Design Patterns: Singleton and Iterator

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

What are design patterns?

- Solutions to problems that arise when software is being developed within a particular context.
  - Heuristics for structuring your code so that it can be systematically maintained and extended.
  - *Caveat* : A pattern is only suitable for a particular problem.
  - Therefore, always understand *problems* before *solutions*!

Singleton Pattern: Motivation

Consider two problems:

1. Bank accounts share a set of data.

   e.g., interest and exchange rates, minimum and maximum balance, *etc.*
2. Processes are regulated to access some shared, limited resources.

Shared Data through Inheritance

Client:

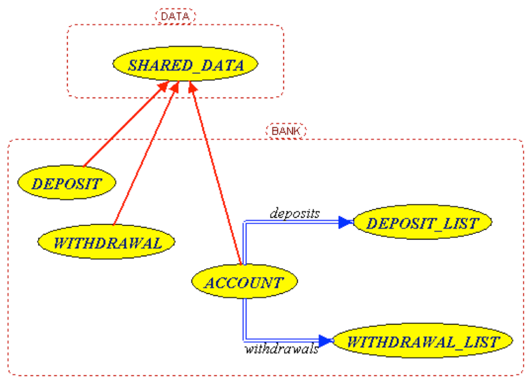
```
class DEPOSIT inherit SHARED_DATA
      ...
end

class WITHDRAW inherit SHARED_DATA
      ...
end

class ACCOUNT inherit SHARED_DATA
feature
      deposits: DEPOSIT_LIST
      withdraws: WITHDRAW_LIST
      ...
end
```

Supplier:

```
class
  SHARED_DATA
feature
  interest_rate: REAL
  exchange_rate: REAL
  minimum_balance: INTEGER
  maximum_balance: INTEGER
  ...
end
```

Problems?

Sharing Data through Inheritance: Architecture



- Irreverent features are inherited, breaking descendants' **cohesion**.
- Same set of data is duplicated as instances are created.

Sharing Data through Inheritance: Limitation

- Each instance at runtime owns a separate copy of the shared data.
- This makes inheritance *not* an appropriate solution for both problems:
  - What if the interest rate changes? Apply the change to all instantiated account objects?
  - An update to the global lock must be observable by all regulated processes.

Solution:
  - Separate notions of *data* and its *shared access* in two separate classes.
  - Encapsulate the shared access itself in a separate class.

Introducing the Once Routine in Eiffel (1.1)

```
1  class A
2  create make
3  feature -- Constructor
4    make do end
5  feature -- Query
6    new_once_array (s: STRING): ARRAY[STRING]
7        -- A once query that returns an array.
8      once
9        create {ARRAY[STRING]} Result.make_empty
10       Result.force (s, Result.count + 1)
11     end
12   new_array (s: STRING): ARRAY[STRING]
13       -- An ordinary query that returns an array.
14     do
15       create {ARRAY[STRING]} Result.make_empty
16       Result.force (s, Result.count + 1)
17     end
18 end
```

L9 & L10 executed **only once** for initialization.
L15 & L16 executed **whenever** the feature is called.

Introducing the Once Routine in Eiffel (1.2)

```
test_query: BOOLEAN
  local
    a: A
    arr1, arr2: ARRAY[STRING]
  do
    create a.make

    arr1 := a.new_array ("Alan")
    Result := arr1.count = 1 and arr1[1] ~ "Alan"
    check Result end

    arr2 := a.new_array ("Mark")
    Result := arr2.count = 1 and arr2[1] ~ "Mark"
    check Result end

    Result := not (arr1 = arr2)
    check Result end
  end
```

Introducing the Once Routine in Eiffel (1.3)

```
test_once_query: BOOLEAN
 local
   a: A
   arr1, arr2: ARRAY[STRING]
 do
   create a.make

   arr1 := a.new_once_array ("Alan")
   Result := arr1.count = 1 and arr1[1] ~ "Alan"
   check Result end

   arr2 := a.new_once_array ("Mark")
   Result := arr2.count = 1 and arr2[1] ~ "Alan"
   check Result end

   Result := arr1 = arr2
   check Result end
end
```


Introducing the Once Routine in Eiffel (2)

```
r (...): T
      once
            -- Some computations on Result
            ...
      end
```

- The ordinary **do** . . . **end** is replaced by **once** . . . **end**.
- The first time the **once** routine *r* is called by some client, it executes the body of computations and returns the computed result.
- From then on, the computed result is "cached".
- In every subsequent call to *r*, possibly by different clients, the body of *r* is not executed at all; instead, it just returns the "cached" result, which was computed in the very first call.
- **How does this help us?**
  Cache the reference to the same shared object!

Introducing the Once Routine in Eiffel (3)

- In Eiffel, the `once` routine:
  - Initializes its return value `Result` by some computation.
  - The initial computation is invoked only once.
  - Resulting value from the initial computation is cached and returned for all later calls to the once routine.
- Eiffel `once` routines are *different* from Java `static` accessors
  In Java, a `static` accessor
  - Does not have its computed return value "cached"
  - Has its computation performed ***freshly*** on every invocation
- Eiffel `once` routines are *different* from Java `static` attributes
  In Java, a `static` attribute
  - Is a value on storage
  - May be initialized via some simple expression
    e.g., `static int counter = 20;`
    but cannot be initialized via some sophisticated computation.
  - **Note**. By putting such initialization computation in a constructor, there would be a ***fresh*** computation whenever a new object is created.

Singleton Pattern in Eiffel

Supplier:

```
class BANK_DATA
create {BANK_DATA_ACCESS} make
feature {BANK_DATA_ACCESS}
  make do ... end
feature -- Data Attributes
  interest_rate: REAL
  set_interest_rate (r: REAL)
end
```

```
expanded class
  BANK_DATA_ACCESS
feature
  data: BANK_DATA
     -- The one and only access
    once create Result.make end
invariant data = data
```

Client:

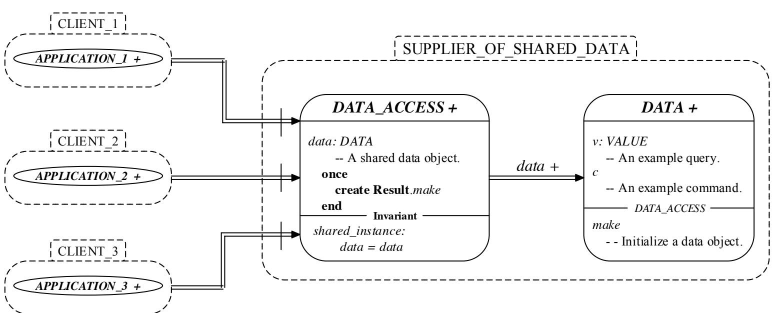
```
class
  ACCOUNT
feature
  data: BANK_DATA
  make (...)
      -- Init. access to bank data.
    local
      data_access: BANK_DATA_ACCESS
    do
      data := data_access.data
      ...
    end
end
```

Writing **create** *data.make* in client's *make* feature does not compile. Why?

Testing Singleton Pattern in Eiffel

```
test_bank_shared_data: BOOLEAN
   -- Test that a single data object is manipulated
 local
   acc1, acc2: ACCOUNT
 do
   comment("t1: test that a single data object is shared")
   create acc1.make ("Bill")
   create acc2.make ("Steve")

   Result := acc1.data ~ acc2.data
   check Result end

   Result := acc1.data = acc2.data
   check Result end

   acc1.data.set_interest_rate (3.11)
   Result := acc1.data.interest_rate = acc2.data.interest_rate
 end
```


Singleton Pattern: Architecture



Important Exercises: Instantiate this architecture to both problems of shared bank data and shared lock. Draw them in `draw.io`.

Iterator Pattern: Motivation

Supplier:

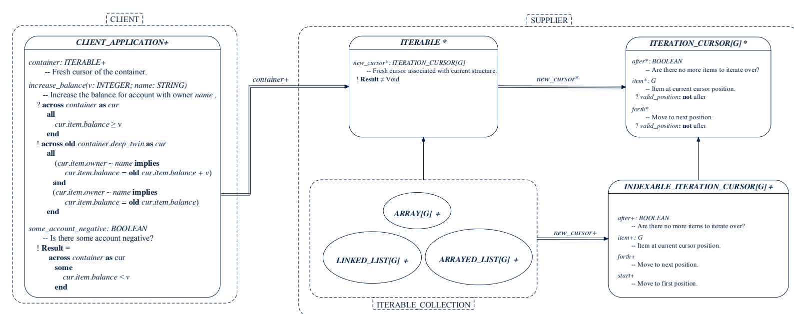
```
class
  CART
feature
  orders: ARRAY [ORDER]
end

class
  ORDER
feature
  price: INTEGER
  quantity: INTEGER
end
```

Client:

```
class
  SHOP
feature
  cart: CART
  checkout: INTEGER
   do
     from
       i := cart.orders.lower
     until
       i > cart.orders.upper
     do
       Result := Result +
         cart.orders[i].price
         *
         cart.orders[i].quantity
       i := i + 1
     end
   end
end
```

Problems?

Iterator Pattern: Architecture



Iterator Pattern: Supplier's Side

- **Information hiding**: changing the secret, internal workings of data structures should not affect any existing clients.
  e.g., changing from *ARRAY* to *LINKED_LIST* in the *CART* class
- Steps:
  1. Let the supplier class inherit from the deferred class *ITERABLE[G]*.
  2. This forces the supplier class to implement the inherited feature: *new_cursor: ITERATION_CURSOR [G]*, where the type parameter *G* may be instantiated (e.g., *ITERATION_CURSOR[ORDER]*).
     - **2.1** If the internal, library data structure is already *iterable* e.g., *imp: ARRAY[ORDER]*, then simply return *imp.new_cursor*.
     - **2.2** Otherwise, say *imp: MY_TREE[ORDER]*, then create a new class *MY_TREE_ITERATION_CURSOR* that inherits from *ITERATION_CURSOR[ORDER]*, then implement the 3 inherited features *after*, *item*, and *forth* accordingly.

Iterator Pattern: Supplier's Implementation (1)

```
class
  CART
inherit
  ITERABLE[ORDER]

...

feature {NONE} -- Information Hiding
  orders: ARRAY[ORDER]

feature -- Iteration
  new_cursor: ITERATION_CURSOR[ORDER]
    do
      Result := orders.new_cursor
    end
```

When the secrete implementation is already *iterable*, reuse it!

Iterator Pattern: Supplier's Imp. (2.1)

```
class
  GENERIC_BOOK[G]
inherit
  ITERABLE[ TUPLE[STRING, G] ]
...
feature {NONE} -- Information Hiding
  names: ARRAY[STRING]
  records: ARRAY[G]
feature -- Iteration
  new_cursor: ITERATION_CURSOR[ TUPLE[STRING, G] ]
    local
      cursor: MY_ITERATION_CURSOR[G]
    do
      create cursor.make (names, records)
      Result := cursor
    end
```

No Eiffel library support for iterable arrays ⇒ Implement it yourself!

Iterator Pattern: Supplier's Imp. (2.2)

```
class
  MY_ITERATION_CURSOR[G]
inherit
  ITERATION_CURSOR[ TUPLE[STRING, G] ]
feature -- Constructor
  make (ns: ARRAY[STRING]; rs: ARRAY[G])
    do ... end
feature {NONE} -- Information Hiding
  i: cursor_position
  names: ARRAY[STRING]
  records: ARRAY[G]
feature -- Cursor Operations
  item: TUPLE[STRING, G]
    do ... end
  after: Boolean
    do ... end
  forth
    do ... end
```

You need to implement the three inherited features: *item*, *after*, and *forth*.

Exercises

1. Draw the BON diagram showing how the iterator pattern is applied to the *CART* (supplier) and *SHOP* (client) classes.
2. Draw the BON diagram showing how the iterator pattern is applied to the supplier classes:
   - *GENERIC_BOOK* (a descendant of *ITERABLE*) and
   - *MY_ITERATION_CURSOR* (a descendant of *ITERATION_CURSOR*).

Iterator Pattern: Client's Side

Information hiding: the clients do not at all depend on *how* the supplier implements the collection of data; they are only interested in iterating through the collection in a linear manner.
Steps:

1. Obey the *code to interface, not to implementation* principle.
2. Let the client declare an attribute of type *ITERABLE[G]* (rather than *ARRAY*, *LINKED_LIST*, or *MY_TREE*).

   e.g., `cart: CART`, where *CART* inherits `ITERATBLE[ORDER]`
3. Eiffel supports, in both implementation and *contracts*, the **across** syntax for iterating through anything that's *iterable*.

Iterator Pattern: Clients using `across` for Contracts (1)

```
class
  CHECKER
feature -- Attributes
  collection: ITERABLE [INTEGER]
feature -- Queries
  is_all_positive: BOOLEAN
      -- Are all items in collection positive?
    do
      ...
    ensure
      across
        collection as cursor
      all
        cursor.item > 0
      end
  end
```

- Using **all** corresponds to a universal quantification (i.e., ∀).
- Using **some** corresponds to an existential quantification (i.e., ∃).

Iterator Pattern: Clients using `across` for Contracts (2)

```
class BANK
...
  accounts: LIST [ACCOUNT]
  binary_search (acc_id: INTEGER): ACCOUNT
      -- Search on accounts sorted in non-descending order.
    require
      across
        1 |..| (accounts.count - 1) as cursor
      all
        accounts [cursor.item].id <= accounts [cursor.item + 1].id
      end
    do
      ...
    ensure
      Result.id = acc_id
    end
```

This precondition corresponds to:

$$\forall i : INTEGER \mid 1 \leq i < accounts.count \bullet accounts[i].id \leq accounts[i+1].id$$

Iterator Pattern: Clients using `across` for Contracts (3)

```
class BANK
...
  accounts: LIST [ACCOUNT]
  contains_duplicate: BOOLEAN
      -- Does the account list contain duplicate?
    do
      ...
    ensure
        ∀i, j : INTEGER |
          1 ≤ i ≤ accounts.count ∧ 1 ≤ j ≤ accounts.count •
            accounts[i] ~ accounts[j] ⇒ i = j
    end
```

- **Exercise:** Convert this mathematical predicate for postcondition into Eiffel.
- **Hint:** Each **across** construct can only introduce one dummy variable, but you may nest as many **across** constructs as necessary.

Iterator Pattern: Clients using Iterable in Imp. (1)

```
class BANK
  accounts: ITERABLE [ACCOUNT]
  max_balance: ACCOUNT
      -- Account with the maximum balance value.
    require ??
    local
      cursor: ITERATION_CURSOR[ACCOUNT]; max: ACCOUNT
    do
      from max := accounts [1]; cursor := accounts. new_cursor
      until cursor. after
      do
        if cursor. item .balance > max.balance then
          max := cursor. item
        end
        cursor. forth
      end
    ensure ??
    end
```


Iterator Pattern: Clients using Iterable in Imp. (2)

```
1  class SHOP
2    cart: CART
3    checkout: INTEGER
4        -- Total price calculated based on orders in the cart.
5      require ??
6      local
7        order: ORDER
8      do
9        across
10         cart as cursor
11       loop
12         order := cursor. item
13         Result := Result + order.price * order.quantity
14       end
15     ensure ??
16   end
```

- Class *CART* should inherit from *ITERABLE[ORDER]*.
- **L10** implicitly declares: `cursor: ITERATION_CURSOR[ORDER]`

Iterator Pattern: Clients using Iterable in Imp. (3)

```
class BANK
  accounts: ITERABLE [ACCOUNT]
  max_balance: ACCOUNT
      -- Account with the maximum balance value.
    require ??
    local
      max: ACCOUNT
    do
      max := accounts [1]
      across
        accounts as cursor
      loop
        if cursor.item.balance > max.balance then
          max := cursor. item
        end
      end
    ensure ??
    end
```

Index (1)

Index (2)

Index (3)