Programming in the large

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]
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" \( x \) on stack \( \text{representation} \):

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

```
[array_up]
  x
  |
  |
  count := x
  representation

[array_down]
  x
  |
  |
  free := free - 1
  representation

"Push" x on stack representation:
  representation [count] := x
  count := count + 1

"Push" x on stack representation:
  representation [free] := x
  free := free - 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*:

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

“Push” operation:

\[
\begin{align*}
\text{new}(n) \\
n.\text{item} & := x \\
n.\text{previous} & := \text{last} \\
\text{head} & := n
\end{align*}
\]
Stack: An Abstract Data Type (ADT)

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

- **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

\[ \text{put}(\text{s}, \text{x}) = \text{s}' \]
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: $\mathbb{R} \leftrightarrow \mathbb{R}$, 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 \( x: G, s: \text{STACK} \ [G] \)
  \[
  \text{item} \ (\text{put} \ (s, x)) = x
  \]
  \[
  \text{remove} \ (\text{put} \ (s, x)) = s
  \]
  \[
  \text{empty} \ (\text{new})
  \]
  (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: `put` becomes a partial function.
End of lecture 3