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](image.png)
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

`x` (array_up)

```
capacity
  x

  count

1

representation
```

“Push” `x` on stack `representation`:

```
representation [count] := x
count := count + 1
```
A stack, concrete object

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

\[
\text{representation}[\text{count}] := x \\
\text{count} := \text{count} + 1
\]
A stack, concrete object

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

\[
\begin{align*}
\text{representation}[\text{count}] & := x \\
\text{count} & := \text{count} + 1
\end{align*}
\]

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

\[
\begin{align*}
\text{representation}[\text{free}] & := x \\
\text{free} & := \text{free} - 1
\end{align*}
\]

“Push” operation:

\[
\begin{align*}
new(n) \\
n.\text{item} & := x \\
n.\text{previous} & := \text{last} \\
\text{head} & := n
\end{align*}
\]
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 $\rightarrow\nRightarrow$, is a function that may not be defined for all possible arguments.

- Example from elementary mathematics:
  - inverse: $\mathbb{R} \nRightarrow \mathbb{R}$, such that

\[
\text{inverse} (x) = 1 / x
\]
The STACK ADT (continued)

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

- **Axioms:** For all \( x: G, s: \text{STACK} [G] \)
  
  \[
  \text{item (put (} s, x\text{)) = } x
  \]
  
  \[
  \text{remove (put (} s, x\text{)) = } s
  \]
  
  \[
  \text{empty (new)}
  \]
  
  \[
  \text{(or: empty (new) = True)}
  \]
  
  \[
  \text{not empty (put (} s, x\text{))}
  \]
  
  \[
  \text{(or: empty (put (} s, x\text{)) = False)}
  \]
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

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

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

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

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

Stack 1

Stack 1

```plaintext
x7
x8
```
Expression reduction

\[
\text{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))

, 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

\[
\text{value} = \text{item} ( \!
\text{remove} ( \!
  \text{put} ( \!
  \text{remove} ( \!
    \text{put} ( \!
      \text{put} ( \!
        \text{put} ( \!
          \text{remove} ( \!
            \text{put} ( \!
              \text{put} ( \!
                \text{put} ( \!
                  \text{new}, x8), x7), x6) \!
            ) \!
          ) \!
        ) \!
      ) \!
    ) \!
  ) \!
) \!
\text{Stack 1} \quad \text{Stack 2}
\]
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) 
        ) , x7) 
      ) , item 
  ) , item 
)
Expression reduction

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

Stack 1  Stack 2

```
x7
x8
```

```
x4
x5
```
Expression reduction

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

Stack 1

Stack 2

x7
x8
x4
x5
Expression reduction

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

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

Stack 1  Stack 2

x2
x5
x7
x8
Expression reduction

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

Stack 1

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

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

\[
\begin{align*}
\text{value} &= \text{item} \left( \text{remove} \left( \text{put} \left( \text{remove} \left( \text{put} \left( \text{put} \left( \text{remove} \left( \text{put} \left( \text{put} \left( \text{new}, x_8 \right), x_7 \right), x_6 \right) \right), x_7 \right), x_6 \right) \right), x_6 \right) \right), x_6 \right) \\
&\quad \left( \text{item} \left( \text{remove} \left( \text{put} \left( \text{put} \left( \text{new}, x_5 \right), x_4 \right) \right), x_2 \right), x_1 \right) \right)
\end{align*}
\]

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

\[
\text{value} = \text{item} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{put} (\text{put} (\text{put} (\text{put} (\text{new}, x8), x7), x6), x5, x4)))), x2), x1))
\]
Three forms of functions in the 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`

Sufficiently complete specification: a “Query Expression” of the form:

\[ f (...) \]

where $f$ is a query, may be reduced through application of the axioms to a form not involving $T$
Stack: An Abstract Data Type

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

- Functions (Operations):

  \[ put: STACK [G] \times G \rightarrow STACK [G] \]
  \[ remove: STACK [G] \rightarrow STACK [G] \]
  \[ item: STACK [G] \rightarrow G \]
  \[ empty: STACK [G] \rightarrow BOOLEAN \]
  \[ new: STACK [G] \]
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:

\[
\begin{align*}
\text{count} & := \text{count} + 1 \\
\text{representation}[\text{count}] & := x
\end{align*}
\]
Application 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 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-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

add_space_before
add_space_after

FIGURE

space_before
space_after

PARAGRAPH

word_count
justified

add_word
remove_word

justify
unjustify

WORD

FEATURES

QUERIES
length
font

COMMANDS
set_font
hyphenate_on
hyphenate_off

Inheritance

Client

Chair of Software Engineering

Software Architecture
A very deferred class

defered class
  \textit{COUNTER}

feature
  \textit{item}: \textsc{integer} is
  \texttt{deferred}
  \texttt{end}

up is
  \texttt{deferred}
  \texttt{ensure}
  \texttt{item} = \texttt{old item} \+ 1
  \texttt{end}

down is
  \texttt{deferred}
  \texttt{ensure}
  \texttt{item} = \texttt{old item} \- 1
  \texttt{end}

invariant
  \texttt{item} \texttt{>= 0}
  \texttt{end}
End of lecture 3