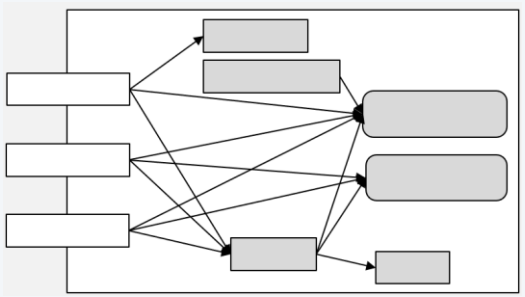
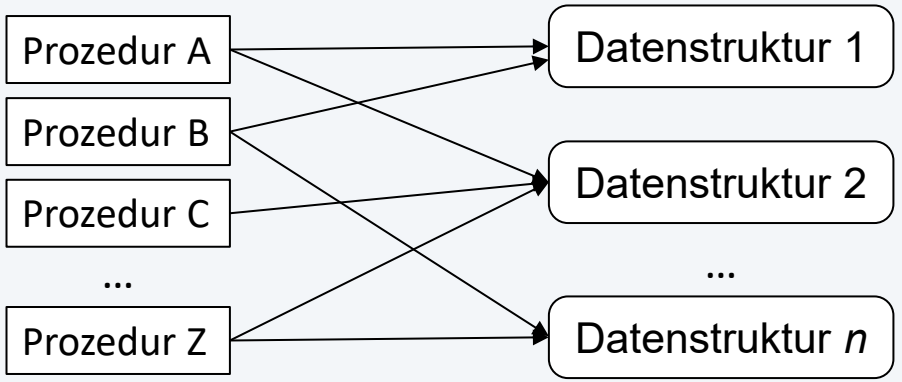
Studiengang Software Engineering
Fakultät für Informatik, Kommunikation, Medien
Campus Hagenberg

12 Datenkapseln und Module



12.1 Unbeschränkter Zugriff auf Datenobjekte

Zerlegung von Algorithmen nach funktionalen Gesichtspunkten führt häufig zu unbeschränkten Datenobjektzugriffen



Konsequenz

- Alle Algorithmen (A bis Z) müssen Details der Datenobjekte (1 bis n) kennen
- Beim Ändern einer Datenstruktur ist es schwierig festzustellen, wo sie überall verwendet wird
- Unabsichtliche Verwendung (Zerstörung) der Daten möglich

Beispiel 1

Beispiel: Unbeschränkter Zugriff auf Stack (Realisierung mittels Feld)

Deklarationen

```
const
  size = ...
var
  stack: array [1:size] of int
  top: int
```

Initialisierung

```
top := 0
```

Operationen Push und Pop

```
top := top + 1
stack[top] := 42
```

```
x := stack[top]
top := top – 1
```

ohne `top:=top + 1` wird letztes Element überschrieben

Fehlerbehandlung (z.B. Stack leer oder voll) ist bei jedem Zugriff auf Datenobjekt `stack` erforderlich.

Beispiel 2

Beispiel: Unbeschränkter Zugriff auf Bankkonto-Datenobjekt

Deklarationen

```
const
  size = …
type
  Timestamp = …
  Change = compound
    time: Timestamp
    purpose: string
    amount: real
  end -- compound
  Account = compound
    balance: real
    initialCredit: real
    nChanges: int
    changes: array [1:size] of Change
  end -- compound
var
  a: Account
```

Anfangsstand bei Eröffnung,
Kontobewegungen, aktueller Saldo

Kontodaten müssen konsistent sein
balance = initialCredit + changes[1:nChanges].amount

Anzahl der Kontobewegungen muss mit
tatsächlich vorhandenen Änderungen
übereinstimmen

Beispiel 2

Verwendung

```
-- init account
a.initialCredit := 0.0
a.balance := 0.0
a.nChanges := 0
```

```
-- new change
a.nChanges := a.nChanges + 1
a.changes[a.nChanges].time := ...
a.changes[a.nChanges].purpose := "Von Oma"
a.changes[a.nChanges].amount := 1000.0
a.balance := a.balance + 1000.0
```

ohne Anweisung würde letzter Eintrag überschrieben

ohne Anweisung würde Kontostand nicht stimmen

Probleme

- Gefahr von inkonsistenten Zuständen
- Undurchsichtiger Datenfluss
- Hoher Änderungsaufwand, z.B. Stack-Repräsentation Feld durch Liste ersetzen

Beispiel 3

Einfaches Zeichenprogramm (ca. 1984)

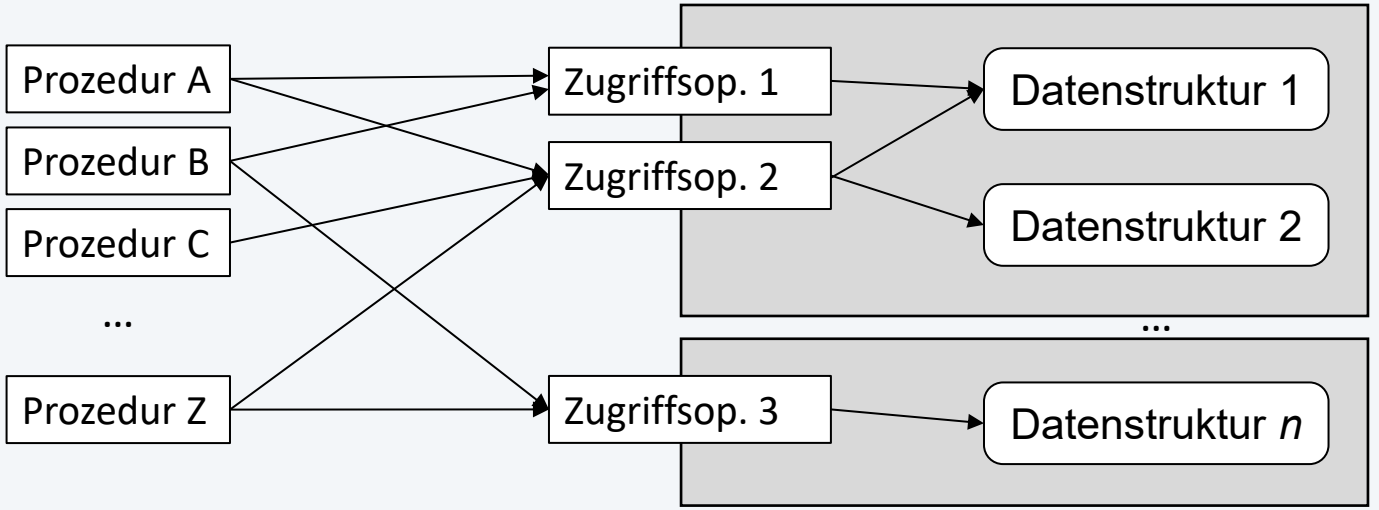
- ca. 5.800 Zeilen Pascal-Code
- ca. 3.800 Zeilen Assembler-Code

Pascal-Code

- eine Datei
- ca. 150 globale Variablen
- ca. 150 Prozeduren u. Funktionen

https://computerhistory.org/blog/macpaint-and-quickdraw-source-code/

12.2 Beschränkter Zugriff auf Datenobjekte



Konsequenz

- Nur noch wenige Algorithmen (Zugriffsoperationen) greifen auf die Datenobjekte zu
- Datenobjekte und Zugriffsoperationen bilden abgeschlossene Einheit
- Schutz vor Zerstörung des Inhalts der Datenobjekte/Datenstruktur (DS)
- Implementierung der gekapselten Datenstruktur kann geändert werden
- Indirekter Zugriff kostet Zeit

Geheimnisprinzip (*Principle of Information Hiding*)

nach David Parnas, 1972

Datenabstraktion

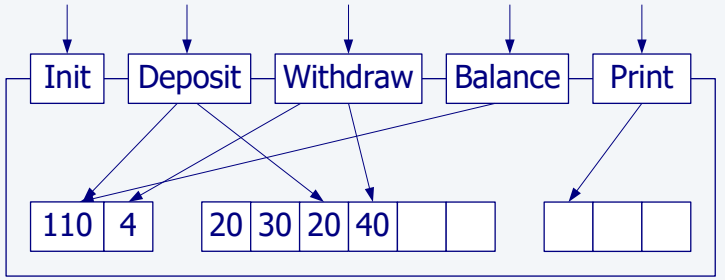
- Datenobjekte, d.h. ihr konkreter Aufbau, dürfen nur denjenigen Algorithmen bekannt sein, die sie zu ihrer Implementierung brauchen
- Unterscheidung zwischen Implementierungsaspekt und Anwendungsaspekt

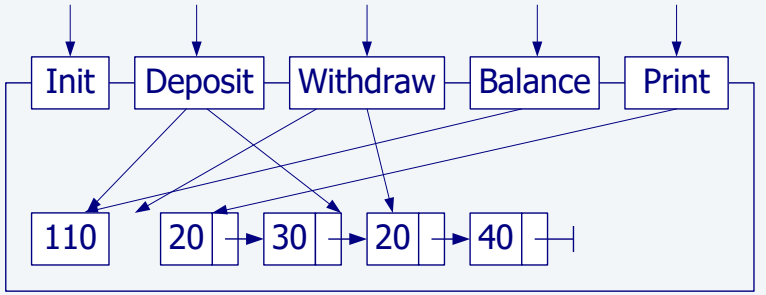
Datensicherheit und Änderungsfreundlichkeit

- Die Datenobjekte können nicht direkt, sondern nur mittels zugelassener Operationen (Zugriffsoperationen) manipuliert werden
- Die Änderung der konkreten Implementierung von Datenobjekten erfordert keine Änderungen in den sie verwendenden Algorithmen (bei gleichbleibender Schnittstelle der Zugriffsoperationen)

Datenkapsel

- Die Datenobjekte selbst bleiben der "Außenwelt" verborgen (Daten sind gekapselt – *information hiding*)
- Zugriffsoperationen sind einzige Möglichkeit zur Manipulation des Inhalts und zum Zugriff auf den Inhalt der Datenobjekte





- Benutzer-Algorithmus kennt "Inhalt" der Datenkapsel nicht

```
Init()
Deposit(↓"Spende von Oma" ↓1000.0)
Withdraw(↓"iPhone" ↓199.0)
if Balance() > 499.0 then
  Withdraw(↓"Mac Mini" ↓499.0)
end -- if
```

12.3 Module zur Realisierung von Datenkapseln

Datenkapselungsprinzip:

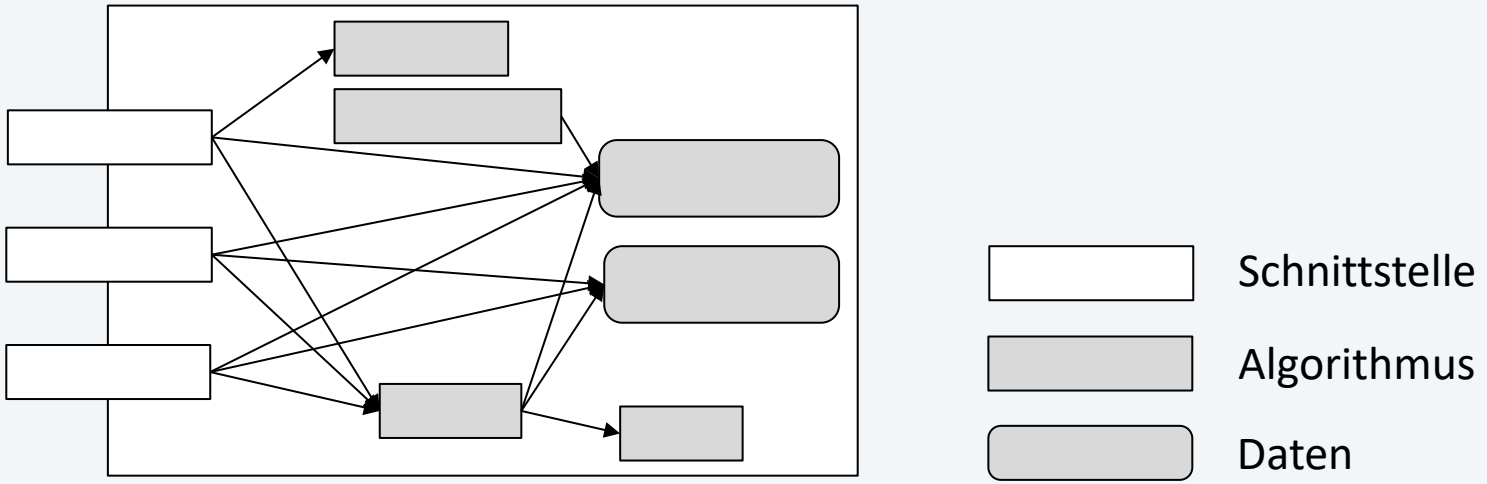
- Datenkapsel entspricht Sammlung von Algorithmen (inkl. Zugriffsoperationen) und ihnen gemeinsam zugänglichen Datenobjekten
- die Datenobjekte sind statische Variablen, auf die alle Algorithmen (inkl. Zugriffsoperationen) zugreifen können (lokale globale Variable)

Realisierungsform:

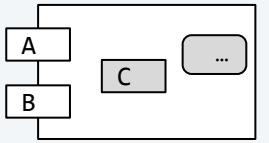
- Bildung einer Programmkomponente/Übersetzungseinheit
- Verschiedene Konzepte Modul (z.B. Modula-2), Package (z.B. Java), Unit (z.B. Free Pascal)
- Pascal-Dialekte (z.B. Free Pascal) bieten Möglichkeit, Programmsysteme auf mehrere Übersetzungseinheiten aufzuteilen
- Systemzerlegung wird Modularisierung genannt

Das Modulkonstrukt (Wdh. Kapitel 5)

- Ein Modul ist eine Sammlung von Datenobjekten und Algorithmen.
- Er kommuniziert mit der Außenwelt nur über eine eindeutig definierte Schnittstelle.
- Die Benutzung eines Moduls setzt keine Kenntnisse seines inneren Aufbaus voraus.
- Die Implementierung eines Moduls setzt keine Kenntnisse über seine Benutzung voraus.
- Die Korrektheit ist ohne Kenntnis der Einbettung ins Gesamtsystem prüfbar.



Aufbau



Module bestehen aus zwei Teilen: *Schnittstelle* und *Implementierung*

```
interface of M
  const …
  type …
  A(…)
  B(…)
end M
```

Schnittstelle (*interface*)

- Konstanten
- Datentypen
- Schnittstellen der Zugriffsoperationen (Signatur)

```
implementation of M
  var …
  A(…) begin …C()… end A

  B(…) begin …C()… end B

  C(…) begin … end C

init
 …
end M
```

Implementierung (*implementation*)

- benötigte Datenobjekte/-strukturen
- Implementierung der Zugriffsoperationen
- weitere Algorithmen, die nicht direkt von außen benutzt werden können
- ggf. Initialisierung

Beispiel: Modul für Integer-Stack

```
interface of IntStack
  InitStack()
  Push(↓e: int)
  Pop(↑e: int)
  IsEmpty(): bool
end IntStack
```

```
implementation of IntStack
  const
    size = 100
  var
    stack: array[1:size] of int
    top: int

  InitStack()
  begin
    top := 0
  end InitStack

  IsEmpty(): bool
  begin
    return top = 0
  end IsEmpty
```

```
…
  Pop(↑e: int)
  begin
    e := stack[top]
    top := top - 1
  end Pop

  Push(↓e: int)
  begin
    top := top + 1
    stack[top] := e
  end Push

init
  InitStack()
end IntStack
```

Beispiel: Modul für Integer-Stack

Verwendung Modul *IntStack*

```
program P
  import IntStack
  var
    i: int
begin
  Push(↓13)  Push(↓21)
  Push(↓34)  Push(↓55)
  while not IsEmpty() do
    Pop(↑i)
    Write(↓i ↓" ")
  end -- while
  top := 0
  Write(↓stack[4])
end P
```

```
interface of IntStack
  InitStack()
  Push(↓e: int)
  Pop(↑e: int)
  IsEmpty(): bool
end IntStack

implementation of IntStack
  const size = 100
  var stack: array[1:size] of int
  var top: int
  …
```

nicht möglich (Geheimnisprinzip!)

Verwendung von Modulen

Häufige Unterscheidung

- funktionsorientierte Module
  ... fassen verwandte Funktionen zusammen
  (z.B. Mathematik-Funktionen, grafische Ausgabe)

- aufgabenorientierte Module
  ... lösen eine Teilaufgabe eines größeren Problems
  (z.B. Mailverarbeitung)

- datenorientierte Module.
  ... kapseln Daten von der Umgebung und stellen Zugriffsfunktionen zur Verfügung (z.B. Stack, Konto, Hashtabelle, Stichwortverzeichnis)

Richtlinien zur Modulbildung

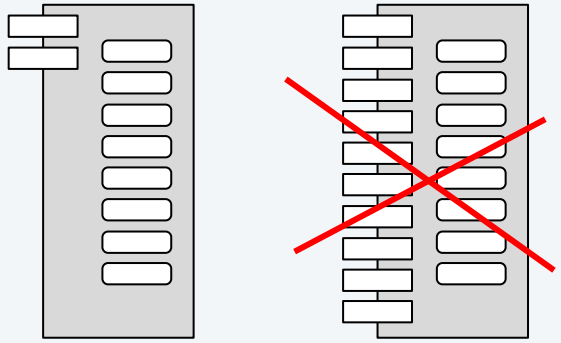
Sicherstellen der Modulgeschlossenheit

- möglichst starke Bindung zwischen Operationen und Daten,
- es soll keine Operationen und Daten geben, die in keinem Zusammenhang zueinander stehen (gilt für aufgaben- und datenorientierte Module)

Sicherstellung einer geringen Modulkopplung (MK)

- Modulkopplung … wie stark sind versch. Module untereinander verbunden
- geringe MK: Maß für die Unabhängigkeit der Module
- hohe MK: Hinweis darauf, das logisch zusammengehörende Operationen über mehrere Module verteilt sind
- globale (exportierte) Datenobjekte führen oft zu hoher MK

Richtlinien zur Modulbildung



Sicherstellung beherrschbarer Schnittstellen

- kleine Schnittstelle -> kleine Modulkopplung
- keine globale Datenobjekte

Sicherstellung der Testbarkeit

Sicherstellung von Interferenzfreiheit

- sie ist nicht gegeben, wenn ein Modul mehrere Aufgaben bearbeitet oder eine Aufgabe auf mehrere Module verteilt ist

12.4 Abstrakte Datenstruktur (ADS)

- Menge von Datenobjekten die bestimmte Abstraktion darstellen (z.B. Stack, Baum, Konto, Autorenverzeichnis, …)
- Beschreibung, welche Datenobjekte verwaltet werden (z.B. Konto)
- Beschreibung der erlaubten Operationen
- Keine Beschreibung, wie Datenstruktur implementiert ist

Beispiel: Abstrakte Datenstruktur für Bankkonto

- Konto mit Anfangsguthaben, Liste von Kontobewegungen und Saldo
- Operationen
  - Initialisieren
  - Einzahlen und Abheben
  - Saldo abfragen
  - Kontoauszug drucken

Zugriffsoperationen

Implementierung der Operationen durch Prozeduren/Funktionen mit Parametern

```
interface of Account
  -- Initialisiere Konto
  InitAccount(↓value: real)
  -- Zugriffsoperation Einzahlung
  Deposit(↓purpose: string ↓amount: real)
  -- Zugriffsoperation Auszahlung
  Withdraw(↓purpose: string ↓amount: real)
  -- Zugriffsoperation Saldo
  Balance(): real
  -- Zugriffsoperation Kontoübersicht
  PrintAccount()
end Account
```

Gesamtheit aller Zugriffsoperationen mit Parametern wird *Signatur* einer ADS genannt

Dokumentation der Zugriffsoperationen

Zugriffsoperationen müssen dokumentiert werden, um ihre Semantik und Benutzung zu erläutern

Schnittstelle	`Withdraw(↓purpose: string ↓amount: real)`
Wirkung	Fügt neue Kontobewegung mit angegebenen Zweck, Betrag und der aktuellen Zeit hinzu und aktualisiert den Saldo.
Vorbedingungen	`InitAccount()` wurde aufgerufen `purpose` ist nicht leer `amount` > 0 `Balance()` ist größer als `amount`
Nachbedingung	`Balance()` liefert den um `amount` verringerten Betrag
Fehlerverhalten	Wenn eine Vorbedingung verletzt ist, hat diese Operation keine Wirkung.

- Keine Angabe über Aufbau der Datenstruktur
- Keine Angabe über Implementierung der Zugriffsoperationen

Dokumentation der Zugriffsoperationen

Beispiele

pop

Schnittstelle

```
public E pop()
```

Wirkung

Removes the object at the top of this stack and returns that object as the value of this function.

Returns:

The object at the top of this stack (the last item of the `Vector` object).

Throws:

`EmptyStackException` - if this stack is empty.

Vorbedingung und Fehlerbehandlung

get

```
E get(int index)
```

Returns the element at the specified position in this list.

Parameters:

`index - index of the element to return`

Returns:

`the element at the specified position in this list`

Throws:

`IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())`

12.5 Abstrakte Datentypen (ADT)

- Abstrakte Datentypen definieren Menge gleicher ADS
- ADT können wie echte Datentypen verwendet werden

Beispiel: Mehrere Bankkonten

```
var
  savingsAccount: Account
  account: Account
```

```
InitAccount(↓↑account ↓1000)
Withdraw(↓↑account ↓"iPhone" ↓449)

if Balance(↓↑account) > 200 then
  Withdraw(↓↑account ↓"Sparbuch" ↓200)
  InitAccount(↓↑savingsAccount ↓200)
else
  InitAccount(↓↑savingsAccount ↓0)
end -- if
```

jede Zugriffsoperation wird auf eine best. DS ausgeübt

Datenstrukturen (Aufbau) bleiben geheim!

Schnittstelle

- Datenobjekte bleiben wieder vor der Außenwelt verborgen
- Manipulation der (abstrakten) Datenobjekte wieder nur über Zugriffsoperationen möglich
- Jede Zugriffsoperation hat einen zusätzlichen Parameter: das zu manipulierende Exemplar des ADT

Beispiel: Abstrakter Datentyp `IntStack`:

```
interface of IntStack
  type
    Stack

  InitStack(↑s: Stack)
  Push(↓↑s: Stack ↓e: int)
  Pop(↓↑s: Stack ↑e: int)
  IsEmpty(↓s: Stack): bool
  DisposeStack(↓↑s: Stack)
end IntStack
```

Name muss in Schnittstelle bekannt sein – Typ soll aber geheim bleiben

Änderung im Vergleich zur Schnittstelle von abstrakten Datenstrukturen

Aufräumarbeiten z.B. wenn als Liste realisiert

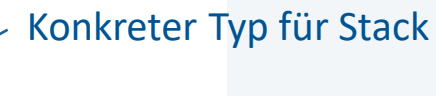
Implementierung

legt konkreten Typ der gekapselten Datenobjekte fest

```
implementation of IntStack
  type
    Stack = compound
      stack: array[1:100] of int
      top: int
    end -- compound

  InitStack(↑s: Stack)
  begin
    s.top := 0
  end InitStack

  Push(↓↑s: Stack ↓e: int)
  begin
    s.top := s.top + 1
    s.stack[s.top] := e
  end Push
  …
end IntStack
```



Vorteile abstrakter Datentypen

Können verwendet werden, um flexibel beliebig viele Exemplare komplexer Datenstrukturen (z.B. Vektoren, Mengen, Verzeichnisse wie Hashtabellen) zu bilden und zu verwenden,

- deren konkreter Aufbau und deren Komplexität verborgen bleibt (Abstraktion),
- deren konkrete Implementierung ohne Wissen der Klienten geändert/ausgetauscht werden kann (Änderungsfreundlichkeit),
- deren missbräuchliche Verwendung verhindert wird (Sicherheit).

Zusammenfassung

- Eine abstrakte Datenstruktur (*abstract data structure*), kurz ADS, ist ein algorithmischer Baustein, der über eine spezielle Schnittstelle eine Menge von (abstrakten) Zugriffsoperationen zur Verfügung stellt, mit denen eine Menge gekapselter, d.h. abstrakter Datenobjekte (deren konkrete Realisierung dem Benutzer einer Zugriffsoperation verborgen bleibt) einzeln oder als Ganzes manipuliert werden kann.
- Ein abstrakter Datentyp (*abstract data type*), kurz ADT, definiert eine Menge von Datenobjekten, die alle dieselbe abstrakte Datenstruktur haben und auf die nur mittels (abstrakter) Zugriffsoperationen zugegriffen werden kann.
- Unter einem Modul (*module*) im Sinne der Informatik verstehen wir eine Sammlung von Algorithmen und Datenobjekten zur Bearbeitung einer in sich abgeschlossenen Aufgabe.

Ausblick auf objektorientierte Programmierung

Eines der wichtigsten Konzepte der objektorientierten Programmierung sind erweiterbare abstrakte Datentypen (z.B. in Form von Klassen)

Implementation:

```
const
  size = 100
type
  Stack = class
    data: array [1:size] of int
    top: int
    Init()
    Push(↓e: int)
    Pop(↑e: int)
    IsEmpty(): bool
  end -- Stack

 Stack.Push(↓e: int)
 begin
   top := top + 1
   data[top] := e
 end Stack.Push
```

Verwendung:

```
var
  a, b: Stack
```

```
a.Push(↓42)
b.Push(↓12)
if a.IsEmpty() then
   …
end -- if
```