Palladio Componentmodel

Entwurfsbeschreibung

Marko Hoyer
Marko.Hoyer@informatik.uni-oldenburg.de

4. November 2005

Inhaltsverzeichnis

1 Einleitung

2 Architektur

In diesem Kapitel wird die derzeitige Architektur des Komponentenmodells vorgestellt. Sie setzt sich aus den in Abbildung 1 dargestellten und im Folgenden kurz erläuterten Bestandteilen zusammen. An der Umrandung der Blöcke ist abzulesen, ob diese bereits entworfen oder lediglich als Erweiterungen geplant sind. Details und Informationen zur Umsetzung der Bestandteile des Komponentenmodells sind Inhalt der folgenden Kapitel dieses Dokuments.

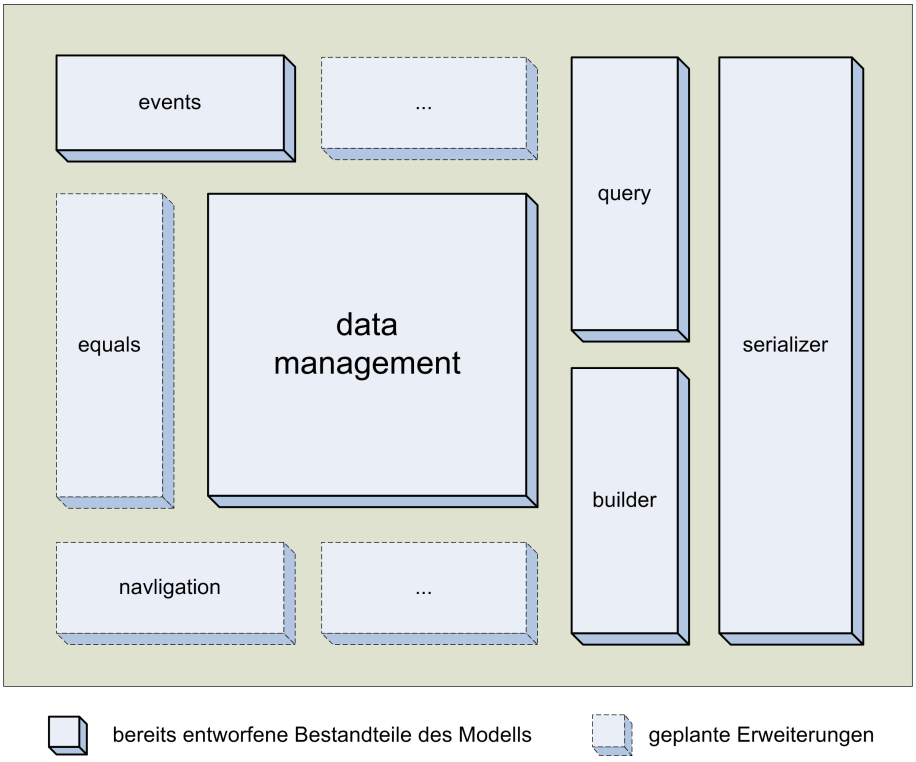

Abbildung 1: Architektur des Komponentenmodells

Zentrum der Architektur bildet die in der Abbildung mit *data management* bezeichnete Datenhaltung. Sie dient der lokalen Speicherung der Entitäten und Relationen des Modells zur Laufzeit der nutzenden Anwendung. Hierfür stellt sie Möglichkeiten zum Lesen und Schreiben der Daten zur Verfügung. Die Konsistenz der Daten ist in dieser Schicht lediglich in Bezug auf die verwendeten Datenstrukturen zu gewährleisten. Semantische Fehler im Sinne des theoretischen Komponentenmodells sind von der Datenhaltung zu

tollerieren, um unabhängig von möglichen Änderungen dieser Semantik zu bleiben. Aufgrund dessen ist der das Modell nutzenden Anwendung keine Möglichkeit zu gewähren, direkt auf die Datenhaltung zuzugreifen, da sonst Korrektheit im Sinne des theoretischen Modells nicht mehr gewährleistet werden kann. Zugriff ist erst nach Überprüfung durch entsprechende Zwischenschichten zu gestatten.

Schreibende Änderungen der Anwendung am Modell sind hierbei durch den in der Architekur mit *builder* bezeichneten Block vorgesehen. Dieser stellt eine Infrastruktur bereit, welche Änderungen an verschiedenen Stellen des Modells zulässt und den korrekten Aufbau gemäß dem theoretischen Modell sicherstellt. Es können an dieser Stelle bereits durch geschickte Wahl der Zugriffsmethoden Fehler ausgeschlossen werden. Eine Möglichkeit der Umsetzung dieser Schicht unter Beachtung der Hierarchie des Komponentmodells wird in Kapitel 5 vorgestellt.

Der lesende Zugriff auf das Modell kann je nach Bedarf durch verschiedene Schichten erfolgen. Die in Abbildung 1 mit *query* bezeichnete Schicht dient der Abfrage von Attributen der Entitäten und der Beziehungen zwischen diesen. Eine direkte Abfragemöglichkeit der Datenhaltung ist prinzipiell möglich, jedoch aufgrund fehlenden Wissens über das theoretische Modell unpraktikabel. Abstraktionen in diesem Sinn sind also ebenfalls Aufgabe der Abfrageschicht.

Die persistente Speicherung des Modells ist Aufgabe der Serialisierungsschicht. Diese soll nach Möglichkeit unabhängig von der genauen Art der Speicherung bleiben. Sowohl der Ex- und Import von Xml-Dateien, wie auch die Speicherung in einer relationalen Datenbank sollen möglich sein. Ebenfalls ist die binäre Speicherung in einem eigenen Datenformat denkbar. Eine Entkopplung vom Kern des Komponentenmodells durch austauschbare Module bietet hierbei die größte Flexibelität. Der Zugriff der Serialisierungsschicht auf die lokal gehaltenen Daten kann auf zwei unterschiedliche Arten realisiert werden, direkter Zugriff auf den Kern oder indirekter Zugriff über die Abstraktionsschichten *query* und *builder*. Vorteil der ersten Variante ist die freie Wahl der Zugriffsmethoden auf die Daten. Je nach Art der Speicherung können Anfragen gezielt angepasst werden, um effizient schreiben oder lesen zu können. Nachteil hierbei ist jedoch die Prüfung der Konsistenz und Korrektheit des Modells. Diese müsste redundant zur *builder*-Schicht implementiert werden. Weiterhin wird eine Abhängigkeit zwischen Kern und Serialisierungsschicht geschaffen, die im Falle der zweiten Variante nicht besteht. Die Nutzung der beiden Abstraktionschichten im zweiten Fall bilden weiterhin eine gute Möglichkeit zur effizienten Qualitätssicherung. So schlagen Serialisierungstests fehl, wenn sich entweder Fehler in der Serialisierungsschicht selber oder in der Implementierung der Zugriffsmethoden befinden.

Der Benachrichtigung der nutzenden Software bei Änderungen des Modells dient die in der Abbildung mit *events* bezeichnete Schicht. Ziel hierbei ist die Bereitstellung vollständiger Überwachungsmöglichkeit des Modells ohne direkte Verbindung zu den

Quellen der Veränderung. So sollen sowohl die Änderungen des Modells durch die Builder als auch durch die Deserialisierung überwacht werden. Setzen sich zur Wahrung der Konsistenz und Korrektheit Aktionen transitiv fort, so sind auch diese Änderungen der Überwachung mitzuteilen.

Die in Abbildung 1 mit *navigation* und *equals* bezeichneten Module sind als Erweiterungen geplant. Zweck des ersten Moduls ist die Navigation durch das Komponentenmodell z.B. entlang des Kontrollflusses unter Verwendung verschiedenster Strategien. Das zweite Modul befasst sich mit der Äquivalenz von Bestandteilen eines Modells oder gar von gesamten Modellen. Die Möglichkeit der unterschiedlichen Definition von Äquivalenz anhand verschiedener Kriterien ist hierbei wichtigste Anforderung an dieses Modul.

Die Architektur kann je nach Bedarf beliebig um Module und Schichten erweitert werden. Diese können entweder direkt oder indirekt über Abstraktionsschichten auf den Kern zugreifen. Bei direkten schreibenden Zugriffen ist wie oben bereits erläutert zu beachten, dass die Konsistenz im Sinn des theoretischen Modells eingehalten werden muss. Keinesfalls dürfen Erweitungsmodule der nutzenden Anwendung den direkten Zugriff auf den Kern ermöglichen. Ist direkter Zugriff nicht zwingend erforderlich, so sind die durch die entsprechenden Abstraktionsschichten angebotenen Schnittstellen vorzuziehen.

In den folgenden Kapiteln werden die für die aktuelle Version des Komponentenmodells entworfenen Module und Schichten vorgestellt, Entwurfsentscheidungen begründet und bekannte Probleme erläutert.

3 Datenhaltung im Modellkern

Der Modellkern, dessen Hauptaufgabe die Verwaltung der Daten darstellt, bildet den wichtigsten Bestandteil des Komponentenmodells mit den im ersten Teil des Kapitels erläuterten Anforderungen. Es folgt die Präsentation von drei Ideen zu deren Umsetzung. Abschließend wird die in dieser Version des Komponentenmodells implementierte Variante ausführlich erläutert.

3.1 Anforderungen an den Modellkern

- **Speicherung**
  Der Modellkern muss in der Lage sein, alle Daten zur Laufzeit des nutzenden Programms zu speichern. Zu den Daten gehören die Entitäten des Komponentenmodells mit ihren Attributen. Die Struktur der Attribute beschränkt sich hierbei nicht auschließlich auf Standartdatentypen. Es müssen beliebige z.T. zur Entwurfszeit unbekannte Datenstrukturen speicherbar sein. Weiterhin müssen die Beziehungen

zwischen den Entitäten (z.B. Komponente A enthält Komponente B) festgehalten werden.

- **Konsistenzprüfung**
  Wie eingangs in der Architekturbescheibung erläutert besteht die Aufgabe des Modellkerns nicht in der Implementierung der Konsistenzprüfung des theoretischen Modells. Somit sollte im Idealfall prinzipiell erst einmal alles abspeicherbar sein. Da die Umsetzung dieser Anforderung viele ungenutzte und zu Lasten der Komplexität fallende Möglichkeiten bietet, ist die Nutzung von Wissen über das theoretische Modell bei der Konzeption der Datenhaltung sinnvoll einzubringen. Verstöße gegen die sich hierraus ergebenen Beschränkungen sind dann jedoch durch den Modellkern abzufangen und entsprechend zu behandeln. Soll beispielsweise entsprechend dem o.g. Beispiel die Komponente B der Komponente A hinzugefügt werden, so ist vom Modell diese Beziehung zu speichern. Setzt die gewählte Speicherstruktur hierbei das vorhandensein von Komponente A vorraus, so ist das durch den Modellkern sicherzustellen. Dieser kann dann entweder die Speicherung ablehnen oder selbständig eine Komponente A erzeugen.

- **Zugriffsmethoden**
  Die dritte Anforderung an den Modellkern stellen die Zugriffsmethoden dar. Da in die Datenhaltung, wie oben erläuert, nicht das vollständige Wissen über das theoretische Modell zu implementieren ist, können keine hierauf zugeschnittenen Zugriffsmethoden zur Verfügung gestellt werden. Es ist also eine Schnittstelle zu schaffen, die flexiblen Zugriff auf alle gespeicherten Daten bereitstellt. Bestehen im Modell der Datenhaltung bereits Beziehungen zwischen den Daten, so bietet sich deren Nutzung beim Zugriff an. Weiterhin wichtig ist sowohl bei den Zugriffsmethoden als auch bei der Speicherung die Geschwindigkeit. Dieser Teil des Modells bildet, wie bereits erläutert, die Datenhaltung für die laufende Anwendung. Sorgt die Arbeit auf dem Modell für zu hohe Latenz, so leidet die Nutzbarkeit der Anwendung hierrunter stark.

Nachdem die grundlegenden Anforderungen an den Modellkern erarbeitet wurden, folgt die Vorstellung von Ideen zu deren Umsetzung.

3.2 Ideen zur Umsetzung

Zur Umsetzung des Modellkerns kommen eine Reihe von Strategien in Frage, von denen drei im Folgenden gegeneinander abgegrenzt werden.

Die erste Strategie bedient sich ausschließlich objektorientierter Konzepte. Hierbei werden die Entitäten durch Klasseninstanzen und Beziehungen zwischen diesen durch Referenzen auf andere Instanzen modelliert. Vorteile dieser Variante ergeben sich aus guter

Modellierbarkeit von Spezialisierung, problemloser Speicherung von Attributen unbekannten Typs und hoher Geschwindigkeit. Erfahrungen haben gezeigt, dass sich bei der Umsetzung dieses Konzeptes Probleme hinsichtlich Wartbarkeit und Erweiterbarkeit ergeben, die sich auf die starke Abhängigkeit der Klassen untereinander zurückführen lassen. Ebenfalls schwierig zu modellieren sind auf diese Art zirkuläre Abhänigkeiten.

Der zweite Ansatz bedient sich einer relationalen oder einer objektrelationalen Datenbank. Die Entitäten werden hierbei in entsprechenden Tabellen der Datenbank gespeichert. Die Beziehungen zwischen den Entitäten lassen sich in der Datenbank entsprechend als Beziehungen zwischen den Tabellen modellieren. Für Details zum Entwurf solcher Datenbankschemata und deren Nutzung sei an dieser Stelle auf entsprechende Literatur (z.b. [EN02]) verwiesen. Vorteil dieses Ansatzes liegt in der guten Infrastruktur zum Speichern, Laden und Anfragen von Daten. Nachteilig ist jedoch das zur Entwurfszeit festgelegte Datenschema, welches den unbekannten Attributen nicht gerecht wird. Weiterhin bringt eine Datenbank viel Funktionalität mit, welche i.A. im Rahmen der Datenhaltung einer Anwendung (z.B. eines Editors für das Komponentenmodell) zuviel Aufwand bedeutet. Seitens der Geschwindigkeit kann die Datenbank mit dem ersten Ansatz in Hinblick auf den Anwendungsfall Komponentenmodell nicht mithalten.

Die dritte hier vorgestellte und ebenfalls umgesetze Idee kombiniert die Vorteile der beiden vorherigen Varianten. Die durch ihre Attribute charakterisierten Entitäten werden gemäß dem ersten Ansatz in Form von Objektinstanzen gespeichert. Die Beziehungen zwischen diesen Entitäten hält eine Art Datenbank. Somit lassen sich problemlos alle Art von (auch unbekannten) Attributen speichern. Die verwendete Datenbank bietet performante und flexible Möglichkeiten, Beziehungen zwischen den Entitäten zu erfragen.

Es folgt die detailierte Beschreibung der von uns umgetzten Variante.

3.3 Beschreibung der umgesetzten Variante

Wie im vorherigen Abschnitt kurz erläutert handelt es sich bei den Entitäten um reine Datenkontainer. Es existieren im Komponentenmodell zwei Arten von Entitäten, interne und externe Enitäten. Alle internen Entitäten besitzen neben einer ID einen Namen und eine Liste von zur Laufzeit frei wählbaren Attributen. Je nach Typ der Entität existieren zusätzliche jedoch zur Entwurfszeit festgelegte Attribute (z.B. der Typ einer Komponente). Zu den internen Entitäten gehören Komponenten, Schnittstellen, Verbindungen und Signaturen.

Externe Entitäten zeichnen sich durch ihre zur Entwurfszeit unbekannte Struktur aus. Ihnen ist ausschließlich eine ID und eine Typ-ID gemeinsam. Die ID sorgt für die Eindeutigkeit der Instanz in einem Modell, die Typ-ID dient der Identifikation der verwendeten Implementation der externen Entität. Im Komponentenmodell gehören Protokolle und

Service-Effekt-Spezifikationen (siehe [Reu01]) zu den externen Entitäten, da diese auf verschiedenste Weise (z.B. durch Finite State Machines) implementierbar sind.

Die Schnittstellen aller Entitäten sind in der derzeitigen Implementierung des Komponentenmodells im Namensraum `Palladio.ComponentModel.ModelEntities` zu finden. Es folgt eine genauere Beschreibung der Realisierung der internen und externen Entitäten.

3.3.1 Realisierung interner Entitäten

Das Interface `IComponentModelEntity` bildet die Basisschnittstelle aller internen Entitäten. Es definiert Zugriffsmethoden auf die ID, den Namen und die Attribute, wobei auf die ID auschließlich lesend, auf den Name und die Attribute zusätzlich auch schreibend zugegriffen werden kann. Während der Name direkter Bestandteil der Schnittstelle ist, ist der Zugriff auf die Attribute im Interface `IAttributable` definiert. Dieses ist Teil des Palladio-Attribut-Konzepts und wird von `IComponentModelEntity` geerbt. Ebenso ist die ID nicht direkt im Interface definiert sondern wird von einem Basisinterfaces ererbt. Eine detailierte Betrachtung der Identifizierung von Entitäten ist in Kapitel 3.3.3 zu finden.

Alle internen Entitäten des Komponentenmodells erhalten spezialisierte Schnittstellen, die pro Entität zusätzliche Attribute definieren. Sowohl Basisinterface als auch die spezialisierten Schnittstellen sind von außerhalb zugreifbar. Die Implementierungen der Interfaces sind verborgen. Sie lassen sich unter Verwendung der Klasse *EntityFactory* instanzieren. In der derzeitigen Version des Komponentenmodells ist diese Fabrik ebenfalls nicht von außen nutzbar. Sie wird ausschließlich durch die Builder-Schicht (vgl. Kapitel 5) genutzt. Grund für diese Entscheidung ist die Vermeidung von fremden möglicherweise fehlerhaften Implementierungen der Enitäten.

Diverse Anforderungen nach Erweiterbarkeit der Entitäten um spezialisierte direkt über die Schnittstellen zugreifbare Attribute erzwingen an dieser jedoch möglicherweise eine Änderung der Designentscheidung. So besteht die Möglichkeit, die Implementierung der Entitäten in Form einer Fabrik von außen konfigurierbar zu gestalten.

3.3.2 Realisierung der externen Entitäten

Im Gegensatz zu den internen Entitäten besteht bei den externen Entitäten bereits auch in dieser Version des Kompomentenmodells eine Unabhängigkeit zur Implementierung. Dies ist notwendig, da zur Entwurfszeit weder Informationen über Attribute der Entitäten noch über interne Strukturen bekannt sind. Die einzigen in den Schnittstellen definierten Attribute sind eine ID zur eindeutigen Identifizierung im Modell und eine Typ-ID zur Zuordnung der Implementierung.

Aufgrund dieser Tatsache bestehen seitens des Komponentenmodells keinerlei Anfrage- oder Modifikationsmöglichkeiten der externen Entitäten. Es ist Aufgabe der nutzenden Anwendung hier eine geeignete Infrastruktur zur Verfügung zu stellen.

Nachdem die Realisierung der sowohl internen als auch externen Entitäten verdeutlicht wurde, folgt nun die Beschreibung des verwendeten ID-Konzepts und der Speicherung der Entitäten im Modell.

3.3.3 Identifizierung und lokale Speicherung von Entitäten

Die Identifizierung der Entitäten ist fundamentale Aufgabe im Modell. Sämtliche Anfragen an das Modell oder Modifikationen an der Struktur des Modells verwenden nicht die Entitäten selber sondern deren IDs. Vorteil hierbei ist die Unabhängigkeit der gesamten Infrastruktur des Komponentmodells von den Implementierungen der Entitäten. Bessere Wartbarkeit und geringere Fehleranfälligkeit sind die Konsequenz hieraus. Weiterhin erhält die nutzende Anwendung die Möglichkeit der granularen Zuordnung von Rechten. So kann bestimmten Teilen die Verwendung von beispielsweise der Builder-Schicht gewährt werden, ohne direkten Zugriff auf die Entitäten selber zulassen zu müssen. Proxy-Konzepte sind unter anderem auf diese Art mit wenig Aufwand umsetzbar.

Zur Vermeidung von Fehlern bei der Nutzung des Komponentenmodells wurden die IDs typisiert. Dies bedeutet, dass beispielsweise die Anfrage an ein Interface nur unter Verwendung einer Interface-ID gestellt werden kann. Um Anfragen die Möglichkeit der Übergabe einer beliebigen ID zu ermöglichen[1], erben alle IDs von einem Basisinterface.

Sowohl das Basisinterface aller IDs als auch das Basisinterfaces der Entitäten, in dem die lesende Zugriffsmethode auf die ID definiert ist, sind Bestandteil des vom Komponentenmodell unabhängigen Projektes `Palladio.Identifier`. Ziel dieser Ausgliederung ist die Wiederverwendbarkeit des ID-Konzepts und die dadurch gewonnene Kombinationsmöglichkeit sonst unabhängiger Projekte. Zusätzlich zu den beiden Interfaces werden Möglichkeiten zur Verwaltung von Entitäten und IDs zur Verfügung gestellt.

Das Basisinterface aller IDs definiert die ausschließlich lesbare Eigenschaft `Key` vom Type `System.String`. Dieser Umweg musste gewählt werden, um die oben erwähnte Möglichkeit zur Verwendung von typisierten IDs zu schaffen. Der Datentyp `System.String` ist weder durch ein Interface gekapselt, noch ist er durch Ableitung erweiterbar. Die ebenfalls im Projekt zu findene Standartimplementierung einer ID bedient sich der String-Repräsentation einer GUID als Key.

Das derzeitige Komponentenmodell stellt zur Erzeugung von IDs die von außen nutzbare Fabrik `ComponentModelIdentifier` zur Verfügung. Es wird von dieser Fabrik für alle

[1] QueryRepository bietet beispielsweise die Möglichkeit, die zu einer übergebenen ID gehörige Entität egal welcher Art zurückzugeben.

erzeugten IDs dieselbe Klasse `InternalEntityIdentifier` verwendet. Als Folge hieraus lassen sich IDs unterschiedlichen Typs ineinander casten. Dies sollte bei der Nutzung des Komponentenmodells keinerlei Probleme bereiten, da die Klasse selber nicht von außen sichtbar ist.

Wie in Kapitel 3.2 bereits angedeutet, werden die Entitäten zur Laufzeit der Anwendung direkt als Objektinstanzen gespeichert. Die hierfür verwendete gekapselte Hashtabelle ist ebenfalls Bestandteil des Identifier-Projekts. Jede Entität des Komponentenmodells ist in dieser Tabelle mit ihrer ID als Schlüssel zugreifbar. Iteratoren ermöglichen einfachen Zugriff auf alle Entitäten.

Nachdem das Konzept der internen und externen Entitäten inklusive der lokalen Speicherung und Identifizierung hinreichend dargestellt wurde, geht der folgende Abschnitt auf die Realisierung der Beziehungen zwischen den Entitäten ein.

3.3.4 Lokale Speicherung der Beziehungen

Die Speicherung der Beziehungen zwischen den Entitäten ist derzeit unter der Verwendung des *ADO.Net-Dataset* realisiert. Es handelt sich hierbei um eine Datenverwaltung ähnlich einer relationalen Datenbank, welche jedoch lokal im Kontext der Anwendung instanziert ist. Realisiert sind Datasets in einem Objektmodell bestehend aus Zeilen und Spalten innerhalb von Tabellen. In Datasets können ebenso wie in realtionalen Datenbanken Beziehungen und Einschränkungen definiert werden. Die Struktur eines Datasets wird in Form eines XML-Schemas definiert. Die enthaltenen Daten sind direkt in XML serialisierbar. Datasets lassen sich in typisiert und nicht typisiert unterteilen. Typisierte Datasets erweitern die Basisklasse `Dataset` um Zugriffsmethoden, welche direkt auf dem zugrundeliegenden Schema aufbauen[Mir].

Die Verwendung eines Datasets im Kontext des Komponentenmodells zeichnet sich durch viele Vorteile aus. Die vollständige Integration in die Anwendung erfordert keinerlei zusätzliche Infrastruktur. Die Objektnatur der Datasets ermöglicht Navigation über Tabellen hinweg durch untereinander verlinkte Tabellenzeilen. Anfragen an eine Tabelle können in SQL ähnlicher Form gestellt werden.

Nachteilig wirkt die fehlende Möglichkeit der Anfrage über mehrere Tabellen hinweg in Form von beispielsweise Joins. Hier müssen Tabellen einzeln abgefragt und die Ergebnisse für weitere Anfragen verwendet werden. Sinnvoll wäre weiterhin die Möglichkeit der freien Wahl der Datentypen in einem Dataset. Die bisher in einer Hashtabelle gehaltenen Entitäten (vgl. 3.3.3) könnten so direkt in eine Tabelle des Datasets integriert werden. Das Konzept der XML-Serialisierbarkeit von Dataset erschwert möglicherweise die Umsetzung einer solchen Erweiterung.

Es folgt die Beschreibung des für das Komponentenmodell entworfenen Schemas welches in Abbildung 2 dargestellt ist.

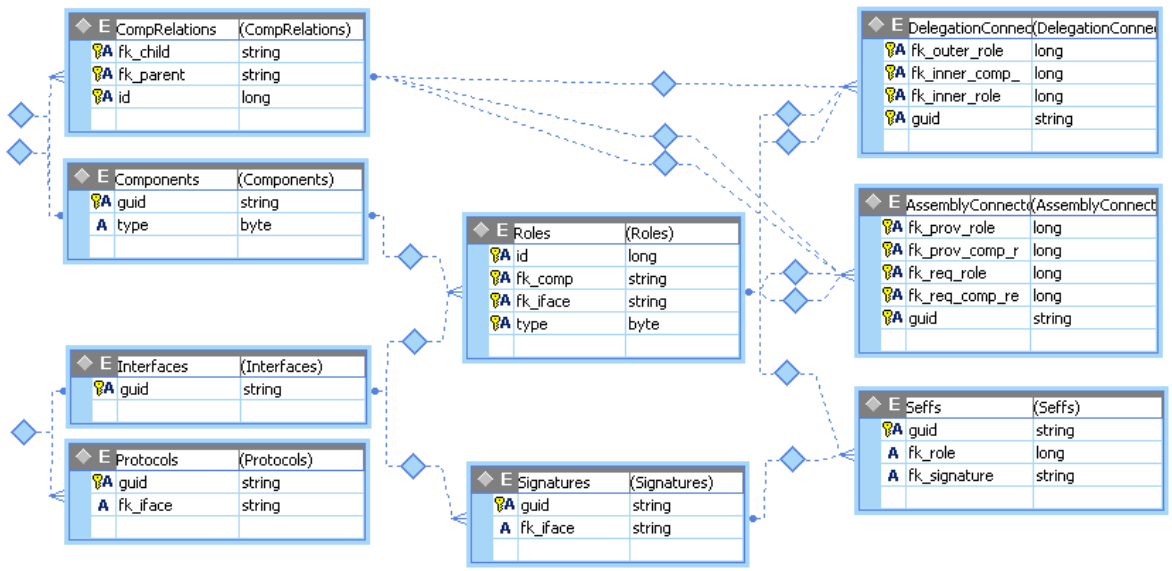

Abbildung 2: .NET Dataset des Modellkerns

Die anfänglich geplante strsikte Trennung von *Type-*, *Implementierungs-* und *Deployment*-Ebene gemäß [Pal] konnte aufgrund von Abhängigkeiten anderer Projekte in dieser Version des Modells nicht eingehalten werden. Aufgrund dieser Tatsache werden zukünftige Änderungen starken Einfluss auf das bisherige Schema haben. Unabhängig davon soll zum besseren Verständnis des derzeitigen Modells das aktuelle Schema erläutert werden. Dies kann ebenfalls als praktisches Beispiel zum Umgang mit Datasets im Kontext des Komponentenmodells dienen.

Um in relationalen Datenbanken Beziehungen zwischen Entitäten herstellen zu können, müssen die Entitäten selber in Form einer Tabelle in der Datenbank vorliegen. Analog verhält sich die Speicherung der Beziehungen in einem Dataset. Da die Entitäten selbst jedoch als Objektinstanzen in einer Hashtabelle (vgl. 3.3.3) vorliegen, sollten diese zur Vermeidung von Redundanz nicht vollständig in der Tabelle des Datasets gehalten werden. An dieser Stelle genügen Schlüssel- und Fremschlüsselattribute der Entitäten, um Beziehungen zwischen diesen modellieren zu können. Die Primärschlüssel der Entitätentabellen entsprechen jeweils den IDs der Entitäten und sind durchgehend mit `guid` bezeichnet. Die folgende Auflistung erläutert alle Tabellen des Datasets.

- **Components**
  Das theoretische Komponentenmodell enthält gemäß [Pal] zwei Arten von Komponenten, zusammengesetzte Komponenten (*composite components*) und Basiskomponenten (*basic components*). Beide sind Spezialisierungen von Komponenten, die sich jedoch nur in den Beziehungen zu anderen Entitäten unterscheiden. Die Art

und Anzahl der Attribute beider Komponententypen sind gleich. Somit bietet sich als Modellierungsart die Auflösung der Spezialisierung unter Verwendung eines Typ-Attributs (vgl. [EN02]) an. Somit setzt sich die Tabelle *Components* aus den Attributen `guid` (ID der Komponente) und `type` (Typ der Komponente) zusammen.

- **CompRelations**
  Zusammengesetzten Komponenten (*composite components*) können andere Komponenten enthalten, welche jedoch zusätzlich Bestandteil anderer Komponenten sein können. Die so entstandene *n zu m*-Beziehung der Komponenten untereinander wurde wie in [EN02] vorgeschlagen unter Verwendung einer zusätzlichen Tabelle realisiert. Diese Tabelle enthält ausschließlich Fremdschlüsselbeziehungen zu den beiden Teilnehmern der Beziehung. Im Falle des Komponentenmodells handelt es sich um die Tabelle `CompRelations` mit den Fremdschlüsselbeziehungen `fk_parent` und `fk_child` zu der Tabelle `Components`. Diese Tabelle enthält abweichend von [EN02] eine zusätzliche Spalte vom Typ *long*. Diese bildet den Primärschlüssel der Tabelle und dient der Modellierung von Beziehungen, welche bei der Beschreibung der Konnektor-Tabellen erläutert werden. Die Kombination der beiden Fremdschlüssel bildet einen Sekundärschlüssel der Tabelle, da die Beziehung zwischen zwei Komponenten nur einmalig möglich sein darf.

- **Interfaces**
  Neben dem Primärschlüssel `guid` sind in der Tabelle keine weiteren Attribute definiert.

- **Signatures**
  Signaturen sind Teil eines Interfaces. Ein Interface kann mehrere Signaturen enthalten, jede Signatur gehört jedoch ausschließlich zu einem Interface. Diese *1 zu n*-Beziehung wurde unter Verwendung des Fremdschlüssels `fk_iface` realisiert.

- **Protocols**
  Protokolle sind ebenso wie Signaturen Teil einer Schnittstelle (Interface). Die Schnittstelle kann auch hier mehrere Protokolle besitzen[2], wobei jedoch auch hier wieder ein Protokoll zu genau einer Schnittstelle gehört. Realisiert wurde diese Beziehung auch hier unter Verwendung des Fremdschlüssels `fk_iface`.

- **Roles**
  Diese Tabelle modelliert die *n zu m*-Beziehung zwischen Komponenten und Interfaces. Ein Interface kann Bestandteil mehrerer Komponenten sein. Eine Komponente kann ebenfalls mehre Interfaces besitzen. Es existieren zwei Arten von

[2]Das theoretische Modell sieht prinzipiell nur ein Protokoll für jede Schnittstelle vor. Dieses muss jedoch in unterschiedlicher Form (z.B. als Finite State Machines und Petri-Netz) möglicherweise redundant abspeicherbar sein.

Beziehungen zwischen Komponenten und Interfaces. Im ersten Fall werden die im Interfaces definierten Dienste von der Komponente angeboten (*ProvidesInterface*), im anderen Fall werden sie benötigt(*RequiresInterface*). Die Spezialisierung wird ebenso wie bei den Komponenten durch das Typ-Attribut `type` aufgelöst. Die Kombination der beiden Fremdschlüssel `fk_iface` und `fk_comp` mit dem Typ der Beziehung bildet einen Sekundärschlüssel der Tabelle. Primärschlüssel ist aus Gründen der besseren Modellierbarkeit der Beziehungen zu den Konnektoren eine eigene ID vom Typ *long*.

- **AssemblyConnections**
  Assembly-Konnektoren verbinden die Bedarfsschnittstelle (requires interface) einer Komponente mit der Angebotsschnittstelle (provides interface) einer anderen. Hierbei ist zu beachten, dass nicht die Komponenten selber sondern die Verwendung der Komponenten (modelliert durch die Vaterschaftsbeziehung zwischen Komponenten) zu verbinden sind, da dieselbe Komponente je nach Verwendung mit verschiedenen Komponenten verbunden sein kann. Somit erhalten Assembly-Konnektoren Fremdschlüsselbeziehungen zur Tabelle `CompRelations` und nicht zu `Components` selber. Die beiden Fremdschlüssel `fk_prov_role` und `fk_req_role` modellieren die Beziehungen zu den Schnittstellen, welche an der Verbindung beteiligt sind. Die Fremdschlüsselbeziehung in die Tabelle `Roles` enthält hierbei jedoch Redundanz, da die Komponenten sowohl in dieser Beziehung als auch in der Fremdschlüsselbeziehung zur Tabelle `CompRelations` definiert sind. Möglicherweise ist an dieser Stelle die Referenzierung auf die Schnittstellentabelle selber besser geeignet.

- **DelegationConnections**
  Im Gegensatz zu Assembly-Konnektoren können Delegation-Konnektoren ausschließlich in zusammengesetzten Komponenten (*composite components*) vorkommen. Sie verbinden die Schnittstelle einer äußeren mit der Schnittstelle einer inneren Komponente entweder auf der Angebots- oder auf der Bedarfsseite. Während Assembly-Konnektoren die Verwendung zweier Komponenten verbinden, gehören Delegation-Konnektoren zu den Komponenten selber. Es genügt also an dieser Stelle, die Beziehungen zu den Komponenten und Schnittstellen durch Fremdschlüssel-Beziehungen in die Tabelle `Roles` zu modellieren. Die zusätzliche Fremdschlüsselbeziehung in die Tabelle `CompRelations` zeigt auf die Kombination von innerer und äußerer Komponente, zwischen denen der Konnektor eine Verbindung bildet. Sie ist an dieser Stelle redundant und dient lediglich der Vereinfachung von Anfragen.

- **Seffs**
  Die Tabelle `Seffs` enthält die Schlüssel der Dienstbefarfsautomaten (*service effect*

specifications). Ein Dienstbedarfsautomat gehört zu einer Signature in einem Interface, welches von einer Basiskomponente (*basic components*) angeboten wird. Die Verbindung zur Kombination zwischen Interface und Komponente modelliert hierbei die Fremdschlüsselbeziehung in die Tabelle `Roles`. Analog hierzu zeigt die Beziehung in die Tabelle `Signatures` auf die Signatur, zu der der Automat gehört.

3.3.5 Schnittstelle nach außen

Alle Schnittstellen, welche den Zugriff auf den Modellkerns ermöglichen, befinden sich im Namensraum `Palladio.ComponentModell.ModelDataManagement`. Gekapselt ist der Kern durch das Interfaces `IModelDataManager`. Dieses bietet Zugriffsmöglichkeiten zu Bestandteilen des Komponentenmodells, welche direkt auf dem Dataset oder der Hashtabelle arbeiten. Hierzu gehören die Anfrageschicht, die Benachrichtigungsschicht und eine Schicht, welche Basisfunktionalität zum Verändern des Modells implementiert.

Die Anfrageschicht ist durch das Interface `IQuery` gekapselt und wird in Kapitel 7.1 detailiert beschrieben. Das Interfaces `IEventInterface` stellt Möglichkeiten der Registrierung für Ereignisse des Modells zur Verfügung. Eine Auflistung möglicher Ereignisse ist in Kapitel 6 zu finden. Veränderungen am Modell können unter Verwendung der im Interface `ILowLevelBuilder` definierten Methoden vorgenommen werden. Mit diesen Methoden ist der gesamte Aufbau eines Modells möglich. Es fehlt an dieser Stelle jedoch die Überprüfung der Korrektheit im Sinne des theoretischen Modells. Lediglich Verletzungen gegen die referentielle Integrität des zugrundeliegenden Datenbankschemas werden durch diese Schicht abgefangen. Für Erweiterungen eben beschriebener Art ist eine eigene Komponente (vgl. Kapitel *builder*) zuständig. Diese stellt neben flexiblen Überprüfungsmöglichkeiten zusätzlich eine der Struktur des Modells angepasste Methode zu dessen Veränderung zur Verfügung.

3.3.6 Implementierung

Die Implementierung der Hauptschnittstelle des Modellkerns übernimmt die Klasse `ModelDataManger`. Hier werden das typisierte Dataset (implementiert in der Klasse `ModelDataSet`) und die für die Speicherung der Entitäten zuständige Hashtabelle instanziert und gehalten. Ebenfalls werden beim Aufruf des Konstruktors die Implementierungen der im vorherigen Kapitel vorgestellten Zugriffsinterfaces instanziert und bei Bedarf von der entsprechenden Methode zurückgegeben.

Die Implementierung der Anfrage- und der Benachrichtigungsschicht wird in eigenen Kapiteln vorgstellt, die das Interface `ILowLevelBuilder` implementierende Klasse soll an dieser Stelle kurz erläutert werden. Für jede Entität des Kompomentenmodells existiert jeweils eine Methode zum Hinzufügen und eine zum Löschen. Die der Erzeugung

dienende Methode erwartet die Entität selber und Informationen über Beziehungen zu anderen Entitäten. Bevor eine Entität dem Modell hinzugefügt wird, erfolgt die Überprüfung von Einschränkungen, welche durch das Datenbankschema definiert sind. Hierzu wird eine zur Signatur identische Methode der Klasse `ModelConstrainsCheck` aufgerufen, welche die Einhaltung der Einschränkungen überprüft. Im Fehlerfall wird eine entsprechende Ausnahme ausgelöst. Sind Entität und Beziehungen überprüft, so werden alle Einträge im Dataset vorgenommen und die Entität der Hashtabelle hinzugefügt. Abschließend werden alle registierten Event-Listener benachrichtigt. Dem Löschen von Enitäten oder Beziehungen ist keine weitere Überprüfung vorangestellt. Existieren die zu löschenden Objekte nicht, so wird die Methode ohne Auswirkungen beendet. Ist durch den Löschvorgang die referentielle Integrität des Dataset gefährdet, so sorgt kaskadiertes Löschen für die Wahrung von Konsistenz. Ist dieses Verhalten nicht gewünscht, so ist eine entsprechende Überprüfung vor dem Aufruf der Methode durchzuführen. Die registrierten Event-Listener werden im Fall von kaskadiertem Löschen in einer Reihenfolge benachrichtigt, die zu jedem Event ein konsistentes Schema garantiert. Soll beispielsweise eine zusammengesetzte Komponente gelöscht werden, so werden erst alle enthaltenen Bestandteile der Komponente gelöscht[3], bevor sie selber entfernt wird.

Mit der Beschreibung der in dieser Version des Komponentenmodells implementierten Variante des Modellkerns schließt dieses Kapitel. Es folgen Hinweise zur Instanzierung und Nutzung des Modells unter Betrachtung einiger spezieller Entwurfsdetails.

4 Instanzierung des Modells

Das Komponentenmodell ist durch die im Namensraum `Palladio.ComponentModel` definierte Klasse `ComponentModelEnvironment` gekapselt. Diese läßt sich unter Verwendung des parameterlosen Konstruktors instanzieren. Mehrfache Instanzen des Komponentmodells sind problemlos möglich, da auf die Verwendung von Klassen gemäß dem statischen Singleton-Pattern []. Beim Entwurf von Erweiterungen, die direkt in das Komponentenmodell einfließen, ist dieses Konzept beizubehalten, um Kompatibelitätsprobleme zu vermeiden. Konzepte zur Synchronisation zweier Modelle sind in der aktuellen Version des Komponentmodells nicht vorgesehen. Bestehen Anforderungen dieser Art, so sind diese unter Verwendung der Benachrichtigungsmechanismen in Verbindung mit der Anfrage und Builder-Schicht zu realieren.

Nach der Instanzierung steht ein leeres Modell zur Verfügung. Wahlweise kann unter Verwendung der Builder (vgl. Kapitel 5) ein neues Modell erstellt oder ein persistent gespeichertes geladen werden. Alle Schnittstellen zu den jeweiligen Schichten sind über die das Modell kapselnde Klasse `ComponentModelEnvironment` erreichbar.

[3]Nach jedem Löschvorgang wird das entsprechendes Event ausgelöst.

Literatur

[EN02] Elmasri, Ramez und Sham Navathe: *Grundlagen von Datenbanksystemen.* Pearson Studium, 3., überarb. Aufl. Auflage, 2002.

[Mir] Mircosoft Corporation: *Einführung in Datasets.* MSDN Library.

[Pal] Palladio Research Group: *The Palladio Component Model.*

[Reu01] Reussner, Ralf H.: *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten.* Logos Verlag, Berlin, 2001.