EECS3311 Fall 2020
Software Design
Instructor: Chen-Wei Wang
Practice Questions Solutions
2020-12-11

Name (Print): ____________________

Prism Login ____________________

Signature ____________________

This question set contains 10 pages (including this cover page) and 4 problems.

Check to see if any pages are missing.

Enter **all** requested information on the top of this page before you start the questions, and put your initials on the top of every page, in case the pages become separated.

This is a closed book test, and **no** data sheets are permitted.

Attempt **all** questions. Answer each question in the boxed space provided.

The following rules apply:

- **NO QUESTIONS DURING THE TEST.**
- **If a question is ambiguous or unclear, then please write your assumptions and proceed to answer the question.**
- **Write in valid Eiffel syntax** wherever required.
- Where descriptive answers are requested, use complete sentences and paragraphs. Be precise and concise.
- **Organize your work**, in a reasonably neat and coherent way, in the space provided. Work scattered all over the page without a clear ordering will receive very little credit.
- **Mysterious or unsupported answers will not receive credit**. A correct answer, unsupported by calculations or explanation will receive no credit; an incorrect answer supported by substantially correct calculations and explanations might still receive partial credit.
- All answers must appear in the boxed areas in this booklet. In the worst case, if you feel you need more space, use the back of the pages; **clearly indicate when you have done this**.

Do not write in this table which contains your raw mark scores.

Problem	Points	Score
1	10	
2	15	
3	15	
4	40	
Total:	80	

Writing Unit Tests for Contracts

1. Consider the following Eiffel code for: **1)** the contract view of the *ACCOUNT* class; and **2)** its (client) test class:

```
class ACCOUNT
create make
feature
    balance: INTEGER
    credit: INTEGER
    make (new_credit: INTEGER)
        ensure
            balance = 0 and credit = new_credit
    withdraw (a: INTEGER)
            -- Withdraw amount 'a'.
        require
            positive_amount: a > 0
            enough_balance: balance + credit - a >= 0
        ensure
            balance = old balance - a and credit = old credit
invariant
    positive_credit: credit > 0
    balance_not_too_low: balance + credit >= 0
end
```

```
class
    TEST_ACCOUNT
inherit
    ES_TEST
create
    make
feature
    make
        do
            -- Add tests here.
        end
feature
    -- Define test features here.
end
```

You can assume that the two invariant constraints are correct: the credit is always positive, and the balance may go negative, provided that it is not smaller than $-credit$ (i.e., $0 - credit$).

(a) You are required to write a test case which verifies that the current precondition for the *withdraw* feature in class *ACCOUNT* is <u>not too weak</u>. Consider the following use case: say an account object *acc* is created with an initial credit value of 10, and a subsequent call of *acc.withdraw*(11) should cause a precondition violation with the corresponding tag.

Your have **two** tasks (both written in valid Eiffel syntax): **1)** Convert this use case to a feature *test_withdraw_precondition_not_too_weak*; and **2)** Write the line of code, appearing in the *make* feature of class *TEST_ACCOUNT*, that adds this feature as a test case.

Hint: You should first decide whether to implement this feature as a command or a query.

Solution:

```
test_withdraw_precondition_not_too_weak
    local
        acc: ACCOUNT
    do
        create acc.make (10)
        acc.withdraw (11)
    end
add_violation_case_with_tag (
    "enough_balance", agent test_withdraw_precondition_not_too_weak)
```

[of 5 points]

(b) You are required to write a test case which verifies that the current precondition for the *withdraw* feature in class *ACCOUNT* is not too strong. Consider the following use case: say an account object *acc* is created with an initial credit value of 10, and a subsequent call of *acc.withdraw*(10) should *not* cause any precondition violations.

Your have **two** tasks (both written in valid Eiffel syntax): **1)** Convert this use case to a feature *test_withdraw_precondition_not_too_strong*; and **2)** Write the line of code, appearing in the *make* feature of class *TEST_ACCOUNT*, that adds this feature as a test case.

Hint: You should first decide whether to implement this feature as a command or a query.

Solution:

```
test_withdraw_precondition_not_too_strong: BOOLEAN
  local
    acc: ACCOUNT
  do
    create acc.make (10)
    acc.withdraw (10)
    Result := acc.balance = −10 and acc.credit = 10
  end
add_boolean_case (agent test_withdraw_precondition_not_too_strong)
```

[of 5 points]

Information Hiding and the Iterator Pattern

2. Consider the following three classes:

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

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

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

Each shop object contains a cart of orders. The *checkout* feature calculates the total amount that is due for the current cart of orders.

(a) The above design violates the principle of **information hiding**. How?
Your answer should clearly explain **all** of the following:

- who the supplier is and who the client is;
- the problem on the supplier side; and
- the problem on the client side.

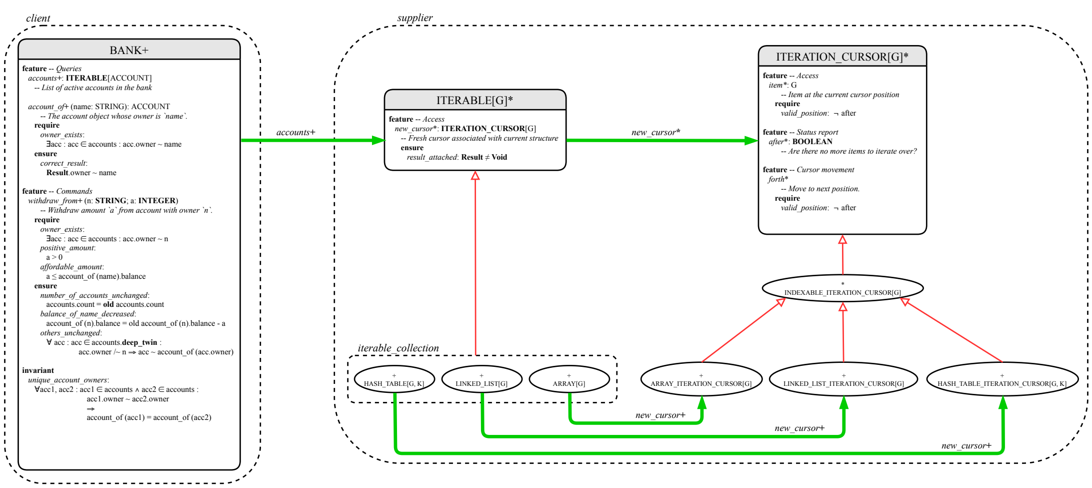
Solution: The client is the *SHOP* class and the supplier is the *CART* class. The problem of the supplier is that it does not hide the implementation secret (i.e., *orders* as a linked list) that is subject to changes. The problem of the client is that its code relies on the part of supplier that is subject to changes (i.e., *start*, *after*, and *forth* features from *LINKED_LIST*).

[of 5 points]

(b) One way to resolve the above problem is to apply the iterator pattern to it. Your task is to draw a BON diagram detailing the new design after the iterator pattern is implemented. Your diagram must include **all** of the following:

- all necessary deferred and effective classes and features;
- all necessary client-supplier and inheritance relations;
- an *expanded* view of the *SHOP* class showing how the *checkout* feature is changed.

Solution:



[of 10 points]

Genericity: Design

3. Figure 1 shows the design (omitting contracts) of a book that stores people's records of any types, implemented using two arrays. It is assumed that the stored records are indexed by the set of names (i.e., an existing name maps to a single record, whereas an existing record might be associated with multiple names).

Consider the following Eiffel test case for the above design of book (Figure 1). The feature *day_of_the_week* is a query defined in the *DATE* class, which returns an integer value, ranging

```
class BOOK
create make
feature
    make
        -- Initialize an empty book.
    add (r: ANY; n: STRING)
        -- Add an entry to the book.
    get (n: STRING): ANY
        -- The associated record of person with name 'n'.
    find (r: ANY): ARRAY[STRING]
        -- Names of people whose associated records are equal to 'r'.
feature {NONE} -- Implementation
    names: ARRAY[STRING]
    records: ARRAY[ANY]
end
```

Figure 1: Design of A Book of Any Records

from 1 to 7, representing the current date's day of the week (1 for Sunday, 7 for Saturday, and so on).

```
test_book: BOOLEAN
    local
        b: BOOK
        birthday: DATE
        phone_number: STRING
    do
        create b.make
        create phone_number.make_from_string ("416−967−1010")
        b.add (phone_number, "Jared")
        create birthday.make (1975, 4, 10)
        b.add (birthday, "David")
        Result := b.get ("David").day_of_the_week = 4
    end
```

Figure 2: A test case for the book

(a) The above Eiffel code (Figure 2) does not compile, which line? Why?

Solution: Line 12: feature *get* returns a value of type *ANY*, but is used as a *DATE*.

[of 3 points]

(b) Write, in valid Eiffel syntax, the fix for making the identified line in part (a) compile.
Hint: Consider an explicit *cast* via the *attached* expression in Eiffel.

Solution:

```
if attached {DATE} b.get ("David") as david_birthday then
    Result := david_birthday.day_of_the_week = 4
end
```

[of 3 points]

(c) Improve the design shown in Figure 1 (page6) by creating a new class *GENERIC_BOOK*. This new class declares a generic parameter for the type of stored records. In your answer, show both the class declaration and feature signatures (do not worry about implementations or contracts).

Solution:

```
class GENERIC_BOOK[G]
feature
    make
    add (r: G; n: STRING)
    get (n: STRING): G
    find (r: G): ARRAY[STRING]
feature {NONE} -- Implementation
    names: ARRAY[STRING]
    records: ARRAY[G]
end
```

[of 3 points]

(d) Consider the above test case in Figure 2 (page6). Say the client decides to have the local variable *b* as a book that stores dates only. How should the declaration in Line 3 be changed using a generic book?

Solution:

```
b: GENERIC_BOOK[DATE]
```

[of 3 points]

(e) After the fix from part (d) on Figure 2 (page6), the code does not compile, which line? Why?

Solution: Line 9: from the declaration in Line 3, the book is constrained to store dates only.

[of 3 points]

Genericity: Contracts and Implementations

4. All parts of this question are related to your new design of a generic book from Question 3 (c).

Contracts

(a) An invariant for the *GENERIC_BOOK* class is formally specified as:

$$\forall i, j : INTEGER \mid names.valid\_index(i) \wedge names.valid\_index(j) \bullet names[i] \sim names[j] \Rightarrow i = j$$

That is, there are no duplicates of strings stored in the *names* array (since book records are indexed by string names). Convert this mathematical expression to valid Eiffel using the *across* syntax. **Hints:** Consider nesting two *across* expressions, and using the |..| operator to create *iterable* integer interval expressions.

Solution:

```
across names.lower |..| names.upper is i all
    across names.lower |..| names.upper is j all
        names[i] = names[j]
            implies i = j
    end
end
```

[of 10 points]

(b) The precondition of feature *add(r, n)* is formally specified as:

$$\forall name : STRING \mid name \in names \bullet \neg (name \sim n)$$

That is, each string in the *names* array is not equal to the argument name n to be added. Convert this mathematical expression to valid Eiffel using the *across* syntax.

Solution:

```
across names as cursor all cursor.item /~ n end
```

[of 3 points]

(c) The postcondition of feature *add(r, n)* asserts that: **1)** sizes of the *names* and *records* arrays are both incremented by one; and **2)** the argument name n and record r are inserted

to the end of the *names* array and *records* array, respectively. Write this postcondition in valid Eiffel syntax.
Hint: Consider using the *count*, *lower*, and/or *upper* features from the *ARRAY* class.

Solution:

names.count = **old** *names.count* + 1
records.count = **old** *records.count* + 1
names[*names.upper*] ∼ *n*
records[*dates.upper*] ∼ *r*

[of 4 points]

(d) The precondition of feature *get(n)* is formally specified as:

$$\exists name : STRING \mid name \in names \bullet name \sim n$$

That is, there exists a string in the *names* array that is equal to the argument name *n*. Convert this mathematical expression to valid Eiffel using the *across* syntax.

Solution:

across *names* **as** *cursor* **some** *cursor.item* ∼ *n* **end**

[of 3 points]

(e) The postcondition of feature *find(r)* asserts that if the argument record *r* exists in the book, then the returned array is non-empty. Convert this into valid Eiffel syntax.
Hints: Do not use the **if** . . . **then** . . . **else** . . . **end** instruction to write this contract; instead, consider using a combination of the logical negation and implication, and the *is_empty* and *has* features from the *ARRAY* class.

Solution:

records.has (*r*) **implies not Result**.*is_empty*

[of 3 points]

(f) Since both features *get(n)* and *find(r)* are queries, they should **not** modify the state of the current account. So they have the same postcondition which asserts that the *pre-state* values of the two implementation arrays *names* and *records* are equal to their *post-state* values. Write these two constraints in valid Eiffel syntax.

Solution:

names ∼ **old** *names.deep_twin*
records ∼ **old** *records.deep_twin*

[of 6 points]

Implementations

(g) Write in valid Eiffel syntax the implementation for the *add* feature. Start your answer with the signature of *add*. **Hints:** Write your implementation in terms of the two array attributes *names* and *dates*. You may declare local variables if necessary. Consider using the *force(v: G; i: INTEGER)* or *put(v: G; i: INTEGER)* feature from the *ARRAY* class.

Solution:

```
add(r: G; n: STRING)
    do
        names.force (n, names.upper + 1)
        records.force (r, records.upper + 1)
    end
```

[of 4 points]

(h) Write in valid Eiffel syntax the implementation for the *find* feature. Start your answer with the signature of *find*. **Hints:** Write your implementation in terms of the two array attributes *names* and *dates*. You may declare local variables if necessary. Consider using the *force(v: G; i: INTEGER)* or *put(v: G; i: INTEGER)* feature from the *ARRAY* class.

Solution:

```
find(r: G): ARRAY[STRING]
    local
        i: INTEGER
    do
        create Result.make_empty
        from
            i := records.lower
        until
            i > records.upper
        loop
            if records[i] ~ r then
                Result.force (names[i], Result.upper + 1)
            end
            i := i + 1
        end
    end
```

[of 7 points]