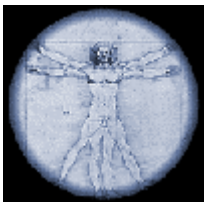
UML



Script

Projektierung, Darstellung und Einsatz von UML 2.0
mit Sourcen und Übungen

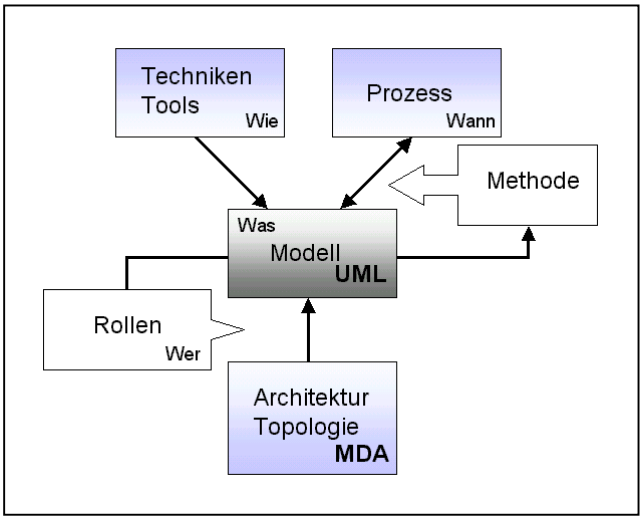
Script Version 2.5, 30.05.2005
Max Kleiner

much communication in a motion
without conversation or a notion...
roxy music

Inhalt:

1 Absicht

1.1 Übersicht



Bei UML steht das Modell mit den Geschäftsprozessen im Mittelpunkt, der eigentliche Prozess regelt die zeitlichen Projektphasen

Abb. 1.1: Das Modell im Mittelpunkt

Seit den ersten Schritten des Software Engineering versuchen Entwickler mit Methode die Komplexität in den Griff zu bekommen. Doch die Methoden waren keine oder nur teilweise Standards und der Begriffswirrwar wurde von Jahr zu Jahr grösser. Die verschiedenen Methoden haben auch unterschiedlich gewichtete Stärken (und auch Schwächen). Zudem lassen sich dabei für gleiche «Dinge» unterschiedliche Begriffe verwenden und unter dem gleichen Begriff versteht man nicht die selben Dinge. Haben Sie schon einmal versucht, Modell und Methode zu definieren. Beobachtet man den heutigen Stand in der Entwicklung der verschiedenen Methoden oder Vorgehensmodelle, so zeigt sich schon heute die Tendenz, dass verschiedene Methoden zu einem universellen Modell zusammengefasst werden. Genauer gesagt ist es eine universelle Methodik zum Modellieren, UML oder Unified Modeling Language genannt.
Die Debatte ob Modell oder Methode wird anhalten, ursprünglich hiess UML auch schon Unified Method Language. Seit der 1.4 ist es aber klar eine Modellsprache.

Schwierigkeitsgrad eines jeden Projektes:
- hohe Interdependenzen der Fachbereiche ohne klare Anforderungen
- teilweise dynamischer Datenfluss der Bewegungsdaten
- Organisatorische Lösung während der Einführung
- Heterogene, proprietäre Umgebung
- komplizierte Geschäftsprozesse (nicht standardisiert)

1.1.1 UML als Industriestandard

Seit 2001 arbeitet man an der Version 2.0, die den modellgetriebenen Ansatz (MDA) in den Vordergrund stellen will. Ivar Jacobson[1] erwähnte an einem Seminar im Dezember 02 in Zürich den Begriff „Executable UML“, welcher angeblich einen der fünf Macrotrends im künftigen Softwarebau sein soll.
Präzise wie ein Gebet, dessen Code Jünger bereits erste Proberunden im Orbit drehen, ist die Aussage von Ivar Jacobson unmißverständlich:
Aus einer Bibliothek von Modellen läßt sich die Zielsprache wählen, die dann auf die entsprechende Plattform portiert und generiert wird.
So wird man wohl weniger programmieren und vermehrt einen Modellkatalog konsultieren müssen. Dies bedingt aber eine stärkere Formalisierung, welche als Gesamtziel des neuen Release hervorgeht. All diese Vorhaben in UML 2.0 laufen unter dem Begriff MDA (Model Driven Architecture) zusammen, welche Modelle in ausführbaren Code transformieren will.

[1] Er gilt als Vater der Use Case Modellierung

1.1.2 Die Bereiche der UML

Am 6. Januar 2003 wurde die überarbeitete Fassung der OMG übergeben. Am Ende dieses Jahres, da erfahrungsgemäß die Finalization Task Force 9 Monate dauert, erwartet die Entwicklergemeinde eine Freigabe der Version 2, die sich in drei separate aber zusammenhängende Gebiete aufteilt:

1. **UML Infrastructure**, welche eine Vereinfachung und Modularisierung des Metamodells vorsieht und keinen Einfluß auf den Benutzer hat. Dieses Gebiet ist für die Toolhersteller und Methodiker interessant, die an Profilen und möglichen Stereotypen Freude und Nutzen haben. Zudem wird das Metamodell von sprachabhängigen Konstrukten (wie C++ oder ADA) gesäubert und sprachneutral, d.h. in OCL definiert.
2. **UML Superstructure**, mit den eigentlichen Erweiterungen bezüglich der Notation und Syntax und somit für den Benutzer sichtbar und verwertbar. Ziemlich aufwerten will die OMG die komponentenbasierte Entwicklung mit z.B. einer präziseren Definition einer Schnittstelle mit erweiterten Signalen oder Nachrichten. Einige statische und dynamische Elemente werden auf Echtzeitfähigkeit getrimmt.
3. **UML OCL/Interchange**, die eine erhöhte Integration in das Metamodell und die Syntax vorsieht, so daß die Spezifikation durch ein Modell zum Code auch eine formale Sprache beinhaltet. Die OCL wird von einer Sprache der Bedingungen zu einer generellen Ausdruckssprache erweitert. Weiter folgt der verbesserte Modellaustausch durch die XMI (XML Metadata Interchange) z.B. mit Layout Informationen zwischen den einzelnen Tools.

1.1.3 Was will die UML ?

UML ist die Abk. für Unified Modeling Language, die eine Vereinheitlichung der unterschiedlichen Ansätze zur Objektorientierten Modellierung von Grady Booch, James Rumbaugh und Ivar Jacobsen, Anfang 1997 bei der OMG (Object Management Group) zur Standardisierung eingereicht, darstellt.

Original-Ton aus www.rational.com:
The Unified Modeling Language [UML97] is a third-generation object-oriented modeling language for specifying, visualizing, and documenting the artifacts of an object-oriented system under development. It fuses the concepts of Booch-93 [BOO94], OMT-2 [OMT91], and OOSE [JACO92]. The result is a single, common, and widely usable modeling language for users of these and other methods.
Die wichtigsten Ziele:

- Konzepte zur durchgängigen Dokumentation
- Kein Strukturbruch zwischen Design und Code
- Standardisierung im Software Engineering
- Codegenerierung
- Methodisches Vorgehensmodell wählbar
- Die Sichten (Fachlich, Technisch) sind wählbar

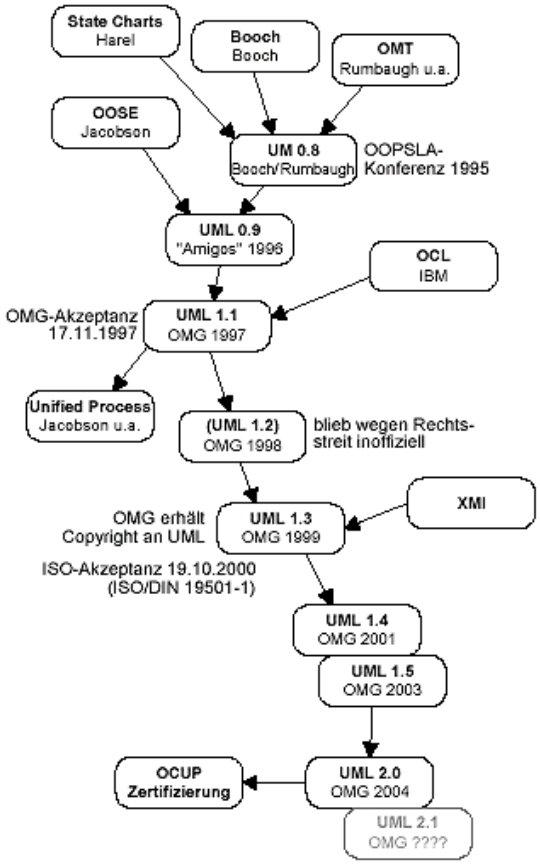


Abb. 1.2: Geschichte der UML

1.2 Konventionen

1. Die Modelle und Konventionen werden gemäss UML 2.0 /1.4 dargestellt.
2. Code-Beispiele sind in OP gehalten, Extrakt vom Buch „Patterns konkret“
3. Die UML-Spezifikation baut auf <OCL> Object Constraint Language auf
4. Mit Prozess ist der Prozess als Vorgehen in Projekten gemeint (sonst näher definiert)
5. Abbildungen (Abb.) wie Tabellen (Tab.) sind separat indexiert.
6. Befehle, Code oder Statements wie Pfadangaben erscheinen in `Courier Schrift`
7. Die meisten Diagramme (s. auch Abb. 6.35) werden gemäss folgender Liste abgekürzt.

Englischer Begriff	**Deutscher Begriff**	**Abkürzung**
use case	Anwendungsfall	UC
activity	Aktivitätsdiagramm	AD
class diagram	Klassendiagramm	CD
object diagram	Objektdiagramm	OD
state event	Zustandsdiagramm	SE
protocol machine	Protokoll Automat	PA
sequence diagram	Sequenzdiagramm	SEQ
interaction overview	Interaktionsübersicht	IAO
communication	Kommunikationsdiagramm	COL
timing diagram	Zeitdiagramm	TI
package	Paketdiagramm	PAC
collaboration	Kollaborationsdiagramm	CL

component	Komponenten	COM
subsystem	Teilsystem	SUB
composition	Komposition	CM
deployment	Verteilungsdiagramm	DEP

Tab. 1.1: Alle UML-Diagramme in 2.0

1.3 UML und MDA

Daß ein Architekturstil nicht nur etwas mit dem Bau und dem Zusammenspiel von Komponenten zwischen den Schichten zu tun hat, sondern auch mit den Personen und Rollen in der Organisation, die solche Komponenten generieren, macht MDA als Technik wie auch als Prozeß interessant.
Die bereits erwähnten Prozessmodelle als Prozesspattern sind vor allem Use Case getrieben, auch XP setzt mit den User Stories die Anforderungen an den Anfang. Wenn aber die Anforderungen schon im Modellansatz bestehen, spricht wenig dagegen, diese modellgetriebene Architektur bei ähnlichen Anforderungen immer wieder einzusetzen. MDA gibt es bereits für existierende Software, die sich mittels Reverse Engineering umwandeln läßt. Zumal MDA auf der höheren Abstraktion ja sprachunabhängig ist. Nun, was meint Model Driven Architecture im Alltag?
Jede Sprache mit der zugehörigen Entwicklungsumgebung ist von einer inneren Architektur von Subsystemen abhängig, z.B. dem Framework der GUI, der Struktur zwischen Interface und Implementierung oder den typischen Datenbankzugriff-Komponenten mit ihren Connection-Strings.

> *Ein UML-Generator 2.0 sollte diese Architektur kennen, damit man sich in den Modellen nicht um diese technischen Details kümmern muß.*

Mein Paketdiagramm sollte also bereits wissen, ob ich z.B. dbExpress mit SOAP statt ADO mit COM einsetzen werde, wenn ich das Modell zum Generieren einsetzen will!
Um diese „Probleme“ bei der Generierung zu lösen, schlägt die OMG vor, ein „Platform Independent Model“ (PIM) durch einen MDA-Generator zu erzeugen. Ein PIM ist ein UML-Modell, in dem die technischen Details nicht ersichtlich sind. Somit befindet sich das Modell auf einer höheren Abstraktionsebene und ist unabhängig von technischen Fakten wie der bekannten Programmiersprache oder Datenbanktechnik.
Das PIM muß aber genug Details besitzen, um das generische Modell durch definierte Abbildungsregeln und Architekturmuster mit einem PSM auf die Zielsprache abzubilden, d.h. innerhalb der Plattform mit zu generieren.

> *Ein PSM (Platform Specific Model) ist das spezifische und technisch abhängige Model, das die Architektur erzeugt.*

Wer sich jetzt fragt, wie die Abgrenzung zu Design Patterns zu setzen ist, der weiß aus eigener Erfahrung, daß ein Tool aus einem Patternkatalog direkt sprachspezifischen Code produziert. So etwas wie ein Design Pattern Generator für Geschäftsprozesse gibt es noch nicht.

> *Also positioniert sich die MDA einen Schritt vor dem Design und ermöglicht bereits in der Analyse sprachunabhängige Klassendiagramme etc. zu generieren.*

Kernidee der MDA (Abb. 1.3) ist also die schrittweise Verfeinerung von der Analyse zum Design ausgehend von einer Modellierung der fachlichen Applikations-/Geschäftslogik, die völlig unabhängig von der technischen Implementierung und der umgebenden IT-Infrastruktur ist. So ein simples PIM-Modell könnte grob so aussehen:
Das kleine Modell soll das abstrakte Typensystem verdeutlichen, ähnlich einer IDL (Interface Definition Language), d.h. jeder mögliche Wert der `queryID` kann als Typ eine Zahl sein, z.B. integer, longint etc. Der neuralgische Punkt des MDA ist die exakte Definition einer Plattform. Wenn nun das Tool aus dem PIM ein plattformspezifisches Modell (PSM) generiert, sollte Einigkeit über die zu implementierende Plattform herrschen. Ist nun eine Plattform CLX, .NET oder J2EE, dann sollten die entsprechenden Klassen und Typen zum Zuge kommen. Was aber ist, wenn die Plattform eine Makro-Sprache mit Objektmodell oder sogar eine Spezifikation wie CORBA ohne Applikationsserver hat.

Der PSM UML-Generator muss also nicht nur die Zielsprache, sondern auch die Zielarchitektur der Plattform kennen.

Im Hinblick auf die Ausarbeitung solcher Details steht die MDA-Bewegung in Bezug auf die vorhandenen Tools noch am Anfang.

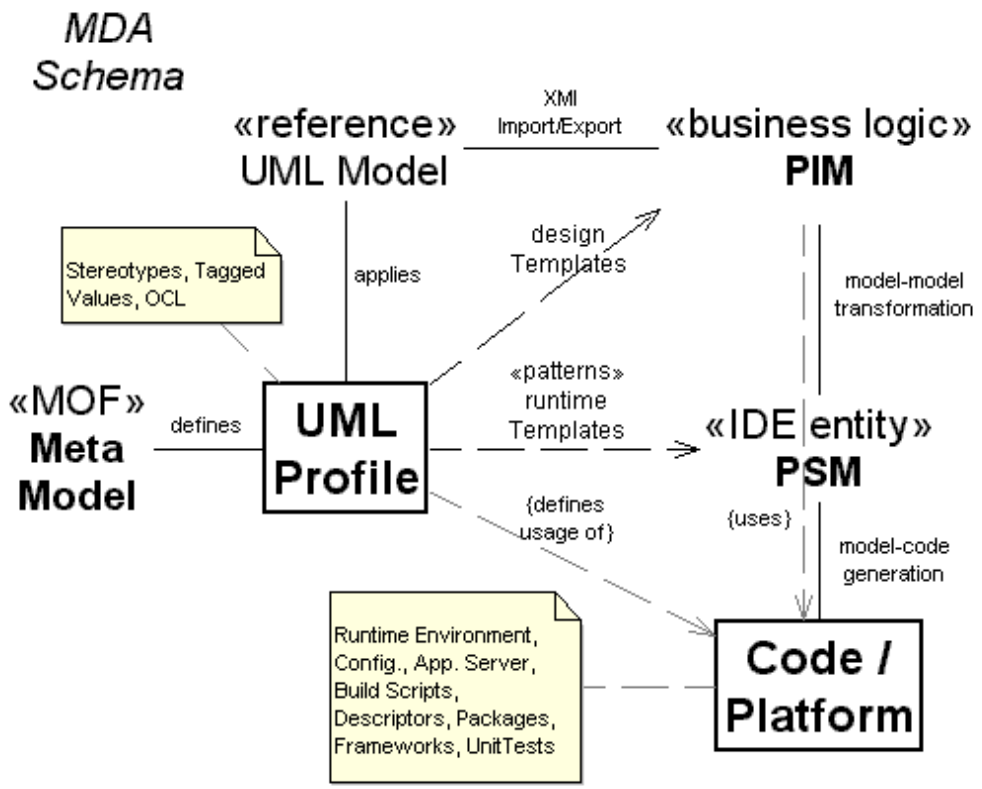


Abb. 1.3: Prinzip von MDA

1.4 Übungsaufgaben und Workshop

Die Aufgaben inklusive Code sind auf http://max.kleiner.com/download/umlscript2.zip

1.4.1 Randbedingungen

- Der Benutzer dieser Unterlagen ist als su eingeloggt
- Das System ist auf Viren und Würmer geprüft.
- Das Geschäftsmodell einer Bank sollte bekannt sein.
- ModelMaker ist vorhanden und auf den Stationen installiert.
- Delphi, FreePascal oder C++ ist installiert

2 Das Metamodell

2.1 Spezifikation

UML baut auf einem dreistufigen Metamodell auf (eigentlich gibt es eine vierte Stufe, die aber hier nicht massgebend ist), das ein logisches Modell repräsentiert. Diese Spezifikation ist für die Tool-Hersteller von entscheidender Bedeutung, bauen sie doch die zugehörige Plausibilisierung und Syntaxprüfung in die Engine ein, damit die Modellierung auch überprüfbar wird. Das Metamodell wird selbst in UML als CD definiert. Machen wir ein erstes konkretes Beispiel anhand eines Klassendiagrammes (CD) im Metamodell:

Layer	**Beschreibung**	**Beispiel**
Metamodell (Syntax)	Hier werden pro Diagramm die möglichen Elemente und Bezeichner definiert	Klasse, Kardinalität, Interface, Methoden, Eigenschaften, Ereignisse
Modell (Notation)	Die konkreten Klassen, Beziehungen und Bezeichner im Modell der zu bauenden Software	TTransaction, SetTransactionHist, Umsatz TrsaktHistory, getData FPosten, FMonat, FTurnover
Objekte Daten (Semantik)	Die konkreten Instanzen des Modells in Laufzeit, d.h. die Daten als Werte oder Code	<Transaction 1001-236> <4523.->, Buchen auf Sparkonto <4>, <1>, <14512.->

Tab. 2.2: Das Metamodell des Klassendiagrammes

The UML metamodel is a logical model and not a physical (or implementation) model. The advantage of a logical metamodel is that it emphasizes declarative semantics, and suppresses implementation details.
Jedes andere Diagramm hat auch entsprechende Instanzen:

Phase	**Modell**	**Instanzen**
Analyse	Use Case	Fach Szenario
Analyse	Activity	Business Units
Analyse/Design	Class Diagram	Objekte
Design	State Event	Zustandswerte
Implementierung	Sequence	Objektszenarios
Implementierung	Package	Versionen
Integration	Component	Files
Integration	Deployment	Geräte

Tab. 2.3: Die Instanzen des Metamodell

2.2 Modelldefinition

A *model* is an abstraction of a modeled system with all diagrams, specifying the modeled system from a certain viewpoint and at a certain level of abstraction.
A model is complete in the sense that it fully describes the whole modeled system at the chosen level of abstraction and viewpoint (tech.-business).

3 Die Diagramme

In der folgenden Aufzählung ist der Versuch einer zeitlichen Abfolge der Hauptdiagramme mit ihren Einsatzgebieten dargestellt, wobei diese «Linearität der Phasen» nur eine Orientierung sein soll. Als Grundlage dient das V-Modell, dass bei den meisten Case-Tools auch Pate stand. Im Prinzip sagt uns V, dass vom Use-Case bis zum State Event ein Top Down-Approach innerhalb Analyse und Design gilt und dann von den Packages zum Deployment der Bottom Up Ansatz in der Implementierung und Einführung gilt (s. Abb. 4.34):

Use Case (UC)
Initialisierung der Anforderungsermittlung
Anforderungen und Soll-Situation darstellen
Geschäftsprozesse im Kontext
allgemeine Diskussionsgrundlage
Definieren der Benutzervertreter als Akteure

Activity Diagram (AD)
Geschäftsprozesse im Ablauf
Ablaufszenarien und Alternativen als Szenarien eines UC
Parallele Prozesse oder Threads
Workflow innerhalb der Organisation

Class Diagram (CD)
Strukturierung v. Methoden und Eigenschaften
OOP-Entitäten als Fachklassen
Finden der Attribute mit erstem Datenmodell oder GUI Design
Relationen der Klassen und so gut wie überall...

State Event (SE)
dynamisches Verhalten innerhalb der Klassen
Objektlebenszyklus der möglichen Zustände
Gültige Zustandsübergänge durch Ereignisse verfolgen

Sequence Diagram (SEQ)
Nachrichtenfluss der Objekte, Testen der UC Scenarios
Zeitliche Aufrufstruktur / Aufrufkaskaden
dynamischer Ablauf zwischen den Objekten
Feinstruktur mit Iteration, Selektion oder Subroutine

Packages (PAC)
Auswirkung von Änderungen durch Abhängigkeiten
Fachliche Gruppierung von Klassen zur Designzeit
Vorbereitung zum Komponentenbau und Modularisieren einer Architektur
Libraries, Übersetzungseinheiten, Versionierung

Components (COM)
Software-Architektur (Implementierung), Schichtenmodell (Multi-Tier etc.) zur Laufzeit
Abhängigkeiten zwischen Prozessen des Systems
ausführbare Einheiten in File/Binärsicht

Deployment (DEP)
Systemarchitektur im Netz / Topologie
Zusammenspiel der Komponenten
Hardwarearchitektur der Geräte und Peripherie
physische Aspekte der Verteilung

3.1 Use Case

Gleich zu Beginn eines Projekts wartet eine kritische Phase: Wenn man die Anforderungen an die zu entwickelnde Applikation nicht verständlich genug definiert, ist es schwieriger, ein genaues Ergebnis zu erzielen. Am genauesten wissen die zukünftigen Anwender, was die Applikation ihnen bieten soll bzw. in welchen Fällen sie die Anwendung einsetzen wollen. Eine nicht verständliche Anforderung am Anfang kann trotz präzisem Design später in einer falschen Anforderung enden. UML trägt dieser Überlegung Rechnung und baut in der Analyse auf die zukünftigen Anwendungsfälle. Die Anforderungsanalyse resultiert in einem (oder mehreren) Anwendungsfall Diagrammen, dessen Elemente **Akteure** einerseits und **Anwendungsfälle** andererseits sind. Die UML stellt für die Darstellung von Anwendungsfällen eine einfache Notation bereit, die wir gleich behandeln.

3.1.1 Die Anforderungen

Use Cases werden definiert, um die Anforderungen der Anwender an die zu entwickelnde Applikation zu analysieren. Für jeden UC lässt sich untersuchen, welche externen Akteure mit dem UC in Beziehung treten und welches Verhalten der UC aus einem spezielleren oder einem weniger pauschalen Anwendungsfall übernehmen kann. Im Diagramm lassen sich Kommunikationsbeziehungen zwischen Akteur und Anwendungsfall sowie Beinhaltet-, Vererbungs- oder Erweiterungsbeziehungen zwischen Anwendungsfällen anlegen. In UML 1.3 ist nebst den Beinhaltet- <include> und Erweitert- <extend> noch die **Vererbungsbeziehung** <generalization> hinzugekommen.

Als gute Quelle zum Finden von UC haben sich externe Ereignisse im Sinne von Anfangsauslösern erwiesen. Mit Hilfe von Interviews und dem Durchsehen von internen Dokumenten der Privatbank wie Verträge, Rechnungen, Anweisungen oder Betriebsrichtlinien erstellen wir eine Ereignisliste:

Nr.	Ereignisse	Link	Bemerkungen
10.	Anlagemöglichkeiten beraten und bestehende Kunden anfragen oder neue Kunden akquirieren	11,15,18	Kommt via Telefon rein oder als schriftliche Anfrage zur Kreditaufnahme
11.	Kontostand abfragen / Transaktionen analysieren	✓	Kunde kennt letzten Saldostand
12.	Mitteilung Gebührenänderung	10	Individuell abmachen
13.	Neue Kunden in Stamm aufnehmen	10,14!	
14.	Konto eröffnen	13,(10)	Vertrag erfassen
15.	Zahlungseingang /Ausgang verbuchen und Transaktionen checken	✓	Zeitstempel als Valuta
16.	Umsatz der Konten ermitteln	15	Nur Tagesumsatz, dann Jahr
17.	Jahresauszug erstellen	11,15	Inklusive Vermögen
18.	Kreditlimite anpassen	✓	Bonität checken

Tab. 3.4: Die Ereignisliste als erste Anforderungen

Als Beschreibung und besseren Recherche von Ereignissen und den entstehenden UC sollte man nach der Ereignisliste ein UseCase-Script erstellen (s. Tab 3.5). Es ist auch möglich, dass man mehrere Ereignisse in einen UC zusammenfasst, weil sie fachlich zusammengehören, und in eine Richtung voneinander abhängig sind. In unserer Ereignisliste aus Tab. 3.4 ist „Nr.13 Neuer Kunde aufnehmen“ evtl. von „Nr.14 Konto eröffnen“ abhängig, ich kann aber auch später bei einem schon bestehenden Kunden ein zusätzliches Konto eröffnen. Der **Link** gilt als Vorbedingung.
Wichtig für die Identifikation von UC ist, dass ein Ereignis am **Anfang einer Kette** steht. „Transaktion speichern“ ist kaum ein Ereignis, da es als Folge von etwas zu sehen ist. Das

UC-Script kann später auch als Teil eines Rahmenvertrag der Software dienen. Exemplarisch beschreiben wir nun unseren ersten UC <Buchen> der aus dem Ereignis Nr. 15 entstanden ist:

Ziel im fachlichen Kontext von <Buchen>	Welche Ziele sind mit dem Anwendungsfall erreichbar? Die Speicherung der Buchungstatsachen
Preconditions	In welchem Zustand befindet sich die fachliche Welt vor Beginn des Anwendungsfalls? Geld und Buchungstatsache sind bewilligt oder vorhanden
Postcondition im Positivfall	In welchem Zustand ist die fachliche Welt nach erfolgreichem Abschluss des UC? (Happy Day Case) Gespeichert und als Transaktion protokolliert
Postcondition im Negativfall	In welchem Zustand ist die fachliche Welt nach Abbruch oder Fehler des Anwendungsfalls? (Bad Day Case) Die Buchung ist nicht gespeichert, Kontostand neu einlesen
Sekundäre Akteure	Wen benötigt das System um den UC auszuführen? Datenbank und Transaktionsmonitor
Event	Welche Aktion löst den UC aus? (Es kann sich dabei auch um ein internes Zeitereignis handeln. Zahlungsein- oder Ausgang (Kauf, Verkauf, Kreditaufnahme)

Tab. 3.5: UseCase-Script zur Beschreibung

3.1.1.1 Anforderungsermittlung

Die Hauptschwierigkeit der Anforderungsermittlung ist sicher das Umsetzen in definierte Regeln in der Designphase (Business Rules), die den UC als Gesamtes dann technisch überprüfen können. Angenommen in der Privatbank existiert ein UC <Arbeitszeit erfassen> der an sich simpel daherkommt, später aber zu bösen Überraschungen führt, weil die Anforderung unterschätzt wurde. Nun was heisst unterschätzt. Die Offerte war wieder einmal zu optimistisch. Es erfolgen weitere Besuche in der Firma, bis wir folgenden Ausschnitt der Richtlinien gefunden haben:

- Die Stunde die weniger gearbeitet werden muss, bezieht sich ausschliesslich auf den Nachmittag.
- Bezieht ein MA Ferien und in diese Ferien fällt ein Feiertag, so ist auf der Stempelkarte dem Ferientag vor dem Feiertag eine Stunde abzuziehen und einzutragen.
- Arbeitet ein MA am Vormittag vor dem Feiertag und bezieht Ferien am Nachmittag, ist am Nachmittag ebenfalls eine Stunde abzuziehen.

So ein Regelwerk in Code umzusetzen kostet in der Regel wieder Tage, die als Entwicklungsaufwand den Zeitplan verzögern. Wir haben uns damit abgefunden, dass die Anforderungsermittlung mühsam ist und haben den Begriff Columbo-Methode geprägt, die zum Ziel hat, solange anzuklopfen bis ein Fall geklärt ist, auch wenn die Firma langsam ungeduldig wird. Denn der Inspektor weiss, wenn die Analyse am Anfang etwas übersieht, ist der Fall später schwieriger zu lösen.

3.1.2 Notation

John Tra Volt: EVERY NIGHT FEVER

Kommen wir zur Notation eines UC. Anhand der beteiligten Aktoren kann es auch zwei oder drei Diagramme geben. In der Zwischenzeit haben wir erfahren, dass der Aktor <Bearbeiter> berechtigt ist, Konfigurationen vornehmen zu müssen. Die diversen Ereignisse betreffend Systemeinstellungen fassen wir im UC <Konfiguration einstellen> zusammen. Für den Broker sind im Moment aus den Ereignissen resultierend folgende Anwendungsfälle bekannt:

- Geld einzahlen, anlegen, abheben
- Kreditaufnahme eingeben
- Kreditlimite prüfen

- Kontostand zeigen
- Chart generieren, darstellen und optimieren
- Kunde / Konto eröffnen
- Kontoauszug / Vermögen
- Bewegungsliste / Transaktionen
- Speichern / Exportieren /Drucken
- Umsatz ermitteln
- Limiten / Gebühren eingeben
- Zinsrechnung und Abschluss
- Konfiguration vornehmen

3.1.3 Der Akteur

Akteure lassen sich durch Kommunikationsbeziehungen mit Anwendungsfällen verbinden, können aber nicht mit anderen Akteuren in Verbindung treten. Der Akteur ist ein menschliches oder dingliches, systemähnliches Wesen (kann ein anderes System sein), das an einem oder vielen UC beteiligt ist. In der Vergangenheit wurde viel Zeit vertan, um Akteure z.B. als Kunden oder Akteure als Mitarbeiter in Bezug auf das System zu finden. Diese Abgrenzung ist aber künftig weniger eindeutig, da auch Kunden innerhalb des E-Commerce vermehrt direkten Zugang zum System finden. Durch diese **Wechselwirkung** ist der wahre Akteur eher subjektiv festzulegen.

3.1.3.1 Zwei Sichten

Versuchen Sie trotzdem diese Sicht zu trennen um zwei Arten von UC zu verstehen, die Technische Sicht oder die Fachliche Sicht eines UC. In beiden Fällen wollen die Akteure etwas vom System, doch aus der Art der Bedienung ist die Technische Sicht näher an der Datenverarbeitung. Wir haben uns einerseits für die Business Sicht entschieden und sehen den Akteur <Anleger> wie den Akteur <Broker> durch den engen Kontakt eher als Partner der Privatbank.
In einem zweiten UC Diagramm eher technischer Natur kommt der Akteur <Bearbeiter> ins Spiel. Hier bedeutet dann ein UC eher die Schnittstelle zwischen Mensch und Maschine innerhalb der Privatbank.[2]
Um einen Akteur anzulegen, muss sein Name definiert sein. Ein UC lässt sich auch durch eine <annotation>, d.h. eine kleine Notiz ergänzen. Wer oder was genau sich hinter einem UC verbirgt, lässt sich so in ein paar Stichworten ergänzen, zudem wird ja jeder UC durch ein Script näher beschrieben. Die Kommunikationslinie ist von <Broker> zu <Buchen> noch näher mit <Handel festlegen> bezeichnet (Abb. 3.6), d.h. hinter einem Fachvorgang verbirgt sich meistens eine Technische Aktion.

> *Kommunikationsbeziehungen in UC lassen sich nur zwischen Akteur und Anwendungsfall anlegen, nicht zwischen den Akteuren selbst.*

[2] Aus dieser Sicht entsteht das Oberflächen Prototyping

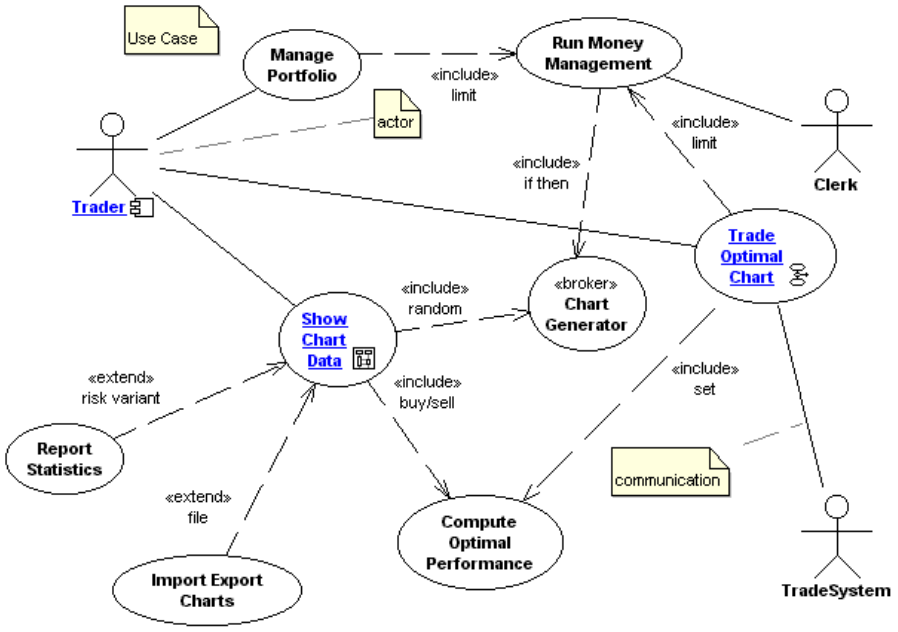


Abb. 3.4: Unser erstes UC Diagramm als Gesamtkontext

3.1.4 Erweitert Notation

Da ein UC auch den Ablauf von Arbeitsgängen und Interaktionen mit der Anforderung an das System beschreibt, ist meistens die normale Variante des Ablaufs im Diagramm sichtbar. Der UC beschreibt ihn somit in einer direkten Weise, so dass sich keine Varianten oder ein alternatives Verhalten ausdrücken lässt. Doch in Geschäftsabläufen gibt es je nach Zustand oder Situation die Möglichkeit, den Arbeitsschritt mit einem anderen Schritt zu erweitern.
Um eben diese **Varianten** dennoch in der Notation des UC zu ermöglichen, die jeweils unterschiedliche Arbeitsschritte darstellen können, sieht UML das Anlegen eines alternativen UC zu einem bestehenden vor. Somit kann man mit der <extend> Notation einem bestehenden UC ein zusätzliches Verhalten verpassen, d.h. er wird um eine Variante reicher, sprich erweitert. Ein zweites Kriterium für den Gebrauch von <extend> ist die schwache Notwendigkeit des anderen UC, d.h. der UC könnte auch ohne die Erweiterung funktionieren.

> *Man benutzt die <extend> Notation bei einem UC, der einem bestehenden UC ähnlich ist, ihn aber um eine Variante erweitert. Der bestehende UC ist nicht unbedingt vom anderen UC abhängig.*

3.1.4.1 Im Fall BROKER

Der Akteur <Trader> benötigt eine Statistik seiner Handelstätigkeit. Die Kommunikation sieht den UC <Report Statistics> in Abb. 3.4 vor, der mit umfangreichen Daten zur Situation des Kunden oder Traders beginnt. Der UC <Show ChartData> wird also um einen Arbeitsschritt erweitert, der bei geringem Handel nicht immer notwendig ist. Nach dessen Schritt kann die Rückkehr zur Ausgangslage erfolgen.

3.1.4.1.1 Semantik

In diesem Kontext des UC ist die **Facharbeit** von Bedeutung. Die Pfeilrichtung der graphischen Notation ist übrigens auch von Bedeutung, da sie gleich der Leserichtung ist: UC <Report Statistics> erweitert UC <Show ChartData>.
Ein UC, den man von einem anderen UC erweitern lässt, schliesst also dessen Variation nicht **jedesmal** in seinen Ablauf ein. Hier lässt sich nun ein Unterschied zur nächsten Notation, der <include> Notation, anbringen: Der Ablauf eines UC, der einen anderen UC beinhaltet, ist für den Ablauf des Ersteren notwendig.
Kurz noch ein letztes Beispiel zu <extend>:

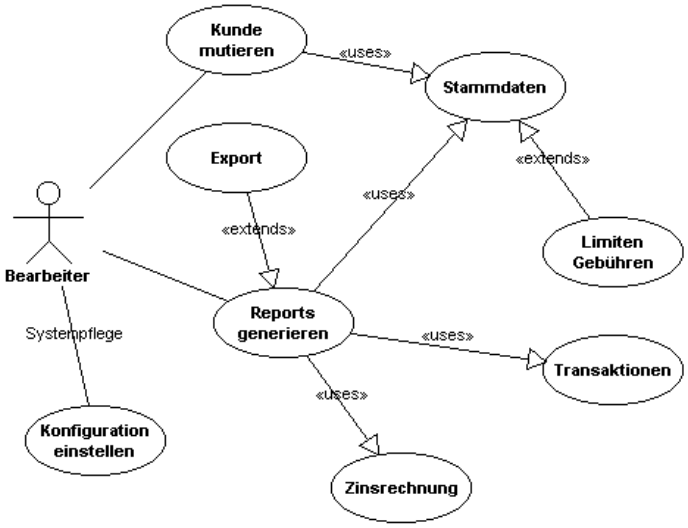


Abb. 3.5: Das zweite UC Diagramm mit <extend> Notation

In Abb. 3.5 gibt es zwei <extend> Notationen, die wir kurz interpretieren: UC <Export> erweitert UC <Reports generieren>, hier ist eine zusätzliche Funktion zum Erstellen der Berichte vorhanden, was aber nicht heisst, dass die Möglichkeit des Druckens auch enthalten ist. Hier ist eben wieder die Genauigkeit der Spezifikation zuständig. Auch die <Limiten Gebühren> erweitern den UC <Stammdaten>, von Zeit zu Zeit, da beim Bearbeiten der Stammdaten nicht jedes Mal das Setzen von Limiten oder das Festlegen von Gebühren notwendig ist.

In der UML 1.3 hat der Stereotyp<extends> zu <extend> gewechselt. Der Stereotyp <uses> hat der Spur nach zu <include> gewechselt. Unser Tool ModelMaker zeigt noch die alte Notation, die Sie des öfteren noch vorfinden.

3.1.5 Beinhaltet Notation

Bei der <include> Notation ist der Ablauf eines UC vom anderen abhängig. Zudem versucht man mit dieser Notation, mehrfach gebrauchte Anforderungen in einen eigenen gemeinsam benutzten UC **auszulagern**, im wahrsten Sinne des Wortes outsourcen. D.h. wenn ein einzelner Arbeitsschritt nicht nur im Rahmen dieses einen UC notwendig ist, sondern auch für einen oder mehrere andere UC vonnöten oder nützlich ist, haben wir einen Kandidat für die <include> Notation gefunden. Die anderen UC beinhalten den ausgelagerten UC.
Später sehen wir, dass bereits Prinzipien vorhanden sind, die sich im Klassenbau wiederfinden. Das heisst auch, dass bereits beim Finden eines UC Ideen wie Wiederverwendbarkeit durch Auslagern (*Komponente*) oder Erweiterbarkeit durch Abgrenzung (*Modul*) vorherrschen. UML ist schlussendlich „Use Case Driven“, nur so ist eine vernünftige Durchgängigkeit möglich.

Man benutzt die <include> Notation bei einem UC, wenn man sich in zwei oder vielen UC wiederholt und diese Wiederholung auslagern und gemeinsam nutzen möchte. Der bestehende UC ist vom UC den er beinhaltet abhängig. Typisch ist auch, dass ein Akteur selten mit einem <include> UC kommuniziert.

3.1.5.1 Im Fall BROKER

Wenn ein <Anleger> in Abb. 3.6 sein Geld anlegen möchte kommuniziert er mit dem UC <Geld anlegen>. <Geld anlegen> wiederum beinhaltet den UC <Buchen>, der auch von einem anderen UC benötigt wird. Das Verhalten für das Buchen der Geldabwicklung ist daher nicht zweimal in einem hypothetischen UC <Geld anlegen buchen> und <Kreditaufnahme buchen> notiert, sondern eben einmal als <Buchen> notiert. Ein evtl. zusätzlicher UC hat dann Zugriff auf diesen ausgelagerten UC.

3.1.5.1.1 Semantik

Durch das Auslagern mehrfach benötigten Verhaltens werden die zwei UC, mit denen der <Anleger> kommuniziert, weniger kompliziert, und das Diagramm lässt sich einfacher lesen. Doch muss gemeinsam benutztes Verhalten auch notwendig sein? Was nun, wenn ein <in-

clude> UC notwendig ist, aber trotzdem nicht mehrfach benötigt wird wie im Falle <Reports generieren> beinhaltet <Zinsrechung> in Abb. 3.5. Dann hat das Kriterium der Notwendigkeit eben Vorrang und die mögliche Mehrfachnutzung ist sekundär. Die Notwendigkeit impliziert, dass z.B. der UC <Buchen> nicht ohne den UC <Transaktionen> ablaufen kann.
Die Definition der <include> Notation kommt von daher, dass ein Verhalten **Teil** möglichst vieler UC werden soll. Darin unterscheidet sich ein <include> von einem <extend> der nur unter besonderen Fällen für einen oder wenige UC brauchbar ist.
Die Leserichtung ist auch hier die Pfeilrichtung, d.h. <Kontostand anzeigen> beinhaltet <Transaktionen>.

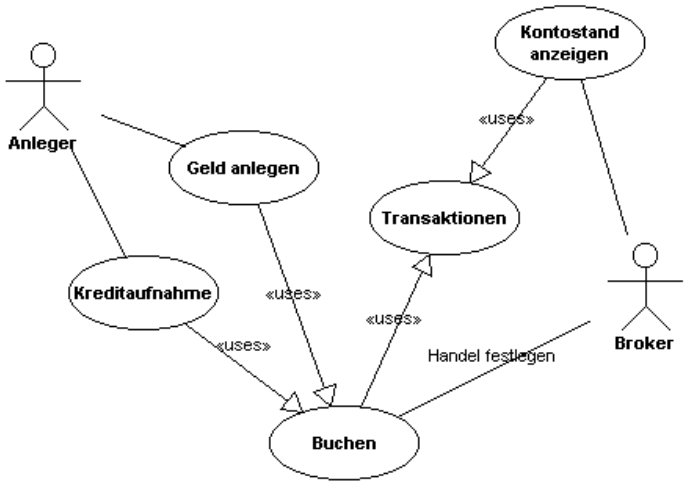


Abb. 3.6: Teil des UC <run money management>

3.1.6 Vererbt Notation

Seit UML 1.3 existiert auch die Vererbt Notation <generalization> die kein Sterotype kennt, d.h. ohne Notation im Diagramm auskommt. Stereotypen sind Notationen in speziellen Klammern wie «include» oder «extend». Die Vererbung ist dem <extend> ähnlich, so dass es bei mehreren Erweiterungen einsetzbar ist, die voneinander abhängig sind. Wenn also der selbe UC durch mehr als ein<extend> erweitert wird und eine Basisbeziehung besteht, sollte man sich eine Vererbung überlegen.

> *Man benutzt die Vererbung in einem UC, wenn eine Erweiterung stark vom Basisverhalten abhängig ist und mehrfach als Spezialisierung vorkommen kann. Somit vermeidet man ein <extend> mit vielen Pfeilen, wenn eine Vererbung präziser ist und die Notation vereinfachen kann.*

3.1.6.1 Im Fall BROKER

Eine mögliche Erweiterung ist der UC <Geld anlegen>. Da man zeigen möchte, dass ein Basisverhalten vorhanden ist, legt man die Erweiterung von <Sparkonto> und <Wertschriften> als Vererbung an. Diese Vererbung ist stärker als eine <extend> Notation, die einem UC nur ähnlich ist.

3.1.6.1.1 Semantik

Die UC <Sparkonto> und <Wertschriften> sind eine Spezialisierung der Generalisierung <Geld anlegen>, d.h. in der Bedeutung ist ein Sparkonto schon ein Spezialfall vom simplen Geld anlegen, das auch nur eine Grundanlage ohne Zins sein könnte. Über gemeinsame Methoden nachzudenken, ist in dieser Phase noch verfrüht, nur die Fachtätigkeit aus der Sicht des Anlegers zählt, und die tendiert dazu, mehrere ähnliche Varianten aus einem Basisverhalten ableiten zu können.

3.2 Activity

«Verstehen und beschreiben was mein Kunde will und braucht»
Wenn wir im folgenden von Prozessen sprechen, sind immer die zu modellierenden Geschäftsprozesse gemeint. Ein Geschäftsprozess selber lässt sich weiter in eine Sammlung von Funktionen auftrennen. Wir modellieren die Geschäftsprozesse von BROKER, somit ist das Modell mit den Abläufen der eigentliche Betrachtungsgegenstand von UML.

> *Der Sinn nach einem Standardprozess kann noch etliche Fragen aufwerfen. Denn zwischen der Exaktheit einer Methode und ihrer Anwendbarkeit besteht ein Zielkonflikt, welcher besagt, je exakter eine Methode desto eingeschränkter ihre anwendbare Nützlichkeit. D. h. exakte Methoden sind meistens nur gezielt und spezifisch einsetzbar.*

3.2.1.1 Business Rules

The Bitles: YELLOW SUBROUTINE

Kommen wir zurück zu unseren Geschäftsprozessen.
Ein Geschäftsprozess ist eine **Folge** von Transaktionen oder Funktionen zum Erreichen betrieblicher Abläufe und Ergebnisse. Gegenstand der Transaktion oder Funktion kann eine Wechselwirkung von Daten und Ergebnissen (Funktion) sein. Diese unternehmensspezifischen Aktivitäten lassen sich nun in einer **Aktivität** (AD) in einen logischen und zeitlichen Zusammenhang bringen.
Prozessorientierte Diagramme haben den Vorteil, eine verständlichere Semantik zu kommunizieren, notwendigerweise auch aus dem Grunde, weil eben die meisten Fachleute nicht in Datenmodellen oder Klassen denken.

3.2.2 Die Aktion im Workflow

Die Elemente eines AD bewähren sich bestens z.B. für die Modellierung von Workflowlösungen und zur Analyse von Anwendungsfällen. Denn die klassischen Workflowsysteme erfordern eine Menge Designvorarbeit vom Unternehmen. In diesem Design enthalten sind die Straffung der Geschäftsabläufe und deren digitale Umsetzung. Die Aktion in einem AD können bereits viele Funktionen einer Workflow-Lösung abdecken.
Dazu gehören etwa Gruppenfunktionen, das Erstellen von Verteilerlisten, Aktivieren von Prozessen und die Überwachung der Abläufe insgesamt. Auch in der Analyse eines UC, bei der die Aktionsabhängigkeiten modelliert werden, setzen wir AD vor allem auch zur Überwindung von Verständnisschwierigkeiten bei Kunden ein. Folgende Ziele treten dabei in den Vordergrund:

- Finden von Aktionen aus Ereignislisten
- Festlegen der statischen Zusammenhänge zwischen den Aktionen
- Festlegen der zeitlichen Abläufe und der dynamischen Beziehungen
- Prüfen der Ereignisliste als Vorbereitung zur Spezifikation
- Beschreibung der Trigger mit ihren Bedingungen
- Suchen nach Parallelitäten zur Synchronisation

In den früheren Versionen von UML war das AD eine Art Zustandsdiagramm, mit dieser ambivalenten Haltung ist in der Version 2 Schluß. Das Diagramm hat eine eigene Basis im Metamodell erhalten und heißt neu schlicht und ergreifend <activity> oder Aktivität.
Gemeinsam sind die beim Zustandsübergang ausgelösten Aktionen, die wiederum Folgeaktivitäten (Ereignisse) auslösen können. Bei viel Ablauflogik werden AD schnell unübersichtlich und «überarrangiert». In dem Fall kann eine konventionelle **Entscheidtabelle** (Bedingung, Regel und Aktionen) die bessere Darstellung sein (s. Tab. 3.6).

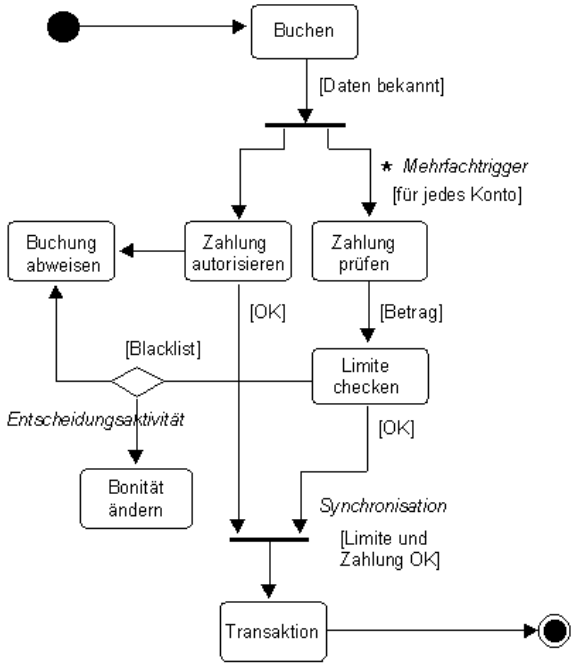


Abb. 3.7: Ein AD ausgehend vom UC <Buchen>

3.2.3 Notation

Zudem eignen sich AD, um die Abschnitte eines Anwendungsfalls (Use Case) detaillierter darzustellen. Ein UC ist ja eine typische Interaktion zwischen Anwender und System, der seine Fortsetzung in einem AD finden kann.
Hinweise, dass sich ein AD für die genauere Betrachtung in einem Anwendungsfall (UC) eignen, finden wir schon in [Kleiner2000][3]. Anwendungsfälle sind die Abschnitte in einem Geschäftsprozess, die durch das System unterstützt werden sollen. Für die Grösse eines UC gilt, dass man einen Ablauf in mehrere UC zerlegen sollte, wenn er sich zeitlich signifikant abgrenzen lässt oder ein Mitarbeiter diesen Prozess ohne grosse Unterbrüche alleine erledigen kann.

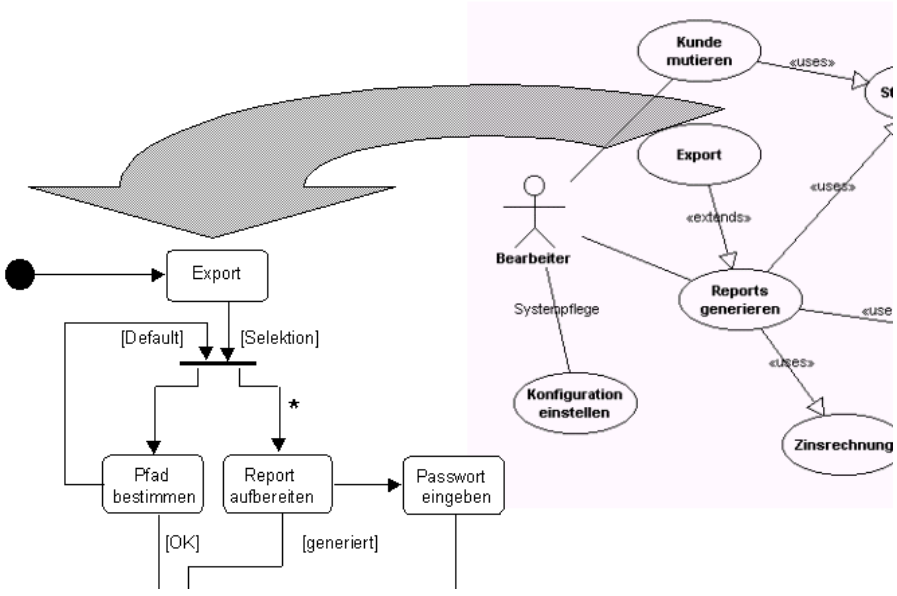


Abb. 3.8: Direkte Ableitung eines UC zum konkreteren AD

Durch die direkte Anbindung eines AD in einem Anwendungsfall, besitzen wir einen Vorteil

[3] Kleiner M., UML mit Delphi, 2000, S. 115, S&S Verlag

gegenüber den Prozessketten, nämlich die Durchgängigkeit. Nach der Kurzbeschreibung von AD kennen Sie die Hauptelemente Aktion und Aktivität. Aktion wie Aktivität stellen eine betriebswirtschaftliche, prozessorientierte Tätigkeit dar.

3.2.3.1 Die Aktionen

Das zentrale Symbol ist die Aktion, wobei man hier von Anfang an die Perspektive festlegen muss aus dem man das Diagramm modelliert. Konkret kann ich eine Aktion auf Stufe Geschäftsprozess, auf Stufe Objekt und Methode oder sogar auf Stufe Multithreading modellieren. Somit ist die Interpretation der Aktion von der **Perspektive** (Sicht) abhängig und nicht umgekehrt.
Viele Entwickler setzen ein AD auch als Ersatz für die Flussdiagramme (Flowchart) ein[4], sozusagen für das Programmieren im Kleinen. Somit lassen sich sehr detailliert auch Algorithmen und Abläufe einzelner Methoden modellieren. Wie gesagt, die Perspektive soll das festlegen.
Jede Aktion kann nun von einer anderen gefolgt werden, d.h. ein AD beschreibt die Reihenfolge mit ihren Bedingungen in einer sequentiellen, aber auch **parallelen** Form. Hier liegt auch der wesentliche Unterschied zu einem Flussdiagramm, welches keine Nebenläufigkeiten darstellen kann. Für die Modellierung von Geschäftsprozessen sind parallele Vorgänge von Bedeutung, da man hier oft unnötige Sequenzialisierungen und Bedingungen oder sogar Leerläufe aufdecken kann. Zudem verbessern parallele Abläufe die Durchlaufzeiten.

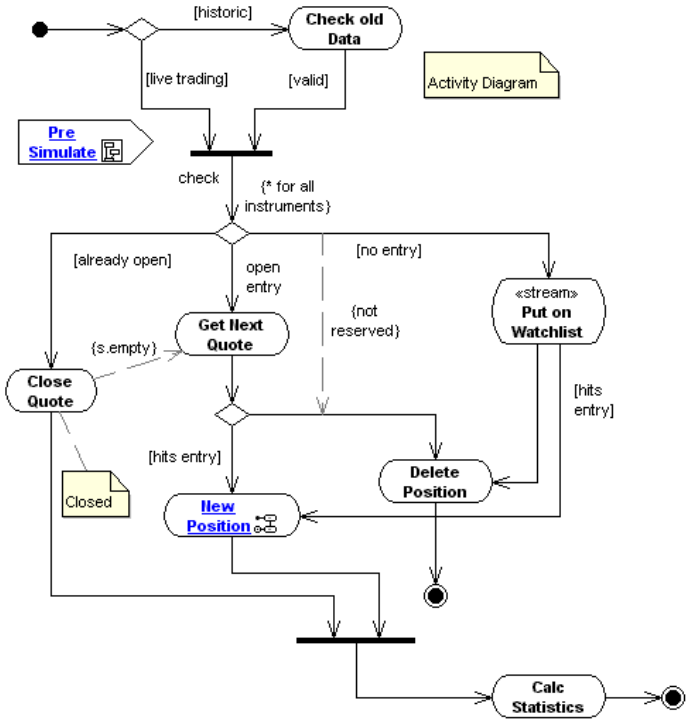


Abb. 3.9: Ein AD mit Kontrollfluss und Bedingungen

3.2.3.2 Tokens (Bedingungsfolge)

Die Aktionen lassen sich nun in einer zeitlichen und logischen Folge darstellen, bei der jede Aktion von mindestens einem Token ausgelöst wird und die Aktivität selbst wieder ein oder mehrere Token auslösen kann. Ein Token ist ein markenbasiertes Teil, welches den Kontrollfluss oder Objektfluss symbolisiert (Abb. 3.10). Es stellt den aktuellen Punkt dar, an dem sich der Ablauf befindet. Das heisst, jeder Prozess als Aktion mit mind. einem Token beginnt und mit mind. einem Token endet.
Da die Token in den AD als Zustände auch im Sinne eines Wertes gebraucht werden, z.B. <if <Get Next Quote> [hits entry] then <OK>>, die zu der Aktion <New Position> führt, ist

[4] Als Ersatz für Datenflussdiagramme nur bedingt geeignet

auch hier die Idee eines Überganges gegeben, nämlich von <Get Next Quote> zu <New Position> in Abb. 3.9. Zudem lassen sich die Token an **Bedingungen** knüpfen, die mit Synchronisationsbalken verknüpft werden oder auf die nächsten Aktionen Einfluss nehmen können. Meistens ist ein Token zum Weiterfliessen sowieso an eine Bedingung gebunden.

3.2.3.3 Synchronisationsbalken

Paralleles Verhalten kann aber muss nicht synchronisiert werden. Bei einer Abhängigkeit dient dazu der Synchronisationsbalken als Kontrollknoten, der die einzelnen Bedingungen oder Token aufnehmen kann. Ein einfacher Synchronisationsbalken bedeutet, dass der ausgehende Token erst in Aktion tritt, wenn alle eingehenden vorhanden sind. Dies entspricht einer **UND-Bedingung**.
In unserem Beispiel in Abb. 3.9 wird das <Calc Statistics> erst möglich, wenn die Bedingungen der Aktionen <New Position> und <Close Quote> erfüllt sind. Wenn z.B. <Close Quote> nicht OK ist, wird evtl. von vorne begonnen oder ein weiterer Kontrollfluss gewartet (Semantisch: die Statistik soll neue wie geschlossene Positionen zeigen).

3.2.3.4 Mengenverarbeitung

Eine weitere Parallelitätsquelle stellt das Auslösen einer Mengenverarbeitung <expansion region> oder {stream} bei einem Objektfluss dar. Im Beispiel Abb. 3.9 wird für jedes Instrument (Wertpapier) die zugehörige Abfrage der Kurse und deren Positionen aufbereitet. Es lassen sich auch asynchrone Parallelitäten modellieren, indem die ausgehenden Token nicht in einen gemeinsamen Balken münden, d.h. die Token müssen nicht aufeinander warten (n-Endknoten).

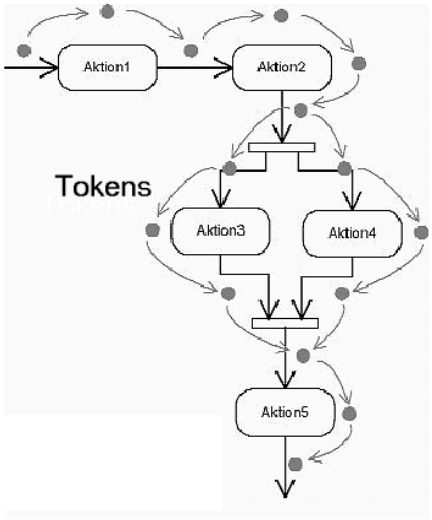


Abb. 3.10: Synchronisation mit Mehrfachtrigger

In Abb. 3.8 haben wir als Variation ein Rückwärtstoken eingebaut, der bei einem nicht gültigen Pfad einen Default Pfad setzt. Am Anfang muss <Default> und <Selektion> als gültige Bedingung vorliegen, damit die Token **feuern** und durch die Synchronisation die beiden Aktionen <Pfad bestimmen> und <Report aufbereiten> aktivieren können. Nun lässt sich immer noch der Default Pfad übersteuern, bei nicht gültigem Pfad feuert der Token aber wieder mit einem Default Pfad zum Synchronisationsbalken hin. Mit diesem Konstrukt lässt sich übrigens auch ein Deadlock simulieren, denn wenn der Default Pfad nicht stimmt, kann ich keinen eigenen Pfad mehr bestimmen, da es sich ja beim Balken um eine UND-Bedingung handelt!

3.2.3.5 Selektion

Ein weiteres Element von AD ist die Entscheidungsraute (Kontrollknoten), die uns in unserem Beispiel die Beschreibung beliebig verschachtelter Entscheidungen erlaubt. Die erste Entscheidung in Abb. 3.7 des AD <Buchen> betrifft das Checken der Limite. Sollte die Bezugslimite überschritten sein, zweigt die Aktion ab, d.h. wir gelangen zur zweiten Entschei-

dung betreffend der Frage, ob der Kunde auf der Blacklist ist oder nicht. Wenn ja wird die Buchung abgewiesen. Der Geschäftsprozess modelliert somit, auch ein Kunde der auf der Blacklist steht, hat noch die Chance, einen minimalen Betrag zu überziehen.

> *Sequenz, Selektion, Iteration und Subroutine, d.h. die Atome einer Programmiersprache, sind in einem AD abbildbar...*

Auch ODER-Bedingungen lassen sich modellieren. In die Aktion <Buchung abweisen> führen zwei Trigger, welche eine ODER-Bedingung darstellen. Entweder wird die Zahlung nicht autorisiert oder der Kunde ist auf der Blacklist. (UML 2.0 explizit markieren).

3.2.3.6 Verknüpfungen

Genaugenommen können wir es auch mit einer XOR-Bedingung zu tun haben, da die Bedingung [auf Blacklist] und [Zahlung nicht autorisiert] sich gegenseitig ausschliessen können. Somit wäre es falsch, wenn beide Bedingungen eintreten, maximal eine genügt. Die XOR-Notation in einem AD entspricht nicht der Version 2, wir halten sie aber für notwendig. Erweiterungen der Syntax in UML lassen sich mit sogenannten Stereotypen oder <{constraints}> vornehmen.

3.2.3.7 Partitionen

Bei beiden Modellen ist ein eigentlicher Datenfluss nicht vorhanden. Durch die Erweiterung um Datenobjekte und Organisationseinheiten lässt sich die semantische Aussagekraft sicher erhöhen. Das Hinzuziehen von Datenobjekten ermöglicht es aufzuzeigen, auf welche Daten eine Aktion wirkt. Bei den Organisationseinheiten hat UML vorgesorgt. Standardmässig kennen die AD sogenannte Verantwortlichkeitsgrenzen <swimlanes>, die beschreiben wer was in den Aktionen tut. D.h. die AD müssen in durch Linien abgetrennte vertikale Bereiche aufgeteilt sein.

> *Jeder Bereich, durch eine <swimlane> markiert, repräsentiert dann die Verantwortlichkeit einer Klasse oder einer Abteilung, je nach Modellperspektive.*

Wenn sich die Visualisierung aber durch die Umstellung zu kompliziert erweist (d.h. damit wir senkrechte Linien im AD erhalten können), sollte man manchmal eine Unterteilung in nicht gerade verlaufende Linien vorziehen. Eine andere Möglichkeit wäre, die Aktionen mit einer Farbe gemäss den verschiedenen Klassen oder Abteilungen einzufärben. Eine Erweiterung mit Datenobjekten ist bei den AD grundsätzlich nicht nötig (ausser Objektknoten), da hierzu andere Diagramme in UML diese Aufgabe objektorientiert übernehmen. [i][FS98]

> *In der OO-Welt werden grundsätzlich keine Datenflüsse modelliert, da man davon ausgeht, die einzelnen Elemente wie Aktionen oder Klassen kommunizieren untereinander via Nachrichten mit Parameterübergabe und greifen dann autonom mit Methoden auf die benötigten Daten zu.*

3.2.4 Zusammenfassung AD

Aktionen und Token lassen sich mit Verknüpfungsoperatoren verbinden, die mit <UND>, <ODER>, und <XOR> bestimmte Abläufe an Bedingungen knüpfen. Die einzige Restriktion besteht darin, dass Token selbst keine Entscheidungen treffen können, will heissen, nur nach Aktionen oder Knoten sind Entscheidungen erlaubt.

- UND entspricht dem Synchronisationsbalken mit ein- und ausgehenden Token
- ODER entspricht der Abbildung, wenn mehr als ein Token direkt in eine Aktivität einfliesst
- XOR lässt sich zusätzlich als Stereotyp einführen, wenn max. eine Bedingung genügt

Da die Token meistens an eine Bedingung gebunden sind, lassen sich auch mehrere, von einer Aktion ausgehende Token, steuern.
Ein AD läßt sich als Kontrolle der einzelnen Bedingungen auch mit einer Entscheidtabelle darstellen. Ein zusätzliches Feature an die Tool Hersteller, indem aus einem AD eine solche Entscheidtabelle generiert wird. Folgend sehen Sie das AD <Buchen> als Entscheidtabelle:

Bedingung	Regel A	Regel B	Regel C	Regel D
[Betrag] OK	Ja	Ja	Ja	Ja
Limite [OK]	Ja	Ja	Nein	Nein
Autorisiert [OK]	Ja	Nein	-	Ja
Auf [Blacklist]	-	-	Nein	Ja
Aktivität				
Buchung abweisen		X		X
Bonität ändern			X	
Transaktion	X			

Tab. 3.6: Vom AD zur Entscheidtabelle

3.3 Class Diagram

Nun kommen wir zum Stolz von UML, dem Design von Klassen. Ein Objekt kann ja eine Methode eines anderen Objekts aufrufen, auch wenn es das Objekt einer anderen Klasse ist. Damit ein solcher Botschaftsfluss realisierbar ist, müssen die beteiligten Klassen einander „kennen“. Solche Beziehungen zwischen Klassen, welche die Nachrichten erst ermöglichen, lassen sich im statischen Klassendiagramm (CD) darstellen, denn diese Beziehungen sind dauerhaft, ob Nachrichten fliessen oder nicht.
Graphisch lassen sich diese Beziehungen, auf **Instanzenebene** als Assoziation oder als Aggregation festlegen und auf **Klassenebene** als Generalisierung im CD bestimmen. Diese zeigen alle zentralen Aspekte des statischen Modells im Überblick: die Klassen mit ihren Methoden und Eigenschaften sowie die Beziehungen zwischen den Klassen. Doch aller Anfang ist erst einmal die Klasse zu finden, die meisten Bücher schweigen sich da aus und kommen jeweils mit den fertigen Diagrammen daher.

3.3.1 Klassenfindung

Gibt es ein Leben nach dem Code ?

Der Unterschied zwischen Komponenten- und Anwendungsentwicklern besteht darin, dass die ersteren neue Klassen bauen, während letztere Instanzen von Klassen erzeugen, bearbeiten und einsetzen. Als Ausgangslage zur Klassenfindung dient uns der UC und die AD. Als Ziel wollen wir die wichtigsten UC in Klassen abgebildet sehen.
Eine **Klasse** ist im Grunde genommen ein **Typ**. Als Entwickler arbeiten Sie immer mit Typen und Instanzen (auch dann, wenn Sie diese Begriffe nicht benutzen). Sie deklarieren bspw. Variablen eines bestimmten Typs, z.B. vom Typ `Currency`. Klassen sind normalerweise schwieriger als Datentypen, funktionieren aber auf ähnliche Weise. Durch Zuweisen unterschiedlicher Werte an Instanzen desselben Typs können Sie unterschiedliche Aufgaben mit denselben Methoden ausführen.

3.3.1.1 Entwurfsmetaphern

In der OO-Welt ist nicht alles neu, was neu erscheint. Bestehende Klassendesign gibt es zuhauf, nur finden muss man sie. Lassen Sie sich von Design Patterns,[ii] Libraries oder Komponenten inspirieren. Den Design Patterns widmet sich ein eigenes Buch.[iii] Meistens sind Klassen nur nicht dokumentiert und werden demzufolge auch nicht verwaltet. Ab und zu werden Klassen-Templates mit generischen Parametern angelegt.
Sie dienen als Muster, nach dem Sie nicht-generische Klassen anlegen können.
Sind effektiv keine bestehenden Strukturen vorhanden, sind Unterlagen über Patterns aus bestehenden Projekten ein Hort der Kreativität.
Beispiel «Builder» als Design Pattern mit folgender Formvorschrift:

> *Trenne die Konstruktion eines komplexen Objekts von seiner Repräsentation, so dass derselbe Konstruktionsprozess unterschiedliche Repräsentationen zur Laufzeit erzeugen kann.*

In unserem Workshop gehen wir noch näher auf diese Trennung ein.

> *Es sollte auch bekannt sein, was für Code bereits vorhanden ist und welche Anforderungen er löst. Es braucht also wie gesagt eine gut dokumentierte Bibliothek, um das Rad nicht immer neu zu erfinden.*

Doch aufgepasst: Bei existierendem Code können Namenskonflikte (Name Spaces) auftreten oder bestehende Teile gestört werden infolge fehlender Kapselung.
Um Klassen wiederverwenden zu können, sollten sie möglichst wenig mit anderen Klassen kommunizieren, denn je weniger eine Klasse von Ihrer Umgebung weiss, desto universeller ist sie einsetzbar (Autonomie). Testen Sie auch bereits bestehende Komponenten mit Source Code um ev. ähnliche Lösungen oder Ansätze zu finden. Wissen wo wissen ist, ist die Devise.

```
type // als wrapper pattern
    TLinkEdit = class(TEdit)
    private
    protected
      FDataLink : TFieldDataLink;
```

```
    function GetDataSource: TDataSource;
    procedure SetDataSource(Value: TDataSource);
  public
    constructor Create(AOwner: TComponent ); override;
    destructor Destroy; override;
  published
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
```

3.3.1.2 Textanalyse

Betriebsinterne Vorgaben und Geschäftsprozesse lassen sich notieren oder bestehende Gesetze , Verordnungen oder Verträge nach Wörtern analysieren. Jeder Betrieb hat so seine Weisungen und Richtlinien dokumentiert. Haben Sie ein Wort – meistens ein Substantiv – gefunden, das eine Klasse treffend bezeichnen könnte, können Sie aus dem Wort eine provisorische Klasse erstellen.
In einem Tool, funktioniert das z.B. folgendermassen: Markieren Sie das Wort, und klicken Sie auf die entsprechende Schaltfläche in der Symbolleiste. Es öffnet sich daraufhin ein Dialog, in dem Sie den Namen der neuen Klasse, ihr Stereotyp und ihre Packagezugehörigkeit bestimmen können.

> *Ein Stereotyp ist ein Mittel der Modellierungssprache, um immer gleiche Zusammenhänge auszudrükken. Stereotype als Erweiterung der Sprache, lassen sich dann anderen Beschreibungselementen zuordnen.*

Das markierte Wort in der Beschreibung wird kursiv, um es als Basis für eine Klasse zu kennzeichnen, und die neue Klasse wird in die OO-Datenbank aufgenommen. Sie wird nun vom Tool genauso konsistent gehalten und in den entsprechenden Fenstern angezeigt wie alle anderen Klassen auch.

3.3.1.3 Gebrauch von CRC-Karten

Das Arbeiten mit CRC-Karten ist eine relativ einfache und wirkungsvolle Technik (nach Kent/ Beck / Cunningham) für das Auffinden von Klassen (**Class**), um deren Aufgabe (**Responsibility**) und Beziehungen zu anderen Klassen (**Collaborations**) darzustellen.
Für jede Klasse verwendet man eine Karte aus Papier, auf der man die gefundenen Ergebnisse notiert um sie später in Gruppenszenarien durchspielen zu lassen.

Klasse: TKonto	Abstrakt: nein
Oberklasse: TBusObj	
Unterklassen: TKreditkonto, TStockkonto	
Verantwortung, Engagement:	Zusammenarbeit, Mitwirkung
GetData	TStringList
Buchen	TTransaktion
GetKreditLimit	
SetKreditLimit	
CollectData	TStringList

Tab. 3.7: CRC aus der Basisklasse TKonto

Bei geeigneter Wahl der Klassen- und Methodennamen lassen sich die Abläufe wie in der Umgangssprache beschreiben und sind somit auch für Nichtfachleute verständlich. Die Methodennamen dienen der **Identifikation** der zu lösenden Problemen. Jedes Objekt steht in Beziehungen zu anderen Objekten. Die Beziehungen charakterisieren die Nachrichten, welche die Objekte untereinander senden / empfangen können. Auf der rechten Seite der Karten stehen die Kollaborateure, die wiederum andere Karten mit Klassen bezeichnen.

> *Die CRC-Technik (Class, Responsibility and Collaboration), wie sie von Kent, Beck und Ward Cunninggham: A Laboratory For Teaching Object-Oriented Thinking, OOPSLA '89 Proceedings, beschrie-*

ben wird. Die Idee beruht auf dem Entwurf, den man durch Karteikarten oder Memo-Zettel repräsentiert. Darauf notiert man sich die Klassennamen (Class), Methoden (Responsibility) und Beziehungen (Collaboration). Die Beziehungen sind eigentlich auch Methoden, die sich jedoch über andere Klassen aufrufen lassen oder mit ihnen zusammenarbeiten.

Solche Szenarien bauen vor allem auf dem interaktiven Spiel einer Gruppe auf, wo jedes Team-Mitglied eine oder n-Karten in der Hand hält und versucht, Verantwortungen (also Methoden) unter Mitwirkung anderer Klassen zu erfüllen.
Bspw. benötigt meine Klasse eine Parser-Methode einer anderen Klasse, um HTML zu erzeugen. D.h. Verantwortlichkeiten brauchen unter Umständen die Mitwirkung anderer Objekte, damit sie erfüllt werden können. Dies trifft sicher bei Beziehungen zu, bei anderen Operationen sofern notwendig. Bei den mitwirkenden Klassen ist es übersichtlicher, sie auf der **gleichen** Zeile auf der CRC-Karte zu notieren, auf der die entsprechende Verantwortlichkeit steht.
Hier noch ein pragmatisches Ablaufschema:

1. Anforderungen / Ziele analysieren
2. Texte markieren oder Metaphern suchen
3. CRC mal einfach so ausfüllen, zuerst eigene Methoden
4. Beziehungen festlegen (probeweise) mit fremden Methoden
5. Prüfen: Rollen spielen, evtl. Karten tauschen und sich gegenseitig aufrufen
6. Prüfen: Vollständigkeit anhand von Szenarien oder Reviews

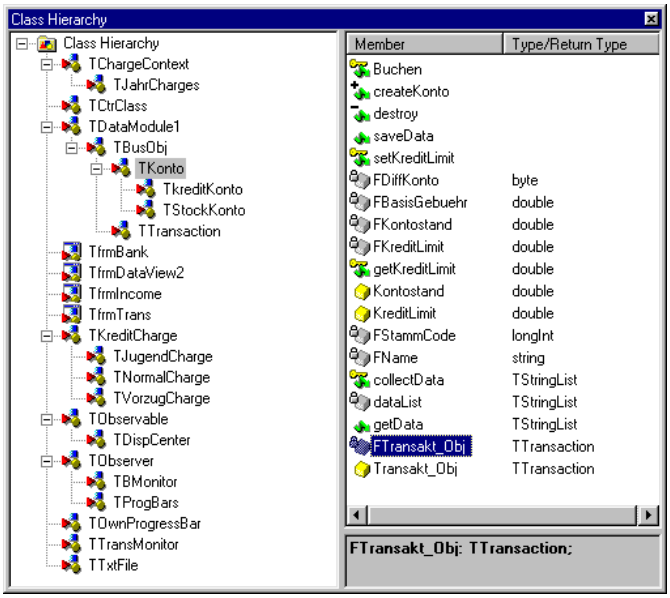


Abb. 3.11: Klassen von BROKER

Nachdem wir nun wissen, wie man Klassen findet, hier ein Inventar der gefundenen Klassen. In Abb. 3.11 finden Sie eine Übersicht der Klassen, die im BROKER zum Einsatz kommen. Klassen die sich in externen DLL's befinden, erscheinen später bei den Packages. Beginnen wir mit der Notation exemplarisch mit den Klassen `TBusObj` und `TKonto`, die mit anderen Klassen zwei erste Diagramme ergeben.

3.3.2 Notation

Simon & Forunkel: BIT OVER TROUBLED DATA
Lassen Sie uns mit dem vorliegenden Klassendiagramm beginnen (Design or not design, ist die Frage), dass die Vorstufe zur Klasse `TKonto` und `TChartGen` bedeutet. Das Diagramm in Abb. 3.12 stellt die Datenbasis für die Chartklassen dar. Sie erben die Methoden, die das Datenmodul in Delphi zur Verfügung stellt. Bevor wir zur Codierung schreiten, hier vorerst ein Einstieg in das Klassendiagramm mit der Vererbung.
Klassen und Interfaces erhalten neu einen <part>, das ist ein Teil einer Klasse. Dieses Teil ermöglicht in einer frühen Phase den Typ einer Klasse zu bestimmen, der dann später in der Implementierung mit Operationen und Attributen ausgearbeitet wird.

3.3.2.1 Die Klasse

Eine Klasse wird mit einem Rechteck symbolisiert, das mit dem entsprechenden Namen beschriftet ist. Wir arbeiten weiterhin mit ModelMaker, nein dies soll kein Werbeblock sein, sondern auf die enge Bindung von Tool und Notation aufmerksam machen. Jedes Tool hat da so seine Feinheiten. Ist die Klasse von einem anderen Stereotyp als dem <Class>-Stereotyp, läßt sich auch der Name des Stereotyps, in spitze Klammern setzen.
In unserem Beispiel haben wir den Stereotyp <Service> gleich in den Namen `TChartVisitor` eingebaut, das für **ServiceObject** steht. Die Klasse ist somit spezifisch instanzierbar; wenn die Schrift des Klassennamen aber kursiv ist, handelt es sich um eine abstrakte Klasse *`TChartMember`*.
Gemäss UML muss eine Klasse mindestens drei Unterteilungen in der Notation enthalten: Klassenname – Eigenschaften – Methoden.
Zusätzliche Attribute wie Felder und Events lassen sich den Eigenschaften und Methoden zuordnen. In Abb. 3.12 ist die Vererbung mit einem Pfeil zur Oberklasse dargestellt, d.h. `TElementChartMA` erbt von `TChartMember`, die Leserichtung entspricht also der Aufrufrichtung.

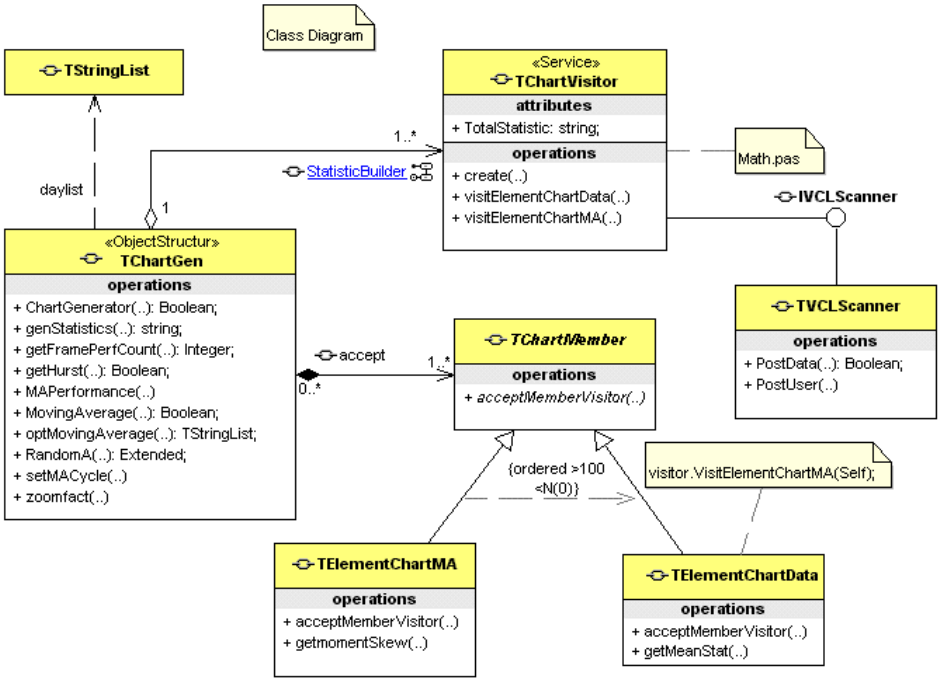


Abb. 3.12: Ein CD mit zweifacher Vererbung

3.3.2.2 Vererbung und Polymorphie

Objekte sind in sich geschlossene Einheiten. Durch diese Unabhängigkeit ist es möglich, dem gleichen Methodennamen für verschiedene Objekte zu verwenden. Jedes Objekt weiss bei Erhalt einer Nachricht (durch die VMT), dass seine Methode `self` gemeint ist. Die Notation für die Vererbung ist ein Pfeil; ohne Vererbung keine Polymorphie.

> *Als Definition zur Polymorphie: Eine identische Nachricht wie `Buchen` aus der Klasse `TKonto` kann von verschiedenen Instanzen aus anderen Klassen (Exemplaren) durch verschiedene Methoden, die aber immer gleich heissen, umgesetzt werden.*

Realisierung: Mit dem Überschreiben von geerbten Methoden. Nutzen: Flexible Erweiterung von Applikationen ohne dass man bestehenden Code ändern muss, indem einfach die Methoden mit dem gleichen Namen erweitert werden (siehe `Buchen`).

3.3.2.3 Beziehungen und Multiplizität

Die Navigationsrichtung ist definiert. Eine Assoziation ist die einfachste Beziehung zwischen zwei Klassen. Bei einer Assoziation ist deren Verhältnis am häufigsten <1..*>. In einer Assoziation kann auch eine Klasse die Multiplizität 0 besitzen, weil es ausreicht, wenn nur die

Instanz einer der beteiligten Klassen die Instanz der anderen Klasse kennt, die aber noch nicht instanziert ist. Wir sehen gleich, wie das konkret funktioniert.

> *Als Multiplizität findet man auch das Wort Kardinalität, das soviel wie Auftretenshäufigkeit heisst und im englischen als multiplicity übersetzt ist. Daher die zwei Wörter die dasselbe bedeuten.*

Es folgen die sechs Arten von Beziehungen inklusive Notation zwischen Klassen, die wir gleich im Klassenbau näher vorstellen:

1. <**generalisation**> hier versteht man die eigentliche Vererbung von Methoden und die Polymorphie, die meist vom Allgemeinen als Basisklasse zum Besonderen läuft und mit einem abgeschlossenen Pfeil notiert wird `TElementChartData`.
2. <**association**> eine Beziehung zur Laufzeit mit loser Kopplung zwischen Objekten (Instanzenbeziehung), z.B. ein DLL-Aufruf oder Dialoge, temp. Strukturen, in Delphi und C++ pointer, wird mit einem offen Pfeil dargestellt.
3. <**aggregation**> besitzt die Klasse ein Objekt (enge Kopplung) zur Designzeit (Klassenbeziehung), z.B. `TButton` in `TForm`, im Konstruktor, in Delphi in C++ member, als offene Raute bei Klasse `TKonto` notiert, d.h. die Klasse `TChartGen` aggregiert die Klasse `TChartVisitor`.
4. <**composition**> ist stärker als <aggregation> und meint physik. Einbindung, z.B. `memo.lines.add`, meistens in Komponenten mit existenzabhängigen Klassen zu finden und als schwarz gefüllte Raute notiert `TChartMember`.
5. <**realisation**> eine Beziehung zur Designzeit, die das Interface mit der zu realisierenden Klasse aufzeigt. Die realisierende Klasse hat eine gestrichelte Linie mit geschlossenem Pfeil zum Interface oder zur abstrakten Klasse. Alternativ ist auch ein Lollipop (offener Kreis) darstellbar `IVCLScanner`.
6. <**dependency**> eine reine Typisierungsbeziehung, entweder man ist abhängig von einem Typ, z.B. einen Exception oder man braucht eine Standardklasse wie `TStringList`, die im Fachmodell nichts zu suchen hat, aber trotzdem ins Modell einfließen soll. Wird mit einer gestrichelten Linie und offenem Pfeil dargestellt.

Die Aggregation kommt häufig vor: Eine Teil-Klasse wie `TChartVisitor` oder `TTransaction` gehört zu genau einem Aggregat wie `TChartGen`, und ein Aggregat kann aus keiner, einer oder vielen Teil-Klassen bestehen. Der zulässige Wertebereich reicht von 0 bis 32767, höhere Zahlen lassen sich in einem Diagramm durch das <*>, das in UML vom ERD her ein <N> bedeutet, darstellen und wir auch schon vom AD her als <multiplicity marker> kennen.

3.3.2.4 Rollen zwischen Klassen

Rollen repräsentieren potentielle Beziehungen zwischen den Klassen. Zusätzlich zur Multiplizität, die UML für Aggregationen und Assoziationen vorsieht, läßt sich auch die Rolle einer jeden Klasse sichtbar machen, wenn auf der Multiplizität der Klasse eine gegenseitige Rolle vorhanden ist.
Als Beispiel einer Rolle dient eine Klasse Person die 1..1 zur Klasse Immobilien als Bewohner wirkt aber auch 1..* zu Immobilien als Eigentümer wirkt. Die Klasse Person hat also zwei verschiedene Rollen zur **gleichen** Klasse. Es sind auch Rollen zu **verschiedenen** Klassen denkbar, z.B. eine Person hat zur Klasse Auftrag die Rolle des Kunden, zur Klasse Abteilung aber die Rolle des Angestellten.

3.3.3 Klassenbau im BROKER

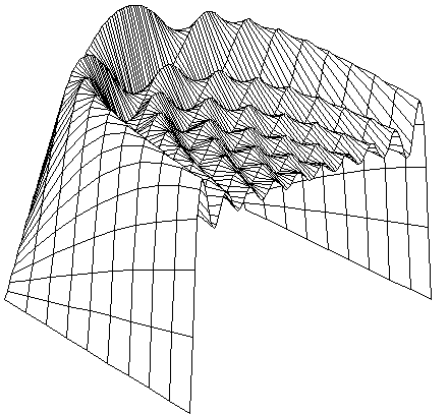


Abb. 3.13: Klassen als strukturiertes Verhalten

Lassen Sie uns mit dem Bau der Klassen beginnen. Wir starten gemäss Abb. 3.14 mit den Kontenklassen. Ein Klassendiagramm enthält im Prinzip Klassen, die in einem fachlichen oder technischen Zusammenhang stehen, die sich mit den Beziehungen und Rollen verdeutlichen lassen. Zu diesem Zeitpunkt, d.h. in der Designphase sind erste Gedanken zur Architektur einzubringen.

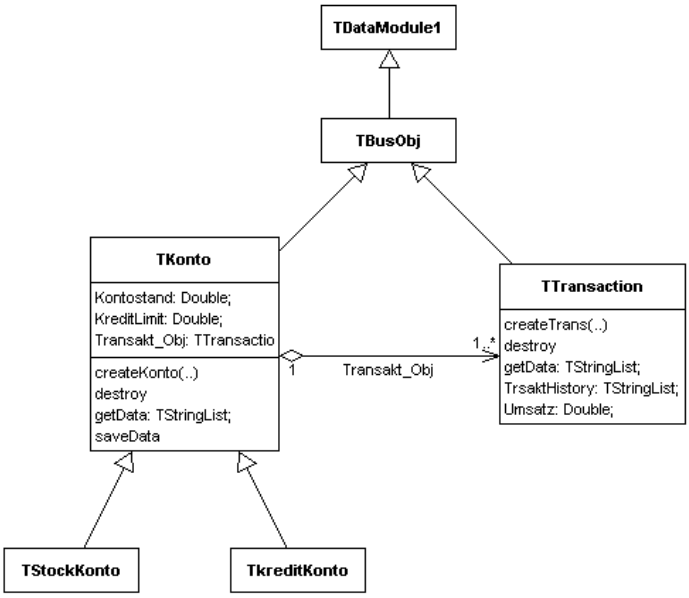


Abb. 3.14: Erste Beziehungen zwischen den Klassen

Eine erste Grobeinteilung mündet in der Frage: „Welche Klassen gehören zusammen in ein Package“, denn bei adäquater Einteilung nach fachlichen Kriterien entspricht am Anfang ein Klassendiagramm ungefähr einem Package, wobei sich Packages dann noch verschachteln lassen. Weiteres zu Paketen und Units folgt später.
Nachdem die Klassen durch die erwähnten Techniken der Klassenfindung gefunden wurden, bestimmen wir das nähere Verhalten und die Architektur gemäß der folgenden Schritte, die aber hochgradig iterativ ablaufen. Einerseits nehmen wir einen Teil der Packages vorweg, andrerseits baut man Beziehungen auf, ohne die volle Implementierung zu kennen, die mit den Sequenzdiagrammen und möglichen State Event pro Klasse dann verfeinert wird.

3.3.3.1 Einteilung in Units (Packages) vornehmen

Das Document (Model), Controller, View (DCV) Prinzip ist eine Technik zur Verteilung der

Funktionalität auf bestimmte Klassen, das auch als Vorbereitung zu mehrschichtigen (multi-tier) Klassen/Komponenten dient: Verarbeitungsdominante Klassen realisieren Kontrolle und Steuerung von 1 oder n UC und werden auf entity- oder control-Klassen verteilt, die man auch <business class> nennt.
Dialogdominante Klassen sollten wenig „Wissen“ enthalten und sind in sogenannten boundary-Klassen zu finden. Sie entsprechen dem View. Es geht also darum, die Klassen in die entsprechenden Units schichtenartig zu verteilen. Im BROKER wurde mal folgende Aufteilung in die ersten 4 Units vorgenommen (Abb. 3.15):

- `TDataModule1` und `TBusObj` in der Unit *bankData* (enthält Zugriff auf InterBase und die Exportschnittstelle)
- `TKonto` und `TTransaction` und weitere Verarbeitungsklassen in der Unit *bankLogic* (Verarbeitung und Business Rules)
- `TCtrClass`, `TObserver` und weitere Controller in der Unit *bankControll* (vermittelt zwischen dem View und der Logik)
- `TfrmBank` , `TBMonitor` und weitere Viewklassen in der Unit *bankView* (die Darstellung der Dialoge und Forms)

Nachdem diese Einteilung erfolgt ist, beginnt man sich den einzelnen Klassen zu widmen, ähnlich während eines Umzuges, wo die Kisten mal in die zweckbestimmten Zimmer geworfen werden und sodann sich Zimmer für Zimmer einrichten lassen. Dass dabei ab und zu eine Kiste (Klasse) das Zimmer (Unit) wechselt liegt auf der Hand respektive im Hirn.

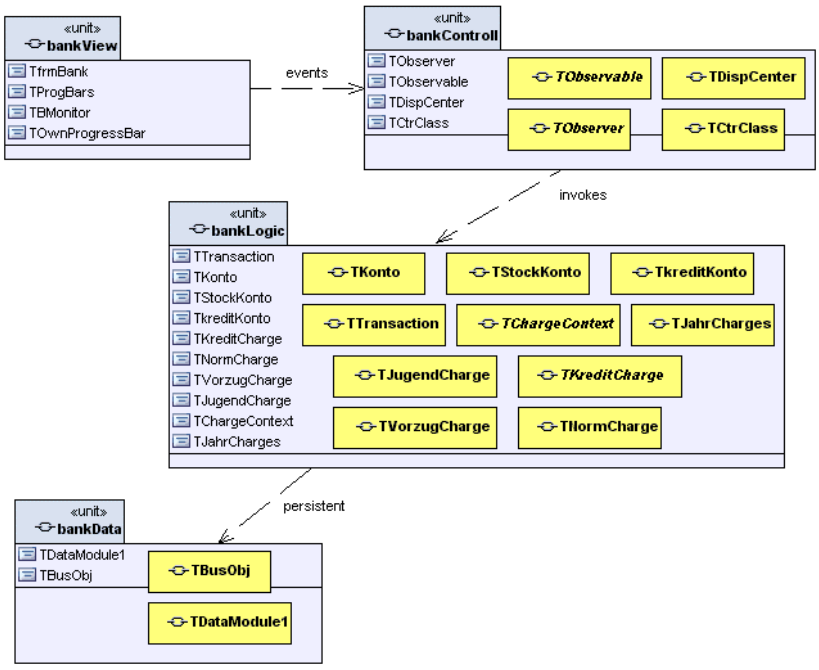


Abb.3.15: Die Klassen in den Units

3.3.3.2 Sichtbarkeiten und Zugriffsmethoden definieren

Alle Attribute von Klassen, einschließlich der Felder, Methoden und Eigenschaften befinden sich auf einer bestimmten Schutz- oder Sichtbarkeitsebene. Diese Sichtbarkeiten gilt es nun zu bestimmen. Inwieweit der Zugriff auf die Attribute und Methoden erlaubt ist, können Sie an der Kennzeichnung (ab UML 1.1) erkennen:

Öffentlich	<public>:	< **+** >
Geschützt	<protected>:	< **#** >
Privat	<private>:	< **-** >
Paket	<package>:	< **~** >

Meist werden Sie Methoden in Klassen oder Komponenten als <public> oder <protected> deklarieren. Nur selten ist das Sichtbarkeitsattribut <private> nötig, das bewirkt, daß nicht einmal abgeleitete Klassen ausserhalb einer Unit Zugriff erhalten. Am meisten läßt es sich kombiniert mit properties einsetzen. Deklarieren Sie Attribute als <private>, wenn sie nur in

der Klasse verfügbar sein sollen, in der sie auch definiert werden.

3.3.3.3 Beziehungen zwischen den Klassen festlegen

Weiter läßt sich die Kommunikation zwischen den Klassen wie Vererbung (Polymorphie), Aggregation und Assoziation erstellen und „relationships“ zu Klassen einbinden. Gedanklich sind diese Beziehungen meistens schon bei der **Klassenfindung** erarbeitet worden, werden aber in dieser Phase stets neu hinterfragt und schrittweise implementiert. Denn während das statische Modell im CD die Beziehungen abbildet, beschreibt das spätere dynamische Modell im SEQ, wie diese Beziehungen sich aktiveren, d.h., welche Botschaften entlang den Instanzen fließen.
Eine Methode umfaßt nur in den seltensten Fällen die Ausführung einer einzigen Funktion. Wesentlich häufiger sind verkettete Aufrufe mehrerer Methoden und Kaskadierungen möglich, die aber nicht in derselben Klasse angesiedelt sind. Somit wird klar, daß diese Phase eben stark iterativ geprägt ist und einer wiederholten **Selbstprüfung** nichts im Wege stehen sollte.

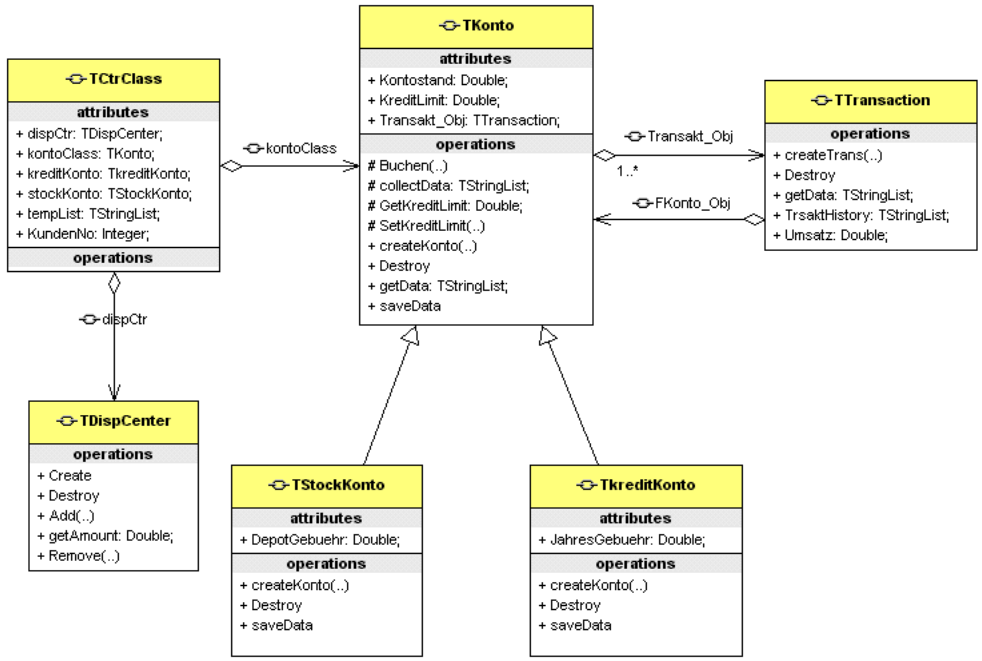


Abb. 3.16: Weitere Methoden im CD dazugekommen

Gehen wir weiter in Richtung der ersten Implementierung der Klassen. Den lauffähigen Code finden Sie mit allen Units und Files in der Zip-Datei (s.S. 8). Angenommen, Sie entwickeln eine einfache Kontenverwaltung, bei der sich Kundenkonten führen lassen. Hier läßt sich die Klasse `TKonto`, durch Vererbung (auch Generalisierung genannt) spezialisieren. Das Wertschriften- und das Kreditkartenkonto erben vom Basiskonto `TKonto`, welches wir als Sparkonto einsetzen wollen (Abb. 3.16):

```
TKonto = class(TBusObj)
TStockKonto = class(TKonto)  //Wertschriften
TKreditKonto = class(TKonto) //Kredite, Hypotheken
```

3.3.3.4 Kapselung durch Zugriffsmethoden

Angenommen Sie haben eine Eigenschaft `KreditLimit` eingebaut. Die Anforderung wird nun dahingehend erhöht, daß jede Änderung der Kreditlimits eine **Autorisierung** mit einem Passwort erfordert. Mit einem normalen Feld oder einer mehrfach vorhandenen Funktion wären Sie gezwungen, an allen Stellen im System die entsprechende Validierung einzufügen. Nicht so mit einer Eigenschaftskapselung, die einfach und sicher ist:

```
  Protected
    function getKreditLimit: double;
    procedure setKreditLimit(newLimit: double);
  public
    property KreditLimit: double
         read getKreditLimit write setKreditLimit;
```

Hier ist wieder das eigentlich Feld `FKreditLimit` geschützt im privaten Bereich, jeder Zugriff muß also zwangsläufig über die Eigenschaft `KreditLimit` erfolgen, die bei Lesen oder Schreiben die entsprechende Methode z.B. `setKreditLimit` feuert:

```
procedure TKonto.setKreditLimit(newLimit: double);
  begin
    if FKreditLimit <> newLimit then begin
      if getPassword(frmTrans.password) then begin
        FKreditLimit:= newLimit
      end else MessageDlg(frmTrans.strLit[2]+'  '+   //multilang
                          frmTrans.strLit[3], mtWarning,[mbok],0);
    end;
  end;
```

Zunächst wird beim versuchten Schreiben eines neuen Kreditlimits ein Passwortdialog aufgerufen (der übrigens von einer DLL stammt), der bei erfolgreicher Autorisierung das private Feld `FKreditLimit` ändert. Ein direkter Zugriff auf das Feld ist somit nicht mehr möglich und die Methode ist künftig erweiterbar z.B. mit einer Datumskontrolle, ein Kontosperrflag oder einer Limitenberechnung. Auch dient diese Technik als Bedingung zum Komponentenbau mit dem `published` Attribut.

3.3.3.5 Relationen im BROKER

Kommen wir nun zu den Beziehungen zwischen den Klassen und deren Interpretation. Beziehungen sind die einzige Möglichkeit, um überhaupt via **Nachrichten** mit anderen Klassen oder Komponenten zu **kommunizieren**.
Assoziation bedeutet, daß einander bekannte Objekte gegenseitig Methoden zur Laufzeit aufrufen können, aber nicht füreinander zuständig sind. Es heißt auch, daß ein Objekt ein anderes lediglich in einer losen Kopplung kennt.
Aggregation wiederum bedeutet, daß ein Objekt in der Regel schon zur Designzeit ein anderes Objekt besitzt oder dafür zuständig ist. Die Aggregation führt dazu, daß das eingebettete Objekt und sein Besitzer gleich lang leben.[iv]

3.3.3.5.1 Vererbung (Generalisation) und Polymorphie:

```
  TKonto = class(TBusObj)
  TStockKonto = class(TKonto)
  TKreditKonto = class(TKonto)
```

Bei der Polymorphie sind die bereits bekannten Methode `Buchen` und `collectData` vielgestaltig. Das heißt, daß die Methode `Buchen` in allen Klassen gleich lautet, aber je nach Aufruf unterschiedlich reagiert. Es läßt sich also automatisch die richtige Methode zur Laufzeit finden. Dies ermöglicht die `self`-Variable der Virtual Method Table (VMT).
Virtuelle und dynamische Methoden lassen sich im Gegensatz zu statischen Methoden in abgeleiteten Klassen überschreiben. Beim Aufrufen einer überschriebenen Methode bestimmt nicht der deklarierte, sondern der **aktuelle Typ** (also Typ zur Laufzeit) der im Aufruf verwendeten Klassen- bzw. Objektvariable, welche Implementierung von z.B. dem Einsammeln der Daten zwecks Darstellung (`collectData`) aktiviert wird:

```
TKonto
protected
    function collectData: TStringList; virtual;
TStockKonto = class(TKonto)
    function collectdata: TStringList; override;
```

```
TKreditKonto = class(TKonto)
    function collectData: TStringList; override;
```

Je nach Konto wird also zur Laufzeit entschieden, welchen Umsatz man via dem Sammeln von Daten abfragen möchte. Auch bei der Methode `Buchen` wird erst zur Laufzeit festgelegt, auf welches Konto man buchen möchte.
Die Klassen sind übrigens völlig unabhängig von einem Form- oder GUI-Objekt (siehe uses-Anweisung), die Daten zur Darstellung werden deshalb in einer Stringliste gesammelt und dann gemeinsam übergeben. Jede Klasse greift **autonom** auf die Daten zu. Somit läßt sich die Methode `collectData` auch für andere Ausgaben wie Reports, Printer oder EMail verwenden und die Konten sind nicht fix an eine Darstellung gebunden. Der eigentliche Aufruf von außen erfolgt über die `getData`-Methode:

```
function TKonto.getData: TStringList;   //virtual container
  begin
    result:= collectData;   //is polymorphic
  end;
```

Nun sind wir in der Lage, die Methoden `Buchen` und `collectData` in der Basisklasse und den beiden abgeleiteten Klassen auch noch als `protected` zu deklarieren. Hinzu kommt noch die Methode `saveData` als letzte virtuelle Methode:

```
Protected
  procedure Buchen(betrag: double); override;
  function collectData: TStringList; override;
..procedure saveData: override;
```

3.3.3.5.2 Assoziation

Die Assoziation läßt sich gemäß der Definition schnell erklären. In der Methode `SetKredit-Limit` haben wir bereits Bekanntschaft mit dem Passwortdialog gemacht:

```
function GetPassword(const PassWord: string):boolean; far;
   external 'Password.dll';
function SelectDir(dest: PChar): boolean; far;
   external 'Password.dll';
```

Hier erfolgt mit einer losen Kopplung ein Aufruf der DLL zur **Ladezeit** (Loadtime). Sie können auf die Routinen einer DLL auch während der Laufzeit über direkte Aufrufe von API-Funktionen zugreifen (z.B. `LoadLibrary`, `FreeLibrary` und `GetProcAddress`, die in der Delphi-Unit Windows deklariert sind). Noch dynamischer wird es während der Aktivierung eines simulierten Transaktionsmonitors in unserem Bankenbeispiel, bei dem das Fenster mit den darzustellenden Daten während der Laufzeit erzeugt wird (auch eine Assoziation):

3.3.3.5.3 Aggregation

Eigentlich sind die verschiedenen Controls, die eine Form beherbergt, schon Grund genug zur Annahme, hier handele es sich um Aggregation. Denn wird das Form gelöscht, verschwinden auch die eingebetteten Controls. Hier liegt bei Delphi die Verantwortung bei der Eigenschaft `Owner`. Mit dieser Eigenschaft können Sie den Eigentümer einer Komponente ermitteln.
Wir aber wollen in unserem Beispiel noch eine „echte“ Aggregation erschaffen, nämlich eine Klasse, die für den **Tagesumsatz** und Kontoauszug verantwortlich ist. Die Klasse `TTransaction` stellt Protokoll-Dienste zur Verfügung,[v] so daß jede Buchung in die Tabelle <TRANS> geschrieben wird und sich mit Summen ein „Umsatz“ ermitteln läßt:

```
TTransaction = class(TBusObj)
  private
    FPosten: integer;
    FTurnoverAvg: double;
    dataList: TStringList;
  protected
```

```
    FKonto_obj: TKonto;
    procedure SetTransactionHist(betrag: double);
  public
    constructor createTrans(konto: TKonto);
    destructor destroy; override;
    function Umsatz: double;
    function TrsaktHistory: TStringList;
    function getData: TStringList;
  end;
```

Die Trennung in eine eigene Klasse wird bewußt in Kauf genommen. Die Klasse läßt sich künftig als eine Art Service Provider einsetzen, da auch sie autonom auf die Daten von InterBase zugreift. Wie kommt es nun zur Aggregation? Die Verbindung der beiden Klassen läßt sich über eine Objektreferenz regeln, die wir von der Basisklasse aus im Konstruktor instanzieren:

```
constructor TKonto.createKonto(cust No: longInt; std Acc: byte);
...........
    dataList:= TStringList.create;
    FTransakt Obj:= TTransaction.createTrans(self);
  end;
```

Hiermit erbt auch jede Unterklasse diese Instanz, so daß der Umsatz pro Kontoklassen auch schon steht. Der Konstruktor der Transaktionsklasse erhält im Gegenzug eine Referenz vom Objekt `TKonto`. Möglich wird das mit der Variablen `self`, die eine Referenz auf die Klasse darstellt. Denn bei einer Objektreferenz erhält `self` als Wert die Klasse des betreffenden Objekts, aus dem **aufgerufen** wird. Somit ist die Transaktionsklasse imstande, Methoden der Kontoklasse aufzurufen, indem `self` gleich nach der Übergabe einer eigenen Variablen im Konstruktor zugewiesen wird:

```
constructor TTransaction.createTrans(konto: TKonto);
  begin
    FKonto_obj:= konto;  //aggregation
    FPosten:= 0; //Tagesumsatz
    FMonat:= 1;
    dataList:= TStringList.create;
  end;
```

So dass beide Objekte jetzt voneinander wissen und gegenseitigen Zugriff erlauben.
Diese Technik nennt man übrigens in der Pattern Sprache ein **Strategiemuster**, wenn als Parameter gleich ein Objekt übergeben wird. Hier läßt sich nun die enge Kopplung der beiden Klassen zur Laufzeit erahnen, fast könnte man von einer Symbiose sprechen, da beide Objekte ja fast gleich lang leben (bis das der Destruktor euch scheidet):

3.3.3.5.4 Composition

Eine Komposition, die eine noch stärkere Kopplung als die Aggregation bedeutet und die man in der Regel bei Komponenten findet. Das entscheidende Merkmal einer Komposition ist, dass die aggregierten und existenzabhängigen Teile nie mit anderen Objekten geteilt werden! In der Unit *bankView* befinden sich zwei Kompositionen. Die `TMemo`-Komponente beherbergt eine `Lines`-Eigenschaft die wiederum ein `TStrings`-Objekt darstellt, so daß die `TStrings`-Methoden (z.B. add) für `Lines` in `TMemo` verwendet werden können:

```
//da lines von Typ TStrings ist und TMemo lines mit der Methode add stark bindet
//haben wir eine composition
   memMon: TMemo;
   memMon.lines.add(floattoStr((dispCenter as TDispCenter).getAmount));
```

3.4 State Event

Zustandsdiagramme (SE) beschreiben die Sicht auf das dynamische Verhalten der im Klassendiagramm definierten statischen Objekte. Jedes Zustandsdiagramm ist **einer** Klasse zu-

geordnet und beschreibt deren Verhalten genauer. Ist also eine Klasse interessant genug, kommt sie in die Ehre sich in einem State Event zu verewigen. Ein SE kann also nie mehreren Klassen gehören.
Zustandsdiagramme sind jetzt strikter von den Activities (AD) semantisch getrennt, sonst erhalten sie die gleiche Bedeutung als Zustandsmaschine und bei Klassen mit extensiven Zustandswerten:

> *Bei den AD sind die <Token> vor allem an Bedingungen <condition> gebunden, während ein <event> zwischen zwei Zuständen <state> in den SE keine Bedingung haben muss.*

Kommen wir zum Hauptunterschied. Beim AD wird die **Aktivität als Zustand** betrachtet, beim SE steht meistens das **Ergebnis als Zustand** im Vordergrund. Das SE untersucht die möglichen Zustände und Übergänge unabhängig eines zeitlichen Ablaufes. Somit empfehle ich ihnen, Zustände im Diagramm als **Partizip** zu formulieren, wie **Kredit gecheckt** und nicht Kredit checken.
Auch die Sicht ist eine andere, ich meine die Perspektive (Point of View) und Detaillierung; sind es beim SE die konkreten Zustände eines instanzierten Objektes, sind es beim AD vor allem die Fachprozesse innerhalb einer technischen Organisation.

3.4.1 Einsatz

Pro Klasse kann es genau einen Objektlebenszyklus geben, der sich in einem Zustandsdiagramm abbilden lässt, denn das vom Zustand abhängige Verhalten der Instanzen ist immer dasselbe. Wichtig ist vor allem:
Je nach Zustand kann eine Klasse anders auf **dasselbe** Ereignis reagieren.
Nun betreten wir kurz das Feld der Philosophie und überlegen uns: Wie im täglichen Leben gibt es ein Ereignis Schulterklopfen. Wenn ich den Zustand nervös habe, reagiere ich auf Schulterklopfen anders als beim Zustand entspannt.
Fall 1: <entspannt> → Schulterklopfen <überrascht>
Fall 2: <überrascht> → Schulterklopfen <erschreckt>
Fall 3: <erschreckt> → Schulterklopfen <wütend>

Genau das ist es, was uns die Väter der SE mitteilen wollten, so glaube ich wenigstens. Wir kommen auf die Welt, füllen mit der Zeit unseren „Methodenrucksack“ und sind dann fähig auf Ereignisse unterschiedlich zu reagieren. Warum: weil wir uns Zustände merken können.
Ein SE kann auf folgende Fragen Antwort geben:

- Auf welche Ereignisse reagiert ein Objekt?
- Wie reagiert ein Objekt und welche Aktivitäten lassen sich auslösen, wenn es sich in einem bestimmten Zustand befindet?
- In welchen gültigen Folgezustand geht das Objekt in Reaktion auf das Ereignis über (welche Übergänge in Automaten sind möglich)?

Das gesamte System und damit jede Instanz einer Klasse befindet sich zu jedem Zeitpunkt in einem wohldefinierten Zustand (jedenfalls tut ein artiges Objekt so was). Durch eine äussere Einwirkung (Ereignis) kann sich der Zustand eines Objektes ändern. Das Ereignis kann somit eine kurze Transition (Übergang vom Zustand in einen anderen) auslösen. Es können externe **und** interne Ereignisse sein.
Anläßlich einer OO-Schulung wurde mir die wohl beste Definition eines Ereignisses bewußt, die ich je gehört hatte:

> *«An event is an important change in state»*

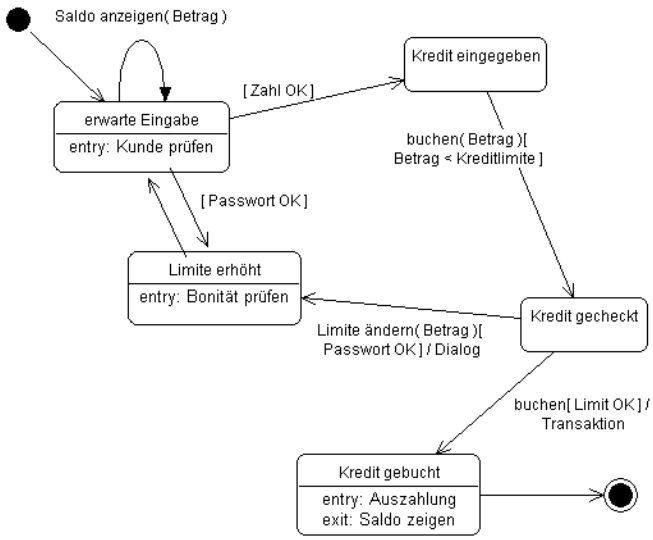


Abb. 3.17: State Event mit möglichen Zuständen der Instanz

In der Abb. 3.17 sehen wir ein erstes konkretes SE der Klasse `TKreditKonto` mit den Ereignissen <buchen> und <Limite ändern>. Nicht jeder Zustand muss gespeichert sein, die meisten Zustände sind Teil einer Validierung (true oder false) entlang einer Zeitachse, und diese Validierung tritt immer wieder in Aktion:

```
procedure TKreditKonto.buchen(betrag: double);
  begin
  if betrag > 0 then
    inherited Buchen(betrag)
  else if Bonitaet(betrag) then //bonitaet, a method from class
    inherited Buchen(betrag)
    else MessageDlg(frmTrans.strLit[10], mtWarning,[mbok],0);
  end;
```

Der eigentliche Zustand <Kredit gecheckt> finden wir in der folgenden Funktion `Bonitaet`, welche die Kreditlimite auf Überziehen überprüft, und dann den Zustand erreicht. Ein Zustand muss **nicht** wahr oder falsch sein, dazu dienen die Bedingungen <guard> zwischen den Zuständen, welche die Übergänge regeln. Erst nach <Kredit gecheckt> kann ich den Zustand <Limite erhöht> oder <Kredit gebucht> erreichen.

```
function TKreditKonto.Bonitaet(betrag: double): boolean;
  begin
    if (KontoStand - -Betrag) >= -FKreditLimit then begin
     result:= true end else result:= false;
  end;
```

3.4.2 Notation

Verbunden mit dem Eintreffen eines Ereignisses sind Aktionen, die sowohl beim Zustandswechsel über eine Transition als auch beim Akzeptieren eines Ereignisses innerhalb eines Zustands (die dann Aktivitäten heissen) durchgeführt werden können.
Sie sehen, innerhalb eines Zustandes läßt sich auch eine Aktivität ausführen, wie beim Zustand <erwarte Eingabe> wo nebenbei die Aktivität <Kunde prüfen> läuft.
In Abb. 3.18 durchleuchten wir die Klasse `TChartGen`, die mit dem Eingehen von Positionen an Börsen zu tun hat.

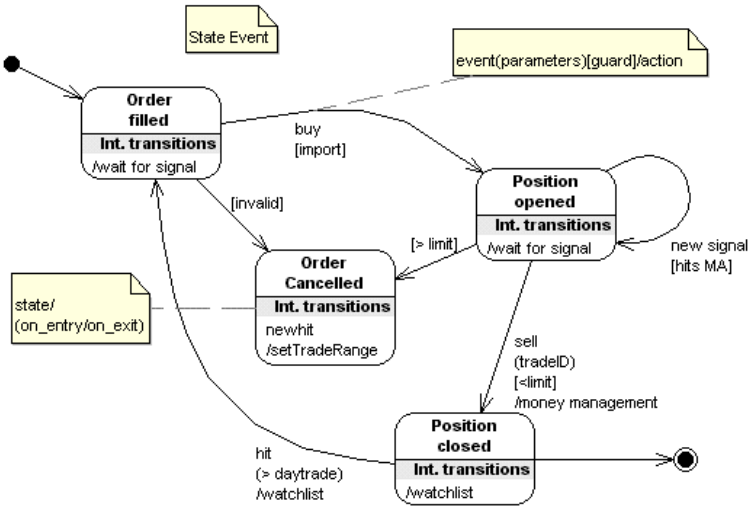


Abb. 3.18: Das SE der Klasse TChartGen

3.4.2.1 Zustand

Ein Zustand beschreibt die momentane Wertemenge der Attribute oder ein Zeitzustand als Aktivität oder Ergebnis eines Objekts. Auf die Antwort eines Objekts auf ein eintreffendes Ereignis kann eine begleitende Aktivität und / oder ein Zustandswechsel erfolgen. Der Pfleil auf den Zustand <opened> gerichtet, läßt sich wie folgt in Abb. 3.18 lesen:
Beim Eintreten des Ereignis <buy> **auf den Zustand** <Order filled> erfolgt ein Wechsel zum neuen Zustand <Position opened>, sofern ein [import] möglich war.
Die Pfeilrichtung zeigt also nicht das Eintreten, sondern den Zustandswechsel an. Weiter im Text: Auf den Zustand <opened> tritt das Ereignis <sell> ein und <opened> wechselt in den Zustand <Position closed>, sofern die Bedingung [<limit], erfüllt ist. Das Event <sell> erzeugt also einen bedingten Zustandswechsel von <opened> nach <closed>.

> *Wie schon gesagt, Zustände haben meistens als Bezeichnung ein **Partizip** (gebucht, reserviert, gecheckt, closed etc.), d.h. das Ereignis oder die Aktion hat bereits stattgefunden.*

In jedem seiner Zustände kann ein Objekt noch eine Aktivität ausführen, die einen Zeitraum in Anspruch nimmt. Aktivitäten sind eine Untermenge der regulären Zustände. Aktivitäten lassen sich auch zur Modellierung einzelner Schritte in der Ausführung eines **Algorithmus** verwenden. Wenn z.B. auf den Zustand <destroyed> das Ereignis <create> eintritt, wird in den Zustand <instanziert> gewechselt, der in seinem Zustand noch die Aktivität <assign> ausführt, wie folgender Code verdeutlicht:

```
constructor TTxtFile.Create(Name : TFileName);
begin
   inherited Create;
   TTextRec(FTextFile).mode := fmClosed;
   FDefaultExt := 'TXT';
   if Length(Name) > 0 then Assign(Name)
end;{Create}
```

> *Aktivitäten sind im Gegensatz zu den Aktionen, die im Übergang stattfinden, nicht atomar, sie lassen sich also während der Ausführung unterbrechen.*

Im Zustandsdiagramm sind normale Zustände als leicht abgerundete Rechtecke dargestellt, die optional einen Namen tragen und mit einer horizontalen Linie in Zustände und Aktivitäten getrennt sind.

3.4.2.2 Ereignis / Aktion

Im Zusammenhang mit den Ports im Component kann ein SE in der Version 2 auch angeben, in welcher Reihenfolge man die Methoden eines Ports benutzen kann.
Eine Transition ist ein Zustandswechsel, der sich durch ein Ereignis aktivieren lässt. Die

Darstellung der Transition erfolgt in Form einer geschlossenen, gerichteten Linie. Der Pfeil zeigt vom aktuellen Zustand zum **Zielzustand**. Die Bezeichnung an der Transition ist vor allem der Name des Ereignisses, das die Transition bewirkt.

> *Wenn die Beschriftung einer Transition kein Ereignis enthält, sondern nur eine Bedingung (<guard>), ist damit ein Zustandswechsel gemeint, der beim Beenden einer Aktivität fast automatisch eintritt. Warum fast, weil eben die Bedingung erfüllt sein muss.*

In eckigen Klammern kann außerdem eine Bedingung angegeben werden, die erfüllt sein muss, damit die Transition feuert. Wenn also in Abb. 3.18 die Bedingung [<limit] nicht erfüllt ist, ist der Zustand <Position closed> unerreichbar.

> *Beim Zustandswechsel ist es möglich, sowohl eine Aktion durchzuführen als auch ein weiteres Ereignis oder Signal zu versenden, das sogenannte Zustandsautomaten anderer Klassen akzeptieren. Hierfür läßt sich auch eine Zielklasse angeben.*

Die Syntax einer Transition kann zusammenfassend folgende optionale Elemente enthalten, wobei mindestens ein Ereignis oder eine Bedingung vorhanden sein muss:
Ereignis (parameter) [Bedingung] / Aktion(parameter)
Eine typische solche Aktion ist das einfache Senden eines Signals an ein anderes Objekt oder das Validieren eines Wertes während des Überganges.
In 3.19 wird buchen()[>limite]/sperren() validiert, die Zustände sollten <Guthaben gesetzt> und <Schulden aktiviert> heissen.

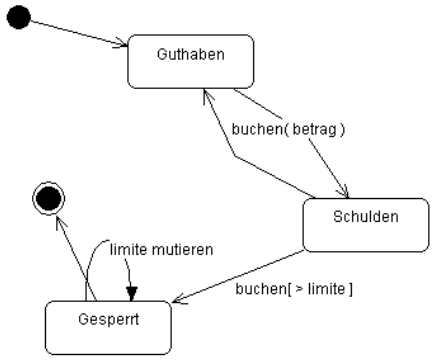


Abb. 3.19: Mögliche Fachzustände der Klasse TKonto

3.4.2.3 Start- und Endzustand

Wir haben kurz den Zustand <instanziert> einer technischen Klasse gestreift. Initiale Zustände lassen sich generell in objektorientierten Sprachen durch Konstruktoren erzeugen. Verschiedene Konstruktoren bzw. unterschiedliche Parameter können dabei Objekte in unterschiedlichen Startzuständen herstellen.

- Ein Anfangszustand wird durch einen kleinen Kreis dargestellt, der optional beschriftet sein kann, um verschiedene Anfangsbedingungen anzuzeigen.
- Ein Endzustand ist als Kreis mit innerem Kern dargestellt. Er kann beschriftet sein, um Endbedingungen zu unterscheiden. Das Erreichen des Endzustands repräsentiert den Abschluss der Aktivitäten und Zustände in der umgebenden Semantik und beendet die Lebensdauer eines Objektes im Idealfall.

3.5 Interaction Diagrams

„Ruf der Wildnis“ Elvis Presley: IN THE GOTO
Grundlegend gibt es von den Interaction Diagrams 4 Ausprägungen:

- Sequenzdiagramm und Kommunikationsdiagramm
- Interaktionsübersicht und Zeitdiagramm

Wir behandeln vor allem die Sequenzdiagramme, die ein wichtiges Werkzeug in der Implementierung aber auch in der Analyse sein können. Wir setzen SEQ gemäss unserem Vorgehensmodell in der Implementierungsphase ein, dadurch läßt sich das Zusammenspiel mit dem Prototyp konkreter testen.
SEQ dienen in dieser Phase der Verfeinerung und besseren **Parametrisierung** des Code. Die Entwicklung eines SEQ kann auch helfen, später noch Methoden einer Klasse zu finden, das Zusammenwirken verschiedener Klassen zu modellieren und die Steuerung des gesamten Systems zu entwerfen.

3.5.1 Bedeutung

Das SEQ beschreibt das Zusammenwirken verschiedener Instanzen oder Objekte für einen bestimmten Anwendungsfall. Es definiert somit die interne Sicht eines UC, beschreibt also, wie sich der UC innerhalb der Anwendung realisieren läßt. Unser SEQ realisiert den UC <Show ChartData> aus Abb. 3.4. Der Nachrichtenaustausch zwischen den Instanzen ist hier in zeitlicher Reihenfolge sichtbar. Und das ist die große Stärke und zugleich das Einsatzgebiet eines SEQ. Die zeitliche Reihenfolge der Methoden (Nachrichten) entspricht ihrem Erscheinen in vertikal absteigender Richtung innerhalb des Diagramms und läßt sich somit dynamisch darstellen. Die Zeitachse verläuft vertikal!
Zwischen Sequenz- und Kommunikationsdiagramm (früher Kollaborationsdiagramm) besteht ein semantischer Zusammenhang, so dass eine Teilmenge darstellbar ist. Die Notation und das Einsatzgebiet der beiden Diagramme differiert aber.
In der Regel gilt als Entscheid:

- Sequenzdiagramm bei wenig Klassen und vielen Nachrichten mit zeitlicher Dynamik
- Kommunikation bei wenig Nachrichten und vielen Klassen mit Patterns
- Timing Diagram bei Echtzeit

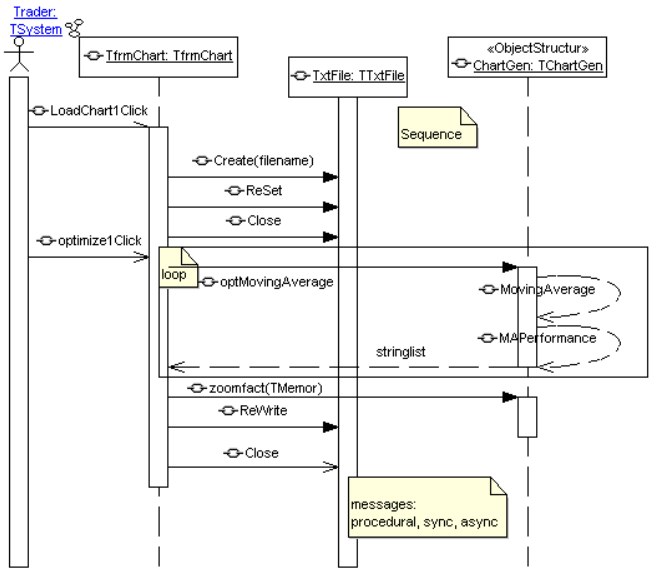


Abb. 3.20: Sequenzdiagramm aus dem UC <Show ChartData>

3.5.2 Relation zu Analyse und Design

Sequenzdiagramme sind auch eine wichtige Technik der Anforderungsanalyse. Die Arbeitsschritte, die ein Anwendungsfall umfasst, lassen sich in diesem frühen Stadium schon durchdenken. Zugleich können Sie eine Grundlage für das Design legen.

Aus dem modellierten Aufrufkaskaden ergeben sich benötigte Klassen und Methoden. Dieses Spiel entsteht bei uns schon während der Klassenfindung mit den CRC-Karten, wo ja eigentlich ein SEQ durchgespielt wird, da jeder den anderen aufruft, wenn er eine Methode oder ein Ergebnis benötigt.
Von der Semantik her ist es richtig, von den Use Cases in die AD umzubrechen, doch je nach Perspektive und Abstraktion kann man auch in einem ersten Schritt direkt vom UC zu einem SEQ gelangen. Die Klassen und bei erhöhtem Reorganisationsbedarf auch die AD, können später folgen. Solche Entscheidungen sind ja im Vorgehensmodell festgelegt.
In d.R. werden also Sequenzdiagramme in der Designphase eingesetzt: Sie realisieren, nebst den AD, eine dynamische Sicht auf den Entwurf. Während CD die statischen Beziehungen zwischen Klassen abbilden, stellen Sequenzdiagramme den Botschaftsfluss zwischen Instanzen dar, der für die Erfüllung einer fachlichen oder technischen Aktivität notwendig ist. Auf diese Weise überprüft man mit der Erstellung von SEQ auch die Korrektheit der Klassendiagramme.

3.5.3 Aufrufkaskaden und Kopplung

Aggregation und Assoziation sind natürliche Wege für Botschaften der Klassen
Verlassen wir kurz die Diagramme, d.h. wir erweitern den Prozess, um die Voraussetzungen und das Einsatzgebiet für ein SEQ besser zu verstehen. Wir befinden uns ja inmitten der Implementierungsphase und da sollten Sie noch mehr Einblick in den BROKER erhalten.
Prozeduren und Funktionen sind vom übrigen Code möglichst unabhängig implementiert. Dennoch bezieht fast jede Prozedur oder Funktion gewisse Werte von aussen und ändert Variablen der restlichen Anwendung.

> *Die Prinzipien der modularen Programmierung verlangen, dass all diese externen Werte und Variablen als Parameter einer Schnittstelle in die Routine übergeben werden, so dass die Parameterliste im Routinenkopf einen vollständigen Überblick über den Austausch von Daten zwischen Programm und Routine liefert.*

Hier ist ObjectPascal auch einmalig mit der **Trennung** Interface und Implementierung.
Ein solches Vorgehen ist eine wichtige Vorbedingung dafür, dass mehrere Personen speditiv und effizient am gleichen Projekt arbeiten können, und es erhöht die Wartbarkeit von Anwendungen.

3.5.3.1 Parametrisierung

Parameter übergeben Daten, das heisst Werte, Konstanten und Variablen, an Prozeduren oder Funktionen und liefern veränderte Werte aus den Routinen zurück an den aufrufenden Code. Exakt das Einsatzgebiet eines SEQ.
Die Implementierung einer Prozedur oder Funktion legt die Liste der formalen Parameter fest. Der Aufruf beinhaltet dann die Liste der aktuellen Parameter, die in Zahl und Typ den formalen Parametern entsprechen müssen.

```
procedure TCtrClass.setKundenNo(custNo: integer); //formaler Parameter
begin
  if custno > CLIENT BASE then FCustNo:= custNo;
end;
```

Jeder Parameter, der in einer Liste der formalen Parameter deklariert ist, ist lokal zu der Prozedur oder Funktion in der er deklariert wird.

- Benennen Sie die Routinen vorsichtig, am Besten mit einem aussagekräftigen Verb gefolgt von einer Beschreibung
- Bauen Sie Routinen mit **starker Kohäsion** und **loser Kopplung**, d.h. mit flexiblen Beziehungen zu anderen Methoden
- Vermeiden Sie globale Variablen und festverdrahtete „Konstanten“

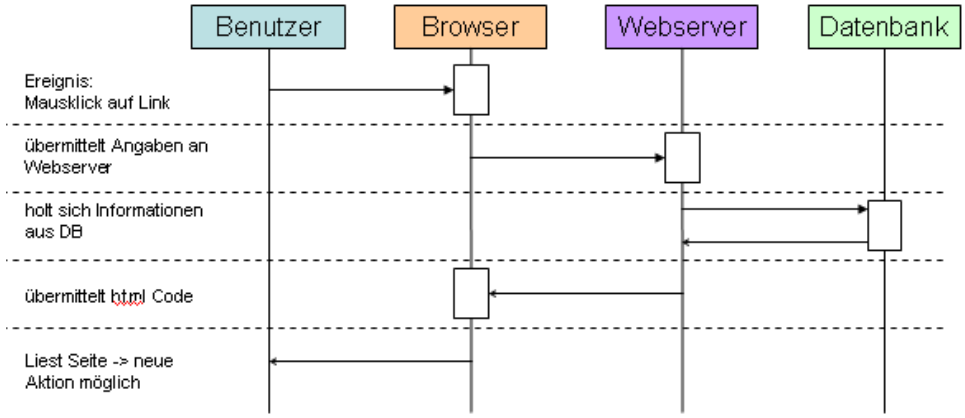


Abb. 3.21: Ein SEQ im Allgemeinen

Die Methode `SetTransactionHist` ist so ein Fall einer starken Kohäsion mit loser Kopplung, da bis zum Ausführen der StoredProcedure auf dem InterBase Server die Kontrolle behält <focus of control>, aber flexibel den Aufruf auch delegieren kann:

```
procedure TTransaction.SetTransactionHist(betrag: double);
begin
 if frmTrans.boolLogSave then begin
   if frmTrans.chkBox_SP.Checked then
     setSP_TransData(betrag, FKonto_Obj.FStammCode) else
     setSQLTransData(betrag, FKonto_Obj.FStammCode);
  end;
 inc(FPosten);  //für Tages-Umsatz
end;
```

Die Variante in der Prozedur dient dazu, die gleiche Aufgabe mal an eine StoredProcedure oder an eine lokale SQL-Anweisung zu delegieren, zudem läßt sich die Aufzeichnung der Transaktionen aus Testzwecken inaktivieren (bei ca. 200 Transaktionen/Minute). Lose Kopplung meint auch, dass der Aufruf einer Methode nicht wieder einen Rückruf benötigt.

3.5.4 Notation

Mit einer Systemgrenze grenzen Sie den Ausschnitt ab, den Sie mit den Instanzen und Botschaften des SEQ darstellen, ab. Die Systemgrenze ist am **linken** Rand zu finden und ist meistens ein Ereignis eines UC. Lässt sich auch direkt, Abb. 3.23, mit einer Instanz beginnen.

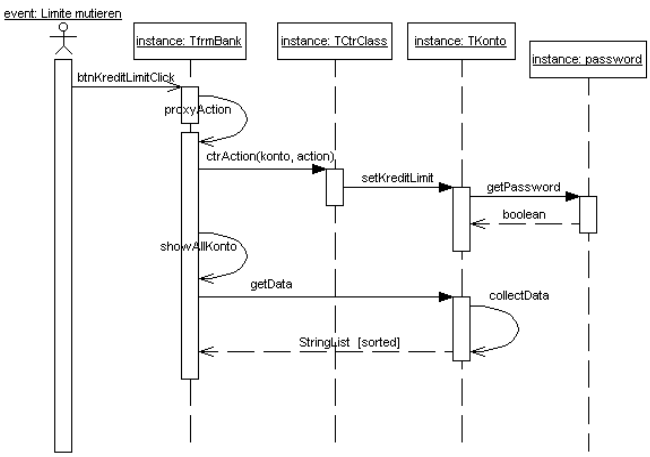


Abb. 3.22: Das SEQ Kreditlimite mutieren

3.5.4.1 Die Instanzen

Eine Instanz (ein Objekt) ist eine interne Einheit des zu modellierenden Systems, die an der

Abarbeitung eines Anwendungsfalls mitwirkt. Sie läßt sich durch eine vertikale gestrichelte Linie <lifeline> darstellen. Oberhalb der Linie werden Klassen- und Objektname in einem Rechteck in der Form <instance:classname> angezeigt, sofern beide schon existieren. Ist nur einer dieser Einträge vorhanden, so wird er allein (mit Doppelpunkt) angezeigt.
Im SEQ habe ich mir die Mühe erspart, die Instanznamen anzugeben, da meistens nur eine Instanz existiert (also keine Exemplare) und bei **Instanzen in Listen** als Zeiger verwaltet eh keine Namen vorhanden sind.

> *Wenn ein Objekt innerhalb der im Diagramm dargestellten Sequenz zerstört wird, läßt sich dies durch eine <destruction> (Schrägkreuz) am Ende der letzten Aktivierung kennzeichnen. Dort endet die Lifeline des Objekts.*

Die Lifelines der Objekte, die nicht im Diagramm zerstört werden, setzen sich über das Ende der letzten Aktivierung hinaus fort. Meistens bleiben die Objekte erhalten, wie in Abb. 3.22 die Instanz von `TfrmBank`. Diese Instanz kennt zwei Lifelines, eine kurze und eine Lange, wobei die Lange bis zum Schluss den <focus of control> behält (bei seq. Aufruf):

```
procedure TfrmBank.btnKreditLimitClick(Sender: TObject);
  begin
    proxyAction(1, edtBetrag.text);
    showAllKonto;
  end;
```

Die Methode `proxyAction` ist eine Selbstaktivierung, die mit einem reflexiven Pfeil diese Tatsache untermalt. Dann behält die Methode `showAllKonto` bis zum Schluss der <lifeline> die Kontrolle der Sequenz:

3.5.4.2 Aktivierungen

Eine Aktivierung als <focus of control> beschreibt den Zeitraum, in dem ein Objekt im Rahmen des modellierten UC oder Ereignis eine oder mehrere Methoden ausführt oder eher ausführen läßt, entweder direkt oder durch Aufruf einer untergeordneten Prozedur. Eine Aktivierung wird demnach durch ein schmales nicht ausgefülltes Rechteck dargestellt, das auf der vertikalen Linie der Instanz <lifeline> verläuft.

> *Beim rekursiven Aufruf eines Objekts innerhalb einer existierenden Aktivierung wird die zweite Aktivierung leicht rechts versetzt zu der ersten angezeigt. Sie wird dann als <child> bezeichnet.*

3.5.4.3 Gates

Ich komme zu den <gates>, also zu den Toren. <gates> Operatoren erlauben <alt>(Verzweigung), <loop>(Schleife), <break>(Ende), <opt>(Optional), <par>(Threads), <ref>(Verweis) Ein -> Ausstieg

> *Auch hier ist die Idee des Diagramms auf andere Sequenzen, wie eine Subroutine, verweisen zu können. Es gleicht einem GOTO, kommt aber noch besser, auch Iterationen und Selektion sind neu möglich. Sequenzen lassen sich dann zerlegen und so auch mehrfach referenzieren.*

Der Filezugriff in Abb. 3.20 ist ein Kandidat für eine eigene Sequenz, die sich dann mit einem Hyperlink, oder eben <gates> verknüpfen läßt. Der Abruf der einzelnen Berechnungen für die Charts ist als Loop dargestellt. Weiterhin ist auch ein Asterix * als Iterator möglich.
Damit diese Ein- und Ausstiegspunkte definiert sind, hat man die <gates> ins Leben gerufen. Zusätzlich hat man mit den erwähnten <gates> die Möglichkeit, Rücksprungbefehle abzufangen, sofern man den Namensraum eines Sequence verläßt um Schnittstellen zu anderen Diagrammen, wie ein Component, definieren zu müssen.

3.5.4.4 Nachrichten

Durch die Funktion der Objekte und durch das Prinzip der Kapselung ergibt sich ein Kommunikationsfluss zwischen den Instanzen. Eine Nachricht ist nichts anderes als ein Aufruf einer Instanz an eine andere Instanz, mit der Bitte, eine Methode auszuführen. Die aufrufende Instanz muss natürlich im Besitze der anderen Instanz sein, ausser es handelt sich um eine Klassenmethode.
Die Nachricht besteht in der Regel aus drei Teilen: aufrufende Instanz, auszuführende Methode und benötigte Parameter:

Instanz.Methode(parameter);

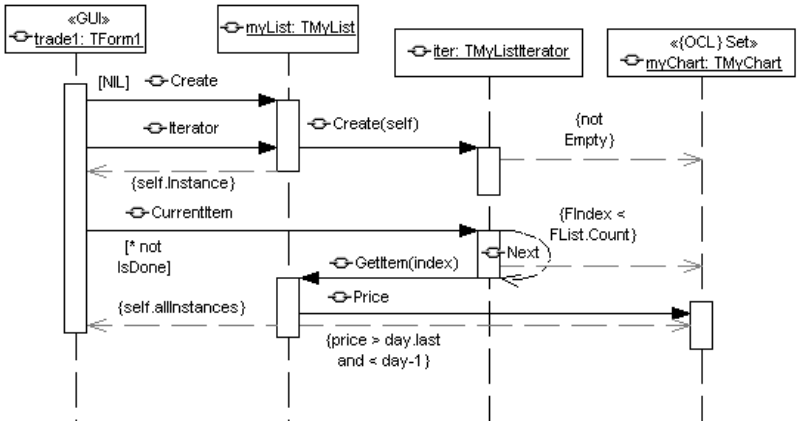


Abb. 3.23: Nachrichten als Aufruf von Methoden

Mit Nachrichten kontrollieren die Objekte den Verlauf in einem SEQ. Sie bestehen aus einem Call, der von einer Instanz im SEQ zu einer anderen Instanz gesendet wird. Eine Nachricht läßt sich durch einen Pfeil darstellen, der vom Objekt, das die Nachricht sendet, zu dem, das sie empfängt reicht.

3.5.4.5 Nachrichtenarten

Zum Schluss noch eine Verfeinerung des Nachrichtenpfeiles. Bis anhin haben wir verschwiegen, wie die Nachricht denn behandelt wird. Kann der Aufrufer auf die Beantwortung warten oder nicht? Es werden zwei Arten von Nachrichten unterschieden, synchrone und asynchrone.

- **Synchrone**: Die Nachricht wird vom Empfänger angenommen und vollständig verarbeitet, erst dann darf der Absender weitermachen, der häufigste Fall. Sequentielle Nachrichten werden als Pfeile mit voller Spitze dargestellt.
- **Asynchrone** Nachrichten werden dagegen gesendet, ohne auf eine Antwort zu warten. Meistens landen die Nachrichten in einer Warteschlange des Empfängers, wann er die Nachricht annimmt, interessiert den Absender nicht (Pfeil mit leerer Spitze).

Nachrichten und Returns können zwischen Objekten, Aktivierungen und der Umgebung verlaufen. Dabei gelten folgende Einschränkungen von Nachrichten an Objekte, die Sie beim Erzeugen und Neuverbinden beachten sollten:

- Objekte können nur Ziele von Nachrichten sein.
- Als Quelle einer Nachricht kommen nur Aktivierungen anderer Objekte oder die Umgebung als Ereignis in Frage.
- Von einer Aktivierung darf nicht mehr als ein Return ausgehen.
- Ein Return ist eine spezielle **asynchrone** Nachricht, die am Ende einer Aktivierung stehen kann aber nicht muss und zur aufrufenden Aktivierung zurückführt.

3.6 Packages

Packages sind nach der Spezifikation der UML ein Mechanismus zur Gruppierung von Design-Objekten. Diese Design Objekte sind klar der Designzeit zuzuschreiben und sind eine Art **Bibliothek von Fachklassen**. Bei den Objekten, nicht im OO-Sinne, handelt es sich methodenabhängig um:

- Operationen, Daten, Module, Operations- und Moduldiagramme (SA/SD),
- Entitäten und Diagramme des konzeptionellen Schemas sowie Datenbanktabellen (ERM/SERM) oder ER Diagramme
- Liste von Gruppen zur Inventarisierung und zur Installation mit entsprechenden Scripts

Unser Fokus ist klar auf die Klassen gerichtet, wobei ein Teil auch die DB mit den StoredProcedures beinhaltet. Gruppierungen von Objekten sind nach unterschiedlichen Gesichtspunkten möglich. Sie sind hierarchisch aufgebaut, d.h. eine Gruppe von Packages läßt sich wiederum zu einer weiteren PAC zusammenfassen. PAC lassen sich also in **Subsysteme** verschachteln.
Jede Sprache hat seine eigene Vorstellung, was ein Package ist. Allen gemeinsam ist die **Gruppierung** von Klassen. Das heisst auch, dass ein fachlich vollständiges Klassendiagramm ein Package ist. Was wiederum bedeutet, eine Unit mit ihren Klassen kann ein Package sein, im Sinne eines Entwurfszeit Package. Auch der BROKER hat zwei Packages (d.h. zwei Diagramme), aber der Reihe nach.

3.6.1 Packages und OOP

Je grösser das Projekt, desto mehr Klassen und Diagramme im Umzug. Eine grosse Zahl von Entwurfselementen läßt sich in beliebig vielen Modelleinheiten, den Packages, zusammenfassen und strukturieren.
Einheiten bestehen aus Elementen. In ein PAC gehören Klassen, die verbindende fachliche oder techn. Themen betreffen und einen intensiven Nachrichtenaustausch betreiben. Die Bindung in den PAC muss stärker sein als zwischen den PAC.
Als öffentlich gekennzeichnete Klassen lassen sich auch in fremden PAC, und sogar in anderen Projekten, benutzen, doch **physisch** als Source Code sind sie nur in einem PAC vorzufinden. Betrachten Sie also in einem PAC nur die Referenz einer Klasse, wenn eine Klasse in mehreren PAC benutzt wird. Dies ist z.B. bei der Vererbung der Fall. Im BROKER erben die Kontenklassen ja von einer Businessklasse. Kein Problem, die Businessklasse bildet mit anderen Klassen ein eigenes PAC, nämlich <bankData>, das zugleich auch die Unit ist:

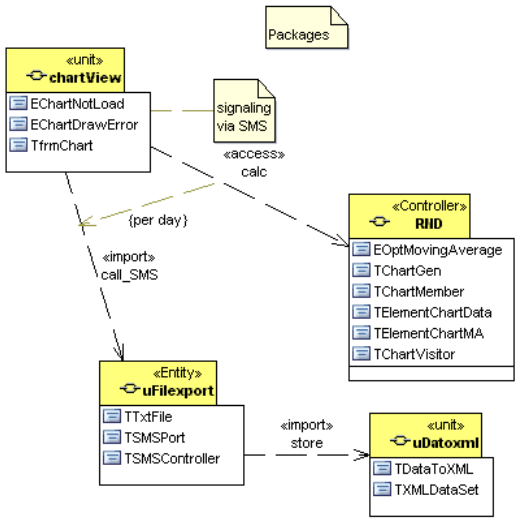


Abb. 3.24: 4 PAC in Abhängigkeit

3.6.1.1 Profiles

Wiederverwendung ermöglichen die PAC auch in anderer Hinsicht, denn Packages lassen sich verschachteln. Es gibt verschiedene Dinge, die man pro PAC festlegen kann: z.B. Na-

menskonventionen, Versionierung, Namensräume, Programmiertechnik, Standardkonstruktoren oder die Ereignisbehandlung. Diese Prinzipien braucht man nicht jedesmal neu zu definieren, sie «erben» die Konfiguration aus dem höher angeordneten PAC, in dem die Definition erfolgte.
Mit dem Erweitern durch sogenannte Profile in UML 2.0, welches eine Art Wörterbuch von Stereotypen darstellt, erhalten auch Packages eine Aufwertung.

> *Auf diese Weise erhalten Klassen oder Pakete innerhalb einer gemeinsamen Fachdomäne ihr eigenes Stereotypenpaket, z.B. das Profil Finanzmarkt.*

Das Profil mit den zugehörigen Typen und sogar Operationen erlaubt, in einem Durchzug und in allen Diagrammen den Typ Integer auf Float zu wechseln!
Ein Profil sollte gemäß der Spezifikation auch Toolübergreifend austauschbar und einsetzbar sein.

3.6.1.2 Indirekte Unit-Referenzen

Die Uses-Klausel eines Moduls muss nur die Namen der Units enthalten, die direkt von diesem Modul verwendet werden. Die folgenden Abhängigkeiten sind zentral für das verstehen der PAC-Notation.

> *Da eine Unit A in ihrer Uses-Klausel die Unit B aufführen kann und Unit B wiederum eine weitere Unit C in ihrer Uses-Klausel aufnehmen kann, ist A indirekt auch von C abhängig.*

Ganze Kaskaden von indirekten Unit-Referenzen sind möglich. Um ein Modul zu kompilieren, muss der Compiler in der Lage sein, alle Units aufzufinden, von denen ein Modul direkt oder indirekt abhängt.
Wenn Sie am Interface-Teil einer Unit Veränderungen vornehmen, dann müssen Sie **alle** Units rekompilieren, die diese geänderte Unit verwenden. Wenn Sie `Project-Build All` verwenden, so übernimmt der Compiler das für Sie.

> *Wenn aber Veränderungen nur am Implementations- oder Initialisierungsteil vorgenommen werden, so reicht es aus, **nur** die veränderte Unit erneut zu kompilieren, da der Interface Teil weiterhin gültig ist!*

Delphi erkennt, dass sich ein Interface-Teil einer Unit verändert hat, da während des Kompilierens eine Versionsnummer (z.B. Hash) der Unit berechnet und verwaltet wird.

3.6.1.3 Zirkuläre Unit-Referenzen

Zirkuläre Referenzen treten auf, wenn Sie wechselseitig abhängige Units haben, d.h. wenn eine Unit A in ihrer Uses-Klausel Unit B aufführt, und wenn die Uses-Klausel von Unit B ihrerseits Unit A referenziert. Solche zirkulären Referenzen sind dann zugelassen, wenn **höchstens eine** der zwei Referenzen im Interface-Teil erscheint.

> *Wenn die Unit A in der Uses-Klausel des Interface-Teils der Unit B erscheint, und die Uses-Klausel im Interface-Teil der Unit B die Unit A aufführt, dann erzeugt Delphi eine Fehlermeldung wegen zirkulärer Unit-Referenzen. Dies natürlich auch beim Erstellen von Packages.*

Wechselseitig abhängige Units können in speziellen Situationen nützlich sein, aber verwenden Sie sie mit Bedacht. Wenn ja, dann machen sie Ihr Programm schwieriger in der Wartung und eher anfällig für Fehler. Im BROKER ist eine einzige wechselseitige Abhängigkeit zwischen der Unit `bankControll` und der Unit `bankLogic` vorhanden, die ihren Grund hat.

```
unit bankLogic;
  implementation
      uses dialogs, trans, bankcontroll;


unit bankControll;
  interface
      uses bankLogic, classes;
```

3.6.2 Schichtenmodell

Generell gesagt, sollte eine Aufteilung zuerst in Packages erfolgen (modularisieren), und dann kommt die Überlegung des Outsourcen von Funktionalität, d.h. Methoden die auch von anderen Packages verwendet werden, lagert man als Komponenten aus.
In der einfachsten Form, die auch als „3-tier“ bezeichnet wird, besteht eine mehrschichtige

Anwendung aus den folgenden Dritteln:

- Client-Anwendung: Stellt die Benutzeroberfläche auf dem Rechner des Benutzers bereit.
- Anwendungsserver (Logik): Befindet sich an einer zentralen Position im Netzwerk, auf die alle Clients zugreifen können. Er stellt die allgemeinen Servicedienste bereit.
- Remote-DB-Server: Stellt das relationale Datenbankverwaltungssystem (RDBMS) bereit.

In diesem 3-tier Modell verwaltet der Anwendungsserver den Datenfluß zwischen den Clients und dem Remote-DB-Server. Er wird deshalb auch als „Daten-Broker„ bezeichnet. Es ist jedoch durchaus möglich, auch eigene Database-Backends zu erstellen oder komplette Frameworks einzusetzen. Alle diese Layers sind aber schon im Zeitpunkt des Designs mit einem PAC zu planen.
Es muss in dieser Phase noch nicht genau bekannt sein, welche Klassen sich im PAC befinden.
In komplizierteren mehrschichtigen Anwendungen können sich zusätzliche Dienste zwischen den Clients und dem Remote-DB-Server befinden. Dabei kann es sich bspw. um einen Broker für Sicherheitsdienste handeln, der sichere Internet-Transaktionen gewährleistet.

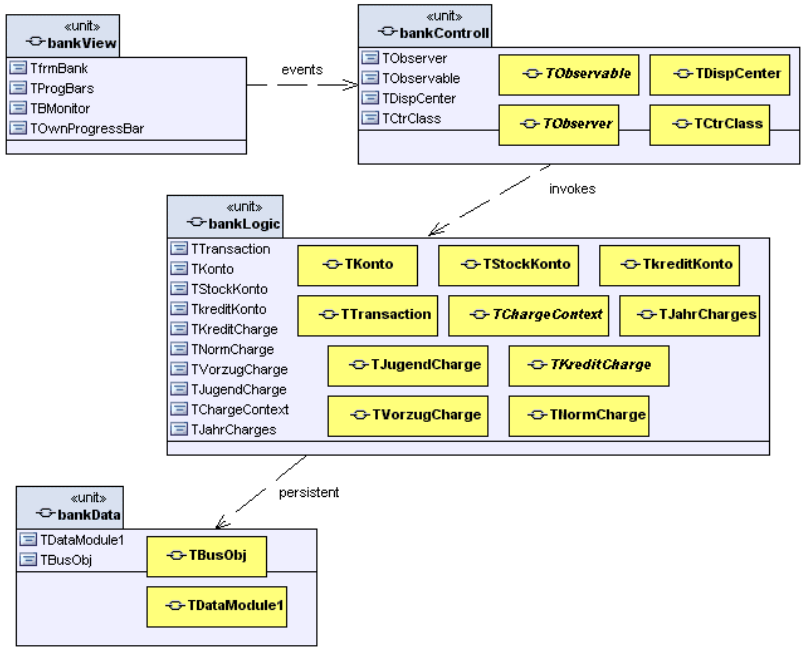


Abb. 3.25: Mehrschichtig mit dem PAC

Wenn Sie TCP/IP oder DCOM verwenden, können Sie die frühe Bindung nicht nutzen, da das Remote-Datenmodul keine duale Schnittstelle besitzt. Sie können jedoch die Schnittstelle des Anwendungsservers einsetzen, um eine bessere Leistung als bei der normalen späten Bindung zu erreichen.

3.6.3 Übergang zu Komponenten

Das Konzept der Packages ist natürlich auch förderlich für die Entwicklung von Componentware. Die Komponenten lassen sich im selben System entwickeln, solide getrennt in einzelnen logischen Packages. Aber die Schnittstellen der Komponenten, d.h. die öffentlichen Klassen, können ihre Funktion schon während der Entwicklung erfüllen, weil sie für die anderen Komponenten – in anderen Packages – zugänglich sind. Hierzu gibt es eine wichtige Regel:
Komponenten entstehen aus den einzelnen Designtime Packages, aber nicht jedes Package wird später eine eigene Komponente. Es gibt auch Packages die nie in eine Komponente einfliessen.
In «The SELECT Perspective» ist folgendes in Bezug auf Komponenten zu lesen:

«The Perspective architecture is a technically neutral framework for model building. However it is designed with the component marketplace very much in mind. Components are ***implementation packages***

of objects which provide services through their interfaces. Components should ideally provide a set of services belonging to a single service category. »

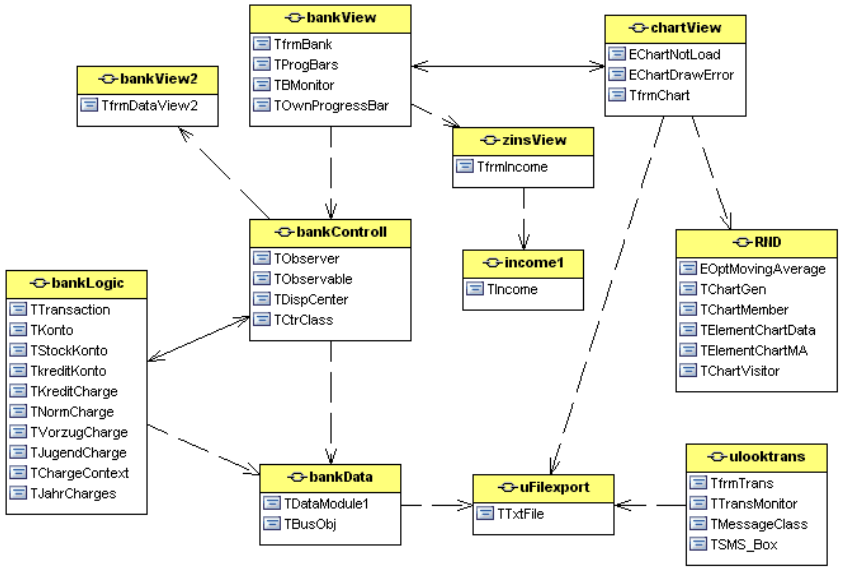


Abb. 3.26: Das PAC von BROKER

3.6.4 Schnittstellenmatrix

In Abb. 3.26 sehen Sie ein PAC. Was unverkennbar ist, das will die enge Bindung von PAC und Unit zeigen. Es hat zwei Verschachtelungen, die im fachlichen Zusammenhang stehen. Nun gibt es eine fantastische Möglichkeit, diese Abhängigkeiten auf eine Matrix zu projizieren. Dies führt zu einem Notationswechsel von einem Diagramm in eine Matrix und zurück. Denn es gibt zur Darstellung der Abhängigkeiten auch eine andere Form, die man Schnittstellenmatrix nennt. Diese Matrix lässt sich auf Stufe Methode, Klasse oder wie in userem Fall auf Stufe Packages wie eine klare Übersicht und Kontrolle anwenden:

Units	**rcvr.**										
Sender	zView	bView2	bView	bCtroll	bLogic	bData	bExp	utrans	inco1	Info1	pfrm
ZinsView									**x**		
bankView2											
bankView	**x**			**x**				**o**		**x**	
bankControll	**o**	**o**			**x**	**o**		**o**			**x**
bankLogic				**o**		**x**		**o**			
bankData							**o**	**o**			
bankExport								**o**			
Income1											
Infobox1											
Passform											

Tab.3.8: Die Schnittstellenmatrix als exakte Grundlage

Wie lesen wir nun diese Matrix. Eigentlich geht es darum: **Wer ruft wenn**. In den Zeilen sehen wir die Units als Sender, will heissen, die Units rufen andere Units auf. Die Empfänger (Receiver) sind in den Kolonnen angeordnet. Nehmen wir die Unit *bankView*, welche die Unit *bankControll, zinsView* etc. aufruft, d.h. von denen abhängig ist.

*Bei einem Kreuz <**x**> ist die Einbindung der Unit im Interface Teil zu finden, bei einem Kreis <**o**> ist die Einbindung der Unit im Implementations Teil zu suchen.*

Auffallend ist die optische Tendenz, eine Diagonale von links oben nach rechts unten zu erkennen. Dies deutet auf ein gutes Schichtenmodell hin, da die oberste Schicht zu den Views gehöret und die unteren (bis zur Unit *utrans*) die Service-Schichten repräsentieren.

Auf der anderen Seite weiter unten erkennen wir auch, dass die Units *utrans*, *income1*, *infobox1* und *passform* keinen Aufruf tätigen, sie gehören zu den reinen Empfänger Units. Die Unit *bankView*, also das Hauptfenster von BROKER, ist eine reine Senderunit, somit haben wir es erreicht, die **Sicht** von der **Logik** zu trennen. Dies erlaubt uns das Form ohne technische Konsequenzen auszuwechseln oder verteilt einzusetzen, da ja keine andere Unit *bankView* einbindet, sprich von ihr abhängig ist.

3.6.5 Notation

Low Read: WALK ON THE FILE SIDE
Gruppierungen von Klassen sind nach div. Gesichtspunkten möglich. Sie sind hierarchisch aufgebaut, d.h. eine Gruppe von Packages läßt sich wiederum zu einer PAC zusammenfassen. Das Collaboration ist neu ein Pattern Diagramm und gilt als PAC-Variante. Packages lassen sich auch in einer Schnittstellenmatrix notieren. Jede PAC kann in sich wieder mehrere PAC enthalten. Package-Diagramme dienen der Darstellung von Abhängigkeitsbeziehungen zwischen Packages. Die **Pfeilrichtung** zeigt die Abhängigkeit <dependencie> an. Der View ist also abhängig von der Logik und nicht umgekehrt. Somit kann sich die Logik auch einen anderen View holen, da keine Anpassung in der Logik nötig ist.
In Abb. 3.27 ist eine erweiterte Sicht eines PAC sichtbar, d.h. es wurde versucht, die Perspektive von BROKER auf eine Gruppierung von Services und Datenbankdiensten zu legen. Bei diesem PAC ist der Fokus auf die **Umsysteme** gerichtet, also DLL's und DB. Bestätigen wir noch einmal die Aussage, dass hinter einem nicht verschachtelten PAC im Prinzip ein Klassendiagramm steckt, das aus einer Unit kommt.

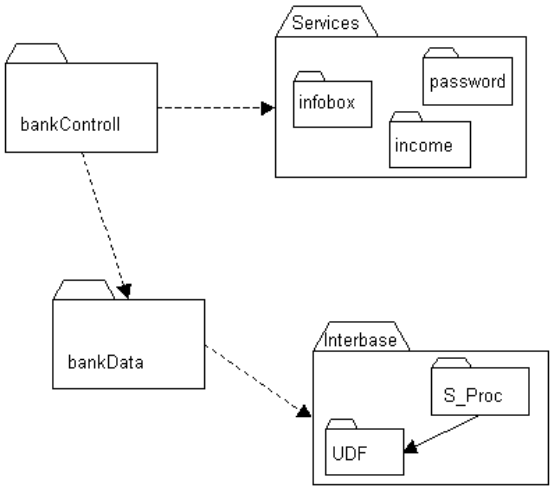


Abb. 3.27: Eine andere Gruppierung der Packages

Werfen wir deshalb einen Blick hinter das PAC <bankControll> (Abb. 3.26), welches eine Steuerungsschicht zwischen dem View und der Logik ist, eine Art Vermittler auch. Alle eintretenden Ereignisse des Form lassen sich in dieser Unit abfangen und nach einer bestimmten Technik **weiterleiten**.

3.6.5.1 Abhängigkeit

Eine Abhängigkeit besteht zwischen Elementen, wenn Änderungen an der Definition eines Elements Änderungen an der Definition des anderen Elementes erzwingen können. Bei Klassen z.B. existieren Abhängigkeiten aus verschiedenen Gründen: Eine Klasse ruft eine andere Klasse auf (sendet eine Nachricht); eine Klasse hat eine andere Klasse (<aggregation>) als Teil ihrer Felder; eine Klasse erwähnt oder übergibt eine andere als self in den Parametern (z.B. im schon erwähnten Strategie-Muster).
Wenn eine Klasse ihr Interface mit vorhandenen Parametern ändert (Signatur), dann ist möglicherweise eine Nachricht einer anderen Klasse nicht mehr gültig, bei einer Änderung in der Implementation tritt die Abhängigkeit in der Regel nicht ein, sofern die Änderung eine Verbesserung bewirkt und nicht neue Fehler erzeugt.

Zudem läßt sich nun zwischen <access> und <import> als Abhängigkeitslinie unterscheiden: Bei <access> greift eine Klasse auf die öffentlichen Elemente der anderen Klasse zu, bei <import> wird der ganze Geltungsbereich eines Paketes dem anderen Paket hinzugefügt oder eben importiert, z.B. bei einer IDL, DLL oder bei Type Libraries.

3.6.5.2 Transitivität

Eine Ähnlichkeit zwischen Paketabhängigkeit und Kompilationsabhängigkeit ist offensichtlich. Jedoch gibt es einen grundsätzlichen Unterschied:

Bei Packages sind die Abhängigkeiten nicht transitiv.

Wenn A > B und B > C dann ist A > C, also transitiv z.B. eine Grössenmetrik. Aber Clinton ist Freund von Gates und Gates ist Freund von Hussein dann ist Clinton nicht unbedingt Freund von Hussein, d.h. diese Folge ist nicht transitiv, oder die Trusts einer Multiple-Domain in NT sind auch nicht transitiv.
Um zu erkennen, warum dies wichtig ist, betrachten wir den BROKER in Abb. 3.27 innerhalb des erweiterten PAC. Wenn sich eine StoredProcedure im <InterBase> Paket ändert, heisst dies nicht, dass im <bankControll> was zu ändern ist, es zeigt höchstens an, dass man auf **Änderungen** hin untersuchen muss. In diesem Fall kapselt oder schirmt <bankData> den <bankControll> vor Änderungen in <InterBase> ab.
Dieses Verhalten ist der klassische Zweck einer geschichteten Architektur: Minimieren von Abhängigkeiten ist die Kunst grosser Systeme!

3.7 Components

3.7.1 Wahl der Architektur

receives a pointer to an interface to the object on return...

Bevor wir zum Bau einer Laufzeitumgebung kommen, noch ein paar Worte, oder eher Sätze, zur Architektur. Grundlegend ist in der jetzigen Phase die Architektur schon **vorbestimmt**. Je modularer unser Design ist, desto skalierbarer in der Einteilung der binären Einheiten.
Bei der Wahl der Architektur sollte man sich drei Punkte vergegenwärtigen:

- Die Wahl der Komponenten, die wir in einem System als Primitiven einsetzen, ist relativ willkürlich und hängt stark vom Ermessen des Entwicklers und deren Entwicklungsumgebung ab. Eine Komponente in UML kann aus Sourcen (bei Interpretern ein Muss), Binaries (z.B. obj-Files, DLL's) oder Executables bestehen.
- Die Beziehung **innerhalb** der Komponenten sind im allgemeinen stärker als die Beziehungen **zwischen** den Komponenten. Aufgrund dieser Tatsache lässt dich die hochfrequente Dynamik der Komponenten trennen von der niederfrequenten Dynamik - das betrifft die Beziehung und Abhängigkeit zwischen den Komponenten.
- Ein komplexes System, das funktioniert, hat sich mit Sicherheit aus einem einfachen System entwikkelt das funktionierte. Ein komplexes System das von Grund auf entworfen wird, funktioniert selten und läßt sich auch nicht dazu bringen zu funktionieren. Sie müssen dann wieder von vorne anfangen und mit einem einfachen, funktionierenden Subsystem beginnen.

Merke: «Die Kunst grosser System besteht darin, Abhängigkeiten zu minimieren»

3.7.1.1 Vermeiden von Abhängigkeiten

Bei der Entwicklung von Komponenten sollte man immer die Sicht des Entwicklers berücksichtigen, der später damit arbeitet. Mit dem Einsatz einer Komponente sollen möglichst wenig Nebenbedingungen verbunden sein, d.h. wir sollten freien funktionalen und zeitlichen Zugriff auf die Komponente haben. Natürlich wird sich das nie vollständig erreichen lassen.
Die folgende Liste nennt einige vage Abhängigkeiten, die man unbedingt vermeiden sollte:

- Man sollte keine zusätzlichen Methoden aufrufen, um die Komponente verwenden zu können.
- Der Aufruf von Methoden sollte nicht an eine bestimmte Reihenfolge gebunden sein.
- Methoden sollten die Komponente nie in einen Zustand versetzen, in dem bestimmte Ereignisse oder Methoden ungültig sind.

Will heissen, man sollte immer Strategien bereitstellen, mit denen solche unerwünschten Verflechtungen abgefangen werden können. Wenn es bspw. eine Methode gibt, welche die

Komponente in einen Zustand versetzt, in dem eine andere Methode ungültig wird, schreiben Sie eine **weitere** Methode, die von der Anwendung aufgerufen wird, um diesen ungünstigen Zustand zu korrigieren. Zumindest sollten Sie dafür sorgen, daß die Laufzeitumgebung eine Exception auslöst.

3.7.2 Notation

Think Floyd: DARK SIDE OF THE CPU

Ein Komponentendiagramm läßt sich verwenden, um die Zuordnung von Klassendiagrammen im Sinne von Packages und physischen Objekten zu Komponenten im physikalischen Systemdesign zu zeigen.

> *Merke: Jede Komponente besteht aus einem oder N - Packages, aber nicht jedes Package wird eine Komponente. Im Gegensatz zu den Packages mit den fachlichen Gruppierungen ist das Interesse von Komponenten auf die technische Bereitschaft gerichtet.*

Für jede COM wird ein Name benötigt. Dieser Name gibt normalerweise den einfachen Namen der physikalischen Datei im Entwicklungssystem an (ohne Endung, diese ist in Verbindung mit einer entsprechenden Liste zu setzen). In unserem COM in Abb. 3.28 finden sie die Extension trotzdem, für den Administrator ein Vorteil.
Die einzige Beziehung, die es zwischen zwei Komponenten (Module) geben kann, ist die Compilations-Abhängigkeit zur **Laufzeit** (C++ gibt dies Abhängigkeit mit `#include` an, OP mit `uses unit1, unit2` etc., die wir bei den Packages schon untersucht haben). Im allgemeinen dürfen die Abhängigkeiten **nicht zyklisch** sein.

> *Abhängigkeiten bei COM sind meistens transitiv. In der Tat ist die Semantik jedoch von der Sprache abhängig. Das Java <import> ist nicht transitiv aufzufassen. Das <include> von C++ ist jedoch transitiv, beim <uses> von ObjectPascal haben wir Wahlfreiheit. Sie sehen, eine transitive Abhängigkeit macht es schwierig, die Neuübersetzung bei Änderungen einzuschränken.*

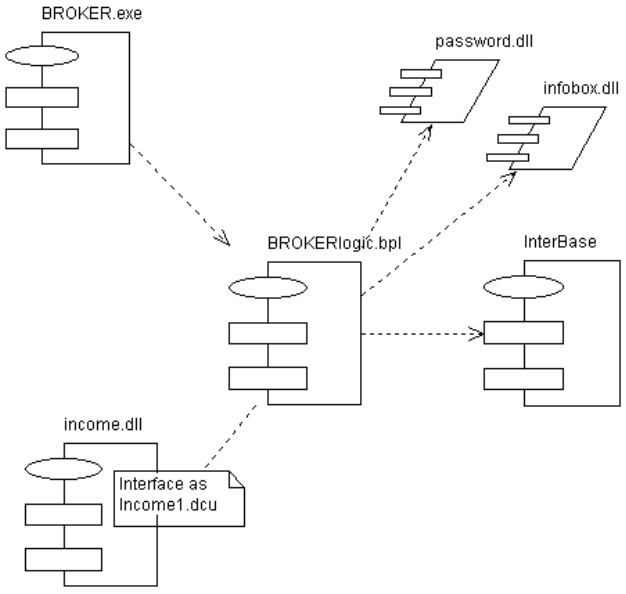


Abb. 3.28: Komponenten und ihre Abhängigkeiten

Unser COM in Abb. 3.28 sollte nun so wie es ist verstanden werden. Die Runtime Package <BROKERLogic> läßt sich nun unabhängig vom View auswechseln oder in eine physische Middleware versetzen. Die Komponente <Income> stellt eine Schnittstelle zur Verfügung, was die Flexibilität noch erhöht. Somit ist das Komponentendiagramm vollständig.
Vermehrt ermöglichen <ports> und <connectors> zwischen einer angebotenen (Technik, Typ, Laufzeit und Signatur) und einer erforderlichen Schnittstelle das mögliche „Zusammenstecken“ prüfen zu lassen um z.B. festzustellen, daß eine MSCOM nicht zu einem EJB oder einer CLX kompatibel ist. Durch die Detailsicht mit Hilfe von <ports> sind neu, geschachtelte Komponenten oder komplizierte Architektur wie ganze Container besser beschreibbar.

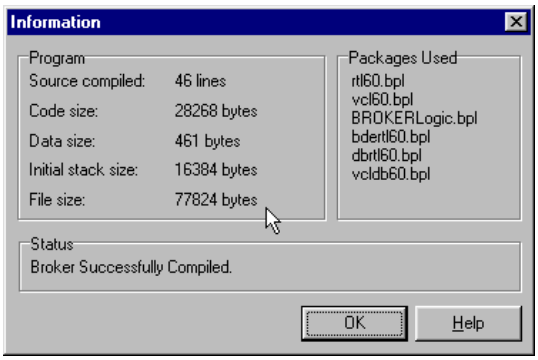


Abb. 3.29: Komponenten kompiliert

3.8 Deployment

Der Entwurf mehrerer Teilprogramme oder Komponenten, die man auf unterschiedlichen Prozessoren oder Systemen ausführt, erfordert einen Designprozess, der sich völlig der Modellierung von Klassen und Objekten unterscheidet. Um das Problem der Verteilung der Prozesse auf die einzelnen **Prozessoren** zu veranschaulichen, zeigt ein **Verteilungsdiagramm** (DEP) die realen Beziehungen zwischen Soft- Hard- und Netzkomponenten an.
Ein DEP (ehemals Prozessdiagramm) enthält die Prozessarchitektur eines lauffähigen Systems. Dabei werden die Prozessoren eines Systems mit den auf ihnen laufenden Prozessen dargestellt, oder auch die angeschlossenen externen Geräte und die Verbindungen zwischen den Hardwarekomponenten[5].
Mit einem DEP lassen sich Plazierung und Migration von Komponenten in einem verteilten System aufzeigen (interessant für CORBA, SOAP oder DCOM aber auch Transaktions-Monitore und Applikationsserver). Ein Dia, das auch den Admin / Netzwerker interessiert.

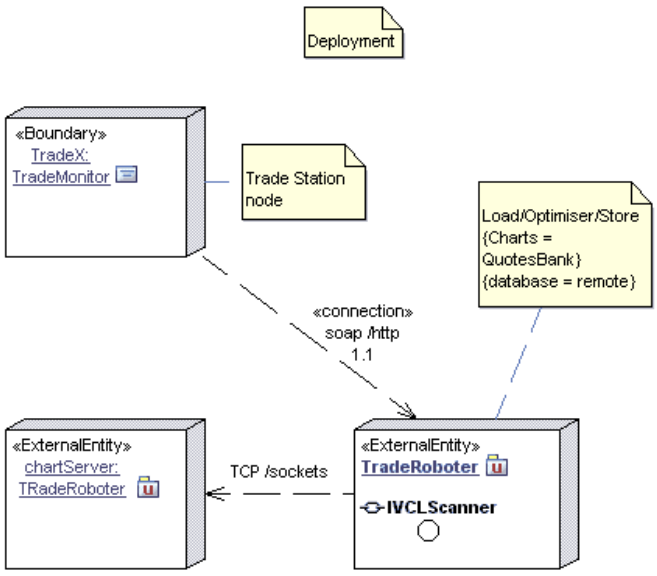


Abb. 3.30: Deployment von BROKER

3.8.1 Verteilte Anwendung

Verteilte Anwendungen lassen sich auf unterschiedlichen Computern und Plattformen ausführen. Die verschiedenen Bestandteile arbeiten (normalerweise über Netzwerk) zusammen, um bestimmte Funktionen durchzuführen. So benötigt bspw. eine Anwendung für unsere Finanzfirma einzelne Client-Anwendungen für die Niederlassungen, einen Hauptserver für die Verarbeitung der Client-Anforderungen und eine Schnittstelle zu einer DB.

[5] Der Begriff Komponente wird sehr vieldeutig gebraucht

Mit Hilfe einer verteilten Client-Anwendung wie dem BROKER, läßt sich in dieser Situation das Pflegen und Aktualisieren der Clients spürbar vereinfachen. Verteilte Anwendungen haben immer mit mehrschichtigen Modellen zu tun, denn was nicht **mehrschichtig** ist, kann man auch nicht **verteilen**.

3.8.1.1 Im Falle BROKER

Nachstehend werden einige der Vorteile des mehrschichtigen Modells zusammengefaßt: Kapselung der Anwendungslogik (Business Rules) in einer freigegebenen Middleware ist mit <BROKERLogic> ansatzweise erfüllt. Die verschiedenen Client-Anwendungen <Broker> greifen alle auf dieselbe Mittelschicht zu. Dadurch lassen sich Redundanzen vermeiden und Wartungskosten einsparen, da man die Business Rules (Logik und Datenzugriff) nicht für jede neue Client-Anwendung erneut implementieren muss.

3.8.1.1.1 Thin Client

Thin Client-Anwendungen lassen sich extrem klein halten, wie die Grösse von ca. 70 KByte zeigt. Das Hauptgewicht der Verarbeitung ist an die Mittelschicht **delegiert**. Daraus ergibt sich der weitere Vorteil einer vereinfachten Weitergabe von BROKER, da keine Installation, Konfiguration und Wartung der Software zur Herstellung der DB-Verbindung (bspw. der BDE) erforderlich ist.

3.8.1.1.2 Datenlogik

Die Verteilung verschiedener Logiken auf mehrere Rechner kann die **Leistung** durch die bessere Lastverteilung erhöhen. Gleichzeitig lassen sich redundante Systeme aktivieren, wenn bspw. ein Server heruntergefahren werden muß. Im BROKER kommen zusätzlich noch StoredProcedure zum Einsatz, die sich zentral warten und kontrollieren lassen.

3.8.1.1.3 Laufzeitverhalten

Sie können kritische Funktionen, wie der Passwortdialog oder ein Zinsserver in Schichten isolieren, die über verschiedene Zugriffsbeschränkungen verfügen. Dies ermöglicht die Einrichtung flexibler Stufen, die sich auch separat auf Performance hin optimieren lassen. Erst nach Verteilung unter hoher Last kann die Zeit kritisch werden.

3.8.2 Notation

Tina Turner: NETWORK CITY LIMIT
Es stehen für verteilte Anwendungen folgende Implementierungsmodelle zur Verfügung:

- TCP/IP-Anwendungen, Sockets und SOAP WebServices
- COM- und DCOM-Anwendungen
- CORBA und RMI -Anwendungen
- Datenbankanwendungen mit Transaction- oder Applicationserver

Diese Information läßt sich nun mit <Knoten> und <Verbindungen> darstellen.

3.8.2.1 Knoten

Jeder Knoten <node> eines DEP repräsentiert eine (wie auch immer geartetete) Verarbeitungseinheit - zumeist eine Stück Hard- oder Middleware. Unser DEP in Abb. 3.30 hat vier Knoten, wovon einer verschachtelt ist. Dieses «Stück» kann ein Router, ein Sensor aber auch ein Mainframe als Gesamtes sein. Hier ist der Fokus (Perspektive und Abstraktion) besonders entscheidend. Ich kann auch von einer Wetterstation ein DEP erzeugen.
Die Komponenten in den Knoten entsprechen meistens exakt den lauffähigen Paketen. Somit zeigt ein DEP an, wo jedes lauffähige Paket im System abläuft.

3.8.2.2 Verbindungen

Eine Verbindung <connection> zwischen den Knoten repräsentieren die Kommunikationspfade, die im System interagieren. Hier kann von TCP/IP bis hin zu einem Infrarot, WLAN oder einem Frame-Relay fast alles gemeint sein. Eine <connection> ist auch stark von der Topologie und dem eingesetzten Netzwerk abhängig. Grundsätzlich sind die DEP in UML 2.0 noch zu wenig spezifiziert.
Jedoch mit den <Deployment Descriptors> erweitert, so daß ein informeller Text in absehba-

rer Zeit auch ein Tool generieren kann und sozusagen als Manifest-Datei in den Knoten einpackt. Diese Angaben lassen sich dann auf der Zielplattform durch MDA in eine Konfigurationsdatei umwandeln (s. Abb. 3.30). Im BROKER benutzen wir zwei Verbindungsprotokolle (soap & sockets), die gleichzeitig auch Transportprotokolle sind.

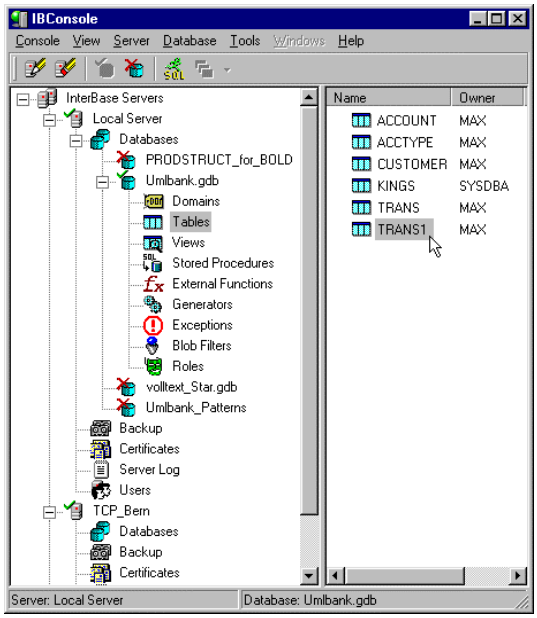


Abb. 3.31: Die DB von BROKER

4 Projektmanagement

Das klassische Wasserfallmodell beschreibt den organisatorischen Ablauf analog eines Wasserfalls. Die einzelnen Lebenszyklen werden in einer fest vorgegebenen Reihenfolge starr durchlaufen. Die einzelnen Phasen laufen sequentiell ab, eine neue Phase wird erst begonnen, wenn die vorhergehende Phase komplett abgeschlossen ist.

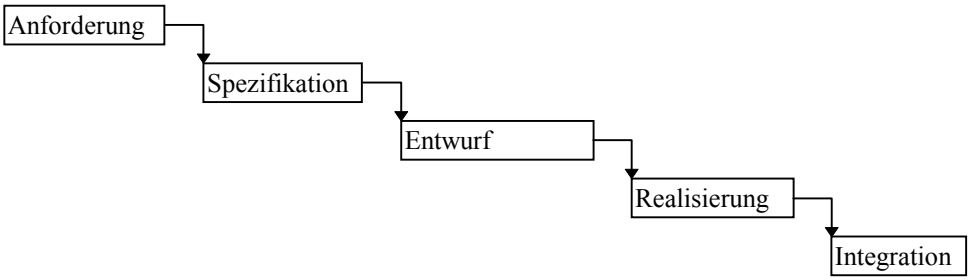
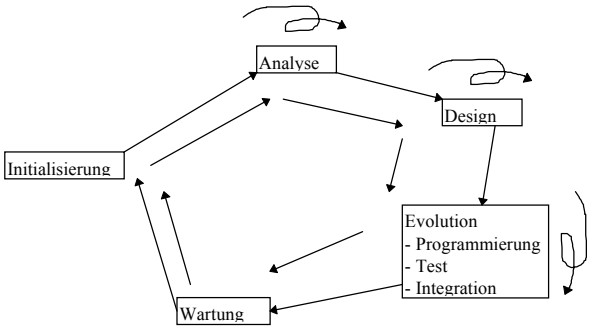


Abb. 4.32: Altes lineares Wasserfallmodell

Ändernde Anforderungen oder neue Erkenntnisse, die während der Entwicklung des Systems entstehen werden nicht berücksichtigt.
Das Projektmanagement läuft parallel zur Entwicklung des Systems. Es umfasst Qualitätssicherung, Organisation, Methodik, Projektmanagement und Kosten / Nutzen - Betrachtungen. Während allen Phasen wird laufend eine Dokumentation aufgebaut.

4.1 Evolution und Zyklen

Um die Problematik der starren Abfolge der einzelnen Phasen schlägt Booch eine Vorgehensweise gemäss einem erweiterten Wasserfallmodell vor. Der von Booch als Macro-Prozess bezeichnete Zyklus bildet den Rahmen für einen spiralförmigen Micro-Prozess, der die Vorgehensweise innerhalb den verschiedenen Phasen beschreibt.
Die einzelnen Phasen bestehen nicht mehr aus einem festen Block, sondern sind Einzelschritte der inkrementellen, iterativen Entwicklung. Iterationen sind phasenübergreifend! Das Gesamtsystem Schritt für Schritt aufgebaut, indem man stets die bestehende Version um eine neue zusätzliche Funktionalität erweitert. Bei jeder Erweiterung eines bestehenden Programms - bspw. einem neuen Produkterelease - wiederholt sich der dargestellte Prozess.



Wichtig :

- frühzeitig sorgfältige , stabile Schnittstellendefinition!
- Entwurfsentscheidungen, die mit hoher Wahrscheinlichkeit wieder geändert werden, einkapseln!

Abb. 4.33: Zyklen sind Teil jeder Entwicklung

4.2 V-Modell und UML

Beim Einsatz der UML mit dem V-Modell ist jedes Darstellungsmittel eindeutig das Ergebnis einer Elementarmethode. Es reicht aus, den Produkttypen Darstellungsmittel zuzuordnen.
Die Produktflüsse des V-Modells halten fest, in welchen Aktivitäten ein Produkt entsteht, und damit ist auch die Verwendung der entsprechenden Methoden impliziert.
Die Frage bleibt noch, wie kann man eine durchgängige Methode begründen, d.h. miteinander kombinierbar und kompatibel machen. Dies ist ein wichtiger Punkt, da die Ergebnisse der Elementarmethoden aufeinander aufbauen und vielfältige über die beschriebenen Schnittstellen hinausgehende **Konsistenzbeziehungen** besitzen sollten. Dieses Dilemma ist zu

lösen, so dass wir im BROKER eine Zuordnung der Produkte (vor allem UML-Diagramme) zu den Phasen im Voraus vornehmen. Wobei es eben auch Produkte gibt, die phasen- begleitend oder übergreifend sind.
Wünschenswert ist es, dass für die Produkttypen nur Diagramme benutzt werden, die sich gegenseitig ergänzen. Die Konsistenz zwischen Produkten lässt sich leichter herstellen.

4.2.1 Erwartete „Produkte“

Die erwarteten Artefakte betreffen einerseits die Grundlagen zu den Reviews und andererseits den geforderten Liefer- und Leistungsumfang von BROKER. Die bereits erstellten Dokumente wie Pflichtenheft, Sicherheitskonzept, Einsatzkonzept etc. sind in der folgenden Liste nicht enthalten. Der Projektleiter ordnet den Projektmitarbeitern auf der Grundlage des V-Modells Aktivitäten zu. Die Durchführung der daraus resultierenden Produkte liegt dann zur Bearbeitung vor. Indirekt ist diese zentrale Liste auch eine Übersicht des Lieferumfanges und gehört ins **Prozesshandbuch**. Folgend nun unsere Version der Zuordnungstabelle, die mit dem Einsatz einer Technik in der entsprechenden zeitlichen Phase auch gleich methodisch stimmt.

4.2.2 Vorgehensmodell im BROKER

Ref	Aktivität (Was)	Technik (Wie)	Produkt (Womit)	Phase (Wann)
1.	Benutzeranforderungen, Ereignisliste erstellen	Interview,Ereignislist Dokustudium	**Use Case** Pflichtenheft	Initialisierung / Analyse
2.	Geschäftsprozesse ermitteln	Reverse Engineering	**Aktivität**	Analyse
3.	GUI und Tech. Anfoderung	V-Template	Spezifikation	Analyse / Design
4.	Datenmodell konstruieren	Relationen, Entitäten	ERD	Design
5.	Fach-Klassen bestimmen, Interfaces definieren	CRC-Karten, Model-Maker	**Klassendiagramm** Statische Struktur	Design
6.	Zustände und Business Rules festlegen	Petri-Netze, Transitions	**State Event** Diagram	Design
7.	Software-Architektur definieren, Schichtenmodell	Schnittstellenmatrix	**Packages** Diagramm	Design / Implementierung
8.	Ersten Prototyp erstellen	GUI Style Guide	Code (Demo)	Implementierung
9.	Dyn. Interaktionen im System, Aufrufkaskaden	Reviews, Testscript	**Sequenzdiagramm**	Implementierung
10.	Modulariseren der Klassen	Package Technik	**Components**	Implementierung
11.	Softwaredokumentation Laufzeitumgebung	Topologie testen	**Deployment** Plattform	Integration
12.	Pilot (Know-How Transfer)	Workshop	Code (Demo)	Integration
13.	Systemtests organisieren	Szenariotechnik	Testplan (QS)	Integration
14.	Abnahme vorbereiten, Korrekturen	V-Template	Prüfprotokoll Mängelliste	Integration
15.	Betrieb & Unterhalt sichern	Recherchen, Nutzenrelation	Wartungsvertrag Helpdesk	Betrieb

Tab.4.9: UML im Vorgehensmodell

Mit einem Tool wie in-Step wäre folgendes möglich: Wählt ein Projektmitarbeiter unter der GUI ein ihm zugeordnetes Produkt einfach per Maus an, so wird es automatisch in der korrekten Version und mit dem richtigen Tool geöffnet, z.B. StarOffice, Delphi oder ModelMaker. Die Bandbreite der Werkzeuge kann von Textverarbeitung, Tabellenkalkulation und Projektmanagementsystemen eben bis zum Compiler reichen. Sie umfasst also alles, was sich bei der Softwareentwicklung nach dem V-Modell einsetzen lässt.

4.2.3 Pflichtenheft

Die fachlichen und vor allem technischen Anforderungen beinhalten folgende Themen:

- **Organisatorische Einbettung**
  Dieses Kapitel enthält eine Beschreibung des organisatorischen Einsatzbereiches des Systems: die Aufbau- und Ablauforganisation beim Anwender, Nutzerklassen, Verantwortlichkeiten und Zuständigkeiten beim Einsatz sowie die weitere Infrastruktur des Anwenders.
  BROKER: Die Anwendung wird als kleines Terminal in einer Privatbank eingesetzt. Schwerpunkt ist der Wertschriftenhandel, die Vermögensberatung sowie die Vermögensverwaltung. Die Nutzer sind Broker und Anlageberater, die sich eine schnelle Übersicht über die finanzielle Situation des Kunden verschaffen wollen. Die meisten Buchungen erfolgen direkt über den Telefonverkauf. Teilweise werden auch Kredite gewährt. Ende des Jahres wird ein Vermögensauszug erstellt, der auch Kosten und Zinsrechnung beinhaltet.
- **Nutzung**
  Hier sind Anforderungen an die Bedienung und Konfiguration des Systems festzuhalten, wie z.B. an mobilen oder stationären Einsatz, Einsatzzeiten, an die Kommunikation von Anwender und System, an den Umfang der vom System automatisch bereitzustellenden Dienste, usw.
  BROKER: Der Broker muss einfach bedienbar sein und Buchungen auf den wichtigsten Kundenkonten ermöglichen. Der Bankauszug muss in einem mobilen Format speicherbar sein. Vermögensnachweis und Zinsdienste sind erforderlich. Zinskosten bei Lombardkrediten sind in einem späteren Release vorgesehen. Nur die Buchung, Mutation und das Reporten steht in diesem Modul im Vordergrund, Wertschriftenverwaltung und Buchhaltung regelt ein anderes Modul.
- **Externe Schnittstellen**
  Anforderungen an die externen Systemschnittstellen zu Nachbarsystemen und im speziellen an die Mensch-Maschine Schnittstelle sind zu spezifizieren.
  Anforderungen hinsichtlich der Ergonomie und Sicherheit sind hier zu berücksichtigen. Die technische Ausgestaltung der Oberfläche wird im Rahmen der Techn. Anforderungen definiert.
  BROKER: Eine Schnittstelle zum Transaktionsserver, welcher alle Transaktionen und Börsenkurse zeigt, muss über ein Protokoll vorhanden sein. Ferner modulare Architektur mit DLL's und einem Schichtenmodell. Die Kreditlimiten müssen durch ein Passwort geschütz sein. Die Oberfläche einfach, da eine Browserlösung im Vordergrund steht.
- **Beschreibung der Funktionalität**
  Dieser Abschnitt nimmt die fachliche Strukturierung der Funktionalität des Systems aus Anwendersicht auf. Es gilt eine Funktionsliste zu erstellen.
  BROKER: Die Anwendung weist folgende Funktionalität auf:
  - Verwalten des Kundenstammes wie der Kontotypen
  - Ein- und Ausbuchen der Kontobewegungen
  - Kontenauszug mit graphischem Portfolio
  - Berechnen der Kostenstruktur (Gebühren) und Kreditlimiten eines Kunden
  - Ermitteln des Transaktionsvolumen (Umsatzberechnung)
  - Verfolgen der Transaktionen und Charts generieren
  - Zinseszinsberechnung und Systemkonfiguration
- **Qualitätsforderungen**
  Es werden Anforderungen hinsichtlich der Qualitätsmerkmale Zuverlässigkeit, Nutzerfreundlichkeit, Effizienz, Wartbarkeit, Übertragbarkeit, Wiederverwendbarkeit, Leistungsdaten usw. formuliert. Basis ist die DIN ISO 9126.
  BROKER: Die Wiederverwendung und Erweiterbarkeit muss im Sinne eines späteren Release gegeben sein. Die Transaktionen müssen redundant ausgelegt sein. Es sind vor allem OO-Techniken beim Softwarebau einzusetzen.

4.3 Die 4 Aufgaben des V-Modells

Ein Vorgehensmodell ist ein Standard, gemäss dem Entwicklungsprozesse in Projektform durchgeführt werden. Ein spezielles Vorgehensmodell für die Systementwicklung ist von der KBSt entwickelt worden. Dieses Vorgehensmodell wird von der MID GmbH als vollständiges Referenzmodell ausgeliefert und lässt sich als Grundlage für ein entsprechendes Projekt ableiten und verwenden.
Im eigentlichen V-Modell- bzw. KBSt-V-Modell-Projekt kann der Benutzer dann das Vorgehensmodell, das als Referenzmodell dient, an das konkrete Projekt anpassen. Hierbei bedient er sich vorab spezifizierter Anpassungsregeln, die zu einem konsistenten und standardkonformen Projekt führen. Ausserdem hat er die Gelegenheit, Informationen zu seinem konkreten Projekt zu erfassen.

- Projektmanagement PM
- Softwareentwicklung SE
- Qualitätssicherung QS
- Konfigurationsmanagement KM

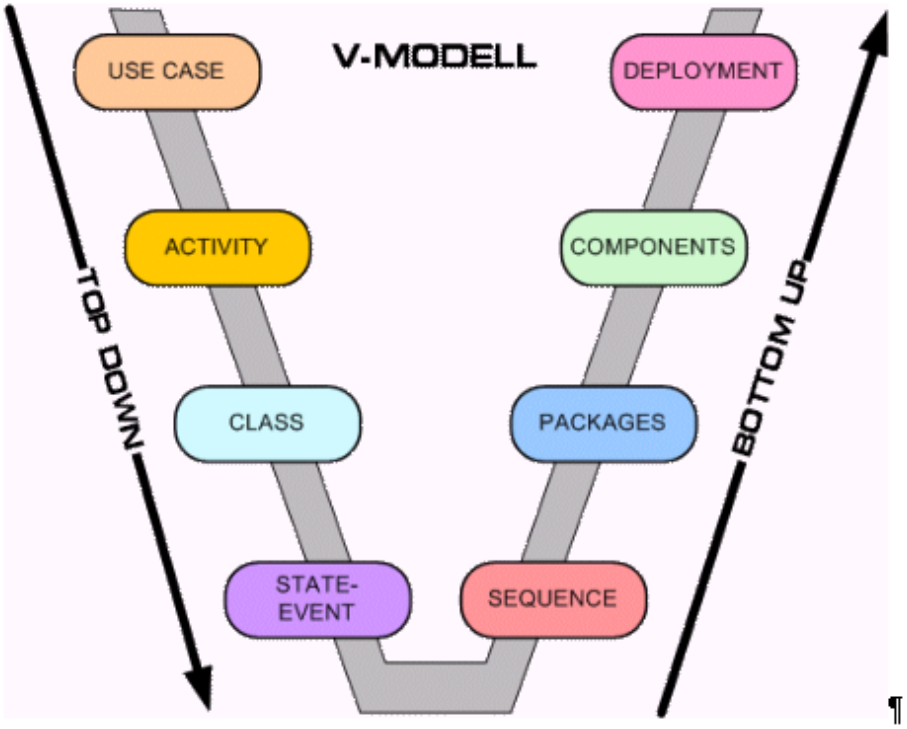


Abb. 4.34: Mit Methode zum V-Modell

5 UML CASE-Tools

Hier wird auf die Beilage verwiesen, die erstmals 1997 im Entwickler erschienen ist, und jedes Jahr aktualisiert wird. Es lassen sich jedes Jahr rund 8 Tools vorstellen, die meisten davon kommen immer wieder: http://max.kleiner.com/umltools.htm

6 Anhang

UML / MDA Diagramme der 2.0

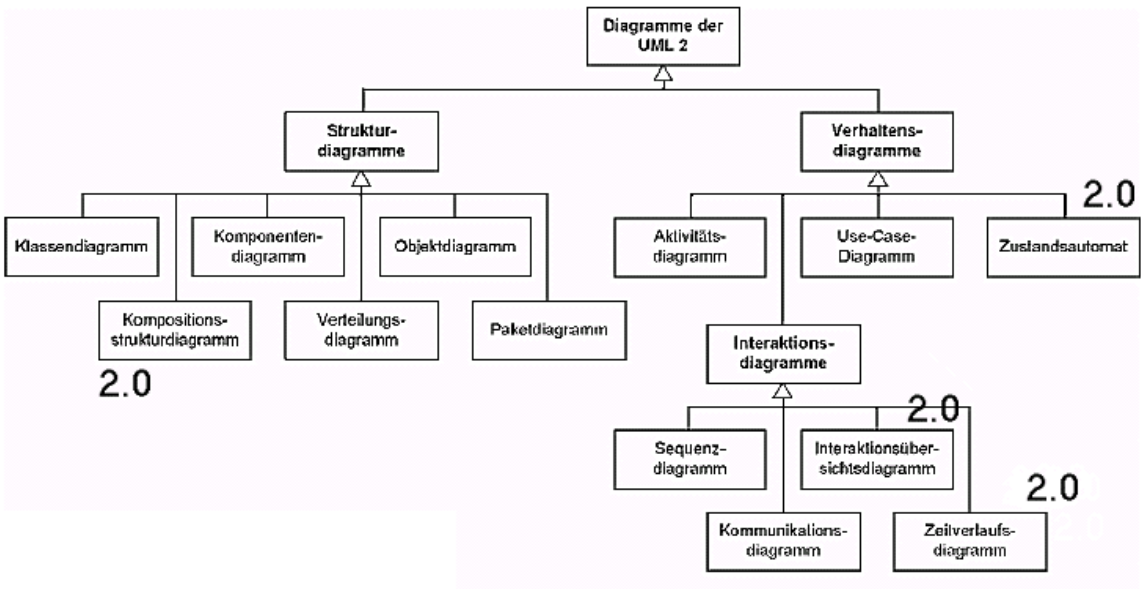


Abb. 6.35: UML Diagram Overview

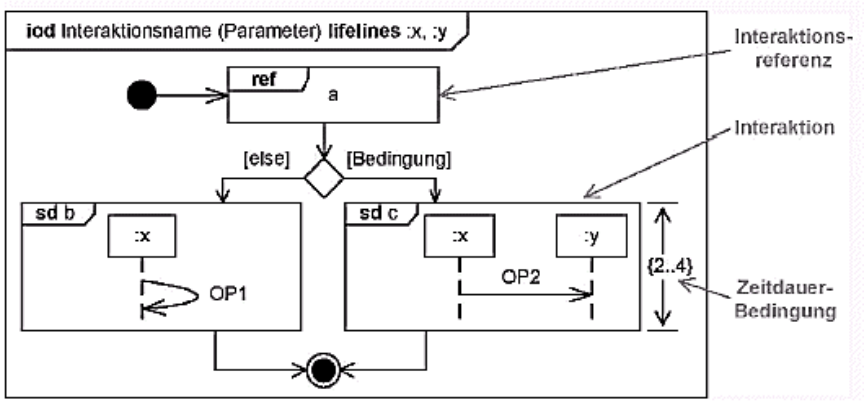


Abb. 6.36: Interaction Overview

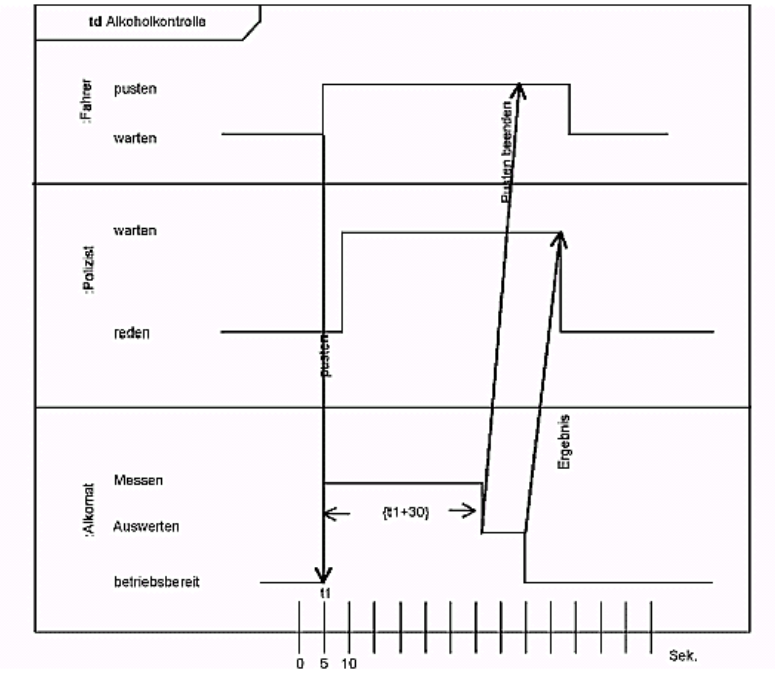


Abb. 6.37: Timing Diagram

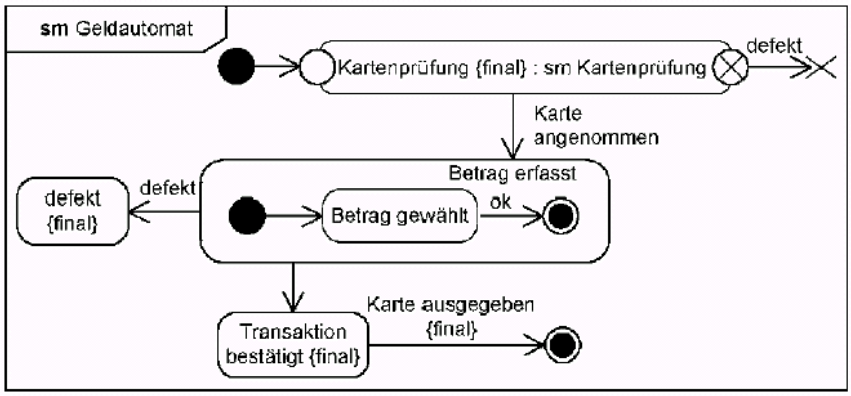


Abb. 6.38: Protokoll Automat BROKER extern

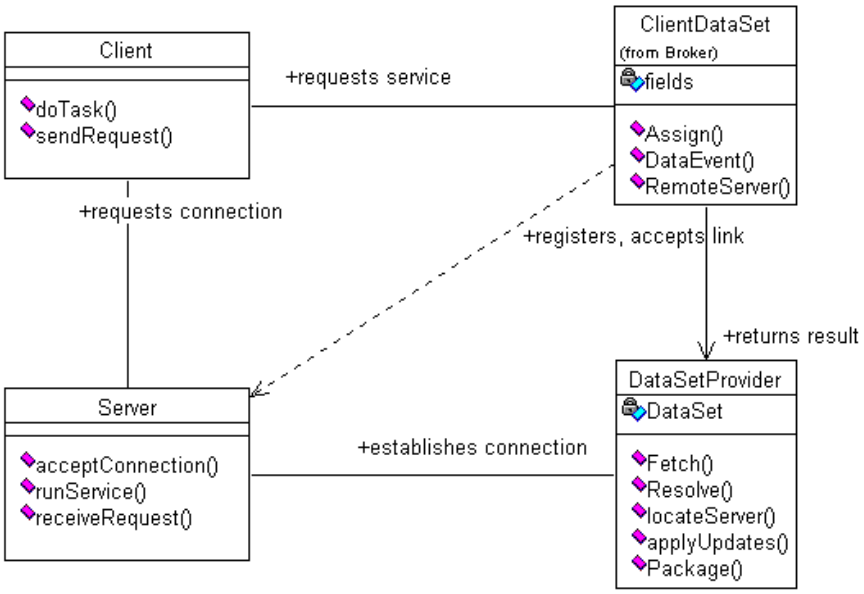


Abb. 6.39: MDA als PSM

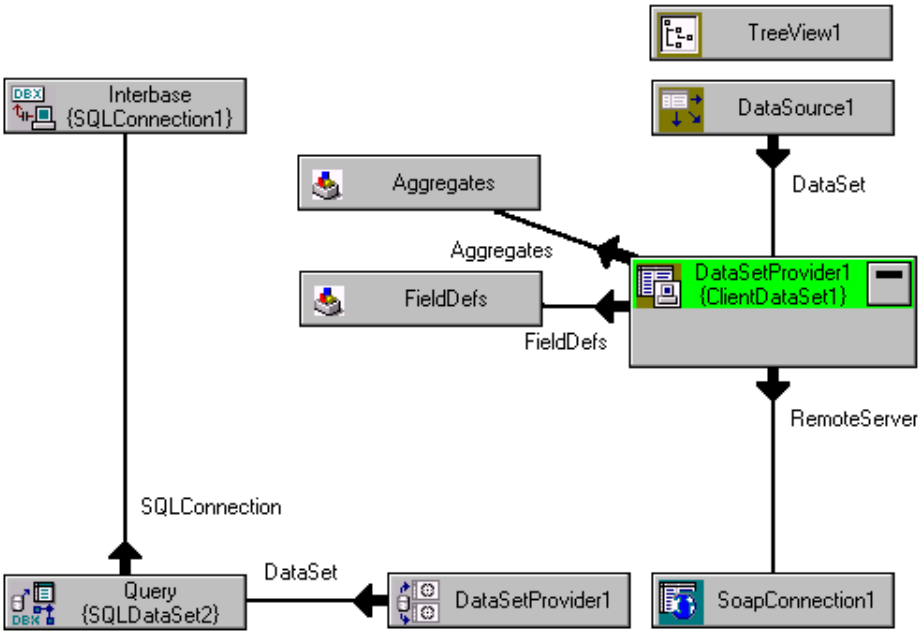


Abb. 6.40: MDA Code in der Zielplattform aus dem PSM

6.1 Kontakt

max@kleiner.com

6.2 Übungen

6.2.1 AD-Task

UEB: Einen Betrieb zu analysieren heisst, die Organisaton und deren Ablauf zu beschreiben. Aus zuerst undokumentierten, diffusen Vorgängen sollen abstrakte Modelle werden, die dann in eine konkrete Implementation münden. Stellen Sie sich «symbolisch» vor, der Betrieb hat folgenden Ablauf:
1. Notieren der betrieblichen Abläufe (Ist-Zustand und Analyse)

1
11
21
1211
111221
312211
Versuchen Sie nun diese Reihe fortzusetzen, suchen Sie nach einer Systematik oder Logik. Geben Sie sich 10 Minuten und Sie werden erstaunliches feststellen: In den meisten Fällen scheitert diese spezielle Folge und Reihe am falschen Denkansatz. Die Reihe lässt sich nicht berechnen, sondern man beschreibt einfach was man sieht, ähnlich einer Mustererkennung. Die Analyse eines Betriebes ist eher pragmatisch orientiert als berechenbar.

2. Modellieren der Lösung (Soll-Zustand und Design)
Die bestehende Zahlenreihe wird übernommen und entsprechend folgendem AD nach 1- und n-Zeichen durchgezählt, aggregiert und wieder übernommen (kopiert):

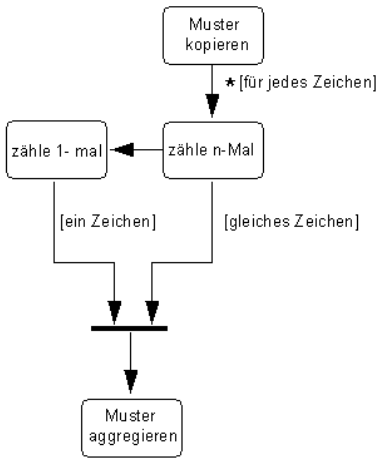


Abb. 6.41: AD-Lösung der Zahlenreihe

3. Implementation
Die Implementation erfolgt in ObjectPascal.

```
function TPattern.runString(Vshow: string): string;
var i: integer;
  Rword, tmpStr: string;
    cntr, nCount: integer;
begin
  cntr:=1; nCount:=0;
  Rword:='';  //initialize
  tmpStr:=Vshow;          // input last result
  for i:=1 to length(tmpStr) do begin
   if tmpStr[i]=tmpStr[i+1] then begin
     inc(cntr);
     nCount:= cntr;
```

```
   end else begin
     if cntr = 1 then nCount:=1 else cntr:=1;     //reinit counter!
     Rword:= Rword + intToStr(ncount) + tmpStr[i]     //+ last char(tmpStr)
   end;
  end; // end for loop
 result:=Rword;
end;
```

6.2.2 Kriterienvergleich

UEB: Vergleichen Sie die Kriterien zwischen den 2 Diagrammtypen CD (UC) und SEQ (AD) und geben eins bis drei Punkte zu den Kriterien, als Beispiel UC versus AD.

Kriterium	UC	AD
Daten-/Informationsfluss	o	o
Ereignisfluss	ooo	ooo
Ressourcen	oo	o
Nebenläufigkeit	o	ooo
Aufgabenhierarchie	oo	ooo
Formalisierungsgrad	oo	ooo
Verständlichkeit	oo	oo

6.2.3 CRC-Karte Auftrag

Wie Sie nun wissen, ist eine der zentralen Aufgaben des Objektorientierten Entwurfs das Finden der «richtigen» Klassen. Erstellen Sie eine weitere Klasse Artikel (Video) nebst der vorhandenen Klasse Auftrag in einer CRC-Karte mit denn Methoden und Eigenschaften. Wo sind vor allem die Collaborateurs, sprich die anderen Klassen, die mit dem Artikel interagieren.

Klasse: `Auftrag`	Abstrakt: ja / nein
Oberklasse:	
Unterklassen	
Inhalt, Verantwortung, Engagement:	Zusammenarbeit, Mitwirkung
- `Auftragsnummer`	
- `Beschreibung`	
- `Stundensatz`	
- `erteilen`	`Kunde, Artikel`
- `stornieren`	`Kunde, Artikel`
- `freigeben`	`Werk, Artikel`
- `kalkulieren (Stundenleistung)`	`Provision, Material`
- `beenden`	`Kunden, Rechnung`

Abb. 6.42: CRC-Karte Auftrag

6.2.4 Beziehungen Übersicht

Lange Zeit war die Terminologie der Klassenbeziehungen, gelinde gesagt, ein Tumult mit Struktur. Als Orientierung dient eine Tabelle welche «alte» Begriffe in die UML-Welt transformieren soll:

UEB: Skizzieren Sie zu jeder Beziehung je ein Beispiel mit der Notation und dem Symbol.

	Alt	Class	Verbal	UML	Symbol
1:	Uses	ja	benutzt von	association	
2:	Has a	ja	besteht aus	aggregation (composition)	
3:	Is a	ja	erbt von	generalization	
4:	Instance of	nein	hat Beziehung zu	link	

Merke: Jede Assoziation besitzt zwei Rollen. Jede Rolle ergibt sich auch als eine Richtung

der Assoziation. Beispiel: Die Assoziation zwischen Kunde und Auftrag enthält zwei Rollen, eine vom Kunden zum Auftrag, die zweite vom Auftrag zum Kunden.

6.2.5 Klassenbeziehungen und ihre Notation

UEB: Suchen Sie zu jedem Fall (1-4) die entsprechenden Diagramme.
Assoziationen zwischen Klassen werden (wenn die entsprechende Option vom Benutzer aktiviert wird) nach dem folgenden Verfahren generiert:

1. Verfügt die Klasse über ein Attribut, dessen Typ eine Referenz/ein Zeiger auf eine Klasse im Repository ist, so wird eine Assoziation zwischen den beiden Klassen erzeugt. Das Attribut wird der Rolle am der Klasse nächstliegenden Ende der Assoziation zugewiesen, und zwar als Implementierungsattribut. Es ist in dieser Richtung navigierbar, ihre Multiplizität ist unspezifiziert.
2. Haben zwei Klassen Attribute, die auf die jeweils andere Klasse verweisen, so wird statt zweier unidirektionaler Assoziation eine einzelne bidirektionale Assoziation erzeugt.
3. In C++/OP werden Assoziationen mit einer Multiplizität grösser 1 mit Hilfe von Container Template Klassen (List, Set, Bag, Array, ...) angelegt. Die so erzeugte Assoziation erhält die Multiplizität (0,*).
4. Soll verhindert werden, dass Assoziationen zu einer bestimmten Klasse erzeugt werden, so muss diese mit dem Stereotyp <<basic>> ausgezeichnet werden. Beispiele hierfür sind String-Klassen oder ähnliches.

6.2.6 Zustände einer Strassenampel / Schachspiel /Bankautomat

UEB: Modellieren Sie kurz die gültigen Zustände einer Strassenampel oder die Prozesszustände eines Betriebssystemes (ready, running, swapped, sleeped) oder eines Schachprogrammes. Überlegen Sie vor allem, welche Übergänge nicht möglich sind.

6.2.7 Nachrichten im Sequenzdiagramm

UEB: Überlegen Sie anhand der folgenden Liste, welche Nachrichtenart beim Starten dieser Dienste übermittelt wird (seq., synch. oder asynch.)

- `Alerter` `Running (Automatic)`
- `Computer Browser` `Running (Automatic)`
- `EventLog (Event log)` `Running (Automatic)`
- `TCP/IP NetBIOS Helper` `Running (Automatic)`
- `Messenger` `Running (Manual)`
- `FTP Publishing Service` `Running (Automatic)`
- `Net Logon (RemoteValidation)` `Running (Manual)`
- `Plug and Play (PlugPlay)` `Running (Automatic)`
- `Remote Procedure Call (RPC) Service` `Running (Automatic)`
- `Schedule` `Running (Automatic)`
- `Spooler (SpoolerGroup)` `Running (Automatic)`
- `Compaq System Shutdown Service` `Running (Manual)`

6.2.8 Packages als Pfad

UEB: Studieren Sie folgenden Batch und finden Sie das Java-Package heraus. Was bedeutet eigentlich <package> in Java?

```
@ECHO OFF
@CLS
ECHO.-------------------------------------------------------
ECHO                    VortexInterface Starten
ECHO.                      (C:\DING\VortexInterface\ch\)
ECHO  -------------------------------------------------------
echo on
rem VortexInterface laden, path setzen.................
rem path=%path%;c:\ding\java\jdk1.1.5\bin;c:\ding\java\bin
set CLASSPATH=.;c:\ding\java\classes\;c:\ding\java\jdk1.1.5\lib\
```

```
classes.zip
c:\ding\java\jdk1.1.5\bin\java.exe ch.dynasystem.io.vortex.VortexInterface
```

6.2.9 Klassen in Packages

Wieviel Klassen befinden sich im PAC der Abb. 3.24, S: 43 und welchen Zusammenhang haben die Klassen zu den Instanzen aus Abb. 3.20, S. 38 ?

6.2.10 Ereignisgesteuerte Systeme

In einem ereignisgesteuerten, sequentiellen System ist die Steuerung in einem Dispatcher oder Monitor enthalten, den die Sprache, das Teilsystem oder das Betriebssystem bereitstellt. An Ereignisse sind Anwendungsprozeduren geknüpft, die der Dispatcher aufruft, wenn das entsprechende Ereignis eintritt ('Callback'). Alle Prozeduren geben die Steuerung an den Dispatcher zurück, statt sie zu behalten, bis die Eingabe vorliegt. Ereignisse wickelt der Dispatcher direkt ab.
Der Programmzustand kann man nicht mit Hilfe des Befehlszählers und des Stacks zwischenspeichern, weil die Prozeduren die Steuerung an den Dispatcher zurückgeben. Prozeduren müssen globale Variablen verwenden um ihren Zustand zu speichern, oder der Dispatcher muss jeweils einen lokalen Zustand mitführen (echt OO-mässig : Ereignis -> Methode in einem Objekt (Interaktion), -> Methode ändert Objektzustand). Welches Diagramm ist ideal für solch ein «event driven system»

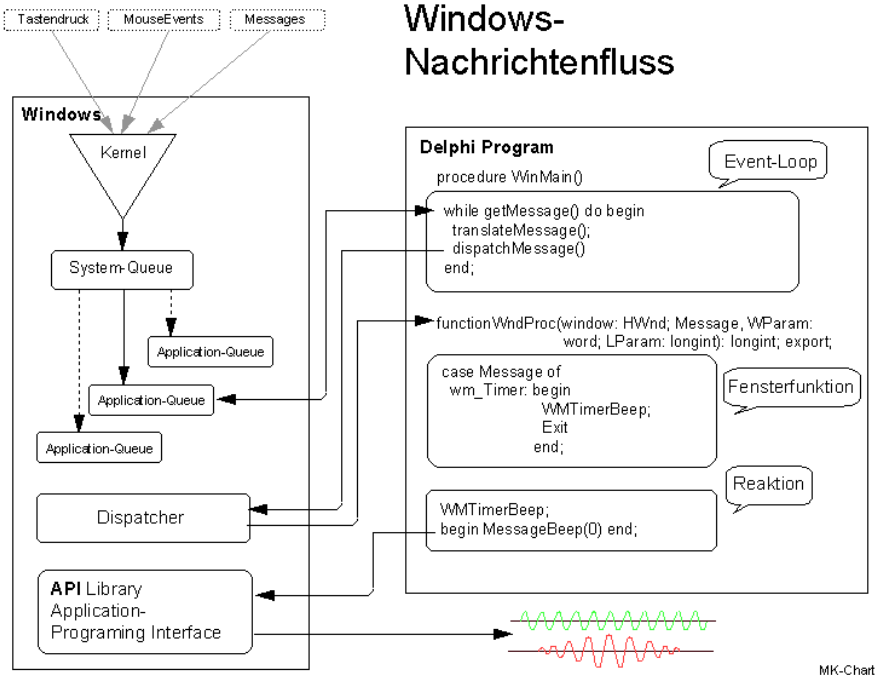


Abb. 6.43: Konvertieren von Free-Style in eine UML-Notation

6.2.11 Taskzuteilung zum V-Modell

UEB: Ordnen Sie die folgenden Tätigkeiten eines SW-Projektes den 4 Domänen (Aufgabengruppen) des V Modells zu:

- Dokumentation, Hilfe und Handbuch
  - Module und Anforderung
  - Schnittstellen
  - Klassen und Datenstrukturen
- Versionsverwaltung
  - PVCS, Revision,Version
  - Archiv und Backups
  - Projekt…

- Testen
  - Testplan
  - Testpersonen (gegentesten)
  - Tools wie Boundschecker, Purify oder Heapwalker
- Internationalisierung
  - Forms...
  - DataDrivenTables
  - Hilfe, Handbuch und Installation
- Installationsroutine
  - Einbinden der BDE
  - C/S oder Ferninstallation
- Updates
  - Customizing
  - Releasing

6.2.12 Zuordnen der Diagramme

UEB: Ordnen Sie alle UML-Diagramme in den Würfel ein.

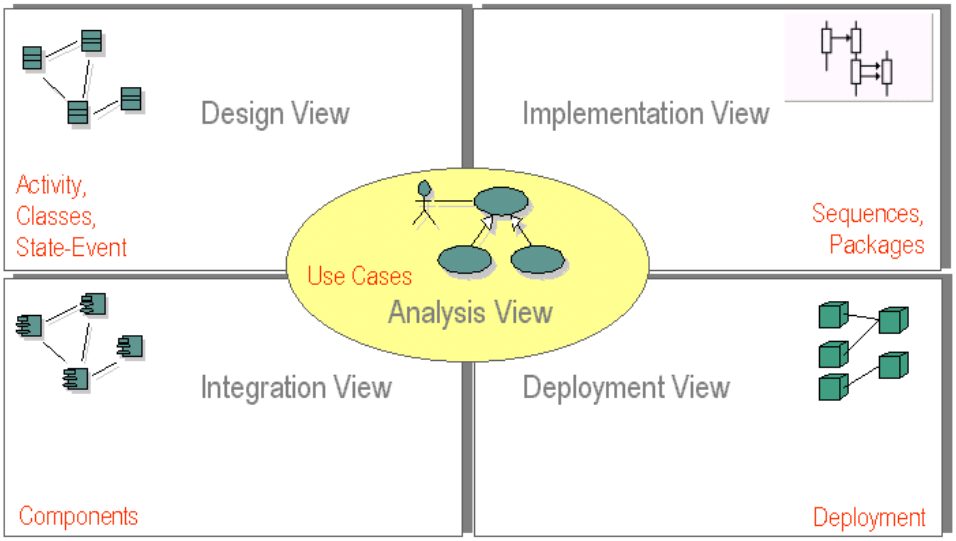


Abb. 6.44: Die verschiedenen Perspektiven auf ein System

6.3 Quellenverzeichnis:

[UML1.1] UML Extension for Objectory Process for Software Engineering, version 1.1, 1.9.97

[Sche94] Scheer A.W.: Wirtschaftsinformatik - Referenzmodelle für industrielle Geschäftsprozesse, 5. Aufl.,Springer-Verlag

[KNS92] Keller, Nüttgens, Scheer, Semantische Prozessmodellierung auf der Grundlage EPK. Uni Saarbrücken

[BuHe95] Bungert, Hess: Objektorientierte Geschäftsprozessmodellierung. Information Management 10, 1995

[Oest97/2] Oestereich B., Objektorientierte Softwareentwicklung: Analyse und Design, 2. Auflage 1997, S. 87, Oldenbourg Verlag

[FS98] Fowler, Scott: UML Distilled, Addison-Wesley 1998

[MK98] Max Kleiner: Einheitlichkeit im Methodendschungel ?, Der Entwickler 3.98

[GHJV] Gamma, Helm etc.: Entwurfsmuster, Addison-Wesley 1996, S. 26

[UMLTool] Erikkson, Penker: UML Toolkit, Wiley&Sons 1998

6.4 Bildverzeichnis

[i] Fowler, Scott: UML Distilled, Addison-Wesley 1998

[ii] Gamma, Helm etc.: Entwurfsmuster, Addison-Wesley 1996

[iii] Kleiner et al: Patterns konkret, S&S Verlag 2003

[iv] Odell James J.:Advanced OO Design Using UML, SIGS Books 1998

[v] www.delphi3000.com - Exception or Event Logger