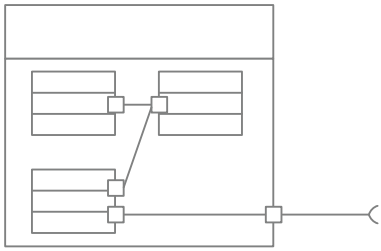
# 9 Kompositionsstrukturdiagramm



Ein Kompositionsstrukturdiagramm gibt Ihnen die Möglichkeit, die interne Struktur eines Classifiers (z. B. einer Klasse) sowie seine Interaktionsbeziehungen zu anderen Systembestandteilen zu beschreiben. Das Kompositionsstrukturdiagramm gibt Ihnen somit die Antwort auf die Frage: **„Wie sind die einzelnen Architekturkomponenten strukturiert und mit welchen Rollen spielen sie dabei zusammen?“**

# 9.1 Überblick

Was sind Kompositionsstrukturen?

Das Kompositionsstrukturdiagramm (Composite Structure Diagram) stellt Teile-Ganzes-Strukturen da. Um dies näher zu beleuchten, müssen wir zunächst klären, was solche Teile-Ganzes-Strukturen – auch Kompositionsstrukturen – genannt, überhaupt sind. Unsere Blickwinkel sind zum einen der *strukturell-statische* und zum anderen der *strukturell-dynamische*. Vereinfachend kürzen wir das Kompositionsstrukturdiagramm im Folgenden mit KSD ab.

### Strukturell-statische Kompositionsstrukturen: Strukturierter Classifier

Stellen Sie sich unter solchen Strukturen z. B. ein System vor, das in Subsysteme zerlegt wird. Jedes Subsystem zerfällt wiederum in Komponenten und diese in Subkomponenten. Subkomponenten zerlegen sich in Klassen, die wiederum Attribute enthalten. Sie gewinnen dadurch eine klassische Top-down-Struktur, die Sie in baumartiger Form, z. B. als Klassendiagramm, darstellen können:

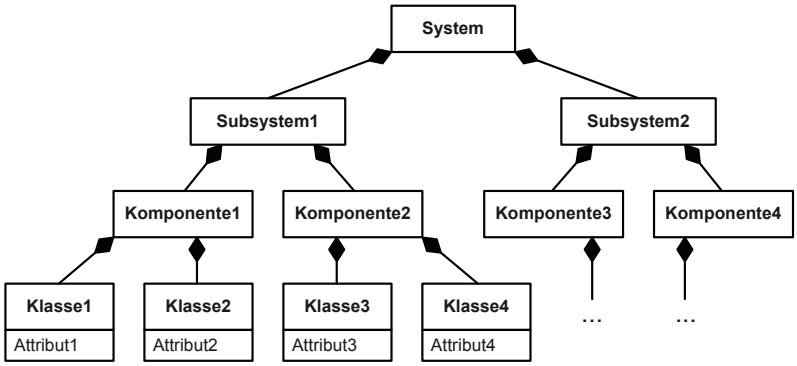



**ABBILDUNG 9.1** Eine Kompositionsstruktur, die die statische Struktur des Systems zeigt

In der Praxis finden Sie derartige Strukturen bei sehr großen Anwendungen (denken Sie an ein Flugzeug oder an ein Schiff) bereits auf Systemebene (Hardware) oder auch rein in Software gegossen (z. B. bei einer GUI-Anwendung: Anwendung – Fenster – Menüleiste – Menüeintrag – Label).

Architekturdiagramm

Da diese Kompositionsstruktur die Grundarchitektur eines Systems widerspiegelt, wird das KSD ab und an auch *Architekturdiagramm* genannt.

### Strukturell-dynamische Kompositionsstrukturen: Kollaboration

Hier stehen weniger die Gesamtstruktur und der Aufbau des Systems oder der Anwendung im Vordergrund, sondern vielmehr die *Funktion* und *welche Teile* des Systems zur Bearbeitung der Funktion nötig sind. Zur Realisierung einer Funktion oder eines Use-Cases sind in aller Regel mehrere Teile, z. B. Klassen, nötig. Die Objekte dieser Klassen arbeiten in gewisser Weise zusammen, rufen gegenseitig Funktionen auf, manipulieren Attribute, stehen irgendwie in Beziehung und realisieren somit die Funktion. Sie arbeiten zusammen (frz. collaboration) oder – in der UML/OO-Sprachwelt – bilden eine *Kollaboration*.

Bedenken Sie, dass eine Kollaboration aus der Menge aller Systembestandteile nur einzelne Teile herausgreift und hier auch nur bestimmte Attribute oder Beziehungen. Kollaborationen bilden also eine Art Sicht auf das System, so dass genau die Teile *in einer bestimmten Konfiguration* herausgegriffen werden, die zur Realisierung nötig sind.

Der methodische Ansatz bzw. die UML-Verwendung von Kollaborationen geht hier noch einen Abstraktionsschritt weiter: Anstatt die konkreten Systembestandteile zu nennen, definiert man zunächst die Aufgabe (den Use-Case) und modelliert die *Rollen,* die nötig sind, um diese Aufgabe zu erfüllen. Vergleichen Sie dies mit Ihrer Arbeit: in Ihrem Projekt muss eine bestimmte Aufgabe erledigt werden, dazu sind gewisse Schritte nötig, die von unterschiedlichen Rollen bearbeitet werden (z. B. brauchen Sie für die Erstellung eines Systems einen Analytiker, einen Designer, einen Programmierer und einen Tester). Wer letztendlich diese Rollen wahrnimmt, ob das Sie in einer One-Man-Show oder ein Team erledigen, wird durch diverse Faktoren bestimmt und kann von Projekt zu Projekt sehr unterschiedlich sein.

Kollaboration vs. Kollaborationsanwendung

Und genau so verhält es sich bei Kollaborationen: Sie definieren Rollen, die zusammenarbeiten und eine Aufgabe erfüllen. Welcher Systembestandteil, welche Klasse oder welche Operation die Rollen zur Laufzeit annehmen, kann variieren. Die UML unterscheidet also zwischen der Definition der Kollaboration (Engl. Collaboration) und ihrer Anwendung (Engl. CollaborationUse).

Was hat das aber mit Kompositionsstrukturen zu tun? Ganz einfach: Betrachten Sie die *Kollaboration als Ganzes* und die *Rollen als Teile*. Die Rollen sind in einer gewissen Konfiguration zusammengestellt (komponiert), um die Aufgabe zu lösen. Beachten Sie, dass es nur um die statische Zusammenstellung geht. Wie die Rollen wirklich zusammenarbeiten (Dynamik! Wer ruft wann was auf?), wird nicht dargestellt. Das bleibt dem Kommunikations- bzw. Sequenzdiagramm überlassen.

Sequenz-, Kommunikationsdiagramm
⇨ 15, 16

Um es konkret zu machen, betrachten Sie Abbildung 9.2, die sowohl die Definition einer Kollaboration `Arbeitsverhältnis` als auch deren Anwendung auf ein `Orchester` zeigt. Dabei wird dem `Dirigenten` die Rolle `Chef` und dem `Musiker` die Rolle `Mitarbeiter` zugewiesen. Die Symbolik erklären wir später.

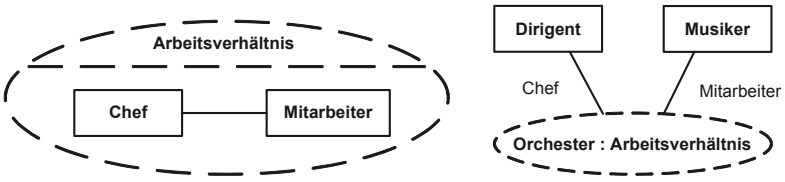



**ABBILDUNG 9.2** Eine Kollaboration und ihre Anwendung

Warum brauche ich einen neuen Diagrammtyp?

Sie werden sich jetzt vielleicht die berechtigte Frage stellen, warum man für die Darstellung von derartigen Strukturen nicht ein Klassendiagramm nimmt und warum denn dazu mit dem Kompositionsstrukturdiagramm ein eigener Diagrammtyp notwendig ist. Warum das Klassendiagramm mitunter eine sehr schlechte und aufwändige Wahl ist, erklären wir im nächsten Abschnitt.

### 9.1.1 Motivation und Hinführung

Um die Notwendigkeit und den Einsatzzweck des KSDs zu verstehen, müssen wir etwas ausholen. Der Name lässt zunächst darauf schließen, dass es hier um die Darstellung von Kompositionen, also enge Teile-Ganzes-Beziehungen, geht. Wir haben diese spezielle Assoziation bereits im Klassendiagramm erläutert. Abbildung 9.3 zeigt nochmals deren Verwendung im Rahmen des Klassendiagramms.

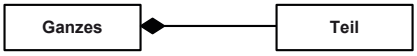



**ABBILDUNG 9.3** Die Komposition im Klassendiagramm

Schwächen der Kompositions-modellierung

Wenn Sie sich intensiv mit der Modellierung solcher *Kompositionsstrukturen* beschäftigen, werden Sie feststellen, dass das Klassendiagramm hier deutliche Schwächen aufweist. Während die Modellierung einfacher Baumstrukturen wie in Abbildung 9.1 keine Probleme bereitet, führt die Einführung *von Assoziationen zwischen den Klassen* bereits zu einem nicht vertretbaren Aufwand. Zudem können einem Klassendiagramm unterschiedliche, korrekte, aber nicht gewünschte Laufzeitstrukturen entstehen. Aus diesem Grund wird die Komposition im KSD präziser gefasst. Abbildung 9.4 zeigt die zugehörige Notation.


**ABBILDUNG 9.4** Die Komposition im KSD

Unterschiedliche Semantik der Kompositionen

Beachten Sie! Die Komposition aus Abbildung 9.3 im Klassendiagramm und die Komposition in Abbildung 9.4 im KSD sind inhaltlich (semantisch) **NICHT** gleich. Es sind nicht nur Darstellungsvarianten! Warum das so ist, möchten wir Ihnen anhand der Probleme des Klassendiagramms im Umgang mit Kompositionen verdeutlichen. Das KSD löst diese Probleme durch implizite Annahmen und Regeln.

Problem 1

**Problem 1:** Im Klassendiagramm ist nicht sichergestellt, dass das Ganze genau *seine* Teile gruppiert.

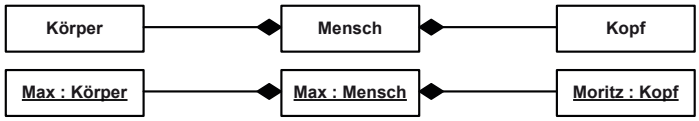



**ABBILDUNG 9.5** Der Kopf von Moritz passt nicht zum Max

Abbildung 9.5 zeigt im oberen Teil ein Klassendiagramm, das die Zusammensetzung eines Menschen aus `Körper` und `Kopf` darstellt. Das darunter abgebildete Objektdiagramm ist konform zum Klassendiagramm, hat aber den Nachteil, dass der `Kopf` von `Moritz` zum `Max` gehört. Zur Laufzeit verklinkt das Objekt `Max` also nicht „seine“ zugehörigen Objekte, sondern nur beliebige Objekte vom Typ `Körper` bzw. `Kopf`. Hier bleibt es also dem Programmierer überlassen, für eine korrekte Kombination zu sorgen. Das KSD der Abbildung 9.6

modelliert den gleichen Sachverhalt. Allerdings *erzwingt die Semantik* des KSDs, dass zum Erzeugungszeitpunkt des Mensch-Objekts auch automatisch die Instanzen von Körper und Kopf erstellt und untrennbar mit der Menschinstanz verbunden werden. Mit der Konsequenz, dass hier keine beliebigen Kombinationen erzeugt werden können und zur Laufzeit eine feste Strukturvorgabe entsteht. So sind die „Max"-Objekte in der gewünschten Weise gruppiert.

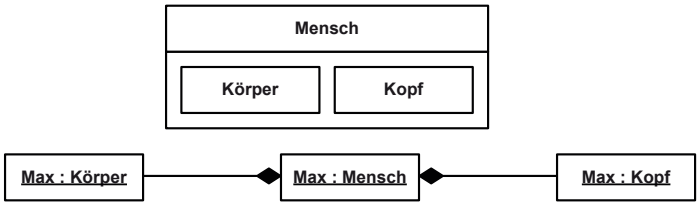



**ABBILDUNG 9.6** Hier passt der Kopf zum Menschen

Part

Sie sehen den kleinen, aber feinen und wichtigen Unterschied zwischen beiden Diagrammen. Beachten Sie außerdem, dass `Körper` und `Kopf` in Abbildung 9.5 „normale" Klassen darstellen. Hingegen stehen sie in Abbildung 9.6 immer in Beziehung mit der Klasse `Mensch`, man sagt auch: sie sind Teil von Mensch. Sie werden mit der Klasse erzeugt und auch zerstört. Deswegen stellen `Körper` und `Kopf` in diesem Fall keine Klassen, sondern so genannte Parts (Teile) dar. Parts sind also immer in einem Kontext zu sehen.

Übrigens: Man hätte den gleichen Effekt wie in Abbildung 9.6 mit einem gewissen Aufwand auch im Klassendiagramm erzeugen können.

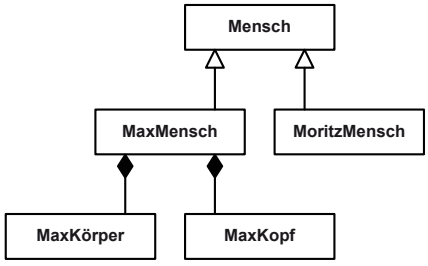



**ABBILDUNG 9.7** Klassendiagramm, das auch einen kompletten Max liefert

Problem 2

**Problem 2:** Im Klassendiagramm ist nicht sichergestellt, dass die richtigen Teile in einem Ganzen korrekt verbunden sind. Abbildung 9.8 zeigt eine gängige Situation in einem Personenfahrzeug. Sie haben 4 Fenster und 2 Kippschalter für die elektrischen Fensterheber. Es sind bewusst nur 2 Kippschalter, da im Fond des Fahrzeugs die Personen noch (manuell) kurbeln müssen, während Fahrer und Beifahrer den Komfort einer elektrischen Bedienung genießen. Die Abbildung zeigt ein konformes Klassendiagramm, das leider wieder nicht das Gewünschte wiedergibt.

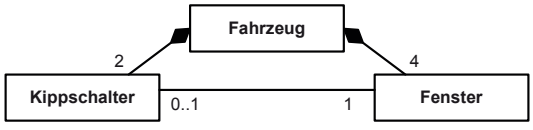



**ABBILDUNG 9.8** Fahrzeug mit Kippschalter für elektrische Fensterheber

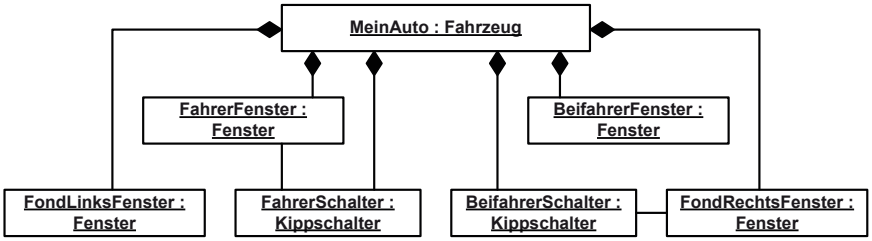



**ABBILDUNG 9.9** Zu Abbildung 9.8 konformes, aber nicht gewünschtes Objektdiagramm

Im Objektdiagramm von Abbildung 9.9 sehen Sie das Problem: Während der `FahrerSchalter` noch mit dem richtigen, dem `FahrerFenster` verbunden ist, ist der `BeifahrerSchalter` mit dem falschen `Fenster` verbunden. Das Problem aus Abbildung 9.5 existiert hier ebenfalls: Wer garantiert, dass das Fahrzeug genau seine Schalter und Fenster verknüpft?

Zusammenfassend lassen sich Mengenverhältnisse (Assoziationen, Links) zwischen Instanzen, die in einer Komposition verbunden sind, mit einem Klassendiagramm nicht hinreichend genau beschreiben. Abhilfe schafft hier wiederum das KSD, das die Semantik enger fasst und in Abbildung 9.10 die Verhältnisse klarlegt.

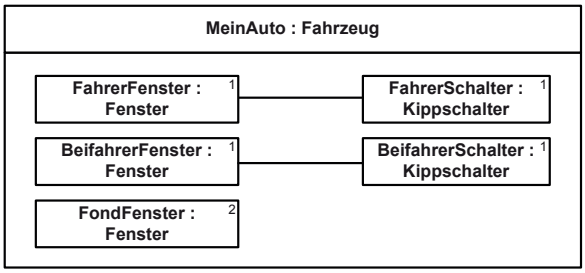



**ABBILDUNG 9.10** Kompositionsstruktur zum Fensterheber

Abbildung 9.10 drückt implizit nämlich Folgendes aus:

1. Sobald eine Instanz von `Fahrzeug` erstellt wird, werden auch 4 Instanzen von `Fenster` und 2 Instanzen von `Kippschalter` erzeugt. Die `Fahrzeug`instanz steht mit diesen 6 Instanzen in Kompositionsbeziehung.
2. Eine `Fenster`instanz nimmt die Rolle `FahrerFenster` ein, eine `Fenster`instanz die Rolle `BeifahrerFenster`, *zwei* `Fenster`instanzen je die Rolle `FondFenster`, eine `Kippschalter`instanz die Rolle `FahrerSchalter` und eine `Kippschalter`instanz die Rolle `BeifahrerSchalter`. Jede Instanz spielt also ihren Part.
3. Die `Fenster`instanz, die die Rolle `FahrerFenster` einnimmt, ist mit der `Kippschalter`-instanz, die den `FahrerSchalter` repräsentiert, verbunden. Die `Fenster`instanz, die die Rolle `BeifahrerFenster` einnimmt, ist mit der `Kippschalter`instanz, die den `BeifahrerSchalter` repräsentiert, verbunden. Die Verbindungen (Linien) werden Konnektoren genannt.

Mit diesen Regeln würde sich eindeutig das Objektdiagramm aus Abbildung 9.10 ergeben.

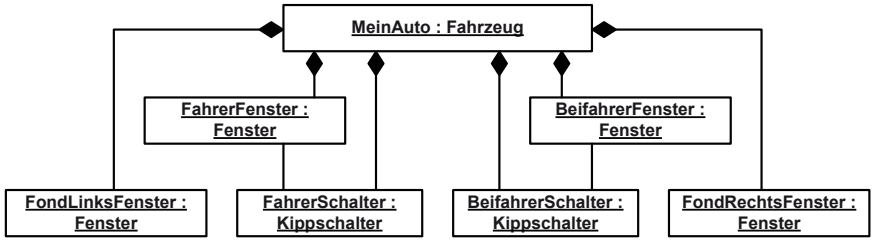



**ABBILDUNG 9.11** Zu Abbildung 9.10 konformes Objektdiagramm

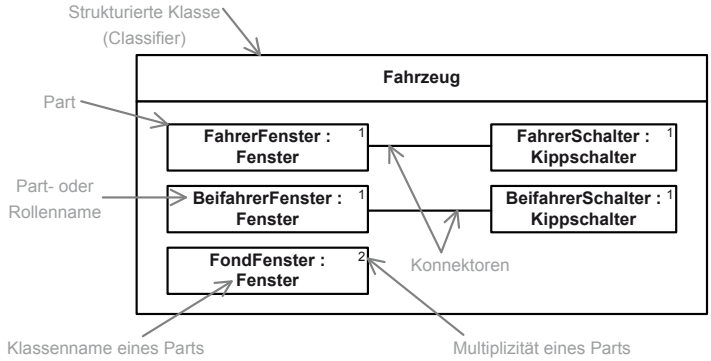



**ABBILDUNG 9.12** Die wichtigsten Notationselemente im KSD

Hauptelemente eines KSD

Die Abbildung zeigt zudem alle wesentlichen Elemente eines KSDs. Die bereits bekannten Parts mit Partname und Klassenname und die so genannten Konnektoren (siehe Abbildung 9.12).

Dabei sind die Bestandteile wie folgt zu interpretieren:

*Strukturierte Klasse (Classifier)* ist die Klasse bzw. der Classifier, der sich aus einzelnen Teilen zusammensetzt (komponiert).

*Part* ist ein Teil eines umschließenden (strukturierten) Classifiers, z. B. ist das Fenster ein Teil des Fahrzeugs. Sie können einen Part vereinfacht auch als Attribut einer Klasse in einem bestimmten Kontext sehen.

Der *Part- oder Rollenname eines Parts* beschreibt die *spezifische* Nutzung eines Parts einer Einheit, d. h. zum Beispiel wird eine Instanz von Fenster als FahrerFenster und eine Instanz als BeifahrerFenster genutzt.

Der *Klassenname eines Parts* repräsentiert den Typ des Parts, d. h. zum Beispiel eine Instanz vom Part Fenster ist vom Typ der Klasse Fenster.

Die *Multiplizität eines Parts*, bestimmt, wie viele Instanzen dieses Parts in einer Instanz des strukturierten Classifiers enthalten sind.

*Konnektoren* sind die Verbindungen zwischen *Parts*, ähnlich wie Assoziationen die Verbindungen zwischen Klassen darstellen. Da aber Parts Klassen in einem bestimmten Kontext unter einer bestimmten Konfiguration mit einer übergeordneten Einheit darstellen, sind die Konnektoren auch eine Verbindung in einem bestimmten Kontext und unterscheiden sich daher von Assoziationen. Sehr vereinfacht ausgedrückt, lässt sich sagen: Konnektoren sind

Assoziationen, für die bestimmte Regeln gelten bzw. die nicht so viele Freiheitsgrade wie Assoziationen zulassen. So verbindet die Assoziation in Abbildung 9.8 einen Kippschalter mit einem Fenster. Der Konnektor in Abbildung 9.9 hingegen verbindet genau einen Kippschalter, der die Rolle FahrerSchalter hat, mit dem Fenster, das das FahrerFenster repräsentiert.

Das KSD ist somit ein eigenständiges Diagramm mit eigenen Notationselementen. Es weist jedoch eine sehr starke Verwandtschaft mit dem Klassendiagramm auf. Streng genommen lassen sich die Restriktionen für die Instanzbildung und -verknüpfung, die das KSD vorgibt – mit sehr viel Aufwand – auch in einem Klassendiagramm erreichen. Dabei wird reichlich von Spezialisierungen und Eigenschaftswerten an Assoziationsenden Gebrauch gemacht. Das Klassendiagramm hat den Nachteil, dass zumindest die Klassenzahl wächst. Auf der anderen Seite modelliert es die Sachverhalte *explizit*, die das KSD *implizit* annimmt.

Typ- vs. Instanzebene

Alle bisher dargestellten KSDe zeigen keine Instanzen, sondern Typen und geben nur Regeln und Grenzen für die Laufzeit vor. Das KSD lässt sich aber auch auf die Instanzebene „ziehen“. Abbildung 9.13 zeigt das KSD von Abbildung 9.10 auf Instanzebene. Sie sehen die aus dem Objektdiagramm bekannte Unterstreichung und den sehr komplexen Namen, der sich wie folgt zusammensetzt:

```
Objektname / Partname : Klassen(Classifer)name
```

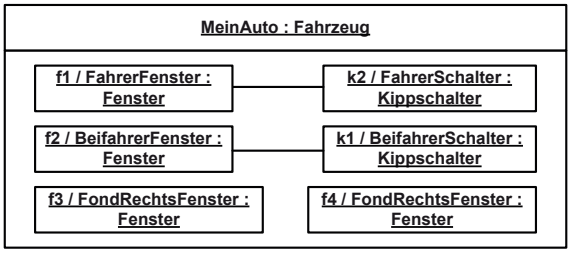



**ABBILDUNG 9.13** Ein Kompositionsstrukturdiagramm auf Instanzebene

Problem 3

**Problem 3:** Es ist nicht ausgeschlossen, dass komplett falsche Instanzkonstellationen durch *gemeinsam genutzte* Klassen entstehen.

Wir ändern dazu das obenstehende Klassendiagramm und nutzen die Klasse Fenster auch im Rahmen einer Klasse Haus. Zusätzlich wird das Fenster mit einer Fensterbank assoziiert.

Probleme der Wiederverwendung

Es ist offensichtlich, dass hier folgende Probleme (neben den bereits erwähnten) entstehen: Ein Fahrzeug könnte ein Fenster mit einer Fensterbank bekommen. Umgekehrt könnten natürlich die Fenster im Haus plötzlich mit Kippschaltern versehen werden. Auch Kombinationen davon sind denkbar (Fenster mit Bank und Schalter). Viel schlimmer ist, dass die Objekte aufgrund der nötigen zu 0..-Beziehungen auch dauerhaft getrennt instanziierbar sind und dabei keine Verbindung (Link) besitzen.

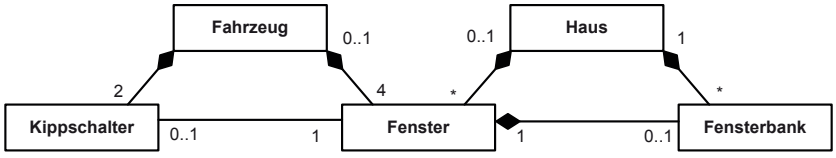



**ABBILDUNG 9.14** Nutzung der Klasse `Fenster` in verschiedenen Anwendungen

Für die Lösung dieses Problems wie auch für alle anderen geschilderten Probleme brauchen Sie *zusätzlich* zum Klassendiagramm ein oder zwei Kompositionsstrukturdiagramme, um die Verhältnisse zur Laufzeit klar zu legen.

### 9.1.2 Modellieren von Kollaborationen

Komposition von Rollen

Nachdem wir in der Hinführung eher die strukturell-statischen Kompositionen betrachtetet haben, wollen wir uns jetzt kurz die strukturell-dynamischen betrachten. Bereits das Beispiel aus Abbildung 9.15 zeigt die Analogie zu den bisher behandelten Kompositionsstrukturen. Statt eines Rechtecks für das Ganze wird eine gestrichelte Ellipse verwendet, die den dynamischen (Use-Case-)Charakter hervorhebt.

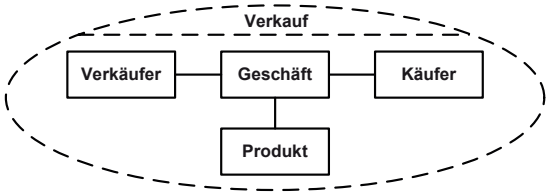



**ABBILDUNG 9.15** Definition der Kollaboration `Verkauf`

Hier wird nicht die statische Struktur abgebildet, sondern das statische Zusammenwirken von Rollen. Diese Rollen sind nötig, um einen `Verkauf` abzuwickeln. Eine Kollaboration definiert sozusagen einen Prototyp oder ein Muster, das Sie auf die Elemente Ihres Anwendermodells legen (Abbildung 9.16).

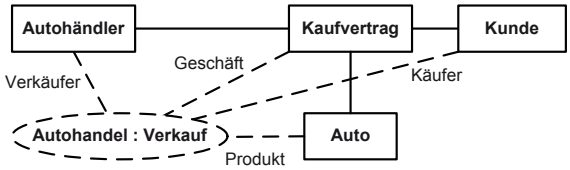



**ABBILDUNG 9.16** Anwendung der Kollaboration `Verkauf`

Die Parts einer Kollaboration stellen den Rollengedanken noch stärker in den Vordergrund. Die Parts werden hier auch durch Konnektoren verbunden. Der wesentliche Unterschied zu statischen Kompositionsstrukturen ist die so genannte Kollaborationsanwendung, also das Anwenden einer Kollaboration, wie in Abbildung 9.16 gezeigt. Ansonsten kennt die UML keinen Unterschied zwischen einer Kollaboration und einer eher statischen Kompositionsstruktur, beide werden als so genannte „Strukturierte Classifier“ bezeichnet. Beide repräsentieren Classifier und dürfen daher spezialisiert (Vererbung) werden. Dies ist in der Praxis aber eher unüblich. Zudem treten meist nur strukturierte Klassen und Komponenten und nicht alle Classifier auf.

Die an einer Kollaboration beteiligten Instanzen existieren bereits vorher und gehören somit nicht zur Kollaboration. Die Kollaboration sorgt lediglich dafür, dass die Instanzen

zusammengebracht und miteinander verbunden werden. Sie zeigt also nur, welche Instanzen beteiligt sind, welche Rollen diese – die oft durch die Schnittstellen beschrieben werden, die sie bietet und nutzt – bekleiden und welche Aufgabe sie durchführen müssen. Sie trifft keine Aussage darüber, welche Classifier zur Spezifizierung der Instanzen herangezogen werden.

### 9.1.3 Kapselung durch Ports

Eng mit dem Partkonzept verbunden sind die so genannten Ports. Dies sind streng genommen nur spezielle Parts. Sie werden daher auch mit dem umgebenden Classifier (Klasse) erzeugt und dürfen einen Typen bzw. eine Rollenangabe besitzen. Ports werden aber anders eingesetzt als Parts. Sie dienen in erster Linie nicht der inneren Strukturierung eines Classifiers, sondern definieren Kommunikationsschnittstellen, die den strukturierten Classifier (die strukturierte Klasse) nach außen abschotten (abkapseln). Die UML spricht daher auch von einem gekapselten Classifier (Encapsulated Classifier), der eine Spezialisierung des strukturierten Classifiers darstellt. Abbildung 9.17 zeigt einen gekapselten Classifier mit Ports, die als kleine Quadrate auf die Kanten des Classifiers oder seiner Parts gesetzt werden.

Encapsulated Classifier

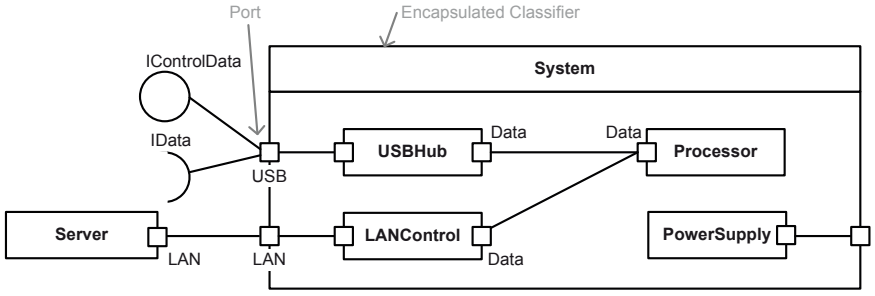



**ABBILDUNG 9.17** Anwendung von Ports bei einem gekapselten Classifier

Mit Ports wird ein Classifier von seiner Umgebung getrennt indem man ihn mit einem Zugangspunkt ausstattet, der die Interaktionen zwischen den internen Strukturen und der Umgebung des Classifiers durchführt. Um verschiedene Interaktionen auszuführen, kann ein Classifier auch über mehrere Ports verfügen. Welche Aktionen gestartet werden, hängt also davon ab, auf welchem Port eine Nachricht eingetroffen ist. Durch diese Trennung ist es möglich, den Classifier entsprechend seiner durch die Ports definierten Einsatzfähigkeiten wieder zu verwenden. Je nach Implementierung kann man den Port als eigenständige Instanz sehen, die beim Erzeugen seines Eigentümers (Classifier, Part) automatisch mit erzeugt wird und die Aufrufschnittstelle des Eigentümers umleitet. Sämtliche Kommunikation läuft dann über den Port. Dieser wird daher in der Literatur manchmal mit einem Briefkasten verglichen.

## 9.2 Anwendungsbeispiel

Abbildung 9.18 zeigt die Beschreibung eines `Einkaufs` in Form eines Kollaborationstyps und seiner Teilnehmer. Ein `Einkauf` hat, wie im oberen Teil der Zeichnung dargestellt, zwei Beteiligte, den `Kunden` und den `Laden`. Im rechten Teil des Bildes ist die interne Struktur eines `Hauses` dargestellt.

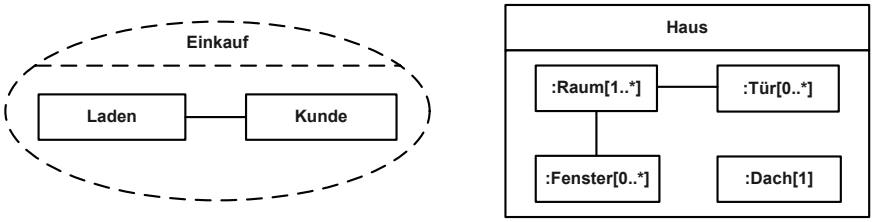



**ABBILDUNG 9.18** Kollaborationstyp `Einkauf` und strukturierte Klasse `Haus`

## 9.3 Anwendung im Projekt

Innerhalb eines Projektes kann das Kompositionsstrukturdiagramm in verschiedenen Entwicklungsphasen eingesetzt werden. Zunächst findet es seinen Platz in den frühen Entwurfsphasen zur abstrakten Dokumentation des Zusammenwirkens der einzelnen Architekturkomponenten. Anschließend können die im Kompositionsstrukturdiagramm dargestellten Systembestandteile iterativ ausdetailliert werden. Hierzu lässt sich ihre interne Struktur schrittweise verfeinern. Kollaborationen werden insbesondere bei der Umsetzung bzw. dem Design von Use-Cases ausgelegt.

### 9.3.1 Darstellung einer Architektur

Das KSD eignet sich auch zur Darstellung von Top-down-Strukturen, wie sie im Grobentwurf der Soft- und Systemarchitektur benötigt werden. Abbildung 9.19 zeigt ein solches „Architekturdiagramm“. Sie sehen mehrfach geschachtelte Parts, die in diesem Fall größere Einheiten und Komponenten repräsentieren. Häufig stehen Parts auch für ganze Schichten eines Systems. Die gesamte Kopplung wird intern über Ports gekapselt und extern über Ports mit Schnittstellen definiert.

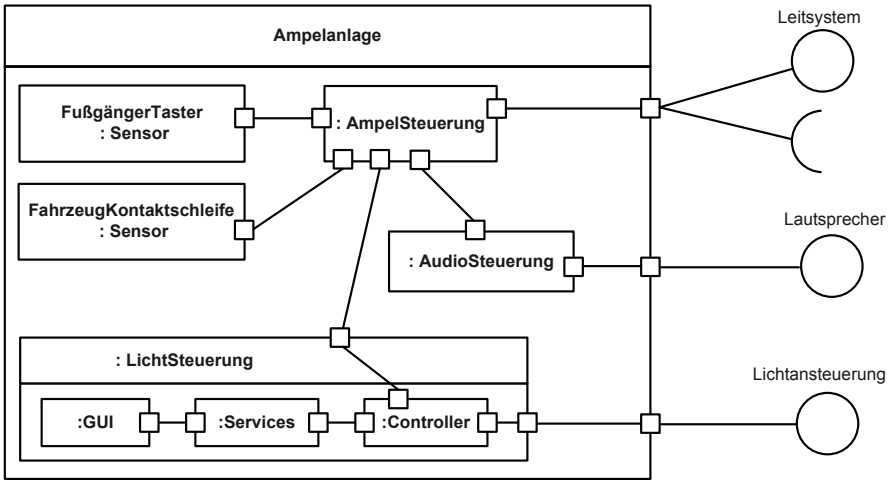



**ABBILDUNG 9.19** Grobarchitektur einer Ampelanlage

## 9.3.2 Struktur einer Use-Case-Realisierung

Design für Use-Cases

Insbesondere im Umfeld des Vorgehensmodells „Rational Unified Process“ ist der Begriff der Use-Case-Realisierung etabliert. Sie stellt den ersten Schritt im Design eines Systems dar, in dem der Designer oder Architekt einen Use-Case aus der Analyse realisiert. Das heißt, er versucht durch das Zusammenspiel von Komponenten bzw. Klassen den Ablauf des Use-Case nachzubilden. Typischerweise wird er dazu auch ein Sequenz- oder Kommunikationsdiagramm einsetzen. Allerdings wird er nie alle Elemente des Systems und die Elemente in verschiedenen Use-Case-Realisierungen auch in verschiedenen Rollen nutzen. Genau dies leistet eine Kollaboration, die bekanntlich einen speziellen Blick auf eine *statische Struktur* wirft. Abbildung 9.20 zeigt eine Kollaboration als „Verfeinerung“ eines Use-Cases. Die Kollaboration könnte auch zur Architekturdokumentation sehr generisch ausgelegt werden, um die Rollen von Komponenten zu zeigen. Typischerweise werden zusätzlich relevante Operationen eingetragen oder die Kollaboration mit einem Verhaltensdiagramm zur Use-Case-Realisierung (Struktur und Dynamik) vervollständigt.

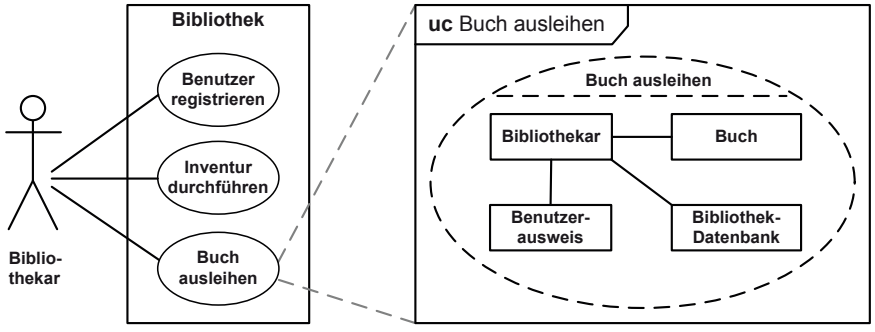



**ABBILDUNG 9.20** Kollaboration `Buch ausleihen` zur Realisierung des Use Cases

### 9.3.3 Kennzeichnung und Visualisierung von Design-Pattern

Die z. B. aus [GOF01] bekannten Design Pattern stellen erprobte Lösungen zu Problemstellungen des Softwaredesigns dar. Die dort definierten Muster beschreiben, bewusst generisch in Rollenform das Zusammenspiel verschiedener Elemente (meist Klassen) zur Lösung einer bestimmten Aufgabe. Die Muster sind dadurch auf gleichartige Probleme wiederholt anwendbar. Genau diese Idee liegt auch den Kollaborationen zu Grunde, so dass diese häufig zur Definition von Mustern herangezogen werden. Die Anwendung eines Design Patterns kann und sollte in einem Modell dokumentiert werden. Dies lässt sich durch eine Kollaborationsanwendung mit UML-Mitteln elegant umsetzen. Abbildung 9.21 zeigt die Definition des ModelViewControllers und seine Anwendung im Modell in sehr vereinfachter Form.

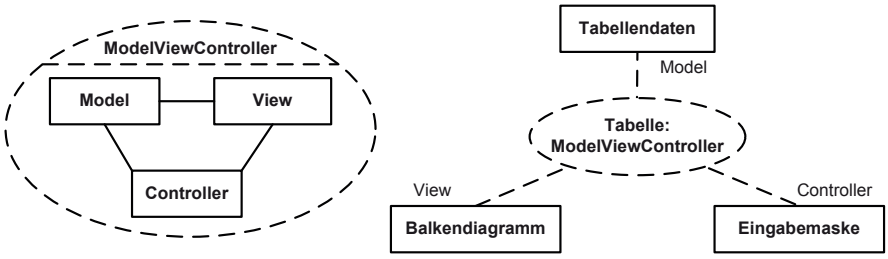



**ABBILDUNG 9.21** Design-Pattern ModelViewController

**Architectural Modeling Capabilities of UML 2**

*Bran Selic, Malina Software Corp, selic@acm.org*

By analogy, the term "architecture" is applied quite commonly when referring to software systems – although other analogies may be more appropriate. It denotes the top-level structural and behavioral organization of a system and the corresponding key design principles and constraints that are behind them. The analogy, of course, stems from building design, in which an architect defines the general form and broad outlines of an edifice, leaving the more detailed aspects (e.g., plumbing, electrical systems, etc.) to be worked out by various domain experts. (Perhaps a more apt analogy lies in the legal domain, where a "software architecture" can be likened to a *constitution* – the set of overarching legal principles to which all specific laws of a country must comply.) Just like their more traditional engineering counterparts, software architects rely heavily on models.

The need to more directly support certain key design patterns that appear in software architecture was identified as a primary requirement for UML 2.0. The following two areas were the subject of particular focus:

- A more refined capability to represent certain role-based structural patterns (as opposed to the more generalized class-based patterns).
- An expanded capability to represent complex end-to-end interaction sequences at higher levels of abstraction.

**Structure Modeling**

For the structural part of architectural modeling, the primary sources for the new constructs came from three distinct sources:

- The ACME language of David Garland and colleagues at Carnegie-Mellon University [GMW97]. The intent of this effort was to capture, in a single metamodel, the basic architectural constructs defined in various architectural description languages (ADLs).
- The UML-RT profile, defined by Selic and Rumbaugh [SR98]. This was an ADL profile of UML originally intended for specifying large-scale real-time software systems.
- The ITU-T SDL standard [ITU02], a domain-specific language for defining telecommunications protocols.

The basic modeling concepts in all three of these were quite closely matched and were easily consolidated and merged with existing role-based modeling capabilities found in UML collaboration diagrams. The result is the general concept of a *structured classifier*. This represents a topology of collaborating objects, called *parts*, whose interconnections are explicitly identified by communication channels called *connectors*.

This general concept is then refined in two ways. One refinement is a slightly evolved version of the traditional UML collaboration concept in which the parts represent abstract roles. The other refinement is the new concept of a structured class. In essence, a structured class is a UML class that can have both an internal and an external structure. The internal structure of such a class can be viewed as an encapsulated collaboration, whose parts and connectors are automatically created when an instance of the structured class is created. Since the parts of this internal structure can themselves be instances of a structured class, it is possible to realize hierarchical structures to an arbitrary depth.

The external structure of a structured class is represented by a set of ports. A *port* is a distinct interaction point on a structured class that provides a means for a structured object to *distinguish* between two or more, possibly concurrent, external transactions. Like the pins on a hardware chip, ports are dedicated and specialized interaction points that also serve to *insulate* the internal parts from explicit knowledge of external collaborators. This decoupling effect is fundamental to realizing a true component-based approach characteristic of sound software architecture.

Note that, from a formal language perspective, a structured class in UML is not differentiated from a "simple" class. A structured class just happens to be a standard UML class with structural attributes such as ports, parts, etc. It is even possible, for example, for a structured class to have a superclass that is not structured. In addition to the extended basic class concept, UML 2 also supports the component construct. This is simply a slightly refined version of the basic (structured) class concept, with a few additional features provided for backward compatibility with earlier versions of UML. However, these differences are of no particular relevance to software architects, so that they can use either UML classes or UML components in their work with no difference in expressive power. (A widespread misunderstanding is that only UML components provide the structural modelling capabilities required by architects.)

**Interaction Modeling**

For modeling behaviour at higher levels of abstraction, the primary architects' tool is the concept of interactions. The primary source for this, was another ITU-T MSC standard for describing interactions in a system of collaborating components [ITU04]. The essential innovation here is the ability to define complex interactions by combining simpler interactions. This kind of hierarchical composition can be applied repeatedly to an arbitrary level. Clearly, this is analogous to the way that the parts of a structured class can be instances of other structured classes. The metamodel of interactions in UML 2 defines a number of common ways of combining interactions into more complex interactions, including parallel composition, alternatives composition, looping constructs, etc.

**Conclusion**

Interaction modeling is closely coupled to structure modeling and, in particular, to structured classifiers, since the participants in an interaction can be identified with the parts of a structured classifier. This combination provides a comprehensive and very powerful set of modeling tools for the software architect capable of describing sophisticated software architectures at many levels of abstraction.

*Bran Selic retired from IBM as an IBM Distinguished Engineer in 2007 and is currently president of Malina Software Corp. He was one of the primary contributors to the original UML standard and also co-leader of the UML 2 design team. For five years (2003 – 2007), he was chair of the official UML 2 language task force at the OMG. He is still an active participant in this group as well as in the team working on a new simplified version of the standard.*

# 9.4 Notationselemente

## 9.4.1 Part

### Definition

A **part** references the properties specifying instances that the classifier owns by composition.

### Notation

Ein Part wird als Rechteck (siehe Abbildung 9.22) innerhalb eines ihn umgebenden strukturierten Classifiers/einer Klasse gezeichnet. Die Syntax für die Partbezeichnung lautet: `Part(Rollen)name : Parttyp [Mulitplizität]`. Alternativ kann die Multiplizität auch ohne eckige Klammern oben rechts im Eck eingetragen werden (Bild Mitte). Das Rechteck wird gestrichelt gezeichnet, wenn die Partinstanz *nicht* mit der Classifierinstanz in Komposition steht, sondern einem anderen Classifier zugeordnet ist. Auf Instanzebene wird der Instanzname des Parts, mit einem Slash abgetrennt, dem Partnamen vorangestellt (Bild rechts).

Multiplizität
⇨ 6.4.2

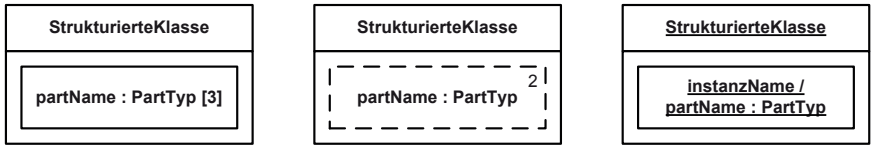



**ABBILDUNG 9.22** Ein Part innerhalb einer Klasse

### Beschreibung

Parts sind Teile eines Classifiers, die bei dessen Instanziierung mit instanziiert werden und danach eine bestimmte Rolle einnehmen. Sie strukturieren den Classifier und stehen mit ihm in Kompositionsbeziehung. Parts können durch Attribute oder andere Classifier gebildet werden. Für eine detaillierte Einleitung und die Regeln im Zusammenhang mit Parts lesen Sie bitte weiter oben Abschnitt 9.1.

⇨ 9.1

Parts sind insbesondere im Zusammenhang mit der Instanziierung/Destruktion des umschließenden Classifiers interessant. Sie werden in aller Regel mit dem Classifier oder kurz danach erzeugt und mit ihm zerstört. Die Menge der Partinstanzen richtet sich nach der Multiplizität des Parts. Durch Parts wird die Laufzeitkonfiguration eines strukturierten Classifiers definiert. So können auch feine Unterscheidungen zwischen den Rollen der Parts getroffen werden.

### Anwendung

Abbildung 9.23 zeigt die typische Zerlegung einer strukturierten Klasse `Anwendungsfenster` in mehrere Teile (Engl. Parts). Die einzelnen Teile werden teilweise nochmals strukturiert. Für das Laufzeitverhalten würden bei Erzeugung eines Fensters automatisch seine Bestandteile mit erzeugt.

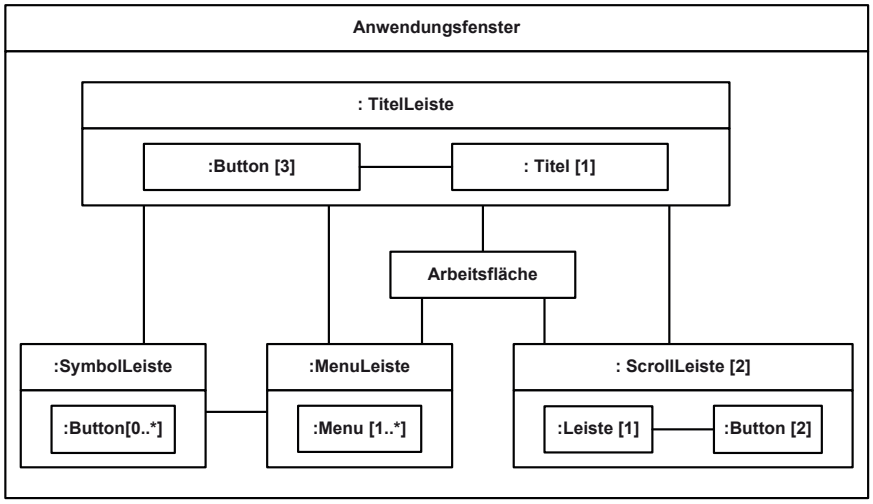



**ABBILDUNG 9.23** Fenster einer Anwendung

## 9.4.2 Konnektor

### Definition

Specifies a link that enables communication between two or more instances.

### Notation

Ein Konnektor (connector) wird ähnlich wie eine Assoziation als einfache durchgezogene Linie dargestellt. Die Beschriftung `Konnektorname:Konnektortyp` ist optional. Die Enden des Konnektors können wie auch die Assoziation mit Namen und Multiplizitäten versehen werden.

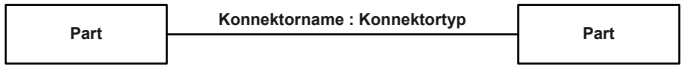



**ABBILDUNG 9.24** Notation eines Konnektors zwischen zwei Parts

### Beschreibung

Konnektor

Ein Konnektor spezifiziert eine Kommunikationsverbindung zwischen so genannten *verbindbaren Elementen (connectable elements*) und damit insbesondere zwischen Parts und Parts, Parts und Ports oder Ports und Ports bzw. genauer zwischen deren Instanzen. Der Unterschied zu einer Assoziation liegt in der Definition und Anwendung von Parts begründet (siehe dazu insbesondere Abschnitt 9.1). Parts sind Teile eines Classifiers, die bei Instanziierung eine *gewisse Rolle unter bestimmten Regeln* einnehmen. Ein Konnektor verbindet dabei genau Parts in einer bestimmten Rolle mit bestimmten Regeln. Eine Assoziation verbindet hingegen allgemeine Klassen, die in beliebigen Rollen agieren können. Denkbar wäre jedoch Folgendes:

⇨ 9.1

Zwei Klassen K1 und K2 bilden die Typen der Parts P1 und P2. Die Klassen sind durch die Assoziation A verbunden. Dann könnten die Parts P1/P2 durch einen Konnektor K *vom Typ* A verbunden sein. Eine Assoziation kann somit als Typ des Konnektors fungieren, denkbar wären aber auch Referenzen (Zeiger), Netzwerkverbindungen, Variablen usw.

Star-Connector-Pattern

Mit Multiplizitäten an den Konnektorenden werden die verbundenen Instanzmengen quantifiziert. Bei identischen Multiplizitäten und der Anzahl von Instanzen werden diese über das so genannte Star-Connector-Pattern verbunden. In Abbildung 9.25 ist jede Instanz `a` mit zwei Instanzen `b` verbunden und umgekehrt.

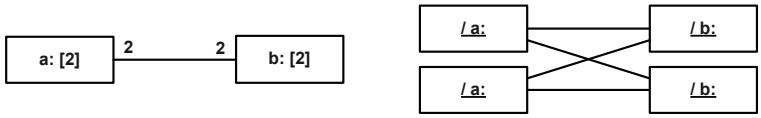



**ABBILDUNG 9.25** Star-Connector-Pattern

Array-Connector-Pattern

Sollte die Multiplizität eines Parts größer als die am Konnektorende sein, ergibt sich das Array-Connector-Pattern aus Abbildung 9.26. Dabei ist jede Instanz `a` mit genau einer Instanz `b` verbunden und umgekehrt.

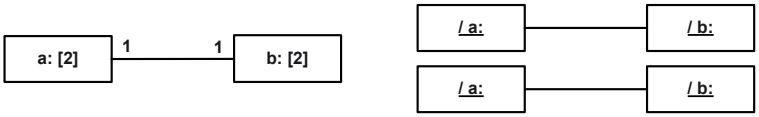



**ABBILDUNG 9.26** Array-Connector-Pattern

## Anwendung

Abbildung 9.27 zeigt die Teile einer `Telefonanlage`, die nur via Ports kommunizieren. Die Kommunikationskanäle sind durch unbeschriftete Konnektoren skizziert.

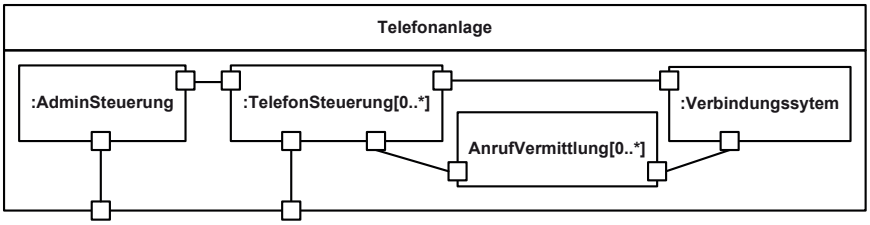



**ABBILDUNG 9.27** Telefonanlage mit Konnektoren zwischen Ports

### 9.4.3 Port

#### Definition

A **port** is a property of a classifier that specifies a distinct interaction point between that classifier and its environment or between the (behavior of the) classifier and its internal parts.

#### Notation

Ein Port wird als kleines Quadrat notiert, das auf der Umrandung eines Classifiers bzw. Parts angebracht ist. Optional kann der Portname und sein Typ angegeben werden. Mehrere unbenannte Ports drücken immer *unterschiedliche* Ports aus.

Ports dürfen mit beiden Schnittstellenarten versehen werden. Zudem wird ein Verhaltensport mit einer Linie zur Ellipse symbolisiert.

Schnittstelle
⇨ 6.4.4

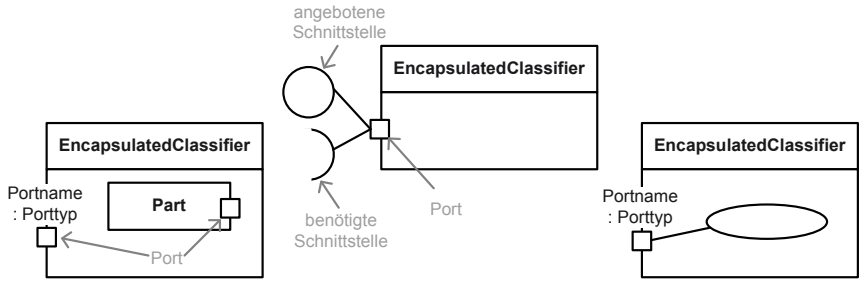



**ABBILDUNG 9.28** Notationen eines Ports

#### Beschreibung

Kommunikationsschnittstelle

Ein Port ist ein Kommunikationspunkt zwischen einem Classifier (einer Klasse) oder einem Part und seiner Außenwelt. Der Begriff Kommunikationspunkt ist dabei sehr abstrakt gefasst und kann in der Implementierung vollkommen unterschiedlich abgebildet werden. Typischerweise wird ein Port durch eine Klasse mit öffentlichen Operationen umgesetzt. Anstatt mit dem eigentlichen Classifier zu kommunizieren, wird die Schnittstelle (die Operationen) der Portklasse genutzt. Man kapselt den Classifier durch Ports ab. Dies gilt übrigens auch umgekehrt, falls vom Inneren eines Classifiers Daten oder Aufrufe nach außen erfolgen. Die Idee der Ports kommt ursprünglich aus der objektorientierten Modellierung von eingebetteten und Echtzeit-Systemen [SGW94]. Ziel ist es, eine Klasse oder eine Komponente auf ihre Ports zu reduzieren. Sind die Ports definiert, könnte jede *beliebige* Klasse diese Ports realisieren, ohne dass sich für die Außenwelt etwas ändert. Ports können dabei diverse Zusatzaufgaben wie Filterung, Caching oder Protokollüberwachung übernehmen. Bei der letzten Aufgabe lassen sie sich hervorragend mit einem Protokollzustandsautomaten verknüpfen. Der Automat definiert dabei die Reihenfolge der Operationen, die an einem Port aufgerufen werden dürfen.

Ein Port kann einen Aufruf über einen Konnektor an einen internen Part weiterleiten. Auch die Weitergabe an den umschließenden (gekapselten) Classifier ist möglich. Dessen Verhalten lässt sich durch ein Verhaltensdiagramm (z. B. Zustandsautomat) beschreiben. Derartige Ports werden auch *Verhaltensports* genannt, da sie die eingehende Kommunikation (z. B. in

Protokollzustandsautomat
⇨ 14.4.16
Signale
⇨ 15.4.3

Form von Signalen) direkt auf das Verhalten des Classifiers umsetzen. Der Signalfluss lässt sich sehr gut mit Informationsflüssen modellieren (siehe Abbildung 9.29).

Ports dürfen mit Schnittstellen versehen werden, die die bereitgestellten und angebotenen Operationen eines Ports definieren. Der Typ des Ports (z. B. eine Klasse, die den Port implementiert) muss entsprechend die Schnittstellenvereinbarungen einhalten. Ein Port stellt einen besonderen Part dar, der immer per Komposition mit seinem Classifier(Part) verbunden ist. Wird ein Classifier zerstört, so auch seine Ports und daran angeschlossene Konnektoren. Ports lassen sich, falls der zugehörige Classifier spezialisiert wird, ebenfalls spezialisieren und neu definieren.

### Anwendung

Informationsfluss
⇨ 6.4.15

Abbildung 9.29 zeigt einen DVD-Recorder, mit zahlreichen Ports (`VideoOut`, `USB`, ...) und Schnittstellen. Der Konnektor vom `AudioIn`-Port zum internen `A/D-Wandler` (Part) wurde mit einem Informationsfluss `Audiostream` versehen.

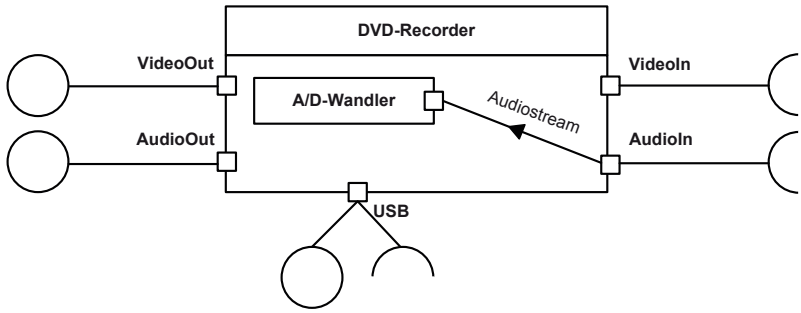



**ABBILDUNG 9.29** Ports an einem DVD-Recorder

## 9.4.4 Kollaboration

### Definition

A **collaboration** describes a structure of collaborating elements (roles), each performing a specialized function, collectively accomplishing some functionality.

### Notation

Eine Kollaboration wird durch eine gestrichelte Ellipse dargestellt, die durch eine weitere gestrichelte Linie in zwei Abschnitte geteilt wird. Im oberen Abschnitt wird der Name der Kollaboration angegeben. Im unteren Abschnitt können Sie dann die Modellelemente, die in diesem Kollaborationstypen zusammenarbeiten, mit Hilfe von Rollen und Konnektoren anführen.

Die in Kollaboration abgebildeten Classifier sind Teile (Parts) dieses Kollaborationstypen und werden daher lediglich mit ihrer Rolle und ihrem Namen versehen, eine weitere Detaillierung (etwa die Auflistung von Operationen) ist denkbar. Dazu wird aber meist die Alternativnotation (rechtes Bild) herangezogen. Da die Rollenangaben Parts sind, gilt die entsprechende Namenssyntax von Abschnitt 9.4.1.

⇨ 9.4.1

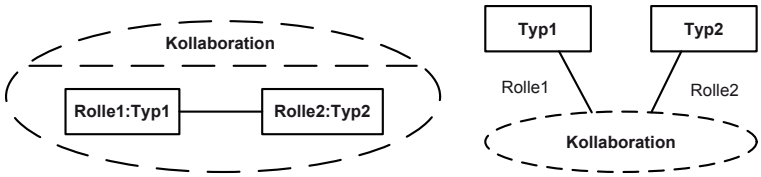



**ABBILDUNG 9.30** Notationen zur Definition einer Kollaboration

## Beschreibung

Eine Kollaboration (Collaboration) ist ein Modellelement, das eine Sicht auf zusammenspielende Modellelemente darstellt. Diese arbeiten zusammen, um eine gemeinsame Aufgabe zu bewältigen. Die Kollaboration ist eine Sonderform des strukturierten Classifiers (siehe dazu Abschnitt 9.1). Eine Kollaboration ist als solche nicht instanziierbar, sondern wird durch die einzelnen Instanzen der enthaltenen Modellelemente in ihrer jeweiligen *Rolle* repräsentiert.

⇨ 9.1

Die in einer Kollaboration verwendeten Rollen beschreiben bestimmte Merkmale, die eine Instanz besitzen muss, um an einer Kollaboration teilnehmen zu können. Dies müssen aber nicht zwingend alle Merkmale einer Instanz sein, es kann sich hierbei auch um eine Untermenge davon handeln, beispielsweise wenn die Instanz eine Schnittstelle implementiert. Dann besitzt die Instanz mindestens die Eigenschaften der Schnittstelle, kann darüber hinaus aber noch weitere Eigenschaften besitzen oder gar weitere Schnittstellen implementieren.

Abstrakte Sichtenbildung

In einer Kollaboration werden nur die für die Erfüllung des Ziels notwendigen Informationen dargestellt. Details werden oft bewusst außer Acht gelassen. Es ist daher auch möglich, dass ein Modellelement gleichzeitig in mehreren Kollaborationstypen auftritt oder dass verschiedene Kollaborationen dieselbe Menge an Modellelementen nur jeweils aus einer anderen Sicht beschreiben.

Beispiele und die Hintergründe zu Kollaborationen finden Sie in Abschnitt 9.1.

⇨ 9.1

## Anwendung

Die Abbildung zeigt die Definition der Kollaboration `Datensicherung` mit 4 Rollen. Die einzelnen Rollen (Parts) sind durch Konnektoren verbunden.

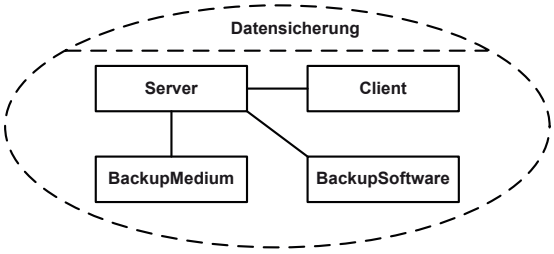



**ABBILDUNG 9.31** Die Definition der Kollaboration `Datensicherung`

### 9.4.5 Kollaborationsanwendung

#### Definition

A **collaboration use** represents the application of the pattern described by a collaboration to a specific situation involving specific classes or instances playing the roles of the collaboration.

#### Notation

Die Anwendung einer Kollaboration wird als gestrichelte Ellipse mit folgender Bezeichnung dargestellt: `Name der Kollaborationsanwendung : Kollaboration`. Auf die Elemente, die die jeweiligen Rollen der Kollaboration einnehmen, zeigen gestrichelte Pfeile, die in der Nähe des Elements mit dem Rollennamen versehen werden.

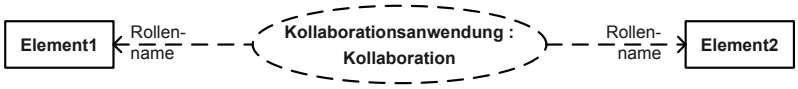



**ABBILDUNG 9.32** Notation einer Kollaborationsanwendung

#### Beschreibung

⇨ 9.4.4

Die *Kollaborationsanwendung* (Collaboration Use) stellt die Anwendung des durch eine Kollaboration (siehe Abschnitt 9.4.4) definierten Musters (Rollenspiels) auf das Modell dar. Dadurch wird ersichtlich, welches Element welche Rolle spielt und damit seinen Beitrag zur Erfüllung der Gesamtaufgabe beiträgt.

#### Anwendung

In der Abbildung sehen wir die Definition einer Kollaboration `CompositePattern` siehe [GOF01] im linken Bild und rechts ihre Kollaborationsanwendung `Dateisystem`. Dabei werden den Elementen die einzelnen Rollen `Blatt (Datei)`, `Komponente (Dateisystemelement)` und `Kompositum (Verzeichnis)` zugewiesen.

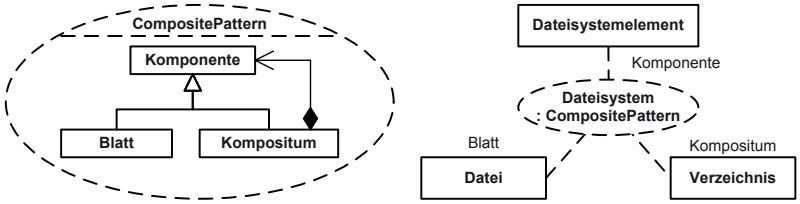



**ABBILDUNG 9.33** Kollaboration CompositePattern (links) und ihre Anwendung (rechts)