DEUTSCHE GEODÄTISCHE KOMMISSION
bei der Bayerischen Akademie der Wissenschaften

Reihe C **Dissertationen** **Heft Nr. 574**

Jens Zimmermann

Konzeption und Umsetzung eines Informationssystems zur geodätischen Deformationsanalyse

München 2004

Verlag der Bayerischen Akademie der Wissenschaften
in Kommission beim Verlag C. H. Beck

ISSN 0065-5325 **ISBN 3 7696 5013 1**

DEUTSCHE GEODÄTISCHE KOMMISSION
bei der Bayerischen Akademie der Wissenschaften

Reihe C | Dissertationen | Heft Nr. 574

Konzeption und Umsetzung eines Informationssystems zur geodätischen Deformationsanalyse

Zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
von der Fakultät für
Bauingenieur-, Geo- und Umweltwissenschaften
der Universität Fridericiana zu Karlsruhe (TH)
genehmigte
Dissertation

von

Dipl.-Inform. Jens Zimmermann

aus Bremerhaven

München 2004

Verlag der Bayerischen Akademie der Wissenschaften
in Kommission beim Verlag C. H. Beck

ISSN 0065-5325 | ISBN 3 7696 50131

Adresse der Deutschen Geodätischen Kommission:

Deutsche Geodätische Kommission

Marstallplatz 8 ● D – 80 539 München

Telefon (089) 23 031 113 ● Telefax (089) 23 031 – 283/ – 100

E-mail hornik@dgfi.badw.de ● http://www.dgfi.badw.de/dgfi/DGK/dgk.html

Tag der mündlichen Prüfung: 16.12.2003

Hauptreferent: Prof. Dr.-Ing. Dr.-Ing.E.h. Günter Schmitt

Korreferent: Prof. Dr.rer.nat. Wolffried Stucky

ISSN 0065-5325 ISBN 3 7696 50131

Danksagung

Ich danke dem Sprecher der kollegialen Institutsleitung und Inhaber des Lehrstuhls für Mathematische und Datenverarbeitende Geodäsie des Geodätischen Instituts der Universität Karlsruhe (TH), Herrn Univ.-Prof. Dr.-Ing. Dr.-Ing. e.H. Günter Schmitt für die Betreuung dieser Arbeit und für die jederzeit gewährte Unterstützung bei der Vollendung dieser Arbeit. Herrn Prof. Dr. rer. nat. Wolffried Stucky vom Institut für Angewandte Informatik und Formale Beschreibungsverfahren der Universität Karlsruhe danke ich für sein Interesse an dieser Arbeit und für die Übernahme des Korreferats.

Ebenso möchte ich mich bei allen Kolleginnen und Kollegen des Geodätischen Instituts der Universität Karlsruhe (TH) bedanken, die zum Gelingen dieser Arbeit beigetragen haben. Hier seien besonders Herr Dipl.-Ing. Ulrich Schmidt und Herr Dr.-Ing. Norbert Rsch hervorgehoben, die durch viele Anregungen zum Gelingen dieser Arbeit beigetragen haben. Meine Dank gilt ebenfalls meiner Familie, die mich während der gesamten Zeit unterstützt hat.

Ganz besonderer Dank gebührt der Deutschen Forschungsgemeinschaft für die Förderung des Forschungsprojektes mit dem Titel *Aufbau eines Informationssystems zur geodätischen Deformationsanalyse unter Einbeziehung heterogener Daten*, aus dem diese Arbeit hervorgegangen ist.

Karlsruhe im Oktober 2003

Kurzfassung

In der vorliegenden Arbeit wird die Konzeption und Umsetzung eines Informationssystems zur geodätischen Deformationsanalyse beschrieben. Es wird der Datenfluss im Rahmen einer Deformationsanalyse dargestellt und auf seine Umsetzung in verschiedenen geodätischen Programmsystemen eingegangen. Hierbei werden verschiedene Probleme im Zusammenhang mit der Verarbeitung des Datenflusses aufgezeigt, die zur Definition von Anforderungen an das Informationssystem verwendet werden. Da die Anforderungen durch den Einsatz der objektorientierten Modellierung und durch den Einsatz von geeigneten Langzeitdatenspeichern, wie z.B. Datenbanksystemen erfüllt werden können, wird darauf ebenfalls eingegangen. Im Zuge der Umsetzung wird die Systemarchitektur und ihre Aufteilung in verschiedene Komponenten beschrieben, wobei im Falle der Langzeitdatenspeicherung auf die Umsetzung des notwendigen Persistenzmechanismus eingegangen wird. Die Anforderungen an potentiell persistente Klassen werden hierbei ebenso beschrieben wie die Funktionalität des Persistenzmechanismus. Jede Stufe des Datenflusses wird in einer eigenen Komponente realisiert, wodurch verschiedene Realisierungen für einzelne Stufen existieren können. Im Zusammenhang mit der Ausgleichung geodätischer Netze wird gezeigt, wie die zu den Beobachtungen gehörenden funktionalen Modelle gekapselt werden können, um diese austauschbar zu machen. Hierdurch werden die Flexibilität und die Erweiterbarkeit der Algorithmen zur Netzausgleichung sichergestellt. Es wird weiterhin gezeigt, wie diese Algorithmen unabhängig von den zu verarbeitenden Beobachtungstypen gestaltet werden können und wie sie mit einer einheitlichen Schnittstelle zur Integration in das System versehen werden. Im Rahmen der Umsetzung der Deformationsanalysen wird gezeigt, wie verschiedene Arten von Analysen unter einer gemeinsamen Schnittstelle zur Datenübergabe vereinigt werden können. Weiterhin wird die Umsetzung von drei statischen Deformationsanalysen in eigenen Komponenten und ihre Integration in das System beschrieben. Zum Abschluss wird die prototypische Realisierung des Systems beschrieben, es werden erste Erfahrungen vom Einsatz geschildert und es wird auf die Umsetzung der anfangs aufgestellten Zielsetzungen sowie auf Möglichkeiten zur Weiterentwicklung eingegangen.

Abstract

This work describes the conception and realisation of an information system for geodetic deformation analysis. The data flow in the context of a geodetic deformation analysis is presented and its implementation in available software is described. Problems that arise when using the listed software to accomplish the data flow are shown and used to define the demands for the information system. Because the demands can be fulfilled using object-oriented modeling techniques and appropriate long term storage systems e.g. database systems, these are mentioned as well. In the course of the system description the different components are presented. In the case of the long term storage the realised persistence mechanism is introduced. The demands for potential persistent classes and the functionality of the persistence mechanism are explained as well. Each level of the data flow is capsuled in a separate component whereby many realisations for each level can exist. In association with the adjustment of geodetic networks a method to capsule the functional models for the various observations is shown. This encapsulation leads to greater exchangeability, flexibility and extensibility. Furthermore it is shown how the algorithms for network equalisation can be constructed to be independent of concrete observations and how the various algorithms can be furnished with a common interface to integrate them into the system. In the context of realising the algorithms for the geodetic deformation analysis a common interface for handing over data to the various types of analyses is presented. The algorithms for three static deformation analyes are realised in separate components. The realisation of a prototype is described as well as first impressions on using the prototype. Finally the initial demands and their realisations are reflected and possibilities for system extensions are pointed out.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Unser tägliches Leben wurde in den letzten Jahren in starkem Ausmaß durch den vermehrten Einsatz von Computertechnik beeinflusst. Mit dieser kommen wir in den unterschiedlichsten Lebensbereichen in Kontakt. Als Beispiele lassen sich hier u.a. Mobiltelefone, Geldautomaten, Steuerungen in Fahrzeugen, internetbasierte Buchungssysteme, Einsatz von Robotern in der Medizin sowie auch Haushaltsgeräte, wie z.B. Waschmaschinen, nennen.

Auch in der Geodäsie hielt die Computertechnik in den letzten Jahren vermehrt Einzug. Messdaten können von den Geräten eigenständig erfasst und digital weiterverarbeitet werden. Sofern die Software zur Weiterverarbeitung nicht von den Herstellern der Messgeräte oder von anderen Herstellern geliefert wird, besteht sie oft aus Eigenentwicklungen, die unter der alleinigen Zielsetzung der Problemlösung implementiert werden. Erweiterungen des Funktionsumfangs hinsichtlich neuer Berechnungsmodelle sind daher, wenn überhaupt, nur mit erheblichem Aufwand durchführbar. Dies kommt besonders zum Tragen, wenn die Ausgangsdaten unter verschiedenen Berechnungsmodellen ausgewertet werden sollen. Da diese Berechnungsmodelle oft in eigenständigen Programmen und manchmal sogar in verschiedenen Programmiersprachen implementiert sind, muss zusätzlich noch die Vergleichbarkeit der Ergebnisse überprüft werden. Der Im- und Export der Daten geschieht hierbei über Dateien, deren Format speziell auf das entsprechende verarbeitende Programm zugeschnitten ist. Um die Daten anderen Programmen zugänglich zu machen, müssen diese vorher in das entsprechende Eingabeformat des jeweiligen Programms transformiert werden.

Diese Vorgehensweise führt im Verlauf des Datenflusses zu einer Menge an Dateien, die jeweils die Ergebnisse des entsprechenden Verarbeitungsschrittes enthalten, deren Format jedoch durch das Programm, das den Verarbeitungsschritt durchgeführt hat, vorgegeben ist. Diese proprietären Dateiformate können sich als problematisch erweisen, wenn sie nicht ausreichend dokumentiert sind oder die Dokumentation nicht mehr verfügbar ist. In solchen Fällen kann es dazu kommen, dass neue Berechnungsmodelle auf solche Daten nicht angewendet werden können, da nicht mehr bekannt ist, an welchen Stellen der Datei die benötigten Informationen zu finden sind.

Bei der geodätischen Deformationsanalyse werden die zu überwachenden Objekte durch Punkte diskretisiert. Diese Objektpunkte werden in bestimmten zeitlichen Abständen, so genannten Epochen, vermessen. Die in jeder Epoche aufgenommenen Daten werden in verschiedenen Schritten verarbeitet, bevor die eigentliche Deformationsanalyse durchgeführt wird. Nach der Aufnahme der Messungen werden diese einer Vorverarbeitung unterzogen, um sie im nachfolgenden Schritt gemeinsam ausgleichen zu können. Bei dieser Ausgleichung werden für die Beobachtungen Verbesserungen und für die Punkte Zuschläge zu den Koordinaten berechnet und an diese angebracht. Bei der sich anschließenden Deformationsanalyse werden die Objektpunkte epochenweise miteinander verglichen, wobei solche Punkte gefunden werden sollen, bei denen es Veränderungen in den Koordinaten gegeben hat, die nicht mehr durch Messfehler erklärt werden können. Die Verfahren zur Deformationsanalyse werden in statische und kinematische Analysen unterteilt. Statische Deformationsanalysen vergleichen jeweils eine Bezugs- und eine Folgeepoche miteinander. Solche Verfahren wurden in [46] vorgestellt. Kinematische Deformationsanalysen vergleichen mehrere Epochen miteinander und versuchen, zusätzlich Bewegungsraten für die sich verändernden Punkte zu ermitteln. Solche Verfahren wurden in [46, 47, 49, 58] vorgestellt.

Mit den in der Geodäsie gebräuchlichen Programmsystemen ist es nicht ohne weiteres möglich, den gerade beschriebenen Datenfluss im Rahmen einer Deformationsanalyse zu verarbeiten. Auch die Erweiterbarkeit der Programmsysteme um neue Analyseverfahren oder Ausgleichungsmodelle ist aus Sicht der Anwender sehr eingeschränkt. Ziel dieser Arbeit ist es daher, ein Konzept für ein Softwaresystem zu entwickeln, mit dessen Hilfe verschiedene Arten von geodätischen Deformationsanalysen durchgeführt werden können. Hierbei soll es möglich

sein, den gesamten Datenfluss innerhalb einer Software durchzuführen, die jedoch nicht den bereits erwähnten Einschränkungen hinsichtlich der Erweiterbarkeit oder Änderbarkeit der verwendeten Algorithmen oder Berechnungsmodelle unterliegt. Das System soll offen für Erweiterungen auf verschiedenen Ebenen sein, es soll also möglich sein, neue Algorithmik bzw. neue Berechnungsmodelle auch zu späteren Zeitpunkten in das System zu integrieren, ohne Seiteneffekte hinsichtlich der bereits bestehenden Algorithmik befürchten zu müssen. Um diese Ziele zu erreichen, wird die objektorientierte Modellierung eingesetzt.

Objektorientierte Modellierung und objektorientierter Entwurf [2, 44, 50] sind eine Art der Problemlösung, in deren Mittelpunkt Modelle stehen, die die Konzepte der realen Welt übernehmen. In der objektorientierten Sichtweise werden Systeme als Kollektion diskreter, miteinander interagierender und kommunizierender Objekte betrachtet, die sowohl eine Datenstruktur als auch ein Verhalten in sich vereinen. Hierdurch wird eine enge Kopplung zwischen den Daten und den darauf angewendeten Operationen erzielt. Die in einem Objekt enthaltene Datenstruktur wird hierbei vor den anderen Objekten verborgen. Zur Interaktion mit dem Objekt und der gekapselten Datenstruktur wird anderen Objekten lediglich eine Schnittstelle zur Verfügung gestellt. Auf diese Weise kann die Implementierung des Objekts geändert werden, ohne das Verhalten des Gesamtsystems zu beeinflussen. Dieser Ansatz fördert auch die Wiederverwendbarkeit von einmal vorgenommenen Modellierungen, indem Objekte auch in anderen Programmen eingesetzt werden können.

Beim Entwurf von objektorientierten Systemen existieren eine Reihe von wiederkehrenden Problemstellungen, für die im Laufe der Zeit verschiedene Lösungsansätze gefunden und implementiert wurden. Ihnen ist gemeinsam, dass sie durch Verwendung wiederkehrender Muster von Klassen und kommunizierenden Objekten definiert werden. Diese Lösungsansätze werden Entwurfsmuster [9, 14] genannt.

Ziel der Entwurfsmuster ist es, diese Muster von Klassen und kommunizierenden Objekten zu klassifizieren. Jedes Entwurfsmuster benennt, erläutert und bewertet einen wichtigen und wiederverwendbaren Entwurf in objektorientierten Systemen. Sie vereinfachen die Wiederverwendung von erfolgreichen Entwürfen und Architekturen. Die Darstellung bewährter Techniken als Entwurfsmuster machen diese Techniken leichter verständlich und helfen, zwischen Entwurfsalternativen, die ein System wiederverwendbar machen, und solchen, die die Wiederverwendbarkeit einschränken, zu unterscheiden. Auch die Wartung und Erweiterbarkeit der Systeme wird durch sie verbessert.

Die Notation der Konzepte geschieht mit Hilfe der Unified Modeling Language (UML) [5, 8, 13, 44, 51]. Die UML ist eine universelle Beschreibungssprache für alle Arten objektorientierter Softwaresysteme. Sie wurde entwickelt, um die große Zahl an Verfahren im objektorientierten Entwurf und die große Zahl an verschiedenen grafischen Notationen in einem einheitlichen Konzept zu erfassen. Bei der Entwicklung der hauptsächlich grafischen Notation wurde auf die Verwendung weniger Konzepte und Symbole, eine einfache Modellierung häufiger Probleme und die Verwendung gleicher Konzepte in allen Bereichen der Notation Wert gelegt. Da die UML unabhängig von der Zielprogrammiersprache einsetzbar ist, bietet sie eine gute Basis für die Modellierung objektorientierter Systeme.

Die vor Ort aufgenommenen oder als Ergebnisse von Berechnungen oder Transformationen entstehenden Daten müssen in geeigneten Speichersystemen dauerhaft abgelegt werden. Hierbei sind vor allem die Sicherheit und die Persistenz der Daten wichtig, da zu ihrer Erfassung oftmals ein nicht unerheblicher personeller und logistischer Aufwand erforderlich ist, weshalb die Daten einen gewissen Wert darstellen. Am Anfang dieses Kapitels wurden bereits die Probleme, die aus der Verwendung proprietärer Dateiformate entstehen können, beschrieben. Bei der Datenspeicherung ist daher darauf zu achten, dass diese auf eine Art und Weise geschieht, die nicht vom verwendeten Betriebssystem abhängt und die eine Weiterverarbeitung der Daten auch mit anderen Programmen, die nicht notwendigerweise in derselben Programmiersprache implementiert sind, geschehen kann. Zur Datenablage bietet sich daher ein Datenbanksystem an.

Datenbanksysteme [31, 34] bestehen aus einer Datenbasis und einem Datenverwaltungssystem, das die Dienstfunktionalität, wie z.B. Speichern von Daten, Ändern von Daten oder Wiederauffinden von Daten, zur Verfügung stellt. Neben den kommerziellen Datenbanksystemen, die z.B. von ORACLE [28, 45] oder IBM [25] hergestellt werden, existieren noch eine Reihe frei verfügbarer Datenbanksysteme, wie z.B. MYSQL [39] oder Postgres [48]. Bei der Ablage der Daten mit Hilfe eines Datenbanksystems fungiert dieses als zentrales Datenhaltungssystem

für verschiedene Benutzer. Die Speicherung der in den einzelnen Objekten enthaltenen Daten geschieht auf der Ebene der einzelnen Attribute, um auch anderen Programmen den Zugriff auf die Daten zu ermöglichen und um die Anfragemöglichkeiten der standardisierten Datenbankanfragesprache SQL [22, 32, 34, 36] nutzen zu können.

Der Export von Daten aus dem System und auch der Import von Daten in das System können notwendig sein, wenn z.B. Daten anderen Forschungseinrichtungen oder Partnern zur Verfügung gestellt werden sollen. Um die eingangs erwähnten Probleme bei der Verwendung proprietärer Dateiformate zu umgehen, wird zum Datenexport und auch zum Datenimport die Extensible Markup Language (XML) eingesetzt.

Bei der Extensible Markup Language (XML) [37] handelt es sich um eine Metasprache, die zur Definition von anderen Sprachen oder Datenstrukturen verwendet werden kann. Hierdurch ist es möglich, Sprachen oder Datenstrukturen zu erschaffen, die auf bestimmte Problemstellungen zugeschnitten sind. Bei der Verwendung der XML ist es möglich, den Daten neben einer Struktur auch eine syntaktische Beschreibung der Bedeutung zu geben, was in vielen proprietären Formaten nicht möglich ist. Durch die Verwendung von Textdateien zur Speicherung der strukturierten XML-Daten wird neben der Unabhängigkeit von bestimmten Programmiersprachen auch die Plattformunabhängigkeit hinsichtlich des Betriebssystems sichergestellt. Die XML kann somit ebenfalls zur Speicherung der Daten verwendet werden, wodurch die Portabilität, wie gerade beschrieben, gewährleistet ist. Die Daten sind, wie auch beim Einsatz eines Datenbanksystems, von unterschiedlichen Programmen verarbeitbar und stehen somit nicht exklusiv dem in dieser Arbeit beschriebenen Programmsystem zur Verfügung. Sie lassen sich aufgrund ihrer textuellen Repräsentation auch von Menschen interpretieren. In Zusammenarbeit mit Datenkompressionsverfahren lassen sich die Daten im XML-Format auch zum Datenaustausch zwischen Programmen oder zur Langzeitarchivierung einsetzen. Mit Hilfe der Extensible Stylesheet Language (XSL) können Transformationen auf den Daten ausgeführt werden, um sie z.B. im Internet zu veröffentlichen oder um Ergebnislisten in verschiedenen Formaten zu erstellen.

Aufgrund der bisher beschriebenen Anforderungen und der strikten Umsetzung der objektorientierten Konzepte wird die Programmiersprache JAVA [11, 18, 27, 30, 52] zur Implementierung eingesetzt. Dies bietet neben der Plattformunabhängigkeit den Vorteil, dass der Zugriff auf Datenbanksysteme [19] sowie die Funktionalität zur Verarbeitung von XML-Daten [37] bereits als Sprachbestandteile integriert sind. Lediglich zur Nutzung von Matrizen und Vektoren zur Berechnung mathematischer Sachverhalte muss auf Software von Drittherstellern [23] zurückgegriffen werden.

1.2 Gliederung

Kapitel 1 beschreibt die Motivation für die vorliegende Arbeit und gibt einen Überblick über die im weiteren Verlauf eingesetzten Konzepte und Techniken. In diesem Zusammenhang wurden die Probleme bei der Durchführung einer geodätischen Deformationsanalyse mit der verfügbaren Software bereits angedeutet.

In Kapitel 2 werden die Ziele der Arbeit dargelegt. Hierzu wird zuerst der Datenfluss im Verlauf einer Deformationsanalyse beschrieben, woran sich ein Überblick über verschiedene Programmsysteme aus dem Bereich der Vermessungstechnik anschließt. In diesem Zusammenhang wird auch darauf eingegangen, inwieweit diese Programmsysteme den vorher beschriebenen Datenfluss realisieren. Den Abschluss des Kapitels bildet ein Abschnitt, in dem die Zielsetzung der Arbeit erläutert und Anmerkungen zur Umsetzung der Ziele gemacht werden. Die Anmerkungen dienen zur Motivation der folgenden Kapitel.

Kapitel 3 gibt einen Einblick in die objektorientierte Modellierung. Hierzu werden die wichtigsten objektorientierten Konzepte vorgestellt, die zum Verständnis der nachfolgenden Kapitel benötigt werden. In einem zweiten Abschnitt wird die grafische Notation der zuvor beschriebenen Konzepte mit Hilfe der Unified Modeling Language (UML) beschrieben. Den Abschluss des Kapitels bildet ein Abschnitt über Entwurfsmuster, in dem zwei für diese Arbeit zentrale Muster erläutert werden.

In Kapitel 4 wird auf einige für diese Arbeit wichtige Teile des Datenmodells eingegangen. Hierbei wird jedoch auf eine detaillierte Beschreibung der Klassen, ihrer Attribute und Methoden verzichtet. Die Klassen werden nur soweit beschrieben, dass ihre Verwendung im Verlauf des Datenflusses einer geodätischen Deformationsanalyse ersichtlich ist.

In Kapitel 5 wird auf die Langzeitspeicherung der in Objekten enthaltenen Daten eingegangen. Hier werden zuerst verschiedene Arten von Objektpersistenz beschrieben. Im Anschluss wird die Abbildung der in Kapitel 3 vorgestellten objektorientierten Konzepte auf eine relationale Tabellenstruktur beschrieben. Hiermit können Objektstrukturen in einer relationalen Datenbank persistiert werden. Der letzte Abschnitt dieses Kapitels beschreibt, wie die objektorientierten Konzepte mit Hilfe der Extensible Markup Language auf eine entsprechende XML-Datei abgebildet werden können, um die in den Objekten enthaltenen Daten exportieren und importieren zu können.

Kapitel 6 beschreibt die Systemarchitektur und die verschiedenen Systemkomponenten sowie ihre Realisierungen. Im ersten Abschnitt wird die Systemarchitektur entwickelt und es werden die Zusammenhänge zwischen den einzelnen Systembestandteilen sowie die jeweiligen Funktionalitäten erläutert. Der folgende Abschnitt befasst sich mit der Umsetzung der im vorigen Kapitel beschriebenen Objektpersistenz unter Verwendung eines relationalen Datenbanksystems. Es werden die Klassen zur Erbringung der notwendigen Funktionalität und die Umsetzung der in Kapitel 5 aufgeführten Konzepte beschrieben. Im Anschluss wir die Realisierung der Funktionalität zum Export und Import von Daten unter Verwendung der Extensible Markup Language (XML) beschrieben. Der nächste Abschnitt erläutert die Umsetzung der Algorithmen zur Netzausgleichung. Es wird beschrieben, wie die verschiedenen Algorithmen mit einer gemeinsamen Schnittstelle versehen werden, um sie unter Verwendung der in Kapitel 3 beschriebenen Entwurfsmuster auf flexible Art und Weise in das System zu integrieren. In diesem Abschnitt wird ebenfalls ein Ansatz beschrieben, um die verschiedenen Algorithmen zur Besetzung der zur Netzausgleichung notwendigen Matrizen und Vektoren zu vereinheitlichen, um das System für die Erweiterung mit neuen Beobachtungstypen vorzubereiten. Hierzu werden die in Kapitel 3 vorgestellten objektorientierten Konzepte verwendet. Der vorletzte Abschnitt des Kapitels befasst sich mit der Integration der Algorithmen zur Deformationsanalyse in das System. Hier wird die Umsetzung von drei Ansätzen zur statischen Deformationsanalyse und die Realisierung in eigenen Komponenten beschrieben. Im letzten Abschnitt wird auf die prototypische Implementierung des Systems eingegangen. Es wird die Realisierung der vorher beschriebenen Funktionalitäten reflektiert, und es werden erste Erfahrungen beim Testen des System beschrieben.

Den Abschluss der Arbeit bildet ein Kapitel, in dem die beschriebenen Konzepte und ihre Umsetzungen reflektiert werden und in dem auf mögliche Erweiterungen des Konzepts eingegangen wird.

Kapitel 2

Zielsetzungen

In diesem Kapitel sollen die Zielsetzungen für ein Informationssystem zur geodätischen Deformationsanalyse beschrieben werden. Hierzu wird zuerst der Datenfluss im Verlauf einer Deformationsanalyse beschrieben. Im Anschluss wird ein Überblick über verschiedene Programmsysteme aus dem Bereich der Geodäsie gegeben, wobei auch darauf eingegangen wird, inwieweit diese Programmsysteme den vorher beschriebenen Datenfluss realisieren. Den Abschluss des Kapitels bildet ein Abschnitt, in dem die Zielsetzungen der Arbeit aufgelistet werden und in dem erste Anmerkungen zur Umsetzung der genannten Ziele gemacht werden.

2.1 Datenfluss bei der geodätischen Deformationsanalyse

Der erste Schritt zur Ermittlung der Anforderungen an ein Informationssystem zur geodätischen Deformationsanalyse besteht darin, den im Rahmen einer Deformationsanalyse entstehenden Datenfluss zu betrachten.

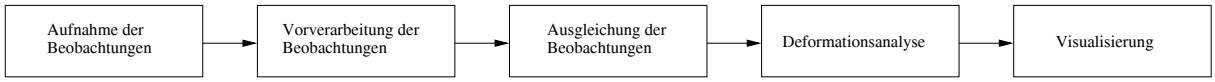

Abbildung 2.1: Datenfluss im Verlauf einer Deformationsanalyse

Abb. 2.1 stellt diesen Datenfluss auf einem relativ hohen Abstraktionsniveau dar. Hierbei werden fünf Stufen innerhalb des Datenflusses deutlich:

- **Stufe 1: Aufnahme der Beobachtungen**
  Die Aufnahme der Beobachtungen bildet den Ausgangspunkt einer geodätischen Deformationsanalyse. In dieser Phase werden die durch Punkte diskretisierten Objekte durch den Einsatz verschiedener Verfahren und unterschiedlicher Geräte vermessen. Die Vermessung aller Objektpunkte wird in diesem Zusammenhang Epoche genannt. Die erfassten Daten werden heute auf Speicherkarten in den Geräten vorgehalten, die durch entsprechende Software der Gerätehersteller ausgelesen werden können.

- **Stufe 2: Vorverarbeitung der Beobachtungen**
  Nachdem die Daten aufgenommen wurden, müssen die Beobachtungen durch verschiedene Vorverarbeitungsschritte so aufbereitet werden, dass sie zu dem in der nächsten Stufe durchzuführenden Ausgleichungsmodell passen. Als Beispiele für solche vorverarbeitenden Berechnungen lassen sich die atmosphärischen und geometrischen Korrekturen bei der Streckenmessung anführen.

- **Stufe 3: Ausgleichung der Beobachtungen**
  In dieser Stufe werden die Beobachtungen einer Ausgleichung nach der Methode der kleinsten Quadrate unterzogen. Hierbei wird mit Hilfe der Beobachtungen ein überbestimmtes Gleichungssystem aufgestellt, dessen Unbekannte den Zuschlägen zu den Näherungswerten der Koordinaten und den Zuschlägen zu den Zusatzparametern entsprechen. Nachdem dieses iterative Rechenverfahren beendet ist und die Zuschläge angebracht sind, können die an die Beobachtungen anzubringenden Verbesserungen berechnet werden.

- **Stufe 4: Deformationsanalyse**
  Bei der Deformationsanalyse werden die Koordinaten der Punkte von ausgeglichenen Netzen epochenweise miteinander verglichen. Unter Einbeziehung der zu den Punkten bzw. ihren Koordinaten gehörenden Genauigkeitsinformationen sollen signifikante Verschiebungen aufgedeckt werden, die nicht durch Messfehler erklärt werden können. Die Deformationsanalyse kann auf verschiedene Arten durchgeführt werden. Bei der statischen Deformationsanalyse werden jeweils eine Bezugs- und eine Folgeepoche miteinander verglichen. Kinematische Deformationsanalysen vergleichen mehrere Epochen miteinander und versuchen, zusätzlich Bewegungsraten für die sich verändernden Punkte zu ermitteln.

- **Stufe 5: Visualisierung**
  Die Ergebnisse der Deformationsanalyse können visualisiert werden, um die Übersichtlichkeit zu erhöhen. Kritische Verschiebungen an einzelnen Punkten können z.B. durch entsprechende Farbgebung oder Symbole angezeigt werden.

Nach dieser allgemeinen Beschreibung des Datenflusses im Rahmen einer geodätischen Deformationsanalyse werden im folgenden Abschnitt verschiedene Programmsysteme aus dem Bereich der Geodäsie vorgestellt.

2.2 Überblick über Software in der Geodäsie

Dieser Abschnitt gibt einen Überblick über verschiedene Programmsysteme, die zur Lösung von Problemstellungen aus der Geodäsie eingesetzt werden. Im Rahmen der Beschreibung wird ein Überblick über die im Zusammenhang mit dem im vorigen Abschnitt vorgestellten Datenfluss bereitgestellte Funktionalität gegeben. Die Liste der in diesem Abschnitt vorgestellten Programmsysteme ist nicht erschöpfend.

2.2.1 GeoMos

GeoMos [33] dient zur automatisierten Datenerfassung mit Hilfe verschiedener Sensoren, die an die Software angeschlossen werden können. Diese Sensoren umfassen Tachymeter und GPS ebenso wie meteorologische oder geotechnische Sensoren. Die erfassten Werte können in einer Datenbank gespeichert werden, um sie zu späteren Zeitpunkten auswerten zu können. Der Export der Daten in verschiedene Standardformate ist ebenso ein Bestandteil des Systems wie ein Meldungsmanagement, über das die Anwender von vorher definierten Ereignissen in Kenntnis gesetzt werden können. Neben der Datenerfassung bietet das System auch noch Funktionen zur Datenanalyse. Diese Funktionen realisieren z.B. die Darstellung der Daten in Weg-Zeit-Diagrammen, mit denen Bewegungstendenzen auf der Basis von Koordinatenvergleichen aufgezeigt werden können.

Bezugnehmend auf den in Abschnitt 2.1 beschriebenen Datenfluss realisiert die Software also die Stufen der Datenerfassung und der Visualisierung. Eine Ausgleichung der Beobachtungen oder eine Deformationsanalyse unter Einbeziehung der Genauigkeitsinformationen ist in das System nicht integriert. Da die Daten nach ihrer Erfassung in einer Datenbank abgelegt werden und somit eine standardisierten Schnittstelle zum Zugriff bereitsteht, können sowohl die Ausgleichung wie auch die Deformationsanalyse durch andere Software durchgeführt werden.

2.2.2 KAFKA

Das geodätische Programmsystem *KAFKA* [3] dient zur Analyse und Ausgleichung beliebiger terrestrisch-geodätischer Beobachtungen. Das System erlaubt hierbei die simultane Verarbeitung verschiedener Beobachtungen. Hierzu gehören unter anderem: Richtungen, elektronisch gemessene Strecken und Messbandstrecken, Zenitdistanzen, Höhenunterschiede, GPS-Koordinaten und Anschlusskoordinaten. Eine Koordinatentransformation von Massenpunkten ist ebenso möglich, wie die Netzanalyse sowie die Fehlersuche in den Daten. Das Programmsystem besteht aus verschiedenen Modulen. Die Dateneingabe ist in einem eigenen Modul realisiert und kann mittels Bildschirmmasken oder über die von den Geräteherstellern zur Verfügung gestellten Dateien erfolgen. Die Module zur Netzausgleichung erlauben jeweils die Ausgleichung der vorgenommenen Lage- und Höhenmessungen. Der Datenexport ist ebenfalls in einem eigenen Modul realisiert und erlaubt z.B. die Speicherung der Daten in einem Format, das mit *Autocad* [1] eingelesen und visualisiert werden kann.

Hinsichtlich des in Abschnitt 2.1 beschriebenen Datenflusses realisiert das Programmsystem die Stufen der Vorverarbeitung und der Ausgleichung. Eine Deformationsanalyse unter Einbeziehung der Genauigkeitsinformationen ist nicht in das System integriert. Auch die Visualisierung von Daten oder Ergebnissen ist nicht Bestandteil des Programmsystems, kann jedoch durch die Möglichkeit des Datenexports realisiert werden.

2.2.3 Neptan

Das Programm *Neptan* [54] dient zur ein-, zwei- und dreidimensionalen Auswertung und Analyse gängiger geodätischer Messungen. Bei den Messungen handelt es sich um Richtungen, GPS-Messungen, Azimute, Zenitdistanzen, Höhenunterschiede, bewegliche Festpunkte und Koordinatendifferenzen. Im Verarbeitungskonzept ist sowohl eine Voranalyse der GPS-Messungen als auch der terrestrischen Messungen enthalten. Die Netzausgleichung kann frei oder unter Einbeziehung von beweglichen Festpunkten geschehen. Optional kann noch eine Komponente zur Deformationsanalyse nachgeschaltet werden, die Punktgruppen in zwei Epochen auf Verschiebungen untersucht und gegebenenfalls die Deformationsparameter berechnet.

Neptan bietet in der Grundversion die Möglichkeit der Vorverarbeitung verschiedener geodätischer Beobachtungen sowie deren gemeinsame Ausgleichung. Es realisiert also die zweite und dritte Stufe des in Abschnitt 2.1 vorgestellten Datenflusses. Im Anschluss an die Ausgleichung der Beobachtungen kann durch ein optionales Programm eine Deformationsanalyse zwischen zwei Epochen durchgeführt werden, was die vierte Stufe des Datenflusses realisiert.

2.2.4 KIVID

KIVID [7] dient zur Auswertung von Vermessungen nach amtlichen Richtlinien. Zur Dateneingabe können die Beobachtungen aus den Dateien der Gerätehersteller übernommen werden. Hierbei ist es auch möglich, gerätespezifische Korrekturen an den Daten vorzunehmen. Die erfassten Daten können dann mit einer Vielzahl geometrischer Verfahren weiterverarbeitet werden. Die Datenhaltung wird durch eine Eigenentwicklung vorgenommen. Die Anbindung von Datenbeständen, die in Datenbanken vorliegen, wird vom Hersteller als Dienstleistung erbracht. Die Ausgleichung der Beobachtungen ist durch externe Programme, wie z.B. das in Abschnitt 2.2.2 beschriebene *KAFKA* möglich, wobei der Datenaustausch über Dateien vonstatten geht. Die Darstellung der Netztopologien bzw. der Ergebnisse geschieht ebenfalls durch die Produkte von Drittherstellern, die an die Software angebunden werden.

Die Software ist zwar in der Lage, Daten zu importieren und diese auch einer gewissen Vorverarbeitung zu unterziehen, ihr Leistungsspektrum erstreckt sich jedoch eher auf geometrische Berechnungsverfahren. Die Ausgleichung von Beobachtungen ist nur durch die Anbindung externer Software möglich, und Deformationsanalysen sind nicht vorgesehen. Die Visualisierung von Daten und Ergebnissen geschieht ebenfalls durch Anbindung einer externen Software.

2.2.5 Geo-Samos

Geo-Samos [6] ist ein grafikgestütztes vermessungstechnisches Berechnungsprogramm zur Unterstützung vermessungstechnischer Abläufe. Die Software ist modular aufgebaut, wobei es zum Grundmodul weitere Ergänzungsmodule gibt. Diese decken die Bereiche der Berechnungen der Kataster- und Ingenieurvermessung, der Tachymeter- und GPS-Messungen sowie des Datenimports und des Datenexports ab. Der Datenimport erlaubt die Übernahme von Daten der verschiedenen Gerätehersteller. Im Bereich der Berechnungen finden sich vorwiegend geometrische Berechnungsverfahren, aber auch die Möglichkeit, digitale Geländemodelle anzulegen und zu verwalten. Die Ausgleichung der Beobachtungen ist durch externe Software möglich. Hierzu kann z.B. das in Abschnitt 2.2.3 beschriebene Programm *Neptan* verwendet werden, wobei der Datenaustausch über Dateien stattfindet. Die Visualisierung der Daten erfolgt durch die Software selbst. Zusätzlich können die Daten in verschiedenen Formaten exportiert werden, um sie auch mit verschiedenen CAD-Systemen bearbeiten zu können.

Im Hinblick auf den in Abschnitt 2.1 vorgestellten Datenfluss realisiert *Geo-Samos* die Vorverarbeitung der Daten sowie die Visualisierung der Ergebnisse. Die Ausgleichung der Beobachtungen ist eine Funktionalität, die zwar angeboten, aber durch Produkte von Drittherstellern realisiert wird. Die Durchführung von Deformationsanalysen ist im Programmsystem nicht vorgesehen.

2.2.6 Netz2D

Netz2D [17] dient zur Ausgleichung zweidimensionaler Netze. Hierbei können Richtungen, Strecken, Azimute, Streckenverhältnisse, relative und absolute GPS-Beobachtungen sowie bewegliche Festpunkte gemeinsam ausgeglichen werden, wobei die Dateneingabe über ein proprietäres Dateiformat erfolgt. Die Vorverarbeitung der Beobachtungen geschieht durch verschiedene Zusatzprogramme, die ihre Ergebnisse in unterschiedlichen Dateiformaten ablegen. Aus diesen Dateien wird dann die Eingabedatei für *Netz2D* erzeugt. Die Ergebnisse der Netzausgleichung werden wiederum in einer Datei mit einem proprietären Format abgelegt. Die Bedienung der Software geschieht durch eine grafische Oberfläche, in die auch die Bedienung der Programme zur Vorverarbeitung integriert ist. Zur Visualisierung der Daten und Ergebnisse stehen Programme zur Verfügung, die aus den Ausgabedateien von *Netz2D* Dateien zum Import in *Autocad* [1] erzeugen. Zur Durchführung von Deformationsanalysen existiert das Programm *CODEKA2D* [16]. Dieses führt eine Deformationsanalyse unter Einbeziehung der Genauigkeitsinformationen durch. Für den Datenimport steht ebenfalls ein proprietäres Datenformat zur Verfügung.

Hinsichtlich des in Abschnitt 2.1 beschriebenen Datenflusses realisiert *Netz2D* die Vorverarbeitung der Beobachtungen sowie deren Ausgleichung. Der Datenaustausch zwischen beiden Stufen findet über proprietäre Dateiformate statt, wobei das eine Format in das andere Format überführt werden muss. Eine Deformationsanalyse kann durch Verwendung eines weiteren Programms im Anschluss durchgeführt werden. Hierbei ist jedoch wiederum die Umformung von proprietären Datenformaten nötig. Die Visualisierung der Daten und Ergebnisse ist nicht Bestandteil des Programms. Sie wird über Datenexport und den Einsatz von Software von Drittherstellern ermöglicht.

2.2.7 PANDA

Das Programmsystem *PANDA* [41] deckt die Bereiche der Vorverarbeitung, der Netzausgleichung und der Deformationsanalyse ab, wobei die einzelnen Verarbeitungsschritte in eigenen Programmmodulen realisiert sind. Das System ist mit einer textuellen Benutzeroberfläche ausgestattet, die die Anwahl der einzelnen Module gestattet. Das Modul zur Netzausgleichung ist in der Lage, Richtungssätze, Höhenunterschiede, Azimute und Horizontalstrecken zu verarbeiten. Zusätzlich können auch noch Anschlusspunkte und Koordinatendifferenzen in die Ausgleichung eingeführt werden. Im Rahmen der Deformationsanalyse können jeweils zwei Epochen miteinander in Beziehung gesetzt werden, die unter Einbeziehung der Genauigkeitsinformationen verarbeitet werden. Die grafische Darstellung der Ergebnisse erfolgt durch das Erzeugen von Dateien, die von Plottern oder Druckern ausgegeben werden können, oder durch das Erzeugen von Dateien, die von CAD-Systemen eingelesen und dargestellt werden können. Die Datenübergabe zwischen den einzelnen Programmmodulen wird durch Dateien vollzogen, die von einem Modul erzeugt und von einem anderen Modul eingelesen werden.

PANDA realisiert die Stufen der Vorverarbeitung, der Netzausgleichung und der Deformationsanalyse des in Abschnitt 2.1 beschriebenen Datenflusses. Die verschiedenen Stufen sind in eigenen Modulen realisiert, wobei der Datenaustausch über proprietäre Datenformate geschieht. Die Visualisierung der Daten und Ergebnisse wird nicht vom Programm übernommen, sondern kann durch den Einsatz von CAD-Systemen oder Druckern bzw. Plottern erfolgen.

2.2.8 Bemerkungen

Hinsichtlich des Datenflusses, wie er in Abschnitt 2.1 beschrieben wird, sind nur drei der in Abschnitt 2.2 beschriebenen Programmsysteme in der Lage, diesen in gewissen Grenzen durchzuführen. Hierbei handelt es sich um die Produkte *Netz2D*, *Neptan* und *PANDA*.

Alle drei sind in der Lage, den Datenfluss unter Zuhilfenahme von zusätzlichen Softwarekomponenten zu realisieren. Die mathematischen Modelle, die zur Vorverarbeitung, zur Netzausgleichung und auch zur Deformationsanalyse verwendet werden, sind fest in der jeweiligen Software verankert und können vom Anwender nicht modifiziert oder ausgetauscht werden. Gerade dies ist jedoch im Umfeld von Forschungseinrichtungen wichtig. Hier werden im Rahmen wissenschaftlicher Arbeiten z.B. neue Berechnungsverfahren zur Vorverarbeitung von Beobachtungen, neue funktionale Modelle zur Ausgleichung der Beobachtungen oder neue Algorithmen zur Durchführung von Deformationsanalysen entwickelt. Diese neuen Erkenntnisse können oftmals nicht in die

bestehenden Produkte integriert werden, was zu Eigenentwicklungen führt, in deren Verlauf verschiedenste Datenmodelle sowie Datenformate für ähnliche oder gleiche Problemstellungen geschaffen werden.

Bei der in Abschnitt 2.2 vorgestellten Software wird deutlich, dass die Datenhaltung und der Datenaustausch zwischen den Programmen und den Modulen von Drittherstellern auf der Basis von proprietären Dateiformaten geschieht. Um die gespeicherten Daten auch mit anderen Programmsystemen nutzen zu können, müssen die Dateien zuerst in die entsprechenden Eingabeformate transformiert werden, wobei im schlimmsten Fall ein manuelles Editieren erforderlich ist.

Im Verlauf des Datenflusses operieren die verschiedenen Komponenten auf dem Datenbestand und verändern hierbei dessen Zustand. Die Langzeitdatenspeicherung geschieht bei vielen der bisher vorgestellten Programmsysteme durch die Verwendung proprietärer Dateiformate, was dazu führt, dass der Zugriff auf die Daten durch andere Programme erschwert wird. Dies ist gerade dann hinderlich, wenn Datenbestände durch die Verwendung verschiedener Programme unter mehreren Gesichtspunkten analysiert werden sollen.

Aus den bisherigen Ausführungen wurde deutlich, welche Einschränkungen die in der Geodäsie verwendeten Programmsysteme hinsichtlich des Datenflusses der Deformationsanalyse, der Erweiterbarkeit und der Langzeitdatenspeicherung unterliegen.

2.3 Zielsetzungen der Arbeit

In diesem Abschnitt werden die Zielsetzungen der Arbeit dargelegt und es werden Anmerkungen zur Umsetzung der genannten Ziele gemacht, um die nachfolgenden Kapitel zu motivieren.

2.3.1 Die Ziele der Arbeit

1. Integration des Datenflusses einer geodätischen Deformationsanalyse in einem Programmsystem.
2. Das System soll auf verschiedenen Ebenen offen für Erweiterungen sein.
3. Das System soll ein Datenmodell verwenden, das von allen am Datenfluss beteiligten Komponenten genutzt werden kann.
4. Die Persistenz der Daten soll durch die Verwendung geeigneter Langzeitdatenspeicher mit standardisierter Zugriffsschnittstelle sichergestellt werden.
5. Der Datenexport soll durch ein strukturiertes Datenformat erfolgen, in dem auch die Bedeutung der Daten hinterlegt werden kann.

Die angegebenen Ziele sollen nun näher erläutert werden, um die Neuerungen gegenüber den existierenden Programmsystemen zu verdeutlichen.

Wie bereits bei der Vorstellung der verschiedenen geodätischen Programmsysteme deutlich wurde, ist keines der Systeme in der Lage, den in Abschnitt 2.1 vorgestellten Datenfluss einer geodätischen Deformationsanalyse vollständig durchzuführen. Wenn eine Deformationsanalyse erfolgen kann, so müssen häufig Daten durch Verwendung proprietärer Datenformate exportiert und von den Analysemodulen wieder importiert werden, da diese oftmals von Drittanbietern bereitgestellt werden oder als Insellösungen implementiert wurden. Ein ähnliches Vorgehen ist oft auch zur Visualisierung der Daten notwendig. Durch die Integration des Datenflusses in einem Programmsystem können die originären Daten einer kompletten Verarbeitung unterzogen werden, ohne Dateiformate transformieren oder editieren zu müssen. Hierdurch wird die Fehleranfälligkeit gesenkt und die Geschwindigkeit der Verarbeitung kann gesteigert werden.

Bei den vorgestellten Programmsystemen sind die zu den verschiedenen Stufen des Datenflusses gehörenden mathematischen Modelle fest in der Software verankert und können vom Anwender nicht modifiziert werden. Das System soll hier offen für Erweiterungen auf verschiedenen Ebenen sein. Im Rahmen der Algorithmen zur Ausgleichung geodätischer Netze soll es möglich sein, zu den vorhandenen Beobachtungen neue funktionale Modelle

hinzuzufügen. Hierdurch wird dem Anwender die Möglichkeit eröffnet, eine Auswertung unter Verwendung verschiedener funktionaler Modelle zu wiederholen bzw. für jede auszuwertende Beobachtung das zu verwendende funktionale Modell vor Beginn der Ausgleichung festzulegen. Das System soll ebenfalls offen für die Einführung neuer Beobachtungstypen sein. Dies ist gerade im Hinblick auf die Zusammenarbeit mit anderen Geowissenschaften wichtig, die z.T. Beobachtungen durchführen, die von den bisherigen geodätischen Programmsystemen nicht verarbeitet werden können. Durch die gemeinsame Auswertung von Beobachtungen aus verschiedenen Geowissenschaften lassen sich möglicherweise Erkenntnisse gewinnen, die bei alleiniger Auswertung geodätischer Beobachtungen nicht sichtbar wären. Die oben aufgestellte Anforderung nach der Offenheit gegenüber verschiedenen funktionalen Modellen gilt natürlich auch für neu eingeführte Beobachtungstypen. Auch die Integration neuer Verfahren zur Vorverarbeitung der aufgenommenen Daten, zur Ausgleichung geodätischer Netze bzw. neuer Ansätze zur Deformationsanalyse soll möglich sein, um aktuelle Forschungsergebnisse anwenden oder testen zu können. In diesem Zusammenhang sei z.B. auf [10] verwiesen, wo ein neuer Ansatz zur Ausgleichung geodätischer Netze vorgestellt wird.

Im Verlauf einer geodätischen Deformationsanalyse werden die originären Daten durch die verschiedenen Berechnungsverfahren modifiziert, wobei die Ergebnisse eines Berechnungsverfahrens die Eingangsdaten des darauffolgenden Berechnungsverfahrens sind. Bei den vorgestellten Programmsystemen wird diese Datenübergabe häufig durch den Einsatz von Dateien mit proprietären Formaten durchgeführt. Hierbei kann es vorkommen, dass diese Dateien vor der Weiterverwendung in ein anderes Format transformiert werden müssen. Nach der Transformation müssen z.T. zusätzliche Informationen von Hand eingefügt werden, um die Daten mit der gewünschten Software weiterverarbeiten zu können. Solche Transformationen sind, vor allem wenn zusätzliche Modifikationen von Hand durchgeführt werden müssen, fehleranfällig und sie fügen zusätzliche Verarbeitungsschritte in den Datenfluss ein. Das System soll auf einem Datenmodell operieren, das allen am Datenfluss beteiligten Komponenten eine Schnittstelle zum Zugriff auf die Daten anbietet, sodass eine Datenübergabe durch Dateien nicht nötig ist. Diese Forderung scheint aus der Sicht der Informatik selbstverständlich zu sein, ist jedoch in den wenigsten geodätischen Programmsystemen verwirklicht, weshalb es zur bereits beschriebenen Transformation von Dateien kommt. Das Datenmodell soll die notwendigen geodätischen Entitäten modellieren und Methoden zum Zugriff und zur Modifikation der enthaltenen Daten bereitstellen.

In Abschnitt 2.1 wurde bereits erwähnt, dass die Vermessung von Objekten in sogenannten Epochen stattfindet. Aufgrund der Art der Objekte kann es vorkommen, dass zwischen zwei Epochen längere Zeitspannen, wie z.B. Monate oder Jahre vergehen. Bei der Deformationsanalyse werden mehrere Epochen zusammen analysiert, um signifikante Verschiebungen an den Objektpunkten aufzudecken, wozu die zu früheren Epochen gehörenden Beobachtungen und die Koordinaten der zu diesen Epochen gehörenden Objektpunkte verfügbar sein müssen. Die Daten dieser frühen Epochen müssen also in geeigneten Langzeitdatenspeichern hinterlegt werden, damit diese später zur Analyse verwendet werden können. Da die Erfassung von Daten oftmals mit einer gewissen Logistik und dem Einsatz von qualifiziertem Personal verbunden ist, stellen die Daten einen nicht unerheblichen Wert dar. Die Auswahl des Langzeitdatenspeichers soll daher vor dem Hintergrund der Zuverlässigkeit und des Vorhandenseins einer standardisierten Schnittstelle zur Abfrage und Manipulation der Daten geschehen. Die Verwendung proprietärer Datenformate scheidet angesichts dieser Kriterien aus, da hier oftmals keine standardisierte Schnittstelle zu den Daten gegeben ist.

In bestimmten Situationen kann es nötig sein, Daten zu exportieren oder Daten zu importieren, um sie z.B. anderen Forschungseinrichtungen oder -partnern zur Verfügung zu stellen. Die von den Programmiersprachen bereitgestellten Exportmöglichkeiten unterstützen oft den Export der Daten in eine Datei. Hierbei wird deren Bedeutung jedoch nicht zwangsläufig ebenfalls in der Datei abgelegt. Die Bedeutung der Daten ist oftmals nur implizit, z.B. durch die Formatierung der Daten innerhalb der Datei vorgegeben, was dazu führen kann, dass die Daten nicht mehr genutzt werden können, wenn das Wissen um deren Bedeutung nicht mehr verfügbar ist. Aufgrund des bereits erwähnten Wertes der Daten ist es notwendig, dass deren Bedeutung ebenfalls exportiert wird. Im Falle der Langzeitdatenspeicherung muss natürlich ebenfalls dafür gesorgt sein, dass die Bedeutung der Daten verfügbar bleibt.

2.3.2 Anmerkungen zur Umsetzung der Ziele

Aus den im vorigen Abschnitt genannten Zielen und Erläuterungen wir deutlich, dass es sich bei der geodätischen Deformationsanalyse um das Zusammenspiel von verschiedenen Berechnungsverfahren auf einem Datenbestand handelt. Zur Umsetzung in ein Programmsystem bietet sich die objektorientierte Modellierung an, da sie die Übernahme von Konzepten der realen Welt erlaubt. Komplexe Systeme werden hierbei als Kollektion miteinander interagierender Objekte verstanden und können anschaulich beschrieben werden. Die geodätische Deformationsanalyse ist das zu modellierende Verarbeitungskonzept, das durch die Interaktion der einzelnen Stufen des Datenflusses charakterisiert wird. Durch die Objektorientierung ist es möglich, ein Datenmodell zu erschaffen, das die verschiedenen Entitäten der Ingenieurgeodäsie modelliert und das allen Komponenten des Datenflusses eine passende Schnittstelle zur Verfügung stellt. Die Integration des Datenflusses in ein Programmsystem kann durch dieses Datenmodell und unter Verwendung einer entsprechenden Systemarchitektur geschehen, die ebenfalls unter objektorientierten Gesichtspunkten umgesetzt wird. Die Konzepte der Objektorientierung werden in Kapitel 3 vorgestellt, wobei ebenfalls eine grafische Notation der Konzepte vorgestellt wird. Das verwendete Datenmodell wird in [53] beschrieben. Zum Verständnis der Systemarchitektur werden einige Teile des Datenmodells in Kapitel 4 vorgestellt.

Auf die Notwendigkeit der Langzeitspeicherung der Daten aufgrund der Zeitspanne zwischen den einzelnen Epochen wurde bereits hingewiesen. Aufgrund der Anforderungen, die an den zu verwendenden Langzeitdatenspeicher gestellt wurden, wird hierzu ein relationales Datenbanksystem eingesetzt. Moderne relationale Datenbanksysteme bieten die geforderte Zuverlässigkeit, um den Schutz der Daten sicherzustellen. Durch die Verwendung der standardisierten Anfragesprache SQL können Daten auf Basis einer einheitlichen Schnittstelle angefragt und manipuliert werden. Da die Repräsentation der Daten im Hauptspeicher des Rechners und in der Datenbank unterschiedlich ist, ist also zur Speicherung der Objekte in der Datenbank eine Transformation der Repräsentationsform notwendig. Mit dieser Problemstellung befasst sich Kapitel 5, in dem verschiedene Abbildungsvorschriften für die in Kapitel 3 vorgestellten objektorientierten Konzepte auf die Tabellenstruktur einer relationalen Datenbank angegeben werden.

Zum Datenexport bzw. -import wird die Extensible Markup Language eingesetzt. Sie bietet die Möglichkeit, Daten strukturiert und mit einer Bedeutung versehen in einer Textdatei zu speichern. Aufgrund ihrer mittlerweile großen Akzeptanz sind verschiedene Programme verfügbar, mit denen die strukturierten Daten angezeigt und manipuliert werden können. Für die verschiedenen Programmiersprachen sind Parser verfügbar, mit denen die Daten und auch die jeweilige Bedeutung eingelesen werden können. Aufgrund der Verwendung von Textdateien zur Ablage der Daten können diese auch auf verschiedenen Betriebssystemplattformen genutzt werden. Auf die Verwendung der Extensible Markup Language wird im Zusammenhang mit der Objektpersistenz in Kapitel 5 eingegangen. Auch hier werden Abbildungsvorschriften angegeben, mit denen die Objekte aus ihrer Repräsentationsform im Speicher in die Struktur der entsprechenden XML-Datei transformiert werden können.

In Kapitel 6 werden die Systemarchitektur und die verschiedenen Systemkomponenten sowie ihre Realisierungen beschrieben, wobei auf die in den vorhergehenden Kapitel Bezug genommen wird.

Kapitel 3

Objektorientierte Modellierung

Objektorientierte Modellierung und objektorientierter Entwurf sind eine Art der Problemlösung, in deren Mittelpunkt Objekte und ihre Beziehungen zueinander stehen. Das grundlegende Konstrukt dieser Denkweise ist das Objekt, das sowohl eine Datenstruktur, als auch ein spezifiziertes Verhalten in sich vereint und mit dem Gegenstände der realen Welt modelliert werden. Mit Hilfe von objektorientierten Modellen lassen sich komplexe Systeme anschaulich beschreiben.

Ein Modell ist eine Abstraktion, die dazu dient, ein System zu verstehen. Weil ein Modell von unwichtigen Details abstrahiert, ist es geeignet verschiedene Szenarien zu durchlaufen, bevor es implementiert wird.

Ein Objektmodell beschreibt die statische Struktur von Objekten in einem System. Hierunter sind die Identität, die Attribute, die Operationen und die Relationen zu anderen Objekten zu verstehen. Das Ziel beim Aufbau eines Objektmodells besteht darin, die Konzepte der realen Welt abzubilden, die für das System wichtig sind. Das Objektmodell sollte frei von Konstrukten sein, die Implementierungsdetails widerspiegeln.

In der objektorientierten Sichtweise werden Systeme als Kollektion diskreter Objekte betrachtet. Hierdurch wird eine enge Kopplung zwischen den Daten und den darauf angewendeten Operationen erzielt. Die folgenden Beschreibungen geben nur die wichtigsten objektorientierten Konzepte wieder und erheben keinen Anspruch auf Vollständigkeit. Eine ausführliche Auflistung der Konzepte findet sich in [2, 30, 50].

3.1 Objektorientierte Konzepte

3.1.1 Klassen und Objekte

Klassen beschreiben Gruppen von Objekten mit ähnlichen Eigenschaften, gemeinsamem Verhalten, gemeinsamen Relationen zu anderen Objekten und einer gemeinsamen Semantik. Die Individualiät der zu einer Klasse gehörenden Objekte ergibt sich aus der Identität, den Attributwerten und den Relationen zu anderen Objekten. Objekte kennen die Klasse zu der sie gehören, womit diese zu einer impliziten Eigenschaft eines Objekts wird. Die Klassenbildung abstrahiert von den eigentlichen Objekten und bietet somit eine Art der Objektspezifikation.

Ein Objekt ist als Gegenstand, Konzept oder Abstraktion zu sehen. Objekte bilden die reale Welt auf natürliche Art und Weise ab und sind ein guter Ausgangspunkt für die Implementierung mit Rechnern. Alle Objekte besitzen eine Identität, die sie eindeutig von anderen Objekten abgrenzt.

Attribute sind Datenwerte, die die Objekte besitzen. Die Attribute werden in der Klassendefinition für alle Objekte spezifiziert, haben in den Objekten jedoch individuelle Werte, die den Zustand des Objekts zum Zeitpunkt der Betrachtung ausmachen. Diese individuellen Werte können z.B. Zahlenwerte bei nummerischen Attributen oder Objektreferenzen sein.

Eine Operation ist eine Funktion oder Transformation, die auf Objekte angewendet werden kann. Jede Operation hat ein Zielobjekt als implizites Argument. Eine Operation kann polymorph sein, also in unterschiedlichen Klassen unterschiedliche Formen annehmen. Die Implementierung einer Operation in einer Klasse wird Methode genannt. Eine Operation kann durch Argumente parametrisiert sein, wobei es möglich ist, Methoden mit gleichem Namen aber unterschiedlichen Signaturen, also unterschiedlichen Parametrisierungen, zu definieren.

Weitere Modellierungselemente sind die abstrakten Klassen, die zur Modellierung von Allgemeinbegriffen dienen. Sie sind immer Oberklassen und dienen zur Spezifikation von allgemeinen Attributen und Methoden, wobei zumindest eine Untermenge der Methoden von den Subklassen implementiert werden muss. Da also mindestens eine Methode einer abstrakten Klasse nicht implementiert ist, können von ihr auch keine Exemplare gebildet werden.

Den abstrakten Klassen ähnlich sind Schnittstellen. Sie definieren lediglich ein Verhalten, jedoch keine Attribute. Klassen, die eine Schnittstelle implementieren, müssen die in ihr definierte Methodik zur Verfügung stellen. Ihr Einsatz bietet sich bei einer arbeitsteiligen Entwicklung an, bei der grundsätzliche Funktionalität von den beteiligten Partnern zusammen definiert, die Realisierung jedoch unabhängig voneinander vorgenommen werden soll.

3.1.2 Identität

Unter Identität versteht man die Zuordnung von Daten zu diskreten, unterscheidbaren Objekten, die konkret oder konzeptionell sein können. Jedes Objekt besitzt seine eigene, systemweit eindeutige Identität, durch die es klar von anderen Objekten, selbst bei gleichen Attributwerten, unterschieden werden kann. Jedes Objekt hat somit einen eigenen Identifikator, der bei seiner Erzeugung vom System vergeben wird.

3.1.3 Kapselung

Bei der Kapselung werden die extern zugreifbaren Daten eines Objekts von den internen Daten und den Implementierungsdetails getrennt. Die Daten und die Implementierungsdetails werden hinter der Schnittstelle verborgen. Die Implementierung der Schnittstelle eines Objekts kann daher ohne Auswirkungen auf das System geändert werden, was die Wartbarkeit erhöht.

3.1.4 Verantwortlichkeit

Jede Klasse ist für genau einen Aspekt des Gesamtsystems verantwortlich. Alle Eigenschaften, die in diesem Verantwortlichkeitsbereich liegen werden in einer Klasse zusammengefasst und nicht auf verschiedene Klassen verteilt. Klassen sollten keine Eigenschaften enthalten, die nicht zu dem Verantwortungsbereich gehören.

3.1.5 Generalisierung und Vererbung

Generalisierung und Vererbung dienen als Abstraktionskonzepte dazu Ähnlichkeiten zwischen Klassen zu teilen und gleichzeitig ihre Unterschiede zu erhalten. Die Generalisierung ist die Relation zwischen einer Klasse und verfeinerten Versionen dieser Klasse. Die Klasse, die verfeinert wird, wird Oberklasse und die verfeinerte Klasse Unterklasse genannt. Jede Unterklasse erbt alle Merkmale der Oberklasse. Generalisierung und Vererbung sind über eine beliebige Anzahl von Ebenen hinweg transitiv. Ein Exemplar einer Unterklasse ist gleichzeitig auch ein Exemplar aller ihrer Oberklassen. Zu den von den Oberklassen geerbten Merkmalen fügen die Unterklassen ihre eigenen Merkmale hinzu. Generalisierung vereinfacht die Modellierung, weil sie Klassen strukturiert und präzise ihre Unterschiede und Ähnlichkeiten erfasst. Durch die Vererbung von Operationen wird die Wiederverwendung von Programmcode unterstützt. Das Konzept der Vererbung wird von objektorientierten Programmiersprachen in hohem Maße unterstützt.

Merkmale der Oberklassen können von Unterklassen überschrieben werden, indem ein Merkmal mit dem gleichen Namen definiert wird. Das neue Merkmal ersetzt oder verfeinert das überschriebene Merkmal der Oberklasse. Das Überschreiben von Merkmalen kann zur präziseren Spezifikation des Merkmals oder zur Verhaltensspezifikation in Abhängigkeit von der Unterklasse dienen.

Abb. 3.1 zeigt ein Beispiel für eine Vererbung bzw. Generalisierung. Die Klasse *Geometrische Figur* ist eine Genralisierung der Klassen *Dreieck*, *Kreis* und *Rechteck*. Sie definiert z.B. eine Methode zur Flächenberechnung, die von den Unterklassen implementiert werden muss, da zur Flächenberechnung von Dreiecken, Kreisen und Rechtecken unterschiedliche Formeln verwendet werden.

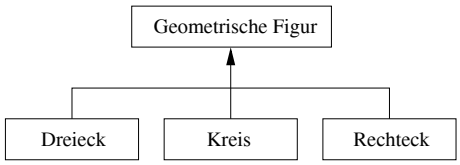

Abbildung 3.1: Beispiel einer Generalisierung bzw. Vererbung.

Die Klassen *Dreieck*, *Kreis* und *Rechteck* sind Unterklassen der Klasse *Geometrische Figur*, weshalb ihre Exemplare anstelle der Klasse *Geometrische Figur* verwendet werden können. Sie erben die Merkmale der Oberklasse wie z.B. die Methode zur Flächenberechnung. Zusätzlich können sie eigene Merkmale wie z.B. Seitenlängen oder einen Radius definieren.

3.1.6 Abstraktion

Bei der Abstraktion werden nur die wichtigsten Aspekte der Objekte betrachtet. Man beschreibt die Funktionalität eines Objekts, ohne festzulegen, wie diese implementiert ist. Hierdurch ist es möglich, Implementierungsdetails so lange wie möglich offen zu halten um sich nicht schon in frühen Entwicklungsstadien Restriktionen aufzuerlegen, die im späteren Entwurf zu Komplikationen führen können.

3.1.7 Assoziationen und Verknüpfungen

Mit Assoziationen und Verknüpfungen werden Beziehungen zwischen Klassen und Objekten hergestellt. Eine Verknüpfung ist eine physikalische oder konzeptuelle Relation auf Objektebene. Eine Assoziation beschreibt eine Gruppe von Verknüpfungen mit gemeinsamer Struktur und Semantik als Relation auf Klassenebene. Assoziationen sind in der Regel bidirektional. Jeder Assoziation ist ein Name zugeordnet, der den Zweck der Assoziation verdeutlicht.

Die Anzahl der an einer Assoziation beteiligten Objekte wird über die an der Assoziation angegebene Multiplizität festgelegt. Der Multiplizitätswert umfasst im Allgemeinen ein Intervall. Durch die Multiplizität können Annahmen, die während des Entwurfs des Modells getroffen wurden, explizit ausgedrückt werden. Ein Beispiel hierfür zeigt Abb. 3.2. Hier wird die Beziehung zwischen einer Firma und ihren Mitarbeitern dargestellt. Durch die Multiplizität wird ausgedrückt, dass jedes Unternehmen beliebig viele Mitarbeiter haben kann und jeder Mitarbeiter bei mindestens einem Unternehmen beschäftigt sein muss.

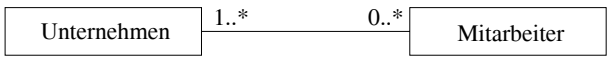

Abbildung 3.2: Beispiel einer Assoziation mit Multiplizitäten.

Genau wie Klassen können auch Verknüpfungen mit Attributen versehen werden. Als Eigenschaft einer Verknüpfung kann das Attribut keinem der beteiligten Objekte ohne Informationsverlust zugeordnet werden.

Assoziationen können auch als Klassen modelliert werden, wodurch jede Verknüpfung ein Exemplar dieser Klasse wird. Diese Art der Modellierung ist nützlich, wenn auf Verknüpfungen Operationen ausgeführt werden sollen. Die Attributierung von Assoziationen, wie oben beschrieben, ist ein Spezialfall der Modellierung von Assoziationen als Klassen.

Die Enden einer Assoziation werden als Rollen bezeichnet. Jede Rolle hat einen Namen, der genauer beschreibt, welche Rolle die jeweiligen Objekte in der Beziehung einnehmen. Rollennamen stehen am Zielende der durchlaufenen Assoziation.

3.1.8 Aggregation und Komposition

Die Aggregation bezeichnet die Relation, in der Objekte die Komponenten eines anderen Objekts repräsentieren. Die Aggregation ist eine enge Form der Assoziation, der eine zusätzliche Semantik zugrunde liegt. Aggregationen sind transitiv und antisymmetrisch. Die Aggregation ist die Zusammensetzung eines Objekts aus einer Menge von anderen Objekten. Für jedes Objekt einer solchen Zusammensetzung existiert eine Aggregation mit zugehöriger Multiplizität. Bei der Aggregation wird das zusammengesetzte Objekt als Einheit betrachtet. Es gibt auch die Möglichkeit, Aggregationen rekursiv zu definieren. Hierbei existiert irgendwo im Aggregationsbaum eine Klasse, die eine über ihr stehende Klasse aggregiert. Bei der Aggregation sind die Lebensdauern der einzelnen aggregierten Objekte nicht von der Lebensdauer der Objektgruppe abhängig.

Abb. 3.3 zeigt als Beispiel für eine Aggregation die Klasse *PKW*, die in dieser vereinfachten Darstellung ein Objekt vom Typ *Motor* und drei oder vier Objekte vom Typ *Rad* aggregiert. Weder der Motor noch die Räder sind vom Rest eines PKW existenzabhängig, da sie auch mit anderen Fahrzeugen des gleichen Typs verwendet werden können.

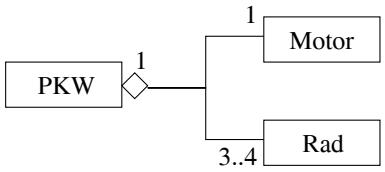

Abbildung 3.3: Beispiel einer Aggregation.

Die enge Form der Aggregation, bei der die Lebensdauern der einzelnen Objekte von der der Objektgruppe abhängen, wird Komposition genannt. Eine solche Komposition zeigt Abb. 3.4. Die Klasse *Rechnung* beinhaltet mehrere Objekte vom Typ *Rechnungsposition*. Es handelt sich hier um eine Komposition, da Rechnungspositionen zu genau einer Rechnung gehören und mit der Löschung dieser ebenfalls gelöscht werden müssen.

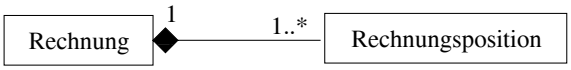

Abbildung 3.4: Beispiel einer Komposition.

3.1.9 Nachrichtenaustausch

Die Kommunikation der Objekte untereinander geschieht durch den Austausch von Nachrichten. Objekte verstehen hierbei genau solche Nachrichten, zu denen sie Operationen besitzen. Der Nachrichtenaustausch ist nicht identisch mit dem Funktions- bzw. Prozeduraufruf, wie er aus den prozeduralen Sprachen bekannt ist, da Objekte Daten und Verhalten kapseln. Ein Objekt enthält Operationen zur Bearbeitung seiner Daten und zur Ausführung zusätzlichen Verhaltens. In prozeduralen Programmiersprachen stehen diese Sachverhalte eher unverbunden hintereinander. Die Operationen bzw. Nachrichten die ein Objekt versteht können nur über das Objekt angesprochen werden. Innerhalb einer Klassenhierarchie können aufgrund der Polymorphie gleichartige Nachrichten von unterschiedlichen Objekten auf verschiedene Art und Weise interpretiert werden.

3.1.10 Polymorphie

Polymorphie bezeichnet die Tatsache, dass sich gleiche Operationen in unterschiedlichen Klassen unterschiedlich verhalten. In Klassenhierarchien bieten verschiedene Unterklassen aufgrund von Vererbung syntaktisch gleiche Operationen an, die sich jedoch entsprechend der Unterklasse unterschiedlich verhalten können. Eine Operation ist hierbei eine Aktion, die ein Objekt ausführt, oder eine Transformation, die auf einem Objekt ausgeführt wird. Eine Operation innerhalb einer Klasse wird Methode genannt. Ein weiterer Aspekt besteht in der Variation der Schnittstelle gleichnamiger Operationen innerhalb einer Klasse. Die Auswahl einer konkreten Implementierung einer Operation kann also über deren Signatur erfolgen.

3.1.11 Verbindung von Daten und Verhalten

Durch die Verbindung von Daten und Verhalten braucht beim Aufruf einer Methode nicht die Anzahl der existierenden Implementierungen berücksichtigt werden. Durch die Polymorphie von Methoden wird die Entscheidung für eine bestimmte Implementierung auf die Klassenhierarchie verlagert. Dies vereinfacht auch die Wartung, da der aufrufende Code beim Hinzufügen neuer Klassen nicht verändert werden muss.

3.2 Die Unified Modeling Language

Die bereits in der Einleitung erwähnte Unified Modeling Language (UML) ist eine universelle Beschreibungssprache für alle Arten objektorientierter Softwaresysteme. Sie wurde von Grady Booch, James Rumbaugh und Ivar Jacobson entwickelt, um die große Zahl an Verfahren im objektorientierten Entwurf sowie die große Zahl an verschiedenen grafischen Notationen in einem einheitlichen Konzept zu erfassen. Eine ausführliche Darstellung der Konzepte und Notationen der UML findet sich in [5, 8, 13, 30, 51].

Bei der Entwicklung der hauptsächlich grafischen Notation wurden u.a. folgende Prinzipien verwirklicht:

- *Einfachheit*: Verwendung weniger Konzepte und Symbole.
- *Priorisierung*: Einfache Modellierung häufiger Probleme, mehr Aufwand bei ungewöhnlichen Situationen.
- *Konsistenz*: Verwendung gleicher Konzepte in allen Bereichen der Notation.

Da die UML unabhängig von der Zielprogrammiersprache einsetzbar ist, bietet sie eine gute Basis für die Modellierung objektorientierter Systeme.

Bei der Modellierung mit der UML wird zwischen der Strukturmodellierung, der Verhaltensmodellierung und der Modellverwaltung unterschieden. Bei der Strukturmodellierung werden die für die Anwendung relevanten statischen Eigenschaften und ihre Beziehungen zu anderen Anwendungen beschrieben. Typische Diagrammarten für die Strukturmodellierung sind Klassendiagramme, Objektdiagramme und Use-Case-Diagramme. Bei der Verhaltensmodellierung werden mit dem Inter-Objekt-Verhalten und dem Intra-Objekt-Verhalten die dynamischen Systemeigenschaften beschrieben. Typische Diagrammarten für die Verhaltensmodellierung sind Sequenzdiagramme, Zustandsdiagramme und Aktivitätsdiagramme. Die Modellverwaltung beschreibt die Organisation der Modelle als hierarchische Gliederung von Einheiten. Hierzu werden auch Klassendiagramme genutzt.

Die UML soll hier nur in Ansätzen beschrieben werden. Es werden die Konzepte vorgestellt, die zum Verstehen der später dargestellten Zusammenhänge nötig sind.

3.2.1 Statische Sicht und Klassendiagramme

Die statische Sicht ist die Grundlage der UML. In ihr sind alle Konzepte, die in einem System Wichtigkeit haben, enthalten. Die Datenstrukturen und die auf ihnen operierende Methodik wird in Klassen zusammengeführt. Sie beschreibt alle Teile eines Systems als Elemente einer diskreten Modellierung, ohne auf das Verhalten der Teile im Verlauf der Lebensdauer des Systems einzugehen. Die wichtigsten Elemente der statischen Sicht sind Klassen und ihre Beziehungen, die durch die Verwendung von Klassendiagrammen modelliert werden. Im Folgenden werden nun die wichtigsten Bestandteile der Klassendiagramme erläutert.

3.2.1.1 Klassen

Wie bereits in Abschnitt 3.1.1 dargelegt, beschreibt eine Klasse eine Menge von Objekten mit gemeinsamer Struktur, gemeinsamem Verhalten, gemeinsamen Beziehungen und gemeinsamer Semantik.

Klassen stellen in sich abgeschlossene Einheiten dar, die einen Zustand sowie Methoden zur Modifikation des Zustands besitzen. Oft sind auch noch Methoden zur Traversierung des Zustands vorhanden. Der Zugriff auf ein Objekt einer Klasse sollte grundsätzlich nur über die, durch die Methoden bereitgestellte Schnittstelle geschehen.

Grafisch können Klassen auf verschiedene Arten dargestellt werden. Im ersten Fall, wie in Abb. 3.5, wird die Klasse als Rechteck mit dem Klassennamen darin dargestellt. Diese Darstellung wird gewählt, wenn zur Darstellung von Beziehungen zwischen Klassen auf einer hohen Abstraktionsebene nur die Klassen, nicht aber die Attribute oder die Methoden der Klassen, benötigt werden.

Abbildung 3.5: Einfache Darstellung einer Klasse

Im zweiten Fall, wie in Abb. 3.6, werden neben dem Klassennamen auch noch die Attribute und die Methoden der Klasse dargestellt. Diese vollständige Darstellung wird gewählt, wenn neben den Beziehungen zu anderen Klassen auch die Spezifikation und das Verhalten der Klasse von Interesse sind.

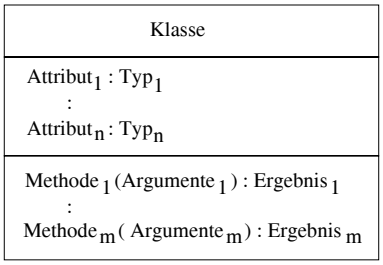

Abbildung 3.6: Klassendarstellung mit Attributen und Methoden

Es gibt noch zwei weitere Arten der Darstellung, die zum einen, wie in Abb. 3.7 zu sehen, nur die Attribute oder zum anderen, wie in 3.8 zu sehen, nur die Operationen einer Klasse wiedergeben.

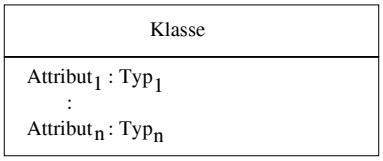

Abbildung 3.7: Klassendarstellung mit Attributen

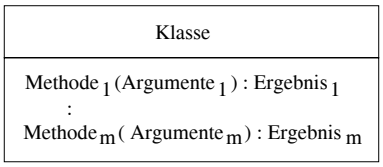

Abbildung 3.8: Klassendarstellung mit Methoden

Attribute sowie auch Methoden können öffentlich oder privat sein. Dieser Sachverhalt kann durch ein, dem Attribut- oder Methodennamen vorangestelltes, Pluszeichen im öffentlichen Fall bzw. einem Minuszeichen im privaten Fall ausgedrückt werden.

Optional können im Kopf der grafischen Notation noch Stereotype oder Eigenschaftswerte dargestellt werden. Eigenschaftswerte, wie z.B. das Schlüsselwort *abstract* zur Kennzeichnung einer abstrakten Klasse, werden in geschweiften Klammern unterhalb des Klassennamens plaziert.

3.2.1.2 Assoziation

Eine Assoziation ist, wie in Abschnitt 3.1.7 beschrieben, die allgemeinste Beziehung zwischen Klassen in einem objektorientierten System. Eine Assoziation zwischen Klassen besteht immer dann, wenn es entweder eine physische oder eine konzeptionelle Beziehung zwischen diesen Klassen gibt. Solche Beziehungen können auch rekursiv sein oder zwischen mehreren Klassen bestehen.

Grafisch wird eine Assoziation, wie in Abb. 3.9, durch eine Kante zwischen den beteiligten Klassen dargestellt. Die Kante kann mit dem Assoziationsnamen beschriftet sein. Durch einen dreieckigen Pfeil an den Seiten des Assoziationsnamens kann die Richtung des Namens der Beziehung dargestellt werden.

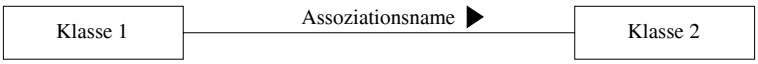

Abbildung 3.9: Assoziation

Durch eine gerichtete Assoziation kann die Navigierbarkeit zwischen den an der Assoziation beteiligten Klassen dargestellt werden. Hierbei hat die Klasse, zu der navigiert werden kann, keine Kenntnis davon, mit wie vielen Klassen sie assoziiert ist. Im Falle einer bidirektionalen Assoziation, wie in Abb. 3.9, werden die Richtungspfeile weggelassen. Ein Beispiel für eine gerichtete Assoziation gibt Abb. 3.10.

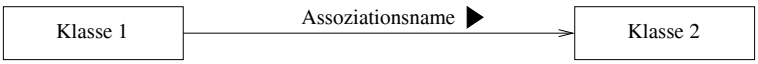

Abbildung 3.10: Gerichtete Assoziation

3.2.1.3 Kardinalitäten

Kardinalitäten geben an, wie viele Objekte der jeweiligen Klassen an einer Beziehung beteiligt sind. Die Kardinalität ist die Anzahl von Exemplaren einer gegenüberliegenden Klasse bezüglich eines Exemplars der gegebenen Klasse. Für jedes Ende der Beziehung kann eine Kardinalität angegeben werden.

Die Kardinalitäten sind hierbei wie folgt definiert:

Kardinalität	**Bedeutung**
1	genau eins
0,1	null oder eins
a..b	zwischen a und b (einschliesslich)
a,b	genau a oder genau b
0..*	grösser oder gleich 0
1..*	grösser oder gleich 1

Tabelle 3.1: Verschiedene Kardinalitäten

Wird keine Kardinalität angegeben, so wird von 0..* ausgegangen.

Ein Beispiel für die Angabe von Kardinalitäten zeigt Abb. 3.11. Hier ist einem Exemplar von Klasse 1 mindestens ein Exemplar von Klasse 2 zugeordnet und einem Exemplar von Klasse 2 ist genau ein Exemplar von Klasse 1 zugeordnet.

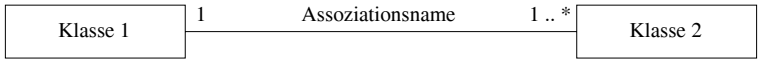

Abbildung 3.11: Kardinalität

3.2.1.4 Rollen

Die Rollen, die Klassen in einer Beziehung spielen, können explizit benannt werden, um Unklarheiten zu vermeiden. Hierbei ist die Rolle die Sichtweise eines Objekts durch das gegenüberliegende Objekt.

Ein Beispiel für den Einsatz von Rollen gibt Abb. 3.12. Hierbei ist Klasse 2 unter Klasse 1 angeordnet.

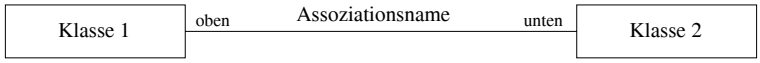

Abbildung 3.12: Rollen

3.2.1.5 Aggregation

Die bereits in Abschnitt 3.1.8 vorgestellte Aggregation zeigt die Zusammengehörigkeit mehrerer Teile zu einem Ganzen an. Das Ganze ist dabei ein zusammengefasstes Objekt, dessen Behandlung auch die Teile betrifft. Das Ganze handelt somit stellvertretend für die Teile. Die Aggregation wird eingesetzt, wenn von zahlreichen Objekten in Bestandteilhierarchien abstrahiert werden soll. Auf diese Weise ist man in der Lage, nur mit den komplexen Objekten umgehen zu müssen, ohne die Einzelkomponenten zu berücksichtigen. Die Aggregation zeigt an, dass eine Klasse in einer anderen enthalten ist, sagt jedoch nichts über die Lebensdauern der einzelnen Objekte aus. Da man eine Aggregation auch als Form der Assoziation verstehen kann, ist die Angabe eines Namens und der Richtung möglich, aber nicht nötig, da beides schon durch die zugrundeliegende Semantik der Teile-Ganzes-Beziehung implizit vorhanden ist. Die Angaben können jedoch zur Verdeutlichung der Sachverhalte dienen. Auch bei der Aggregation ist die Angabe von Kardinalitäten möglich.

Grafisch wird eine Aggregation, wie in Abb. 3.13, durch eine hohle Raute an der Klasse, die die andere enthält, dargestellt.

Abbildung 3.13: Aggregation

3.2.1.6 Komposition

Die Komposition wird als enge Aggregation gesehen, die eine Verbindung der Teile mit dem Ganzen auch hinsichtlich der Lebensdauer der Objekte bedeutet. Die Komposition modelliert Existenzabhängigkeiten zwischen Objekten. Auch bei der Komposition ist die Angabe von Kardinalitäten möglich. Die in 3.13 gemachten Aussagen zur Angabe von Namen und Richtungen gelten auch bei der Komposition.

Hinsichtlich der Angabe von Kardinalitäten ist zu beachten, dass jedes Teil bezüglich der Lebensdauer an ein Ganzes gebunden ist. An der Raute kann also als Kardinalität lediglich der Wert 1 stehen. Abgesehen davon kann ein Teil jedoch beliebig viele andere Assoziationen haben und auch anderen Aggregationen unterliegen.

Grafisch wird die Komposition, wie in Abb. 3.14, durch eine ausgefüllte Raute an der Klasse, die die andere enthält, dargestellt.

Abbildung 3.14: Komposition

3.2.1.7 Generalisierung

Die Generalisierung bezeichnet die Übernahme von Eigenschaften und Operationen einer Beschreibungseinheit von einer anderen. Die Generalisierung wird innerhalb der Modellierung in verschiedenen Ausprägungen eingesetzt, hierbei jedoch oft in Form der statischen Vererbung zwischen Klassen, um die Beziehung einer allgemeinen zu einer spezielleren Klasse zu modellieren. Die speziellere Klasse ist konsistent zur allgemeineren, kann jedoch weitere Merkmale enthalten. Die Generalisierung wurde bereits in Abschnitt 3.1.5 vorgestellt.

Grafisch wird die Generalisierung, wie in Abb. 3.15, durch einen ausgefüllten Pfeil dargestellt, der von der abgeleiteten zur Basisklasse zeigt.

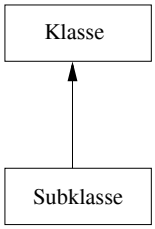

Abbildung 3.15: Generalisierung

3.2.1.8 Stereotype

Durch Stereotype lassen sich die möglichen Verwendungen eines Modellelements klassifizieren. Hierdurch wird die Art der Verwendung des Modellelements verdeutlicht und eine visuelle Unterscheidung möglich. Jedes Element kann durch genau einen Stereotyp klassifiziert werden, wodurch seine Semantik verändert werden kann.

Die wichtigsten Stereotypen sind die deskriptiven und restriktiven. Deskriptive Stereotypen sind solche, die Verwendungskontexte beschreiben. Sie lassen sich als eine Art der standardisierten Kommentierung nutzen. Hierdurch können z.B. Klassen einer bestimmten Architekturschicht zugeordnet werden. Restriktive Stereotypen definieren formale Einschränkungen auf vorhandene Modellierungselemente. Sie können das Vorhandensein bestimmter Eigenschaften erzwingen. Als Beispiel lässt sich hier der Stereotyp $\ll interface \gg$ heranziehen, der eine so gekennzeichnete Klasse auf die Verwendung von abstrakten Methodendeklarationen einschränkt.

Die grafische Darstellung geschieht, wie in Abb. 3.16 dargestellt, dadurch, dass der Name des Stereotyps in spitze Klammern eingeschlossen über dem Elementnamen platziert wird.

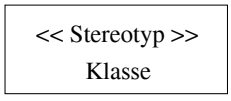

Abbildung 3.16: Stereotyp

3.2.1.9 Realisierung

Die Realisierung modelliert die Beziehung einer Klasse, die ein Verhalten implementiert, zu der Schnittstelle, die die Verhaltensbeschreibung enthält. Die implementierende Klasse muss hierbei die gesamte Funktionalität der Spezifikation implementieren. Schnittstellen wurden bereits in Abschnitt 3.1.1 beschrieben.

Grafisch wird die Realisierung, wie in Abb. 3.17, durch einen nicht ausgefüllten Pfeil mit einer gestrichelten Linie dargestellt, der von der implementierenden Klasse zur Schnittstelle gerichtet ist.

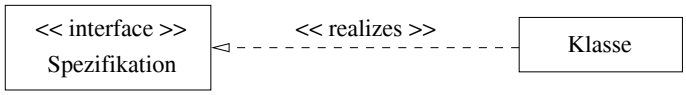

Abbildung 3.17: Realisierung

Optional kann, wie in Abb. 3.17 dargestellt, über der Linie der Stereotyp ≪ *realizes* ≫ platziert werden.

3.2.2 Dynamische Sicht und Aktivitätsdiagramme

Die dynamische Sicht beschreibt das Verhalten eines Systems über den Verlauf der Zeit. Dieses Verhalten kann z.B. als Serie von Veränderungen, die sich im System abspielen, beschrieben werden. Diese Veränderungen können sowohl das Inter-Objekt-Verhalten als auch das Intra-Objekt-Verhalten widerspiegeln.

3.2.2.1 Aktivitätsgrafen

Ein Aktivitätsgraf ist eine Variante eines Zustandsdiagramms, das den Programmfluss bzw. die Ausführung von Operationen modelliert. Die Knoten eines Aktivitätsgrafen repräsentieren Aktivitäten, also einzelne Schritte in einem Verarbeitungsablauf. Jede Kante zwischen den Aktivitäten repräsentiert eine Aktivitätstransition, die sofort nach Beendigung einer Aktivität zur nächsten Aktivität durchlaufen wird. Es wird angenommen, dass die Berechnungen ohne externe Ereignisse voranschreiten.

Aktivitätsgrafen können Verzweigungen sowie Aufsplittungen und Zusammenführungen des Kontrollflusses enthalten.

3.2.2.2 Aktivitätsdiagramme

Ein Aktivitätsdiagramm ist die grafische Repräsentation eines Aktivitätsgrafen. Es enthält grafische Symbole für die oben beschriebenen Möglichkeiten der Modellierung.

Eine Aktivität wird durch ein Rechteck mit abgerundeten Ecken dargestellt. Innerhalb des Rechtecks ist eine Beschreibung der Aktivität notiert. Ein Beispiel für eine Aktivität zeigt Abbildung 3.18.

Abbildung 3.18: Grafische Darstellung einer Aktivität

Sequentielle Abhängigkeiten und Aktivitätstransitionen werden durch gerichtete Kanten zwischen den Aktivitätszuständen dargestellt. Ein Beispiel einer Aktivitätstransition zeigt Abbildung 3.19.

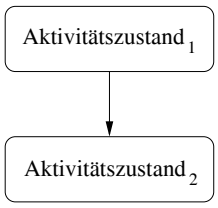

Abbildung 3.19: Grafische Darstellung einer Aktivitätstransition

Verzweigungen des Programmflusses werden durch Rauten dargestellt. Im Falle einer Verzweigung werden die Aktivitätstransitionen der Verzweigung mit den entsprechenden Bedingungen, die zur Wahl der jeweiligen Verzweigung führen, beschriftet. Ein Beispiel für das Verzweigen des Kontrollflusses zeigt Abbildung 3.20.

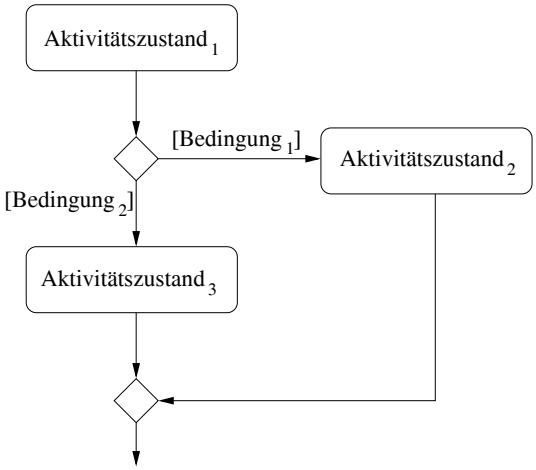

Abbildung 3.20: Grafische Darstellung einer Verzweigung

Das Aufsplitten oder Zusammenführen des Kontrollflusses wird durch einen Synchronisationsbalken dargestellt, von dem aus mehrere Aktivitätstransitionen ausgehen oder in den mehrere Aktivitätstransitionen münden. Ein Beispiel für das Aufsplitten bzw. das Zusammenführen des Kontrollflusses zeigt Abbildung 3.21. Hierbei wird durch den oberen Balken das Aufsplitten des Kontrollflusses auf drei nebenläufige Aktivitäten dargestellt. Durch den unteren Balken wird eine Synchronisation des verzweigten Kontrollflusses vorgenommen. Der Kontrollfluss schreitet erst dann voran, wenn alle drei nebenläufigen Aktivitäten beendet sind.

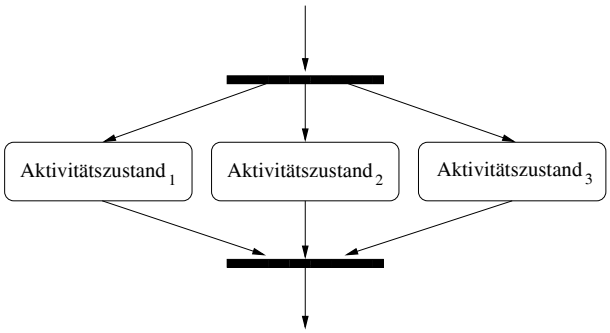

Abbildung 3.21: Grafische Darstellung des Aufsplittens und Zusammenführens des Kontrollflusses

Ein Aktivitätsdiagramm kann neben dem Kontrollfluss auch den Zustand eines Ein- oder Ausgabeobjekts zeigen. Bei einem Eingabeobjekt wird eine gestrichelte, gerichtete Kante vom Objekt zum Aktivitätszustand gezogen. Bei einem Ausgabeobjekt wird eine gestrichelte, gerichtete Kante vom Aktivitätszustand zum Objekt gezogen. Hat ein Aktivitätszustand mehr als ein Ausgabeobjekt, so werden die Kanten von einem Synchronisationsbalken

aus gezogen. Hat ein Aktivitätszustand mehr als ein Eingabeobjekt, so werden die Kanten zu einem Synchronisationsbalken hin gezogen. Ein Beispiel für einen Aktivitätszustand mit mehreren Ein- und Ausgabeobjekten zeigt Abbildung 3.22.

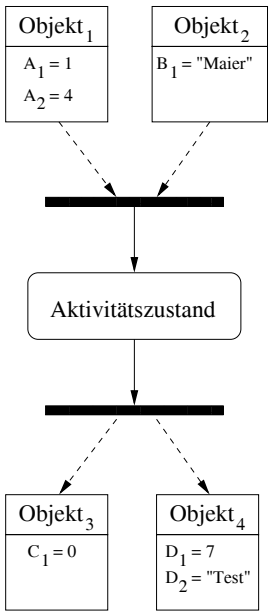

Abbildung 3.22: Grafische Darstellung einer Aktivität mit mehreren Ein- und Ausgangsobjekten

Beispiel eines Aktivitätsdiagramms

Ein einfaches Beispiel für ein Aktivitätsdiagramm findet sich in Abb. 3.23, wo das aufsteigende Sortieren dreier Zahlen dargestellt ist.

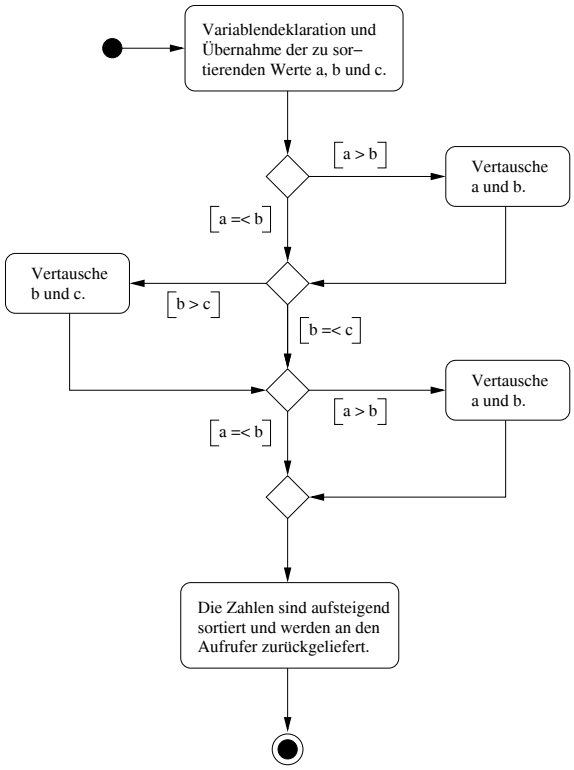

Abbildung 3.23: Beispiel eines Aktivitätsdiagramms

3.3 Entwurfsmuster

Beim Entwurf von objektorientierten Systemen existieren eine Reihe von wiederkehrenden Problemstellungen, für die im Laufe der Zeit verschiedene Lösungsansätze gefunden und implementiert wurden. Ihnen ist gemeinsam, dass sie durch Verwendung wiederkehrender Muster von Klassen und kommunizierender Objekte definiert werden. Ziel der Entwurfsmuster ist es, diese wiederkehrenden Muster zu klassifizieren.

Jedes Entwurfsmuster benennt, erläutert und bewertet einen wichtigen und wiederkehrenden Entwurf in objektorientierten Systemen. Entwurfsmuster vereinfachen die Wiederverwendung von erfolgreichen Entwürfen und Architekturen. Die Darstellung bewährter Techniken als Entwurfsmuster machen diese Techniken leichter verständlich und helfen, zwischen Entwurfsalternativen, die ein System wiederverwendbar machen, und solchen, die die Wiederverwendbarkeit einschränken, zu unterscheiden. Auch die Wartung und Erweiterbarkeit der Systeme wird verbessert.

Ein Entwurfsmuster benennt, abstrahiert und identifiziert die relevanten Aspekte eines Lösungsansatzes für eine bestimmte Problemstellung bei der Softwareentwicklung. Diese Aspekte beschreiben, warum das Muster für die Entwicklung eines wiederverwendbaren, objektorientierten Entwurfs nützlich ist. Das Entwurfsmuster identifiziert die teilnehmenden Klassen und Objekte, die Rollen, die sie spielen, die Interaktion der Rollen und die ihnen zugeteilten Aufgaben. Jedes Entwurfsmuster konzentriert sich auf ein bestimmtes objektorientiertes Entwurfsproblem. Es beschreibt wann es einsetzbar ist, ob es angesichts einschränkender Randbedingungen eingesetzt werden kann und welche Konsequenzen sein Einsatz hat.

Entwurfsmuster werden in erzeugende, strukturorientierte und verhaltensorientierte Muster unterteilt.

Erzeugende Muster verstecken den Prozess der Objekterzeugung und helfen somit ein System unabhängig davon zu machen, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden. Sie sind vor allem dann von Bedeutung, wenn Systeme mehr von Objektkomposition als von Vererbung abhängen. Die Muster kapseln das Wissen um die konkreten, vom System verwendeten Klassen und verstecken, wie Exemplare dieser Klassen erzeugt und zusammengefügt werden. Alles, was eine Anwendung insgesamt über die Objekte weiß, wird durch abstrakte Klassen definiert, deren Subtypen durch die entsprechenden Muster erzeugt werden. Beispiele für erzeugende Muster sind z.B.:

- Das Entwurfsmuster *Fabrik*, das eine Schnittstelle zum Erzeugen von verwandten oder voneinander abhängigen Objekten bietet, ohne ihre konkrete Klasse zu benennen. Die Erzeugung der Objekte geschieht hierbei auf der Basis von Daten, die der Fabrik übergeben werden. Eine Beschreibung dieses Entwurfsmusters befindet sich in Abschnitt 3.25.

- Das Entwurfsmuster *Singleton*, das die Einmaligkeit eines Exemplars einer Klasse sicherstellt. Dies geschieht durch Abfangen von Befehlen zur Erzeugung neuer Exemplare. Ausser einem bereits vorhandenen können keine weiteren Exemplare existieren.

- Das Entwurfsmuster *Erbauer*, das die Konstruktion eines komplexen Objekts von seiner Repräsentation trennt. Hierdurch kann derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen. Neue Repräsentationen können dem System jederzeit hinzugefügt werden.

Strukturmuster befassen sich mit der Komposition von Klassen und Objekten, um größere Strukturen zu bilden. Klassenbasierte Strukturmuster benutzen Vererbung, um Schnittstellen oder Implementierungen zusammenzuführen. Solche Muster sind z.B. hilfreich, um unabhängig voneinander entwickelte Bibliotheken zusammenarbeiten zu lassen. Objektbasierte Strukturmuster beschreiben Mittel und Wege, Objekte zusammenzuführen, um neue Funktionalität zu gewinnen. Durch die Objektkomposition ergibt sich die Möglichkeit, das Kompositionsgefüge zur Laufzeit zu ändern, was mit statischer Klassenkomposition nicht möglich ist. Beispiele für Strukturmuster sind z.B.:

- Das Entwurfsmuster *Adapter*, das die Schnittstelle einer Klasse an eine andere, von den Klienten erwartete Schnittstelle anpasst. Durch dieses Muster können Klassen zusammenarbeiten, die aufgrund von inkompatiblen Schnittstellen sonst hierzu nicht in der Lage wären.

- Das Entwurfsmuster *Fassade*, das eine einheitliche Schnittstelle zu einer Menge von Schnittstelle eines Subsystems bietet. Die einheitliche Schnittstelle vereinfacht die Nutzung des Subsystems.
- Das Entwurfsmuster *Kompositum*, das Objekte zu Baumstrukturen zusammenfügt, um Teile-Ganzes-Hierarchien zu repräsentieren. Dieses Muster ermöglicht es den Klienten, einzelne Objekte sowie Kompositionen von Objekten einheitlich zu behandeln.

Verhaltensmuster befassen sich mit Algorithmen und der Zuweisung von Zuständen zu Objekten. Sie beschreiben nicht nur Muster von Objekten oder Klassen, sondern auch die Muster der Interaktion zwischen ihnen. Klassenbasierte Verhaltensmuster verwenden Vererbung, um das Verhalten unter den Klassen zu verteilen. Objektbasierte Verhaltensmuster verwenden die Objektkomposition anstelle der Vererbung. Die Muster beschreiben, wie eine Gruppe von Objekten zusammenarbeitet, um eine Aufgabe zu erledigen. Ein wichtiger Aspekt hierbei ist, wie die Objekte einander bekannt sind. Beispiele für Verhaltensmuster sind z.B.:

- Das Entwurfsmuster *Strategie*, das eine Familie von Algorithmen definiert. Die Algorithmen sind in eigenen Klassen gekapselt und sind somit austauschbar. Die Algorithmen können unabhängig von den Klienten variiert werden und neue Algorithmen können flexibel in das System integriert werden. Eine Beschreibung dieses Entwurfsmusters befindet sich in Abschnitt 3.3.1.
- Das Entwurfsmuster *Iterator*, das eine Möglichkeit bietet, auf die Elemente eines zusammengesetzten Objekts sequentiell zugreifen zu können, ohne die Kapselung des Objekts zu verletzen. Die Zuständigkeiten für den Zugriff auf das zusammengesetzte Objekt und die Funktionalität zur Traversierung werden dem Iterator-Objekt zugeteilt.
- Das Entwurfsmuster *Besucher*, das eine, auf den Elementen einer Objektstruktur auszuführenden Operation in einem Objekt kapselt. Die Elemente der Objektstruktur enthalten eine Operation zur Entgegennahme eines Besucher-Objekts. Hierdurch können neue Operationen zu späteren Zeitpunkten in das System integriert werden.

In diesem Abschnitt werden nur die Entwurfsmuster beschrieben, die bei der Implementierung des hier beschriebenen Systems verwendet werden. Die grafische Darstellung wird durch die im vorigen Abschnitt beschriebene UML vorgenommen. Das Verständnis der Entwurfsmuster und ihrer Vorteile wird durch die, in den Abschnitten 3.1 und 3.1 vorgestellten objektorientierten Konzepte erleichtert. Einen Überblick über Entwurfsmuster und ihre Anwendung bieten [9, 14].

3.3.1 Das Entwurfsmuster Strategie

Dieses Entwurfsmuster [15] ist ein objektbasiertes Verhaltensmuster. Es definiert eine Familie von Algorithmen, wobei jeder einzelne Algorithmus gekapselt und somit austauschbar wird. Hierdurch ist es möglich, den Algorithmus unabhängig von den ihn nutzenden Klienten zu variieren. Neue Algorithmen können durch diesen Ansatz flexibel in das System integriert werden, wenn sie an eine vorgegebene Schnittstelle anpassbar sind.

Das Strategiemuster bietet sich in den folgenden Fällen zur Implementierung an:

- Es gibt viele verwandte Klassen, die sich nur in ihrem Verhalten unterscheiden. Klassen können so mit verschiedenen Verhaltensweisen konfiguriert werden.
- Es werden unterschiedliche Varianten eines Algorithmus benötigt, die sich z.B. hinsichtlich der Laufzeit und des Speicherbedarfs unterscheiden. Diese Varianten können dann als eigenständige Klassenhierarchie von Algorithmen implementiert werden.
- Ein Algorithmus verwendet Daten, die den Klienten nicht bekannt sein sollen. Die Verwendung dieses Musters vermeidet die Offenlegung von Datenstrukturen des Algorithmus.

Die Struktur des Strategiemusters zeigt Abb. 3.24.

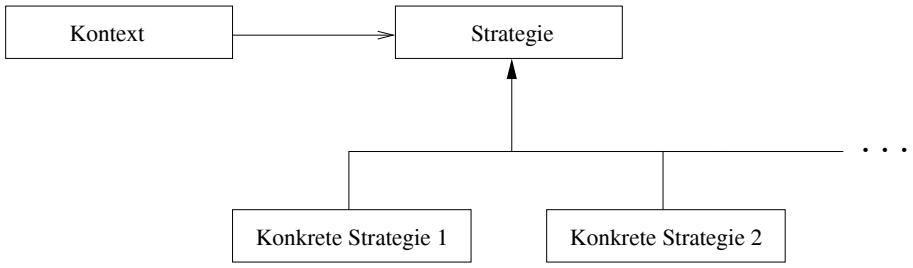

Abbildung 3.24: Grafische Darstellung des Entwurfsmusters *Strategie*

Die Klasse *Strategie* definiert eine Schnittstelle, die von allen unterstützten Algorithmen angeboten wird. Diese Schnittstelle wird von den *Kontext*-Objekten verwendet, um den in *Konkretestrategie* implementierten Algorithmus aufzurufen. Die Klasse *Konkretestrategie* implementiert den Algorithmus unter Verwendung der in *Strategie* definierten Schnittstelle. Die Klasse *Kontext* wird mit einem *Konkretestrategie*-Objekt konfiguriert und kann eine Schnittstelle bereitstellen, um den *Strategie*-Objekten den Zugriff auf seine Daten zu ermöglichen.

Die Nutzung dieses Entwurfsmusters bietet eine Reihe von Vorteilen, die nun aufgelistet werden:

- Hierarchien von Strategieklassen definieren jeweils Familien von Algorithmen und Verhalten, die von Kontextobjekten wiederverwendet werden können.
- Es bietet eine Alternative zur Unterklassenbildung. Es findet keine Vermischung der Implementierungen von Kontext und Algorithmus statt. Bei der Unterklassenbildung können die Algorithmen nicht zur Laufzeit verändert werden.
- Es wird keine Form von Bedingungsanweisungen zur Auswahl des gewünschten Verhaltens benötigt.
- Es kann zwischen verschiedenen Implementierungen mit unterschiedlichem Laufzeitverhalten und Speicherplatzbedarf gewählt werden.

3.3.2 Das Entwurfsmuster Fabrik

Bei diesem Entwurfsmuster handelt es sich um ein objektbasiertes Erzeugungsmuster. Es bietet eine Schnittstelle zum Erzeugen von verwandten oder voneinander abhängigen Objekten, ohne ihre konkrete Klasse zu benennen. Die Erzeugung der Objekte geschieht auf der Basis von Daten, die der Fabrik übergeben werden. Oft haben die Klassen, die durch die Fabrik erzeugt werden können, eine gemeinsame Oberklasse, die das Verhalten spezifiziert. Die Unterklassen sind jeweils für eine bestimmte Aufgabe optimiert.

Das Fabrikmuster bietet sich in den folgenden Fällen zur Implementierung an:

- Das System soll unabhängig davon sein, wie die genutzten Objekte erzeugt werden.
- Die konkrete Implementierung eines Objekts geschieht in einer Unterklasse, basierend auf als Parameter übergebenen Daten.
- Die Erzeugung der Objekte sowie die Art der Erzeugung der Objekte soll in einer Klasse zentralisiert werden.

Eine mögliche Struktur des Fabrikmusters zeigt Abb. 3.25

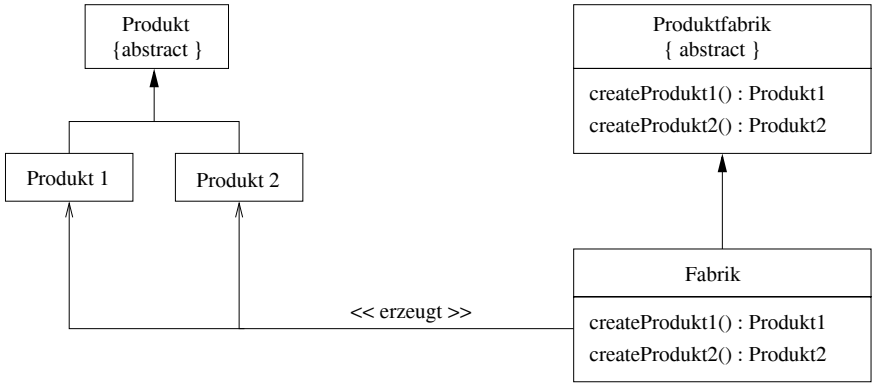

Abbildung 3.25: Grafische Darstellung des Entwurfsmusters *Fabrik*

Die abstrakte Klasse *Produkt* definiert die Grundfunktionalität der durch die Klasse *Fabrik* erzeugten Objekte. Die Klassen *Produkt1* und *Produkt2* definieren Unterklassen mit voller Funktionalität, die von dem Fabrikobjekt zurückgeliefert werden kann. Die Klasse *Produktfabrik* definiert Methoden, um Objekte vom Typ *Produkt* zu erzeugen und an das anfragende Objekt zu liefern. Die Entscheidung, welches Produkt geliefert wird, hängt im Wesentlichen von den an die Produktfabrik übergebenen Parametern ab. Es ist auch möglich, dass Fabriken Methoden zum Erzeugen mehrerer verschiedener Produkthierarchien enthalten.

Die Nutzung dieses Entwurfsmusters bietet eine Reihe von Vorteilen, die nun aufgelistet werden:

- Dieses Entwurfsmuster ermöglicht es, die Klienten von den Implementierungsklassen zu isolieren, da die Fabrik für die Erzeugung der Objekte zuständig ist. Die Namen der Produktklassen sind in der Implementierung der Fabrik isoliert und erscheinen nicht im Code der Klienten.
- Die Klienten brauchen sich nicht um die Erzeugung der Objekte zu kümmern. Sie kennen die Funktionalität der Produkte durch die Schnittstelle der Produktklasse.

Kapitel 4

Das Datenmodell

Nachdem im vorangegangenen Kapitel die Grundlagen der objektorientierten Modellierung sowie die UML als grafische Notation vorgestellt wurden, werden in diesem Kapitel einige zentrale Klassen des Datenmodells beschrieben. Das in diesem Abschnitt beschriebene Klassenmodell der geodätischen Entitäten stammt aus [53]. Es modelliert alle Messungen, Daten, Netztopologien, Punktmengen und Zusatzinformationen, die im Rahmen eines Ingenieurprojekts erfasst und ausgewertet werden sollen. Diese Modellierung dient der Unterstützung eines durchgängigen Datenflusses mit den Verarbeitungsschritten wie sie in Abbildung 2.1 dargestellt sind. Im Rahmen dieser Arbeit sollen nur einige zentrale Klassen vorgestellt werden, auf die in den folgenden Kapiteln Bezug genommen wird. Die vollständige Beschreibung des Klassenmodells findet sich in [53]. Dort werden auch die Entwurfsentscheidungen zur Modellierung erläutert.

4.1 Die Klasse Punkt

In der Ingenieurgeodäsie wird das zu überwachende Objekt durch eine Menge eindeutig bestimmbarer Punkte diskretisiert. Diese Punkte werden in zeitlichen Abständen beobachtet, um ihre Koordinaten zu ermitteln. Der Punkt ist ein zentrales Element der Datenmodellierung und ist daher ein komplexes Aggregat aus verschiedenen Klassen, was in Abb. 4.1 dargestellt ist.

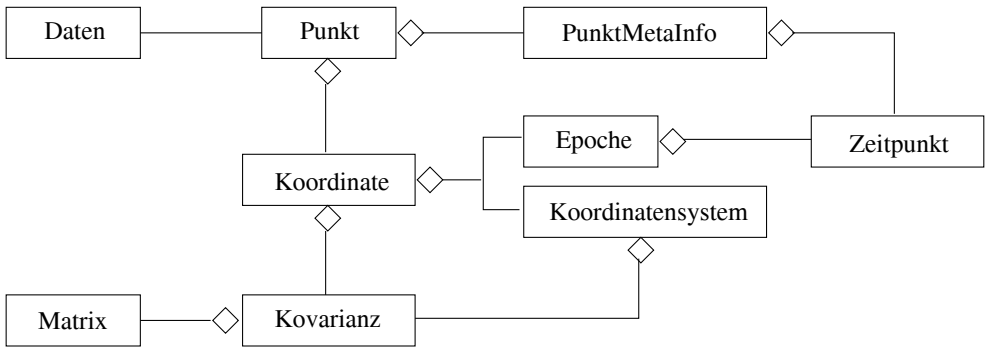

Abbildung 4.1: Die Klasse *Punkt* und ihre Beziehungen

Die Klasse *Punkt* besitzt Attribute zur Aufnahme eines Punktnamens und einer Punktnummer sowie die in Abbildung 4.1 dargestellten Assoziationen. Die Klasse *PunktMetaInfo* dient zur Modellierung von Zusatzinformationen, die für die nummerischen Auswertungen nicht von Bedeutung sind, aber bei der Planung und zur Koordinierung von Messungen durchaus von Interesse sein können. Die Klasse *Koordinate* modelliert die räumliche Position des Punktes. Sie erlaubt die Angabe dieser Position sowohl durch geozentrische Koordinaten wie auch durch ellipsoidische Koordinaten. Die Umrechnung zwischen beiden Systemen [21] kann durch eine Methode der Klasse erfolgen. Um auch eine geometrische dreidimensionale Netzausgleichung zu unterstützen, werden auch Lotabweichungsparameter modelliert. Die Klasse *Koordinatensystem* modelliert den Bezug der Koordinate zu einem Globalsystem. Die Klasse enthält die Parameter einer Ähnlichkeitstransformation, die auf die Koordinaten angewendet werden muss, um diese in das Globalsystem zu transformieren. Die Klasse *Kovarianz* dient zur Modellierung der Varianz- und Kovarianzinformationen, die im Rahmen eines vermessungstechnischen Projekts anfallen können. Sie besteht aus einer Kofaktormatrix und einem Multiplikator. Die Klasse *Epoche* modelliert die Informationen über den Zeitpunkt, an dem ein Punkt vermessen wurde. Hierzu wird die Klasse *Zeitpunkt* verwendet, die Informationen über Jahr, Monat, Tag, Stunde, Minute, Sekunde und Millisekunde kapselt. Zusätzlich kann ein Exemplar der Klasse *Epoche* mit einem Namen versehen werden.

Die Exemplare der Klasse *Punkt* können Beziehungen zu verschiedenen Exemplaren der Klasse *Daten* haben. Diese modellieren die verschiedenen Daten, die während der Beobachtungen an einem Punkt aufgenommen werden. Hierbei kann es sich z.B. um Temperaturen, die Luftfeuchtigkeit oder den Luftdruck handeln. Die

Modellierung der Daten enthält eine breite Klassenhierarchie, wobei die Klasse *Daten* die wichtigsten Modellierungselemente kapselt. Da die Modellierung der Daten für diese Arbeit nicht weiter von Interesse ist, wird sie nur der Vollständigkeit wegen erwähnt.

Um die verschiedenen in der Geodäsie vorkommenden Punkttypen zur Verfügung stellen zu können, existieren vier Subtypen der Kasse *Punkt*, was in Abbildung 4.2 dargestellt ist.

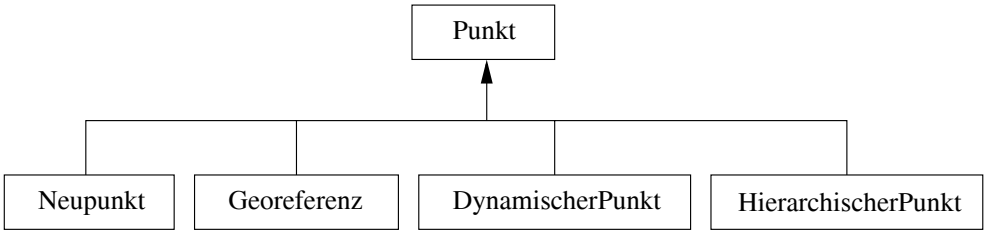

Abbildung 4.2: Subtypen der Klasse *Punkt*

Die Klasse *Neupunkt* dient zur Modellierung von Neupunkten, deren Koordinaten durch die Netzausgleichung bestimmt werden. Die Klasse *Georeferenz* dient zur Modellierung eines Netzpunktes für geotechnische und geophysikalische Messungen, die nicht zwangsläufig mit den Punkten der geodätischen Beobachtungen zusammenfallen. Die Klasse *DynamischerPunkt* dient zur Modellierung von stochastischen Anschlusspunkten im Rahmen der Netzausgleichung und die Klasse *HierarchischerPunkt* zur Modellierung von hierarchischen Anschlusspunkten im Rahmen der Netzausgleichung.

4.2 Die Klasse Messungstyp

Neben den Daten, die direkt an einem Punkt erfasst werden, existieren in der Ingenieurgeodäsie zahlreiche Messungen, die zwischen zwei bzw. drei Punkten vorgenommen werden. Zur Modellierung dieser Messungen existiert eine Klassenhierarchie, deren Wurzel die Klasse *Messungstyp* bildet. Diese abstrahiert von den eigentlichen Messungen und stellt Attribute zur Aufnahme von Werten zur Verfügung, die alle Messungen gemeinsam haben. Hierzu gehören z.B. der Messwert und seine Varianz, der Standpunkt, der Zielpunkt oder die im Zuge der Netzausgleichung berechnete Verbesserung. Zusätzlich zu diesen fachlichen Attributen enthält die Klasse noch Attribute, die von den verschiedenen Verarbeitungsschritten genutzt werden, um die Art der Verarbeitung abzulegen. Auch die Klasse *Messungstyp* ist ein komplexes Aggregat aus verschiedenen anderen Klassen, was in Abbildung 4.3 dargestellt ist.

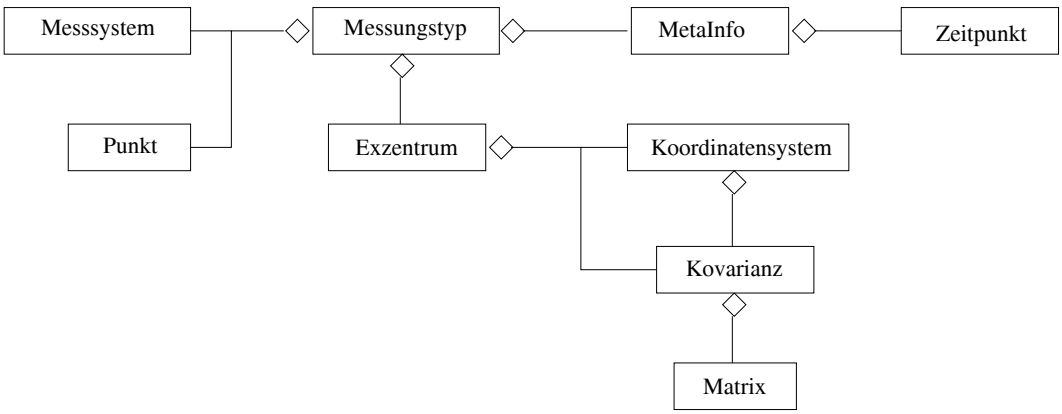

Abbildung 4.3: Die Beziehungen der Klasse *Messungstyp*

Die Klasse *MetaInfo* dient zur Modellierung von Zusatzinformationen, die für die nummerischen Auswertungen nicht von Bedeutung sind, aber bei der Planung und zur Koordinierung von Messungen durchaus von Interesse sein können. Die Klasse *Exzentrum* dient zur Modellierung des Zusammenhangs zwischen den Messbezugspunkten und den vermarkten Punkten. Sie enthält Attribute, mit denen die Abweichung in einem lokalen, horizontalen Koordinatensystem modelliert werden kann. Die Klasse *Messsystem* dient zur Modellierung des

verwendeten Messsystems. Wie schon bei den Daten und den Beobachtungstypen ist auch sie die Wurzel einer Vererbungshierarchie und kapselt Attribute, die alle Messsysteme gemeinsam haben. Die Klassen zur Modellierung der verschiedenen Messgeräte sind Spezialisierungen dieser Klasse. Diese Spezialisierungen sind für diese Arbeit nicht von Bedeutung, deshalb wird hier auch nicht weiter darauf eingegangen.

Zur Modellierung der verschiedenen Beobachtungstypen werden Subklassen der Klasse *Messungstyp* verwendet. In diesen Subklassen können Daten abgelegt werden, die über die bereits modellierten hinausgehen. Durch die Subklassenbildung und die Kapselung der zentralen Daten in der Klasse *Messungstyp* können die Algorithmen zur Ausgleichung geodätischer Netze auf eine übersichtliche Art und Weise dargestellt und implementiert werden. Es werden keine *if-then*-Anweisungsblöcke oder *switch*-Anweisungen im Programmcode benötigt, um konkrete Beobachtungstypen voneinander zu unterscheiden, was in Abschnitt 6.3.2 erläutert wird. Zurzeit existieren Subklassen zur Modellierung der in Tabelle 4.1 dargestellten Beobachtungstypen. Für eine genauere Betrachtung dieser Beobachtungstypen sei auf [53] verwiesen.

Beobachtungstypen
Alignementbezugslinie
Azimut
Bildkoordinaten
Extensometermessung
GPS-Koordinatendifferenz
gleichzeitig gegenseitige Zenitdistanzmessung
Höhenunterschied
Horizontalstrecke
Potentialdifferenz
Richtung
Schlauchwaagenmessung
Schrägstrecke
Schweredifferenz
Strainmetermessung
Streckenverhältnis
terrestrische Koordinatendifferenz
Winkelverhältnis
Zenitdistanzmessung

Tabelle 4.1: Die vorhandenen Beobachtungstypen

4.3 Die Klasse Messungsnetz

Die während einer Epoche aufgenommenen Beobachtungen bilden eine Netztopologie, die durch verschiedene Stufen des Datenflusses verarbeitet wird. Eine solche Netztopologie wird durch die Klasse *Messungsnetz* modelliert. Sie enthält Exemplare der Klassen *Punkt* und *Messungstyp*, die die Beobachtungen zwischen den Punkten modellieren und bietet Methoden zum Zugriff auf die Punkte und Beobachtungen sowie Attribute, mit denen Eigenschaften der Algorithmen zur Netzausgleichung gesteuert werden können. Durch die Aggregation eines Exemplars vom Typ *Epoche* kann einer Netztopologie ein übergeordneter Zeitstempel gegeben werden. Die Abbruchbedingung für den Ausgleichungsalgorithmus kann neben anderen nummerischen Schranken in einem Exemplar des Typs *Netzfehlerschranken* abgelegt werden. Die Klasse bietet ebenfalls die Möglichkeit, stochastische Anschlusspunkte in die Netztopologie mit aufzunehmen, die nicht durch Messungen, sondern nur durch Kovarianzinformationen mit den anderen Punkten verbunden sind.

4.4 Die Klasse Punktnetz

Die Algorithmen zur Netzausgleichung modifizieren die Netztoplogie und haben als zusätzliches Ergebnis eine Menge von Punkten mit den zugehörigen Genauigkeitsinformationen. Diese Punktmenge dient als Eingangsparameter für die geodätische Deformationsanalyse und wird durch die Klasse *Punktnetz* modelliert, die mit ihren Beziehungen in Abbildung 4.4 dargestellt ist.

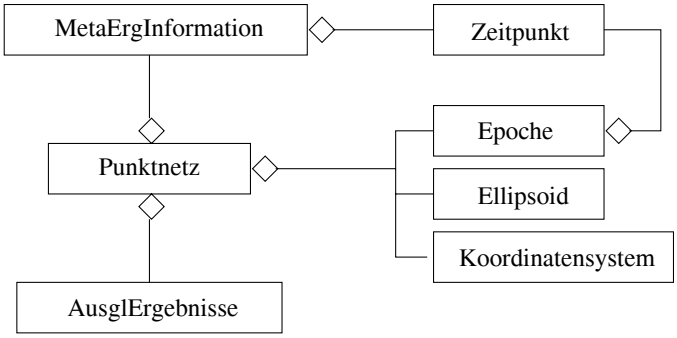

Abbildung 4.4: Die Beziehungen der Klasse *Punktnetz*

Die Klasse *AusglErgebnisse* modelliert die Ergebnisse einer Netzausgleichung. Hierzu ge-hören die Punkte mit ihren Koordinaten, die verbesserten Beobachtungen und die verschiedenen Genauigkeitsinformationen. In dem Exemplar der Klasse *MetaErgInformation* können Informationen über die Ausgleichung, wie z.B. den Zeitpunkt der Ausgleichung oder die ausführende Person abgelegt werden. Auch ein Exemplar der Klasse *Punktnetz* kann durch die Verwendung einer *Epoche* mit einem Zeitstempel versehen werden. Das Ellipsoid, das den enthaltenen Punkten zugrunde liegt sowie das Koordinatensystem der Punkte werden ebenfalls modelliert.

Kapitel 5

Objektpersistenz

In den Abschnitten 2.3.1 und 2.3.2 wurde dargelegt, dass die im Rahmen einer geodätischen Vermessung erhobenen Daten aufgrund des personellen und des logistischen Aufwands einen wirtschaftlichen Wert darstellen. Auf die z.T. langen Zeitspannen zwischen zwei Epochen, in denen Vermessungen durchgeführt werden, wurde in diesem Zusammenhang ebenfalls hingewiesen. Aus den genannten Gründen besteht also die Notwendigkeit, die in den Objekten enthaltenen Daten dauerhaft zu speichern.

5.1 Persistenz

Objekte werden durch Prozesse erschaffen und auch wieder zerstört. Die Lebensdauer eines Objekts im Speicher beginnt i.A. mit der Allokation von Speicherplatz durch den Aufruf des Konstruktors. Die Lebensdauer eines Objekts kann auf unterschiedliche Art und Weise enden:

- durch den Aufruf eines Destruktors, der den vom Objekt belegten Speicherplatz wieder freigibt,
- durch den Einsatz einer Garbage-Collection, wenn keine Referenzen auf das Objekt mehr existieren oder
- durch die Terminierung des erzeugenden Prozesses.

Nach [4, 44] bezeichnet Persistenz die Fähigkeit von Objekten, die Existenz des sie erzeugenden Prozesses zu überdauern. Hieraus folgt, dass solche persistenten Objekte spätestens zu dem Zeitpunkt, an dem der sie erzeugende Prozess endet, in einem nichtflüchtigen Speicher untergebracht werden müssen. Im Gegensatz zu persistenten Objekten stehen transiente Objekte, deren Existenz spätestens zu dem Zeitpunkt, an dem der sie erzeugende Prozess terminiert, endet.

Objektpersistenz kann in unterschiedlichen Ausprägungen auftreten, die in [4] wie folgt angegeben werden.

5.1.1 Persistenz durch Vererbung

Persistenz durch Vererbung setzt voraus, dass alle Objekte, die die Fähigkeit zur Persistenz besitzen sollen, von einer speziellen Basisklasse erben. In dieser Basisklasse sind Mechanismen zur Realisierung der Persistenz enthalten. Die Entscheidung, ob eine Klasse potentiell persistent ist, erfolgt also beim Klassenentwurf. Ein Exemplar der potentiell persistenten Klasse kann dann entweder direkt zum Erzeugungszeitpunkt persistiert werden oder es existiert zuerst als transientes Objekt im Speicher und wird zu einem späteren Zeitpunkt persistiert.

5.1.2 Persistenz zum Erzeugungszeitpunkt

Persistenz zum Erzeugungszeitpunkt verschiebt den Zeitpunkt der Entscheidung über die Persistenz von Objekten vom Klassenentwurf auf die Anwendung der Klassen. Es wird also vom Entwickler entschieden, ob ein neues Exemplar persistent sein soll oder nicht. Dieses Konzept ist flexibler, da die Entscheidung über die Persistenz der Objekte anhand der einzelnen Objekte gefällt werden kann. Ein wesentlicher Nachteil ist jedoch, dass ein nicht persistent angelegtes Objekt zu einem späteren Zeitpunkt nicht mehr persistent gemacht werden kann.

5.1.3 Explizite Persistenz

Das Konzept der expliziten Persistenz sieht vor, dass Objekte durch den Aufruf einer speziellen Methode persistent gemacht werden. Steht hierzu eine inverse Methode zur Verfügung, mit deren Hilfe Objekte wieder transient gemacht werden können, so ist die Objektpersistenz auf flexible Art und Weise verwirklicht. Der Einsatz der expliziten Persistenz erfordert die Implementierung der angesprochenen Methoden in den potentiell persistenten Klassen, was bei Klassenhierarchien mit einer großen Anzahl an Klassen einen nicht unerheblichen Aufwand nach sich ziehen kann.

5.1.4 Persistenz durch Erreichbarkeit

Bei diesem Konzept wird jedes Objekt persistent, das direkt oder indirekt von einem persistenten Objekt aus erreichbar ist. Erreichbarkeit bedeutet hier, dass zwischen zwei Objekten eine Beziehung besteht, die von dem einen zu dem anderen Objekt durchlaufen werden kann. Hierzu wird ein Objektgraf betrachtet, bei dem die Persistenz eine topologische Eigenschaft darstellt. Dies ist die allgemeinste und flexibelste Form der Persistenz.

5.2 Datenbankgestützte Objektpersistenz

5.2.1 Datenhaltung und Datenbanksysteme

Über die Datenhaltung wird eine Wechselwirkung zwischen Anwendern über zeitliche Distanzen hinweg möglich [34]. Diese Wechselwirkung zwischen den Anwendern ist jedoch im Allgemeinen nicht planbar. Die Erzeuger von Daten wissen nichts über die späteren Nutzer und müssen somit auch eine syntaktische Beschreibung der Bedeutung der Daten in die Datenhaltung mit einbringen.

Ein Datenhaltungssystem setzt sich aus den Daten, der sogenannten Datenbasis und einem Datenverwaltungssystem zusammen. Das Datenverwaltungssystem stellt die Dienstfunktionalität, wie z.B. Speichern von Daten, Ändern von Daten oder Wiederauffinden von Daten zur Verfügung. Datenhaltungssysteme werden im Allgemeinen als Datenbanksysteme bezeichnet. Ein Datenbanksystem muss jedoch eine Reihe von Qualitätsmerkmalen erfüllen, um einen sicheren Betrieb zu gewährleisten. Diese Qualitätsmerkmale sind:

- *Angemessenheit*: Sachverhalte müssen sich mit den Dienstfunktionen und Datenbanksprachen auf eine dem Problem angepasste Art und Weise beschreiben lassen. Diese Forderung ist wichtig, um auch die Semantik, die hinter den Daten steht, ausdrücken zu können.
- *Zuverlässigkeit*: Die Datenbank muss die Modelltreue durch Kongruenz oder Konsistenz der Datenbasis und auch die Unverletzlichkeit der Datenbasis durch Persistenz sicherstellen. Hieraus folgt, dass die Datenbank auch die Konfliktfreiheit in Form von geregelter Konkurrenz sicherstellen muss.
- *Technische Leistung*: Die Datenbank muss auf eine Anfrage hin, innerhalb einer angemessenen Antwortzeit, ein Ergebnis liefern. Bei einer größeren Anzahl an Dienstnehmern spielt auch der Datendurchsatz eine wesentliche Rolle.
- *Schutz*: Da Daten möglicherweise langfristig gespeichert werden, müssen sie durch geeignete Maßnahmen vor unbefugtem Zugriff geschützt werden. Dies ist besonders wichtig, da potentielle Angreifer oft genügend Zeit haben, einen Angriff auf die Daten durchzuführen. Die Palette der möglichen Angriffe reicht vom Ausspionieren der Daten über das Verändern der Daten bis hin zur gezielten Vernichtung von Daten.

Einen umfassenden Überblick über Datenbanken, Anfragesprachen und Implementierung von Datenhaltungssystemen und Datenverwaltungssystemen liefern [31, 32, 34, 36].

5.2.2 Die Abbildung von Objekten auf eine relationale Datenbank

5.2.2.1 Vergleich des objektorientierten und des relationalen Ansatzes

Im objektorientierten Ansatz werden Objekte betrachtet, die aus Daten und einem spezifizierten Verhalten bestehen. Zwischen Objekten können verschiedene Beziehungen, wie z.B. die in Abschnitt 3.1.7 beschriebene Assoziation, die in Abschnitt 3.1.8 beschriebene Aggregation oder die in Abschnitt 3.1.5 beschriebene Generalisierung, bestehen. Objekte sind, wie in Abschnitt 3.1.2 beschrieben, über ihre Identität eindeutig identifizierbar. Neben dieser Identität können auch die Zustände von Objekten zu einem Zeitpunkt verglichen werden. Zwei Exemplare einer Klasse können somit zu einem Zeitpunkt gleiche Zustände haben ohne identisch zu sein.

Beim relationalen Ansatz [34] wird die Speicherung von Daten in einer Tabellenstruktur betrachtet. Eine Datenzeile wird durch eine Menge von Attributen, die Primärschlüssel genannt wird, eindeutig identifiziert. Die Spalten einer Tabelle können Werte enthalten, die in anderen Tabellen Schlüssel sind. Werden sie zur Referenz auf andere Tabellen verwendet, spricht man von einem Fremdschlüssel. Um die referentielle Integrität der Daten

sicherzustellen, muss für jeden Fremdschlüssel der Datensatz existieren, der referenziert wird. Relationale Datenbankmanagementsysteme bieten durch die Anfragesprache SQL ein Werkzeug zur Organisation und Abfrage der in den Tabellen enthaltenen Daten.

Vergleicht man die beiden Ansätze miteinander, so fällt auf, dass die interne Repräsentation der Daten auf unterschiedliche Arten gehandhabt wird, zum Zugriff auf die Daten jedoch Schnittstellen vorhanden sind. Im objektorientierten Fall sind dies die Schnittstellen der Klassen und im Fall des relationalen Ansatzes die Anfragesprache SQL. Zur Speicherung von Objekten in einer relationalen Datenbank müssen also Transformationen durchgeführt werden. Für diese Transformationen werden Abbildungsvorschriften für die Daten der Objekte sowie auch für die in Abschnitt 3.1 vorgestellten objektorientierten Konzepte benötigt. Solche Abbildungsvorschriften werden in den folgenden Abschnitten beschrieben.

5.2.2.2 Eindeutigkeit der Objekte

Wie bereits in Abschnitt 5.2.2.1 erwähnt, können Objekte über ihre Identität identifiziert werden. In relationalen Datenbanken wird zur eindeutigen Identifizierung von Datensätzen das Konzept des Primärschlüssels benutzt.

Startet man einen Prozess mehrere Male hintereinander, so haben die Objekte, die er erzeugt, nicht notwendigerweise immer dieselbe Identität. Wird die vom System erzeugte Objektidentität als Primärschlüssel verwendet, so kann es passieren, dass ein aus der Datenbank geladenes Objekt vom System eine andere Identität zugewiesen bekommt. In einem solchem Fall müssten dann alle Beziehungen zwischen den Identitäten der geladenen Objekte und den Primärschlüsseln in der Datenbank geprüft werden, um die referenzielle Integrität der Datenbasis sicherzustellen.

Da die Verwendung der systemerzeugten Objektidentifikatoren also gewisse Schwierigkeiten mit sich bringt, bietet sich als Alternative die Verwendung künstlich erzeugter Objektidentifikatoren an. Sie werden zusammen mit dem Objektzustand in der Datenbank abgelegt und dienen hier als Primärschlüssel. Die Objektidentifikatoren werden bei der Persistierung der Objekte erzeugt und sie werden durch die, die Objekte verwendenden Programme nicht modifiziert. Als Objektidentifikatoren können z.B. Zahlen, Zeichenketten oder Kombinationen aus beidem verwendet werden. Hierbei ist die Eindeutigkeit der Objektidentifikatoren jedoch wichtig, da sie die Objektidentität sicherstellen sollen.

5.2.2.3 Die Abbildung von Klassen auf relationale Tabellen

Der Zustand eines Objekts zu einem bestimmten Zeitpunkt wird durch die Belegung der Attribute zu diesem Zeitpunkt bestimmt. Um den Zustand eines Objekts zu persistieren, muss also die Belegung der Attribute in der Datenbank hinterlegt werden.

Hierzu bietet es sich an, zu den Klassen Tabellen anzulegen, in denen Spalten für die Attribute der Klassen vorgesehen sind. Für die Basistypen nahezu aller objektorientierten Programmiersprachen gibt es Entsprechungen in den Daten-Definitionssprachen von Datenbankmanagementsystemen. Attribute, die Referenzen auf andere Objekte enthalten, können durch die in den Abschnitten 5.2.2.4 und 5.2.2.5 vorgestellten Konzepte behandelt werden. Auf die Speicherung von ererbten Attributen wird in Abschnitt 5.2.2.6 eingegangen.

Mengenwertige Attribute oder Aufzählungstypen existieren in Java nicht in der Form wie z.B. in C++. Um diese nachzubilden werden statische Klassenvariablen benutzt.

5.2.2.4 Die Abbildung der Aggregation auf eine relationale Tabellenstruktur

Zur Abbildung der Aggregation auf eine relationale Tabellenstruktur kann eine Fremdschlüssel-Beziehung genutzt werden. Hierbei wird der Objektidentifikator des aggregierten Objekts in die zum entsprechenden Attribut gehörende Spalte des aggregierenden Objekts eingetragen. Zum Laden eines Objekts kann die Ermittlung der Attributbelegung des aggregierten Objekts dann entweder durch ein Tabellen-Join oder durch jeweils eine Datenbankabfrage pro aggregiertem Objekt erfolgen.

5.2.2.5 Die Abbildung von Kollektionen auf eine relationale Tabellenstruktur

Zur Abbildung von Assoziationen, wie sie bei der Verwendung von Kollektionen oder assoziativen Datenstrukturen vorkommen, auf eine relationale Tabellenstruktur werden für die verschiedenen Kollektionstypen zusätzliche Tabellen benutzt. Diese Tabellen nehmen die Objektidentifikatoren der enthaltenen Objekte und je nach Kollektionstyp auch den Index des zugehörigen Objekts auf. Durch diese Art der Abbildung wird die, aus der Sicht der Kollektion bestehende 1:n-Beziehung, abgebildet. Eventuell vorkommende n:m-Beziehungen werden auf eigene Tabellen abgebildet. Diese enthalten die Objektidentifikatoren der an der jeweiligen Beziehung beteiligten Objekte.

5.2.2.6 Die Abbildung der Vererbung auf eine relationale Tabellenstruktur

Bei der Abbildung der Vererbung auf eine relationale Tabellenstruktur bestehen mehrere Möglichkeiten, um die von den Oberklassen geerbten Attribute auf die Tabellenstruktur abzubilden:

- **Ansatz 1: Eine Tabelle für alle Subklassen einer Klasse:** Hierbei wird eine Klasse mit all ihren Subklassen auf eine einzige Tabelle abgebildet. Alle vorkommenden Attribute werden auf jeweils eine Tabellenspalte abgebildet. Der Zugriff auf die in der Tabelle gespeicherten Objektzustände geschieht für alle Subklassen nahezu gleich und mit relativ hoher Geschwindigkeit, da nur auf eine Tabelle zugegriffen werden muss. Werden neue Attribute in die Klassenhierarchie eingefügt, so müssen alle auf die Tabelle zugreifenden Objekte modifiziert werden. Dieser Ansatz bringt Nachteile beim Speicherbedarf mit sich, da die Spalten zum großen Teil mit NULL-Werten gefüllt werden müssen.

- **Ansatz 2: Eine Tabelle für alle Attribute einer Klasse:** Hierbei werden die Attribute der Klasse und alle Attribute der jeweiligen Oberklassen auf die Spalten der Tabelle abgebildet. Alle Attribute der Klasse sind also nur in einer Tabelle abgebildet und es gibt für jede Klasse eine eigene Tabelle. Bei der Speicherung eines Objektzustandes wird, wie in Ansatz 1, nur auf eine Tabelle zugegriffen. Hierbei handelt es sich jedoch - im Gegensatz zu Ansatz 1 - je nach Klasse, um unterschiedliche Tabellen. Dieser Ansatz bringt den gleichen Geschwindigkeitsvorteil mit sich wie Ansatz 1. Das Einfügen neuer Attribute in die Klassenhierarchie ist hierbei jedoch besonders aufwendig, da nicht nur die zur entsprechenden Klasse gehörende Tabelle, sondern auch die zu den Subklassen gehörenden Tabellen und die auf sie zugreifenden Objekte modifiziert werden müssen. Hinsichtlich des Speicherbedarfs ist auch dieser Ansatz nicht optimal, da für vererbte Attribute mehrere Spalten in unterschiedlichen Tabellen existieren.

- **Ansatz 3: Eine Tabelle für die in einer Klasse definierten Attribute:** Hierbei werden nur die in der Klasse definierten Attribute auf die zur jeweiligen Klasse gehörende Tabelle abgebildet. Die von den Oberklassen geerbten Attribute sind in den zu den Oberklassen gehörenden Tabellen abgelegt. Um die Beziehung zwischen den in einer Tabelle gespeicherten Attributwerten und den restlichen, in den zu den Oberklassen gehörenden Tabellen gespeicherten Attributwerten herzustellen, werden die Objektidentifikatoren der Oberklassen benutzt. Tabellen zur Aufnahme von Klassen, die von Oberklassen abgeleitet sind, enthalten eine zusätzliche Spalte zur Aufnahme dieses Objektidentifikators. Zum Zugriff auf den Objektzustand müssen die entsprechenden Tabellen miteinander in Beziehung gesetzt werden, was eine Geschwindigkeitseinbuße mit sich bringt. Beim Hinzufügen eines neuen Attributs zur Klassenhierarchie muss jedoch, im Gegensatz zu den anderen Ansätzen, nur eine Tabelle geändert werden. Dies kann sich als Erleichterung erweisen, wenn es im Zuge der Umsetzung eines Entwurfs zu Änderungen an einzelnen Klassen kommt. Im Hinblick auf den Speicherbedarf ist dieser Ansatz besser als die vorherigen Ansätze, da es keine unbenutzten oder doppelten Tabellenspalten gibt.

Vergleicht man die drei Ansätze, so fällt auf, dass eine einfache Implementierbarkeit - wie in Ansatz 1 - zwar mit einer hohen Geschwindigkeit einhergeht, aber mehr Aufwand bei der Erweiterung nach sich zieht. Eine leichte Erweiterbarkeit - wie in Ansatz 3 - erfordert jedoch einen höheren Implementierungsaufwand und geht wiederum zu Lasten der Geschwindigkeit aufgrund der größeren Zahl von Tabellen, die miteinander in Beziehung gesetzt werden müssen. Trotz des höheren Implementierungsaufwands und der Geschwindigkeitseinbußen erscheint Ansatz 3 für dieses Projekt geeignet. Aufgrund der Tatsache, dass im Zuge einer Vermessung eine nicht unerhebliche Datenmenge anfällt, sollte keine redundante Speicherung der Daten durchgeführt werden.

5.3 Im- und Export von Daten durch Verwendung der Extensible Markup Language

Bei der Extensible Markup Language (XML) handelt es sich um eine Metasprache, die zur Definition von anderen Sprachen oder Dateistrukturen verwendet werden kann. Hierdurch ist es möglich, Sprachen oder Dateistrukturen zu erschaffen, die auf bestimmte Problemstellungen zugeschnitten sind.

XML-Dateien haben einen ähnlichen Aufbau wie HTML-Dateien, wie sie zur Beschreibung von Internetseiten verwendet werden. Im Hinblick auf den Anwendungsbezug ergeben sich jedoch gravierende Unterschiede zwischen beiden Sprachen:

- Durch die Verwendung von XML können Daten und eine syntaktische Beschreibung ihrer Bedeutung in einem Dokument spezifiziert werden. Dies geht über die Layoutdefinition und die Formatierungsanweisungen von HTML-Dateien hinaus.
- Bei der Verwendung der XML können eigene Strukturen und Schlüsselwörter durch den Entwickler entworfen werden, um Daten bzw. ihre Bedeutung darzustellen. HTML hingegen besitzt einen festen Satz an Schlüsselwörtern und Strukturen.
- Da HTML zur Beschreibung eines Seitenlayouts entworfen wurde, bietet es immer die gleiche Sicht auf die Daten. Bei der Verwendung von XML stehen jedoch die Daten im Vordergrund, wodurch es möglich ist, an die jeweilige Anwendung angepasste Sichten auf die Daten zu erzeugen.

Einen Überblick über die XML, den Aufbau von XML-Dateien sowie die vielseitigen Anwendungsmöglichkeiten gibt [37].

Da mittlerweile kommerzielle sowie frei verfügbare Programmpakete zur Verarbeitung von XML-Daten für nahezu alle gängigen Programmiersprachen verfügbar sind, bietet sich die Verwendung der XML auch beim Datenaustausch zwischen verschiedenen Programmen an. Durch die Verwendung von Textdateien zur Speicherung der XML-Daten wird neben der Unabhängigkeit von bestimmten Programmiersprachen und Herstellern auch die Plattformunabhängigkeit bezüglich des verwendeten Betriebssystems sichergestellt. Aufgrund der Möglichkeit, Dateistrukturen zu erschaffen, die auf unterschiedlichen Betriebssystemplattformen und mit Hilfe von Parsern unter Zuhilfenahme verschiedener Programmiersprachen eingelesen werden können, eignet sich die XML auch zum Im- und Export von Daten.

5.3.1 Die Abbildung von Objekten auf eine XML-Datei

Bei der Persistierung von Objekten mit Hilfe der XML müssen deren Objektidentitäten und Zustände in einer XML-Datei abgelegt werden. Um die in Abschnitt 3.1 beschriebenen objektorientierten Konzepte wie Vererbung, Aggregation oder Assoziation bei Kollektionen zu verarbeiten, werden geeignete Abbildungen für diese Konzepte benötigt. Diese Abbildungen werden in den folgenden Abschnitten beschrieben.

5.3.1.1 Objektidentität

Auch bei der Speicherung von Objekten in einer XML-Datei werden Objektidentifikatoren benötigt, um z.B. Aggregationen oder Assoziationen abzubilden. Anders als bei der datenbankgestützten Form der Persistierung müssen diese jedoch nur innerhalb der XML-Datei eindeutig sein. Hierzu können wieder künstlich generierte Objektidentifikatoren dienen, die zusätzliche Metainformationen enthalten. Der Name der Datei, in die der Objektgraf persistiert werden soll, wird als Objektidentifikator im Wurzelobjekt des Objektgrafen abgelegt.

5.3.1.2 Die Abbildung von Klassen

Zur Speicherung des Objektzustandes muss die Attributbelegung zum Zeitpunkt der Speicherung in der XML-Datei abgelegt werden. Zur Speicherung des Klassentyps werden Tags verwendet, die den Namen der Klasse enthalten. Diese rahmen die Tags zur Speicherung des Objektidentifikators und der Attribute ein. Die Attributwerte werden durch Tags umrahmt, die die jeweilige Attributbezeichnung enthalten. Hierbei werden alle Attribute, also auch die von Oberklassen geerbten, gespeichert.

Dieser Sachverhalt soll durch den nachfolgenden Ausschnitt aus einer XML-Datei verdeutlicht werden.

```
<Klassenname>
  <OID>Objektidentifikator< \OID>
  <Attributname_1 >Attributwert_1 < \Attributname_1 >
  ...
  <Attributname_n >Attributwert_n < \Attributname_n >
< \Klassenname>
```

Abbildung 5.1: Abbildung der Objektidentität und der Attribute einer Klasse auf eine XML-Datei

5.3.1.3 Die Abbildung der Aggregation

Bei Attributen, die Referenzen auf andere Objekte enthalten, wird anstelle eines Attributwertes der Objektidentifikator des referenzierten Objekts abgelegt. Auf diese Weise muss jedes Objekt nur einmal in der XML-Datei hinterlegt werden und eine Vervielfältigung von Objekten aufgrund von mehrfacher Speicherung innerhalb einer Datei wird ausgeschlossen.

Dieser Sachverhalt soll durch den nachfolgenden Ausschnitt aus einer XML-Datei verdeutlicht werden.

```
<Klassenname>
  <OID>Objektidentifikator_1 < \OID>
  <Attributname_1 >Objektidentifikator_2 < \Attributname_1 >
< \Klassenname>

<Klassenname>
  <OID>Objektidentifikator_2 < \OID>
  <Attributname_1 >Attributwert_1 < \Attributname_1 >
  ...
  <Attributname_n >Attributwert_n < \Attributname_n >
< \Klassenname>
```

Abbildung 5.2: Abbildung der Aggregation auf eine XML-Datei

5.3.1.4 Die Abbildung der Assoziation bei Kollektionen

Die in 5.3.1.3 beschriebene Verfahrensweise kann auch bei Kollektionen und assoziativen Datentypen angewandt werden. Hier werden die Objektidentifikatoren der enthaltenen Objekte durch spezielle Tags umrahmt, die sie als Elemente der Kollektion kennzeichnen, was in Abbildung 5.3 dargestellt ist.

```
<Klassenname>
  <OID>Objektidentifikator_1 < \OID>
  <Element>Objektidentifikator_2 < \Element>
  ...
  <Element>Objektidentifikator_l < \Element>
< \Klassenname>

<Klassenname>
  <OID>Objektidentifikator_2 < \OID>
  <Attributname_1 >Attributwert_1 < \Attributname_1 >
  ...
  <Attributname_n >Attributwert_n < \Attributname_n >
< \Klassenname>

  ...

<Klassenname>
  <OID>Objektidentifikator_l < \OID>
  <Attributname_1 >Attributwert_1 < \Attributname_1 >
  ...
  <Attributname_m >Attributwert_m < \Attributname_m >
< \Klassenname>
```

Abbildung 5.3: Abbildung der Assoziation bei Kollektionen

Kapitel 6

Umsetzung der Anforderungen in ein Informationssystem

Dieses Kapitel beschreibt die Umsetzung der in Abschnitt 2.3.1 aufgestellten Zielsetzungen und Anforderungen sowie die Unterteilung des Systems in seine Komponenten. In einem ersten Abschnitt wird die Systemarchitektur erläutert und es wird auf die Zuständigkeiten der Systemkomponenten eingegangen. Hieran schließen sich weitere Abschnitte an, in denen die Umsetzungen einzelner Systemkomponenten beschrieben werden.

6.1 Systemarchitektur

In Abschnitt 2.3.1 wurde die Integration des Datenflusses einer geodätischen Deformationsanalyse in einem Programmsystem als vorrangiges Ziel dieser Arbeit genannt. Dieser Datenfluss wurde in Abschnitt 2.1 aus geodätischer Sicht beschrieben und soll mit Hilfe der in Kapitel 3 beschriebenen Objektorientierung in einem Informationssystem umgesetzt werden. Die einzelnen Stufen des Datenflusses entsprechen fachlichen Einheiten in der Geodäsie, weshalb ihre Umsetzung in Komponenten Sinn macht. Eine weitere Zergliederung des Datenflusses würde zu einer zu großen Anzahl an Komponenten führen und die Vorgabe aus der Geodäsie damit nicht mehr exakt widerspiegeln. Durch die Hintereinanderausführung der Komponenten kann der Datenfluss durchgeführt werden. Die Realisierung des Datenflusses durch eine Kette von Komponenten bietet folgende Vorteile:

- Die aus der Geodäsie bekannte Struktur des Datenflusses kann in dem Programmsystem nachgebildet werden.
- Die Komponenten zur Realisierung der einzelnen Stufen des Datenflusses können auf verschiedene Arten kombiniert werden, um verschiedene Analysen zu ermöglichen.
- Die einzelnen Komponenten können unabhängig voneinander entwickelt und konfiguriert werden, was die Wartbarkeit des Systems erhöht und die Erweiterbarkeit sicherstellt.
- Neue Berechnungs- und Analyseverfahren können in eigenen Komponenten realisiert und somit leicht in das System integriert werden.

Die einzelnen Komponenten bieten ihre Funktionalität über Schnittstellen an, über die sie mit anderen Komponenten interagieren. Durch die Verwendung von Schnittstellen wird die Kopplung zwischen den einzelnen Komponenten vermindert, was die Flexibilität des Systems erhöht. Hinter den Schnittstellen werden zum Teil komplexe fachliche Arbeitsabläufe verborgen, die durch die bereitgestellte Methodik aufgerufen werden können. Die Komponenten enthalten hauptsächlich anwendungsbezogenen Code, der die fachlichen Aspekte der jeweiligen Stufe des Datenflusses umsetzt. Nichtfachliche Aspekte, wie z.B. die Datenspeicherung, werden in anderen Teilen des Systems umgesetzt. Bei der Umsetzung des Datenflusses wird die Stufe der Datenerfassung ausgelassen, da diese in der Regel nicht vollautomatisch durchgeführt wird. Obwohl im Datenfluss, wie er in Abb. 2.1 dargestellt ist, die Stufe der Visualisierung direkt auf die Deformationsanalyse folgt, ist es sinnvoll, wenn die entsprechende Komponente auch die Ergebnisse vorangegangener Stufen darstellen kann.

Da die Komponenten ihre Funktionalität durch eine Schnittstelle bereitstellen, kann ihre Integration in das System durch das in Abschnitt 3.3.1 vorgestellte Entwurfsmuster *Strategie* erfolgen. Wie bereits erläutert, realisieren die einzelnen Komponenten Berechnungsverfahren, die im Verlaufe der geodätischen Deformationsanalyse durchgeführt werden, wobei sie den Zugriff auf diese hinter einer Schnittstelle verbergen. Die Komponenten stellen somit die *KonkreteStrategie*-Objekte aus Abb. 3.24 dar, deren Funktionalität über die bereitgestellte Schnittstelle von den *Kontext*-Objekten genutzt werden kann. Dieses Vorgehen führt dazu, dass zu jeder Stufe des Datenflusses verschiedene Komponenten existieren können, die den gleichen fachlichen Aspekt auf unterschiedliche Art und Weise realisieren. Dies wird z.B. an der Stufe der Deformationsanalyse deutlich, wo verschiedene Ansätze zur Deformationsanalyse in separaten Komponenten realisiert werden können, die jedoch alle dieselbe Schnittstelle realisieren.

Nachdem nun die Umsetzung des Datenflusses erläutert wurde, soll dieser noch einmal aus der Sicht der Datenverarbeitung betrachtet werden. Nach der Durchführung der Messungen bilden die Beobachtungen und die aufgenommenen Daten eine Netztopologie, die als Eingangsparameter für die Vorverarbeitung dient. Diese Netztopologie wird durch die in Abschnitt 4.3 beschriebene Klasse *Messungsnetz* modelliert. Die Algorithmen zur Vorverarbeitung operieren auf der Netztopologie und bringen diese in einen Zustand, in dem die Netztopologie als Eingangsparameter für eine anschließende Netzausgleichung dienen kann. Auch die Algorithmen zur Netzausgleichung operieren auf einer Netztopologie. Als Ergebnis der Netzausgleichung enthält die Netztopologie Zuschläge, die an die Koordinaten der Punkte anzubringen sind sowie Verbesserungen, die an den Beobachtungen anzubringen sind. Weiterhin liefern die Algorithmen zur Netzausgleichung zu jeder Netztopologie je eine Menge an ausgeglichenen Punkten mit den zugehörigen Genauigkeitsinformationen, die durch die in Abschnitt 4.4 beschriebene Klasse *Punktnetz* modelliert wird. Anders als die Algorithmen zur Vorverarbeitung, die als Ergebnis nur eine modifizierte Netztopologie liefern, haben die Algorithmen zur Netzausgleichung sowohl eine modifizierte Netztopologie wie auch eine Punktmenge mit Genauigkeitsinformationen als Ergebnis, die als Eingangsparameter für die Algorithmen zur Deformationsanalyse dient. Die Algorithmen zur Deformationsanalyse operieren - je nach Typ - auf zwei oder mehreren Punktmengen, die durch die Klasse *Punktnetz* modelliert werden, um signifikante Verschiebungen der Punkte aufzudecken. Die als Parameter übergebene Punktmenge wird durch die Algorithmik zur Deformationsanalyse nicht modifiziert. Die Visualisierung dient zur grafischen Darstellung der Ergebnisse der vorangegangenen Stufen, um die Interpretation des Zahlenmaterials zu unterstützen. Die Komponente zur Visualisierung soll in der Lage sein, die Ergebnisse aller vorangegangenen Teilschritte als Eingangsdaten zu akzeptieren.

Nach der Betrachtung des Datenflusses aus Sicht der Datenverarbeitung wird deutlich, dass die verschiedenen Komponenten auf den in [53] beschriebenen Klassen zur Modellierung der geodätischen Entitäten operieren. Hierdurch wird also eine Trennung zwischen den Klassen zur Datenmodellierung und den Klassen zur Datenverarbeitung vollzogen. In Abschnitt 2.3.2 wurde ausgeführt, dass zur Langzeitdatenspeicherung ein relationales Datenbanksystem eingesetzt werden soll und in Kapitel 5 wurde beschrieben wie Objekte und die zugehörigen Konzepte auf eine relationale Tabellenstruktur transformiert werden können. Aufgrund der beschriebenen Trennung der Klassen zur Datenmodellierung und der Klassen zur Datenverarbeitung müssen also nur erstere in der Datenbank gespeichert werden. Bei der Umsetzung der Objektpersistenz bietet sich neben der Implementierung entsprechender Methoden in den datenhaltenden Klassen auch noch der Einsatz einer Schichtenarchitektur an, was folgende Vorteile bietet:

- In den Klassen zur Datenmodellierung muss kein Code zur Interaktion mit dem verwendeten Langzeitdatenspeicher vorhanden sein. Diese Klassen können somit rein fachliche Aspekte modellieren.
- Der Persistenzmechanismus bietet den Klienten eine Schnittstelle zur Interaktion an. Die Transformation der Objekte geschieht für diese völlig transparent.
- Bei Änderungen am zugrundeliegenden Langzeitdatenspeicher muss nur die Schicht, nicht aber die Klienten oder die Klassen zur Datenmodellierung angepasst werden. Dies erhöht die Wartbarkeit der gesamten Anwendung.

Aufgrund dieser Vorteile wird der Persistenzmechanismus durch eine Schichtenarchitektur realisiert.

In Abschnitt 2.3.1 wurde der Datenexport ebenfalls als Zielsetzung genannt. Dieser soll durch die Verwendung der Extensible Markup Language realisiert werden, weshalb in Abschnitt 5.3 Abbildungsvorschriften angegeben wurden, um die Objekte und die zugehörigen Konzepte auf den Inhalt einer XML-Datei abzubilden. Im Rahmen des Datenexports ist man im Allgemeinen am Export aller in den Objekten enthaltenen Daten interessiert. Da durch die Schnittstelle des Persistenzmechanismus bereits die notwendige Funktionalität definiert ist, bietet es sich an diese ebenfalls zum Datenexport zu verwenden. Dies bietet den Vorteil, dass Programmcode und Verfahrensweisen, die zur Realisierung der Objektpersistenz mit einer relationalen Datenbank verwendet werden, wiederverwendet werden können.

Auch bei der Realisierung der Benutzerschnittstelle bietet sich die Schichtenarchitektur an. Die Benutzerschnittstelle umfasst die Klassen zur Ansteuerung der Komponenten und zur Eingabe der Daten in Dialogfenstern. Die Visualisierungskomponente dient zwar auch der Präsentation der Daten am Bildschirm, sie soll jedoch nicht

der Benutzerschnittstelle zugerechnet werden, da sich ihre Aufgaben lediglich auf die Präsentation der Daten und nicht auf die Interaktion mit dem Benutzer beziehen. Durch die Verwendung einer Schichtenarchitektur zur Umsetzung der Benutzerschnittstelle ist es möglich, die Sicht auf die Daten von den Klassen zu deren Modellierung zu entkoppeln, wodurch sich folgende Vorteile gegenüber einer Implementierung der Funktionalität zur Präsentation der Daten in den Klassen zur Datenmodellierung ergeben:

- In den Klassen zur Datenmodellierung muss kein Code zur Präsentation der Daten vorhanden sein. Diese Klassen können somit rein fachliche Aspekte modellieren.
- Das Datenmodell wird von der Sicht auf die Daten entkoppelt. Zu jeder Klasse, die zur Modellierung von Daten dient, können daher mehrere Klassen zur Anzeige der Daten vorhanden sein.
- Neue Sichten auf die Daten können dem System hinzugefügt werden, ohne die Klassen zur Modellierung der Daten modifizieren zu müssen.
- Die Entkopplung der Daten von der Sicht auf die Daten vereinfacht die Wartung der Klassen der Benutzerschnittstelle.

Da die Benutzerschnittstelle die Interaktion mit dem Benutzer übernimmt, hat sie sowohl Zugriff auf die Schnittstellen der Komponenten als auch auf die Klassen zur Modellierung der geodätischen Entitäten und die Funktionalität des Persistenzmechanismus.

In Abbildung 6.1 ist die Umsetzung der in Abschnitt 2.1 vorgestellten Stufen des Datenflusses der geodätischen Deformationsanalyse in einzelne Komponenten zu sehen. Der Sachverhalt, dass die einzelnen Stufen durch verschiedene Komponenten mit unterschiedlichen Implementierungen realisiert werden können, soll durch die übereinanderliegenden Kästchen angedeutet werden. Alle am Datenfluss beteiligten Komponenten operieren auf den Klassen zur Modellierung der geodätischen Entitäten, um ihre Funktionalität erbringen zu können. Die Schichtenarchitektur des Persistenzmechnismus ist im unteren Teil der Abbildung zu erkennen. Hier wird deutlich, dass der Zugriff auf den Langzeitdatenspeicher vor den Komponenten des Datenflusses, dem Klassenmodell zur Datenhaltung und der Benutzerschnittstelle verborgen wird. Die Benutzerschnittstelle ist ebenfalls durch eine Schichtenarchitekur realisiert, wodurch die Sicht auf die Daten vom eigentlichen Datenmodell entkoppelt werden kann. Da die Benutzerschnittstelle zur Ansteuerung der einzelnen Komponenten und zur Speicherung der Daten genutzt wird, hat sie Zugriff auf die einzelnen Komponenten des Datenflusses, auf das Klassenmodell und auf die Funktionalität des Persistenzmechanismus.

Die beschriebene Systemarchitektur trennt die technischen Aspekte wie die Langzeitdatenspeicherung oder die Benutzerschnittstelle von den fachlichen Aspekten wie der Modellierung der geodätischen Entitäten und dem Datenfluss der Deformationsanalyse, woraus sich zusätzlich zu den bereits genannten noch weitere Vorteile ergeben:

- Bei der Realisierung neuer Berechnungsverfahren auf den verschiedenen Stufen des Datenflusses brauchen nur fachliche Aspekte betrachtet und modelliert werden. Technische Aspekte wie z.B. die Langzeitdatenspeicherung müssen nicht betrachtet werden.
- Neue Berechnungsverfahren können durch die Realisierung der entsprechenden Schnittstelle und des Entwurfsmusters *Strategie* auf einfache Art und Weise in das System integriert werden.
- Durch verschiedene Kombinationen der Komponenten des Datenflusses können verschiedene Analysen auf den gleichen Ausgangsdaten durchgeführt werden.
- Die Verarbeitung der Daten kann nach jeder Stufe des Datenflusses eingestellt werden. Das System kann also auch nur zur Ausgleichung geodätischer Netze dienen, wenn keine Deformationsanalyse benötigt wird.
- Durch die Realisierung verschiedener Visualisierungskomponenten können unterschiedliche Sichten auf die Ergebnisse der verschiedenen Komponenten angeboten werden.
- Durch die Langzeitdatenspeicherung stehen die Daten vergangener Epochen auch für spätere Auswertungen zur Verfügung.

- Der Langzeitdatenspeicher kann ohne Auswirkungen auf die fachlichen Aspekte des Systems gewechselt werden.
- Die Benutzerschnittstelle kann ohne Auswirkungen auf die Fähigkeit zur Langzeitdatenspeicherung oder die Realisierung des Datenflusses gewechselt werden.

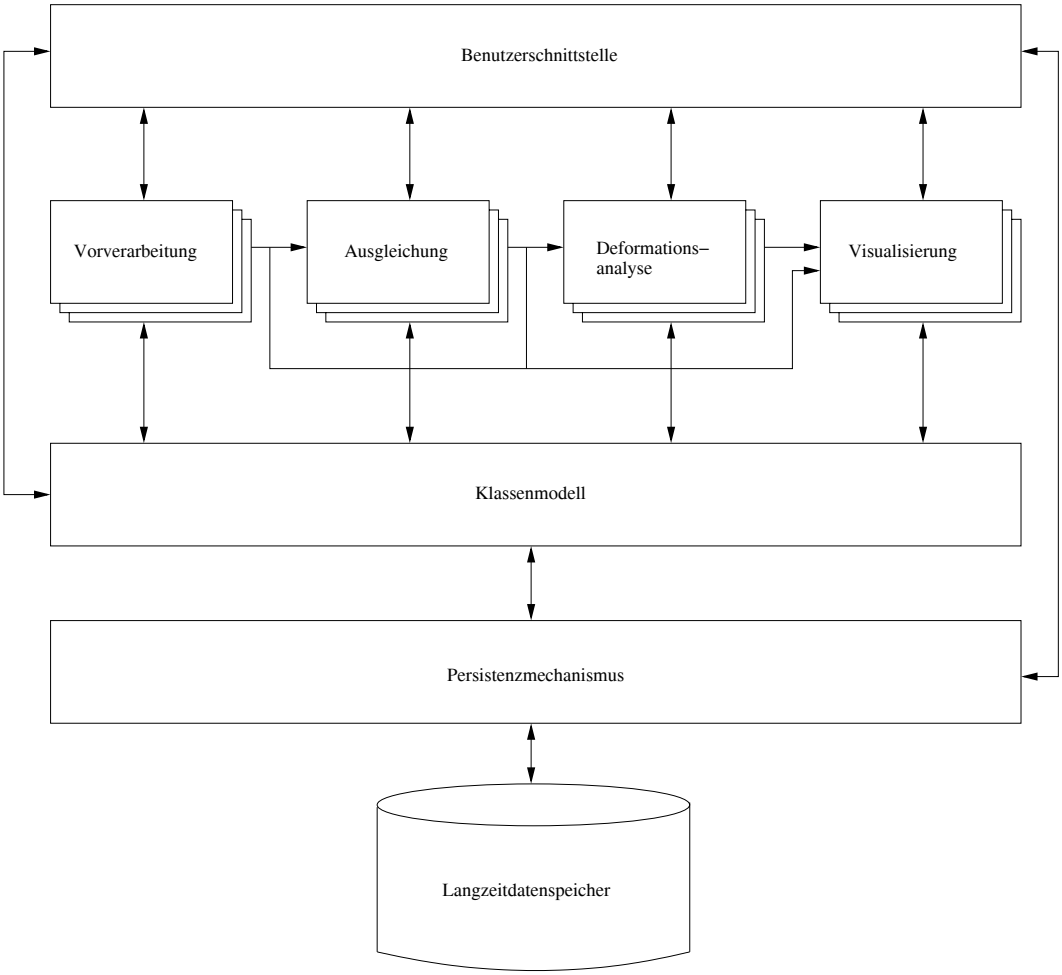

Abbildung 6.1: Architektur des Informationssystems zur Deformationsanalyse

Im Rahmen dieser Arbeit wurden die Komponenten zur Netzausgleichung und zur Deformationsanalyse sowie der Persistenzmechanismus und eine Benutzerschnittstelle realisiert.

Die Benutzerschnittstelle dient als Steuerzentrale des Gesamtsystems. Sie ist mit den in Abb. 6.1 dargestellten Systemkomponenten assoziiert und ruft deren Funktionalität entsprechend den Benutzereingaben auf. Sie bietet keine grafische Repräsentation der Daten, wie z.B. der bearbeiteten Netztopologie, und dient lediglich als Rahmen zur Darstellung der zur Verfügung stehenden Komponenten des Datenflusses, der das Aufrufen der durch die Komponenten bereitgestellten Funktionalität unterstützt. Die Umsetzung der Benutzerschnittstelle wird in dieser Arbeit nicht beschrieben.

Die Komponenten zur Netzausgleichung kapseln die Algorithmik zur Ausgleichung von ein- und zweidimensionalen geodätischen Netzen. Die Umsetzung dieser Algorithmik in die entsprechenden Komponenten und deren Schnittstelle zur Integration in das System wird in Abschnitt 6.3 beschrieben. In diesem Zusammenhang wird auch erläutert, wie die Erweiterbarkeit des Systems hinsichtlich neuer funktionaler Modelle und neuer Beobachtungstypen realisiert ist.

Die Komponenten zur Deformationsanalyse kapseln verschiedene Verfahren zur geodätischen Deformationsanalyse. Die Umsetzung der Verfahren in die entsprechenden Komponenten und deren Schnittstelle zur Integration in das System wird in Abschnitt 6.4 beschrieben.

Die Klassen zur Datenmodellierung, die funktionalen Modelle zur Netzausgleichung und die geodätischen Grundlagen der Deformationsanalyse werden in [53] beschrieben. Die Realisierung der Verfahren zur Vorverarbeitung der Daten ist ein bisher noch offener Punkt und daher nicht Bestandteil der Arbeit. Die Visualisierungskomponente ist ebenfalls nicht Bestandteil dieser Arbeit, da die Visualisierung von Daten im Kontext der Geodäsie eine komplexe Aufgabenstellung ist, die separat behandelt werden sollte.

Der folgende Abschnitt befasst sich mit der Realisierung des Persistenzmechanismus und mit der Umsetzung der in Kapitel 5 beschriebenen Transformationen.

6.2 Der Persistenzmechanismus

6.2.1 Allgemeine Anmerkungen

Auf die Notwendigkeit der Langzeitspeicherung der Daten wurde bereits in den Abschnitten 2.3.1 und 2.3.2 eingegangen. In Kapitel 5 wurden Abbildungsvorschriften für die in Abschnitt 3.1 vorgestellten objektorientierten Konzepte angegeben, auf deren Umsetzung innerhalb eines Persistenzmechanismus in diesem Abschnitt eingegangen wird. Der Persistenzmechanismus ist für alle Aktivitäten zuständig, die im Zusammenhang mit der Speicherung von Objekten auf einem dauerhaften Speichermedium, deren Wiederbeschaffung oder Entfernung von dem Speichermedium stehen.

Die Durchführung dieser Funktionen geschieht für eine Applikation vollständig transparent. Sie kennt lediglich die Schnittstelle des Persistenzmechanismus, die sie zur Kommunikation mit diesem nutzt.

Da es bei der Programmentwicklung auch während der Umsetzung des Entwurfs zu Änderungen an den Modellierungen der Klassen kommen kann, war es notwendig, einen Persistenzmechanismus zu entwerfen, der keine starren Verarbeitungsvorschriften für bestimmte Klassen beinhaltet, sondern die zu persistierenden Objekte auf flexible Art und Weise analysieren und verarbeiten kann. Diese Art des Entwurfs war von Vorteil, da es zu späteren Zeitpunkten noch zu Änderungen an den Modellierungen der geodätischen Entitäten kommen konnte. Solche Änderungen in fortgeschrittenen Entwicklungsstadien sollten ohne starke Seiteneffekte hinsichtlich des Persistenzmechanismus durchgeführt werden können.

In diesem Abschnitt werden zunächst die Anforderungen an potentiell persistente Klassen, die mit Hilfe des Persistenzmechanismus persistiert werden können, beschrieben. In diesem Zusammenhang wird auch auf die Erschaffung persistenter Versionen der in JAVA vorhandenen Kollektionen eingegangen. Hiernach wird die nach außen hin sichtbare Funktionalität des Persistenzmechanismus erläutert, um im Anschluss auf die jeweiligen Klassen zur Umsetzung der Funktionalität einzugehen.

Im Rahmen der Zielsetzung wurden die Gründe für die Verwendung eines relationalen Datenbanksystems zur Langzeitdatenspeicherung dargelegt. Abschnitt 6.2.4 befasst sich daher mit der Realisiserung der datenbankgestützten Objektpersistenz.

Zum Abschluss wird in Abschnitt 6.2.5 die Realisierung des Datenexports und des Datenimports unter Verwendung der XML erläutert. Hierfür wurden bereits in Kapitel 5 Abbildungsvorschriften angegeben.

6.2.2 Potentiell persistente Klassen

Klassen, bei denen die Möglichkeit bestehen soll, dass ihre Exemplare mit Hilfe des Persistenzmechanismus verarbeitet werden können, müssen von der Klasse *PersistentObject* erben, die wie in Abb.6.2 dargestellt, die Schnittstelle *PersistenceInterface* realisiert.

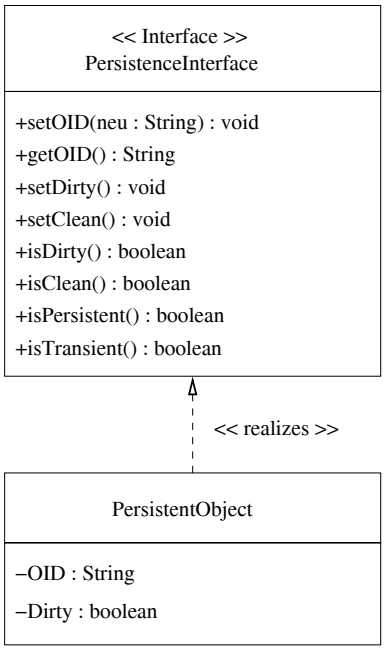

Abbildung 6.2: Die Realisierung der Schnittstelle *PersistenceInterface* durch die Klasse *PersistentObject*

Klassen, die die dargestellte Schnittstelle realisieren oder von der Klasse *PersistentObject* erben, werden im Folgenden als potentiell persistent bezeichnet. Die in Abb. 6.2 dargestellte Schnittstelle definiert Methodik, die vom Persistenzmechanismus zur Persistierung von Exemplaren solcher Klassen genutzt wird:

- Die Methoden `getOID()` und `setOID(···)` dienen zum Ermitteln bzw. Setzen des aktuellen Objektidentifikators des Objekts. Diese Objektidentifikatoren werden vom Persistenzmechanismus erzeugt, gesetzt und ggf. auch wieder entfernt. Objekte, deren Objektidentifikatoren nicht gesetzt sind, sind transient. Das Attribut zur Aufnahme des Objektidentifikators wird von den potentiell persistenten Klassen bereitgestellt.

- Die Methoden `isDirty()`, `isClean()`, `setDirty()` und `setClean()` dienen dazu, zu ermitteln, ob der Objektzustand seit dem letzten schreibenden Zugriff auf das Speichermedium verändert wurde, bzw. dazu, eine solche Änderung zu protokollieren. Diese vier Methoden werden hauptsächlich von der datenbankgestützten Realisierung des Persistenzmechanismus genutzt, um zu entscheiden, ob bei der erneuten Speicherung eines bereits persistenten Objektes Änderungen an der Datenbasis durchzuführen sind.

- Die Methoden `isPersistent()` und `isTransient()` dienen dazu zu ermitteln, ob ein potentiell persistentes Objekt persistent oder transient ist.

Bei der Verarbeitung eines potentiell persistenten Objekts ermittelt der Persistenzmechanismus den Objektzustand, also die momentane Belegung der Attribute, mit Hilfe der in JAVA zur Verfügung stehenden *Reflection* [11], die es ermöglicht, Informationen über Objekte und Objektstrukturen zur Laufzeit zu ermitteln. Bei der Ermittlung des Objektzustands werden jedoch keine transienten oder abstrakten Attribute und auch keine Klassenattribute betrachtet. Zyklen in Objektstrukturen werden vom Persistenzmechanismus nicht verarbeitet.

Um potentiell persistente Versionen der in JAVA vorhandenen Kollektionen zur Verfügung zu stellen, existieren die Schnittstellen *PersistentListInterface*, *PersistentSetInterface* und *PersistentMapInterface*, die sowohl die Funktionalität der Schnittstelle *PersistenceInterface* wie auch die der Schnittstellen *java.util.List*, *java.util.Set* bzw. *java.util.Map* erben. Realisierungen dieser Schnittstellen werden durch Subklassenbildung der in JAVA bereits vorhandenen Kollektionen gebildet. Für die angegebenen, potentiell persistenten Versionen der Kollektionen sind Verarbeitungsvorschriften in den Persistenzmechanismus integriert. Einen Überblick über die vorhandenen Realisierungen bietet Tabelle 6.1.

realisierte Schnittstelle	Klasse	Subtyp von
PersistentListInterface	PersistentVector	java.util.Vector
	PersistentArrayList	java.util.ArrayList
	PersistentLinkedList	java.util.LinkedList
PersistentSetInterface	PersistentHashSet	java.util.HashSet
	PersistentTreeSet	java.util.TreeSet
PersistentMapInterface	PersistentHashMap	java.util.HashMap
	PersistentTreeMap	java.util.TreeMap

Tabelle 6.1: Die Realisierungen der Schnittstellen *PersistentListInterface*, *PersistentSetInterface* und *PersistentMapInterface*

Exemplare von Klassen, die nicht der von *PersistentObject* ausgehenden Vererbungshierarchie entstammen, können aufgrund der in JAVA fehlenden Mehrfachvererbung nicht ohne weiteres mit dem Persistenzmechanismus verarbeitet werden. Die relevanten Attribute solcher Klassen können jedoch in einer potentiell persistenten Klasse nachgebildet werden, mit deren Hilfe die Objektzustände dann persistiert werden können.

Durch den hier beschriebenen Ansatz wird die, in Abschnitt 5.1.1 beschriebene, Persistenz durch Vererbung realisiert.

6.2.3 Die Funktionalität des Persistenzmechanismus

Die Funktionalität des Persistenzmechanismus wird durch die Schnittstelle *PersistenceLayerInterface* definiert, die in Abb. 6.3 dargestellt ist.

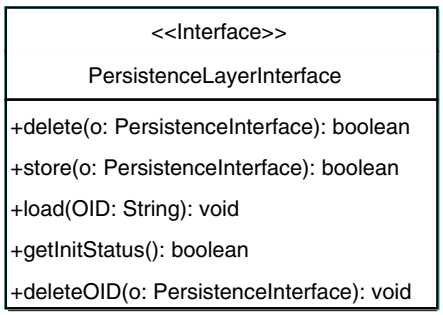

Abbildung 6.3: Die Schnittstelle *PersistenceLayerInterface*

Mit Hilfe dieser Funktionalität kann eine Applikation potentiell persistente Objekte auf einem dauerhaften Datenträger ablegen, sie manipulieren, laden oder löschen:

- Die Methode `delete(···)` dient dazu, das als Parameter übergebene Objekt und alle von ihm aus erreichbaren Objekte vom Speichermedium zu entfernen. Erreichbare Objekte sind in diesem Zusammenhang solche, auf die das als Parameter übergebene Objekt eine Referenz besitzt. Die Erreichbarkeit ist hier eine rekursive Eigenschaft. Nach der Entfernung vom Speichermedium ist das Objekt im Speicher als transient markiert.
- Die Methode `store(···)` dient dazu, das als Parameter übergebene Objekt und alle von ihm aus erreichbaren Objekte auf dem Speichermedium abzulegen. Nach dieser Operation ist das Objekt im Speicher als persistent markiert.

- Die Methode `load(···)` dient dazu, das Objekt, dessen Objektidentifikator als Parameter übergeben wird, vom Speichermedium in den Speicher einzulesen. Hierbei werden alle Objekte, die zum Zeitpunkt der letzten Speicherung erreichbar waren, ebenfalls in den Speicher eingelesen und es werden die ursprünglichen Assoziationen zwischen den Objekten wieder hergestellt.
- Die Methode `getInitStatus()` dient dazu, den Status des Persistenzmechanismus nach der Initialisierung abzufragen. Diese Methode wird bei der datenbankgestützten Realisierung verwendet, um zu ermitteln, ob die Verbindung zur Datenbank aufgebaut werden konnte.
- Die Methode `deleteOID(···)` entfernt den Objektidentifikator des als Parameter übergebenen Objekts sowie die Objektidentifikatoren aller von ihm aus erreichbaren Objekte. Diese Methode kann z.B. benutzt werden, um Objekte mit mehreren Identitäten auf dem Speichermedium abzulegen.

Durch die gerade beschriebene Funktionalität wird die in Abschnitt 5.1.4 beschriebene Persistenz durch Erreichbarkeit realisiert.

Zur Schnittstelle *PersistenceLayerInterface* existieren zwei Realisierungen, die potentiell persistente Objekte mit Hilfe eines relationalen Datenbankmanagement-Systems und durch Verwendung der XML persistieren. Diese beiden Realisierungen werden in den folgenden Abschnitten beschrieben.

6.2.4 Die Klassen zur Realisierung der datenbankgestützten Objektpersistenz

Die Klassen zur Realisierung der datenbankgestützten Objektpersistenz realisieren die in der Schnittstelle *PersistenceLayerInterface* definierte Funktionalität unter Verwendung eines relationalen Datenbankmanagement-Systems. Einen Überblick über diese Klassen liefert Abb. 6.4.

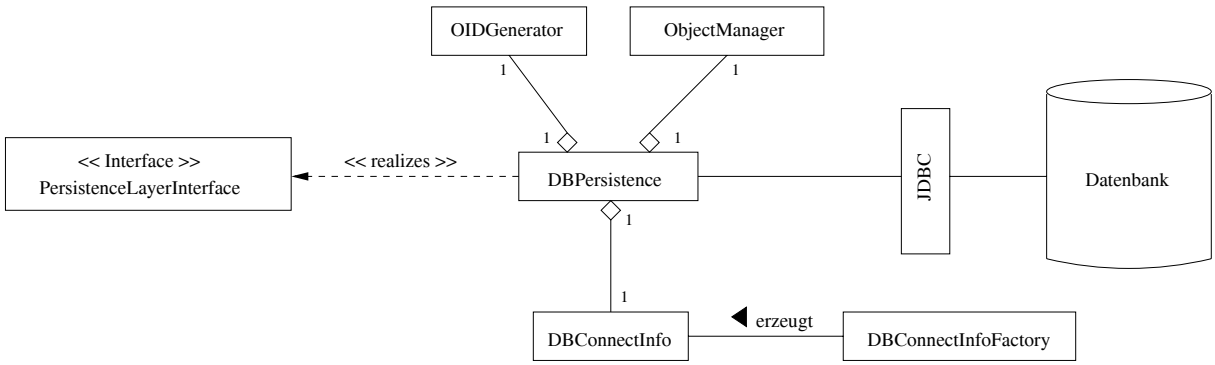

Abbildung 6.4: Die Klassen zur Realisierung der datenbankgestützten Objektpersistenz

Die Abbildung der Objekte auf die Tabellen der Datenbank geschieht durch die in den Abschnitten 5.2.2.3, 5.2.2.4, 5.2.2.5 und 5.2.2.6 vorgestellten Konzepte. Bei der Vererbung wurde Ansatz 3 aus Abschnitt 5.2.2.6 umgesetzt.

Um die in Abschnitt 6.2.3 beschriebene Funktionalität mit einer Datenbank nutzen zu können, müssen in der Datenbank einige Tabellen vorhanden sein:

- Zu jeder potentiell persistenten Klasse, deren Exemplare in der Datenbank abgelegt werden sollen, muss eine Tabelle vorhanden sein, die den Namen der Klasse trägt. Für jedes in der Klasse definierte Attribut enthält die Tabelle eine Spalte, die wie das Attribut bezeichnet ist. Für die primitiven Datentypen und ihre Wrapper-Objekte existieren hierbei die in Tabelle 6.2 angegebenen Zuordnungen zu den Spaltentypen. Objektreferenzen werden von Spalten aufgenommen, die Zeichenketten speichern können.

  Zusätzlich zu den Spalten, die durch die Attribute der Klassen definiert werden, enthält jede Tabelle eine Spalte zur Aufnahme des Objektidentifikators. Diese Spalte trägt den Namen der Klasse und ist in der Lage, eine Zeichenkette aufzunehmen. Bei Klassen, die nicht direkt von *PersistentObject* erben,

muss eine weitere Spalte zur Aufnahme des Objektidentifikators der direkten Oberklasse vorhanden sein. Der beschriebene Sachverhalt ist in Abbildung 6.5 dargestellt. Die Beziehung zwischen den dargestellten Unterklassen von *PersistentObject* wird durch einen Fremdschlüssel hergestellt.

Java-Datentyp	Spaltentyp
boolean	Zeichenkette
byte	nummerischer Typ
char	Zeichenkette
double	nummerischer Typ
float	nummerischer Typ
int	nummerischer Typ
long	nummerischer Typ
short	nummerischer Typ
String	Zeichenkette

Tabelle 6.2: Zuordnungen der primitiven Datentypen zu den Spaltentypen

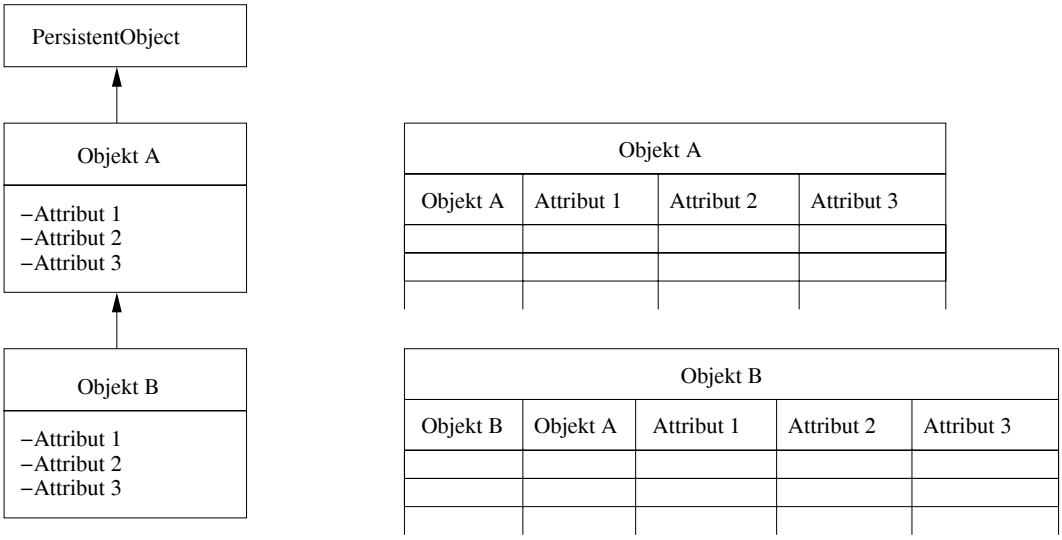

Abbildung 6.5: Zuordnung der Klassen zu Tabellen

- Zusätzlich existiert eine Tabelle, die zu jedem Objekt, das in der Datenbank abgelegt wird, die Beziehung zwischen dem Objektidentifikator und dem Klassennamen speichert. Diese Tabelle wird im Folgenden als Objektkatalog bezeichnet. Die in ihm enthaltenen Zuordnungen werden hauptsächlich beim Laden von Objekten verwendet, um zu einem Objektidentifikator die zugehörige Klasse zu ermitteln.

- Zu den Schnittstellen *PersistentList*, *PersistentSet* und *PersistentMap* existiert ebenfalls jeweils eine Tabelle. In diesen werden die Identitäten und Beziehungen abgelegt, die in den die Schnittstellen realisierenden Objekten enthalten sind. In Abbildung 6.6 ist die zur Schnittstelle *PersistentList* gehörende Tabelle abgebildet. Die Spalte *PersistentList* nimmt hierbei den Objektidentifikator der Liste auf. In der Spalte *OID* werden die Objektidentifikatoren der enthaltenen Objekte abgelegt und die Spalte *Idx* enthält die Indices der in der Liste enthaltenen Elemente.

PersistentList		
PersistentList	Idx	OID

Abbildung 6.6: Die Tabelle zur Schnittstelle *PersistentList*

- Zur Generierung der Objektidentifikatoren wird eine Tabelle mit Metadaten verwendet. Diese Tabelle trägt den Namen *SEQUENCES* und enthält eine Spalte, die zur Speicherung einer Zahl verwendet wird, die zur Generierung von künstlichen Objektidentitäten verwendet wird. Auf die Verwendung der Tabelle wird in Abschnitt 6.2.4.1 eingegangen.

In den folgenden Abschnitten werden die Zuständigkeiten der in Abb. 6.4 dargestellten Klassen beschrieben. Auf Implementierungsdetails wird hierbei nur soweit eingegangen, wie dies zum Verständnis der Funktionalität der jeweiligen Klassen notwendig ist.

6.2.4.1 Die Klasse OIDGenerator

Die Klasse *OIDGenerator* kapselt die Funktionalität zur Generierung von eindeutigen Objektidentifikatoren. Wie in Abschnitt 5.2.2.2 erläutert, wird zur eindeutigen Identifizierung von Datensätzen in Datenbanken ein Schlüsselattribut benötigt. Die Schwierigkeiten, die sich hierbei durch die Wahl von ungeeigneten Werten ergeben können, wurden bereits beschrieben.

Die Klasse *OIDGenerator* stellt die Methode `getNewOID()` zur Verfügung, die einen neuen, bisher nicht verwendeten Objektidentifikator erzeugt. Die erzeugten Objektidentifikatoren bestehen aus Zeichenketten, die sich aus einem invarianten Bestandteil, der die Zeichenkette als Objektidentifikator kennzeichnet sowie aus zwei Zahlenwerten, `High` und `Low`, zusammensetzt. Der Wert von `High` wird während der Initialisierung des Objekts aus der Tabelle *SEQUENCES* bestimmt, während der Wert von `Low` im Speicher verwaltet wird. Nach jeder Anforderung eines Objektidentifikators durch den Persistenzmechanismus wird das folgende Codefragment ausgeführt:

```
if (Low == Schwellwert)
  {
    Low = 0;
    High++;
    storeToDB(High);
  }
else
  {
    Low++;
  }
```

Abbildung 6.7: Codefragment zur Generierung der Objektidentifikatoren

Der Wert `Low` ist durch den 64-Bit Datentyp *long* implementiert. Für den Schwellwert wurde die größte durch diesen Datentyp darstellbare Zahl gewählt, wodurch ein Datenbankzugriff zur Ermittlung des nächsten Wertes für den Wert `High` erst nach der Generierung von $2^{63} - 1$ Objektidentifikatoren nötig wird. Der Wert `High` wird durch den Datentyp *java.math.BigInteger* implementiert, da der Wertebereich dieses Datentyps nur durch den der Laufzeitumgebung zur Verfügung stehenden Speicher begrenzt wird.

Die auf diese Weise ermittelten Objektidentifikatoren sind eindeutig und eignen sich daher als Primärschlüssel für die Datenbank. Ein Exemplar dieser Klasse wird im Zusammenhang mit der Speicherung von Objekten in die Datenbank verwendet.

6.2.4.2 Die Klasse ObjectManager

Die Klasse *ObjectManager* stellt die Funktionalität zur Aufnahme von Objektidentifikatoren und den zugehörigen Objekten zur Verfügung. Während eines Ladevorgangs wird für jedes zu ladende Objekt geprüft, ob es während des aktuellen Vorgangs bereits geladen wurde. Ist dies der Fall, so wird das Objekt mit Hilfe des Objektidentifikators ermittelt und zurückgeliefert. Andernfalls wird das Objekt aus der Datenbank geladen und der Objektidentifikator und das zugehörige Objekt werden im *ObjectManager* registriert. Nachdem alle zum aktuellen Ladevorgang gehörenden Objekte geladen wurden, werden alle Einträge im *ObjectManager* gelöscht. Hierdurch wird die Vervielfältigung von Objekten aufgrund von mehrfachen Ladeanforderungen vermieden. Diese Klasse wird im Zusammenhang mit dem Laden von Objekten aus der Datenbank verwendet.

6.2.4.3 Die Klasse DBConnectInfo

Die Klasse *DBConnectInfo* kapselt die Informationen, die zum Verbindungsaufbau mit der Datenbank benötigt werden. Sie stellt eine abstrakte Basisklasse dar, von der Subklassen für spezielle Datenbanktypen abgeleitet werden müssen.

Die Klasse definiert die Attribute zur Aufnahme der Verbindungsinformationen. Hierzu zählen die Namen der zu verwendenden Treiberklasse sowie des Treibertyps zum Verbindungsaufbau, der Schemaname zum Zugriff auf die Tabellen, der Benutzername und das Passwort zur Anmeldung bei der Datenbank, der Name des Datenbankservers sowie die Portnummer zum Verbindungsaufbau und der Datenbankname. Lediglich die Methodik für die Erzeugung des *Connect-Strings*, der für den Verbindungsaufbau zur Datenbank benötigt wird, muss von den Subklassen implementiert werden. Die Klasse besitzt Methoden, um ihren Zustand in einer XML-Datei zu persistieren bzw. ihn aus einer solchen einzulesen. Da das System ein Datenbankmanagementsystem vom Typ *Oracle 8i* zur Persistierung der Objekte benutzt, existiert eine entsprechende Subklasse.

6.2.4.4 Die Klasse DBConnectInfofactory

Die Klasse *DBConnectInfofactory* realisiert eine Objektfabrik [14], mit der Subtypen der Klasse *DBConnectInfo* erzeugt werden können. Auf das zugehörige Entwurfsmuster wurde bereits in Abschnitt 3.25 eingegangen. Es können wahlweise nur der Name des Datenbanktyps oder auch eine zusätzliche Liste der in Abschnitt 6.2.4.3 definierten Attribute als Parameter übergeben werden. Das Fabrikobjekt erzeugt dann den zum Datenbanktyp passenden Untertyp der Klasse *DBConnectInfo*. Falls lediglich der Name eines Datenbanktyps als Parameter übergeben wurde, müssen die zum Verbindungsaufbau benötigten Attribute nachträglich besetzt werden.

6.2.4.5 Die Klasse DBPersistence

Die Klasse *DBPersistence* realisiert die durch die Schnittstelle *PersistenceLayerInterface* definierte Funktionalität unter Verwendung eines relationalen Datenbankmanagementsystems. Hierzu wird auf die Funktionalität der in den vorangegangenen Abschnitten beschriebenen Klassen zurückgegriffen.

Diese Klasse realisiert die zentrale Methodik zur Persistierung der potentiell persistenten Objekte unter Verwendung einer relationalen Datenbank. Einen Überblick über diese Methodik liefert Abb. 6.8, in der sich nicht nur die in der Schnittstelle *PersistenceLayerInterface* definierten, sondern auch weitere, zur Erbringung der Funktionalität notwendige Methoden finden.

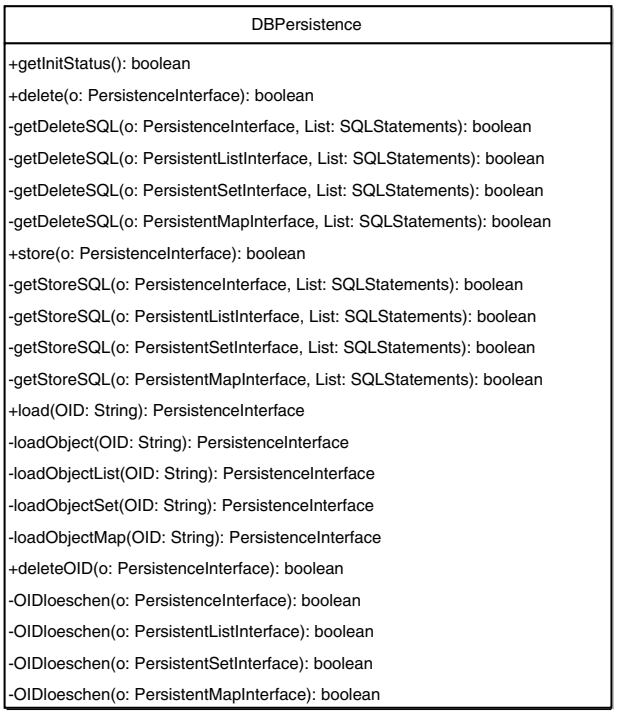

Abbildung 6.8: Die Funktionalität der Klasse *DBPersistence*

Wie aus Abb. 6.4 ersichtlich ist, sind Exemplare dieser Klasse mit jeweils einem Exemplar der Klassen *DBConnectInfo*, *ObjectManager* und *OIDGenerator* assoziiert. Die Realisierung der in Abb. 6.8 dargestellten Methoden wird nun beschrieben.

• Die Methode `getInitStatus()` dient zur Prüfung, ob das Objekt fehlerfrei initialisiert werden konnte. Ist dies der Fall, so konnte die Verbindung zur Datenbank aufgebaut werden, und alle assoziierten Objekte konnten ebenfalls fehlerfrei initialisiert werden.

• Die Methode `delete(···)` entfernt das als Parameter übergebene Objekt aus der Datenbank, sofern dieses persistent ist. Hierbei werden auch alle erreichbaren Objekte aus der Datenbank entfernt. Die hierzu nötigen SQL-Anweisungen werden durch die verschiedenen Versionen der Methode `getDeleteSQL(···)` erzeugt, die ebenfalls alle vom übergebenen Objekt erreichbaren Objekte verarbeiten. Die Arbeitsweise dieser Methoden soll durch das folgende Programmfragment und die folgenden Aktivitätsdiagramme veranschaulicht werden:

```
public boolean delete(PersistenceInterface o) {
  List StatementList = new List();
  if(getDeleteSQL(o,StatementList))
    {
      Connection conn = DriverManager.getConnection(ConInfo.getConnectString());
      if (executeStatements(StatementList,conn))
        {
          OIDloeschen(o);
          return true;
        }
      else return false;
    }
  else return false;
}
```

Abbildung 6.9: Programmfragment der Methode `delete(···)`

Die Methode `executeStatements(···)` führt die in der als Parameter übergebenen Liste enthaltenen SQL-Anweisungen innerhalb einer Datenbanktransaktion durch. Der Ausführungsstatus der Datenbanktransaktion entscheidet über den Rückgabewert der Methode. Bei einer fehlerfreien Ausführung wird der Wert *wahr* zurückgeliefert, ansonsten der Wert *falsch*.

Die Methode `getDeleteSQL(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistenceInterface* realisieren, erzeugt zuerst die SQL-Anweisungen, um das als Parameter übergebene Objekt aus der Datenbank und dem Objektkatalog zu entfernen. Im Anschluss wird über die Attribute des Objekts iteriert, wobei bei allen Referenztypen die entsprechende Version der Methode `getDeleteSQL(···)` aufgerufen wird. Das Aktivitätsdiagramm der Methode ist in Abb. 6.10 dargestellt.

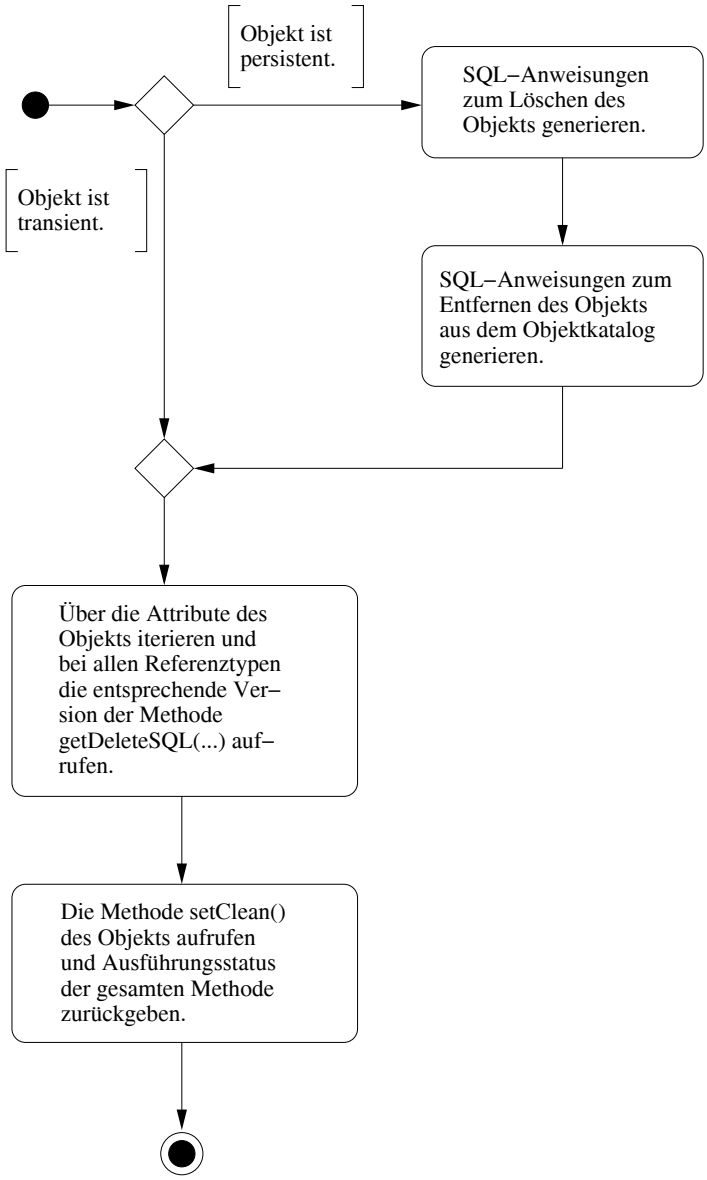

Abbildung 6.10: Das Aktivitätsdiagramm der Methode `getDeleteSQL(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistenceInterface* realisieren

Abb. 6.11 zeigt das Aktivitätsdiagramm der Methode `getDeleteSQL(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistentListInterface* realisieren. Der wesentliche Unterschied zu der in Abb. 6.10 dargestellten Version der Methode ist, dass in diesem Fall nicht über die Attribute, sondern über die enthaltenen Referenzen iteriert wird.

Die Versionen der Methode `getDeleteSQL(···)` zur Verarbeitung von Objekten, die eine der Schnittstellen *PersistentListInterface*, *PersistentSetInterface* oder *PersistentMapInterface* realisieren, sind in ihrer Funktionsweise analog der in Abb. 6.11 dargestellten.

Sämtliche Versionen der Methode `getDeleteSQL(···)` liefern als Rückgabewert *wahr*, wenn alle Verarbeitungsschritte fehlerfrei ausgeführt werden konnten, und andernfalls den Wert *falsch*. In einem solchen Fall wird dies durch die Persistenzschicht an das aufrufende Objekte übergeben um dieses von dem nicht gelungenen Löschvorgang in Kenntniss zu setzen. Die Methode `delete(···)` liefert den Wert *wahr*, wenn das Objekt erfolgreich aus der Datenbank entfernt werden konnte und andernfalls den Wert *falsch*.

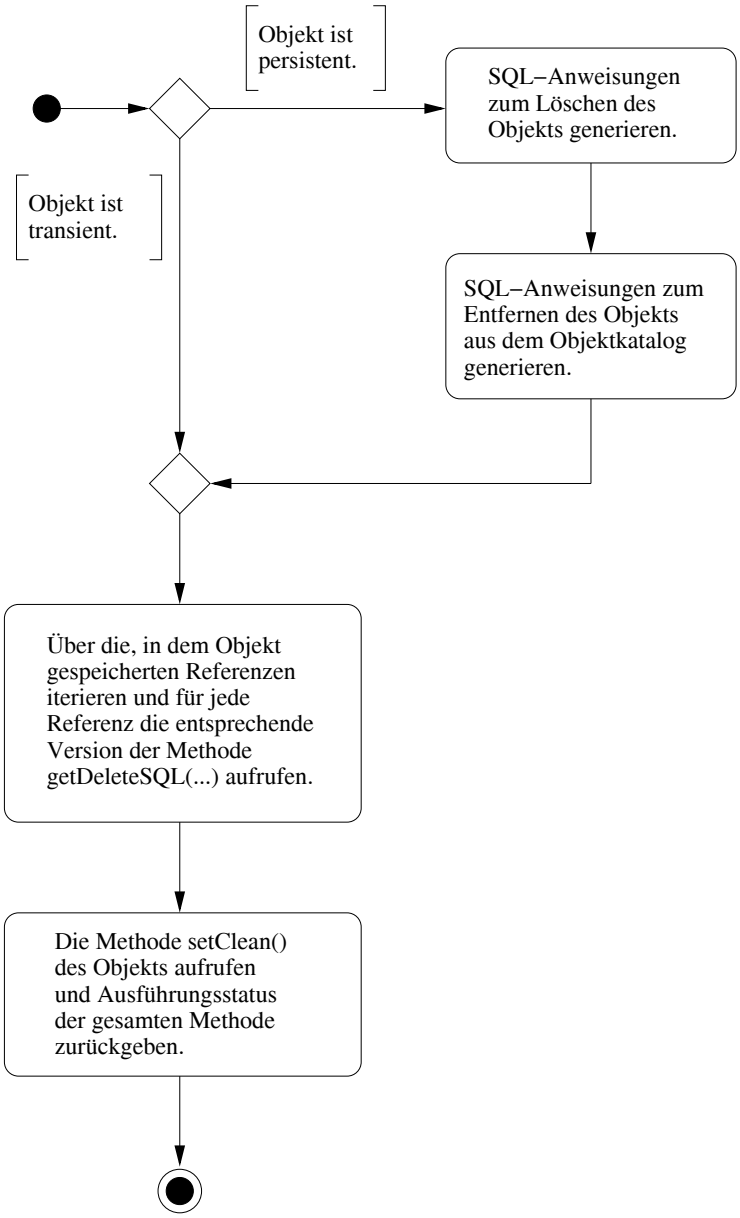

Abbildung 6.11: Das Aktivitätsdiagramm der Methode `getDeleteSQL(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistentListInterface* realisieren

• Die Methode `store(···)` speichert das als Parameter übergebene Objekt in der Datenbank, sofern dieses transient ist. Hierbei werden auch alle erreichbaren Objekte in der Datenbank gespeichert. Bei persistenten Objekten wird geprüft, ob die Attributbelegung seit der letzten Speicherung des Objekts geändert wurde und deshalb Änderungen in der Datenbasis durchzuführen sind. Die nötigen SQL-Anweisungen werden durch die verschiedenen Versionen der Methoden `getStoreSQL(···)` und `getUpdateSQL(···)` erzeugt, die ebenfalls alle vom übergebenen Objekt erreichbaren Objekte verarbeiten. Die Arbeitsweise dieser Methoden soll durch das folgende Programmfragment und die folgenden Aktivitätsdiagramme veranschaulicht werden:

```
public boolean store(PersistenceInterface o) {
  List StatementList = new List();
  boolean status = false;
  if(o.isTransient)
    { status = getStoreSQL(o,StatementList);}
  else
    { status = getUpdateSQL(o,StatementList);}
  if(status)
    {
      Connection conn = DriverManager.getConnection(ConInfo.getConnectString());
      if (executeStatements(StatementList,conn))
        { return true; }
      else
        {
          OIDLoeschen(o);
          return false;
        {
    }
  else return false;
}
```

Abbildung 6.12: Programmfragment der Methode `store(···)`

Die Methode `getStoreSQL(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistenceInterface* realisieren, prüft, ob das als Parameter übergebene Objekt persistent ist. Ist dies der Fall, so wird das Objekt an die entsprechende Version der Methode `getUpdateSQL(···)` übergeben. Andernfalls wird vom assoziierten Exemplar der Klasse *OIDGenerator* ein neuer Objektidentifikator angefordert. Bei der Iteration über die Attribute des zu verarbeitenden Objekts wird für alle Referenztypen die entsprechende Version der Methode `getStoreSQL(···)` aufgerufen und der ermittelte Objektidentifikator wird in dem Objekt abgelegt. Hierdurch ist sichergestellt, dass zu dem Zeitpunkt, an dem die SQL-Anweisungen zur Speicherung der Attributbelegung eines Objekts generiert werden, alle von diesem Objekt aus erreichbaren Objekte mit einem Objektidentifikator versehen sind und die SQL-Anweisungen zur Speicherung dieser Objekte vorliegen. Bei der Generierung der SQL-Anweisungen wird für den Fall, dass ein Attribut einen Referenztyp enthält, dessen Objektidentifikator als Wert des Attributs in die SQL-Anweisung eingefügt. Im Anschluss werden noch die SQL-Anweisungen zum Speichern der aktuellen Attributbelegung des Objekts in die Datenbank sowie der entsprechenden Zuordnung im Objektkatalog erzeugt. Das Aktivitätsdiagramm der beschriebenen Methode ist in Abb. 6.13 dargestellt.

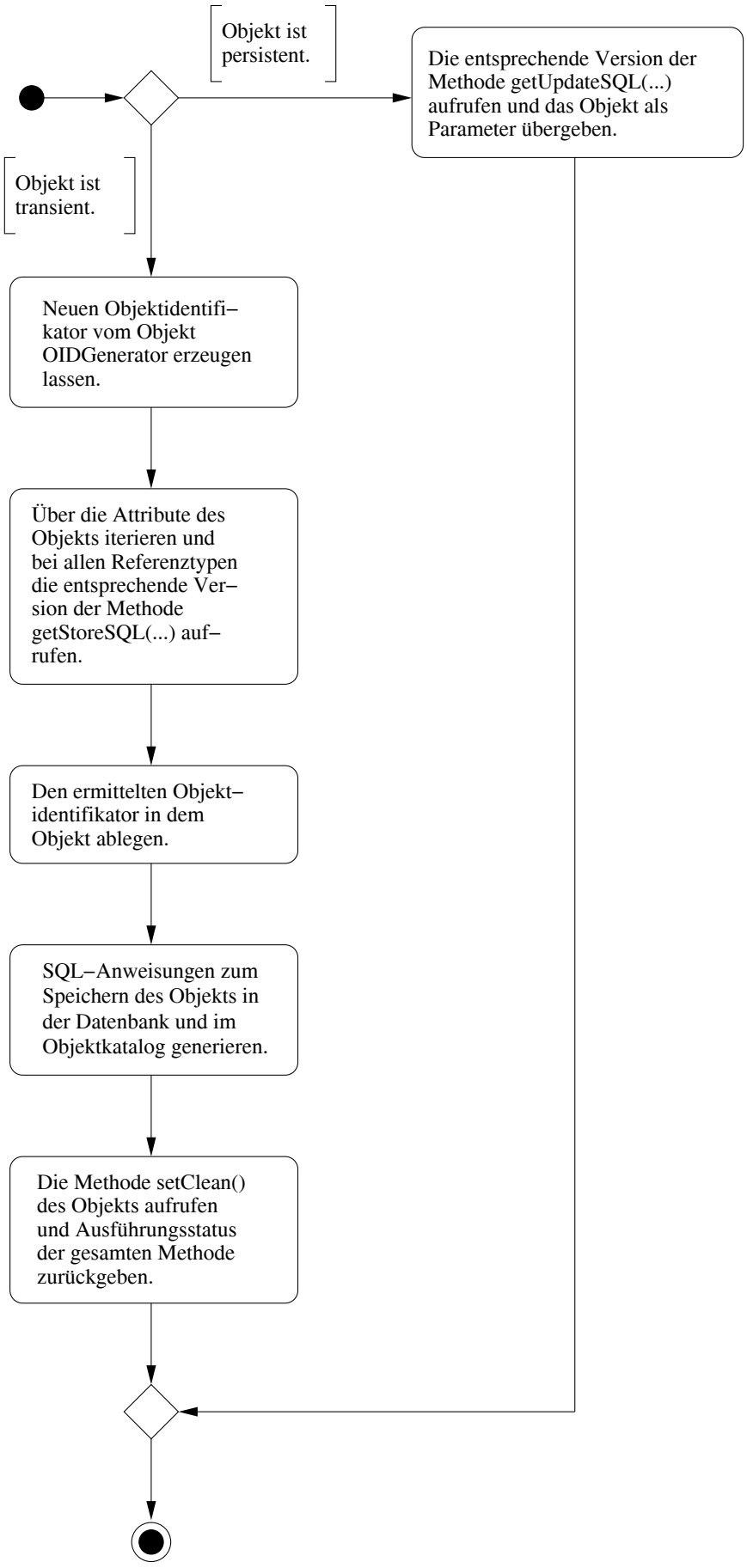

Abbildung 6.13: Das Aktivitätsdiagramm der Methode `getStoreSQL(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistenceInterface* realisieren

Abb. 6.14 zeigt das Aktivitätsdiagramm der Methode `getStoreSQL(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistentListInterface* realisieren. Der wesentliche Unterschied zu der in Abb. 6.13 dargestellten Version der Methode ist, dass in diesem Fall nicht über die Attribute, sondern über die enthaltenen Referenzen iteriert wird. Aus diesem Grund werden bei der Generierung der SQL-Anweisungen zum Speichern des Objekts solche Anweisungen erzeugt, die die im Objekt enthaltenen Objektidentifikatoren in der entsprechenden Reihenfolge in der Datenbank ablegen.

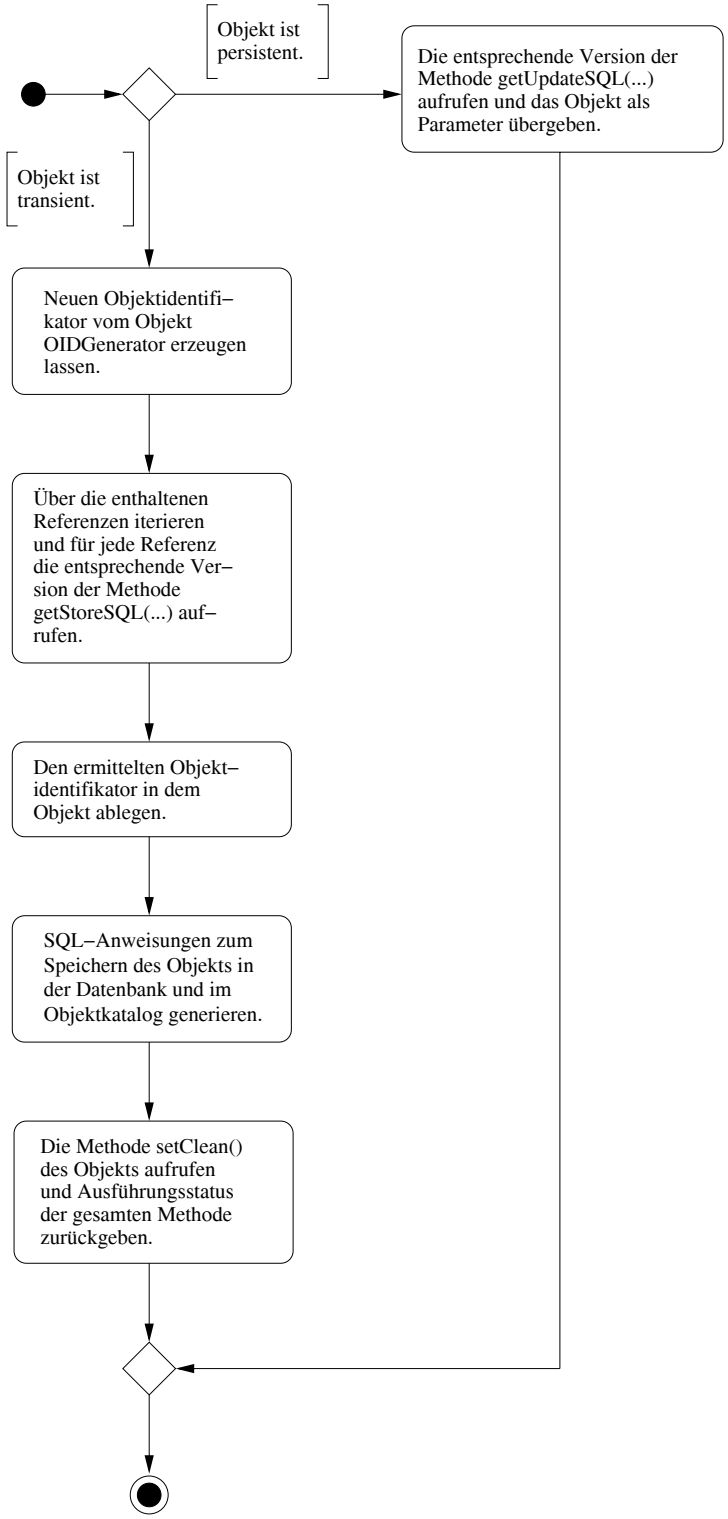

Abbildung 6.14: Das Aktivitätsdiagramm der Methode `getStoreSQL(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistentListInterface* realisieren

Die Versionen der Methode `getStoreSQL(···)` zur Verarbeitung von Objekten, die eine der Schnittstellen *PersistentSetInterface* oder *PersistentMapInterface* realisieren, sind in ihrer Funktionsweise analog der in Abb. 6.14 dargestellten.

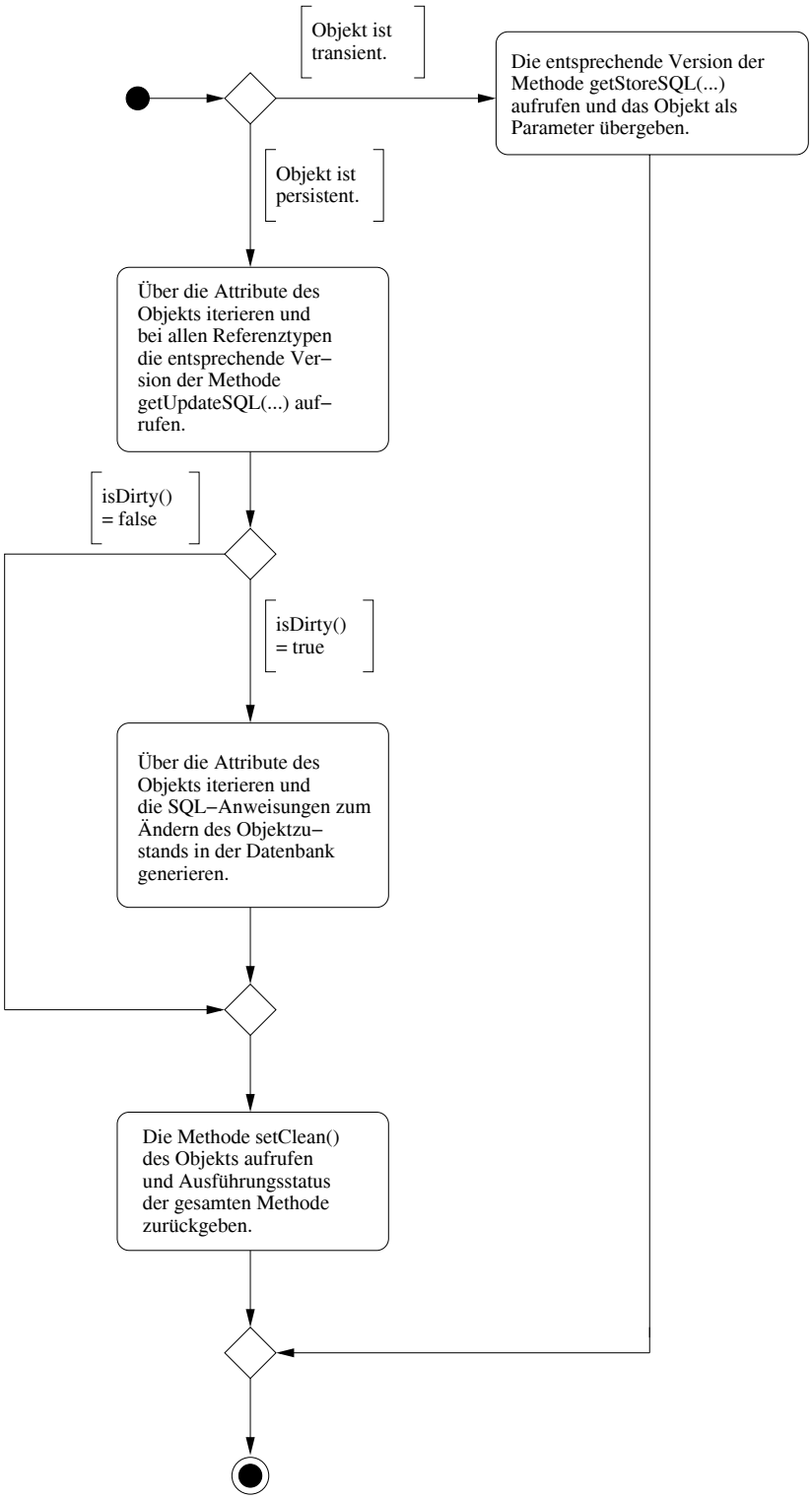

Abbildung 6.15: Das Aktivitätsdiagramm der Methode `getUpdateSQL(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistenceInterface* realisieren

Die Methode `getUpdateSQL(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistenceInterface* realisieren, prüft, ob das als Parameter übergebene Objekt persistent ist. Ist dies der Fall, so wird das Objekt an die entsprechende Version der Methode `getStoreSQL(···)` übergeben. Andernfalls wird über die Attribute

des Objekts iteriert und mit den Referenztypen die entsprechende Version der Methode `getUpdateSQL(···)` aufgerufen. Im Anschluss wird geprüft, ob der Objektzustand seit dem letzten schreibenden Zugriff auf die Datenbank geändert wurde. In einem solchen Fall werden SQL-Anweisungen zur Anpassung des in der Datenbank gespeicherten Zustands generiert. Diese Anpassung wird protokolliert und der Ausführungsstatus der Methode zurückgeliefert. Das Aktivitätsdiagramm dieser Methode ist in Abb. 6.15 dargestellt.

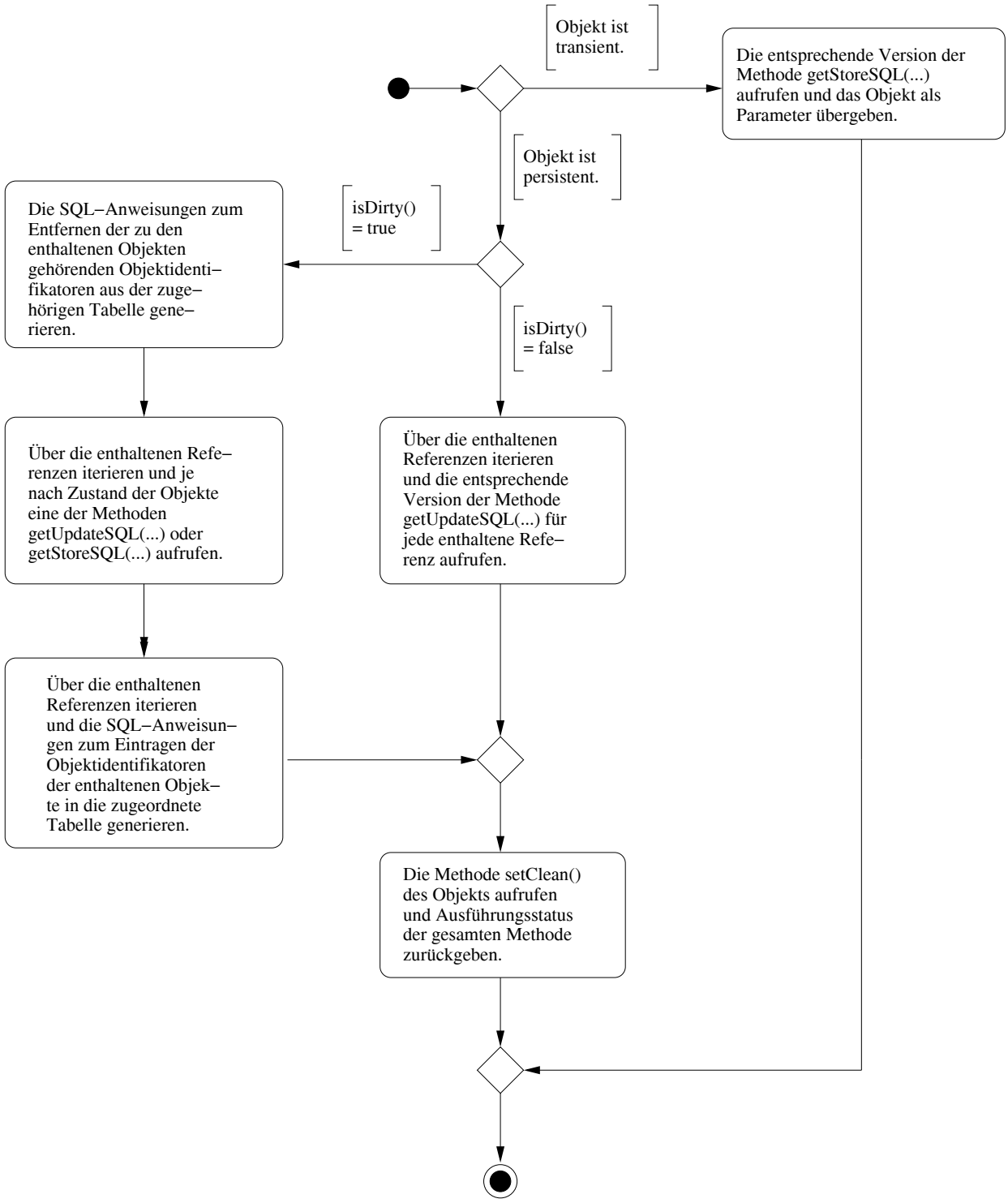

Abbildung 6.16: Das Aktivitätsdiagramm der Methode `getUpdateSQL(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistentListInterface* realisieren

Die in Abb. 6.16 dargestellte Version der Methode `getUpdateSQL(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistentListInterface* realisieren, unterscheidet sich in einigen Punkten von der oben beschriebenen. Nach der Prüfung, ob es sich bei dem übergebenen um ein persistentes Objekt handelt, wird wiederum geprüft, ob der Objektzustand seit dem letzten schreibenden Datenbankzugriff verändert wurde.

Ist dies der Fall, so wurden Objekte entfernt bzw. hinzugefügt. Es werden dann SQL-Anweisungen erzeugt, die die zu dem Objekt gehörenden Einträge aus der dieser Klasse zugeordneten Tabelle entfernen. Im Anschluss wird über die im Objekt enthaltenen Referenzen iteriert, und es werden, je nach Zustand der einzelnen Objekte, die entsprechenden Versionen der Methoden `getUpdateSQL(···)` bzw. `getStoreSQL(···)` aufgerufen. Hierdurch wird sichergestellt, dass nach diesem Schritt alle enthaltenen Objekte mit einem gültigen Objektidentifikator versehen sind und dass die SQL-Anweisungen zum Anlegen oder Ändern der Objektzustände der enthaltenen Objekte bereits generiert sind. Bei einer weiteren Iteration über die enthaltenen Objekte werden SQL-Anweisungen erzeugt, um diese Objektidentifikatoren in der zur Klasse gehörenden Tabelle abzulegen.

Wurden seit dem letzten schreibenden Datenbankzugriff keine Objekte entfernt bzw. hinzugefügt, so besteht die Möglichkeit, dass die enthaltenen Objekte seither verändert wurden. Daher wird über die enthaltenen Objekte iteriert und es wird für jedes Objekt die entsprechende Version der Methode `getUpdateSQL(···)` aufgerufen.

Nachdem die enthaltenen Objekte auf die eine oder andere Art verarbeitet wurden, wird diese Verarbeitung protokolliert und der Ausführungsstatus der Methode wird zurückgeliefert.

Die Versionen der Methode `getUpdateSQL(···)` zur Verarbeitung von Objekten, die eine der Schnittstellen *PersistentSetInterface* oder *PersistentMapInterface* realisieren, sind in ihrer Funktionsweise analog der in Abb. 6.16 dargestellten.

Sämtliche Versionen der Methoden `getStoreSQL(···)` bzw. `getUpdateSQL(···)` liefern als Rückgabewert *wahr*, wenn alle Verarbeitungsschritte fehlerfrei ausgeführt werden konnten, andernfalls den Wert *falsch*. Die Methode `store(···)` liefert den Wert *wahr*, wenn das übergebene Objekt erfolgreich in der Datenbank gespeichert werden konnte, andernfalls den Wert *falsch*.

• Die Methode `load(···)` lädt das zu dem als Parameter übergebenen Objektidentifikator gehörende Objekt aus der Datenbank, sofern ein solches existiert. Hierbei werden alle Objekte, die zurzeit der Speicherung erreichbar waren, ebenfalls geladen und es werden die ursprünglichen Referenzen wieder hergestellt. Das Laden der Objekte und die Datenbankzugriffe werden von den verschiedenen Versionen der Methode `loadObject(···)` durchgeführt. Die Arbeitsweise der verschiedenen Methoden wird durch das folgende Programmfragment und die folgenden Aktivitätsdiagramme verdeutlicht.

```
public PersistenceInterface load(String OID) {
  PersistenceInterface Objekt = null;
  Connection conn = DriverManager.getConnection(ConInfo.getConnectString());
  Objekt = loadObject(OID,conn);
  ObjectManager.reset();
  return Objekt;
}
```

Abbildung 6.17: Programmfragment der Methode `load(···)`

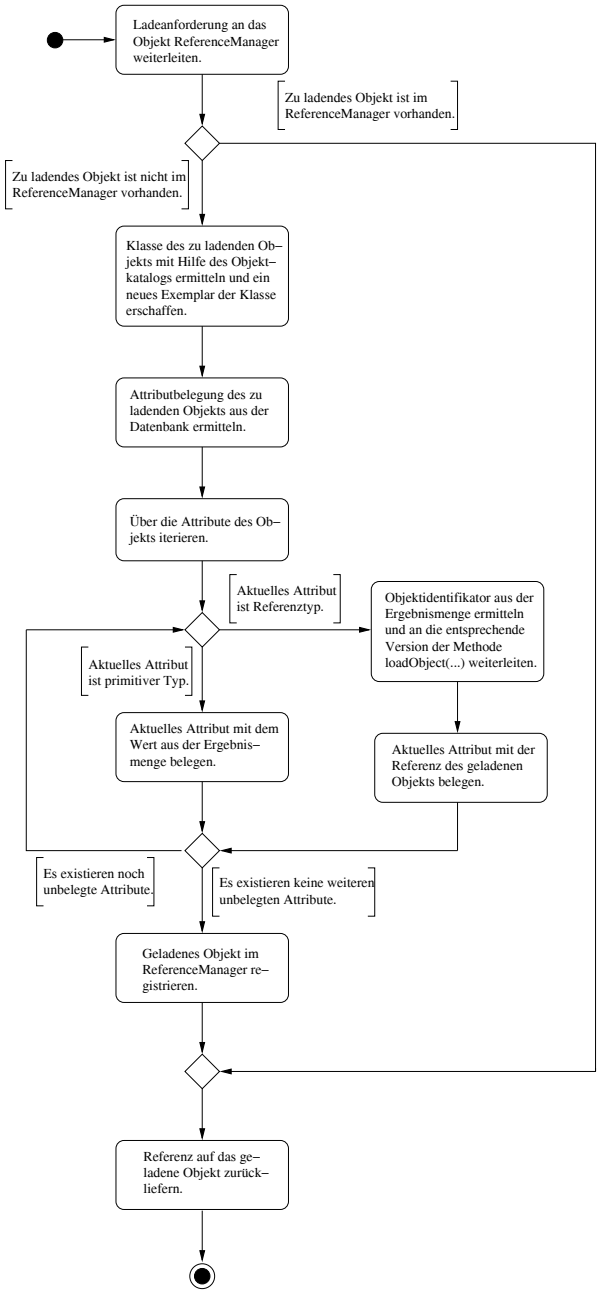

Abbildung 6.18: Das Aktivitätsdiagramm der Methode `loadObject(···)`

Die in Abb. 6.18 dargestellte Methode `loadObject(···)` leitet die Anfrage zuerst an das assoziierte Exemplar des *ObjectManager* weiter. Sollte die entsprechende Referenz nicht im *ObjectManager* vorhanden sein, so wird der Typ des zu ladenden Objekts mit Hilfe des Objektkatalogs ermittelt und ein neues Objekt dieses Typs erzeugt. Danach wird die in der Datenbank gespeicherte Attributbelegung geladen und über alle Attribute des neu erschaffenen Objekts iteriert. Alle primitiven Attribute werden sofort mit den in der Ergebnismenge ermittelten Werten belegt. Bei Referenztypen wird der ermittelte Wert als Objektidentifikator interpretiert, dessen zugehöriges Objekt wiederum durch die entsprechende Version der Methode `loadObject(···)` beschafft wird. Nachdem das Objekt vollständig geladen wurde, wird es beim Exemplar des *ObjectManager* registriert. Hierdurch wird eine Vervielfältigung von Objekten aufgrund von mehreren Ladeanforderungen verhindert.

Die in Abb. 6.19 dargestellte Methode `loadObjectList(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistentListInterface* realisieren, ermittelt ebenfalls zuerst den Typ des zu ladenden Objekts mit Hilfe des Objektkatalogs. Im Anschluss wird die Liste der Objektidentifikatoren der Objekte, die zum Zeitpunkt der Persistierung des Objekts enthalten waren, ermittelt. Diese Objektidentifikatoren werden der Reihe nach an die entsprechende Version der Methode `loadObject(···)` weitergeleitet, um diese zu laden. Nachdem ein Objekt

geladen wurde, wird es der Kollektion hinzugefügt. Nachdem alle Objekte geladen wurden, wird die Kollektion beim Exemplar des *ObjectManager* registriert.

Die Methoden `loadObjectSet(···)` und `loadObjectMap(···)` zur Verarbeitung von Objekten, die eine der Schnittstellen *PersistentSetInterface* oder *PersistentMapInterface* realisieren, sind in ihrer Funktionsweise analog der in Abb. 6.19 dargestellten.

Die Methode `load(...)` liefert die Referenz auf das aus der Datenbank geladene Objekt zurück, sofern ein Objekt zum übergebenen Objektidentifikator ermittelt werden konnte. In diesem Fall werden auch die Einträge aus dem *ObjectManager* entfernt. Falls kein Objekt zum übergebenen Objektidentifikator geladen werden konnte, so wird die Referenz auf das Nullobjekt zurückgeliefert.

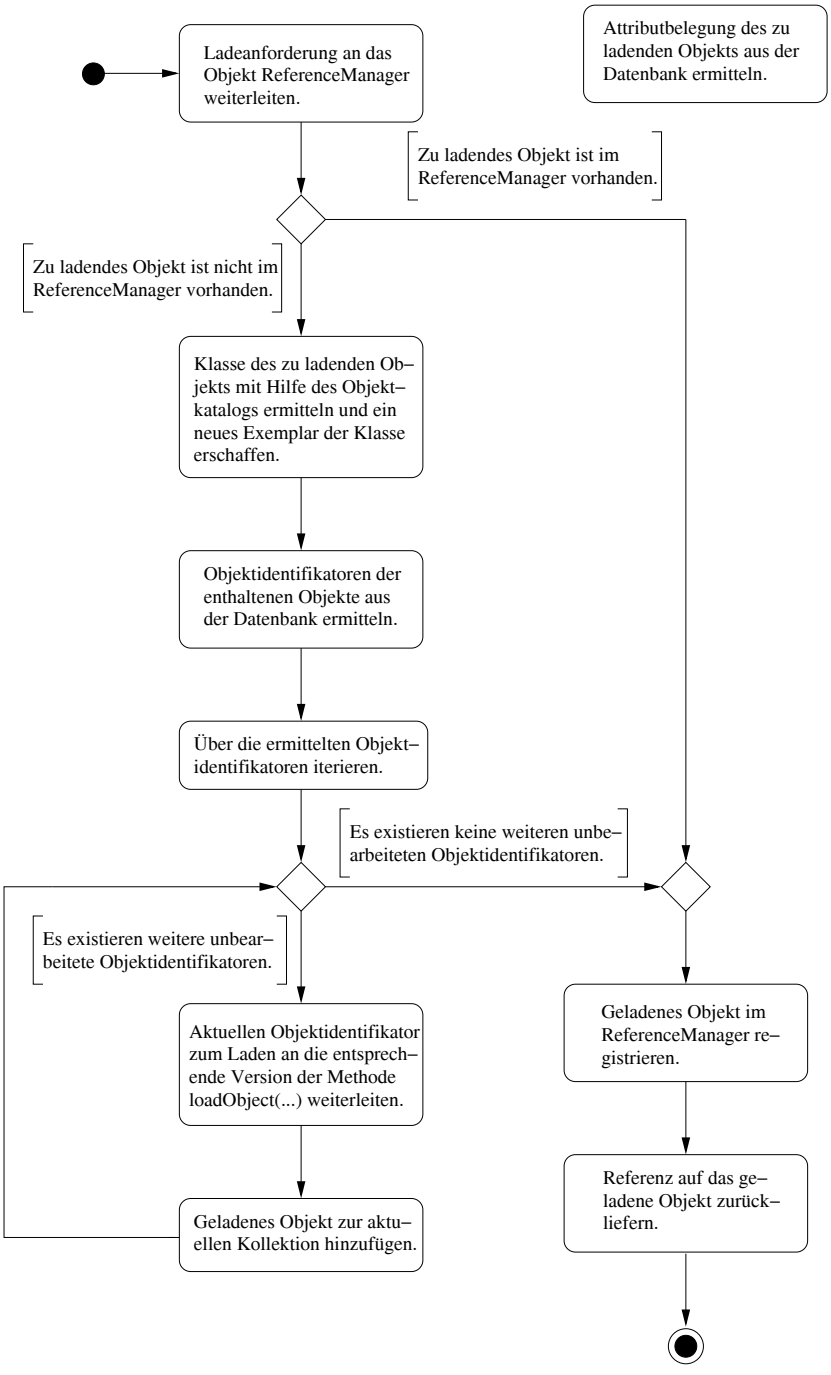

Abbildung 6.19: Das Aktivitätsdiagramm der Methode `loadObjectList(···)`

• Die Methode `deleteOID(···)` entfernt die Objektidentifikatoren aus dem als Parameter übergebenen und allen erreichbaren Objekten. Hierzu wird die Methode `OIDloeschen(···)` aufgerufen, die den Objektidentifikator des übergebenen Objekts entfernt und ebenfalls alle erreichbaren Referenztypen verarbeitet. Zur Verarbeitung von Objekten, die eine der Schnittstellen *PersistentListInterface*, *PersistentSetInterface* oder *PersistentMapInterface* realisieren, existieren überladene Versionen der Methode `OIDloeschen()`. Diese unterscheiden sich in ihrer Funktionsweise von der oben beschriebenen dadurch, dass sie die Methode `OIDloeschen(···)` für alle enthaltenen Referenztypen aufrufen.

6.2.5 Die Klassen zur Realisierung des Datenim- und exports unter Verwendung der XML

Die Klassen zur Realisierung der Objektpersistenz unter Verwendung der Extensible Markup Language (XML) [37] realisieren ebenfalls die in der Schnittstelle *PersistenceLayerInterface* definierte Funktionalität.

Die Abbildung der Objekte auf die XML-Dateien geschieht durch die in den Abschnitten 5.3.1.1, 5.3.1.2, 5.3.1.3 und 5.3.1.4 beschriebenen Konzepte. Das zu persistierende und alle von ihm aus erreichbaren Objekte werden hierbei in derselben Datei abgelegt bzw. aus derselben Datei geladen. Auch in diesem Fall wird also neben der in Abschnitt 5.1.1 beschriebenen Persistenz durch Vererbung noch die in Abschnitt 5.1.4 beschriebene Persistenz durch Erreichbarkeit realisiert. Einen Überblick über die an der Realisierung beteiligten Klassen gibt Abb. 6.20.

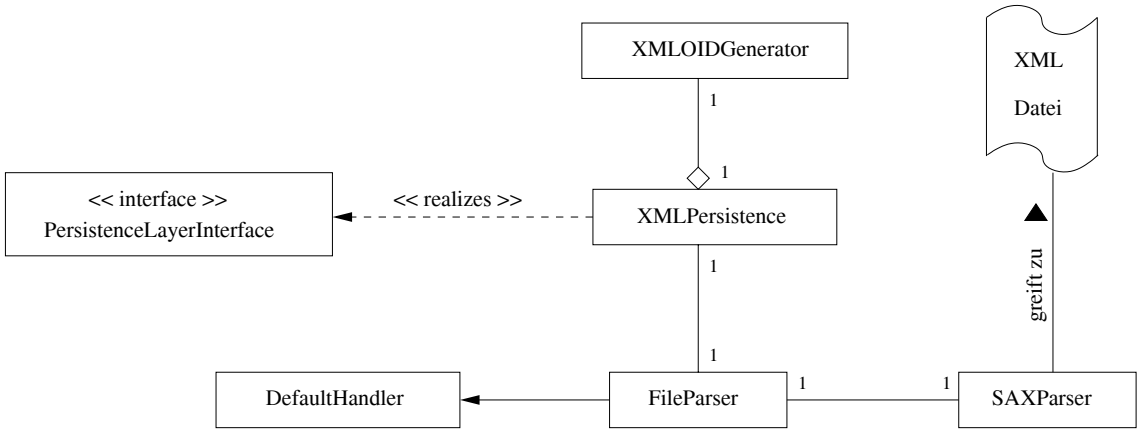

Abbildung 6.20: Die Klassen zur Realisierung der Objektpersistenz unter Verwendung der XML

In den folgenden Abschnitten werden die Zuständigkeiten der dargestellten Klassen beschrieben. Auf Implementierungsdetails wird hierbei nur soweit eingegangen, wie dies zum Verständnis der Funktionalität der jeweiligen Klassen notwendig ist.

6.2.5.1 Die Klasse XMLOIDGenerator

Die Klasse *XMLOIDGenerator* kapselt die Funktionalität zur Generierung von eindeutigen Objektidentifikatoren. Auf die notwendigen Objektidentifikatoren im Zusammenhang mit der Persistierung unter Verwendung der XML wurde bereits in Abschnitt 5.3.1.1 eingegangen.

Die Klasse *XMLOIDGenerator* stellt die Methode `getNewOID(···)` zur Verfügung, die einen neuen, bisher nicht verwendeten Objektidentifikator, erzeugt.

Die verwendeten Objektidentifikatoren bestehen aus Zeichenketten, die sich aus einem invarianten Bestandteil, der die Zeichenkette als Objektidentifikator kennzeichnet sowie aus einem Zahlenwert zusammensetzt. Dieser wird durch einen internen Zähler verwaltet, der vor der Persistierung eines Objekts den Wert Null hat. Nach jeder Anforderung eines Objektidentifikators wird der interne Zähler inkrementiert. Nachdem das Objekt und mit ihm alle erreichbaren Objekte persistiert wurden, wird der Zähler wieder auf den Wert Null zurückgesetzt. Wenn sich zu späteren Zeitpunkten die Beziehungen des in der Datei abgelegten Objekts geändert haben und

diese neuen Beziehungen ebenfalls abgespeichert werden sollen, so wird der Inhalt der Datei, basierend auf dem zu speichernden Objekt und seinen aktuellen Beziehungen, neu angelegt.

Die auf diese Weise ermittelten Objektidentifikatoren sind innerhalb der XML-Datei eindeutig. Diese Klasse wird im Zusammenhang mit der Speicherung von Objekten verwendet.

6.2.5.2 Die Klasse FileParser

Die Klasse *FileParser* stellt die Funktionalität zum Laden von Objekten aus den durch Exemplare der Klasse *XMLPersistence* erzeugten XML-Dateien bereit. Sie realisiert somit einen Teil der in der Schnittstelle *PersistenceLayerInterface* definierten Methodik, ohne jedoch die Schnittstelle als solche zu realisieren. Zum Parsen der XML-Dateien werden die in JAVA vorhandenen Klassen zur XML-Verarbeitung eingesetzt. Diese stellen verschiedene Arten von Parsern sowie die Methodik zur Ausnahmebehandlung bereit. Das Exemplar der Klasse *FileParser* ist ein Untertyp der Klasse *DefaultHandler*, einer JAVA-Basisklasse, die eine Schnittstelle zur Verarbeitung der vom Parser generierten Ereignisse bereitstellt. Solche Ereignisse werden z.B. durch öffnende bzw. schließende Tags oder durch Zeichenketten in der zu verarbeitenden XML-Datei erzeugt.

Das Laden der Objekte aus einer XML-Datei geschieht in zwei Schritten. Im ersten Schritt wird die XML-Datei verarbeitet. Der Parser liefert als erstes die Klasse des Objekts, mit der dann ein neues Exemplar erzeugt wird. Die nächsten vom Parser gelieferten Elemente sind die Namen der Attribute und die zugehörigen Werte, die direkt in dem anfänglich erzeugten Exemplar abgelegt werden. Bei Objektreferenzen wird anstelle des Attributwertes der Objektidentifikator geliefert, der in dem jeweils referenzierten Objekt abgelegt wird. Nachdem alle Attribute des Objekts durch die vom Parser gelieferten Werte besetzt wurden, wird das Objekt in einer assoziativen Datenstruktur abgelegt, die Schlüssel-Objekte und zugehörige Wert-Objekte aufnehmen kann. Hierbei bilden Objektidentifikator und Objekt das benötigte Schlüssel-Wert-Paar. Da das erste in der XML-Datei enthaltene Objekt die Wurzel des Objektgrafen und somit den Ausgangspunkt bei der Wiederherstellung der Objektreferenzen darstellt, wird die Referenz auf dieses Objekt gesondert abgespeichert. Diese Vorgehensweise wird in Abb. 6.21 dargestellt.

Nachdem alle in der XML-Datei enthaltenen Objekte auf die oben beschriebene Art und Weise geladen und in der assoziativen Datenstruktur abgelegt sind, werden im zweiten Schritt die Assoziationen zwischen den geladenen Objekten wieder hergestellt, die zur Zeit der Speicherung bestanden haben. Hierzu existieren verschiedene Versionen der Methode `generateObject(···)`, die die Attributlisten der Objekte bzw. die enthaltenen Objektidentifikatoren verarbeiten. Zu Beginn des zweiten Verarbeitungsschrittes wird die Methode `generateObject(···)` mit dem Wurzelobjekt als Parameter aufgerufen. Diese iteriert über die Attribute des Objekts. Bei allen Objektreferenzen wird das zugehörige Objekt mit Hilfe des Objektidentifikators aus der assoziativen Datenstruktur ermittelt. Bevor mit der Verarbeitung der Attribute des Wurzelobjekts fortgefahren wird, wird die beschriebene Vorgehensweise rekursiv auf das gerade ermittelte Objekt angewendet. Abb. 6.22 zeigt das Aktivitätsdiagramm dieses zweiten Verarbeitungsschrittes.

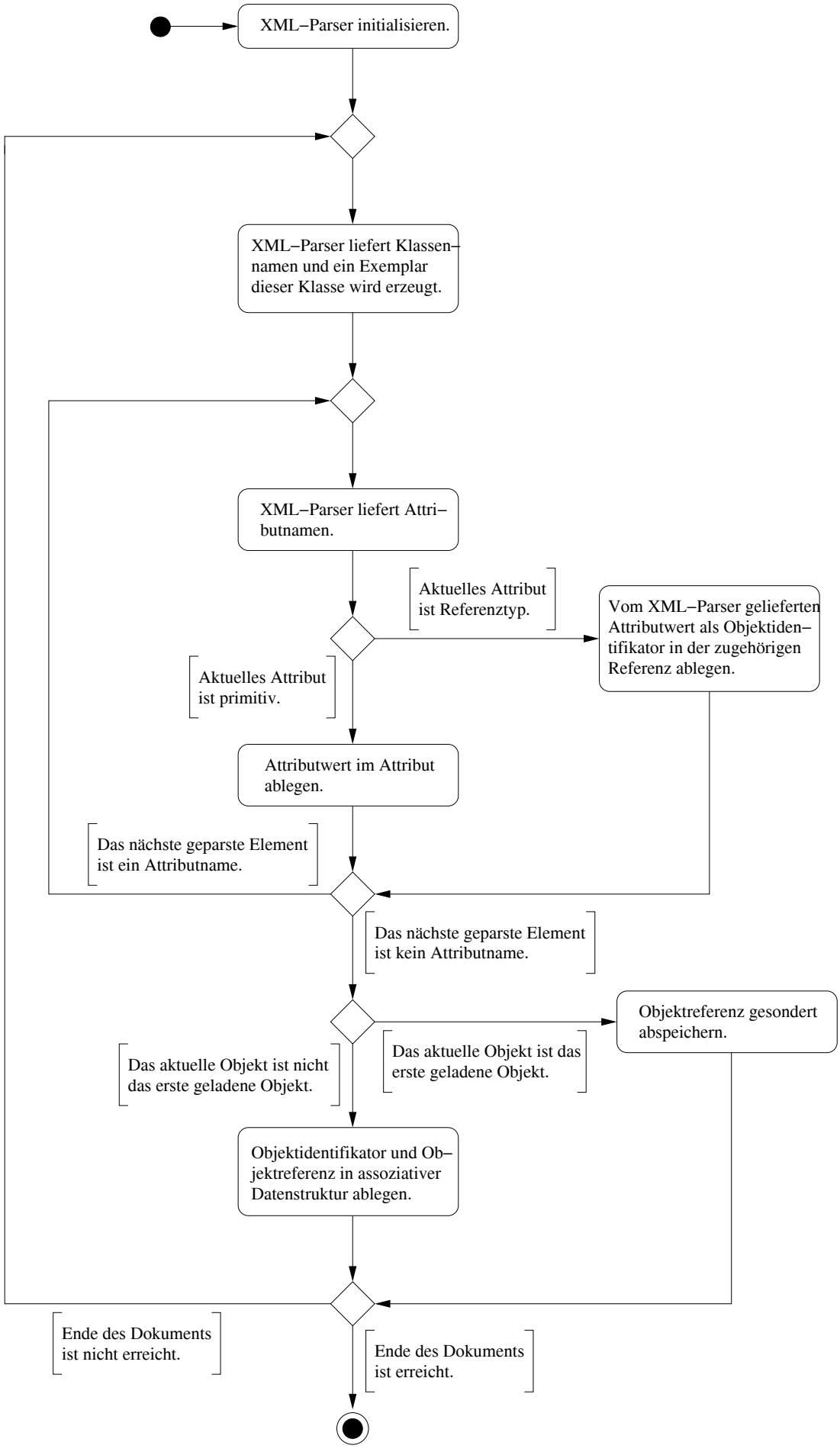

Abbildung 6.21: Die Vorgehensweise beim Parsen der XML-Datei

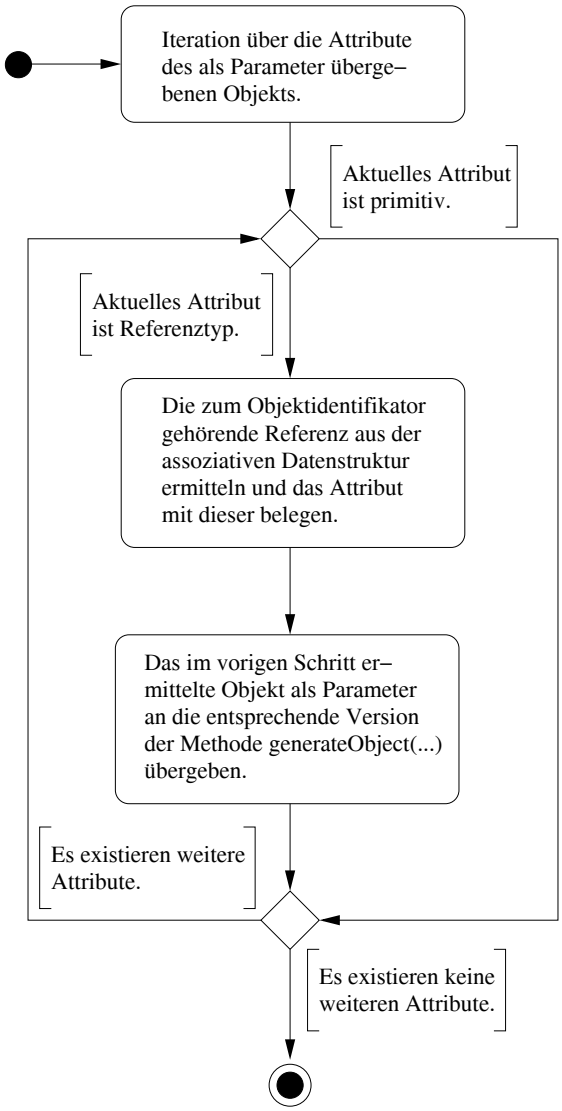

Abbildung 6.22: Die Methode `generateObject(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistenceInterface* realisieren

Die Versionen der Methode `generateObject(···)` zur Verarbeitung von Objekten, die eine der Schnittstellen *PersistentListInterface*, *PersistentSetInterface* oder *PersistentMapInterface* realisieren, unterscheiden sich von der in Abb. 6.22 dargestellten dadurch, dass nicht über die Attribute des Objekts, sondern über die enthaltenen Objektidentifikatoren iteriert wird.

Im Anschluss an das Parsen der XML-Datei und die Wiederherstellung der ursprünglichen Referenzen wird die Referenz auf das geladene Objekt den assoziierten Klassen zur Verfügung gestellt.

6.2.5.3 Die Klasse XMLPersistence

Die Klasse *XMLPersistence* realisiert die durch die Schnittstelle *PersistenceLayerInterface* definierte Funktionalität. Hierzu wird auf die Funktionalität der in vorangegangenen Abschnitten beschriebenen Klassen zurückgegriffen. Diese Klasse realisiert den Teil der in der Schnittstelle *PersistenceLayerInterface* definierten Methodik, der schreibende Zugriffe auf die XML-Dateien durchführt. Einen Überblick über die Methodik dieser Klasse liefert Abb. 6.23. Hier finden sich nicht nur die in der Schnittstelle *PersistenceLayerInterface* definierten Methoden, sondern auch weitere, zur Erbringung der Funktionalität notwendige.

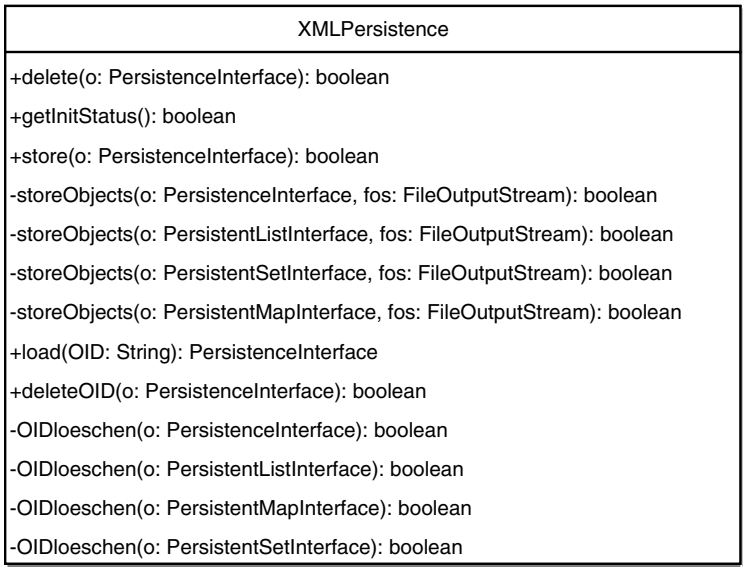

Abbildung 6.23: Die Funktionalität der Klasse *XMLPersistence*

Wie aus Abb. 6.20 ersichtlich ist, sind Exemplare dieser Klasse mit jeweils einem Exemplar der Klassen *FileParser* und *XMLOIDGenerator* assoziiert. Die Realisierung der in Abb. 6.23 dargestellten Methoden wird nun beschrieben.

• Die Methode `getInitStatus()` dient zur Prüfung, ob das Objekt fehlerfrei initialisiert werden konnte. Wenn dies der Fall ist, so konnten auch alle assoziierten Objekte fehlerfrei initialisiert werden und es wird der Wert *wahr* zurückgeliefert.

• Die Methode `delete(···)` entfernt die Datei, deren Bezeichnung als Objektidentifikator des Objekts angegeben ist vom Speichermedium, sofern diese vorhanden ist. Verläuft die Entfernung der Datei erfolgreich, so wird der Wert *wahr* zurückgeliefert.

• Die Methode `store(···)` persistiert das als Parameter übergebene Objekt. Der Name der XML-Datei, in die das Objekt persistiert werden soll, wird hierbei als Objektidentifikator des als Parameter übergebenen Objekts bereitgestellt. Von diesem Objekt ausgehend werden ebenfalls alle erreichbaren Objekte persistiert. Vor der Persistierung werden die Objektidentifikatoren aller erreichbaren Objekte entfernt und durch neue, von einem Exemplar der Klasse *XMLOIDGenerator* erzeugte, ersetzt. Hierdurch ist sichergestellt, dass zum Zeitpunkt der Persistierung alle erreichbaren Objekte mit einem Objektidentifikator versehen sind. Im Anschluss werden die Attributbezeichnungen und -werte der Objekte durch die verschiedenen Versionen der Methode `storeObjects(···)` in der XML-Datei abgelegt.

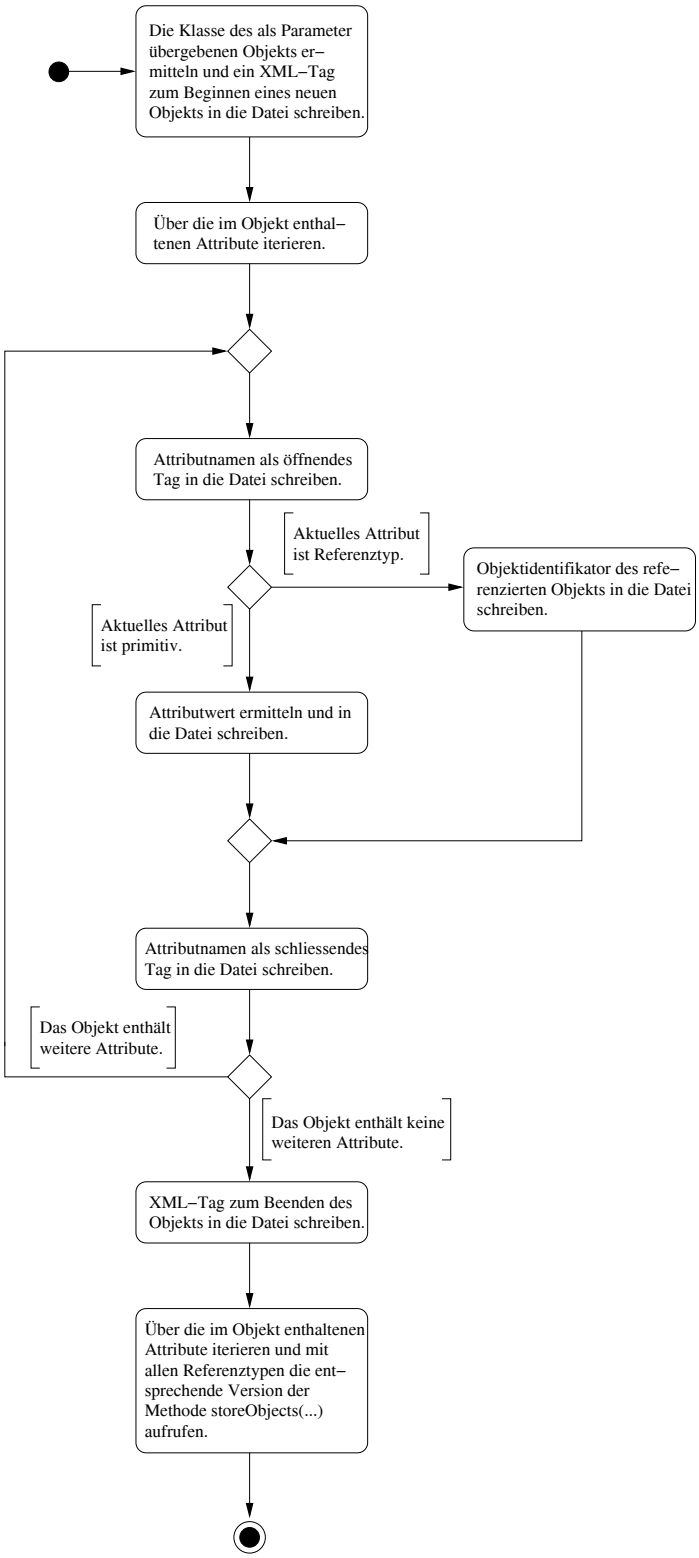

Abbildung 6.24: Die Methode `storeObjects(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistenceInterface* realisieren

Abb. 6.24 zeigt das Aktivitätsdiagramm der Methode `storeObjects(···)` zur Verarbeitung von Objekten, die die Schnittstelle *PersistenceInterface* realisieren. Die Methode ermittelt zuerst den Namen der Klasse des als Parameter übergebenen Objekts und legt ein entsprechendes Tag zum Beginnen eines neuen Objekts in der Datei ab. Hiernach wird über die Attribute des Objekts iteriert und es wird jeweils ein öffnendes Tag, das den Attributnamen enthält, in die Datei geschrieben. Wenn es sich bei dem Attribut um einen Referenztypen handelt, dann wird der Objektidentifikator des referenzierten Objekts als Attributwert in die Datei geschrieben. Andernfalls wird der Attributwert ermittelt und in die Datei geschrieben. Im Anschluss wird jeweils ein schließendes

Tag, das den Attributnamen enthält, in die Datei geschrieben. Nachdem alle Attribute auf diese Art und Weise in der Datei abgelegt wurden, wird erneut über die Attribute des Objekts iteriert. Bei allen Referenztypen wird die entsprechende Version der Methode `storeObjects(···)` mit dem jeweiligen Objekt als Parameter übergeben. Hierdurch werden alle vom ursprünglichen Objekt aus erreichbaren Objekte ebenfalls in der Datei abgelegt.

Die Versionen der Methode `storeObjects(···)` zur Verarbeitung von Objekten, die eine der Schnittstellen *PersistentListInterface*, *PersistentSetInterface* oder *PersistentMapInterface* realisieren, unterscheiden sich in ihrer Funktionsweise von der oben beschriebenen nur dadurch, dass sie eine Liste der Objektidentifikatoren der enthaltenen Objekte in der XML-Datei ablegen und danach rekursiv die Methode `storeObjects(···)` mit den enthalten Objekten als Parameter aufrufen.

Die Methode `store(···)` liefert als Ergebnis den Wert *wahr*, wenn während der Generierung der XML-Datei kein Fehler aufgetreten ist, andernfalls den Wert *falsch*.
• Die Methode `load(···)` erzeugt ein neues Exemplar der Klasse *FileParser*, das die XML-Datei, deren Name der Methode als Parameter übergeben wird, verarbeitet und aus dem Inhalt das zu ladende Objekt rekonstruiert. Die Art und Weise der Rekonstruktion wurde bereits in Abschnitt 6.2.5.2 beschrieben.
• Die Methode `deleteOID(···)` entfernt die Objektidentifikatoren aus dem als Parameter übergebenen und allen erreichbaren Objekten. Hierzu wird die Methode `OIDloeschen(···)` aufgerufen, die den Objektidentifikator des übergebenen Objekts und aller von diesem aus erreichbaren Objekten entfernt. Zur Verarbeitung von Objekten, die eine der Schnittstellen *PersistentListInterface*, *PersistentSetInterface* oder *PersistentMapInterface* realisieren, existieren überladene Versionen der Methode `OIDloeschen(···)`. Diese unterscheiden sich in ihrer Funktionsweise von der oben beschriebenen dadurch, dass sie die Methode `OIDloeschen(···)` für alle enthaltenen Referenztypen aufrufen.

6.3 Die Umsetzung der Algorithmen zur Netzausgleichung

In Abschnitt 6.1 wurde erläutert, dass die einzelnen Stufen des Datenflusses der geodätischen Deformationsanalyse in eigenen Komponenten gekapselt werden. Es wurde ebenfalls darauf eingegangen, dass alle Komponenten, die zu einer bestimmten Stufe des Datenflusses gehören, eine gemeinsame Schnittstelle haben, um sie unter Verwendung des Entwurfsmusters *Strategie* auf flexible Art und Weise in das System integrieren zu können. Dieser Abschnitt beschreibt die Umsetzung der ein- und zweidimensionalen Netzausgleichung in eigenen Komponenten und die gemeinsame Schnittstelle aller Komponenten zur Netzausgleichung. Weiterhin wird ein Ansatz zur Entkopplung der funktionalen Modelle von den zugehörigen Beobachtungstypen beschrieben, durch den sich, in Verbindung mit der in Abschnitt 4.2 beschriebenen Generalisierung von Beobachtungstypen, die Darstellung der Algorithmen zur Netzausgleichung vereinfachen lässt. Auf die verschiedenen Verfahren zur Ausgleichung geodätischer Netze wird nicht eingegangen. Diese werden z.B. in [24, 29, 40, 56, 57] beschrieben.

6.3.1 Integration der Algorithmen in das System

Die Integration der Algorithmen zur Netzausgleichung in das System findet unter Zuhilfenahme des Entwurfsmusters *Strategie* [14, 15] statt, da es sich aufgrund der in Abschnitt 3.3.1 gemachten Ausführungen anbietet. Hierbei werden die verschiedenen Komponenten mit einer Schnittstelle versehen, die den Zugriff durch andere Komponenten vereinheitlicht. Dieser Zusammenhang ist in Abb. 6.25 veranschaulicht.

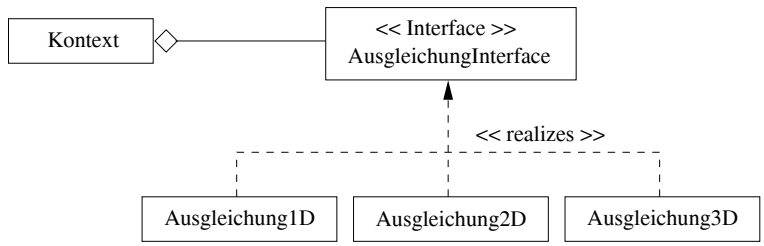

Abbildung 6.25: Integration der Algorithmen zur Netzausgleichung in das System unter Verwendung des Entwurfsmusters *Strategie*

Die durch die gemeinsame Schnittstelle der Komponenten zur Netzausgleichung definierte Methodik ist in Abb. 6.26 dargestellt. Die Schnittstelle *AusgleichungInterface* definiert die Methode `ausgleichung(···)`, der die Topologie des auszugleichenden Netzes als Parameter übergeben wird und mit der gleichzeitig die Ausgleichung gestartet wird.

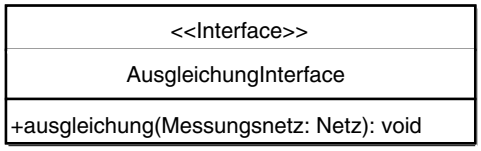

Abbildung 6.26: Die Schnittstelle *AusgleichungInterface*

Da die Schnittstelle *AusgleichungInterface* lediglich eine Methode zum Starten der Netzausgleichung definiert, wird die Verantwortung für die Ermittlung der zur Ausgleichung nötigen Parameter an die realisierenden Klassen delegiert. Da sich die Parameter nach den verschiedenen Dimensionen unterscheiden, wird darauf in den entsprechenden, zu den verschiedenen Ausgleichungen gehörenden Abschnitten eingegangen.

6.3.2 Der Ansatz zur Vereinheitlichung der Matrixbesetzung

Die Rechenverfahren zur Ausgleichung geodätischer Netze in verschiedenen Dimensionen unterscheiden sich auf Formelebene nicht voneinander. Lediglich die Art und Weise, wie die in den Formeln enthaltenen Matrizen und Vektoren besetzt werden, hängt von der Dimension und den zu betrachtenden Beobachtungen ab. Je nach Dimension der Ausgleichung nehmen nur bestimmte Beobachtungstypen an der Ausgleichung teil, wobei jede konkrete Beobachung zu einer oder zu mehreren Zeilen in den Matrizen und Vektoren korrespondiert. Die in diese Matrizen und Vektoren einzutragenden Werte werden hierbei durch beobachtungsspezifische Formeln errechnet. Die im Folgenden beschriebenen Maßnahmen führen zu einer Vereinheitlichung und Vereinfachung der Algorithmen bei einer gleichzeitigen Steigerung der Flexibilität, der Erweiterbarkeit und der Wartbarkeit.

Wie bereits beschrieben, korrespondiert jede Beobachtung zu einer oder zu mehreren Zeilen in den Matrizen und Vektoren, wobei die Anzahl der Zeilen vom Typ der Beobachtung abhängt. Um den Algorithmus zur Besetzung der Matrizen und Vektoren von den zu einer bestimmten Dimension gehörenden Beobachtungstypen unabhängig zu machen, erben alle Klassen, die zur Modellierung von Beobachtungstypen dienen, von einer gemeinsamen Oberklasse. Diese in Abschnitt 4.2 beschriebene Oberklasse trägt den Namen *Messungstyp* und stellt Methodik bereit, um z.B. die an einer Messung beteiligten Punkte, den Messwert und die zugehörige Varianz oder das verwendete Meßsystem zu ermitteln. Weiterhin wird auch Methodik bereitgestellt, mit der geprüft werden kann, ob ein Beobachtungstyp an der Ausgleichung einer bestimmten Dimension teilnehmen kann. Eine detaillierte Beschreibung der Klassen zur Modellierung der verschiedenen Beobachtungstypen findet sich in [53]. Da der Algorithmus zur Besetzung der Matrizen und Vektoren durch die beschriebene Modellierung weitestgehend unabhängig von konkreten Beobachtungstypen ist, können neue Beobachtungstypen mit nur wenigen Modifikationen des Algorithmus eingeführt werden. Es ist lediglich notwendig, dass solche neuen Beobachtungstypen von der Klasse *Messungstyp* erben. Durch die beschriebene Generalisierung der Beobachtungstypen und die daraus resultierende Vereinfachung der Algorithmen wird die Zielsetzung der Erweiterbarkeit des Systems auf der Ebene der Beobachtungen erfüllt. Eine solche Erweiterbarkeit ist bei keinem der in Abschnitt 2.2 vorgestellten geodätischen Programmsysteme auf eine so einfache Art und Weise möglich. Die Algorithmen zur Besetzung der Matrizen und Vektoren werden in den Abschnitten 6.3.3.3 und 6.3.4.2 beschrieben.

Betrachtet man die zu den jeweiligen Beobachtungen korrespondierenden Zeilen der Matrizen und Vektoren, so fällt auf, dass die einzutragenden Werte zwar durch beobachtungsspezifische Berechnungsvorschiften erzeugt werden, dass aber die jeweilige Art der Werte auf einer gewissen Abstraktionsebene ähnlich ist. Jede an einer Ausgleichung teilnehmende Beobachtung liefert, je nach Typ der Ausgleichung, Werte für den Stand- bzw. Ziel-

punkt, die Zusatzparameter, den Messwertvektor sowie den Absolutgliedvektor. Einige der genannten Werte werden unter Zuhilfenahme der beobachteten Größen berechnet. Obwohl die Berechnungsvorschriften beobachtungsspezifisch sind, lässt sich trotzdem eine gemeinsame Schnittstelle zum Zugriff auf diese finden. Die exakte Definition dieser Schnittstelle hängt von der Dimension der Ausgleichung ab und wird in den Abschnitten 6.3.3.2 und 6.3.4.2 beschrieben.

Durch die Möglichkeit, eine gemeinsame Schnittstelle zum Zugriff auf die Berechnungsvorschriften der zu einer bestimmten Dimension gehörenden Beobachtungstypen definieren zu können, ist es auch möglich, die jeweiligen Berechnungsvorschriften in einzelnen Klassen zu kapseln. Der Bezug eines Beobachtungstyps zu der Klasse, die die zugehörigen Berechnungsvorschriften kapselt, kann über ein Attribut des Beobachtungstyps hergestellt werden. Durch die Kapselung ist es möglich, neue Berechnungsvorschriften zu einzelnen Beobachtungstypen in das System zu integrieren, ohne die Beobachtungstypen modifizieren zu müssen. Weiterhin können die Berechnungsvorschriften für jede einzelne Beobachtung vor Beginn der Ausgleichung festgelegt werden.

Abb 6.27 zeigt die Trennung der Berechnungsvorschriften von den Beobachtungstypen und den Zugriff eines Algorithmus auf die gemeinsame Schnittstelle aller zu einer Dimension gehörenden Berechnungsvorschriften.

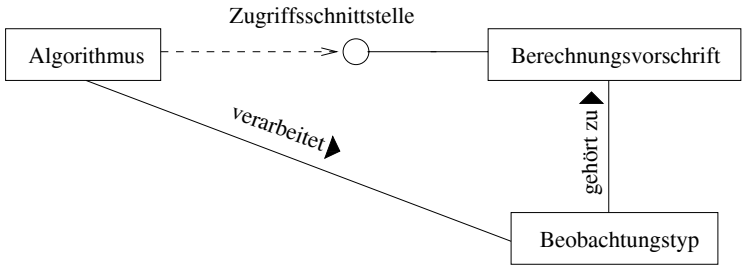

Abbildung 6.27: Zusammenhang zwischen Beobachtungstypen und zugehörigen Berechnungsvorschriften

Durch die Kapselung der funktionalen Modelle in eigenen Klassen und die damit verbundene Entkopplung von den Beobachtungstypen wird die Zielsetzung der Erweiterbarkeit des Systems auf der Ebene der funktionalen Modelle erfüllt. Auch dies ist bei den in Abschnitt 2.2 vorgestellten geodätischen Programmsystemen nicht ohne weiteres möglich.

Der Algorithmus zur Besetzung der Matrizen und Vektoren benötigt aufgrund der gemeinsamen Oberklasse aller Beobachtungstypen keine Kenntnis über die konkreten Beobachtungen, die zu den jeweiligen Zeilen und Einträgen führen. Während der Besetzung der Matrizen und Vektoren wird zu jeder Beobachtung ein Exemplar der Klasse erzeugt, das die entsprechenden Berechnungsvorschriften kapselt. Der Algorithmus greift dann über die gemeinsame Schnittstelle auf die Berechnungsvorschriften zu, um mit Hilfe der Beobachtung die notwendigen Werte zu berechnen und diese in die entsprechenden Matrizen und Vektoren einzutragen.

Die Vorteile, die sich durch die Verwendung des beschriebenen Ansatzes ergeben, werden nun noch einmal aufgeführt:

- Durch die Verwendung einer gemeinsamen Oberklasse, von der alle Beobachtungstypen erben und die Definition einer Schnittstelle zum Zugriff auf die Berechnungsvorschriften, kann die Verarbeitung der einzelnen, zu einer bestimmten Dimension gehörenden Beobachtungen vereinheitlicht werden. Es sind daher keine *if-then*-Anweisungsblöcke oder *switch*-Anweisungen nötig, um die Algorithmik zu den unterschiedlichen Beobachtungstypen aufzurufen. Hierdurch wird der Programmcode übersichtlicher und leichter wartbar.

- Durch die Verwendung einer gemeinsamen Oberklasse für alle Beobachtungstypen und durch die bereits beschriebene Vereinheitlichung der Verarbeitung können neue Beobachtungstypen mit nur geringfügigen Modifikationen der Algorithmik in das System integriert werden, solange sie von den notwendigen Oberklassen erben.

- Durch die Trennung von Beobachtungstypen und Berechnungsvorschriften in verschiedene Klassen ist es möglich, jede einzelne Beobachtung vor Beginn der Ausgleichung mit den gewünschten Berechnungsvorschriften zu parametrisieren. Weiterhin können neue Berechnungsvorschriften unter Realisierung der jeweiligen Schnittstellen auf einfache Art und Weise in das System integriert werden. Modifikationen bestehender Berechnungsvorschriften ziehen keinerlei Seiteneffekte hinsichtlich der Beobachtungen nach sich.

6.3.3 Eindimensionale Netzausgleichung

Dem Algorithmus zur eindimensionalen Netzausgleichung wird die Referenz auf die Topologiebeschreibung des auszugleichenden Netzes als Parameter übergeben. Aus dieser Topologiebeschreibung extrahiert der Algorithmus alle nötigen Daten, um die notwendigen Matrizen und Vektoren zur Netzausgleichung aufzubauen.

6.3.3.1 Vorbereitende Schritte

Vor der Durchführung der Ausgleichung müssen verschiedene Parameter vom Anwender in Erfahrung gebracht werden. Dies sind der Erdradius in Metern, die erste Näherung für den Refraktionskoeffizienten, der Gewichtseinheitsfehler sowie die Information, ob die Refraktionskoeffizienten an den Zenitdistanzen bzw. den Strecken geschätzt werden sollen. Weiterhin wird die Art der durchzuführenden Ausgleichung in Erfahrung gebracht. Hier stehen eine freie Netzausgleichung mit Gesamtspurminimierung, eine freie Netzausgleichung mit Teilspurminimierung, eine dynamische oder eine hierarchische Netzausgleichung zur Auswahl. Die Informationen werden vom Anwender in ein Dialogfenster eingegeben.

Bevor die zur Ausgleichung nötigen Matrizen und Vektoren besetzt werden können, müssen in einigen Vorverarbeitungsschritten die hierzu nötigen Informationen aus der Netztopologie extrahiert werden. Diese Vorverarbeitungschritte sind:

1. Mit Hilfe der Netztopologie werden verschiedene Listen gebildet, die die bei der Ausgleichung zu berücksichtigenden Punkte und Beobachtungen enthalten. Diese Listen sind im einzelnen:
   - Die Liste HA der hierarchischen Anschlusspunkte.
   - Die Liste DA der dynamischen Anschlusspunkte.
   - Die Liste M, der bei der Ausgleichung zu berücksichtigenden Beobachtungen. Beim Aufbau dieser Liste werden nur Beobachtungen betrachtet, die zur gewählten Dimension der Ausgleichung passen.
   - Die Liste AP der Punkte, die bei der Ausgleichung berücksichtigt werden. Diese Liste enthält je nach Art der Ausgleichung verschiedene Punkttypen:
     - Im Falle einer freien Netzausgleichung enthält sie alle Punkte, die an den zu berücksichtigenden Beobachtungen beteiligt sind.
     - Im Falle einer dynamischen Netzausgleichung enthält sie alle Neupunkte und alle dynamischen Punkte, die an den zu berücksichtigenden Beobachtungen beteiligt sind.
     - Im Falle einer hierarchischen Netzausgleichung enthält sie alle Neupunkte, die an den zu berücksichtigenden Beobachtungen beteiligt sind.
2. Basierend auf den zu berücksichtigenden Beobachtungen werden die Spaltenindizes der Zusatzparameter bestimmt. Diese Indizes sind während der gesamten Netzausgleichung invariant und werden von den verschiedenen Algorithmen bei der Bestimmung der Zeilen der Designmatrix verwendet.
3. Im Falle einer dynamischen Netzausgleichung wird eine Liste der Sessions, in denen die dynamischen Punkte enthalten sind, erstellt. Die dynamischen Punkte in der Liste DP werden entsprechend der Zugehörigkeit zu den Sessions und ihrem Rang innerhalb der jeweiligen Session umsortiert. Die Genauigkeitsinformationen von nicht vorhandenen Punkten werden ggf. aus den Sessions entfernt.
4. Im Falle einer freien Ausgleichung mit Teilspurminimierung wird die Ränderungsmatrix aufgebaut. Diese besteht aus einer Spalte und $card(\mathrm{AP})$ Zeilen. Die Elemente, deren Index mit denen der Datumspunkte in AP übereinstimmen, sind 1, alle anderen sind 0.

Nachdem die beschriebenen Schritte durchgeführt sind, kann mit der Besetzung der zur Ausgleichung notwendigen Matrizen und Vektoren begonnen werden.

6.3.3.2 Besetzung der Matrizen und Vektoren

Die generelle Vorgehensweise zur Besetzung der zur Ausgleichung notwendigen Matrizen und Vektoren wurde bereits in Abschnitt 6.3.2 beschrieben. An dieser Stelle wird auf die Schnittstelle zum Zugriff auf die Berechnungsvorschriften im eindimensionalen Fall eingegangen.

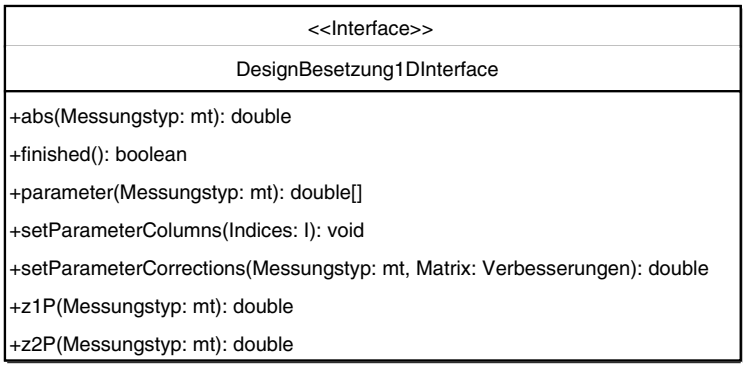

Abbildung 6.28: Die Schnittstelle *DesignBesetzung1DInterface*

Abb. 6.28 zeigt die Schnittstelle *DesignBesetzung1DInterface*, die zum Zugriff auf die Berechnungsvorschriften verwendet wird. Die Schnittstelle definiert folgende Methoden:

- Die Methode `setParameterColumns(···)` übergibt die während der Vorverarbeitung ermittelten Spaltenindizes der Zusatzparameter in der Designmatrix. Diese sind erforderlich, um die beobachtungsspezifischen Zusatzparameter zusammenzustellen und, um diese dann innerhalb eines Zahlenblocks zurückzuliefern. Daher ist es erforderlich, diese Methode vor allen anderen aufzurufen.
- Die Methode `z1P(···)` liefert den in die Designmatrix einzutragenden Wert des Standpunkts, der mit der übergebenen Beobachtung berechnet wurde. Die Methode `z2P(···)` arbeitet analog für den Zielpunkt der Beobachtung.
- Die Methode `abs(···)` liefert das zur übergebenen Beobachtung berechnete Absolutglied.
- Die Methode `parameter(···)` liefert die zur betrachteten Beobachtung gehörenden Werte der Zusatzparameter als Zahlenblock, der an die entsprechende Stelle der Designmatrix eingetragen wird.
- Mit der Methode `setParameterCorrections(···)` werden die im Zuge der Ausgleichung berechneten Verbesserungen an den Zusatzparametern der als Parameter übergebenen Beobachtung angebracht. Hierzu wird der Vektor der Verbesserungen ebenfalls als Parameter übergeben.
- Die Methode `finished()` dient zur Überprüfung, ob alle Werte zur aktuellen Beobachtung ermittelt wurden. Diese Methode wird benötigt, da es Beobachtungstypen gibt, die mehrere Zeilen in den Matrizen und Vektoren belegen.

Im folgenden Abschnitt wird der Algorithmus zur Durchführung der eindimensionalen Netzausgleichung beschrieben. In diesem Zusammenhang ist auch die Verwendung der in diesem Abschnitt beschriebenen Schnittstelle und die Umsetzung der in Abschnitt 6.3.2 vorgestellten Maßnahmen zu erkennen.

6.3.3.3 Algorithmus zur Durchführung der eindimensionalen Netzausgleichung

Der hier vorgestellte Algorithmus dient zur Durchführung der eindimensionalen Netzausgleichung. Er ist in einer eigenen Komponente gekapselt, die die in Abschnitt 6.3.1 beschriebene Schnittstelle realisiert. Der Algorithmus macht von dem in Abschnitt 6.3.2 vorgestellten Ansatz zur Kapselung der zu den Beobachtungen gehörenden Formeln der funktionalen Modelle unter Verwendung der in Abschnitt 6.3.3.2 vorgestellten Schnittstelle Gebrauch. Die Darstellung des Algorithmus erfolgt in Pseudocode, der nur die grundsätzlichen Verarbeitungsschritte wiedergibt. Der tatsächliche Programmcode ist selbstverständlich komplexer.

```
1   boolean Ende = false;
2   while (!Ende)
3     {
4       for (M_i ∈ M; i < card(M_i))
5         {
6           Exemplar Alg des in M_i benannten Algorithmus erzeugen;
7           Alg.setParameterColumns(Indizes);
8           while (!Alg.finished())
9            {
10             z_1 = Alg.x1P(M_i); z_2 = Alg.x2P(M_i);
11             params = Alg.parameter(M_i);
12             abs = Alg.abs(M_i);
13             if (! M_i.getP1() ∈ HP)
14              { A.set(i,AP.indexOf(M_i.getP1()),z_1);  }
15             if (! M_i.getP2() ∈ HP)
16              { A.set(i,AP.indexOf(M_i.getP2()),z_2);  }
17             A.set(i,Paramindex,params);
18             P.set(i,i,M_i.getVarianz()^-1 · σ_0^2);
19             a.set(i,0,abs);
20             l.set(i,0,M_i.getMesswert());
21           } // end while
22       } // end for
23       if (dynamische Ausgleichung)
24         {
25           Zeilen für dynamische Punkte in Designmatrix erzeugen;
26           Zeilen für dynamische Punkte in Kovarianzmatrix erzeugen;
27           Zeilen für dynamische Punkte im Messwertvektor erzeugen;
28           Zeilen für dynamische Punkte im Absolutgliedvektor erzeugen;
29         } // end if
30       P = Q_ll^-1
31       if (dynamische Ausgleichung) ∨ (hierarchische Ausgleichung)
32         { x̂ = (A^T PA)^-1 A^T Pl; }
33       if (freie Ausgleichung Gesamtspurminimierung)
34         { x̂ = (A^T PA)^+ A^T Pl; }
35       if (freie Ausgleichung Teilspurminimierung)
36         { x̂ = (A^T PA + GG^T)^-1 A^T Pl; }
37       Ende = checkCorrections(x̂);
38       if (!Ende)
39         {
40           Verbesserungen an den Punkten anbringen;
41           Verbesserungen an den Zusatzparametern der Beobachtungen anbringen;
42         } // end if
43     } // end while
44   v = Ax̂ − (l − a);
45   Verbesserungen und statistische Größen an den Beobachtungen anbringen;
```

• Zeile 4: Es wird über die Liste der bei der Ausgleichung zu betrachtenden Beobachtungen iteriert. Die in der Iteration jeweils aktuelle Beobachtung wird durch M_i bezeichnet.

• Zeile 6: Ein Exemplar der Klasse, die die zur aktuellen Beobachtung gehörenden Berechnungsvorschriften kapselt, wird basierend auf der in dem Beobachtungsobjekt abgelegten Zuordnung erzeugt.

• Zeile 7: Dem Algorithmusobjekt werden die Spaltenindizes der Zusatzparameter übergeben.

• Zeilen 10-12: Die in der Designmatrix einzutragenden Werte werden mit Hilfe der aktuell betrachteten Beobachtung M_i unter Verwendung der Schnittstelle *DesignBesetzung1DInterface*, wie in Abschnitt 6.3.2 beschrieben, berechnet.

• Zeilen 13-14 : Wenn es sich beim Standpunkt der Messung nicht um einen hierarchischen Punkt handelt, so wird der berechnete Wert in die Designmatrix eingetragen.

• Zeile 17: Die Zusatzparameter werden in die Designmatrix eingetragen.

• Zeile 18: Die Varianz wird in die Kovarianzmatrix eingetragen.

• Zeile 19: Der Wert des Absolutgliedes wird in den Absolutgliedvektor eingetragen.

• Zeile 20: Der Messwert wird in den Messwertvektor eingetragen.

• Zeile 25: Wenn es sich um eine dynamische Ausgleichung handelt, werden noch die Zeilen, in denen die dynamischen Anschlusspunkte als Beobachtungen eingeführt werden, in die Designmatrix eingefügt.

• Zeile 26: Die Genauigkeitsinformationen der dynamischen Punkte werden in die Kovarianzmatrix eingefügt.

• Zeile 27: Die Koordinaten der dynamischen Punkte werden in den Messwertvektor eingefügt.

• Zeile 28: Die Koordinaten der dynamischen Punkte werden in den Absolutgliedvektor eingefügt.

• Zeile 30: Die Gewichtsmatrix wird erzeugt.

• Zeilen 31-32: Wenn es sich um eine dynamische oder hierarchische Ausgleichung handelt, dann werden die Zuschläge zu den Punktkoordinaten berechnet durch: $\hat{x} = (A^T PA)^{-1} A^T Pl$.

• Zeilen 33-34: Wenn es sich um eine freie Ausgleichung mit Gesamtspurminimierung handelt, dann werden die Zuschläge zu den Punktkoordinaten berechnet durch: $\hat{x} = (A^T PA)^{+} A^T Pl$.

• Zeilen 35-36: Wenn es sich um eine freie Ausgleichung mit Teilspurminimierung handelt, dann werden die Zuschläge zu den Punktkoordinaten berechnet durch: $\hat{x} = (A^T PA + GG^T)^{-1} A^T Pl$. G ist hierbei die in der Vorverarbeitungsphase erzeugte Ränderungsmatrix.

• Zeile 37: Es wird geprüft, ob die Zuschläge zu den Punktkoordinaten dem Abbruchkriterium genügen. Hierzu müssen alle Verbesserungen kleiner 0,001 m sein.

• Zeile 38: Es wird über die Liste AP iteriert und an den jeweiligen Punkten die berechneten Zuschläge angebracht.

• Zeile 41: Es wird über die Liste der Beobachtungen iteriert und an den jeweiligen Zusatzparametern werden die berechneten Verbesserungen angebracht. Hierzu werden wieder, wie schon beim Besetzen der Designmatrix, Exemplare der Klassen zur Kapselung der Berechnungsvorschriften zu den entsprechenden Beobachtungen erzeugt und mit diesen die Verbesserungen an den Zusatzparametern angebracht.

• Zeile 44: Die Verbesserungen an den Beobachtungen werden berechnet durch:
$v = A\hat{x} - (l - a)$.

• Zeile 45: Es wird über die Liste der Beobachtungen iteriert und an den jeweiligen Beobachtungen werden die berechneten Verbesserungen angebracht.

6.3.4 Zweidimensionale Netzausgleichung

Wie im eindimensionalen Fall wird auch dem Algorithmus zur zweidimensionalen Netzausgleichung die Referenz auf die Topologiebeschreibung des auszugleichenden Netzes als Parameter übergeben, damit dieser alle nötigen Daten ermitteln kann. Auch dieser Algorithmus ist in einer eigenen Komponente realisiert und implementiert die in Abschnitt 6.3.1 beschriebene Schnittstelle.

6.3.4.1 Vorbereitende Schritte

Vor der Durchführung der Ausgleichung werden wiederum einige Parameter vom Anwender mittels eines Dialogfensters in Erfahrung gebracht. Es handelt sich hierbei um den Gewichtseinheitsfehler sowie die Art der durchzuführenden Netzausgleichung. Nachdem alle Parameter zur Durchführung bekannt sind, werden ebenfalls die in Abschnitt 6.3.3.1 beschriebenen Vorverarbeitungsschritte durchgeführt.

6.3.4.2 Besetzung der Matrizen und Vektoren

Die Besetzung der Designmatrix geschieht im Wesentlichen wie im eindimensionalen Fall, also durch die in Abschnitt 6.3.2 beschriebene Vorgehensweise. Die Schnittstelle zum Zugriff auf die Berechnungsvorschriften der Beobachtungstypen, die an einer zweidimensionalen Netzausgleichung teilnehmen können, ist in Abb. 6.29 dargestellt.

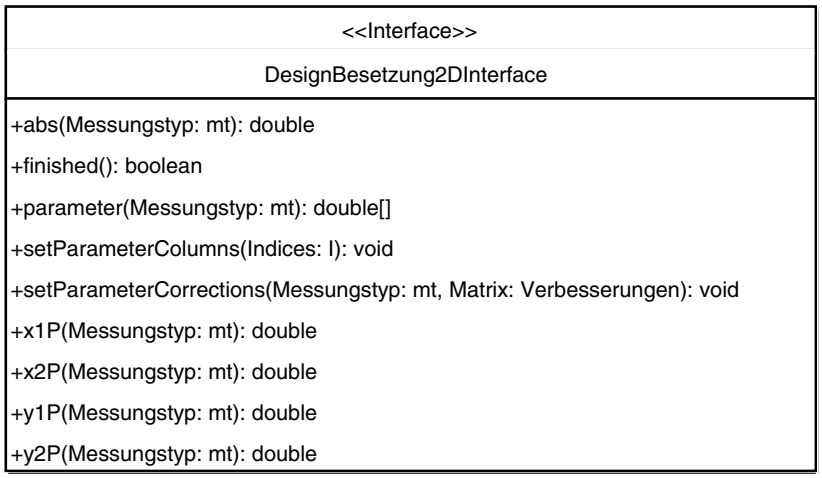

Abbildung 6.29: Die Schnittstelle *DesignBesetzung2DInterface*

Da der größte Teil der in der Schnittstelle *DesignBesetzung2DInterface* definierten Methoden bereits in Abschnitt 6.3.3.2 beschrieben wurde, wird nun nur noch auf die verbleibenden Methoden eingegangen. Die Methoden `x1P(···)` und `y1P(···)` arbeiten analog der in Abschnitt 6.3.3.2 beschriebenen Methode `z1P(···)`. Anstelle der Höhenkomponente liefert diese Schnittstelle die Lagekomponenten der an der Beobachtung beteiligten Punkte. Die Methoden `x2P(···)` und `y2P(···)` arbeiten analog der Methode `z2P(···)`.

Der Algorithmus zur Durchführung der zweidimensionalen Netzausgleichung ist im Wesentlichen analog zu dem in Abschnitt 6.3.3.3 beschriebenen. Anstelle der in den Zeilen 14 und 16 berechneten Höhenkomponenten werden im zweidimensionalen Fall die Lagekomponenten der jeweiligen Beobachtung in die Designmatrix eingetragen.

6.4 Umsetzung der Algorithmen zur Deformationsanalyse

Wie in Abschnitt 6.1 beschrieben werden auch die verschiedenen Ansätze zur geodätischen Deformationsanalyse in eigenen Komponenten realisiert und mit einer gemeinsamen Schnittstelle versehen. Dieser Abschnitt beschreibt diese gemeinsame Schnittstelle und ihre Verwendung bei der Integration der Komponenten in das System. Im Anschluss wird die Umsetzung von drei verschiedenen Ansätzen zur statischen Deformationsanalyse und ihre Integration in das System beschrieben. Auf die unterschiedlichen Ansätze zur geodätischen Deformationsanalyse wird in diesem Zusammenhang nicht weiter eingegangen. Diese werden z.B. in [46, 47, 49, 58] beschrieben.

6.4.1 Integration der Algorithmen in das System

Die Integration der Algorithmen zur Deformationanalyse geschieht, aufgrund der in Abschnitt 3.3.1 gegebenen Beschreibung, ebenfalls unter Verwendung des Entwurfsmusters *Strategie* [14, 15]. Die verschiedenen Algorithmen zur geodätischen Deformationsanalyse wurden mit einer einheitlichen Schnittstelle versehen, die dem aufrufenden Kontext die Übergabe der Parameter und den Start des Algorithmus ermöglichen. Die Umsetzung des Entwurfsmusters zeigt Abb. 6.30.

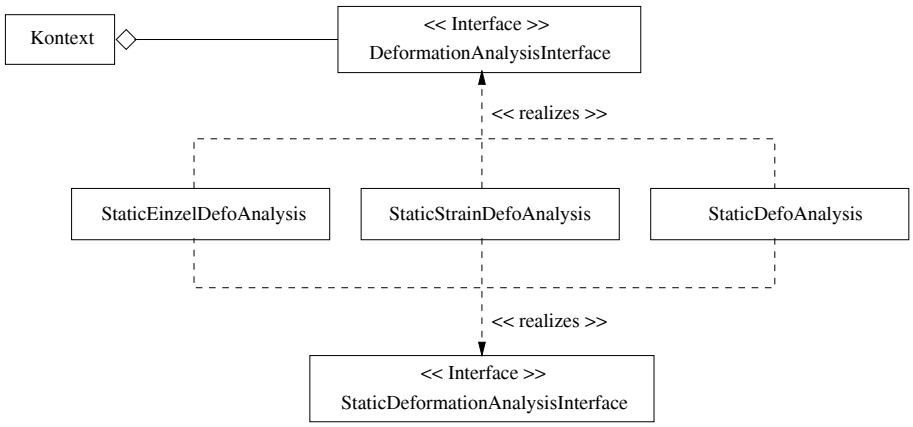

Abbildung 6.30: Integration der Algorithmen zur Deformationsanalyse in das System

Die in Abb. 6.30 dargestellte Schnittstelle *DeformationAnalysisInterface* definiert zwei Methoden, die von den Komponenten, die Algorithmik zur Deformationsanalyse implementieren, zu realisieren ist. Die Methode `setPunktnetze(···)` dient zur Übergabe der Punktmengen, mit denen die Analyse durchgeführt werden soll. Im Falle einer statischen Deformationsanalyse werden nur die ersten zwei der als Parameter übergebenen Punktmengen betrachtet. Die Methode `startAnalysis(···)` dient zum Starten der eigentlichen Analyse. Diese umfasst neben der eigentlichen Deformationsanalyse auch die Ermittlung von Parametern mittels Dialogfenstern sowie die Aufbereitung der Ergebnisse. Die Reihenfolge der Methodenaufrufe ergibt sich aus der obigen Erläuterung der Funktionsweise.

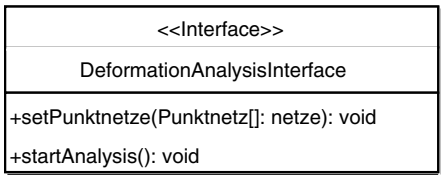

Abbildung 6.31: Die Schnittstelle *DeformationAnalysisInterface*

Die in Abb. 6.30 dargestellte Schnittstelle *StaticDeformationAnalysisInterface* definiert keine Methodik. Sie dient lediglich zur Markierung der Algorithmen und deren Unterteilung in disjunkte Mengen. Von dieser Markierung können Dialogfenster Gebrauch machen, um nur statische Algorithmen zur Auswahl anzubieten. Bei einer Erweiterung des Systems um neue Deformationsanalysen, die auch kinematisch oder dynamisch sein können, können weitere markierende Schnittstellen eingeführt werden.

Die Klassen *StaticDefoAnalysis*, *StaticEinzelDefoAnalysis* und *StaticStrainDefoAnalysis* realisieren verschiedene statische Deformationsanalysen. Die zugehörigen Algorithmen werden in den Abschnitten 6.4.2, 6.4.3 und 6.4.4 beschrieben.

Durch die Verwendung des Entwurfsmusters *Strategie* [14, 15] bei der Integration der Komponenten ist das System flexibel hinsichtlich der Auswahl der bereits vorhandenen Algorithmen. Durch die einheitliche Schnittstelle kann der im aufrufenden Kontext notwendige Programmcode vereinheitlicht werden. Die Erweiterbarkeit des Systems zu späteren Zeitpunkten ist ebenfalls sichergestellt, da neue Algorithmik unter Realisierung der Schnittstelle *DeformationanalysisInterface* sowie der Schnittstelle *StaticDeformationAnalysisInterface* auf einfache Art und Weise hinzugefügt werden kann.

Der Aufruf eines Algorithmus zur Durchführung einer Deformationsanalyse erfordert vom Benutzer eine Reihe von Angaben. Diese können unterteilt werden in Angaben, die zur Erschaffung eines Exemplars einer Algorithmusklasse benötigt werden und Parametern, die zur Durchführung eines konkreten Algorithmus benötigt werden. Die Angaben zur Erschaffung eines Exemplars einer Algorithmusklasse sind von denen zur Durchführung unabhängig und erfolgen immer in derselben Reihenfolge:

1. Auswahl, welche Art von Deformationsanalyse durchgeführt werden soll. Diese Auswahl hat Einfluss auf das Dialogfenster zur Angabe der zu analysierenden Punktmengen und auf das Dialogfenster zur Auswahl der Algorithmen.

2. Auswahl des auszuführenden Algorithmus. Je nachdem, welche Art von Deformationsanalyse durchgeführt werden soll, werden unter Verwendung der in Abb. 6.30 dargestellten, markierenden Schnittstelle solche zur Auswahl angeboten.

3. Auswahl der Punktmengen, mit denen die gewählte Analyse durchgeführt werden soll. Im statischen Fall können zwei Punktmengen ausgewählt werden und bei anderen Arten der Deformationsanalyse mehrere.

Mit Hilfe dieser Angaben wird ein Exemplar der ausgewählten Algorithmusklasse erschaffen. Die Ermittlung algorithmenspezifischer Parameter wird an die Algorithmusklassen delegiert, die dies eigenständig durchführen. Die folgenden Abschnitte beschreiben die Algorithmen zur statischen Deformationsanalyse. Es wird für jeden beschriebenen Algorithmus zuerst eine allgemeine Beschreibung der Funktionsweise gegeben. Im Anschluss wird der Algorithmus in Pseudocode dargestellt und es wird auf einzelne Schritte eingegangen. In diesem Zusammenhang sei noch einmal erwähnt, dass der vorgestellte Pseudocode nur grundsätzliche Verarbeitungsschritte wiedergibt. Der tatsächliche Programmcode ist natürlich komplexer.

6.4.2 Deformationsanalyse mit stufenweiser Stützpunktsuche

6.4.2.1 Allgemeine Beschreibung des Algorithmus

Dieser Algorithmus basiert auf einem Testverfahren [26, 42, 43], mit dem eine Menge von Punkten gemeinsam auf Stabilität getestet werden kann. Von der zu testenden Punktmenge werden Untermengen gebildet, die hinsichtlich ihrer Kardinalität absteigend sortiert sind. Diese Untermengen werden dann nacheinander auf Stabilität getestet. Die erste Untermenge, die den Stabilitätstest erfüllt, ist die gesuchte Menge von Stabilpunkten.

Vor Beginn der eigentlichen Deformationsanalyse werden Vorverarbeitungsschritte durchgeführt, um die zu den Epochen gehörenden Punktmengen so zu reduzieren, dass in beiden Mengen nur noch die in beiden Epochen gemessenen Punkte enthalten sind. Hierbei werden auch die zu den entfernten Punkten gehörenden Zeilen und Spalten aus den jeweiligen Varianz-Kovarianz-Matrizen entfernt. Im Anschluss werden die Punkte in der zur Folgeepoche gehörenden Menge in die Reihenfolge gebracht, die diese auch in der Punktmenge der Bezugsepoche haben. Hierbei werden auch die Zeilen und Spalten der zugehörigen Varianz-Kovarianz-Matrix umsortiert.

Bei der Durchführung der Deformationsanalyse werden in einem ersten Schritt alle Punkte gemeinsam auf Stabilität getestet. Versagt dieser Test, so wird, ausgehend von der Grundmenge M_0, die alle Punkte enthält, eine Untermenge von stabilen Punkten gesucht, die hinsichtlich ihrer Kardinalität maximal ist.

Hierzu werden, ausgehend von der ursprünglichen Punktmenge, nacheinander verschiedene Punktkombinationen entfernt. Die zu entfernenden Punktkombinationen sind hierbei alle Untermengen $M^i_{M_0}$ von M_0, wobei gilt: $0 < card(M^i_{M_0}) < card(M_0) - D + 1$. D ist hierbei die Dimension der Analyse. Die $M^i_{M_0}$ werden aufsteigend nach ihrer Kardinalität geordnet und es wird bei der Entfernung mit dem $M^i_{M_0}$ begonnen, dessen Kardinalität minimal ist. In die nach der Entfernung verbleibenden Punkte wird mittels einer Schwerpunkttransformation das Datum gelegt, wonach sie gemeinsam auf Stabilität getestet werden. Die erste Punktmenge, bei der dieser Stabilitätstest nicht versagt, ist die hinsichtlich der Kardinalität maximale Menge an Stabilpunkten und somit die gesuchte Punktmenge.

6.4.2.2 Algorithmus zur stufenweisen Stützpunktsuche

```
1   P_0 = Bezugsepoche
2   P_1 = Folgeepoche
3   M_0 = Menge der Punkte der Bezugsepoche
4   M_1 = Menge der Punkte der Folgeepoche
5   if ((P_0.getAPrioriVarianz() == P_1.getAPrioriVarianz()) ∧
      (P_0.getRangdefekt() == P_1.getRangdefekt()) ∧
      (P_0.getKoordinatensystem() == P_1.getKoordinatensystem()) ∧
      (P_0.getDatumspunkte() == P_1.getDatumspunkte()))
6       {
7         M_S = M_0 ∩ M_1; M'_0 = M_0 ∩ M_S; M'_1 = M_1 ∩ M_S;
8         Datum in die in M'_0 bzw. M'_1 enthaltenen Punkte legen;
9         Punkte in M_1 in die Reihenfolge von M_0 bringen;
10        Blockdiagonalmatrix aufstellen;
11        Differenzenvektor aufstellen;
12        B-Matrix aufstellen;
13        Testgröße T berechnen;
14        if (T < F_{b,∞}(α)) { Alle Punkte sind stabil; }
15        else
16          {
17            M_{M_0} = {M^j_{M_0} | M^j_{M_0} ⊂ M_0 ∧ M^j_{M_0} ≠ ∅ ∧ card(M^j_{M_0}) < card(M_0) − D + 1};
18            stabile = false;
19            for (M^j_{M_0} ∈ M_{M_0}; j < card(M_{M_0}))
20              {
21                M'_0 = M_0/M^j_{M_0}; M'_1 = M_1/M^j_{M_0};
22                Datum in die in M'_0 enthaltenen Punkte legen;
23                Blockdiagonalmatrix aufstellen;
24                Differenzenvektor aufstellen;
25                B-Matrix aufstellen;
26                Testgrößen T und T' berechnen;
27                if ((T < F_{b,∞}(α)) ∧ (T' < F_{b,∞}(α)))
28                  { M'_0 enthält die stabilen Punkte; stabile = true; break; }
29              } // end for
30            if (stabile == true) { M'_0 ausgeben; }
31            else { Es wurden keine stabilen Punkte gefunden; }
32          } // end else
33      } // end if
```

• Zeile 1-4: Zu Beginn werden die in den einzelnen Epochen gemessenen Punkte zur Verarbeitung in Mengen zusammengefasst. Hierbei kann auch eine Vorannahme über bereits als stabil bekannte Punkte eingeführt werden, indem in der Menge M_0 nur diese Punkte erfasst werden.

• Zeile 5: Bevor die Analyse durchgeführt werden kann, muss sichergestellt sein, dass die a-priori-Varianzen, die Rangdefekte, die Koordinatensysteme und die Datumspunkte der beiden Epochen gleich sind.

• Zeile 7: Durch diese Berechnungen enthalten M_0 und M_1 nur noch die in beiden Epochen gemessenen Punkte. Hierbei müssen auch die zu den Epochen gehörenden Varianz-Kovarianz-Matrizen entsprechend modifiziert werden.

• Zeile 8: Das Datum wird in die in M_0 und M_1 enthaltenen Punkte gelegt. Hierzu werden die Matrizen C_{xx_i} transformiert durch $C_{xx_i} = S_i^* C_{xx_i} S_i^{*T}$, wobei die C_{xx_0} die Varianz-Kovarianz-Matrix der Bezugsepoche ist und die C_{xx_1} die der Folgeepoche. Die S_i^* werden durch $S_i^* = I - G_i^*(G_i^{*T} E_i G_i^*)^{-1} G_i^{*T} E_i$ gebildet. Die E_i sind hierbei Matrizen, die an den Stellen auf der Hauptdiagonalen, die den in M_0 bzw. M_1 enthaltenen Punkten entsprechen, mit 1 besetzt sind und ansonsten mit 0. Die Matrizen G_i^* werden aus Submatrizen aufgebaut, die im zweidimensionalen Fall die Form

$$G^j = \begin{bmatrix} 1 & 0 & -y \\ 0 & 1 & +x \end{bmatrix}$$

und im dreidimensionalen Fall die Form

$$G^j = \begin{bmatrix} 1 & 0 & 0 & 0 & +z & -y \\ 0 & 1 & 0 & -z & 0 & +x \\ 0 & 0 & 1 & +y & -x & 0 \end{bmatrix}$$

haben. Der Punkt aus P_i, zu dem der jeweilige Block gehört, wird durch j angegeben.

• Zeile 9: Die in M_1 enthaltenen Punkte werden in die gleiche Reihenfolge gebracht, die diese auch in M_0 haben. Hierbei müssen auch die Zeilen und Spalten der zugehörigen Varianz-Kovarianz-Matrix entsprechend umsortiert werden.

• Zeile 10: Die Blockdiagonalmatrix hat die Form $C_{xx} = \begin{pmatrix} C_{xx_0} & 0 \\ 0 & C_{xx_1} \end{pmatrix}$.

• Zeile 11: Der Differenzenvektor hat die Form
$\bar{x} = \begin{bmatrix} x_1^0 & y_1^0 & z_1^0 & \cdots & x_n^0 & y_n^0 & z_n^0 & x_1^1 & y_1^1 & z_1^1 & \cdots & x_n^1 & y_n^1 & z_n^1 \end{bmatrix}^T$, wobei die x_j^0, y_j^0, z_j^0 die Koordinaten der Punkte der Bezugsepoche sind und die x_j^1, y_j^1, z_j^1 die der Folgeepoche.

• Zeile 12: Die B-Matrix hat die Form $B = \begin{pmatrix} -I & I \end{pmatrix}$. Die Matrizen $-I$ bzw. I sind Einheitsmatrizen der Dimension $(D \cdot card(M_0) \times D \cdot card(M_0))$, wobei D die Dimension der Deformationsanalyse ist.

• Zeile 13: Die Testgröße T berechnet sich durch $T = \frac{R/b}{\sigma_0^2}$.
Hierbei ist $R = \sigma_0^2 (Bx)^T (BC_{xx}B^T)^+ (Bx)$ und $b = D \cdot s - d - M$, mit

D	=	Netzdimension
s	=	Stabilpunktanzahl im aktuellen Test
d	=	Netzdefekt
M	=	Anzahl der Epochenmaßstäbe bezüglich der Referenzepoche.

• Zeile 14: Die Testgröße wird gegen das Fraktil der Fisher-Verteilung mit den Freiheitsgraden b und ∞ bei vorgegebener Irrtumswahrscheinlichkeit α getestet. Bei bestandenem Test sind alle Punkte als stabil zu betrachten.

• Zeile 17: Die Menge M_{M_0} enthält alle nichtleeren Mengen von Punkten $M_{M_0}^j$, die sich aus M_0 bilden lassen und für die gilt: $card(M_{M_0}^j) < card(M_0)$-$D$+1. D ist hierbei die Dimension der Analyse. M_{M_0} ist eine nach der Kardinalität der enthaltenen Mengen aufsteigend sortierte Menge.

• Zeile 19: Iteration über alle in M_{M_0} enthaltenen Mengen. $M^j_{M_0}$ bezeichnet die, im aktuellen Iterationsschritt betrachtete Menge aus M_{M_0}.

• Zeile 22: Das Datum wird mittels einer Schwerpunktstransformation in die in M'_0 bzw. M'_1 enthaltenen Punkte gelegt. Hierzu werden die Matrizen $S_i = I - G_i(G_i^T E G_i)^{-1} G_i^T E$, gebildet. E ist hierbei eine Matrix, die an den Stellen auf der Hauptdiagonalen, die den in M'_0 bzw. M'_1 enthaltenen Punkten entsprechen, mit 1 besetzt ist und ansonsten mit 0. Die Matrizen G_i werden aus Submatrizen aufgebaut, die dieselbe Form wie die in Zeile 8 verwendeten haben. Der Punkt aus M_i, zu dem der jeweilige Block G^j gehört, wird durch j angegeben. Mit Hilfe der S_i werden die C'_{xx_i} berechnet zu $C'_{xx_i} = S_i \cdot C_{xx_i} \cdot S_i^T$. Durch die Schwerpunktstransformation wird der Defekt in die in M'_0 enthaltenen Punkte gelegt. Nach der Transformation werden die Zeilen und Spalten der in $M^j_{M_0}$ enthaltenen Punkte aus den C'_{xx_i} entfernt.

• Zeile 23: Die Blockdiagonalmatrix C'_{xx} wird, wie in der Beschreibung zu Zeile 9, mit den Matrizen C'_{xx_0} und C'_{xx_1} aufgestellt.

• Zeile 24: Der Differenzenvektor wird, wie in der Beschreibung zu Zeile 10, mit den in M'_0 und M'_1 enthaltenen Punkten aufgestellt.

• Zeile 25: Die B-Matrix wird, wie in der Beschreibung zu Zeile 11, aufgestellt. Sie hat die Dimension $(D \cdot card(M'_0) \times D \cdot card(M'_0))$

• Zeile 26: Die Testgrößen T und T' werden, wie in Zeile 12 beschrieben, berechnet. Bei der Berechnung des T' wird anstatt der vollständigen C'_{xx} eine Matrix verwendet, die nur die Hauptdiagonale der C'_{xx} enthält.

• Zeile 27: Die Testgrößen werden gegen das Fraktil der Fisher-Verteilung mit den Freiheitsgraden b und ∞ bei vorgegebener Irrtumswahrscheinlichkeit α getestet. Bei bestandenen Tests sind die in M'_0 enthaltenen Punkte als stabil zu betrachten und die Iteration kann beendet werden, da M'_0 hinsichtlich der Kardinalität maximal ist.

6.4.3 Stützpunktsuche mit Strainanalyse

6.4.3.1 Allgemeine Beschreibung des Algorithmus

Dieser Algorithmus [20, 38, 55] basiert darauf, die auf Stabilität zu untersuchende Punktmenge in Dreiecke zu zerlegen, die dann einzeln auf Ähnlichkeit und Kongruenz getestet werden. Dieser Algorithmus ist daher nicht anwendbar, wenn die Dimension kleiner als zwei ist. Die Dreiecke müssen hierbei nicht notwendigerweise aus einem Punkt und seinen beiden nächsten Nachbarn gebildet werden. Für Dreiecke, bei denen der Ähnlichkeits- oder der Kongruenztest versagt, wird eine Strainanalyse durchgeführt. Sind beide Tests erfolgreich, so wird das Dreieck in die Liste der kongruenten Dreiecke aufgenommen. Nachdem alle Dreiecke getestet wurden, werden aus der Liste der kongruenten Dreiecke zusammenhängende, kongruente Teilnetze gebildet.

Vor Beginn der eigentlichen Deformationsanalyse werden Vorverarbeitungsschritte durchgeführt, um die zu den Epochen gehörenden Punktmengen so zu reduzieren, dass in beiden Mengen nur noch die in beiden Epochen gemessenen Punkte enthalten sind. Hierbei werden auch die zu den entfernten Punkten gehörenden Zeilen und Spalten aus den jeweiligen Varianz-Kovarianz-Matrizen entfernt. Im Anschluss werden die Punkte in der zur Folgeepoche gehörenden Menge in die Reihenfolge gebracht, die diese auch in der Punktmenge der Bezugsepoche haben. Hierbei werden auch die Zeilen und Spalten der zugehörigen Varianz-Kovarianz-Matrix umsortiert.
Bei der Durchführung der Deformationsanalyse wird über die in der Bezugsepoche gemessenen Punkte iteriert. Es werden für jeden Punkt die k nächsten Punkte ermittelt und die resultierenden Punktmengen werden zur Bildung von Dreiecken genutzt, indem alle möglichen Kombinationen von jeweils drei Punkten ermittelt werden. Die ermittelten Punktkombinationen werden in der Menge D gesammelt, wodurch D am Ende der Iteration Punktkombinationen enthält, die alle in den Epochen gemessenen Punkte mindestens einmal enthalten.

Im Anschluss wird über die in D enthaltenen Dreiecke iteriert. Das Datum wird in das im aktuellen Iterationsschritt betrachtete Dreieck D_i gelegt. Für das Dreieck, das die Koordinaten der in D_i enthaltenen Punkte

bezüglich der Folgeepoche aufspannen, wird der Maßstab in Bezug auf das durch die Koordinaten der Bezugsepoche aufgespannte Dreieck berechnet. Dieser wird an die Koordinaten der betrachteten Punkte der Folgeepoche angebracht. Mit Hilfe der Koordinatendifferenzen der in D_i enthaltenen Punkte wird die Testgröße T für den Ähnlichkeitstest berechnet, die gegen die χ^2-Verteilung getestet wird. Versagt dieser Test, so sind die Dreiecke nicht ähnlich und somit auch nicht kongruent. Sind die Dreiecke ähnlich, so werden sie einer weiteren Transformation unterzogen und es wird die Testgröße für den Kongruenztest ermittelt, die ebenfalls gegen die χ^2-Verteilung getestet wird. Versagt dieser Test, so sind die Dreiecke nicht kongruent. Andernfalls wird das Dreieck in die Liste der kongruenten Dreiecke übernommen.

Nachdem alle Dreiecke auf Kongruenz getestet wurden, werden aus den kongruenten Dreiecken zusammenhängende kongruente Teilnetze gebildet. Hierzu wird ein beliebiges Dreieck als Startpunkt ausgewählt. Zu diesem wird ein weiteres Dreieck gesucht, das mit ihm mindestens einen gemeinsamen Punkt hat. Die beiden Dreiecke werden zu einem zusammenhängenden Polygon vereinigt und aus der Liste der kongruenten Dreiecke entfernt. Zu dem ermittelten Polygon werden nun iterativ weitere Dreiecke gesucht, die mit ihm mindestens einen gemeinsamen Punkt haben. Sollten sich keine weiteren Dreicke mehr finden lassen, so ist das Polygon ein kongruentes Teilnetz aus Stabilpunkten. Das Verfahren wird wiederholt, solange die Liste der kongruenten Dreiecke nicht leer ist.

6.4.3.2 Algorithmus zur Stützpunktsuche mit Strainanalyse

```
1   P_0 = Bezugsepoche
2   P_1 = Folgeepoche
3   M_0 = Menge der Punkte der Bezugsepoche
4   M_1 = Menge der Punkte der Folgeepoche
5   if (((Dimension == 2) ∧ (P_0.getNetzdefekt() ≤ 3)) ∨
       ((Dimension == 3) ∧ (P_0.getNetzdefekt() ≤ 6)) ∨
       ((Dimension == 2) ∧ (P_1.getNetzdefekt() ≤ 3)) ∨
       ((Dimension == 3) ∧ (P_1.getNetzdefekt() ≤ 6)))
6      {
7        M_S = M_0 ∩ M_1; M'_0 = M_0 ∩ M_S; M'_1 = M_1 ∩ M_S;
8        Datum in die in M'_0 bzw. M'_1 enthaltenen Punkte legen;
9        Punkte in M_1 in die Reihenfolge von M_0 bringen;
10       Dreiecke D_j aus den Punkten in M_0 bilden und in der Menge D ablegen;
11       for (D_j ∈ D; j < card(D))
12         {
13           Koordinatenvektoren aufstellen;
14           Datum in die in D_j enthaltenen Punkte legen;
15           Maßstab M von D^1_j in Bezug auf D^0_j bestimmen;
16           M an den zu D^1_j gehörenden Koordinatenvektor anbringen;
17           Differenzenvektoren berechnen;
18           Testgröße T_Ähnl für den Ähnlickeitstest ermitteln;
19           if (¬(T_Ähnl < χ²_f(α))) { Die Dreiecke sind nicht ähnlich; Strainanalyse; }
20           else
21             {
22               S-Transformation durchführen;
23               Testgröße T_Kong für den Kongruenztest ermitteln;
24               if (¬(T_Kong < χ²_f(α))) { Die Dreiecke sind nicht kongruent; Strainanalyse; }
25               else { D_i zur Menge D_Kong der kongruenten Dreiecke hinzufügen; }
26             } // end else
27         } // end for
28       while (D_Kong ≠ ∅)
29         {
30           P_Kong = ∅;
31           P_Kong = P_Kong ∪ D^1_Kong;
32           P'_Kong = ∅;
```

```
33              while (D_Kong ≠ ∅) ∧ (card(P'_Kong) < card(P_Kong))
34                {
35                  P'_Kong = P_Kong;
36                  for (D^j_Kong ∈ D_Kong; j < card(D_Kong))
37                    {
38                      if (D^j_Kong ∩ P_Kong ≠ ∅ )
39                        { P_Kong = P_Kong ∪ D^j_Kong; D_Kong = D_Kong/D^j_Kong; }
40                    } // end for
41                } // end while
42              P_Kong zu den kongruenten Teilnetzen hinzufügen;
43            } // end while
44        } // end if
```

• Zeile 1-9: Die Beschreibung der Funktionsweise dieser Zeilen findet sich in 6.4.2.2.

• Zeile 10: Zur Ermittlung der Dreiecke, mit denen die Ähnlichkeits- bzw. Kongruenztests durchgeführt werden, werden zu jedem Punkt aus M_0 die k nächsten Punkte ermittelt, $2 < k < card(M_0)$. Hierzu wird die Adjazenzmatrix der gesamten Punktmenge M_0 aufgestellt, in die die Entfernung der jeweiligen Punkte voneinander eingetragen wird. Um zu einem Punkt p_j die k nächsten Punkte zu bestimmen, werden aus der zu dem Punkt gehörende Zeile Tupel aus der Entfernung und dem zugehörigen Punkt gebildet. Diese Tupel werden den Entfernungen nach aufsteigend sortiert. Die ersten k Tupel, die eine von Null verschiedene Entfernung aufweisen, enthalten die gesuchten Punkte. Diese Punkte werden in der Menge PK_{p_j} gesammelt. Sollten die Tupel an den Positionen k und k+1 die gleiche Entfernung haben, so wird auch der Punkt an der Position k+1 zur Ermittlung der Dreiecke verwendet. Zu jeder Menge PK_{p_j} werden alle möglichen Tupel D_j gebildet, die jeweils drei verschiedene Punkte enthalten. Diese Tupel repräsentieren die Dreiecke, die in der Menge D gesammelt werden.

• Zeile 11: Iteration über alle in D enthaltenen Mengen. D_j bezeichnet das im aktuellen Iterationsschritt betrachtete Dreieck aus D.

• Zeile 13: Die Koordinatenvektoren $\bar{x}_i$ haben die Form
$\bar{x}_i = \begin{bmatrix} x_1 & y_1 & z_1 & \cdots & x_n & y_n & z_n \end{bmatrix}^T$, wobei die x_k, y_k, z_k die Koordinaten der Punkte der jeweiligen Epoche sind, zu der der Koordinatenvektor gehört.

• Zeile 14: Das Datum wird mittels einer Schwerpunktstransformation in die in D_j enthaltenen Punkte gelegt. Hierzu werden die Matrizen $S_i = I - G_i(G_i^T E G_i)^{-1} G_i^T E$ sowie $S_i^\star = I - G_i^\star(G_i^{\star T} E G_i^\star)^{-1} G_i^{\star T} E$ gebildet. E ist hierbei eine Matrix, die an den Stellen auf der Hauptdiagonalen, die den in D_j enthaltenen Punkten entsprechen, mit 1 besetzt ist und ansonsten mit 0. Die Matrizen G_i werden aus Submatrizen aufgebaut, die im zweidimensionalen Fall die Form

$$G^k = \begin{bmatrix} 1 & 0 & -y & x \\ 0 & 1 & x & y \end{bmatrix}$$

und im dreidimensionalen Fall die Form

$$G^k = \begin{bmatrix} 1 & 0 & 0 & 0 & +z & -y & x \\ 0 & 1 & 0 & -z & 0 & +x & y \\ 0 & 0 & 1 & +y & -x & 0 & z \end{bmatrix}$$

haben. Der Punkt aus M_i, zu dem der jeweilige Block gehört, wird durch k angegeben. Die Matrizen $G_i^\star$ werden aus den G_i aufgebaut, durch

$$G_i^\star = \begin{cases} (1.\text{ bis } 3.\text{ Spalte von } G_i \text{ , falls Dimension} = 2, \\ (1.\text{ bis } 6.\text{ Spalte von } G_i \text{ , falls Dimension} = 3. \end{cases}$$

Mit Hilfe der S_i werden die C'_{xx_i} berechnet durch: $C'_{xx_i} = S_i \cdot C_{xx_i} \cdot S_i^T$. Durch die Schwerpunktstransformation wird der Defekt in die in D_j enthaltenen Punkte gelegt. Nach der Transformation werden die Zeilen und Spalten

der nicht in D_j enthaltenen Punkte aus den C'_{xx_i} entfernt. Die Koordinatenvektoren werden durch $\bar{x}_i^\star = S_i^\star \cdot \bar{x}_i$ ebenfalls transformiert. Nach der Transformation werden auch hier die Zeilen der nicht in D_j enthaltenen Punkte entfernt.

• Zeile 15: Der Maßstab des Dreiecks D_j^1 in Bezug auf das Dreieck D_j^0 wird berechnet. D_j^1 ist das, mit den in D_j enthaltenen Punkten aus den Koordinaten der Folgeepoche und D_j^0 das mit den Koordinaten der Bezugsepoche gebildete Dreieck. Zu jedem Punkt aus D_j^1 und D_j^0 wird m_l gebildet durch $m_l = \frac{\sqrt{x_0^2+y_0^2}}{\sqrt{x_1^2+y_1^2}} = \frac{s_0}{s_1}$ bzw. durch $m_l = \frac{\sqrt{x_0^2+y_0^2+z_0^2}}{\sqrt{x_1^2+y_1^2+z_1^2}} = \frac{s_0}{s_1}$. Die x_0, y_0, z_0 sind hierbei die Koordinaten des betrachteten Punktes in der Bezugsepoche und die x_1, y_1, z_1 die Koordinaten in der Folgeepoche. Zu jedem m_l werden die folgenden Werte berechnet: $f_1 = \frac{\partial m_l}{\partial x_0} = \frac{x_0}{s_0 s_1}$, $f_2 = \frac{\partial m_l}{\partial y_0} = \frac{y_0}{s_0 s_1}$, $f_3 = \frac{\partial m_l}{\partial z_0} = \frac{z_0}{s_0 s_1}$, $f_4 = \frac{\partial m_l}{\partial x_1} = \frac{s_0 \cdot x_1}{-s_1^3}$, $f_5 = \frac{\partial m_l}{\partial y_1} = \frac{s_0 \cdot y_1}{-s_1^3}$ und $f_6 = \frac{\partial m_l}{\partial z_1} = \frac{s_0 \cdot z_1}{-s_1^3}$. Mit diesen werden im zweidimensionalen Fall die Vektoren

$$F_1 = \begin{bmatrix} f_1 & f_2 & 0 & 0 & 0 & 0 & f_4 & f_5 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$F_2 = \begin{bmatrix} 0 & 0 & f_1 & f_2 & 0 & 0 & 0 & 0 & f_4 & f_5 & 0 & 0 \end{bmatrix}$$

$$F_3 = \begin{bmatrix} 0 & 0 & 0 & 0 & f_1 & f_2 & 0 & 0 & 0 & 0 & f_4 & f_5 \end{bmatrix}$$

und im dreidimensionalen Fall die Vektoren

$$F_1 = \begin{bmatrix} f_1 & f_2 & f_3 & 0 & 0 & 0 & 0 & 0 & 0 & f_4 & f_5 & f_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$F_2 = \begin{bmatrix} 0 & 0 & 0 & f_1 & f_2 & f_3 & 0 & 0 & 0 & 0 & 0 & 0 & f_4 & f_5 & f_6 & 0 & 0 & 0 \end{bmatrix}$$

$$F_3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & f_1 & f_2 & f_3 & 0 & 0 & 0 & 0 & 0 & 0 & f_4 & f_5 & f_6 \end{bmatrix}$$

gebildet. Die Blockdiagonalmatrix C'_{xx} wird durch $C'_{xx} = \begin{pmatrix} C'_{xx_0} & 0 \\ 0 & C'_{xx_1} \end{pmatrix}$ definiert und der Maßstab M berechnet sich zu $M = \frac{m_1 \cdot {C_{mm_1}}^{-1} + m_2 \cdot {C_{mm_2}}^{-1} + m_3 \cdot {C_{mm_3}}^{-1}}{{C_{mm_1}}^{-1} + {C_{mm_2}}^{-1} \cdot {C_{mm_3}}^{-1}}$. Hierbei gilt $C_{mm_k} = F_k (C_{xx})^+ F_k^T$, $k \in \{1, 2, 3\}$.

• Zeile 16: Den Maßstab M an den Koordinatenvektor der Folgeepoche anbringen durch $\bar{x}'_1 = M \cdot \bar{x}_1^\star$.

• Zeile 17: Der Differenzenvektor $d_{\ddot{A}hnl}$ berechnet sich durch $d_{\ddot{A}hnl} = \bar{x}_0^\star - \bar{x}'_1$. Der Differenzenvektor d_{Kong} berechnet sich durch $d_{Kong} = \bar{x}_0^\star - \bar{x}_1^\star$.

• Zeile 18: Die Testgröße $T_{\ddot{A}hnl}$ berechnet sich durch $T_{\ddot{A}hnl} = {d_{\ddot{A}hnl}}^T (C'_{xx_0} + C'_{xx_1})^+ d_{\ddot{A}hnl}$.

• Zeile 19: Die Testgröße $T_{\ddot{A}hnl}$ wird gegen das Fraktil der χ^2-Verteilung mit dem Freiheitsgrad f bei vorgegebener Irrtumswahrscheinlichkeit α getestet. Der Freiheitsgrad f ist definiert durch

$$f = \begin{cases} 2 \cdot card(M_0) - 4, \text{ falls Dimension} = 2, \\ 3 \cdot card(M_0) - 7, \text{ falls Dimension} = 3. \end{cases}$$

Bei bestandenem Test sind die beiden Dreiecke ähnlich. Andernfalls muss eine Strainanalyse durchgeführt werden. Diese wird durch $u = (B^T (\frac{1}{\sigma_0^2} C_{dd})^+ B)^+ B^T (\frac{1}{\sigma_0^2} C_{dd})^+ d_{Kong}$ berechnet. Hierbei gilt $C_{dd} = (S_0^\star Q_{xx_0} S_0^{\star T} + S_1^\star Q_{xx_1} S_1^{\star T})$. Die Matrix B wird aus Submatrizen aufgebaut, die im zweidimensionalen Fall die Form

$$B^k = \begin{bmatrix} x & y & 0 & -y & 1 & 0 \\ 0 & x & y & +x & 0 & 1 \end{bmatrix}$$

und im dreidimensionalen Fall die Form

$$B^k = \begin{bmatrix} x & 0 & 0 & y & z & 0 & 0 & +z & -y & 1 & 0 & 0 \\ 0 & y & 0 & x & 0 & z & -z & 0 & +x & 0 & 1 & 0 \\ 0 & 0 & z & 0 & x & y & +y & -x & 0 & 0 & 0 & 1 \end{bmatrix}$$

haben. Die x, y, z sind die Koordinaten der Punkte aus M_0, die in dem aktuell betrachteten Dreieck D_j enthalten sind. Der Punkt aus D_j wird durch den Index k angegeben.

• Zeile 22: Durch eine S-Transformation wird der Defekt auf 3 geändert und in die Punkte des aktuell betrachteten Dreiecks D_j gelegt. Die Transformation wird durchgeführt durch $C^{\star}_{xx_i} = \sigma_0^2 S_i^{\star} Q_{xx_j} S_i^{\star T}$. Im Anschluss an die Transformation werden die Zeilen und Spalten der nicht in D_j enthaltenen Punkte aus den $C^{\star}_{xx_i}$ entfernt.

• Zeile 24: Die Testgröße T_{Kong} wird gegen das Fraktil der χ^2-Verteilung mit dem Freiheitsgrad f bei vorgegebener Irrtumswahrscheinlichkeit α getestet. Der Freiheitsgrad f ist definiert durch

$$f = \begin{cases} 2 \cdot card(M_0) - 3, \text{ falls Dimension} = 2, \\ 3 \cdot card(M_0) - 6, \text{ falls Dimension} = 3. \end{cases}$$

Bei bestandenem Test sind die beiden Dreiecke kongruent und das aktuell betrachtete Dreieck wird in die Liste der kongruenten Dreiecke aufgenommen. Andernfalls muss eine Strainanalyse durchgeführt werden.

• Zeile 31: Die im ersten Dreieck aus D_{Kong} enthaltenen Punkte zur Menge P_{Kong} hinzufügen.

• Zeile 32: Wiederholen, solange sich noch Dreiecke in D_{Kong} befinden und sich die Menge P_{Kong} in der letzten Iteration vergrößert hat.

• Zeile 36: Iteration über die in D_{Kong} enthaltenen Dreiecke.

• Zeile 39: Wenn das in der aktuellen Iteration betrachtete Dreieck D^j_{Kong} mindestens einen gemeinsamen Punkt mit der Menge P_{Kong} hat, so werden die in D^j_{Kong} enthaltenen Punkte mit denen aus P_{Kong} vereinigt. Danach wird D^j_{Kong} aus D_{Kong} entfernt.

• Zeile 42: Da keine weiteren Dreiecke ermittelt werden konnten, die gemeinsame Punkte mit P_{Kong} haben, ist P_{Kong} ein kongruentes Teilnetz.

6.4.4 Einzelpunktanalyse

6.4.4.1 Allgemeine Beschreibung des Algorithmus

Dieser Algorithmus basiert auf einem Testverfahren [35], das für alle Punkte einzeln eine Testgröße ermittelt. Die Schwerpunkte, der in den Epochen gemessenen Netze, werden hierzu in den Nullpunkt der Bezugsepoche gelegt. Gleichzeitig werden die Koordinaten der Punkte der Folgeepoche bestmöglich an die der Bezugsepoche angepasst. Im Anschluss wird der Punkt mit der kleinsten Testgröße auf Stabilität gestestet. Das Datum wird iterativ in die Stabilpunkte gelegt und es werden wiederum die Testgrößen für die noch nicht als stabil erkannten Punkte ermittelt.

Vor Beginn der eigentlichen Deformationsanalyse werden Vorverarbeitungsschritte durchgeführt, um die zu den Epochen gehörenden Punktmengen so zu reduzieren, dass in beiden Mengen nur noch die in beiden Epochen gemessenen Punkte enthalten sind. Hierbei werden auch die zu den entfernten Punkten gehörenden Zeilen und Spalten aus den jeweiligen Varianz-Kovarianz-Matrizen entfernt. Danach werden die Punkte in der zur Folgeepoche gehörenden Menge in die Reihenfolge gebracht, die diese auch in der Punktmenge der Bezugsepoche haben. Hierbei werden auch die Zeilen und Spalten der zugehörigen Varianz-Kovarianz-Matrix umsortiert.

Zur Deformationsanalyse werden die zu den Punktmengen der Epochen gehörenden Koordinatenvektoren zuerst einer Schwerpunkttransformation unterzogen. Für jeden Punkt wird mit Hilfe der Koordinatendifferenzen sowie den mittleren Punktfehlern eine Testgröße ermittelt. Die minimale Testgröße wird dann gegen die Fisher-Verteilung auf Stabilität geprüft.

Die verbleibenden Punkte werden einer Transformation unterzogen, um das Datum in den als stabil ermittelten Punkt zu legen. Hiernach werden wieder die Testgrößen für die verbleibenden Punkte ermittelt. Die minimale Testgröße wird wiederum gegen die Fisher-Verteilung auf Stabilität geprüft.

Im Anschluss wird über die verbleibenden, bisher nicht als stabil erkannten Punkte iteriert. Mit Hilfe von Schwerpunkttransformationen wird das Datum in die bereits als stabil erkannten Punkte gelegt, wonach für alle anderen Punkte wieder die Testgröße ermittelt wird. Die minimale Testgröße wird wieder gegen die Fisher-Verteilung auf Stabilität geprüft. Die Iteration endet, wenn kein stabiler Punkt mehr ermittelt werden kann oder keine Punkte zum Testen mehr vorhanden sind.

6.4.4.2 Algorithmus zur Einzelpunktanalyse

```
1   P0 = Bezugsepoche
2   P1 = Folgeepoche
3   M0 = Menge der Punkte der Bezugsepoche
4   M1 = Menge der Punkte der Folgeepoche
5   if (((Dimension == 2) ∧ (P0.getNetzdefekt() ≤ 3)) ∨
       ((Dimension == 3) ∧ (P0.getNetzdefekt() ≤ 6)) ∨
       ((Dimension == 2) ∧ (P1.getNetzdefekt() ≤ 3)) ∨
       ((Dimension == 3) ∧ (P1.getNetzdefekt() ≤ 6)))
6      {
7        MS = M0 ∩ M1; M0' = M0 ∩ MS; M1' = M1 ∩ MS;
8        Datum in die in M0' bzw. M1' enthaltenen Punkte legen;
9        Punkte in M1 in die Reihenfolge von M0 bringen;
10       G-Matrizen aufstellen;
11       Koordinatenvektoren aufstellen;
12       Schwerpunkttransformation auf den Koordinatenvektoren durchführen;
13       Differenzenvektor aufstellen;
14       Testgrößen Tj berechnen und minimale Testgröße Tmin bestimmen;
15       if (¬(Tmin < F1,f(α))) { Keiner der Punkte ist stabil; break; }
16       else
17         {
18           Datum in den zu Tmin gehörigen Punkt legen;
19           Differenzenvektor aufstellen;
20           Testgrößen Tj für alle nicht stabilen Punkte berechnen und minimale
             Testgröße Tmin bestimmen;
21           if (¬(Tmin < F1,f(α))) { Kein weiterer Punkt ist stabil; break; }
22           else
23             {
24               while (mind. 1 Punkt ist nicht stabil)
25                 {
26                   Datum in die stabilen Punkte legen;
27                   Differenzenvektor aufstellen;
28                   Testgrößen Tj für alle nicht stabilen Punkte berechnen und minimale
                     Testgröße Tmin bestimmen;
29                   if (¬(Tmin < F1,f(α))) { Kein weiterer Punkt ist stabil; break; }
30                   else { Den zu Tmin gehörenden Punkt als stabil markieren; }
31                 } // end while
32             } // end else
33         } // end else
34     } // end if
```

- Zeile 1-9: Die Beschreibung der Funktionsweise dieser Zeilen findet sich in 6.4.2.2.

• Zeile 10: Die Matrizen G_i werden aus Submatrizen aufgebaut, die im zweidimensionalen Fall die Form

$$G^j = \begin{bmatrix} 1 & 0 & -y \\ 0 & 1 & +x \end{bmatrix}$$

und im dreidimensionalen Fall die Form

$$G^j = \begin{bmatrix} 1 & 0 & 0 & 0 & +z & -y \\ 0 & 1 & 0 & -z & 0 & +x \\ 0 & 0 & 1 & +y & -x & 0 \end{bmatrix}$$

haben. Der Punkt aus M_i, zu dem der jeweilige Block gehört, wird durch j angegeben.

• Zeile 11: Die Koordinatenvektoren $\bar{x}_i$ haben die Form
$x_i = \begin{bmatrix} x_1 & y_1 & z_1 & \cdots & x_n & y_n & z_n \end{bmatrix}^T$, wobei die x_j, y_j, z_j die Koordinaten der Punkte der jeweiligen Epoche sind, zu der der Koordinatenvektor gehört.

• Zeile 12: Auf den Koordinatenvektoren wird eine Schwerpunkttransformation durchgeführt. Hierzu wird die Matrix $S_x = I - G_0(G_0{}^T I G_0)^{-1} G_0{}^T I$ gebildet. Mit Hilfe der S_x werden die x_i' berechnet durch $x_i' = S_x \cdot \bar{x}_i$. Hierdurch werden die Schwerpunkte der durch die Koordinaten aufgespannten Netze in den Nullpunkt der Bezugsepoche gelegt. Gleichzeitig werden die Koordinaten der Folgeepoche bestmöglich an die der Bezugsepoche angepasst.

• Zeile 13: Der Differenzenvektor wird gebildet durch $d_x = x_1' - x_0'$.

• Zeile 14: Die Testgrößen T_j für die einzelnen Punkte werden berechnet durch

$$T_j = \frac{\Delta_j^2}{\sigma_{P_m}^2 + \sigma_{P_n}^2}$$

mit

$$\sigma_{P_k}^2 = \begin{cases} (\sigma_{z_k}^2) & \text{, falls Dimension} = 1, \\ (\sigma_{x_k}^2 + \sigma_{y_k}^2) & \text{, falls Dimension} = 2, \\ (\sigma_{x_k}^2 + \sigma_{y_k}^2 + \sigma_{z_k}^2) & \text{, falls Dimension} = 3. \end{cases}$$

und

$$\Delta_j^2 = \begin{cases} (\Delta Z_j^2) & \text{, falls Dimension} = 1, \\ (\Delta X_j^2 + \Delta Y_j^2) & \text{, falls Dimension} = 2, \\ (\Delta X_j^2 + \Delta Y_j^2 + \Delta Z_j^2) & \text{, falls Dimension} = 3. \end{cases}$$

wobei die $\Delta X_j^2, \Delta Y_j^2$ und ΔZ_j^2 in d_x enthalten sind.

• Zeile 15: Die minimale Testgröße T_{min} wird gegen das Fraktil der Fisher-Verteilung mit den Freiheitsgraden 1 und f bei vorgegebener Irrtumswahrscheinlichkeit α getestet. Der Freiheitsgrad f ist definiert durch

$$f = \sum(diag(\frac{1}{\sigma_0^2}(C_{vv_0} \cdot (\frac{1}{\sigma_0^2} \cdot C_{ll_0})^{-1})) + \sum(diag(\frac{1}{\sigma_0^2}(C_{vv_1} \cdot (\frac{1}{\sigma_0^2} \cdot C_{ll_1})^{-1}))$$

Bei bestandenem Test ist der Punkt, der zu T_{min} gehört als stabil zu betrachten. Andernfalls ist keiner der Punkte als stabil zu betrachten.

• Zeile 18: Um das Datum in den ermittelten Stabilpunkt zu legen, wird zunächst der maximale Wert w_{max} der Absolutbeträge aus x_0' bestimmt. Zu diesem wird ein k bestimmt, sodass gilt: $10^{k+1} > w_{max} > 10^k$. Weiterhin werden Matrizen $G_i^\star$ und G_i', $i \in \{0, 1\}$, bestimmt, die aus Submatrizen aufgebaut werden, die im zweidimensionalen Fall die Form

$$G^{\star j} = \begin{bmatrix} -y \\ +x \end{bmatrix}$$

bzw.

$$G'^{j} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

und im dreidimensionalen Fall die Form

$$G^{\star j} = \begin{bmatrix} 0 & +z & -y \\ -z & 0 & +x \\ +y & -x & 0 \end{bmatrix}$$

bzw.

$$G'^{j} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

haben. Der Punkt aus M_i, zu dem der jeweilige Block gehört, wird durch j angegeben. Mit den $G_i^\star$ werden die $C^\star_{xx_i}$ berechnet zu $C^\star_{xx_i} = C_{xx_i} + (10^k \cdot (G_i^\star \cdot G_i^{\star T}))$. Die G'_i werden zur Berechnung der S'_i verwendet, die sich zu $S'_i = I - G'_i(G_i'^T E G'_i)^{-1} G_i'^T E$ ergeben. Die Matrix E ist an den Stellen auf der Hauptdiagonalen, die zum bisher ermittelten Stabilpunkt gehören, mit 1 besetzt und ansonsten mit 0. Mit Hilfe der S'_i werden die $C'_{xx_i} = S'_i \cdot C^\star_{xx_i} \cdot S_i'^T$ und die $x''_i = S'_i \cdot x'_i$ berechnet. Der Differenzenvektor wird mit den x''_i gebildet zu $d'_x = x''_1 - x''_0$.

• Zeile 20: Die Testgrößen für die noch nicht als stabil erkannten Punkte werden wie in der Beschreibung zu Zeile 14 berechnet. Zur Bestimmung der $\sigma^2_{P_k}$ werden die C'_{xx_i}, und zur Bestimmung der Δ^2_j der Differenzenvektor d'_x herangezogen.

• Zeile 21: Die minimale Testgröße T_{min} wird gegen das Fraktil der Fisher-Verteilung mit den Freiheitsgraden 1 und f bei vorgegebener Irrtumswahrscheinlichkeit α getestet. Bei bestandenem Test ist der Punkt, der zu T_{min} gehört, als stabil zu betrachten. Andernfalls sind keine weiteren stabilen Punkte vorhanden.

• Zeile 24: Iterieren, solange noch Punkte vorhanden sind, die bisher nicht als stabil erkannt wurden.

• Zeile 26: Das Datum wird mittels einer Schwerpunkttransformation in die bisher als stabil erkannten Punkte gelegt. Hierzu werden die notwendigen $S_i = I - G_i(G_i^T E G_i)^{-1} G_i^T E$, berechnet und mit diesen die $C^\diamond_{xx_i} = S_i \cdot C_{xx_i} \cdot S_i^T$. Die $x_i^\diamond = S_i \cdot x_i$ sind die transformierten Koordinatenvektoren. Die Matrix E ist an den Stellen auf der Hauptdiagonalen, die zu den bisher ermittelten Stabilpunkten gehören, mit 1 besetzt und ansonsten mit 0.

• Zeile 27: Aus den $x_i^\diamond$ wird der Differenzenvektor $d_x^\diamond = x_1^\diamond - x_0^\diamond$ gebildet.

• Zeile 28: Die Testgrößen für die noch nicht als stabil erkannten Punkte werden wie in der Beschreibung zu Zeile 13 berechnet. Zur Bestimmung der $\sigma^2_{P_k}$ werden die $C^\diamond_{xx_i}$ und zur Bestimmung der Δ^2_j der Differenzenvektor $d_x^\diamond$ herangezogen.

• Zeile 29: Die minimale Testgröße T_{min} wird gegen das Fraktil der Fisher-Verteilung mit den Freiheitsgraden 1 und f bei vorgegebener Irrtumswahrscheinlichkeit α getestet. Bei bestandenem Test ist der Punkt, der zu T_{min} gehört, als stabil zu betrachten. Andernfalls sind keine weiteren stabilen Punkte vorhanden.

6.5 Prototypische Realisierung

Die in Abschnitt 6.1 beschriebene Systemarchitektur wurde prototypisch realisiert. Zu dieser prototypischen Realisierung gehören eine Benutzerschnittstelle, je eine Komponente zur eindimensionalen und zur zweidimensionalen Netzausgleichung, drei Komponenten zur Durchführung von verschiedenen Ansätzen zur statischen Deformationsanalyse, das in [53] beschriebene Klassenmodell der geodätischen Entitäten sowie der Persistenzmechanismus. Komponenten zur Vorverarbeitung der Beobachtungen oder zur Visualisierung der Ergebnisse wurden im Rahmen dieser Arbeit nicht realisiert.

Die Benutzerschnittstelle stellt die verfügbaren Komponenten des Datenflusses zur Auswahl bereit und unterstützt das Laden und Speichern von Daten unter Verwendung des Persistenzmechanismus und des Langzeitdatenspeichers. Die manuelle Dateneingabe mit Hilfe von Dialogfenstern ist für einige wichtige Beobachtungstypen und für Netztopologien ebenfalls realisiert. Die Realisierung von Dialogfenstern für die übrigen bereits vorhandenen Beobachtungstypen sowie zur Eingabe von Punktmengen für die Deformationsanalyse sind derzeit noch offene Punkte. Der Import von Netztopologien, die mit Software von Drittherstellern vorverarbeitet wurden ist aufgrund der Vielzahl der vorhandenen proprietären Formaten ebenfalls noch nicht realisiert.

Die in den Abschnitten 6.3.3.3 und 6.3.4 beschriebenen Algorithmen zur Netzausgleichung sind in eigenständigen Komponenten realisiert. Die Interaktion mit diesen Komponenten geschieht durch die in Abschnitt 6.3.1 beschriebene Schnittstelle. Die in Abschnitt 6.3.2 beschriebene Verfahrensweise zur Entkopplung der funktionalen Modelle von den Beobachtungstypen ist in diesem Rahmen ebenfalls in beiden Komponenten umgesetzt. Zum Testen der eindimensionalen Netzausgleichung wurden synthetische Daten verwendet, bei denen die Ergebnisse vorher bekannt waren. Der erste Datensatz bestand aus neun Höhenunterschieden und sechs Punkten. Die vorausberechneten Ergebnisse konnten für diese Netztopologie nachgewiesen werden. Der zweite Datensatz bestand aus vier Höhenunterschieden, vier Zenitdistanzen, vier Schrägstrecken und fünf Punkten. Auch für diesen Datensatz konnten die vorausberechneten Ergebnisse nachgewiesen werden. In einem dritten Test wurden dem System einige neue funktionale Modelle hinzugefügt, um die Funktionsfähigkeit des in Abschnitt 6.3.2 beschriebenen Verfahrens zu testen. Auch für diese neuen funktionalen Modelle konnten die vorausberechneten Ergebnisse nachgewiesen werden. Zum Testen der zweidimensionalen Netzausgleichung wurden ebenfalls synthetische Daten verwendet. Hier standen ebenfalls zwei verschiedene Netztopologien zur Verfügung. Die erste Netztopologie bestand aus 19 Horizontalstrecken und zehn Punkten. Für diese konnten die vorausberechneten Ergebnisse nachgewiesen werden. Die zweite Netztopologie bestand aus drei Richtungssätzen mit jeweils vier Richtungsmessungen, einem Azimutsatz mit vier Azimutmessungen, vier Streckenverhältnissen, neun Horizontalstrecken, zwei terrestrischen Koordinatendifferenzen, zwei Winkelverhältnissen und zehn Punkten. Auch für diese Netztopologie konnten die vorausberechneten Ergebnisse nachgewiesen werden.

Im Rahmen der Ausgleichung geodätischer Netze konnte durch die Testdatensätze gezeigt werden, dass die in [53] beschriebenen und in eigenen Klassen gekapselten funktionalen Modelle ihre Funktion erfüllen. Es konnte durch Hinzufügen abgewandelter funktionaler Modelle auch die Funktionsfähigkeit des in Abschnitt 6.3.2 beschriebenen Verfahrens nachgewiesen werden. Das eingesetzte Verfahren, bei dem zur Laufzeit der Netzausgleichung zum gerade zu verarbeitenden Beobachtungstyp das jeweilige zu verwendende Exemplar des funktionalen Modelle erzeugt wird, führt zu Einbußen bei der Geschwindigkeit. Es ist jedoch effizient hinsichtlich des Speicherbedarfs, was besonders bei einer großen Anzahl an Beobachtungen zum Tragen kommt. Wenn jedoch die Geschwindigkeit des Systems optimiert werden soll, dann ist es möglich, alle benötigten Exemplare der funktionalen Modelle vor der Durchführung des Ausgleichungsalgorithmus zu erzeugen. Aufgrund der Implementierung des Systems in JAVA sind die einzelnen Komponenten langsamer als die korrespondierenden fachlichen Einheiten anderer geodätischer Programmsysteme. Da die Geschwindigkeit bei der Entwicklung des Systems kein vorrangiges Ziel war, ist dieser Umstand tolerierbar. Das System ist im Gegenzug in der Lage, weit mehr Beobachtungstypen zu verarbeiten als vergleichbare Programmsysteme und es ist durch die verwendete Systemarchitektur offen für Erweiterungen auf verschiedenen fachlichen Ebenen. Die Geschwindigkeit des Systems kann jedoch bei Bedarf durch den Einsatz spezieller Compiler, die Maschinencode für die unterschiedlichen Prozessorarchitekturen erzeugen, gesteigert werden.

Die in den Abschnitten 6.4.2, 6.4.3 und 6.4.4 beschriebenen Ansätze zur Deformationsanalyse sind in jeweils eigenständigen Komponenten realisiert. Die Interaktion mit den Komponenten geschieht durch die in Abschnitt 6.4.1 beschriebene Schnittstelle. Zum Testen der Komponenten wurden zwei Datensätze mit jeweils acht Punkten verwendet. Für die Tests wurden die Koordinaten einiger Punkte des Datensatzes der Folgeepoche so modifiziert, dass die Analyseverfahren signifikante Koordinatenänderungen erkannten. Auf diese Weise konnte die Funktionsfähigkeit der verschiedenen Verfahren nachgewiesen werden.

Der in Abschnitt 6.2 beschriebene Persistenzmechanismus wurde als Projektlösung realisiert. Die Transformation der Objekte geschieht durch die in den Abschnitten 5.2.2.3, 5.2.2.4, 5.2.2.5 und 5.2.2.6 beschriebenen Abbildungen. Die Verfahren zur Umsetzung der Abbildungen wurden in Abschnitt 6.2.4.5 beschrieben. Bei der Generierung der notwendigen SQL-Anweisungen wurden keine datenbankspezifischen SQL-Erweiterungen

eingesetzt, um das System nicht auf einen bestimmten Datenbanktyp einzuschränken. Bei der prototypischen Realisierung wurde ein Datenbanksystem vom Typ *Oracle 8i* als Langzeitdatenspeicher eingesetzt. Bei der Speicherung werden die Objekte zur Laufzeit analysiert, um den aktuellen Objektzustand zu ermitteln. Hierdurch ist die Realisierung flexibel und es sind nur geringe Anpassungen notwendig, wenn Änderungen am Datenmodell notwendig werden. Die Funktionsfähigkeit des Persistenzmechanismus wurde mit den Datensätzen zum Testen der Netzausgleichung und der Deformationsanalyse getestet. Diese wurden mit Hilfe des Persistenzmechanismus im Langzeitdatenspeicher abgelegt und bei den Tests der verschiedenen Komponenten aus diesem geladen.

Bei der Realisierung des Persistenzmechanismus werden die Objekte zur Laufzeit analysiert, um die entsprechenden Transformationen durchzuführen, bzw. um die notwendigen SQL-Anweisungen erzeugen zu können. Diese Laufzeitanalyse bringt den Vorteil mit sich, dass der Persistenzmechanismus auf Änderungen am Datenmodell flexibel reagieren kann. Während der prototypsichen Realisierung war diese Flexibilität von Vorteil, da mehrfach Änderungen am Datenmodell durchgeführt wurden. Die erreichte Flexibilität geht jedoch zu Lasten der Geschwindigkeit, was im Betrieb des Systems störend wirken kann. Die Realisierung von festen Abbildungsvorschriften für die Klassen des Datenmodells anstatt der Laufzeitanalyse bringt einen Geschwindigkeitsvorteil auf Kosten der Flexibilität mit sich. Dies muss jedoch nicht von Nachteil sein, wenn das Datenmodell so ausgereift ist, dass keine Änderungen daran mehr zu erwarten sind. Aufgrund der verwendeten Schichtenarchitektur kann eine solche Realisierung von festen Abbildungsvorschriften in einer neuen Schicht geschehen, die dann gegen die bisherige ausgetauscht wird.

Die in Abschnitt 6.2.5 beschriebene Realisierung zum Im- und Export von Daten verwendet ebenfalls die in Abschnitt 6.2.3 vorgestellte Schnittstelle des Persistenzmechanismus. Der Datenexport und auch der Datenimport wurden ebenfalls mit den Testdatensätzen zur Netzausgleichung und zur Deformationsanalyse getestet. Da auch bei dieser Realisierung die Objektzustände zur Laufzeit ermittelt werden, ergibt sich die gleiche Flexibilität bei Änderungen am Datenmodell wie bei der Realisierung des Persistenzmechanismus.

Auch im Rahmen des Datenexports wird die Laufzeitanalyse zur Ermittlung der Objektzustände eingesetzt. Die gerade gemachten Ausführungen bezüglich der Geschwindigkeit und der Flexibilität einer solchen Realisierung gelten sinngemäß auch hier. Zur Geschwindigkeitssteigerung können auch in diesem Fall feste Abbildungsvorschriften anstatt der Laufzeitanalyse verwendet werden.

Bei den Tests der Netzausgleichung und der Deformationsanalyse wurde deutlich, wie schwer die Ergebnisinterpretation anhand von reinem Zahlenmaterial ist. Eine Unterstützung des Benutzers durch eine Visualisierungskomponente ist vor allem bei umfangreichem Datenmaterial erforderlich. Die Eingabe der Testdatensätze wurde durch die vorhandenen Dialogfenster zwar vereinfacht, war jedoch noch immer recht mühselig. Aufgrund der Tatsache, dass die Messergebnisse oft auf Speicherkarten abgelegt werden, ist es notwendig, Daten von den Programmsystemen der Messgerätehersteller importieren zu können. Vor allem vor dem Hintergrund umfangreicher Messreihen erspart ein solcher Datenimport Zeit und schließt eine mögliche Fehlerquelle aus.

Durch die durchgeführten Tests konnte die Funktionsfähigkeit der bisher realisierten Komponenten, das Zusammenspiel der Komponenten, die Funktionsfähigkeit des Persistenzmechanismus und auch die Funktionsfähigkeit der Benutzerschnittstelle gezeigt werden. Auch der in Abschnitt 6.3.2 vorgestellte Ansatz zur Entkopplung der funktionalen Modelle von den Beobachtungstypen hat seine Funktionsfähigkeit im Rahmen der Tests zur Ausgleichung geodätischer Netze bewiesen.

Kapitel 7

Schlussfolgerungen und Ausblick

7.1 Schlussfolgerungen

In dieser Arbeit wurde ein Konzept für ein Informationssystem zur geodätischen Deformationsanalyse sowie dessen Umsetzung beschrieben. Es wurden die Systemarchitektur und die verschiedenen Systembestandteile sowie ihre Funktionalität und die jeweilige Umsetzung beschrieben.

Im Anschluss an die Einleitung wurde in Kapitel 2 der Datenfluss einer Deformationsanalyse aus geodätischer Sicht beschrieben. Hierbei wurde der Datenfluss in fünf Stufen zerlegt, die im Einzelnen vorgestellt wurden. Im zweiten Abschnitt des Kapitels wurde ein Überblick über Programmsysteme gegeben, die überwiegend in der Geodäsie eingesetzt werden. Im Rahmen der Vorstellung der einzelnen Programmsysteme wurde auch darauf eingegangen, welche Stufen des zuvor beschriebenen Datenflusses diese verarbeiten können. Im letzten Abschnitt des Kapitels wurden dann die Anforderungen und Zielsetzungen an ein Informationssystem zur geodätischen Deformationsanalyse definiert. Diese Anforderungen beinhalten zum einen, dass der Datenfluss innerhalb eines Softwaresystems verarbeitet werden kann und zum anderen, dass es sich um ein auf verschiedenen Ebenen erweiterbares System handelt. Auch die Beseitigung der im vorigen Abschnitt beschriebenen Probleme, die bei der Durchführung einer geodätischen Deformationsanalyse mit Hilfe der vorhandenen Programmsysteme auftreten, ist in den Zielsetzungen enthalten. Auch die Notwendigkeit der Langzeitdatenspeicherung der Daten und die Realisierung des Datenim- und exports gehören zu den formulierten Zielsetzungen.

In Kapitel 3 wurde auf die Grundlagen der objektorientierten Modellierung und ihrer Konzepte eingegangen, da diese zum Verstehen der in späteren Kapiteln gemachten Ausführungen notwendig sind. Als grafische Notation wurde die Unified Modeling Language vorgestellt, die ebenfalls ansatzweise vorgestellt wurde, um die Modellierung des Systems auch grafisch darstellen zu können. Durch die vorgestellten objektorientierten Konzepte war es möglich, die Anforderung nach einem Datenmodell, das alle Komponenten des Datenflusses nutzen können sowie die geforderte Flexibilität hinsichtlich neuer Beobachtungstypen und neuer Berechnungsmodelle zu erfüllen. Auch die Verarbeitung des Datenflusses innerhalb einer Software war durch ihren Einsatz möglich. Den Abschluss des Kapitels bildete ein Abschnitt über Entwurfsmuster, in dem zwei, für diese Arbeit zentrale Muster, vorgestellt wurden. Durch ihre Verwendung war es möglich, die Zielsetzung bezüglich der Kapselung der Algorithmik in eigenen Komponenten zu erfüllen und auch sicherzustellen, dass neue Algorithmik auch zu späteren Zeitpunkten in das System integriert werden kann.

In Kapitel 4 wurden einige zentrale Klassen des verwendeten Datenmodells vorgestellt. Hierbei wurde nicht im Detail auf einzelne Attribute oder Methoden der Klassen eingegangen. Vielmehr sollte das Zusammenspiel mit den anderen Klassen des Datenmodells erläutert werden. Die vorgestellten Klassen wurden in späteren Kapiteln zur Illustration der Verfahrensweisen benötigt.

In Kapitel 5 wurde die Langzeitdatenspeicherung der in den Objekten enthaltenen Daten behandelt. Es wurde auf Datenbanksysteme eingegangen und auch darauf, dass mit der Structured Query Language (SQL) eine standardisierte Anfragesprache für diese zur Verfügung steht. Hierdurch erfüllen Datenbanksysteme also die Zielsetzung des Einsatzes eines Langzeitdatenspeichers mit einer standardisierten Zugriffsschnittstelle. In diesem Kapitel wurde ebenfalls gezeigt, wie die in Kapitel 3 vorgestellten objektorientierten Konzepte auf eine relationale Tabellenstruktur abgebildet werden können, um die Zustände von Objekten in einer Datenbank zu speichern. In einem weiteren Abschnitt wurde auf den Datenexport und den Datenimport durch Verwendung der Extensible Markup Language eingegangen. Aufgrund der Möglichkeit, die Daten mit einer Struktur zu versehen und der Verwendbarkeit auf unterschiedlichen Betriebssystemplattformen eignet auch sie sich, um die genannte Zielsetzung zu erfüllen. Die Abbildung der objektorientierten Konzepte auf eine XML-Datei zum Ex- und Import von Objektzuständen wurde ebenfalls in diesem Kapitel beschrieben.

Im Rahmen der Beschreibung der Systemarchitektur wurde in Kapitel 6 auf die Umsetzung des Datenflusses der geodätischen Deformationsanalyse in dem System eingegangen. Hierbei wurden die einzelnen Teilschritte des Datenflusses und die Zuordnungen zu den entsprechenden Systemkomponenten aus Sicht der Datenverarbeitung beschrieben.

Im Falle der Datenspeicherung wurde gezeigt, wie ein Persistenzmechanismus erschaffen wurde, der eine Applikation von allen Aufgaben, die mit der Speicherung, dem Laden oder Löschen von Daten in Verbindung stehen, entlastet. Die vom Persistenzmechanismus angebotene Funktionalität stellt sich einer Applikation auf Objektebene dar. Die Objekte können mit Hilfe eines relationalen Datenbankmanagementsystems persistiert werden. Da die Struktur und der Zustand der zu persistierenden Objekte zur Laufzeit untersucht und mit diesen Informationen die notwendigen SQL-Anweisungen generiert werden, ist der Persistenzmechanismus in der Lage, die Objekte ohne vorgebene, objektspezifische Verarbeitungsvorschriften zu persistieren. Dies ist insbesondere von Bedeutung, da sich die Objektstrukturen auch in späteren Phasen einer Entwicklung noch verändern können. Es wurde ebenfalls beschrieben, wie die Klassen, deren Objekte mit Hilfe des Persistenzmechanismus verarbeitet werden sollen, beschaffen sein müssen. Durch den beschriebenen Persistenzmechanismus wird die in Kapitel 2 aufgestellte Zielsetzung nach Verwendung eines geeigneten Langzeitdatenspeichers erfüllt. Aufgrund der Verwendung eines Datenbanksystems zur Speicherung der Objektzustände sind diese auch für andere Programmsysteme verfügbar, da die Zugriffe über eine standardisierte Schnittstelle möglich sind. Die zugreifende Software kann hierbei auf unterschiedlichsten Plattformen und unter Verwendung unterschiedlichster Programmiersprachen implementiert sein. Die Struktur und z.T. auch die Bedeutung der gespeicherten Daten ist durch das Tabellenmodell vorgegeben.

Beim Export und beim Import von Daten wurde gezeigt, wie die in Abschnitt 5.3 vorgestellten Abbildungsvorschriften von Objektstrukturen auf eine XML-Datei umgesetzt wurden. Die Realisierung der Funktionalität verbirgt sich hinter der schon vom Persistenzmechanismus bekannten Schnittstelle, da einige Verfahrensweisen ähnlich sind. Auch beim Datenexport werden die Objekte zur Laufzeit untersucht, um die Objektzustände zu ermitteln. Im Rahmen des Datenimports wurden die Vorgänge beim Parsen der XML-Datei und bei der Rekonstruktion der ursprünglichen Objektbeziehungen beschrieben.

In den Erläuterungen zu Abb. 6.1 wurde beschrieben, dass die verschiedenen Stufen des Datenflusses entweder auf einer Netztopologie oder auf einer Punktmenge operieren. Durch die objektorientierte Modellierung ist es möglich, eine Trennung in datenhaltende Entitäten und darauf operierender Algorithmik vorzunehmen. Da die datenhaltenden Entitäten der Algorithmik eine Schnittstelle zum Zugriff auf alle enthaltenen Daten anbieten, kann der gesamte Datenfluss also innerhalb eines Softwaresystems durchgeführt werden. Das Klassenmodell dieser geodätischen Entitäten wird in [53] beschrieben. Durch dieses Klassenmodell wird die Zielsetzung nach einem Datenmodell, auf das von allen Komponenten zugegriffen werden kann, erfüllt.

Bei der Umsetzung der Netzausgleichung wurde gezeigt, wie die Formeln zur Berechnung der Werte in den zur Ausgleichung notwendigen Matrizen und Vektoren in eigenen Klassen gekapselt und somit von den Beobachtungstypen entkoppelt werden können. Hierdurch ist es möglich, für jede Beobachtung die zu verwendende Berechnungsvorschrift vor Beginn der Ausgleichung auszuwählen. Neue Berechnungsvorschriften können dem System auf flexible Art und Weise hinzugefügt werden indem die einzelnen Formeln in eigenen Klassen gekapselt werden. Durch diese Modellierung wird die in Kapitel 2 aufgestellte Zielsetzung der Offenheit des Systems auf der Ebene der funktionalen Modelle erfüllt. Durch die beschriebene Kapselung der Berechnungsvorschriften und da die Klassen zur Modellierung der Beobachtungstypen von einer gemeinsamen Oberklasse erben, konnten die Algorithmen zur Besetzung der Matrizen und Vektoren unabhängig von konkreten Beobachtungstypen gestaltet werden. Durch diesen Ansatz ist es möglich, neue Beobachtungstypen mit nur geringem Aufwand in das System zu integrieren. Hierdurch wird die Zielsetzung der Offenheit des Systems auf der Ebene der Beobachtungstypen erfüllt. Dies kommt besonders zum Tragen, wenn im Rahmen einer Netzausgleichung Beobachtungen aus anderen Geowissenschaften in die Berechnungen einfließen sollen. Die Algorithmen zur Netzausgleichung wurden in eigenständigen Komponenten realisiert und unter Verwendung des Entwurfsmusters *Strategie* in das System integriert, wozu sie mit einer gemeinsamen Schnittstelle versehen wurden.

Es wurde gezeigt, wie die verschiedenen Algorithmen zur Deformationsanalyse unter einer gemeinsamen Schnittstelle zusammengefasst werden konnten, wodurch auch hier das Entwurfsmuster *Strategie* eingesetzt werden konnte. Diese Schnittstelle kann sowohl für statische wie auch für kinematische Deformationsanalysen verwendet werden. Jeder Algorithmus wurde in einer eigenen Komponente realisiert, was die Möglichkeit eröffnet, das System zu späteren Zeitpunkten um neue Algorithmen zu erweitern, ohne Seiteneffekte hinsichtlich der bereits vorhandenen Algorithmen befürchten zu müssen. Für die realisierten Algorithmen wurde eine allgemeine Beschreibung der Vorgehensweise gegeben und es wurde Pseudocode mit zusätzlichen Erklärungen angegeben.

Durch die Verwendung des Entwurfsmusters *Strategie* bei der Integration der Algorithmen zur Netzausgleichung und auch der Deformationsanalyse war es möglich, die Algorithmik in eigenen Komponenten zu kapseln. Hierdurch werden diese austauschbar und neue Algorithmik kann unter Realisierung der jeweiligen Schnittstelle in das System integriert werden. Dieser Ansatz kann auch bei der Realisierung von Komponenten zur Vorverarbeitung verfolgt werden, um die Zielsetzung der Integration des gesamten Datenflusses in einem Programmsystem zu erfüllen.

In einem abschließenden Abschnitt wurde auf die prototypische Realisierung des Systems eingegangen. Hier wurden die Tests der einzelnen Komponenten beschrieben und es wurde ein erster Erfahrungsbericht vom Umgang mit dem Prototypen gegeben.

7.2 Ausblick

Es besteht eine Reihe von Erweiterungsmöglichkeiten, die in nachfolgenden Arbeiten durchgeführt werden können, um den Funktionsumfang des Systems zu erhöhen, und um die bisher nicht betrachteten Teilschritte des Datenflusses einer geodätischen Deformationsanalyse in das System zu integrieren. Diese Erweiterungsmöglichkeiten werden in diesem Abschnitt beschrieben.

Die datenbankgestützte Realisierung des Persistenzmechanismus enthält Zuordnungsvorschriften, um den jeweiligen Klassennamen auf die Tabellen, bzw. um die Attributnamen auf die Spalten der zugehörigen Tabelle in einer relationalen Datenbank abzubilden. Um ein größeres Maß an Flexibilität zu erreichen, wäre es möglich, die Zuordnungsvorschriften für die einzelnen Klassen in einer Datei zu externalisieren. Diese Zuordnungsvorschriften werden dann bei der Initialisierung des Persistenzmechanismus geladen. Zur Externalisierung der Zuordnungsvorschriften bietet sich die XML an, da es mit ihr möglich ist, diese Zuordnungsvorschriften mit einem Texteditor anzulegen oder zu modifizieren. Durch die Verwendung von assoziativen Datenstrukturen können die Zuordnungen zur Laufzeit ermittelt werden, um die notwendigen SQL-Anweisungen zu generieren. Zur Optimierung der Geschwindigkeit des Persistenzmechanismus können auch feste Abbildungsvorschriften für die einzelnen Klassen des Datenmodells in einer weiteren Schicht realisiert werden, die dann gegen die bisherige ausgetauscht wird. Dieses Vorgehen bietet sich an, wenn das Datenmodell so ausgereift ist, dass keine Änderungen daran mehr zu erwarten sind. Ein ähnliches Vorgehen kann auch zur Geschwindigkeitsoptimierung beim Datenexport eingesetzt werden.

Die Komponenten zur Deformationsanalyse wurden mit einer einheitlichen Schnittstelle zur Übergabe der Parameter und zur Ausführung der Analyse versehen. Da ihre Integration in das System unter Verwendung des Entwurfsmusters *Strategie* erfolgte, ist es auf einfache Art und Weise möglich, neue Algorithmen in das System zu integrieren. Solche neuen Algorithmen können unter Verwendung der Klassen zur Modellierung der geodätischen Entitäten erschaffen und auf bereits bestehenden Daten getestet werden. Der Entwickler wird somit von den Aufgaben, die sich im Zusammenhang mit der Ein- und Ausgabe und der Entwicklung notwendiger Datenstrukturen ergeben, entlastet. Prototypen können daher zum Testen einzelner Schritte des Algorithmus leicht in das System integriert werden. Es ist vorstellbar, dass auf dieser Grundlage komplexere, als die bisher vorhandenen Algorithmen entwickelt werden, die auch eine größere Zahl an Daten in die Analyse mit einbeziehen.

Aufgrund der breiten Basis an Klassen zur Modellierung von geodätischen Entitäten ist es möglich, weitere Algorithmik zur Lösung von Problemstellungen aus der Geodäsie zu implementieren und in das System zu integrieren. Hierzu notwendige Erweiterungen an den Klassen sind ohne Seiteneffekte hinsichtlich der bestehenden Algorithmik durchführbar und werden vom Persistenzmechanismus flexibel verarbeitet.

Die vorhandene Benutzerschnittstelle dient bisher lediglich zur Auswahl der Algorithmen und deren Durchführung. Netztopologien oder Punktmengen können bisher nicht angezeigt, eingegeben oder modifiziert werden. Als sinnvolle Erweiterung bietet sich hier die Implementierung einer Visualisierungskomponente an. Diese sollte in der Lage sein, Netztopologien grafisch darzustellen. Zur Anbindung einer solchen Komponente an das System bieten sich - je nach Art der Komponente - verschiedene Verfahrensweisen an. Im Falle einer eher statischen Komponente, die keine Interaktion mit dem Benutzer zulässt, kann die Anbindung unter Verwendung des Entwurfsmusters *Beobachter* [9, 14] geschehen. Die Visualisierungskomponente wird in diesem Fall über Änderungen in der dargestellten Topologie informiert und passt ihre Darstellung daraufhin an. Eine Komponente, die auch zur Interaktion mit dem Benutzer in der Lage sein soll, muss auf andere Art und Weise an das System angebunden werden. Da der Benutzer in die Lage versetzt wird, die Netztopologie und die zugrundeliegenden Daten zu editieren, muss die Visualisierungskomponente in die Lage versetzt werden, Änderungen an der zugrundeliegenden Datenbasis vorzunehmen. Hierzu bietet sich die Verwendung des Entwurfsmusters *Model-View-Controller* [12] an, da es eine klare Trennung nach dem Datenmodell, der Sicht darauf und der Komponente zur Modifikation des Datenmodells vollzieht.

Unabhängig davon, wie die Visualisierungskomponente gestaltet ist, fehlt dem Anwender bisher die Möglichkeit, Daten mit Hilfe von Dialogfenstern in das System einzugeben oder sie auf diese Weise zu modifizieren. Es ist notwendig, die Benutzerschnittstelle dahingehend zu erweitern, dass der Anwender in der Lage ist, geodätische Entitäten, wie z.B. Beobachtungen, Punkte oder auch meteorologische Daten mit Hilfe von Dialogfenstern einzugeben oder zu ändern. Bei der Gestaltung der Dialogfenster ist es wichtig, auf die Zusammenhänge zwischen bestimmten Beobachtungstypen und möglichen Gruppierungen dieser zu achten.

Auch die Übernahme von Daten, die aus firmenspezifischer Software der Gerätehersteller oder von vollautomatischen Erfassungssystemen stammen, ist bisher nicht möglich. Hier wäre es notwendig, Komponenten zu realisieren, die aus solchen firmenspezifischen Darstellungen der Daten eine Netztopologie oder eine Punktmenge erstellen, die mit dem System weiterverarbeitet werden kann.

Um den Einsatz des Systems nicht auf den deutschsprachigen Raum zu beschränken, kann eine Internationalisierung vorgenommen werden. Zu dieser Problemstellung bietet JAVA Mechanismen unter Verwendung assoziativer Datenstrukturen an. Hierbei werden Schlüssel-Wert-Paare genutzt, wobei die Schlüssel im Programmtext referenziert werden und die entsprechenden Textbausteine in der assoziativen Datenstruktur vorgehalten werden. Alternativ wäre auch eine Realisierung durch das in Abschnitt 3.25 beschriebene Entwurfsmuster *Fabrik* möglich, wobei die Textbausteine die Rolle der Produkte übernehmen. Die Lokalisierung, also das Übersetzen in eine andere Sprache, ist in beiden Fällen auf einfache Art und Weise möglich.

Literaturverzeichnis

[1] Autodesk: *Autocad.* http://www.autodesk.de, 2003.

[2] Balzert, H.: *Lehrbuch der Objektmodellierung: Analyse und Entwurf.* Spektrum Akademischer Verlag, Heidelberg, Berlin, Oxford, 1999.

[3] Benning: *KAFKA - 3D Ausgleichungsprogramm für beliebige Lage- und Höhennetze.* http://www.gia.rwth-aachen.de:80/Forschung/Kafka, 2002.

[4] Berzen, N.: *Entwurf und Implementierung eines portablen persistenten Objektspeichers für C++ in heterogenen Rechnerumgebungen.* Doktorarbeit, RWTH Aachen, 2000. Veröffentlichungen des Geodätischen Instituts der Rheinisch-Westfälischen Technischen Hochschule Aachen.

[5] Booch, G., J. Rumbaugh und I. Jacobson: *Das UML-Benutzerhandbuch.* Addison-Wesley, Bonn, 1999. Original: The unified modelling language user guide.

[6] Breining: *Vermessungstechnisches Berechnungsprogramm Geo-Samos.* http://www.breining.de, 2002.

[7] Burg: *KIVID - Kataster- und Ingenieur-Vermessung im Dialog.* http://www.kivid.de, 2002.

[8] Burkhardt, R.: *UML - Unified Modeling Language: objektorientierte Modellierung für die Praxis.* Addison-Wesley, Bonn, 2., aktualisierte Auflage, 1999.

[9] Cooper, J. W.: *The Design Patterns - Java Companion.* Addison-Wesley Design Patterns Series. Addison-Wesley, 1998.

[10] Dinter, G.: *Generalisierte Orthogonalzerlegungen in der Ausgleichungsrechnung*, Band 559 der Reihe *C, Deutsche Geodätische Kommission, München.* 2002.

[11] Eckel, B.: *Thinking in Java.* Prentice Hall, Upper Saddle River, New Jersey, 2000.

[12] Enode_Inc.: *Model-View-Controller Pattern.* http://www.enode.com/x/markup/tutorial/mvc.html, 2002.

[13] Fowler, M. und K. Scott: *UML konzentriert: eine strukturierte Einführung in die Standard-Objektmodellierungssprache.* Addison-Wesley, München, 2., aktualisierte Auflage, 2000.

[14] Gamma, E., R. Helm, R. Johnson und J. Vlissides: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, München, 5., korrigierte Auflage, 2001. Nachdruck.

[15] Geary, D.: *Strategy for success.* http://www.javaworld.com/javaworld/jw-04-2002/jw-0426-designpatterns_p.html, 2002. Version - 26. April 2002.

[16] Geodätisches Institut, Universität Karlsruhe: *CODEKA - Koordinatenbezogene Deformationsanalyse.* http://www.gik.uni-karlsruhe.de/software/deform.html, 2002.

[17] Geodätisches Institut, Universität Karlsruhe: *Netz2D - Ausgleichung zweidimensionaler Netze mit GPS-Integration.* http://www.gik.uni-karlsruhe.de/software/netzaus.html, 2002.

[18] Gosling, J., B. Joy und Guy Steele: *Java(TM) Die Sprachspezifikation.* Addison-Wesley-Longman, Bonn, 1997.

[19] Hamilton, G.: *JDBC - Datenbankzugriff mit Java - die offizielle Dokumentation von JavaSoft.* Java - Series. Addison-Wesley-Longman, Bonn, 1998.

[20] Hartmann, P.: *Strainanalyse ebener geodätischer Deformationsnetze.* Diplomarbeit, Universität Karlsruhe (TH), 1985. 666 - GIK (unveröffentlicht).

[21] Heck, B.: *Rechenverfahren und Auswertemodelle der Landesvermessung.* Herbert Wichmann Verlag, Karlsruhe, 1995.

[22] Heuer, A. und G. Saake: *Datenbanken: Konzepte und Sprachen.* mitp, Bonn, 2., aktualisierte und erweiterte Auflage, 2000.

[23] Hicklin, J., R. F. Boisvert, C. Moler, B. Miller, P. Webb, R. Pozo und K. Remington: *JAMA: A Java Matrix Package.* http://math.nist.gov/javanumerics/jama, 2000.

[24] Höpcke, W.: *Fehlerlehre und Ausgleichsrechnung.* Walter de Gruyter, Berlin, New York, 1980.

[25] IBM_DB2. http://www-3.ibm.com/software/data/db2, 2001.

[26] Jäger, R. und E. Drixler: *Deformationsanalyse - Verfahren am Geodätischen Institut der Universität Karlsruhe; Konzepte - Vergleiche - Software - Ausblick; Stand Februar 1990.* Interner Bericht. Geodätisches Institut der Universität Karlsruhe (GIK), 1990.

[27] Java: *The Source for Java(TM) Technology.* http://java.sun.com. Sun Microsystems.

[28] Koch, G. und K. Loney: *Oracle8 - Die umfassende Referenz.* Carl Hanser Verlag, München, Wien, 1998.

[29] Koch, K.-R.: *Parameterschätzung und Hypothesentests in linearen Modellen.* Ferdinand Dümmler Verlag, Bonn, 3., bearbeitete Auflage, 1997.

[30] Krüger, G.: *Go to Java 2: Handbuch der Java-Programmierung.* Addison-Wesley, München, 2. Auflage, 2000.

[31] Lang, S. M. und P. C. Lockemann: *Datenbankeinsatz.* Springer - Verlag, Berlin, Heidelberg, 1995.

[32] Lans, R. F. van der: *Introduction to SQL: mastering the relational database language.* Addison-Wesley, Harlow, München, 3. Auflage, 2000.

[33] Leica_Geosystems: *Geodätisches Monitoring System GeoMos.* http://www.leica-geosystems.com/surveying/product/software/geomos_de.html, 2001.

[34] Lockemann, P. C., G. Krüger und H. Krumm: *Telekommunikation und Datenhaltung.* Carl Hanser Verlag, München, Wien, 1993. unter Mitwirkung von Radermacher, K. und Schill, A.

[35] Ludwig, H.: *Ausgleichungsrechnung Ib.* Skriptum zur Vorlesung an der Fachhochschule Würzburg, 1996.

[36] Marsch, J. und J. Fritze: *Erfolgreiche Datenbankanwendung mit SQL: effiziente Wege und zielführendes Know-how für den professionellen Einsatz.* Vieweg, Braunschweig, Wiesbaden, 5., überarbeitete Auflage, 1999.

[37] McLaughlin, B.: *Java und XML.* O'Reilly Verlag, Peking, Köln, 2. Auflage, 2002.

[38] Mierlo, J. van: *Ausgleichungsrechnung III.* Skriptum zur Vorlesung am Geodätischen Institut der Universität Karlsruhe (GIK), 1998.

[39] MySQL. http://www.mysql.com, 2001.

[40] Niemeier, W.: *Ausgleichungsrechnung: eine Einführung für Studierende und Praktiker des Vermessungs- und Geoinformationswesens.* Walter de Gruyter, Berlin, New York, 2002.

[41] Niemeier, W. und D. Tengen: *PANDA The Software Package for Precise Engineering Networks.* http://www.slac.standford.edu/grp/met/TOC_S/Papers/WNiem90.pdf, 1990. Version - September 1990.

[42] Nkuite, G.: *Ausgleichung mit singulärer Varianzkovarianzmatrix am Beispiel der geometrischen Deformationsanalyse*, Band 501 der Reihe *C, Deutsche Geodätische Kommission, München.* 1998.

[43] Obergfell, K.: *Automatisierte Stützpunktsuche nach dem Karlsruher Deformationsanalyse Modell.* Diplomarbeit, Universität Karlsruhe (TH), 1985. 697 - GIK (unveröffentlicht).

[44] Oestereich, B.: *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language.* R. Oldenbourg Verlag, München, 5., völlig überarbeitete Auflage, 2001.

[45] Oracle_Corporation. http://www.oracle.com, 2001.

[46] Pelzer, H.: *Zur Analyse geodätischer Deformationsmessungen*, Band 164 der Reihe *C, Deutsche Geodätische Kommission, München.* 1971.

[47] Pelzer, H.: *Deformationsuntersuchungen auf der Basis kinematischer Bewegungsmodelle.* Allgemeine Vermessungsnachrichten (AVN), 94(2):49–62, 1987.

[48] PostgreSQL. http://www.postgresql.org, 2001.

[49] Rawiel, P.: *Dreidimensionale kinematische Modelle zur Analyse von Deformationen an Hängen*, Band 533 der Reihe *C, Deutsche Geodätische Kommission, München.* 2001.

[50] Rumbaugh, J.: *Objektorientiertes Modellieren und Entwerfen.* Carl Hanser Verlag, München, Wien, 1994. deutsche Ausgabe besorgte Märtin, Doris.

[51] Rumbaugh, J., I. Jacobson und G. Booch: *The unified modeling language reference manual: UML.* Addison-Wesley, 1999.

[52] Schader, M. und L. Schmidt-Thieme: *Java - Einführung in die objektorientierte Programmierung.* Springer - Verlag, Berlin, Heidelberg, New York, Barcelona, Budapest, Hongkong, London, Mailand, Paris, Santa Clara, Singapur, Tokio, 1998.

[53] Schmidt, U.: *Modellierung der mehrdimensionalen multisensoralen objektorientierten Deformationsanalyse.* Geodätisches Institut der Universität Karlsruhe (GIK), 2003. Dissertation (in Druck).

[54] Technet_GmbH: *Neptan - Ausgleichung und Analyse geodätischer Netze.* http://www.technet-gmbh.com, 2002.

[55] Wiechert, H.: *Theoretische Untersuchungen und programmtechnische Umsetzung der Sensitivitätsanalyse bei STRAIN.* Diplomarbeit, Universität Karlsruhe (TH), 1996. 722 - GIK (unveröffentlicht).

[56] Wolf, H.: *Ausgleichungsrechnung - Formeln zur praktischen Anwendung.* Ferdinand Dümmler Verlag, Bonn, 1975.

[57] Wolf, H.: *Ausgleichungsrechnung - Aufgaben und Beispiele zur praktischen Anwendung.* Ferdinand Dümmler Verlag, Bonn, 1979.

[58] Zippelt, K.: *Modellbildung, Berechnungsstrategie und Beurteilung von Vertikalbewegungen unter Verwendung von Präzisionsnivellements*, Band 343 der Reihe *C, Deutsche Geodätische Kommission, München.* 1988.