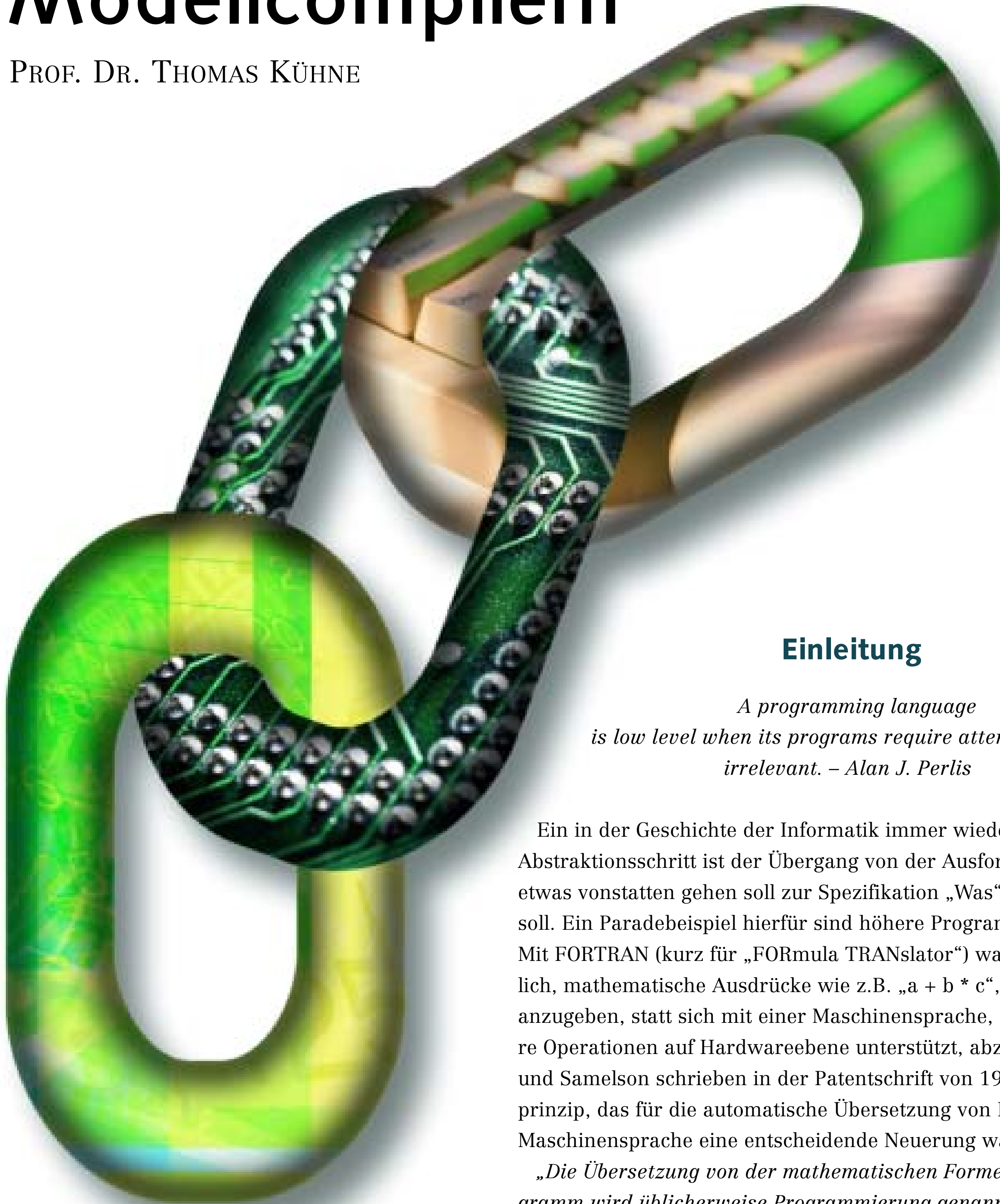
Automatisierte Softwareentwicklung mit Modellcompilern

PROF. DR. THOMAS KÜHNE



Einleitung

A programming language is low level when its programs require attention to the irrelevant. – Alan J. Perlis

Ein in der Geschichte der Informatik immer wieder vorgenommener Abstraktionsschritt ist der Übergang von der Ausformulierung „Wie" etwas vonstatten gehen soll zur Spezifikation „Was" erreicht werden soll. Ein Paradebeispiel hierfür sind höhere Programmiersprachen: Mit FORTRAN (kurz für „FORmula TRANslator") war es endlich möglich, mathematische Ausdrücke wie z.B. „a + b * c", einfach als solche anzugeben, statt sich mit einer Maschinensprache, die nur elementare Operationen auf Hardwareebene unterstützt, abzumühen. Bauer und Samelson schrieben in der Patentschrift von 1957 zum Kellerprinzip, das für die automatische Übersetzung von Formeln in Maschinensprache eine entscheidende Neuerung war:

„Die Übersetzung von der mathematischen Formelsprache ins Programm wird üblicherweise Programmierung genannt; sie hat sich in praxi als eine zeitraubende und fehleranfällige, im allgemeinen nur lästige Angelegenheit herausgestellt. Für den Mathematiker stellt die Programmiersprache eine ungewohnte Formulierung dar, die überdies noch von Anlagentyp zu Anlagentyp wechselt." Auslegeschrift 1094019, 30.3.1957

Man beachte, dass im obigen Text mit „Programmiersprache" die Maschinensprache einer bestimmten Hardware gemeint ist. Mittler-

Automated Software Development with Modelcompilers

The first FORTRAN compiler represents a milestone in computer science, since for the first time ever it was possible to specify the „what“ instead of the „how“ of a solution. Programmers using a high level programming language became independent of the particularities of specific hardware and – at the same time – were enabled to think and construct with much more powerful concepts. The vision of a model compiler implies a similar boost in programmer productivity; the advancement from programming with data and pointers to modeling with components and connectors. Important foundations for turning this vision into reality, such as domain specific modeling, aspect-oriented architectures, and model transformation techniques are currently the focus of intense research and development activities.

weile, durch die zur Selbstverständlichkeit gewordene Benutzung von Übersetzern, bedeutet „Programmierung“ die in Anspruchnahme sehr viel ausdrucksstärkerer Mechanismen (wie z.B., Polymorphismus oder impliziter Methodenaufruf).

Allerdings stellen die heutigen programmiersprachlichen Ausdrucksmittel auch das niedrigste Abstraktionsniveau dar, auf das man freiwillig zurückgreift. Bevor es zur Realisierung von Software mit Hilfe von Programmiersprachen kommt, werden sehr viel prägnantere Spezifikationsformen eingesetzt. Steht der Lösungsentwurf erst einmal in detaillierter Form zur Verfügung, dann ist die programmiersprachliche Umsetzung oft nur eine monotone Überführung von Entwurfsentscheidungen in Codierungsmuster.

Die oben zitierte Patentschrift wird plötzlich wieder hochaktuell, wenn man einfach einige wenige Termini ersetzt (Ersetzungen sind hervorgehoben):

„Die Übersetzung von der *diagrammatischen Modellierungssprache* ins Programm wird üblicherweise *Entwurf/Realisierung* genannt; sie hat sich in praxi als eine zeitraubende und fehleranfällige, im allgemeinen nur lästige Angelegenheit herausgestellt. Für den *Modellierer* stellt die *Implementierungssprache* eine ungewohnte Formulierung dar, die überdies noch von *Plattform* zu *Plattform* wechselt.“

In dieser „modernisierten“ Fassung sind zwei Aspekte berücksichtigt:

1. Der Übergang vom Entwurf (Modellierung) zur Realisierung (Programmierung).
2. Die Anpassung des allgemeinen Entwurfs an spezifische Realisierungsmöglichkeiten.

Die Realisierungsmöglichkeiten werden durch den Ausführungskontext (Plattform) bestimmt. Je nach gewählter technologischer Basis („middleware“) ergeben sich andere Grundoperationen. So unterstützen viele Standardplattformen heutzutage bereits Persistenz oder verteilte Komponentenausführung. Je nach Ausrichtung und Anbieter unterscheiden sich diese Ausführungsumgebungen jedoch im Detail und verlangen Realisierungsentscheidungen, deren Konsequenzen bis in den Entwurf zurückreichen.

Wenn es gelänge, einen Übersetzer aus der Taufe zu heben, dessen Quellen aus eindeutigen Entwurfsspezifikationen bestünden und dessen Ausgabe die gewünschten Realisierungen – angepasst an die ausgewählte Zielplattform – wären, dann wäre ein weiterer Meilenstein in der Informatik erreicht: Der Modellcompiler.

Modellcompiler: Die Vorteile

Könnte man Entwickler von den Einzelheiten der Implementierung befreien, sie also von Programmierern zu Modellierern machen, hätte das eine Reihe von Vorteilen:

- Analog zum Vergleich „höhere Programmiersprache“/„Maschinensprache“ kann auf viel höherem Abstraktionsniveau entwickelt werden. Hieraus ergibt sich unmittelbar eine höhere Produktivität.
- Viele Details der Realisierung, die heutzutage die „Bäume“ darstellen wegen derer man den „Wald“ nicht mehr erkennen kann, tauchen im Entwurf gar nicht auf. Daraus ergibt sich eine verbesserte Kommunikationsgrundlage zwischen Entwicklern und eine einfachere Wartbarkeit.
- Fehlerhafte oder suboptimale Umsetzungen des Entwurfs in Implementierungen werden vermieden. Zwar wird es immer spezielle Herausforderungen geben, die sich einer vollständigen Automatisierung widersetzen – sowie auch heute noch Maschinensprache manchmal unumgänglich ist –, auf der anderen Seite sind aber mechanisierte, optimierte Übersetzungen den manuell erzielten Ergebnissen von Nicht-Experten oft überlegen.
- Ein einmal entwickelter Entwurf kann für verschiedene Plattformen (wieder-) benutzt werden. Angesichts der heutigen kurzen Innovationszyklen

im Bereich der Plattformtechnologien ist dies ein nicht zu unterschätzender Vorteil.

Ein Modellcompiler ist also eine wünschenswerte Vision, aber ist diese auch realisierbar?

Eine utopische Vision?

Die OMG (Object Management Group) liefert mit ihrer MDA- (Model Driven Architecture) Initiative [1] im Prinzip die Zielvorgabe für den Modellcompiler; die Ablösung des Codes als primären Softwareartefakt durch die entsprechenden Entwurfsmodelle. Letztere sollen Mittelpunkt der Softwareentwicklung werden und der Code zum abgeleiteten Produkt „degenerieren". Um die große Lücke zwischen einem modellierten Entwurf und einer implementierten Realisierung schließen zu können, sieht der MDA-Ansatz zwei verschiedene Modelltypen vor: Ein plattformunabhängiges Modell (PIM), das – nach Auswahl einer Plattform – anschließend zu einem plattformabhängigen Modell (PSM) überführt wird, welches wiederum anschließend implementiert wird. Der MDA-Ansatz teilt also einige angestrebte Vorteile mit dem Modellcompiler, hat jedoch weder einen vollständigen Automatisierungsanspruch, noch enthält er konkrete Hinweise, wie die Umsetzung auszusehen habe.

Kritische Stimmen könnten sogar behaupten, dass „MDA" nur eine Marketingblase ohne technischen Inhalt sei, deren Hauptzweck darin bestehe, den nicht sehr erfolgreichen CORBA-Standard als Legitimation der OMG durch die Unified Modeling Language (UML) abzulösen. Tatsächlich ist die UML ein sehr erfolgreicher OMG-Standard, der auch im Rahmen der MDA-Initiative als Mittel der Wahl genannt wird. Es sei dahingestellt wieviel „Nährwert" die MDA-Initiative anfangs hatte; es steht fest, dass sie bereits jetzt ein gemeinsames Ziel für Industrie und Forschung darstellt. Allerdings muss man für ein Gelingen der MDA-Initiative bzw. für die Machbarkeit eines Modellcompilers noch einige dringende Verbesserungen einfordern.

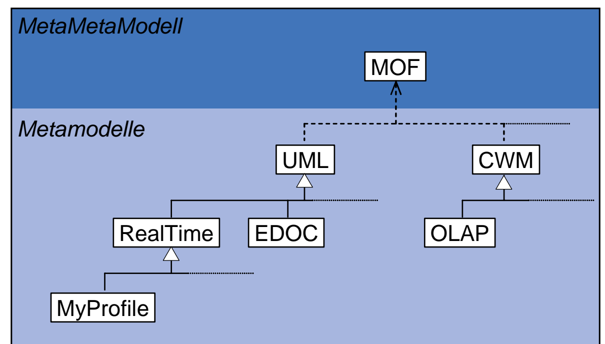
Anforderungen an die Modellierungssprache

Eine Notation für Entwurfsmodelle – grafisch oder nicht – , die als Grundlage für abgeleiteten Code dienen soll, muss natürlich eine eindeutige Semantik haben. Über die Bedeutung eines Diagramms, das einen bestimmten Aspekt des gesamten Modells darstellt, darf es keine Diskussionen geben. Gerade die im MDA Zusammenhang in das Rampenlicht gedrängte UML lässt hier aber einiges zu wünschen übrig. Bevor sie also für den gedachten Zweck tatsächlich geeignet ist, muss sie zunächst von einem Notationsstandard zu einem Modellierungsstandard reifen, indem sie eine formale Semantik erhält. Dass diese eine Semantik möglicherweise nicht allen UML-Benutzern gerecht wird, kann durch UML-Varianten begegnet werden (siehe Abb. 1).

Je nach Anwendungsbereich wird eine geeignete Variante/Verfeinerung der UML zum Einsatz kommen können. Abb. 1 macht nebenbei auch deutlich, dass die UML nur einer von vielen Modellierungsstandards sein soll; alle jedoch durch das gleich Meta-Meta-Modell, die Meta-Object-Facility (MOF), beschrieben werden können. Damit ist es möglich, modellierte Daten und Modelle über Modellierungssprachgrenzen hinaus auszutauschen. Die UML-Definition wird übrigens als Metamodell bezeichnet, da sie ein Modell aller benutzererstellten Modelle – also ein Meta-Modell – ist.

Neben der unbedingt notwendigen Präzisierung der UML gibt es aber noch eine weitere Voraussetzung für die Machbarkeit der MDA-Idee: Modellierer werden nur dann in der Lage sein, ihre Softwaresystem hinreichend genau und ökonomisch zu modellieren, wenn ihnen die Möglichkeit gegeben wird, die UML an ihre spezifischen Anforderungen anzupassen. Die UML-Variantenbildung darf also nicht nur auf standardisierte Verfeinerungen beschränkt sein, sondern der Modellierer selbst muss in der Lage sein, das Modellierungsvokabular zu erweitern. Der bisherige Weg, dies zu ermöglichen, bestand darin, die UML als Sprache zu erweitern. Jedoch haben sich insbesondere die dafür vorgesehenen „Stereotypen" als

Abb. 1: UML Varianten
UML variants



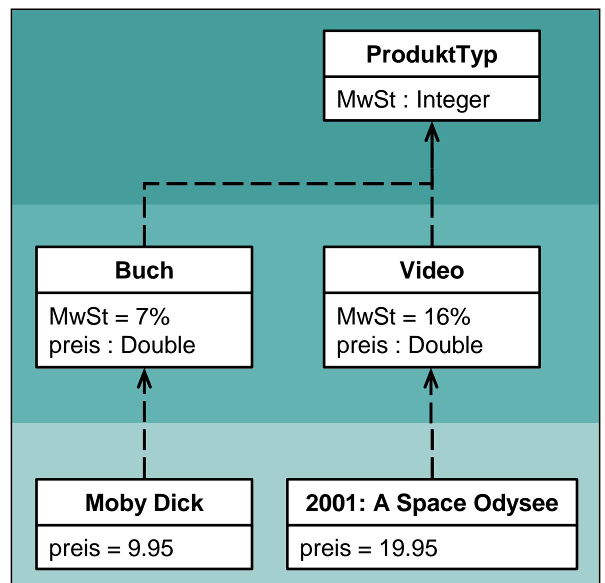


Abb. 2: Modellieren mit Metaklassen
Modeling with metaclasses

stark umstrittenes und wenig mächtiges Mittel herausgestellt [2].

Sehr viel ausdruckstärker und dabei konzeptionell einfacher ist der Ansatz die Beziehung Instanz/Klasse mehrfach zu wiederholen, also, z.B. Klassen von Klassen zu modellieren (siehe Abb. 2)

Die Klassen Buch und Video selbst werden als Instanzen der Meta-Klasse Produkttyp modelliert. Damit können nicht nur Spielregeln für erlaubte Beziehungen unterhalb von Klassen formuliert werden –, z.B. „es dürfen nur Produkttypen mit Warenkorbarten verknüpft werden" – sondern auch Klasseneigenschaften (z.B. die einem bestimmten Produkttyp zugeordnete Mehrwertsteuer) natürlich beschrieben werden [3].

Selbst wenn jetzt schon eine, wie oben beschriebene, Modellierungssprache zur Verfügung stünde, würde immer noch ein wichtiges Teil im „Modellcompiler"-Puzzle fehlen: Es zeigt sich bereits heute, dass eine bloße Unterteilung laut MDA-Ansatz in PIM, PSM und Realisierung unzureichend ist.

Stratifizierung

Es liegt in der Natur einiger Systemaspekte wie z.B. „Verteilung", dass sie sich einer transparenten Behandlung widersetzen. Es ist zwar mit einem PIM möglich einen Systementwurf unabhängig von am Markt existierenden technischen Lösungsmöglichkeiten zu formulieren, es wird aber nicht gelingen das System als „nicht verteiltes System" zu beschreiben und die Verteilung dann per Umformung in das PSM zu erreichen. Der Verteilung inhärente Eigenschaften wie „unzuverlässige Kommunikation", „weite Streuung von Laufzeiten für Prozeduraufrufe" usw. verlangen nach expliziter Behandlung und können deshalb nicht einfach automatisch hinzugeneriert werden. Ein von der Verteilung unabhängiger Systementwurf wäre nicht nur plattformunabhängig (PIM), sondern sogar aspektunabhängig (CIM = Concern Independent Model).

Ein solches CIM wäre natürlich nützlich, da eine von der Verteilung abstrahierende Systembeschreibung erstens leichter verständlich ist und zweitens wiederverwendet werden kann, wenn sich die Konzeption des Verteilungsaspekts einmal ändern sollte. Aus diesem Grund sollten auf dem Weg von einer möglichst lösungsunabhängigen Beschreibung bis zum ausführbaren Code mindestens drei Abstraktionsstufen liegen.

In Abb. 3 sieht man links drei Versionen einer Systembeschreibung, deren Abstraktionsgrad nach unten hin abnimmt. Die unterste, detaillierteste Ebene kann dann mechanisch in ausführbaren Code transformiert werden (rechts in Abb. 3). Die Abstraktionsebenen stehen also untereinander in einer Verfeinerungsrelation [4][5], sind aber nicht wie üblich als temporale Abfolgen zu interpretieren, sondern als Gesamtheit zu sehen. Die gesamte Abstraktionshierarchie auf der linken Seite von Abb. 3 stellt die Systemarchitektur dar, so dass jederzeit jede gewünschte Abstraktionssicht auf das System verfügbar ist.

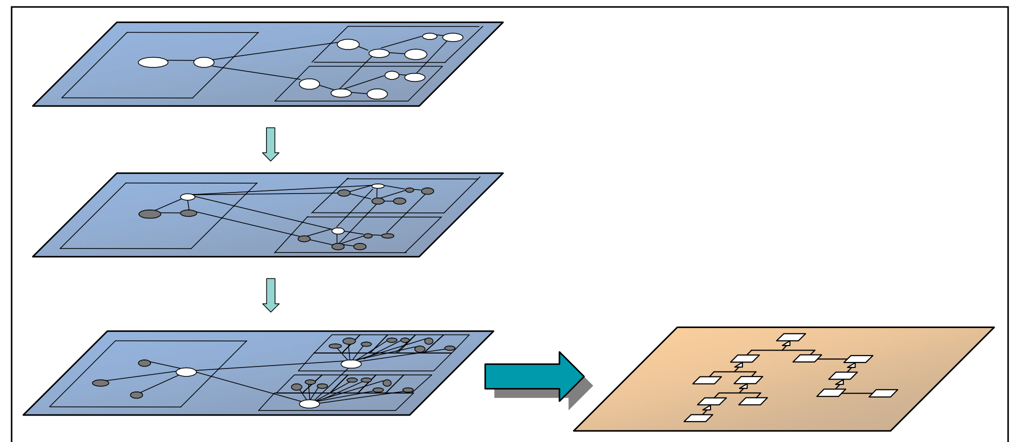


Abb. 3: Architekturstratifikation

Stratified architecture

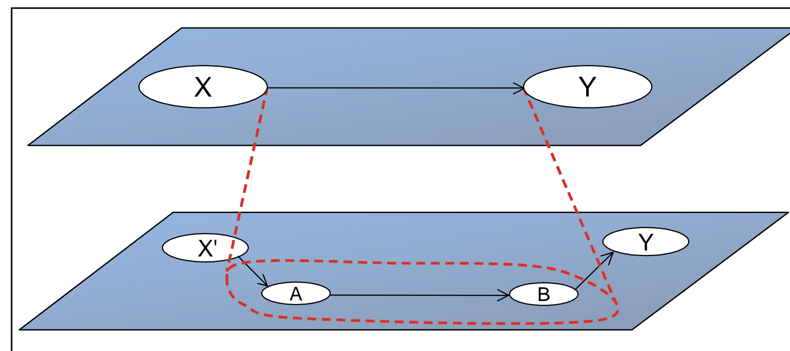


Abb. 4: Beziehungsverfeinerung

Relationship refinement

Die einzelnen Beschreibungen stehen hierbei in einer festen Relation zueinander (siehe Abb.4).

Eine Beziehung zwischen zwei Systemkomponenten X und Y wird hier auf der nächsten niedrigeren Ebene repräsentiert durch

- neu eingeführte Komponenten A und B,
- neu eingeführte Beziehungen,
- und eine Adaption von Komponente X an die neuen Gegebenheiten.

Beispielsweise wird vom Übergang von Abb. 5a nach Abb. 5b deutlich, dass dem „international"-Aspekt der „Bank"/„Account" Kommunikation durch die Einführung

Abb. 5a: Eine einfache Architektur

A simple architecture

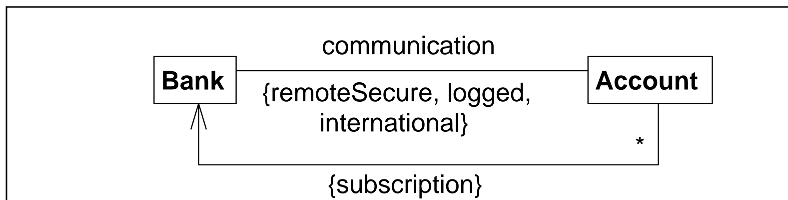


Abb. 5b: Wenige Details: Hinzufügen der Währung

Low detail: Adding Currency

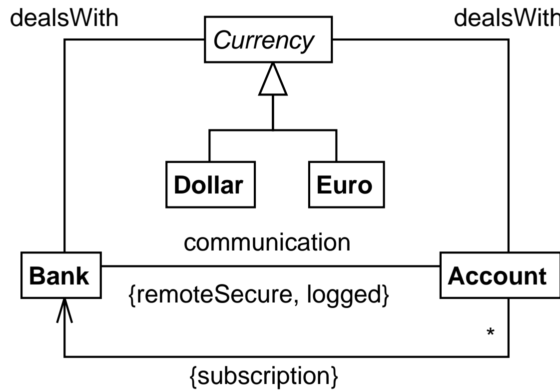


Abb. 5c: Mehr Details: Hinzufügen der Protokollierung

Medium detail: Adding Logging

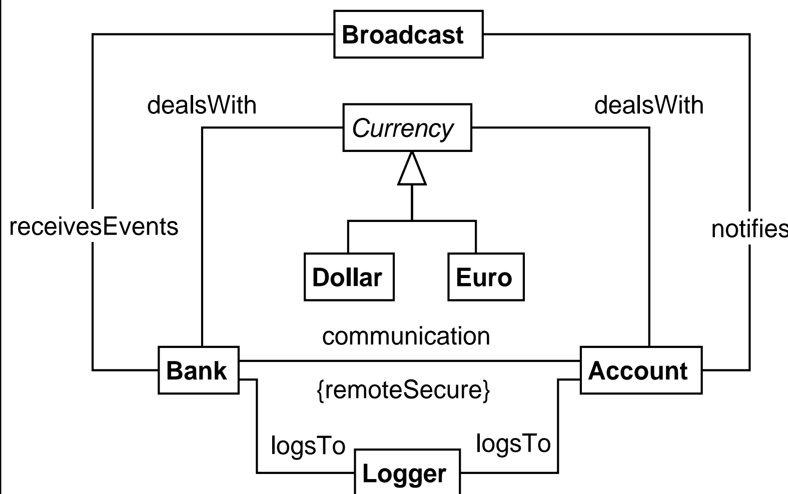
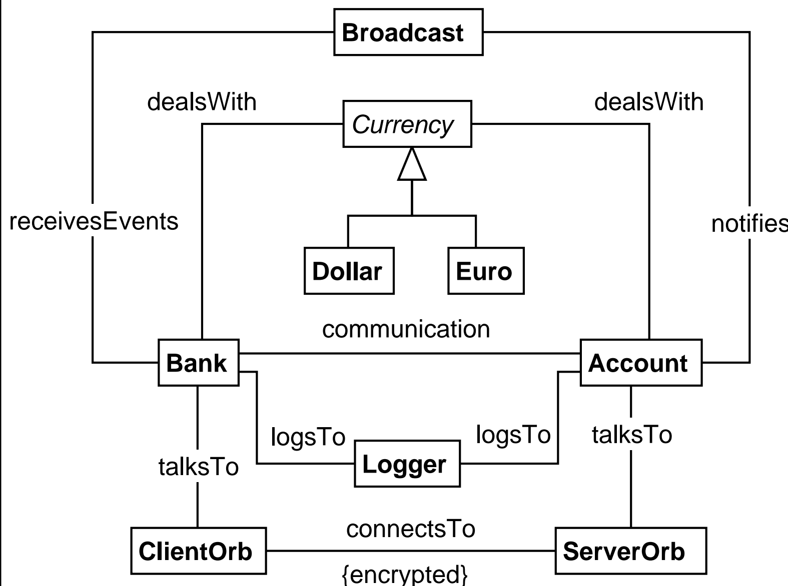


Abb. 5d: Viele Details: Hinzufügen der verteilten Kommunikation

High detail: Adding remote Communication



entsprechender Währungskomponenten Rechnung getragen wird. Wie man sieht, wurde also eine sehr komplexe Beziehung zwischen zwei Komponenten durch eine weniger komplexe ersetzt und das gewünschte Verhalten durch die Einführung neuer Komponenten erzielt. Solche Übergänge, wie in Abb. 4 schematisch gezeigt, werden durch Relationen (z.B., bidirektionale Graphentransformationen) beschrieben, so dass die Übergänge automatisiert herbeigeführt werden können. Entweder

- „vorwärts“ durch Auflösung von Beziehungsannotierungen wie „international“ oder
- „rückwärts“ durch Analyse der nächst niedrigeren Stufe nach entsprechenden Mustervorkommen.

Es wird deutlich, dass dieses Vorgehen zu mehr als drei Abstraktionsstufen führt, sobald Beziehungen genügender Komplexität eingeführt werden. Tatsächlich ist dies gewollt, denn so kann – Schritt für Schritt – pro Stufe ein neuer Realisierungsaspekt eingeführt werden.

Aspektorientierung

Werden beim Übergang von einer Verfeinerungsstufe zur nächsten genau diejenigen Komponentenbeziehungen ausgewählt, die einem bestimmten Systemaspekt zugeordnet werden können, erhält man eine saubere Trennung dieser Aspekte und erreicht, dass sie aufeinander aufbauen können. So wird zum Beispiel beim Übergang von Abb. 5b nach Abb. 5c Systemprotokollierung eingeführt und beim Übergang von Abb. 5c nach Abb. 5d Verteilung; hier die Kommunikation über sogenannte „Object Request Broker“. Die Verschlüsselung, von der in dieser Stufe noch

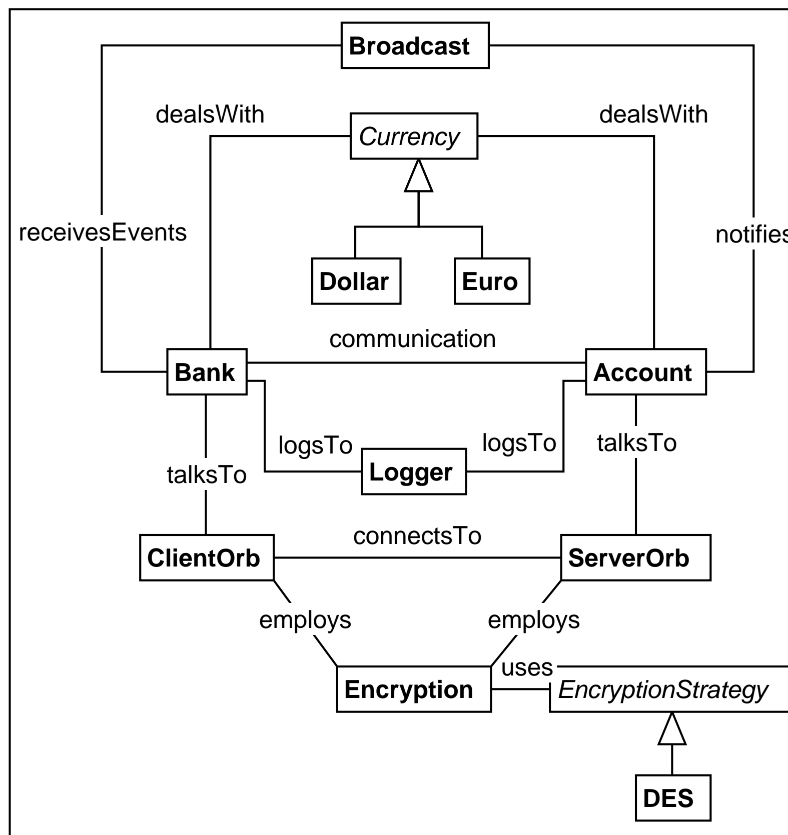


Abb. 6: Vollste Details: Hinzufügen der Verschlüsselung

Most details: Adding Encryption

abstrahiert wird, ist schließlich erst in Abb. 6 konkretisiert.

Vor allem der Kontrast zwischen Abb. 5a und Abb. 6 macht deutlich, wie hilfreich die verschiedenen Abstraktionsstufen sind: Für einen allgemeinen Überblick sind hohe Abstraktionsniveaus ideal, für die Klärung von Detailfragen sind jeweils entsprechend angemessene Sichten verfügbar. Die in unteren Niveaus gegebenenfalls angetroffene Komplexität ist keinesfalls ein Nachteil, sondern unumgänglich. Die Klärung von Fragen z.B. die Serialisierung von Daten oder deren Verschlüsselung betreffend, lassen sich erst klären wenn der entsprechende Kontext (z.B. Kommunikation) vorher geklärt wurde und für die Entscheidungsfindung zur Verfügung steht.

Tabelle 1: Kategorisierung von Aspekten

Categorizing Aspects

Aspekt-Kategorie	***Aspekt***	***Unter-Aspekt***
Infrastruktur	Koordination	Lastverteilung
		Synchronisation
	Verteilung	Fehlertoleranz
		Kommunikation
		Transaktionen
	Persistenz	
Services	Sicherheit	Verschlüsselung
		Autorisierung
	Wiederherstellung	
	Rückgängig machen	
Paradigmen	Funktion	
	Ereignisse	

Bisherige Ansätze zur Aspektorientierung, also der Trennung von Systemaspekten vom eigentlichen Systementwurf [6], haben Probleme, wenn es darum geht, mehrere Aspekte mit dem Basisentwurf zusammenzuweben. Das liegt zum einen daran, dass das Weben auf Codeebene vollzogen wird, also keine Ebenen mit Komponenten oder Beziehungen von höherem Abstraktionsgrad zur Verfügung stehen, an die man die Aspekte verankern könnte. Des Weiteren ist für einen abhängigen Aspekt (z.B. Verschlüsselung) der Kontext nie explizit, da dieser erst durch das automatische Verweben des Basissystems mit den vorausgehenden Aspekten (z.B. verteilte Kommunikation) entsteht. Die Folge ist, dass es an Stellen, an denen sich mehrere Aspekte überlagern, zu Konflikten zwischen diesen kommt. Je nach Überlagerungsstelle muss oft jeweils ein anderer Aspekt dominierend behandelt werden, d.h. eine allgemeingültige Strategie zur Verwebung konkurrierender Aspekte ist nicht formulierbar. Bei der Architekturstratifikation [7] dagegen – wie oben vorgestellt – liegt der entsprechende Kontext für eine Aspektanwendung immer explizit auf der nächst höheren Abstraktionsstufe vor. Somit werden die Aspekte nicht nur voneinander getrennt, sondern es wird auch deren hierarchischer Natur Rechnung getragen.

Tabelle 1 macht deutlich, dass gewisse abhängige Unteraspekte überhaupt erst in Erscheinung treten, wenn deren auslösende Aspekte bereits eingeführt wurden.

Fazit

Die heutigen Generatoren für Benutzungsschnittstellencode zeigen welch hohe Automation bereits (in diesem eingeschränkten Bereich) möglich ist. Das Beschreiben des „Was", also das Zusammenstellen der Bedienelemente, genügt bereits, um das „Wie", den ereignisgesteuerten Code, zu erzeugen. Der nächste logische Schritt besteht darin, diesen Automatisierungsgrad für alle Systemaspekte zu erzielen.

Wie beschrieben sind allerdings noch einige wichtige Schritte auf diesem langen Weg zu gehen. Während die Beschreibung der reinen Systemstruktur heute schon gut möglich wäre, besteht immer noch eine große Herausforderung darin, Systemverhalten probat zu beschreiben, d.h. hier nicht auf Programmiersprachenniveau herabzugleiten.

Auch die Idee der Architekturstratifizierung mit ihren verket-

teten Abstraktionsstufen ist zwar vielversprechend, muss aber noch im Einzelnen ausgearbeitet werden. Eine möglichst intuitive und allgemeine Form der Beschreibung der Verfeinerungsrelationen ist ebenso von Nöten wie eine optimale Unterstützung durch Werkzeuge.

Auch wenn der Modellcompiler noch nicht in greifbarer Nähe ist, so scheint dessen Realisierung früher oder später unvermeidlich. Es gilt zwar noch einige Hürden zu überwinden, aber ist – bezogen auf die Softwareentwicklung – ein lohnenderes Ziel als den nächsten geschichtlichen Quantensprung zu vollziehen überhaupt denkbar?

Literatur

[1] OMG: Executive overview: Model driven architecture. http://www.omg.org/mda, 2001

[2] Colin Atkinson, Thomas Kühne, Brian Henderson-Sellers: Stereotypical Encounters of the Third Kind. Proceedings of the 5th International Conference on the UML, Sept. 30 - Oct. 4, Dresden, 2002

[3] Colin Atkinson and Thomas Kühne: Rearchitecting the UML Infrastructure. To appear in the the ACM journal „Transactions on Modeling and Computer Simulation", 2003

[4] Nicholas Wirth: „Program Development by Stepwise Refinement." Communications of the ACM, vol. 14, no. 4, 1971, pp. 221 - 227.

[5] M. Broy: Towards a Mathematical Concept of a Component and Its Use, TUM Report I9746, 1997

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier and J. Irwin: Aspect-Oriented Programming, In proceedings of the European Conference on Object-Oriented Programming, Finland. Springer-Verlag LNCS 1241, 1997

[7] Colin Atkinson and Thomas Kühne: Aspect-Oriented Development with Stratified Frameworks. To appear in IEEE Software, 2003

Informationen zum Fachgebiet Metamodellierung und deren Anwendungen im Fachbereich Informatik der TU Darmstadt

Das Themengebiet Metamodellierung hat aktuelle Anwendungen in der Informatik (UML, XML, E-Commerce, etc.), seine Wurzeln reichen jedoch zurück bis in die griechische Antike. Grundsätzliche Modellierungsfragen wurden schon von Aristoteles und Plato behandelt und später von Mathematikern wie Frege präzisiert, was letztendlich zur Stufenbildung (Metaisierung) durch Russell und nachfolgend zur Typisierung führte.

Die Abwesenheit einer Systematik der Metaebenen (Definitionshierarchie) kann zu paradoxen Situationen führen und sollte daher möglichst vermieden werden. Aktuelle Forschungsthemen des Fachgebiets sind die Anwendung von bekannten aber auch innovativen Metamodellierungstechniken auf die Definition und Benutzung der UML sowie die Beschreibung und Entwicklung von Systemarchitekturen auf der Grundlage einer Hierarchie von Verfeinerungsebenen. Am Ende all dieser Ansätze steht die Vision einer möglichst automatisierten, modellgetriebenen Softwareentwicklung.

Kontakt

Dr. Thomas Kühne, Juniorprofessor
Tel. 0 61 51 / 16-61 88
kuehne@informatik.tu-darmstadt.de
TU Darmstadt, Fachbereich Informatik
Fachgebiet Metamodellierung
und deren Anwendungen
Wilhelminenstr. 76, 4283 Darmstadt
Fax 0 61 51 / 16-54 72
http://www.mm.informatik.tu-darmstadt.de

Wir sind ein international tätiges Ingenieurbüro mit weltweit mehr als 700 Mitarbeitern in Niederlassungen in verschiedenen Ländern (Deutschland, Österreich, Schweiz, Tschechische Republik, Slowakische Republik, Italien, Rußland, USA, Nigeria, Malaysia, Bulgarien, Türkei u.a.) und befassen uns überwiegend mit der Planung und Abwicklung von Projekten in den Bereichen:

Pipeline Transportsysteme **Anlagen für die Öl- u. Gasindustrie** **Verkehrsanlagen** **Kommunikations- u. Automatisierungstechnik**

Aus folgenden Fachrichtungen werden Ingenieure bei uns laufend gesucht:

ELEKTROTECHNIK
INFORMATIONS-/NACHRICHTENTECHNIK
ALLGEMEINER MASCHINENBAU
VERFAHRENSTECHNIK
WIRTSCHAFTSINGENIEURWESEN

Gute Englischkenntnisse in Wort und Schrift und idealerweise eine zweite Fremdsprache erleichtern den Einstieg bei unseren internationalen Projekten. Da wir als Dienstleister am Markt auftreten, sollten Sie gleichzeitig auch Interesse am Vertrieb von Ingenieurleistungen haben.

Die Positionen erfordern Mobilität und Flexibilität, Verantwortungsbewußtein, Selbständigkeit, freundlichen Umgang mit Kunden, Lieferanten und Mitarbeitern sowie Zuverlässigkeit und Ehrlichkeit.

Wir bieten Ihnen ein interessantes und abwechslungsreiches Aufgabengebiet, ein angenehmes Betriebsklima, einen sicheren Arbeitsplatz und eine der Tätigkeit angemessene Vergütung. Sollten wir Ihr Interesse an einer Mitarbeit in unserem Unternehmen in München geweckt haben, dann schicken Sie uns bitte Ihre ausführlichen Bewerbungsunterlagen an folgende Adresse:

I L F Beratende Ingenieure GmbH
z. Hd. Frau P. Lischka
Arabellastraße 21
D-81925 München
Tel.: +49 / 89 / 92 80 08-17
Fax: +49 / 89 / 92 80 08-30
Patrizia.Lischka@muc.ilf.com

ILF
BERATENDE INGENIEURE
CONSULTING ENGINEERS
INGENIEURS CONSEILS

www.ilf.com