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

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.

Arguments for using objects

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

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.

Theoretical basis

- The main issue: How to describe program objects (data structures):
  - Completely
  - Unambiguously
  - Without overspecifying?
    - (Remember information hiding)

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

A stack, concrete object

```
<table>
<thead>
<tr>
<th>capacity</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“Push” operation: count := count + 1
representation [count] := x

<table>
<thead>
<tr>
<th>free</th>
<th>representation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“Push” operation: free := free + 1
representation [free] := x

<table>
<thead>
<tr>
<th>prev</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“Push” operation: new (v)
   n.item := x
   n.previous := last
   last := n
```
A stack, concrete object

- capacity
- count
- representation

- free
- previous
- head

"Push" operation:
- `count := count + 1`
- `representation[count] := x`

Using functions to model operations

```
put (s, x) = s',
```

Stack: An abstract data type

- Types:
  - `STACK [G]`  
    -- `G`: Formal generic parameter

- Functions (Operations):
  - `item: STACK [G] ↦ G`
  - `empty: STACK [G] → BOOLEAN`
  - `new: STACK [G]`

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: ℝ ↦ ℝ`, such that
    - `inverse (x) = 1 / x`
The STACK ADT (cont’d)

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

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

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: \( \text{put} \) becomes a partial function.

Formal stack expressions

\[
\text{value} = \text{item} \ (\text{remove} \ (\text{put} \ (\text{remove} \ (\text{put} \ (\text{put} \ (\text{put} \ (\text{put} \ (\text{put} \ (\text{put} \ (\text{put} \ (\text{new}, x8), x7), x6)), x5), x4))), x3)), x2)), x1)))
\]

Expressed differently

\[
\text{value} = \text{item} \ (\text{remove} \ (\text{put} \ (\text{remove} \ (\text{put} \ (\text{put} \ (\text{put} \ (\text{put} \ (\text{put} \ (\text{put} \ (\text{put} \ (\text{new}, x8), x7), x6)), x5), x4))), x3)), x2)), x1)))
\]

- \( g1 = \text{new} \)
- \( x2 = \text{put} \ (\text{new}, x2), x1) \)
- \( x3 = \text{remove} \ (x2) \)
- \( g1 = \text{new} \)
- \( x4 = \text{put} \ (\text{new}, x4), x3) \)
- \( x5 = \text{remove} \ (x2) \)
- \( x2 = \text{new} \)
- \( x6 = \text{put} \ (\text{new}, x6), x5) \)
- \( x7 = \text{remove} \ (x2) \)
- \( x3 = \text{new} \)
- \( x8 = \text{put} \ (\text{new}, x8), x7) \)
- \( x4 = \text{remove} \ (x2) \)
- \( x1 = \text{new} \)
- \( x9 = \text{put} \ (\text{new}, x9), x8) \)
- \( x7 = \text{remove} \ (x2) \)
- \( x5 = \text{new} \)
- \( x6 = \text{put} \ (\text{new}, x6), x5) \)
- \( x4 = \text{remove} \ (x2) \)
Expression reduction (9/10)

```plaintext
value = Item
```

Expression reduction (10/10)

```plaintext
value = x5
```

Expressed differently

```plaintext
value = Item [remove (put (remove (put (remove (put (put (new, x8), x7), x6), x5)))]
```

An operational view of the expression

```plaintext
value = Item [remove (put (remove (put (put (remove (put (put (new, x8), x7), x6)))]
```

```plaintext
x8 x7 x6 x5 x4
s3 s2 s1 (empty)
```

```plaintext
x2 x1
x5 s5 s6
```

```plaintext
x7
```

```plaintext
x8
```

```plaintext
s7 = (s9, s11)
```

```plaintext
value = Item [remove (put (remove (put (put (new, x8), x7), x6)))]
```
Sufficient completeness

- Three forms of functions in the specification of an ADT \( T \):
  - Creators:
    \[ \text{OTHER} \to T \]  e.g. \text{new}
  - Queries:
    \[ T \times \ldots \to \text{OTHER} \]  e.g. \text{item, empty}
  - Commands:
    \[ T \times \ldots \to T \]  e.g. \text{put, 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 \( T \).

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.

Stack: An abstract data type

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

- Functions (Operations):
  - \text{put}: \text{STACK} [G] \times G \to \text{STACK} [G]
  - \text{remove}: \text{STACK} [G] \leftrightarrow \text{STACK} [G]
  - \text{item}: \text{STACK} [G] \leftrightarrow G
  - \text{empty}: \text{STACK} [G] \to \text{BOOLEAN}
  - \text{new}: \text{STACK} [G]

Implementing an ADT

- Three components:
  1. The ADT’s specification: functions, axioms, preconditions. (Example: stacks.)
  2. Some representation choice. (Example: \text{<representation, count>}).
  3. 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 \text{put, remove, item, empty, new.})
A choice of stack representation

- "Push" operation:
  - `count := count + 1`
  - `representation[count] := x`

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.

Application to information hiding

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

A very deferred class

```plaintext
defefined class COUNTER
feature
  item: INTEGER is deferred end
  -- Counter value
  up is deferred
  ensure
    item = old item + 1
end
  down is deferred
  ensure
    item = old item - 1
end
  invariant
    item >= 0
end
```