

Design Pattern: Iterator



EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG



What are design patterns?

- Solutions to *recurring 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*!

3 of 22

Learning Objectives



Upon completing this lecture, you are expected to understand:

1. Motivating Problem of the Iterator Design Pattern
2. Supplier: Implementing the Iterator Design Pattern
3. Client: Using the Iterator Design Pattern
4. A Challenging Exercise (architecture & generics)

2 of 22

Iterator Pattern: Motivation (1)



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
```

Supplier:

```
class
  CART
  feature
    orders: ARRAY[ORDER]
  end
class
  ORDER
  feature
    price: INTEGER
    quantity: INTEGER
  end
```

Problems?

4 of 22

Iterator Pattern: Motivation (2)



Supplier:

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

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

Client's code must be modified to adapt to the supplier's *change on implementation*.

5 of 22

Client:

```
class
  SHOP
  feature
    cart: CART
    checkout: INTEGER
  do
    from
      cart.orders.start
    until
      cart.orders.after
    do
      Result := Result +
        cart.orders.item.price
      *
      cart.orders.item.quantity
    end
  end
end
```

Iterator Pattern: Supplier's Side



- Information Hiding Principle:**

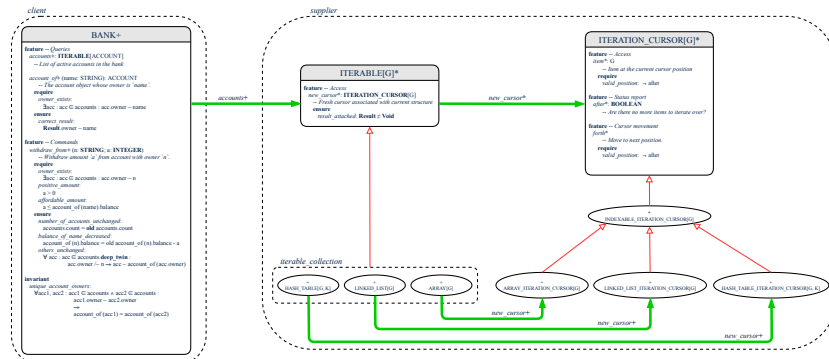
- Hide design decisions that are *likely to change* (i.e., *stable API*).
- Change of secrets* does not affect clients using the existing API.
 - e.g., changing from *ARRAY* to *LINKED_LIST* in the *CART* class

- Steps:**

- Let the supplier class inherit from the deferred class *ITERABLE[G]*.
- 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]*).
 - If the internal, library data structure is already *iterable* e.g., *imp: ARRAY[ORDER]*, then simply return *imp.new_cursor*.
 - 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.

7 of 22

Iterator Pattern: Architecture



6 of 22

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 secret implementation is already *iterable*, reuse it!

8 of 22

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
end
    
```

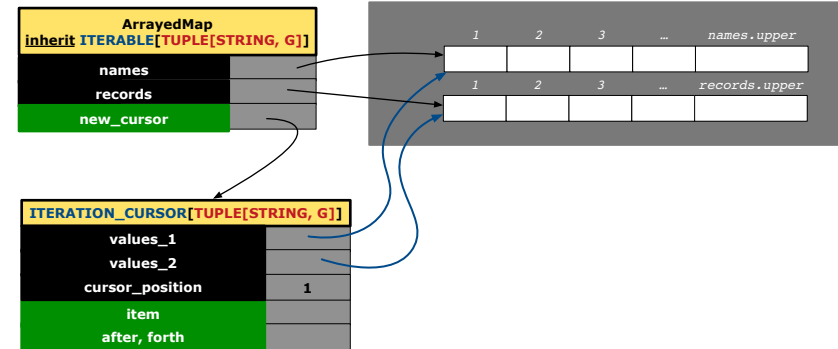
No Eiffel library support for iterable arrays ⇒ Implement it yourself!

9 of 22

Iterator Pattern: Supplier's Imp. (2.3)



Visualizing iterator pattern at runtime:



11 of 22

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
  cursor_position: INTEGER
  names: ARRAY[STRING]
  records: ARRAY[G]
feature -- Cursor Operations
  item: TUPLE[STRING, G]
  do ... end
  after: Boolean
  do ... end
  forth
  do ... end
end
    
```

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

10 of 22

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*).

12 of 22

Resources



- [Tutorial Videos on Generic Parameters and the Iterator Pattern](#)
- [Tutorial Videos on Information Hiding and the Iterator Pattern](#)
- [Tutorial on Making a Birthday Book \(implemented using HASH_TABLE\) ITERABLE](#)

13 of 22

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 **interface** type **ITERABLE[G]** (rather than **implementation** type **ARRAY**, **LINKED_LIST**, or **MY_TREE**).
e.g., `cart: CART`, where `CART` inherits `ITERABLE[ORDER]`
3. Eiffel supports, in **both** implementation and **contracts**, the **across** syntax for iterating through anything that's *iterable*.

14 of 22

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 is item
  all
    item > 0
  end
end
```

- Using **all** corresponds to a universal quantification (i.e., \forall).
- Using **some** corresponds to an existential quantification (i.e., \exists).

15 of 22

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) is i
  all
    accounts [i].id <= accounts [i + 1].id
  end
do
  ...
ensure
  Result.id = acc_id
end
```

This precondition corresponds to:

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

16 of 22

Iterator Pattern: Clients using across for Contracts (3)

```
class BANK
...
accounts: LIST [ACCOUNT]
contains_duplicate: BOOLEAN
  -- Does the account list contain duplicate?
do
...
ensure
   $\forall i, j: \text{INTEGER} \mid$ 
     $1 \leq i \leq \text{accounts.count} \wedge 1 \leq j \leq \text{accounts.count} \bullet$ 
     $\text{accounts}[i] \sim \text{accounts}[j] \Rightarrow i = j$ 
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.

17 of 22

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   do
7     across
8       cart is order
9     loop
10      Result := Result + order.price * order.quantity
11    end
12  ensure ??
13 end
```

- Class *CART* should inherit from *ITERABLE[ORDER]*.
- L10 implicitly declares `cursor: ITERATION_CURSOR[ORDER]` and does `cursor := cart.new_cursor`

19 of 22

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 cursor := accounts.new_cursor; max := cursor.item
  until cursor.after
  do
  if cursor.item.balance > max.balance then
    max := cursor.item
  end
  cursor.forth
  end
  ensure ??
end
```

18 of 22

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

```
class BANK
  accounts: LIST [ACCOUNT] -- Q: Can ITERABLE [ACCOUNT] work?
  max_balance: ACCOUNT
  -- Account with the maximum balance value.
  require ??
  local
  max: ACCOUNT
  do
  max := accounts [1]
  across
  accounts is acc
  loop
  if acc.balance > max.balance then
    max := acc
  end
  end
  ensure ??
end
```

20 of 22

Beyond this lecture ...



- Tutorial Videos on Iterator Pattern
- Exercise: Architecture & Generics

21 of 22

Index (1)



Learning Objectives

What are design patterns?

Iterator Pattern: Motivation (1)

Iterator Pattern: Motivation (2)

Iterator Pattern: Architecture

Iterator Pattern: Supplier's Side

Iterator Pattern: Supplier's Implementation (1)

Iterator Pattern: Supplier's Imp. (2.1)

Iterator Pattern: Supplier's Imp. (2.2)

Iterator Pattern: Supplier's Imp. (2.3)

Exercises

22 of 22

Index (2)



Resources

Iterator Pattern: Client's Side

Iterator Pattern:

Clients using `across` for Contracts (1)

Iterator Pattern:

Clients using `across` for Contracts (2)

Iterator Pattern:

Clients using `across` for Contracts (3)

Iterator Pattern:

Clients using `Iterable` in Imp. (1)

Iterator Pattern:

Clients using `Iterable` in Imp. (2)

23 of 22

Index (3)



Iterator Pattern:

Clients using `Iterable` in Imp. (3)

Beyond this lecture ...

24 of 22