Object-Oriented Software Construction

Bertrand Meyer
Reading assignment

- OOSC2
  - Chapter 10: Genericity
Lecture 4:
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

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

“Push” operation:
\[
\begin{align*}
  \text{free} & := \text{free} - 1 \\
  \text{new}(n) & := \text{x} \\
  n.\text{previous} & := \text{last} \\
  \text{head} & := n
\end{align*}
\]
A stack, concrete object

“Push” operation:
\[ \text{count} := \text{count} + 1 \]
\[ \text{representation}[\text{count}] := x \]

“Push” operation:
\[ \text{free} := \text{free} - 1 \]
\[ \text{representation}[\text{free}] := x \]

“Push” operation:
\[ \text{new}(n) \]
\[ n.\text{item} := x \]
\[ n.\text{previous} := \text{last} \]
\[ \text{head} := n \]
Stack: An abstract data type

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

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

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

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

- Example from elementary mathematics:
  - inverse: \( \mathbb{R} \not
  \rightarrow \mathbb{R} \), such that
  \[
  \text{inverse} \ (x) = 1 / x
  \]
The STACK ADT (cont’d)

- **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)`
    (or: `empty (new) = True`)
  - `not empty (put (s, x))`
    (or: `empty (put (s, x)) = False`)

```
put ([ ], [ ] ) = [ ]
```

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

- $s_1 = new$
- $s_2 = put (put (s_1, x8), x7), x6)$
- $s_3 = remove (s_2)$
- $s_4 = new$
- $s_5 = put (put (s_4, x5), x4)$
- $s_6 = remove (s_5)$
- $y_1 = item (s_6)$
- $s_7 = put (s_3, y_1)$
- $s_8 = put (s_7, x2)$
- $s_9 = remove (s_8)$
- $s_{10} = put (s_9, x1)$
- $s_{11} = remove (s_{10})$
- $value = item (s_{11})$
Expression reduction (1/10)

value = item (  
  remove (  
    put (  
      remove (  
        put (  
          put (  
            remove (  
              put (put (new, x8), x7), x6)  
            , item (  
              remove (  
                put (put (new, x5), x4)  
              , x2)  
            , x1)  
          )  
        )  
      )  
    )  
  )  
)
Expression reduction (2/10)

\[
\text{value} = \text{item} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{new}, x_8), x_7), x_6) ) ) ) ) ) , x_2), x_1)
\]
Expression reduction (3/10)

\[ \text{value} = \text{item} ( \text{remove} ( \text{put} ( \text{remove} ( \text{put} ( \text{put} ( \text{new}, x_8), x_7), x_6) \text{, item} ( \text{remove} ( \text{put} ( \text{new}, x_5), x_4) \text{, } x_2) \text{, } x_1) \text{, } x_2) \text{)} \text{, } x_1) \text{)} \]
value = item (  
  remove (  
    put (  
      remove (  
        put (  
          put (  
            remove (  
              put (put (new, x8), x7), x6)  
            , item (  
              remove (  
                put (put (new, x5), x4)  
              )  
            )  
          )  
        )  
      )  
    )  
  )  
)
Expression reduction (5/10)

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

value = item (remove (put (remove (put (put (new, x8), x7), x6))
               , item (remove (put (new, x5), x4))
               , x2))
               , x1)
Expression reduction (7/10)

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

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

, x2)

, x1)

) ) ) ) ) )

Stack 3
Expression reduction (8/10)

\[ value = item ( \) \]
\[ \text{remove (} \) \]
\[ \text{put (} \) \]
\[ \text{remove (} \) \]
\[ \text{put (} \) \]
\[ \text{remove (} \) \]
\[ \text{put (put (put (new, x8), x7), x6)} \]
\[ , item (} \]
\[ \text{remove (} \) \]
\[ \text{put (put (new, x5), x4)} \]
\[ , x2) \]
\[ , x1) \]
\[ ) \]
Expression reduction (9/10)

\[
\text{value} = \text{item(}
\text{put(}
\text{remove(}
\text{put(}
\text{remove(}
\text{put(}
\text{remove(}
\text{put(}
\text{put(}
\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)}\)
\text{), x2)}\)
\text{), x1)}\)
\text{)})}
\]

Stack 3
value = item(
  remove(
    put(
      remove(
        put(
          remove(
            put (put (put (new, x8), x7), x6)
        )
      )
    )
  )
)
)

⇒ value = x5
Expressed differently

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

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

```
value = item (remove (put (remove (put (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: $\text{OTHER} \rightarrow T$  
    e.g. $\text{new}$
  - Queries: $T \times \ldots \rightarrow \text{OTHER}$  
    e.g. $\text{item, empty}$
  - Commands: $T \times \ldots \rightarrow T$  
    e.g. $\text{put, remove}$

- Sufficiently complete specification: a “Query Expression” of the form:
  
  $f (\ldots)$

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

- **Functions (Operations):**
  - **put:** \( \text{STACK } [G] \times G \rightarrow \text{STACK } [G] \)
  - **remove:** \( \text{STACK } [G] \leftrightarrow \text{STACK } [G] \)
  - **item:** \( \text{STACK } [G] \leftrightarrow G \)
  - **empty:** \( \text{STACK } [G] \rightarrow \text{BOOLEAN} \)
  - **new:** \( \text{STACK } [G] \)
Abstract data types provide an ideal basis for modularizing software.

- Identify every module with an implementation of an abstract data type, i.e. the description of a set of objects with a common interface.

- The interface is defined by a set of operations (implementing the functions of the ADT) constrained by abstract properties (the axioms and preconditions).

- The module consists of a representation for the abstract data type and an implementation for each of the operations. Auxiliary operations may also be included.
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:

\[ count := count + 1 \]
\[ \text{representation} [\text{count}] := x \]
Application to information hiding

Secret part:
• Choice of representation (E2)
• Implementation of functions by features (E3)

Public part:
ADT specification (E1)
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.
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
- space_before
- space_after
- add_space_before
- add_space_after

FIGURE
- word_count
- justified?

PARAGRAPH
- add_word
- remove_word
- justify
- unjustify

WORD
- length
- font
- set_font
- hyphenate_on
- hyphenate_off

FEATURES

QUERIES
COMMANDS

Inheritance

Client
deferred class COUNTER

feature

item: INTEGER is deferred end

-- Counter value

up is
-- Increase item by 1.
defered
ensure

item = old item + 1
end

down is
-- Decrease item by 1.
defered
ensure

item = old item – 1
end

invariant

item >= 0

end
End of lecture 4