

Komponentenorientierter Entwurf von PIMs und CIMs mit der KobrA-Methode

Colin Atkinson, Matthias Gutheil, Oliver Hummel

Lehrstuhl für Softwaretechnik
Universität Mannheim
L15, 16
68131 Mannheim
{atkinson, gutheil, hummel}@informatik.uni-mannheim.de

Abstract: Unternehmenssoftware wächst beständig in Größe und Komplexität, weshalb es immer wichtiger wird, Softwareentwicklungs-Paradigmen anzuwenden, die ein großes Problem abstrahieren und in kleinere Teile zerlegen können (divide and conquer). Die Modellgetriebene und komponentenbasierte Entwicklung ermöglichen genau dies, sie werden bisher allerdings, auf Grund ihrer getrennten Entwicklungsgeschichte nur sehr begrenzt gemeinsam genutzt. Das Problem liegt im Wesentlichen darin, dass das komponentenbasierte Paradigma nur auf der untersten Ebene der modellgetriebenen Entwicklung, d.h. für Platform Specific Models Verwendung findet. Höhere, abstraktere Ebenen werden derzeit für ein solches Vorgehen nicht genutzt. Wir beschreiben in diesem Papier, wie die KobrA-Methode diese Herausforderung angeht, indem sie die Konzepte „Komponente“ und „System“ vereinheitlicht und für die komplette Palette der UML-Diagramme zugänglich macht. Somit ermöglicht KobrA die komponentenorientierte Modellierung sowohl von Platform Independent Models als auch von Computation Independent Models.

1 Einführung

Seit jeher ist die Software-Industrie bestrebt, bessere Unternehmenssoftware-Systeme anzubieten und diese nahtloser miteinander zu integrieren. Im Wesentlichen wurden bisher zwei grundlegende Strategien entwickelt, um dem damit einher gehenden Anstieg der Komplexität Herr zu werden. Eine Möglichkeit – die sogenannte komponentenorientierte Software-Entwicklung [Szy98] – besteht darin, das Problem in kleinere, möglichst standardisierte Bausteine zu zerlegen, deren Größe und Komplexität beherrschbar bleiben. Die andere, bekannt als Model Driven Architecture (MDA), versucht das Problem so weit zu abstrahieren, dass essentielle Anwendungslogik und Geschäftsregeln unabhängig von plattformspezifischen Implementierungsdetails entwickelt werden können. Diese beiden Ansätze haben zwar ein hohes Potential sich

gegenseitig zu ergänzen, sie wurden allerdings unabhängig voneinander entwickelt und daher werden momentan die Vorteile beider nur sehr begrenzt gemeinsam genutzt.

Das grundlegende Problem besteht darin, dass diese beiden Paradigmen nicht vollständig orthogonal zueinander sind. Im Model Driven Development werden – zumindest in der Vision der Model Driven Architecture der Object Management Group (OMG) – drei elementare Abstraktionsebenen definiert, mit denen Unternehmenssoftware repräsentiert werden kann. Von „unten nach oben“ sind dies zunächst das Platform Specific Model (PSM), das das System in seiner kompletten Größe mit allen Middleware- und Plattform-Details verwoben mit der Geschäftslogik zeigt. Darauf folgen das Platform Independent Model (PIM), das von plattformspezifischen Middleware-Details abstrahiert sowie das Computation Independent Model (CIM), welches das Problem, das durch ein System gelöst werden soll, ohne Bezug zur IT beschreibt. Obwohl alle genannten Abstraktionsebenen von der Komponenten-Orientierung profitieren könnten, wird komponentenorientierte Entwicklung heutzutage meist nur auf der untersten Stufe, also auf Ebene einer bestimmten Plattform verwendet. Alle vier derzeit gängigen Komponenten-Technologien (.NET, J2EE, CORBA, Web Services) werden als Plattform im Sinne der MDA betrachtet und bis heute sind beinahe alle Möglichkeiten, Komponenten in der Unified Modelling Language (UML) – der Modellierungs-Sprache der MDA – zu beschreiben, praktisch ausschließlich auf die unterste Ebene ausgerichtet.

Mit anderen Worten, die ursprünglichen Versionen dieser Modellierungssprache bieten nur die Möglichkeit der Erstellung von komponentenbasierten PSMs, PIMs oder CIMs dagegen, bleiben weitgehend außen vor. Um logische Komponenten (d.h. solche, die auf Grund ihrer logischen Struktur und nicht nur bloßes Zusammenpacken gekoppelt sind) zu modellieren, benötigt man eine spezialisierte komponentenbasierte Entwicklungsmethode wie Catalysis [SOU98], Kobra [At02] oder UML Components [Che00]. Die UML 2.0 [OMG03] verbessert diese Situation zwar leidlich, indem sie bessere Abstraktionen zur Modellierung von logischen Komponenten anbietet. Wie Kobra erlaubt UML 2.0 nun auch, die Beschreibung einer Komponente in zwei getrennte Teile zu splitten, in eine Spezifikation und eine Realisierung. Es ist weiterhin möglich, interne Classifier und Strukturen einer Komponente zu zeigen, indem Teile von Klassendiagrammen oder ähnliche Strukturen in die Komponente aufgenommen werden. Allerdings besteht auch UML 2.0 noch darauf, die Konzepte, die zur Modellierung von Komponenten-Internas und Verbund-Strukturen verwendet werden können, auf eine Teilmenge derer zu beschränken, die zur Modellierung kompletter Systeme verwendet werden. Sie gibt keinerlei Hinweise darauf, wie UML-Diagramme oder Teile davon zu den beschränkten strukturellen Informationen hinzugefügt werden sollen, um z.B. die nach außen sichtbaren Zustände einer Komponente oder ihren internen Nachrichtenfluss zu beschreiben.

Da logische Komponenten in moderner Unternehmenssoftware extrem groß und komplex werden können, ist es sehr schwierig, diese umfassend mit den derzeit in der UML vorhandenen Möglichkeiten darzustellen. Um das Komponenten-Paradigma auf der PIM-Ebene voll einsetzen zu können, ist es notwendig, die willkürliche Trennung zwischen einem *System*, dem alle Diagrammtypen der UML zur Verfügung stehen und einer *Komponente*, die nur sehr eingeschränkt modelliert werden kann, aufzuheben. In

unseren Augen gibt es keine Rechtfertigung für eine Unterscheidung zwischen diesen beiden Konzepten, da des einen System durchaus eines anderen Komponente sein kann und umgekehrt.

Durch die Beseitigung dieser Unterscheidung wird es möglich, sowohl PIMs als auch CIMs in einer komponentenorientierten Art und Weise zu organisieren und dadurch für die komponentenbasierte und die modellgetriebene Entwicklung vollständige Orthogonalität zu erreichen. In diesem Papier beschreiben wir die Vorzüge dieses Ansatzes, den wir mit der sogenannten Kobra-Methode umgesetzt haben. Diese ist von Grund auf darauf abgestimmt, Systeme und Komponenten gleich zu behandeln, was als das Prinzip der Gleichförmigkeit (principle of uniformity) bezeichnet wird. Im nächsten Abschnitt beschreiben wir zunächst, wie Systeme/Komponenten in Kobra mit Hilfe zusammenhängender UML-Diagramme modelliert werden, bevor wir darauf eingehen, wie diese hierarchisch zur zusammengesetzten Struktur einer komplexen Unternehmenssoftware organisiert werden. In Abschnitt drei diskutieren wir den Prozess, mit dessen Hilfe komponentenorientierte PIMs und CIMs entwickelt werden. Der darauf folgende Abschnitt zeigt, wie sich prinzipiell PSMs daraus ableiten lassen. Diesen theoretischen Erläuterungen stellen wir im Teil fünf eine einfache Fallstudie zur Seite, die grundlegende Ideen noch einmal illustriert, bevor wir im sechsten und letzten Abschnitt die Vorteile unseres Ansatzes noch einmal abschließend zusammenfassen.

2 Modellierung von logischen Komponenten mit Hilfe der UML

Die Art und Weise, in der in Kobra Komponenten beschrieben werden, basiert, wie Abbildung 1 illustriert, auf dem Ansatz, den Fusion [Col93] zur System-Beschreibung nutzt. Dieser wiederum basiert auf der ursprünglichen OMT (Object Modelling Technique, [Rum91]), die die objektorientierte Software-Entwicklung in den frühen 90er Jahren entscheidend geprägt hat.

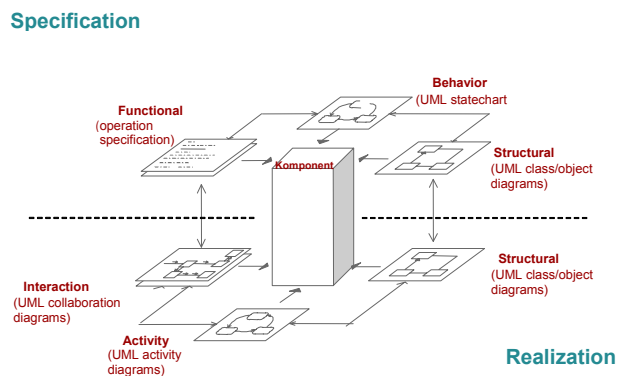


Abbildung 1: Kobra's UML-basierte Darstellung von Komponenten

Wie Abbildung 1 zeigt, besteht die Beschreibung einer Komponente in Kobra aus zwei Teilen: der Spezifikation und der Realisierung. Diese sind entfernt mit den Analyse- und

Design-Sichten von „traditionellen“, nicht-komponentenbasierten Systemen verwandt. Das bedeutet, die Spezifikation beschreibt die nach außen sichtbaren Eigenschaften einer Komponente mit Hilfe der folgenden drei Modelle: Eines oder mehrere statische Struktur-Diagramme beschreiben die *strukturelle Sicht*, eine Menge von Operations-Spezifikationen zeigt die *funktionale Sicht* und ein Zustands-Diagramm spiegelt das *Verhalten* der Komponente wieder. Diese drei Perspektiven entsprechen genau den drei Dimensionen der objektorientierten Analyse, die schon die OMT propagiert und Fusion später aufgegriffen hat.

Die Realisierung andererseits beschreibt, wie eine Komponente ihre durch die Spezifikation festgelegten Eigenschaften in Interaktion mit anderen Komponenten, die Clients oder interne Sub-Komponenten sein können, umsetzt. Dies wird wiederum durch die Verwendung von drei Teilmodellen ermöglicht: Statische Struktur-Diagramme zeigen die *strukturelle Sicht* auf der Design-Ebene, eine Anzahl von Interaktions-Diagrammen (Kommunikations- oder Sequenz-Diagramme) beschreiben die *interaktionsorientierte Sicht* und eine Menge von Aktivitätsdiagrammen verdeutlicht die *algorithmische Sicht*. Diese Sichten basieren im Wesentlichen auf Fusions Konzept der objektorientierten Design-Artefakte, dem noch Activity-Diagramme hinzugefügt und dessen Klassen-Beschreibungen durch Klassen-Diagramme ersetzt wurden.

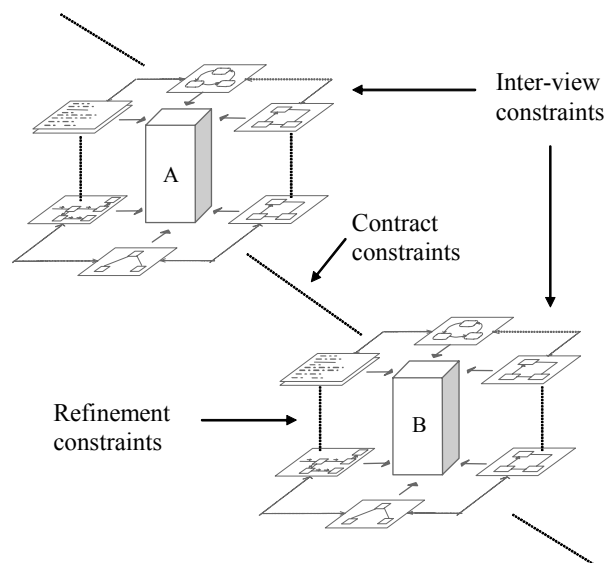


Abbildung 2: Constraints in Kobra-Modellen

Einer der Hauptgründe dafür, die Kobra-Modelle auf jenen von Fusion aufzubauen, liegt darin, deren strenge Vorgaben und die gute Qualitätskontrolle auszunutzen. Es ist gemeinhin anerkannt, dass die Fusion-Methode auf Grund ihrer sehr strengen Konsistenz-Vorgaben zwischen den Modellen und des sehr systematischen Entwicklungsprozesses einer der planvollsten Ansätze für die objektorientierte Software-Entwicklung ist. Abbildung 2 zeigt, wie Kobra diese Ideen aufgegriffen und zu einer umfassenden Menge von Konsistenz-Vorgaben ausgebaut hat. Diese bieten sich nicht

nur dazu an, die Korrektheit von Modellen untereinander zu prüfen, sondern eröffnen auch einen Weg, die Vollständigkeit eines Modells zu definieren. Das bedeutet z.B. konkret, dass Informationen, in einem Modell, die nicht von anderen Modellen genutzt werden, überflüssig sind und entfernt werden können.

Der Hauptunterschied zwischen KobrA und “traditionellen” OO-Methoden wie Fusion oder OMT ist der, dass dieser Modellierungsansatz rekursiv angewendet wird, um eine geschachtelte Hierarchie von (Sub-)Komponenten zu erhalten, die alle auf die gleiche Art und Weise beschrieben sind und in einer Baumstruktur organisiert werden. Wie in Abbildung 2 zu sehen ist, werden die Konsistenz-Vorgaben zwischen Spezifikation und Realisierung (Refinement Constraints) und die Inter-View-Constraints zwischen den drei Dimensionen der objektorientierten Analyse durch sog. Contract Constraints zwischen Komponenten ergänzt. Diese garantieren, dass die in der Spezifikation einer Subkomponente definierten Eigenschaften den Vorgaben entsprechen, die die entsprechende Superkomponente in ihrer Realisierung festgelegt hat.

Wichtig ist, dass nicht alle der oben genannten Modelle für jede Komponente erstellt werden müssen, da es vorkommen kann, dass ein Model keine nutzbringende Information enthält. Dies kann beispielsweise bei sehr kleinen oder vollständig passiven Komponenten der Fall sein. Allerdings gilt in KobrA die Auffassung, dass ein Modell, wenn es für notwendig erachtet wird, genau die oben angesprochene Form haben muss. Weitere, hier bisher nicht genannte Artefakte, wie z.B. Test-Cases oder Data-Dictionaries können für eine umfassendere Dokumentation von Komponenten verwendet werden.

3 Plattformunabhängige Modellierung

Bekanntermaßen beschreibt eine Software-Entwicklungsmethode zwei grundlegende Teile: einen *Prozess*, mit dessen Hilfe ein *Produkt* erzeugt wird. Die komponentenorientierte, modellgetriebene Architektur, die im letzten Abschnitt vorgestellt wurde, beschreibt KobrAs Komponenten-Hierarchie, also das Produkt eines KobrA-Projektes. Der nun folgende Abschnitt beschreibt den dazugehörigen Entwicklungs-Prozess.

Genau genommen ist ein KobrA-Produkt allerdings unabhängig von einem bestimmten Prozess definiert, d.h. es kann jeder beliebige Prozess zur Erstellung des Produkts verwendet werden. So lange alle Artefakte entwickelt werden, die oben angesprochen wurden, ist es unwichtig wie diese erstellt werden. In der Praxis aber benötigen Firmen, die KobrA verwenden möchten, einen definierten Prozess. Mehr noch, je einfacher und systematischer dieser Prozess ist, desto einfacher und weniger fehleranfällig ist der Einsatz von KobrA letztendlich möglich.

3.1 Entwicklungsprozess für komponentenorientierte PIMs

Diese Einfachheit erreicht KobrA hauptsächlich durch die Verwendung eines rekursiven Entwicklungsprozesses, der sich sehr elegant in Verbindung mit dem KobrA-Produkt,

also der verschachtelten Komponenten-Hierarchie einsetzen lässt. Mit anderen Worten, das Endprodukt kann letztlich durch die wiederholte Anwendung einfacher Entwicklungsaktivitäten erzeugt werden, wodurch der Prozess auch naturgemäß leicht für größere Projekte skalierbar ist. Dies garantiert nicht nur die verlangte Einfachheit des Prozesses, sondern ermöglicht auch sehr klare Vorgaben, welche Aktivitäten für welches Produkt notwendig sind.

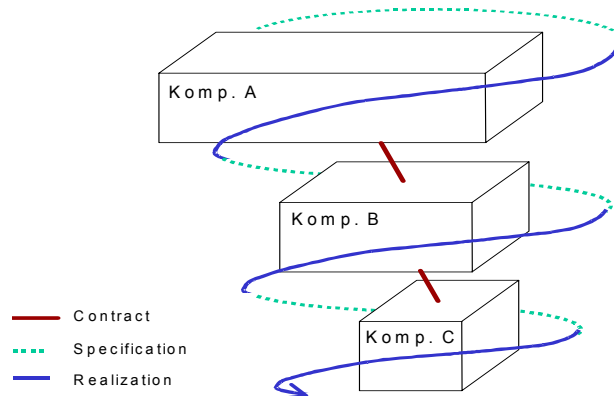


Abbildung 3: Der rekursive Prozess zur Erzeugung einer Komponenten-Hierarchie

Abbildung 3 illustriert die grundlegende Idee, die hinter dem Entwicklungsprozess in KobrA steckt. Spezifikations- und Realisierungs-Aktivitäten, die grob gesprochen den Analyse- und Design-Phasen in herkömmlichen Prozessen entsprechen, werden rekursiv angewendet, um eine Komponenten-Hierarchie zu erzeugen.

Um die Spezifikation einer Komponente (z.B. Komp. B in Abb. 3) erstellen zu können, wird als Vorgabe die Realisierung einer anderen Komponente (beispielsweise Komp. A) benötigt. Daraus kann dann jeweils der, für das Zusammenspiel der Komponenten notwendige „Contract“ abgeleitet werden. Eine Unterkomponente kann danach sowohl „from scratch“ entwickelt werden oder es kann auch, um alle Vorteile eines komponentenorientierten Ansatzes nutzen zu können, eine bereits vorhandene Komponente wiederverwendet werden. Dies stellt den Entwickler vor das „berühmte Make-or-Buy-Problem“, also ob er durch den Zukauf oder die Wiederverwendung einer bereits vorhandenen Komponente eine eigene Realisierung einsparen kann.

Die erste Schwierigkeit, die sich dabei stellt, ist das Finden einer Komponente deren angebotene Services und Eigenschaften eine Wiederverwendung überhaupt lukrativ machen würden. Der optimale Fall wäre eine bereits existierende Komponente, deren Realisierung genau zu der vorgegebenen Spezifikation passt, so dass sie ohne weitere Modifikationen genutzt werden kann. Dies wird in der Praxis allerdings selten oder überhaupt nicht der Fall sein, normalerweise sind immer zumindest kleine Änderungen erforderlich. Der nächste Schritt ist daher das „Aushandeln“ einer für beide Seiten akzeptablen Schnittstelle. Da in vielen Fällen aber die Implementierung eines Reuse-Kandidaten unbekannt ist (z.B. bei COTS-Komponenten), müssen die Änderungen meist

an der Realisierung der Super-Komponente getätigt werden, was natürlich noch weitere Änderungen im Komponenten-Baum nach sich ziehen kann.

3.2 Kontext-Modellierung

Eine grundlegende Frage drängt sich förmlich auf, wenn man den eben beschriebenen rekursiven Entwicklungsprozess betrachtet: Wo beginnt dieser Prozess? Kobra bietet mit der Kontext-Realisierung darauf eine konkrete Antwort an. Diese ist eine spezielle Variante einer regulären Realisierung, deren Aufgabe darin besteht, den Anfangspunkt für die Systementwicklung vorzugeben. Sie enthält die gleichen Modelle wie eine reguläre Realisierung, nur ohne dabei eine vorherige Spezifikation zur Verfügung zu haben. Die Kontext-Realisierung wird daher in Kobra als eine Pseudo-Komponente betrachtet, die das Umfeld des zu entwickelnden Systems aus Business-Prozess-Sicht darstellt und somit einem CIM im Sinne der MDA entspricht.

Als Ersatz für die nicht vorhandene Kontext-Spezifikation beginnt Kobra mit der Erfassung der Anforderungen an das System. Dies kann auf Basis bereits existierender oder neu zu erstellender Business-Modelle oder ganz klassisch mit Hilfe einer Use-Case-Analyse erfolgen. Daraus werden die erforderlichen drei Realisierungs-Dimensionen (Structural-, Usage- und Interaction-Models) abgeleitet, die als Ausgangspunkt für den plattformunabhängigen Entwurf des Systems dienen. Damit wird Kobras Kontext-Realisierung bereits um einiges konkreter und weitaus besser nutzbar, als dies gemeinhin innerhalb der MDA bei einem CIM der Fall ist. Die Fallstudie in Abschnitt 5 verdeutlicht dies noch einmal an Hand von konkreten Auszügen aus einem „Kobra-CIM“.

4 Ableitung komponentenorientierter PSMs

Die Spezifikations- und Realisierungs-Aktivitäten aus den vorhergehenden Abschnitten ergeben eine abstrakte Beschreibung der Komponenten-Hierarchie. Diese ist jedoch, da sie aus plattformunabhängigen UML-Modellen besteht, nicht ausführbar, d.h. sie muss mit weiteren Schritten in eine ausführbare Form gebracht werden. Eine wichtige Grundregel in Kobra ist, dass diese Implementierungsschritte orthogonal zu der in den UML-Modellen durchgeführten Dekomposition sind. Der konkrete Nutzen davon ist, dass Kobra im Hinblick auf gängige Implementierungssprachen eine sehr große Flexibilität zur Verfügung stellt. Wie Abbildung 4 verdeutlicht, werden ganz im Sinne der MDA die plattformunabhängigen Komponenten-Realisierungen noch einmal in plattformspezifische Implementierungs-Modelle verfeinert (Refinement), bevor sie dann in den Sourcecode übersetzt werden (Translation).

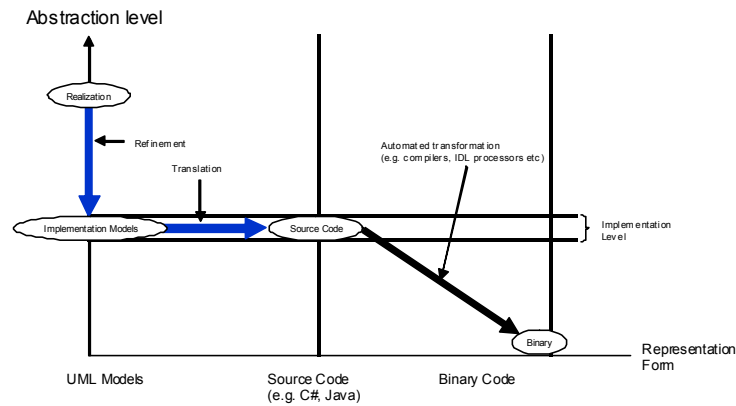


Abbildung 4: PSM- und Code-Erzeugung in Kobra

Diese Aufteilung in zwei Schritte garantiert eine saubere Trennung von Entwurfsentscheidungen, die bei der Überführung in ein plattformspezifisches Modell getroffen werden müssen sowie der anschließenden Umsetzung in den Source-Code. Beide Schritte können bei entsprechender Tool-Unterstützung auch automatisiert durchgeführt werden. Kobra beschreibt mit einem Implementierungsmodell bereits das komplette System, so dass eine Überführung in den zugehörigen Source-Code nur noch eine Übersetzung in eine andere Darstellungsart bedeutet. Der Abstraktions- bzw. Detaillierungsgrad der Darstellung wird auf dieser Ebene nicht mehr verändert. Für diese vorhergehende Konkretisierung nutzt Kobra sogenannte UML-Implementierungs-Profile, die Komponenten-Realisierungen in Implementierungs-Modelle überführen. Es sind also bereits alle Grundlagen für eine MDA-gerechte, automatisierte Code-Erzeugung in Kobra vorhanden.

In Kobra ist es nicht zwingend, dass die logischen Komponenten eines PIM eins zu eins in ein PSM bzw. „physische“ (ausführbare) Einheiten übersetzt werden. Abbildung 5 zeigt, dass es ohne weiteres möglich ist, zwei logische Komponenten zu einer ausführbaren Komponente zusammenzufassen. Da eine logische Komponente in Kobra die kleinste Granularitätsebene ist, macht die umgekehrte Vorgehensweise allerdings keinen Sinn.

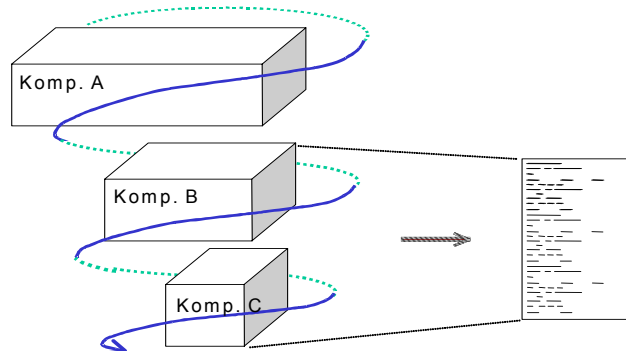


Abbildung 5: Sub-Komponenten in der Komponenten-Hierarchie als eigenes System

Gemäß dem Prinzip der Gleichförmigkeit macht Kobra keinen Unterschied zwischen System und Komponente, weshalb es ebenso möglich, ein erzeugtes „Executable“ als komplettes System und nicht nur als Komponente zu betrachten. Wichtig dabei ist, dass nicht alle Komponenten, notwendigerweise in jedes „Executable“ gepackt werden müssen. Es müssen immer nur die Sub-Komponenten einer ausgewählten Komponente ebenfalls in ausführbare Form gebracht werden.

5 Exemplarische Fallstudie

Dieser Abschnitt verdeutlicht mit Hilfe einer Fallstudie, wie mit der Kobra-Methode Modelle komponentenorientierter Software im Sinne der MDA entwickelt werden. Dabei erläutern wir exemplarisch einige Modelle, die während der Modellierung mit Kobra erzeugt werden und beschreiben Teile des Entwicklungsprozesses. Als Fallstudie dient die Modellierung einer „Simple-International-Bank“ (SIB). Kunden der SIB können Ein- und Auszahlungen in verschiedenen Währungen vornehmen, die das Banksystem automatisch umrechnet.

5.1 Kontext-Modellierung – CIM

Das Ziel der Kontext-Realisierung ist es, den Geschäftsprozess mit Hilfe von interagierenden Aktoren zu beschreiben. Ein oder mehrere dieser Aktoren können als automatisierte, computerbasierte Systeme realisiert werden und somit als Software-System im traditionellen Sinne behandelt werden.

Zur Kontext-Modellierung in der Kobra-Methode gehören verschiedene Modelle:

- Geschäftsmodell (Geschäftskonzeptdiagramme, Geschäftsprozessdiagramme)
- Strukturmodell (Klassen- und Objektdiagramme, Aktivitätenspezifikation)
- Usage Modell (Aktivitätsdiagramme)
- Interaktionsmodell (Sequenz- und Kommunikationsdiagramme)

Zusätzlich können je nach Bedarf weitere Artefakte wie beispielsweise Use Cases oder ein Data Dictionary für die Kontext-Modellierung verwendet werden. Im Folgenden beschreiben wir nur die in den Abbildungen vorgestellten Diagramme, da eine ausführliche Beschreibung aller Modelle den Rahmen dieser Arbeit sprengen würde.

Das Klassendiagramm (siehe Abbildung 6) stellt das Domain Model dar, welches die grundlegenden Konzepte der Domäne veranschaulicht. In unserem Beispiel sind dies die Bank, ein oder mehrere Bankangestellte, Kunden und deren Konten.

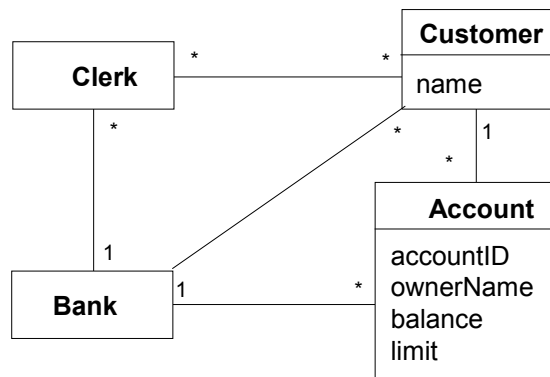


Abbildung 6: Domain Model der SIB

Das Usage Model (siehe Aktivitätsdiagramm in Abbildung 7) beschreibt, wie das System die vom Benutzer benötigten Operationen realisiert. Die Beschreibung erfolgt ausschließlich auf der Ebene der Systemoperationen und verdeutlicht, welche Abläufe notwendig sind, um eine gewünschte Funktionalität zu realisieren.

Das nachfolgende Aktivitätsdiagramm zeigt einen Geschäftsprozess der SIB, nämlich die withdraw-Operation, mit deren Hilfe ein Geldbetrag von einem Konto abgeboben werden kann. Am Ablauf dieses Geschäftsprozesses sind neben den beiden Objekten Bank und Account auch die Aktoren Customer und Clerk beteiligt.

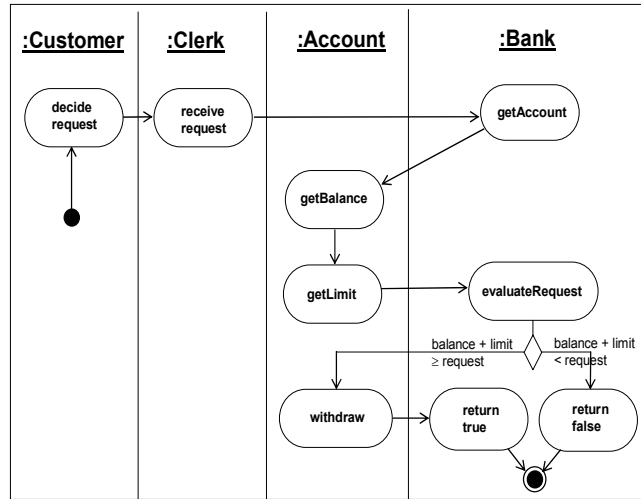


Abbildung 7: Aktivitätsdiagramm der withdraw-Operation

Vor dem Entwickeln einer Kontext-Realisierung mit der Kobra-Methode steht die Struktur des zu entwickelnden Systems üblicherweise in groben Zügen fest. Dies muss jedoch nicht der Regelfall sein. Die Kontext-Realisierung kann auch aus dem Grund erstellt werden, um den Geschäftsprozess und seine Umgebung in Form von interagierenden Aktoren, welche als Black-Box betrachtet werden, zu verstehen. Dies entspricht der von der MDA vorgesehene Verwendung eines CIM. Die Entscheidung einen Actor als Softwarekomponente zu realisieren, muss nicht endgültig während der Kontext-Realisierung erfolgen, sondern kann während des Entwicklungsprozesses zu jedem Zeitpunkt getroffen werden.

5.2 Spezifikation/Realisierung - PIM

Die Spezifikation und die Realisierung einer Software-Komponente nach der Kobra-Methode entsprechen der traditionellen Black-Box- und White-Box-Sicht auf die Software-Komponente. D.h. Spezifikation sowie Realisierung erfolgen in plattform-unabhängiger Weise und gehen somit nicht auf die gewünschte Zielplattform ein.

Die Kobra-Spezifikation einer Software-Komponente besteht hauptsächlich aus drei Modellen:

- Strukturelles Modell (Klassen- und Objektdiagramme)
- Verhaltensmodell (Zustandsdiagramme)
- Funktionales Modell (Operations-Spezifikation)

Die folgende Abbildung 8 zeigt das Klassendiagramm der Spezifikation der SIB. Es wird nur die äußere Sicht auf die Komponente beschrieben. Somit werden im Diagramm alle von außen sichtbaren Konzepte dargestellt, wie z.B. der Account, der Name der Bank oder die Anzahl vorhandener Accounts. Das Modell enthält OCL-Constraints um

z.B. Initialwerte und Invarianten zu beschreiben. Die Komponente deren Spezifikation dargestellt wird, wird durch den Stereotyp <<subject>> gekennzeichnet.

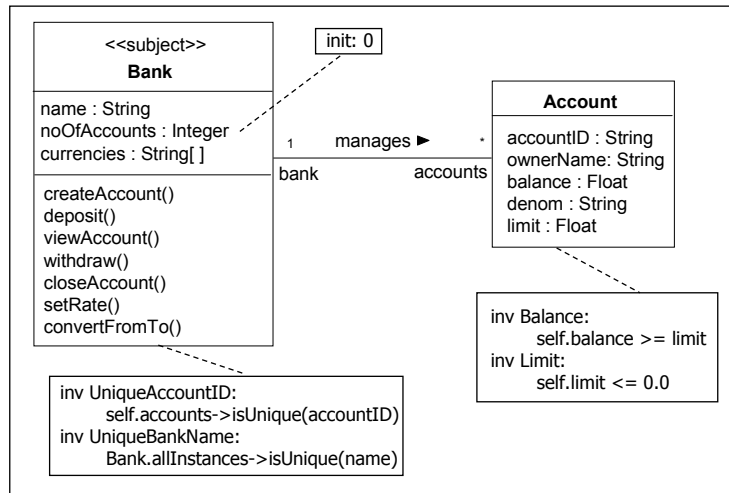


Abbildung 8: Spezifikation (Strukturelles Modell) der SIB

In der Spezifikation einer Komponente werden auch benötigte Komponenten dargestellt, sofern sie für den Benutzer sichtbar sein müssen. Mit anderen Worten eine Komponente muss aufgenommen werden, wenn sie zur Laufzeit von der, in der Spezifikation beschriebenen Komponente benötigt wird und keine Subkomponente von ihr ist.

Die folgende Tabelle (Abbildung 9) zeigt die Operations-Spezifikation der withdraw-Operation, in der ihre nach außen sichtbaren Effekte beschrieben werden. Dies kann mehr oder weniger umgangssprachlich oder wie im Beispiel gezeigt, auch formal mit OCL erfolgen.

Name	withdraw
Informal Description	An amount of money in a particular currency is withdrawn from an account.
Receives	ID : String, currency : String, amount : Real
Returns	wdpossible(ID : String, currency : String, amount : Real) : Boolean = let acc : Account = self.accounts->select(accountID = ID)->asSequence()->first() in currencies->includes(currency) and amount > 0 and acc.balance - convertFromTo(amount, currency, acc.denom) > acc.limit
Changes	account with accountID = ID
Rules	let acc : Account = self.accounts->select(accountID = ID)->asSequence()->first() in if currency = acc.denom then equivalentAmount = amount else equivalentAmount = convertFromTo(amount, currency, acc.denom) endif
Result	post: if (wdpossible(ID, currency, amount)) then acc.balance = acc.balance@pre – equivalentAmount else acc.balance = acc.balance@pre endif

Abbildung 9: withdraw-Operations-Spezifikation

Die Kobra-Realisierung einer Komponente beschreibt detailliert, wie die Komponente die gewünschten Anforderungen realisiert. Zu einer Kobra-Realisierung gehören folgende Modelle:

- Strukturelles Modell (Klassen- und Objektdiagramme)
- Aktivitätsmodell (Aktivitätsdiagramme)
- Interaktionsmodell (Sequenz- und Kommunikationsdiagramme)

Abbildung 10 zeigt das Klassendiagramm der Realisierung unseres Fallbeispiels, für das das Modell der Spezifikation als Grundlage diente. Während des Entwurfsprozesses wird das Modell der Spezifikation um die zur Realisierung benötigten Elemente ergänzt. In unserer Fallstudie werden beispielsweise zusätzlich die Komponenten Converter und Teller benötigt. Zur Unterscheidung von einfachen Klassen, die nur elementare Konzepte der Domäne oder Datenstrukturen repräsentieren (wie z.B. Account und PersistentAccount), werden Kobra-Komponenten mit dem Stereotyp <<Komponent>> gekennzeichnet. Werden diese Komponenten selbst entwickelt, so muss im Rahmen des rekursiven Entwicklungsprozesses für diese Komponenten eine Spezifikation und eine Realisierung erfolgen. Handelt es sich um sogenannte COTS-Software (Commercial-Off-The-Shelf), sollte bereits eine Spezifikation dieser Komponente vorliegen und es ist keine eigene Modellierung mehr erforderlich.

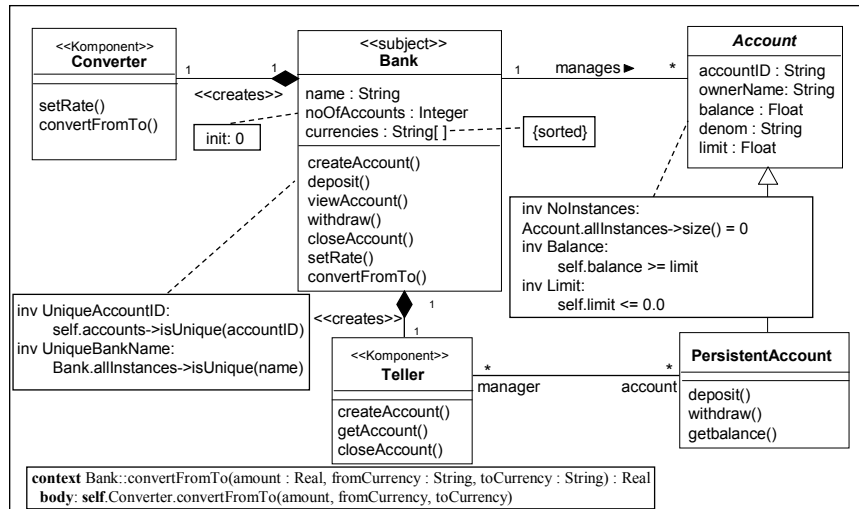


Abbildung 10: Realisierung (Strukturelles Modell der SIB)

Das in Abbildung 11 folgende Kommunikationsdiagramm verdeutlicht die Realisierung der withdraw-Operation. Es beschreibt wie die einzelnen Objekte miteinander interagieren und somit zur Realisierung beitragen.

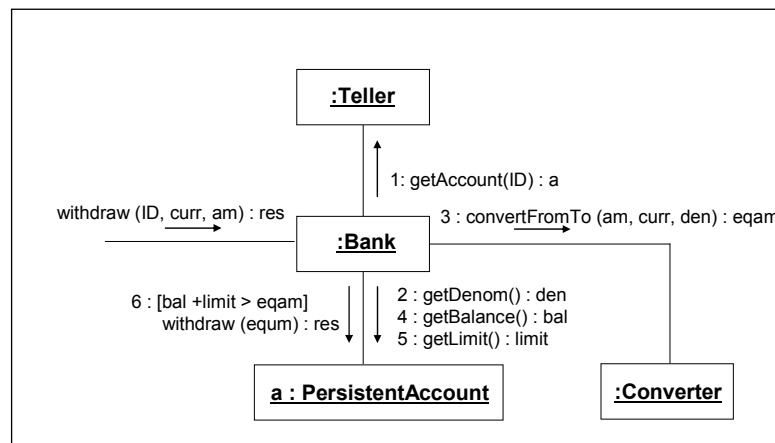


Abbildung 11: withdraw-Kommunikationsdiagramm

Anhand des Beispiels haben wir verdeutlicht, wie es möglich ist, mit Hilfe der Kobra-Methode alle Aspekte einer Software-Komponente sowohl auf der Ebene des PIM als auch auf der Ebene des CIM zu beschreiben. Im nächsten Abschnitt fassen wir noch einmal alle Vorteile der Kobra-Methode zusammen.

Zusammenfassung

Wir haben in diesem Papier gezeigt, dass es nicht nur notwendig ist, komponentenbasierte PSMs erstellen zu können, um die volle Synergie von modellgetriebener und komponentenbasierter Entwicklung zu nutzen, sondern dass es genauso wichtig ist, PIMs und CIMs angemessen in komponentenorientierter Form darstellen zu können. Das bedeutet allerdings nicht, dass PIMs und CIMs auf typischer plattformspezifischer Ebene mit Hilfe der gängigen Komponenten-Technologien wie J2EE, .NET oder Webservices beschrieben werden sollen. Ganz im Gegenteil muss ein PIM, um vollständig plattformunabhängig zu sein, frei von „physischen“ Aufteilungsentscheidungen und Konstrukten einer spezifischen Plattform gehalten werden. Und zwar deswegen, weil die Wahl einer Plattform die physikalische Konfiguration der Hardware genauso festlegt, wie die unterstützende Middleware bzw. Komponenten-Technologie. Werden also solche Entscheidungen bereits in den PIM-Entwurf eingebunden, wird das System später unnötig eingeschränkt.

Die vorgestellte Kobra-Methode vermeidet genau das, da sie ein System in Form von logischen Komponenten beschreibt, ohne dabei zu spezifizieren, wie diese „physisch“ zu Einheiten zusammengefasst werden sollen. Mit anderen Worten, Kobra erlaubt komponentenorientierte PIMs unabhängig von ausführbaren Komponenten. Der Schlüssel um dies zu erreichen, liegt darin, alle verhaltensreichen Objekte unabhängig von ihrer jeweiligen Position in der Teile-/Ganzes-Hierarchie in einem System, auf die gleiche Art und Weise zu modellieren. Dies erlaubt es, einen logischen Teil des Systems auf eine separat ausführbare, physische Komponente oder auf interne Software-Elemente abzubilden, je nachdem, wie es die Ziel-Plattform verlangt.

Es ist mit der UML 2.0 mittlerweile auch möglich, Konzepte und Notationen der physischen Komponenten-Technologien auf Ebene der PIMs zu benutzen. Die Abbildungen 12 und 13 zeigen, wie die externen Eigenschaften einer internen Struktur des Bank-Beispiels, aus dem letzten Abschnitt mit Hilfe der neuen Interface-, Komponenten- und Verbund-Fähigkeiten der UML 2.0 dargestellt werden können.

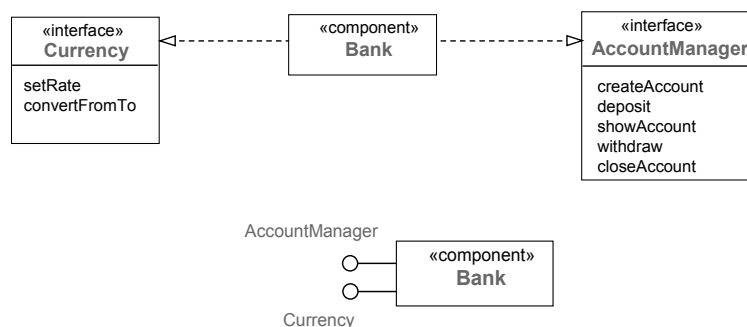


Abbildung 12: UML 2.0 Interface-Notation

Die Abbildung 12 zeigt nach außen sichtbare (black box) Eigenschaften der Bank in Form von zwei verschiedenen Interfaces, nämlich Currency und AccountManager. Abbildung 13 dagegen zeigt die interne Struktur (white box) der Bank in Form von zwei internen Komponenten, die die Implementierung der eben gezeigten Currency- und AccountManager-Interfaces darstellen.

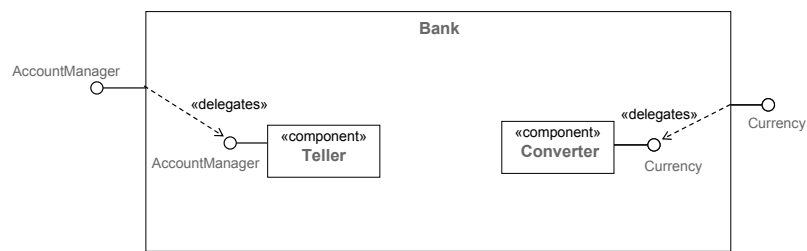


Abbildung 13: Interne Struktur einer Komponente in UML 2.0

Die Interpretation dieser Darstellung als PIM wirft allerdings das Problem auf, dass sie, streng genommen, eine voreilige Abbildung von logischen Eigenschaften auf Implementierungskonstrukte impliziert. Interfaces, Komponenten und Verbindungen sind konkrete Konzepte und Mechanismen von physischen Komponenten-Plattformen. Wenn diese benutzt werden, ein PIM zu beschreiben, wird es schwierig, wenn nicht gar unmöglich, diese Zuordnung später noch einmal zu ändern. Es ist daher wesentlich besser, ein PIM von solchen Konzepten frei zu halten und die logischen Komponenten so abstrakt und flexibel wie möglich zu beschreiben, wie es in Kobra geschieht.

Eine Stufe höher in der Abstraktionshierarchie gelten für die Beschreibung des CIM die gleichen Prinzipien. Während das Ziel eines PIM ist, die logischen Teile eines Software-Systems so zu beschreiben, dass ihre Implementierung möglichst offen bleibt, ist das Ziel des CIM, die logischen Teile eines Geschäftsprozesses oder einer Umgebung so zu beschreiben, dass eine Implementierung als Software-System möglich wird. Somit ist der einzige wirkliche Unterschied zwischen PIM und CIM der, dass logische Teile im CIM entweder von Software-Systemen oder z.B. auch von Menschen übernommen werden können. In einem PIM dagegen werden alle logischen Teile durch Software repräsentiert, sei es nun eine separate physische Komponente oder eine „reguläre“, logische Komponente. In beiden Fällen besteht die Herausforderung darin, eine Gemeinschaft von Aktoren auf Basis von gegenseitig abgestimmten Contracts zu beschreiben. Einige dieser Aktoren mögen dann als Software-System zu realisieren sein und müssen daher als Komponenten in PIMs beschrieben werden, die wiederum in einem PSM verfeinert werden.

Das in Kobra geltende Prinzip der Gleichförmigkeit unterstützt diese strukturelle Ähnlichkeit von PIMs und CIMs ganz automatisch, da es verlangt, dass Systeme (also der logische Teil eines CIM) genauso modelliert werden, wie logische Komponenten (d.h. der logische Teil eines PIM). Kobra bekräftigt dies noch einmal durch die

Benutzung derselben Menge von Modellen, um sowohl ein CIM, als auch die Realisierung einer logischen Komponente in einem PIM zu beschreiben.

Zukünftig ist gerade für die modellgetriebene Entwicklung, aber auch für komponentenbasierte Ansätze mit einer weiteren Popularitätssteigerung zu rechnen. Dies wird dazu führen, dass mehr und mehr Projekte beide Konzepte gemeinsam nutzen. Wir haben mit den in diesem Papier vorgestellten Ideen einige Ansatzpunkte gegeben, die es ermöglichen, den größtmöglichen Nutzen aus der naturgemäßen Synergie dieser beiden Ansätze zu ziehen.

Literaturverzeichnis

- [Atk02] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J. und Zettel, J.: Component-Based Product Line Engineering with UML, The Component Software Series. Addison-Wesley Publishing Company, 2002.
- [Che00] Cheesman, J., Daniels, J.: UML Components, Addison-Wesley, 2000.
- [Col93] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H. Hayes, F., and Jeremaes, P.: Object-Oriented Development: The Fusion Method. Prentice Hall, 1993.
- [Dec96] Deck, M.: Cleanroom and object-oriented software engineering: A unique synergy. In Proceedings of the Eighth Annual Software Technology Conference, Salt Lake City, USA, April 1996.
- [JBR98] Jacobson, I., Booch, G. und Rumbaugh, J.: The Unified Software Development Process, Addison-Wesley, 1998.
- [LA99] Laitenberger, O., Atkinson, C.: Generalizing Perspective-based Inspection to handle Object-Oriented Development Artefacts, ICSE'99, 1999.
- [OMG03] OMG: Unified Modeling Language, UML 2.0 Superstructure Final Adopted Specification. OMG document ptc/03-08-02, 2003.
- [Rum91] Rumbaugh, J. et. al: Object-Oriented Modeling and Design, Prentice Hall, 1991.
- [SOU98] D'Souza, D. and Wills, A.C.: Catalysis: Objects, Frameworks, and Components in UML, Addison-Wesley, 1998.
- [Szy98] Szyperski, C.: Component Software - Beyond Object-Oriented Programming, Addison-Wesley, 1998.