GERHARD ANDREAS PÜLZ

AdConsol-1D

Entwicklung eines offenen und erweiterbaren Programms
zur Untersuchung neuer Algorithmen
der Berechnung des zeitlichen Setzungsverhaltens
mehrschichtiger Böden

Diplomarbeit an den Lehrstühlen *Informatik im Bauwesen & Bodenmechanik*

BAUHAUS - UNIVERSITÄT WEIMAR

In dieser Arbeit werden eingetragene Warenzeichen, Handelsnamen und Gebrauchsnamen verwendet. Auch wenn diese nicht als solche gekennzeichnet sind, gelten die entsprechenden Schutzbestimmungen.

Vorwort

Wenn es nur eine einzige Wahrheit gäbe,
könnte man nicht hundert Bilder über dasselbe Thema malen.

[Picasso]

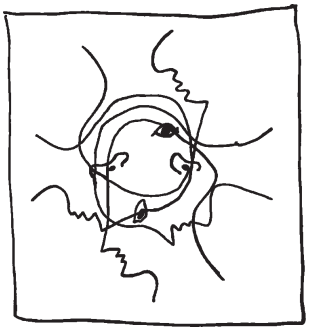

Bild 1: "Das unerkannte Meisterwerk" *Picasso (1926)*

Versuchen wir die Informatik als das zu nehmen, was sie ist: Die simple Aneinanderreihung einfacher, logischer Regeln. Wer hinter komplexen Systemen die Einfachheit sieht, wird sie verstehen. Es gibt bei der Programmierung nicht die eine Wahrheit und auch nicht den einen richtigen Weg - oftmals ist die einfachste Lösung gleichsam die Beste. Um zu dieser Erkenntnis zu gelangen, bedarf es einer vielseitigen Betrachtung des Problems.

Computer können uns das Verstehen, Planen und Entwickeln nicht abnehmen. Ohne die von Menschen erdachten Programme sind Computer nichts als nutzlose Maschinen. Gleich einer Hilfskraft können sie uns jedoch wiederkehrende Prozesse abnehmen und dadurch mühsame Arbeiten beschleunigen.

Wem es gelingt seine Denkweisen in ein einfaches System zu bringen, gehört zu den wahren Entwicklern in der modernen, elektronischen Welt.

Weimar, im Mai 2003
Gerhard Andreas Pülz

Inhalt

Vorwort **I**

Inhaltsverzeichnis **V**

Abkürzungsverzeichnis **VII**

Tabellenverzeichnis **IX**

Abbildungsverzeichnis **XI**

Literaturverzeichnis **XIV**

1 Einführung **1**
1.1 Einleitung . . . 1
1.2 Motivation . . . 1
1.3 Aufbau der Arbeit . . . 2

2 Konsolidationstheorie **5**
2.1 Einführung . . . 5
2.2 Berechnung in *AdConsol-1D* . . . 5
2.3 Setzungsberechnung . . . 6
2.3.1 Methoden der Setzungsberechnung . . . 6
2.3.2 Setzungsberechnung in *AdConsol-1D* . . . 6
2.4 Konsolidation . . . 10
2.4.1 Gegenstand . . . 10
2.4.2 Porendruck-Formulierung . . . 10
2.4.3 Konsolidationsberechnung in *AdConsol-1D* . . . 11
2.4.4 Geschichteter Boden . . . 12
2.5 Formelzeichen . . . 14

3 Allgemeine Umsetzung **15**
3.1 Problematik des Prototypings . . . 15
3.2 Objektorientierte Programmierung . . . 15
3.2.1 Einführung in die OOP . . . 15
3.2.2 Begriffe der OOP . . . 16
3.2.3 Vorteile der OOP . . . 17
3.2.4 Merkmale einer objektorientierten Sprache . . . 17

3.3 Anforderungen an ein Windows-Programm . . . 18
3.3.1 Notwendigkeit einer Windows gerechten Oberfläche . . . 18
3.3.2 Grafische Interaktion zwischen Anwender und Programm . . . 19
3.3.3 Richtlinien einer MDI Anwendung . . . 19
3.4 Implementierung der Berechnungsalgorithmen . . . 20
3.4.1 Skripting . . . 20
3.4.2 Berechnungsklasse . . . 21

4 Die Programmiersprache *Visual Basic* 23
4.1 Einführung in *Visual Basic* . . . 23
4.1.1 Historische Entwicklung von *Visual Basic* . . . 23
4.1.2 *Visual Basic* als Entwicklungssystem . . . 25
4.2 Objektorientierung in *Visual Basic 6* . . . 25
4.2.1 Abstraktion - abstraction . . . 25
4.2.2 Kapselung - encapsulation . . . 26
4.2.3 Vererbung - object inheritance . . . 26
4.2.4 Polymorphie - polymorphism . . . 27
4.2.5 Ereignis - event . . . 28
4.2.6 Konstruktoren und Destruktoren . . . 29
4.2.7 Wichtige Objekte in *Visual Basic* . . . 29
4.3 Programmstrukturierung in *Visual Basic* . . . 30
4.3.1 Modultypen . . . 30
4.3.2 Unterprogramme . . . 30
4.4 Namenskonventionen . . . 31
4.5 Grunddatentypen . . . 32
4.6 Einschätzung der Sprache *Visual Basic* . . . 32

5 Umsetzung von *AdConsol-1D* 35
5.1 Programmentwurf . . . 35
5.1.1 Eingabe . . . 35
5.1.2 Ausgabe . . . 36
5.1.3 Speicherung der Eingabedaten . . . 38
5.1.4 Hilfedatei . . . 39
5.2 Modellierung der Datenstruktur . . . 39
5.2.1 Vorgehensweise . . . 39
5.2.2 Eingabe-Klassen . . . 40
5.2.3 Container-Klassen . . . 40
5.2.4 Klasse InputGenerator . . . 41
5.2.5 Berechnungs-Klasse . . . 41
5.2.6 Visualisierungs-Klasse . . . 41
5.2.7 Andere Module . . . 42
5.3 Programmstruktur . . . 43
5.3.1 Aufbau des Programms . . . 43
5.3.2 Klassenmodell . . . 44
5.3.3 Aufbau der einzelnen Klassen . . . 46
5.4 Berechnungsklasse . . . 55

5.4.1 Aufbau der Berechnungsklasse . . . 55
5.4.2 Beschreibung der Variablen . . . 58
5.4.3 Ergebnisse der Konsolidationsberechnung . . . 60
5.5 Grafik in *AdConsol-1D* . . . 61
5.5.1 Grafikprogrammierung unter Windows . . . 61
5.5.2 Einführung in die *OpenGL* . . . 62
5.5.3 *OpenGL* Programmierung . . . 63

6 Ausblick **69**
6.1 Erweiterungsmöglichkeiten . . . 69
6.1.1 Gitteroptimierung . . . 69
6.1.2 Mehrere Bodenprofile . . . 69
6.1.3 Programmregistrierung . . . 70
6.1.4 Erweiterungen der Ergebnisausgabe . . . 70
6.1.5 Zeitlich variabler Grundwasserspiegel . . . 70
6.2 Beschleunigung der Berechnung . . . 70
6.2.1 Optimierung des Programmcodes . . . 70
6.2.2 Erstellung eigener DLLs . . . 71
6.3 Komponentenbasierte Softwareentwicklung . . . 71
6.3.1 Software-Komponenten . . . 71
6.3.2 Geschichte . . . 71
6.3.3 Softwareentwicklung heute . . . 71

Zusammenfassung **73**

A Inhalt der CD **75**

B Systemanforderungen & Installation **77**
B.1 Systemanforderungen . . . 77
B.2 Installation . . . 78

C Ausdruck der Hilfedatei **79**

Erklärung **i**

Thesen **iii**

Abkürzungsverzeichnis

API	Application Programming Interface = *Anwendungsprogrammierschnittstelle*
ASCII	American Standard Code of Information Interchange
BASIC	Beginners Allpurpose Symbolic Instruction Code
COM	Component Object Model = *Komponentenobjektmodell*
DBMS	Data Base Management System = *Datenbankverwaltungssystem*
DCOM	Distributed Component Object Model
DIN	Deutsche Industrie Norm
DLL	Dynamic Link Libraries = *Programmbibliothek*
DNA	Distributed Net Application Architecture
EDV	Elektronische Datenverarbeitung
EVA	Eingabe - Verarbeitung - Ausgabe
etc.	et cetera = *und so weiter*
exe	executable Program = *ausführbares Programm*
FDM	Finite Differenzen Methode
FEM	Finite Elemente Methode
MDI	Multiple Document Interface
MTS	*Microsoft* Transaction Server
MVC	Model - View - Controller
ODBC	Open Data Base Connectivity = *Zugriffsmechanismus auf Datenbanken*
OOP	Objektorientierte Programmierung
RAD	Rapid Application Development
VB	*Visual Basic*
z.B.	*zum Beispiel*

Tabellen

2.1 Verwendete Formelzeichen in der Konsolidationstheorie 14

4.1 Namenskonventionen in *Visual Basic* 31
4.2 Grunddatentypen in *Visual Basic* 32

5.1 Tabellarische Darstellung der Klasse clsKern 46
5.2 Tabellarische Darstellung der Klasse clsInputGenerator 46
5.3 Tabellarische Darstellung der Klasse clsField 47
5.4 Tabellarische Darstellung der Klasse clsOptions 47
5.5 Tabellarische Darstellung der Klasse clsLoadList 48
5.6 Tabellarische Darstellung der Klasse clsLoad 49
5.7 Tabellarische Darstellung der Klasse clsPointList 50
5.8 Tabellarische Darstellung der Klasse clsPoint 50
5.9 Tabellarische Darstellung der Klasse clsSoilList 51
5.10 Tabellarische Darstellung der Klasse clsSoil 52
5.11 Tabellarische Darstellung der Klasse clsCalculation 53
5.12 Tabellarische Darstellung der Klasse clsVisualisation 53
5.13 Tabellarische Darstellung der Klasse cls3DModell 54
5.14 Tabellarische Darstellung der Klasse clsDiagramm 54
5.15 Tabellarische Darstellung der Klasse clsIsoSurface 55
5.16 Tabellarische Darstellung der Klasse clsWaitCursor 55
5.17 Berechnungsvariablen 58
5.18 Eindimensionale Berechnungs-Arrays 58
5.19 Bodenprofil Arrays 59
5.20 Zweidimensionale Berechnungs-Arrays 60
5.21 Eindimensionale Ergebnis-Arrays 60
5.22 Zweidimensionale Ergebnis-Arrays 61
5.23 *OpenGL* Namenskonventionen 63
5.24 *OpenGL* Grafikprimitive 65

Bilder

1 "Das unerkannte Meisterwerk" *Picasso (1926)* I

2.1 Schlaffe und starre Last 7
2.2 Superposition von Lastflächen 7
2.3 Abschnitte aus der Oedometerkurve zur Bestimmung des Funktionals . . . 9
2.4 Punkte des Berechnungsgitters über die Zeit und Tiefe 12

4.1 *Visual Basic*-Logo 24

5.1 Bodenschichtmodell in *AdConsol-1D* 36
5.2 Anordnung von Last- und Bodenmodell in *AdConsol-1D* 37
5.3 Isofläche und 3DModell in *AdConsol-1D* 38
5.4 Onlinehilfe von *AdConsol-1D* 39
5.5 Darstellung der Container-Klassen am Beispiel *clsPointList* 41
5.6 Darstellung der Klasse *clsInputGenerator* 42
5.7 Darstellung der Klasse *clsVisualisation* 43
5.8 Klassenmodell von *AdConsol-1D* 45
5.9 Blockdiagramm der Ablaufsteuerung 56
5.10 Darstellung der Berechnungsvariablen 57
5.11 *OpenGL*-Logo 62
5.12 Translation in *OpenGL* 66
5.13 Rotation in *OpenGL* 66
5.14 Skalierung in *OpenGL* 66
5.15 Kameraposition in *AdConsol-1D* 67
5.16 Perspektive in *OpenGL* 68

Literaturverzeichnis

Barth, R. (2002), Visualisierung und graphische Standards-Teil 1 - OpenGL, Technical report, Institut für Praktische Informatik und Medieninformatik.

Doberenz/Kowalski (1999*a*), *Programmieren lernen in Visual Basic*, Hanser Verlag, München.

Doberenz/Kowalski (1999*b*), *Visual Basic 6 Kochbuch*, Hanser Verlag, München.

edx (April 2003), 'Internetseite', http://is6.pacific.net.hk/ edx/contents.htm.

EVB (1993), *Empfehlungen Verformungen des Baugrundes bei baulichen Anlagen EVB*, Ernst & Sohn, Berlin.

Gnoth, M. (2000), Charitime, Master's thesis, Humboldt Universität zu Berlin.

Gnoth, M. (April 2003), 'Internetseite', http://www.gnoth.net/Job/WebTech/VBPage.html.

Holzer, S. M. (1997), 'Gestaltung ingenieurgemäßiger Statiksoftware', *Bauingenieur-Bauinformatik* **Heft 3**.

Kirstein, M. (1. Auflage 1998), *Visual Basic 6*, Data Becker, Düsseldorf.

Krause, G. (1977), 'Konsolidationstheorien im Rahmen der Mechanik der Gemische', *Die Bautechnik* **3**(3), 95 – 103.

Krumsiek, J. (Mai 2003), 'Internetseite Programmiersprachenstreit', http://www.activevb.de/tutorials/tut_sprachenstreit/programmiersprachen.html.

Kuska, J.-P. (2000), 'Graphik-Programmierung mit OpenGL'.

Melchart/Pojar (April 2003), 'Internetseite', http://3dcoders.bugpower.ch/ opengluebe r.htm.

Microsoft Corp. (2000), 'MSDN Library Visual Studio', Visual Basic CD.

Mönch/Queck (2001), Programmieren mit OpenGL, Technical report, Institut für Angewandte Mathematik II, Freiberg.

Morris, P. H. (2002), 'Analytical solutions of linear finite-strain one-dimensional consolidation', *Journal of Geotechnical and Geoenvironemental Engineering* **128**(4), 319 – 325.

Packulat (April 2003), 'Internetseite', http://homepages.compuserve.de/DiplPackulat/Default.htm.

Petroutsos, E. (1. Auflage 1998), *Visual Basic 6 - Das Buch*, Sybex Verlag, Düsseldorf.

Schels/Schels (2000), *Programmieren lernen*, Markt & Technik Verlag, München.

Schmidt, H.-H. (1996), *Grundlagen der Geotechnik: Bodenmechanik - Grundbau*, B. G. Teubner, Leipzig, Stuttgart.

Schubert, D. (Mai 2003), 'Internetseite Basicstory', http://www.activevb.de/tutorials/tut_basicstory/basicstory.html.

Schwetlick/Kretzschmar (1991), *Numerische Verfahren für Naturwissenschaftler und Ingenieure*, Fachbuch Verlag GmbH Leipzig.

Shadoware (April 2003), 'Internetseite', http://www.shadoware.de/.

Teetz, M. (2002), 'Entwicklungssysteme im Vergleich', *IT FOKUS* .

Terlinden, A. (1996), Objektorientierte Programmierung mit Visual Basic, Technical Report 108, Fachbereich Wirtschaftsinformatik.

Winkler, P. (2002), *Computerlexikon*, Markt & Technik Verlag, München.

Kapitel 1

Einführung

1.1 Einleitung

Bereits die Baumeister der Antike wussten, dass ein solider Baugrund im wahrsten Sinne des Wortes die beste 'Grundlage' für ein dauerhaftes Bauwerk darstellt. Welcher Stellenwert dem Baugrund schon früher zugeordnet wurde, lässt sich erahnen, wenn man unseren Wortschatz etwas näher betrachtet. So findet man Redewendungen wie *etwas ist auf Sand gebaut* oder *frisch gelerntes muss sich setzen*. Diese Metaphern aus dem Ingenieurbereich helfen in einem neuen Kontext verwendet Sachverhalte zu veranschaulichen.

Die Setzungsproblematik beschäftigt heutzutage Ingenieure nicht nur am "schiefen Turm von Pisa". Setzungen können überall auftreten, bei Neubauten genauso wie an historischen Gebäuden. Absinkendes Grundwasser, veränderte Grundwasserströme oder falsch eingeschätzte Baugrundsituationen können Ursachen für ungewollte Bodenverformungen sein. Das Problem ist, dass Setzungen in der Regel nicht plötzlich auftreten, sondern der Untergrund oft schleichend über Jahre hinweg konsolidiert.

1.2 Motivation

Die Berechnung von zeitabhängigen Verformungen mehrschichtiger Böden ist ein komplexes Thema der Geotechnik. Die Schwierigkeit liegt vor allem darin, dass die Berechnungen im dreidimensionalen Raum zugleich orts- als auch zeitabhängig ablaufen. Es liegt nahe, diese rechenintensive Aufgabe computergestützt zu erledigen.

Ein weiteres Problem ist, dass nur für einfache Problemstellungen der Konsolidationsrechnung geschlossene Lösungen existieren. Größere Probleme erfordern numerische Lösungswege. Dazu ist es nötig, neue Algorithmen zu entwickeln oder Vorhandene zu modifizieren.

Ziel dieser Diplomarbeit ist es, ein *Windows*-Programm zu entwickeln, mit dem neue Algorithmen für die Berechnung des zeitlichen Setzungsverhaltens mehrschichtiger Böden untersucht werden können. Als Grundlage dient hierfür das Programm *AdConsol-1D* Version 1, das bereits im Sommer 2002 als Studienarbeit angefertigt wurde. Dieses Programm soll komplett überarbeitet und erweitert werden. Gegenüber der Vorgängerversion sind

dabei folgende funktionalen Erweiterungen vorgesehen:

- Unbegrenzte Anzahl von Bodenschichten bei konstantem Bodenprofil im Untersuchungsgebiet
- Unbegrenzte Anzahl von beliebig positionierbaren, konstanten, zeitabhängigen Teilflächenlasten
- Verformungsberechnung eines beliebig positionierbaren Punktes im Untersuchungsgebiet ('Punktberechnung')
- Verformungsberechnung des Gesamtgebietes ('Feldberechnung')
- Grafische Darstellung der Verformungen im Berechnungsmodus 'Feldberechnung' als Isofläche und 3D-Oberfläche mit veränderbarem Ansichtspunkt
- MDI-Anwendung mit Projektverwaltung (Laden, Bearbeiten, Abspeichern, Workspace)

Die Programmstruktur ist so zu gestalten, dass *AdConsol-1D* leichter erweiterbar, anpassbar und wartbar wird. Dazu ist ein objektorientiertes Datenmodell (Generator-, Kern-, Berechnungs- und Visualisierungsmodell) zu entwickeln. Zum Testen neuer Berechnungsalgorithmen muss es leicht möglich sein diese auszutauschen. Die Programmoberfläche von *AdConsol-1D* soll 'windows-konform' gestaltet werden und mit Maus oder Tastatur bedienbar sein. Eine Möglichkeit zum Einlesen und Speichern von Daten ist im Programm vorzusehen. Zur Gewährleistung der Modellkonsistenz sind geeignete Plausibilitätskontrollen ins Programm zu integieren. Damit das Programm gut weitergegeben werden kann, ist eine Setup-Routine und Hilfedatei erwünscht.

Das Programm ist in *Microsoft Visual Basic* zu implementieren und besitzt Prototypcharakter. Die Diplomarbeit schließt die Entwicklung des Programms zum größten Teil ab.

1.3 Aufbau der Arbeit

Grundlage für die Umsetzung des Programms ist das Verständnis des geotechnischen Problems. Aus diesem Grund wird am Anfang dieser Arbeit kurz auf die Konsolidationstheorie eingegangen. Im Anschluss folgt ein allgemeiner Teil, in dem die objektorientierte Programmierung grob erläutert wird und die Anforderungen an ein modernes Windows-Programm erörtert werden. Zum Testen neuer Berechnungsalgorithmen sollen diese in *AdConsol-1D* leicht austauschbar sein. Welche Möglichkeiten es dafür gibt, wird ebenfalls in diesem Teil der Arbeit erläutert. Das darauffolgende Kapitel befasst sich mit der Programmiersprache *Visual Basic*. Hier wird untersucht inwiefern *Visual Basic 6* zur Umsetzung geeignet ist und wie *Visual Basic* Programme sinnvoll strukturiert werden können. Die Erläuterung der Umsetzung von *AdConsol-1D*, vom Programmentwurf über die Modellierung der Datenstruktur bis hin zur Grafikprogrammierung, ist Gegenstand des letzten Teils dieser Arbeit. Im Ausblick werden Möglichkeiten zur Erweiterung und Beschleunigung des

Programms vorgestellt und abschließend die Ergebnisse dieser Arbeit kurz zusammengefasst. Einen guten Überblick, welche Programmdetails in dieser Diplomarbeit umgesetzt wurden und was *AdConsol-1D* leistet, bietet die 'Onlinehilfe'. Im Anhang liegt diese in gedruckter Form vor. Ebenso sind hier die Systemanforderungen für *AdConsol-1D* und eine Installationsanleitung zu finden.

Kapitel 2

Konsolidationstheorie

Seit die Mathematiker über die Relativitätstheorie hergefallen sind,
verstehe ich sie selbst nicht mehr.

[Albert Einstein]

Anmerkung:

Dieses Kapitel basiert auf den noch unveröffentlichten Ansätzen zur Konsolidationsrechnung von *Dipl. -Ing. Martin Zimmerer*.

2.1 Einführung

Setzungen entstehen im Wesentlichen infolge einer Verringerung des Porenvolumens bei Lastaufbringung. Sind alle Poren des Bodens mit Wasser gefüllt, muss das Porenwasser beim Zusammendrücken entweichen. Die Durchlässigkeit nichtbindiger Böden (drainierte Böden) ist so groß, dass das Porenwasser sofort entweichen kann. Die Setzungen stellen sich somit fast augenblicklich mit der Lastaufbringung ein. Bei bindigen Böden (undrainierte Böden) dagegen ergibt sich aufgrund der geringeren Durchlässigkeit nur ein langsames Entweichen des Porenwassers. Damit setzt sich auch der Boden nur langsam.

Für den Fall, dass der Boden wassergesättigt und der Spannungszustand in jedem vertikalen Schnitt gleich ist, kann das Porenwasser nur in vertikaler Richtung entweichen. Es handelt sich um eindimensionale Konsolidation. Hierfür gibt *Terzaghi* ein anschauliches mechanisches Modell mit einer geschlossenen mathematischen Lösung an.

2.2 Berechnung in *AdConsol-1D*

In *AdConsol-1D* wird die Möglichkeit gegeben, das zeitliche Setzungsverhalten mit Hilfe von spannungsabhängigen Steifigkeiten und porenzahlabhängigen Durchlässigkeiten zu

berechnen. Die Konsolidationstheorie stützt sich dabei auf die Fließgesetze von *Darcy* und die Konsolidationstheorie von *Terzaghi*.

Zur Berücksichtigung von spannungsabhängigen Steifemodulen und porenzahlabhängigen Durchlässigkeiten wurde die *Terzaghi*-Theorie ingenieurmäßig modifiziert. Auf die neue Herleitung einer umfassenden partiellen Differentialgleichung des Konsolidationsverhaltens mit dem Porenwasserdruck als abhängige Variable wurde verzichtet. Vielmehr wird in jedem Zeitschritt ein neuer Steifemodul, eine neue Durchlässigkeit und ein neuer Drainweg berechnet.

2.3 Setzungsberechnung

2.3.1 Methoden der Setzungsberechnung

Grundsätzlich kann die Setzungsberechnung nach DIN 4019 auf zwei Arten erfolgen. Mit Hilfe der direkten Setzungsermittlung, die auf einer Setzungsformel basiert, oder über eine indirekte Setzungsberechnung, die mit Hilfe der lotrechten Spannungen im Baugrund arbeitet.

2.3.2 Setzungsberechnung in *AdConsol-1D*

In *AdConsol-1D* wird ausschließlich die Methode der indirekten Setzungsermittlung angewandt. Dabei wird vereinfachend angenommen, dass die Verteilung der vertikalen Spannung $\sigma(z)$ im Boden von dessen Materialeigenschaften unabhängig ist. Im Zuge der Berechnungen nach der indirekten Methode müssen die Spannungen $\sigma(z)$ unter dem betrachteten Punkt aus Eigenlast des Bodens und der Bauwerkslast über die Berechnungstiefe z berechnet werden. (Schmidt 1996)

Die Spannungsabminderung über die Tiefe bei einer begrenzten Lasteintragung kann dabei mit Hilfe verschiedener Funktionen berechnet werden. In *AdConsol-1D* wurden die Formeln von *Steinbrenner* implementiert.

$$i\,(z) = \frac{1}{2\pi}\left[\arctan\left(\frac{ab}{z\sqrt{a^2+b^2+z^2}}\right) + \frac{abz}{\sqrt{a^2+b^2+z^2}}\left(\frac{1}{a^2+z^2}+\frac{1}{b^2+z^2}\right)\right] \tag{2.1}$$

dabei gilt:

a = Belastungslänge
b = Belastungsbreite
z = Tiefe im Boden

Aus Gleichung 2.1 wird ersichtlich, dass die Funktion von Steinbrenner an der Schichtoberkante bei $z = 0$ keine Ergebnisse liefern kann. Daher wird im Programm der Spannungseinfluss an der Schichtoberkante gleich dem einfachen Wert der aufgetragenen Spannung angenommen ($i(0) = 1$).

Für die Spannung $\sigma(z)$ in der Tiefe z ergibt sich damit:

$$\sigma(z) = \sigma_0 \cdot i(z) \tag{2.2}$$

dabei gilt:

σ_0 = Spannung an der Oberfläche
$i(z)$ = Steinbrennerwert nach Gleichung 2.1

Diese Art der Spannungsabminderung über die Tiefe kann direkt nur für schlaffe Lasten beziehungsweise biegeweiche Fundamente angewendet werden. Die dabei auftretenden Setzungsunterschiede von Lastflächenmitte (Fundamentmitte) und Lastflächenrand (Fundamentrand) gleichen sich jedoch bei starren Bauwerken je nach Grad der Steifigkeit der Lasteintragungsfläche aus, da die auftretenden Biege- und Schubbeanspruchungen in der Konstruktion aufgenommen werden (siehe Bild 2.1). (EVB 1993)

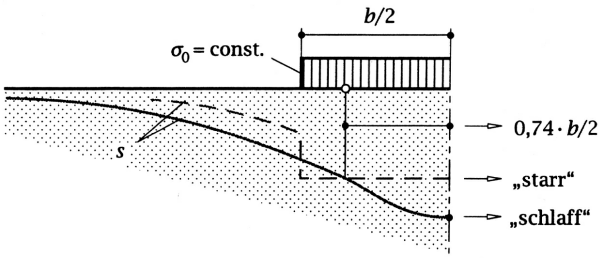

Bild 2.1: Schlaffe und starre Last

Die Ermittlung des Spannungseinflussbeiwertes $i(z)$ erfolgt bei mehreren vorhandenen Lastflächen nach dem Superpositionsprinzip. So kann man durch Überlagerung die Vertikalspannungen an beliebigen Stellen komplizierter Belastungsflächen ermitteln. Bild 2.2 soll dies verdeutlichen. Die Spannungseinflussbeiwerte werden hier jeweils für den kennzeichnenden Punkt (KP) ermittelt.

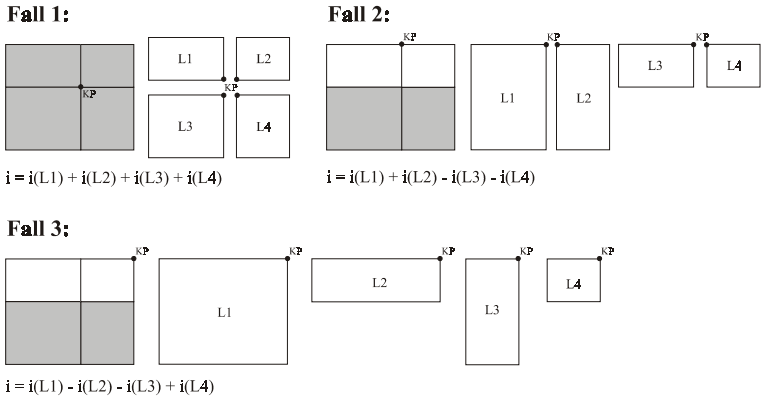

Bild 2.2: Superposition von Lastflächen

Im vollständig wassergesättigten Boden unterscheidet man folgende Vertikalspannungen:

Totale Spannung $\sigma(z)$**:** Spannung infolge Gewichtskraft der Festmasse des Bodens, der Gewichtskraft des Wassers und der Bodenpressung durch äußere Lasten $\Delta\sigma(z)$ im betrachteten Horizont z.

Porenwasserdruck $u_{stat}(z)$**:** hydrostatischer Wasserdruck im betrachteten Horizont z. $u_{stat}(z) = \gamma_w \cdot h_w$ mit h_w = hydrostatische Druckhöhe und γ_w = Wasserwichte.

Porenwasserüberdruck $u(z,t)$**:** Porenwasserspannung aus äußeren Lasten im Horizont z, die über die Zeit t an das Korngerüst abgegeben wird.
$u(z,0) = \Delta\sigma(z)$; $u(z,t) = \Delta\sigma(z) - \Delta\sigma'(z,t)$

Effektive Spannung $\sigma'(z,t)$**:** Spannung die im betrachteten Horizont z zum Zeitpunkt t im Korngerüst herrscht:

$$\sigma'(z,t) = \sigma(z) - [u_{stat}(z) + u(z,t)] \tag{2.3}$$

Der Baugrund wird in der indirekten Setzungsberechnung auf Grundlage einer Bodenerkundung in mehrere Bodenschichten mit konstanten Steifigkeiten unterteilt. Die entsprechenden Steifemodule werden hierbei aus Oedometerversuchen abgeleitet. Der ständige Abgleich (Spannung - Steifigkeit) mit den Versuchsdiagrammen kann in der nachfolgenden Berechnung entfallen, da die Steifigkeiten als Tangentenmodule näherungsweise durch die Funktion aus dem Ohdschen Ansatz beschrieben werden.

$$E_s = E_{oed}^{ref} \left(\frac{\sigma'(z) + \sigma_\gamma(z)}{\sigma_{ref}} \right)^m \tag{2.4}$$

dabei gilt:

$\sigma_\gamma(z)$ = Spannung aus Bodeneigengewicht
E_{oed}^{ref} = normierte Steifigkeit
σ_{ref} = Referenzspannung aus dem Ansatz von *Ohde*
m = Exponent aus Regressionskoeffizienten einer Näherungsfunktion zur Bestimmung von E_{oed}^{ref}

In *AdConsol-1D* ist es ferner möglich die Berechnung mit Erst- und Wiederbelastungsmodulen durchzuführen und die Bodenschichten in kleinere Δz - Elemente (Inkremente) zu unterteilen. Dabei wird jedem Element seine spezifische Steifigkeit aufgrund der Auflast und Bodeneigenlast zugeordnet.

Die Setzung wird schließlich durch die Funktion 2.5 berechnet.

$$S = \int_0^h \int_{\sigma_\gamma(z)}^{\sigma_\gamma(z)+\sigma'(z)} \frac{i(z) d\sigma' dz}{E_s\left(\sigma'(z)\right)} \tag{2.5}$$

Bei mehrschichtigen Böden lautet die Funktion aus 2.5

$$S = \int_0^{h_1} \int_{\sigma_\gamma(z)}^{\sigma_\gamma(z)+\sigma'(z)} \frac{i(z)d\sigma' dz}{E_{s_1}(\sigma'(z))} + \cdots + \int_{h-1}^{h} \int_{\sigma_\gamma(z)}^{\sigma_\gamma(z)+\sigma'(z)} \frac{i(z)d\sigma' dz}{E_{s_n}(\sigma'(z))} \tag{2.6}$$

Bei überkonsolidierten Böden oder Belastungshistorien mit Ent- und Wiederbelastungen werden die Funktionen 2.5 und 2.6 zu den Funktionalen 2.7, 2.8 und 2.9 nach *Ohde* zusammengefaßt. Dabei wird zwischen drei Bereichen in der Oedometerkurve unterschieden (siehe Bild 2.3):

Für Spannungen die **unter** der Überkonsolidationsspannung σ_{OC} liegen, gilt:
$\sigma'_q(z) \leq \sigma_{oc}$ (Bereich 1 in Bild 2.3):

$$\varepsilon(z) = \int_{\sigma_\gamma(z)}^{\sigma_\gamma(z)+\sigma'_q(z)} \frac{i(z)d\sigma'}{E_{ur}(\sigma'(z))} \tag{2.7}$$

Für Spannungen die **über** der Überkonsolidationsspannung σ_{OC} liegen, gilt:
$\sigma'_q(z) > \sigma_{oc}$ (Bereich 2 in Bild 2.3):

$$\varepsilon(z) \int_{\sigma_\gamma(z)}^{\sigma_\gamma(z)+\sigma'_{oc}} \frac{i(z)d\sigma'}{E_{ur}(\sigma'(z))} + \int_{\sigma_\gamma(z)+\sigma'_{oc}}^{\sigma_\gamma(z)+\sigma'_q(z)} \frac{i(z)d\sigma'}{E_s(\sigma'(z))} \tag{2.8}$$

Bei erneuter Entlastung oder Wiederbelastung gilt:
$\sigma_{oc} < \sigma'_q(z)$ und $\sigma'_q(z) < \sigma_{max}$ (Bereich 3 in Bild 2.3):

$$\varepsilon(z) = \int_{\sigma_\gamma(z)}^{\sigma_\gamma(z)+\sigma'_{oc}} \frac{i(z)d\sigma'}{E_{ur}(\sigma'(z))} + \int_{\sigma_\gamma(z)+\sigma'_{oc}}^{\sigma_\gamma(z)+\sigma'_{max}} \frac{i(z)d\sigma'}{E_s(\sigma'(z))} + \int_{\sigma_\gamma(z)+\sigma'_{max}}^{\sigma_\gamma(z)+\sigma'_q(z)} \frac{i(z)d\sigma'}{E_{ur}(\sigma'(z))} \tag{2.9}$$

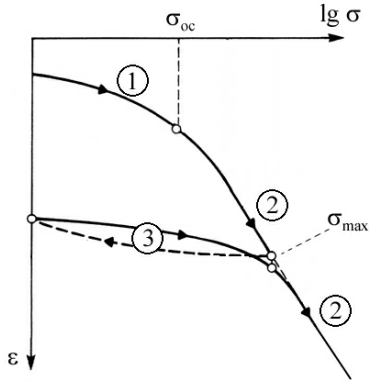

Bild 2.3: Abschnitte aus der Oedometerkurve zur Bestimmung des Funktionals

2.4 Konsolidation

2.4.1 Gegenstand

Die 'Konsolidationstheorie' beschreibt mathematisch die Verformung eines porösen Mediums bei gleichzeitigem 'Auspressen' des Porenwassers.

Das poröse Medium besteht dabei aus 'Feststoffpartikeln', die sich zu einem Skelett zusammenfügen. Der Raum zwischen den 'Partikeln' ist dabei mit Wasser gefüllt. Setzt man das Medium einer Spannung aus, wird das Skelett zusammengedrückt und das Porenwasser mit einer bestimmten Geschwindigkeit herausgepresst.

Es gibt zwei Formulierungen der Konsolidationstheorie. Die *volumenabhängige Theorie* ('Volume fraction') verwendet die Porenzahl e oder den Porenanteil n als die abhängige Variable der Grundgleichung. Die *Porendruck-Formulierung* verwendet den Porenwasserüberdruck als die abhängige Variable.

Die erste Differentialgleichung zur Beschreibung des Porenwasserdrucks für den eindimensionalen Fall wurde 1925 von *Terzaghi* angegeben. (Krause 1977).

2.4.2 Porendruck-Formulierung

Die einfachste, nicht empirische Methode, die zeitabhängige, eindimensionale Konsolidation von weichen, feinkörnigen Materialien zu bestimmen, ist die Lösung zur Bestimmung des Konsolidationsgrades in Abhängigkeit eines dimensionslosen Zeitfaktors und des Porenwasserdrucks (*Cargill 1984; Duncan 1993; Fox 1999*) (Morris 2002). Die Lösungen basieren meist auf der 'konventionellen' Theorie der kleinen Verformungen von *Terzaghi* (1924), ausgedrückt in der Gleichung:

$$c_v \frac{\partial^2 u}{\partial z^2} = \frac{\partial u}{\partial t} \tag{2.10}$$

mit c_v als Konsolidationsbeiwert (2.11), dem Porenwasserüberdurck u, der vertikalen Koordinate z und der Zeit t.

$$c_v = \frac{k \cdot E_s}{\gamma_w} \tag{2.11}$$

Die Gleichung 2.10 basiert auf vereinfachenden Annahmen, dass sowohl die Durchlässigkeit k, als auch die Steifigkeit E_s unter der Spannungs- beziehungsweise Lastveränderung konstant bleibt. Dies führt häufig zu stark von der Realität abweichenden Ergebnissen. Der auftretende Fehler hängt dabei im Wesentlichen von der Magnitude des Lastinkrementes und den daraus resultierenden Porenzahlveränderungen ab.
Wird die Durchlässigkeit als Funktion der Porenzahl $k(e)$ und die Steifigkeit als Funktion der effektiven Spannung $E_s(\sigma')$ geschrieben, ergibt sich die Funktion 2.10 nach *Zimmerer* wie folgt:

$$\frac{\partial u(z)}{\partial t} = \frac{-k\,(e)\,E_s\,(\sigma'(z))}{\gamma_w} \frac{\partial^2 u(z)}{\partial z^2} \tag{2.12}$$

2.4.3 Konsolidationsberechnung in *AdConsol-1D*

Die numerische Umsetzung erfolgt in *AdConsol-1D* nach der 'finiten Differenzen Methode' (FDM), dem Schema von *Braja M. Das* und Erweiterungen durch *Zimmerer (2001).*

Zur Lösung können mehrere Wege beschritten werden. Die gängigste Variante ist die Auflösung der Differenzen-Quotienten nach einem expliziten Schema, wobei die Gleichungssysteme nach je einer Unbekannten aufgelöst werden. Dabei lassen sich aus dem Differenzenschema von Zeitschritt zu Zeitschritt fortschreitend, unter Benutzung der Anfangs- und Randbedingungen, die gesuchten Näherungen berechnen. Um die Gleichungssysteme lösen zu können und im Grenzbereich ein oszillieren der expliziten Lösung zu vermeiden, muss dabei folgende Bedingung eingehalten werden: (Schwetlick/Kretzschmar 1991)

$$\Delta t \leq 0,5 \cdot \frac{\Delta z^2 \cdot \gamma_w}{k \cdot E_s} \tag{2.13}$$

Durch die Bedingung 2.13 ist das Lösungsverhalten abhängig von den Ausgangswerten Δt, Δz, k und E_s. Dies stellt einen erheblichen Nachteil der expliziten Lösungsvariante dar.

Aus diesem Grund wurde in *AdConsol-1D* ein impliziter Lösungsweg gewählt. Dabei ermittelt man aus dem bekannten Zeitschritt k im Zeitschritt $k+1$ je drei Unbekannte (siehe Gleichung 2.17 und Bild 2.4). Die Differentiale aus Gleichung 2.12 werden durch Zentraldifferenzen 2. Ordnung approximiert. Gleichzeitig führt der implizite Lösungsweg zu deutlich geringeren Rechenzeiten und einer höheren Genauigkeit der Ergebnisse. In impliziter Schreibweise erhält man:

$$\frac{\partial u\,(z,t)}{\partial z} = \frac{u\,(z+\Delta(z,t)+\Delta t) - u\,(z-\Delta(z,t)+\Delta t)}{2\Delta z} \tag{2.14}$$

$$\frac{\partial^2 u\,(z,t)}{\partial z^2} = \frac{u\,(z+\Delta(z,t)+\Delta t) - 2u\,(z,t+\Delta t) + u\,(z-\Delta(z,t)+\Delta t)}{(\Delta z)^2} \tag{2.15}$$

$$\frac{\partial u\,(z,t)}{\partial t} = \frac{u\,(z,t+\Delta t) - u\,(z,t)}{\Delta t} \tag{2.16}$$

Die Schreibweise der Gleichung 2.12 in Form von finiten Differenzen lautet nach Umformungen:

$$\frac{u_i^{k+1} - u_i^k}{\Delta t} = c_v \frac{u_{i+1}^{k+1} - 2u_i^{k+1} + u_{i-1}^{k+1}}{(\Delta z)^2} \tag{2.17}$$

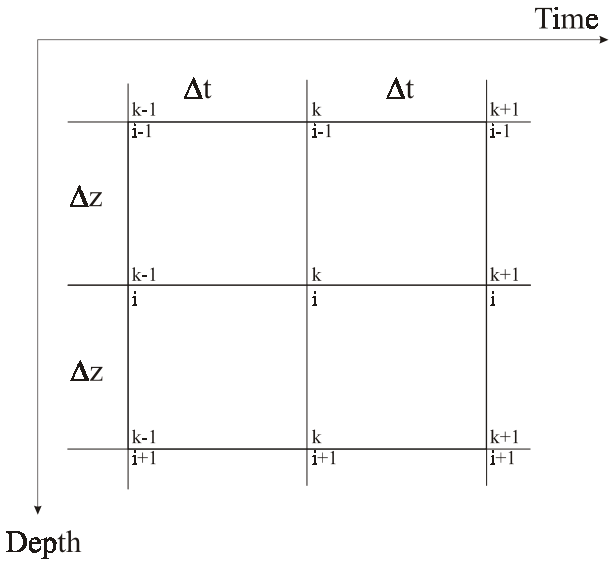

Bild 2.4: Punkte des Berechnungsgitters über die Zeit und Tiefe

2.4.4 Geschichteter Boden

Es ist nicht immer möglich in geschichteten Böden eine geschlossene Lösung für den Konsolidationsvorgang zu finden. Voneinander abweichende Eigenschaften der Bodenschichten (z.B. unterschiedliche Durchlässigkeiten, Schichtdicken oder Konsolidationsbeiwerte), führen in jeder Schicht zu einem anderen Konsolidationsverhalten. In den Berechnungen muss jedoch immer gewährleistet sein, dass der Porenwasserdurchfluss von einer Schicht zur nächsten kontinuierlich ist.

Dementsprechend ist die Gleichung 2.17 zur Berechnung von Porenwasserüberdrücken an der Schnittstelle zweier verschiedener Bodenarten entsprechend zu modifizieren.

Aus Gleichung 2.12 wird nach *Scott*:

$$\begin{array}{ccc} \frac{k}{c_v}\frac{\partial u}{\partial t} & = & k\frac{\partial^2 u}{\partial z^2} \\ \uparrow & & \uparrow \\ Volumen- & & Unterschied\,zwischen \\ änderung & & den\,Fließraten \end{array} \tag{2.18}$$

Basierend auf Gleichungen der Strömungsmechanik wird die Gleichung 2.18 zu Gleichung 2.19.

$$\left(\frac{k_1}{c_{v1}}+\frac{k_2}{c_{v2}}\right)\frac{u_i^{k+1}-u_i^k}{\Delta t}=\frac{1}{(\Delta z)^2}\,(k_1+k_2)\left(\frac{2k_1}{k_1+k_2}u_{i-1}^{k+1}-2u_i^{k+1}+\frac{2k_2}{k_1+k_2}u_{i+1}^{k+1}\right) \tag{2.19}$$

Die Gleichung 2.19 wird aufgelöst und zu einer impliziten Lösungsmatrix umformuliert. Diese implizite Formulierung entspricht nur dann einer wirklich impliziten Lösung, wenn die Durchlässigkeiten k und die Konsolidationsbeiwerte c_v konstanten Werten entsprechen und nicht vom zeitlichen Verhalten des Bodens abhängig sind.

In der von *Zimmerer (2001)* (und Anderen) erweiterten Konsolidationstheorie nach *Terzaghi* wird die Durchlässigkeit in Abhängigkeit der initialen Durchlässigkeit und der aktuellen Porenzahl nach einem Ansatz von *Biot* zu:

$$log\frac{k}{k_0} = \frac{\Delta e}{c_k} \tag{2.20}$$

mit

Δe = Veränderung der Porenzahl
k = Durchlässigkeit
k_0 = initiale Durchlässigkeit
c_k = Faktor zur Beschreibung der Veränderung der Durchlässigkeit

Die Steifigkeit (und damit c_v) wird nach dem Ansatz von *Schanz (1998)* und *Ohde (1939)* (Gleichung 2.4) formuliert. Damit ergeben sich stets vom aktuellen Zustand (Porenzahl beziehungsweise effektive Spannung) abhängige Steifigkeiten und Durchlässigkeiten. Da die effektive Spannung immer vom aktuellen Porenwasserüberdruck u abhängt, kann man Gleichung 2.4 auch umformulieren zu:

$$E_s(u) = E_{oed}^{ref}\left(\frac{\sigma(z) - u(z,t) + \sigma_\gamma(z)}{\sigma_{ref}}\right)^m \tag{2.21}$$

Die Steifigkeit wird damit eine vom Porenwasserüberdruck u abhängige Variable.

Bei einem rein impliziten Verfahren würden die Porenzahl e, die effektive Spannung σ' und der Porenwasserüberdruck u ebenfalls aus dem aktuellen Zeitschritt $k+1$ (siehe Bild 2.4) entnommen, wodurch das zu lösende Gleichungssystem jedoch nichtlinear würde. Um schnelle Lösungsalgorithmen zur Berechnung linearer Gleichungssysteme (z.B. 'Thomas-Algorithmus') nutzen zu können, wurden die Koeffizienten der Porenwasserdrücke aus dem bekannten Zeitschritt k entnommen.

2.5 Formelzeichen

Tabelle 2.1: Verwendete Formelzeichen in der Konsolidationstheorie

Formelzeichen	**Beschreibung**
a	Belastungslänge bei *Steinbrenner*
b	Belastungsbreite bei *Steinbrenner*
c_k	Faktor zur Beschreibung der Veränderung der Durchlässigkeit in Gleichung 2.20
c_v	Konsolidationsbeiwert nach Gleichung 2.11
Δe	Veränderung der Porenzahl
Δz	Schichtinkrement
e	Porenzahl
E_{oed}^{ref}	normierte Steifigkeit
E_s	Steifigkeit
γ_w	Wasserwichte
h_w	hydrostatische Druckhöhe
i	Steinbrennerwert nach Gleichung 2.1
k	Durchlässigkeit
k_0	initiale Durchlässigkeit
m	Exponent aus Regressionskoeffizienten einer Näherungsfunktion zur Bestimmung von E_{oed}^{ref}
n	Porenanteil
σ	Totale Spannung
σ'	Effektive Spannung
σ_0	Spannung an der Oberfläche
σ_γ	Spannung aus Bodeneigengewicht
σ_{OC}	Überkonsolidationsspannung
σ_{ref}	Referenzspannung aus dem Ansatz von Ohde
t	Zeit
u	Porenwasserüberdurck
u_{stat}	Porenwasserdurck
z	Tiefe im Boden

Kapitel 3

Allgemeine Umsetzung

3.1 Problematik des Prototypings

Bei der Entwicklung einer neuen Software sind im Wesentlichen folgende drei Entwicklungsstadien zu durchlaufen, die man am effektivsten in dieser Reihenfolge abhandelt:

- Spezifikation (Problemanalyse und Lösungsplanung)
- Entwurf (Lösungsbeschreibung und Lösungsüberprüfung)
- Implementierung (Programmcodierung und Programmtest)

Leider lässt sich dieses Entwurfsschema nur selten so exakt einhalten. Auch bei der Entwicklung von *AdConsol-1D* konnten gewisse Umwege nicht vermieden werden. Vielmehr entwickeln sich oftmals Programmdetails aus einer bereits lauffähigen Testversion heraus; Features werden wieder verworfen und neue Elemente kommen hinzu. Dieses Entwicklungsverhalten ist jedoch typisch für Prototypen wie *AdConsol-1D*. Folgende Definition erläutert den Begriff 'Prototyping' näher:

> *Unter Prototyping wird eine Vorgehensweise verstanden, bei der möglichst früh bereits eine erste ablauffähige Programmversion erstellt wird. Ziel ist es, wichtige, noch nicht zur Gänze geklärte Fragen (z.B. im Bereich der Benutzeroberfläche oder Funktionalität) mit den Benutzern zu klären und praktische Erfahrungen zu sammeln. Die erste vorläufige Programmversion wird aufgrund der Benutzerkommentare in der Folge vervollkommnet oder sie bildet mitsamt den Verbesserungsvorschlägen die Vorlage für die Neuerstellung einer effizienten, benutzergerechten Endversion.*
>
> [Hans-Robert Hansen]

3.2 Objektorientierte Programmierung

3.2.1 Einführung in die OOP

Software für heutige Anforderungen wird zunehmend komplexer. Der Prozess der Entwicklung von Anwendungen besteht zum großen Teil darin diese Komplexität in den Griff

zu bekommen. Alles, was dazu beiträgt große Probleme in kleine, handhabbare Einheiten aufzuteilen, ist daher von Vorteil. Die Objektorientierung als grundlegende Programmierungstechnik ist 'das' Werkzeug um "große Probleme" in "kleine Objekte" zu zerlegen.

> *Object-Oriented programming is not so much a coding technique as it is a code packaging technique, a way for code suppliers to encapsulate functionality for delivery to consumers.*
>
> [Brad J. Cox]

Mit dieser Aussage degradiert *Cox* die objektorientierte Programmierung schlicht zur 'Verpackungstechnik'. Sieht man sich die Grundbegriffe der objektorientierten Programmierung an, so wird man erkennen, dass diese Betrachtungsweise weitgehend zutrifft. (Terlinden 1996)

3.2.2 Begriffe der OOP

Klassen - class

Eine Klasse ist eine Softwarekonstruktion, die Mitgliedsvariablen und Methoden eines Objekts definiert. Die Klasse selbst ist kein Objekt, sie ist lediglich die Vorlage (Prägestempel), die definiert wie das Objekt auszusehen hat und wie es sich verhalten soll. Ein großer Vorteil von Klassen besteht darin, dass sie die Wiederverwendbarkeit von Programmcode ermöglichen. Oft wird anstatt 'Klasse' auch der Begriff 'Objekttyp' verwendet. Man erhält ein konkretes Objekt indem man eine Instanz der Klasse bildet. Mit einer einzigen Klassendefinition können viele Objekte erzeugt werden, gleichsam der Fertigung vieler Autos nach den selben Bauplänen auf dem Montageband. (Doberenz/Kowalski 1999*a*)

Objekt - object

Die Instanzen einer Klasse nennt man Objekte. Beim Anlegen einer Instanz (man spricht dabei von Instanzierung) wird die Klasse als Vorlage für das neue Objekt benutzt. Ein Objekt ist ein abgeschlossener Speicherbereich mit einer genau definierten Datenstruktur (den Eigenschaften) und fest umrissenen Aufgaben. (Terlinden 1996)

Attribute - properties

Die Attribute (Eigenschaften) bilden die Datenstruktur eines Objekts. Sie speichern dessen Zustand und können in der Regel nur über Methoden verändert werden. (Terlinden 1996)

Methoden - methods

In der objektorientierten Programmierung werden die Daten eines Objekts nicht direkt bearbeitet, sondern über Methoden manipuliert. Die Methoden beschreiben wie sich das Objekt zu verhalten hat. Die Methoden bestimmen somit das Verhalten der Objekte. (Terlinden 1996)

Ereignis - event

Ereignisse sind Aktionen oder Zustandsänderungen auf die ein Programm reagieren kann. Typische Ereignisse sind zum Beispiel Mausbewegungen, das Drücken einer Taste oder das Klicken auf Schaltflächen.

Die Kommunikation mit einem Objekt erfolgt über Nachrichten. Jedes Objekt weiß auf welche Nachrichten es mit welchen Methoden zu reagieren hat. Benutzeraktionen wie beispielsweise ein Mausklick, lösen beim betreffenden Objekt eine Methode mittels einer vom Betriebssystem gesendeten Nachricht aus.

Ereignisse können aber auch innerhalb von Objekten ausgelöst werden. Sie gehören deshalb zum sogenannten "outgoing Interface". Im Gegensatz dazu stellen Attribute und Methoden das "incoming Interface" dar, da sie von außerhalb des Objekts aufgerufen werden. In den Event-Handlern der ereignisauslösenden Objekte können andere Objekte auf diese Ereignisse reagieren. (Doberenz/Kowalski 1999*b*) (Terlinden 1996)

3.2.3 Vorteile der OOP

Nachdem eine Klasse implementiert wurde und damit ihre Datenstrukturen angelegt und der interne Code zur Bearbeitung dieser Datenstrukturen geschrieben und getestet wurde, kann man diese Klasse nach dem 'Black-Box-Prinzip' uneingeschränkt einsetzten. Somit braucht man sich keine Gedanken darüber zu machen, ob Änderungen in einem Teil des Programms in Konflikt mit der Funktionsweise eines Objekts geraten, oder ob andere Funktionen oder Objekte zufällig die Datenstrukturen innerhalb des Objekts beeinträchtigen könnten. Auf die Attribute und Methoden eines Objekts kann nur über die öffentlichen Schnittstellen (Interfaces) zugegriffen werden. Dieser Gedanke geht sogar noch weiter. Man kann Objekte benutzen ohne deren Implementierung zu kennen.

Diese Trennung zwischen Objekten kann zur langfristigen Zuverlässigkeit einer Anwendung beitragen, insbesondere, wenn mehrere Entwickler daran schreiben. Auch wird das Hinzufügen neuer Features zu einer Anwendung vereinfacht. Es genügt die Interfaces einiger Klassen zu erweitern und die Methoden zu implementieren, anstatt den gesamten Code der Anwendung zu überarbeiten oder neu zu schreiben. Ein weiterer Vorteil der objektorientierten Programmierung ist, dass fertig implementierte und getestete Klassen auch in anderen Projekten wiederverwendet werden können. Dadurch reduziert sich der erforderliche Testaufwand. (Terlinden 1996) (Doberenz/Kowalski 1999*a*) (Shadoware April 2003)

3.2.4 Merkmale einer objektorientierten Sprache

Folgende vier Merkmale werden als die wichtigsten Anforderungen an objektorientierte Sprachen angesehen:

1. Abstraktion (engl. abstraction)
2. Kapselung (engl. encapsulation)
3. Vererbung (engl. inheritance)
4. Polymorphismus (engl. polymorphism)

Abstraktion - abstraction

Unter Abstraktion versteht man die selektive Auswahl bestimmter Aspekte eines Gegenstands oder eines Problems. Dabei wird versucht die Wirklichkeit in Klassen zu modellieren, die durch Eigenschaften und Methoden beschrieben werden. Wichtige Aspekte werden von unwichtigen getrennt und ein abstrakter Datentyp kreiert. Die Wahl des richtigen Grades der Abstraktion bildet ein zentrales Problem bei der Entwicklung objektorientierter Software.

Kapselung - encapsulation

Kapselung ist die Fähigkeit Daten eines Objekts vor direktem Zugriff zu schützen. Das Objekt kann nur über die öffentlichen Schnittstellen (Interfaces) manipuliert werden. Datenbezogene Fehler sind damit leicht lokalisierbar. Dies geschieht in der Regel durch die Zugriffsspezifikationen *Private* und *Public* (Terlinden 1996)

Vererbung - inheritance

Bei der Vererbung wird die Definition einer Klasse für weitere Klassen als Grundlage verwendet. Man kann eine Unterklasse (Subklasse, Kindklasse) bilden, die alle Attribute und Methoden ihrer Oberklasse (Superklasse, Vaterklasse) erbt. Zu den geerbten Eigenschaften der Unterklasse können neue hinzugefügt oder vorhandene, geerbte Eigenschaften überschrieben werden. (Terlinden 1996)

Polymorphismus - polymorphism

Polymorphie bezieht sich darauf, dass die korrekten Methoden aufgerufen werden können, wenn Objekte über eine Objekt-Variable referenziert werden. Die meisten objektorientierten Sprachen realisieren Polymorphie über *Vererbung*. Der Vorteil von Polymorphie liegt im Beitrag zur Reduktion von Komplexität. Andernfalls müssten für jedes Objekt andere Befehle definiert werden.

Wenn beispielsweise eine Variable vom Typ *Object* angelegt wurde und diese Variable eines aus vielen Objekten referenzieren kann, die alle über eine 'Clear-Methode' verfügen, wird immer korrekt die 'Clear-Methode' des gerade von der Variablen referenzierten Objekts aufgerufen. (Doberenz/Kowalski 1999*b*)

3.3 Anforderungen an ein Windows-Programm

3.3.1 Notwendigkeit einer Windows gerechten Oberfläche

Rechner dienen in der täglichen Arbeit des Ingenieurs traditionell als reines Rechenwerkzeug. Mit der allgemeinen Durchdringung der Arbeitswelt durch immer preiswertere und leistungsfähigere Computer verlagern sich jedoch immer mehr Tätigkeiten, die bisher mit Papier und Bleistift ausgeführt wurden, ebenfalls auf den Rechner. Der Computer hat sich damit als Allzweckwerkzeug auch bei nicht numerischen Aufgabenstellungen wie Textverarbeitung und Datenbanken durchgesetzt.

Der Durchschnittsbenutzer eines Computers arbeitet in der Regel nicht ständig und täglich mit demselben Programm, sondern wechselt zwischen unterschiedlichen Applikationen. Wenn er sich dabei jeweils in einer anders gestalteten Umgebung zurechtfinden und mit einer anderen Eingabephilosophie auseinandersetzen muss, sind Fehler und langsames, ineffizientes Arbeiten vorprogrammiert.

Im technischen Büro werden fast ausschließlich *PCs* mit *Intel*-Prozessor und einer Variante des Betriebssystems *Windows* von *Microsoft* eingesetzt. Die Produkte, die von *Microsoft* für den breiten Markt der allgemeinen Bürosoftware (Textverarbeitung, Tabellenkalkulation, relationale Datenbanken) angeboten werden, folgen heute detailliert gefassten, einheitlichen Gestaltungsrichtlinien. Demgegenüber zeichnet sich die derzeit verfügbare kommerzielle Bausoftware, soweit eine grafische Oberflächen vorhanden ist, teilweise noch durch ein buntes Vielerlei unterschiedlichster Oberflächengestaltungen aus. Dies ist einer der Gründe dafür, dass sich Benutzer beim Einarbeiten in bauspezifische Programme oft schwer tun, während bei allgemeiner Bürosoftware selbst Mitarbeiter, die kaum Hintergrundwissen über Computer besitzen, sehr schnell mit den Programmen zurechtkommen. Einheitliches Aussehen und Funktionalität von Programmen steigern aber nicht nur die Effizienz der Arbeit, sondern helfen auch Interpretations- und Bedienungsfehler zu vermeiden. Sie dienen damit der Qualitätssicherung computergestützter Arbeit. Viele Menüpunkte, die in allgemeiner Bürosoftware notwendig sind, können auch in einem Statikprogramm sinnvoll sein. Der Anwender sollte sie dort an derselben Stelle mit gleicher Funktionalität vorfinden. Auch Maus- und Tastaturfunktionalitäten sollten sich so eng wie möglich an die Vorgaben durch Standardbüroprogramme anlehnen. Vereinheitlichung von Programmen durch Orientierung an industriellen 'Quasistandards' ist deswegen das Ziel. (Holzer 1997)

3.3.2 Grafische Interaktion zwischen Anwender und Programm

Kaum eine kommerzielle Statiksoftware kommt heutzutage noch ohne eine grafische Oberfläche aus, und die Attraktivität grafischer Ein- und Ausgaben ist mit Recht ungebrochen. Die Zeichnung ist die Sprache des Ingenieurs. Deshalb trägt direktes grafisches Feedback viel dazu bei, Fehler und Irrtümer zu vermeiden. Im Idealfall sollten sofort alle Auswirkungen einer Aktion sichtbar werden. Dies ist jedoch, vor allem bei rechenintensiven Verfahren wie FEM (Finite Elemente Methode), nur bedingt möglich.

Ein Programm, das zur grafisch-interaktiven Konstruktion eines statischen Systems dient, hat viel Ähnlichkeit mit einem CAD-Programm, und Techniken, die im CAD die Arbeit vereinfachen, sind auf das Statik-Programm übertragbar. Nach dem Prinzip "objektorientierter Interaktion" funktionieren inzwischen auch die meisten Standardbüroprogramme. Dabei selektiert der Benutzer zuerst ein oder mehrere Objekte in der Oberflächengrafik. Erst danach wird eine Aktion ausgewählt, die auf die selektierten Objekte angewendet werden soll. (Holzer 1997)

3.3.3 Richtlinien einer MDI Anwendung

AdConsol-1D wird als MDI Anwendung (Multiple Document Interface) implementiert. Folgende Richtlinien muss jede MDI-Anwendung erfüllen:

- Ein Fenster spielt die Rolle des Hauptfensters, in dessen Innenfläche sich die anderen Fenster (Kindfenster) bewegen.
- Wird ein Kindfenster maximiert, nimmt es die Größe des Hauptfensters an. Ein Kindfenster wird folglich nie größer als das Hauptfenster.
- Wird ein Kindfenster minimiert, wird es am unteren Rand des Hauptfensters angeordnet.
- Das Hauptfenster besitzt ein Fenster-Menü über dessen Kommandos die Fenster angeordnet und aktiviert werden können.
- Ist ein Kindfenster nur teilweise sichtbar, erhält das Hauptfenster automatisch eine Bildlaufleiste zum Scrollen.
- Wird ein Kindfenster mit eigener Menüleiste aktiv, wird diese im Hauptfenster angezeigt.

(Schels/Schels 2000)

3.4 Implementierung der Berechnungsalgorithmen

Um neue Berechnungsalgorithmen untersuchen zu können, sollen diese in *AdConsol-1D* leicht austauschbar sein. Es gibt hierfür im wesentlichen zwei mögliche Lösungsansätze: Der Austausch der Berechnungsalgorithmen zur Laufzeit des Programms und das Auswechseln von Bechnungsklassen vor der Kompilierung. Der erste Ansatz lässt sich mit 'Skripten' verwirklichen, der zweite durch die feste Integration der Berechnungsalgorithmen ins Klassenmodell als eigene 'Berechnungsklasse'. Die Vor- und Nachteile dieser beiden Methoden sollen im folgenden Abschnitt erörtert werden.

3.4.1 Skripting

Skripte

Ein Skript ist ein Programm oder eine Sequenz von Befehlen, die von einem anderen Programm ausgeführt werden kann. Diese liegen in einer Textdatei vor. So werden zum Beispiel *JavaScripts* oder *VBScripts* vom Browser gelesen, interpretiert und ausgeführt. Im Gegensatz zu 'normalen' Programmen benötigen Skripte keinen Compiler. Sie werden zur Laufzeit interpretiert und sind somit sowohl im Internet als auch im Systembereich ideal für einfache Programmierlösungen geeignet. (Winkler 2002)

Vorteile des Skriptings

Visual Basic stellt mit dem *Script-Steuerelement-Objekt* eine effektive Möglichkeit bereit, Programme mittels *VBScript* zu steuern. Mit dem *Script-Steuerelement* kann sowohl auf *ActiveX*-Objekte als auch auf Methoden und Attribute programminterner Klassen zugegriffen werden. Damit ist man in der Lage projektspezifische Befehle zu entwickeln und damit eine eigene, auf das Programm zugeschnittene Skriptsprache zu definieren.

Die Berechnungsalgorithmen von *AdConsol-1D* können mit Skripten leicht ausgetauscht und modifiziert werden. Die Skripte liegen entkoppelt vom Programm in eigenen ASCII-Dateien vor und können separat editiert werden. Zur Laufzeit werden die Skripte ausgewählt, geladen, interpretiert und ausgeführt. (Petroutsos 1. Auflage 1998)

Nachteile des Skriptings

In der Dokumentation wird *VBScript* für kleinere Anwendungen selbst nur dann empfohlen, wenn man die Einfachheit mit Performance-Verlust bezahlen will. *VBScript* ist (wie alle Skriptsprachen) eine interpretierte Sprache und kann bei weitem nicht mit der Ausführungsgeschwindigkeit kompilierter *Visual Basic* Anwendungen mithalten. Noch gravierender ist, dass *VBScript* eine typenlose Sprache ist. Demnach werden alle Variablen als *Variant* gespeichert. Vor der Verarbeitung müssen somit alle Variablen erst in den richtigen Datentyp konvertiert werden. Dies macht sich erheblich in der Performance bemerkbar. Ein weiterer Nachteil einer typenlosen Sprache ist, dass Objekte nicht früh gebunden werden können (*early Binding*). Dies stellt wiederum einen Performance-Verlust dar. Die Berechnung jedoch, ist der zeitkritische Teil von *AdConsol-1D*. Gerade hier kommt es auf Geschwindigkeit an.

Die nicht vorhandene 'Fehlerbehandlung' ist ein weiteres Defizit des *Script-Steuerelements*. Nur wenige Syntaxfehler werden korrekt erkannt. Die meisten Fehler erzeugen eine generelle 'Automatisierungsfehlermeldung' und es gibt keine Möglichkeit die Fehlerquelle zu lokalisieren, oder nähere Informationen über den Fehler zu erhalten. Man bekommt somit bei der Fehlersuche im Code kaum Unterstützung. Bei einem umfangreichen Projekt mit mehreren hundert Zeilen Berechnungscode und teilweise sehr komplexen Formeln, wie in *AdConsol-1D*, ist dies ein nicht zu unterschätzendes Problem, vor allem wenn man die komfortablen Debugging Möglichkeiten der *Visual Basic* Entwicklungsumgebung gewohnt ist.

Ein weiteres Manko des Scriptings liegt in der Gewährleistung des Urheberrechts. Die Berechnungsroutinen sind Teil der Promotionsarbeit von *Dipl. -Ing. Martin Zimmerer* und sollten vor Veröffentlichung dieser Arbeit nicht für jedermann zur Einsicht bereitstehen. Da das Programm aber kurzfristig auch an Dritte weitergegeben wird, muss dieser Teil des Programms vor Einblicken und Eingriffen geschützt sein. *VBScripts* liegen jedoch als ASCII-Datei vor, die mit jedem Texteditor leicht editier- und lesbar sind. (Petroutsos 1. Auflage 1998)

3.4.2 Berechnungsklasse

Die oben genannten Nachteile haben dazu geführt den Berechnungsteil von *AdConsol-1D* nicht als Scripting Modell zu implementieren.

Da am Lehrstuhl Bodenmechanik bereits Grundkenntnisse im Umgang mit *Visual Basic* vorhanden sind, hat sich die Möglichkeit einer festen Implementierung der Berechnungsalgorithmen als Berechnungsklasse angeboten.

Vorteile der Berechnungsklasse

In kompilierter Form ist ein Zugriff auf den Berechnungsquelltext nicht mehr möglich. Auch die Ausführungsgeschwindigkeit ist dabei wesentlich besser als in *VBScript*. Seit Version 5 kann *Visual Basic* 'System-Code' kompilieren. Version 6 enthält einen 'System-Code-Compiler', der, ähnlich wie der *Visual C++* Compiler, optimierten Code erstellen kann (Leider bedeutet dies aber nicht, dass auch die Ausführungsgeschwindigkeit ähnlich einem in *C++* erstellten Programm ist). Zudem besteht die Möglichkeit, während der Implementierung den Debugger und die Überwachungstools der *Visual Basic* Entwicklungsumgebung zu nutzen.

Um den Austausch der Berechnungsalgorithmen so einfach wie möglich zu gestalten, werden alle Berechnungsalgorithmen in eine Klasse implementiert. Wenn der Berechnungscode nicht über das gesamte Programm verstreut ist, lässt sich *AdConsol-1D* durch Austauschen dieser Berechnungsklasse leicht auf eine komplett neue Berechnungsweise umstellen.

Die Zusammenfassung der Berechnungsalgorithmen in eine einzige Klasse hat noch weitere Vorteile. Sie hilft dabei, über den Ablauf der Berechnung einen besseren Überblick zu behalten. Da die übrigen Klassen bei Änderungen des Berechnungsalgorithmus unberührt bleiben, sichert dieses Vorgehen auch die langfristige Stabilität des Programms.

Nach dem 'Black-Box-Prinzip' stellt die Berechnungsklasse definierte Schnittstellen zur Verfügung. Über diese Schnittstellen können alle anderen Objekte des Programms mit der Berechnungsklasse kommunizieren. Die ausgetauschte Berechnungsklasse muss dabei immer exakt diese Schnittstellen verwenden.

Kapitel 4

Die Programmiersprache *Visual Basic*

Rechenautomaten haben etwas von den Zauberern im Märchen.
Sie geben einem wohl, was man sich wünscht,
doch sagen sie einem nicht, was man sich wünschen soll.

[Norbert Wiener]

AdConsol-1D soll in der Programmiersprache *Visual Basic* Version 6 (Professional-Edition) der Firma *Microsoft* umgesetzt werden. Diese konnte im Rahmen der Aktion 'Start-Developing' kostenlos bezogen werden. Auf die Entwicklung, Möglichkeiten, Grenzen und Probleme dieser Sprache wird im folgenden Kapitel eingegangen.

4.1 Einführung in *Visual Basic*

4.1.1 Historische Entwicklung von *Visual Basic*

Die Entwicklung von *BASIC*

Der Stammbaum von *Visual Basic* lässt sich fast 40 Jahre zurückverfolgen. 1963 beschäftigten sich die Professoren *Thomas E. Kurtz* und *John G. Kemeny* am *Dartmouth College* (USA) damit, ihren Studenten die Arbeit am Computer näher zu bringen und eine leicht zu erlernende Alternative zu den Programmiersprachen *ALGOL* und *FORTRAN* zu entwickeln. So kam es, dass am 1. Mai 1964 das erste *BASIC*-Programm auf einem Großrechner vom *Typ GE 225* geboren wurde. Die Programmiersprache *BASIC* - das Kurzwort steht für "Beginners Allpurpose Symbolic Instruction Code" - wurde rasch populär. Im Laufe der Jahre kam eine wahre Flut von *BASIC*-Varianten auf den Markt. *Microsoft* entwickelte *MBASIC*, das auf allen Kleincomputern eingesetzt werden konnte. *MBASIC* wurde Anfang 1980 zu *GW-BASIC* weiterentwickelt und damit zum 'Standard-*BASIC*-Dialekt' für *IBM*-kompatible *PCs*. Von 1985 bis 1989 vertrieb *Microsoft QuickBASIC* als semiprofessionelle Programmiersprache. Sie stellte einen kompletten Umbruch zur gewohn-

ten *GW-BASIC*-Philosophie dar. Von nun an waren Zeilennummern nur noch optional - es gab etliche Konstrukte zur strukturierten Programmierung und viele neue *BASIC*-Befehle wurden integriert. Sogar eine Möglichkeit zum Debuggen war vorhanden. Der gravierendste Unterschied war jedoch, dass unter *QuickBASIC* entwickelte Programme nicht mehr durch einen Interpreter ausgeführt, sondern mit einem Compiler in 'Stand-Alone' Programme übersetzt werden konnten. (Schubert Mai 2003) (Gnoth April 2003)

Von *BASIC* zu *Visual Basic*

Mit Beginn der 'Windows-Ära' erkannte *Microsoft*, dass das textbasierte *QuickBASIC* nicht als 'fensterorientierte' Programmiersprache eingesetzt werden konnte. *Microsoft* entwickelte in Folge dessen *Visual Basic* und stellte im August 1991 die Version 1.0 in Deutschland vor.

Ursprünglich war *Visual Basic* als Einsteigersprache vorgesehen. Von Version zu Version entwickelte es sich jedoch immer mehr zu einer ernstzunehmenden Programmiersprache. In *Visual Basic 4* wurde die '32-bit Entwicklungsebene' hinzugefügt. Dadurch sind Anwendungen, die ab *Visual Basic 4* erstellt worden sind, nur noch unter *Windows 95/Windows NT* (und höher) lauffähig. Gleichzeitig wurde durch Einführung der Klassenmodule der Prozess eingeleitet *Visual Basic* in eine objektorientierte Programmiersprache umzuwandeln.

Visual Basic 6 wird in einer *Learning-, Professional-* und *Enterprise-Edition* angeboten. Der Unterschied zwischen den drei Versionen besteht zum einen im Sprachumfang der jeweiligen Version, den mitgelieferten 'Add-In´s' und nicht zuletzt auch im Preis. Dieser beträgt für die Enterprise-Edition ca. 1300 EUR und unterstreicht damit den mittlerweile professionellen Anspruch von *Visual Basic*. Durch den stark gestiegenen Sprach- und Funktionsumfang hat die Übersichtlichkeit jedoch ein wenig gelitten. So wurde zum Vergleich Version 1 auf zwei $5\frac{1}{4}$ Zoll Disketten (ca. 1,2 MB) ausgeliefert, die Version 6 füllt mit Programmhilfe drei CDs (ca. 2100 MB). Trotzdem ist *Visual Basic* immer noch eine relativ leicht zu erlernende Sprache, mit der man beachtliche Ergebnisse erzielen kann. (Kirstein 1. Auflage 1998) (Gnoth April 2003)

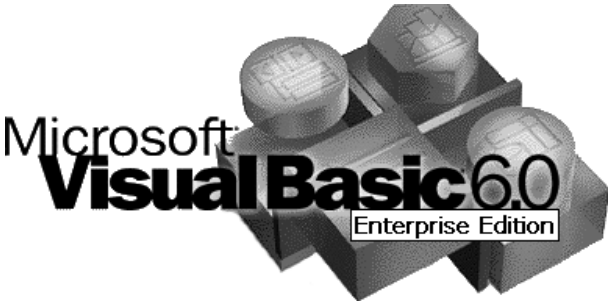

Bild 4.1: *Visual Basic*-Logo

Hat *Visual Basic* noch Gemeinsamkeiten mit *BASIC*?

Wer sich mit dem System beschäftigt, merkt schnell: *Visual Basic* ist *BASIC* zwar ähnlich, aber doch etwas völlig anderes. Von *QuickBASIC* wurde das prozedurale, modulare System übernommen, im Hinblick auf objektorientiertes Programmieren jedoch entsprechend erweitert. Neu für *BASIC* waren Klassen und Objekte. An die Stelle der linearen Programmabläufe trat die Steuerung durch Ereignisse. Der markanteste Unterschied zu *QuickBASIC* ist die schnelle und einfache Möglichkeit grafische Benutzeroberflächen zu erstellen. Wo sonst viele Zeilen Programmcode erforderlich waren um ein kleines Fenster auf den Bildschirm zeichnen zu können, werden jetzt komfortable Benutzeroberflächen einfach per Mausklick erstellt. (Schubert Mai 2003)

4.1.2 *Visual Basic* als Entwicklungssystem

Die *Visual Basic* Entwicklungsumgebung

Wie jedes fortschrittliche Entwicklungssystem für Windows-Applikationen ist auch *Visual Basic* eine visuelle Programmiersprache, das heißt, sie besitzt eine grafische Benutzeroberfläche. Mit Hilfe dieser Oberfläche werden die Formulare und Steuerelemente der Applikation, die der Anwender später auf dem Bildschirm sieht und mit denen er interagiert, zusammengestellt. Die Funktionalität der Elemente wird anschließend, während der eigentlichen Programmierung, im Programmcode implementiert. Die Entwicklungsumgebung ist ein mächtiges Werkzeug. Sie bietet neben dem Editor weitere intuitive Hilfsmittel an, die dem Programmierer große Teile seiner Arbeit abnehmen. Dazu zählen auch Assistenten und Plugins wie beispielsweise der *Installationsassistent*. Außerdem stellt *Visual Basic* dem Programmierer während der Anwendungserstellung viele Hilfe-, Überwachungs- und Debugging-Funktionen bereit.

Installationsassistent von *Visual Basic*

Der *Installationsassistent* von *Visual Basic* ermöglicht es, ein *Setup-Programm* zu erstellen. Dieses Paket enthält neben der kompilierten Version des Programms auch alle zugehörigen Dateien wie *Programmhilfe* und *Resourcendateien*, sowie die zur Ausführung eines *Visual Basic* Programms notwendigen *Laufzeitdateien*, *Steuerelemente* und *Programmbibliotheken*. Damit lassen sich Programme leicht auf Fremdrechnern installieren. Die Setup-Routine kann benutzerspezifisch angepasst werden.

4.2 Objektorientierung in *Visual Basic 6*

Visual Basic ist in seinen wesentlichen Zügen objektorientiert. Inwieweit es die vier Konzepte der objektorientierten Programmierung unterstützt, soll in folgendem Teilkapitel erläutert werden.

4.2.1 Abstraktion - abstraction

Abstraktion geschieht auf zwei Arten. Zum einen abstrahiert der Programmentwickler reale Objekte um damit die Komplexität zu verringern. Auf der anderen Seite muss zur Umsetzung jede objektorientierte Programmiersprache geeignete Mittel zur Verfügung stellen.

In *Visual Basic* geschieht dies durch Objekte, die in Klassenmodulen beschrieben werden. (Terlinden 1996)

4.2.2 Kapselung - encapsulation

In *Visual Basic* können Daten über den Gültigkeitsbereich von Variablen und Prozeduren gekapselt werden. Möglich sind Gültigkeiten auf Prozedur-, Modul- oder Projektebene. Die Deklaration erfolgt dementsprechend mit *Dim*, *Private* oder *Public*. Durch die Möglichkeit eigene Klassen zu definieren und diese mit 'privaten Daten' (Attributen) und Funktionen (Methoden) zu versehen, stellt *Visual Basic* die Kapselung in vollem Umfang zur Verfügung.

Mit *Property*-Prozeduren können 'private Daten' einer Klasse öffentlich gemacht werden. Sie stellen damit die Eigenschaften einer Klasse dar.(Doberenz/Kowalski 1999*a*)

Codebeispiel:

```
'Kapselung auf Prozedurebene:
Private Sub Prozedurnname()
Dim h As Byte
    'Implementierung
End Sub

'Kapselung auf Modulebene:
Private h As Byte

'Kapselung in Klassen:
Private attrSpeed As Single

Private Property Let Speed(ByVal vData As Single)
    attrSpeed = vData
End Property

Private Property Get Speed() As Single
    Speed = attrSpeed
End Property
```

4.2.3 Vererbung - object inheritance

Vererbung im klassischen Sinne ist mit *Visual Basic 6* nicht möglich. *Visual Basic* unterstützt bis Version 6 nur die *Schnittstellenvererbung* bei der eine Klasse nicht den Code, sondern nur das *Interface* einer Klasse erbt. Damit entspricht *Visual Basic* nicht der klassischen Definition einer objektorientierten Sprache. (Doberenz/Kowalski 1999*b*) (Packulat April 2003)

Codebeispiel:

```
'Vaterklasse Fahrzeug-Klasse:
'Eigenschaft Speed
'Methoden Draw und Faster)

Public Speed As Single 'Attribut Speed (auch ohne Property möglich)

Public Sub Draw()
    'Leere Methode (wird von Kindklasse implementiert)
End Sub

Public Sub Faster()
    Speed = Speed + 1
End Sub

'Kindklasse Auto:
Implements Fahrzeug Private fhzg As New Fahrzeug

Private Property Let Fahrzeug_Speed(ByVal vData As Single)
    fhzg.Speed = vData
End Property

Private Property Get Fahrzeug_Speed() As Single
    Fahrzeug_Speed = fhzg.Speed
End Property

Public Sub Fahrzeug_Draw()
    'Hier steht die Implementierung
End Sub

Public Sub Fahrzeug_Schneller()
    Call fhzg.Faster 'geerbte Methode aufrufen ...
    If fhzg.Speed > 130 Then fhzg.Speed = 130 '... und ergänzen
End Sub
```

4.2.4 Polymorphie - polymorphism

Polymorphie bedeutet, dass viele Klassen dieselbe Eigenschaft oder Methode bereitstellen können und beim Aufruf der Eigenschaft beziehungsweise Methode dem Nutzer nicht bekannt sein muss, zu welcher Klasse das betreffende Objekt gehört. (Microsoft Corp. 2000)

Die meisten objektorientierten Sprachen realisieren Polymorphie über *Vererbung*; *Visual Basic* verwendet dazu die Mehrfachschnittstellen des Komponentenobjektmodells (COM = Component Object Model). (Microsoft Corp. 2000) (Doberenz/Kowalski 1999*b*)

Codebeispiel:

```
'Methode in Klasse clsTestA:
Public Sub Zeige()
    MsgBox "Ich bin ein Objekt vom Typ clsTestA"
End Sub

'Methode in Klasse clsTestB:
Public Sub Zeige()
    MsgBox "Ich bin ein Objekt vom Typ clsTestB"
End Sub

'Testprozedur:
Private Sub Testprozedur()
Dim Obj As Object
    If InputBox("Wählen Sie A oder B") = "A" Then
        Set Obj = New clsTestA
    Else
        Set Obj = New clsTestB
    End If
    Call Obj.Zeige
End Sub
```

4.2.5 Ereignis - event

Ein Ereignis wird durch eine Benutzeraktion (z.B. *Mausklick*) oder durch das System (z.B. *Timer*) ausgelöst. In *Visual Basic* ist es auch möglich Ereignisse für eigene Objekte zu implementieren. In einem *Klassenmodul* werden dazu die Ereignisse definiert und ausgelöst. In den Ereignisprozeduren (*Event-Handler*) der ereignisauslösenden Objekte kann auf diese Ereignisse reagiert werden.(Doberenz/Kowalski 1999*a*)

Codebeispiel:

```
'Klasse Auto:
Public Event Schreck() 'Definition "Schreck"-Ereignis

Public Sub hupen() 'Hupen-Methode
    RaiseEvent Schreck '"Schreck"-Ereignis auslösen
End Sub

'"Irgendwo" im Programm:
Private WithEvents Auto As New clsAuto

Private Sub Auto_Schreck() 'Auto hat gehupt
    'Reaktion auf "Schreck"-Ereignis
End Sub
```

4.2.6 Konstruktoren und Destruktoren

Beim Erzeugen oder Entfernen eines Objekts müssen oftmals Aufgaben, wie das Zuweisen von Standardeigenschaften oder das Bereinigen des Speichers, ausgeführt werden. In vielen Programmiersprachen wird dies durch *Konstruktoren* und *Destruktoren* erledigt. Auch *Visual Basic* stellt dafür eine Möglichkeit in Form von Ereignissen zur Verfügung.

Das Ereignis *Initialize()* tritt auf, wenn ein Formular, MDI-Formular oder eine Klasse instanziert wird.

Das Ereignis *Terminate()* tritt auf, wenn alle Verweise auf eine Instanz eines Formulars, MDI-Formulars oder einer Klasse aus dem Speicher entfernt worden sind.

Codebeispiel:

```
'Initialisierung eines Objekts
Private Sub Class_Initialize()
    'Code, der beim Erzeugen des Objekts aufgerufen wird.
End Sub

'Terminierung eines Objekts
Private Sub Class_Terminate()
    'Code, der beim Entfernen des Objekts aufgerufen wird.
End Sub
```

4.2.7 Wichtige Objekte in *Visual Basic*

Außer den selbstdefinierten Objekten, Steuerelementen und Formularen gibt es eine Vielzahl weiterer Objekte, die Visual Basic zur Verfügung stellt. Durch die Nutzung dieser Objekte wird zum Beispiel der Zugriff auf Systemkomponenten wesentlich erleichtert. An dieser Stelle sollen nur einige wichtige Objekte genannt werden die in *AdConsol-1D* Verwendung finden: (Kirstein 1. Auflage 1998) (Terlinden 1996)

- FileSystemObject (Hauptobjekt des Dateisystem-Modells)
- Collection (Auflistungen)
- PropertyBag (Speichern von Eigenschaften eines Objekts)
- Screen (Bildschirm-Objekt)
- Printer (Drucker-Objekt)
- Font (Schriften Objekt)
- App (Informationen über die Anwendung)
- Err (Fehler Objekt)
- Clipboard (Zwischenablage)

FileSystemObject

Das *FileSystemObject* stellt das Hauptobjekt des *Dateisystem-Modells* von *Visual Basic* dar. Es besitzt Methoden die Operationen mit Ordnern und Dateien (Erstellen, Löschen etc.), sowie das Abrufen von Informationen über Laufwerke ermöglichen.

Collection-Objekte

Collection-Objekte (Auflistungen) sind dynamische Datenfelder mit Methoden und Eigenschaften zum Zugriff auf die Daten. Vom System werden die Auflistungen *Controls*, *Forms* und *Printers* zur Verfügung gestellt. Benutzerdefinierte Auflistungen haben den Klassennamen *Collection*. Die Auflistungen speichern Daten mit dem Datentyp *Variant*.

4.3 Programmstrukturierung in *Visual Basic*

Visual Basic Programme können leicht unübersichtlich werden. Code ist in vielen kleinen Ereignisprozeduren verstreut. Gleiche Funktionalität wird oft mehrfach implementiert. Deswegen ist es besonders wichtig sich im Voraus mit den Strukturierungsmöglichkeiten in *Visual Basic* auseinanderzusetzen.

4.3.1 Modultypen

Ein *Visual Basic*-Programm besteht meist aus mehreren *Modulen*. Ein *Modul* fasst Daten und funktionalen Code zusammen, die in einem logischen Zusammenhang stehen. Es gibt in *Visual Basic* folgende Modultypen: (Kirstein 1. Auflage 1998)

Code-Module sollten vor allem für allgemeine Programmkomponenten, allgemein nutzbare Prozeduren, benutzerdefinierte Datentypen und globale Konstanten verwendet werden.

Klassen-Module dienen in der objektorientierten Programmierung zur Definition neuer Objekttypen, die über ihre Methoden und Attribute angesprochen werden. Klassen-Module eignen sich besonders gut zur Wiederverwendung in verschiedenen Projekten.

Form-Module sind gewissermaßen der 'Codeteil' eines Formulars, das als Programmfenster oder Dialog während der Programmausführung angezeigt wird. Form-Module sollten nach Möglichkeit nur Ereignis-Prozeduren der Steuerelemente (Code zur 'Bedienung' der Steuerelemente) enthalten.

4.3.2 Unterprogramme

Jedes Modul in *Visual Basic* kann in Unterprogramme aufgegliedert werden. Unterprogramme sind Anweisungsfolgen, die zur Ausführung durch einen expliziten Aufruf (*Call*) in einer anderen Prozedur oder durch Ereignisse (*Events*) aufgerufen werden. Unterprogramme werden wie folgt aufgegliedert: (Kirstein 1. Auflage 1998)

Sub Routinen führen eine (Teil-) Aufgabe oder Berechnung aus.

Functions führen eine Berechnung aus und geben einen Wert zurück.

Properties definieren Eigenschaften mit individueller Schreib- und Leseberechtigung bei der Implementierung von Eigenschaften in Klassen- und Form-Modulen.

4.4 Namenskonventionen

Objekte müssen eindeutige Namen besitzen, damit im Programm exakt auf sie Bezug genommen werden kann. Bei der Erstellung eines Steuerelements schlägt *Visual Basic* automatisch Namen vor. Da diese jedoch nicht sehr aussagekräftig sind, hat es sich eingebürgert, jeden Objekttyp mit einem aus mindestens drei Buchstaben bestehendem Präfix zu bezeichnen, an das sich die Funktion des Objekts anschließt.

So wird beispielsweise eine *TextBox* für die Eingabe einer Zeit mit dem Namen 'txtTime' benannt. In Tabelle 4.1 sind häufig verwendete Präfixe zusammengefasst.

Tabelle 4.1: Namenskonventionen in *Visual Basic*

Objekt	**Klasse**	**Präfix**
Anzeige	Image	img
Befehlsschaltfläche	CommandButton	cmd
Benutzerdefinierte Klassen	Class	cls
Benutzerdefinierte Module	Modul	mod
Bezeichnungsfeld	Label	lbl
Bildfeld	PictureBox	pic
Formular	Form	frm
Fortschrittsleiste	ProgressBar	prg
Kombinationsfeld	ComboBox	cbo
Kontrollkästchen	CheckBox	chk
Listenfeld	ListBox	lst
Menü	Menu	mnu
Optionsfeld	OptionButton	opt
Rahmen	Frame	fra
RTF	RichTextBox	rtf
Schieberegler	Slider	sld
Textfeld	TextBox	txt

Anmerkung: Obwohl *Microsoft* selbst diese Namenskonventionen vorschlägt, generiert *Visual Basic* automatische Namen, die gänzlich von diesen Konventionen abweichen! (Kirstein 1. Auflage 1998)

4.5 Grunddatentypen

Die in *Visual Basic* zur Verfügung stehenden Grunddatentypen sind in Tabelle 4.2 zusammengefasst. (Kirstein 1. Auflage 1998) (Shadoware April 2003)

Tabelle 4.2: Grunddatentypen in *Visual Basic*

Variablentyp	**Speicher**	**Wertebereich**
Byte (Ganzzahl)	1 Byte	0 bis 255
Boolean	2 Bytes	True oder False
Integer (Ganzzahl)	2 Bytes	-32.768 bis 32.767
Long (Ganzzahl)	4 Bytes	-2.147.483.648 bis 2.147.483.647
Single (Fließkomma)	4 Bytes	$-3,4 \cdot 10^{38}$ bis $1,4 \cdot 10^{-45}$ und $1,4 \cdot 10^{-45}$ bis $3,4 \cdot 10^{38}$
Double (Fließkomma)	8 Bytes	$-1,7 \cdot 10^{308}$ bis $-4,9 \cdot 10^{-324}$ und $4,9 \cdot 10^{-324}$ bis $1,7 \cdot 10^{308}$
Currency (Festkomma)	8 Bytes	-922.337.203.685.477,5808 bis 922.337.203.685.477,5807
Date	8 Bytes	1. Jan 100 bis 31. Dez 9999
Object	4 Bytes	beliebiger Verweis auf ein Objekt
String (variable Länge)	Textlänge	max. 2 Milliarden Zeichen
String (feste Länge)	Textlänge + 10 Bytes	max. 2 Milliarden Zeichen
Variant (mit Zahlen)	16 Bytes	Numerische Werte wie Double
Variant (mit Zeichen)	Textlänge + 22 Bytes	Wie variable Strings

4.6 Einschätzung der Sprache *Visual Basic*

Visual Basic 6 wird in drei Versionen angeboten. Auf dem Markt der Entwicklungssysteme und Programmiersprachen ist es zweifelsfrei das mit großem Abstand dominierende Produkt. Laut *Microsoft* setzen heute weltweit rund 50 Prozent der professionellen Entwickler diese Programmiersprache ein. Damit verfügt *Visual Basic* über eine sehr große Entwicklergemeinde und ein entsprechend großes Reservoir an 'Knowhow' und 'Manpower'. *Visual Basic*-Entwickler sind am Arbeitsmarkt sicher am einfachsten zu finden. Im Laufe eines mehr als 20-jährigen Engagements für *BASIC* ist es *Microsoft* mittlerweile auch gelungen, diese Sprache für professionelle Aufgaben im Unternehmens- und Web-Bereich zu positionieren. Trotzdem haftet an *Visual Basic* immernoch ein Hauch von 'Heimwerker-Image'.

Angesichts des Herstellers kann es nicht erstaunen, dass *Visual Basic* sehr gut in die Windows-Architektur integriert ist und zu Technologien wie *DCOM* (Distributed Component Object Model), *MTS* (*Microsoft* Transaction Server) oder *DNA* (Distributed Net Application Architecture) passt. *COM*-Objekte können mit *Visual Basic* sowohl verarbei-

tet als auch erzeugt werden. Es verfügt über eine Reihe von visuellen Datenbanktools und einer RAD-Umgebung (Rapid Application Development). Aufgrund der zahlreichen, für *Windows*-Benutzer vergleichsweise einfach zu bedienenden visuellen Tools, können auch weniger erfahrene Entwickler schnell sichtbare Resultate erzielen.

Die leichte Erlernbarkeit und die breite Basis von *Visual Basic* setzen jedoch auch die Grenzen. *Microsoft* muss auf die Ansprüche von sehr unterschiedlich ausgerichteten Anwendern eingehen, wofür die Dreiteilung der Editionen eine kaum ausreichende Differenzierung darstellt. So war die Objektorientierung bis Version 6 nicht vollständig verwirklicht. Erst *Visual Basic.NET* ist voll objektorientiert. Die in *Visual Basic* als Datenbank enthaltene *Jet-Engine* kann höheren Ansprüchen nicht genügen. Professionelle Datenbank-Anbindung erfolgt über *ODBC* (Open Data Base Connectivity). Ein Manko ist das Fehlen von Features, die man bei der Realiserung größerer Projekte benötigt. So gibt es in Visual Basic keine Übersichtsmöglichkeit über den kompletten Programmcode und automatische Generierungsmöglichkeiten von UML-Diagrammen oder Struktogrammen. Dies beeinträchtigt die Wartbarkeit von Software Projekten erheblich. Hinzu kommt, dass *Visual Basic* keine Managementfunktionen für das Arbeiten in Teams bietet. Diese Aspekte begrenzen *Visual Basic* nach oben. Es eignet sich daher vor allem für die Programmierung von überschaubaren Modulen oder kleineren Anwendungen und Prototypen, weniger hingegen für große Projekte.

Jedoch kein Nachteil ohne Vorteil. Gerade so lässt sich *Visual Basic* - auf Grund seiner guten Zusammenarbeit mit *DNA*, *COM* oder *OLE DB* - gut als ergänzendes Tool für andere Entwicklungssysteme einsetzen. (Teetz 2002)

Kapitel 5

Umsetzung von *AdConsol-1D*

Code writing is not the problem
unterstanding the problem
is the problem.

[Petkoff]

5.1 Programmentwurf

5.1.1 Eingabe

AdConsol-1D soll zur Berechnung des zeitlichen Setzungsverhaltens mehrschichtiger Böden dienen.

Primär muss hierfür ein Bodenausschnitt definiert werden, in dem alle Aktionen und Berechnungen stattfinden. Der Bodenausschnitt kann als Kubus abstrahiert werden. Die Ausdehnung diese Kubuses in x- und y-Richtung wird durch die Eingabe eines rechteckigen Berechnungsfeldes im Feld-Dialog (*frmField*) beschrieben, die Ausdehnung in z-Richtung durch die Gesamthöhe des Bodenschichtmodells. In Bild 5.1 wird dies verdeutlicht.

Zur Eingabe des Bodenmodells müssen im 'Boden-Dialog' (*frmSoil*) die einzelnen Bodenschichten mit ihren geotechnischen Kenngrößen, ihrer Anordnung und Mächtigkeit festgelegt werden. Die mögliche Anzahl von Schichten soll praktisch unbegrenzt sein. In dieser Programmversion wird vereinfachend ein einheitliches Schichtprofil über das gesamte Untersuchungsgebiet angenommen. Dies ist zulässig und sinnvoll, da die Bodenschichten im Berechnungsmodell nur eindimensional (nach oben oder unten hin) drainieren können.

Im 'Last-Dialog' (*frmLoad*) erfolgt die Eingabe des Lastmodells. Hier werden die einzelnen rechteckigen, konstanten Teilflächenlasten in ihrer zeitlichen Folge und Topologie festgelegt. Dabei ist es nicht notwendig die anzahlmäßig unbeschränkten Lasten zeitlich zu sortieren.

Die Eingaben für Last- und Bodenmodell werden grafisch dargestellt. Da beide Modelle miteinander in Bezug stehen (sozusagen als Grundriss und Schnitt) ist es sinnvoll, diese

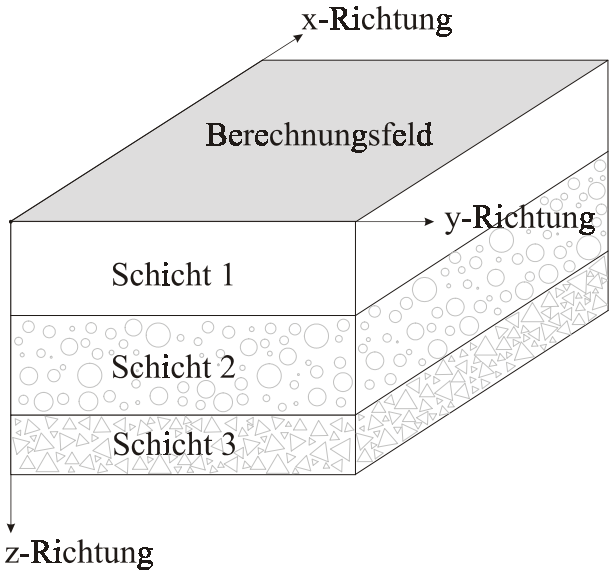

Bild 5.1: Bodenschichtmodell in *AdConsol-1D*

nebeneinander in einem Fenster, dem 'Modell-Fenster' (*frmModell*), anzuordnen (siehe Bild 5.2). Der Zugriff auf alle Eingabedaten muss zu jeder Zeit gewährleistet sein. Um dies zu vereinfachen, ist die Eingabe von Lasten und Bodenschichten auch interaktiv, durch klicken mit der Maus auf die Grafik, möglich.

Das Programm soll Berechnungen in zwei Betriebsarten durchführen können: einer Feldberechnung und einer Punktberechnung. Das Umschalten zwischen den beiden Berechnungsmodi und die benutzerspezifische Anpassung des Programms, sowie das Einstellen der Berechnungseinheiten und das Festlegen der Berechnungsparameter etc., erfolgt in einem eigenen 'Optionen-Dialog' (*frmOptions*).

Für den Punktberechnungsmodus muss explizit ein Berechnungspunkt festgelegt werden in dem die Verformungsberechnung durchgeführt wird. Dies geschieht im 'Punkt-Dialog' (*frmPoint*). Der Berechnungspunkt kann auch interaktiv, durch Klicken mit der rechten Maustaste im 'Modell-Fenster', gesetzt werden.

Für den Feldberechnungsmodus wird automatisch ein regelmäßiges Netz aus Berechnungspunkten über das Berechnungsfeld generiert. Prinzipiell ist es sinnvoll diese Raster an das Lastmodell anzupassen. Die Programmierung des komplexen Optimierungsalgorithmus ist allerdings nicht Bestandteil dieser Arbeit gewesen. Bei Bedarf kann das Programm jedoch um diese Funktion erweitert werden. In der jetzigen Version kann der Anwender das Raster im 'Optionen-Dialog' verfeinern. In jedem der Rasterpunkte werden während der Berechnung die Verformungen ermittelt.

5.1.2 Ausgabe

Die Ergebnisse der Berechnung werden alphanumerisch und grafisch ausgegeben. Die komplette Steuerung der Ergebnisausgabe findet im 'Ergebnis-Fenster' (*frmResults*) statt. Hier

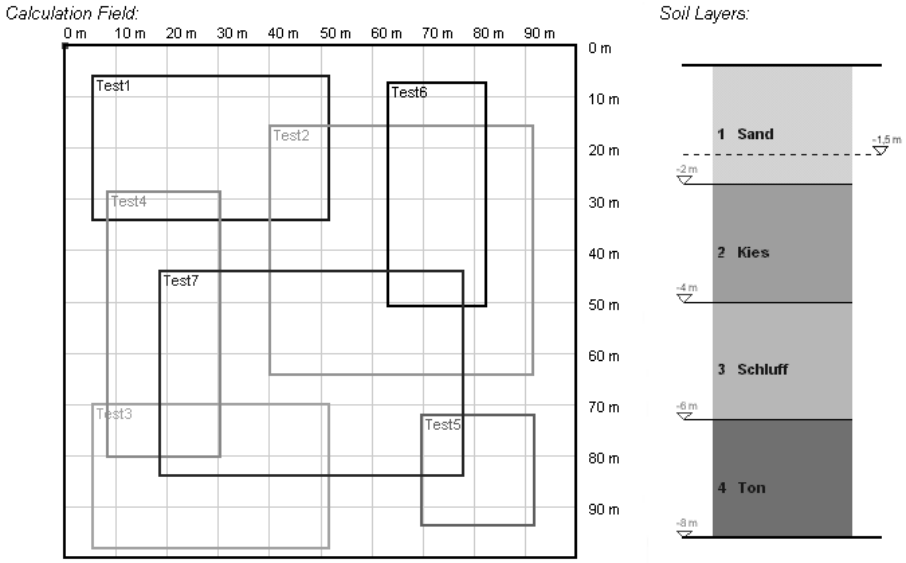

Bild 5.2: Anordnung von Last- und Bodenmodell in *AdConsol-1D*

werden nach der Berechnung zunächst die wichtigsten Ergebnisse für jeden Berechnungspunkt angezeigt. Dies sind die 'Endsetzung', sowie 'maximale Setzung' und 'maximale Hebung' über den Zeitverlauf. Um ein Projekt sinnvoll und nachvollziehbar ausdrucken oder abspeichern zu können, bietet das 'Ergebnis-Fenster' die Möglichkeit, die Eingabedaten strukturiert in den Ergebnistext einzufügen. Damit können Eingaben und Ergebnisse in einer ASCII-Datei (*.txt) abgespeichert oder als Report ausgedruckt werden.

Für die 'Punktrechnung' werden die zeitlichen Setzungen der einzelnen Bodenschichten sowie die Summe der Setzungen über alle Schichten als 'Zeit-Setzungs-Diagramm' dargestellt. Dies geschieht im 'Diagramm-Fenster' (*frmDiagramm*). Die Grafik kann mit der Exportfunktion als Bitmap Datei (*.bmp) gespeichert werden.

Zur Kontrolle der Berechnung und Aufarbeitung der Ergebnisse mittels externer Programme wie *Excel* oder *MathCAD*, ist es wichtig, Zugriff auf die Eingabedaten und Berechnungswerte zu bekommen. Dazu stellt *AdConsol-1D* die Möglichkeit bereit, die wichtigsten Datenfelder der Berechnung im 'Export-Dialog' (*frmOutputSelection*) auszuwählen und als ASCII-Datei (*.dat) abzuspeichern. Diese Dateien können dann von anderen Programmen eingelesen werden.

Im 'Feldberechnungs-Modus' können die Verformungen grafisch als '3D-Modell' mit veränderlichem Ansichtspunkt oder als 'Isoflächen' dargestellt werden (siehe Bild 5.3).

Die Darstellung des '3D-Modells' geschieht in einem eigenen Fenster (*frm3DModell*), das in der Ergebnisausgabe aufgerufen wird. Die Ansicht kann benutzerspezifisch angepasst werden. So ist es möglich, das Modell mit der Maus oder Tastatur in alle Richtungen zu drehen und zu zoomen. Die Darstellung ist als Drahtgitter- oder Flächenmodell möglich. Farben, Überhöhungsfaktor und Betrachtungszeitpunkt können beliebig verändert wer-

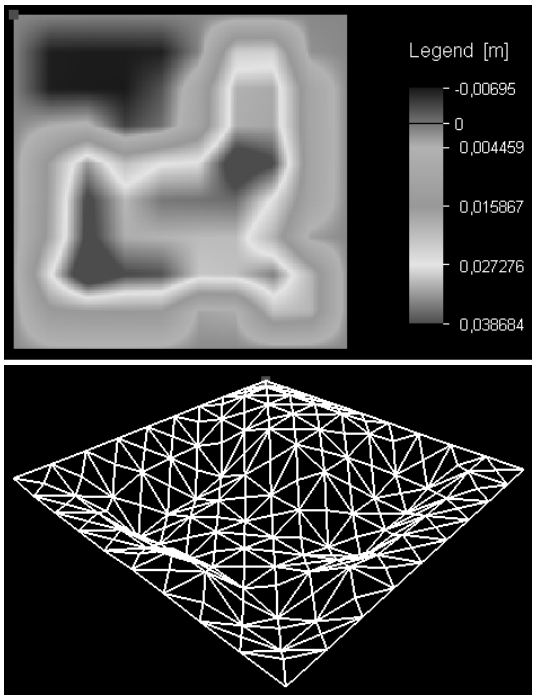

Bild 5.3: Isofläche und 3DModell in *AdConsol-1D*

den. Ebenso kann jede Schicht für sich oder die Summe der Setzungen über alle Schichten hinweg betrachtet werden. Die eingebaute Animationsfunktion erlaubt es den zeitlichen Verlauf der Setzungen zu visualisieren. Die Grafik kann als Bitmap-Datei exportiert werden.

Die Darstellung der 'Isoflächen' geschieht in einem eigenen Fenster (*frmIsoSurface*). Das Berechnungsfeld wird hier als Grundriss dargestellt. Die Verformungen werden über das Farbspektrum von blau nach rot (blau-cyan-grün-gelb-rot) visualisiert. Die Bedeutung der Farbwerte kann in einer Legende abgelesen werden. Auch hier ist es möglich den Betrachtungszeitpunkt zu verändern und zwischen den einzelnen Bodenschichten oder der Setzungssumme über alle Schichten zu wechseln. Die Grafik kann als Bitmap-Datei exportiert werden.

Wie bei der 'Punktberechnung' können auch bei der 'Feldberechnung', Zeit-Setzungs-Diagramme für jeden Berechnungspunkt angezeigt werden. Der Datenexport als DAT-File beschränkt sich hier jedoch, aus speichertechnischen Gründen, auf die Zeitsetzungsdaten der einzelnen Schichten für jeden Berechnungspunkt.

5.1.3 Speicherung der Eingabedaten

Um umfangreiche Last- und Bodenmodelle nicht bei jedem Programmstart neu eingeben zu müssen, ist es sinnvoll eine Möglichkeit zur persistenten Speicherung der Eingabedaten zu bieten. Damit können diese Eingabedaten jederzeit wieder ins Programm geladen werden. In Version 2 ist dies vollkommen anders umgesetzt als in der Vorgängerversion . Sowohl

das Bodenmodell als auch das Lastmodell stellen eine Import- und Export-Funktion zur Verfügung. Dadurch können diese Modelle getrennt voneinander abgespeichert und geladen werden. Die Speicherung erfolgt in strukturierten ASCII-Dateien mit der Endung '*.load' für Last- und '*.soil' für Bodenmodelle. Damit können diese Dateien auch separat editiert werden. Die Speicherung der Projekteinstellungen, wie Berechnungsoptionen, Feldgröße etc. erfolgt in Binärdateien (*.eva) mit der 'Speichern-Funktion'.

5.1.4 Hilfedatei

Damit die englische Programmoberfläche von *AdConsol-1D* für deutschsprachige Anwender besser verständlich wird, wurde die zugehörige Hilfedatei auf Deutsch verfasst. Prinzipiell gibt es zwei Möglichkeiten für Windows Programme Onlinehilfen zu erstellen: html basierte Hilfedateien (*.chm) und herkömmliche Hilfe-Dateien (*.hlp). Da herkömmliche Hilfedateien wesentlich schneller geöffnet werden und auch *Windows95* (ohne IE 3.0) problemlos mit diesen Dateien zurecht kommt, wurde zur Umsetzung diese Variante gewählt.

Dazu muss die Programmhilfe als Richtext-Datei (*.rtf) geschrieben werden. Durch bestimmte Textformatierungen und Fußnoten wird die spätere Funktionalität der Programmhilfe wie das Setzen von 'Hyperlinks' oder die Definition eines 'Stichwortverzeichnisses' festgelegt. Die Kompilierung zur Hilfe-Datei (*.hlp) geschieht mit dem Hilfecompiler 'hcw.exe' der Firma *Microsoft*. Zur Vereinfachung der Hilfeprogrammierung kann auch ein Programm wie *Oasis PRO* verwendet werden.(Doberenz/Kowalski 1999*a*)

Bild 5.4: Onlinehilfe von *AdConsol-1D*

5.2 Modellierung der Datenstruktur

5.2.1 Vorgehensweise

Im Sinne der Objektorientierung wurde beim Entwurf der Datenstruktur von *AdConsol-1D* versucht sinnvolle Objekte zu bilden.

In einem ersten Abstraktionsschritt stellt sich die Frage, welche Daten generell gebraucht werden. Insbesondere die Untersuchung der Berechnungsalgorithmen ist hier von großer Bedeutung. Aber auch alle anderen verwendeten Algorithmen, die Programmoberfläche und Ergebnisausgabe werden untersucht. Am Ende legt man Eingabe-, Ausgabe-, Arbeitsdaten und Konstanten fest.

In einem zweiten Schritt werden diese Daten sinnvoll strukturiert. Die Entscheidungen die hier getroffen werden, können später viel Arbeit einsparen oder auch bereiten. Im Sinne der Abstraktion ist es wichtig logische Datenstrukturen zu identifizieren, um zusammengehörige Daten in Objekte kapseln zu können. Innerhalb der Datenstruktur sollten geeignete Standarddatentypen der Programmiersprache verwendet werden.

Entscheidend für den Zugriff auf die Daten ist die Frage, wo diese Daten zum Einsatz kommen: Lokal in einzelnen Verarbeitungsschritten, im gesamten Modul oder im ganzen Programm. Dementsprechend erfolgt die Deklarierung.

Um den Quellcode leichter lesen zu können, werden für alle Variablen und Objekte sinnvolle Namen und Bezeichnungen, die der Anwendersicht entsprechen, vergeben. Dabei sind die Namenskonventionen von *Visual Basic* (siehe Tabelle 4.1) zu beachten.

5.2.2 Eingabe-Klassen

Zunächst werden die Klassen Last (*clsLoad*) und Bodenschicht (*clsSoil*) gebildet. Diese kapseln alle baumechanischen und geotechnischen Kenngrößen, die für die Berechnung und Darstellung benötigt werden. Zudem stellen sie Methoden zum Zeichnen des Objekts (Paint) oder zur Berechnung der Steinbrennerwerte einer Last (Steinbrenner) zur Verfügung. Die Klasse Punkt (*clsPoint*) dient zur Speicherung und Darstellung eines Berechnungspunktes. Die ermittelten Ergebnisse für jeden Berechnungspunkt werden ebenfalls in dieser Klasse gespeichert damit sie in der Ergebnisausgabe weiterverarbeitet werden können.

Auch für das Berechnungsfeld (*clsField*) und die Berechnungsoptionen (*clsOptions*) werden entsprechende Klassen entwickelt, die spezifische Attribute und Methoden zur Verfügung stellen.

5.2.3 Container-Klassen

Die Klassen Last (*clsLoad*), Bodenschicht (*clsSoil*) und Punkt (*clsPoint*) können während der Programmausführung mehrfach instanziert werden. Daraus ergibt sich der Bedarf, die Instanzen sinnvoll verwalten zu können. In *AdConsol-1D* werden dazu eigene *Container-Klassen* (*clsLoadList, clsSoilList, clsPointList*) verwendet. Jede Container-Klasse enthält ein privates *Collection-Objekt*. Auf dieses *Collection-Objekt* kann nur über Schnittstellen zugegriffen werden, die in der Container-Klasse implementiert werden. Diese Schnittstellen stellen in der Regel die vier Standardmethoden (*Add, Remove, Count, Item*) des *Collection-Objekts* zur Verfügung. Über diese Methoden können die Objekte in der *Collection* manipuliert werden. Darüberhinaus besteht aber auch die Möglichkeit, die *Collection-Objekte* indirekt über ihre Standardmethoden hinaus zu erweitern. So ist zum Beispiel der

komplette Daten-*Import* und -*Export* für Lasten und Bodenschichten in diese *Container-Klassen* integriert. Ebenso sind bestehende Methoden der *Collection-Objekte* durch Methoden wie *Update* oder *RemoveAll* ergänzt worden (siehe Bild 5.5).

Diese Vorgehensweise stellt in *Visual Basic* die übersichtlichste und robusteste Möglichkeit dar, eine Menge gleichartiger Objekte sinnvoll und strukturiert zu verwalten.

5.2.4 Klasse InputGenerator

Die Klasse *clsInputGenerator* übernimmt die Verwaltung aller Eingabe-Klassen. Zur persistenten Speicherung der Eingabedaten stellt diese Klasse die Methode *WriteBinaryFile* bereit. Mit *ReadBinaryFile* können die Daten wieder eingelesen werden. Auch eine Methode zum Umrechnen der Einheiten im Programm ist hier implementiert. Änderungen der Eingabedaten werden in den Attributen *ModellChanged*, *LoadChanged* und *SoilChanged* registriert. Damit kann beim Beenden des Programms nachgefragt werden, ob die Änderungen gespeichert werden sollen (siehe Bild 5.6).

5.2.5 Berechnungs-Klasse

Wie in Kapitel 3.4 erörtert, sind die Berechnungsalgorithmen fest in die Programmstruktur von AdConsol-1D integriert. Um den Austausch der Berechnungsklasse (*clsCalculation*) zu erleichtern, sind alle Berechnungsalgorithmen gut strukturiert in einem einzigen Klassenmodul zusammengefasst. Damit das Programm mit den neuen Berechnungsalgorithmen außerhalb der Entwicklungsumgebung verwendet werden kann, muss es neu kompiliert werden. Der Aufbau der 'Berechnungsklasse' wird in Kapitel 5.4.1 näher beschrieben.

5.2.6 Visualisierungs-Klasse

Die Klasse *clsVisualisation* übernimmt die Verwaltung der Ergebnisausgabe. Sie stellt Methoden zum Aufrufen der einzelnen Ausgabefenster ('Diagramm-Fenster', 'Isoflächen-Fenster', '3D-Modell-Fenster' und 'Export-Dialog') bereit. Mit der Methode *GenerateResults* wird der Ergebnistext generiert. Die Eingabedaten werden von der Methode *InsertInputData* in die Ergebnisausgabe eingefügt. Um das 'DAT-File' mit den Ergebnissen und Berechnungswerten zu erzeugen, steht die Methode *CreateDatFile* zur Verfügung.

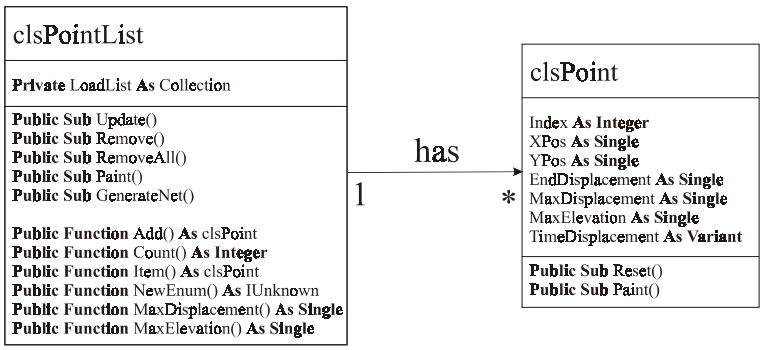

Bild 5.5: Darstellung der Container-Klassen am Beispiel *clsPointList*

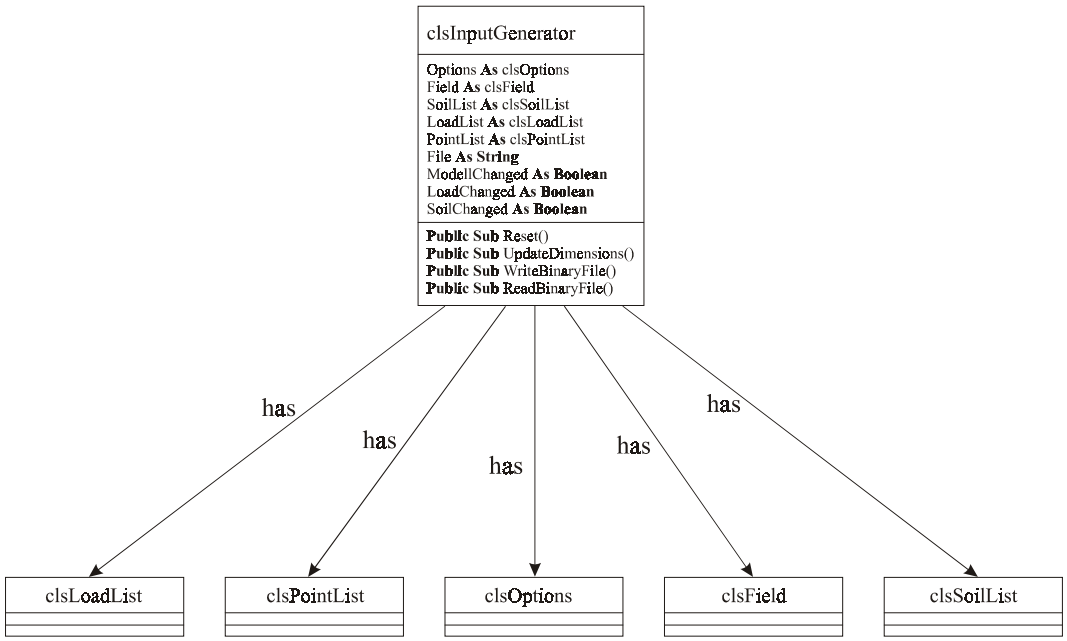

Bild 5.6: Darstellung der Klasse *clsInputGenerator*

Die einzelnen Ausgabefenster der grafischen Benutzeroberfläche werden vom Anwender im 'Ergebnis-Fenster' aufgerufen. Intern werden die Befehle vom 'Ergebnis-Fenster' an die 'Visualisierungs-Klasse' weitergereicht und erst dort die entsprechenden Fenster aufgerufen. Durch die zentrale Steuerung aller Ausgabemöglichkeiten von einer Klasse aus, ist es leicht möglich die Ergebnisausgabe zu erweitern (siehe Bild 5.7).

5.2.7 Andere Module

In den Formular-Modulen sind lediglich die Ereignis-Prozeduren der Steuerelemente implementiert. Nur das 'Ergebnis-Fenster' stellt eine eigene Methode zum Drucken der Ergebnisse bereit. Dadurch kann das komplette Ergebnisfenster leicht in einem neuen Projekt wiederverwendet werden.

Allgemeine Programmkomponenten und allgemein nutzbare Prozeduren wie *Validierungs-*, *Rundungs-* und *Selektions*-Methoden, sind in einem separaten Modul (*modMethods*) implementiert. Zur Unterbringung globaler Konstanten steht das Modul *modConstants* zur Verfügung. Gestartet wird *AdConsol-1D* über eine 'Sub Main Prozedur' im Modul *modApplication*.

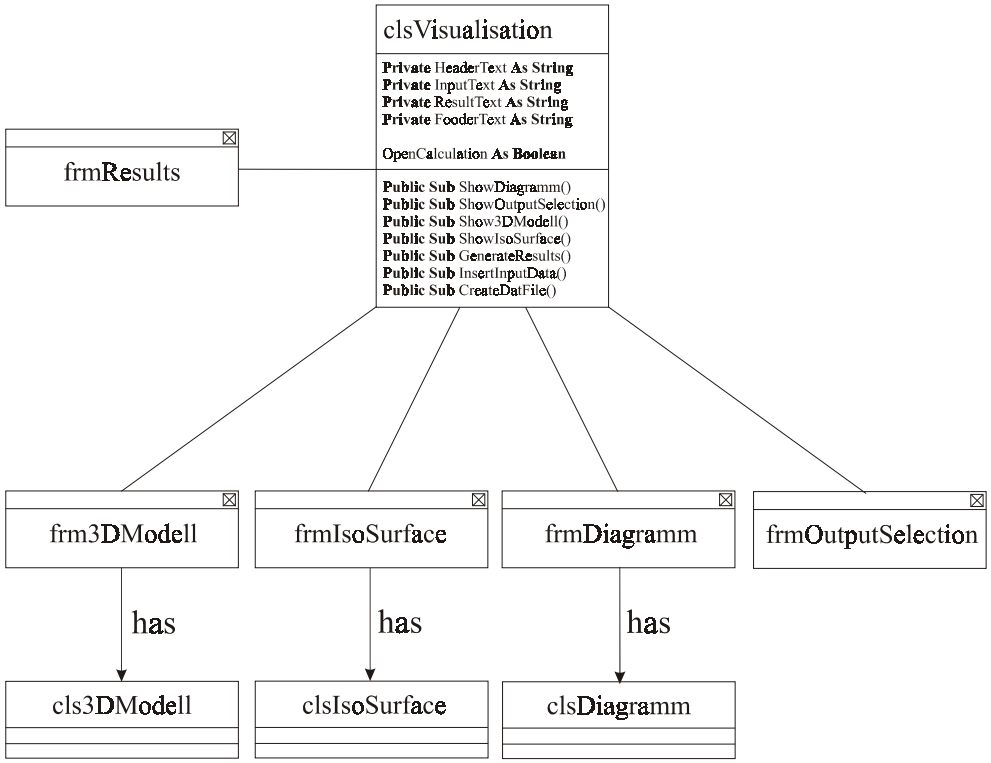

Bild 5.7: Darstellung der Klasse *clsVisualisation*

5.3 Programmstruktur

5.3.1 Aufbau des Programms

Die Programmstruktur von *AdConsol-1D* ist so aufgebaut, dass im Wesentlichen ein 'Kernmodell' (*clsKern*) existiert, das beim Starten des Programms instanziert wird. Dieses 'Kernmodell' instanziert wiederum in die Klassen *clsInputGenerator*, *clsCalculation* und *clsVisualisation*, die für die Eingabe, Berechnung und Ausgabe zuständig sind. Die Klasse *clsInputGenerator* enthält weitere Unterobjekte (*clsField, clsOptions, clsLoadList, clsSoilList*und *clsPointList*), die ebenfalls beim Start geladen werden. Die Instanzen der Klassen *clsLoad*, *clsSoil* und *clsPoint* werden zur Laufzeit angelegt und in den Container-Klassen (*clsLoadList*, *clsSoilList* und *clsPointList*) gespeichert.

Gestartet wird *AdConsol-1D* durch eine 'Sub Main Prozedur' im Modul *modApplication*. Diese Prozedur erzeugt eine Instanz des 'Kernmodells'. Durch die Deklaration *Public* wird diese Instanz als Variable *Kern* dem gesamten Programm zur Verfügung gestellt. Über diese Variable erhält man Zugriff auf alle Objekte des Klassenmodells. Dies ist für die Erweiterbarkeit und Übersichtlichkeit des Programms von großem Vorteil, da sonst alle Variablen in den Methodenköpfen übergeben werden müssten. Weiterhin ruft die 'Main Prozedur' das 'Intro-Fenster' und schließlich die grafische Benutzeroberfläche mit der 'MDI-Umgebung' und dem 'Modell-Fenster' auf. Für die 'MDI-Umgebung' wurde die Formularvorlage *MDI-Formular* verwendet. Diese erleichtert in *Visual Basic* die Erstellung von MDI-Anwendungen, da die in Kapitel 3.3.3 genannten Eigenschaften bereits

vordefiniert sind.

Die Klassen *clsDiagramm*, *clsIsoSurface* und *cls3DModell* werden beim Aufruf der jeweiligen Formulare (*frmDiagramm*, *frmIsoSurface* und *frm3DModell*) instanziert. Sie sind somit in der Programmstruktur fest an diese Formulare gebunden.

Die Klasse *clsWaitCursor* ist eine Sevice-Klasse zur Anzeige des 'Sanduhrsymbols' während längeren Berechnungen. Sie wird von verschiedenen Klassen und Modulen, als lokale Instanz, nur in Prozeduren verwendet.

5.3.2 Klassenmodell

Das Klassenmodell (siehe Bild 5.8) stellt die komplette Programmstruktur in einer modifizierten UML-Schreibweise dar. Darin sind alle Formulare, Module und Klassen des Programms zu finden. Außerdem ist dargestellt welche Dateien im Programm gelesen (*read*) oder geschrieben (*write*) werden.

Eigentlich können Steuerelemente auf Formularen in *Visual Basic* von überall aus angesprochen werden. Die Verbindungslinien sollen in der Darstellung symbolisieren, welche Abhängigkeiten tatsächlich bestehen. Wenn nicht anders angegeben, sind die Kardinalitäten stets eins zu eins. Pfeile mit der Anmerkung 'has' verdeutlichen, dass eine Klasse Bestandteil einer Anderen ist.

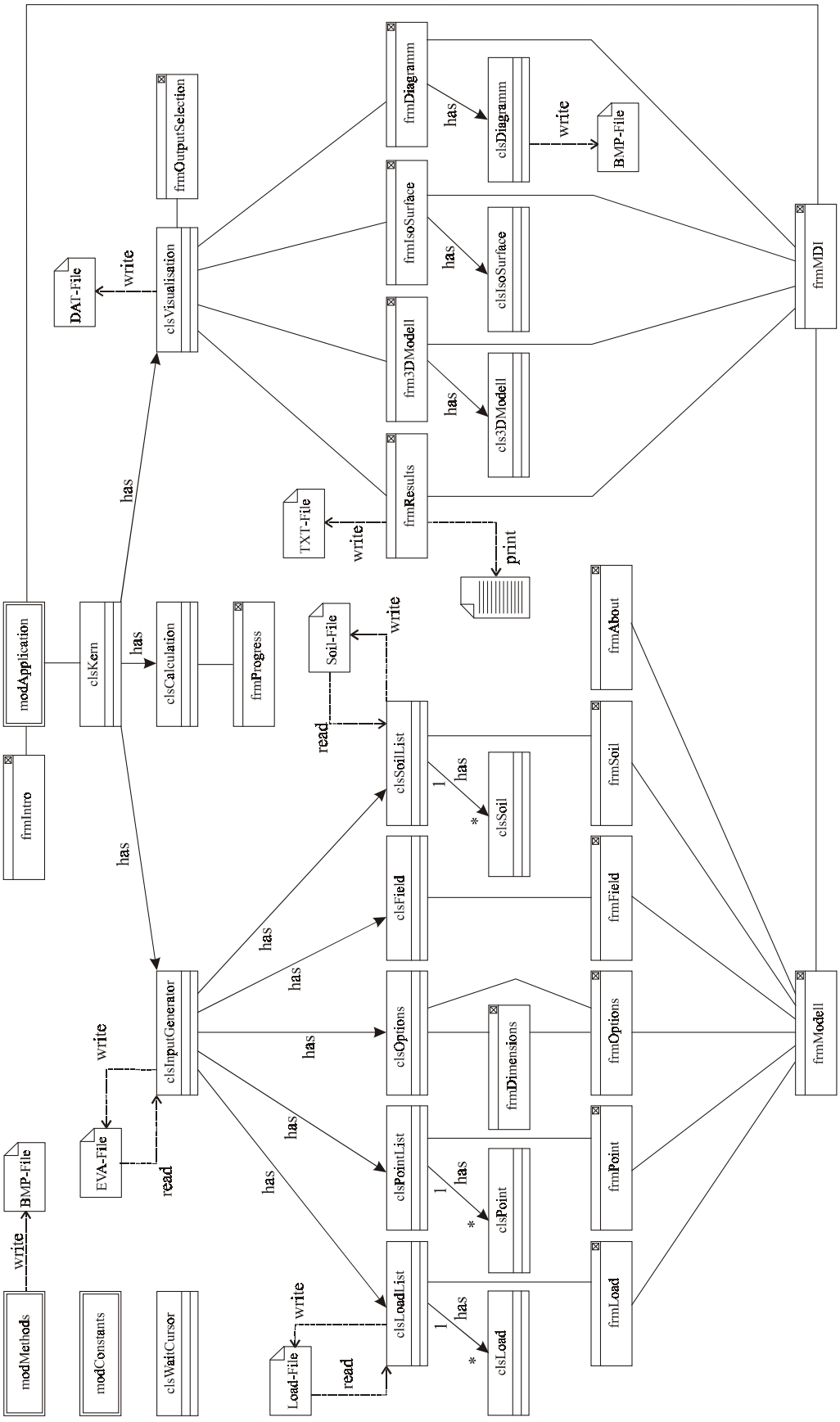

Bild 5.8: Klassenmodell von *AdConsol-1D*

5.3.3 Aufbau der einzelnen Klassen

Im Folgenden werden die einzelnen Klassen tabellarisch dargestellt. Dabei werden die öffentlichen Attribute und Methoden (=*Interface* der jeweiligen Klasse) mit Namen, Deklaration und Übergabeparametern angegeben.

Tabelle 5.1: Tabellarische Darstellung der Klasse clsKern. Sie beinhaltet das komplette Datenmodell von *AdConsol-1D*.

clsKern	
Attribute:	*Typ:*
InputGenerator	clsInputGenerator
Visualisation	clsVisualisation
Calculation	clsCalculation
Keine öffentlichen Methoden	

Tabelle 5.2: Tabellarische Darstellung der Klasse clsInputGenerator. Diese stellt eine Art 'Verwaltungsklasse' für alle Eingabeklassen dar.

clsInputGenerator	
Attribute:	*Typ:*
Options	clsOptions
Field	clsField
SoilList	clsSoilList
LoadList	clsLoadList
PointList	clsPointList
File	String
ModellChanged	Boolean
LoadChanged	Boolean
SoilChanged	Boolean
Methoden:	*Übergabeparameter:*
Reset()	-
WriteBinaryFile()	-
ReadBinaryFile()	-
UpdateDimensions()	MultiTime As Single, MultiLength As Single, MultiForce As Single, NewTimeDim As String, NewLengthDim As String, NewForceDim As String

Tabelle 5.3: Tabellarische Darstellung der Klasse clsField, die alle Eingabedaten des Berechnungsfeldes beinhaltet.

clsField	
Attribute:	*Typ:*
FieldWidth	Single
FieldHeight	Single
GroundwaterLevel	Single
GlobalInitialLoad	Single
Methoden:	*Übergabeparameter:*
Reset()	-
Paint()	-

Tabelle 5.4: Tabellarische Darstellung der Klasse clsOptions, die für die Verwaltung der Berechnungsoptionen und Programmeinstellungen zuständig ist.

clsOptions	
Attribute:	*Typ:*
DeltaT	Integer
CalculationTime	Long
GravityWater	Single
Inkrement	Integer
ShowZero	Byte
PointCalculation	Boolean
FieldCalculation	Boolean
NumberOfColumns	Byte
NumberOfRows	Byte
DimensionTime	String
DimensionLength	String
DimensionForce	String
Methoden:	*Übergabeparameter:*
Reset()	-

Tabelle 5.5: Tabellarische Darstellung der Klasse clsLoadList, die als Collection-Klasse dient und das komplette Lastmodell beinhaltet.

clsLoadList	
Keine Attribute	
Methoden:	*Übergabeparameter:*
Reset()	-
UpdateListBox()	-
Remove()	Index As Integer
RemoveAll()	-
Paint()	-
Export()	-
Import()	-
Update()	Index As Integer, Name As String, Color As Long, Time As Long, Value As Single, Selekted As Boolean, Aktivated As Byte, InitialLoad As Byte, XPos As Single, YPos As Single, Width As Single, Height As Single
Add() As clsLoad	Name As String, Color As Long, Time As Long, Value As Single, Selekted As Boolean, Aktivated As Byte, InitialLoad As Byte, XPos As Single, YPos As Single, Width As Single, Height As Single
Count() As Integer	-
NewEnum() As IUnknown	-
Item() As clsLoad	Index As Integer

Tabelle 5.6: Tabellarische Darstellung der Klasse clsLoad, die zum Speichern der Eingabedaten einer Last dient.

clsLoad	
Attribute:	*Typ:*
Index	Integer
Name	String
Color	Long
Time	Long
Value	Single
Selected	Boolean
InitialLoad	Byte
Aktivated	Byte
XPos	Single
YPos	Single
Width	Single
Height	Single
Methoden:	*Übergabeparameter:*
Reset()	-
Paint()	-
ToString() As String	-
Steinbrenner() As Single	Depth As Single, XPoint As Single, YPoint As Single

Tabelle 5.7: Tabellarische Darstellung der Klasse clsPointList, die als Collection-Klasse dient und alle Berechnungspunkte beinhaltet.

clsPointList	
Keine Attribute	
Methoden:	*Übergabeparameter:*
Update()	Index As Integer, XPos As Single, YPos As Single
Remove()	Index As Integer
RemoveAll()	-
Paint()	-
GenerateNet()	-
Add() As clsPoint	XPos As Single, YPos As Single
Count() As Integer	-
Item() As clsPoint	Index As Integer
NewEnum() As IUnknown	-
MaxDisplacement() As Single	-
MaxElevation() As Single	-

Tabelle 5.8: Tabellarische Darstellung der Klasse clsPoint, die zur Speicherung der Eingabedaten eines Berechnungspunktes inklusive dessen Berechnungsergebnissen dient.

clsPoint	
Attribute:	*Typ:*
Index	Integer
XPos	Single
YPos	Single
EndDisplacement	Single
MaxDisplacement	Single
MaxElevation	Single
TimeDisplacement	Variant
Methoden:	*Übergabeparameter:*
Reset()	-
Paint()	-

Tabelle 5.9: Tabellarische Darstellung der Klasse clsSoilList, die als Collection-Klasse dient und alle Bodenschichten beinhaltet.

clsSoilList	
Attribute:	*Typ:*
TotalThickness	Single
Methoden:	*Übergabeparameter:*
Reset()	-
Update()	Index As Byte, Name As String, Color As Long, Drained As Boolean, Thickness As Single, Gravity As Single, GravityBuoyanced As Single, FirstLoading As Single, PowerFirstLoading As Single, Reloading As Single, PowerReloading As Single, Permeability As Single, FrictionAngle As Single, Cohesion As Single, InitialVoidRatio As Single, ChangePermeability As Single, OverconsolidationStress As Single, Dummy1 As Single, Dummy2 As Single, Dummy3 As Single, Dummy4 As Single
RemoveAll()	-
Remove()	Index As Byte
Paint()	-
Export()	-
Import()	-
Add() As clsSoil	Name As String, Color As Long, Drained As Boolean, Thickness As Single, Gravity As Single, GravityBuoyanced As Single, FirstLoading As Single, PowerFirstLoading As Single, Reloading As Single, PowerReloading As Single, Permeability As Single, FrictionAngle As Single, Cohesion As Single, InitialVoidRatio As Single, ChangePermeability As Single, OverconsolidationStress As Single, Dummy1 As Single, Dummy2 As Single, Dummy3 As Single, Dummy4 As Single
Count() As Byte	-
Item() As clsSoil()	Index As Byte
NewEnum() As IUnknown	-

Tabelle 5.10: Tabellarische Darstellung der Klasse clsSoil, die zum Speichern der Eingabedaten einer Bodenschicht dient.

clsSoil	
Attribute:	*Typ:*
Index	Byte
Depht	Single
Name	String
Color	Long
Drained	Boolean
Thickness	Single
UnitWeight	Single
UnitWeightBuoyant	Single
FirstLoading	Single
PowerFirstLoading	Single
Reloading	Single
PowerReloading	Single
Permeability	Single
FrictionAngle	Single
Cohesion	Single
InitialVoidRatio	Single
ChangePermeability	Single
OverconsolidationStress	Single
Dummy1	Single
Dummy2	Single
Dummy3	Single
Dummy4	Single
Methoden:	*Übergabeparameter:*
Reset()	-
Paint()	-
DeadWeight() As Single	Depth As Single

Tabelle 5.11: Tabellarische Darstellung der Klasse clsCalculation, die den kompletten Berechnungsalgorithmus beinhaltet.

clsCalculation	
Attribute:	*Typ:*
Canceled	Boolean
StartTime	Single
ErgDepthM	Variant
ErgSigmaM	Variant
ErgDeadWeight	Variant
ErgSigmaSummeM	Variant
ErgK	Variant
ErgE	Variant
ErgU	Variant
ErgSigmaEffektiv	Variant
ErgDisplacement	Variant
ErgTimeDisplacement	Variant
ErgSigmaInitArray	Variant
Methoden:	*Übergabeparameter:*
Calculate()	-

Tabelle 5.12: Tabellarische Darstellung der Klasse clsVisualisation zur Verwaltung der Ergebnisausgabe.

clsVisualisation	
Attribute:	*Typ:*
OpenCalculation	Boolean
Methoden:	*Übergabeparameter:*
ShowDiagramm()	-
ShowOutputSelection()	-
Show3DModell()	-
ShowIsoSurface()	-
GenerateResults()	-
InsertInputData()	-
CreateDatFile()	File As String

Tabelle 5.13: Tabellarische Darstellung der Klasse cls3DModell, die als Steuerungsklasse zur Darstellung des 3D-Modells im '3D-Modell-Fenster' (frm3DModell) dient.

cls3DModell	
Attribute:	*Typ:*
Bank	Long
Grid	Boolean
Bleed	Byte
GridColorRed	Byte
GridColorGreen	Byte
GridColorBlue	Byte
SurfaceColorRed	Byte
SurfaceColorGreen	Byte
SurfaceColorBlue	Byte
Intervall	Integer
Layer	Byte
Methoden:	*Übergabeparameter:*
Initialize()	Hwnd As Long, Hdc As Long
Display()	AddPhi As Single, AddPsi As Single, MultiZoom As Single
OnSize()	Width As Long, Height As Long
SetWorldList()	-
MakeCurrent()	-

Tabelle 5.14: Tabellarische Darstellung der Klasse clsDiagramm, die als Steuerungsklasse zur Darstellung des Diagramms im 'Diagramm-Fenster' (frmDiagramm) dient.

clsDiagramm	
Keine Attribute:	
Methoden:	*Übergabeparameter:*
Initialize()	-
Paint()	-
Export()	-

Tabelle 5.15: Tabellarische Darstellung der Klasse clsIsoSurface, die als Steuerungsklasse zur Darstellung der Isoflächen im 'Isoflächen-Fenster' (frmIsoSurface) dient.

clsIsoSurface	
Attribute:	*Typ:*
Intervall	Integer
Layer	Byte
Methoden:	*Übergabeparameter:*
Initialize()	Hwnd As Long, Hdc As Long
OnSize()	Width As Long, Height As Long
DrawField()	-
Display()	-
MakeCurrent()	-

Tabelle 5.16: Tabellarische Darstellung der Klasse clsWaitCursor, die als Service-Klasse zur Darstellung der Sanduhr bei längeren Berechnungen dient.

clsWaitCursor
Keine Attribute:
Keine öffentlichen Methoden

5.4 Berechnungsklasse

5.4.1 Aufbau der Berechnungsklasse

Die Berechnungsklasse gliedert sich in folgende vier Abschnitte:

- Ablaufsteuerung (Koordination des Berechnungsablaufs)
- Vorberechnung (Vorbereiten der Daten und Datenfelder)
- Konsolidationsberechnung (Berechnung der Verformungen)
- Nachberechnung (Aufbereiten der Ergebnisse)

Die Ablaufsteuerung besteht aus der öffentlichen Methode *Calculation*. Diese steuert den kompletten Berechnungsvorgang und ruft dabei die Methoden der Vorberechnung,

Nachberechnung und die Konsolidationsberechnung selbst auf. In Abhängigkeit des Berechnungsmodus erfolgt die Berechnung der Verformungen für einen Berechnungspunkt ('Punktberechnung') oder über alle Berechnungspunkte im generierten Netz ('Feldberechnung') (siehe Bild 5.9).

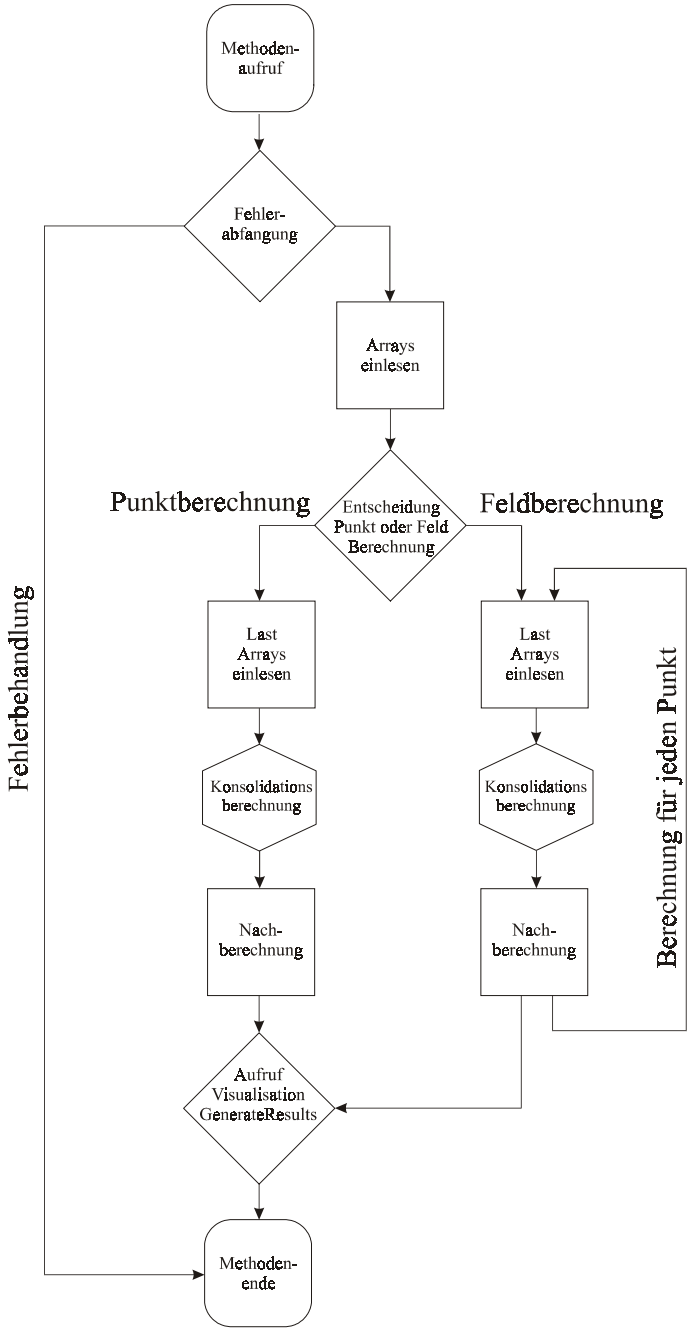

Bild 5.9: Blockdiagramm der Ablaufsteuerung

Die Konsolidationsberechnung erfolgt in jedem Berechnungspunkt nach zwei Dimensionen: der Zeit und der Tiefe. Die Anzahl der Zeitintervalle (*AnzInt*) wird durch die Länge der Berechnungszeit (*CalculationTime*) und der Zeitschrittweite (*DeltaT*) ausgedrückt. Jede Bodenschicht wird nochmals in 'Inkremente' unterteilt. Dadurch ergibt sich aus der Anzahl der Bodenschichten (*AnzLayer*) und der Anzahl der Inkremente (*Inkrement*) die Anzahl der Berechnungsschritte über die Tiefe (*AnzInk*) (siehe Bild 5.10).

Zur Vereinfachung des Berechnungsalgorithmus werden in der Konsolidationsberechnung dynamische Arrays verwendet. Um die Handhabung der einzelnen Bodenschichten im Berechnungsalgorithmus zu verbessern, werden die geotechnischen Kenngrößen der einzelnen Schichten inkrementweise in eindimensionale Arrays geschrieben. Dies hat den Vorteil, dass nicht während der Berechnung entschieden werden muss, welches Inkrement beispielsweise in welcher Bodenschicht liegt und welcher entsprechende Bodenkennwert verwendet werden muss.

Die Zuordnung der Werte zu den Arrays und die Dimensionierung erfolgt in der Vorberechnung.

In der Nachberechnung werden die Berechnungsergebnisse aufgearbeitet, sodass sie in der Ergebnisausgabe visualisiert werden können. Dazu werden die 'maximale Setzung', die 'maximale Hebung', die 'Endsetzung' und das 'schichtbezogene Zeit-Setzungs-Array' berechnet und als Attribute den Berechnungspunkten zugewiesen.

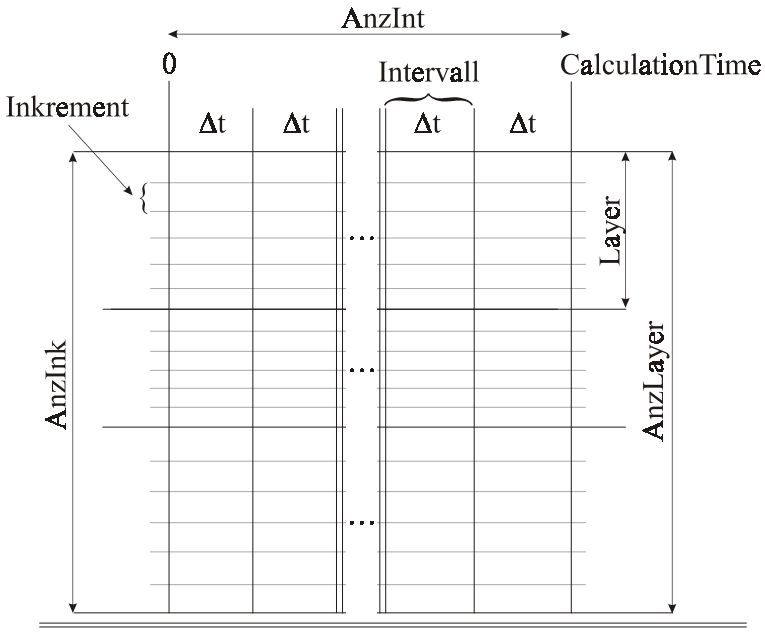

Bild 5.10: Darstellung der Berechnungsvariablen

5.4.2 Beschreibung der Variablen

Um die Berechnung so effektiv und so schnell wie möglich zu gestalten, werden die Variablen nicht größer als nötig deklariert. Die Aufgabenstellung fordert eine 'unbegrenzte' Anzahl von Bodenschichten und Lasten. Da jedoch alle Variablen nur einen bestimmten Wertebereich umfassen können, ist dies praktisch unmöglich. Deswegen kann man sich nur darauf beschränken die Variablen so groß zu dimensionieren, dass der 'normale' Anwender im praktischen Einsatz niemals an die Grenzen der Dimensionierung stößt. Bei der Implementierung von *AdConsol-1D* wurden die Anzahl der Bodenschichten (*AnzLayer*) zum Beispiel als Byte-Wert deklariert, da in der Praxis ein Schichtmodell kaum mehr als 255 Bodenschichten umfasst. Ebenso wurde eine mögliche Anzahl von 32.767 Lasten als ausreichend angesehen und damit die Variable *AnzLoad* als Integer-Wert deklariert.

Tabelle 5.17: Berechnungsvariablen

Variable	**Typ**	**Beschreibung**
AnzInk	Integer	Gesamtanzahl der Berechnungsinkremente
AnzInt	Long	Gesamtanzahl der Berechnungsintervalle
AnzLayer	Byte	Gesamtanzahl der Bodenschichten
AnzLoad	Integer	Gesamtanzahl der Lasten
Inkrement	Integer	Anzahl der Inkremente je Bodenschicht
DeltaT	Integer	Zeitschrittweite
GravityWater	Single	Wasserwichte
TotalThickness	Single	Gesamtdicke des Bodenprofils
GlobalSigmaInit	Single	Globale Initiallast
Cancel	Boolean	Steuervariable zum Programmabbruch

Tabelle 5.18: Eindimensionale Berechnungs-Arrays

Array	**Typ**	**Beschreibung je Inkrement**
Depth(0 To AnzInk)	Single	Tiefe an den Inkrement-Grenzen
DepthM(1 To AnzInk)	Single	Tiefe in der Inkrement-Mitte
DeadWeight(0 To AnzInk)	Single	Eigengewicht an den Inkrement-Grenzen
DeadWeightM(1 To AnzInk)	Single	Eigengewicht in der Inkrement-Mitte
LayerNr(1 To AnzInk)	Byte	Zugehörigkeit zum Bodenschicht-Index
SigmaInitArray(0 To AnzInk)	Single	Initiallast an den Inkrement-Grenzen

Tabelle 5.19: Bodenprofil Arrays

Bodenprofil Arrays	**Typ**	**Beschreibung je Inkrement**
Drained(0 To AnzInk + 1)	Boolean	drainiert oder nicht drainiert
Thickness(0 To AnzInk)	Single	Dicke
UnitWeight(1 To AnzInk)	Single	Wichte
UnitWeightBuoyant(1 To AnzInk)	Single	Wichte unter Auftrieb
FirstLoading(1 To AnzInk)	Single	Erstbelastungswert
PowerFirstLoading(1 To AnzInk)	Single	Exponent der Erstbelastung
Reloading(1 To AnzInk)	Single	Wiederbelastung
PowerReloading(1 To AnzInk)	Single	Exponent der Wiederbelastung
Permeability(1 To AnzInk)	Single	Durchlässigkeit
FrictionAngle(1 To AnzInk)	Single	Reibungswinkel
Cohesion(1 To AnzInk)	Single	Kohäsion
InitialVoidRatio(1 To AnzInk)	Single	Initiale Porenzahl
ChangePermeability(1 To AnzInk)	Single	Änderung der Durchlässigkeit
OverconsolidationStress(1 To AnzInk)	Single	Überkonsolidationsspannung
Dummy1(1 To AnzInk)	Single	Testvariable 1
Dummy2(1 To AnzInk)	Single	Testvariable 2
Dummy3(1 To AnzInk)	Single	Testvariable 3
Dummy4(1 To AnzInk)	Single	Testvariable 4

Tabelle 5.20: Zweidimensionale Berechnungs-Arrays

Array	**Typ**	**Beschreibung je Inkrement und Zeitintervall**
Sigma(0 To AnzInk, 0 To AnzInt)	Single	Spannungen an den Inkrement-Grenzen
SigmaM(1 To AnzInk, 0 To AnzInt)	Single	Spannungen in der Inkrement-Mitte
SigmaSumme(0 To AnzInk, 0 To AnzInt)	Single	Über die Intervalle summierte Spannungen an den Inkrement-Grenzen
SigmaSummeM(1 To AnzInk, 0 To AnzInt)	Single	Über die Intervalle summierte Spannungen in der Inkrement- Mitte
Displacement(1 To AnzInk, 0 To AnzInt)	Single	Verformung in der Inkrement-Mitte
SigmaEffektiv(0 To AnzInk, 0 To AnzInt)	Single	Effektive Spannung an den Inkrement-Grenzen
SigmaEffektivM(1 To AnzInk, 0 To AnzInt)	Single	Effektive Spannung in der Inkrement-Mitte
SigmaEffektivMMax(1 To AnzInk, 0 To AnzInt)	Single	Maximale effektive Spannung in der Inkrement-Mitte
TimeDisplacement(0 To AnzLayer, 0 To AnzInt)	Single	Zeit-Setzungs-Array je Bosenschicht und Setzungssumme aller Schichten

5.4.3 Ergebnisse der Konsolidationsberechnung

Folgende Ergebnisse und Zwischenergebnisse stellt die Berechnungsklasse *clsCalculation* als Attribute dem Programm zur weiteren Verarbeitung bereit.

Tabelle 5.21: Eindimensionale Ergebnis-Arrays

Array	**Typ**	**Beschreibung je Inkrement**
DepthM(1 To AnzInk)	Single	Tiefe in der Inkrement-Mitte
SigmaInitArray(0 To AnzInk)	Single	Initiallast an den Inkrement-Grenzen
DeadWeight(0 To AnzInk)	Single	Eigengewicht an den Inkrement-Grenzen

Tabelle 5.22: Zweidimensionale Ergebnis-Arrays

Array	**Typ**	**Beschreibung je Inkrement und Zeitintervall**
SigmaM(1 To AnzInk, 0 To AnzInt)	Single	Spannungen in der Inkrement-Mitte
SigmaSummeM(1 To AnzInk, 0 To AnzInt)	Single	Über die Intervalle summierte Spannungen in der Inkrement-Mitte
k(1 To AnzInk, 0 To AnzInt)	Single	Durchlässigkeit in der Inkrement-Mitte
e(1 To AnzInk, 0 To AnzInt)	Single	Porenzahl in der Inkrement-Mitte
u(0 To AnzInk, 0 To AnzInt)	Single	Porenwasserdruck an den Inkrement-Grenzen
SigmaEffektiv(0 To AnzInk, 0 To AnzInt)	Single	Effektive Spannung an den Inkrement-Grenzen
Displacement(1 To AnzInk, 0 To AnzInt)	Single	Verformung in der Inkrement-Mitte
TimeDisplacement(0 To AnzLayer, 0 To AnzInt)	Single	Zeit-Setzungs-Array je Bodenschicht und Setzungssumme aller Schichten

5.5 Grafik in *AdConsol-1D*

5.5.1 Grafikprogrammierung unter Windows

Einführung

Mit der rasanten Entwicklung der Computertechnologie stehen die Möglichkeiten der grafischen Datenverarbeitung einer immer breiteren Masse von Anwendern zur Verfügung. Dementsprechend stieg in den letzten Jahren die Nachfrage für entsprechende Anwendungsprogramme stetig an. Davon betroffen sind nicht nur Unterhaltungsprogramme, wie Computerspiele. Auch im technischen Bereich werden beispielsweise Rendering-Funktionen in CAD- oder FEM-Systeme integriert.

Grafik-Schnittstellen

Um die Anwendungsprogrammierer von der oft komplexen Aufgabe zu befreien, die Methoden der Computergrafik algorithmisch zu erfassen und zu implementieren, gibt es seit einiger Zeit Bemühungen um einheitliche Grafikschnittstellen (APIs).

Im Bereich der 3D-Grafik unter *Windows*, gibt es heutzutage nur zwei große APIs, die in direkter Konkurrenz zueinander stehen. *Direct3D* aus dem *DirectX*-Paket von *Microsoft* und die freie *OpenGL*. Beide APIs können in *Visual Basic* Programmen verwendet werden.

Vorteile von *OpenGL*

Als einer der erfolgreichsten und langlebigsten Schnittstellen-Standards hat sich die *OpenGL* herauskristallisiert. Heutzutage werden *OpenGL* Runtime Bibliotheken mit allen *Unix-, MacOS-, Linux-* und *Windows*-Systemen ausgeliefert. Im Gegensatz dazu existiert *Direct3D* nur in der *Windows*-Welt.

Da die 'Type Library' (vbogl.tlb), die jedem Setup Paket mitgegeben werden muss, lediglich 539 kB groß ist und *OpenGL* auf fast jedem *Windows*-System dargestellt werden kann (vorausgesetzt die Grafikkarte kann *OpenGL*-Befehle verarbeiten), wird zur Umsetzung von *AdConsol-1D* auf diese Grafikschnittstelle zurückgegriffen. (edx April 2003)(Melchart/Pojar April 2003)

OpenGL-Befehle werden wesentlich schneller verarbeitet als die *Visual Basic* eigenen Grafikbefehle. Deshalb wurde sowohl das '3D-Modell' als auch die 'Isoflächen' in *OpenGL* implementiert.

5.5.2 Einführung in die *OpenGL*

Was ist die *OpenGL*?

OpenGL steht für 'Open Graphics Library' und ist eine Grafikbibliothek zur Beschreibung und Darstellung zwei- und dreidimensionaler Objekte.

Ihre historischen Wurzeln hat die *OpenGL* in der von *Silicon Graphics* entwickelten Bibliothek *SGI IRIS GL*. Nach der Lizenzfreigabe wurde sie sozusagen zum 'Industriestandard'. Seitdem wird die *OpenGL* von einem firmenübergreifenden Konsortium - *Architecture Review Board (ARB)* genannt - standardisiert, gepflegt und weiterentwickelt. Die *OpenGL* ist heutzutage durch Public Domain Portierungen für alle gängigen Plattformen verfügbar und gehört im Allgemeinen zur Standarddistribution eines Betriebssytems. Die aktuelle Revision von *OpenGL* ist 1.3.

Als hardwareunabhängige Grafikschnittstelle kann die *OpenGL* auf verschiedenen Rechner- und Betriebssystemplattformen eingesetzt und in unterschiedlichen Programmiersprachen implementiert werden. Aus diesem Grund enthält *OpenGL* jedoch keine Befehle zur Verwaltung von Fenstern und zur Verarbeitung von Benutzereingaben. *OpenGL* ist primär für die Darstellung von Grafik und das Rendering verantwortlich.

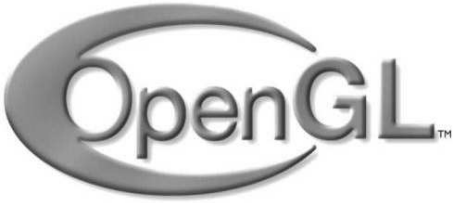

Bild 5.11: *OpenGL*-Logo

Die *OpenGL* ist aus zwei getrennten Bibliotheken aufgebaut: der *Basis GL* (*Graphics Library*) sowie der *GLU* (*OpenGL Utility Library*). Darüber hinaus gibt es vom System abhängige Erweiterungen und Zusatzbibliotheken. Auch *Microsoft* Windows besitzt *OpenGL*-Erweiterungen (*WGL*). Diese sind bereits in der Basis-Bibliothek (opengl32.dll) implementiert.

Da die *OpenGL* auf maximale Geschwindigkeit bei der Darstellung von 3D-Objekten ausgelegt ist, beherrscht sie keine Raytracing- und Radiosity-Rechnungen. Für Effekte wie beispielsweise Schatten und Spiegelungen muss der Anwendungsprogrammierer mit viel Aufwand selbst sorgen. *OpenGL* ist deshalb nichts für photorealistische Bilder, sie eignet sich jedoch aufgrund ihrer hohen Geschwindigkeit ideal für Virtual-Reality-Anwendungen oder 3D-Modellierungsprogramme. Mit dieser Funktion steht *OpenGL* zwar als Grafikbibliothek nicht allein, doch ist sie im Profi-Bereich (*Silicon Graphics*) unumstrittener Standard. (Melchart/Pojar April 2003)

Zusatzbibliotheken

Als prozedurales Grafiksystem kennt *OpenGL* zunächst nur ganz einfache geometrische Objekte wie Punkte, Linien, Dreiecke und Polygone. Zusatzbibliotheken dienen zur Vereinfachung der Handhabung und der Ergänzung von *OpenGL*. Die wichtigsten sind die *OpenGL Utility Library* zur Darstellung komplexer geometrischer Objekte wie Kugeln, Zylinder, Splines und das *OpenGL Utility Toolkit*, das Funktionen zur Fenster- und Ereignisbehandlung, wie Maus- und Tastaturabfragen, bietet. (Mönch/Queck 2001)

5.5.3 *OpenGL* Programmierung

Namenskonventionen

In der *OpenGL* existieren Konventionen hinsichtlich der Namensgebung von Funktionen, Variablen und Parametern. Diese sind in Tabelle 5.23 dargestellt:

Der *OpenGL*-Befehlssatz besteht aus circa 150 verschiedenen Funktionen. Für viele dieser Funktionen existieren mehrere Varianten, die sich in Anzahl und Datentyp ihrer Parameter unterscheiden. Das Suffix des Funktionsnamen gibt Auskunft über Art und Anzahl der zu erwartenden Parameter. Der Befehl *glBefehl3f()* beispielsweise bedeutet, dass die Funktion als Argument drei Zahlen vom Typ *Float* erwartet. Wird allgemein von einer Funktion gesprochen, schreibt man *glBefehl*()*.

Tabelle 5.23: *OpenGL* Namenskonventionen

OpenGL	**Syntax**	**Beispiel**
OpenGL-Funktion	glBefehl()	glBegin()
OpenGL-Parameter	GL_PARAMETER	GL_ENABLED
OpenGL-Typ	GLtyp	GLfload

Raumpunkte im Koordinatensystem

Ein Raumpunkt (*Vertex*) wird durch seine Koordinaten im kartesischen Koordinatensystem beschrieben. Überträgt man das Koordinatensystem auf den Bildschirm, so zeigt die positive z-Koordinate aus dem Bildschirm heraus. Der Befehl für einen Raumpunkt lautet:

Codebeispiel:

```
glVertex* (x, y, z)
```

Programmierung eines Primitives

Als 'Primitive' werden in der Computergrafik die einfachsten Ausgabeelemente bezeichnet. Zu den 'Primitiven' gehören Punkte, Linien und Polygone. Aus ihnen kann man komplexe grafische Objekte zusammensetzen. Der Programmblock zur Programmierung eines 'Primitives' hat folgende Struktur:

Codebeispiel:

```
glBegin(GL_PRIMITIV_NAME)
    glVertex*(x,y,z)
        '...
    glVertex*(x,y,z)
glEnd()
```

Eingeleitet wird der Programmblock mit dem Befehl *glBegin()*, abgeschlossen mit *glEnd()*. Dabei bestimmt der Parameter *GL_PRIMITIV_NAME*, wie das Ergebnis aussieht, beziehungsweise welches Primitiv gezeichnet werden soll. Dazwischen befindet sich eine Liste von Raumpunkten. Diese stellen die Eckpunkte eines Primitives dar. In Tabelle 5.24 sind die wichtigsten *OpenGL*-Grafikprimitive aufgeführt: (Barth 2002) (Kuska 2000)

Tabelle 5.24: *OpenGL* Grafikprimitive

Primitivname	**Beschreibung des Primitivs**
POINTS	*Einzelne Punkte:* Jeder Raumpunkt spezifiziert die Positionen eines Punktes.
GL_LINES	*Liniensegmente:* Zwei aufeinanderfolgende Raumpunkte beschreiben eine Linie.
GL_LINE_STRIP	*Linienzug:* Die Raumpunkte werden entsprechend ihrer Reihenfolge in Form eines Linienzuges verbunden.
GL_LINE_LOOP	*Geschlossener Linienzug:* Die Raumpunkte werden entsprechend ihrer Reihenfolge in Form eines Linienzuges verbunden. Der Ringschluß erfolgt automatisch.
GL_POLYGON	*Polygon:* Die Raumpunkte beschreiben eine Fläche, die gefüllt ist. Polygone müssen konvex sein!
GL_QUADS	*Viereck:* Jeweils vier aufeinanderfolgende Raumpunkte bilden ein Viereck.
GL_QUAD_STRIP	*Verbundene Vierecke:* Zusammenhängende ausgefüllte Vierecke
GL_TRIANGLES	*Dreieck:* Jeweils drei aufeinanderfolgende Raumpunkte bilden ein Dreieck.
GL_TRIANGLE_FAN	*Dreiecksfächer:* Um den Anfangspunkt (0) gruppieren sich die Dreiecke.
GL_TRIANGLE_STRIP	*Verbundene Dreiecke:* Zusammenhängende ausgefüllte Dreiecke

Transformationen

Die Platzierung der Grafikprimitive im Raum kann durch Transformationen geändert werden. Die drei wichtigsten Transformationen sind *Translation* (Verschiebung), *Rotation* (Drehung) und *Skalierung* (Größenänderung). (Barth 2002)

Translation: Verschiebung eines Objekts um den Vektor (dx, dy, dz)

Codebeispiel:

```
glTranslate*(dx, dy, dz)
```

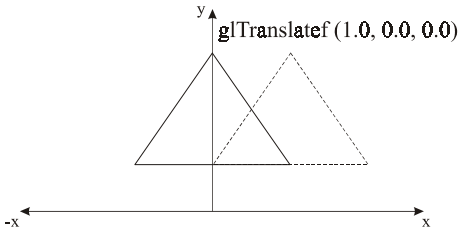

Bild 5.12: Translation in *OpenGL*

Rotation: Drehung eines Objekts um die Achse, parallel zum Vektor (x, y, z) durch den Ursprung

Codebeispiel:

```
glRotate*(Winkel, x, y, z)
```

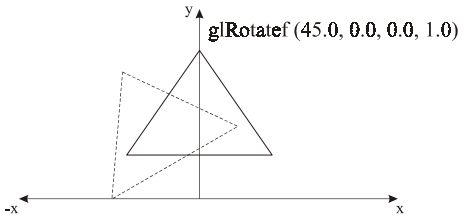

Bild 5.13: Rotation in *OpenGL*

Skalierung: Größenveränderung eines Objekts je Achse

Codebeispiel:

```
glScale*(sx, sy, sz)
```

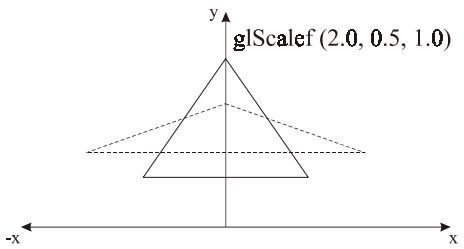

Bild 5.14: Skalierung in *OpenGL*

Projektions- und Ansichtstransformationen

Um eine Szene mit dreidimensionalen Körpern betrachten zu können, muss man dem Grafiksystem mitteilen, von welcher Position aus in welche Blickrichtung gesehen wird. Dies kann in *OpenGL* mit dem *gluLookAt* Befehl geschehen. Das 'Auge des Betrachters' (*Viewpoint*) befindet sich dabei an der Position mit den Koordinaten (eyex, eyey, eyez). Der Punkt, der betrachtet wird (*Center*), liegt auf den Koordinaten (centerx, centery, centerz). Die Parameter upx, upy und upz legen fest, welche Richtung oben ist. (Barth 2002)

Codebeispiel:

```
gluLookAt(eyex,eyey,eyez,centerx,centery,centerz,upx,upy,upz)
```

In *AdConsol-1D* sind die Koordinaten des *Center* immer (0, 0, 0). Die Koordinaten des *Viewpoints* werden programmintern durch Polarkoordinaten (φ und ψ) beschrieben. Dadurch lässt sich die Kamera leichter um das Berechnungsfeld bewegen, das durch eine *Translation* in den Ursprung geschoben wird. Der Abstand zwischen *Center* und *Viewpoint* kann mit dem 'Zoomfaktor' Θ verändert werden. Der Befehl *gluLookAt* benötigt die Koordinaten des *Viewpoints* jedoch als karthesische Koordinaten. Mittels einfacher Koordinatentransformation werden diese aus den Polarkoordinaten errechnet (siehe Bild 5.15).

$$eyeX = \Theta \cdot \cos(\varphi) \cdot \cos(\psi) \tag{5.1}$$

$$eyeY = \Theta \cdot \sin(\varphi) \cdot \cos(\psi) \tag{5.2}$$

$$eyeZ = \Theta \cdot \sin(\psi) \tag{5.3}$$

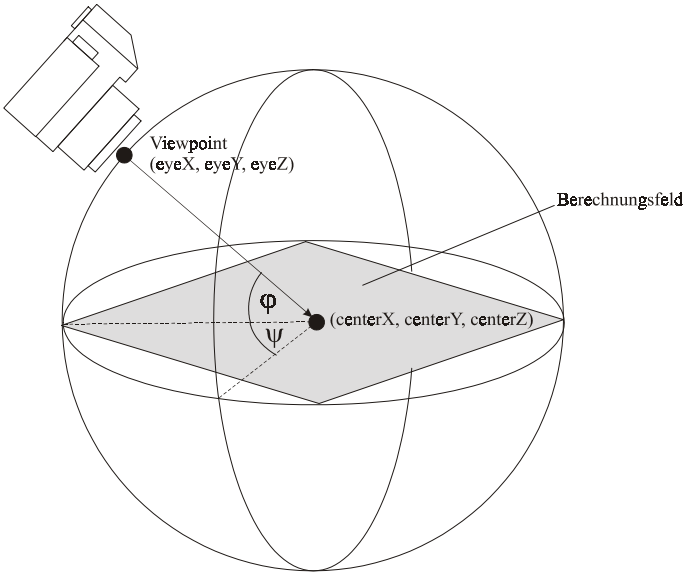

Bild 5.15: Kameraposition in *AdConsol-1D*

Nach Festlegung des *Viewpoints* und der Ansichtstransformation, werden die Kameraparameter und die Art der Projektion definiert. Prinzipiell kann man bei den projektiven Abbildungen zwei Typen unterscheiden. Die 'Parallelprojektion' (parallele Sehstrahlen) und die 'Perspektivprojektion' (perspektivische Sehstrahlen). In *AdConsol-1D* wurde die 'Perspektivprojektion' gewählt. Dies kann in *OpenGL* mit dem Befehl *gluPerspective* erfolgen (siehe Bild 5.16). (Barth 2002)

Codebeispiel:

```
gluPerspective(alpha, aspect, near, far)
```

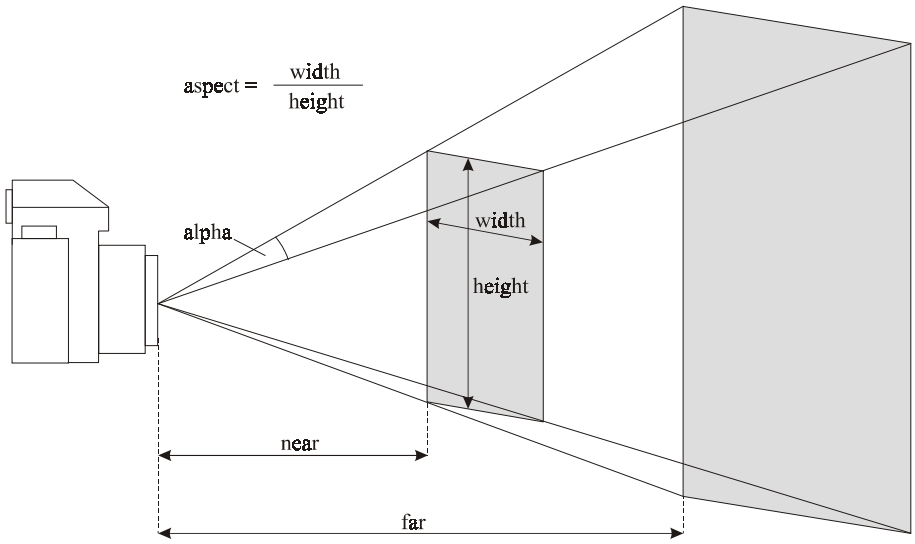

Bild 5.16: Perspektive in *OpenGL*

Kapitel 6

Ausblick

Things are more complex than they seem to be!
Things take longer than expected!
If something can go wrong - it will!

[Murphy's Law]

Murphy was an Optimist!

[Corollar]

6.1 Erweiterungsmöglichkeiten

Durch den objektorientierten Aufbau und die gewählte Programmstruktur ist es leicht möglich *AdConsol-1D* weiter auszubauen. Beispielhaft werden hier einige Möglichkeiten genannt.

6.1.1 Gitteroptimierung

Es wäre sinnvoll, das automatisch erzeugte Gitternetz für das jeweilige Lastmodell zu optimieren. Dazu müsste lediglich ein geeigneter Algorithmus entwickelt werden, der den aktuellen Algorithmus zur Erzeugung des regelmäßigen Gitters ersetzt. Dieser befindet sich in der Methode *GenerateNet()* der Klasse *clsPointList*.

6.1.2 Mehrere Bodenprofile

Im Moment wird idealisiert angenommen, dass für das gesamte Berechnungsfeld ein einheitliches Bodenprofil besteht. Tatsächlich wird dies niemals den realen Gegebenheiten in der Natur entsprechen. Deswegen wäre es sinnvoll, das Programm in diese Richtung weiter zu entwickeln. Dies könnte so gestaltet werden, dass die Reihenfolge der Bodenschichten weiterhin für das komplette Berechnungsfeld gilt, die Dicken der einzelnen Bodenschichten jedoch, je nach gewähltem Profil, über das Berechnungsfeld variiert werden können. Dazu müsste das Klassenmodell durch die 'Container-Klasse' *clsProfilList* und die Klasse

clsProfil ergänzt werden. Das Attribut *Thickness* in der Klasse *clsSoil* würde entfallen und die Berechnungsalgorithmen müssten entsprechend modifiziert werden.

6.1.3 Programmregistrierung

Von Seiten der Geotechnik wurde der Wunsch geäußert, das Programm geschützt weitergeben zu können. Um zu Verhindern, dass das Programm kopiert wird und auf fremden Rechnern zum Einsatz kommt, besteht die Möglichkeit *AdConsol-1D* durch einen 'Software-Dongle' zu schützen. Dazu müsste die Programmstruktur lediglich durch eine Registrierungsklasse *clsRegister* erweitert werden, in der zum Beispiel die Festplattennummer mit einem verschlüsselten Freischaltcode verglichen wird.

6.1.4 Erweiterungen der Ergebnisausgabe

Auch die Ergebnisausgabe lässt sich einfach erweitern. Werden neue Ergebnisdarstellungen gewünscht, müssen nur Kleinigkeiten in der Visualisierungsklasse und im Formular der Ergebnisausgabe verändert werden. Die neuen Ergebnis-Klassen und -Fenster können damit in die Programmstruktur integriert werden.

6.1.5 Zeitlich variabler Grundwasserspiegel

Um zum Beispiel die Auswirkungen einer Grundwasserabsenkung zu simulieren, wäre es sinnvoll, den Grundwasserspiegel zeitabhängig eingeben zu können. Bei der Umsetzung müssten lediglich Änderungen in den Klassen *clsField* und *clsCalculation* vorgenommen werden. Außerdem müsste man dem Programm ein neues Formular zur Bearbeitung der Grundwasserlinie hinzufügen.

6.2 Beschleunigung der Berechnung

Visual Basic bietet effektive Möglichkeiten, grafische Benutzeroberflächen für *Windows* zu konstruieren. Leider gehören jedoch *Visual Basic* Programme, von der Ablaufgeschwindigkeit her gesehen, nicht zu den schnellsten. Dieses Manko macht sich auch beim Berechnungsalgorithmus in *AdConsol-1D* bemerkbar.

6.2.1 Optimierung des Programmcodes

Eine Möglichkeit den Berechnungsablauf zu beschleunigen, besteht darin die verwendeten Algorithmen zu optimieren. Zu großen Teilen ist dies in *AdConsol-1D* schon geschehen. So wurde beispielsweise in der Berechnungsklasse der aufwändige 'Gauss-Algorithmus' durch den 'Thomas-Algorithmus' ersetzt. Zur Optimierung von Algorithmen gehört ebenfalls die Verwendung der richtigen Datentypen für Variablen und (nach Möglichkeit) die Vermeidung des Datentyps *Variant*. Darüber hinaus gibt es noch eine ganze Reihe weiterer Möglichkeiten, *Visual Basic* Code zu optimieren, wie zum Beispiel die Verwendung von *With* beim mehrmaligen Referenzieren eines Objekts oder das Ersetzen der Potenzierung in mathematischen Ausdrücken. Nach einiger Zeit gelangt man jedoch mit diesen Methoden der Programmoptimierung an die Grenzen der verwendeten Programmiersprache. Eine

zusätzliche Geschwindigkeitssteigerung ist dann kaum mehr möglich. (Shadoware April 2003)

6.2.2 Erstellung eigener DLLs

Hohe Performance, wie sie mit *C++* erreicht wird, ist nicht in jedem Teil eines Programms notwendig. Ob beispielsweise die Überprüfung einer Eingabe eine Millisekunde oder zehn Millisekunden dauert, ist für den Anwender nicht wahrnehmbar. Nur die sogenannten 'Flaschenhälse' eines Programms müssen deswegen entsprechend optimiert werden.

So ist es möglich, Funktionen bei denen es auf hohe Performance ankommt, in *C++* zu schreiben und als *Dynamic Link Libraries (DLLs)* zu exportieren. Diese DLL kann anschließend in *Visual Basic* als API-Funktion verwendet werden. Bei der Erstellung von Setup-Routinen müssen die DLLs mitgegeben werden. (Shadoware April 2003) (Krumsiek Mai 2003)

6.3 Komponentenbasierte Softwareentwicklung

6.3.1 Software-Komponenten

Moderne Softwaresysteme bestehen heute nicht mehr aus einem monolithischen Block, sondern sind aus einer Vielzahl von Softwarekomponenten aufgebaut. Eine Komponente ist für die Außenwelt eine 'Black Box', die Implementierung ist vor dem Benutzer verborgen. Auf eine Komponente kann nur über ihr öffentliches Interface zugegriffen werden. Komponenten sind sprachunabhängig und liegen meist in Binärcode vor. Durch sie werden komplexe Probleme in mehrere kleine Teilprobleme zerlegt. Daraus ergeben sich in der Entwicklung eine Reihe von Vorteilen. (Gnoth 2000)

6.3.2 Geschichte

Eines der ersten in der Praxis erfolgreichen Systeme, das eine komponentenbasierte Softwareentwicklung ermöglichte, war *Visual Basic*. Bereits in Version 1.0 konnten sogenannte *VBX-Controls* (*Visual Basic eXtensions*) in ein Programm integriert werden. Diese Controls waren in erster Linie dafür vorgesehen, die *Windows*-Programmierung zu vereinfachen. Bereits 1993 gab es einen etablierten Markt an *VBX-Controls* von Drittanbietern. *VBX-Controls* waren nicht nur für Oberflächenelemente verfügbar, sondern auch für effiziente mathematische Algorithmen, Zugriff auf Netzprotokolle, Bildverarbeitung und natürlich Datenbanken. Durch Ankauf von Komponenten mussten Entwickler nicht mehr alle Teile einer Anwendung selbst entwerfen. (Gnoth 2000)

6.3.3 Softwareentwicklung heute

Heutige Softwareentwicklung unter *Windows* wird nicht mehr von der Wahl des Programmierwerkzeuges bestimmt. Wichtiger ist es die richtigen Komponenten auszuwählen und diese mit vergleichsweise wenigen Codezeilen zu einem funktionierenden Ganzen zusammenzubauen. Darin liegt auch der Erfolg von 'High Level Sprachen' wie *Visual Basic* und *Java* bei der Erstellung von Informationssystemen begründet. Diese Sprachen bieten einen höheren Abstraktionsgrad und verbergen viele (für die Problemlösung unerhebliche)

technische Details. Es werden Programmierfehler vermieden und die Entwicklungszeit beschleunigt.

Für die Erstellung von effizienten Komponenten wiederum sind 'Low Level Sprachen' wie *C++* aufgrund ihrer größeren Systemnähe besser geeignet. Der Entwickler braucht gute Kenntnisse der technischen Interna des Betriebssystems. Hier ist eindeutig ein Umbruch in der Softwareindustrie erkennbar. Einige Firmen haben sich auf die Erstellung solcher Komponenten spezialisiert, sodass man heute für nahezu jede Aufgabenstellung Komponenten erwerben kann. Andere Firmen bauen Informationssysteme und setzen dafür solche Komponenten ein. (Gnoth 2000)

Zusammenfassung

Als Resultat dieser Diplomarbeit ist das Programm *AdConsol-1D* entstanden mit dem es möglich ist das zeitliche Setzungsverhalten mehrschichtiger Böden mit Hilfe von spannungsabhängigen Steifigkeiten und porenzahlabhängigen Durchlässigkeiten zu berechnen. Die Ergebnisse können grafisch und numerisch dargestellt werden.

Grundlage ist die Setzungsberechnung nach DIN 4019 (Methode der indirekten Setzungsberechnung) und eine von *Dipl. -Ing. Martin Zimmerer* angepasste Konsolidationstheorie. Diese stützt sich auf die Fließgesetze von *Darcy* und die Konsolidationstheorie von *Terzaghi*. Zur Berücksichtigung von spannungsabhängigen Steifemodulen und porenzahlabhängigen Durchlässigkeiten wurde die *Terzaghi*-Theorie ingenieurmäßig modifiziert.

Aus der Vorgängerversion, die bereits im Sommer 2002 als Studienarbeit angefertigt wurde, konnten nur einige bewährte Konzepte der grafischen Benutzeroberfläche übernommen werden. Die interne Programm- und Datenstruktur wurde komplett neu entwickelt. Im Sinne der Erweiterbarkeit wurde dabei besonderes Augenmerk auf die Nutzung objektorientierter Konzepte gelegt.

Durch die Dreiteilung der Programmstruktur in Eingabemodell, Berechnung und Visualisierung ist ein flexibles, leicht erweiterbares Programm entstanden. Über eine Kernklasse werden die drei Teile des Programms angesprochen. Auf diese Kernklasse kann von überall im Programm zugegriffen werden. Dies vereinfacht die zukünftige Erweiterbarkeit von *AdConsol-1D* erheblich.

Um neue Berechnungsalgorithmen testen zu können, wurden diese in einer einzigen 'Berechnungsklasse' zusammengefasst. Durch einfaches Austauschen dieser Klasse kann das Programm auf neue Berechnungsweisen umgestellt werden.

Die Funktionalität der ersten Programmversion wurde vollkommen neu überarbeitet und um etliche wesentliche Features erweitert. So sind Lasten nicht mehr an eine vorgegebene Zellenstruktur gebunden, sondern können beliebig positioniert werden. Auch ein freies Setzen der Berechnungspunkte im Berechnungsfeld ist jetzt möglich. Die Einführung des Berechnungsmodus 'Feldberechnung' erforderte eine Neuentwicklung der Ergebnisausgabe. Zur Darstellung der zeitabhängigen Verformungen in einem dreidimensionalen Modell und zur Ausgabe von Isoflächen wurden eigene Ausgabefenster entwickelt. Eine Onlinehilfe unterstützt den Anwender bei der Arbeit mit dem Programm.

Kompiliert läuft *AdConsol-1D* relativ schnell und stabil. Lediglich die komplexen Berechnungsalgorithmen bremsen das Programm, vor allem bei umfangreichen Modellen, etwas aus. Mit *Visual Basic* allein ist hier jedoch kaum mehr Nachbesserung möglich. Um höhere Ausführungsgeschwindigkeiten zu erreichen, müssten neue Wege eingeschlagen werden.

Anhang A

Inhalt der CD

Verzeichnis Code:

Hier sind alle zum Kompilieren des Programms notwendigen Dateien zu finden. Um den Quellcode besser nachvollziehen zu können, ist dieser mit Kommentaren versehen. Dies betrifft vor allem selbst definierte Prozeduren und Methoden deren Funktion jeweils am Methodenkopf kurz erläutert wird. Längere Prozeduren werden durch Kommentare untergliedert. Ereignisprozeduren sind nicht näher kommentiert, da sich deren Funktion in der Regel von selbst erklärt.

Verzeichnis Dokumentation:

Die Datei *AdConsol.pdf* stellt die vorliegende Diplomarbeit in 'digitaler Form' dar.

Verzeichnis Hilfe:

Beinhaltet alle Dateien, die zur Erstellung der Hilfedatei mit *Oasis PRO 1.4* notwendig sind

Verzeichnis Latex:

Diese Diplomarbeit wurde mit LaTeX angefertigt. Alle Dateien, die zur Erstellung notwendig sind liegen in diesem Verzeichnis.

Verzeichnis Programm:

Enthält das ausführbare Programm *AdConsol.exe*, die Hilfedatei *AdConsol.hlp* und die Help-Contents-Datei *AdConsol.cnt*

Verzeichnis Setup:

Installationsprogramm für *AdConsol-1D* inklusive Programmhilfe

Verzeichnis Support:

- *OpenGL* Type Library (vbogl.tlb)

- Service Pack 5 für *Visual Basic 6*
- *Oasis PRO 1.4* als Shareware Version
- Hilfecompilier *hcwsetup.exe* von *Microsoft*
- *Windows*-System DLL zur Darstellung von *OpenGL* (GLU32.DLL)
- *Windows*-System DLL zur Darstellung von *OpenGL* (OPENGL32.DLL)

Anhang B

Systemanforderungen & Installation

B.1 Systemanforderungen

Um das Programm einsetzen zu können, sollte ihr Rechner folgende Mindestausstattung und Systemkonfiguration aufweisen.

- 486er Prozessor mit 16 MB Arbeitsspeicher
- *SVGA* Grafikkarte die *OpenGL* darstellen kann
- Mindestens 8 MB freier Speicher auf der Festplatte
- Installiertes *Windows*-Betriebssystem (ab *Windows95*)
- Bildschirmauflösung von mindestens 600 x 800 Pixel
- Mindestens 16 Bit Farbtiefe
- Dezimaltrennzeichen = 'Komma'
- Symbol für Zifferngruppierung = 'Punkt'

Anmerkung: Da die Berechnungen extrem rechenintensiv sind, sollte nach Möglichkeit ein schneller Prozessor mit viel Arbeitsspeicher verwendet werden.

B.2 Installation

1. Schließen sie alle laufenden Programme.

2. Klicken sie die Datei *setup.exe* im Verzeichnis *Setup* auf der CD doppelt an. Das Setupprogramm leitet sie dann menügeführt durch die Installation.

3. Unter Umständen sind einige Installationsdateien bereits auf Ihrem System in einer neueren Version vorhanden. Es erscheint dann die Meldung "Die Datei ... ist bereits in einer neueren Version vorhanden... Soll die Datei beibehalten werden?" Diese Frage sollten sie mit [JA] beantworten.

4. Ebenso kann es passieren, dass zur Durchführung der Installation ihre Systemdateien aktualisiert werden müssen (insbesondere wenn sie *Windows95* oder eine ältere Version von *Windows98* installiert haben). Dies übernimmt ebenfalls das Installationsprogramm für sie. Es erscheint dann die Meldung "Setup kann nicht fortgesetzt werden, da einige Systemdateien auf ihrem System nicht mehr aktuell sind ...". Um die Aktualisierung abzuschließen, starten sie den Computer neu und rufen sie das Installationsprogramm nochmals auf. (Datei *setup.exe* doppelt anklicken etc.)

5. Unter Umständen kann es erforderlich sein die Programmhilfe manuell in das Programm einzubinden. Beim Aufrufen der Hilfe in *AdConsol-1D* erscheint dann die Meldung "... wollen Sie selbst nach der Hilfe suchen?" Diese Frage beantworten sie mit [JA]. Daraufhin geben sie den Pfad ein. Die Hilfedatei *AdConsol.hlp* befindet sich im Installationsverzeichnis.

Anhang C

Ausdruck der Hilfedatei

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe.

Weimar, 22. Mai 2003

Thesen

Bei der Gestaltung grafischer Benutzeroberflächen besitzen marktbeherrschende industrielle Produkte normsetzende Kraft. Um den Anwender nicht zu verwirren, sollte man sich deshalb beim Design einer Software möglichst an die quasi 'Industriestandards' halten.

Durch das Zusammenwirken von komfortabler grafisch-interaktiver Handhabung und standardisierter Gestaltung der Programmoberfläche können praxisorientierte Statikprogramme entwickelt werden. Diese gestatten dem Benutzer zu jeder Zeit den Überblick zu behalten und dennoch die Flexibilität und Leistungsfähigkeit moderner Berechnungsmethoden voll auszuschöpfen.

Visual Basic ist ideal zum Erstellen von 'Prototypen' im 'Windows-Standard' geeignet. Durch die RAD-Entwicklungsumgebung können effizient grafische Oberflächen erstellt und auf die Ansprüche des 'Auftraggebers' schnell und flexibel eingegangen werden.

Auch mit *Visual Basic* lassen sich gut erweiterbare und übersichtlich strukturierte Programme, nach objektorientierten Prinzipien entwickeln. Nicht alle Projekte benötigen dabei die 'echte' Vererbung von Programmcode. So spielt bei der Implementierung von *AdConsol-1D* diese Manko von *Visual Basic* ebenfalls keine Rolle.

Durch die Dreiteilung eines Programms in *Eingabe*, *Berechnung* und *Visualisierung* (*MVC-Konzept*) können Berechnungsprogramme gut strukturiert werden. Diese Aufteilung gleicht dem logischen Datenfluss und ist deshalb für ingenieurmäßige Anwendungen besonders gut geeignet. Ebenso entspricht sie dem traditionellen Denkmodell der Datenverarbeitung *EVA* (Eingabe - Verarbeitung - Ausgabe).

Berechnungsalgorithmen können durch Skripte oder Berechnungsklassen ausgetauscht werden. Je nach Komplexität sind Skripte eher für kleinere Aufgaben, Berechnungsklassen mehr für komplexe Algorithmen geeignet. Die Ausführungsgeschwindigkeit ist bei Skripten wesentlich geringer, da diese interpretiert werden.

Die Umsetzung von *AdConsol-1D* hat gezeigt, dass der Einsatz von 'Berechnungsklassen' für die Implementierung der geotechnischen Algorithmen gut geeignet sind.

Ergebnisse lassen sich unterschiedlich visualisieren. Für die ingenieurmäßige Darstellung von Verformungen einer Belastungsfläche haben sich besonders 'Isoflächen' bewährt. Dreidimensionale Visualisierungen sind besser zur Veranschaulichung für Laien geeignet.