3 OO im Überblick

If you can't draw a diagram [of forces] of it, it isn't a pattern.

—Alexander, 79

3.1 Einführung

In diesem Kapitel werden einige grundlegende Bausteine vorgestellt, die zur Entwicklung von Softwaresystemen notwendig sind. Ich werde die Entwicklung der benutzten Implementierungsmodule verfolgen und dabei die standardisierte Unified Modeling Language (UML) als Schreibweise zur Systemdokumentation einsetzen. UML ist mehr als nur eine grafische Notation, sie definiert Syntax und Semantik, um Konzepte ausdrücken zu können; daher ist sie tatsächliche eine Sprache. Es ist wichtig, die Grundlagen zu legen, so dass in den folgenden Kapiteln davon ausgegangen werden kann, dass alle dasselbe Verständnis haben, dieselbe Terminologie verwenden und damit jeweils dasselbe meinen.

Schauen wir zuerst, wie sich die hauptsächlichen Programmeinheiten während der Entwicklung der Software-Entwicklungspraktiken geändert haben. Ursprünglich war die Haupteinheit einfach das »Programm« (siehe Abbildung 3.1). Schwierigere Probleme führten zu größeren Programmen.

Abbildung 3.1: Ein einfaches Programm

Als die zunehmende Größe der Programme sie immer schwieriger wartbar machten, wurde die Idee geboren, sie in größere Funktionseinheiten, »Module« genannt, aufzuteilen. Dies führte zu strukturierten und Top-Down-Designpraktiken (Abbildung 3.2).

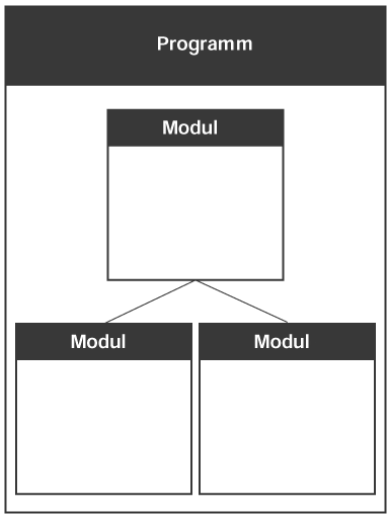

Abbildung 3.2: Strukturiertes Design

In den späten 60ern machte David Parnas einige Beobachtungen über die Zerlegung von Systemen in Module. Der Schlüssel zur Erstellung einer weniger komplexen Software lag darin, Funktionen und ihre Daten zusammenzulegen. Dies war der Anfang der Kapselung, einem essenziellen Element der Entwicklung von OOD-Techniken. Es führte zur Modellierung des abstrakten Datentyps (ADT) (Abbildung 3.3), der die Grundlage des objektbasierten Designs bildet. Dieser Ansatz erlaubt es, den internen Zustand oder interne Daten zu verstecken und den Zugriff auf Operationen zu beschränken, die den Zustand setzen oder lesen. Auf diese Weise kann man die Abbildung der Daten ändern, ohne den Klienten (oder Anwender) des ADT zu beeinflussen. Operationen werden dann ausgeführt, wenn eine Nachricht von einem Klienten empfangen wurde.

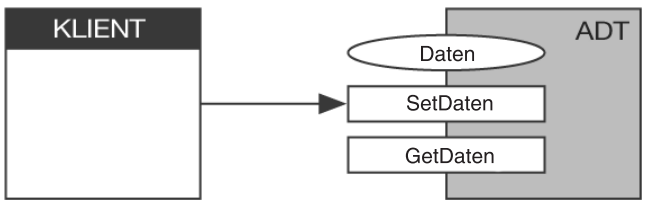

Abbildung 3.3: Abstrakter Datentyp

Meinung

Es ist für mich fast unvorstellbar, dass es beinahe 15 Jahre her ist, als ich zum ersten Mal mit einer objektorientierten Machbarkeitsprüfung zu tun hatte (Ich verkrafte das Älterwerden durch Verleugnen.). Damals untersuchten viele Organisationen die Nutzung von Objekten durch kleine Projekte, um die Technologie zu »verifizieren«. Dieser Ansatz (der in den letzen Jahren auch bei Mustern angewendet wird) funktioniert ziemlich gut, wenn das Team, das den Prototypen erstellt, die zu untersuchende Technologie korrekt anwenden kann. Verschiedene Organisationen, mit denen ich zusammenarbeitete, statteten diese Projekte erst einmal mit Personen aus, die kein echtes Verständnis der Objekttechnologie hatten; daher zeigten diese Untersuchungen, dass Objekte im Umfeld X nicht anwendbar waren.

Ich habe Ähnliches in den letzten Jahren bei Mustern beobachtet. Eine Gruppe, mit der ich zu tun hatte, stoppte die Nutzung von Mustern, nachdem einer der Entwickler sich so in Muster verliebt hatte, dass er versuchte, alle 23 Muster in einem einzelnen Programm anzuwenden. Natürlich widerspricht dies dem ganzen Gedanken der Entwurfsmuster, der auf dem Kontext fußt, und zeigt nur die Wissenslücken der Gruppe. Allerdings führte dies dazu, dass das Management der Idee von Mustern sehr kritisch gegenüberstand. Es hat mich einige Mühe gekostet, diesen Schaden zu beheben. Ich empfehle diese Machbarkeitsstudien auch heute noch zum Sammeln von Erfahrung. Ich empfehle allerdings immer, einen externen Experten dabei zu haben, der dabei hilft, das Projekt zu starten.

Man kann dieses Konzept erweitern, indem man eine Sammlung von zusammenhängenden Operationen zusammenfasst und damit die Signatur oder den »Typ« definiert. Dies wird allgemein als Schnittstelle (engl. Interface) bezeichnet. Diese wird in UML als dreigeteiltes Rechteck dargestellt und zeigt den Namen (obligatorisch), Attribute (normalerweise nicht in einer Schnittstelle) und die Operationen(optional), wie in Abbildung 3.4 zu sehen ist.

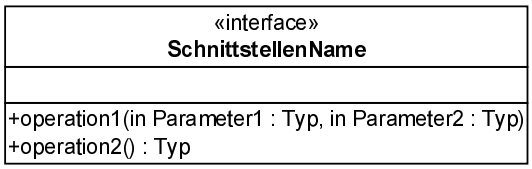

Abbildung 3.4: Schnittstellenschreibweise

Die Schnittstelle definiert nur die Signatur. Man muss dann noch die Implementierung dieser Signatur beschreiben, die man dann »Klasse« nennt. Es kann mehrere Klassen geben, die eine oder mehrere Schnittstellen implementieren. Dies kann man in UML

ähnlich einer Schnittstelle ausdrücken, als dreigeteiltes Rechteck mit Namen (obligatorisch), Attributen (optional) und Operationen (optional), wie in Abbildung 3.5 gezeigt.

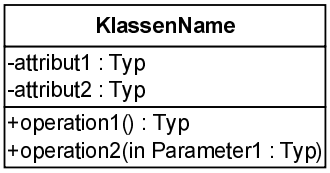

Abbildung 3.5: Klassennotation

Man kann auf zwei Arten darstellen, dass eine Klasse eine Schnittstelle implementiert. Dies kann einmal durch eine gerichtete Linie von der Klasse zur Schnittstelle geschehen (Abbildung 3.6). Diese Linie ist gestrichelt, um zu zeigen, dass eine Schnittstelle implementiert wird (eine durchgezogene Linie bedeutet Klassenableitung).

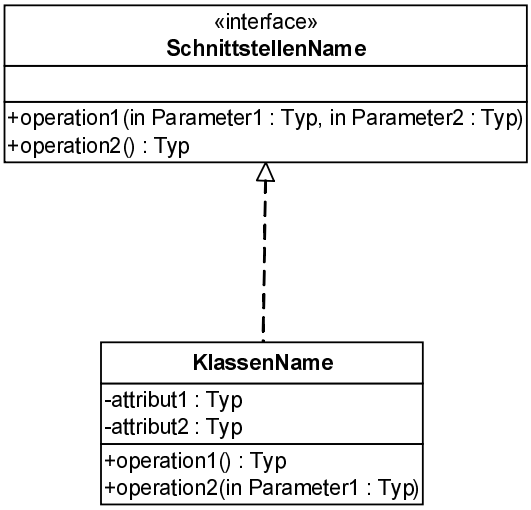

Abbildung 3.6: Eine Klasse, die eine Schnittstelle implementiert

Die zweite Möglichkeit zu zeigen, dass eine Klasse eine Schnittstelle implementiert, ist die kanonische Weise, mit der durch eine Klasse oder eine andere Entität implementierte Schnittstellen als lutscherähnliche Auswüchse dargestellt werden (Abbildung 3.7).

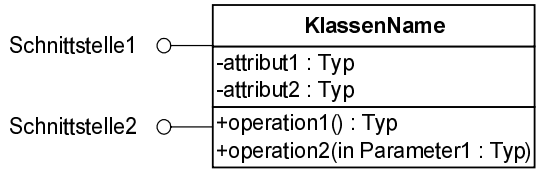

Abbildung 3.7: Kanonische Form der Implementierung einer Schnittstelle

Programmiersprachen

Letztens erlebte ich eine Überraschung bei einem Vorstellungsgespräch für die Position eines normalen Softwareingenieurs. Ich stellte die Frage »Wie implementieren Sie ein Interface in C++?«. Die schockierende Antwort lautete: »C++ unterstützt keine Interfaces; dies tut nur Java.«. Es wurde mir schnell klar (als ich weiterbohrte), dass vielen das Verständnis für den konzeptionellen Unterschied zwischen Design und direkter Unterstützung durch eine Programmiersprache fehlt. In den ersten Versionen von COBOL gab es keine direkte Unterstützung für strukturierte Programmierung, es ließen sich aber trotzdem Techniken zur Unterstützung dieses Modells finden. Im Falle von C++ und Java sind die Sprachunterschiede nicht so groß, wie viele Leute meinen (wobei die von vielen Leuten nicht erfassten Unterschiede zwischen Betriebssystem, virtueller Maschine und Bytecode recht groß sind!). Es folgt eine Auflistung von einfachen Konzepten und wie sie in C++ und Java realisiert werden können. Es sollte klar sein, dass Java, da es nach C++ entwickelt wurde, mehr semantische Möglichkeiten hat, die verbreiteten Konzepte aus C++ auszudrücken. Dieser semantische Reichtum erleichtert die Arbeit eines Entwicklers. Daher halte ich Java für eine deutliche bessere Sprache zum Vermitteln von Inhalten, auch wenn das Endprodukt in C++ erstellt werden mag.

	C++	**Java***
Schnittstelle (Interface)	Benutzen Sie eine abstrakte Klasse.	Benutzen Sie das Schlüsselwort *interface*.
(Abstrakte) Klasse	Benutzen Sie (mindestens) eine pure virtuelle Funktion.	Benutzen Sie das Schlüsselwort *abstract* für die Klasse (oder für Methoden).
(Konkrete) Klasse	Benutzen Sie das Schlüsselwort *class* (ohne pure virtuelle Funktionen).	Benutzen Sie das Schlüsselwort *class* (ohne abstrakte Funktionen).
Parametrisierung	Benutzen Sie das Schlüsselwort *template* (oder Makros).	Keine Unterstützung in der Sprache (häufig funktionieren gemeinsame Basisobjekte).

* Wir betrachten hier nur diese Kernkonzepte. Die wesentlich gravierenderen Unterschiede liegen in Dingen wie der virtuellen Maschine, Garbage Collection und Bibliotheken.

3.2 Vererbung

Vererbung ist eine Möglichkeit, Generalisierung oder »ist eine Art von«-Beziehungen zu beschreiben. Wenn man also sagt, Rechteck erbt von Form, behauptet man, dass ein Rechteck eine Art von Form ist.[1]

Ich vergleiche Vererbung gerne mit Zeichensätzen in einer Textverarbeitung. Die meisten Leute erinnern sich noch an das erste Dokument, das sie erstellt haben, nachdem sie die Möglichkeiten von Zeichensatzeinstellungen entdeckt hatten. Es verstieß offensichtlich gegen das erste Gesetz des Desktop-Publishings: Dein Dokument soll nie wie ein Erpresserbrief aussehen. Das Gute an diesem Phänomen war, dass normalerweise der erste Leser dies bemängelte und auf den Missbrauch dieses Werkzeugs hinwies. Nun, etwas Ähnliches passiert bei der Nutzung von Vererbung. Der Code-Review-Prozess hätte, genau wie das Korrekturlesen des Dokuments, herausfinden müssen, dass das Werkzeug falsch angewendet wurde. Häufig genug hat der Prüfer jedoch keine bessere Vorstellung von der richtigen Nutzung des Werkzeugs als der ursprüngliche Programmierer. Dies führt zu einem ernstzunehmenden Missbrauch der Vererbung, der unglücklicherweise dazu führt, dass viel von den allgemeinen und Modellierungsvorteilen dieses Werkzeugs verloren gehen.

Die »richtige« Anwendung von Vererbung wird häufig nicht erkannt. Vererbung sollte nur benutzt werden, wenn tatsächlich eine »ist eine Art von«-Beziehung vorliegt. Eine gute Möglichkeit zur Prüfung bietet Liskovs Ersetzungsprinzip [Lis, 88]. Grundsätzlich gilt, dass Vererbung anwendbar ist, wenn eine Unterklasse (oder ein Untertyp) oder eine Implementierungsklasse in jeder beliebigen Situation mit ihrer Basisklasse austauschbar ist. Um dies zu erreichen, darf die Unterklasse keine zusätzlichen Einschränkungen denen ihrer Oberklasse hinzufügen. Abbildung 3.8 zeigt ein Beispiel für »schlechte« Vererbung.

Ein klassisches Beispiel sind Quadrat und Rechteck. Man könnte eine Operation auf einem Quadrat definieren, die die Höhe und Breite durch die Seiten eines Quadrats abbildet.

So weit so gut. Doch nun betrachte man den folgenden Pseudocode, der ein Rechteck verbreitern soll, bis seine Breite größer als seine Höhe ist:

```
void groesseAendern(Rechteck r)
{
   while (r.getHoehe() <= r.getBreite())
   {
      r.setBreite(r.getBreite() + 1)
   }
}
```

1 Für diese Betrachtung ist es wichtig, zwischen der mathematischen und der programmtechnischen Definition zu unterscheiden. Mathematisch ist ein Quadrat ein Rechteck. Wie wir in Kürze sehen werden, ist dies programmtechnisch nicht so klar.

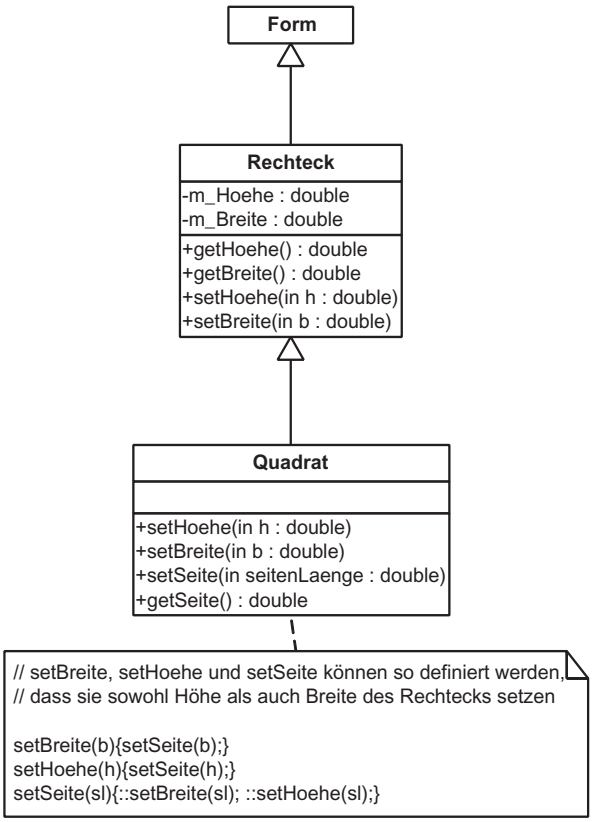

Abbildung 3.8: Eine »schlechte« Anwendung von Vererbung

Wenn dieser Code für ein Rechteck aufgerufen wird, arbeitet er prima, wenn wir ihn allerdings mit einem Quadrat benutzen, funktioniert er nicht wie erwartet. Die Breite wird immer größer werden, bis sie überläuft, da durch einen Seiteneffekt unserer Implementierung die Höhe mit vergrößert wird. Das Problem liegt darin, dass ein Quadrat mehr Beschränkungen als ein Rechteck unterworfen ist, bei dem die Seiten ja unterschiedlich lang sein können, was zu einem Dilemma führt. Entweder erlaubt man, die Seiten des Quadrats unabhängig voneinander zu setzen (was die Definition von Quadrat verletzt) oder dokumentiert Einschränkungen bei der Benutzung eines Quadrats. Dies bedeutet leider in beiden Fällen, dass jeder Aufrufer einer Funktion mit einem Rechteck als Parameter alle Details der Ausführung dieser Funktion kennen muss. Entsprechend muss man die Kapselung durchbrechen, die man eigentlich erreichen möchte.

Zusätzlich gibt es Datenelemente im Rechteck, die nicht Teil eines Quadrats sind. Es werden doppelt soviel Daten wie nötig gespeichert (Höhe und Breite statt nur Kantenlänge). Dies führt zu einer generellen Richtlinie für die Arbeit mit Vererbungshierarchien.

Als wichtige Richtlinie bei der Vererbung sollte man gemeinsamen Code möglichst oben in der Hierarchie ansiedeln, damit er wiederbenutzt werden kann (im Unterschied zu Datenelementen gibt es keine zusätzlichen Kosten je Instanz für Code, der nicht benutzt wird) und Datenelemente so weit wie möglich unten anordnen (siehe Abbildung 3.9). Dies stellt sicher, dass man keine Extrakosten für unbenutzte Datenelemente hat, aber trotzdem die Nutzung gemeinsamen Codes maximiert.

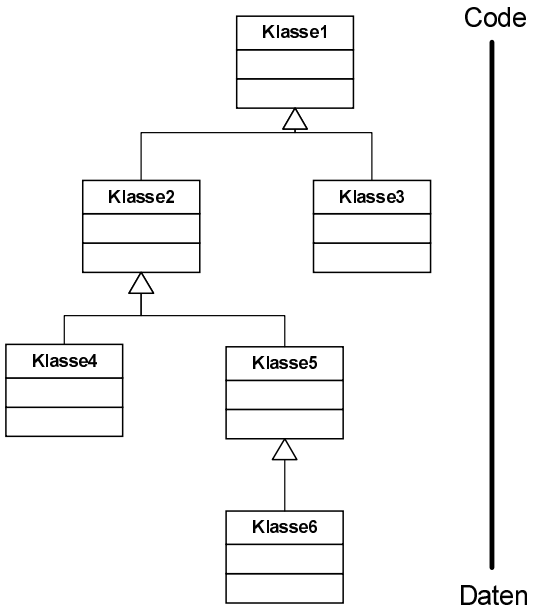

Abbildung 3.9: Code sollte immer möglichst weit oben in der Hierarchie angesiedelt sein und Daten möglichst weit unten.

Eine bessere Lösung wäre eine gemeinsame Polygonklasse, um die allen Polygonen gemeine Definition von Höhe und Breite als abstrakte getHoehe()- und getBreite()-Methoden zu enthalten (siehe Abbildung 3.10). Man muss sicherstellen, dass (wie oben beschrieben) die Datenelemente im kleinsten gemeinsamen Nenner gespeichert werden, in diesem Fall im Quadrat und Rechteck selbst

Zwei andere zentrale Konzepte sind abstrakte und konkrete Klassen. Eine abstrakte Klasse definiert eine gemeinsame Basisklasse, die nicht instanziiert werden kann. Dies bedeutet, dass keine Objektinstanzen der abstrakten Klasse erzeugt werden können. Eine abstrakte Klasse kann nur eine Basis definieren, von der andere Klassen abgeleitet werden können. Eine konkrete Klasse ist jede nicht abstrakte Klasse. Sie kann zur

Erzeugung einer Instanz benutzt werden. Abstrakte Klassen stellen häufig Konzepte oder Generalisierungen wie Form oder Fortbewegungsmittel dar. Konkrete Klassen sind eher spezifischere Entitäten wie Quadrat oder Toyota Celica.

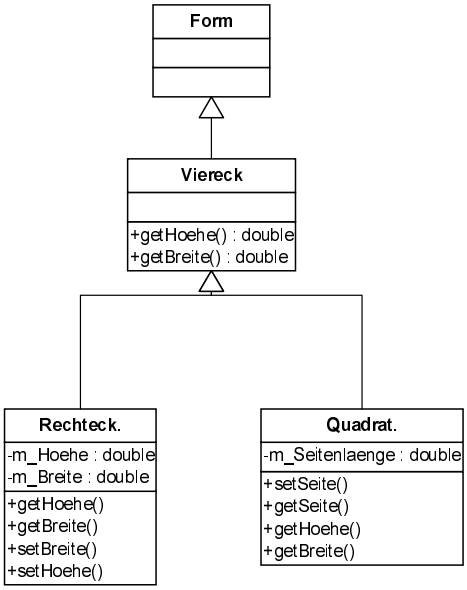

Abbildung 3.10: Eine »bessere« Anwendung von Vererbung

Obwohl man von konkreten Klassen ableiten kann, sollte man dabei vorsichtig sein. Scott Meyers [Mey, 96] geht dabei so weit, zu sagen, dass man von konkreten Klassen nie ableiten soll. Dies trifft meinen Stil bei der Implementierung von Frameworks, da es die sauberste Realisierung mit der Möglichkeit späterer Erweiterungen ergibt.

Ein weiteres Problem der Vererbung ist der Verlust eines großen Teils der Kapselung eines Objekts. Dies zwingt den Implementierer dazu, die gesamte Hierarchie zu überblicken. In Abbildung 3.10 muss z.B. der Implementierer von Rechteck die *Implementierung* von Form und Viereck kennen, da sie die Kosten verursachen. Zusätzlich muss dies bei der Übersetzung und nicht zur Laufzeit definiert werden, d.h., wenn der Implementierer eines Rechtecks sich entscheidet, Teile einer anderen Klasse wieder zu verwenden, funktioniert das nicht.

Wenn Vererbung nicht immer angebracht ist, was gibt es dann für Alternativen? Abbildung 3.11 zeigt eine Richtlinie zur Vererbung.

Durch Komposition (d.h. die Delegation von Funktionsaufrufen) kommt man zu einer viel besseren generellen Struktur, wie Abbildung 3.12 zeigt.

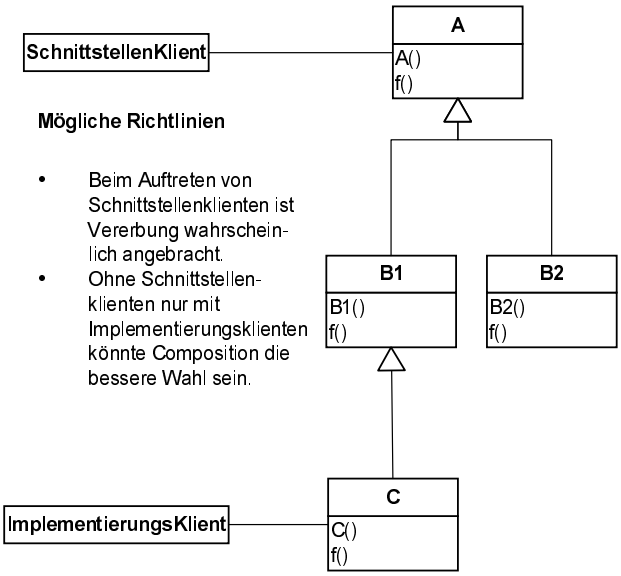

Abbildung 3.11: Eine Richtlinie zur Vererbung

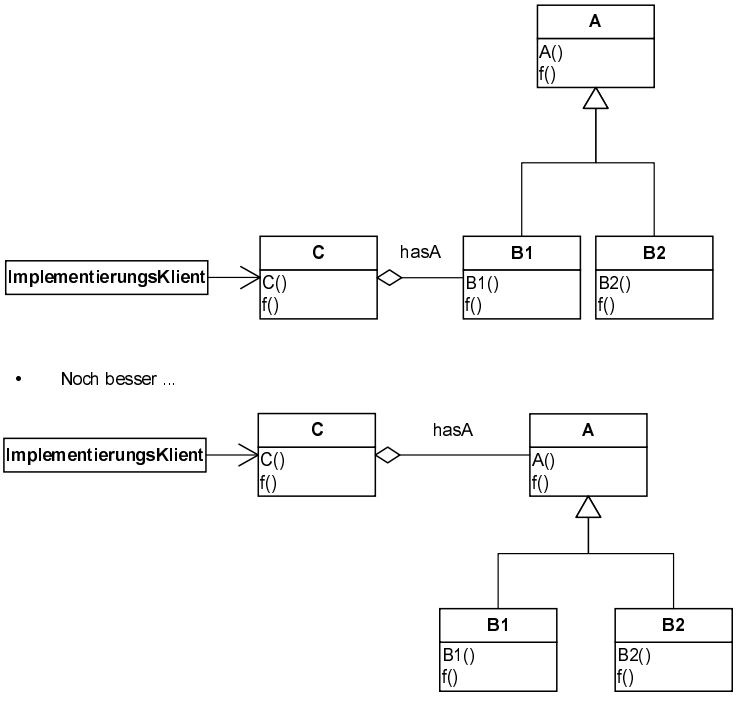

Abbildung 3.12: Delegation

3.3 Komponenten

Während sich Klassen entwickelten, gestaltete sich die Code-Wiederverwendung immer schwieriger. Es wurde akzeptiert, dass eine bessere Aufteilung der Verteilung nötig war. Damit wurden auf Komponenten beruhende Verteilungsmodelle wichtiger. Bei diesem Ansatz werden eine oder mehrere Schnittstellen durch eine Einheit abgebildet, die wir Komponenten nennen wollen. Der Ansatz kann auf verschiedene Weise realisiert werden, indem diese Komponenten in *shared libraries* oder ausführbaren Programmen zusammengefasst werden. Dies wird durch Microsofts COM (das sich zum .NET Framework weiterentwickelt hat) oder OMGs[2] CORBA unterstützt wird, die Kommunikation und Marshalling (die Konvertierung von Datentypen und Verteilung) koordinieren. Sprachspezifische Zusammenstellungen wie etwa Enterprise Java Beans (EJB) unterstützen ebenfalls diesen Ansatz.

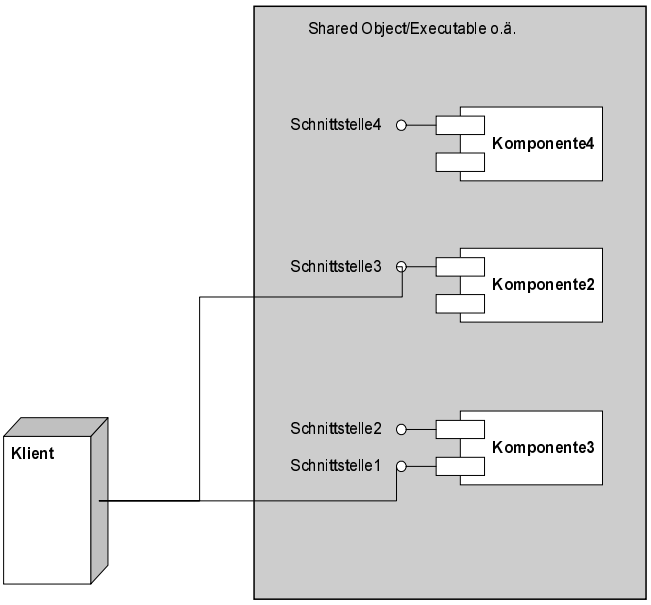

Abbildung 3.13: Komponenten

Der aktuelle Trend ist die Erweiterung um Transaktionsunterstützung, die es Komponenten gestattet, sich an Ereignissen zu beteiligen, die mehrere Komponenten betreffen. Transaktionsmonitore erscheinen sowohl in COM durch den MTS als auch in Lösungen anderer Hersteller, einschließlich hybrider Lösungen, die einen traditionellen Transaktionsmonitor nehmen und einen CORBA- oder Java-Überbau (wie etwa BEA Systems

2 A.d.Ü.: OMG = Object Management Group, CORBA=Common Object Request Broker Architecture, MTS=Microsoft Transaction Server

Produkt Jolt) anbieten. Des Weiteren werden sprachspezifische Java Application Server immer stabiler und können für geschäftskritische Frameworks benutzt werden. Abbildung 3.13 zeigt einige Komponenten.

3.4 Zusammenfassung

In diesem Kapitel wurde die Entwicklung der Strukturierung von Softwaresystemen besprochen – vom Programm als Ganzem hin zu in sich abgeschlossenen Komponenten, die ein komplexes Framework bilden. Zudem wurden die Grundlagen von Objekten, Vererbung und Komposition abgehandelt. Diese Basisterminologie erlaubt es, die nächsten Schritte bei der Systementwicklung zu gehen.

UML bietet eine semantisch reiche Notation, die Klassendiagramme, Objektdiagramme, Anwendungsfalldiagramme, Sequenzdiagramme, Kollaborationsdiagramme, Zustandsdiagramme, Aktivitätsdiagramme, Komponentendiagramme und Verteilungsdiagramme mit einschließt. Die richtige Auswahl an Diagrammen hängt jeweils davon ab, was man darstellen möchte. In diesem Kapitel, das dem Leser lediglich einen Überblick zu verschaffen sucht, kann ich der UML nur oberflächlich gerecht werden. Ich empfehle dem Einsteiger *UML Distilled* von Martin Fowler [Fow, 97b] und für Fortgeschrittene das *UML Users Guide* von Grady Booch und anderen [Boo, 99].[3]

3 A.d.Ü.: Eine ausgezeichnete deutsche Einführung ist auch *Objektorientierte Softwareentwicklung: Analyse und Design mit der UML* von Bernd Oestereich.