

SEMINARARBEIT

Implementierung des Design Patterns *Command*  
zur Datenbankanbindung des Geschäftsanalysetools  
*PAT*

von

Christian Eichenberger  
`christianmarkus.eichenberger@unifr.ch`

Universität Freiburg i.Ü./CH

Referent: Prof. Dr. Andreas Meier  
Betreuer: Daniel Frauchiger

Freiburg, Herbst 2002

# Abstract

## **Weiterentwicklung des Geschäftsprozess-Analysetools *PAT***

In Zusammenarbeit mit einem international tatigen Beratungs- und Revisionsunternehmen ist ein in Java geschriebener Prototyp entstanden, der die graphische Modellierung von Geschaftsprozessen erlaubt und dank seines generischen Ansatzes die Analyse der Prozesse auf verschiedenste Kriterien ermoglicht. Diese Seminararbeit hat zum Ziel, dem existierenden Prototypen eine Datenbank zu unterlegen: In einem ersten Schritt wird die Datenstruktur modelliert, in einem zweiten die Datenbankanbindung implementiert. Gleichzeitig bekommt die Befehlsstruktur ein neues, sauberes Design.

# Inhaltsverzeichnis

<b>Abstract</b>	<b>1</b>
<b>Inhaltsverzeichnis</b>	<b>2</b>
<b>1 Einleitung</b>	<b>3</b>
1.1 Problemstellung	3
1.2 Zielsetzung	3
1.3 Aufbau	3
<b>2 Datenmodell</b>	<b>5</b>
2.1 <i>APN</i>	5
2.1.1 Ablaufdarstellung	5
2.1.2 Abstraktion und Hierarchisierung	6
2.2 Formalisierung im Entitäten-Beziehungsmodell	10
2.3 Vereinfachung des Schemas	12
2.4 Überführen des Entitäten-Beziehungsmodells in ein relationales Datenbankschema	14
2.5 Exkurs: Petri-Netze	15
<b>3 Design und Implementierung der Datenbankanbindung</b>	<b>19</b>
3.1 Vorüberlegungen zur Datenbankanbindung	19
3.2 Die Klasse <code>Command</code>	21
3.2.1 Das Design Pattern <i>Command</i>	21
3.2.2 Das Interface der Klasse <code>Command</code>	21
3.2.3 Die <i>ConcreteCommands</i>	22
3.3 Die Klasse <code>History</code>	25
3.3.1 Das Interface der Klasse <code>History</code>	25
3.4 Die Klasse <code>dbConnection</code>	27
3.4.1 Das Design Pattern <i>Adapter</i>	27
3.4.2 Das Interface der Klasse <code>dbConnection</code>	28
<b>4 Konklusion</b>	<b>30</b>
<b>Literatur, Soft- und Hardwareverzeichnis</b>	<b>31</b>
<b>Abbildungsverzeichnis</b>	<b>33</b>
<b>Tabellenverzeichnis</b>	<b>34</b>

# Kapitel 1

## Einleitung

### 1.1 Problemstellung

*PAT* – Akronym für *Process Analysis Tool*<sup>1</sup> – wurde als Prototyp ohne Datenmodellierung angegangen; deshalb müssen nachträglich Begriffe fixiert und die Daten modelliert werden.

Zudem erlaubt es *PAT* nicht, die eingegebenen Daten in einem anderen Anwendungen zugänglichen Format abzulegen. Dies soll durch das Speichern der erzeugten Diagramme in einer MySQL-Datenbank ermöglicht werden.

### 1.2 Zielsetzung

In einem ersten Schritt soll die *PAT* zugrundeliegende Datenstruktur modelliert werden, sodass der Implementierung der Datenbankbindung ein sauberes Datenmodell zugrunde liegt. Dieses soll einerseits dazu dienen, die Möglichkeiten von *PAT* genau festzulegen, und andererseits dazu, das relationale Datenbankschema zu generieren.

Die in *PAT* erstellten Diagramme sollen in eine MySQL-Datenbank abgebildet und wieder eingelesen werden können. Für die Aufbereitung der Daten soll ein geeigneter Mechanismus verwendet werden, der es erlaubt, die Datenbankbindung sauber implementieren zu können, und der für den Benutzer von *PAT* möglichst viel Bedienungskomfort bietet. Dies kann wie folgt geschehen: Die durch das Benutzer-Interface eingegebenen Befehle werden aufgezeichnet – dann können auf Befehl die Veränderungen am Diagramm in die Datenbank gespeichert werden oder bei Bedarf auch Modifikationen innerhalb von *PAT* widerrufen werden.

### 1.3 Aufbau

In Kapitel 2 wird das Datenmodell besprochen, aus dem sodann die Tabellen für eine relationale Datenbank abgeleitet werden. Es wird zuerst ein Modell erklärt, das vollständige Diagramme beschreibt; in einem zweiten Schritt wird

---

<sup>1</sup> Dt.: *Prozessanalysewerkzeug*

dieses Modell so gelockert, dass auch Diagramm-Entwürfe gespeichert werden können.

In Kapitel 3 werden wichtige Design-Entscheidungen der Datenbankanbindung erläutert: Die Befehle, die Daten erstellen oder verändern, müssen bestimmt werden; der Widerrufungsmechanismus muss organisiert werden; Datenmodifikationen müssen in entsprechende Befehle an die Datenbank übersetzt werden können.

Ein kurzes Konklusionskapitel fasst die wichtigsten Punkte der Arbeit noch einmal zusammen.

## Kapitel 2

# Datenmodell

### 2.1 *APN*

Bei dem *PAT* zugrundeliegenden Modell handelt es sich um eine aus den *Petri-Netzen* abgeleitete Notation mit dem inoffiziellen Akronym *APN* (für *Actually Petri Net*<sup>1</sup>). Es handelt sich um eine den Petri-Netzen äquivalente Notation, die versucht, einfacher verständlich zu sein. Auf die Frage der Transformationsregeln kann in dieser Arbeit nicht eingegangen werden; sie sind Forschungsgegenstand der Gruppe für Informationssysteme des Departements für Informatik an der Universität Freiburg.

Anhand von *APN* können Geschäftsabläufe dargestellt werden; durch Verzweigungen und Vereinigungen unter bestimmten Eingangs- und Ausgangsbedingungen kann der Fluss gesteuert werden.

#### 2.1.1 Ablaufsdarstellung

Die zentralen Elemente eines Geschäftsablaufes sind *Aktivitäten*; sie beschreiben je eine isolierte Tätigkeit durch verschiedene Parameter wie Input, Output, Verantwortlichkeiten, Risiko etc. Ein Geschäftsablauf in *APN* besteht – grob gesagt – aus einer gerichteten, beidseits terminierten Verknüpfung von Aktivitäten.

Um Ablaufvarianten in das Modell einzubinden, gibt es nicht nur *einfache Aktivitäten*, d.h. Aktivitäten mit genau einem Ein- und Ausgang, sondern auch *komplexe*, d.h. Aktivitäten mit mehreren Ein- und Ausgängen. *APN* bietet zwei Typen von komplexen Aktivitäten an: Einerseits den *Kollektor*, dessen Eingänge UND- und dessen Ausgänge ODER-geschaltet sind;<sup>2</sup> andererseits den *Distributor*, dessen Eingänge ODER- und dessen Ausgänge UND-geschaltet sind; bei letzterem Typ sind die Ausgänge mit *Kriterien* besetzt, sodass jeweils genau ein Ausgang geschaltet wird. Abbildung 2.1 zeigt die graphische Repräsentation eines *APN*-Diagrammes anhand eines einfachen Beispiels: Einfache Aktivitäten werden durch ein Rechteck dargestellt, Kollektoren durch eine entgegen der Flussrichtung beflaggte und Distributoren durch eine in Flussrichtung beflaggte Linie, die quer zum Fluss liegt; der Fluss in einem Task beginnt genau an einem Punkt und mündet in einem Punkt – der *Start* wird graphisch als un-

---

<sup>1</sup> Dt.: *Eigentlich Petri-Netz*

<sup>2</sup> Hier wie weiter unten ist immer ein exklusives ODER (*XOR*) gemeint.

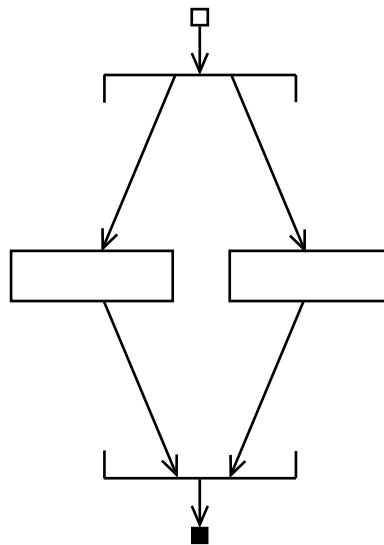


Abbildung 2.1: Beispiel für die Flusssteuerung in APN.

ausgefülltes Quadrat, das *Ende* eines Tasks als ausgefülltes Quadrat dargestellt; die Abbildung zeigt also einen Fluss, der sich zuerst verzweigt und dann wieder zusammengeführt wird.

Abbildung 2.2 zeigt Kombinationen von komplexen Aktivitäten und ihre Interpretation.

### 2.1.2 Abstraktion und Hierarchisierung

Ein von anderen Abläufen organisatorisch eigenständiger Geschäftsablauf heisst *Projekt*; sein Ablauf ist unabhängig von dem anderer Projekte. Teile eines Projektes können zusammengefasst und abstrahiert werden zu einem *Task*. Ein Projekt insgesamt wird auch als Task betrachtet, der jedoch per definitionem selber nicht Teil eines übergeordneten Tasks sein kann; in Baumdarstellung können die Knoten als Tasks, die Äste als Abstraktions- und Hierarchisierungsbeziehungen und der Wurzelknoten als Projektstask, angesehen werden.

Durch eine Aggregation von Tasks kann also ein Projekt logisch strukturiert (i.e. ein Teil bildet eine Abstraktionseinheit) und hierarchisiert (i.e. gewichtet) werden.

Schematisch kann der Abstraktions- und Hierarchisierungsmechanismus durch Tasks wie in Abbildung 2.3 verbildlicht werden. Abbildung 2.4 nimmt das Beispiel aus Abbildung 2.1 noch einmal auf, fasst jedoch alle vorhandenen Aktivitäten zusammen in einer abstrakten Aktivität und ordnet sie in einem neuen Task unter.

Untergeordnete Tasks erscheinen im übergeordneten Task als *abstrakte Aktivitäten*: Dem übergeordneten Task erscheint eine abstrakte Aktivität syntaktisch wie eine einfache Aktivität; semantisch sind es aber erweiterte Aktivitäten, denn sie nehmen auch die Eigenschaften des Tasks, den sie repräsentieren, an.

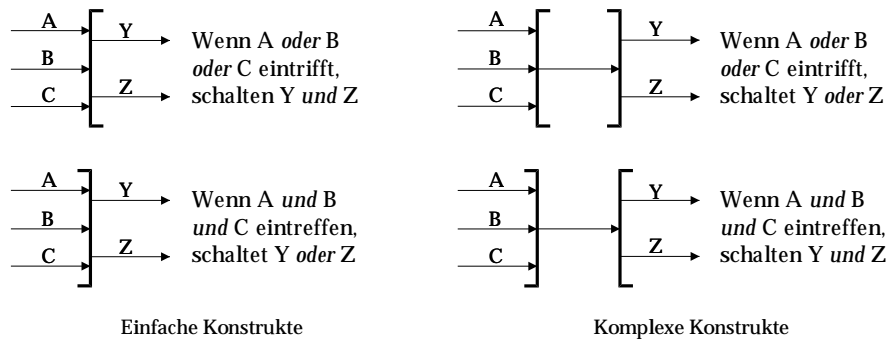


Abbildung 2.2: Mögliche APN-Konstrukte und ihre Interpretation. Aus: [Frauchiger, 78].

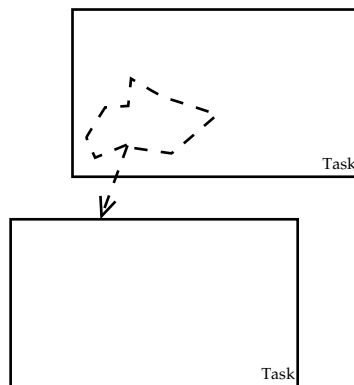


Abbildung 2.3: Unterordnung eines Teiles eines Tasks.








	Start
	Ende
	einfache Aktivität
	abstrakte Aktivität
	Distributor
	Kollektor
	Verknüpfung

Tabelle 2.1: Graphische Repräsentation der *APN*-Elemente.

Abstrakte Aktivitäten werden graphisch durch Rechtecke mit gestrichelten Kanten dargestellt.

Tabelle 2.1 fasst die graphischen Darstellungen der verschiedenen Elemente eines *APN*-Diagrammes zusammen.

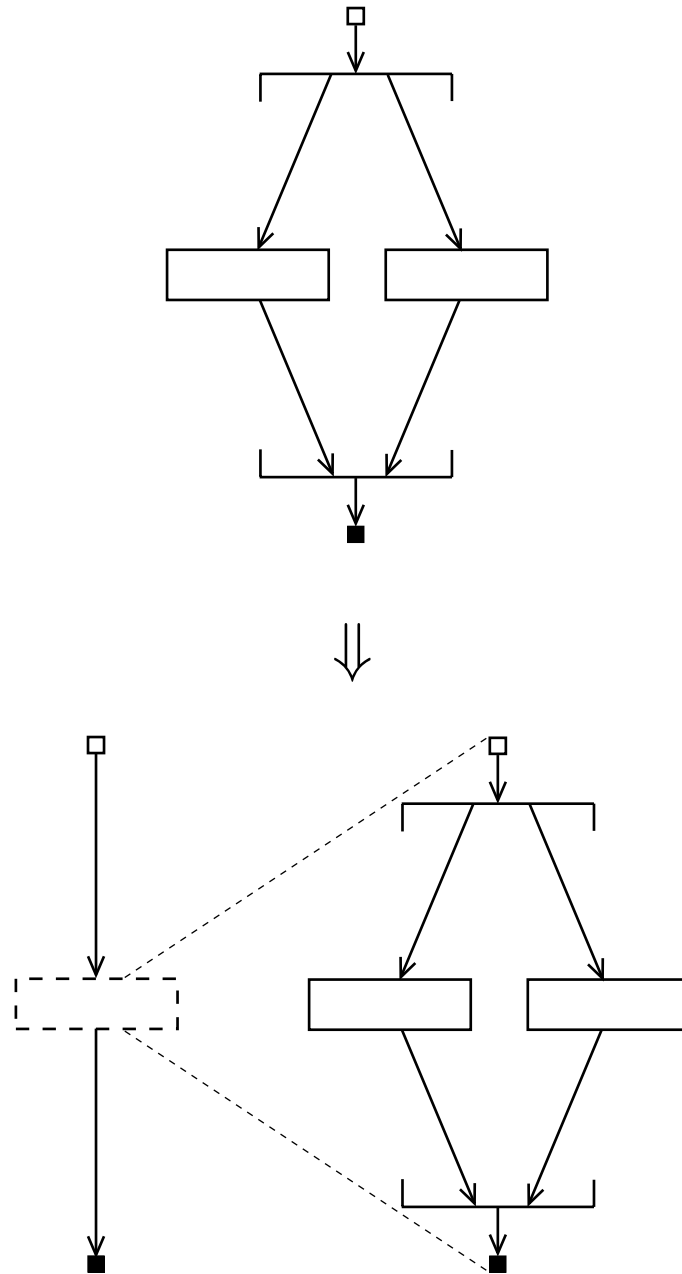


Abbildung 2.4: Abstraktion und Hierarchisierung eines Tasks.

## 2.2 Formalisierung im Entitäten-Beziehungsmodell

Der Zwischenschritt zwischen Datenanalyse und relationalem Datenbankschema (also den Tabellen) besteht darin, die Daten in einem Entitäten-Beziehungsmodell darzustellen ([Meier, 13ff.]).

Dieser Schritt hat den grossen Vorteil, dass die Datenmodellierung unabhängig von Tabellen geschehen kann, wobei jedoch die anschliessende Überführung in Tabellenform nicht verkompliziert wird, da einfache Abbildungsregeln bekannt sind.

Ein Entitäten-Beziehungsmodell kann in ein UML-Klassendiagramm überführt werden anhand der Regeln in Tabelle 2.2. (In der Gegenrichtung bestehen Einschränkungen, weil UML-Klassendiagramme Möglichkeiten haben, die in einem Entitäten-Beziehungsmodell nicht ausgedrückt werden können; cf. nächster Abschnitt). Im folgenden werden deshalb die sich entsprechenden Begriffe synonymisch verwendet; so wird z.B. über Abbildung 2.5 in beiden Terminologien gesprochen werden können. Die UML-Notation bietet sich auch deshalb an, weil *PAT* ja in der objektorientierten Programmiersprache Java implementiert ist. Für Details zu UML sei auf [Fowler] verwiesen.

Das UML-Klassendiagramm in Abbildung 2.5 zeigt einen Datenmodellierungsvorschlag für *APN*;<sup>3</sup> es macht sowohl syntaktische, wie auch inhaltliche Zusammenhänge zwischen den Elementen von *APN* klar. Auf der inhaltlichen Seite zu erwähnen ist die Vererbung der Aktivitätseigenschaften auf alle vier Aktivitätstypen (resp. die Generalisierung der Aktivitätstypen), sowie die Repräsentationsbeziehung zwischen abstrakter Aktivität und Task.

Der syntaktische Aspekt der Aktivitäten wird durch die Vererbung der *Connectee*-Eigenschaften deutlich gemacht. Im weiteren zeichnet sich ein Task durch genau einen *Start* und ein *Ende* aus, die über eine Kette von Connectees durch *Verknüpfungen* verbunden sind. Um alle Verknüpfungsmöglichkeiten von komplexen und von nicht-komplexen Connectees genau zu erfassen, sind *Ein- und Ausgang* von den Aktivitäten getrennt als abstrakte Klassen modelliert; die konkreten Unterklassen sind dann jeweils für einfache oder mehrfache Zugänge angelegt.

Entitäten-Beziehungsmodell	UML-Klassendiagramm
Entität	Klasse
Relation	Assoziation
Vererbung	Generalisierung
disjunkt-vollständige Generalisierung	abstrakte Vererberklasse
Attribut	Attribut

Tabelle 2.2: Entsprechungen im Entitäten-Beziehungsmodell und im UML-Klassendiagramm.

<sup>3</sup> Ausgelassene Beschriftungen der Assoziationen in UML-Diagrammen sind als nicht näher benennbare “hat”- oder “gehört zu”-Beziehungen zu verstehen.

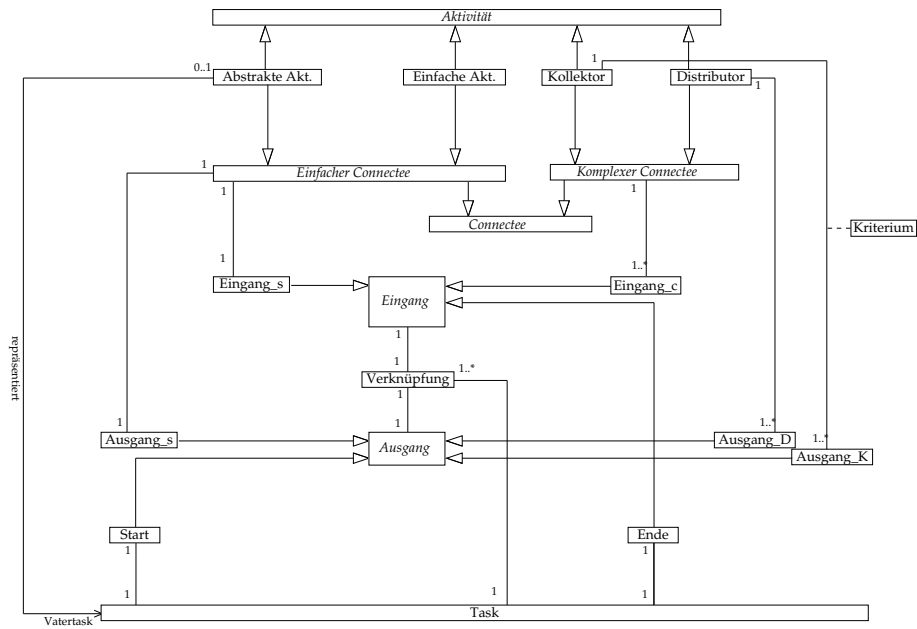


Abbildung 2.5: Vorschlag eines Entitäten-Beziehungsmodells für APN.

## 2.3 Vereinfachung des Schemas

Das Schema in Abbildung 2.5 beschreibt vollständige *APN*-Diagramme; in der Praxis sollen aber auch Entwürfe in die Datenbank abgebildet werden können. Folgende Mechanismen sind denkbar zur Abänderung des Schemas:

1. *Lockerung des Schemas*: Z.B. Ersetzen einer einfachen Relation durch eine komplexe. Zur Einhaltung der Regeln müssen die Lockerungen durch datenbankexterne Vorkehrungen aufgefangen werden.
2. *Restriktion des Schemas*: Z.B. Ersetzen einer komplexen Relation durch eine einfache. Zur Einhaltung der betroffenen Regeln müssen keine besonderen Vorkehrungen getroffen werden.

Folgende Punkte sind “vernünftige” Vereinfachungen und setzen einen der oben genannten Mechanismen um:

- *Lockerung der Verbindungssyntax* im Entitäten-Beziehungsmodell: Die Unterscheidung zwischen einfachen und komplexen Connectees wird aufgehoben. Dass aber einfache und abstrakte Aktivitäten nur einen Vorgänger und einen Nachfolger haben dürfen, wird durch Einführen von *UML-constraints* ins Schema gesichert (in einem UML-Diagramm durch geschwungene Klammern um natürlichsprachigen Text angezeigt); diese können dadurch eingehalten werden, dass die Verantwortung an *PAT* übergeben wird, i.e. *PAT* darf UML-constraints verletzende Eingaben weder annehmen, noch an die Datenbank weiterleiten.
- In einem Task sollen auch *unverknüpfte* (bzw. unvollständig verknüpfte) *Aktivitäten* vorkommen dürfen. Dies ist eine nötige Lockerung, da man auch einen unvollständig ausgearbeiteten Entwurf speichern können möchte. Sobald alle Aktivitäten korrekt verknüpft sind, erfüllt aber das Diagramm die *APN*-Regeln. Dies bedingt, dass die Zugehörigkeit einer Aktivität zu einem Task nicht über die Verkettung der Aktivitäten (und schliesslich über den Start oder das Ende eines Tasks), sondern über die einzelnen Aktivitäten ausgedrückt werden muss.
- Ein Task soll *mindestens eine Aktivität* enthalten: Jedes Diagramm mit dieser Einschränkung erfüllt die *APN*-Regeln. Die Start- und die Endaktivität eines Tasks sind im Falle nur einer Aktivität identisch.

Mit diesen drei Massnahmen kann das zu implementierende Modell beträchtlich vereinfacht werden, wie Abbildung 2.6 zeigt. Insbesondere kann damit die Zahl der Generalisierungen vermindert werden, was im Umgang mit relationalen Datenbanken einen Vorteil darstellt.

Folgende Bemerkung zur Handhabung des *Kriteriums* drängt sich auf: In der ersten Version des Entitäten-Beziehungsmodells wird das Kriterium als Attribut eines Ausganges eines Kollektors betrachtet, in der vereinfachten als Attribut jeder Verknüpfung zweier Aktivitäten, wobei dann die nicht von Kollektoren ausgehenden Verknüpfungen mit leeren Kriterien zu versehen sind. Es ist vielleicht eine unentscheidbare Frage, welche Version “recht hat”, denn die Kriterien sind inhärent systemfremd (nicht in ein Element des Systems “einbaubar”), da sie vom aktuellen Systemzustand abhängig entschieden werden müssen (z.B. durch Benutzereingabe).

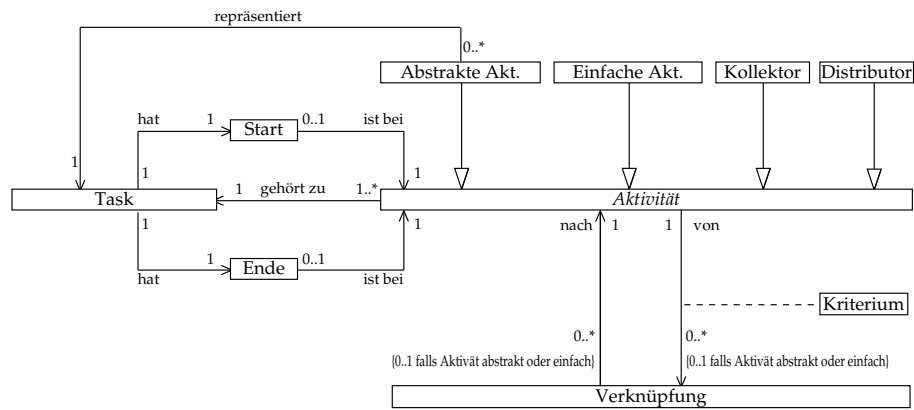


Abbildung 2.6: Vereinfachtes Entitäten-Beziehungsmodell für APN.

## 2.4 Überführen des Entitäten-Beziehungsmodells in ein relationales Datenbankschema

Anhand der Regeln in [Meier, 25ff.] kann das vorgestellte Modell in Tabellen überführt werden – ausgedrückt in tabellendefinierenden MySQL-Befehlen sieht das folgendermassen aus:

```
CREATE TABLE task
(
    id INT NOT NULL AUTO_INCREMENT,
    abstract_id INT,
    name varchar (60),
    teamManager varchar(60),
    teamMembers varchar(60),
    annotations varchar(200),
    date varchar (30),
    PRIMARY KEY (id)
) TYPE=INNODB;

CREATE TABLE activity
(
    id INT NOT NULL AUTO_INCREMENT,
    type ENUM ('simple', 'abstract', 'collector',
              'distributor') NOT NULL,
    taskC INT NOT NULL,
    input varchar(100),
    output varchar(100),
    place varchar(100),
    agent varchar(100),
    viceAgent varchar(100),
    activity varchar(100),
    target varchar(100),
    annotation varchar(100),
    preCondition varchar(100),
    postCondition varchar(100),
    risk varchar(100),
    isControl BOOL,
    periode INT,
    controls0 BOOL,
    controls1 BOOL,
    controls2 BOOL,
    controls3 BOOL,
    controls4 BOOL,
    controls5 BOOL,
    controls6 BOOL,
    X INT,
    Y INT,
    PRIMARY KEY (id),
    FOREIGN KEY (taskC) REFERENCES task (id)
) TYPE=INNODB;
```

```

CREATE TABLE criterion
(
  id INT NOT NULL AUTO_INCREMENT,
  collector INT NOT NULL,
  criterion varchar(200),
  PRIMARY KEY (id),
  FOREIGN KEY (collector) REFERENCES activity (id)
) TYPE=INNODB;

```

```

CREATE TABLE connection
(
  id INT NOT NULL AUTO_INCREMENT,
  start INT NOT NULL,
  stop INT NOT NULL,
  PRIMARY KEY (id),
  FOREIGN KEY (start) REFERENCES activity (id),
  FOREIGN KEY (stop) REFERENCES activity (id)
) TYPE=INNODB;

```

Zwei Anmerkungen zu diesen Tabellendefinitionen:

- *Enumerationstyp und erste Normalform*: Um eine Tabelle einzusparen, die einfach die Aktivitätstypen auflisten würde und deren Einträge dann über einen Fremdschlüssel aus der Tabelle `activity` referenziert werden müssten, scheint diese Verletzung der ersten Normalform vertretbar; auch die verschiedenen Aktivitätstypen in vier zusätzlichen Tabellen zu führen (je eine pro Typ), macht keinen Sinn, solange diese alle dieselben Attribute haben.
- *Repräsentations-Beziehung zwischen Tasks und abstrakten Aktivitäten*: Nach [Meier, 30] ist es natürlicher, eine einfach-konditionale Beziehung in der Tabelle der Entität mit dem einfachen Assoziationstyp auszudrücken (durch eine Referenz in die Tabelle der Entität mit dem konditionalen Assoziationstyp), weil dann Nullwerte vermieden werden können.<sup>4</sup> Im Fall der Repräsentations-Beziehung kann aber durch die andere Option eine Generalisierung verhindert werden (cf. oben, Abschnitt 2.3), indem auch die abstrakten Aktivitäten nur durch die Tabelle `activity` ausgedrückt werden.

## 2.5 Exkurs: Petri-Netze

Da *APN* auf Petri-Netzen basiert, kann ein Vergleich einige Aspekte von *APN* noch von einer andern Seite beleuchten.<sup>5</sup>

<sup>4</sup> I. Ggs. zu UML wird im Entitäten-Beziehungsmodell die Multiplizität nicht bei der Entität, auf die sich die Mengenangabe bezieht, angegeben (cf. [Fowler, 53] und [Meier, 18ff.]).

<sup>5</sup> Dieser Abschnitt basiert auf [Frauchiger, 76ff.], wo *APN* entwickelt worden ist.

	Stelle
	Transition
	Verbindungspfeil

Tabelle 2.3: Graphische Repräsentation der Elemente von Petri-Netzen.

Syntaktisch gesehen bestehen Petri-Netze aus *Stellen* (graphisch dargestellt als Kreise) und *Transitionen* (graphisch dargestellt als Rechtecke); *Verbindungspfeile* führen jeweils von einer Stelle zu einer Transition oder von einer Transition zu einer Stelle; die einzelnen Stellen und Transitionen können je mehrere ein-, bzw. ausgehende Verbindungspfeile haben. Tabelle 2.3 stellt die Elemente der Petri-Netze synoptisch dar.

Die semantische Interpretation ist folgende: Eine Stelle kann erreicht, resp. verlassen werden, sobald mindestens ein Ein-, resp. Ausgang geschaltet ist; eine Transition wird hingegen erst aktiv, sobald alle Eingänge geschaltet sind, i.e. sobald alle Inputargumente, die die Transition benötigt, vorhanden sind – analog werden alle Ausgänge der Transition geschaltet.

Um Geschäftsabläufe modellieren zu können, ist es aber oft praktischer, anstatt von Stellen und Transitionen, von *Verteil- und Sammelpunkten* auszugehen: Verteilstellen leiten eintreffende Signale unmittelbar an alle Ausgänge weiter, Sammelpunkte leiten ankommende Signale erst beim Eintreffen aller Signale an genau einen Ausgang (von möglicherweise mehreren) weiter.

Für *APN* heisst dies erstens, dass die Kreise der Petri-Netze in die Pfeile von *APN* integriert sind; zweitens werden mehrere Aktivitäten unterschieden. Abbildung 2.7 zeigt ein einfaches Beispiel der Transformation eines Petri-Netzes in *APN*; die gestrichelten Pfeile deuten die einzelnen Transformationsschritte an.

Aus dieser Herleitung von *APN* folgt, dass die Verbindungspfeile in *APN* als Entitäten zu betrachten sind, und nicht nur als Verbinder von Stellen und Transitionen wie in den Petri-Netzen.

Tabelle 2.4 stellt die (syntaktischen) Möglichkeiten der Petri-Netze und von *APN* einander gegenüber; dabei steht in der dritten Spalte die *Implikation* auf die Eingänge links und diejenige auf die Ausgänge rechts von einem Gedankenstrich; die *Eigenschaft* eines Elementes gibt an, ob es bestimmen kann, welche Ein-, bzw. Ausgänge geschaltet werden, oder nicht; die nicht-komplexen Aktivitäten sind als Spezialfälle von Kollektoren oder Distributoren mit jeweils nur einem Ein- und Ausgang und wegen ihrer Herleitung aus den Transitionen (wohl) als *aktiv* zu betrachten, obwohl die Wahl auf nur eine Möglichkeit beschränkt ist – aus letzterem Grund kann (wohl) auch eine UND-Implikation angenommen werden.

Ein Punkt, wo *APN* die Petri-Netze erweitert, ist der Abstraktions-, bzw. Hierarchisierungsmechanismus durch eine Taskaggregation (cf. [Frauchiger, 76]). Auch dieser Punkt schränkt die Kompatibilität von *APN* zu *PN* nicht ein, da

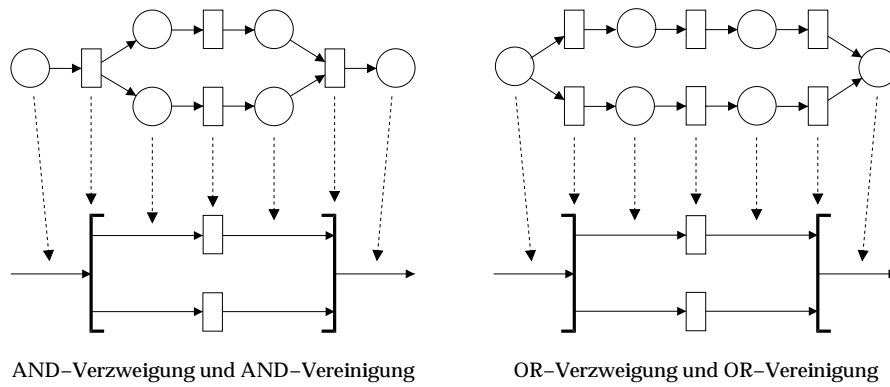


Abbildung 2.7: Herleitung von *APN* aus den Petri-Netzen. Aus: [Frauchiger, 77].

Modell	Element	Implikation	Eigenschaft
<i>PN</i>	Transition	UND-UND	aktiv
	Stelle	ODER-ODER	passiv
	Pfeil	Fluss	passiv
<i>APN</i>	einfache/abstrakte A.	(UND-UND)	(aktiv)
	Kollektor	UND-ODER	aktiv
	Distributor	ODER-UND	aktiv
	Pfeil	Fluss+Stelle	passiv

Tabelle 2.4: Gegenüberstellung der Elemente der Petri-Netze (*PN*) und von *APN*.

die Inhalte der untergeordneten Tasks rekursiv in die übergeordneten Tasks integriert werden können – natürlich unter Verlust der Information über diese Inhalte als abstrahierte Einheiten.

## Kapitel 3

# Design und Implementierung der Datenbankanbindung

### 3.1 Vorüberlegungen zur Datenbankanbindung

In diesem Kapitel sollen grundlegende Überlegungen zur Datenbankanbindung gemacht und die Wahl des unten beschriebenen Designs erklärt werden.

Für eine Datenbankanbindung gibt es grundsätzlich zwei Möglichkeiten:

1. Es wird direkt auf der Datenbank gearbeitet, d.h jede Modifikation muss von der Datenbank angenommen sein, bevor sie gültig ist.
2. Es wird grundsätzlich mit den Daten im Hauptspeicher gearbeitet und erst bei explizitem Befehl in die Datenbank geschrieben.

Im zweiten Fall sind wiederum zwei Szenarien denkbar:

1. Die im Hauptspeicher befindliche Struktur wird als ganze einem Überföhrungsalgorithmus übergeben. Obwohl keine Informationen über die Zwischenschritte vorhanden sind, müssen die Struktur in der Applikation und die in der Datenbank abgeglichen werden:
  - *Komplett*: Die Struktur wird komplett in die Datenbank geschrieben, die alte Version wird nachträglich (oder ev. vorgängig) gelöscht. Dies ist zwar einfach zu implementieren, bei jedem Speichern werden jedoch neue Datensätze (mit neuen Primärschlüsseln) angelegt für schon vorhandene, die im Gegenzug gelöscht werden müssen; dies verletzt die Idee, dass ein Objekt in der Datenbank nur gelöscht werden darf, wenn das entsprechende in der Applikation gelöscht worden ist.
  - *Individuell*: Die beiden Strukturen müssen in beide Richtungen verglichen werden, Neues muss ergänzt, Veraltetes muss erkannt und gelöscht werden. Dies ist kompliziert und fehleranfällig, weil der Weg, der vom Zustand in der Datenbank zum Zustand in der Applikation

geführt hat, nicht bekannt ist; insbesondere reicht es nicht, nur die Objekte in der Applikation zu betrachten, da ja in der Applikation gelöschte Objekte so nicht erkannt und dann in der Datenbank gelöscht werden können; vielmehr müssen dafür alle Datensätze nach ihrer Existenz hin überprüft werden.

2. Ausgangspunkt für die Überführung ist nicht der aktuelle Zustand im Hauptspeicher, sondern es werden die Modifikationsschritte, die zum aktuellen Zustand geführt haben, aufgezeichnet. Für jeden Einzelschritt kann bei Bedarf die entsprechende Modifikation auf die Datenbank ausgeführt werden und so die Datenbank in den aktuellen Zustand gebracht werden. Dies hat einerseits den Vorteil, dass die einzelnen Schritte isoliert implementiert und getestet werden können, so dass Fehler besser lokalisierbar sind. Andererseits ist es dann einfach, einen Widerrufungsmechanismus zu implementieren, weil ja für jeden Schritt Informationen für die Datenbanktransaktion gesammelt werden; je nachdem müssen zusätzliche Informationen für das Widerrufen angelegt werden (z.B. müssen Referenzen auf gelöschte Objekte erhalten bleiben) – dies kann aber am gleichen Ort geschehen wie für das Speichern. Ein Problem dieses Ansatzes könnte die Performanz sein, denn wenn die Granularität der Kommandos zu klein ist, kann es sein, dass die Modifikationsgeschichte viele kleine Kommandos enthält, die etwas Ähnliches machen – z.B. pro Attribut einer Aktivität ein Kommando zum Setzen des Wertes. Ein anderer Optimierungsansatzpunkt können Gruppen von Kommandos sein, die immer in der gleichen Abfolge auftreten – z.B. können die Attribute eines Objektes schon beim Instanzieren gesetzt oder erst nachträglich (d.h. nach anderen Arbeiten) “geändert” werden – im ersten Fall ist nur ein Kommando nötig für beides.

Das vorgeschlagene Design basiert aus den genannten Überlegungen auf letzterem Szenario und kann durch folgende zwei Grundregeln präzisiert werden:

- Alle Informationen, die benötigt werden, um die ursprüngliche Struktur in der Applikation, die den Zustand in der Datenbank hervorgebracht hat, vollständig wiederherzustellen, müssen in der Datenbank gespeichert werden. Es sollte beachtet werden, dass das heißt, dass nur die relevanten Informationen gespeichert werden müssen, dass die dafür notwendige Datenstruktur hingegen ganz anders aussehen kann als die in der Applikation; das ist ja auch der Fall (Java-Objekte vs. Tabellen).
- Nach einer Transaktion ist der Stand der Datenbank Ausgangspunkt für alle weiteren Modifikationen, d.h. “eine Transaktion löscht die Vergangenheit.” Weder die Transaktion selber, noch die Modifikationen vor der Transaktion können widerrufen werden; ein vorgängiger Zustand kann nur mehr durch “positive” Modifikationen (i. Ggs. zu Widerrufen) erreicht werden.

In den folgenden Abschnitten werden einzelne Aspekte dieses Designs beleuchtet, namentlich die Klassen `Command`, `History` und `dbConnection`.

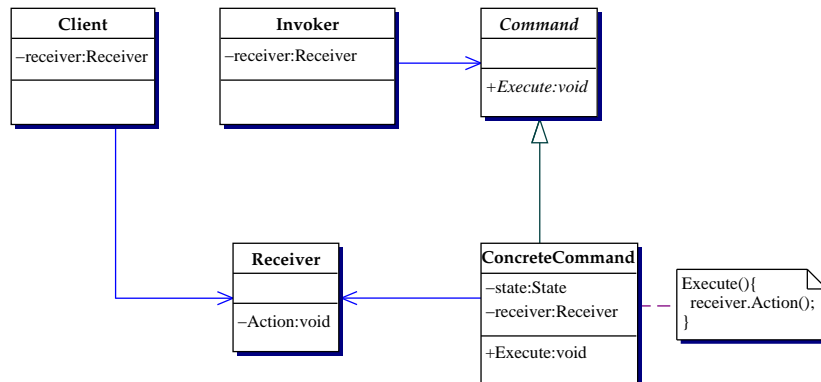


Abbildung 3.1: Das Design Pattern *Command*. Nach: [Gamma, 236].

## 3.2 Die Klasse Command

In diesem Abschnitt wird zuerst der Mechanismus des Design Patterns *Command* erläutert, dann werden das Interface der Klasse *Command* und die *ConcreteCommands* hinsichtlich ihrer Implementierung erörtert.

### 3.2.1 Das Design Pattern *Command*

Die Idee des Design Patterns *Command* ([Gamma, 233ff.]) ist es, zusammenhängende Operationen nicht in einer Methode, sondern in einer Klasse zu abstrahieren, sodass es möglich ist, die Command-Instanzen zu referenzieren und somit zu speichern:

Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations. [Gamma, 233]

Das Pattern ist in Abbildung 3.1 wiedergegeben. *Command* definiert das Interface für die Operationen, die auf ein Kommando ausgeführt werden können sollen, also mindestens eine Operation, das Kommando auszuführen (*execute*). *Invoker* führt eine Operation auf ein Kommando aus; oft ist ein Invoker gebunden an graphische Komponenten (wie z.B Menü-Einträge oder Maus-Clicks an bestimmten Stellen). Ein *ConcreteCommand* implementiert die abstrakten Methoden von *Command* so, dass es eine bestimmte Aufgabe erfüllt (wie z.B. Erstellen eines neuen Dokumentes). Ein *Receiver* gibt den Zielkontext für ein Kommando vor; der *Client* ist der allgemeine Kontext, in dem die oben genannten Teile des Patterns stehen (also oft die Applikation als ganze).

### 3.2.2 Das Interface der Klasse Command

Zur Realisierung des Widerrufs eines Kommandos und zur Transaktion ist eine Erweiterung des Command-Interfaces nötig. Abbildung 3.2 zeigt einen Vorschlag dafür. Die drei abstrakten Methoden `concreteExecute`, `concreteUnexecute` und `concreteTransact` müssen von den konkreten Unterklassen im-

plementiert werden; diese Methoden werden indirekt aufgerufen, entweder über die Methode `execute` auf eine Kommandoinstanz, oder über die Methoden `transact`, `undo` oder `redo` der `History`.

Die Methode `concreteExecute` gibt einen `boolean`-Wert zurück, damit entschieden werden kann, ob ein Kommando in die `History` eingefügt werden soll; dies erlaubt es, einerseits auch Kommandos zu definieren, die selber nicht in die Kette der widerrufbaren Kommandos gehängt werden, sondern diese nur verwalten (siehe unten: 3.2.3); andererseits sollen Kommandos, die versagt haben, nicht in die Modifikationsgeschichte aufgenommen werden.

Die Parameterübergabe – die Referenz auf den Receiver von Abbildung 3.1 – geschieht bei der Instanzierung über die Argumente des Konstruktors. Dies ist auch genau der wunde Punkt dieses Design Patterns: Da die Commands per definitionem abgekapselt von ev. verschiedenen Receivern agieren, müssen alle notwendigen Referenzen explizit übergeben werden; zudem liegt es im Gegensatz zum Pattern in der Praxis oft am Invoker, ein Command zu instanzieren; in diesem Fall obliegt ihm, den Receiver zu übergeben, direkt oder via Client. Ein Ansatzpunkt für das Problem der Parameterübergabe könnte sein, dass die Kommandos aufgeteilt würden, sodass jeweils nur ein Receiver pro Kommando vorhanden wäre – die logische Granularität (à la “ein Click – ein Kommando” oder “ein Menüpunkt – ein Kommando”) schien aber wichtiger als die Implementationsnachteile. Die logische Gliederung könnte allerdings trotz auf einen Receiver beschränkter Kommandos mit *Makrokommandos* hergestellt werden (siehe dazu [Gamma, 235]).

### 3.2.3 Die *ConcreteCommands*

Folgende Kommandos werden benötigt (angegeben sind jeweils der Name der Unterklasse von `Command` und eine Kurzbeschreibung):

`Command_newProject` Erstellen eines neuen Projektes

`Command_addSubTask` Erstellen eines neuen Tasks und der entsprechenden abstrakten Aktivität (ev. inkl. Setzen von Eigenschaften und Ausrichten der abstrakten Aktivität)

`Command_newSimpleActivity`, `Command_newCollector`, `Command_newDistributor` Erstellen einer neuen einfachen Aktivität, eines Kollektors oder eines Distributors (ev. inkl. Setzen der Eigenschaften und Ausrichten)

`Command_changeTaskProperties` Ändern der Eigenschaften eines Tasks<sup>1</sup>

`Command_changeProcessProperties` Ändern der Eigenschaften einer Aktivität

`Command_displace` Verschieben einer Aktivität (ev. inkl. Ausrichten)

`Command_remove` Löschen einer Aktivität; beim Löschen einer abstrakten Aktivität wird der zugehörige Task rekursiv mitgelöscht.<sup>2</sup>

<sup>1</sup> `Command_changeTaskProperties` ist eine innere Klasse von `TaskPropertiesDialog`.

<sup>2</sup> Die Rekursion ist noch nicht implementiert, d.h. es können nur Blätter-Tasks gelöscht werden.



`Command_connect` Verknüpfung zweier Aktivitäten erstellen (bei den Aktivitäten mit nur einem Ein- und Ausgang kann dies bedeuten, dass zuerst eine bestehende Verknüpfung gelöscht werden muss)

`Command_disconnect` Verknüpfung zweier Aktivitäten löschen

Dazu kommt folgendes Kommando, das nicht in die Modifikationsgeschichte geschrieben werden muss:

`Command_open` Laden eines Projektes aus der Datenbank

Wünschbar wäre auch, dass der Inhalt eines Untertasks (unter Verlust der Information über die abstrakte Einheit) wieder in den übergeordneten Task eingliedert und auch dass invers ein (existierender) Teil eines Tasks “tel quel” untergeordnet werden könnte (wie in [Abbildung 2.3](#) angedeutet); im zweiten Fall könnten allerdings nicht beliebige Elemente aus einem Task ausgewählt werden, denn es müsste immer sichergestellt sein, dass der Teil in eine (abstrakte) Aktivität mit einem Ein- und einem Ausgang überführt werden kann.

## 3.3 Die Klasse History

### 3.3.1 Das Interface der Klasse History

Die Klasse `History` dient – wie der Name schon sagt – dazu, Kommandos zu widerrufen, aber auch widerrufen Kommandos zu wiederholen oder den aktuellen Zustand in die Datenbank schreiben zu lassen. Daraus ergibt sich folgendes Interface:

- *Funktionalität*: Anhängen eines Kommandos an die Kette der gespeicherten Kommandos, Widerrufen, Wiederholen und Speichern

```
protected void addToHistory(Command c)
public void redo()
public void undo()
public void transact()
```

- *Administration*: Übergabe und Abfrage der `dbConnection` einer `History`; die `dbConnection` wird zur Transaktion benötigt.

```
public void initConnection(dbConnection dbconnection)
public dbConnection getConnection()
```

Der Mechanismus funktioniert folgendermassen: Abbildung 3.3(a) zeigt eine leere `History`. Der Zeiger auf das erste Kommando der Kette und der Zeiger auf das letzte sind dargestellt durch eine vertikale Linie mit horizontalen Fähnlein in Richtung der dazwischenliegenden Kommandokette und je einem horizontalen Pfeil auf deren Anfangs- und Endkommando. Ein Kreis mit einer Linie in der Diagonale zeigt an, dass die beiden Zeiger auf das leere Kommando zeigen (in Java ein Referenz auf `null`), dass die Kette also leer ist. Ein Beispiel einer nicht-leeren Kette zeigt Abbildung 3.3(b). Kreise stehen dabei für Kommandos, die Minuskeln innerhalb der Kreise sollen Kommandoinstanzen eindeutig identifizieren. Die gepunkteten horizontalen Linien deuten die lineare Verkettung der Kommandos an und zwar vom Anfang der Kette (links) zum Ende (rechts). Die vertikalen gestrichelten Pfeile mit Spitze auf einem Kommando sind beschriftet mit einer Zahl, gefolgt von einem Wort: Die Spitze zeigt auf das “aktuelle” Kommando zum Zeitpunkt, der durch die Zahl angegeben wird, also auf das letzte Kommando, das zum Zustand zu jenem Zeitpunkt beigetragen hat; die Zustände sind in aufsteigender Reihenfolge geordnet; die ausgeführte Operation wird dabei durch das Wort in der Beschriftung spezifiziert.

Das Wort kann *do*, *undo*, *redo* oder *transact* sein:<sup>3</sup> Ein mit *do* beschrifteter Pfeil zeigt also auf das zuletzt ausgeführte Kommando; ein mit *undo* beschrifteter Pfeil zeigt auf das Kommando links vom durch *undo* widerrufenen Kommando; ein mit *redo* beschrifteter Pfeil zeigt auf das wiederholte Kommando, wie in Abbildung 3.3(c) dargestellt. Nicht wie in anderen Applikationen heben sich zwei unmittelbar aufeinander folgende *undo*-Befehle nicht auf.

<sup>3</sup> Die Begriffe *undo*, *redo* und *transact* beziehen sich auf die gleichnamigen Methoden `undo`, `redo` und `transact` der Klasse `History`, *do* bezieht sich (synonymisch) auf die Methode `execute` der Klasse `Command`.

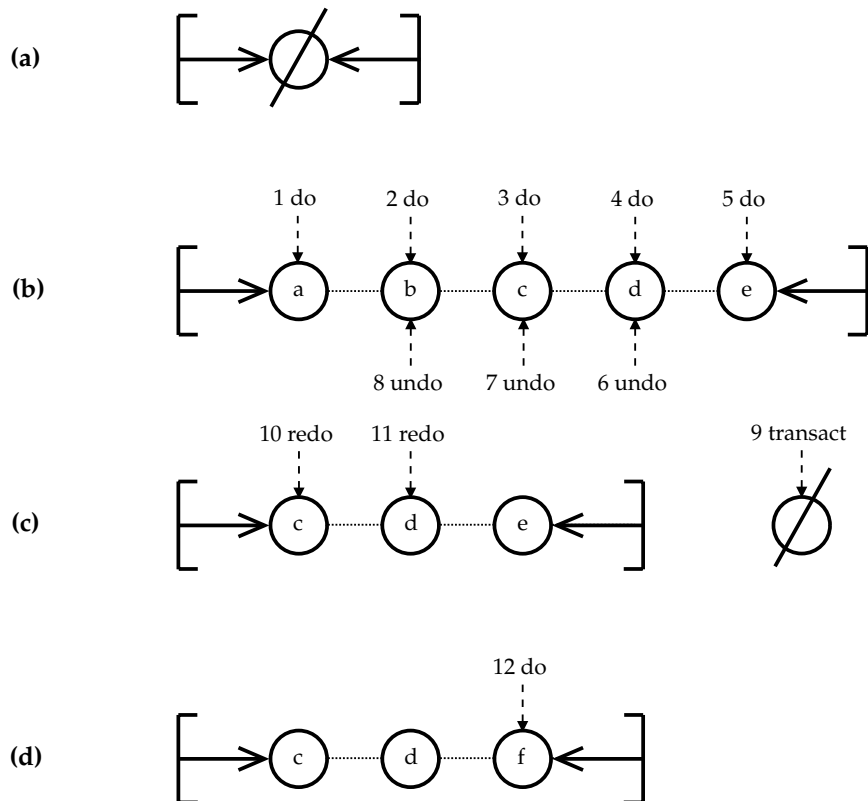


Abbildung 3.3: Entwicklung der History-Kette nach verschiedenen Modifikationen.

In dieser Darstellung werden einfach nur die Elemente im neuesten Zustand angegeben – so enthielt die Kette z.B. zum Zeitpunkt ‘3’ (i.e. nach dem dritten *do*) nur die Kommandos ‘a’, ‘b’ und ‘c.’

Abbildung 3.3(c) zeigt, wie in Abschnitt 3.1 angedeutet, dass ein *transact* die “Vergangenheit” löscht, also alle Kommandos vom Anfang bis zum und mit dem “aktuellen” Kommando vor der Transaktion.<sup>4</sup> Abbildung 3.3(d) zeigt, dass ein *do* die “Zukunft” (die Kommandos rechts vom “aktuellen” Kommando vor der Ausführung) unwiderruflich löscht.

Mit diesen Ausführungen wird auch die Bedeutung der zweiten unter 3.1 aufgeführten Grundregel klar: Zwar könnten für alle Kommandos “*untransact*”-Methoden definiert werden, doch müsste deren Wirkung dann sofort in die Datenbank geschrieben werden, weil Kommandos, die widerrufen, aber nicht wiederholt werden, nicht mehr in der Kette figurieren, sobald neue Kommandos ausgeführt werden, d.h. die “voherige Zukunft” wird – im Gegensatz zu einem *redo* – durch ein *do* gelöscht; somit ginge Information verloren und die Datenbank wäre nicht mehr in konsistentem Zustand.

<sup>4</sup> Deshalb ist das “aktuelle” Kommando nach einer Transaktion leer; die History-Kette selber muss nicht leer sein, wie das Beispiel zeigt.

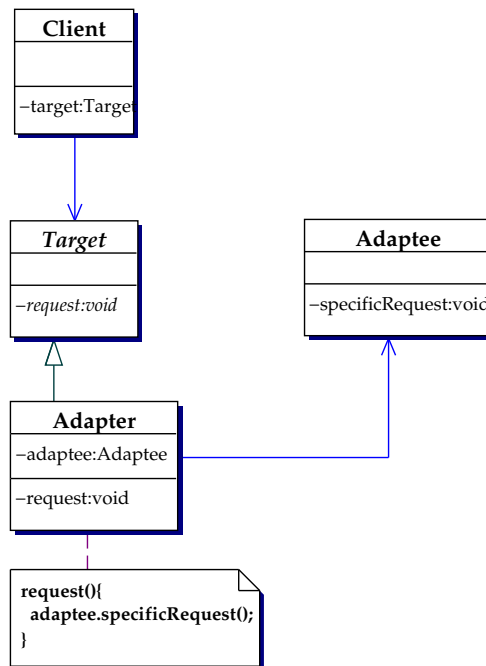


Abbildung 3.4: Das Design Pattern *Adapter*. Nach: [Gamma, 141].

### 3.4 Die Klasse `dbConnection`

Die Klasse `dbConnection` hat zweierlei Grundfunktionen: Erstens dient sie zur Erstellung und Administration einer Netzwerkverbindung zu einer MySQL-Datenbank; zweitens bietet ihr Interface Methoden an, die die Transaktion der durch die verschiedenen `ConcreteCommands` definierten Operationen ermöglichen. Daneben nimmt die Initialisierung der Tabellen der Datenbank eine Mittelstellung ein, da dies einerseits über die rohe Verbindung zur Datenbank hinausgeht, andererseits aber nur die Tabellen für die Datentransaktionen der `ConcreteCommands` vorbereitet.

#### 3.4.1 Das Design Pattern *Adapter*

Sinn und Struktur der Klasse `dbConnection` orientieren sich am *Adapter*-Pattern ([Gamma, 139ff.]). Die Abbildung 3.4 zeigt das Pattern. Der *Client* kommuniziert mit einem *Target* über dessen Interface; *Adaptee* definiert das zu adaptierende Interface; *Adapter* adaptiert das Interface von *Adaptee* nach dem *Target*-Interface. In weniger abstrakten Worten heisst das, dass das *Target* eine dem *Client* geläufige Sprache zeigt (die “Quellsprache”), die der *Adapter* dann hinter den Kulissen in die des *Adaptee* (die “Zielsprache”) übersetzt.

Da in *PAT* nur ein *Adapter* existiert, d.h. nur eine Implementierung eines Übersetzers, muss das *Target*-Interface nicht in einer separaten abstrakten Klasse, sondern kann implizit in der *Adapter*-Klasse `dbConnection` direkt definiert werden. Im gegebenen Fall entspricht dem *Adaptee* eine MySQL-Datenbank,

und die Klasse `dbConnection` übernimmt die Funktionen des Adapters und des Targets. Es wäre in der Zukunft mit beschränktem Aufwand möglich, den bestehenden durch einen alternativen Adapter zu ergänzen oder zu ersetzen, indem das Target-Interface separiert, das Pattern also komplett ausimplementiert würde (e.g. für andere JDBC-Treiber). Auch könnte das Target-Interface beibehalten werden, wenn das Sichern der *APN*-Diagramme in einer MySQL-Datenbank durch ein anderes Zielmedium ersetzt oder ergänzt werden müsste (e.g. eine Übersetzung in *PN*).

### 3.4.2 Das Interface der Klasse `dbConnection`

Das Interface der Klasse `dbConnection` enthält zwei Arten von Methoden:

- *Administration der Verbindung:*

```
public void setUpConnection(String userName,
                           String passWord,
                           String host,
                           String dbName)
                           throws Exception

public String getUsername()
public String getPassword()
public String getHost()
public String getDbName()
public boolean isSetUp()
public ProjectDescriptor[] getProjects()
```

- *Methoden für die Datenbankbindung;* sie entsprechen den unter [3.2.3](#) erwähnten Kommandos, die sie also bei einer Transaktion in die Sprache des Datenbankprogrammes – konkret in MySQL-Befehle – übersetzen; einzige Ausnahme ist, dass es für jeden Aktivitätstyp eine separate Methode gibt.

```
public void newProject(Task task)
public void addSubTask(Task newTask, Task parent)

public void newSimpleActivity(Activity act, Task task)
public void newAbstractActivity(Activity act, Task task)
public void newDistributor(Activity act, Task task)
public void newCollector(Activity act, Task task)

public void updateTask(Task task)
public void updateActivity(Activity act)

public void displace(Activity act)

public void removeActivity(Activity act)
```

```
public void connect(Activity from, Activity to)
public void disconnect(Activity from, Activity to)

public Project getProject(int id)
public void transactProject(Project project)
```

## Kapitel 4

# Konklusion

Bei der Modellierung der Datenstruktur von *APN* war die grosse Knacknuss die Verknüpfungssyntax der Aktivitäten, die schliesslich durch das Separieren der Ein- und Ausgänge von den Aktivitäten und sodann durch ihre Generalisierung gelöst werden konnte.

Eine weitere Schwierigkeit mit dem bestehenden Prototypen *PAT* lag darin, abschätzen zu können, welche Funktionalitäten weggelassen werden können, so dass *APN* vereinfacht würde, aber dennoch mächtig bliebe. Insbesondere lieferte erst die Lösung für die Verbindungssyntax von *APN* die nötige Argumentation, den Abstraktionsmechanismus über eine Extraktionsoperation zu formulieren, um in dieser Sichtweise fremde Funktionalitäten zu eliminieren.

Der Schritt der Vereinfachung von *APN* war einerseits aus konzeptioneller Sicht notwendig, andererseits war zu Übungszwecken schon eine Testversion der Datenbank in Betrieb, die so nur noch leicht abgeändert werden musste.

Die Implementierung der Datenbankanbindung verlief grösstenteils reibungslos, da die Idee mit dem Design Pattern Command schon sehr bald festgestanden hatte und das Datenbankschema – aus *APN* abgeleitet – auch vorgegeben war.

# Literaturverzeichnis

- [Flanagan] David FLANAGAN. *Java in a Nutshell, A Desktop Quick Reference*, 3rd edition. O'Reilly: Sebastopol, 1999.
- [Fowler] Martin FOWLER with Kendall SCOTT. *UML distilled: a brief guide to the standard object modeling language*, 2nd edition. Addison-Wesley: Boston, 2000.
- [Frauchiger] Daniel FRAUCHIGER. *Vorschlag eines Informationssystems zur Beurteilung von operationellen Risiken an Hand eines Prozessmodells als Hilfsmittel*, Diplomarbeit. Universität Freiburg, 2001.
- [Gamma] Erich GAMMA [et al]. *Design Patterns, Elements of Reusable Object-Oriented Software* Addison-Wesley: Boston, 1995.
- [Meier] Andreas MEIER. *Relationale Datenbanken: Leitfaden für die Praxis*, 4. Auflage. Springer: Berlin, 2001.
- [lshort] Tobias OETIKER, Hubert PARTL, Irene HYNA and Elisabeth SCHLEGL. *The Not So Short Introduction to L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>. Or L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> in 95 minutes*, Version 3.20. August 2001. <http://www.ctan.org/tex-archive/info/lshort>.

# Soft- und Hardware

*PAT* wurde weiterentwickelt und getestet für folgende Softwarekonfiguration:

- MM.MySQL 2.0.14
- mysql Ver 11.18 Distrib 3.23.51, for pc-linux-gnu (i686)
- Java<sup>TM</sup>2 Runtime Environment, Standard Edition (build 1.4.0.01-b03)

Dafür wurde folgende Hardware- und Betriebssystemkonfiguration verwendet:

- CPU: Intel Celeron 2/Pentium III Coppermine 1GHz, 256KB cache
- RAM: 512 MB
- OS: SuSE Linux 8.0 mit Kernel 2.4.18-4GB unter KDE 3.0.0

Daneben wurden für diese Seminararbeit die folgenden Programme verwendet:

- Borland<sup>®</sup> JBuilder<sup>TM</sup>Personal 7.0.155.0
- Together ControlCenter 6.0 Build 1914
- Sketch 0.6.12
- KWrite 4.0
- Ktexmaker2 1.7
- OpenOffice 641c-23
- pdflatex: pdfTeX (Web2C 7.3.7) 3.14159-1.00a-pretest-20011114-ojmw, kpath-sea version 3.3.7
- EPSTOPDF 2.7, 2001/03/05
- Acrobat<sup>®</sup> Reader 4.0 x86 linux

# Abbildungsverzeichnis

2.1	Beispiel für die Flusssteuerung in <i>APN</i> . . . . .	6
2.2	Mögliche <i>APN</i> -Konstrukte und ihre Interpretation. . . . .	7
2.3	Unterordnung eines Teiles eines Tasks. . . . .	7
2.4	Abstraktion und Hierarchisierung eines Tasks. . . . .	9
2.5	Vorschlag eines Entitäten-Beziehungsmodells für <i>APN</i> . . . . .	11
2.6	Vereinfachtes Entitäten-Beziehungsmodell für <i>APN</i> . . . . .	13
2.7	Herleitung von <i>APN</i> aus den Petri-Netzen. . . . .	17
3.1	Das Design Pattern <i>Command</i> . . . . .	21
3.2	Das Interface der Klasse <i>Command</i> . . . . .	23
3.3	Entwicklung der <b>History</b> -Kette nach verschiedenen Modifikationen. . . . .	26
3.4	Das Design Pattern <i>Adapter</i> . . . . .	27

# Tabellenverzeichnis

2.1	Graphische Repräsentation der <i>APN</i> -Elemente. . . . .	8
2.2	Entitäten-Beziehungsmodell und UML-Klassendiagramm. . . . .	10
2.3	Graphische Repräsentation der Elemente von Petri-Netzen. . . . .	16
2.4	<i>PN</i> und <i>APN</i> . . . . .	17