

Abstrakte Datentypen

Algebraische Spezifikation und Anwendung
beim Entwurf komplexer Programme

Copyright 2002 © BerniSoft Corp.

Übersicht

1. Einführung
2. Software-Entwicklung
3. Abstrakte Datentypen
4. Erweitertes Beispiel
5. Abstrakte Datentypen und Software-Entwicklung
6. Fazit

1. Einführung

In den 70-er Jahren kristallisierten sich drei
Anforderungen an Software heraus:

- Korrektheit
- Wartbarkeit
- Wiederverwendbarkeit

1.1 Vorhandene Konzepte

In diesem Zuge wurden neue Konzepte und
Programmiersprachen entwickelt:

- Information Hiding (Parnas)
- Simula 67 (Dahl, Nygaard)
- Strukturierte Programmierung (Hoare)

2 Software-Entwicklung

Grundsätzliches Problem:
Der Mensch kann nur einen geringen Grad an Komplexität überschauen.

Lösung: Herunterbrechen der Komplexität durch

Abstraktion und Strukturierung

2.1 Strukturierung

Strukturierung:

- Aufteilen in mehrere Unterprogramme/Prozeduren
- Behandlung dieser als Black Boxes

Defizit:

- Beschreibung abstrakter Objekte kaum möglich
- Schnittstellen schlecht beschreibbar
- Programmablauf im Mittelpunkt

2.2 Abstraktion

Abstraktion:

- Vergessen von Details von Datenobjekten
- Beschränkung auf wesentliche Eigenschaften und Beziehungen

Mittel:

- Repräsentationsunabhängige Definition von Attributen und Methoden

3 Abstrakte Datentypen

Abstrakte Datentypen:

Sind Objekte, verbunden mit den Operationen, die auf ihnen arbeiten. Dabei werden die Operationen als eine abstrakte Schnittstelle repräsentationsunabhängig beschrieben.

3.1 Spezifikation von Datenobjekten

Klassifikation der Attribute:

- Repräsentationsabhängig
Repräsentation der Daten, Implementation der Operationen
- Repräsentationsunabhängig
Namen und abstrakte Bedeutung der Operationen

Allgemeine Spezifikation der Operationen

Beschreibung abstrakter Datentypen:

- Syntax
- Semantik:
 - Operational
 - Axiomatisch

Algebraische Spezifikation

In Anlehnung an heterogene Algebren:

- Syntax: Menge von Namen, Wertebereichen
- Semantik:
Menge von Relationen der Operationen untereinander in Form von Axiomen

3.2 Beispiel Warteschlange

Syntaktische Spezifikation:

NEW: \rightarrow Queue
ADD: Queue x Item \rightarrow Queue
FRONT: Queue \rightarrow Item
REMOVE: Queue \rightarrow Queue
IS_EMPTY: Queue \rightarrow Boolean

Axiome:

2. $IS_EMPTY(NEW) = true$
3. $IS_EMPTY(ADD(q,i)) = false$
4. $FRONT(NEW) = error$
5. $FRONT(ADD(q,i)) = if\ IS_EMPTY(q)\ then\ i\ else\ FRONT(q)$
6. $REMOVE(NEW) = error$
7. $REMOVE(ADD(q,i)) = if\ IS_EMPTY(q)\ then\ NEW\ else\ ADD(REMOVE(q),i)$

Achtung

Besonderheiten:

- Item nicht spezifiziert
- Boolean als bekannt vorausgesetzt
- Keine Implementationsvorgaben

Gefahren:

- Widersprüche
- Unzureichende Spezifikation (nicht alle Fälle berücksichtigt)

Lösung:

Heuristiken zur Erstellung ausreichender und widerspruchsfreier Spezifikationen

4 Erweitertes Beispiel

Das folgende Beispiel soll unter anderem einen Einblick in den Zusammenhang von Abstraktion und Repräsentation geben.

Im folgenden wird der Datentyp *SymbolTable* modelliert. Er dient der Auflistung/ Verwaltung der Variablenbezeichner einer blockorientierten Programmiersprache in eine für Compiler ausgelegte Struktur.

SymbolTable

Syntaktische Spezifikation:

- INIT: $\rightarrow SymbolTable$
ENTERBLOCK: $SymbolTable \rightarrow SymbolTable$
LEAVEBLOCK: $SymbolTable \rightarrow SymbolTable$
ADD: $SymbolTable \times Identifier \times Attributelist \rightarrow SymbolTable$
IS_INBLOCK: $SymbolTable \times Identifier \rightarrow Boolean$
RETRIEVE: $SymbolTable \times Identifier \rightarrow Attributelist$

SymbolTable

Semantische Spezifikation:

2. LEAVEBLOCK(INIT) = error
3. LEAVEBLOCK(ENTERBLOCK(symtab)) = symtab
4. LEAVEBLOCK(ADD(symtab, id, attrs)) = LEAVEBLOCK(symtab)
5. IS_INBLOCK(INIT, id) = false
6. IS_INBLOCK(ENTERBLOCK(symtab), id) = false
7. IS_INBLOCK(ADD(symtab, id, attrs); idl) =
if IS_SAME(id, idl)
then true
else IS_INBLOCK(symtab, id)
8. RETRIEVE(INIT, id) = error
9. RETRIEVE(ENTERBLOCK(symtab), id) = RETRIEVE(symtab, id)
10. RETRIEVE(ADD(symtab, id, attrs), idl) =
if IS_SAME(id, idl)
then attrs
else RETRIEVE(symtab, idl)

Stack

Syntaktische Spezifikation:

- NEWSTACK: \rightarrow Stack
- PUSH: Stack x Array \rightarrow Stack
- POP: Stack \rightarrow Stack
- TOP: Stack \rightarrow Array
- IS_NEWSTACK: Stack \rightarrow Boolean
- REPLACE: Stack x Array \rightarrow Stack

Stack

Semantische Spezifikation:

2. IS_NEWSTACK(NEWSTACK) = true
3. IS_NEWSTACK(PUSH(stk, arr)) = false
4. POP(NEWSTACK) = error
5. POP(PUSH(stk, arr)) = stk
6. TOP(NEWSTACK) = error
7. TOP(PUSH(stk, arr)) = arr
8. REPLACE(stk, arr) = if IS_NEWSTACK(stk)
then error
else PUSH(POP(stk), arr)

Array

Syntaktische Spezifikation:

- EMPTY: \rightarrow Array
- ASSIGN: Array x Identifier x Attributelist \rightarrow Array
- READ: Array x Identifier \rightarrow Attributelist
- IS_UNDEFINED: Array x Identifier \rightarrow Boolean

Array

Semantische Spezifikation:

2. IS_UNDEFINED(EMPTY, id) = true
3. IS_UNDEFINED(ASSIGN(arr, id, attrs), idl) =
if IS_SAME(id, idl)
then false
else IS_UNDEFINED(arr, idl)
4. READ(EMPTY, id) = error
5. READ(ASSIGN(arr, id, attrs), idl) =
if IS_SAME(id, idl)
then attrs
else READ(arr, idl)

SymbolTable-Repräsentation

- INIT': \rightarrow Stack
- ENTERBLOCK': Stack \rightarrow Stack
- LEAVEBLOCK': Stack \rightarrow Stack
- ADD': Stack x Identifier x Attributelist \rightarrow Stack
- IS_INBLOCK': Stack x Identifier \rightarrow Boolean
- RETRIEVE': Stack x Identifier \rightarrow Attributelist

SymbolTable-Repräsentation

Implementierung:

- INIT' :: PUSH(NEWSTACK, EMPTY)
- ENTERBLOCK'(stk) :: PUSH(stk, EMPTY)
- LEAVEBLOCK'(stk) :: if IS_NEWSTACK(POP(stk))
then error
else POP(stk)
- ADD'(stk, id, attrs) :: REPLACE(stk, ASSIGN(TOP(stk), id, attrs))
- IS_INBLOCK'(stk, id) :: if IS_NEWSTACK(stk)
then false
else not IS_UNDEFINED(TOP(stk), id)
- RETRIEVE'(stk, id) :: if IS_NEWSTACK(stk)
then error
else if IS_UNDEFINED(TOP(stk), id)
then RETRIEVE'(POP(stk), id)
else READ(TOP(stk), id)

SymbolTable-Interpretation

Interpretationsfunktion I:

2. I(error) = error
3. I(NEWSTACK) = error
4. I(PUSH(stk, EMPTY)) = if IS_NEWSTACK(stk)
then INIT
else ENTERBLOCK(I(stk))
5. I(PUSH(stk, ASSIGN(arr, id, attrs))) =
ADD(I(PUSH(stk, arr)), id, attrs)

SymbolTable-Korrektheit

- Um zu überprüfen, ob die Repräsentation mit der abstrakten Definition übereinstimmt, müssen die Axiome nachgeprüft werden.
- Für Axiome 1 bis 8 ist dies ein mechanisches Unterfangen. Für Axiom 9 braucht man noch folgende Annahme:

$ADD'(symtab, id, attrs) \Rightarrow IS_NEWSTACK(symtab) = false$

SymbolTable-Änderung

Grundlegende Änderungen sind relativ einfach zu vollziehen.

Beispiel-Änderung:

- In einem Block sollen nur Variablen gültig sein, die :
 - Lokal definiert
 - In einer speziellen Liste aufgeführt sind
- Globale Variablen gibt es nicht mehr

SymbolTable-Änderung

Nur Axiome zu berücksichtigen, in denen ENTERBLOCK vorkommt.

3. $IS_INBLOCK(ENTERBLOCK(symtab, klist), id) = false$
4. $LEAVEBLOCK(ENTERBLOCK(symtab, klist)) = symtab$
5. $RETRIEVE(ENTERBLOCK(symtab, klist), id) =$
if $IS_IN(klist, id)$
then $RETRIEVE(symtab, id)$
else error

Knowlist

Syntax:

2. CREATE: \longrightarrow Knowlist
3. APPEND: Knowlist x Identifier \longrightarrow Knowlist
4. IS_IN: Knowlist x Identifier \longrightarrow Boolean

Semantik:

- $IS_IN(CREATE) = false$
- $IS_IN(APPEND(klist, id), idl) =$ if $IS_SAME(id, idl)$
then true
else $IS_IN(klist, idl)$

5 Abstrakte Datentypen und Software-Entwicklung

Abstrakte Datentypen haben großen Einfluss auf die Entwicklung der Softwaretechnik genommen:

- Hauptaugenmerk auf Datenobjekte
- Programme mathematisch handhabbar
- Hilfe bei der Erfüllung der anfangs genannten Kriterien:
 - Korrektheit
 - Wartbarkeit
 - Wiederverwendbarkeit

5.1 Korrektheit

Abstrakte Spezifikation:

In der Spezifikation sind alle wesentlichen Merkmale genau aufgeführt. Somit ist eine Überprüfung gewünschter Eigenschaften leichter zu führen.

Repräsentation:

Durch Überprüfen der Axiome kann die Übereinstimmung mit der Spezifikation nachgewiesen werden.

5.2 Wartbarkeit

Veränderung:

- Lokale Änderungen der Implementation haben keine globalen Auswirkungen. (Information Hiding)
- Schnell erkennbar, wo Änderungen anzusetzen sind

Fehlerlokalisierung:

Da die Daten nur mit den aufgeführten Operationen verändert werden können und diese Schnittstellen präzise beschrieben sind, ist ein Fehler schnell zu lokalisieren.

5.3 Wiederverwendbarkeit

Wegen ihrer Repräsentationsunabhängigkeit können abstrakte Datentypen in verschiedenen Programmen problemlos wieder verwendet werden.

5.4 Auswirkungen auf Entwicklungs-Prozess

- Hilfe bei Behandlung komplexer Probleme
- Spezifikation von Schnittstellen zwischen Modulen
- Entscheidung für eine Repräsentation hinausgezögert
- Codierung zu einem späten Zeitpunkt
- Codierung/Testen nur unter Verwendung der Spezifikationen fördert Prinzip Information Hiding

5.5 Nachteile

- Wichtige Attribute leicht wegzuabstrahieren
- Effizienzkriterien nicht berücksichtigt
- Codierung nicht gleich ableitbar

6 Fazit

- Die Algebraische Datenspezifikation macht Software für die Mathematik handhabbar. Damit stehen mathematische Methoden zur Analyse bereit.
- Außerdem bietet ihre Repräsentationsunabhängigkeit Vorteile in Bezug auf Wiederverwendbarkeit und den Entwicklungs-Prozess