

Was, bitte, bedeutet Objektorientierung?

Stefan Jähnichen · Stephan Herrmann

Wir begrüßen die Offenheit des Informatik Spektrums, Diskussionen über essentielle Fragen der Informatik aufzunehmen und zu publizieren. In diesem Sinne begrüßen wir auch den Artikel von Broy und Siedersleben [5], da er eine offensichtlich notwendige Diskussion anstößt und eine unbequeme Position bezieht, die einen vielfach stillen Konsens zum Allheilmittel Objektorientierung angreift. Allerdings enthält der genannte Artikel einige Ungenauigkeiten, die klargestellt werden sollten; an vielen Stellen regt er uns aber auch an, den Autoren direkt zu widersprechen.

Richtung objektorientierte Konzepte verändert und ergänzt werden müssen, um den Anforderungen der Zukunft gerecht zu werden. Wir greifen den Artikel von Broy und Siedersleben als Anlass auf, diese Fragen zu erörtern.

Wir möchten durch ausdrückliche Einbeziehung des geschichtlichen Aspekts die Richtung vergangener und zukünftiger Entwicklung thematisieren. Vieles über diese Entwicklungsdynamik können

1 Einleitung

Mehr noch motiviert der Beitrag aber zu einer genauen Bestimmung und Einordnung des Gegenstandes der Objektorientierung im *Gesamtkontext* der Softwaretechnik, da die Bedeutungsvielfalt dieses Begriffes ansonsten zu heillosen Missverständnissen führen muss. Wenn wir nach der *Bedeutung* der Objektorientierung fragen, so suchen wir nicht einfach nach einem Wörterbucheintrag, sondern fragen auch nach der Relevanz der Objektorientierung und danach, welche Rolle sie für die Softwareentwicklung spielt. Es ist unsere Überzeugung, dass es längst keine Frage mehr ist, *ob* Objektorientierung in der Softwaretechnik eine Bedeutung hat, sondern vielmehr, in welcher

wir besser begreifen, wenn wir uns darüber klar werden, wie wir dahin kamen, wo wir jetzt stehen.

1.1 Ist Objektorientierung kritikfähig?

Lohnt es überhaupt, über Objektorientierung zu streiten? Auf der einen Seite begrüßen wir den Vorstoß von Broy und Siedersleben, der eine stillschweigende Übereinkunft aufzubrechen versucht, mit der Objektorientierung sei die Entwicklung von Programmiersprachen zu ihrem Abschluss gelangt. Seit dem Aufkommen von Java ist es in der Tat schwierig, über einige grundlegende Fragen zu diskutieren, da die an vielen Stellen anzutreffende Zufriedenheit mit dieser Sprache und ihren begleitenden Bibliotheken und Werkzeugen für derlei Kritik blind macht. (Ob diese Zufriedenheit durch C# nachhaltig erschüttert ist, oder ob das nur ein letztes Zucken der Entwicklung ist, bleibt abzuwarten.)

Auf der anderen Seite sehen wir in dem Artikel in vielen Punkten eine Vereinfachung des Gegenstandes, die den Nutzen der Diskussion gefährdet. Drei Dimensionen sollen in diesem Beitrag untersucht werden, die einen reichhaltigen Raum von Konzepten aufspannen, eine kombinatorische Vielfalt, die es vielleicht sogar rechtfertigt, Objektorientierung nicht als eine Technik, sondern als ein Paradigma einzustufen. Diese Dimensionen sind:

- Design von (objektorientierten) Programmiersprachen,
- Methodik für die Entwicklung objektorientierter Programme,
- technische Infrastruktur für (objektorientierte) Systeme.

Es dürfte Konsens sein, die Unabhängigkeit dieser Diskursbereiche anzustreben. Wir werden jedoch

Stefan Jähnichen, Stephan Herrmann
Fraunhofer FIRST,
Technische Universität Berlin
E-Mail: Stefan.Jaehnichen@first.fhg.de;
stephan@cs.tu-berlin.de

später noch auf Grenzbereiche zu sprechen kommen, die es teilweise in Frage stellen, hier von unabhängigen „Dimensionen“ zu sprechen.

Bei allem muss mit der historischen Entwicklung die Zeit als eigenständige Dimension berücksichtigt werden. Für eine sinnvolle Diskussion muss stets klar sein, über welchen Ort in diesem komplexem Raum gesprochen wird.

1.2 Herkunft der Objektorientierung

Bevor wir die drei Hauptdimensionen – Sprachen, Methoden, Technik – auffächern, zunächst eine kurze Rekapitulation, wo die Softwaretechnik stand, als die Objektorientierung aufkam. Wir erlauben uns, für die „Geburt“ der Objektorientierung einen Zeitraum von 1967 bis 1988 anzusetzen, der wesentlich durch die Programmiersprachen Simula, Smalltalk und Eiffel geprägt wurde. Diese drei Sprachen illustrieren drei Beweggründe, objektorientiert zu arbeiten (die Zuordnung dieser Beweggründe zu den Sprachen ist natürlich eine grobe Vereinfachung).

Es ist wohl kein Zufall, dass der erste Ansatz zur Objektorientierung aus dem Bereich der Simulation kam. Nachdem z. B. COBOL erlaubte, die Daten eines Formulars zu einer Datenstruktur zusammenzufassen, sollten nun auch reale „Objekte“ im Rechner repräsentiert werden, die durch ein gewisses Eigenleben gekennzeichnet sind. Eine treibende Kraft war mithin die *natürliche Modellierung* von Realität, wobei das Konzept „Objekt“ eine aussagekräftige Metapher bietet.

Nahezu unabhängig von dieser Metapher entwickelte die Informatik Techniken zur *mehrfachen Nutzung* gemeinsamer Berechnungen etc. Nach den ersten Erfolgen durch Unterprogramme, die von verschiedenen Programmstellen aus aufgerufen werden können, verlagerte insbesondere die Entwicklungsumgebung von Smalltalk den Fokus von den Prozeduren auf jene „Dinge“, die teilweise gemeinsame *Eigenschaften* haben. Das klassische Beispiel für solche „Dinge“ sind die Elemente einer graphischen Benutzungsoberfläche, wie sie von Smalltalks Entwicklungsumgebung bereitgestellt wurden. Nahezu alle dieser Elemente können z. B. Mausevents empfangen und auf dem Bildschirm gezeichnet werden. Teilweise kann dabei dieselbe Implementierung verwendet werden, teilweise ist nur die Fähigkeit die gleiche, jedoch mit unterschiedlicher Implementierung. Die Diskussionen, durch

welche Mechanismen Objekte gemeinsame Eigenschaften teilen können („sharing“), mündete in dem „Treaty of Orlando“ [39], wo u. a. die Äquivalenz von Vererbung (zwischen Klassen) und Delegation (zwischen Objekten) festgehalten wurde.

Dritter Einflussfaktor war das Konzept der *abstrakten Datentypen* (ADT). Nachdem die Konzepte von ADTs zur formalen Spezifikation etabliert waren, bot sich die aufkommende Objektorientierung an, um diese Konzepte auch für die Programmierung nutzbar zu machen. Hier dient die Sprache Eiffel als Vorbild der Umsetzung folgender Konzepte: Objekte werden *gekapselt*, indem der interne Zustand derart versteckt wird, dass er nur noch durch die Prozeduren einer expliziten Schnittstelle verändert werden kann. Es wird also zwischen interner Repräsentation des Zustandes und dem beobachtbaren Verhalten unterschieden. Zweitens gewährleistet die Klassifizierung von Objekten durch *Typen*, dass Objekte nur entsprechend der Schnittstelle ihres Typs benutzt werden können. Drittens ist die Semantik eines Typs formal (wenn auch selten vollständig) *spezifiziert*, was im Falle von Eiffel durch Vor-, Nachbedingungen und Invarianten erfolgt.

Diese drei Bereiche, natürliche Modellierung, *Sharing* und ADTs geben – auch ohne präzise Definition verwendeter Mechanismen – das *Wesen* der Objektorientierung seit ihrer frühen Phase wieder.

Im nächsten Abschnitt werden wir uns den konstituierenden inneren Eigenschaften der Objektorientierung widmen. Erst mit dem dabei gewonnenen Wissen können wir mögliche Defizite identifizieren und ihre Ursache lokalisieren. Zum Schluss wagen wir einen Blick in die (gar nicht so ferne) Zukunft, die nach unserer Einschätzung weitgehend von der Objektorientierung geprägt sein wird, ohne dass noch ausdrücklich darüber gesprochen werden muss: Objektorientierung wird eine selbstverständliche Basistechnik sein, Diskussionen und Innovationen werden auf höheren Abstraktionsebenen stattfinden.

Dass wir der Objektorientierung solch eine Rolle zuschreiben, soll nicht sagen, dass wir sie für perfekt halten, sondern eher, dass die Objektorientierung in unseren Augen eine geeignete Grundlage ist, um viele weitergehende Konzepte zu integrieren.

2 Dimensionen der Objektorientierung

Nach der historischen Sicht wollen wir uns nun der Objektorientierung auf analytische Art und Weise nä-

hern. Die oben erwähnten Dimensionen sollen uns dabei zur Strukturierung der Betrachtung dienen: Sprachen, Methoden und technische Infrastrukturen.

2.1 Objektorientierte Programmiersprachen

Welche Eigenschaften zeichnen eine objektorientierte Programmiersprache aus? Zunächst ist es wichtig, nicht nur ein oder zwei Sprachen zu betrachten, sondern sich die Vielfalt vor Augen zu führen. Neben dem Hauptstrang der rein imperativen, klassenbasierten, (überwiegend) statisch typisierten Sprachen gibt es mindestens folgende Varianten:

- Sprachen ohne statische Typisierung (z. B. Smalltalk) [14], die das Konzept der ADTs nur unvollständig umsetzen;
- klassenlose, prototypbasierte Sprachen (Self) [45], die sich für Delegation anstatt Vererbung entschieden haben;
- objektorientierte Derivate von funktionalen Sprachen (CLOS, O'Caml) [19, 37];
- Erweiterung von Java um funktionales Programmieren (Pizza) [32];
- beta weicht so gründlich vom Hauptstrang ab, dass die Unterschiede in dieser kurzen Auflistung keinen Platz finden [11, 25].

Trotz ihrer Vielfalt erfüllen alle diese Sprachen das Wesen der Objektorientierung z. B. im Sinne der oben genannten Aspekte „natürliche Modellierung“, „Sharing“ und ADTs. Will man diese Diskussion technisch präziser führen, so lohnt ein Blick in eine Arbeit von Wegner aus dem Jahr 1987 [47]. Darin hat Wegner den minimalen, orthogonalen Satz von Konzepten identifiziert, um die bislang diskutierten Dinge zu klassifizieren (unsere Übersetzung):

- **Objekte:** „modulare Berechnungsagenten“,
- **Typen:** „Mechanismus zur Klassifizierung von Ausdrücken“,
- **Delegation:** „Mechanismus zur gemeinsamen Nutzung von Ressourcen“,
- **Abstraktion:** „Mechanismus der Schnittstellenspezifikation“.

Die Liste, die Broy und Siedersleben zur Definition der Objektorientierung angeben, erscheint uns vergleichsweise willkürlich und ungenau. Sie kann aber auf Wegners vier Konzepte abgebildet werden:

- Klassen implementieren Objekte und definieren gleichzeitig ihren Typ (eine Vermischung von Konzepten, die immer wieder kontrovers diskutiert wird).
- Schnittstellen¹ sind Abstraktionen.

- Objektidentität² ist Bestandteil des Objektkonzeptes.
- Objekterzeugung³ verknüpft das Objektkonzept mit Delegation, da Instanzen gemeinsame Implementierung teilen; Methodenaufrufe an das Objekt werden an seine Klasse bzw. sein *Parent*-Objekt delegiert.
- Vererbung⁴ ist ein Spezialfall von Delegation; Polymorphie ergibt sich in der Kombination mit Typisierung.

Bereits Wegner kommt zu dem Ergebnis, dass die orthogonalen Grundkonzepte nicht unbedingt das sind, was Programmierer direkt in die Hand bekommen sollten. Während die vier genannten Konzepte und Mechanismen für die sprachtheoretische Diskussion wichtig sind, ist beim Programmieren möglicherweise Vererbung interessanter, obwohl oder weil sie eine spezifische Mischung von Konzepten darstellt (vgl. auch [34]).

Am Rande noch ein Gedanke, den wir eigentlich bei Broy und Siedersleben erwartet hätten: Es gibt auch eine Welt außerhalb der Objektorientierung. In diesem Sinne haben funktionale und logische Programmierung nach wie vor ihre Berechtigung. Interessanterweise gibt es für beide Fälle Systeme, die diese Paradigmen mit der Objektorientierung verbinden [4, 32].

Alle weiteren Betrachtungen müssen neben den Konzepten der Programmierung auch die Umwelt mit einbeziehen: Es sind Menschen, die in einem kooperativen, kreativen und/oder ingenieurmäßigen Prozess Realität modellieren und diese Modelle in Softwaresysteme umsetzen. Eine Schlüsselfunktion auf dieser Ebene sind *Methoden* zur Softwareentwicklung. Dabei klammern wir aus dieser Diskussion organisatorische Rahmenbedingung und Notwendigkeiten der Softwareentwicklung (z. B. Projektmanagement, Konfigurationsmanagement) aus, da diese nur noch sehr indirekt mit der Objektorientierung in Verbindung gebracht werden können.

¹ Listen von Methodensignaturen – nicht, wie in [5] definiert, Methoden.

² Objektidentität wie in [5] durch Speicheradressen zu definieren ist eine hilfreiche, aber zu naive Anschauung. Relozierende Garbage Collectoren z. B. verschieben Objekte im Speicher, ohne ihre Identität zu verletzen. In rein objektorientierten Sprachen (was insbesondere C++ ausschließt) sind Referenzen opaque, d. h. ihre Implementierung durch Speicheradressen ist vollständig gekapselt.

³ Erzeugung braucht nicht zwangsläufig durch Klassen definiert zu sein, wie in [5] ausgeführt; auch Cloning erfüllt denselben Zweck.

⁴ Die Diskussion über objektbasierte Vererbung erlebt in jüngster Zeit eine Renaissance (vgl. [23, 35]).

2.2 Objektorientierte Methoden

Spätestens bei der Diskussion über Methoden muss klar werden, dass Softwareentwicklung sehr unterschiedlich aussieht, abhängig davon, welche Art von Software erstellt werden soll. Eine Methode bündelt Erfahrungswissen aus einer Klasse von Entwicklungsprojekten. Diese Klassen werden einerseits durch den Gegenstandsbereich und andererseits durch Qualitätsanforderungen an das Produkt bestimmt. Eine Webdatenbank wird anders entwickelt als eine Satellitensteuerung; das Ziel der Wartbarkeit erfordert eine andere Methode als das der optimalen Performanz bei beschränkten Ressourcen.

Methoden sind teilweise speziell auf Objektorientierung zugeschnitten, andere Anteile mögen universell einsetzbar sein. Zum Beispiel schreibt die *Fusion-Methode* [7] vor, in der Analyse ein Klassendiagramm ohne Feinstruktur der einzelnen Klassen zu zeichnen.⁵ Eine Anleitung zur Testfallermittlung für das Blackbox-Testen soll unabhängig von allen verwendeten Implementierungstechniken sein.

Man darf niemals vergessen, dass gute Software nur entwickelt werden kann, wenn die Menschen, die daran arbeiten, gute Arbeit leisten. Programmiersprachen können nur einen äußeren Rahmen schaffen. Methoden versuchen deutlich konstruktivere Hilfe anzubieten, sind allerdings von deutlich heuristischer, d. h. auch vager Natur. Am Rande sei bemerkt, dass es kein grundsätzliches Problem darstellt, objektorientiert zu entwerfen und den Entwurf anschließend in ein C-Programm umzusetzen. Die Korrespondenz zwischen Methode und Programmiersprache kann also ggf. durch wohldefinierte Transformationen hergestellt werden. In diesem Sinne können und sollen Methoden Abstraktionsmöglichkeiten bieten, die durch die benutzten Sprachen allein nicht gegeben sind.

Zum Beispiel lassen sich Entwurfsmuster [13] der Methodik zuschlagen. Sie geben den Entwicklern Konzepte an die Hand, die durch bekannte Arbeitsschritte in der jeweiligen Sprache und im jeweiligen Kontext umgesetzt werden. Häufig ist gerade zu beobachten, dass Entwurfsmuster dort an Bedeutung gewinnen, wo die Programmiersprache Wünsche offen lässt (vgl. z. B. die vielen Muster zur Nachahmung von Funktionen höherer Ordnung). Langfristig muss also die Erfahrung über Entwurfs-

muster in die Entwicklung von Programmiersprachen einfließen, um die mühsamsten Muster, die am häufigsten zur „Reparatur“ der Programmiersprache benutzt werden, allmählich überflüssig zu machen. Entwurfsmuster entschärfen in der Regel den Leidensdruck nur mittelfristig.

Anders formuliert, die Grenze zwischen den Problemen, die durch Programmiersprachen bzw. die Methode gelöst werden, ist nicht starr.

2.3 Technische Infrastruktur

Eine Methode steht gewissermaßen begleitend neben der eigentlichen Programmierung. Zusätzlich ist stets auch eine technische Umgebung nötig, in die das Programm eingebettet wird. Zunächst wollen wir dazu das jeweilige Betriebssystem zählen, das dem Programm Zugriff auf Geräte und andere Ressourcen gibt. Auch die Definition von Prozessen und Threads gehört in die Domäne von Betriebssystemen. Plötzlich lässt sich die Diskussion um Nebenläufigkeit gar nicht mehr allein auf der Sprachebene diskutieren, sondern mindestens das Betriebssystem muss mit hinzugezogen werden.

Nun kann es passieren, dass die Synchronisierungs- und Schedulingdienste des Betriebssystems zu unstrukturiert sind, so dass man sich eine zusätzliche, abstraktere Kapselung durch die Programmiersprache wünscht. In der Praxis ergibt sich folgendes Spektrum: die Thread-Unterstützung von Java ist stark simplifiziert und überlässt viele Konzepte der Kommunikation und Synchronisierung der händischen Implementierung (wobei bestimmte Anforderungen bereits an der ungenügenden Spezifikation des Scheduling scheitern mögen). Die meisten Betriebssysteme bieten da per se schon differenziertere Möglichkeiten. Ada [43] am anderen Ende des Spektrums ist für Anforderungen an die Kommunikation und Synchronisation zwischen Threads und für Aspekte des Scheduling deutlich besser ausgestattet. Dass beide Sprachen (mit ihrem immensen Unterschied gerade in diesen Punkten!) Parallelität „auf einer sehr implementierungsnahen Ebene“ unterstützen [5], ist eine Kritik, auf die eine Erwiderung ohne Spekulation über die Intention der Aussage schwer fällt. Was soll eine Programmiersprache anders sein als implementierungsnah?

Aber zurück zu den technischen Infrastrukturen. Wenn man nun Annotationen bezüglich der Synchronisation nicht überall im Programm verstreut sehen will, so bleibt z. B. der Griff nach einer

⁵ Dies macht die Kritik „Dinge wie Polymorphie, Konstruktoren, Destruktoren ... haben in der Spezifikation nichts verloren“ [5] gegenstandslos.

Komponenteninfrastruktur wie EJB [30] oder CCM [2], die es erlaubt, Transaktionsschutz und Ähnliches weitgehend deklarativ und außerhalb des Programmtextes zu definieren. Wenn man so will, sind Komponentencontainer mit ihren Services die direkte Weiterentwicklung von TP-Monitoren⁶ und ähnlichen Techniken. Vergleicht man nun Komponentencontainer und Applikationsserver mit Betriebssystemen, so findet man viele Ähnlichkeiten, merkt aber auch, dass es nicht nur darum geht, dem Programm Zugriff auf „darunter liegende“ Ressourcen zu verschaffen, sondern es werden auch neue Abstraktionen eingeführt, die helfen, Systeme zu strukturieren.

Wegners Liste von Konzepten der Objektorientierung [47] ist oben unvollständig zitiert. Er führt zusätzlich noch Nebenläufigkeit und Persistenz an. Beide Konzepte stehen jedoch aus heutiger Perspektive an der Schwelle zwischen programmiersprachlichen Lösungen und separaten Techniken, die entweder im Programm explizit aufgerufen oder durch generative Techniken (z. B. Präprozessoren) eingeflochten werden. Auch die explizite Konfiguration von Komponenten mittels eines Deployment-Deskriptors fällt in diese Kategorie von Techniken, die die verwendete Programmiersprache ergänzen.

2.4 Überlappung der Diskursbereiche

Das Problem mit der Diskussion über die Diskursbereiche Sprache, Methode und Infrastruktur ist, dass diese nicht orthogonal sind. Wir haben Beispiele gegeben, wo die Grenzen fließen, d. h. die Lösung für ein und dasselbe Problem in verschiedenen Bereichen gefunden werden kann. Genau aus diesem Grund ist es schwierig, einen Bereich einzeln zu diskutieren, zumal wenn äußerlich allein auf der Sprachebene diskutiert wird, aber implizit die Objektorientierung schlechthin in Frage gestellt wird.

3 Problemfelder der Objektorientierung

Nachdem eine differenzierte Betrachtung der Objektorientierung einige mögliche Kritikpunkte bereits beantwortet hat, wollen wir uns im Folgenden diesen vier Themengebieten näher zuwenden:

1. Objektidentität,
2. Referenzierung, Aliasing und Kapselung,
3. Vererbung,
4. Modularisierung.

Im Zuge der Diskussion dieser Themengebiete werden wir versuchen aufzuzeigen, dass in keinem dieser Bereiche die Entwicklung stehen geblieben ist. Während viele hilfreiche Ansätze einzelne Probleme in Isolation gelöst haben, ist offenbar eine Vielzahl von separaten Inseln entstanden, so dass der Nutzen jeweils nur einer kleinen Gemeinde zugute kommt. Die Vision, die wir der geäußerten Kritik entgegensetzen wollen, handelt von einem Kontinuum von Sprachtechnik und Werkzeugen, so dass Softwareentwicklungsprojekte nach ihren individuellen Anforderungen eine Methode nebst Werkzeugkasten zusammenstellen können, die den speziellen Qualitätsanforderungen und Rahmenbedingungen optimal Rechnung tragen. Nur so können wir von dem unheilvollen „one size fits all“ wegkommen, ohne die technologische Landschaft so weit zu zersplittern, dass Informatiker aus unterschiedlichen Projekten sich eines Tages gar nicht mehr verständigen können.

3.1 Objektidentität

Broy und Siedersleben beklagen, dass es schwierig sei, Welten mit unterschiedlichen Objektbegriffen „(z. B. die Java-Objekte im Hauptspeicher und die Objekte in einer OO-Datenbank)“ zusammenzuführen. Eine Lösung für diesen „Medienbruch“ erfordert allerdings nur je eine Proxyklasse für jede Datenbankklasse und eine Fabrikmethode oder -klasse (sofern nicht die Datenbank diese oder eine ähnliche Technik bereits mehr oder weniger transparent zur Verfügung stellt). Der Effekt ist, dass viele Projekte diese Lösung lieber schnell neu erfinden, als zu suchen, ob eine existierende Lösung verwendet werden kann.

Diese Lösung ist also höchstens „zu einfach“. Es ist ja gerade eine Essenz der Objektorientierung (die sie natürlich mit anderen Paradigmen teilt), dass Abstraktionen programmiert werden können, die dem Benutzer eines Programmmoduls etwas „vorgaukeln“, das in Wirklichkeit anders programmiert wurde. Eine Proxyklasse sieht *exakt* wie eine normale Klasse aus. Dies schließt auch die Objektidentität mit ein, die einfach durch eine Fabrikmethode oder -klasse gewährleistet werden kann. Die-

⁶ TP-Monitore seien hier nur erwähnt, um die Klage, früher, in der Zeit von CICS sei alles besser gewesen [5], zu entkräften. Ja, man kann auch heute noch Funktionalität von bestimmten anderen Aspekten trennen. Und dabei ist man nicht auf einen Dienst (Transaktionen) beschränkt, sondern kann auf ähnlichem Wege auch Dienste wie z. B. Security und Benachrichtigungen bekommen.

se einfache Technik garantiert, dass die Identität von zwei „externen“ Objekten (in einer Datenbank oder einem sonst wie entfernten Adressraum) genau mit der Gleichheit der Referenzen innerhalb des lokalen Programms zusammenfällt. Eine Unterscheidung ist nicht nötig!

3.2 Referenzierung, Aliasing und Kapselung

Es ist wahr, dass man auch mit objektorientierten Sprachen undurchschaubare Programme schreiben kann. Diese Beobachtung ist nicht neu („real programmers can program Fortran in any language“, will sagen, schlechter Stil ist nicht an eine Programmiersprache gebunden). Natürlich sind solche Programme am besten verständlich und analysierbar, bei denen Datenstrukturen, Programmstruktur sowie alle Steuerflüsse streng hierarchisch und statisch bekannt sind. Bei derartigen Programmen können keine undefinierten Fernwirkungen oder gar Bumerangeffekte auftreten. Und wenn streng hierarchische Programme Seiteneffekte haben, so finden diese nur in einem wohldefinierten Bereich statt.

„Leider“ ist die Welt nicht streng hierarchisch. Das heißt, die *angemessenste* Lösung für viele Modellierungsprobleme enthält Strukturen, die alles andere als hierarchisch sind. Wollte man Programmierer zwingen, beliebig komplexe Zusammenhänge allein durch Hierarchien abzubilden, so würden diese häufig genug dahin entarten, dass ein Großteil der eigentlich gekapselten Daten bei nahezu allen Funktionsaufrufen als Parameter übergeben werden müsste. Dies ist vergleichbar dazu, von vornherein (nahezu) alle Daten global zu deklarieren. Zwingt man eine Lösung in eine hierarchische Struktur, so erfordert dies häufig, wenige monströse Module zu erzeugen, da deren weitere hierarchische Zerlegung zu unlösbaren Konflikten führt.

Es kann als ein typisches Problem von objektorientierten Programmen angesehen werden, dass dynamisch Strukturen erzeugt werden (Graphen von sich gegenseitig referenzierenden Objekten), bei denen viele (z. T. statisch nicht bekannte) Pfade zu ein und demselben Objekt führen können (Aliasing). Werden mehrere dieser Pfade benutzt, um dasselbe Objekt zu modifizieren, so können schwer verständliche Situationen entstehen. Nebenbei bemerkt ist dies nicht eine Erfindung der Objektorientierung. Eine verzeigte Struktur von Modula-2-Records zeigt weitgehend dieselbe Eigenschaft.

Will man einerseits natürliche Modellierungen für nicht baumartige Datenstrukturen erlauben und andererseits Seiteneffekte bei Aliasing kontrollieren, so helfen diverse Techniken von Annotationen und Analysen, die am konkreten Programm nachweisen sollen, dass keine Anomalien auftreten können. Als ein aktueller Ansatz in diesem Bereich sei das Konzept der *Universes* angeführt [31]. Dieser Ansatz führt mit Mitteln des Typsystems eine Partitionierung aller Laufzeitobjekte – d. h. zusätzliche Kapselung – ein, so dass keine „unerwarteten“ Seiteneffekte erzeugt werden können. Derartige Programme können vollständig modular verifiziert werden.

Was die Implementierung von Wertsemantiken (Objekte, auf die keine Referenzen existieren) angeht, wäre es interessant, das Konzept der *expanded types* der Sprache Eiffel genauer mit *structs* in C# zu vergleichen. Da letztere eher aus Leistungsgründen in die Sprache aufgenommen wurden, ist ihre Verwendung für semantische Zwecke eher fraglich. Dabei könnte sich auch als hinderlich herausstellen, dass keine Möglichkeit besteht, die *Value*-Eigenschaft attributweise zu vergeben, da es sich schlicht um eine zweite Variante von *Klassen* handelt.

3.3 Vererbung

Vererbung ist in der Tat ein schillernder Begriff. Wir hatten bereits ausgeführt, dass Vererbung bewusst im Katalog der orthogonalen Konzepte *nicht* aufgezählt wird. Sie ist ein Konglomerat von verschiedenen Konzepten (vgl. [34, 41]), das ohne weitere Differenzierung leicht zu Missverständnissen und Programmfehlern führen kann. Mindestens müssen unterschieden werden:

- Subtyp-Beziehung,
- Erwerb von Merkmalen (der Oberklasse) (bei Wegner: Delegation).

Während beide Konzepte jeweils noch weiter unterteilt werden können, wird wenigstens diese Zweiteilung von einigen Programmiersprachen mehr oder weniger strikt unterstützt. Es geht hierbei darum, den üblichen breiten Begriff von Vererbung weiterhin benutzen zu können, ohne daraus falsche Schlussfolgerungen über die typmäßige Substituierbarkeit zu ziehen. Mit anderen Worten: es sollten nur ausgewählte Vererbungsbeziehungen als subtypbildend gekennzeichnet werden. Am weitesten geht hier Sather [40], aber auch Javas *inter-*

faces sind ein (kleiner) Schritt in dieser Richtung.

Ein Begriff, der ganz eng mit der Vererbung verknüpft ist, ist die Polymorphie. Ein Problem dieses Begriffes liegt darin, dass er recht unterschiedliche Konzepte zusammenfasst. Die Klassifikation von Cardelli und Wegner [6] nennt vier Kategorien, von denen für diese Betrachtung insbesondere parametrische Polymorphie (auch: Generizität) und Inklusionspolymorphie (auch: Subtyppolymorphie) relevant sind.

Broy und Siedersleben diskutieren Polymorphie anhand des Beispiels der algebraischen Struktur „Ring“. Von den Problemen dieses Beispiels sei hier zunächst herausgegriffen, dass eine Signatur

```
Object add (Object x, Object y);
```

suggeriert, man könnte tatsächlich beliebige Objekte miteinander addieren. Das kann keine Implementierung erfüllen, also ist solch eine Schnittstelle nicht *typsicher*. Eine erste Verbesserung kann in der Syntax von Generic Java (voraussichtlich wird dies auch mit JDK 1.5 möglich sein) angegeben werden:

```
interface Ring <T>{  
    T add (T x, T y);  
    /*...*/  
}
```

Die Einführung von parametrischer Polymorphie macht diese Schnittstelle *typsicher*. Auch nach dieser Verbesserung bleiben mindestens noch zwei Probleme: (1) Dieser Typ *Ring* modelliert keine eigenständigen Objekte sondern hat eher die Form einer traditionellen Funktionsbibliothek. (2) Selbst wenn man *Ring*-Objekte erzeugen würde, wären diese (aufgrund von Kovarianz) nicht inklusionspolymorph nutzbar.

All dies trifft aber die tatsächlichen Probleme von Polymorphie nicht, und zeigt allenfalls, dass Objektorientierung für das Konzept „Ring“ eine unpassende Lösung nahezulegen scheint.

Eine kritische Diskussion der Polymorphie muss benennen, dass Polymorphie nur dort sinnvoll ist, wo sie auf echtem Subtyping beruht, sprich: auch echter Austauschbarkeit.

Dass Vererbung nur unter bestimmten Bedingungen eine Subtypbeziehung erzeugt, hatten wir oben angedeutet.

Wie kann man nun „echtes“ Subtyping von Vererbung unterscheiden? Der Schlüssel liegt natürlich (wie Broy und Siedersleben korrekt ansetzen) auf der Austauschbarkeit. Diese hat zwei Ebenen: Syntax und Semantik. Wenn eine Entität vom Typ *T* durch ein Objekt eines Subtyps *T'* belegt werden soll, so muss dieses alle Operationen von *T* auf konformer Art und Weise unterstützen. Auf *syntaktischer* Ebene könnte Konformität hier erlauben, dass bei gleicher Argumentanzahl Argumenttypen aufgeweitet und Ergebnistypen eingeschränkt werden, was aber in *Mainstream-Programmiersprachen* erstaunlicherweise nicht ausgenutzt wird. Eine zweite, *semantische* Ebene einzuziehen bedeutet, in irgendeiner Weise zu spezifizieren, was denn eigentlich der Effekt einer Methode sein soll, und auch auf dieser Ebene eine Konformität zu definieren. Hier steht Eiffel traurigerweise seit vielen Jahren allein auf weiter Flur. Eiffels Konzept des *Design-by-Contract* mit methodisch sauberer Unterstützung für Subtyping darf bei einer Diskussion über die Semantik der Vererbung nicht übergangen werden.⁷ Eine echte verhaltensmäßige Konformität [16, 24] zu erzwingen, erfordert natürlich weiteren Aufwand in Form von Spezifikationstechnik und Werkzeugeinsatz, der bislang nur unter bestimmten Bedingungen angemessen erscheint.

3.3.1 Steigerung der Wiederverwendung

Vererbung ist populär geworden, weil sie ein hohes Maß an Wiederverwendung verspricht. Im Gegensatz dazu preisen Broy und Siedersleben die „Wiederverwendung durch Unterprogramme, die uns seit den 50er Jahren vertraut ist“. Unterprogramme mögen Probleme der 1950er Jahre gelöst haben, aber das dritte Jahrtausend ist in dieser Hinsicht wesentlich anspruchsvoller. Längst reicht einfache Parametrisierung nicht mehr aus, um Programmfragmente für unterschiedliche Verwendungen anzupassen. Der Kern jeder ernst zu nehmenden Wiederverwendung ist eine Analyse von Gemeinsamkeiten (*commonalities*) und Variationen (*variabilities*) [8]. Dabei werden die Gemeinsamkeiten in einem *client interface* eingefroren, und für die Variationen muss ein geeigneter Adaptionsmechanismus gefunden werden. Für viele Arten der Anpassung

⁷ Ein wahrer „Schandfleck“, der Objektorientierung ist die Art, wie Java um „Zusicherungen“ erweitert wurde: das *assert*-Schlüsselwort wirft uns in diesem Punkt weit hinter 1988 [26] zurück.

muss das *adaptation interface* die Kapselung des *client interface* brechen⁸.

Die Alternative lautet also häufig: Wiederverwendung durch Vererbung (unter lokaler Verletzung der Kapselung) oder Codeduplikation statt Wiederverwendung. Die zweite Wahl ist häufig in diskutabel, da sie ein immenses Wartungsproblem nach sich zieht. Kapselung existiert natürlich beim Codekopieren gar nicht mehr. Wenn man andererseits die etwas intimere Bekanntschaft innerhalb einer Vererbungshierarchie zur Kenntnis nimmt, lässt sich auch hier das Schlimmste verhindern: Wieder hilft das Konzept des Design-by-Contract, das – konsequent angewandt – die Adaptionen einer Unterklasse in geregelten Bahnen halten kann.

Nur mit der Technik der Vererbung ist es gelungen, die Wiederverwendung über die Ebene von Funktions- und Klassenbibliotheken zu transzendieren: *Frameworks* stellen komplexe, partielle Programme dar, deren Instantiierung zu einer Applikation ohne Vererbung nicht denkbar wäre. Die wahre Flexibilität erhalten *Frameworks* dabei dadurch, dass eine applikationsspezifische Subklasse i. Allg. jede Methode ihrer Oberklasse überschreiben kann. Nur so sind *unvorhergesehene* Anpassungen möglich.

Nur muss deutlich gesagt werden, dass diese Flexibilität einen hohen Preis hat: Die Entwicklung und vor allem Dokumentation von *Frameworks* erfordert sehr viel Erfahrung und Disziplin.

Durch wahlfreies Überschreiben von Methoden entstehen hochkomplexe Steuerflüsse, die den Framework-Code und den applikationsspezifischen Code (die häufig von unterschiedlichen Entwicklern stammen) eng miteinander verzahnen.

Dadurch wird u. a. die Fehlersuche deutlich erschwert. Aus diesem Kontext kommt wohl die Beobachtung, den Steuerfluss einer objektorientierten Anwendung zu verfolgen, sei wie eine Landkarte durch einen Strohalm zu betrachten. Die Praxis zeigt jedoch, dass der Gewinn durch *Frameworks* deutlich höher ist als der Preis, der in Form von Komplexität bezahlt werden muss.

3.3.2 Evolution

Was mehrfach verwendet wird, muss sich auch neuen Anforderungen anpassen können, ohne zu alten Anwendungen inkompatibel zu werden. Fra-

meworks entstehen überhaupt erst durch Entwicklung in mehreren Iterationen. Objektorientierung ist umgeben von einer Vielzahl von Praktiken, die mehr oder weniger systematische Softwareevolution erlauben. Aus einer anderen Welt scheint dieses Zitat zu stammen: „Niemand ändert ein laufendes System nur um der Schönheit willen“ [5]. Um präzise zu sein: Die große Menge der Programmierer, die laufende Systeme strukturell verbessern, tun dies des „Geruchs“ willen: Die verbreitete, praxisnahe Bewegung des *Refactoring* basiert auf dem Konzept der „bad smells“, die auf strukturelle Schwächen in Programmen hinweisen, so dass durch gezielte strukturierte *Refactorings* die Qualität des Quelltextes verbessert werden kann, ohne dabei seine Funktionalität zu gefährden [10, 12].

3.4 Modularisierung

Zum Thema der Modularisierung sprechen Broy und Siedersleben zwei Probleme an: a) das Dilemma, Programme nur nach Daten *oder* Funktion strukturieren zu können, und b) die Suche nach Modulen, die größer sind als Klassen.

In der Tat wäre es unbefriedigend, wenn die Entwicklung von Blöcken über Prozeduren zu Modulen nun bei Klassen stehen bleiben sollte. Es ist allerdings nicht zeitgemäß, sich zu beklagen, die Objektorientierung würde keinen Komponentenbegriff liefern. Fakt ist,

- dass die Informatik sich allmählich mühsam auf einen Komponentenbegriff geeinigt hat (in dieser Richtung ist Szyperskis Buch [42] tatsächlich ein Referenzbuch), dessen Kern die Möglichkeit unabhängigen Deployments bildet;
- dass dieser Begriff über Fragen der Programmiersprachen weit hinausgeht;
- dass Komponenten i. d. R. objektorientiert implementiert werden;
- dass Komponenten in der Tat die Basis für Softwarearchitekturen bilden.⁹

So gesehen ist Komponententechnik etwas, das auf Objektorientierung aufbaut, aber deutlich darüber hinausgeht. Dieses Mehr liegt dabei zu erheblichen Anteilen im Bereich technischer Infrastrukturen, die bis dato von der Objektorientierung noch etwas vernachlässigt wurden. Es ist wichtig, die gemeinsame Basis, aber auch die Unterschiede zu sehen. So wie

⁸ Kiczales z. B. spricht von *client interface* und *meta interface* [22].

⁹ Uneinigkeit herrscht allerdings noch darüber, wie Konnektoren in Entwurf und Implementierung behandelt werden sollten [3, 18].

die Objektorientierung alle Konzepte von Unterprogrammen und strukturierter Programmierung in sich aufnahm, so sind Komponenten heute kaum ohne Objektorientierung vorstellbar.

Zwar scheint die Forderung (b) nun durch Komponenten erfüllt zu sein, es zeigt sich aber, dass erhebliche Probleme bestehen bleiben. Alle Beobachtungen gehen dahin, dass eine effektive Modularisierung großer komplexer Systeme erfordert, Programme als mehrdimensionale Strukturen aufzufassen. Das einfachste Beispiel sind die zwei Dimensionen Daten und Funktionen, die z. B. das Visitemuster nur notdürftig zu trennen versucht. In der Kritik dieses Musters stimmen wir Broy und Siedersleben im Wesentlichen zu. Viele Begriffe wurden in den letzten 10 Jahren geprägt, um diese Vielschichtigkeit von Software (vgl. [44]) auf grundsätzlichere Art in den Griff zu bekommen: Subjekte [15], Aspekte [21], Sichten (diverse Ansätze aus den Bereichen Datenbanken [38], verteilte Systeme [29], Programmiersprachen [18, 33]). Kollaborationen schließlich bezeichnen ein Konzept, das auf Entwurfsebene seit langem mit Erfolg eingesetzt wird [9, 36] und erst zögerlich auch beim Programmiersprachenentwurf berücksichtigt wird [17, 28, 46].

Alle diese Ansätze zeigen uns, dass neue Konzepte für Modularisierung und *Komposition* [27] erforderlich sind, um komplexe Systeme langfristig wartbar zu halten. Die verschiedenen Entwicklungen verbinden sich unter dem Dach der aspektorientierten Softwareentwicklung. Die erste internationale Konferenz zu diesem Thema [20] hat jüngst gezeigt, wie viel Innovationspotential in den Konzepten der Aspektorientierung steckt. Auch die ersten Erfolgsberichte von der industriellen Anwendung liegen bereits vor.

Hier ist gerade jetzt ein Zusammenfließen verschiedener Strömungen in Aussicht und wird uns in allernächster Zukunft erlauben, unsere Programme nach vielfältigen Kriterien viel radikaler zu modularisieren, als es die pure Objektorientierung erlaubt. Aber auch diese Ansätze bauen de facto auf der Objektorientierung auf. Warum? Weil die Objektorientierung das einzig sinnvolle Paradigma darstellt? Nein, eher weil die Objektorientierung einerseits vielen Informatikern sehr gut handhabbar erscheint und andererseits flexibel genug ist, um als Plattform für neue Strukturierungskonzepte zu dienen. An dem Übergang zur Aspektorientierung ste-

hen einige Konzepte der Objektorientierung erneut auf dem Prüfstand und es besteht die Gelegenheit, einige Unzulänglichkeiten auszumerzen. Die erneute Diskussion um objektbasierte Vererbung ist ein Beispiel dafür. Auch Kapselung wird hier völlig neu diskutiert, die explizite Unterscheidung von *client interface* und *adaptation interface* wird immer wichtiger. Diese Diskussionen sind nicht neu, führen aber unter neuen Rahmenbedingungen hoffentlich zu verbesserten Lösungen.

4 Fazit und Ausblick

Manche Kritikpunkte, die Broy und Siedersleben vorbringen, sind nicht weiter diskussionsfähig, da der Bezug zur angegebenen Zielsetzung „hochverteilte[r], interoperative[r] Software“ nicht hergestellt werden kann. Somit fehlen de facto Kriterien, Rahmenbedingungen oder Qualitätseigenschaften, an denen Technik und Methodik letztendlich gemessen werden sollen. Wieder beschleicht uns das Gefühl, es wird nach einer Universallösung für eine unbekannte Menge von Problemen gesucht. Ein solches Unterfangen wäre unseriös. Eingebettete Systeme erfordern andere Techniken als Telekommunikationssoftware, als ERP-Systeme, als Suchmaschinen und Onlinekataloge. Die Balance zwischen Flexibilität und Analysierbarkeit erfordert viele unterschiedliche Strategien in unterschiedlichen Projekten.

Um diese Vielfalt zuzulassen, ohne unüberwindliche Verständigungsprobleme unter Informatikern zu erzeugen, bedarf es einer gemeinsamen Basis. Diese Basis ist für eine große Mehrheit von Informatikern und Ingenieuren derzeit die Objektorientierung. Anzustreben sind darüber hinaus Programmiersprachen, die Lösungen für verschiedene der hier diskutierten Probleme anbieten. Von derart breit angelegten Sprachen werden dann in Projekten konkrete Profile angewendet, d. h., durch Konventionen und darauf abgestimmte Werkzeuge kann die Mächtigkeit der Sprache dann wieder eingeschränkt werden. In der Ada-Welt ist es gang und gäbe, durch sog. Profile bzw. Restriktionen [43] die Sprache für bestimmte Verwendungszwecke hin einzuschränken und dadurch analysierbare und letztendlich beherrschbare Programme zu erhalten. Die Einhaltung der Einschränkungen kann leicht durch Werkzeuge überprüft werden, da die Sprache Ada mittels einer standardisierten Schnittstelle den Zugriff auf den vollständigen abstrakten Syntaxbaum eines Programms unterstützt [1].

In der Diskussion über Wertsemantik klagen Broy und Siedersleben: „Auch hier lässt uns die Objektorientierung allein.“ Grund genug, die Hände in den Schoß zu legen und zu klagen, früher war alles viel besser? Nein, auch sie setzen zu guter Letzt auf Verbesserung. Gibt es ein zentrales Problem, das gelöst werden muss? Broy und Siedersleben sagen: asynchrone, nebenläufige Ausführung. Das ist jedoch nur ein kleiner Aspekt, in dem nach unserer Einschätzung ohnehin vergleichsweise wenig Handlungsbedarf bestand. Die angerissene Lösung mit ihrer Fixierung auf Zustandsautomaten wird wahrscheinlich nur einer kleinen Klasse von Applikationen wirklich dienen.

Wir haben versucht aufzuzeigen, wie vielfältig die Bestrebungen sind, einzelne Probleme heutiger Programmiersprachen und Methoden zu reparieren, und dabei sicherlich nur einen winzigen Ausschnitt berührt. Wichtig ist, dass wir nur dann eine Chance haben, verschiedene dieser Neuerungen gemeinsam einzusetzen, wenn diese auf einer gemeinsamen Basis aufsetzen. Und das ist für uns die Objektorientierung, mit all ihren Mehrdeutigkeiten und Varianten!

Um Studierende für künftige Entwicklungen zu wappnen, muss die Informatikausbildung darauf hinweisen, dass Objektorientierung ein Ding mit vielen Gesichtern ist und dass bestimmte Konflikte schwierige Abwägungen erfordern. Gerade die mächtigsten Techniken (wie z. B. das *Template&Hook*-Muster) erlauben eine sehr effektive Wiederverwendung (z. B. im Falle von Frameworks) und stellen gleichzeitig eine akute Gefahr für die Korrektheit des Programms dar. Nur Informatiker, die beide Seiten kennen, können sich ein eigenes Urteil bilden, in konkreten Projekten die geeignetsten Techniken auswählen und vielleicht an einer Verbesserung unserer Techniken und Methoden forschen. Aber vielleicht spricht dann schon keiner mehr von der Objektorientierung, so wie heute kaum noch vom strukturierten Programmieren die Rede ist – gewisse Dinge sind irgendwann einfach selbstverständlich.

Literatur

1. ISO/IEC 15291:1999 Information Technology – Programming Languages. Ada Semantic Interface Specification (ASIS)
2. CORBA Components, vol. I, joint revised submission. OMG Document orbos/99–07–01. Object Management Group 1999
3. Beugnard, A., Ogor, R.: Encapsulation of Protocols and Services in Medium Components to Build Distributed Applications. Proc. of Engineering Distributed Objects (EDO '99), ICSE 99 Workshop (1999)
4. Brichau, J., Meuter, W.D.: Qsoul Manual (Draft). <http://prog.vub.ac.be/poolresearch/qsoul/QSOULManual.pdf>
5. Broy, M., Siedersleben, J.: Objektorientierte Programmierung und Softwareentwicklung – eine kritische Einschätzung. Informatik Spektrum 25(1), 3–11 (2002)
6. Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism. Computing Surveys (1985)
7. Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., Jermaes P.: Object-Oriented Development: The Fusion Method. Prentice-Hall 1994
8. Coplien, J.O.: Multi-Paradigm Design. Ph.D. thesis, Vrije Universiteit Brussel, 2000
9. D'Souza, D., Wills, A.: Objects, Components, and Frameworks with UML – The Catalysis Approach. Addison-Wesley 1998
10. Dudziak, T., Wloka, J.: Tool-supported Discovery and Refactoring of Structural Weaknesses in Code. Diploma thesis, Technische Universität Berlin, 2002
11. Ernst, E.: gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance. PhD thesis, Department of Computer Science, University of Århus, 1999
12. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley 1995
14. Goldberg, A., Robson, D.: Smalltalk 80: The Language and its Implementation. Addison-Wesley 1983
15. Harrison, W., Ossher, H.: Subject-Oriented Programming: a Critique of Pure Objects. Proc. of OOPSLA'93, ACM, 1993, pp. 411–428
16. Helke, S., Santen, T.: Mechanized Analysis of Behavioral Conformance in the Eiffel Base Libraries. Lecture Notes in Computer Science, vol. 2021. Berlin Heidelberg New York: Springer 2001, pp. 20–42
17. Herrmann, S., Mezini, M.: Combining Composition Styles in the Evolvable Language LAC. Proc. of ASoC Workshop at the 23rd ICSE, 2001
18. Herrmann, S., Mezini, M.: Connectors for Bridging Mismatches between the Components of a Software Engineering Environment. IEE Proceedings – Software, 2001
19. Keene, S.E.: Object-Oriented Programming in Common LISP: A Programmer's Guide to CLOS. Reading/MA: Addison-Wesley 1989
20. Kiczales, G. (ed.): Proc. of First International Conference on Aspect Oriented Software Development, ACM Press, 2002. <http://trese.cs.utwente.nl/aosd2002/index.php>, .
21. Kiczales, G., Lamping, J., Mendhekar, A., et al.: Aspect Oriented Programming. Proc. of ECOOP '97. LNCS, vol. 1241. Berlin Heidelberg New York: Springer 1997, pp. 220–243
22. Kiczales, G.: Why Are Black Boxes So Hard to Reuse? Transcript from Presentation at OOPSLA'94, 1994
23. Kniesel, G.: Type-safe Delegation for Run-time Component Adaptation. Proc. of ECOOP'99. LNCS, vol. 1628. Berlin Heidelberg New York: Springer 1999, pp. 351–366
24. Liskov, B., Wing, J.: A Behavioral Notion of Subtyping. ACM Transactions on Programming Languages and Systems 16(6), 1811–1841 (1994)
25. Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: Object-Oriented Programming in the BETA Programming Language. Addison-Wesley 1993
26. Meyer, B.: Object Oriented Software Construction. New York: Prentice Hall Int. 1988
27. Mezini, M., Haupt, M.: Neue Paradigmen des Softwareengineering: Integrationsorientierte Programmierung. ObjektSpektrum, Feb. 2001
28. Mezini, M., Lieberherr, K.: Adaptive Plug-and-Play Components for Evolutionary Software Development. Proc. OOPSLA'98. ACM SIGPLAN Notices 33, 97–116 (1998)
29. Milli, H., Milli, A., Dargham, J., Cherkaoui, O., Godin, R.: View Programming: Towards a Framework for Decentralized Development and Execution of OO Programs. Proc. of TOOLS'99 (1999)
30. Monson-Haefel, R.: Enterprise Java Beans. O'Reilly 1999
31. Müller, P., Poetzsch-Heffter, A.: Universes: A Type System for Alias and Dependency Control. Technical Report 279, Fernuniversität Hagen, 2001
32. Odersky, M., Wadler, P.: Pizza into Java: Translating Theory into Practice. Proc. 24th ACM Symposium on Principles of Programming Languages, January 1997
33. Odersky, M.: Objects + Views = Components? In: Abstract State Machines 2000. LNCS. Berlin Heidelberg New York: Springer 2000
34. Ostermann, K., Mezini, M.: Object-Oriented Composition Untangled. Proc. of OOPSLA 2001. ACM Sigplan Notices 36, 283–299 (2001)
35. Ostermann, K.: Dynamically Composable Collaborations with Delegation Layers. Proc. of ECOOP 2002. LNCS. Berlin Heidelberg New York: Springer 2002

36. Reenskaug, T.: Working with Objects – The OORAM Software Engineering Method. Prentice Hall 1996
37. Rémy, D., Vouillon, J.: Objective ML: An Effective Object-Oriented Extension to ML. Theory and Practice of Objects Systems 4(1), 27–50 (1998)
38. Santos, C., Delobel, S., Abiteboul, S.: Virtual Schemas and Bases. Proc. of the International Conference on Extending Database Technology. LNCS, vol. 779. Berlin Heidelberg New York: Springer 1994
39. Stein, L.A., Lieberman, H., Ungar, D.: A Shared View of Sharing: The Treaty of Orlando. In: Kim, W., Lochovsky, F.H. (eds.): Object-Oriented Concepts, Databases and Applications. ACM Press/Addison-Wesley 1989, pp. 31–48
40. Szyperski, C., Omohundro, S., Murer, S.: Engineering a Programming Language: The Type and Class System of Sather. Proc. of Programming Languages and System Architectures. LNCS, vol. 782. Berlin Heidelberg New York: Springer 1994, pp. 208–227
41. Szyperski, C.: Import Is Not Inheritance – Why We Need Both: Modules and Classes. Proc. of ECOOP'92, LNCS, vol. 615. Berlin Heidelberg New York: Springer 1992, pp. 19–32
42. Szyperski, C.: Component Software – Beyond Object-Oriented Programming. Addison-Wesley 1998
43. Taft, T., Duff, R. (eds.): Ada 95 Reference Manual, Language and Standard Libraries, International Standard ISO/IEC 8652. LNCS, vol. 1246. Berlin Heidelberg New York: Springer 1997
44. Tarr, P., Ossher, H., Harrison, W., Sutton S. Jr.: *N* Degrees of Separation: Multi-Dimensional Separation of Concerns. Proc. of the 21st ICSE, 1999
45. Ungar, D., Smith, R.B.: Self: The Power of Simplicity. Proc. of OOPSLA'87, 1987
46. VanHilst, M., Notkin, D.: Using Role Components to Implement Collaboration-based Design. Proc. of OOPSLA'96. ACM SIGPLAN Notices 28(10), 1996
47. Wegner, P.: Dimensions of object-based language design. Proc. of OOPSLA'87, 1987