John Guttag
Abstrakte Datentypen

von Thomas Haufe

Problemseminar

„Wegweisende Arbeiten in der Softwaretechnik“

Volker Gruhn, Ralf Laue

Sommersemester 2004

Inhaltsverzeichnis:

1. Einleitung

2. Einführung der abstrakten Datentypen 1975
2.1. Widerstände gegen abstrakte Datentypen
2.2. Ursachen für die Einführung der abstrakten Datentypen

3. Spezifikation abstrakter Daten
3.1. Abstrakte Daten
3.2. Spezifikationen
3.2.1. Erweiterung um Spezifikationen
3.2.2. Exemplarisch imperative Spezifikation
3.2.3. Axiomatisch imperative Spezifikation
3.2.4. Exemplarisch applikative Spezifikation
3.2.5. Axiomatisch applikative Spezifikation
3.3. Abstrahierende Funktion

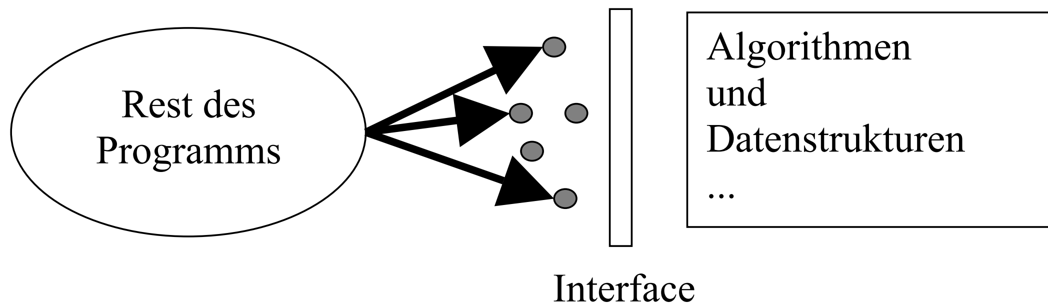
4. Probleme

5. Zusammenfassung

6. Quellen

1. Einleitung

Abstrakte Datentype bestehen aus einer Menge von Objekten und Operationen auf diesen Objekten.
Durch eine Spezifikation der Operatoren wird ein Interface zwischen den abstrakten Datentypen und dem Rest des Programms definiert. Dieses Interface beschreibt das Verhalten der Operationen. Dabei werden die Algorithmen und Datenstrukturen der Operationen vom Rest des Programms isoliert.



John Guttag griff bei der Entwicklung der abstrakten Datentypen auf teilweise vorhandene Ideen zurück:
Parnas erkannt die Vorteile des Information Hiding, Dahl und Nygaard legten die Grundlagen der Abstraktion in Simula 67 fest, Wirth und Liskov haben gezeigt wie man mit abstrakten Datentypen programmieren für den Programmierer sicherer machen kann und Hoare zeigte wie man Monitore benutzen kann, um Programme und dazugehörige Axiome zu strukturieren.

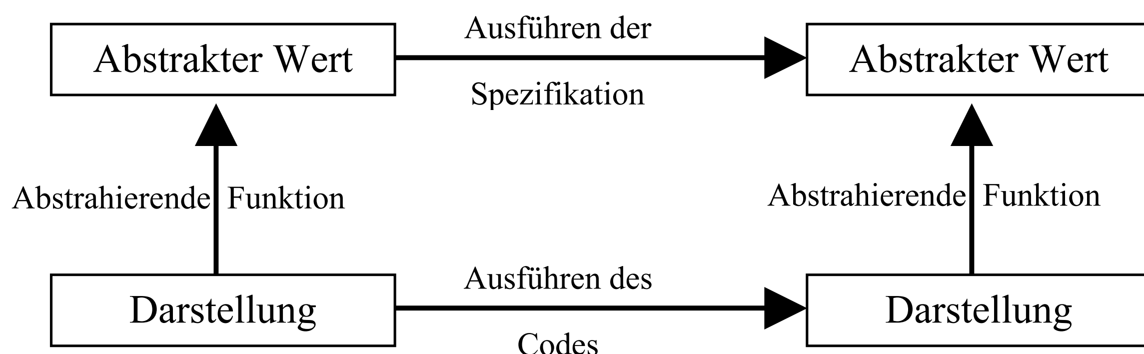
2. Einführung der Abstrakten Datentypen 1975

2.1. Widerstände gegen Abstrakte Datentypen

1975 entwickelte John Guttag in seiner Dissertation unter Leitung von Jim Horning das Konzept der abstrakten Datentypen.
Zu dieser Zeit wurde das Konzept allerdings mehrheitlich von Wirtschaft und Forschung abgelehnt. Die Gründe hierfür liegen in den unterschiedlichen Annahmen die den Programmen bei der Entwicklung zugrundelagen.
Die erste wichtige Annahme Guttags war, dass Software kein Fraktal ist.
Dies stand im Widerspruch zur allgemeinen Annahmen „it's the same thing all the way down". Was den Grundsatz beschreibt, dass die Softwareentwicklung in allen Entwicklungsstufen, von Design bis zur Implementierung , ein rekursiver Prozess ist.
Eine weiter Annahme die den abstrakten Datentypen zugrunde liegt ist, dass Wissen gefährlich ist. Menschen können nur eine begrenzte Menge Wissen aufnehmen und Programmiermethoden, die annehmen dass man viel versteht, sind sehr riskant. Zuviel über die Struktur eine Programms zu wissen ist meist schlechter als zu wenig zu wissen.
Auch diese Ansicht wurde meist nicht geteilt. Besonders Studenten waren häufig der Meinung, dass der Quellcode eines Software-Projektes jedem Mitarbeiter gleichermaßen gehört.
Diese Ansichten hatten weitreichende Folgen bis heute. Wäre das Prinzip der abstrakten Datenstrukturen in den 70er Jahren bereits verstanden und angewandt wurden, hätte es beispielsweise keine „Jahr 2000 Problem" gegeben und die Umstellung der Zeitskala wäre sehr einfach zu realisieren gewesen.
Ein weiterer Kritikpunkt war, dass durch das Interface nur ein begrenzter Zugriff auf die Datenstrukturen möglich ist, der die Kosten für diesen Zugriff erheblich erhöht. Dem entgegen stellt Guttag seine Annahme, das die Performance eines Programms nicht ausschließlich durch lokale Entscheidungen bei den Datenzugriffen beeinflusst wird, sondern auch durch die globale Struktur des Programms. Dies kommt besonders bei Modifikationen und Wartungen zum tragen, da an diesen Stellen die Performance ohne abstrakte Datenstruktur erheblich sinkt.
Eine letzte Annahme die Guttag den Programmen zugrunde legte war, dass durch domainspezifische Datentypen bessere Programme erzeugt werden können als ohne.
Auch diese Meinung wurde nicht von allen geteilt, oft waren Programmierer der Ansicht, dass acht Datentypen ausreichend seinen.

2.2. Gründe für Einführung der Abstrakten Datentypen

Der Grund für die Weiterentwicklung dieser Annahmen, ist der Versuch kommerzielle Softwareentwicklung zu verbessern, dass heißt Zeit, Kosten und Qualität zu optimieren. Die Verbesserung der Programmstruktur durch abstrakte Datentype dient dabei nur als ein Hilfsmittel.
Das Ziel Guttags war es zu verstehen wie man am besten Software herstellt, um damit die Herstellung einfach zu machen. Heute glaubt aber auch Guttag nicht mehr daran, dass es möglich ist Software einfach herzustellen.
Das einzige was möglich ist, ist eine Steigerung der Effizienz. Dazu steht nun zusätzlich zur Dekomposition der Mechanismus der Abstraktion zur Verfügung.
Durch die Abstraktion ist es möglich bestimmte Details zu unterdrücken und so domainspezifische Datentypen zu erzeugen die eine bequeme Dekomposition erlauben.
Ein weiterer Vorteil der Abstraktion ist die Darstellungsunabhängigkeit.
Die Implementierung eines abstrakten Datums besteht aus mehreren Teilen:
der Implementierung der Operationen, den Datenstrukturen die die Werte enthalten und einer Vereinbarung wie die Operationen auf die Datenstrukturen angewendet werden können. Diese Operationen sind die Darstellungsinvariante und die abstrahierende Funktion.
Die Darstellungsinvarianten legen fest wie instanzierte Datenstrukturen behandelt werden und die abstrahierende Funktion ist eine Abbildung der Werte der implementierten Datenstruktur auf die abstrakten Werte.



Das kommutative Diagramm zeigt die Darstellungsunabhängigkeit.
Die Anwendung einer konkreten Operation auf einen instanzierten Wert (links unten) erzeugt einen weiteren konkreten Wert (rechts unten). Wird auf diesen die abstrahierende Funktion angewandt, erhält man den gleichen abstrakten Wert (rechts oben), den man erhalten würde wenn man erst die abstrahierende Funktion auf den konkreten Ausgangswert anwendet und dann auf den erhaltenen abstrakten Wert (links oben) die Spezifikation ausführt.
Das Diagramm zeigt also, dass das Verhalten des Programms durch die Spezifikation bzw. das Interface bestimmt wird. Als Konsequenz daraus wird die Konstruktion von Clientprogrammen erleichtert, da der Client nicht mehr die Implementierung der abstrakten Daten beachten muss.

3. Spezifikation Abstrakter Daten

3.1. Abstrakte Daten

Die Eigenschaften eines Datenobjektes lassen sich in 2 Klassen einteilen: Eigenschaften, die die Implementierung des Objektes beschreiben und Eigenschaften, die die Namen und abstrakten Bedeutungen eines Objektes definieren.
Diese zwei Eigenschaften sind unabhängig und definieren unterschiedlich logische Konzepte.
Die einzigen implementierungsunabhängigen Informationen sind der Definitionsbereich und der Wertebereich der Operationen. Betrachten wir das Beispiel eines Stack:

new:	$\rightarrow$ Stack
push:	Stack x Item $\rightarrow$ Stack
top:	Stack $\rightarrow$ Item
pop:	Stack $\rightarrow$ Stack
empty:	Stack $\rightarrow$ Boolean

Da keine Implementierung zu den Operationen angegeben ist, lässt sich eine Bedeutung nur durch passende Namen festlegen. Lässt man die Bezeichnungen für die Operationen und Trägermengen außer acht, könnten die Operationen auch eine Queue definieren und keinen Stack, da die Operationen der beide Datentypen die gleichen Definitions- und Wertebereiche haben.
Sich ausschließlich auf die Bedeutung der Namen zu verlassen kann gefährlich sein. Auch das Arbeiten mit unbekannten Datentypen ist so fast unmöglich.

3.2. Spezifikationen

3.2.1. Erweiterung um Spezifikationen

Um die Semantik der abstrakten Daten festzulegen, braucht man zusätzlich zu den Definitions- und Wertebereichen eine Spezifikation der abstrakten Funktionen.
Es gibt unterschiedliches Arten von Spezifikationen:
Bei den imperativen Spezifikationen bedient man sich der Mittel höherer Programmiersprachen. Applikative Spezifikationen haben mathematisch oder logische Konzepte als Grundlage.

3.2.2.Exemplarisch imperative Spezifikation

Bei der exemplarisch imperativen Spezifikationen werden die Datenelemente wie in Programmiersprachen dargestellt. Am Beispiel des Stack:

Stack als Array

Operationen werden als Prozeduren angesehen:

new ist die Prozedur, die einen leeren Stack erzeugt
push ist die Prozedur, die dem Stack ein Element hinzufügt
top ist die Prozedur, die das oberste Element das Stack ausliest
pop ist die Prozedur, die das oberste Element aus dem Stack entfernt
empty ist die Prozedur, die prüft ob der Stack leer ist

Der Nachteil dieser Spezifikation ist, das sie nicht abstrakt genug ist, sondern eher einer Implementierung entspricht.

3.2.3.Axiomatisch imperative Spezifikation

Bei der axiomatisch imperativen Spezifikation wird eine Programmlogik benutzt, die aus drei Teilen besteht, mit der Notation {x}S{y}.
Der erste und dritte Teil sind Formeln eines Logikkalküls, die die Vor- und Nachbedingungen der Operation angeben. Der zweite Teil ist der Programmteil.
Die Operationen werden also durch ihre Vor- und Nachbedingungen axiomatisiert.
Betrachten wir das Beispiel Stack:

{true} s = new() {empty(s) = true}
{true} s = push(t, x) {empty(s) = false, top(s) = x}
{empty(s) = false} x = top(s) {true}
{empty(t) = false} s = pop(t) {true}

Der Vorteil dieser Methode ist ein guter Abstraktionsgrad, der die Anforderung der Spezifikation erfüllt.
Die axiomatisch imperativen Methoden sind gut geeignet für die Beschreibung von Speichertransformationen in Datenbanken und für die Semantikbeschreibung imperativer Programmiersprachen.
Sie können auch zur Programmverifikation verwendet werden.

3.2.4.Exemplarisch applikative Spezifikation

Bei der exemplarischen applikativen Spezifikationsmethode werden Konzepte wie in der denotationalen Semantik angewandt. Es wird ein mathematisches Modell für die Operationen und Datenelemente angegeben.
Am Beispiel des Stack:

$$Stack = Item^* = \{ (i_1, i_2, ..., i_n) \mid n \in Nz, i \in Item \} \cup \{()\}$$

$$push((i_1, i_2, ..., i_n), x) = (x, i_1, i_2, ..., i_n)$$

$$pop(i_1, i_2, ..., i_n) = \begin{cases} (i_2, ..., i_n), \text{ falls } n>0 \\ \text{undefiniert, falls } n=0 \end{cases}$$

$$top(i_1, i_2, ..., i_n) = \begin{cases} i_1, \text{ falls } n>0 \\ \text{undefiniert, falls } n=0 \end{cases}$$

$$empty(i_1, i_2, ..., i_n) = \begin{cases} \text{false, falls } n>0 \\ \text{true, falls } n=0 \end{cases}$$

Die exemplarische applikative bietet einen guten Abstraktionsgrad und eine gute Verständlichkeit, ist aber für die Beschreibung von polymorphen ADTs ungeeignet.

3.2.5.Axiomatisch applikative Spezifikation

Bei der axiomatisch applikativen Spezifikation werden die Eigenschaften der abstrakten Datentypen mittels einer geeigneten logischen Sprache angegeben.
Am Beispiel des Stack:

$new() \in Stack$
$\forall s \forall i \ (s \in Stack \wedge i \in Item \rightarrow push(s, i) \in Stack \}$
$\forall s \ (s \in Stack \wedge s \neq new() \rightarrow pop(s) \in Stack \}$
$\forall s \ (s \in Stack \wedge s \neq new() \rightarrow top(s) \in Item \}$
$\forall s \ (s \in Stack \rightarrow empty(s) \in \{false, true\} \}$
$\forall s \forall i \ (s \in Stack \wedge i \in Item \rightarrow push(s, i) \neq new() \}$
$\forall s \forall i \forall j \ (s \in Stack \wedge i, j \in Item \rightarrow (push(s, i) = push(s ,i) \rightarrow i = j) \}$
$\forall s \forall t \forall i \ (s, t \in Stack \wedge i \in Item \rightarrow (push(s ,i) = push(t, i) \rightarrow s = t) \}$
$\forall s \forall i \ (s \in Stack \wedge i \in Item \rightarrow top(push(s ,i)) = x \wedge pop(push(s, i)) = s \}$
$\forall s \forall i \ (s \in Stack \wedge i \in Item \rightarrow empty(push(s, i)) = false \}$
$empty(new()) = true$

Die axiomatisch applikative Methode bietet einen sehr hohen Abstraktionsgrad und erlaubt es dem Spezifizierer genau die gewünschten Eigenschaften zu beschreiben. Der Nachteil an dieser Spezifikationsart ist, dass die Prädikatenlogik in ihrer Allgemeinheit zu mächtig ist. Widerspruchsfreiheit und Vollständigkeit sind nur schwer oder gar nicht entscheidbar. Deshalb ist es sinnvoll die Axiome auf Termgleichungen einzuschränken.
Am Beispiel des Stack:

empty(new()) = true
empty(push(s,x)) = false
top(new()) = undefiniert
top(push(s,x)) = x
pop(new()) = undefiniert
pop(push(s,x)) = s

Für diese algebraische Gleichungsspezifikation existieren ausgearbeitete Theorien und für die Frage der Widerspruchsfreiheit existieren Regeln.
Ein weiterer wichtiger Vorteil der axiomatischen Definition ist, dass die Spezifikation bei Erweiterung der Komplexität nicht so schnell anwächst wie bei anderen Spezifikationsarten, die eine erhebliche Überspezifikation erfordern.

3.3. Abstrahierende Funktion

Eine Instanzzierung eines abstrakten Datentyps besteht aus einer Interpretation oder Implementierung der Operationen des Datentyps, so das diese ein Modell der Spezifikationen ist, und aus einer abstrahierenden Funktion Φ, die die Terme des Modells auf die abstrakten Daten abbildet. Die abstrahierende Funktion hat keine eindeutige inverse Funktion, da es zu jedem abstrakten Datentyp mehrere Instanzierungen geben kann.
Ein Beispiel zum Stack:

x = new()	x=new()
x = push(x,A)	x=push(x,A)
x = push(x,B)	x=push(x,C)
	x=pop(x)
	x=push(x,B)

oder die isomorphe Darstellung dazu:

push(push(new(),A),B)	push(pop(push(push(new(),A),C)),B)

Die beiden Implementierungen sind verschieden, beschreiben aber den gleichen abstrakten Wert - nämlich einen Stack, der A und B enthält. Die abstrahierende Funktion liefert also für beide den gleichen Wert.

4. Probleme

Die hier vorgestellte Form von abstrakten Datentypen hat einige Schwächen.
Operationen die auf abstrakten Datentypen arbeiten, müssen beliebig viele Eingabe- und genau einen Ausgabewert haben. Häufig gibt es aber mehrere Rückgabewerte, zum Beispiel durch Übergabe von Pointer, oder gar keinen Rückgabewert.
Es gibt keine Möglichkeit Performance-Constraints zu spezifizieren.

Der Übergang von algebraischer Spezifikation zum Programmcode ist schwieriger als der, von einer operationaler Spezifikation.
Grenzbedingungen können leicht übersehen werden und bei der Erstellung der Spezifikation können Inkonsistenzen oder Unvollständigkeit entstehen.
Betrachtet man zum Beispiel die Spezifikation des abstrakten Datentyps Menge:

Operationen:

new: → set
insert: set x nat → set
del: set x nat → set

Spezifikation:

insert(insert(s,n),n) = insert(s,n)
insert(insert(s,n),m) = insert(insert(s,m),n)

del(new(),n) = new()
del(insert(s,n),n) = s

Obwohl die Spezifikation auf den ersten Blick konsistent erscheint, ist sie es nicht.
Alle Elemente von set fallen mit new() zusammen:

new() = del(insert(new(),n),n)
= del(insert(insert(new(),n),n),n)
= insert(new(),n)

Ein weiteres Problem ist, dass der Designer der ADTs leicht vermeintlich unwichtige Funktionen übersehen kann und diese nicht spezifiziert. Fällt das Fehlen erst bei der Anwendung des Datentyps auf, wird der Benutzer das Interface umgehen, direkt auf den Datenstruktur zugreifen und so das Prinzip der ADTs aushebeln.
Die Erzeugung von algebraische Spezifikation und die Vermeidung dieser Fehler erfordern viel Erfahrung.

5. Zusammenfassung

Heute ist die Datenabstraktion selbstverständlich, sie ist ausführlich erforscht und wird von viele Programmiersprachen unterstützt.
Die Forschung an abstrakten Datenstrukturen führte zu einem tieferen Verständnis von Datenstrukturen im allgemeinen. Das Betrachten eines Programms als eine Menge von Datenstrukturen anstelle einer Menge von Operationen führte zu einem neuen Organisationsprinzip. Durch Abstraktion ist Programmieren nicht mehr das Schreiben von Quellkode, vielmehr ist es das Anordnen von Modulen.
Die schrittweise Verfeinerung dieser Module führte zu großen Komponenten und damit zu einer flachen Programmstruktur. Diese Struktur hatte zur Folge, dass die Module wenig spezialisiert und so sehr gut wiederverwendbar sind.
Die ausgedehnte Benutzung von ADTs führt zu besseren Programmstrukturen, die qualitativ besserer und billigerer Programme ermöglicht.
Algebraische Spezifikation sind weitestgehend implementierungsunabhängig und verzögern den Zeitpunkt der Implementierung. Diese Vermeidung einer frühzeitigen Entscheidung über Speicherstrukturen und Zugriffsarten ist von Vorteil, weil der Designer der ADTs häufig nur wenig Einblicke in die relative Häufigkeit der Benutzung der verschiedenen Operationen hat.
Die algebraische Spezifikation abstrakter Datentypen kann auch benutzt werden um Programmeigenschaften zu beweisen.

5. Quellen

John V. Guttag “Software pioneers: Abstract Data Types”

Prof. Hartwig ”Syntax, Semantik und Spezifikationen”

Prof. Gruhn “Ausgewählte Kapitel der Software-Technologie: Spezifikationen”

John V. Guttags Homepage am MIT
http://csbi.mit.edu/faculty/steering/Members/JohnGuttag
http://nms.lcs.mit.edu/~guttag/

Bild Guttags
http://www.uni-koblenz.de/~ghkamp/sdm2001/guttag.html

Oracle PL/SQL Programming, 2nd Edition
http://www.hk8.org/old_web/oracle/prog2/