Sieben Thesen zur erfolgreichen Verwirrung von Anfängern der objektorientierten Programmierung

Axel Schmolitzky

In sieben Thesen werden Aussagen zur objektorientierten Programmierung dargestellt, die aus Sicht des Autors für Programmieranfänger problematisch sind. Alle Aussagen sind sinngemäß Lehrbüchern oder wissenschaftlichen Publikationen zum Thema „Einführung in die objektorientierte Programmierung" entnommen. Bewusst kontrovers und zugespitzt dargestellt, sollen die Thesen in erster Linie als Diskussionsgrundlage dienen.

Kenntnisse der *objektorientierten Programmierung* (OOP) sind für Informatikerinnen und Informatiker heutzutage unerlässlich. Entsprechend fließt die OOP immer stärker in die grundlegende Programmierausbildung an den Hochschulen ein, bis hin zu Ansätzen, die Objekte und Klassen vom ersten Tag der Programmierausbildung an thematisieren (*Objects First*). Dieses „Hineinwachsen" in die Ausbildung von Programmieranfängern geschieht auf höchst unterschiedliche Weise und möglicherweise an manchen Stellen eher ad hoc als gründlich reflektiert.

Eines der Hauptprobleme dabei ist die Vermischung von OOP und *objektorientierter Modellierung* (OOM). Während die OOP sehr mechanisch orientiert ist und auf programmiersprachlichen Konzepten aufsetzt, ist die OOM eine Sicht auf die Welt, die von der Beschreibung von Anwendungsdomänen bis hin zum Entwurf von Softwaresystemen reicht. Insbesondere beim Entwurf ist die Nähe zur OOP groß, doch es besteht ein wichtiger Unterschied: Beim Entwurf geht es um die Deklaration, wie ein Softwaresystem *sein soll*, während die OOP festlegt, wie Softwaresysteme *sind*. Beide Aspekte werden durch die *Unified Modeling Language* (UML) unterstützt; es ist somit offensichtlich, dass diese beiden Bereiche der Objektorientierung eng beieinander liegen. Eine Basisthese dieses Artikels ist jedoch, dass eine zu starke Vermischung von OOP und OOM Programmieranfängern den Einstieg erschwert.

Der Einstieg in die OOP wird heutzutage überwiegend mit Hilfe der Programmiersprache Java [4] gelehrt. Neben wirtschaftlichen Vorteilen (Java ist inzwischen eine relevante Sprache in der kommerziellen Informationsverarbeitung) spricht insbesondere das vergleichsweise einfache Objektmodell für diese Sprache in der Ausbildung: Java kennt nur einen Übergabemechanismus für Parameter (per Wert) und alle Objekttypen sind Referenztypen (ihre Exemplare können ausschließlich über Referenzen manipuliert werden); Zeigerarithmetik ist nicht möglich und das Einhalten von Objektgrenzen wird durch die Sprache garantiert. Dennoch ist Java weit davon entfernt, eine ideale Sprache für Programmieranfänger zu sein; sie ist, im übertragenen Sinne Churchills, eher die am wenigsten schlechte der zur Auswahl stehenden objektorientierten Programmiersprachen.

Alle der folgenden Thesen sind, wenn auch meist weniger zugespitzt, in Lehrbüchern oder wissenschaftlichen Publikationen zum Thema „Ein-

DOI 10.1007/s00287-006-0130-8

Dr. Axel Schmolitzky
Arbeitsbereich Softwaretechnik, Department Informatik,
Universität Hamburg,
Vogt-Kölln-Str. 30, 22527 Hamburg
E-Mail: schmolit@informatik.uni-hamburg.de

führung in die objektorientierte Programmierung mit Java“ zu finden. Sie sollen aus Sicht des Autors natürlich nicht dem im Titel genannten Zweck dienen, sondern vielmehr den Blick dafür schärfen, welche Fallstricke in der Lehre an Hochschulen vermieden werden sollten. Nach längerem Abwägen wurde für diesen Artikel entschieden, die jeweiligen Quellen nicht im Detail zu benennen, denn es geht nicht um das „Anschwärzen“ der Arbeit von Kolleginnen und Kollegen, sondern eher um das Aufzeigen von Mustern in der Lehre, die der Autor als problematisch ansieht.

„Objekte haben Namen“

In der objektorientierten Modellierung ist es überwiegend nützlich, ein Objekt und seinen Namen als eine Einheit zu betrachten. Entsprechend bietet auch die UML die Möglichkeit, die Objekte in einem Objektdiagramm mit einem Namen zu versehen. Bei genauerem Hinsehen stellt man jedoch fest, dass in einem Objektdiagramm der UML, das einen Schnappschuss eines Java-Objektgeflechts zur Laufzeit darstellen soll, das Benennen der Objekte problematisch ist; nur die Attribute, die die Objekte referenzieren, sollten Namen tragen.

Entsprechend ist diese These aus Sicht der OOP (fast immer) falsch. Objekte *haben* nicht per se einen Namen, wir geben ihnen höchstens Namen. Manchmal definieren wir für einen Objekttyp, dass seine Exemplare ein Attribut „Name“ haben; dann kann jedes Exemplar über eine geeignete Operation nach seinem Namen gefragt werden. Und immer benennen wir die *Variablen*, über die wir auf Objekte zugreifen. Die dynamisch erzeugten Objekte selbst sind jedoch anonyme Einheiten (deren synthetische Identität meist über eine Speicheradresse definiert wird und nicht mit dem hier verwendeten Verständnis eines Namens zu verwechseln ist). Diese Benennung vom Benannten zu trennen, ist eines der größten Probleme für Programmieranfänger; es sollte nicht mit nachlässiger Terminologie verschärft werden.

„Die Zustandsfelder von Objekten definieren ihre Attribute“

In der OOP wird der Zustand eines Objektes in seinen *Zustandsfeldern* (häufig nur kurz: Feldern) gehalten. Ein *Attribut* hingegen ist umgangssprachlich etwas, das eine nach außen sichtbare Eigenschaft beschreibt: Eine Person hat eine Haarfarbe, ein Fahrzeug hat eine Geschwindigkeit, ein Konto hat einen Kontostand. In der OOP ist es nicht zwingend, dass solche Attribute auch unmittelbar als Zustandsfelder implementiert werden müssen (statt eines Zustandsfeldes für den Saldo eines Kontos könnten auch alle Kontobewegungen eines Kontos protokolliert werden und der Saldo auf Nachfrage berechnet werden). Und umgekehrt gibt es viele Zustandsfelder, die gerade nicht nach „außen“ (also für Klienten eines Objektes) sichtbar werden sollen: In einer Implementierung des Datentyps *Liste* als verkettete Liste etwa ist die Referenz auf den Listenkopf üblicherweise nicht Teil der Schnittstelle. Diese Beispiele zeigen, dass die Unterscheidung zwischen den Zustandsfeldern der OOP und den Attributen der OOM nützlich sein kann. Die UML als eine Sprache, mit der sowohl bestehende Systeme dokumentiert als auch zukünftige Systeme spezifiziert werden können, erzwingt jedoch, beide Abstraktionen als Attribute zu beschreiben. Entsprechend wird der Begriff Attribut vor allem in Lehrbüchern aus dem Bereich der Softwaretechnik, die heutzutage auf der Terminologie der UML aufsetzen, vereinheitlichend verwendet; auf diese Weise werden Unterschiede verwischt, die in der Programmierausbildung relevant sind.

„Objekte enthalten Objekte“

In der objektorientierten Modellierung mit der UML werden mehrere Abhängigkeitsformen zwischen Klassen bzw. Objekten unterschieden, von denen die *Komposition* eine *enthält*-Beziehung modelliert. Es ist häufig sehr nützlich bei der Modellierung, klare Beziehungen zwischen einer Aggregation und ihren enthaltenen Teilen formulieren zu können, wie etwa zwischen einem Fahrzeug und seinem Motor. In Lehrbüchern zur OOP mit Java ist diese These jedoch problematisch, denn in Java sind Objekte mit anderen Objekten ausschließlich über Referenzen verknüpft; eine echte *enthält*-Beziehung kann also nicht programmiert werden. Dieses, verglichen mit beispielsweise C++, sehr einfache Objektmodell sollte anfangs konsequent vermittelt werden, denn Referenzen sind für Programmieranfänger schon für sich genommen schwierig genug. Erst wenn ausreichend Erfahrung im Umgang mit Objekten und insbesondere Referenzen vorhanden ist, können mächtigere Objektmodelle vermittelt werden, die auch eingebettete Objekte (siehe etwa die *expanded types* in Eiffel [10] oder die *value types* von C# [6]) zulassen.

„Mit Vererbung bilden wir Begriffshierarchien ab“

Diese Aussage findet sich in vielen Lehrbüchern, meist veranschaulicht durch eine Begriffshierarchie, in der beispielsweise Säugetiere über *ist-ein*-Beziehungen miteinander in Verbindung gebracht werden. Das ist für eine erste Vermittlung eines intuitiven Verständnisses eventuell hilfreich, es führt die meisten Programmieranfänger jedoch auf eine trügerische Fährte. Sie neigen dann schnell zu der Meinung, dass alle *ist-ein*-Beziehungen mit Vererbung modelliert werden sollten. Es gibt jedoch etliche solcher Beziehungen, für die die Vererbungskonzepte von Programmiersprachen eher ungeeignet sind (siehe dazu u.a. [3, 8]); beispielsweise, wenn ein Unterkonzept weniger Operationen anbietet als sein übergeordnetes Konzept (klassische Veranschaulichung: Ein Emu *ist ein* Vogel, kann aber nicht fliegen), wenn Operationen in einem Unterkonzept eingeschränkt werden (im mathematischen Sinne *ist* ein Quadrat *ein* spezielles Rechteck, es kann aber nicht in allen Kontexten als ein Rechteck angesehen werden, wenn bei einem Rechteck eine Seite verändert werden darf), oder wenn schlicht der Bezug zur Programmierung fehlt („die Demokratie *ist eine* Staatsform“).

Sehr viel wichtiger für die OOP sind die „*kann verwendet werden als*“-Beziehung (Subtyp-Polymorphie bzw. Ersetzbarkeit, engl. subtyping und substitutability) und die „*übernimmt Verhalten und Struktur*“-Beziehung (Code- oder Implementationsvererbung, engl. inheritance), siehe etwa [5]. Beides sind sehr technische Beziehungen, deren Mechanik gut verstanden sein will. Erst wenn dieses technische Verständnis vorhanden ist, kann die Modellierung von realen Anwendungsproblemen mit der notwendigen Schärfe vorgenommen werden.

„Interfaces sind spezielle abstrakte Klassen“

Die Problematik dieser These ist eigentlich konträr zu den vorherigen, denn sie ist nicht zu wenig, sondern zu stark technisch motiviert. Sie ist eine technisch und historisch korrekte Erläuterung von *Interfaces* (dem durch Java populär gewordenen Sprachkonstrukt), die Anfänger von der Verwendung des Konstruktes jedoch eher abschreckt. Mit einem Interface kann aber die bzw. eine Schnittstelle einer Klasse explizit beschrieben werden, eine Tätigkeit, die unabhängig von der ausführlichen Beschäftigung mit Vererbung und abstrakten Klassen nützlich ist. So kann der Umgang mit *Datenabstraktion* [9] (beispielsweise über ein Interface *List* mit Implementierungen *LinkedList* und *ArrayList*) schon im ersten Semester der Programmierausbildung geübt werden, während die Fallstricke rund um die Vererbung bequem erst im zweiten Semester thematisiert werden können [13].

„Wir beginnen mit Objekten ohne Methoden (reinen Datenobjekten), da dies einfacher ist“

Der datengetriebene Ansatz dieser These ist in einigen universitären Kontexten zu finden, in denen als einführende Programmiersprache eine funktionale gewählt wurde und in denen erst im zweiten Semester zu Java gewechselt wird. Eine zweite Linie mit großem Einfluss auf diese These ist in der klassischen imperativen Programmierung zu finden. So wird in der C- und C++-Welt das Klassenkonstrukt lediglich als eine alternative Form zu den, mit dem Schlüsselwort `struct` definierten Verbünden angesehen, ähnlich wie in der Linie von Pascal über Modula-2 und Oberon hin zu Oberon-2 Objekte lediglich als Erweiterungen von Record-Strukturen gesehen werden, auf Java angewendet etwa in [11] zu finden.

Dem datengetriebenen steht ein anderer Ansatz gegenüber, der Objekte primär über ihr Verhalten definiert, genauer über ihre *Zuständigkeiten* (*responsibility-driven approach*, erstmals beschrieben in [14]). Im Verhaltensansatz sind die Datenfelder der Objekte gekapselt, Veränderungen des Zustands erfolgen ausschließlich über Aufrufe von Operationen. Diese Sichtweise auf die objektorientierte Programmierung hat sich inzwischen weitgehend durchgesetzt und kann auch in der Programmierausbildung von Anfang an vermittelt werden.

„Wir benutzen Java zu Anfang rein funktional, um Zustände möglichst spät einzuführen“

Auch diese These ist in Kontexten entstanden, in denen zuerst funktional und erst im zweiten Semester objektorientiert programmiert wird, siehe etwa [12]. Durch eine anfangs rein funktionale Benutzung von Java soll der Wechsel der verwendeten Sprache erleichtert werden. Ein Effekt dieses Vorgehens ist, dass Java als eine Sprache empfunden wird,

in der elegante funktionale Konzepte kantig und umständlich werden.

Java ist jedoch explizit eine objektorientierte Programmiersprache und in der OOP stehen Zustände im Mittelpunkt; Objekte sind Einheiten eines laufenden Systems, die einen Zustand haben und u.a. auf Basis dieses Zustandes Dienstleistungen für Klienten anbieten. Bei der OOP geht es gerade darum, einen disziplinierten Umgang mit der Komplexität von zustandsbasierten Systemen zu erlernen; dem steht ein Vermeiden von Zuständen eher entgegen.

Fazit

Bei der intensiven Auseinandersetzung mit verschiedenen Lehransätzen und vor dem Hintergrund umfangreicher Lehrerfahrung mit Java sind dem Autor an verschiedenen Stellen (sowohl in Lehrbüchern als auch in wissenschaftlichen Publikationen) Aussagen begegnet, die er insbesondere für Programmieranfänger für problematisch hält. Die auffälligsten von ihnen sind in den vorgenannten (Anti-)Thesen destilliert. Während die ersten vier Thesen im Spannungsfeld zwischen OOP und OOM stehen und entweder falsch oder zumindest problematisch sind, drücken die letzten drei Thesen eher ein didaktisches Grundverständnis aus.

Die aufgeführten Thesen werden möglicherweise nur vom Autor als problematisch angesehen und können somit auch als Ausgangspunkte für kontroverse Diskussionen dienen. Der Autor selbst favorisiert einen konsequenten *Objects-First*-Ansatz, der durch die integrierte Entwicklungsumgebung BlueJ [7] ermöglicht wird und exemplarisch beschrieben ist in dem Lehrbuch „Objects First with Java – A Practical Introduction using BlueJ“ [2] (deutsche Übersetzung auch bei Pearson Education Deutschland [1]). Aber selbst dieses gute Lehrbuch und auch BlueJ sind nicht frei von Tendenzen zu einigen dieser Thesen.

Literatur

1. Barnes, D., Kölling, M.: Java lernen mit BlueJ – Eine Einführung in die objektorientierte Programmierung (3. Auflage). Pearson Education, Deutschland (2006)
2. Barnes, D., Kölling, M.: Objects First with Java – A Practical Introduction Using BlueJ (3rd Edition). Pearson Education, UK (2006)
3. Evered, M., Keedy, J.L., Schmolitzky, A., Menger, G.: “How Well Do Inheritance Mechanisms support Inheritance Concepts?”, Proc. Joint Modular Languages Conference (JMLC) ’97, Linz, Austria. In: Lecture Notes in Computer Science 1204, Springer-Verlag, S. 252–266 (1997)
4. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification (3rd Ed.). Reading, MA: Addison-Wesley (2005)
5. Halbert, D.C., O’Brien, P.D.: “Using Types and Inheritance in Object-Oriented Languages”, Proc. ECOOP ’87, Paris, France. In: Lecture Notes in Computer Science 276, Springer-Verlag, S. 20–31 (1987)
6. Hejlsberg, A., Wiltamuth, S., Golde, P.: The C# Programming Language. Addison-Wesley (2003)
7. Kölling, M., Quig, B., Patterson, A., Rosenberg, J.: “The BlueJ system and its pedagogy”. J. Comput. Sci. Edu. 13(4), 249–268 (2003) Special issue on Learning and Teaching Object Technology
8. LaLonde, W., Pugh, J.: “Subclassing $\neq$ subtyping $\neq$ Is-a”. J. Object-Orient. Program., January, 57–62 (1991)
9. Liskov, B.: “Data Abstraction and Hierarchy”, Proc. OOPSLA ’87 (Addendum), Orlando, Florida. In: ACM SIGPLAN Notices, Vol. 23, 5 (1988)
10. Meyer, B.: Eiffel: the Language. New York: Prentice-Hall (1992)
11. Mössenböck, H.-P.: Sprechen Sie Java? Eine Einführung in das systematische Programmieren (3. überarb. Auflage). Dpunkt Verlag (2005)
12. Proulx, V.K., Gray, K.E.: “How to Design Class Hierarchies”, Proc. Ninth Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts, ECOOP 2005, Glasgow, UK (2005)
13. Schmolitzky, A.: “Teaching Inheritance Concepts with Java”, Proc. Principles and Practices of Programming in Java (PPPJ), Mannheim, Germany; in ACM Press (2006)
14. Wirfs-Brock, R., Wilkerson, B.: “Object-oriented Design: A Responsibility-driven Approach”, Proc. OOPSLA ’89, New Orleans, Louisiana. In: ACM SIGPLAN Notices, 24, 12, S. 71–75 (1989)