Arbeiten mit Java-Schnittstellen und -Klassen

Teil 2 von 2

Dirk Riehle, Ubilab, UBS

Bahnhofstrasse 45, CH-8021 Zürich

Die Unterscheidung von Schnittstellen und Implementierungen ist ein in der Softwaretechnik schon lange bekanntes Konzept. Java bietet diese Unterscheidung zwar an, allerdings ist sie für viele Java-Entwickler anfangs ungewohnt. Dieser zweiteilige Artikel soll dem abhelfen. Ging es im ersten Teil um Schnittstellen und Implementierungen, geht es nun um Schnittstellen- und Klassenvererbung und die Faktorisierung von Implementierungen. Zur Illustration dient wiederum das Beispiel eines einfachen Namensdienstes, wobei neben der sauberen und änderungsfreundlichen Modellierung insbesondere die Codewiederverwendung betont wird.

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, wie zum Beispiel der Abstract-Window-Toolkit (AWT), daß der sinnvolle Einsatz von Schnittstellen in Java nicht immer selbstverständlich ist.

Dies ist der letzte von zwei Artikeln, von denen der erste in der letzten Java-Spektrum Ausgabe erschienen ist. Im vorigen Artikel habe ich die Bedeutung von Schnittstellen und ihre Unterscheidung von Klassen diskutiert. Ich greife dies hier noch einmal kurz auf, um dann zu fortgeschritteneren Themen der Vererbung und Codefaktorisierung zu gelangen. Als Beispiel verwende ich dabei den Entwurf eines Namensdienstes (naming service), wie er in der CORBA Object Services Specifikation (COSS) definiert wird.

2 Schnittstellen und Implementierungen

In der objektorientierten Modellierung von Anwendungsdomänen verwendet man primär die Konzepte Klasse und Objekt. Eine Klasse ist die Abstraktion von mehreren ähnlichen Phänomenen, die man in einem oder mehreren Anwendungsbereichen vorfindet. Objekte selbst sind die konkreten Phänomene.

In der Java-basierten Implementierung von Softwaresystemen drückt man die in der Modellbildung definierten Klassen durch Java-Schnittstellen aus und implementiert diese Schnittstellen durch Java-Klassen. Es ist auch denkbar, eine Modellklasse direkt durch eine Java-Klasse zu implementieren, was aber zu Problemen führen würde. Da Schnittstelle und Implementierung direkt in einer Klasse zusammengefaßt sind, können die Implementierungen nicht mehr frei variiert und ausgewählt werden. Weiterhin führen Änderungen an der Implementierung der Klasse zu direkten Rückwirkungen auf Klientencode, zum Beispiel zum notwendigen Anpassen des Codes oder auch zu Neukompilierungen. Der Einsatz von expliziten Schnittstellen hilft, diese Probleme weitgehend zu vermeiden und höhere Flexibilität, Anpassbarkeit und Wartungsfreundlichkeit zu erreichen.

Im vorigen Artikel habe ich dies zuerst am Beispiel der Implementierung des Namenskonzepts gezeigt. Ein Name wie “net/projects/Geo/Vroom” besteht aus mehreren Namenskomponenten und kann auf verschiedene Weisen implementiert werden. Es wurde die Schnittstelle Name definiert sowie zwei Implementierungen illustriert. Zum einen wurde Name durch die Klasse StringNameImpl implementiert, welche den Namen als einen einzelnen String verwaltet und die Namenskomponententrennzeichen (“/”) maskiert; zum anderen wurde Name durch die Klasse VectorNameImpl implementiert, welche die einzelnen Namenskomponenten als Strings in einem Vector verwaltet. Keine der Implementierungen ist der anderen überlegen—es kommt auf den Anwendungszweck an, welche Implementierung die jeweils am besten geeignete ist.

In der Fortführung des Beispiels habe ich dann die Aufgabe eines Namensdienstes beschrieben und zwei Implementierungsvariationen diskutiert. Ein Namensdienst dient dazu, Objekte mit einem Namen belegen, sie

unter diesem Namen ablegen, und sie schließlich unter diesem Namen später wieder erfragen zu können. Der folgende Code beschreibt den Namensdienst als Schnittstelle NamingContext; diese Schnittstelle wurde direkt aus der CORBA Object Services Specification für den Namensdienst (naming service) entnommen [1].

```
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);
}
```

Die Operationen bind() und resolve() dienen dem Ablegen und Abfragen von Objekten unter Namen. Mit folgendem Code kann ein Kundenbrowser-Objekt abgelegt und später wieder ergefragt werden:

```
// unter Namen ablegen
Browser browser = new BrowserImpl();
Name browserName = new StringNameImpl("KundenBrowser");
LokaleDienste.gibNamensDienst().bind(browserName, browser);
...
// unter Namen abfragen
Name browserName = new StringNameImpl("KundenBrowser");
Object object = LokaleDienste.gibNamensDienst().resolve(browserName);
Browser browser = (Browser) object;
...
```

Die Implementierung der NamingContext-Schnittstelle geschieht mithilfe von abstrakten und konkreten Klassen. Im konkreten Fall benötigen wir zwei verschiedene Implementierungen. Die eine basiert auf einer im Hauptspeicher verwalteten Hash-Tabelle, die Namen auf Objektreferenzen abbildet. Die andere verwaltet Namen und Objektreferenzen in abgespeicherter Form in einer oder mehreren Dateien. Die Hauptspeichervariante ist schnell und funktioniert mit kleinen bis mittelgroßen Menge von Namen/Objektreferenz-Paaren. Sie ist aber unsicher, weil die Abbildungsinformation nur im Hauptspeicher gehalten wird und bei einem Absturz verloren geht. Deswegen gibt es eine weitere dateibasierte Implementierung, welche jedes Namen/Objektreferenz-Paar sofort auf die Festplatte schreibt und somit auch Systemabstürze überlebt. Die Konsequenz ist, daß diese Implementierung zumeist langsamer ist als die reine hauptspeicherbasierte Variante. Bei der Verwendung von Zwischenspeicherung (Caching) gilt dies zumindest für den ersten Zugriff nach Starten des Dienstes.

Diese zwei Implementierungen wurden als die Klassen InMemoryNamingContextImpl und FileBasedNamingContextImpl definiert. Ihre Gemeinsamkeiten wurden in der abstrakten Oberklasse NamingContextDefImpl festgehalten. Abbildung 1 zeigt den resultierenden Entwurf.

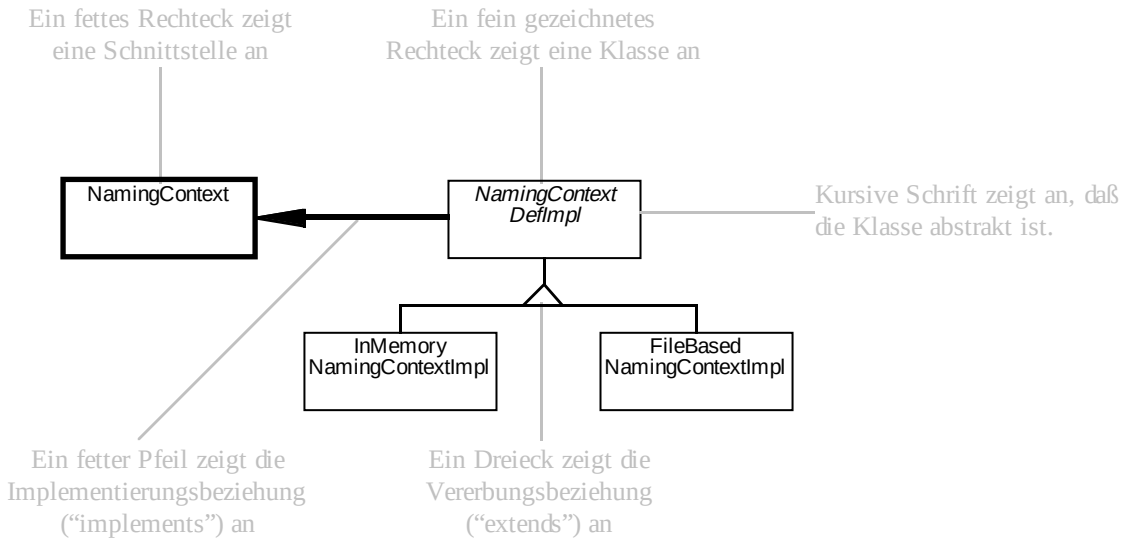

Abbildung 1: Das Beispiel der Namenskontext-Schnittstelle mit zwei Implementierungsklassen und einer gemeinsamen abstrakten Oberklasse

Die abstrakte Klasse NamingContextDefImpl stellt ein Gerüst für ihre Unterklassen dar, welches von ihnen auszufüllen ist. Sie implementiert die gemeinsamen Aspekte ihrer vorweg gedachten Unterklassen (DefImpl steht für Default-Implementierung). Dazu verwendet es Konzepte wie Fabrik- und Schablonenmethoden [2], gemeinhin auch als Prinzip der schmalen Vererbungsschnittstelle bekannt (narrow inheritance interface principle). NamingContextDefImpl sieht folgendermaßen aus:

```
public abstract class NamingContextDefImpl
```

```
  implements NamingContext
{

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 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, der von Unterklasse zu Unterklasse variiert. Diese variablen Teile sind im Beispiel die in NamingContextDefImpl definierten abstrakten Operationen doBind, doUnbind and doContains. Diese Operationen stellen die Vererbungsschnittstelle (“protected”-Schnittstelle) dar.

Die Operationen doBind, doUnbind und doContains werden dann in der Unterklasse InMemoryNamingContext mithilfe einer Hash-Tabelle und in der Klasse FileBasedNamingContextImpl mithilfe einer Datei implementiert. Die Kunst beim Entwurf der abstrakten Oberklasse NamingContextDefImpl besteht darin, die passende Vererbungsschnittstelle zu finden, auf deren Basis NamingContextDefImpl implementiert werden kann. Unterklassen füllen dann nur noch diese Vererbungsschnittstelle aus.

An dieser Stelle setzen wir jetzt wieder auf und führen die Diskussion aus dem ersten Artikel weiter.

3 Schnittstellen- und Klassenvererbung

Bis jetzt habe ich gezeigt, wie man eine Schnittstelle implementiert und wie man dabei unter Verwendung von abstrakten und konkreten Klassen eine gute Codewiederverwendung erreichen kann. Natürlich aber bestehen Softwaresysteme aus komplexen Schnittstellenhierarchien, so daß wir uns in diesem Abschnitt die Vererbung auf Schnittstellenseite anschauen, und uns überlegen, wie wir dies mit den Konzepten für Implementierungsklassen und Codewiederverwendung zusammenbringen können.

Dazu greifen wir wieder auf das Beispiel des Namensdienstes zurück. Ein Objektname wie zum Beispiel “http://swt-www.informatik.uni-hamburg.de/~riehle/SwissPhone1.html” ist eine recht komplexe Angelegenheit. Er besteht aus mehreren Teilen, die alle komplizierte Lookup-Operationen verlangen. Zuerst interpretiert ein Web-Browser das verwendete Protokoll, dann liefert der Internet-Domain-Name-Service eine Ethernet-Adresse für “swt-www.informatik.uni-hamburg.de” und schließlich liefert der http-Server den Inhalt von “~riehle/SwissPhone1.html”. Bei der Interpretation des Dateinamens im lokalen Dateisystem kann es auch wieder beliebig kompliziert werden, etwa wenn symbolische Links über NFS-Mount-Punkte Dateisystem-übergreifend auf Dateien verweisen.

Diese unterschiedlichen und zum Teil sehr komplizierten und implementierungstechnisch anspruchvollen Aspekte beim Auflösen eines komplexen Namens möchte man zumeist voneinander trennen, so daß sie austauschbar werden, ohne daß die nicht geänderten Teile in Mitleidenschaft gezogen werden.

Dazu kann man das Kompositummuster (Composite design pattern) aus [2] verwenden. Es beschreibt, wie rekursive Baumstrukturen aufgebaut werden. Es funktioniert ausgezeichnet für hierarchische Namensräume, welche eine Baumstruktur aufweisen [3]. Man kann sich das am einfachsten mithilfe des Beispiels eines hierarchischen Dateisystems klarmachen. Jedes Verzeichnis definiert den Namensraum für darin enthaltene Objekte und ermöglicht es, auf diese Objekte zuzugreifen, ohne den vollen Pfadnamen angeben zu müssen. Abbildung 2 zeigt einen einfachen Dateibaum.

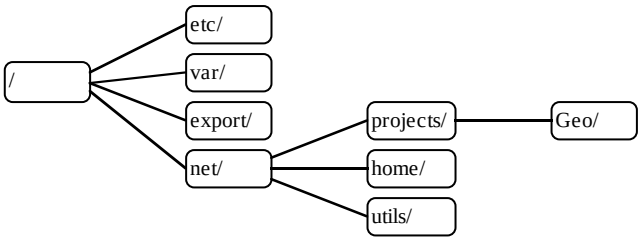

Abbildung 2: Ein hierarchischer Namensraum, hier am Beispiel eines Dateisystems

Die Datei “/net/projects/Geo/Geo.proj” kann entweder von “/” unter dem Namen “net/projects/Geo/Geo.proj”, oder von “/net/” unter dem Namen “projects/Geo/Geo.proj”, oder von “/net/projects/” unter dem Namen “Geo/Geo.proj”, oder von “/net/artktis/Geo/” unter dem Namen “Geo.proj” erfragt werden. Die hinter dem Trennzeichnen “/” stehenden Objekte stellen sogenannte Namenskontexte dar, welche einen hereingereichten Namen interpretieren und das passende Objekt zurückgeben können. Wir haben die Schnittstelle solcher Namenskontextobjekte bereits als NamingContext definiert.

Nun macht es meist keinen Sinn, wenn das “/”-Namenskontextobjekt den Namen “net/projects/Geo/Geo.proj” selbst vollständig auflöst. Es ist besser, wenn es seine Subkontexte kennt und die Aufgabe an diese, die selbst normale Namenskontexte sind, delegiert. Subkontextobjekte von “/” sind zum Beispiel laut obiger Abbildung “etc/”, “var/”, “export/” und “net/”. Beim Auflösen eines Namens zu einem Objekt entfernt das Namenskontextobjekt zuerst jenen Teil des Namens, der zum ausgewählten Subkontext führt. “/” entfernt also “net/” aus dem Namen “net/projects/Geo/Geo.proj”, so daß “projects/Geo/Geo.proj” bleibt. Mit dem verkürzten Namen wendet es sich nun an das Subkontextobjekt, welches seinerseits dasgleiche macht, bis das gesuchte Objekt gefunden ist. Es ergibt sich eine Tiefensuche in den Baum hinein, die durch den ursprünglich definierten Namen festgelegt ist.

In der Praxis realisiert man natürlich nicht jedes Verzeichnis als ein Namenskontextobjekt, sondern geht sehr viel sparsamer damit um. Eine Möglichkeit ist zum Beispiel, für jedes physikalische Dateisystem ein eigenes Namenskontextobjekt zu haben, welches dann Subkontexte an den Mount-Punkten einhängt.

Um nun Subkontexte in Kontextobjekten einhängen zu können, müssen wir die Schnittstelle von NamingContext um die benötigte Verwaltungsfunktionalität erweitern. Dies führt zur Schnittstelle CompositeNamingContext (zusammengesetzter Namenskontext):

```
public interface CompositeNamingContext
  extends NamingContext
{
  public void bindSubContext(Name name, NamingContext nc);
  public void rebindSubContext(Name name, NamingContext nc);
  public void unbindSubContext(Name name);
  public NamingContext resolveSubContext(Name name);
  public boolean containsSubContext(Name name);
  public Vector getSubContexts();
}
```

Ein Objekt, das die CompositeNamingContext-Schnittstelle erfüllt, ist also ein Objekt, das zuerst einmal wie ein normales Namenskontextobjekt funktioniert, da es von NamingContext erbt. Zusätzlich kann man für Teile des von ihm aufgespannten Namensraum Subkontexte einhängen, welche dann für diesen Teilbereich zuständig sind. Dies ist die typische Funktionalität zur Verwaltung von Kind-Objekten in einem Baum, also getSubContexts(), bindSubContext(...), und so weiter. Die obige Schnittstelle wurde aus der CORBA COSS-Spezifikation für Namensdienste abgeleitet. Interessanterweise gibt es dort keine getSuperContext()-Operation, was bedeutet, daß praktisch nie den Baum hoch gesucht wird.

Das zusammengesetzte Namenskontextobjekt wird das Heraussuchen eines Objekts zu einem Namen genau dann an einen Subkontext delegieren, wenn aus dem Namen ersichtlich ist, daß dieser Subkontext für den im Namen impliziten Namensraum zuständig ist.

Eine CompositeNamingContext-Implementierung muß also zwei Aufgaben zusätzlich zur NamingContext-Implementierung erfüllen. Sie muß es ermöglichen, daß man untergeordnete Namensräume als Subkontextobjekte einhängt, und sie muß die Funktionalität von NamingContext daraufhin anpassen.

- Die Implementierung der Operationen bindSubContext, rebindSubContext, usw. kann analog zu den bind, rebind, usw. Operationen von NamingContext geschehen. Zumeist mag es sinnvoll sein, mittels einer im Hauptspeicher verwalteten Hash-Tabelle herauszufinden, ob der hereingereichte Name im lokalen Namensraum liegt, mitunter aber wird man auch andere Implementierungen wählen, wie zum Beispiel die bereits diskutierte dateibasierte Lösung.

- Hinzukommt, daß man die bind, rebind, usw. Operationen von NamingContext anpassen muß. Bei jedem Aufruf dieser bereits in NamingContext definierten Operationen muß zuerst geprüft werden, ob der aufzulösende Name im lokal verwalteten Namensraum liegt, oder ob die Aufgabe des Auflösens an einen Subkontext weitergereicht werden muß.

Ich habe bis zu dieser Stelle bewußt keinen Code gezeigt, um die Unterschiedlichkeit der verschiedenen Implementierungsaspekte zu verdeutlichen. Von der Art der gewünschten Codewiederverwendung hängt es ab, wie man den Vererbungsbaum schneidet.

Nehmen wir an, die Klasse InMemoryCompositeNamingContextImpl verwendet eine Hauptspeicher-Implementierung zur Verwaltung der Subkontexte. Soll sie von InMemoryNamingContextImpl erben? Dann hätten wir eine Implementierung, bei der sowohl das Auflösen eines Namens wie auch das Verwalten von Subkontexten im Hauptspeicher geschieht.

Wollen wir aber eine Implementierung verwenden, welche die Namen mittels einer dateibasierten Datenbank verwaltet und die Subkontexte im Speicher hält, so wäre es besser von FileBasedNamingContextImpl erben. Und wollen wir die Namenstabellen im Hauptspeicher halten, und die Subkontexte in Dateien, müssen wir noch eine weitere Kombination implementieren.

Hinzukommt, daß die konkreten Klassen wie InMemoryNamingContextImpl das sind, was man "semantisch geschlossen" nennt: Ihr Verhalten ist durch ihre Implementierung vollständig bestimmt, so daß weitere (nicht-triviale) Unterklassen die Implementierung nur noch "verschlimmbessern" können. Dies ist Teil des sogenannten Offen-Geschloßen-Prinzips (open-closed principle [4]).

Der nächste Abschnitt beschreibt eine Lösung zu diesem Problem. Im Augenblick bleibt uns nur die Erkenntnis, daß wir von konkreten Implementierungsklassen normalerweise nicht erben sollten, weil wir die Semantik der Implementierung durcheinander bringen würden und mit einer Explosion von Implementierungsvarianten rechnen müssen.

Es ist deswegen am besten, sich auf das im letzten Abschnitt vorgestellte Konzept der abstrakten Default-Implementierungen zu besinnen. Dies führt zum in Abbildung 3 dargestellten Klassenbaum.

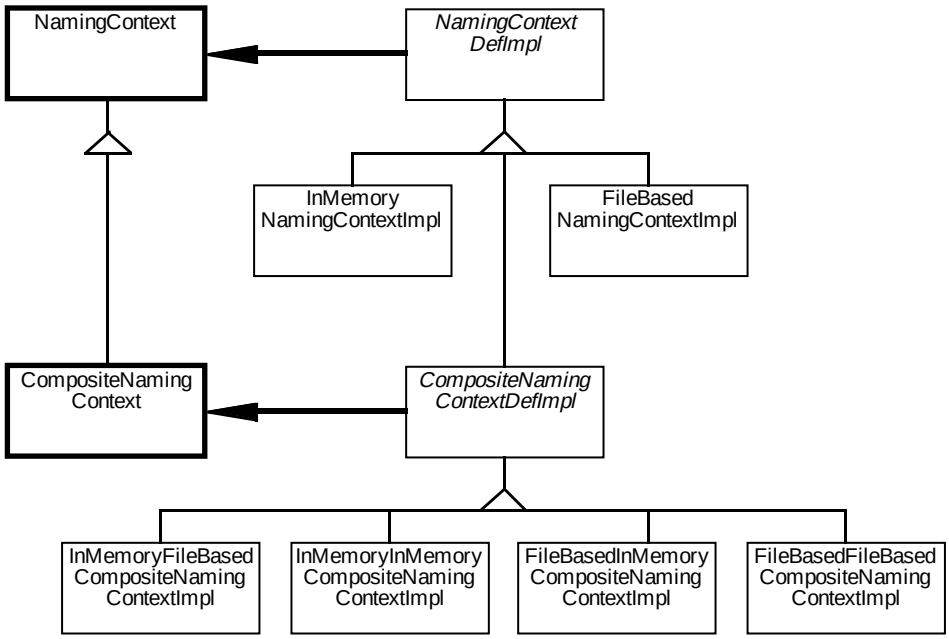

Abbildung 3: Illustration der Vererbung von Schnittstellen und Default-Implementierungen

Hierbei gibt es wieder eine abstrakte Default-Implementierungsklasse namens CompositeNamingContext-DefImpl, welche von NamingContextDefImpl erbt. Wie NamingContextDefImpl implementiert sie ihre allgemeine Funktionalität mithilfe einer schmalen Vererbungsschnittstelle:

```
public abstract class CompositeNamingContextDefImpl
   extends NamingContextDefImpl
   implements CompositeNamingContext
{

... Konstruktoren und Initialisierungscode

public void bindSubContext(Name name, NamingContext nc)
{
```

```
  if ( ! contains(name) ) doBindSubContext(name, nc);
}

public void rebindSubContext(Name name, NamingContext nc)
{
  if ( ! contains(name) ) return;
  doUnbind(name);
  doBind(name, nc);
}

... und die anderen Operationen

// hier jetzt die Vererbungsschnittstelle
protected abstract void doBindSubContext(Name name, NamingContext nc);
protected abstract void doUnbindSubContext(Name name);
protected abstract void doContainsSubContext(Name name);

}
```

Das ganze funktioniert also analog zu NamingContextDefImpl. Der Hauptteil der Implementierung wird auf Basis einer kleinen Vererbungsschnittstelle erstellt, so daß Unterklassen nur noch diese Vererbungsschnittstelle ausfüllen müssen.

Die bind, rebind, usw. Operationen müssen natürlich auch noch angepasst werden. Dazu bedienen wir uns einer Hilfsoperation, die herausfindet, ob ein Name im Namensraum eines registrierten Subkontexts liegt:

```
// gehoert zu CompositeNamingContextImpl
protected NamingContext findSubContext(Name name)
{
  for ( int i = 0; i < name.noOfComponents(); i++ )
  {
    Name context = name.context(i);
    if ( containsSubContext(context) )
    {
      return resolveSubContext(context);
    }
  }
  return null;
}
```

Mit dieser Hilfsoperation, kann zum Beispiel resolve() herausfinden, ob ein aufzulösender Name lokal verwaltet wird, oder ob die Operation die Aufgabe an einen Subkontext weitergeben soll:

```
// gehoert zu CompositeNamingContextDefImpl
public Object resolve(Name name)
{
  NamingContext subContext = findSubContext(name);
  if ( subContext != null )
  {
    Name subName = name.without(subContext.getName());
    return subContext.resolve(subName);
  }
  else
  {
    return super.resolve(name);
  }
}
```

Konkrete Unterklassen von CompositeNamingContextDefImpl müssen jetzt nur noch die offenen Stellen implementieren. Sie müssen die Vererbungsschnittstelle von NamingContextDefImpl zur Namensverwaltung sowie die Vererbungsschnittstelle von CompositeNamingContextDefImpl zur Subkontextverwaltung implementieren. Gibt es jeweils zwei Implementierungsvariationen, so kommen wir in der Kombination auf vier verschiedene Unterklassen, wie in obiger Abbildung auch dargestellt. Dieser kombinatorischen Explosion kann mithilfe der im nächsten Abschnitt geschilderten Implementierungsfaktorisierung entgegenwirken.

An dieser Stelle soll lediglich noch einmal das Prinzip festgehalten werden. Zu jeder Schnittstelle gibt es eine abstrakte Default-Implementierung, die parallel zum Schnittstellenbaum verläuft. Sie ist nicht instantiierbar. Je tiefer der Baum, desto mehr offene Vererbungsschnittstellen gibt es, die von Unterklassen implementiert werden müssen. Durch die Verwendung von Default-Implementierungen aber erreicht man bereits einen Grad an direkter Codewiederverwendung, der angesichts der Einschränkung in Java, das Klassen nur Einfachvererbung kennen, sehr gut ist.

Wenn das durch die Schnittstelle modellierte Konzept als instantiierbar betrachtet werden soll, dann sollte es auch konkrete Unterklassen dieser abstrakten Implementierungsklasse geben, die von Klienten direkt genutzt werden können. Dies ist pragmatisch sinnvoll, weil man nicht erst selbst eine Unterklasse erstellen muß, und psychologisch hilfreich, damit sich niemand über die Komplexität der vorgestellten Entwurfskonzepte beschwert.

Sie werden sich vermutlich fragen, was passiert, wenn die Schnittstellenseite keinen Einfachvererbungsbaum darstellt, sondern Mehrfachvererbung anwendet. Das ist natürlich möglich, aber man muß dabei jedesmal gucken,

aus welchem Grund die Mehrfachvererbung verwendet wird. Meine Erfahrung ist es, daß sich von der fachlich motivierten Modellierung her immer ein Einfachvererbungsbaum ergibt. Mehrfachvererbung macht nur Sinn, um sogenannte Protokolle in einen Einfachvererbungsbaum zu mischen. Diese Protokollschnittstellen sind im Englischen leicht an ihrer Endung “-able” zu erkennen. Sie kommen zumeist nicht mit einer eigenen Default-Implementierung daher; wenn doch, so kann auch diesem Problem durch die im nächsten Abschnitt geschilderten Konzepte geholfen werden.

4 Faktorisierung von Implementierungen

Wie der vorige Abschnitt gezeigt hat, bestehen komplexe Objekte oftmals aus mehreren, implementierungstechnisch zum Teil gut trennbaren Aspekten. Ein zusammengesetzter Namenskontext muß sich sowohl um die Verwaltung des Namensraums wie auch die Verwaltung von Subkontexten kümmern. Beides geschieht zwar in ein- und demselben Objekt; die Implementierungen dieser Teilaspekte hängen oftmals nicht voneinander ab.

Die Einführung einer abstrakten Oberklasse, welche das Standardverhalten einer Schnittstelle implementiert, führte gleichzeitig zum Konzept der Vererbungsschnittstelle. Eine abstrakte Oberklasse definiert eine Vererbungsschnittstelle so, daß sie sich in ihrer eigenen Implementierung auf diese Schnittstelle abstützen kann, sie aber noch nicht selbst implementiert. Abbildung 4 greift das Beispiel wieder auf und zeigt die Vererbungsschnittstelle für die Verwaltung des Namensraums und wo sie implementiert wird an. Ein dicker Strich stellt die Definition einer Vererbungsschnittstelle in einer Klasse dar und ein in einer Unterklasse darumgezeichnetes Rechteck zeigt die Implementierung an.

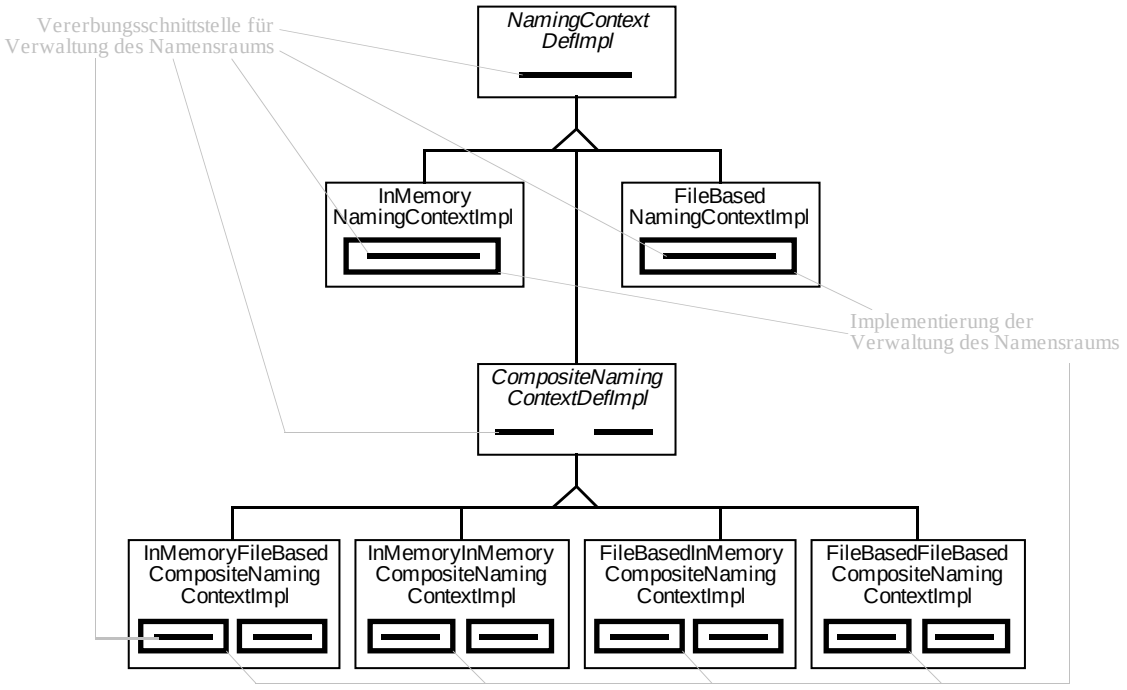

Abbildung 4: Illustration der Vererbungsschnittstellen abstrakter Oberklassen

Auf Stufe CompositeNamingContextDefImpl kam die Vererbungschnittstelle für die Verwaltung von Subkontexten hinzu, was zur kombinatorischen Explosion von Unterklassen führt. Dies sind die Klassen InMemoryFileBased-, InMemoryInMemory-, FileBasedInMemory-, und FileBasedFileBasedCompositeNamingContextImpl (ich belasse diese fürchterlichen Namen aus Gründen der Illustration so lang—in der konkreten Anwendung kann man dann immer noch geeignete Abkürzungen wählen oder die überlangen Namen hinter Bequemlichkeitsoperationen verstecken).

Die Präfixe InMemory und FileBased benennen die gewählte Implementierung für die Verwaltung des Namensraums und der Subkontexte. Die beiden Klassen InMemoryFileBased- und InMemoryInMemoryCompositeNamingContextImpl implementieren beide eine hauptspeicherbasierte Verwaltung des Namensraums. Sie können diese Implementierung aber nicht von einer gemeinsamen Oberklasse erben (wie zum Beispiel InMemoryNamingContextImpl), ohne andere Vorteile zu verlieren. Dieser andere Vorteil wäre im konkreten Beispiel die Oberklasse CompositeNamingContextDefImpl, welche das Standardverhalten für

zusammengesetzte Namenskontexte definiert. Sie könnten es natürlich trotzdem versuchen, würden aber bei jeder weiteren Kombination auf neue Widersprüche stoßen. Die Widersprüche ergeben sich daraus, daß Sie auf Klassenebene nur Einfachvererbung einsetzen können. Wie auch immer man es dreht und wendet: In Java bekommt man mit Vererbung allein nie eine maximale Codewiederverwendung hin.

Wie kann man nun die Redundanz in den Implementierungen und die daraus folgende Wartungs- und Änderungsunfreundlichkeit vermeiden?

Indem man Objektkomposition statt Vererbung als Mittel der Codewiederverwendung verwendet. Den wichtigsten Grundstein haben wir mit dem Konzept der Vererbungsschnittstelle bereits gelegt.

Wir delegieren die Implementierung der Vererbungsschnittstelle an ein neues Objekt, statt sie in jeder Unterklasse zu implementieren. Dieses neue Objekt ist ein Exemplar einer Klasse, welches die Vererbungsschnittstelle (und nur diese) implementiert. Da die Objekte variieren können, können auch die Implementierungen sich ändern, solange sie nur die Vererbungsschnittstelle erfüllen. Im Beispiel führt dies zu folgender Klassenhierarchie:

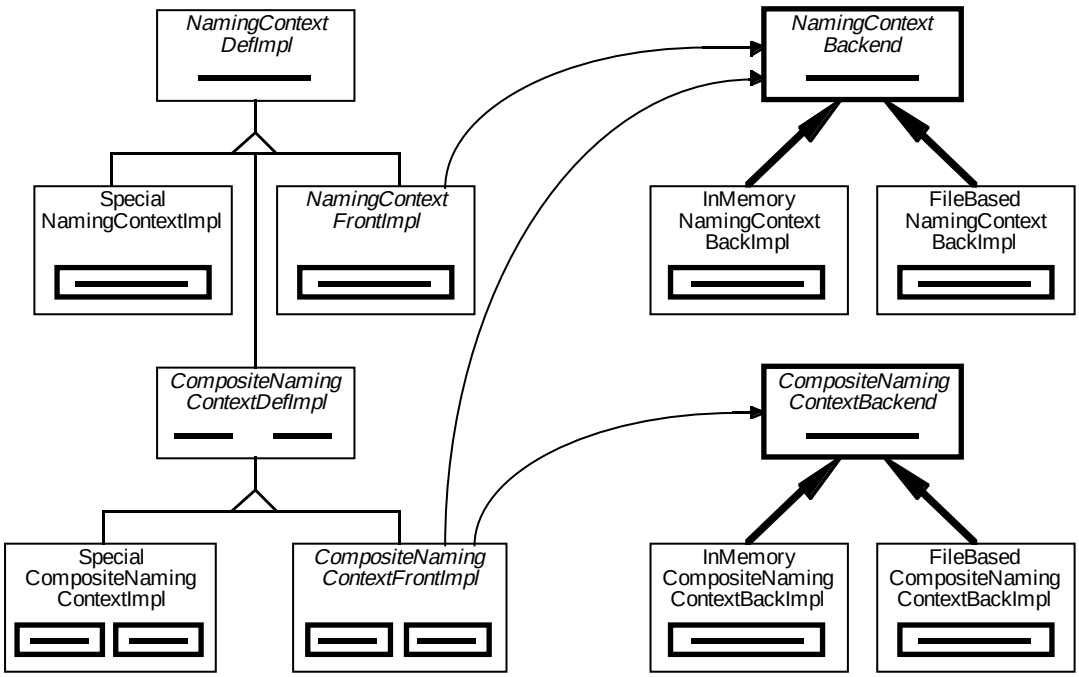

Abbildung 5: Parametrisierung von Implementierungsklassen mit Implementierungsstrategien

Die neue Unterklasse NamingContextFrontImpl von NamingContextDefImpl implementiert die Vererbungsschnittstelle, in dem sie die Operationsaufrufe an ein Objekt delegiert, welches die Schnittstelle NamingContextBackend erfüllt. Im Beispiel können diese Objekte Exemplare der Klassen InMemoryNamingContextBackImpl oder FileBasedNamingContextBackImpl sein.

Ebenso gibt es eine Klasse CompositeNamingContextFrontImpl, welche die Implementierung der Verwaltung des Namensraums wie auch der Subklasse an weitere Objekte delegiert. Für jeden Implementierungsaspekt gibt es eine eigene “Backend”-Klasse, welche genau diesen Aspekt ausfüllt, und deren Objekte zur Implementierung der “Frontend”-Klassen dienen.

Die Implementierung der Vererbungsschnittstelle von NamingContextFrontImpl sieht also folgendermaßen aus:

```
public class NamingContextFrontImpl
   extends NamingContextDefImpl
{

public NamingContextFrontImpl(NamingContextBackend ncb)
{
   namespace = ncb;
}

protected void doBindSubContext(Name name, NamingContext nc)
{
   namespace.bindSubContext(name, nc);
}

protected void doUnbindSubContext(Name name);
{
   namespace.unbindSubContext(name)
}
```

```
protected boolean doContainsSubContext(Name name);
{
   return namespace.containsSubContext(name);
}

// Implementierungszustand
protected NamingContextBackend namespace;

}
```

Möchte jetzt ein Klient ein Namenskontextobjekt erzeugen, so schreibt er folgenden Code:

```
// Auswahl der Implementierung
NamingContextBackend ncb = new InMemoryNamingContextBackImpl();
// Erzeugen des Namenskontextobjekts
NamingContext nc = new NamingContextFrontImpl(ncb);
```

Analog geschieht dies im Fall des zusammengesetzten Namenskontexts mit dem Unterschied, daß dieser mit zwei Implementierungsobjekten parametrisiert wird.

```
public class CompositeNamingContextFrontImpl
   extends CompositeNamingContextDefImpl
{

public CompositeNamingContextFrontImpl(NamingContextBackend ncb,
   CompositeNamingContextBackend cncb)
{
   initialize(ncb, cncb);
}

protected initialize(NamingContextBackend ncb,
   CompositeNamingContextBackend cncb)
{
   namespace = ncb;
   subcontexts = cncb;
}

// Implementierung der Vererbungsschnittstelle von
// NamingContextDefImpl durch Delegation

...

// Implementierung der Vererbungsschnittstelle von
// CompositeNamingContextDefImpl durch Delegation

...

// Implementierungszustand
protected NamingContextBackend namespace;
protected CompositeNamingContextBackend subcontexts;

}
```

Tritt eine bestimmte Kombination von Implementierungsobjekten besonders häufig auf, so kann man diese Kombination in einer eigenen Unterklasse festhalten. Die Kombination einer dateibasieren Namensraumverwaltung und einer hauptspeicherbasierten Subkontextverwaltung würde also zur folgenden Klasse führen:

```
public class FileBasedInMemoryCompositeNamingContextFrontImpl
   extends CompositeNamingContextFrontImpl
{

public FileBasedInMemoryCompositeNamingContextFrontImpl()
{
   NamingContextBackend ncb = new InMemoryNamingContextImpl();
   CompositeNamingContextBackend cncb = new InMemoryNamingContextBackend();
   initialize(ncb, cncb);
}

}
```

Ein Klient kann dann Objekte dieser wohl-konfigurierten Klasse direkt erzeugen, ohne den Konfigurationscode selbst schreiben zu müssen.

Das beschriebene Prinzip der Delegation von Implementierungen an eigenständige Objekte entspricht weitgehend dem Entwurfsmuster Brücke aus [2]. Dieses Entwurfsmuster beschreibt, wie eine Abstraktion (Frontend, zum Beispiel NamingContext) mit einer Implementierung (Backend, zum Beispiel NamingContextBackend) parametrisiert wird. Man könnte die Implementierungsobjekte auch als Strategien (ein weiteres Entwurfsmuster aus [2]) betrachten. Da aber Strategien gern zur Laufzeit gewechselt werden, und dies hier kein wichtiges Kriterium ist, trifft diese Einschätzung aber meines Erachtens nicht gut zu.

5 Zusammenfassung

Diese zwei Artikel haben die verschiedenen Aspekte des Arbeitens mit Java-Schnittstellen und -Klassen aufgezeigt, die sich bei Konzentration auf die Modellierung von Konzepten und ihre Implementierung ergeben. Es wurde sowohl die saubere und änderungsfreundliche Modellierung betont, als auch gezeigt, wie ein hoher Grad an Codewiederverwendung erreicht werden kann.

Die verschiedenen Konzepte “Direkte Verwendung von Klassen”, “Trennung von Schnittstelle und Implementierung”, “Abstrakte Default-Implementierung”, “Konkrete Implementierung”, “Parallele Hierarchien” und “Objektbasierte Codefaktorisierung” ermöglichen eine feingliedrige Abstufung des Ausmaßes and Flexibilität, welches man erreichen und einsetzen können möchte.

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

Literaturverweise

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

[2] 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*).

[3] Sape Mullender. *Distributed Systems.* Addison-Wesley, 1992.

[4] Bertrand Meyer. *Object-Oriented Software Construction.* 2nd Edition. Prentice-Hall, 1997.

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* (siehe [2]) sowie Autor des frisch bei Addison-Wesley Deutschland erschienenen Buchs *Entwurfsmuster für Softwarewerkzeuge.*

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