# EECS3311 Software Design
# Summer 2015
# Lab Exercises of Week 3

Jackie Wang

**Abstract**

In this week's lab session, you are asked to do three exercises on the iterator pattern. It is critical for you to understand, and appreciate, how **information hiding** is applied here: the supplier's secret representation of the collection (e.g., *ARRAY*, *LINKED_LIST*, *etc.*) is completely hidden from the client; and clients only depend on a uniform interface that is defined in the *ITERABLE* and *ITERATION_CURSOR* classes. In the first exercise (Section 1), you will act as a client of an iterable class: use the Eiffel *across* constructs to write both contracts and implementations for iterating through items in the collection in a linear fashion. In the other two exercises, you will act as a supplier of an iterable class: implement the interface of the *ITERABLE* class. In the second exercise (Section 2), you will simply reuse the implementation from Eiffel library classes. In the second exercise (Section 3), you will develop the implementation of an iterable class on your own. Click on the heading of each section to link to its associated video.

## Contents

**1 Acting as a Client: Using the across Constructs** **1**

**2 Acting as a Supplier: an Iterable CART Class** **3**

**3 Acting as a Supplier: an Iterable GENERIC_BOOK[G] Class** **4**

## 1 Acting as a Client: Using the across Constructs

In this exercise, you will be acting as a *client* of objects that are *iterable*. More precisely, you will use the **across** for both contracts (which correspond to the universal and existential quantifiers) and implementation (which corresponds to a loop).

Consider the *ITERABLE_UTILITIES* class:

```
class
    ITERABLE_UTILITIES
create
    make
feature -- Attributes
    collection: ITERABLE [INTEGER]
feature -- Constructors
    make (new_collection: ITERABLE [INTEGER])
            -- Initialize the iterable object for processing.
        do
            collection := new_collection
        end
```

```
invariant
    all_non_negative:
        ∀item : INTEGER | item ∈ collection • item ≥ 0
```

The *ITERABLE_UTILITIES* class above provides utility functions for inquiring about an iterable object (or a collection). There are two constraints: 1) the collection stores integers only; and 2) all stored integers should be non-negative.

**Question:** How should these constraints be reflected in the *ITERABLE_UTILITIES* class?

**Task 1:** Create the *ITERABLE_UTILITIES* class above and covert the mathematical pre- and post-conditions, as well as invariants, into Eiffel using the **across** constructs.

Now, consider the two new queries *min* and *has* below.

```
feature -- Queries
    min: INTEGER
            -- Minimum value in 'collection'.
        local
            cursor: ITERATION_CURSOR [INTEGER]
        do
            cursor := collection.new_cursor
            -- Your task: Write a loop that uses this returned cursor to find the minimum.
        ensure
            result_is_minimum:
                ∀item : INTEGER | item ∈ collection • Result ≤ item
        end

    has (v: INTEGER): BOOLEAN
            -- Is there a value in 'collection' equal to 'v'?
        local
            cursor: ITERATION_CURSOR [INTEGER]
        do
            cursor := collection.new_cursor
            -- Your task: Write a loop that uses this returned cursor to find the minimum.
        ensure
            result_valid:
                Result = (∃item : INTEGER | item ∈ collection • item = v)
        end
```

**Task 2:** Add the above two queries to the *ITERABLE_UTILITIES* class by converting the mathematical pre- and post-conditions into Eiffel using the **across** constructs.

**Task 3:** As for the body implementation of these two queries, try both possibilities:

1. a **from** ... **until** ... **do** ... **end** loop

2. an **across** ... **as** ... **loop** ... **end** construct

**Task 4:** Is your implementation of the *has* feature efficient? That is, if the number of integers stored in the collection is substantial, will your loop exit as soon as the item is found? Or will it examine the entire list no matter what?

*Hint.* Use a local variable *item_found: BOOLEAN* that is used as part of the loop exit condition (i.e., part of the *until* condition), and is set *True* if the current iteration finds the item.

## 2 Acting as a Supplier: an Iterable CART Class

This exercise builds on a previous tutorial video on information hiding, and the project resulted from that exercise can be downloaded here as the starter code for this exercise. However, you are supposed to finish that exercise by yourself before attempting the current exercise. Follow these steps:

1. Create a new class *GOOD_SHOP2* whose text is copied and pasted from *GOOD_SHOP*.

2. Create a new class *GOOD_CART2* whose text is copied and pasted from *GOOD_CART*.

3. Make the class *GOOD_CART2* inherit from *ITERABLE[ORDER]*.

   **Question:** Why not inherit from *ITERABLE[G]*?

4. This will now force the inherited feature *new_cursor* to be effected (i.e., implemented) in *GOOD_CART2*.

5. Since the implementations suggested to you, i.e., *ARRAY* and *LINKED_LIST*, are both Eiffel library classes that are both already *ITERABLE*. This implies that both *ARRAY* and *LINKED_LIST* already support some concrete implementations for the *new_cursor* feature.

6. Therefore, the *new_cursor* feature in *GOOD_CART2* has a one-line implementation:

```
class
  GOOD_CART2
inherit
  ITERABLE[ORDER]
...
feature -- Iteration
  new_cursor: ITERATION_CURSOR[ORDER]
      -- A fresh cursor for iterating through orders in current cart.
    do
      Result := imp.new_cursor
    end
feature -- Implementation
  imp: ARRAY[ORDER] -- Or the type of imp can be LINKED_LIST[ORDER]
```

   **Question.** When the supplier's secret, i.e., the detailed representation of the collection of orders, changes from *ARRAY* to *LINKED_LIST*, or vice versa, will the one-line implementation for *GOOD_CART2*'s *new_cursor* feature be affected? Justify your answer.

7. Now that you have implemented all features to make the *CART* class iterable, you can use it as a client in the *GOOD_SHOP2* class:

```
class
  GOOD_SHOP2
...
feature -- Attributes
  cart: CART
feature -- Queries
  checkout: INTEGER
      -- Total price of orders in current cart.
    do
      -- Your task to complete the calculation.
    end
```

   **Question:** In the implementation body of *checkout*, you can no longer write *cart.orders*, why not?

# 3 Acting as a Supplier: an Iterable GENERIC_BOOK[G] Class

This exercise builds on a previous tutorial video on genericity, and the project resulted from that exercise can be downloaded here as the starter code for this exercise. However, you are supposed to finish that exercise by yourself before attempting the current exercise.

Recall that to implement the mappings from names to records, we use two arrays (or two linked lists). Consequently, making this suppliers class (i.e., *GENERIC_BOOK[G]*) iterable will be a more challenging exercise compared with the case of *CART* (see Section 2). This is why:

- The *new_cursor* feature is already supported in the *ARRAY* and *LINKED_LIST* classes (since they are both descendant classes of *ITERABLE*).
- But there is no implementation of the *new_cursor* feature for two arrays (or for two linked lists), unless you implement one yourself!

Follow these steps:

1. First of all, you need to consider: What should be the return type for the *new_cursor* feature in *GENERIC_BOOK[G]*, if it inherits from *ITERABLE*? It cannot be simply *new_cursor: ITERATION_CURSOR[G]*, because it will then only revel the record, but hide its associated name! That is, as clients iterate through a book object, in each iteration step they should be able to retrieve both a name and its associated record.

2. To achieve this, we introduce the *TUPLE* type in Eiffel. Each tuple object contains a list of values, each of which can be of a distinct type. For example, to declare a tuple type for name-date pairs, e.g., [“*Jim*″, 1970/03/20], [“*Jeremy*″, 1969/04/28], *etc*, we write either

```
TUPLE [STRING, DATE]
```

or

```
TUPLE [name: STRING; record: DATE]
```

Both of the above tuple types are valid. Note that member types in the first tuple type is separated by commas, whereas in the second tuple type they are separated by semi-colons. Furthermore, in the first tuple type, we can only use indices, starting from 1, to refer to tuple elements. In the second tuple type, we declare names for the members, which allows clients to refer to elements using those names. Here is an example of declaring and using tuples:

```
test_tuple: BOOLEAN
  local
    pair: TUPLE[STRING, DATE]
    pair2: TUPLE[name: STRING; record: DATE]
    d: DATE
  do
    create d.make (1970, 4, 23)
    pair := ["Jim", d]
    Result := pair[1] ~ "Jim" and pair[2] ~ d

    pair2 := ["Jim", d]
    Result := pair2.name ~ "Jim" and pair2.record ~ d
    check Result end
  end
```

3. Before making the *GENERIC_BOOK* class iterable, we first need a new class that implements the cursor for iterating through two arrays. Having introduced the *TUPLE* type, create a new class *TWO_ARRAY_ITERATION_CURSOR* which inherits from the *ITERATION_CURSOR* class:

```
class
    TWO_ARRAY_ITERATION_CURSOR[G]
inherit
    ITERATION_CURSOR[TUPLE[STRING, G]]
...
end
```

   It is important to note that we instantiate the formal generic parameter *G* in the *ITERATION_CURSOR* class by *TUPLE[STRING, G]*, meaning that the cursor is going to let clients iterate through a collection of string-record tuples.

4. Since *TWO_ARRAY_ITERATION_CURSOR* inherits from *ITERATION_CURSOR*, you will be forced to implement three inherited features that are deferred: *after*, *item*, and *forth*.

```
class
    TWO_ARRAY_ITERATION_CURSOR[G]
inherit
    ITERATION_CURSOR[TUPLE[STRING, G]]
create
    make
feature
    make (ns: ARRAY[STRING]; rs: ARRAY[G])
            -- Initialize a cursor from two arrays.
        do
            ...
        end
feature
    after: BOOLEAN
        do
            ...
        end
    item: TUPLE[STRING, G]
        do
            ...
        end
    forth
        do
            ...
        end
end
```

   **Note.** The return type of the *item* feature is a tuple type, and its members are given names that can be referenced by clients (see the Boolean test case *test_iterable_book* below).

   The implementation of the above features, in terms of two arrays, is left to you as an exercise. You might need additional attributes to keep track of the current position of the cursor. Notice that these auxiliary attribute should be hidden!

5. Having defined your own version of an iteration cursor for two arrays, now inherit the *GENERIC_BOOK* class from *ITERABLE*, which will force the inherited feature *new_cursor* to be implemented:

```
class
    GENERIC_BOOK[G]
inherit
    ITERABLE[TUPLE[STRING, G]]
...
feature
    new_cursor: ITERATION_CURSOR[TUPLE[STRING, G]]
        local
            ic: TWO_ARRAY_ITERATION_CURSOR[G]
        do
            create ic.make (names, records)
        end
```

**Question**: Contrast this *inherit* clause with the one for making the *GOOD_CART2* class iterable (Section 2). Why in one case *G* is instantiated as *ORDER*, but in the other case it is instantiated by *TUPLE[STRING, G]*?

6. How would you use an iterable book? Here is an example test case:

```
test_iterable_book: BOOLEAN
    local
        book: GENERIC_BOOK[DATE]
        today, d1, d2: DATE
        all_born_today: BOOLEAN
        pair: TUPLE[name: STRING; record: DATE]
    do
        create book.make
        create today.make_now
        create d1.make_now
        create d2.make_now
        book.add ("Jim", d1)
        book.add ("Jeremy", d2)
        Result :=
            across
                book as cursor
            all
                cursor.item [2] ~ today
            end
        check Result end

        all_born_today := true
        across
            book as cursor
        loop
            pair := cursor.item
            if pair.record /~ today then
                all_born_today := false
            end
        end
        check Result end
    end
```