Software Architecture

Abstract Data Types

Mathematical description

- An ADT is a mathematical specification
- Describes the properties and the behavior of instances of this type
- Doesn‘t describe implementation details (therefore it‘s *abstract*)
- An example: STACK

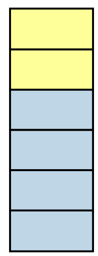
Stack

- LIFO (last in, first-out) Queue
- Operations:
  - put: Put something onto the STACK (Command)
  - remove: Remove the top element of the STACK (Command)
  - item: Return the value of the top item (Query)
  - empty: Is the STACK empty? (Query)

Abstract data types

One data type – many implementations. E.g. for STACK:

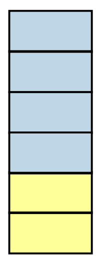
Variant 1: array up



"Push" *x* on stack *representation*:

representation [*count*] := *x*
count := *count* + 1

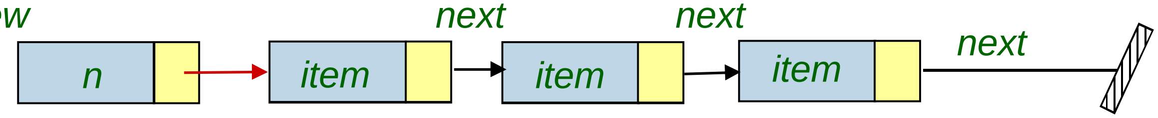
Variant 2: array down



"Push" *x* on stack *representation*:

representation [*free*] := *x*
free := *free* - 1

Variant 3: linked list



"Push" operation:

new (*n*)
n.*item* := *x*
n.*next* := *first*
first := *new*

Formal description of a STACK

Types:

The type(s) that are described by the ADT.

Functions:

The functions that can be applied to the ADT.

Preconditions:

Preconditions that need to be fulfilled to apply a feature.

Axioms:

Axioms that the ADT fulfills.

Abstract data types

Creators:

Create a new instance of an ADT.

{OTHERS} $\longrightarrow$ ADT

Queries:

Functions that have a return value and do not change the instance.

ADT {x OTHERS} $\longrightarrow$ OTHERS

Commands:

Functions without return value that change the instance.

ADT {x OTHERS} $\longrightarrow$ ADT

Partial function $\nrightarrow$

There are cases where no valid return value can be given for a function e.g.. division by 0

Abstract data types

Types

STACK [G]

-- G: Formal generic parameter

Functions

put: STACK [G] $\times$ G $\rightarrow$ STACK [G]

remove: STACK [G] $\nrightarrow$ STACK [G]

item: STACK [G] $\nrightarrow$ G

empty: STACK [G] $\rightarrow$ BOOLEAN

new: STACK [G]

Abstract data types

Preconditions

remove (s: STACK [G]) **require** not empty (s)

item (s: STACK [G]) **require** not empty (s)

Axioms For all x: G, s: STACK [G]

1. item (put (s, x)) = x
2. remove (put (s, x)) = s
3. empty (new)
4. not empty (put (s, x))

Well-formed and correct terms

Well-formed: all functions get a right number of arguments of right types

Correct: preconditions of all functions are satisfied

empty (item (put (new, 3)))	ill-formed
item (put (new, 3))	3
item (remove (put (new, 3)))	incorrect
empty (remove (put (new, 7)))	True
item (put (put (remove (put (new, 4)), 3), 2))	2

Structural induction

Goal: Prove that a property P is valid for all correct terms T of the ADT.

Induction basis (step 0):

Prove that P holds for all creators of the ADT.

Induction hypothesis (step n-1):

Assume that P holds for any correct term T_{sub}.

Induction step (step n):

Prove for all commands that can be applied correctly to T_{sub} that P will still hold afterwards.

Sufficient completeness

An ADT is sufficiently complete if and only if:

1. For every term you can determine whether it is **correct** or not using the axioms of the ADT.
2. Every correct term where **the outermost function is a query** of the ADT can be reduced, using the axioms of the ADT, into **a term not using any function of the ADT**.

Your turn: Design an ADT (types, functions, preconditions, axioms)

We have the following requirements for a BANK_ACCOUNT class:

1. Every BANK_ACCOUNT has an owner and a balance.
2. The balance is recorded in “Rappen” (as an INTEGER).
3. The owner is recorded with his/her name (as a STRING).
4. It should always be possible to retrieve the balance and owner for any given BANK_ACCOUNT.
5. It is possible to deposit money to and withdraw money from the BANK_ACCOUNT.
6. The balance on the BANK_ACCOUNT is adjusted accordingly.
7. The balance of any BANK_ACCOUNT should never become negative.

ADT BANK_ACCOUNT

TYPES

BANK_ACCOUNT

FUNCTIONS

new_account: STRING $\rightarrow$ BANK_ACCOUNT

owner: BANK_ACCOUNT $\rightarrow$ STRING

balance: BANK_ACCOUNT $\rightarrow$ INTEGER

deposit: BANK_ACCOUNT $\times$ INTEGER $\nrightarrow$ BANK_ACCOUNT

withdraw: BANK_ACCOUNT $\times$ INTEGER $\nrightarrow$ BANK_ACCOUNT

ADT BANK_ACCOUNT (cont'd)

PRECONDITIONS (with $v \in$ INTEGER, $a \in$ BANK_ACCOUNT)

withdraw (a, v) **require** balance (a) $\geq$ v **and** v $\geq$ 0

deposit (a, v) **require** v $\geq$ 0

AXIOMS (with $o \in$ STRING, $v \in$ INTEGER, $a \in$ BANK_ACCOUNT)

A1: balance (new_account (o)) = 0

A2: owner (new_account (o)) = o

A3: balance (deposit (a, v)) = balance (a) + v

A4: balance (withdraw (a, v)) = balance (a) – v

To Do

- Prove by structural induction of bank accounts that the value returned by "balance" is never negative.
- The specification is not sufficiently complete; show why. Add axiom(s) to make it sufficiently complete, and prove that, with such an extension, it is sufficiently complete.

Proof: balance non-negative

- We prove this by induction over the structure of correct bank accounts:
  - Base case: The bank account is of the form new_account(o), and we know balance(new_account(o)) = 0 and that 0 >= 0.
  - Step case: The bank account can have one of two forms, where *a* is a correct bank account with balance(*a*) >= 0:
    - Form deposit(*a*,i) where i >= 0. By axiom A3, we know that balance(deposit(*a*,i)) = balance(*a*) + i which is non-negative because of the induction hypothesis and i >= 0
    - Form withdraw(*a*,i) where balance(*a*) >= i >= 0. From axiom A4 it follows that balance(withdraw(*a*,i)) >= 0.

Proof: sufficient completeness (1)

The ADT is not sufficiently complete, since we cannot determine the owner of an account if a deposit or withdrawal was made.

To make it sufficiently complete, we have to add the axioms:

A5: owner(deposit(a,v)) = owner(a)

A6: owner(withdraw(a,v)) = owner(a)

Proof: sufficient completeness (2)

Let P(*n*) be the property "for all terms *a* of type BANK_ACCOUNT with at most *n* applications of deposit and withdraw, it can be proven 1) whether *a* is correct or not and 2) whether balance(*a*) and owner(*a*) are correct or not and if correct, whether they can be reduced to terms not involving new_account, owner, balance, deposit and withdraw"

Base case n=0: *a* is new_account(o), which is correct, and balance(*a*) = 0 and owner(*a*) = o. Thus P(0) holds.

Proof: sufficient completeness (3)

- Step case: We assume the induction hypothesis (IH) P(n-1) and have to prove P(n).
  First case: *a* is deposit(*b*,i) and the IH applies to terms *b* and i.

1. Term *a* is correct iff *b* and i are correct, which we can determine by IH, and i > 0, which we can determine (since we can reduce i to a term not using functions of BANK_ACCOUNT by IH).
2. * balance(*a*) is correct iff *a* is correct, which we can determine (see 1). If balance(*a*) is correct, then balance(*a*) = balance(b) + i, which can be reduced to a term not using functions of BANK_ACCOUNT by IH.
   * owner(*a*) is correct iff *a* is correct, which we can determine (see 1). If owner(*a*) is correct, then owner(*a*) = owner(*b*), which can be reduced to a term not using functions of BANK_ACCOUNT by IH.

Proof: sufficient completeness (4)

- Step case (continued)

  Second case: *a* is withdraw(*b*,i) and the IH applies to terms *b* and i.

1. Term *a* is correct iff *b* and i are correct, which we can determine by IH, and balance(*b*) >= i >= 0, which we can determine (since we can reduce balance(*b*) and i to terms not using functions of BANK_ACCOUNT by IH).
2. * balance(*a*) is correct iff *a* is correct, which we can determine (see 1). If balance(*a*) is correct, then balance(*a*) = balance(b) - i, which can be reduced to a term not using functions of BANK_ACCOUNT by IH.
   * owner(*a*) is correct iff *a* is correct, which we can determine (see 1). If owner(*a*) is correct, then owner(*a*) = owner(*b*), which can be reduced to a term not using functions of BANK_ACCOUNT by IH. **QED**

Supplementary reading

Object-oriented Software Construction, Second Edition, by Bertrand Meyer, pp. 148-159