Abstract Data Types (ADTs),
Classes, and Objects

Readings: OOSC2 Chapters 6, 7, 8

Abstract Data Types (ADTs)

• Given a problem, you are required to filter out irrelevant details.
• The result is an abstract data type (ADT), whose interface consists of a list of (unimplemented) operations.

Supplier’s Obligations:
- Implement all operations
- Choose the “right” data structure (DS)

Client’s Benefits:
- Correct output
- Efficient performance

The internal details of an implemented ADT should be hidden.

Why Java Interfaces Unacceptable ADTs (1)

It is useful to have:
- A generic collection class where the homogeneous type of elements are parameterized as E.
- A reasonably intuitive overview of the ADT.

Building ADTs for Reusability

• ADTs are reusable software components e.g., Stacks, Queues, Lists, Dictionaries, Trees, Graphs
• An ADT, once thoroughly tested, can be reused by:
  - Suppliers of other ADTs
  - Clients of Applications
• As a supplier, you are obliged to:
  - Implement given ADTs using other ADTs (e.g., arrays, linked lists, hash tables, etc.)
  - Design algorithms that make use of standard ADTs
• For each ADT that you build, you ought to be clear about:
  - The list of supported operations (i.e., interface)
  - The interface of an ADT should be more than method signatures and natural language descriptions:
    - How are clients supposed to use these methods? [preconditions]
    - What are the services provided by suppliers? [postconditions]
    - Time (and sometimes space) complexity of each operation
**Why Java Interfaces Unacceptable ADTs (2)**

Methods described in a *natural language* can be *ambiguous*:

```java
set
E set(int index, E element)
```

Replaces the element at the specified position in this list with the specified element (optional operation).

---

**Why Eiffel Contract Views are ADTs (1)**

```eiffel
class interface ARRAYED_CONTAINER
feature -- Commands
assign_at (i: INTEGER; s: STRING)
-- Change the value at position 'i' to 's'.
require
valid_index: 1 <= i <= count
ensure
size_unchanged: imp.count = (old imp.twin).count
item_assigned: imp[i] = s
others_unchanged: \( \forall 1 \leq j \leq \text{imp.count} : j \neq i \rightarrow \text{imp}[j] = (\text{old imp.twin})[j] \)
end

count: INTEGER
```

---

**Uniform Access Principle (1)**

- We may implement `Point` using two representation systems:
  - The *Cartesian system* stores the *absolute* positions of \( x \) and \( y \).
  - The *Polar system* stores the *relative* position: the angle (in radian) \( \phi \) and distance \( r \) from the origin \((0,0)\).

- How the `Point` is implemented is irrelevant to users:
  - **Imp. 1**: Store \( x \) and \( y \) \([ Compute \ r \ and \ \phi \ on \ demand ]\)
  - **Imp. 2**: Store \( r \) and \( \phi \) \([ Compute \ x \ and \ y \ on \ demand ]\)

- As far as users of a `Point` object \( p \) is concerned, having a *uniform access* by always being able to call \( p.x \) and \( p.y \) is what matters, despite **Imp. 1** or **Imp. 2** being current strategy.
Uniform Access Principle (2)

class POINT
create
make_cartisian, make_polar
feature -- Public, Uniform Access to x- and y-coordinates
  x : REAL
  y : REAL
end

• A class Point declares how users may access a point: either get its x coordinate or its y coordinate.
• We offer two possible ways to instantiating a 2-D point:
  ○ make_cartisian (nx: REAL; ny: REAL)
  ○ make_polar (nr: REAL; np: REAL)
• Features x and y, from the client’s point of view, cannot tell whether it is implemented via:
  ○ Storage [ x and y stored as real-valued attributes ]
  ○ Computation [ x and y defined as queries returning real values ]

Uniform Access Principle (3)

Let’s say the supplier decides to adopt strategy Imp. 1.

class POINT -- Version 1
feature -- Attributes
  x : REAL
  y : REAL
feature -- Constructors
make_cartisian(nx: REAL; ny: REAL)
  do
    x := nx
    y := ny
  end
end

• Attributes x and y represent the Cartesian system
• A client accesses a point p via p.x and p.y.
  ○ No Extra Computations: just returning current values of x and y.
• However, it’s harder to implement the other constructor: the body of make_polar (nr: REAL; np: REAL) has to compute and store x and y according to the inputs nr and np.

Uniform Access Principle (4)

Let’s say the supplier decides (secretly) to adopt strategy Imp. 2.

class POINT -- Version 2
feature -- Attributes
  r : REAL
  p : REAL
feature -- Constructors
make_polar(nr: REAL; np: REAL)
  do
    r := nr
    p := np
  end
feature -- Queries
  x : REAL do Result := r × cos(p) end
  y : REAL do Result := r × sin(p) end
end

• Attributes r and p represent the Polar system
• A client still accesses a point p via p.x and p.y.
  ○ Extra Computations: computing x and y according to the current values of r and p.

Uniform Access Principle (5.1)

Let’s consider the following scenario as an example:

Note: 360° = 2π
The **Uniform Access Principle**: 

- Allows clients to use services (e.g., p.x and p.y) regardless of how they are implemented.
- Gives suppliers complete freedom as to how to implement the services (e.g., Cartesian vs. Polar).
  - No right or wrong implementation; it depends!
  - Choose for **storage** if the services are frequently accessed and their computations are expensive.
    e.g., balance of a bank involves a large number of accounts
    ⇒ Implement balance as an attribute
  - Choose for **computation** if the services are not keeping their values in sync is complicated.
    e.g., update balance upon a local deposit or withdrawal
    ⇒ Implement balance as a query
- Whether it’s storage or computation, you can always change secretly, since the clients’ access to the services is **uniform**.

---

```plaintext
test_points: BOOLEAN
local
A, X, Y: REAL
p1, p2: POINT
do
cm: (*test: two systems of points*)
A := 5; X := A × √3; Y := A
create (POINT) p1.make_cartisian (X, Y)
create (POINT) p2.make_polar (2 × A, π/2)
Result := p1.x = p2.x and p1.y = p2.y
end
```

- If strategy **Imp. 1** is adopted:
  - L8 is computationally cheaper than L9. [x and y attributes]
  - L10 requires no computations to access x and y.
- If strategy **Imp. 2** is adopted:
  - L9 is computationally cheaper than L8. [r and p attributes]
  - L10 requires computations to access x and y.

---

**Generic Collection Class: Motivation (1)**

```plaintext
class STRING_STACK
feature (NONE) -- Implementation
| imp: ARRAY[STRING]; i: INTEGER
feature -- Queries
| count: INTEGER do Result := i end
  -- Number of items on stack.
| top: STRING do Result := imp [i] end
  -- Return top of stack.
feature -- Commands
| push (v: STRING) do imp[i] := v; i := i + 1 end
  -- Add ‘v’ to top of stack.
pop do i := i - 1 end
  -- Remove top of stack.
end
```

- Does how we implement integer stack operations (e.g., top, push, pop) depends on features specific to element type STRING (e.g., at, append)? [NO!]
- How would you implement another class ACCOUNT_STACK?

---

**Generic Collection Class: Motivation (2)**

```plaintext
class ACCOUNT_STACK
feature (NONE) -- Implementation
| imp: ARRAY[ACCOUNT]; i: INTEGER
feature -- Queries
| count: INTEGER do Result := i end
  -- Number of items on stack.
| top: ACCOUNT do Result := imp [i] end
  -- Return top of stack.
feature -- Commands
| push (v: ACCOUNT) do imp[i] := v; i := i + 1 end
  -- Add ‘v’ to top of stack.
pop do i := i - 1 end
  -- Remove top of stack.
end
```

- Does how we implement integer stack operations (e.g., top, push, pop) depends on features specific to element type ACCOUNT (e.g., deposit, withdraw)? [NO!]
Generic Collection Class: Client (1.2)

As client, declaring \texttt{ss: STACK[ACCOUNT]} instantiates every occurrence of G as ACCOUNT.

```plaintext
class STACK[\# ACCOUNT]  
| feature {NONE} -- Implementation  
| imp: ARRAY[\# ACCOUNT]; i: INTEGER  
| feature -- Queries  
| count: INTEGER do Result := i end  
| top: \# ACCOUNT do Result := imp [i] end  
| feature -- Commands  
| push (v: \# ACCOUNT) do imp[i] := v; i := i + 1 end  
| pop do i := i - 1 end  
```

Generic Collection Class: Client (1.1)

As client, declaring \texttt{ss: STACK[STRING]} instantiates every occurrence of \texttt{G} as \texttt{STRING}.

```plaintext
class STACK[\# STRING]  
| feature {NONE} -- Implementation  
| imp: ARRAY[\# STRING]; i: INTEGER  
| feature -- Queries  
| count: INTEGER do Result := i end  
| top: \# STRING do Result := imp [i] end  
| feature -- Commands  
| push (v: \# STRING) do imp[i] := v; i := i + 1 end  
| pop do i := i - 1 end  
```

Generic Collection Class: Client (2)

As client, instantiate the type of \texttt{G} to be the one needed.

```plaintext
test_stacks: BOOLEAN  
local  
| ss: STACK[STRING]; ss: STACK[ACCOUNT]  
| s: STRING; a: ACCOUNT  
| do  
| ss.push("A")  
| ss.push(create ACCOUNT.make ("Mark", 200))  
| s := ss.top  
| a := ss.top  
| sa.push(create ACCOUNT.make ("Alan", 100))  
| sa.push("B")  
| a := sa.top  
| a := sa.top  
```

- L3 commits that \texttt{ss} stores \texttt{STRING} objects only.
  - L8 and L10 valid; L9 and L11 invalid.
- L4 commits that \texttt{sa} stores \texttt{ACCOUNT} objects only.
  - L12 and L14 valid; L13 and L15 invalid.
Expanded Class: Modelling

- We may want to have objects which are:
  - Integral parts of some other objects
  - Not shared among objects

  e.g., Each workstation has its own CPU, monitor, and keyword. All workstations share the same network.

Expanded Class: Programming (2)

```
class KEYBOARD ... end
class CPU ... end
class MONITOR ... end
class NETWORK ... end
class WORKSTATION
k: expanded KEYBOARD
c: expanded CPU
m: expanded MONITOR
n: NETWORK
end
```

Alternatively:

```
expanded class KEYBOARD ... end
expanded class CPU ... end
expanded class MONITOR ... end
class NETWORK ... end
class WORKSTATION
k: KEYBOARD
c: CPU
m: MONITOR
n: NETWORK
end
```

Expanded Class: Programming (3)

```
test_expanded: BOOLEAN
local
ebl, eb2: B
do
  Result := eb1.i = 0 and eb2.i = 0
  check Result end
  Result := eb1 = eb2
  check Result end
  eb2.change_i(15)
  Result := eb1.i = 0 and eb2.i = 15
  check Result end
  Result := eb1 /= eb2
  check Result end
end
```

- L5: object of expanded type is automatically initialized.
- L9 & L10: no sharing among objects of expanded type.
- L7 & L12: = between expanded objects compare their contents.

Reference vs. Expanded (1)

- Every entity must be declared to be of a certain type (based on a class).
- Every type is either referenced or expanded.
- In reference types:
  - y denotes a reference to some object
  - x := y attaches x to same object as does y
  - x = y compares references
- In expanded types:
  - y denotes some object (of expanded type)
  - x := y copies contents of y into x
  - x = y compares contents

  \[ x \sim y \]
Reference vs. Expanded (2)

**Problem:** Every published book has an author. Every author may publish more than one book. Should the author field of a book reference-typed or expanded-typed?

<table>
<thead>
<tr>
<th>reference-typed author</th>
<th>expanded-typed author</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;The Red and the Black&quot; 1840 341</td>
<td>&quot;The Red and the Black&quot; 1840 341</td>
</tr>
<tr>
<td>&quot;Life of Bismarck&quot; 1823 307</td>
<td>&quot;Life of Bismarck&quot; 1823 307</td>
</tr>
<tr>
<td>&quot;Nineteen Eighty Forty&quot; 1842</td>
<td>&quot;Nineteen Eighty Forty&quot; 1842</td>
</tr>
</tbody>
</table>

Copying Objects

Say variables \( c_1 \) and \( c_2 \) are both declared of type \( C \). [ \( c_1, c_2 : C \) ]

- There is only one attribute \( a \) declared in class \( C \).
- \( c_1.a \) and \( c_2.a \) may be of either:
  - expanded type or
  - reference type

Copying Objects: Reference Copy

- Copy the address stored in variable \( c_2 \) and store it in \( c_1 \).
- \( \Rightarrow \) Both \( c_1 \) and \( c_2 \) point to the same object.
- \( \Rightarrow \) Updates performed via \( c_1 \) also visible to \( c_2 \). [ aliasing ]

Copying Objects: Shallow Copy

- Create a temporary, behind-the-scene object \( c_3 \) of type \( C \).
- Initialize each attribute \( a \) of \( c_3 \) via reference copy: \( c_3.a := c_2.a \)
- Make a reference copy of \( c_3 \): \( c_1 := c_3 \)
- \( \Rightarrow \) \( c_1 \) and \( c_2 \) are not pointing to the same object. [ \( c_1 /= c_2 \] ]
- \( \Rightarrow \) \( c_1.a \) and \( c_2.a \) are pointing to the same object.
- \( \Rightarrow \text{Aliasing still occurs: at 1st level (i.e., attributes of } c_1 \text{ and } c_2 \)
Copying Objects: Deep Copy

Deep Copy

- Create a temporary, behind-the-scene object \( c_3 \) of type \( C \).
- **Recursively** initialize each attribute \( a \) of \( c_3 \) as follows:
  - **Base Case:** \( a \) is expanded (e.g., INTEGER). \( \Rightarrow c_3.a := c_2.a \).
  - **Recursive Case:** \( a \) is referenced. \( \Rightarrow c_3.a := c_2.a.deep_twin \).
- Make a reference copy of \( c_3 \):
  - \( c_1 := c_3 \) (\( c_1 \) and \( c_2 \) are not pointing to the same object).
  - \( c_1.a \) and \( c_2.a \) are not pointing to the same object.
  - **No aliasing** occurs at any levels.

Copying Objects: Example

Initial situation:

- \( b := a \)
- \( c := a.twin \)
- \( d := a.deep_twin \)

Result of:

- \( b := a \)
- \( c := a.twin \)
- \( d := a.deep_twin \)