Essay zur Objektorientierten Programmierung

Paradigmen, Dogmen und Konzepte

Fabian Steeg

30. Dezember 2007

Objektorientierte Programmierung (OOP) ist das Mainstream-Programmierparadigma der heutigen Zeit. Dies zeigt sich nicht nur durch die Tatsache, dass die beiden industriell wichtigsten Programmiersprachen Java und C++ diesem Paradigma folgen, sondern auch etwa darin, dass die verbreitetsten, sogenannten sprachunabhängigen Programmierhilfen, nämlich *Design Patterns* und die *Unified Modeling Language* (UML) dieses Paradigma voraussetzen. Nach einer Klärung der grundlegenden Begriffe beschreibe ich Eigenarten und Vorzüge der OOP, sowie im Anschluss Probleme dieses Paradigmas und geäusserte Kritik; anschließend gehe ich auf Konzepte alternativer Paradigmen (Logik- und funktionale Programmierung) vor dem Hintergrund der Frage ein, ob diese eine Lösung der Probleme der OOP darstellen können. Hier stellt sich die Frage, inwiefern es notwendig ist, in bestimmten Paradigmen zu bleiben oder ob sich diese nicht vielmehr in Konzepte zerlegen lassen, welche sich im Sinne pragmatischer Lösungen neu zusammensetzen lassen.

1 Programmierparadigmen

Der Begriff *Paradigma* beschreibt in diesem Kontext eine wissenschaftliche Schule, d.h. eine bestimmte Art, zu programmieren, oder genauer die Art, wie Probleme in einer Programmiersprache formuliert und gelöst werden. Ein Paradigma stellt damit eine Art Lehrmeinung dar. Der Begriff hat eine gewisse semantische Verwandschaft mit einem allgemeingültigen, unwiderlegbaren Lehrsatz, einem Dogma. Diese Intuition wird bestärkt durch Vertreter bestimmter Paradigmen oder Sprachen, die ihre jeweiligen Ansätze für die einzig richtigen halten.

Traditionell werden imperative und die deklarative Programmierung als Gegensatz beschrieben, wobei deklarative Programmierung weiter in funktionale und Logikprogrammierung unterteilt wird. Auf diese Weise wird funk-

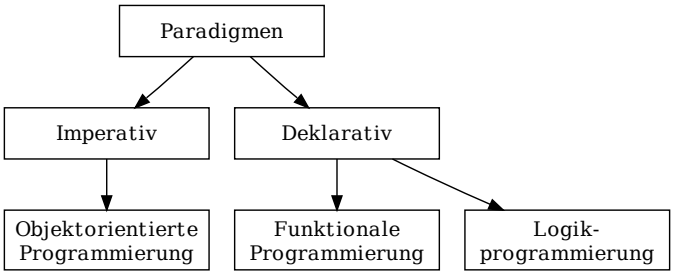

Abbildung 1: Ein konstruierter Gegensatz

tionale und imperative Programmierung als entgegengesetzte Positionen beschrieben, doch ist es schon in den grundlegendsten Formen der imperativen Programmierung möglich, einzelne Instruktionen (etwa Maschinencode zur Zuweisung von Werten in bestimmten CPU-Registern) zusammenzufassen zu Funktionen mit Parametern und Rückgabewert (etwa eine Operation zur Multiplikation zweier Werte). In diesem Sinn ist es etwa in imperativen Sprachen möglich, funktional zu programmieren, auch wenn eine Sprache es nicht erzwingt, wie auch umgekehrt. Dies deutet bereits an, dass Paradigmen nicht in sich abgeschlossen sind, sondern aus Konzepten bestehen, die in verschiedenen Paradigmen unterschiedliche Rollen spielen, welche daher nicht zu Dogmen werden sollten.

2 OOP

Grundlegendes Ziel der OOP ist die Vereinfachung der Softwareentwicklung durch die Verwendung abgeschlossener, wiederverwertbarer Softwarebausteine. Die erste objektorientierte Programmiersprache war Simula-67. In den 1980ern wurde OOP zunehmend verbreitet, insbesondere durch Smalltalk-80. In den 1990ern schließlich fand die OOP mit den Programmiersprachen C++ und Java Einzug in den Programmier-Mainstream. Im Folgenden werde ich auf die zwei zentralen Aspekte der OOP eingehen, nämlich Objekte und Vererbung.

2.1 Objekte

Kennzeichen der OOP ist die Verwendung von Objekten in Verbindung mit Vererbung. Programmieren mit Objekten, aber ohne Vererbung wird auch als objektbasierte Programmierung bezeichnet (Roy & Haridi 2004).

Grundkonzept von Objekten ist die Abbildung von Objekten des Problembereichs in der Welt auf Objekte der Lösung in der Software. In diesem Sinn wird die Welt in der Software modelliert. Den Bauplan der Objekte bilden Klassen, welche die Struktur und das Verhalten der Objekte beschreiben. Die Klassen bestehen aus Attributen (Eigenschaften) und Methoden (Aktionen).

Eine Sicht auf Objekte ist, dass in ihnen Daten und die darauf zulässigen Operationen gekapselt sind. Diese Sicht kann jedoch problematisch sein, etwa werden Klassen mit vielen Operationen (z.B. auf einem generischen Baum) auf diese Weise aufgebläht, zudem sind sie nicht erweiterbar, es kann etwa keine Operation ergänzt werden, wenn die Klasse nicht verändert werden kann oder soll. Eine andere Sicht auf Objekte, die weniger nachteilige Konsequenzen für das Design hat, ist die von Funktionen mit einem Gedächtnis in Form eines festen Zustandes.

Häufig wird die OOP im Kontext einer architekturgetriebenen Softwareentwicklung eingesetzt, die mehr oder weniger scharf die Phasen Analyse, Entwurf (Design), Programmierung und (manchmal auch) Testen trennt. Eine solche Einbettung ist aber nicht unbedingt nötig, oder gar wünschenswert. Die Herangehensweise einer architekturgetriebenen Entwicklung versucht, als erstes sauberen Code zu entwerfen, der dann verändert wird, um zu funktionieren, was häufig schwierig ist und zudem den Code oft wieder verschlechtert. Einen anderen Ansatz verfolgt etwa die testgetriebene Entwicklung (TDD), die das Funktionieren voranstellt und sauberen Code in einem zweiten, aber unmittelbar folgenden und notwendigen Schritt verfolgt (Beck 2003).

2.2 Vererbung

Grundlegende Überlegung zur Vererbung ist, dass die Abstraktionen, die in Objekten kodiert sind, einander oft sehr ähnlich sind. Zur Vermeidung von dupliziertem Code werden daher Klassen von anderen abgeleitet. Diese erben die Eigenschaften der Superklasse und lediglich der Unterschied muss spezifiziert werden. So kann Vererbung etwa auch für die Entwicklung generischer Frameworks eingesetzt werden, etwa in Java vor Version 5, wo die generischen Collection-Klassen Objekte vom Typ *Object* enthalten, einer Klasse, von der alle anderen Klassen erben.

Mit dem Konzept der Verwrbung sind trotz der beschriebenen Vorteile und Möglichkeiten drei grundsätzliche Probleme verbunden: erstens ist eine intime Kenntnis der Oberklasse nötig, zweitens eröffnet die Vererbung eine zusätzliche Schnittstelle zu der Klasse und drittens schließlich wird die Implementierung auf die Klasse und alle Superklassen ausgebreitet. Alle diese Punkte durchbrechen die Kapselung und laufen damit den Zielen der OOP zuwider. Die wichtigste Alternative zur Vererbung ist die Delegation, die im Gegensatz zur Sein-Relation der Vererbung eine Haben-Relation ausdrückt. Im wirkungsreichen Buch *Design Patterns* (Gamma *et al.* 1995) wird daher

empfohlen, mit Vererbung sparsam umzugehen und grundsätzlich Delegation zu bevorzugen, wenn dies sinnvoll möglich ist.

2.3 Kritik

Es lassen sich unterschieldiche Arten von Kritik an der OOP ausmachen. Ein wichtiger Punkt ist dabei die oben schon angedeutete Anwendung der Vererbung. Hier ist wichtig, dass die Sein-Relation, die die Vererbung ausdrückt auch tatsächlich eine solche ist. Ausgedrückt wird diese Forderung durch das Liskovsche Ersetzungsprinzip (*Liskov Substitution Principle*, LSP), das besagt, dass eine Subklassen-Instanz eine Instanz der Superklasse ersetzen kann, ohne dass Verhalten des Programms zu verändern.

Ein Beispiel für eine Vererbung, die dieses Prinzip verletzt, wäre etwa die Ableitung eines Kontos mit Gebühren von einem Konto (ohne Gebühren). Das Konto mit Gebühren ist nicht wirklich ein Konto (ohne Gebühren) und würde daher bei Ersetzung der Oberklasse das Verhalten des Programms verändern und damit den Vertrag der Oberklasse brechen. Eine Modellierung, die dem LSP entspricht, wäre etwa, beide speziellen Konten (nämlich mit und ohne Gebühren) von einem abstrakten Konto abzuleiten.

Weitere Kritikpunkte an der OOP sind etwa eine inkonsistente Umsetzung in bestimmten Sprachen (z.B. in Java durch die Duplizierung der elementaren Datentypen), Schwierigkeiten beim Zusammenspiel mit anderen Paradigmen (etwa OO-Modellierung im Zusammenspiel mit einer relationalen DB, der sog. *object-relational impedance mismatch*) sowie die Erhöhung der Programmkomplexität durch Objekte. Dass Objekte ihre Semantik auf Raum (in Form von einander erbenden Klassen) und Zeit (das Verhalten eines Objektes hängt von den im Lauf der Zeit empfangenen Nachrichten ab) ausweiten, erschwert das Verständnis, Beweisen und Debuggen von Programmen.

3 Alternative Paradigmen

Einige der beschriebenen Probleme der OOP treten im Kontext anderer, weniger mächtiger Paradigmen nicht auf. Im Folgenden möchte ich der Frage nachgehen, inwiefern diese möglicherweise Alternativen zur OOP darstellen.

3.1 Deklarative Programmierung allgemein

Deklarative Programmierung, etwa in Form von funktionaler oder von Logikprogrammierung ist allgemein dadurch gekennzeichnet, dass aufgrund des fehlenden festen Zustandes alle Operationen unabhängig und deterministisch sind. Dies hat grundsätzlich zwei Vorteile: Elemente lassen sich einfach zusammensetzen und Programme sind einfach zu verstehen, zu beweisen und zu debuggen.

3.2 Logikprogrammierung

Logikprogrammierung ist eine Form von deklarativer Programmierung, die vor allem durch einen eingebauten Suchmechanismus und eine prädikatenlogische Syntax gekennzeichnet ist. Ein System zur Logikprogrammierung besteht aus den 3 wesentlichen Elementen der *Axiome*, einer *Anfrage* (*query*) und einen *Beweismechanismus*, der die Anfrage auf Basis der Axiome und mithilfe von Deduktion beweist oder widerlegt.

Ein grundsätzliches Problem eines solchen Ansatzes ist, dass jede Verzweigung bei der Suche nach einer Lösung den Suchraum mit der Anzahl der Optionen an der Verzweigung multipliziert. Aufgrund dieser Einschränkung eignet sich Logikprogrammierung vor allem für Anwendungen mit kleinem Suchraum und als Erkundungssystem für begrenzte Beispiele.

Konkrete Anwendungen liegen etwa im Bereich von Datenbanken, so entspricht die die Anfrage der Suchanfrage einer Datenbank, der Suchmechanismus entspricht dem RDMS (*relational database management system*) einer traditionellen DB. Vorteile der Logikprogrammierung sind hier sehr flexible Anfragen in Form von prädikatenlogischen Ausdrücken sowie die Fähigkeit zur Deduktion, wodurch nicht-explizites Wissen geschlossen werden kann. Ein weiterer Einsatzbereich von Logikprogrammierung ist die Konstruktion von Parsern, wofür etwa Prolog in Form der DCG eine problemnahe Syntax bereitstellt.

Insgesamt kann aber festgehalten werden, dass Logikprogrammierung aufgrund der beschriebenen Eigenschaften zu speziell ist, um eine Lösung der beschriebenen Probleme der OOP darzustellen.

3.3 Funktionale Programmierung

Eine rein funktionale Sprache erlaubt als Operationen lediglich mathematische Funktionen, d.h. es gibt keinen expliziten Zustand. Vertreter der reinen funktionalen Programmierung argumentieren, dass dies die beste Basis für zuverlässige, erweiterbare Software darstellt: die Berechnung sollte stets ohne Seiteneffekte erfolgen (d.h. die Parameter sollten nicht verändert werden) und Software sollte schrittweise und iterativ mit einem Interpreter entwickelt werden, mit dem die einzelnen, unabhängigen Teile getestet werden (Graham 1993).

Die starke Einschränkung, die einige Vorteile mit sich bringt, führt aber dazu, dass im Vergleich zur OOP auf einige Dinge verzichtet werden muss, insbesondere festen Zustand, Vererbung und eine gestufte Kapselung (z.B. public, protected, private).

4 Pragmatische Lösungen

Die beschriebenen alternativen Paradigmen sind zum Teil nicht allgemein einsetzbar (Logikprogrammierung) oder bringen neben Vorteilen auch Nachteile mit sich (funktionale Programmierung). Zudem ist der totale Umstieg auf andere Sprachen mit völlig anderen Paradigmen und Konventionen nicht praktikabel, etwa für bestehende Projekte oder aufgrund des vorhandenen Entwickler-Know-Hows. Im folgenden möchte ich daher einige pragmatische Lösungsansätze beschreiben.

4.1 Best-Practices und Hilfsmittel in OOP-Sprachen

Die Komplexität, welche die OOP zugleich so vorteilhaft macht und die zugleich wohl ihr größtes Problem ist, kann durch unterschiedliche Best-Practices und Hilfsmittel gebändigt werden, etwa die Bevorzugung von Delegation oder die Verwendung bestimmter Design Patterns. Zur Vermeidung fehlerhafter Ableitungshierarchien unter Berücksichtigung des LSP kann etwa *Design-by-Contract* führen. Den schrittweisen Aufbau von zuverlässigen Softwarebausteinen, den Anhänger der reinen funktionalen Programmierung ihrem Paradigma und einem Interpreter zuschreiben, lässt sich auch durch die TDD erreichen, bei der die Tests einen interaktiven, inkrementellen Aufbau einer verlässlichen Codebasis erlauben.

4.2 Kompatible Spracherweiterungen und neue Sprachen

Ein anderer Ansatz ist die Weiterentwicklung von bestehenden oder die Entwicklung neuer, mit bisherigen Sprachen kompatiblen Sprachen, die bestimmte Aspekte verbessern. Als Beispiel hierfür kann *Scala* dienen, eine mit Java und .Net kompatible Sprache, die zum Ziel hat, funktionale und objektorientierte Programmierung zu verbinden und etwa für Komponenten spezielle Unterstützung liefert (Odersky & al. 2004).

Kennzeichen von Scala ist einerseits ein konsequent objektorientierter Ansatz, bei dem etwa alle Methoden Objekte sind und auch Infix-Operatoren selbst definiert werden können, so entspricht in Scala der Ausdruck 3+5 dem Aufruf 3.+(5), d.h. + ist eine Methode des Objekts 3. Auch umgeht Scala einige Probleme der OOP, so verhindert etwa die Kennzeichnungspflicht von Methoden, die Methoden einer Superklasse überschreiben sowohl ein versehentliches Überschreiben, als auch ein versehentliches Umwandeln einer überschreibenden Methode zu einer überladenden (durch Änderungen in der Superklasse). Zugleich erlaubt aber Scala sehr funktionales Programmieren mit einer aufgeräumteren Syntax als etwa in Java (etwa durch Methoden als Variablen) und fördert so die Entwicklung besserer, einfacherer Programme.

Die Entwicklung einer neuen Sprache bietet darüberhinaus Spielraum für die Integration verschiedener Konzepte, so unterstützt etwa Scala XML di-

rekt im Code, hat spezielle Mechanismen für Komponentenentwicklung und auch Eigenschaften von Logiksprachen, etwa Unterstützung für Pattern Matching und Unifikation sowie Hornklauseln.

Die komplette Integration mit Java (es können Klassen aufgerufen und ererbt, Interfaces implementiert werden, etc.) macht Scala so zu einer sehr pragmatischen Lösung verschiedener Probleme der OOP und schlägt eine Brücke zwieschen den sonst häufig als unvereinbar betrachteten Paradigmen der auf der imperativen Programmierung aufbauenden OOP und der deklarativen Programmierung, wodurch deutlich wird, dass es sich bei einer solchen Unterscheidung nicht um diskrete Kategorien handelt, sondern vielmehr um ein Kontinuum, vom Programmieren mit Zustand zum ganz zustandsfreien Programmieren.

Literatur

Beck, K.: 2003, *Test-Driven Development by Example*, The Addison-Wesley Signature Series, Addison-Wesley.

Gamma, E., R. Helm, R. Johnson & J. Vlissides: 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.

Graham, P.: 1993, *On LISP: Advanced Techniques for Common LISP*, Prentice Hall.

Odersky, M. & al.: 2004, 'An Overview of the Scala Programming Language', Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland.

Roy, P. V. & S. Haridi: 2004, *Concepts, Techniques, and Models of Computer Programming*, The MIT Press, Cambridge, Mass.