Software Architecture

Bertrand Meyer
Reading assignment for next week

- OOSC2:
  - Chapter 3: Modularity
  - Chapter 6: Abstract Data Types
Lecture 3: Abstract Data Types
Abstract Data Types (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 analysis and design method:
  - Process-based decomposition: classical (routines)
  - Object-oriented decomposition
Arguments for using objects

- **Reusability**: Need to reuse whole data structures, not just operations
- **Extendibility, Continuity**: Objects remain more stable over time.

Diagram:
- Employee information
- Hours worked
- Produce Paychecks
- Paychecks
Object-oriented software construction is the approach to system structuring that bases the architecture of software systems on the types of objects they manipulate — 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 design

- 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)
A stack, concrete object

```
\text{\textcolor{red}{x}}
```

(array_up)

```
capacity
\textcolor{red}{x}
count
1
```

representation

“Push” $x$ on stack $\text{\textit{representation}}$:

\begin{align*}
\text{\textcolor{red}{count}} & := \text{\textcolor{red}{count}} + 1 \\
\text{\textit{representation}}[\text{\textcolor{red}{count}}] & := x
\end{align*}
A stack, concrete object

(array_up)

(capacity)

(count)

(1)

(representation)

("Push" \(x\) on stack \textit{representation}:

\(\text{count} := \text{count} + 1\)

\(\text{representation}[\text{count}] := x\)

(array_down)

(representation)

(free)

(1)

("Push" \(x\) on stack \textit{representation}:

\(\text{representation}[\text{free}] := x\)

\(\text{free} := \text{free} - 1\)
A stack, concrete object

“Push” \( x \) on stack representation:

\[
\text{representation}[\text{count}] := x
\]
\[
\text{count} := \text{count} + 1
\]

“Push” \( x \) on stack representation:

\[
\text{representation}[\text{free}] := x
\]
\[
\text{free} := \text{free} - 1
\]

“Push” operation:

\[
\text{new}(n)
\]
\[
n.\text{item} := x
\]
\[
n.\text{previous} := \text{head}
\]
\[
\text{head} := n
\]
Stack: An Abstract Data Type (ADT)

- **Types:**
  - \( \text{STACK} \ [G] \)
  - \( G \): Formal generic parameter

- **Functions (Operations):**
  - \( \text{put}: \text{STACK} \ [G] \times G \rightarrow \text{STACK} \ [G] \)
  - \( \text{remove}: \text{STACK} \ [G] \rightarrow \text{STACK} \ [G] \)
  - \( \text{item}: \text{STACK} \ [G] \rightarrow G \)
  - \( \text{empty}: \text{STACK} \ [G] \rightarrow \text{BOOLEAN} \)
  - \( \text{new}: \text{STACK} \ [G] \)
Using functions to model operations

\[ \text{put } (s, x) = s' \]
Reminder: Partial functions

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

- Example from elementary mathematics:
  - inverse: $\mathbb{R} \rightarrow \mathbb{R}$, such that
    $$\text{inverse} \ (x) = \frac{1}{x}$$
The STACK ADT (continued)

- **Preconditions:**
  
  \[
  \begin{align*}
  \text{remove} & \ (s : \text{STACK} \ [G]) \ \text{require not empty} \ (s) \\
  \text{item} & \ (s : \text{STACK} \ [G]) \ \text{require not empty} \ (s)
  \end{align*}
  \]

- **Axioms:** For all \( x : G, s : \text{STACK} \ [G] \)
  
  \[
  \begin{align*}
  \text{item} \ (\text{put} \ (s, x)) & = x \\
  \text{remove} \ (\text{put} \ (s, x)) & = s \\
  \text{empty} \ (\text{new}) & \\
  \quad \text{(or: empty} \ (\text{new}) & = \text{True}) \\
  \text{not empty} \ (\text{put} \ (s, x)) & \\
  \quad \text{(or: empty} \ (\text{put} \ (s, x)) & = \text{False})
  \end{align*}
  \]
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.
value = item (remove (put (remove (put (put (remove (put (put (new, x8), x7), x6)), item (remove (put (put (new, x5), x4)))))), x2)), x1)))
Expressed differently

\[
\text{value} = \text{item (remove (put (remove (put (put (put (put (put (new, x8), x7), x6)), item (remove (put (put (new, x5), x4))), x2), x1)))}
\]

- \(s1 = \text{new}\)
- \(s2 = \text{put (put (put (s1, x8), x7), x6)}\)
- \(s3 = \text{remove (s2)}\)
- \(s4 = \text{new}\)
- \(s5 = \text{put (put (s4, x5), x4)}\)
- \(s6 = \text{remove (s5)}\)
- \(y1 = \text{item (s6)}\)
- \(s7 = \text{put (s3, y1)}\)
- \(s8 = \text{put (s7, x2)}\)
- \(s9 = \text{remove (s8)}\)
- \(s10 = \text{put (s9, x1)}\)
- \(s11 = \text{remove (s10)}\)
- \(\text{value} = \text{item (s11)}\)
value = item (  
    remove (  
        put (  
            remove (  
                put (  
                    put (  
                        remove (  
                            put (put (put (new, x8), x7), x6)  
                        , item (  
                            remove (  
                                put (put (new, x5), x4)  
                            )  
                        , x2)  
                    )  
                , x1)  
            )  
        )  
    )  
)
value = item (remove (put (remove (put (remove (put (put (put (new, x8), x7), x6), item (remove (put (put (new, x5), x4), x2), x1), x1), x2), x1), x1), x1), x1)
value = item (remove (put (remove (put (put (put (new, x8), x7), x6) , item (remove (put (put (new, x5), x4) ) , x2) ) , x1) ) )

Stack 1

x7
x8
Expression reduction

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

Stack 1
Expression reduction

\[
value = \text{item} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{put} (\text{put} (\text{new}, x8), x7), x6) ) \\
\text{item} (\text{remove} (\text{put} (\text{put} (\text{new}, x5), x4) \\
\text{x2}) ) \\
\text{x1}) ) \\
) )
\]

Stack 1

Chair of Software Engineering

Software Architecture
Expression reduction

\[
value = \text{item (}
    \text{remove (}
        \text{put (}
            \text{remove (}
                \text{put (}
                    \text{put (}
                        \text{remove (}
                            \text{put (}
                                \text{put (}
                                    \text{new }, x8), x7), x6)
                        )}
                    )}
                )}
            )}
        )}
    )
\]
Expression reduction

\[
value = item (\text{remove (put (remove (put (put (new, x8), x7), x6) , item (remove (put (put (new, x5), x4) , x2) , x1) , x2) , x1)})
\]
Expression reduction

\[
\begin{align*}
value & = \text{item} ( \\
& \quad \text{remove} ( \\
& \quad \quad \text{put} ( \\
& \quad \quad \quad \text{remove} ( \\
& \quad \quad \quad \quad \text{put} ( \\
& \quad \quad \quad \quad \quad \text{put} ( \\
& \quad \quad \quad \quad \quad \quad \text{remove} ( \\
& \quad \quad \quad \quad \quad \quad \quad \text{put} ( \\
& \quad \quad \quad \quad \quad \quad \quad \quad \text{put} ( \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{remove} ( \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{put} ( \text{put} (\text{new}, x8), x7), x6) \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{item} ( \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{remove} ( \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{put} (\text{put}(\text{new}, x5), x4) \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{item} ( \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{remove} ( \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{put} (\text{new}, x2), x1) \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{put} (\text{new}, x1), x0) \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{put} (\text{new}, x0), x0)
\end{align*}
\]
value = item (remove (put (remove (put (put (remove (put (put (put (put (new, x8), x7), x6) put (put (new, x5), x4) ) ) ) ) ) x2), x1) ) )

Stack 1
Stack 2
value = item
  remove (put (remove (put (put (new, x8), x7), x6)
      remove (put (new, x5), x4)
      )
      x2)
  )
  x1)
Expression reduction

\[
\text{value} = \text{item} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{new}, x8), x7), x6)) , x5), x4)) , x2) , x1))
\]
value = item ( remove ( put ( remove ( put ( remove ( put ( put (new, x8), x7), x6) , item ( remove ( put (new, x5), x4) ) , x2) , x1) ) ) )

Stack 1
Stack 2
Expression reduction

\[
\text{value} = \text{item} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{put} (\text{new}, x8), x7), x6) \; , \; \text{item} (\text{remove} (\text{put} (\text{put} (\text{new}, x5), x4) \; , \; x2) \; , \; x2) \; , \; x1) \; , \; x1) \; , \; x1) \; , \; x1)
\]

Stack 1

Stack 2

x2
x5
x7
x8
Expression reduction

\[
\text{value} = \text{item} \left( \text{remove} \left( \text{put} \left( \text{remove} \left( \text{put} \left( \text{put} \left( \text{new}, x_8 \right), \ x_7 \right), \ x_6 \right) \right) \right), \ \text{item} \left( \text{remove} \left( \text{put} \left( \text{put} \left( \text{new}, x_5 \right), \ x_4 \right) \right) \right), \ x_2 \right), \ \text{put} \left( \text{put} \left( \text{new}, x_8 \right), \ x_7 \right), \ x_6 \right) \right) \right) \right) \right) \right)
\]
value = item (remove (put (remove (put (remove (put (put (new, x8), x7), x6) , item (remove (put (put (new, x5), x4) ), x2) , x1) ), x5), x7), x8) , x7) , x6)
Expression reduction

value = item (remove (put (remove (put (remove (put (put (put (new, x8), x7), x6)
      , item (remove (put (put (new, x5), x4)
      )
      , x2)
      , x1)
      ))
      , x8))
      , x7)
      , x5)
      Stack 1                  Stack 2
Expressed differently

\[
\text{value} = \text{item} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{put} (\text{put} (\text{put} (\text{new}, x_8), x_7), x_6)), \text{item} (\text{remove} (\text{put} (\text{put} (\text{new}, x_5), x_4))), x_2), x_1))
\]

- \( s_1 = \text{new} \)
- \( s_2 = \text{put} (\text{put} (s_1, x_8), x_7), x_6) \)
- \( s_3 = \text{remove} (s_2) \)
- \( s_4 = \text{new} \)
- \( s_5 = \text{put} (\text{put} (s_4, x_5), x_4) \)
- \( s_6 = \text{remove} (s_5) \)
- \( y_1 = \text{item} (s_6) \)
- \( s_7 = \text{put} (s_3, y_1) \)
- \( s_8 = \text{put} (s_7, x_2) \)
- \( s_9 = \text{remove} (s_8) \)
- \( s_{10} = \text{put} (s_9, x_1) \)
- \( s_{11} = \text{remove} (s_{10}) \)
- \( \text{value} = \text{item} (s_{11}) \)
An operational view of the expression

\[
value = \text{item} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{put} (\text{put} (\text{put} \text{new}, x8), x7), x6), \text{item} (\text{remove} (\text{put} (\text{put} \text{new}, x5), x4))), x2), x1))
\]
Properties of an ADT specification

- **Consistent**
  
The axioms do not lead to contradiction

  “Only the truth”

- **Complete**
  
The axioms cover all needed properties

  “All the truth”
Completeness

- Cannot be ascertained in general: it would have to be relative to some higher-level specification of the system, for which the same problem arises.

- Useful notion in practice: sufficient completeness.
Sufficient completeness

An ADT specification for a type $T$ is **sufficiently complete** if and only if any correct “query Expression” may be reduced, through application of the axioms, to a form not involving $T$. 
“Correct” ADT expression

An expression of which we can prove that all arguments to ADT functions satisfy the precondition if any

\[
\text{put} (\text{put} (\text{new}, x5), x4) \quad \text{-- Correct}
\]

\[
\text{remove} (\text{put} (\text{put} (\text{new}, x5), x4)) \quad \text{-- Correct}
\]

\[
\text{item} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{put} (\text{put} (\text{put} (\text{put} (\text{put} (\text{put} (\text{put} (\text{new}, x8), x7), x6)), \text{item} (\text{remove} (\text{put} (\text{put} (\text{new}, x5), x4))))), x2)), x1)))) \quad \text{-- Correct}
\]

\[
\text{remove} (\text{new}) \quad \text{-- Not correct}
\]
Three kinds of functions in specification of an ADT $T$:

- **Creators:**
  \[ \text{OTHER} \rightarrow T \]
  e.g. *new*

- **Queries:**
  \[ T \times \ldots \rightarrow \text{OTHER} \]
  e.g. *item, empty*

- **Commands:**
  \[ T \times \ldots \rightarrow T \]
  e.g. *put, remove*
Query expression

An expression in which the outermost function is a query

Examples (correct expressions):

\[
\text{item (put (new, x5))}
\]
\[
\text{item (remove (put (put (new, x5), x4))))}
\]
\[
\text{item (remove (put (remove (put (put (remove (put (put (put (put (new, x8), x7), x6)),
\text{item (remove (put (put (new, x5), x4))))), x2)), x1))))}
\]

But not:

\[
\text{new}
\]
\[
\text{put (new, x5)}
\]
\[
\text{remove (put (new, x5))}
\]
Sufficient completeness

An ADT specification for a type $T$ is **sufficiently complete** if and only if any correct “query Expression” may be reduced, through application of the axioms, to a form not involving $T$. 
Exercise

Prove that the specification of stacks is sufficiently complete.
Stack: An Abstract Data Type

- Types:
  \[ \text{STACK} [G] \]
  -- \( G \): Formal generic parameter

- Functions (Operations):
  \( \text{put}: \text{STACK} [G] \times G \rightarrow \text{STACK} [G] \)
  \( \text{remove}: \text{STACK} [G] \rightarrow \text{STACK} [G] \)
  \( \text{item}: \text{STACK} [G] \rightarrow G \)
  \( \text{empty}: \text{STACK} [G] \rightarrow \text{BOOLEAN} \)
  \( \text{new}: \text{STACK} [G] \)
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: \(<representation, count>\).)
  - (E3) A set of subprograms (routines) and attributes, each implementing one of the functions of the ADT specification (E1) in terms of the chosen representation (E2).
    (Example: routines put, remove, item, empty, new.)
A choice of stack representation

“Push” operation:

\[ \text{count} := \text{count} + 1 \]

\[ \text{representation}[\text{count}] := x \]
Application to information hiding

Public part:
ADT specification \((E_1)\)

Secret part:
- Choice of representation \((E_2)\)
- Implementation of functions by features \((E_3)\)
Object technology: A first definition

- Object-oriented software construction is the approach to system structuring that bases the architecture of software systems on the types of objects they manipulate — not on “the” function they achieve.
Object technology: More precise definition

- Object-oriented software construction is the construction of software systems as structured collections of (possibly partial) abstract data type implementations.
Classes: The fundamental structure

- 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

CHUNK

FIGURE

PARAGRAPH

WORD

FEATURES

QUERIES

COMMANDS

add_word
remove_word
justy
unjusty

add_space_before
add_space_after

set_font
hyphenate_on
hyphenate_off

length
font

Inheritance

Client

Chair of Software Engineering

Software Architecture
deferred class
   
   COUNTER

feature
   
   item: INTEGER is
      -- Counter value
      
      deferred end
   
   up is
      -- Increase item by 1.
      
      deferred ensure
      item = old item + 1
      end

   down is
      -- Decrease item by 1.
      
      deferred ensure
      item = old item - 1
      end

invariant
   item >= 0
end
End of lecture 3