EECS-3311 – Lab – Sorted Variants

L3

1/26/19 8:01:54 PM

1 Design Goals

```
require
     across 0 |..| 2 as i all lab_completed(i.item) end
     read accompanying document: Eiffel101[1]
ensure
     submitted on time
     no submission errors
rescue
     ask for help during scheduled labs
     attend office hours for TA William
```

- Lab1: Sorted Trees (use of `SEQ[G]` from Mathmodels)
- Lab2: Sorted Map (use of `FUN[K, V]` from Mathmodels)
- Lab3: Sorted Variants and the Iterator Design Pattern

See the architectural BON diagram in Figure 3. We provide you with the cluster `sorted-collections` (you did some related work in Lab1). And we also provide you with the cluster `sorted-map` (you did some related work in Lab2). Your task is to complete the cluster `student-design`, in which you *adapt*[2] the machinery of `sorted-collections` to create a variety of efficient sorted maps (the most efficient being a sorted red-black tree `SORTED_RBT_MAP[K, V]`). You must create and design all classes in the `student-design` cluster.

You may start with three subclasses of `SORTED_MAP_DESIGN`, and discover that there is much duplicated code that can be *pulled-up*[3] to a common parent class `SORTED_MAP_DESIGN`. Most of the implementation machinery for creating these variants of sorted map is provided to you in the cluster `sorted-collections`. In this design, you are expected to discover the power of inheritance, polymorphism, and dynamic binding. In addition, you will also implement the *Iterator Design Pattern*.[4] As before, you will also need to understand Design-by-Contract (some of the critical contracts are in Figure 3).

[1] See: https://www.eecs.yorku.ca/~eiffel/eiffel101/Eiffel101.pdf

[2] The adapter design pattern is one of the twenty-three well-known GoF design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse. The adapter pattern is a software design pattern (also known as wrapper) that allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without the need to modify their source code, just reusing already developed machinery via the API.

[3] Why Refactor using *pull-up*? Subclasses may be developed independently of one another, but have nearly identical features. Benefits: Gets rid of duplicate code (which is a common “code smell”). If you need to make changes to a feature, it’s better to do so in a single place than have to search for all duplicates of the feature in subclasses.

[4] See Eiffel101 and http://svn.eecs.yorku.ca/repos/sel-open/misc/tutorial/iterator/index.html (login with anonymous), for some actual code.

2 Getting started

These instructions are for when you work on one of the EECS Linux Workstations or Servers (e.g *red*). You should not compile on *red* as it is a shared server; compile on an EECS workstation. See the course wiki for how to use the SEL-VM or your Laptop.

2.1 Retrieve Lab3

```
> ~sel/retrieve/3311/lab3
```

This will provide you with a starter directory `sorted-variants` with the following structure:

```
sorted-variants
├── model
|   ├── sorted-collections
|   |   ├── kv_pair.e
|   |   ├── node.e
|   |   ├── sorted_adt.e
|   |   ├── sorted_bst.e
|   |   ├── sorted_linear.e
|   |   ├── sorted_rbt.e
|   |   ├── sorted_tree.e
|   |   └── util.e
|   ├── sorted-maps
|   |   ├── sorted_map_adt.e
|   |   └── sorted_model_map.e
|   └── student-design
|       ├── sorted_bst_map.e
|       ├── sorted_linear_map.e
|       ├── sorted_map_cursor.e
|       ├── sorted_map_design.e
|       └── sorted_rbt_map.e
├── root
|   └── root.e
├── sorted-variants.ecf
└── tests
    ├── instructor
    |   ├── bst_tests.e
    |   ├── int_key_tests.e
    |   ├── iterable
    |   |   ├── iterable_sorted_bst_map_tests.e
    |   |   ├── iterable_sorted_linear_map_tests.e
    |   |   ├── iterable_sorted_map_tests.e
    |   |   ├── iterable_sorted_model_map_tests.e
    |   |   └── iterable_sorted_rbt_map_tests.e
    |   ├── linear_tests.e
    |   ├── model_tests.e
    |   ├── p_key_tests.e
    |   ├── person.e
    |   ├── rbt_tests.e
    |   └── student_design_tests.e
    └── student
        └── student_tests.e
```

Figure 1 starter directory sorted-variants

2.2 BON Class Diagrams: System Architecture

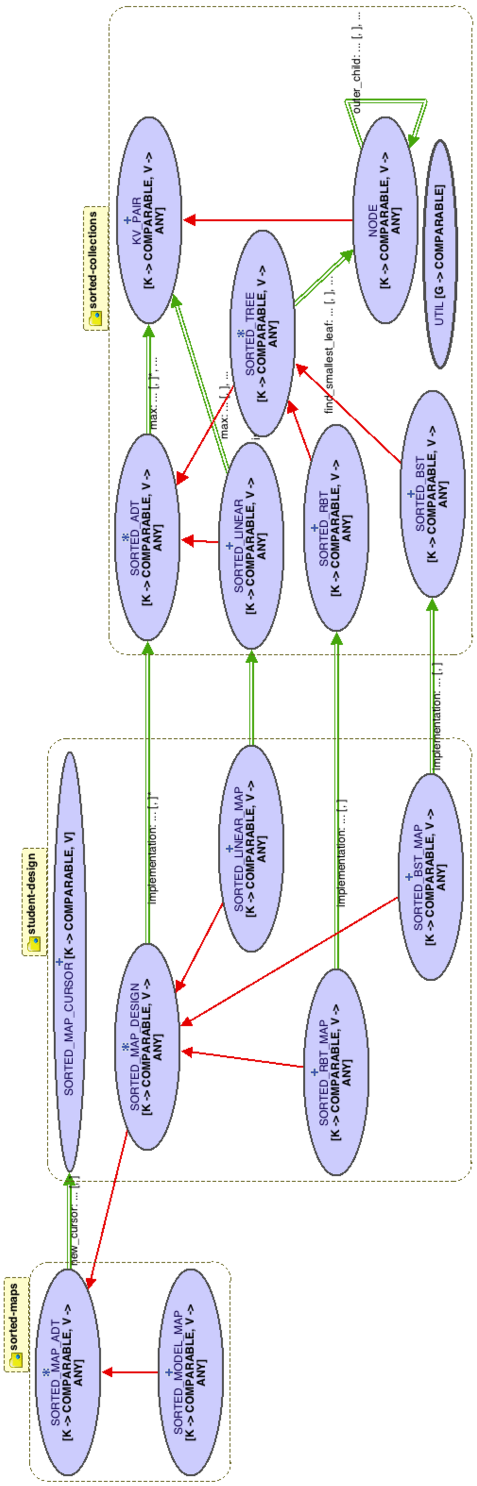

Figure 2 BON diagram generated by the IDE

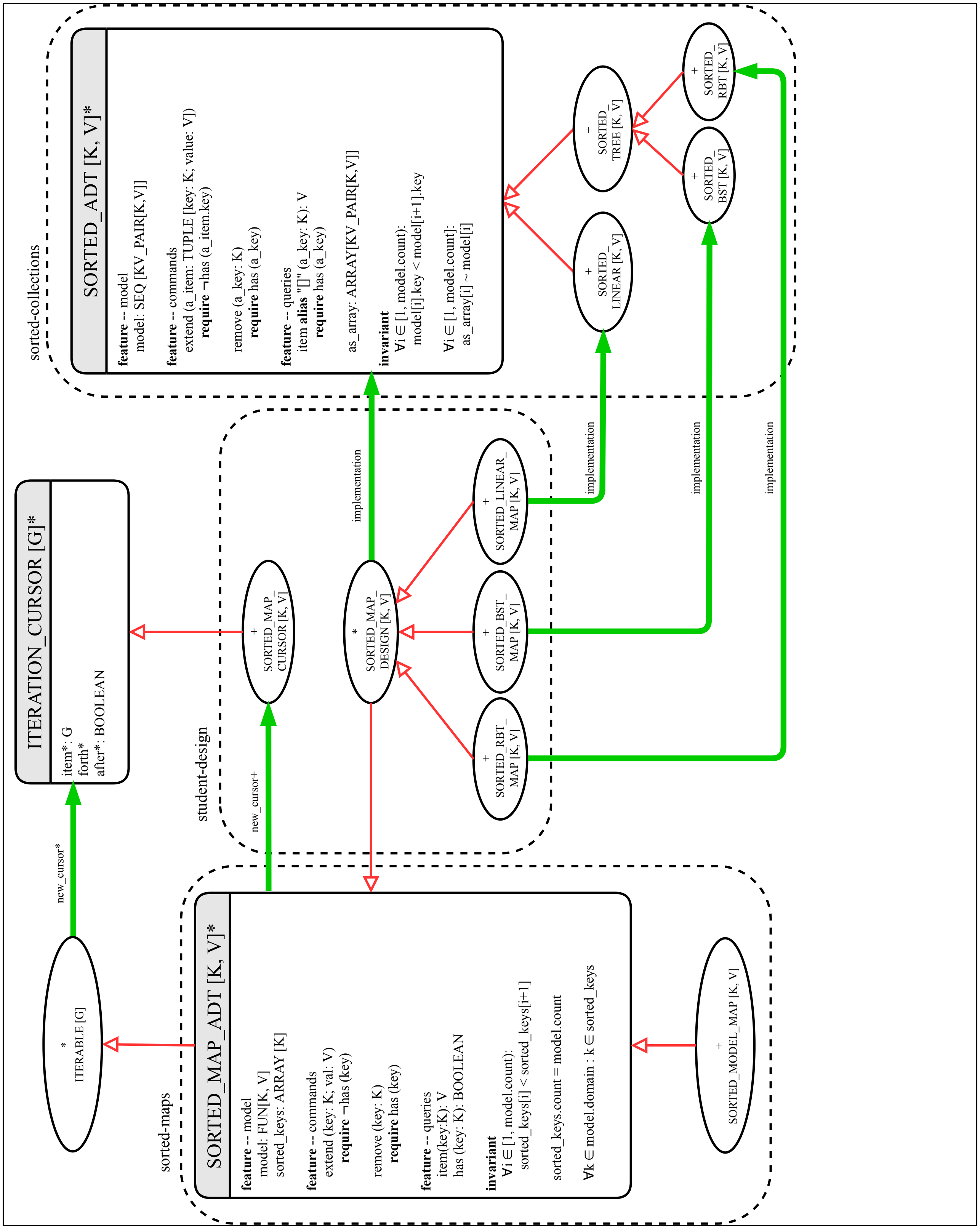

Figure 3 Architectural BON diagram using the template and draw.io

2.3 Architecture is part of design

A significant component of design is to describe the **architecture** of the system under construction. A UML or BON class diagram is an important artifact for describing the design architecture.

See Figure 2 (the BON diagram as generated by the IDE from the source code[5]) and Figure 3 (a higher-level BON diagram using the *draw.io* template[6]). Figure 3 provides a more selective and better structured view of the design architecture; but it was useful to have Figure 2 (from the IDE) as a starting point. In a later lab and the project you will be required to understand and construct these diagrams, so ensure that you obtain the skill now.[7] Ensure that you understand the meaning of all the figures, arrows, and symbols in the diagrams.

2.4 Missing elements of the Design:

- The *deferred* class `SORTED_MAP_ADT` (in Figure 1, this class is in **red**) in cluster `sorted-maps` has **missing contracts**. Complete all contracts with the **TO DO** labels:
    - `prune_tolerant` (postcondition)
    - `merge` (precondition and postcondition)
- The five classes in **red** in cluster `student-design` are **missing**, and you are required to create them and supply them with proper implementations.
- In class `STUDENT_TESTS`, add as many (at least four) test cases as you judge necessary for testing the correctness of your software.

2.5 Compile the Lab

You can now compile the Lab.

```
> estudio sorted-map/sorted-variants.ecf &
```

where `sorted-variants.ecf` is the Eiffel configuration file for this Lab, with the `ROOT` class:

[5] In this Lab, practice using the IDE at: https://www.eiffel.org/doc/eiffelstudio/Diagram_tool.
[6] In this Lab, practice using this template at: http://seldoc.eecs.yorku.ca/doku.php/eiffel/faq/bon. You will need this in a subsequent Lab and the project.
[7] http://seldoc.eecs.yorku.ca/doku.php/eiffel/faq/bon

```
class
    ROOT

inherit
    ARGUMENTS
    ES_SUITE

create
    make

feature {NONE} -- Initialization

    make
            -- Run application.
        do
            add_test (create {STUDENT_TESTS}.make)
            add_test (create {ITERABLE_SORTED_MODEL_MAP_TESTS}.make)
--          add_test (create {MODEL_TESTS}.make)
--          add_test (create {STUDENT_DESIGN_TESTS}.make)
--          add_test (create {BST_TESTS}.make)
--          add_test (create {RBT_TESTS}.make)
--          add_test (create {LINEAR_TESTS}.make)
--          add_test (create {ITERABLE_SORTED_BST_MAP_TESTS}.make)
--          add_test (create {ITERABLE_SORTED_RBT_MAP_TESTS}.make)
--          add_test (create {ITERABLE_SORTED_LINEAR_MAP_TESTS}.make)
            show_browser
            run_espec
        end
end
```

This initial starter project given to you is ***not*** expected to compile. It will have the following error:

```
Rule  Description                                                              Location
VTCT  Type is based on unknown class SORTED_MAP_CURSOR.                        ITERABLE_SORTED_MAP_TESTS (tests)
      Error code: VTCT

      Error: type is based on unknown class.
      What to do: use an identifier that is the name of a class in the universe.

      Class: ITERABLE_SORTED_MAP_TESTS
      Unknown class name: SORTED_MAP_CURSOR
      Line: 58
               m: SORTED_MAP_ADT[INTEGER, STRING]
      ->       ic: SORTED_MAP_CURSOR[INTEGER, STRING]
               tuples: LINKED_LIST[TUPLE [k: INTEGER; v: STRING]]
VTCT  Type is based on unknown class SORTED_MAP_CURSOR.                        SORTED_MAP_ADT (sorted-maps)
      Error code: VTCT

      Error: type is based on unknown class.
      What to do: use an identifier that is the name of a class in the universe.

      Class: SORTED_MAP_ADT [K -> COMPARABLE, V -> ANY]
      Unknown class name: SORTED_MAP_CURSOR
      Line: 332

      ->  new_cursor: SORTED_MAP_CURSOR [K, V]
               -- Fresh cursor associated with current structure
```

Before proceeding to the next page, think about what these errors mean to inform you!

2.6 Implement the Iterator Pattern[8]

Double click on the 2nd compilation error message above:

```
VTCT Type is based on unknown class SORTED_MAP_CURSOR
```

You will then be brought to the `SORTED_MAP_ADT` class:

```
deferred class
    SORTED_MAP_ADT [K -> COMPARABLE, V -> ANY]

inherit

    ITERABLE [TUPLE [K, V]]
        redefine
            is_equal,
            out
        end

feature --iteration

    new_cursor: SORTED_MAP_CURSOR [K, V]
            -- Fresh cursor associated with current structure
        do
            create Result.make (as_array)
        end
```

Recall that to make a class *iterable*, the first step is to make it inherit from the deferred class `ITERABLE[G]`, and the second step is to implement `new_cursor: G`, which is the only inherited *deferred* feature from `ITERABLE[G]`. For example, the above class `SORTED_MAP_ADT` inherits from `ITERABLE[G]` by instantiating the generic parameter as `TUPLE[K, V]`. Consequently, when a client iterates through an *iterable* instance of `SORTED_MAP_ADT` (via the `across` construct; see the next section for an example in class `ITERABLE_SORTED_MAP_TESTS`), each item retrieved from the map is always of the type `TUPLE[K, V]`, regardless of the underlining implementation being a mathematical function, a binary search tree, a red-black tree, or an array. This obeys the **information hiding** design principle.

In order to allow clients to iterate through a `SORTED_MAP_ADT` instance, the `new_cursor` query must be implemented. In the above fragment of code, the `new_cursor` query is already "implemented", but its return type denotes a class `SORTED_MAP_CURSOR` that does not yet exist. Your task is to create this class and implement all the necessary features. Here are some hints:

- What would be the relationship between `SORTED_MAP_CURSOR` and `ITERATION_CURSOR`?
- The one-line implementation for `new_cusor` reads: **create Result**.make (as_array). Where should the make feature be declared? What type of parameter should it have? Should it be a query, command, or constructor?

[8] See Eiffel101 and also below for more examples on implementing the Iterator pattern:
http://svn.eecs.yorku.ca/repos/sel-open/misc/tutorial/iterator/index.html
http://svn.eecs.yorku.ca/repos/sel-open/misc/tutorial/iterator/Documentation/index.html

2.7 Use the Iterator Pattern

Double click on the 1st compilation error message above:

`VTCT Type is based on unknown class SORTED_MAP_CURSOR`

You will then be brought to the `ITERABLE_SORTED_MAP_TESTS` class:

```
test_iteration_cursor: BOOLEAN
    local
        m: SORTED_MAP_ADT[INTEGER, STRING]
        ic: SORTED_MAP_CURSOR[INTEGER, STRING]
        tuples: LINKED_LIST[TUPLE [k: INTEGER; v: STRING]]
    do
        comment ("test_iteration_cursor: test iterating through map cursor")
        m := new_map
        m.extend (3, "three")
        m.extend (1, "one")
        m.extend (4, "four")
        m.extend (2, "two")

        create tuples.make
        from
            ic := m.new_cursor
        until
            ic.after
        loop
            tuples.extend (ic.item)
            ic.forth
        end

        Result :=
                tuples.count = 4
            and tuples[1].k = 1 and tuples[1].v ~ "one"
            and tuples[2].k = 2 and tuples[2].v ~ "two"
            and tuples[3].k = 3 and tuples[3].v ~ "three"
            and tuples[4].k = 4 and tuples[4].v ~ "four"
    end
```

As discussed, each item retrieved from a `SORTED_MAP_ADT[K, V]` is of type `TUPLE[K, V]`. In the above example test feature `test_iteration_cursor`, the following two local variable declarations are related:

```
m: SORTED_MAP_ADT[INTEGER, STRING]
ic: SORTED_MAP_CURSOR[INTEGER, STRING]
```

both declarations instantiate the generic parameter `K` to `INTEGER`, and `V` to `STRING`. Consequently, each item retrieved from map `m` is of type `TUPLE[K, V]`, and once we get the return value from `m.new_cursor`, we can store the tuples accessed by all iterations into a list. The test also illustrates that the order in which a client iterates through a `SORTED_MAP_ADT` instance is sorted according to the keys.

In the SORTED_MAP_ADT class, there is another hint, as to how the SORTED_MAP_CURSOR class should be implemented, in the out feature:

```
out: STRING
        -- New string containing terse printable representation
        -- of current object
    do
        create Result.make_empty
        across
            Current as cursor
        loop
            Result.append ("(" + cursor.item.key.out + "," + cursor.item.value.out + ")")
        end
    end
```

In the above out feature, we can go across the Current map because SORTED_MAP_ADT is declared as *iterable*. Specifically, cursor will be assigned the returned value of the new_cursor feature (which is of type SORTED_MAP_CURSOR). This means that in the SORTED_MAP_CURSOR class there should be an item feature (which is inherited from the ITERATION_CURSOR class), and its return value is a tuple (as constrained by the ITERATBLE[TUPLE[K, V]] declaration at the top of the SORTED_MAP_ADT class).

The above out feature suggests that the returned tuple has a key field and a value field. For a tuple, you can declare names of their fields (so that you can reference them like the above out feature does):

```
item: TUPLE[key: K; value: V]
```

Without the above declarations of field names (i.e., TUPLE[K, V]), you can only refer to fields via their positions (as can be seen in the above test_iteration_cursor feature).

After you have correctly implemented the new SORTED_MAP_CURSOR class, and add proper tests to STUDENT_TESTS, you should obtain a Green Bar:

PASSED (6 out of 6)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	6	6
All Cases	6	6
State	**Contract Violation**	**Test Name**
Test1	STUDENT_TESTS	
PASSED	NONE	t1: describe test t1 here
PASSED	NONE	t2: describe test t2 here
PASSED	NONE	t3: describe test t3 here
PASSED	NONE	t4: describe test t4 here
Test2	ITERABLE_SORTED_MODEL_MAP_TESTS	
PASSED	NONE	test_iterable_map: test iterating through map
PASSED	NONE	test_iteration_cursor: test iterating through map cursor

2.8 Specify SORTED_MAP_ADT Contracts

Now go back to the ROOT class and uncomment the next line from the make feature:

```
add_test (create {MODEL_TESTS}.make)
```

```
    make
            -- Run application.
        do
            add_test (create {STUDENT_TESTS}.make)
            add_test (create {ITERABLE_SORTED_MODEL_MAP_TESTS}.make)
            add_test (create {MODEL_TESTS}.make)
--          add_test (create {STUDENT_DESIGN_TESTS}.make)
--          add_test (create {BST_TESTS}.make)
--          add_test (create {RBT_TESTS}.make)
--          add_test (create {LINEAR_TESTS}.make)
--          add_test (create {ITERABLE_SORTED_BST_MAP_TESTS}.make)
--          add_test (create {ITERABLE_SORTED_RBT_MAP_TESTS}.make)
--          add_test (create {ITERABLE_SORTED_LINEAR_MAP_TESTS}.make)
            show_browser
            run_espec
        end
```

When you execute the Lab in workbench mode (Control-Alt-F5), you will see a Red Bar: there are failing tests from the *ESpec* Unit Testing report. To fix these failing tests, you are required to **specify contracts** for features in the SORTED_MAP_ADT class. The deferred class SORTED_MAP_ADT (in Figure 1, this class is in red) in cluster sorted-maps has missing contracts. Complete all contracts with the TO DO labels such as: prune_tolerant (postcondition); merge (precondition and postcondition).

Follow the **TO DO** labels to see which features and hints. Make sure that you obtain a Green Bar before proceeding to the next section.

Notice that all contracts specified at the level of the *deferred* class SORTED_MAP_ADT will be inherited by all its descendants (i.e., classes which inherit from it). Contracts in the context of inheritance are related to the topic of *subcontracting*, which we will discuss in class. For the purpose of this lab, just be aware that inappropriate contracts at the parent class level might lead to violations at the descendant class level.

Every descendant (or implementation) of SORTED_MAP_ADT will have to satisfy the contracts of this ADT – this will ensure that all the descendants will satisfy the ADT specifications.

Question: What classes in the BON diagram of Figure 3 are descendants of the sorted map ADT? [Hint: there are at more than 3 such classes, although you may not have to implement all of them]

2.9 Study the Given Variant of SORTED_MAP_ADT

Given that the SORTED_MAP_ADT class is *deferred* (because it has unimplemented features), we cannot instantiate it directly at runtime. Instead, we ought to create *effective* descendants of it, each of which adopt a different strategy of efficient implementation (e.g., binary search tree, red-black tree, *etc.*). Each of such *effective* descendant class can be instantiated at runtime.

A complete example is given to you: a descendant class `SORTED_MODEL_MAP` that uses the `FUN` class for **implementing** a sorted map:

```
class
    SORTED_MODEL_MAP [K -> COMPARABLE, V -> ANY]

inherit
    SORTED_MAP_ADT[K,V]

create
    make_empty, make_from_array, make_from_sorted_map

feature -- model
    model: FUN [K, V]
            -- abstraction function
        do
            Result := implementation
        end

feature{NONE} -- attributes
    implementation: FUN[K,V]
            -- inefficient but abstract implementation of sorted map
        attribute
            create Result.make_empty
        end

    instance: like Current
        attribute
            create Result.make_empty
        end
```

We are going to see an interesting development at work. The above class uses the **immutable queries** of `FUN[K,V]` for the model and the **specifications** of the features, but is also uses the analogous **commands** of `FUN[K,V]` to implement the features of the class.[9] This way we get an effective class and we can start writing tests for it directly, tests that will be re-used for many other implementation sub-classes of the map ADT. This implementation will be relativity inefficient by comparison with a red-black tree implementation of a sorted map. But it is executable model that can be constructed quickly, and therein lies its value for exploring and testing sorted map behaviors.

The two private attributes `implementation` and `instance` are an important hint to how you can complete other variants of the `SORTED_MAP_ADT`:

- The `implementation` attribute suggests a client-supplier relationship between the `FUN` class from the Mathmodels library: all other *deferred* features inherited from `SORTED_MAP_ADT` can now be implemented by calling features of class `FUN` on attribute `implementation`.
- All feature contracts in the `SORTED_MAP_ADT` class references the deferred feature `model`. At the level of class `SORTED_MODEL_MAP`, all these inherited contracts become executable, because `model` is implemented as an alias to the implementation attribute.
- The implementation of the `sub_map` feature in `SORTED_MODEL_MAP` references the *deferred* feature `instance`. This means at the level of `SORTED_MAP_ADT`, the `sub_map`

---

[9] For more background, if you are interested, see: https://www.eecs.yorku.ca/~jonathan/publications/2018/MoDRE18.pdf

feature is not directly *executable*, but merely serves as a template. At the lower level of *effective* descendants of SORTED_MAP_ADT, such as SORTED_MODEL_MAP, where instance is implemented, the sub_map feature not only becomes executable, but also when executed, it will call the version of instance of the corresponding effective descendant class (i.e., dynamic binding). For example, the instance attribute in SORTED_MODEL_MAP implements the corresponding *deferred* feature from SORTED_MAP_ADT, and at the runtime calling the sub_map feature on a SORTED_MODEL_MAP instance will trigger the version of instance that is implemented in SORTED_MODEL_MAP.

2.10 Implement a other Variants of SORTED_MAP_ADT

Go back to the ROOT class and uncomment the next line from the make feature:

```
add_test (create {STUDENT_DESIGN_TESTS}.make)
```

```
    make
            -- Run application.
        do
            add_test (create {STUDENT_TESTS}.make)
            add_test (create {ITERABLE_SORTED_MODEL_MAP_TESTS}.make)
            add_test (create {MODEL_TESTS}.make)
            add_test (create {STUDENT_DESIGN_TESTS}.make)
--          add_test (create {BST_TESTS}.make)
--          add_test (create {RBT_TESTS}.make)
--          add_test (create {LINEAR_TESTS}.make)
--          add_test (create {ITERABLE_SORTED_BST_MAP_TESTS}.make)
--          add_test (create {ITERABLE_SORTED_RBT_MAP_TESTS}.make)
--          add_test (create {ITERABLE_SORTED_LINEAR_MAP_TESTS}.make)
            show_browser
            run_espec
        end
```

Now the project is not expected to compile, because the STUDENT_DESIGN_TESTS class references to some non-existing classes. As a hint, study carefully the given test test_sorted_map_adt, which specifies precisely the expected inheritance hierarchy of the missing classes (and you can also see this hierarchy design from the given BON diagram). As an example, when we declare the local variable:

```
m: SORTED_MAP_ADT[INTEGER, STRING]
```

the *static type* of m is declared as SORTED_MAP_ADT, meaning that at runtime, we may assign to m any object whose type is a descendant of SORTED_MAP_ADT. For example, the object creation:

```
create {SORTED_MODEL_MAP[INTEGER, STRING]} m.make_empty
```

changes the *dynamic type* of m to SORTED_MODEL_MAP, which is indeed a descendant of SORTED_MAP_ADT.

You will see other similar declarations and creations of objects, whose *static* and *dynamic* types altogether indicate the expected inheritance hierarchy that you must create.

Now, in order to implement these new classes, here are some hints for you:

- First create these classes and declare the expected inheritance relationship between them, as suggested by the above test.
- To implement these classes, revisit how the `implementation` and `instance` attributes were used to implement in the `SORTED_MODEL_MAP` class, which is an example variant of the deferred class `SORTED_MAP_ADT`. Then, look at the given classes in the `sorted-collections` cluster and find the corresponding classes for your implementation.
- Your final implementation for all these new classes should obey the **single-choice** design principle: for each *deferred* feature that is inherited from `SORTED_MAP_ADT`, there should be a common place to define in its implementation that can be shared by (or inherited to) all *effective* descendants.

Important Note: *To check syntax, a compile (shortcut F7) is sufficient. But it is best to freeze (Control-F7) before running unit tests. Run the unit tests often! (even after very small changes to your code). When you compile, ensure that the compilation succeeded (reported at the bottom of the IDE). When you run unit tests, ensure that all your routines terminate (can also be checked in the IDE). If you keep running the tests without halting the current non-terminating run, you will keep adding new non-terminating processes to the workstation, and the workstation will choke on all the concurrently executing processes. Study how to use your tools effectively and efficiently.*

3 To Submit

1. Add correct implementations and contracts as specified.
2. Work incrementally one feature at a time. Run all regression tests before moving to the next feature. This will help to ensure that you have not added new bugs, and that the prior code you developed still executes correctly.
3. Add at least 4 tests of your own to `STUDENT_TESTS`, i.e. don't just rely on our tests.
4. Don't make any changes to classes other than the ones specified.
5. Ensure that you get a green bar for all tests. Before running the tests, always freeze first.

You must make an electronic submission as described below.

1. On Prism (Linux), *eclean* your system, freeze it, and re-run all the tests to ensure that you get the green bar.
2. *eclean* your directory *sorted-variants* again to remove all EIFGENs.

Submit your Lab from the command line as follows:

```
submit 3311 Lab3 sorted-variants
```

You will be provided with some feedback. Examine your feedback carefully. Submit often and as many times as you like.

Remember

- Your code must compile and execute on the departmental Linux system (Prism) under CentOS7. That is where it must work and that is where it will be compiled and tested for correctness.
- Equip each test t with a *comment* ("t: …") clause to ensure that the ESpec testing framework and grading scripts process your tests properly. (Note that the colon ":" in test comments is mandatory.). An improper submission will not be given a passing grade.
- The directory structure of your folder sorted-variants must be a superset of Figure 1.

4 Exercise Questions

In class `SORTED_ADT`, there are two queries

- find_index
- has

The precondition of `find_index` uses `has` and the postcondition of `has` uses `find_index`.
Question: Are we going to run into infinite recursion with each one calling the other in a circle?

The answer to this question should be that it is ok. See the **Assertion Evaluation Rule** in OOSC2, p402. Think of a security guard at the entrance to a nuclear plant, in charge of inspecting the credentials of the visitors. But who will run the background check on the guards themselves? Contracts are like security guards protecting the business logic. Ensure that you understand this well as it is part of run-time assertion checking.