ETH Zürich
Chair of Software Engineering

# Software Architecture

Bertrand Meyer, Carlo A. Furia, Martin Nordio

ETH Zurich, February-May 2011

## Lecture 5: Modularity and Abstract Data Types

# Three topics (over today & tomorrow)

1. Modularity

2. The theory of abstract data types

3. Object-oriented analysis

# Reading assignment for this week

*OOSC*, chapters

3: Modularity

6: Abstract data types

In particular pp.153-159,
sufficient completeness

# Modularity

General goal:

Ensure that software systems are structured into units (modules) chosen to favor

- Extendibility
- Reusability
- "Maintainability"
- Other benefits of clear, well-defined architectures

# Modularity

Some principles of modularity:

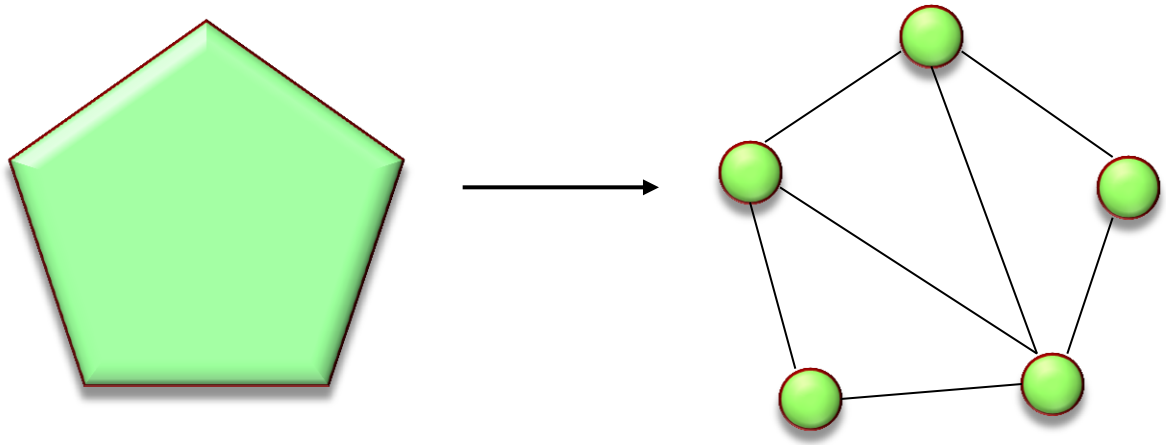
- Decomposability
- Composability
- Continuity
- Information hiding
- The open-closed principle
- The single choice principle

# Decomposability

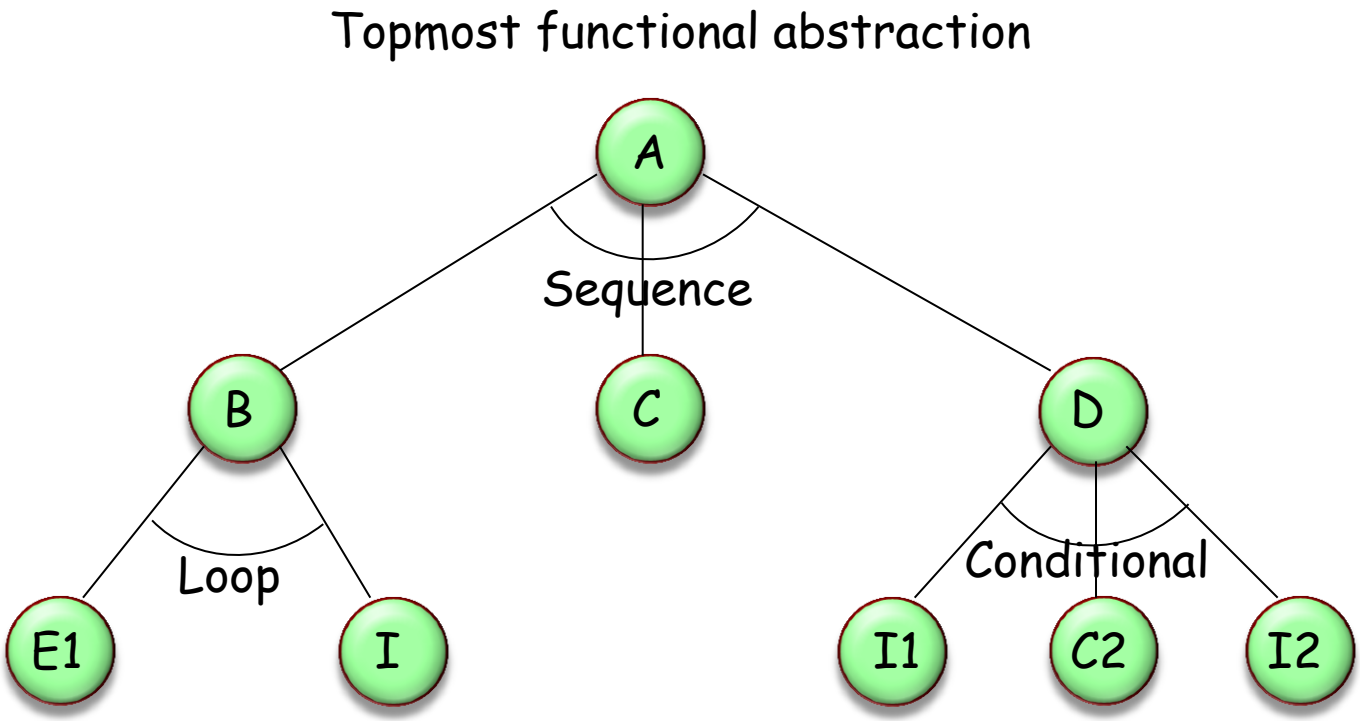
The method helps decompose complex problems into subproblems

COROLLARY: Division of labor.

- Example: Top-down design method (see next).
- Counter-example: General initialization module.



# Top-down functional design



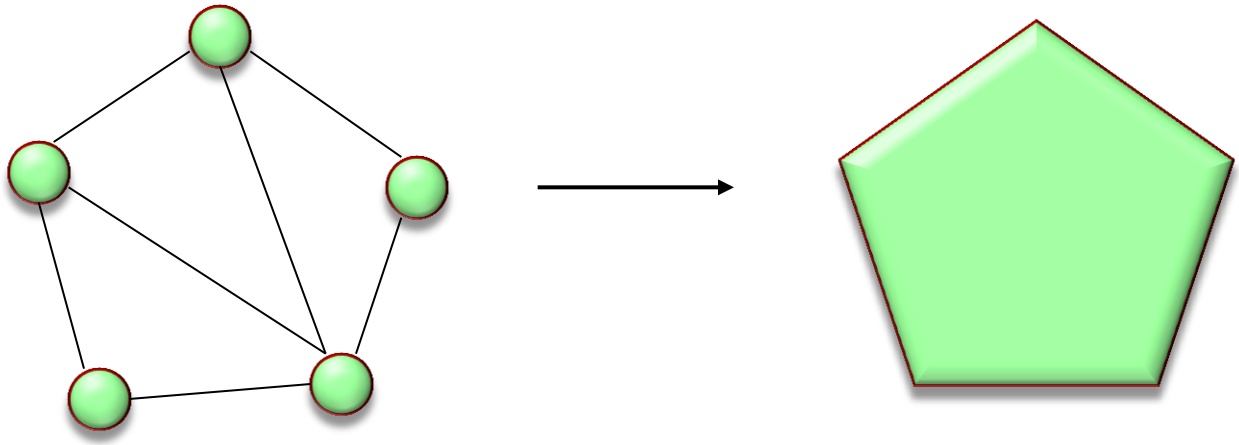
# Top-down design

See Niklaus Wirth, "Program Construction by Stepwise Refinement", Communications of the ACM, 14, 4, (April 1971), p 221-227.

http://www.acm.org/classics/dec95/

# Composability

The method favors the production of software elements that may be freely combined with each other to produce new software



Example: Unix shell conventions

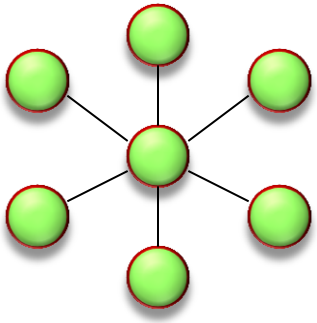
Program1 | Program2 | Program3

# Direct Mapping

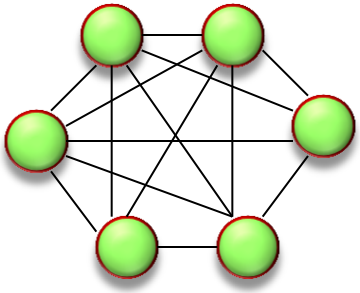
The method yields software systems whose modular structure remains compatible with any modular structure devised in the process of modeling the problem domain

# Few Interfaces principle

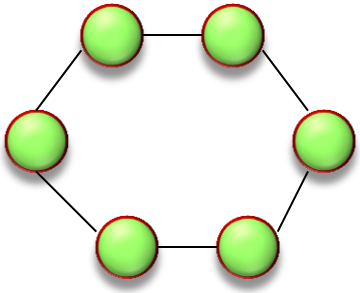
Every module communicates with
as few others as possible



(A)



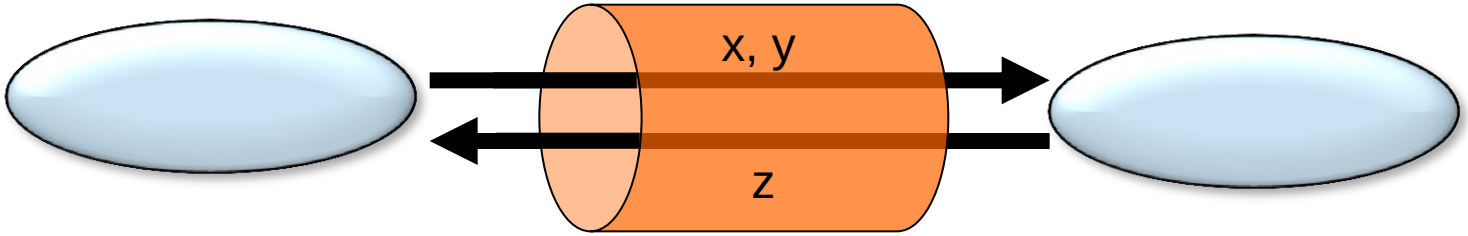
(B)



(C)

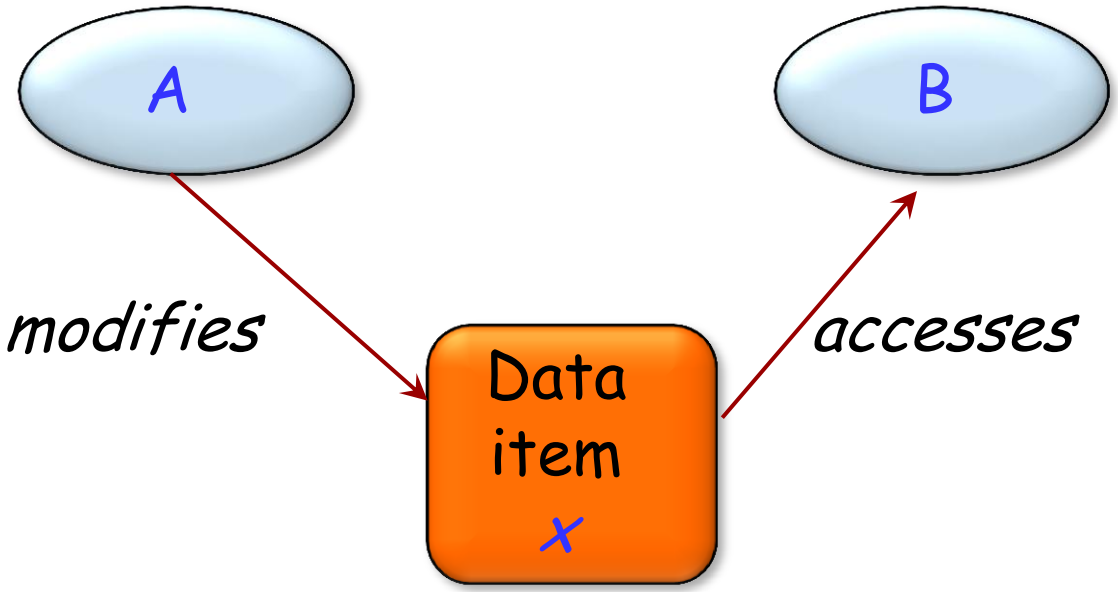
# Small Interfaces principle

If two modules communicate, they exchange as little information as possible



# Explicit Interfaces principle

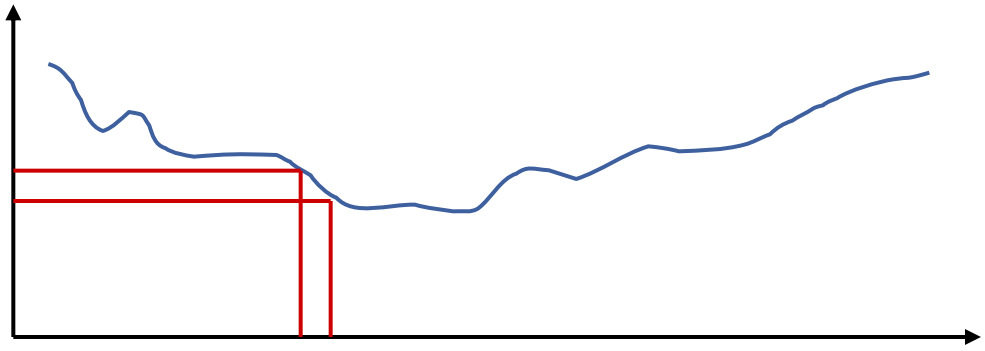
Whenever two modules communicate, this is clear from the text of one or both of them



# Continuity

The method ensures that small changes in specifications yield small changes in architecture.

*Design method*: Specification → Architecture



Example: Principle of Uniform Access (see next)

Counter-example: Programs with patterns after the physical implementation of data structures.

# Uniform Access principle

It doesn't matter to the client whether you look up or compute

A call such as

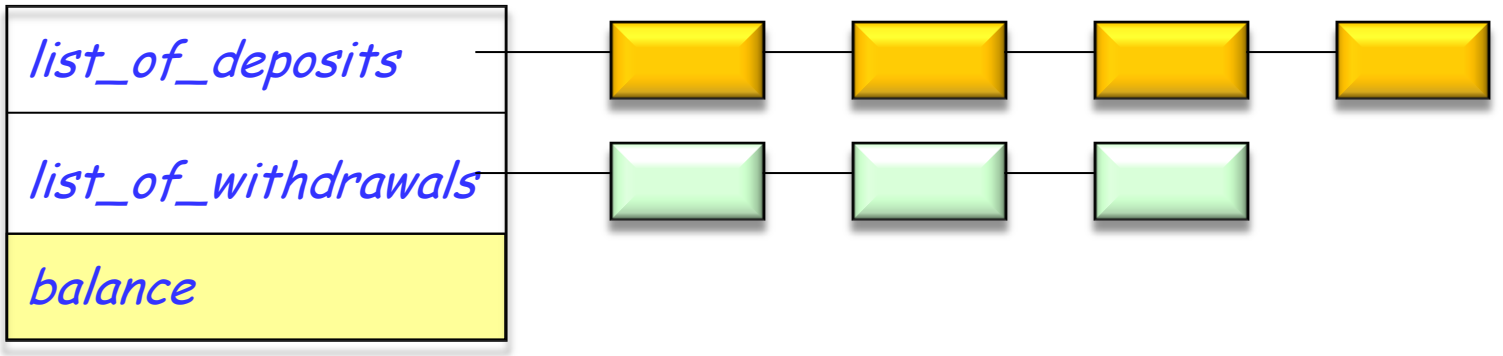
*your_account.balance*

could use an attribute or a function

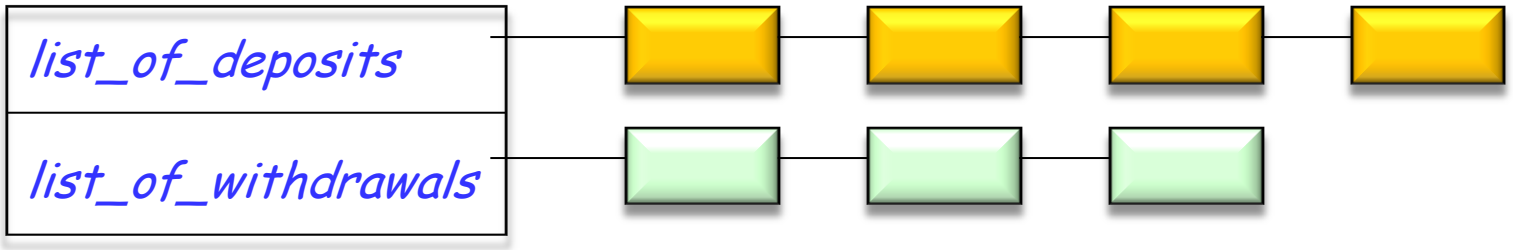
# Uniform Access

*balance = list_of_deposits.total - list_of_withdrawals.total*

(A1)




(A2)




Ada, Pascal, C/C++, Java, C#:

*a.balance*

*balance(a)*  *a.balance()*

Simula, Eiffel:

*a.balance*

# Uniform Access principle

Facilities managed by a module are accessible to its clients in the same way whether implemented by computation or by storage.

Definition: A client of a module is any module that uses its facilities.

# Information Hiding

Underlying question: how does one "advertise" the capabilities of a module?

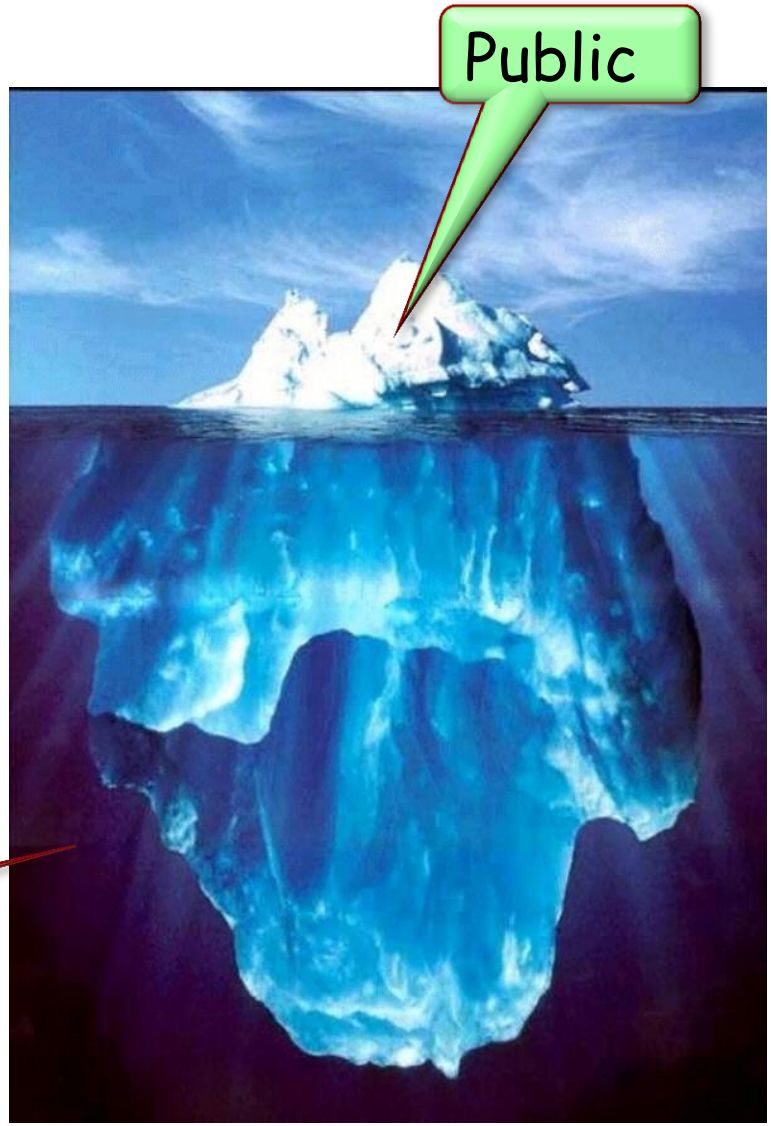
Every module should be known to the outside world through an official, "public" interface.

The rest of the module's properties comprises its "secrets".

It should be impossible to access the secrets from the outside.

# Information Hiding Principle

The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules



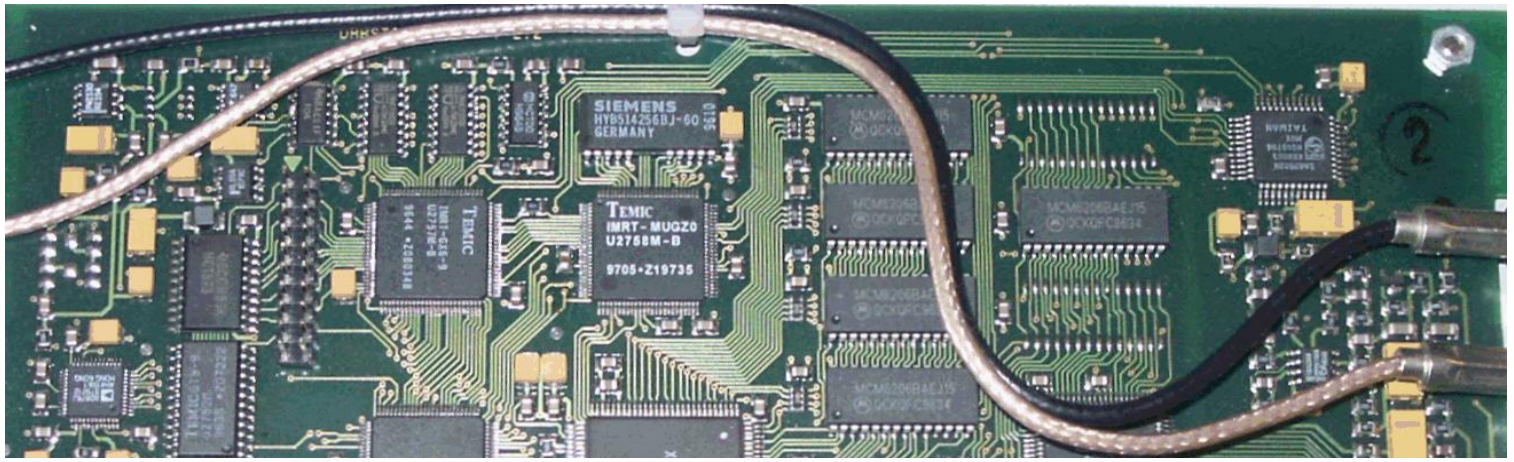
# Information hiding

Justifications:

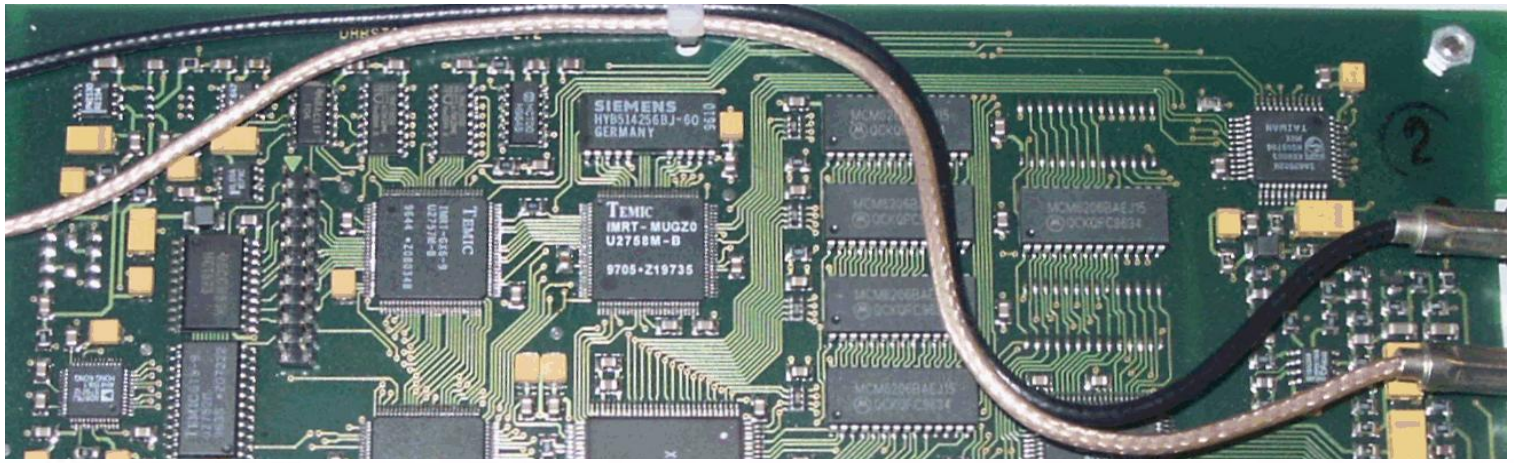
- Continuity
- Decomposability

# An object has an interface

# An object has an implementation




# Information hiding



# The Open-Closed Principle

**Modules should be open and closed**

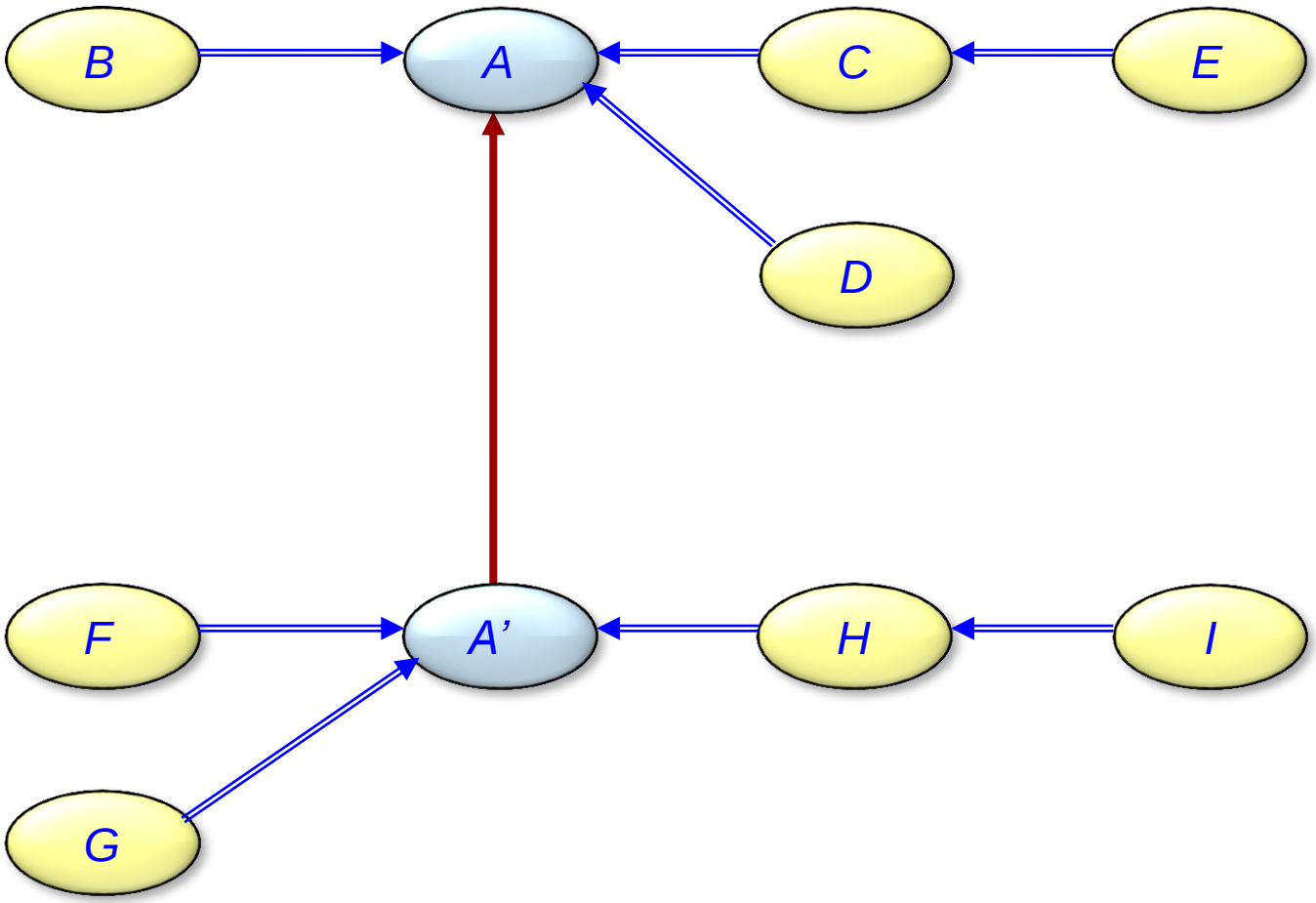
Definitions:

- Open module: May be extended.
- Closed module: Usable by clients. May be approved, baselined and (if program unit) compiled.

The rationales are complementary:

- For closing a module (manager's perspective): Clients need it now.
- For keeping modules open (developer's perspective): One frequently overlooks aspects of the problem.

# The Open-Closed principle



# The Single Choice principle

Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list.

- Editor: set of commands (insert, delete etc.)
- Graphics system: set of figure types (rectangle, circle etc.)
- Compiler: set of language constructs (instruction, loop, expression etc.)

# Reusability: Technical issues

General pattern for a searching routine:

```
has (t: TABLE; x: ELEMENT ): BOOLEAN
                -- Does x appear in t ?
        local
                pos: POSITION
        do
                from
                        pos := initial_position (t, x)
                until
                        exhausted (t, pos) or else found (t, x, pos)
                loop
                        pos := next (t, x, pos)
                end

                Result := found (t, x, pos)
        end
```

# Issues for a general searching module

Type variation:

- What are the table elements?

Routine grouping:

- A searching routine is not enough: it should be coupled with routines for table creation, insertion, deletion etc.

Implementation variation:

- Many possible choices of data structures and algorithms: sequential table (sorted or unsorted), array, binary search tree, file, ...

# Issues

## Representation independence:

- Can a client request an operation such as table search (*has*) without knowing what implementation is used internally?

*has* ( *t1*, *y* )

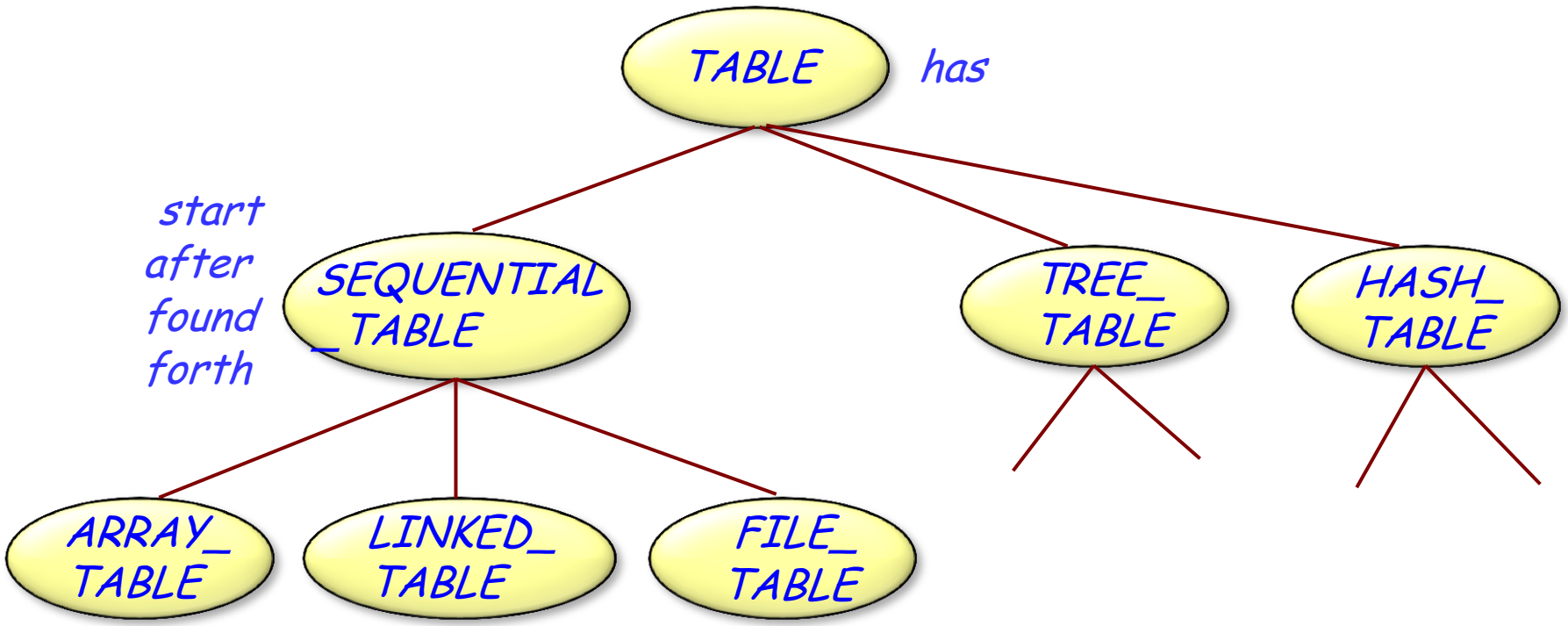
# Issues

Factoring out commonality:

- How can the author of supplier modules take advantage of commonality within a subset of the possible implementations?
- Example: the set of sequential table implementations.
- A common routine text for *has*:

```
has ( ...; x: ELEMENT ): BOOLEAN
        -- Does x appear in t?
        do
        from start until after or else found (x) loop
                forth
        end
        Result := found (x)
 end
```

# Factoring out commonality



# Implementation variants

| | *start* | *forth* | *after* | *found (x)* |
|---|---|---|---|---|
| Array | *i* := 1 | *i* := *i* + 1 | *i* > *count* | *t* [*i*] = *x* |
| Linked list | *c* := *first* | *c* := *c.right* | *c* = **Void** | *c.item* = *x* |
| File | *rewind* | *read* | *end_of_file* | *item* = x |

# Encapsulation languages ("Object-based")

Ada, Modula-2, Oberon, CLU...

Basic idea: gather a group of routines serving a related purpose, such as *has*, *insert*, *remove* etc., together with the appropriate data structure descriptions.

This addresses the Related Routines issue.

Advantages:

- For supplier author: Get everything under one roof. Simplifies configuration management, change of implementation, addition of new primitives.
- For client author: Find everything at one place. Simplifies search for existing routines, requests for extensions.

# The concept of Abstract Data Type (ADT)

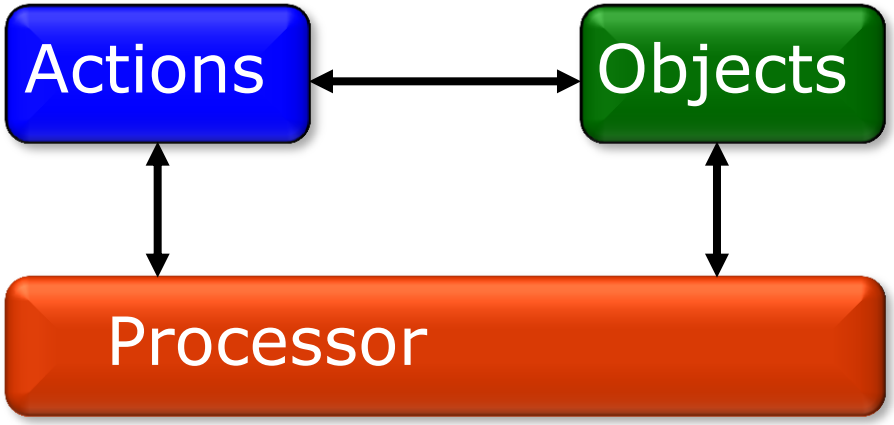
- Why use the objects?
- The need for data abstraction
- Moving away from the physical representation
- Abstract data type specifications
- Applications to software design

# The first step

A system performs certain actions on certain data.

Basic duality:

- Functions [or: Operations, Actions]
- Objects [or: Data]



# Finding the structure

The structure of the system may be deduced from an analysis of the functions (1) or the objects (2)

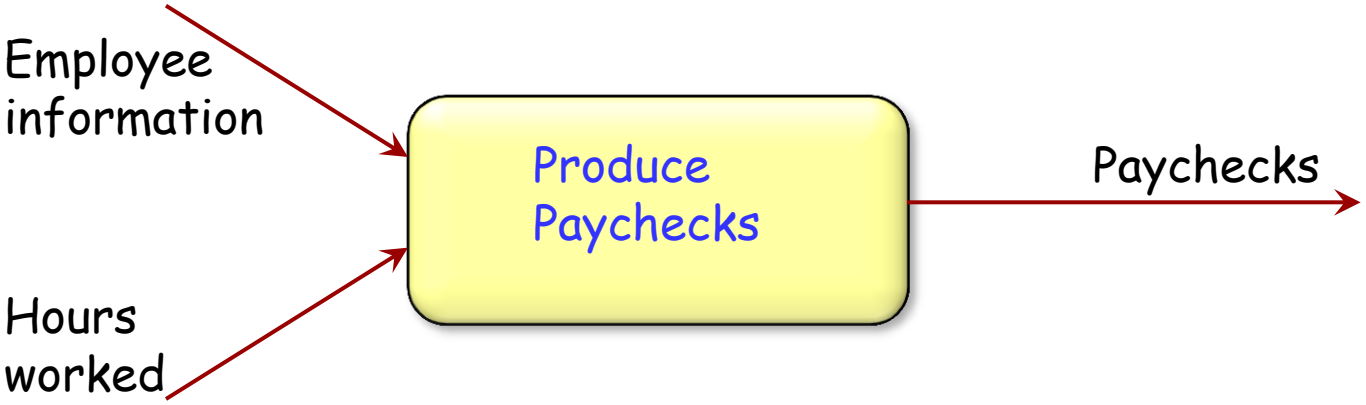
Resulting architectural style and analysis/design method:

- (1) Top-down, functional decomposition
- (2) Object-oriented

# Arguments for using objects

Reusability: Need to reuse whole data structures, not just operations

Extendibility, Continuity: Object categories remain more stable over time.



# Object technology: A first definition

Object-oriented software construction is the software architecture method that bases the structure of systems on the types of objects they handle — not on "the" function they achieve.

# The O-O designer's motto

Ask not first WHAT the system does:

Ask WHAT it does it to!

# Issues of object-oriented architecture

- How to find the object types
- How to describe the object types
- How to describe the relations and commonalities between object types
- How to use object types to structure programs

# Description of objects

Consider not a single object but a type of objects with similar properties.

Define each type of objects not by the objects' physical representation but by their behavior: the services (FEATURES) they offer to the rest of the world.

External, not internal view: ABSTRACT DATA TYPES

# The theoretical basis

The main issue: How to describe program objects (data structures):

- Completely
- Unambiguously
- Without overspecifying?

(Remember information hiding)

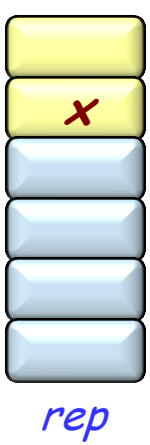
# Abstract Data Types

A formal way of describing data structures

Benefits:

- Modular, precise description of a wide range of problems
- Enables proofs
- Basis for object technology
- Basis for object-oriented requirements

# A stack, concrete object

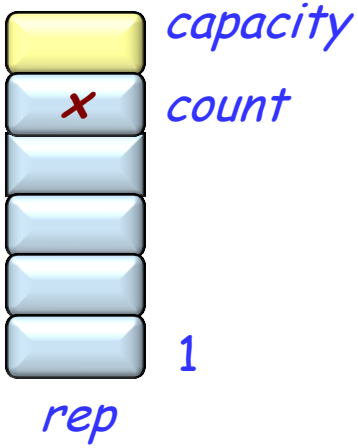

Representation 1:
"Array Up"

Implementing a "PUSH" operation:

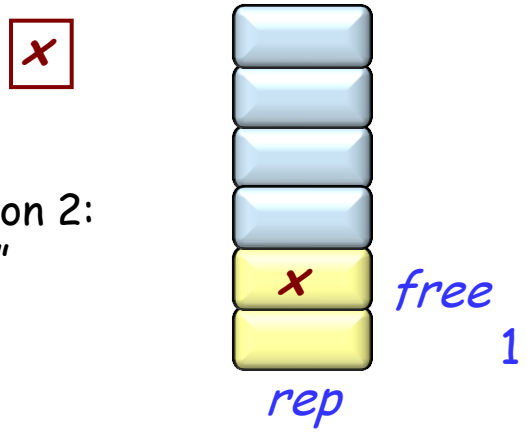
*count* := *count* + 1
*rep* [*count*] := *x*

# A stack, concrete object

Representation 1:
"Array Up"




Representation 2:
"Array Down"



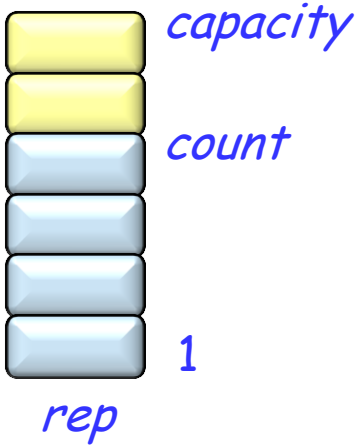

Implementing a "PUSH" operation:

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

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

# A stack, concrete object

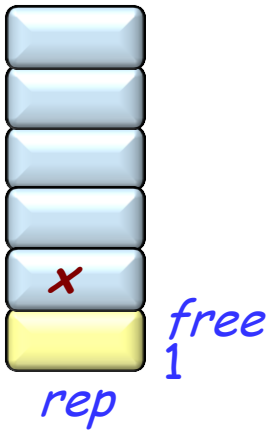
Representation 1:
"Array Up"



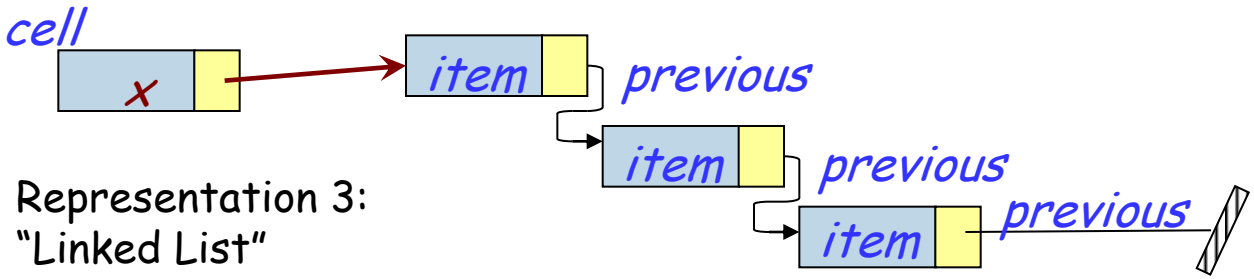

Implementing a "PUSH" operation:

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

Representation 2:
"Array Down"




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




Representation 3:
"Linked List"

**create** *cell*
*cell.item* := *x*
*cell.previous* := *last*
*head* := *cell*

# Stack: An Abstract Data Type (ADT)

Types:

$STACK[G]$

-- $G$: Formal generic parameter

Functions (Operations):

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

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

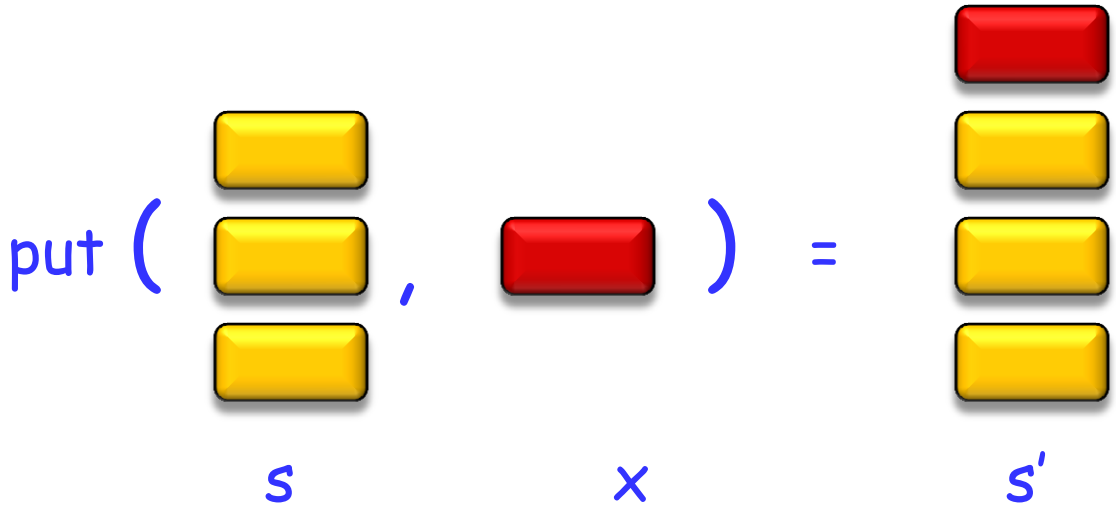
$item : STACK[G] \nrightarrow G$

$is\_empty : STACK[G] \to BOOLEAN$

$new : STACK[G]$

Partial function (see next)

# Using functions to model operations



# Reminder: Partial functions

A partial function, identified here by $\nrightarrow$, is a function that may not be defined for all possible arguments.

Example from elementary mathematics:

- *inverse*: $\mathbb{R} \nrightarrow \mathbb{R}$, such that

$$inverse\,(x) = 1 / x$$

# The *STACK* ADT (continued)

Preconditions:

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

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

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

*item* (*put* (*s*, *x*)) = *x*

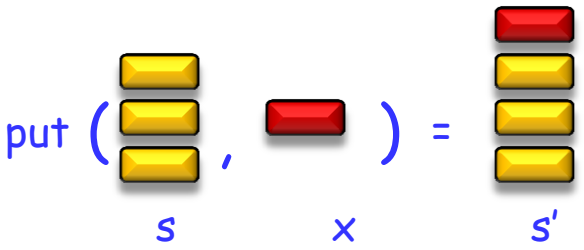
*remove* (*put* (*s*, *x*)) = *s*

*is_empty* (*new*)

(can also be written: *is_empty* (*new*) = **True**)

**not** *is_empty* (*put* (*s*, *x*))

(can also be written: *is_empty* (*put* (*s*, *x*)) = **False**)



# Exercises

Adapt the preceding specification of stacks (LIFO, Last-In First-Out) to describe queues instead (FIFO).

Adapt the preceding specification of stacks to account for bounded stacks, of maximum size *capacity*.

- Hint: *put* becomes a partial function.

# Formal stack expressions

*value =*

*item (remove (put (remove (put (put*

*(remove (put (put (put (new, x8), x7 ), x6 )),*

*item (remove (put (put (new, x5), x4 )))),*

*x2 )), x1 )))*

# Expressed differently

*value = item (remove (put (remove (put (put (remove (put (put (put (new, x8), x7), x6)), item (remove (put (put (new, x5), x4)))), x2)), x1)))*

*s1 = new*

*s2 = put (put (put (s1, x8 ), x7 ), x6 )*

*s3 = remove (s2)*

*s4 = new*

*s5 = put (put (s4, x5 ), x4)*

*s6 = remove (s5)*

*y1 = item (s6 )*

*s7 = put (s3, y1 )*

*s8 = put (s7, x2 )*

*s9 = remove (s8 )*

*s10 = put (s9, x1 )*

*s11 = remove (s10 )*

*value = item (s11 )*

# Expression reduction

```
value = item (
            remove (
                put (
                    remove (
                        put (                                   Stack 1
                            put (
                                remove (
                                    put (put (put ( new , x8 ), x7), x6 )
                                ),
                            item    (
                                remove (
                                    put (put (new, x5), x4)
                                )
                            ),
                        x2)
                    ),
                x1)
            )
    )
```

# Expression reduction

```
value = item (
            remove (
                put (
                    remove (
                                put (
                                    put (
                                        remove (
                                            put (put ( put (  new , x8 ), x7), x6 )
                                        ),
                                        item    (
                                            remove (
                                                put (put (new, x5), x4)
                                            )
                                        ),
                                        x2)
                                ),
                            x1)
                    )
            )
```

x8

Stack 1

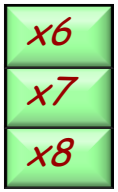
# Expression reduction

```
value = item (
            remove (
                put (
                    remove (
                        put (
                            put (
                                remove (
                                    put (put (put (  new , x8), x7), x6 )
                                ),
                                item    (
                                    remove (
                                        put (put (new, x5), x4)
                                    )
                                )
                            ),
                            x2)
                        ),
                    x1)
                )
        )
```


Stack 1

# Expression reduction

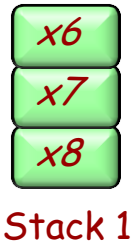
```
value = item (
            remove (
                put (
                    remove (
                        put (
                            put (
                                remove (
                                    put (put (put (  new , x8 ), x7), x6 )
                                )
                                , item    (
                                    remove (
                                        put (put (new, x5), x4)
                                    )
                                )
                            ),
                        x2)
                    ),
                x1)
            )
    )
```




Stack 1

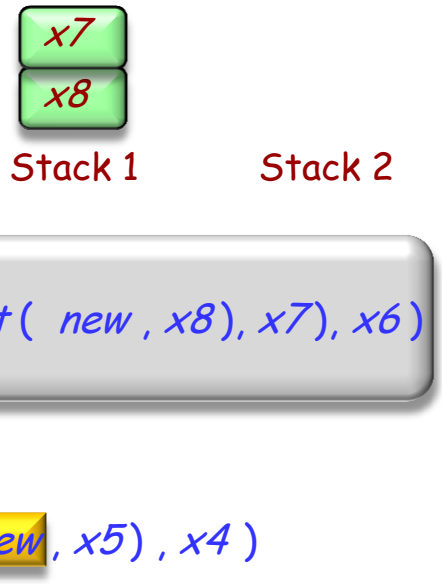
# Expression reduction

```
value = item (
            remove (
                put (
                    remove (
                        put (
                            put (
                                remove (
                                    put (put (put (  new , x8), x7), x6 )
                                ),
                                item    (
                                    remove (
                                        put (put (new, x5), x4)
                                    )
                                )
                            ),
                            x2)
                        ),
                    x1)
                )
            )
```



# Expression reduction

```
value = item (
            remove (
                put (
                    remove (
                        put (
                            put (
                                remove (
                                    put (put (put (  new , x8 ), x7), x6 )
                                ),
                                item    (
                                    remove (
                                        put ( put ( new , x5) , x4 )
                                    )
                                )
                            ),
                            x2)
                        ),
                    x1)
                )
            )
```

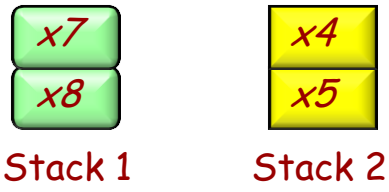


# Expression reduction

```
value = item (
            remove (
                put (
                    remove (
                        put (
                            put (
                                remove (
                                    put (put (put (  new , x8), x7), x6)
                                ),
                                item    (
                                    remove (
                                        put ( put (new, x5) , x4)
                                    )
                                )
                            ),
                            x2)
                        ),
                    x1)
                )
            )
```

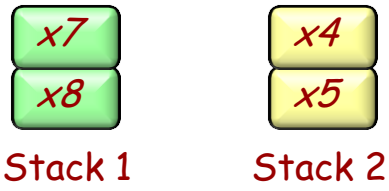
# Expression reduction

```
value = item (
            remove (
                put (
                    remove (
                        put (
                            put (
                                remove (
                                    put (put (put (  new , x8 ), x7), x6 )
                                ),
                                item    (
                                    remove (
                                        put (put ( new, x5 ), x4)
                                    )
                                )
                            ),
                            x2)
                        ),
                    x1)
                )
            )
```



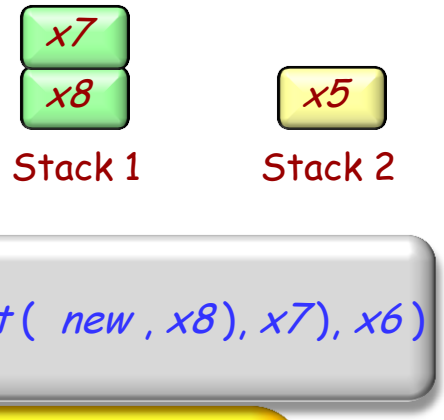
# Expression reduction

```
value = item (
            remove (
                put (
                    remove (
                        put (
                            put (
                                remove (
                                    put (put (put (  new , x8), x7), x6)
                                ),
                                item    (
                                    remove (
                                        put (put (new, x5), x4)
                                    )
                                )
                            ),
                            x2)
                        ),
                    x1)
                )
            )
```



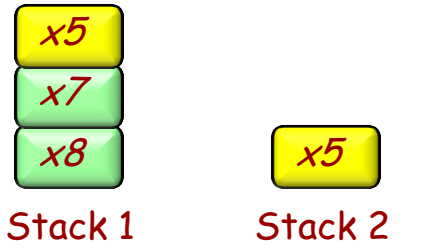
# Expression reduction

```
value = item (
            remove (
                put (
                    remove (
                        put (
                            put (
                                remove (
                                    put (put (put (  new , x8), x7), x6 )
                                ),
                                item    (
                                    remove (
                                        put (put (new, x5), x4)
                                    )
                                ),
                                x2)
                        ),
                    x1)
            )
    )
```



# Expression reduction

```
value = item (
            remove (
                put (
                    remove (
                        put (
                            put (
                                remove (
                                    put (put (put (  new , x8), x7), x6 )
                                ),
                                item    (
                                    remove (
                                        put (put (new, x5), x4)
                                    )
                                )
                            ),
                            x2)
                    ),
                x1)
            )
    )
```

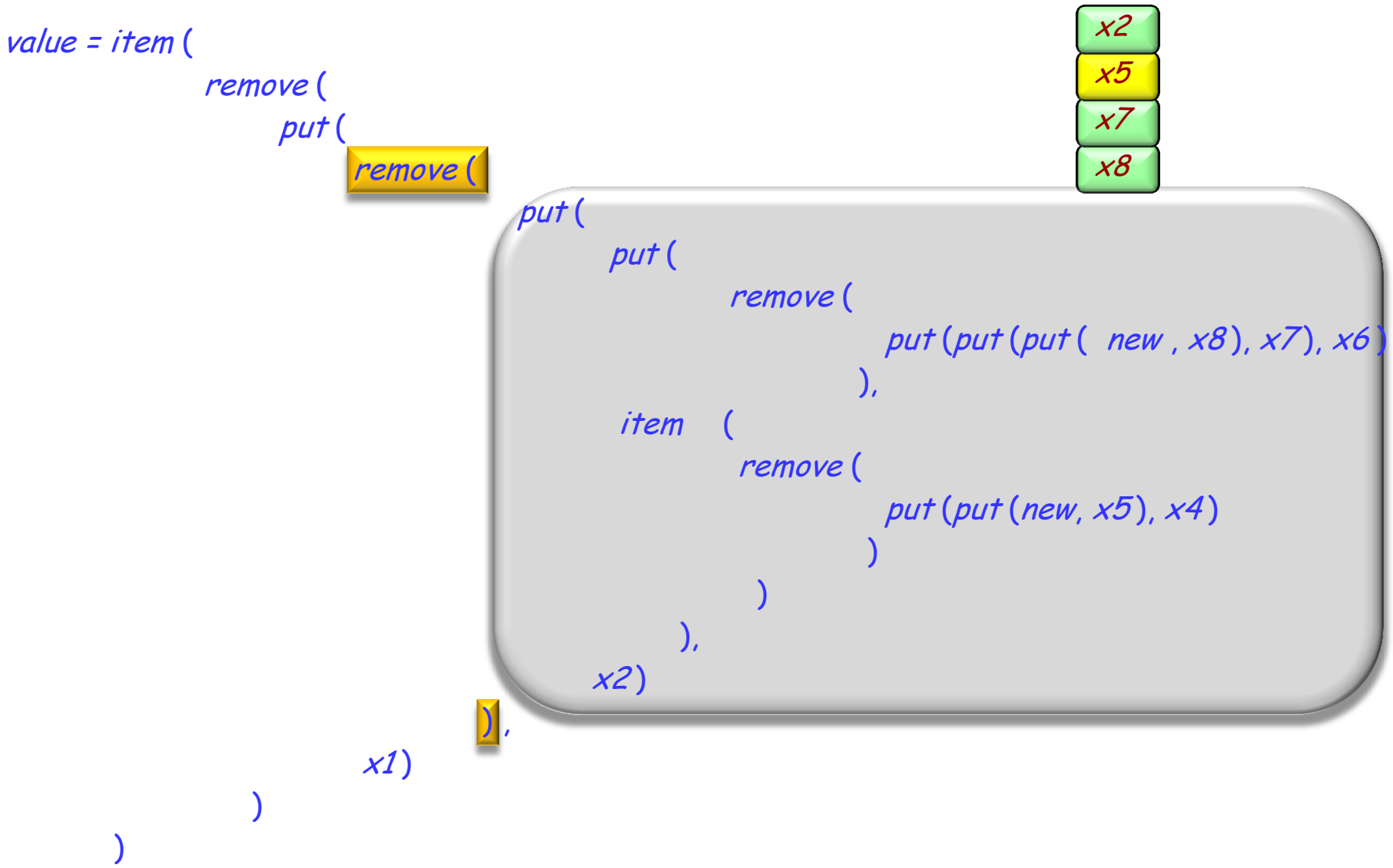


# Expression reduction

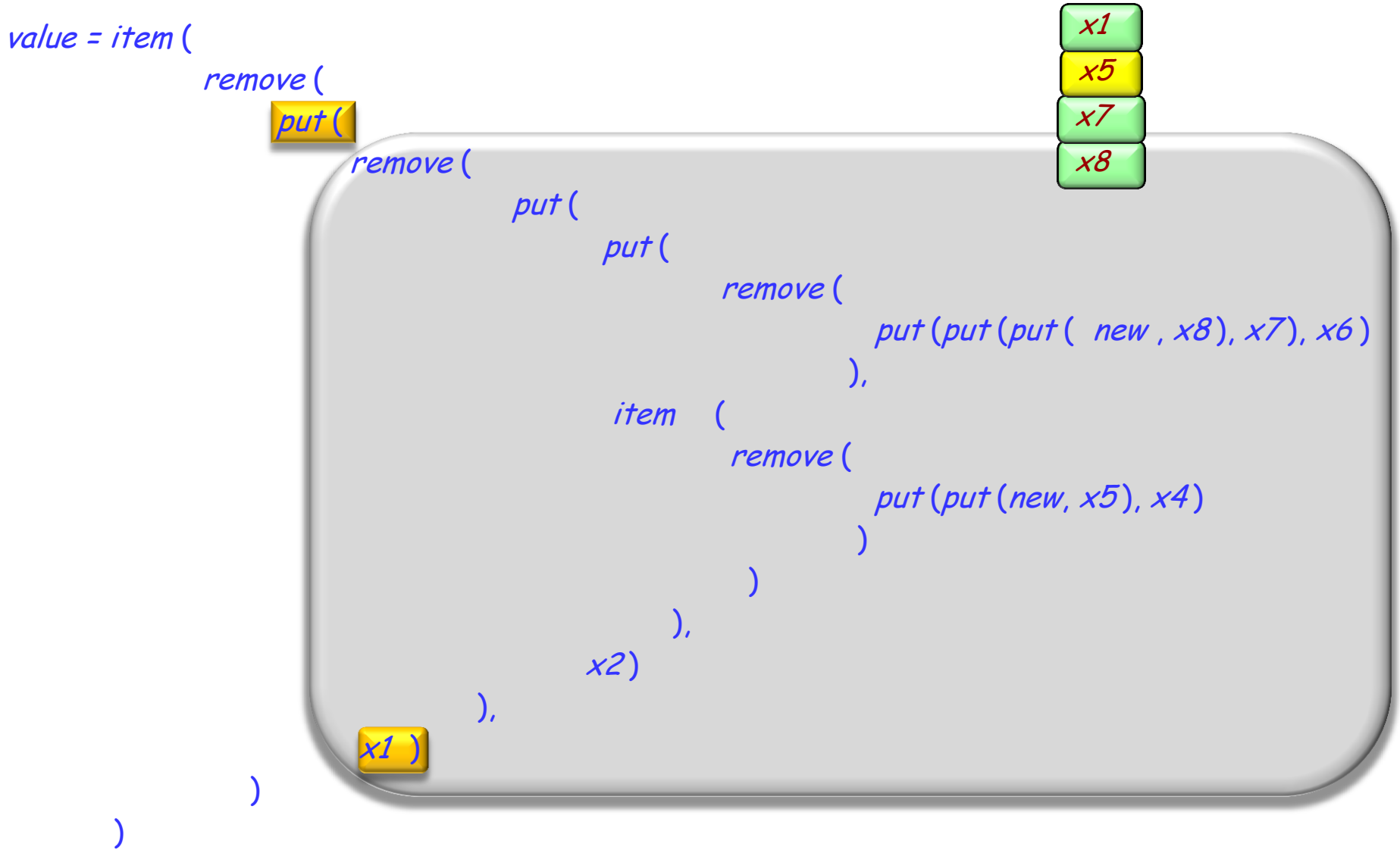
```
value = item (
            remove (
                put (
                    remove (
                        put (
                            put (
                                remove (
                                    put (put (put (  new , x8), x7), x6 )
                                ),
                            item    (
                                remove (
                                    put (put (new, x5), x4)
                                )
                            )
                        ),
                        x2 )
                    ),
                x1)
            )
    )
```

x2
x5
x7
x8

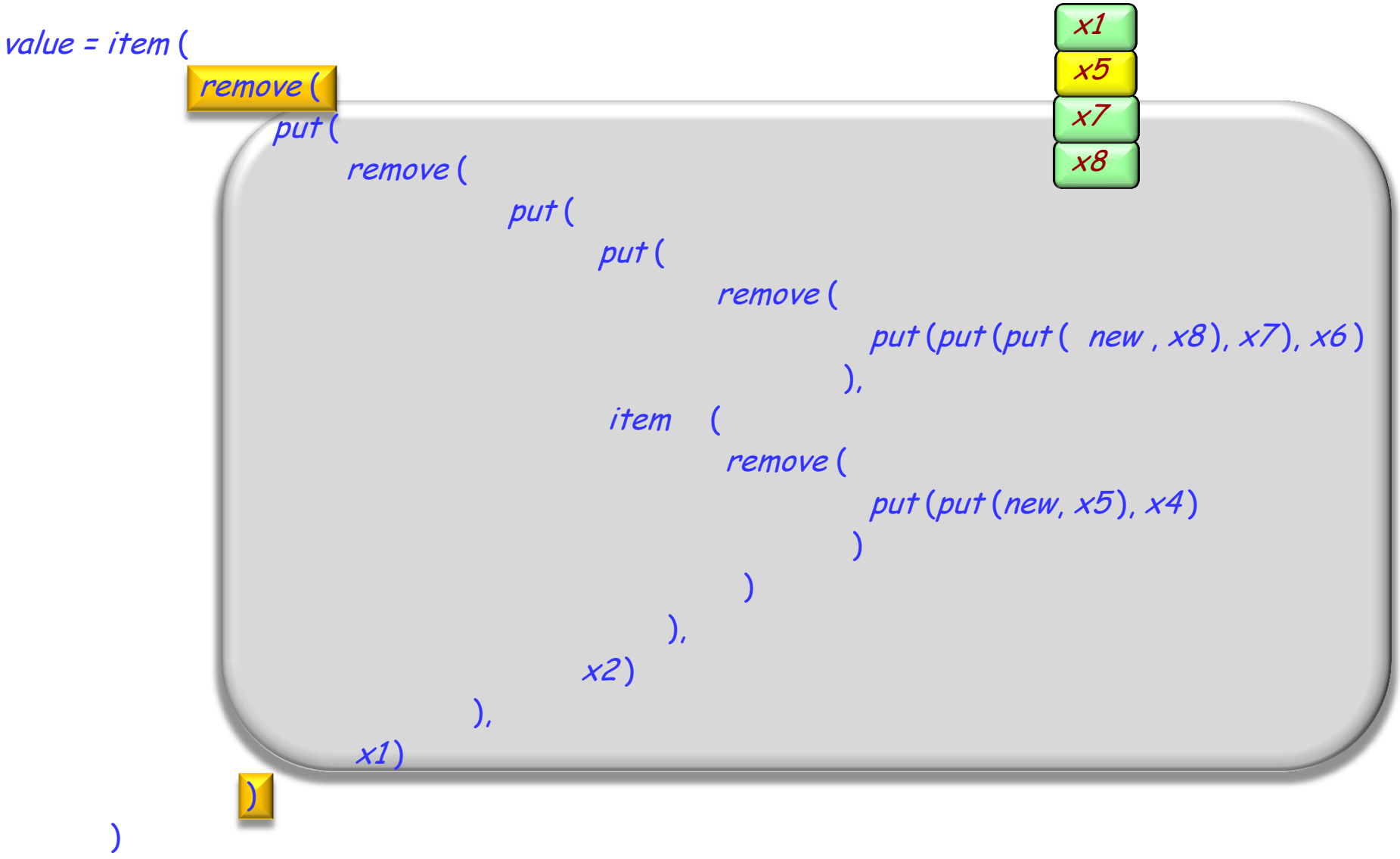
# Expression reduction



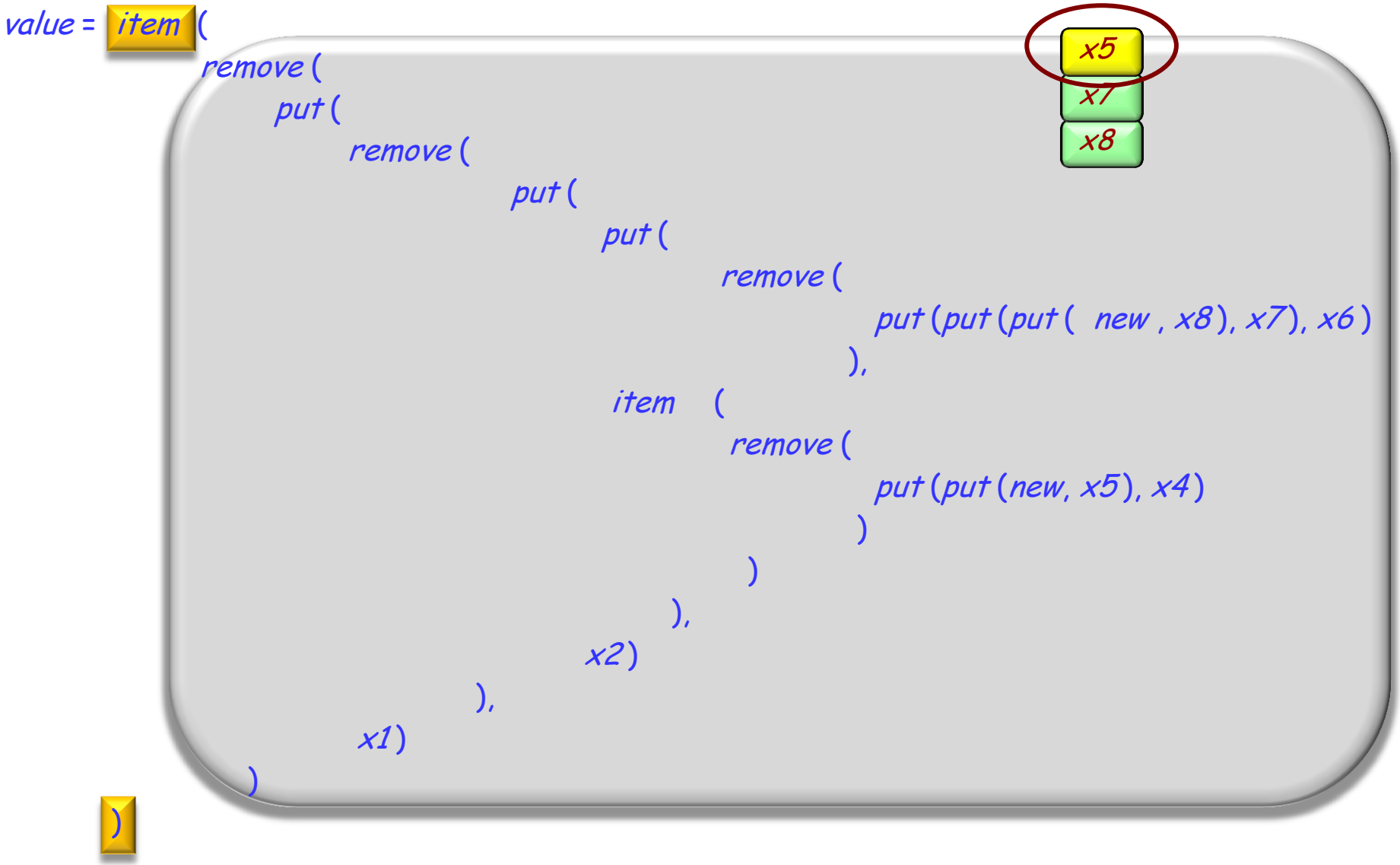
# Expression reduction



# Expression reduction



# Expression reduction



# Expressed differently

*value = item (remove (put (remove (put (put (remove (put (put (put (new, x8), x7), x6)), item (remove (put (put (new, x5), x4)))), x2)), x1)))*

*s1 = new*

*s2 = put (put (put (s1, x8 ), x7 ), x6 )*

*s3 = remove (s2)*

*s4 = new*

*s5 = put (put (s4, x5 ), x4)*

*s6 = remove (s5)*

*y1 = item (s6 )*

*s7 = put (s3, y1 )*

*s8 = put (s7, x2 )*

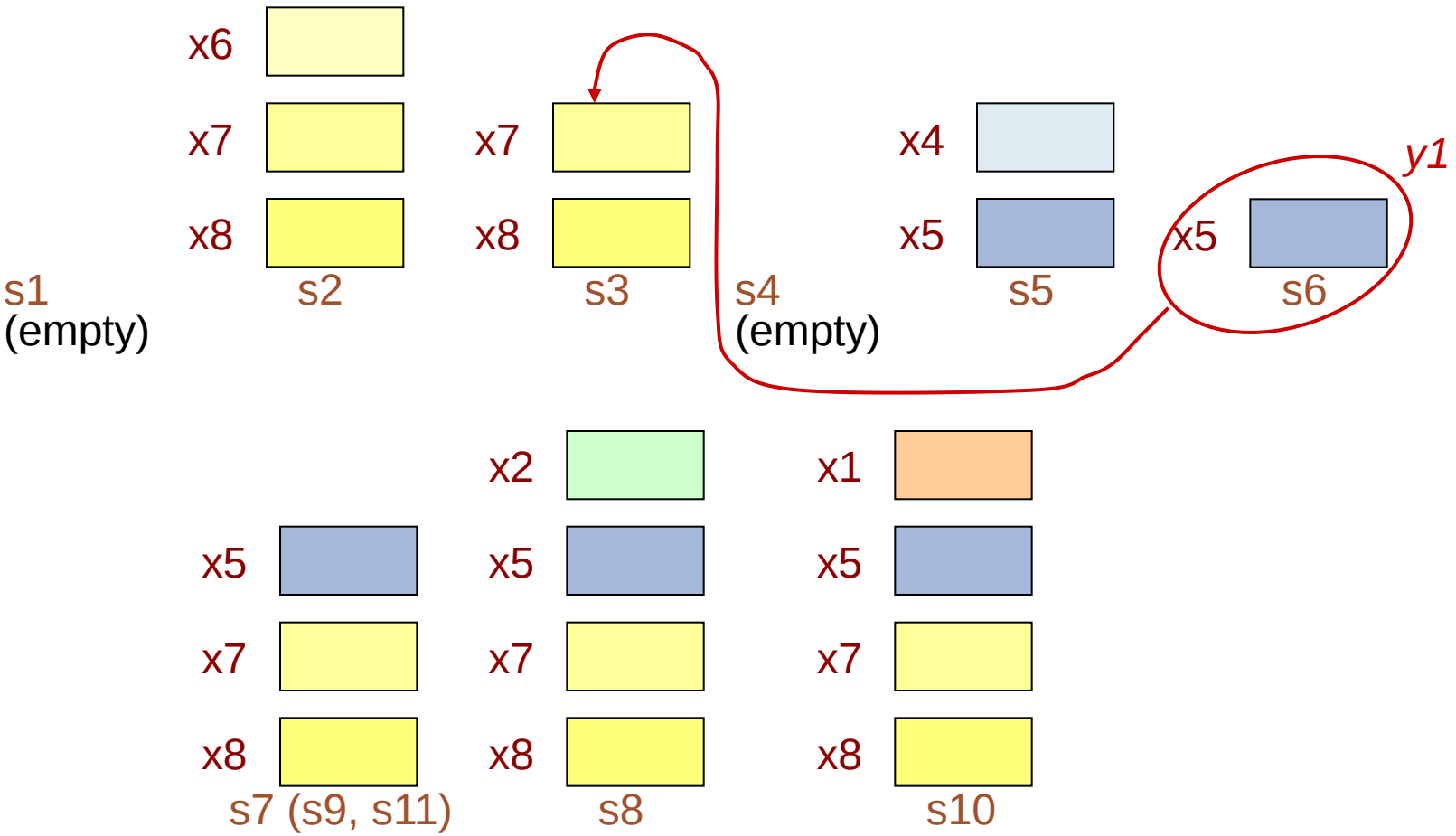
*s9 = remove (s8 )*

*s10 = put (s9, x1 )*

*s11 = remove (s10 )*

*value = item (s11 )*

# An operational view of the expression




value = item (remove (put (remove (put (put (remove (put (put (put (new, x8 ), x7 ), x6)), item (remove (put (put (new, x5 ), x4 )))), x2 )), x1 )))

# Sufficient completeness

Three forms of functions in the specification of an ADT $T$:

- Creators:
  $OTHER \rightarrow T$ e.g. *new*
- Queries:
  $T \times \ldots \rightarrow OTHER$ e.g. *item, empty*
- Commands:
  $T \times \ldots \rightarrow T$ e.g. *put, remove*

Sufficiently Complete specification

An ADT specification with axioms that make it possible to reduce any correct "Query Expression" of the form

$f(\ldots)$

where $f$ is a query, to a form not involving $T$

# Well-formed and correct expression

An expression built from an ADT is **well-formed** if all the function applications it involves use the right number of arguments of the right type, each argument being (recursively) well-formed

Examples:

put (x) is not well-formed

put (new, x) is well-formed

remove (new) is well-formed

An expression is **correct** if (1) it is well-formed (2) the arguments to all function applications satisfy the respective preconditions and (3) all these arguments are (recursively) correct.

# Is the stack specification sufficiently complete?

Types

$STACK[G]$

Functions

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

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

$item: STACK[G] \nrightarrow G$

$empty: STACK[G] \to BOOLEAN$

$new: STACK[G]$

# The *STACK* ADT (continued)

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*]

*item* (*put* (*s*, *x*)) = *x*

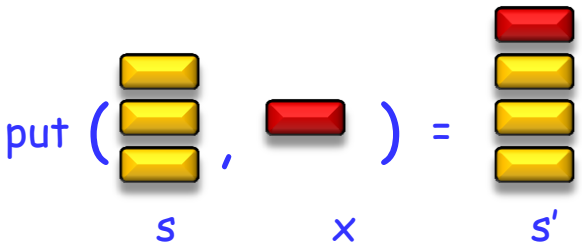
*remove* (*put* (*s*, *x*)) = *s*

*empty* (*new*)

(can also be written: *empty* (*new*) = **True**)

**not** *empty* (*put* (*s*, *x*))

(can also be written: *empty* (*put* (*s*, *x*)) = **False**)



# Proving sufficient completeness

Let us say that an ADT expression f for the ADT STACK [X] is **reducible** if the axioms make it possible to determine the value of f in terms of the types X and BOOLEAN only

To prove sufficient completeness for the stack specification, we must prove that a correct expression f of the form

item (e) or
empty (e)

is reducible

(to terms of values X and BOOLEAN respectively)

# Proof sketch

We prove that a correct expression f = item (e) is reducible

(the case of empty is left as an exercise)

We work by induction on the number of parenthesis pairs in f

**Lemma:**

A correct expression of the form

remove (g)

has a subexpression of the form

remove (put (h, x))

# "Complete" requirements (from the Requirements lecture)

Complete with respect to what?

Definition from IEEE standard (see next) :

*An SRS (Software Requirements Specification) is complete if, and only if, it includes the following elements:*

- *All significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interfaces. In particular any external requirements imposed by a system specification should be acknowledged and treated.*
- *Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations. Note that it is important to specify the responses to both valid and invalid input values.*
- *Full labels and references to all figures, tables, and diagrams in the SRS and definition of all terms and units of measure.*

# Completeness (from the Requirements lecture)

Completeness cannot be "completely" defined

But (taking advantage of the notion of *sufficient completeness* for abstract data types) we can cross-check:

- Commands x Queries

to verify that every effect is defined

# ADTs and software architecture

Abstract data types provide an ideal basis for modularizing software.

Build each module as an *implementation* of an ADT:

- Implements a set of *objects* with same *interface*
- Interface is defined by a set of operations (the ADT's functions) constrained by abstract properties (its axioms and preconditions).

The module consists of:

- A *representation* for the ADT
- An *implementation* for each of its operations
- Possibly, auxiliary operations

# Implementing an ADT

Three components:

(E1) The ADT's specification: functions, axioms, preconditions

(Example: stacks)

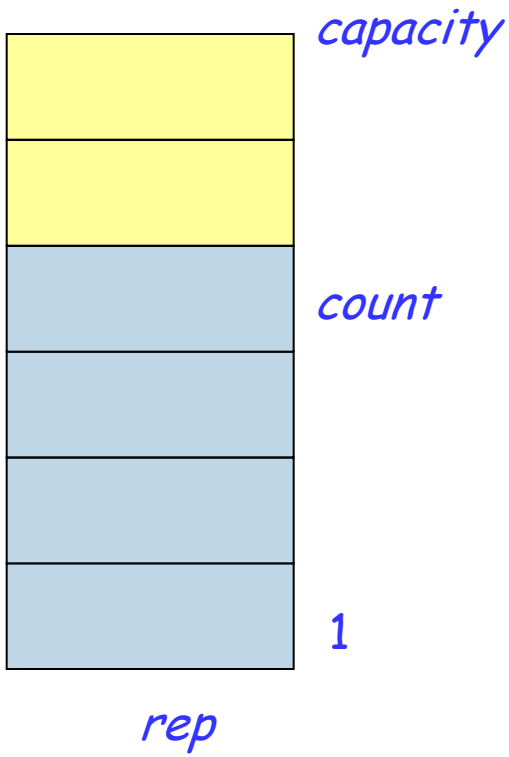
(E2) Some representation choice

(Example: <*rep, count*>)

(E3) A set of subprograms (routines) and attributes, each implementing one of the functions of the ADT specification (E1) in terms of chosen representation (E2)

(Example: routines *put, remove, item, empty, new*)

# A choice of stack representation




(array_up)

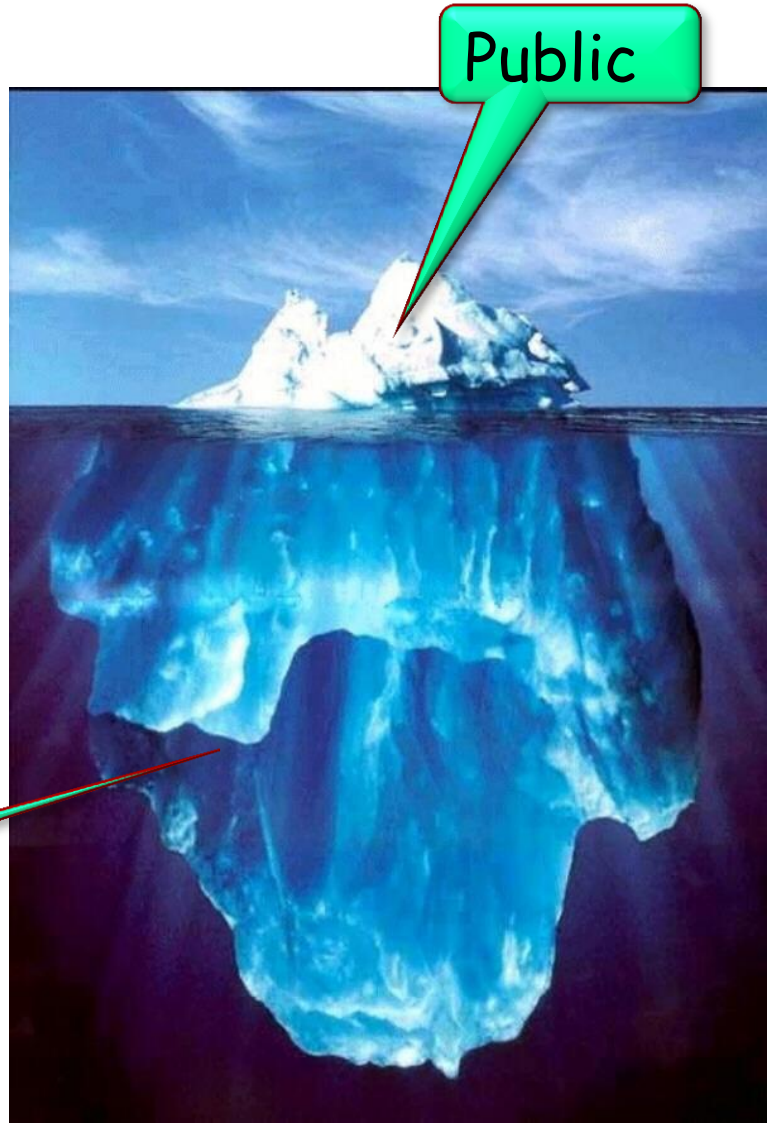
"Push" operation:

*count* := *count* + 1

*rep* [*count*] := *x*

# Information hiding

The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules



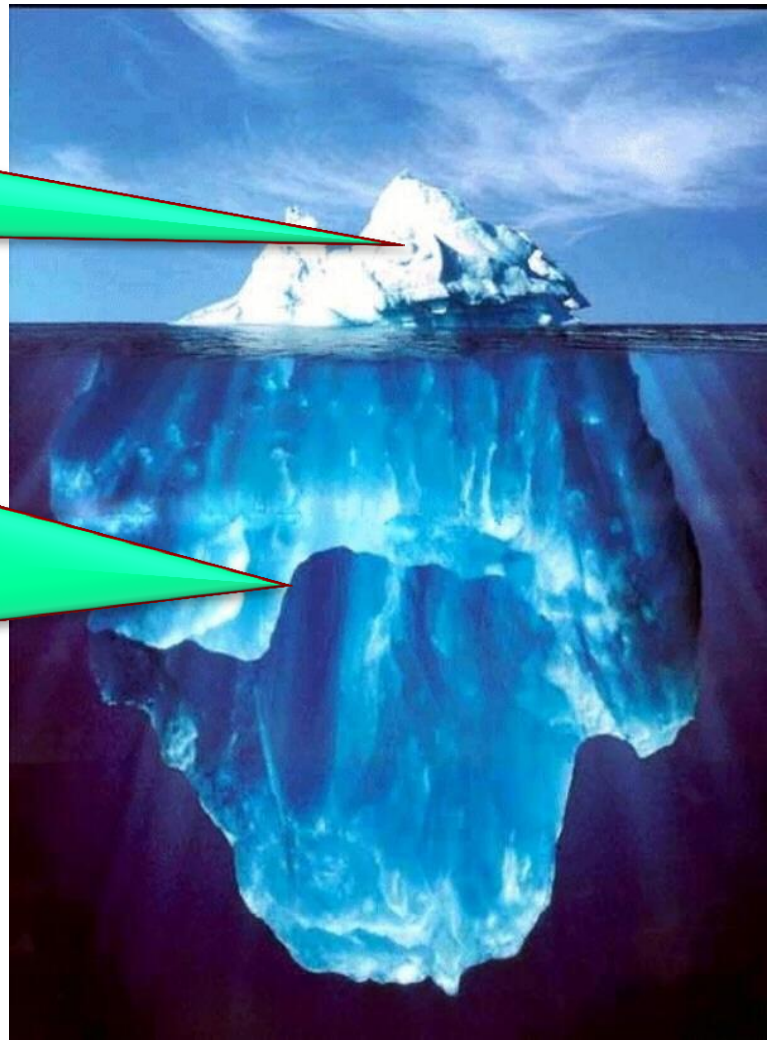
# Applying ADTs to information hiding

**Public part:**

- ADT specification
  (*E1*)

**Secret part:**

- Choice of representation
  (*E2*)
- Implementation of functions by features
  (*E3*)



# Object technology: A first definition

Object-oriented software construction is the software architecture method that bases the structure of systems on the types of objects they handle — not on "the" function they achieve

# A more precise definition

Object-oriented software construction is the construction of software systems as structured collections of (possibly partial) abstract data type implementations

# The fundamental structure: the class

Merging of the notions of module and type:

- Module = Unit of decomposition: set of services
- Type = Description of a set of run-time objects ("instances" of the type)

The connection:

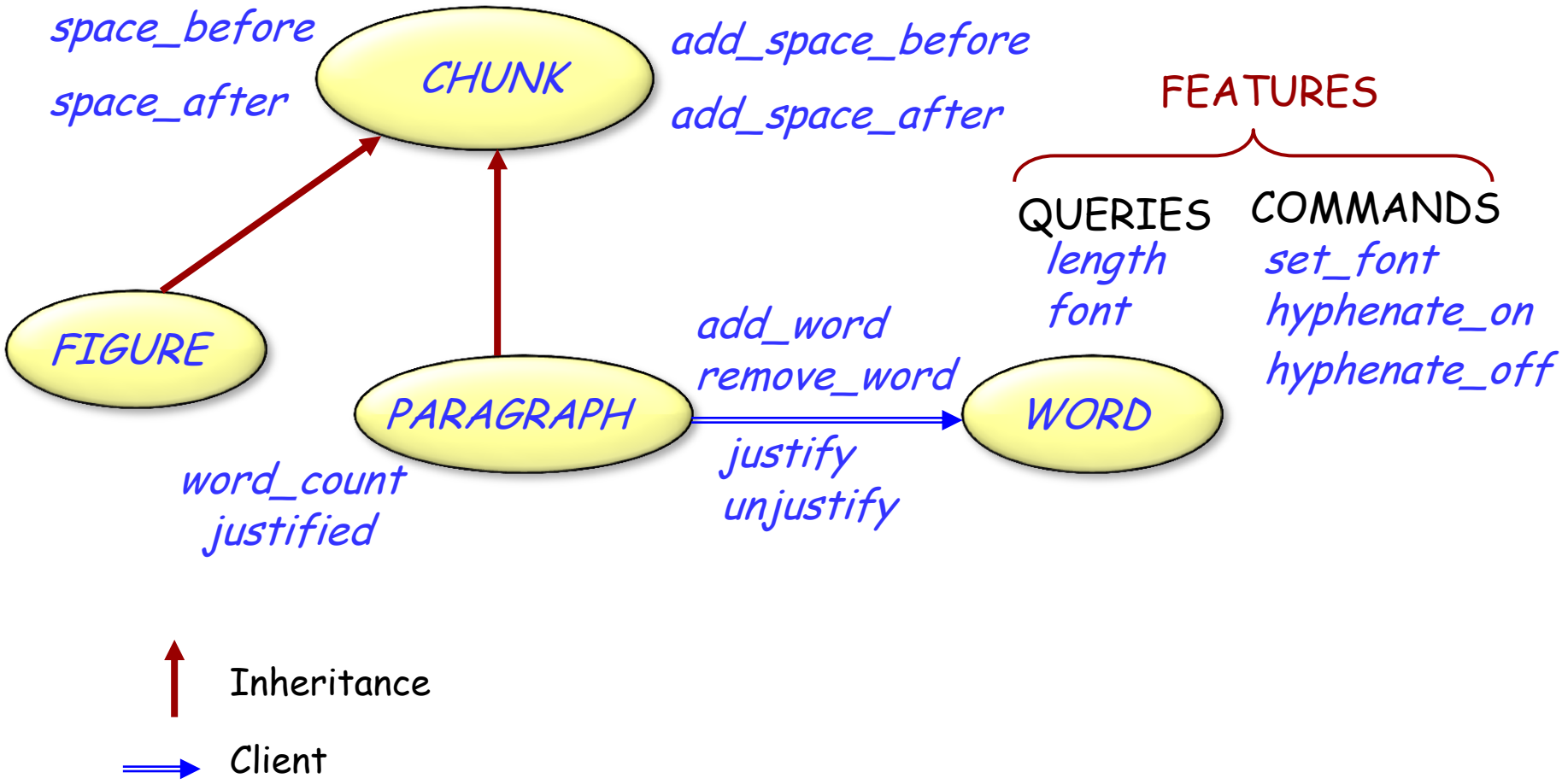
- The services offered by the class, viewed as a module, are the operations available on the instances of the class, viewed as a type.

# Class relations

Two relations:

- Client
- Heir

# Overall system structure



# Bounded stacks

Types:

$$BSTACK[G]$$

Functions (Operations):

$$put : BSTACK[G] \times G \nrightarrow BSTACK[G]$$
$$remove : BSTACK[G] \nrightarrow BSTACK[G]$$
$$item : BSTACK[G] \nrightarrow G$$
$$empty : BSTACK[G] \rightarrow BOOLEAN$$
$$new : BSTACK[G]$$
$$capacity : BSTACK[G] \rightarrow INTEGER$$
$$count : BSTACK[G] \rightarrow INTEGER$$

$$full : BSTACK[G] \rightarrow BOOLEAN$$

# Television station example

Source: OOSC

```
class SCHEDULE feature
    segments: LIST [SEGMENT]
end
```


91

# Schedules

```
note
    description :
        " 24-hour TV schedules"
deferred class SCHEDULE feature

    segments: LIST [SEGMENT]
            -- Successive segments
        deferred
        end

    air_time : DATE is
            -- 24-hour period
            -- for this schedule
        deferred
        end
```

```
    set_air_time (t: DATE)
            -- Assign schedule to
            -- be broadcast at time t.
        require
            t.in_future
        deferred
        ensure
            air_time = t
        end
    print
            -- Produce paper version.
        deferred
        end
end
```


92

# Contracts

Feature precondition: condition imposed on the rest of the world

Feature postcondition: condition guaranteed to the rest of the world

Class invariant: Consistency constraint maintained throughout on all instances of the class


93

# Why contracts

Specify semantics, but abstractly!

(Remember basic dilemma of requirements:

- Committing too early to an implementation
  Overspecification!

- Missing parts of the problem
  Underspecification!

)


94

# Segment

```
note
    description :
      "Individual fragments of a schedule"
deferred class SEGMENT feature
    schedule : SCHEDULE deferred end
            -- Schedule to which
            -- segment belongs
    index: INTEGER deferred end
            -- Position of segment in
            -- its schedule
    starting_time, ending_time :
            INTEGER deferred end
            -- Beginning and end of
            -- scheduled air time
    next: SEGMENT deferred end
            -- Segment to be played
            -- next, if any
```

```
sponsor: COMPANY deferred end
            -- Segment's principal sponsor

rating: INTEGER deferred end
            -- Segment's rating (for
            -- children's viewing etc.)

    ... Commands such as change_next,
    set_sponsor, set_rating omitted ...

Minimum_duration: INTEGER = 30
    -- Minimum length of segments,
    -- in seconds

Maximum_interval: INTEGER = 2
    -- Maximum time between two
    -- successive segments, in seconds
```


95

# Segment (continued)

```
invariant
    in_list: (1 <= index) and (index <= schedule.segments.count)
    in_schedule: schedule.segments.item (index) = Current
    next_in_list: (next /= Void ) implies
                    (schedule.segments.item (index + 1) = next)
    no_next_iff_last: (next = Void) = (index = schedule.segments.count)
    non_negative_rating: rating >= 0
    positive_times: (starting_time > 0 ) and (ending_time > 0)
    sufficient_duration:
          ending_time - starting_time >= Minimum_duration
    decent_interval :
          (next.starting_time) - ending_time <= Maximum_interval
end
```


96

# Commercial

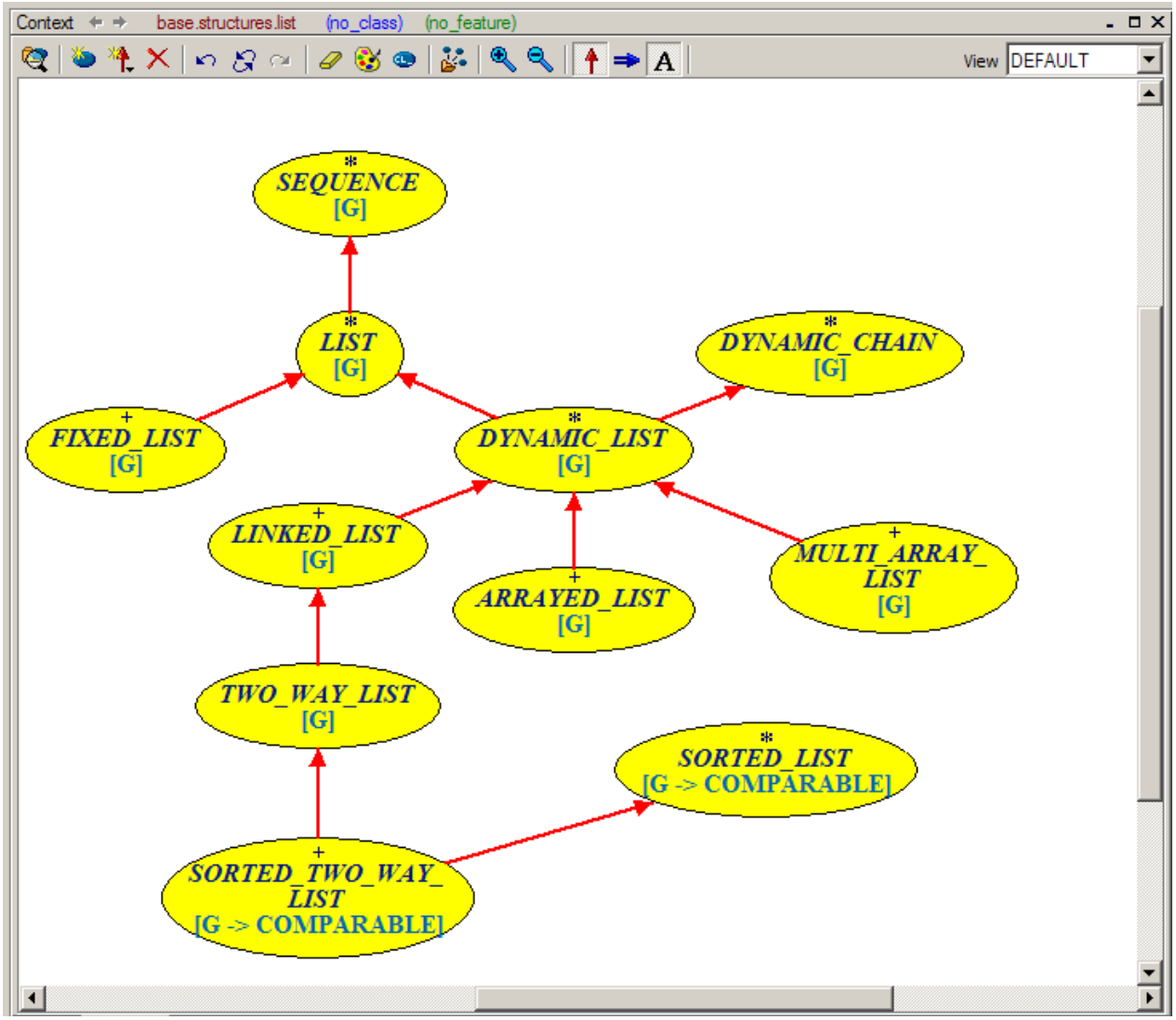
```
note
    description: "Advertizing segment"
deferred class COMMERCIAL inherit
    SEGMENT
        rename sponsor as advertizer end
feature
    primary: PROGRAM deferred
        -- Program to which this
        -- commercial is attached
    primary_index: INTEGER deferred
        -- Index of primary
```

```
set_primary (p: PROGRAM)
        -- Attach commercial to p.
    require
        program_exists: p /= Void
        same_schedule: p.schedule = schedule
        before:
            p.starting_time <= starting_time
    deferred
    ensure
        index_updated:
            primary_index = p.index
        primary_updated: primary = p
    end
```

```
invariant
    meaningful_primary_index: primary_index = primary.index
    primary_before: primary.starting_time <= starting_time
    acceptable_sponsor: advertizer.compatible (primary.sponsor)
    acceptable_rating: rating <= primary.rating
end
```

# Diagrams: UML, BON

## Text-Graphics Equivalence





98

# O-O analysis process

Identify abstractions

- New
- Reused

Describe abstractions through interfaces, with contracts

Look for more specific cases: use inheritance

Look for more general cases: use inheritance, simplify

Iterate on suppliers

At all stages keep structure simple and look for applicable contracts


99

# Your turn! Describe this in an O-O way

Consider a small library database with the following transactions:

1. Check out a copy of a book. Return a copy of a book.
2. Add a copy of a book to the library. Remove a copy of a book from the library.
3. Get the list of books by a particular author or in a particular subject area.
4. Find out the list of books currently checked out by a particular borrower.
5. Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers.

Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints:

- All copies in the library must be available for checkout or be checked out.
- No copy of the book may be both available and checked out at the same time.
- A borrower may not have more than a predefined number of books checked out at one time.

*Practical advice*

Take advantage of O-O techniques from the requirements stage on

Use contracts to express semantic properties

*Practical advice*

Write ADT specifications for delicate parts of the system requirements

# Bounded stacks (continued)

Preconditions:

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

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

*put* (*s*: *BSTACK* [*G*]) **require not** *full* (*s*)

Axioms: For all *x*: *G*, *s*: *BSTACK* [*G*]

*item* (*put* (*s*, *x*)) = *x*

*remove* (*put* (*s*, *x*)) = *s*

*empty* (*new*)

**not** *empty* (*put* (*s*, *x*))

*full* = (*count* = *capacity*)

*count* (*new*) = 0

*count* (*put* (*s*, *x*)) = *count* (*s*) + 1

*count* (*remove* (*s*)) = *count* (*s*) - 1