Objektorientierte Programmierung & Softwareentwicklung

Eine kritische Einschätzung

Manfred Broy
Institut für Informatik
Technische Universität München
D-80290 München, Germany

Johannes Siedersleben
sd&m Research
D-81737 München, Germany

Abstract
In der Softwaretechnik ist die Objektorientierung inzwischen wohl der populärste Ansatz für Programmierung, Entwurf und Analyse. Allerdings hat es die Objektorientierung in ihrer bisherigen Ausprägung nicht geschafft, den Bedürfnissen der heutigen hochverteilten, interoperativen Softwareanwendungen gerecht zu werden. Im folgenden legen wir dar, dass Objektorientierung tatsächlich ohne jede Frage interessante Eigenschaften besitzt, dass jedoch eine Reihe schwerer Unzulänglichkeiten bei dem Einsatz dieser Technik für große verteilte Softwaresystemen auftreten. Dies zeigt, dass objektorientierte Techniken nicht mehr auf dem neuesten Stand einer wissenschaftlich wohlverstandenen Programmiermethode und Softwaretechnik stehen.

Abstract
In software engineering object oriented development is today the most popular programming and design approach. However, object orientation does not manage to address these needs of today's software construction in such a radical and fundamental way as needed in highly distributed interoperating software applications. In the following, we argue that object orientation indeed offers interesting features, but yet suffers from a number of severe shortcomings for engineering large distributed software systems. This shows that object oriented techniques are not in accordance with the state of the art in scientific well-understood programming methodology and software engineering.

1. Einleitung

Objektorientierung wurde von Ole-Johan Dahl und Kristen Nygaard erfunden, als sie Simula 67 entwickelten (vgl. [Simula 67]). Vorher waren die meisten Programmiersprachen entweder durch den Befehlsatz der vorhandenen Maschinen oder durch theoretischen Grundlagen der Berechenbarkeit wie dem λ-Kalkül beeinflußt. Nur langsam erreichten und unterstützten sie eine abstraktere Sicht der Daten und Abläufe und waren immer noch bestimmt von der Vorstellung von kleinen, sequentiellen, nicht-interaktiven Programmen. Ein/Ausgabe wurde normalerweise als nachrangig betrachtet und war deshalb kein Bestandteil der Definition z.B. von ALGOL.

Simula 67 brachte das radikal neue Konzept von Koroutinen und Klassen. Das erleichterte die Entwicklung großer komplexer Systeme, die – verteilt auf viele Rechner – parallel und asynchron agieren.

In den letzten drei Jahrzehnten entwickelte sich die Objektorientierung zum populärsten Ansatz für die Programmierung und den Entwurf von Software. Objektorientierung, so heißt es, biete bessere Möglichkeiten der Strukturierung und flexiblere Konzepte als die konventionellen imperativen, funktionalen oder logischen Programmierstile, vor allem für das Programmieren im Großen. Dies gilt zweifellos bei der Gestaltung graphischer Nutzerschnittstellen und bei umfangreichen Programmbibliotheken (Frameworks).

Allerdings begegnet die Objektorientierung den Erfordernissen der Software-Konstruktion weniger radikal und fundamental als wir uns das wünschen. In vieler Hinsicht bleibt die Objektorientierung innerhalb des konventionellen Programmieransatzes, stark geprägt durch die sequentiellen, nicht vernetzten Maschinen der frühen 60er Jahre.

Man könnte einwenden, dass etwa Java als relativ junge objektorientierte Sprache alle Ansprüche an eine moderne Programmiersprache erfüllt. Tatsächlich aber ist Java in vieler Hinsicht konventionell; die Java-Objektorientierung ist eher hausbacken. Java hat sich durchgesetzt, weil es leicht zu lernen ist, und weil es sich dank Portierbarkeit und Code-Mobilität ideal mit dem Internet verbindet. Dies sind allerdings Eigenschaften einer Programmiersprache, die weitgehend orthogonal zu ihren methodischen Prinzipien liegen.

In den letzten 50 Jahren haben sich Informatik und Software Engineering zu einer geachteten wissenschaftlichen Disziplin entwickelt. Viele neue und tiefe Einsichten wurden gewonnen, umfassende Prinzipien wurden entdeckt, wissenschaftlich analysiert und sind heute Teil des wachsenden Wissens über Programmierung und ingenieurmäßige Erstellung von Software-Systemen. Im Folgenden vertreten wir die These, dass die heute gebräuchliche Objektorientierung zwar eine ganze Reihe interessanter, vorteilhafter Eigenschaften besitzt, dass es aber auch einige gravierende Mängel gibt, die zeigen, dass die Objektorientierung nicht auf dem aktuellen Stand der wissenschaftlich verstandenen Programmiermethodik und des Software-Engineering ist.

Unser Papier ist gedacht als – aus Sicht der Autoren dringend nötiger – kritischer Gegenpol zu den zahlreichen Publikationen, die die Vorteile der Objektorientierung

unkritisch und unreflektiert herausstellen. Wir wollen den Leser wach rütteln und nehmen dabei bewußt in Kauf, dass die Objektorientierung möglicherweise schlechter wegkommt, als sie es verdient hat.

Das Papier darf *nicht* als Kritik an den Pionieren, Erfindern und Protagonisten der Objektorientierung (Dahl, Goldberg und vielen anderen) missverstanden werden. Ihnen zollen wir unseren uneingeschränkten Respekt. Die Programmiersprachen, die sie geschaffen haben, waren ein gewaltiger Schritt nach vorn und für den damaligen Zweck hervorragend geeignet. Allerdings sind heute die Anforderungen an Programmiersprachen und Entwicklungsmethodik wesentlich höher .

2. Objektorientierung – Anspruch und Wirklichkeit

Objektorientierung war zunächst ein Paradigma der Programmierung, wie es von den OO-Sprachen unterstützt wird, und entwickelte sich weiter zu einer umfassenden Methode des Software-Entwurfs, die das ganze Spektrum von Spezifikation, Konstruktion bis zur Implementierung unterstützt. Speziell bei Internet- und Client/Server-Systemen gilt die Objektorientierung als der im Vergleich zu anderen Programmierstilen bessere Weg.

Wir beginnen unsere Diskussion mit einer kurzen Übersicht der wesentlichen Elemente der Objektorientierung:

- Klassen mit Attributen und Methoden als die wesentlichen Einheiten zur Beschreibung und Strukturierung von Programmen,
- Schnittstellen als Listen von Methoden,
- Erzeugung von Objekten als Instanzen von Klassen,
- Speicherung von Daten und Zuständen in Attributen von Klassen und Objekten,
- Objektidentität definiert durch die Speicheradresse,
- Vererbung und Polymorphie.

Die Objektorientierung beansprucht für sich, die folgenden Entwurfsprinzipien zu unterstützen:

- Datenabstraktion,
- Geheimnisprinzip,
- wohldefinierte Schnittstellen,
- Modularität durch Kapselung der Objektdaten,
- Dynamik und Flexibilität durch die Instanziierung von Objekten,
- Wiederverwendung von Code durch Vererbung und Aggregation.

Die ersten drei Prinzipien wurden bereits von nicht objektorientierten Sprachen wie Modula2 und Ada83 unterstützt: Auch dort gibt es Datenabstraktion, Geheimnisprinzip und wohldefinierte Schnittstellen, aber natürlich noch keine Objekte und keine Vererbung. Der eigentliche Mehrwert der Objektorientierung liegt also scheinbar in den letzten beiden Punkten: Instanziierung von Objekten und Vererbung.

Objektorientierung löst trotz ihrer Beliebtheit und Verbreitung längst nicht alle Probleme, und manche hat sie erst geschaffen. Wir nennen zunächst die Probleme, die schon immer da waren und trotz Objektorientierung geblieben sind:

- Die Objektorientierung liefert keinen geeigneten Komponentenbegriff als Basis für Software-Architektur.
- Die Objektorientierung kennt keine Komposition von Klassen.
- Das OO-Ausführungsmodell ist – wie bei den prozeduralen Sprachen – sequentiell.
- Die Objektorientierung sagt uns nicht, wie wir das Verhalten (Funktionen und Interaktion) von Schnittstellen definieren sollen und wie die zu spezifieren ist.

In der Tat bieten neuere objektorientierte Programmiersprachen eine Reihe von Erweiterungen zur klassischen Objektorientierung, um einige dieser Probleme zu umgehen. Beispielsweise bieten syntaktische Klassenschnittstellen ein nützliches Konzept für die Beschreibung der syntaktischen Schnittstellen. In den meisten objektorientierten Programmiersprachen existieren jedoch keine abstrakten semantischen Beschreibungen für Schnittstellen. Experimente und Erfahrungen mit objektorientierten Frameworks zeigen die Schwäche der Objektorientierung in dieser Hinsicht. Dies belegen z.B. die Behälter und Iteratoren von C++ oder von Java.

Die Objektorientierung hat uns zusätzlich ein neues Problem beschert und ein vorhandenes verschärft:

- Vererbung (im Sinn von Implementierungsvererbung) verletzt das Geheimnisprinzip.
- Zeiger (Pointer) waren schon immer die Ursache für oft undurchsichtige Verweistrukturen und schwer zu analysierende Seiteneffekte – und in der Objektorientierung verwenden wir massenhaft Objektreferenzen, die nichts anderes sind als notdürftig verhüllte Zeiger.

Auch die objektorientierten Methoden für Spezifikation und Konstruktion, vor allem UML, haben schwere Mängel, die sie zum Teil mit der objektorientierten Programmierung gemeinsam haben:

- Fehlen einer klaren Semantik,
- mangelnde Integration verschiedener Beschreibungstechniken,
- fehlendes Komponentenkonzept als Basis der Software-Architektur.

Gerade die oft gepriesene UML ist leider eine bloße Sammlung von Konzepten (so wie in den 60igern PL/1 für Programmiersprachen) und eben keine Synthese.

3. Objektorientierte Programmierung – eine Kritik

Dieser Abschnitt befasst sich mit den Problemen der objektorientierten Programmierung. Die Kernaussage lautet: Die Objektorientierung unterstützt uns nicht ausreichend bei der Erstellung großer verteilter Software-Systeme.

3.1. Spezifikation von Klassen

Eine Klasse offenbart die Grundidee eines Moduls bei der objektorientierten Programmierung. Klassen sind die grundliegenden Bausteine objektorientierter Programme. In gewisser Hinsicht sind sie die einzigen Elemente zur objektorientierten Strukturierung.

Bei einer Schnittstellen-Spezifikation beschreiben wir das Verhalten einer Klasse in einer Schnittstellensicht, also ihr erkennbares Verhalten. Somit haben wir eine Beschreibung, die bestimmt, unter welchen Umständen zwei verschiedene Klassen in der gleichen Umgebung ohne jeglichen Unterschied in ihrem Verhalten verwendet werden können. Solch eine Definition ist für den Ansatz einer Spezifikation und eines Entwurfs zwingend, sei es top-down oder bottom-up.

Leider ist es außerordentlich schwierig, das beobachtbare Verhalten von Klassen und Objekten zu beschreiben. Der Grund dafür sind die komplexen Interaktionen von Objekten über Methodenaufrufe, die nichts anderes sind als normale Funktionsaufrufe, ergänzt um den Mechanismus der späten Bindung.

Methodenaufrufe können natürlich den Zustand verändern, der durch die Belegung der Attribute der betreffenden Objekte gegeben ist. Methodenaufrufe können wieder in Unteraufrufe weiterer Methoden resultieren und deshalb nicht nur den Zustand der durch die Aufrufe adressierten Objekte verändern, sondern zusätzlich auch den anderer Objekte. Daher müssen Methodenaufrufe in objektorientierten Programmen als in einem riesigem Zustandsraum – dem globalen Programmzustand – operierend angesehen werden.

Der Effekt eines Methodenaufrufes einschließlich aller Unteraufrufe von Methoden während der Ausführung des Anrufes muß zwangsläufig als eine Zustandsänderung auf dem globalen Zustandsraum beschrieben werden. Diese Aussage läßt sich anhand von zwei Phänomenen präzisieren: Den undefinierten Fernwirkungen und dem Bumerang.

Undefinierte Fernwirkungen
x, *y* seien Instanzen der Klassen *X* bzw. *Y*; *f* sei eine Methode von *X* und *g* eine Methode von *Y*. Die Notation

$$x.f \approx\!\!> y.g$$

bedeutet: *x.f* kann den Aufruf *y.g* verursachen, und zwar direkt (der Aufruf *y.g* steht im Code von *f*) oder indirekt (eine Folge von Unteraufrufen führt vom Aufruf *x.f* zum Aufruf *y.g*). Die in allen OO-Sprachen vorhandene import-Anweisung (z.B. *X imports Y*) sagt nur, welche anderen Klassen zur Kompilierung von *X* benötigt werden, aber nur wenig über den Wirkungsbereich *W(x, f)* des Aufrufs *x.f*:

$$W(x, f) = \{ (y, g) \mid x.f \approx\!\!> y.g \}$$

Er steht bestenfalls im Kommentar; die Objektorientierung unterstützt uns nicht bei seiner Ermittlung. Der eigentliche Vorwurf geht aber vor allem an die Implement-

ierungsvererbung, die die statische Analyse der Fernwirkung verhindert (vgl. hierzu Abschnitt 3.4).

Bumerang
Die Fernwirkung kann auch wieder zum Ausgangsobjekt zurückkommen (hier ist also $x = y$, und X hat eine Methode g; im Fall $f = g$ handelt es sich um eine schlichte Rekursion):

$$x.f \approx\!\!> x.g$$

Dieser Punkt ist trivial, wird aber gern übersehen und führt oft zu Ärger. Die meisten OO-Programmierer unterstellen naiv, dass jedes Objekt am Beginn und am Ende einer jeden Methode in einem konsistenten Zustand ist. Inkonsistenzen sind nur unterwegs, also nach dem Beginn und vor dem Ende der Methode, möglich und erlaubt. Aber im obigen Beispiel sieht der Aufruf *x.g* das Objekt *x* in einem zufälligen, wahrscheinlich inkonsistenten Zwischenzustand, den der unvollständig abgearbeitete Aufruf der Methode *f* herbeigeführt hat. Man sollte also unterscheiden zwischen Methoden, die die genannte Konsistenz unterstellen dürfen, und denen, wo das nicht der Fall ist (hier *g*).

Entscheidend ist folgende Beobachtung. Die Spezifikation des beobachtbaren Verhaltens von Klassen läuft in alle Schwierigkeiten, die wir von den verteilten interaktiven Systemen kennen (bis auf Nebenläufigkeit und Granularität von Aktionen; siehe Abschnitt 3.5). Objekte im Sinn der Objektorientierung brauchen in Wirklichkeit eine Laufzeitumgebung die echte Parallelität beherrscht – aber die liefert uns die Objektorientierung nicht. Alle Versuche, Parallelität in die Objektorientierung einzuführen, konnten nicht überzeugen.

3.2. Objekt-Instanziierung

Ein wesentliches Konzept der Objektorientierung ist die Möglichkeit, Objekte von Klassen zu instanziieren. Diese Idee hat zwei Seiten: Zunächst wird ein Verhalten von einer Beschreibung abgeleitet. Das ist ein geläufiges Konzept in der Programmierung, wo wir Verhalten von Programmen ableiten. Ein zweiter, wesentlich schwierigerer Aspekt ist die Frage nach der Identität von Objekten und der Zugriff auf Objekte mit Hilfe von Referenzen.

Der Objektbegriff der gängigen objektorientierten Sprachen ist sehr implementierungsnah: Zwei Objekte sind identisch, wenn sie denselben Speicherplatz einnehmen; sie werden identifiziert durch ihre Speicheradresse und existieren also nur innerhalb eines Adreßraums. Verbindungssoftware (Corba, RMI) und objektorientierte Datenbanken verwenden künstliche Objektschlüssel, und das sind nichts anderes als Adreßraum-übergreifende Zeiger. Dies konfrontiert den Programmierer mit zwei Problemen:

a) Wie kann er Welten mit verschiedenen Objektbegriffen zusammenführen (z.B. die Java-Objekte im Hauptspeicher und die Objekte in einer OO-Datenbank)?
b) Wie kann er Seiteneffekte kontrollieren, wenn ein Objekt mehrfach referenziert wird?

Die meisten Projekte lösen diese Fragen entweder gar nicht oder bestenfalls ad hoc. Das in [Siedersleben, Denert 00] beschriebene Konzept des *Arbeitsbereichs* ist eine Hilfe beim ersten Punkt; zum zweiten folgende kurze Betrachtung:

Mehrfach-Referenzen sind immer dann kein Problem, wenn unveränderbare Objekte referenziert werden. Ein bekanntes Beispiel ist der Java-String; man vgl. hierzu auch das Flyweight-Muster [Gamma 94], das dieselbe Idee beschreibt, und die funktionale Programmierung, die unter anderem darauf beruht, dass es keine Variablen gibt, sondern nur benannte Werte (ähnlich wie *const*-Variable in C oder *static final*-Variable in Java).

Die Regel könnte also lauten: Keine Mehrfach-Referenzen auf änderbare Objekte! Dies ist im Grunde eine gute Richtschnur, doch leider führt sie in der Praxis oft zu zwei Problemen:

a) Mangelnde Performance (deshalb gibt es in vielen funktionalen Programmiersprachen doch wieder Variable im üblichen Sinn)
b) Unübersichtliche Benachrichtigungsmechanismen, die dann erforderlich sind, wenn ein Objekt zum Zwecke der Vermeidung mehrfacher Referenzen in mehreren Kopien existiert, die von Hand – etwa nach dem Beobachtermuster – zu koordinieren sind.

Auch hier läßt uns die Objektorientierung allein.

3.3. Software-Architektur und Komponenten

Der wichtigste Baustein der Objektorientierung ist die Klasse. Von einem methodischen Standpunkt aus sollte die Klasse deshalb drei zentrale Eigenschaften mitbringen:

- hierarchische Komposition/Dekomposition,
- Spezifikation des Verhaltens von Schnittstellen,
- angemessene Skalierbarkeit.

Das objektorientierte Klassenkonzept wird keiner dieser drei Anforderungen gerecht.

3.3.1. Komposition von Klassen

Die Objektorientierung bietet keine Operation zur Komposition von mehreren Klassen zu einer zusammengesetzten Klasse. Es gibt kein gemeinsames Konzept einer zusammengesetzten Klasse. Die Idee der mehrfachen Vererbung hat nur eine oberflächliche Ähnlichkeit mit der Komposition – in Wirklichkeit hat sie nichts damit

zu tun. Somit unterstützt uns die Objektorientierung nicht bei der Strukturierung von großen Systemen in Komponenten – aber genau das wäre dringend nötig. Die Komposition von Klassen ließe sich ohne Aufwand einführen, und deshalb ist die Abwesenheit dieses Konzepts unverständlich.

3.3.2. Komponenten

Einer der schlimmsten Mängel der Objektorientierung ist die Abwesenheit von Komponenten als Ergänzung zur Klasse. Klassen sind einfach zu klein, zu granular. Sie sind Einheiten der Implementierung, nicht der Konstruktion; große Systeme lassen sich damit kaum strukturieren. Es gibt eine ganze Reihe von Großprojekten, die an dieser schlichten Tatsache gescheitert sind: Ein System, das aus 20.000 nicht weiter strukturierten Klassen besteht, wird nicht laufen und schon gar nicht wartbar sein.

Ernstzunehmende Systeme bestehen aus umfangreichen Komponenten mit definierten Schnittstellen, und nicht aus einem großen Sack von Klassen. Hierarchisch gegliederte Komponenten sind das wichtigste Element der Software-Architektur: die Strukturierung eines Systems in Komponenten und die Prinzipien und Formen der Kooperation und Interaktion.

3.4. Vererbung und Polymorphie

Eines der wichtigsten Merkmale der Objektorientierung ist die Vererbung. In der Praxis wird Vererbung oft mit Implementierungsvererbung gleichgesetzt: Die abgeleitete Klasse *erbt* alle Methoden und Attribute der Vaterklasse. In der abgeleiteten Klasse kann man neue Methoden und Attribute hinzufügen (additive Vererbung); man kann aber auch vorhandene Methoden und Attribute modifizieren oder sogar ungültig machen (modifizierende Vererbung). Letztlich gibt es über die abgeleitete Klasse nur eine einzige, syntaktische Annahme: Sie hat mindestens dieselben Methoden und Attribute wie die Oberklasse.

Neben der Implementierungsvererbung gibt es die Verhaltensvererbung (Behavior Inheritance), die z.B. in Java durch die Vererbung von Schnittstellen möglich ist. Verhaltensvererbung ist ein sehr schlichtes Konzept und in gewisser Weise das Gegenteil der Implementierungsvererbung: Wer von eine Klasse erbt (Implementierungsvererung), der *bekommt etwas geschenkt* – er spart ein paar Programmzeilen. Wer von einer Schnittstelle erbt (Verhaltensvererbung), der *muß etwas tun*, denn er verpflichtet sich, das in der Schnittstelle definierte Verhalten zu implementieren.

3.4.1. Kritik der Implementierungsvererbung

Implementierungsvererbung ist der zentrale Schwachpunkt der Objektorientierung. Dafür gibt es drei Gründe.

Unverständlicher Code
Implementierungsvererbung verletzt das Prinzip der Datenabstraktion, denn wer die *Implementierung* einer Klasse ändert, beeinflußt damit alle Unterklassen in

unvorhersehbarer Weise. Implementierungsvererbung ist im wesentlichen ein technischer Trick, um ein paar Codezeilen zu sparen, aber sie ist kein Hilfsmittel für die Strukturierung großer Systeme. Wiederverwendung mit Hilfe von Vererbung bedingt die genaue Kenntnis der Implementierung der Oberklasse und ist insofern ein wesentlich schwächeres Prinzip als die Wiederverwendung durch Unterprogramme, die uns seit den 50iger Jahren vertraut ist. Implementierungsvererbung führt oft zu unverständlichem Code: Weil die *Implementierung* einer Klasse auf eine beliebige Menge von Oberklassen verteilt sein kann (und dies ohne explizite Funktionsaufrufe!), sieht man ihr beim besten Willen nicht mehr an, was sie eigentlich tut. Unsere schlimmsten Altlasten sind nicht die Assembler-Systeme der späten Sechziger, sondern die C++-Systeme der frühen Neunziger.

Undefinierte Semantik
Zahlreiche Lehrbücher, z.B. [Meyers 96], empfehlen, Implementierungsvererbung ausschließlich im Sinn von Generalisierung/Spezialisierung zu verwenden: Der Firmenkunde *ist ein* spezieller Kunde; der Kunde ist eine Generalisierung von Firmenkunde und Privatkunde. Leider liegt die Einhaltung dieses Prinzips (z.B. durch Beschränkung auf additive Vererbung) allein in der Verantwortung des Programmierers und vielleicht noch der Qualitätssicherung. Es wird in der Praxis häufig mißachtet, was ein Blick auf die Java-Standardbibliothek sofort belegt. In der Praxis ist die Semantik der Vererbung oft reduziert auf die Aussage „Die abgeleitete Klasse *ist so etwas ähnliches wie* die Oberklasse".

Beschränkung auf ein Projekt
Implementierungsvererbung funktioniert so gut wie nie über Projektgrenzen hinweg. Das bewußt simple Beispiel mit den drei Kundenklassen würde in der Praxis so laufen: In Projekt A gibt es nur Privatkunden; Firmenkunden sind nicht absehbar. Deshalb wird nur die Klasse Kunde implementiert – die in Wirklichkeit den Privatkunden darstellt. Ein Jahr später kommt Projekt B; dort sind Firmenkunden zu modellieren. Nun gibt es drei Möglichkeiten:

a) Projekt B ignoriert die Kundenklasse von Projekt A und baut seine eigenen Klassen. Das ist der wahrscheinlichste Fall. Hier findet keine Wiederverwendung statt, aber es entsteht auch kein Schaden.
b) Projekt B leitet seinen Firmenkunden vom Kunden aus Projekt A ab. Damit wird etwas Code gespart; die Semantik der Vererbung wurde allerdings reduziert auf "ist so etwas ähnliches wie". Diese Variante, die in der Praxis gelegentlich zu beobachten ist, endet in einer Katastrophe, wenn man sie im großem Maßstab anwendet, denn der resultierende Code ist völlig undurchschaubar.
c) Projekt A oder Projekt B baut die vorhandene Klasse Kunde (die in Wirklichkeit einen Privatkunden darstellt) so um, dass eine sachgemäße Vererbungshierachie entsteht. Das ist noch nie passiert, denn niemand ändert ein laufendes System nur um der Schönheit willen.

Die Idee der Code-Wiederverwendung mit Hilfe von Implementierungsvererbung ist beim Bau von Prototypen hilfreich. Dort kann die Einsparung von Code-Zeilen den

Gesamtaufwand tatsächlich in gewissen Grenzen reduzieren. Bei der Entwicklung großer Systeme ist die Codemenge aber kein entscheidender Faktor. Wichtig sind vielmehr Lesbarkeit, Änderbarkeit, Wartbarkeit – und die Implementierungsvererbung ist ein probates Mittel, um diese Eigenschaften zu verschlechtern.

Implementierungsvererbung ist höchstens akzeptabel, wenn sie Implementierungsgeheimnis der abgeleiteten Klasse ist. Das ist der Fall in Eiffel und bei der privaten Ableitung in C++.

3.4.2. Polymorphie

Oft wird Polymorphie und – damit eng zusammenhängend – dynamisches Binden für die Objektorientierung ins Feld geführt. In Wirklichkeit gibt es nur eine einzige gesunde Form der Polymorphie: Die Austauschbarkeit unterschiedlicher Implementierungen einer Schnittstelle (eines wohldefinierten, beobachtbaren Verhaltens). Wir betrachten als Beispiel die algebraische Struktur „Ring" der Mathematik als Java-Schnittstelle:

```
public interface Ring
{
   Object add( Object x, Object y );
   Object subtract( Object x, Object y );
   Object multiply( Object x, Object y );
   Object negate( Object x );
   Object zero();
   Object one();
}
```

Abb. 1: Schnittstelle Ring

Die Semantik dieser Schnittstelle haben die Mathematiker genauestens definiert. Man kann jetzt den Ring mithilfe ganz unterschiedlicher Klassen implementieren (z.B. *BigInteger*, *Rational*, *Matrix*) – und jetzt kommt die Polymorphie ins Spiel: Alles, was sich mit Ringen machen läßt (z.B. Polynome, Matrizen) programmiert man mithilfe der Polymorphie ein für alle Mal gegen die Ring-Schnittstelle. Leider sind die Schnittstellen, die uns in der Praxis begegnen, alles andere als klar definiert. Dies reduziert den Nutzen der Verhaltensvererbung, denn oft wissen wir nur ungenau, welches Verhalten wir eigentlich erben.

3.5. Daten und Funktionen verbinden?

Die Verbindung von Daten und Funktionen funktioniert nicht immer: Zur Klasse *Konto* passen zwar die Methoden *einzahlen* und *abheben*, aber nicht *toXML*, *toSQL*, *toEDIFACT*, *toRTF* usw. Das Visitor-Muster [Gamma 94] hilft bei der Vermeidung überfrachteter Klassen, aber der Preis ist hoch: viele zusätzliche Klassen, komplizierte Aufrufsequenzen und mäßige Performanz. *Multiple Dispatching* macht das ungeliebte

Visitor-Muster überflüssig, aber es unterminiert die Objektorientierung. Dies wollen wir kurz erläutern:

Methoden im Sinn der Objektorientierung sind nichts weiter als Funktionen mit einer Sonderbehandlung des ersten Arguments. Beispiel: Die Java-Anweisung

k.print();

wählt in Abhängigkeit vom aktuellen Typ von *k* (z.B. Privatkunde oder Firmenkunde) die passende *print*-Methode aus. Diesen Mechanismus nennt man Single-Dispatching im Gegensatz zu Multiple-Dispatching, wo die auszuführende Methode von allen Argumenten (und ev. sogar vom Rückgabewert) abhängt. Java und C++ implementieren Single-Dispatching, Ada95 und Dylan [Shalit 96] sind Beispiele für Programmiersprachen mit Multiple-Dispatching.

Der Punkt ist nun: Im Widerspruch zu einem Grundgedanken der Objektorientierung trennt Multiple-Dispatching Daten und Funktionen! Bei Multiple-Dispatching ist es nicht möglich, Funktionen einer einzelnen Klasse zuzuordnen, denn alle Argumente sind gleichberechtigt. Daher verlieren die gängigen Sichtbarkeitsregeln auf Klassenebene (public, protected, private) ihren Sinn.

In Ada95 und in Dylan steht nicht die Klasse im Vordergrund der Überlegungen sondern das Paket (Ada95) bzw. das Modul (Dylan). Beide Sprachen sind also nur in einem eingeschränkten Sinn objektorientiert und vermeiden so einige Nachteile der vollen Objektorientierung.

3.6. Sequentialität

Das objektorientierte Paradigma ist von Grund auf sequentiell: Alles geschieht innerhalb von Methodenaufrufen; jede Methode kann andere Methoden nach Belieben rufen und andere Objekte nach Belieben verändern. Somit operiert jede Methode atomar auf einem riesigen globalen Zustandsraum. Der Fortschritt gegenüber den Zeiten vor der Objektorientierung liegt darin, dass die früheren globalen Variablen heute sauber in Objekten verpackt sind – aber der Zustandsraum ist immer noch genauso groß, und in Bezug auf Parallelität wurde nichts gewonnen.

Die objektorientierten Sprachen unterstützen Parallelität entweder gar nicht (wie C++) oder auf einer sehr implementierungsnahen Ebene (wie Ada und Java). Die Einführung von Parallelität bei großen Zustandsräumen beschert uns alle bekannten Probleme: Atomare Aktionen, Koordinierung und Synchronisierung, sowie Wartesituationen.

Selbstverständlich kann man mit den bekannten, implementierungsnahen Verfahren arbeiten – aber dabei verläßt man die Abstraktionsebene der übrigen objektorientierten Welt. Leider ist das unsere einzige Option, denn fast alle großen Systeme laufen hochgradig parallel auf einem Netz von unabhängigen Rechnern.

Aus der Sicht der Praxis interessiert vor allem die folgende Frage: Wie weit kann man Software entwerfen, ohne über Parallelität nachzudenken? Es wäre schlimm, wenn wir auch bei einfachen Klassen (z.B. Strings) oder Anwendungsklassen (z.B.

Buchung, Konto) Parallelität berücksichtigen müssten. In der alten Host-Welt konnte man die gesamte Anwendung (z.B. in Cobol) schreiben als gäbe es nur einen einzigen Prozess und einen einzigen Benutzer; die gesamte Parallelität wurde vom TP-Monitor (z.B. CICS) und der Datenbank behandelt. Neue Anwendungen auf Client/Server-Architekturen oder allgemeinen Rechnernetzen erfordern jedoch ein höheres Maß an Parallelität. Dies benötigt auch eine angemessene Unterstützung durch Sprachkonzepte.

4. Objektorientierte Spezifikation und Konstruktion – eine Kritik

In diesem Abschnitt wollen wir ein paar kritische Punkte bei der objektorientierten Analyse und dem Entwurf anpacken, insbesondere bei Sprachen wie UML. Vieles von dem was wir kritisch über objektorientierte Programmiersprachen angemerkt haben, trifft – sogar in erschwerter Form – auf objektorientierte Modellierungssprachen zu.

4.1. Systemmodell und semantische Integration

Eine Software- und System-Entwicklungsmethode braucht eine System-Philosophie, die ein klares Konzept eines Systemmodells enthält. Solch ein Systemmodell leistet folgendes: Es dient als Richtlinie und Leitfaden bei der Systementwicklung, es liefert eine Basis für das Verstehen der Semantik, es dient der Integration der verschiedenen Beschreibungstechniken und schafft somit einen Integrationsrahmen.

Selbstverständlich erhält man durch Objektorientierung eine spezifische Systemphilosophie und ein spezifisches Systemmodell. Das Systemmodell erfüllt jedoch nicht die oft formulierten Ansprüche, dass die Objektorientierung ein natürliches Modell für die Wirklichkeit liefert. Durch einen Prozeduraufruf können nicht alle möglichen Formen der Interaktion dargestellt werden. Zudem ist Nebenläufigkeit ein Phänomen der realen Welt. Nebenläufigkeit wird jedoch in objektorientierten Modellierungssprachen wie UML überhaupt nicht angesprochen (siehe Abschnitt 4.3).

4.2. Software und Systemarchitektur

Eine der wesentlichen Aufgaben einer Entwurfssprache ist die Beschreibung von Software- und Systemarchitektur. Dies ist wahrscheinlich die größte Schwachstelle heutiger objektorientierter Modellierungssprachen.

Bei UML ist es beispielsweise nicht möglich, ein Komponentenkonzept mit angemessener Granularität festzulegen. Dies ist besonders ärgerlich, da Komponenetenkonzepte heute doch das Kernstück von Softwarearchitektur-Beschreibungen sind.

4.3. Nebenläufigkeit

Heute sind praktisch alle interessanten Systeme nebenläufig, verteilt und interaktiv. Abstrakte Modelle moderner Softwaresysteme erfordern daher gut ausgearbeitete Systemmodelle, die alle Formen von Nebenläufigkeit und Interaktion unterstützen.

Der Formalismus und die Form der Zustandsdiagramme, auf denen die Diagrammtechnik des Zustandsautomaten bei UML basiert, kann leicht auf ein nebenläufiges Ausführungsmodell erweitert werden. Dieses wird jedoch bei den heute verbreiteten OOA/OOD-Techniken nicht verwertet.

4.4. Objektorientierte Spezifikation

Es ist bis heute nicht klar, welche OO-Elemente in der Spezifikation wirklich ihren Platz haben. In der Spezifikation sind die Daten und die Funktionen des Systems zu beschreiben – für die Daten haben wir die E/R-Modellierung, und in der Praxis schauen viele OO-Modelle genauso aus wie die alten E/R-Modelle. Gerade in der Spezifikation sollte es offensichtlich sein, dass wir Objekte durch ihre fachlichen Schlüssel identifizieren. Mit technische Konzepten wie Objekt-Ids befassen wir uns – wenn überhaupt – erst in der Konstruktion

In der Spezifikation ist der Begriff der Klasse als Einheit von Daten und Funktionen ziemlich unerheblich; die ohnehin dünne Unterscheidung zwischen Methode und Funktion ist hinderlich. Der Anwender, der die Spezifikation ja lesen soll, wird sich für solche Feinheiten nicht interessieren; Dinge wie Polymorphie, Konstruktoren, Destruktoren oder Angaben zur Sichtbarkeit (private, protected, public) haben in der Spezifikation nichts verloren.

Zuviel OO-Konzepte in der Spezifikation bergen die sehr reale Gefahr der Überspezifikation: Was in der Spezifikation oft zufällig festgelegt wurde, gilt in späteren Phasen als beschlossen.

5. Schlußbemerkung

Trotz der hier vorgetragenen Kritik soll nicht vergessen werden, daß die Objektorientierung viele wertvolle Konzepte und Prinzipien in die Programmentwicklung eingebracht hat. Allerdings sind Verbesserungen erforderlich. Sind wir in der Lage, Softwaresystem-Entwurf und Programmieren so anzugehen, dass es nicht die Schwächen der Objektorientierung heute aufweist und trotzdem die meisten Vorteile beibehalten werden können? Wir glauben ja! Es gibt Methoden zur Programmierung von verteilten Systemen, ausgeführt von Zustandsautomaten (wie z.B. bei Zustandsdiagrammen), die asynchrone Modelle nebenläufiger Ausführung unterstützen. Eine interessante Methode dieser Art ist ROOM, die in einer sehr konsequenten Form die benötigten Techniken einführt.

Eine Generalisierung diesen Modells entlang der Linie von Focus (s. [Broy 98]) und des Werkzeugs des Prototypen CASE, AutoFocus (s. [AutoFocus 00]) erschließt

viele der oben aufgeführten Eigenschaften . Die Einführung des klassischen Konzeptes der Objektorientierung bei diesem Modell wäre eine interessante Übung.

Auch sind Programmiersprachen denkbar, die die genannten Schächen der heutigen Objektorientierung vermeiden, indem sie unter anderem angemessene Konzepte für Softwarekomponenten und Parallelität bereitstellen. Wie gut aber sind die Aussichten, daß solche Sprachen nicht nur als wissenschaftliche Experimente entstehen, sondern auch ihren Weg in die Praxis finden. Dies erfordert, wie wir wissen, weit mehr als nur gutes Sprachdesign. Voraussetzung ist die breit angelegte Unterstützung der Sprache von maßgeblichen Organisationen oder Unternehmensgruppen und ihre Durchsetzung auf unterschiedlichen Plattformen. Undenkbar? Wer weiß! Auch vor der Einführung von Java konnten sich viele Informatiker kaum vorstellen, dass es ein Leben nach C oder C++ geben könnte.

Danksagung

Wir danken allen, die durch Diskussionen zu unseren Überlegungen beigetragen haben. Namentlich sei Herr Bernhard Rumpe erwähnt, der eine Fassung unseres Beitrags kritisch gegengelesen hat.

6. Literatur

[AutoFocus 00]
P. Braun, H. Lötzbeyer, B. Schätz, O. Slotosch: Consistent Integration of Formal Methods. In: Proc. 6th Intl. Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00), 2000,

[Beeck 94]
M. v. d. Beeck: A Comparison of Statecharts Variants. In: H. Langmaack, W.-P. de Roever, J. Vytopil (eds): Formal Techniques in Real Time and Fault-Tolerant Systems. Lecture Notes in Computer Science 863, Berlin: Springer 1994, 128-148

[Booch 91]
G. Booch: Object Oriented Design with Applications. Benjamin Cummings, Redwood City, CA, 1991

[Broy 91a]
M. Broy: Towards a Formal Foundation of the Specification and Description Language SDL. Formal Aspects of Computing 3, 21-57 (1991)

[Broy 92]
M. Broy: Compositional Refinement of Interactive Systems. Journal of the ACM, Volume 44, No. 6 (Nov. 1997), 850-891. Also in: DIGITAL Systems Research Center, SRC 89, 1992.

[Broy 93]
M. Broy: (Inter-)Action Refinement: The Easy Way. In: Broy, M. (ed.): Program Design Calculi. Springer NATO ASI Series, Series F: Computer and System Sciences, Vol. 118, pp. 121-158, Berlin, Heidelberg, New York: Springer 1993

[Broy 95b]
M. Broy: Advanced Component Interface Specification. In: Takayasu Ito, Akinori Yonezawa (Eds.). Theory and Practice of Parallel Programming, International Workshop TPPP'94, Sendai, Japan, November 7-9, 1994, Proceedings, Lecture Notes in Computer Science 907, Berlin: Springer 1995

[Broy 98]
M. Broy: Compositional Refinement of Interactive Systems Modelled by Relations. In: W.-P. de Roever, H. Langmaack, A. Pnueli (eds.): Compositionality: The Significant Difference. LNCS State of the Art Survey, Lecture Notes in Computer Science 1536, 1998, 130-149

[Dahl et al 72]
O. Dahl, E.W. Dijkstra, C.A.R. Hoare (eds.): Structured Programming. Academic Press 1971

[Gamma94]
Gamma, E. et al.: Design Patterns. Addison-Wesley 1994

[Harel 87]
D. Harel: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8, 1987, 231 - 274

[Jacobson 92]
I. Jacobson: Object-Oriented Software Engineering. Addison-Wesley, ACM Press 1992

[Meyers 96]
S. Meyers: More Effective C++, Addison-Wesley, 1996

[Mitchell 90]
Type Systems for Programming Languages. In: van Leeuwen (Ed.). Handbook of Theoretical Computer Science, Vol B. Elsevier Science Publishers, 1990

[Rumbaugh 91]
J. Rumbaugh: Object-Oriented Modelling and Design. Prentice Hall, Englewood Cliffs: New Jersey 1991

[Philipps, Scholz 95]
J. Philipps, P. Scholz: Compositional Specification of Embedded Systems with Statecharts. In: Theory and Practice of Software Development TAPSOFT'97, Lille, Lecture Notes in Computer Science 1214, Berlin: Springer 1995

[SDL 88]
Specification and Description Language (SDL), Recommendation Z.100. Technical report, CCITT, 1988

[Shalit 96]
The Dylan Reference Manual. Addison-Wesley, 1996

[Siedersleben, Denert 00]
Wie baut man Informationssysteme? Überlegungen zur Standardarchitektur. Informatik Spektrum, Vol. 23, 247-257, 2000

[Simula 67]
Dahl, O.-J., B. Myrhaug, K. Nygaard: Simula 67 - common base language. Technical Report N. S-22, Norsk Regnesentral (Norwegian Computing Center), Oslo.

[UML 97]
G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language for Object-Oriented Development, Version 1.0, RATIONAL Software Cooperation

[Wirsing 90]
M. Wirsing: Algebraic Specification. Handbook of Theoretical Computer Science, Vol. B, Amsterdam: North Holland 1990, 675-788