Object-Oriented Software Construction

Bertrand Meyer

Lecture 3:

Abstract Data Types
Reading assignment

- OOSC2
  - Chapter 10: Genericity
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 showing the relationship between Employee information, Hours worked, Produce Paychecks, and Paychecks]
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.
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

```
(count := count + 1
representation [count] := x

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

(new (n)
 n.item := x
 n.previous := last
 head := n
```
A stack, concrete object

```
\text{"Push" operation:}
\text{\quad count} := \text{count} + 1
\text{\quad representation[\text{count}] := x}

\text{"Push" operation:}
\text{\quad representation[\text{free}] := x}
\text{\quad free} := \text{free} - 1

\text{"Push" operation:}
\text{\quad new (n)}
\text{\quad n.item} := x
\text{\quad n.previous} := \text{last}
\text{\quad head} := n
```

\text{Chair of Software Engineering}
Stack: An abstract data type

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

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

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

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

- Example from elementary mathematics:
  - inverse: $\mathbb{R} \leftrightharpoons \mathbb{R}$, such that
    \[
    \text{inverse} \ (x) = \frac{1}{x}
    \]
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`)

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

- s1 = new
- s2 = 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)
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) 
          )  
        )  
      )  
    )  
  )  
)
value = item (remove (put (remove (put (remove (put (put (new, x8), x7), x6) , item (remove (put (put (new, x5), x4) ) , x2) , x1) ) , x4) ) , x5) )
value = item (remove (put (remove (put (put (put (new, x8), x7), x6) put (item (remove (put (put (new, x5), x4)) , x2)))) , x1))))
value = item ( remove ( put ( remove ( put ( put (new, x8), x7), x6) ) item ( remove ( put (put (new, x5), x4) ) , x2) , x1) ) Stack 1 Stack 2
value = item ( remove ( put ( remove ( put ( remove ( put ( put ( put (new, x8), x7), x6) ) ) ) ) ) ) 
\[ x1, x2 \] 
Stack 1
\[ x5 \] 
Stack 2
Expression reduction (7/10)

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

Stack 3

x5
x7
x8
value = item (  
  remove (  
    put (  
      remove (  
        put (  
          remove (  
            put (put (put (new, x8), x7), x6)  
          , item (  
            remove (  
              put (put (new, x5), x4)  
            , x2)  
          , x1)  
        )  
      )  
    )  
  )  
)
Expression reduction (9/10)

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

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

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

- Three forms of functions in the specification of an ADT $T$:
  - Creators:
    $$\text{OTHER} \rightarrow T$$
    e.g. $\textit{new}$
  - Queries:
    $$T \times \ldots \rightarrow \text{OTHER}$$
    e.g. $\textit{item}, \textit{empty}$
  - Commands:
    $$T \times \ldots \rightarrow T$$
    e.g. $\textit{put}, \textit{remove}$

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

  where $f$ is a well-formed 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):
  - \( \text{put}: \text{STACK} \ [G] \times G \rightarrow \text{STACK} \ [G] \)
  - \( \text{remove}: \text{STACK} \ [G] \leftarrow \leftrightarrow \text{STACK} \ [G] \)
  - \( \text{item}: \text{STACK} \ [G] \leftrightarrow G \)
  - \( \text{empty}: \text{STACK} \ [G] \rightarrow \text{BOOLEAN} \)
  - \( \text{new}: \text{STACK} \ [G] \)
ADT and software architecture

- 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:

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

\[ \text{representation}[\text{count}] := x \]
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
deferred class \textit{COUNTER}

feature

\textit{item}: INTEGER is deferred end

\hspace{1cm}-- Counter value

\textit{up} is

\hspace{1cm}-- Increase \textit{item} by 1.

\hspace{1cm}deferred

\hspace{1cm}ensure

\hspace{1cm}\textit{item} = old \textit{item} + 1

\hspace{1cm}end

\textit{down} is

\hspace{1cm}-- Decrease \textit{item} by 1.

\hspace{1cm}deferred

\hspace{1cm}ensure

\hspace{1cm}\textit{item} = old \textit{item} - 1

\hspace{1cm}end

\textbf{invariant}

\hspace{1cm}\textit{item} \geq 0

end
End of lecture 3