Typisierte Vererbung modelliert durch Aggregation

Franz J. Hauck

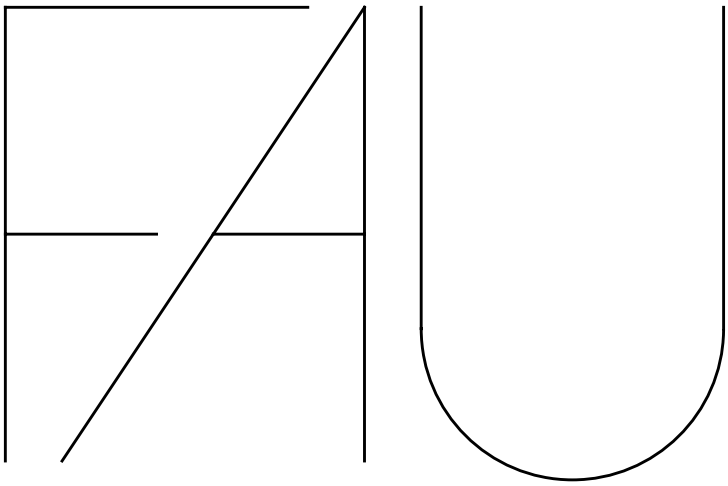
September 1993 TR-I4-9-93

Technical Report

Computer
Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany



Dieser Report wurde auch veröffentlicht in:

Franz J. Hauck: “Vererbung modelliert durch Aggregation”; In: H. Wedekind [Hrsg.] *Verteilte Systeme*, Grundl. und zukünft. Entw. aus d. Sicht d. SFB 182; Bibliographisches Institut, Zürich; erscheint 1994

Typisierte Vererbung modelliert durch Aggregation

Franz J. Hauck
E-Mail: hauck@informatik.uni-erlangen.de

Teilprojekt B2
Entwurf und Implementierung eines an Hardwarearchitektur und Aufgabenklassen adaptierbaren Multiprozessorsystems

Zusammenfassung

Vererbung ist neben der Aggregation das wichtigste Kompositionsmittel in objektorientierten Sprachen. Während bei Aggregation geeignete Typsysteme vorhanden sind, die den typsicheren Austausch von Objekten und Klassen erlauben, gibt es keine Typsysteme für Vererbungsbeziehungen, die einen typsicheren Austausch einer Basisklasse erlauben. Gerade für die Verwaltung von Klassenbibliotheken und langlebigen verteilten Systemen ist dies aber eine wichtige Voraussetzung.

Es wird ein Ansatz zur Typisierung der Vererbungsbeziehung vorgestellt. Er basiert darauf, die Vererbung durch Aggregation zu modellieren und die Typisierung der Vererbung auf die Typisierung der Aggregation abzubilden. Zur Modellierung der vollen Ausdrucksfähigkeit von Vererbung werden parametrierbare Bindungen für erzeugte Objekte eingeführt.

1 Einführung

Vererbung ist ein grundlegendes Konzept der objektorientierten Programmierung. Nach Wegner ist Vererbung ein Mechanismus zur Ressourcenteilung in Klassenhierarchien [Wegn87]. Dabei erben Klassen Methoden und Variablen von Basisklassen. Neben der Vererbung wird ein anderes Kompositionsmittel eingesetzt, die *Aggregation*. Darunter versteht man die Komposition von Objekten über Variablen und Objektreferenzen. Ein Objekt besteht dann aus Verweisen auf andere Objekte oder auf sich selbst.

Während es für Aggregation geeignete Typkonzepte gibt, die von dem konkreten Objekt, auf das ein Verweis zeigt, abstrahieren können, ist dies bei Vererbung nicht festzustellen. Typen beschreiben das abstrakte Verhalten von Objekten. Werden die Verweise einer Aggregationsbeziehung typisiert, so kann nur auf typkonforme Objekte verwiesen werden. Ein Austausch eines dieser Objekte ist dann möglich, wenn der Typ des neuen Objekts konform ist. Bei Vererbung ist ein Austausch einer vererbenden Klasse (*Basisklasse*) nicht vorgesehen. Dies liegt wohl darin begründet, daß Verweise auf andere Objekte üblicherweise variabel sein können, sich also dynamisch während des Programmlaufs ändern können, klassische Vererbungsbeziehungen jedoch nicht.

Betrachtet man dagegen Klassenbibliotheken, von denen Anwendungsprogramme durch Vererbung Gebrauch machen, oder langlebige verteilte Systeme, so tritt genau der Fall ein, daß Klassen ausgetauscht werden sollen, von denen Klassen anderer Programme geerbt haben. Ein solcher Austausch kann z.B. für die Fehlerbeseitigung oder Verbesserung von Klassen nötig sein. Bei Sprachen wie *Eiffel* [Meye92] und *C++* [ElSt90] stehen den Bibliotheksadministratoren keine Möglichkeiten zur Verfügung, einen solchen Austausch typsicher zu vollziehen. So kann bereits die Hinzunahme einer neuen Methode in einer Bibliotheksklasse abgeleitete Klassen kompromittieren, weil dort zufällig eine gleichnamige Methode deklariert wurde. Man benötigt daher eine typisierte Schnittstellen zwischen erbenden und vererbenden Klassen.

Im folgenden wird ein Ansatz vorgestellt, der Vererbung mit einer spezielleren Form der Aggregation modelliert und die Typen von Objekten zur Typisierung der Vererbung benutzt. In Kapitel 2 wird zunächst näher auf die beiden Kompositionsmittel Aggregation und Vererbung eingegangen und es werden Gründe für eine Typisierung der Vererbung angeführt. Das Kapitel beschreibt ein Objektmodell, auf dem alle folgenden Überlegungen basieren. Kapitel 3 zeigt auf, was Vererbung leisten kann, führt die Modellierung durch Aggregation ein und zeigt wie die Typisierung der Vererbung stattfindet. Kapitel 4 bewertet den Ansatz und stellt einige neue Möglichkeiten heraus, wie z.B. das Parametrieren der Basisklasse. In Kapitel 5 werden die Ergebnisse zusammengefaßt.

2 Typisierte Vererbung

Dieses Kapitel beschreibt ein minimales Objektmodell, in dem die beiden Kompositionsmittel Aggregation und Vererbung vorgestellt werden, und motiviert die Typisierung der Vererbungsbeziehung. Zum Abschluß wird auf den Zusammenhang zwischen Vererbung und der Typkonformität eingegangen.

2.1 Komposition durch Aggregation

Alle Objekte bestehen aus Verweisen auf andere Objekte. Über solche Verweise können Methoden aufgerufen werden. Verweise können variabel oder konstant sein. *Variablen*[1] sind benannte Speicher für Verweise. Das Verknüpfen zweier Objekte über eine Variable des einen Objekts wird *Bindung* genannt. Die Komposition von Objekten über solche Bindungen nennt man *Aggregation*. Bindungen können von der Erzeugung eines Objekts an bestehen. Solche Bindungen werden *initial* genannt. Objekte, die initial und konstant gebunden sind, heißen *privat*.

1. Es wird oft auch der Begriff der *Instanzvariablen* verwendet.

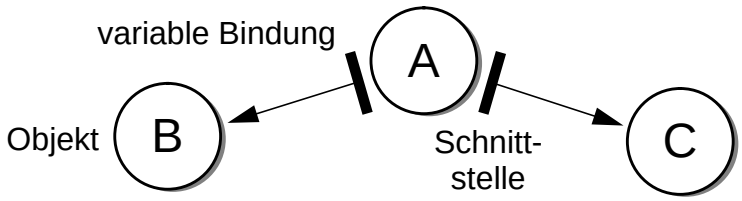


Abb. 2.1 Bindungen zwischen Objekten bei Aggregation

Abb. 2.1 zeigt ein Objekt *A*, an das zwei Objekte *B* und *C* gebunden sind, d.h. *A* hat je einen Verweis auf *B* und *C*. Die Bindungen können variabel sein und damit zur Laufzeit geändert werden. Dies entspricht dem Konzept eines Zeigers in *C++* und einer Variablen in *Eiffel*. Konstante Bindungen werden in *C++* durch Objektdeklarationen[2] und in *Eiffel* durch *Expanded objects* repräsentiert.

Die dicken Querbalken in Abb. 2.1 zeigen das Objekt *A* mit typisierten Variablen. Der Typ einer Variablen bestimmt, welche Objekte an die Variable gebunden werden können. Der Typ beschreibt ein abstraktes Verhalten der zu bindenden Objekte. Er stellt damit eine Art Kontrakt dar, der von den zu bindenden Objekten erfüllt werden muß. Die Typisierung der Schnittstellen erlaubt nun den typsicheren Austausch von Objekten durch Neubindung. Das neue Objekt muß lediglich einen konformen Typ besitzen.

2.2 Komposition durch Vererbung

Vererbung in Sprachen wie *C++* und *Eiffel* stellt sich zunächst als ein Kompositionsmittel zwischen Klassen statt zwischen Objekten dar. Abb. 2.2a zeigt eine Klasse *B* und eine Klasse *A*, die von *B* geerbt hat. Die übliche Sicht ist, daß *A* die Eigenschaften der Basisklasse *B* in sich besitzt – dargestellt durch den eingelassenen Kreis *B*.

Betrachtet man die Objekte der beiden Klassen, so hat ein Objekt der Klasse *A* Eigenschaften eines Objekts der Klasse *B* in sich. Abb. 2.2a könnte also genausogut je eine Instanz von *A* und von *B* darstellen. Die Instanz von *A* enthält auch eine Instanz von *B*.

Versucht man die Beziehung zwischen erbender und vererbender Klasse expliziter zu machen, kommt man zu Abb. 2.2b. Die Klasse *A*' wird über eine Kante „erbt-von“ mit *B* verbunden. *A*' ist jetzt nur eine Differenzklasse, die den Unterschied von *A* aus der Abb. 2.2a und *B* darstellt. Abb. 2.2c zeigt die gleiche Situation. Die Differenzklasse wird jedoch wie eine normale Klasse dargestellt. Vererbung ist damit eine Relation zwischen Klassen bzw. Differenzklassen – in den Bildern mit einem dicken Pfeil repräsentiert. Dies läßt sich, wie oben erwähnt, auch auf die Objekte dieser Klassen übertragen. Ein Objekt der Klasse *A*' besitzt eine besondere Verbindung zu einem Objekt der Klasse *B*. Dies widerspricht der herkömmlichen Sichtweise auf Vererbung, die z.B. bei *C++*

2. Im Gegensatz zu Zeigerdeklarationen.

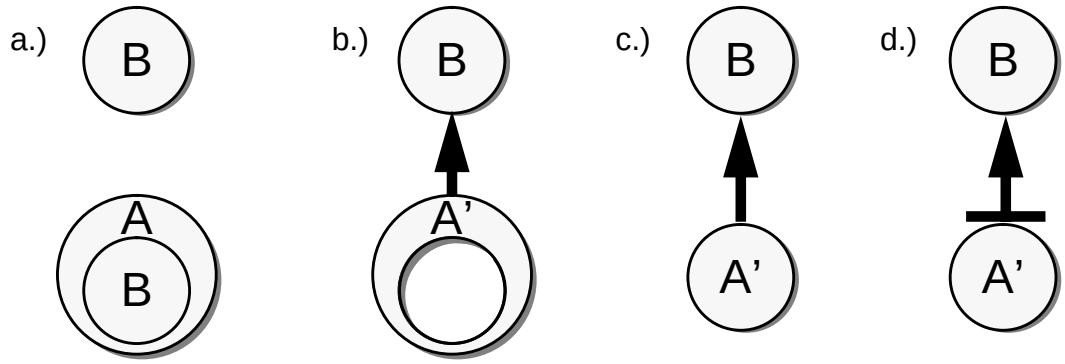


Abb. 2.2 Darstellung der Vererbungsbeziehung

vertreten wird. In den Implementierungen läßt sich die besondere Verbindung der Objekte dagegen leicht ausmachen. So kodiert z.B. C++ in den Speicher jedes Objekts der Klasse *A*' ein Objekt der Klasse *B.*

Eine Typisierung von Schnittstellen zwischen den an der Vererbung beteiligten Klassen findet nicht statt. Insbesondere ist kein Austausch von Basisklassen vorgesehen. Es wird zwar manchmal von Vererbungsschnittstelle gesprochen, z.B. bei *C++*. Diese Schnittstellen sind jedoch fest mit der Klasse verbunden und können nicht von ihr abstrahieren. Insbesondere kann keine andere Klasse dieselbe Schnittstelle besitzen. Die Schnittstelle kann zwar in einer anderen Klasse identisch definiert sein, ist jedoch niemals dieselbe. Diese Schnittstellen sind daher für eine Typisierung unbrauchbar und unterstützen nicht den typsicheren Austausch der Klasse gegen eine andere.

2.3 Warum typisierte Vererbung?

Die Einführung von typisierten Schnittstellen zwischen erbender und vererbender Klasse erlaubt den typsicheren Austausch der Basisklasse, ohne die erbende Klasse anpassen oder verändern zu müssen. Der Typ dient damit als Kontrakt zwischen den beteiligten Klassen.

Ein solcher Austausch ist für Wartungsarbeiten, z.B. zur Fehlerbeseitigung in Klassenbibliotheken oder verteilten, langlebigen Systemen notwendig. Im Fall von Klassenbibliotheken sollen Anwendungsprogrammierer nicht durch Verbesserungen in der Klassenbibliothek zu Veränderungen in ihren Programmen gezwungen werden. Eine einfache Neuübersetzung oder Neubindung sollte genügen.

Im Fall eines verteilten oder langlebigen Systems soll das System nicht abgeschaltet werden, um Klassen gegen verbesserte Versionen auszutauschen. Um einen solchen Austausch durchzuführen, muß es möglich sein, einige Objekte einer erbenden Klasse mit einer neuen Basisklasse zu verknüpfen und andere nicht. Dies liegt daran, daß man die Basisklasse im allgemeinen nicht für aktive Objekte austauschen kann.

Eine zusätzliche Kompositionsmöglichkeit ergibt sich dadurch, daß pro Objekt entschieden werden könnte, von welcher konkreten Basisklasse es erbt. Dies könnte in einer Konfigurationsphase bestimmt werden, wie sie in verteilten Systemen auch für die Aggregationsbeziehung vorgesehen werden kann [Fäus92].

Bei einer typisierten Schnittstelle könnte sich ein Objekt auch erst bei der Instantiierung an eine vorhandene Instanz einer Basisklasse binden[3]. Diese Bindung könnte, ähnlich wie beim dynamischen Binden von Aggregationsbeziehungen, über Nameserver oder Broker hergestellt werden.

2.4 Bemerkung zu Typen

Bei den bisherigen Betrachtungen wurde nicht weiter auf die Repräsentation von Typen eingegangen. Sprachen wie *C++* und *Eiffel* verbinden Typ- mit Klassenrepräsentation. Diese Konzepte voneinander zu trennen ist mittlerweile jedoch weitgehend akzeptiert, siehe z.B. [CoHC90].

Es wird daher im folgenden davon ausgegangen, daß Typen eine eigene Repräsentation im Objektmodell besitzen und eine Vererbung zwischen Typen unabhängig von Klassenvererbung ist. Vererbung zwischen Typen erlaubt die Wiederverwendung bereits bestehender Typbeschreibung und ist eventuell die Grundlage zur Erzeugung konformer Typen. Sicherlich kann Typvererbung aber auch zur Erzeugung nicht-konformer Typen verwendet werden.

Bei einer Trennung von Typ und Klasse muß es Mechanismen geben, getrennt formulierte Typen und Klassen zusammenzubringen. Dazu bestimmt jede Klasse von welchem Typ sie ist und gibt eine Abbildung von den Signaturen im Typ zu ihren eigenen Methoden oder Variablen an.

3 Modellierung durch Aggregation

Zur Modellierung der Vererbung durch Aggregation muß zunächst geklärt werden, was Vererbung zwischen Klassen leistet. Es wird dann gezeigt, wie sich eine solche Modellierung durchführen läßt und wie dadurch eine typisierte Vererbungsschnittstelle entsteht.

3.1 Was ist Vererbung?

Vererbung hat drei wichtige Eigenschaften für eine erbende Klasse bzw. für deren Objekte:

3. In einer besonderen Vererbungs-Bindung.

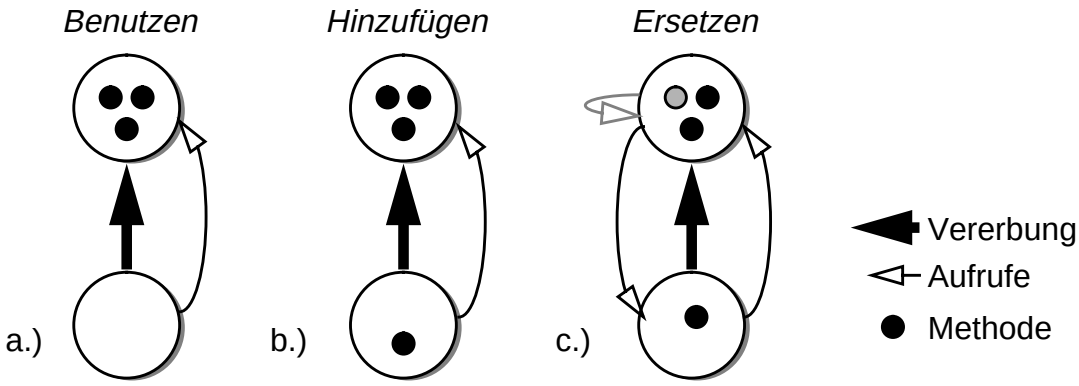


Abb. 3.1 Beziehungen der Subklasse zur Basisklasse bei Vererbung

- Die erbende Klasse kann wie in einer Aggregationsbeziehung alle zugänglichen Komponenten der Basisklasse benutzen (Abb. 3.1a). Im Bild wird dazu ein dünner Pfeil von erbender Klasse zur Basisklasse gezeichnet.
- Die erbende Klasse kann zusätzliche Komponenten definieren – Methoden und Variablen (Abb. 3.1b). Im Bild werden nur Methoden dargestellt.
- Die erbende Klasse kann zusätzliche Methoden definieren, die bereits in der Basisklasse definierte Methoden ersetzen. Das bedeutet, daß in der Basisklasse alle Aufrufe an die ersetzte Methode in die erbende Klasse umgeleitet werden (Abb. 3.1c). Im Bild wird dazu ein dünner Pfeil von der Basisklasse zur erbenden Klasse gezeichnet. Der graue Pfeil deutet die ursprüngliche Bindung dieses Pfeils an – ohne Vererbung. Der dünne Pfeil von erbender Klasse zur Basisklasse soll verdeutlichen, daß in der erbenden Klasse auch die ersetzte Methode explizit angesprochen werden kann, z. B. durch das aus *Smalltalk* bekannte Schlüsselwort *super* [GoRo83].

Vererbung bedeutet hier nicht, daß die erbende Klasse automatisch einen konformen Typ zum Typ der Basisklasse hat. Die Typkonformität soll getrennt beschrieben werden. Es sei dazu auf Abschnitt 2.4 verwiesen.

Zum besseren Verständnis sollen diese Eigenschaften an einem Beispiel demonstriert werden. Angenommen eine Klasse *Port* hätte die Eigenschaft, daß ihre Objekte einen Hardware-Port bedienen können. Auf diesem Port können einzelne Zeichen ausgegeben werden. Dazu bietet die Klasse eine Methode *Putchar* an. Für die Ausgabe von ganzen Zeilen wird eine Methode *Putline* angeboten, die intern auf die Methode *Putchar* abgebildet wird. In Abb. 3.2 (linke Spalte) wird in einer fiktiven Syntax eine stark vereinfachte Beschreibung der Klasse *Port* angegeben. Das Schlüsselwort *self* wird hier zur Verdeutlichung beim Methodenaufruf mitangegeben. In den meisten Sprachen muß *self* nicht explizit angegeben werden, sondern wird vom Compiler ergänzt.

In der gewählten Syntax stehen links von einem Doppelpunkt Namen von Variablen oder Methoden und rechts davon deren Deklaration oder Definition. Bei den Methoden sind die Deklarationen der formalen Parameter der Einfachheit halber weggelassen.

```
port : class
{
     putchar : ()->()
     { ... }

     putline : ()->()
     {    ...
          while ...
               self.putchar()
     }
}
```

```
bufferedport : class inherits port
{
     putchar : ()->()
     {    ...
          self.flush()
          ...
     }

     flush : ()->()
     {    ...
          while ...
               super.putchar()
     }
}
```

Abb. 3.2 Beispiel – Ausgabeport

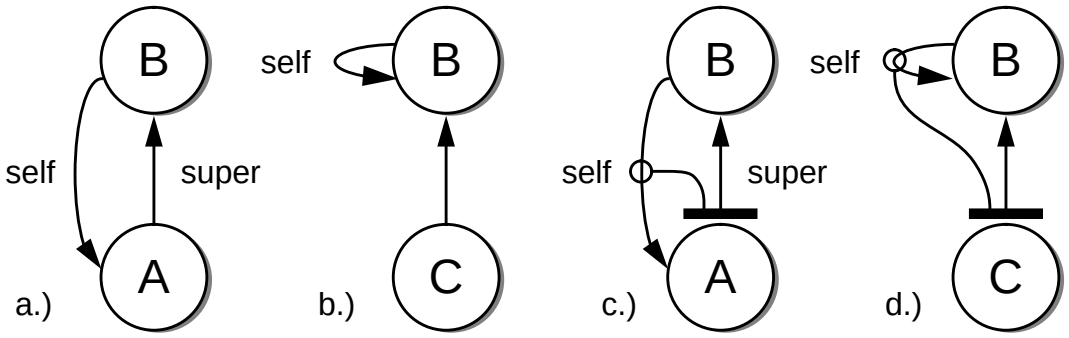


Abb. 3.3 Explizite Bindungen bei Vererbung und Aggregation

Um nun eine Klasse zu bauen, die eine gepufferte Ausgabe auf den Ausgabeport realisieren soll, wird Vererbung eingesetzt. Die Klasse *Bufferedport* (Abb. 3.2 – rechte Spalte) ersetzt die Methode *Putchar* durch eine puffernde Version. Alle Zeichen werden in einen Puffer eingetragen. Ist der Puffer voll, wird eine eigene hinzugefügte Methode *Flush* aufgerufen, die wiederum auf die ursprüngliche *Putchar* Methode zurückgreift, um den Puffer auszugeben. Dazu wird das Schlüsselwort *super* benutzt.

3.2 Vererbung durch Aggregation

Abb. 3.1 und das *Port*-Beispiel aus Abb. 3.2 geben bereits Hinweise, wie die Vererbung durch Aggregation modelliert werden kann. Die Schlüsselworte *self* und *super* werden zu expliziten Variablen. Die Variable *super* bindet ein privates Objekt der Basisklasse. Diese Bindung wird initial mit der Erzeugung eines Objekts der erbenden Klasse gebunden und bleibt konstant[4]. Die Variable *self* in der Basisklasse wird mit dem Objekt der erbenden Klasse gebunden (Abb. 3.3a)[5].

4. Durch Aufhebung dieser Bedingungen lassen sich dynamische Vererbungsbeziehungen modellieren. Darauf soll jedoch nicht näher eingegangen werden.
5. Im Bild wird der Einfachheit halber keine Bindung für die *self*-Variable der erbenden Klasse angegeben.

```
port : class
{
     self : parm port=here
     putchar : ()->()
     { ... }
     putline : ()->()
     {    ...
          while ...
               self.putchar()
     }
}
```

```
bufferedport : class
{
     self : parm bufferedport=here
     super : priv port(self=here)
     putchar : ()->()
     {    ...
          self.flush()
          ...
     }
     flush : ()->()
     {    ...
          while ...
               super.putchar()
     }
}
```

Abb. 3.4 Beispiel – Ausgabeport

Da die Variable *self* auch dann in der Basisklasse definiert wird, wenn deren Objekte nicht in einer Vererbungsbeziehung stehen, muß definiert werden, wie diese Variable für den Fall der Aggregationsbeziehung gebunden ist. Für diesen Fall wird die Variable auf das Objekt der Basisklasse selbst gebunden. Diese Situation zeigt Abb. 3.3b.

Die Bindung der *self*-Variablen eines Objekts der Basisklasse muß also je nachdem, ob das Objekt in einer Vererbungs- oder Aggregationsbeziehung steht, verschieden gebunden werden. Diese Bindung ist jeweils initial und konstant. Sie kann daher wie ein Parameter bei der Erzeugung des Objekts aufgefaßt werden.

3.3 Typisierung

Die Typisierung der Vererbungsbeziehung muß gleichzeitig die Bindung der *super* und der *self* Variablen erfassen. Der Typ der *super*-Variablen beschreibt die Schnittstelle der Objekte der Basisklasse. Dieser Typ sollte daher auch anzeigen, daß diese Objekte eine parametrierbare Bindung namens *self* besitzen, und deren Typ beschreiben. *Self* stellt sich damit wie eine spezielle öffentlich zugängliche Variable der Basisklasse dar. Sie ist deshalb speziell, weil sie nur bei der Erzeugung gebunden werden kann und sich anschließend nicht mehr verändern läßt.

Abb. 3.3c und 3.3d zeigen die Schnittstelle zur Basisklasse durch einen dicken Querbalken. Angedeutet ist die Einbeziehung der Bindung von *self* in den Typ. Abb. 3.3c zeigt den Fall der Vererbungsbeziehung, Abb. 3.3d die Aggregationsbeziehung.

Der Ansatz läßt sich am besten durch das Beispiel aus Abschnitt 3.1 verdeutlichen. Abb. 3.4 zeigt das gleiche Beispiel, aber jetzt mit einer speziellen Aggregationsbeziehung anstatt der Vererbungsbeziehung. In der Klasse *Port* wurde dazu die Variable *self* definiert. Sie ist von der Klasse *Port*[6]. Das Schlüsselwort *parm* soll anzeigen, daß diese Bindung konstant und initial ist und bei der Objekterzeugung parametriert werden kann. *Self* wird dann mit einer vorgegebenen Bindung definiert, die auf das erzeugte Objekt selbst zeigt. Dafür wird das Schlüsselwort *here* benutzt. Im Gegensatz zum

6. Wie bereits erwähnt wird hier nicht auf die Repräsentation der Typen eingegangen. Die Definition der Variablen *self* und *super* müßte mit dem entsprechenden Typ versehen werden.

```
bufferedport : class
{
     self : parm bufferedport=here
     super : priv port(self=self)
     ...
}
```

Abb. 3.5 Veränderung für weitere Vererbung

Schlüsselwort *self* (z.B. in *Smalltalk*) kann *here* durch Vererbung nicht verändert werden. Die übrigen Definitionen in der Klasse *Port* sind identisch zum vorhergehenden Beispiel (Abb. 3.2).

Die Klasse *Bufferedport* erbt jetzt nicht durch ein Schlüsselwort, sondern definiert eine Variable *super*, die an ein privates Objekt der Klasse *Port* gebunden wird. In Klammern wird dabei die Parametrierung der Bindung von *self* in dem erzeugten und gebundenen Objekt vorgenommen[7]. *Here* bezeichnet dort einen Verweis auf das Objekt der Klasse *Bufferedport*. *Self* wird analog zu *self* in *Port* definiert. Die übrigen Definitionen bleiben wiederum gleich.

3.4 Wiederholte Vererbung

Bisher wurden immer nur jeweils zwei Klassen in einer Vererbungsbeziehung betrachtet. Klassen können auch von Basisklassen erben, die selbst wieder von einer anderen Klasse geerbt haben, so daß sich Ketten von erbenden Klassen bilden.

Der vorgestellte Ansatz zur Modellierung der Vererbung durch Aggregation leistet dies bisher nicht. Im Beispiel in Abb. 3.4 wurde zwar in der Klasse *Bufferedport* eine parametrierbare Bindung für *self* definiert, diese hat jedoch bei erneuter Vererbung nicht den gewünschten Effekt. Normalerweise würden alle beteiligten Klassen, also alle Basisklassen und die letztendlich alles erbende Klasse, die gleiche Bindung für ihre *self*-Variablen haben. Dies muß entsprechend modelliert werden. Im Beispiel ist dies nur dann der Fall, wenn *Bufferedport* die letztendlich erbende Klasse ist. Beide Bindungen sind dann mit *here* aus *Bufferedport* gebunden.

Für den allgemeinen Fall muß der Beispielcode verändert werden, wie es in Abb. 3.4 dargestellt ist. *Self* wird wie bisher definiert, die Definition von *super* dagegen wird verändert. Statt das *self* aus der Klasse *Port* mit *here* zu binden, wird es an das eigene *self*

7. Hier wird stillschweigend vorausgesetzt, daß Objekte der Klasse *Bufferedport* typkonform zur Klasse *Port* sind. Wenn dies nicht der Fall ist, muß *self* in *Port* mit einem Typ definiert werden, zu dem der Typ von *Bufferedport* konform ist. Dies läßt sich durch geeignete Typrepräsentation leicht ausdrücken.

gebunden[8]. Damit wird das *self* aus *Port* immer gleich dem *self* aus *Bufferedport* gesetzt, auch für den Fall der Parametrierung von *self* aus *Bufferedport* bei wiederholter Vererbung.

3.5 Mehrfachvererbung

Mehrfachvererbung, d.h. Vererbung mit mehreren Basisklassen, läßt sich leicht modellieren. Es werden mehrere *super*-Variablen definiert, bei denen jeweils die *self*-Variable parametriert wird. Die entsprechenden Variablen zur Bindung der Basisklassen müssen verschieden benannt werden. Die Benennung kann jedoch problemorientiert erfolgen.

4 Bewertung des Konzepts

4.1 Vererbung

Neben dem typsicheren Austausch von Basisklassen, bietet die vorgestellte Art der Vererbung weitere Vorteile. So kann pro Methodenaufruf entschieden werden, ob dieser Aufruf bei Vererbung von der letztendlich erbenden Klasse abgewickelt wird. Dies wird durch den Aufruf über *self* angezeigt. Beim Aufruf über *here* wird immer die eigene Methode aufgerufen. *C++* gestattet eine Auswahl lediglich bei der Methodendeklaration, indem das Schlüsselwort *virtual* entweder vorangestellt wird oder nicht. Bei *Smalltalk* läßt sich eine Fixierung auf die eigene Methode überhaupt nicht beschreiben.

Mit mehreren parametrierbaren Bindungen lassen sich mehrere verschiedene *self*-Variablen definieren, die verschiedenen Zwecken dienen können. Der erbenden Klasse werden mehrere parametrierte Bindungen angeboten, von denen sie einige, nicht unbedingt alle, an sich selbst binden können. Damit kann die erbende Klasse auch darauf Einfluß nehmen, welche Aufrufe aus der Basisklasse von ihr aufgefangen werden sollen und welche nicht.

Über eine parametrierbare Bindung kann man auch die Basisklasse eines Objekts parametrieren. Dazu lassen sich Klassen definieren, die zusätzlich zu ihrer *self*-Bindung eine parametrierbare *super*-Bindung besitzen (Abb. 4.1). Diese Bindung muß bei der Erzeugung abgesättigt werden, ist also nicht voreingestellt. In Abb. 4.1 wird ein Objekt der Klasse *C* gezeigt, das ein Objekt der Klasse *A* benutzt. Dazu muß *C* bei der Erzeugung von *A* ein Objekt der Klasse *B* miterzeugen und die parametrierten Bindungen so setzen, daß *B* wie eine Basisklasse zu *A* wirkt. Dazu muß *B* mit dem Typ der *super*-

8. Man beachte, daß in der Syntax der Parametrierung links vom Gleichheitszeichen ein Name aus der angegebenen Klasse (bzw. deren Typ) steht – *self* aus *Port* – und rechts vom Gleichheitszeichen ein Ausdruck, dessen Namen aus der definierenden Klasse stammen – *self* von *Bufferedport*.

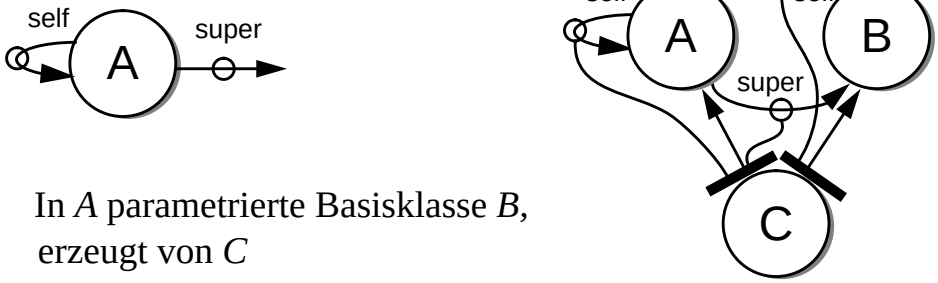


Abb. 4.1 In *A* parametrierte Basisklasse *B*, erzeugt von *C*

Variablen in *A* konform sein. Zu untersuchen ist in diesem Zusammenhang der Vergleich mit der Mixin-Vererbung, z.B. dem Ansatz von [BraCoo90]. Die Klasse von *A* kann hier nämlich wie ein Mixin aufgefaßt werden.

Auf ähnliche Weise läßt sich die sogenannte *Fork-join-inheritance* modellieren. Dabei handelt es sich um mehrfache Vererbung bei der in der Vererbungshierarchie mehrfach auftretende gleiche Basisklassen mit nur einem Objekt repräsentiert sind [Sakk92].

4.2 Komposition

Unabhängig von der Vererbung können die parametrierbaren Bindungen zu weiteren Kompositionsformen herangezogen werden, die über die Modellierung von Vererbung hinausgehen. So lassen sich Objektstrukturen definieren, bei denen die Objekte beliebig miteinander über parametrierbare Bindungen verschaltet werden.

Bei der Komposition von Systemen steht der Anwender nicht mehr vor zwei gänzlich verschiedenen Mechanismen, die sich gegenseitig ausschließen. Stattdessen verwendet er immer Aggregation als Kompositionsmittel und setzt, falls nötig, parametrierbare Bindungen ein.

4.3 Verteilung

Ein Nebeneffekt des vorgestellten Ansatzes ist, daß die Objekte einer Vererbungsbeziehung einfach auf verschiedene Rechnerknoten verteilt werden können, denn es ist bekannt, wie Aggregationsbeziehungen verteilt werden können. Dies geschieht in den Implementierungen mit entsprechenden Mechanismen für die Objektadressierung, z.B. durch Stubs.

Für die Vererbung gibt es bisher keine Untersuchungen über eine Verteilung der Objekte von erbender Klasse und Basisklasse. Der vorgestellte Ansatz der Modellierung durch Aggregation erlaubt die Verteilbarkeit ganz automatisch dadurch, daß nur noch Aggregationsbeziehungen vorhanden sind. Die Verteilung der beteiligten Objekte wird nicht in jedem Fall sinnvoll sein, ist aber möglich.

5 Zusammenfassung

Es wurde ein Konzept vorgestellt, wie sich Vererbungsbeziehungen durch spezielle Aggregationsbeziehungen modellieren lassen. Der Ansatz verdeutlicht zum einen den Vererbungsvorgang und kann zum anderen als Konzept zur Implementierung von Vererbung verstanden werden. Integriert in eine Programmiersprache eröffnet der Ansatz weitere Programmiermöglichkeiten, wobei die traditionelle Vererbung durch geeignete Syntaxunterstützung dem Programmierer erhalten bleiben könnte.

Der Ansatz erlaubt eine Typisierung der Vererbungsbeziehung, die wiederum den Austausch von Bibliotheksklassen oder Klassen in verteilten oder langlebigen Systemen typsicher ermöglicht. Die Einführung von Bindungen, die bei der Objekterzeugung parametriert und in den Typ der Objekte integriert werden können, erlaubt über die Modellierung von Vererbung hinausgehende Kompositionsmöglichkeiten.

An dieser Stelle sei meinem Kollegen und Freund Thomas Eirich gedankt, der frühe Ideen zu dem Thema mitentwickelt hat. Dank gilt auch allen anderen Kollegen, vor allem aus der PM-Gruppe, die in vielen Diskussionen zum besseren Verständnis des Themas beigetragen haben.

Literatur

[BrCo90] G. Bracha, W. Cook: “Mixin-based inheritance”; In: *Proc. of the Conf. on Obj.-Oriented Progr. Sys., Lang., and Appl. / Eur. Conf. on Obj.-Oriented Progr.* – OOPSLA / ECOOP (Ottawa, Ont., Canada, 21.-25. Okt. 1990); SIGPLAN Notices 25(10); Oktober 1990 – pp. 303-311

[CoHC90] W. R. Cook, W. L. Hill, P. S. Canning: “Inheritance is not subtyping”; *Conf. record of the 17th Symp. on Princ. of Progr. Lang.* – PoPL, (San Francisco, Cal., 17-19. Jan. 1990); 1990 – pp. 125-135

[ElSt90] M.A. Ellis, B. Stroustrup: *The annotated C++ reference manual – ANSI base document*; Addison-Wesley, Reading MA, USA; 1990

[Fäus92] M. Fäustle: *Beschreibung der Verteilung in objektorientierten Systemen*; Dissertation, IMMD, Univ. Erlangen-Nürnberg; Arbeitsber. d. IMMD 25(8); Sep. 1992

[GoRo83] A. Goldberg, D. Robson: *Smalltalk-80: the language and its implementation*; Addison-Wesley, Reading MA, USA; 1983

[Meye92] B. Meyer: *Eiffel: the language*; Prentice Hall, New York, NY; 1992

[Sakk92] M. Sakkinen: “A critique of the inheritance principles of C++”; *USENIX, Comp. Sys.* 5(1); Univ. of Calif. Press, Berkeley; 1992 – pp. 69-110

[Wegn87] P. Wegner: “Dimensions of object-based language design”; In: *Proc. of the Conf. on Obj.-Oriented Progr. Sys., Lang., and Appl.* – OOPSLA; N. Meyrowitz [Hrsg.], (Orlando, Fla., 4.-8. Okt. 1987); SIGPLAN 22(12); ACM, New York, NY; Dez. 1987 – pp. 168-182