EECS-3311 – Lab1 – List Graph

1/9/20 3:54:27 PM

1 Goals

```
require
     Lab0 done
     read accompanying document: Eiffel-101
ensure
     submitted on time
     no submission errors
rescue
     ask for help during scheduled labs
     attend office hours for TAs Kevin and Connor
```

In design, the skill you wish to develop is the ability to distill a complex problem into its simplest components, and to organize the components into a cohesive and maintainable product.

Design is the construction of abstractions of data and computation and the organization of these abstractions into a working software application (Software Design with Java, M. Robillard, 2017)

The worldview underlying the Eiffel method is treating the whole process of software development as a continuum; unifying the concepts behind activities such as requirements, specification, design, implementation, verification, maintenance and evolution; and working to resolve the remaining differences, rather than magnifying them. Anyone who has worked in both specification and programming knows how similar the issues are. Formal specification languages look remarkably like programming languages; to be usable for significant applications they must meet the same challenges: defining a coherent type system, supporting abstraction, providing good syntax (clear to human readers and parsable by tools), specifying the semantics, offering modular structures, allowing evolution while ensuring compatibility. The same kinds of ideas, such as an object-oriented structure, help on both sides. Eiffel as a language is the notation that attempts to support this seamless, continuous process, providing tools to express both abstract specifications and detailed implementations. (Bertrand Meyer, 2014).

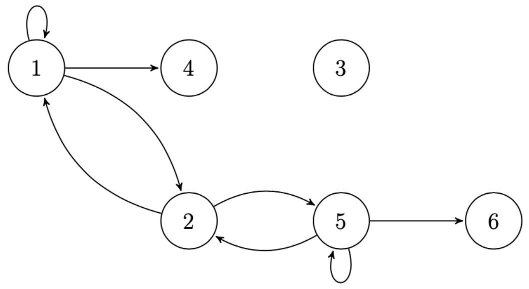
Graphs

A graph G is a structure $G = (V, E)$ where V is a finite set of vertices and E is a finite set of edges. To keep matters simple we will limit our summary to *directed graphs*. For example, in the graph of Fig. 1 the vertices are $\{1, 2, 3, 4, 5, 6\}$ and there are edges such as $1 \mapsto 2$ and $2 \mapsto 5$.[1] A path through the graph is a sequence of vertices, e.g. $\langle 1, 2, 5, \cdots \rangle$. In principle, in a path, vertices may appear more than once. However, often we remove cycles in a path for brevity.

A vertex v of a graph has a set of *outgoing* and *incoming* edges. A function *adjacent* (v) provides a sequence of vertices adjacent (via outgoing edges) to vertex v. For example, $\mathit{adjacent}(1) = \langle 4, 2 \rangle$.

If the vertices of a graph are comparable (such as integers, strings etc.), then *adjacent* (v) is a sequence of vertices adjacent to v in comparable order, and we may then write: $\mathit{adjacent}(1) = \langle 2, 4 \rangle$. A comparable order for vertices allows us to uniquely order paths, topological sorts and other properties that directed graphs might have.

We will denote such graphs as `COMPARABLE_GRAPH[V→COMPARABLE]`, where V is a generic parameter for vertices in the graph. If vertices are not comparable then we denote the graph structure as `GRAPH[V]`.



The above comes from graph-algorithms.pdf (in the docs folder). Refer to it for an abstract description of the breadth first search algorithm Fig. 4. See also Section 3.

Up to and including reading week, you will be dealing with a sequence of three labs involving the design, construction and use of graphs in an application.

- **Lab1**: *List Graph*. Specify and construct directed graphs implemented as an Adjacency List. Construct classes such as `VERTEX[G]` and `EDGE[G]` where `G` is a generic parameter that is comparable. In class `LIST_GRAPH[G]`, a list of *vertices* is of type `LIST[VERTEX[G]]` and an array of *edges* is `ARRAY[EDGE[G]]`. An edge has a *source* and *destination* vertex. A vertex has an *outgoing* and *incoming* list of edges as well as a sorted outgoing array of edges based on the comparable constraint. You gain familiarity working with arrays and lists in Eiffel. Specifications use classical contracts and ensure consistency in the data structures (e.g. see the class invariants in the BON class diagram of Figure 5).
- **Lab2**: *Abstract Graph*. Expands functionality of the `LIST_GRAPH` class to include more advanced queries. Also introduces you to a mathematical model `GRAPH [G]` (from the Mathmodels library). Mathmodels allows for complete specifications beyond what can be described with classical contracts.

- **Lab3**/ETF: *Maze*. Combines material from Lab1 and Lab2 for you to design an interactive game using graph structures.

This Lab and others are accompanied by the accompanying document **Eiffel-101** which you must read and understand in the first 3 weeks of the term[1].

Goals of Lab1:

- Develop familiarity and competence with Eiffel: the language, method and tools
- Review and code with basic OO concepts such as polymorphism, dynamic binding, static typing, generic parameters and recursion.
- Tools Use: Use the IDE to browse code, edit, compile, unit test with ESpec, document the design, and develop competence in the use of the debugger.
- Use *Design by Contract* (preconditions, postconditions and class invariants) for specifying APIs and the use of regression testing with ESpec unit tests for improving the correctness and robustness of your code.
- Constructing implementations that satisfy *specifications*.
- Understand BON class diagrams for describing design decisions.
- Use genericity and void safe programming constructs (**attached** and **detachable**) for abstraction and reliability.
- Introduction to Design principles such as the Command-Query Separation Principle and Information Hiding.
- Use the iterator design pattern using the keyword **across** for specifications of predicates that require the use of quantifiers (∀, ∃), e.g. for asserting that an array is sorted. We study this design pattern early on in the course.
- For more on the above, consult Eiffel-101 as needed.

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 your workstation.

Invoke the Eiffel IDE from the command line as `estudio19.05` (aliased to `estudio`) and the command line compiler is `ec19.05` (aliased to `ec`).

2.1 Retrieve and compile Lab1

```
> ~sel/retrieve/3311/lab1
```

[1] https://www.eecs.yorku.ca/~eiffel/pdf/Eiffel-101.pdf .

This will provide you with a starter directory `list-graph`. The directory has the structure shown in Table 1.

Table 1 Lab1 Directory Structure

```
list-graph/
 ├──── model
 |    └── list_graph.e
 |    └── vertex.e
 |    └── edge.e
 ├──── root
 |    └── root.e
 ├──── list-graph.ecf
 └── tests
        ├──── instructor
        |    └── test_vertex_instructor.e
        |    └── test_list_graph_instructor.e
        |    └── course.e
        └── student
             └── student_test.e
```

The three classes in **red** are **incomplete**, and you are required to complete all the *features* (queries and commands) in these classes, given the *specifications* (preconditions, postconditions and class invariants). The parts you must do are listed as *ToDo*. Also, you are required write at least five unit tests in class `STUDENT_TEST`.

You can now compile the Lab.

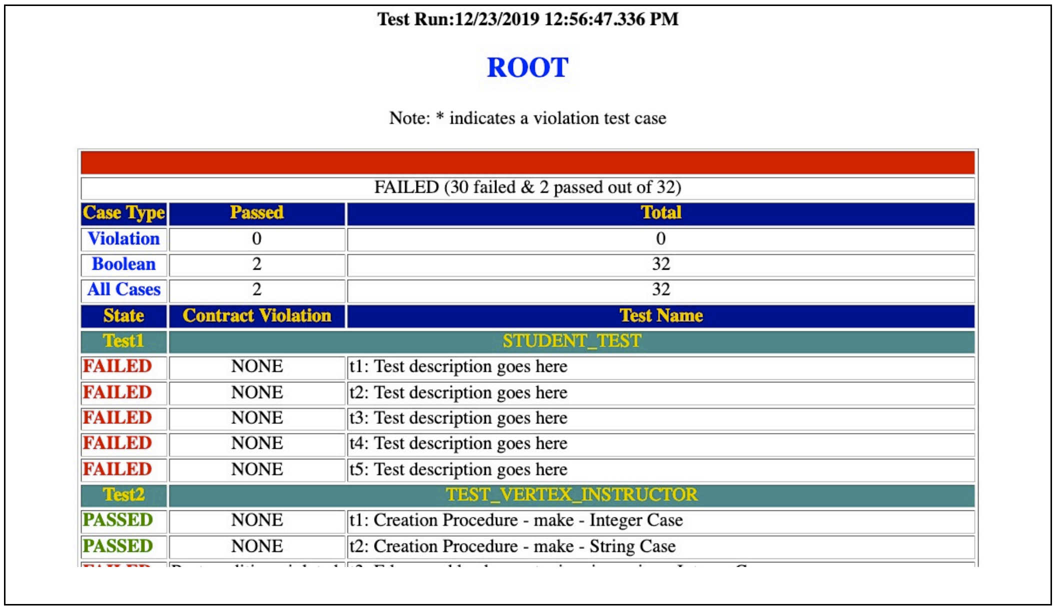
```
> estudio list-graph/list-graph.ecf &
```

where `list-graph.ecf` is the Eiffel configuration file for this Lab. The text of the root class (file `root.e`) is as shown in the box below.

```
note
      description: "ROOT class for Lab1: List Graph"
      author: "JSO and JW and CD"

class ROOT inherit
      ES_SUITE
create
      make

feature {NONE} -- Initialization
      make
                  -- Run tests
            do
                  add_test (create {STUDENT_TEST}.make)
                  add_test (create {TEST_VERTEX_INSTRUCTOR}.make)
--                add_test (create {TEST_LIST_GRAPH_INSTRUCTOR}.make)
                   -- uncomment after the above suites are working
                  show_browser
                  run_espec
            end
end -- class ROOT
```

One of the test suites is commented out. The project will compile and when you execute the Lab in workbench mode (Control-Alt-F5)[2], you will see the following *ESpec* Unit Testing report:

Figure 1: Part of ESpec Test Results



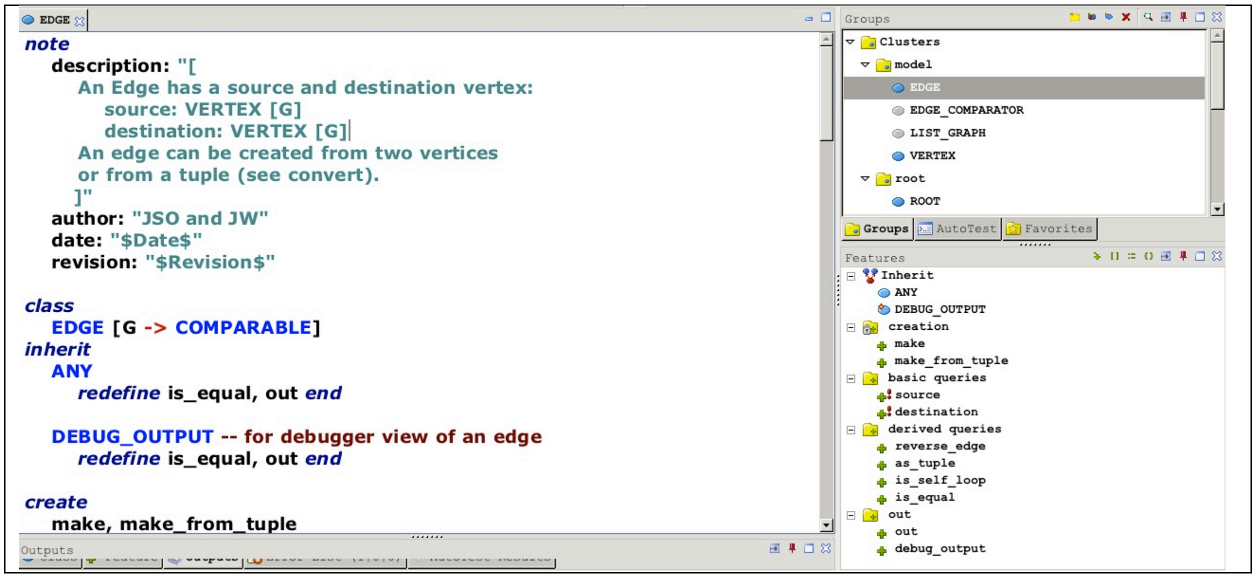
Test Run:12/23/2019 12:56:47.336 PM

ROOT

Note: * indicates a violation test case

FAILED (30 failed & 2 passed out of 32)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	2	32
All Cases	2	32
State	**Contract Violation**	**Test Name**
Test1	STUDENT_TEST	
FAILED	NONE	t1: Test description goes here
FAILED	NONE	t2: Test description goes here
FAILED	NONE	t3: Test description goes here
FAILED	NONE	t4: Test description goes here
FAILED	NONE	t5: Test description goes here
Test2	TEST_VERTEX_INSTRUCTOR	
PASSED	NONE	t1: Creation Procedure - make - Integer Case
PASSED	NONE	t2: Creation Procedure - make - String Case

[2] Read *Eiffel-101* for details.



```
note
    description: "[
        An Edge has a source and destination vertex:
            source: VERTEX [G]
            destination: VERTEX [G]
        An edge can be created from two vertices
        or from a tuple (see convert).
        ]"
    author: "JSO and JW"
    date: "$Date$"
    revision: "$Revision$"

class
    EDGE [G -> COMPARABLE]
inherit
    ANY
        redefine is_equal, out end

    DEBUG_OUTPUT -- for debugger view of an edge
        redefine is_equal, out end

create
    make, make_from_tuple
```

```
Groups
Clusters
  model
    EDGE
    EDGE_COMPARATOR
    LIST_GRAPH
    VERTEX
  root
    ROOT
```

```
Features
Inherit
    ANY
    DEBUG_OUTPUT
creation
    make
    make_from_tuple
basic queries
    source
    destination
derived queries
    reverse_edge
    as_tuple
    is_self_loop
    is_equal
out
    out
    debug_output
```

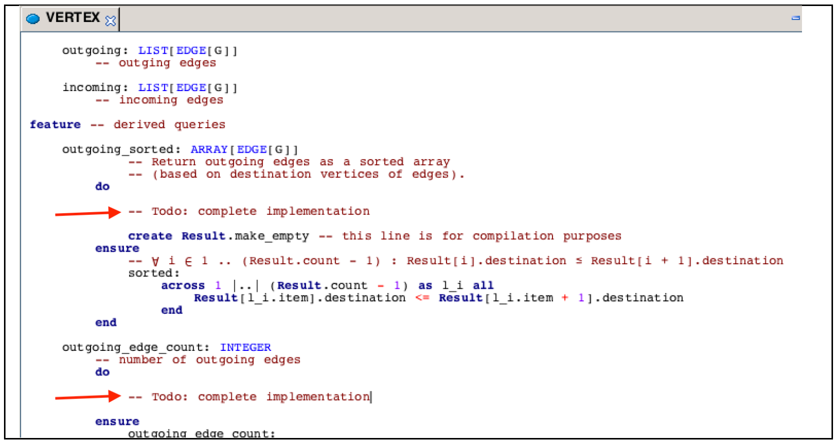
Figure 2 IDE showing clusters such as model (top right) and features (bottom right)

In Figure 2, we show a way of displaying class `EDGE` while at the same time viewing which cluster it is in (`model`) and all the features (queries and commands) of the class. Notice that the feature sections can be tagged (e.g. "basic queries"). It is worthwhile learning how to use the IDE to browse code. See OOSC2 Chapter 26 (A Sense of Style) for how to self-document your code.

2.2 Get the initial Unit Tests working

The Red Bar means that some of the tests fail. You must get all the tests to work and obtain a Green Bar.

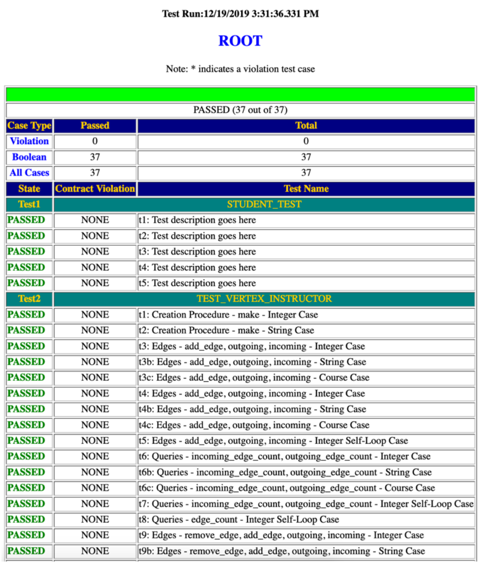
- `STUDENT_TEST`: you must write your own tests and we will check your tests. You may insert as many tests as you wish in this class.
- `TEST_VERTEX_INSTRUCTOR`: We provide you with some basic tests to get all the features in class `VERTEX` working correctly. See the `ToDo` hints in this class.



VERTEX

```
    outgoing: LIST[EDGE[G]]
        -- outging edges

    incoming: LIST[EDGE[G]]
        -- incoming edges

feature -- derived queries

    outgoing_sorted: ARRAY[EDGE[G]]
            -- Return outgoing edges as a sorted array
            -- (based on destination vertices of edges).
        do

            -- Todo: complete implementation

            create Result.make_empty -- this line is for compilation purposes
        ensure
            -- ∀ i ∈ 1 .. (Result.count - 1) : Result[i].destination ≤ Result[i + 1].destination
            sorted:
                across 1 |..| (Result.count - 1) as l_i all
                    Result[l_i.item].destination <= Result[l_i.item + 1].destination
                end
        end

    outgoing_edge_count: INTEGER
        -- number of outgoing edges
        do

            -- Todo: complete implementation

        ensure
            outgoing_edge_count:
```

Once you have all the incomplete features in class `VERTEX` working, you should now obtain a Green Bar:

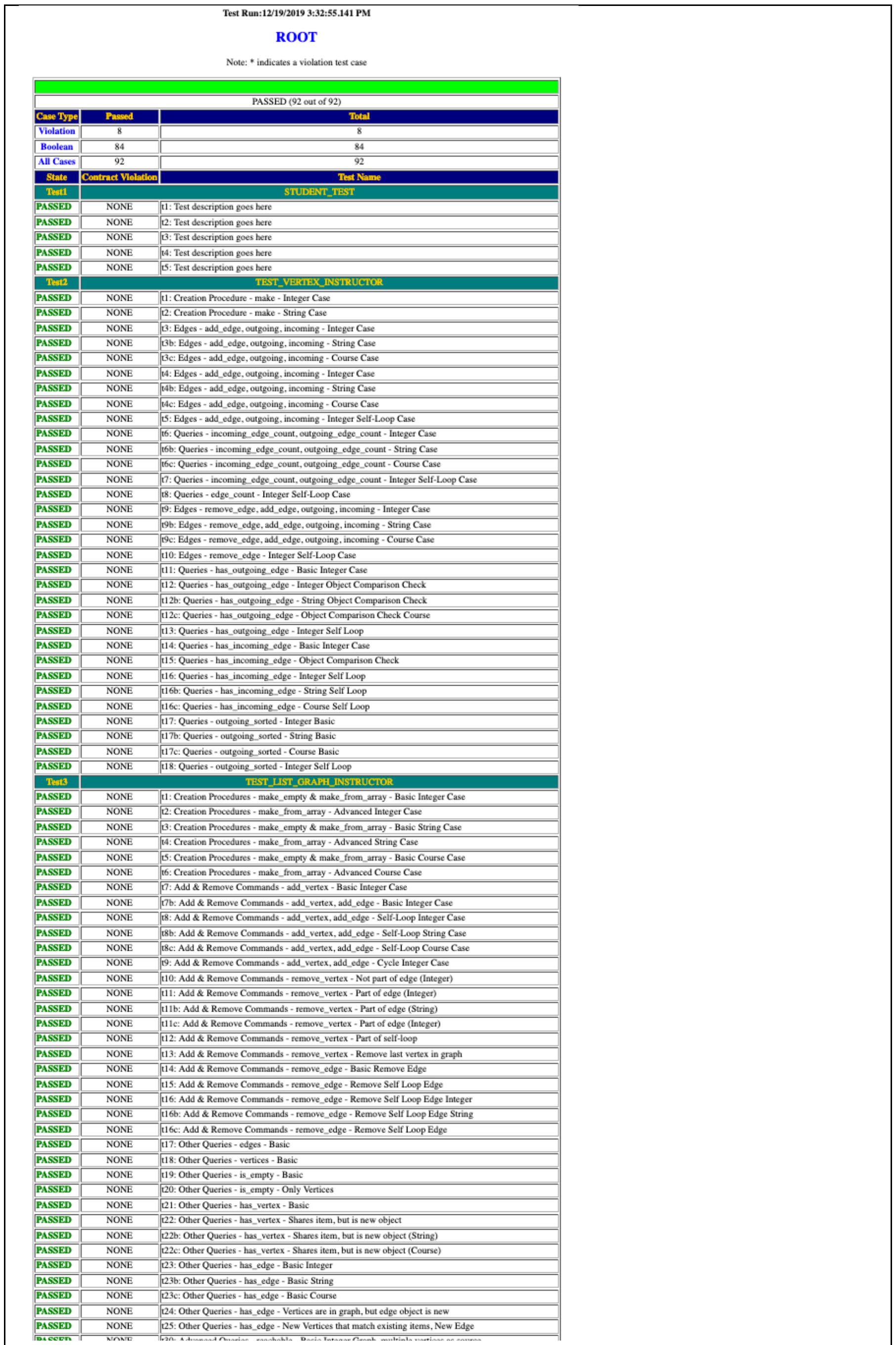


Test Run:12/19/2019 3:31:36.331 PM

ROOT

Note: * indicates a violation test case

🟩		
PASSED (37 out of 37)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	37	37
All Cases	37	37
State	**Contract Violation**	**Test Name**
Test1	STUDENT_TEST	
PASSED	NONE	t1: Test description goes here
PASSED	NONE	t2: Test description goes here
PASSED	NONE	t3: Test description goes here
PASSED	NONE	t4: Test description goes here
PASSED	NONE	t5: Test description goes here
Test2	TEST_VERTEX_INSTRUCTOR	
PASSED	NONE	t1: Creation Procedure - make - Integer Case
PASSED	NONE	t2: Creation Procedure - make - String Case
PASSED	NONE	t3: Edges - add_edge, outgoing, incoming - Integer Case
PASSED	NONE	t3b: Edges - add_edge, outgoing, incoming - String Case
PASSED	NONE	t3c: Edges - add_edge, outgoing, incoming - Course Case
PASSED	NONE	t4: Edges - add_edge, outgoing, incoming - Integer Case
PASSED	NONE	t4b: Edges - add_edge, outgoing, incoming - String Case
PASSED	NONE	t4c: Edges - add_edge, outgoing, incoming - Course Case
PASSED	NONE	t5: Edges - add_edge, outgoing, incoming - Integer Self-Loop Case
PASSED	NONE	t6: Queries - incoming_edge_count, outgoing_edge_count - Integer Case
PASSED	NONE	t6b: Queries - incoming_edge_count, outgoing_edge_count - String Case
PASSED	NONE	t6c: Queries - incoming_edge_count, outgoing_edge_count - Course Case
PASSED	NONE	t7: Queries - incoming_edge_count, outgoing_edge_count - Integer Self-Loop Case
PASSED	NONE	t8: Queries - edge_count - Integer Self-Loop Case
PASSED	NONE	t9: Edges - remove_edge, add_edge, outgoing, incoming - Integer Case
PASSED	NONE	t9b: Edges - remove_edge, add_edge, outgoing, incoming - String Case



Test Run:12/19/2019 3:32:55.141 PM

ROOT

Note: * indicates a violation test case

	PASSED (92 out of 92)	
Case Type	Passed	Total
Violation	8	8
Boolean	84	84
All Cases	92	92
State	Contract Violation	Test Name
Test1	STUDENT_TEST	
PASSED	NONE	t1: Test description goes here
PASSED	NONE	t2: Test description goes here
PASSED	NONE	t3: Test description goes here
PASSED	NONE	t4: Test description goes here
PASSED	NONE	t5: Test description goes here
Test2	TEST_VERTEX_INSTRUCTOR	
PASSED	NONE	t1: Creation Procedure - make - Integer Case
PASSED	NONE	t2: Creation Procedure - make - String Case
PASSED	NONE	t3: Edges - add_edge, outgoing, incoming - Integer Case
PASSED	NONE	t3b: Edges - add_edge, outgoing, incoming - String Case
PASSED	NONE	t3c: Edges - add_edge, outgoing, incoming - Course Case
PASSED	NONE	t4: Edges - add_edge, outgoing, incoming - Integer Case
PASSED	NONE	t4b: Edges - add_edge, outgoing, incoming - String Case
PASSED	NONE	t4c: Edges - add_edge, outgoing, incoming - Course Case
PASSED	NONE	t5: Edges - add_edge, outgoing, incoming - Integer Self-Loop Case
PASSED	NONE	t6: Queries - incoming_edge_count, outgoing_edge_count - Integer Case
PASSED	NONE	t6b: Queries - incoming_edge_count, outgoing_edge_count - String Case
PASSED	NONE	t6c: Queries - incoming_edge_count, outgoing_edge_count - Course Case
PASSED	NONE	t7: Queries - incoming_edge_count, outgoing_edge_count - Integer Self-Loop Case
PASSED	NONE	t8: Queries - edge_count - Integer Self-Loop Case
PASSED	NONE	t9: Edges - remove_edge, add_edge, outgoing, incoming - Integer Case
PASSED	NONE	t9b: Edges - remove_edge, add_edge, outgoing, incoming - String Case
PASSED	NONE	t9c: Edges - remove_edge, add_edge, outgoing, incoming - Course Case
PASSED	NONE	t10: Edges - remove_edge - Integer Self-Loop Case
PASSED	NONE	t11: Queries - has_outgoing_edge - Basic Integer Case
PASSED	NONE	t12: Queries - has_outgoing_edge - Integer Object Comparison Check
PASSED	NONE	t12b: Queries - has_outgoing_edge - String Object Comparison Check
PASSED	NONE	t12c: Queries - has_outgoing_edge - Object Comparison Check Course
PASSED	NONE	t13: Queries - has_outgoing_edge - Integer Self Loop
PASSED	NONE	t14: Queries - has_incoming_edge - Basic Integer Case
PASSED	NONE	t15: Queries - has_incoming_edge - Object Comparison Check
PASSED	NONE	t16: Queries - has_incoming_edge - Integer Self Loop
PASSED	NONE	t16b: Queries - has_incoming_edge - String Self Loop
PASSED	NONE	t16c: Queries - has_incoming_edge - Course Self Loop
PASSED	NONE	t17: Queries - outgoing_sorted - Integer Basic
PASSED	NONE	t17b: Queries - outgoing_sorted - String Basic
PASSED	NONE	t17c: Queries - outgoing_sorted - Course Basic
PASSED	NONE	t18: Queries - outgoing_sorted - Integer Self Loop
Test3	TEST_LIST_GRAPH_INSTRUCTOR	
PASSED	NONE	t1: Creation Procedures - make_empty & make_from_array - Basic Integer Case
PASSED	NONE	t2: Creation Procedures - make_from_array - Advanced Integer Case
PASSED	NONE	t3: Creation Procedures - make_empty & make_from_array - Basic String Case
PASSED	NONE	t4: Creation Procedures - make_from_array - Advanced String Case
PASSED	NONE	t5: Creation Procedures - make_empty & make_from_array - Basic Course Case
PASSED	NONE	t6: Creation Procedures - make_from_array - Advanced Course Case
PASSED	NONE	t7: Add & Remove Commands - add_vertex - Basic Integer Case
PASSED	NONE	t7b: Add & Remove Commands - add_vertex, add_edge - Basic Integer Case
PASSED	NONE	t8: Add & Remove Commands - add_vertex, add_edge - Self-Loop Integer Case
PASSED	NONE	t8b: Add & Remove Commands - add_vertex, add_edge - Self-Loop String Case
PASSED	NONE	t8c: Add & Remove Commands - add_vertex, add_edge - Self-Loop Course Case
PASSED	NONE	t9: Add & Remove Commands - add_vertex, add_edge - Cycle Integer Case
PASSED	NONE	t10: Add & Remove Commands - remove_vertex - Not part of edge (Integer)
PASSED	NONE	t11: Add & Remove Commands - remove_vertex - Part of edge (Integer)
PASSED	NONE	t11b: Add & Remove Commands - remove_vertex - Part of edge (String)
PASSED	NONE	t11c: Add & Remove Commands - remove_vertex - Part of edge (Integer)
PASSED	NONE	t12: Add & Remove Commands - remove_vertex - Part of self-loop
PASSED	NONE	t13: Add & Remove Commands - remove_vertex - Remove last vertex in graph
PASSED	NONE	t14: Add & Remove Commands - remove_edge - Basic Remove Edge
PASSED	NONE	t15: Add & Remove Commands - remove_edge - Remove Self Loop Edge
PASSED	NONE	t16: Add & Remove Commands - remove_edge - Remove Self Loop Edge Integer
PASSED	NONE	t16b: Add & Remove Commands - remove_edge - Remove Self Loop Edge String
PASSED	NONE	t16c: Add & Remove Commands - remove_edge - Remove Self Loop Edge
PASSED	NONE	t17: Other Queries - edges - Basic
PASSED	NONE	t18: Other Queries - vertices - Basic
PASSED	NONE	t19: Other Queries - is_empty - Basic
PASSED	NONE	t20: Other Queries - is_empty - Only Vertices
PASSED	NONE	t21: Other Queries - has_vertex - Basic
PASSED	NONE	t22: Other Queries - has_vertex - Shares item, but is new object
PASSED	NONE	t22b: Other Queries - has_vertex - Shares item, but is new object (String)
PASSED	NONE	t22c: Other Queries - has_vertex - Shares item, but is new object (Course)
PASSED	NONE	t23: Other Queries - has_edge - Basic Integer
PASSED	NONE	t23b: Other Queries - has_edge - Basic String
PASSED	NONE	t23c: Other Queries - has_edge - Basic Course
PASSED	NONE	t24: Other Queries - has_edge - Vertices are in graph, but edge object is new
PASSED	NONE	t25: Other Queries - has_edge - New Vertices that match existing items, New Edge
PASSED	NONE	t30: Advanced Queries - reachable - Basic Integer Graph, multiple vertices as source

There are more LIST_GRAPH tests than shown in this image …

2.3 Get all the remaining Unit Tests working

You may uncomment and compile the rest of the tests (`TEST_LIST_GRAPH_INSTRUCTOR`) in class `ROOT`. You then are required to get all these remaining tests working.

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.

2.4 List of features to implement to Specifications

Complete the following functions in `VERTEX` (in whichever order you deem best):

- `outgoing_sorted`
- `outgoing_edge_count`
- `incoming_edge_count`
- `edge_count`
- `has_outgoing_edge`
- `has_incoming_edge`
- `add_edge`
- `remove_edge`

Complete the following functions of `LIST_GRAPH`:

- `reachable` (use breadth first algorithm)
- `add_vertex` (implementation and contracts)
- `add_edge` (implementation and contracts)
- `remove_edge`
- `remove_vertex`

Notes:

- Get `VERTEX` working first, then `LIST_GRAPH`. Much of `LIST_GRAPH` relies on a correct implementation of VERTEX.
- There is an additional class called `COURSE` in the test folder. This class is used in some of the tests to ensure the algorithms work for arbitrary comparable generic classes as well as base classes such as `INTEGER` and `STRING`.

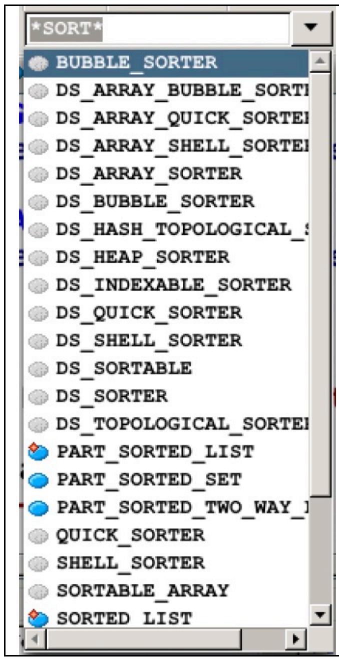
2.4.1 Query *reachable* in class LIST_GRAPH and BFS

Query `reachable` is a function query (i.e. a routine) that returns an array of vertices reachable from a specified source.

```
reachable (src: VERTEX[G]): ARRAY[VERTEX[G]]
                        -- Starting with vertex `src`, return list of vertices
                        -- visited via a breadth-first search.
                        -- Use `outgoing_sorted` so that
                        -- resulting array is uniquely ordered.
                        -- outgoing_sorted` is analogous to `adjacent`
                        -- in the abstract algorithm documentation of BFS.
            require
                                    cl_existing_source:
                                                has_vertex (src) -- given to students
            ensure
                                    cl_valid_vertices:
                                                across
                                                            Result as l_x
                                                all
                                                            has_vertex (l_x.item)
                                                end
                                    -- ∀x ∈ Result: x ∈ Current
            end
```

If you have forgotten how to do a breadth first search, see the accompanying document *graph-algorithms.pdf* on Mathmodels for graphs which provides an abstract algorithm for breadth first search. Notice how the **across** mechanism may be used to write the predicate given by ∀x ∈ *Result*: x ∈ *Current*.

2.4.2 Query in VERTEX: *outgoing_sorted*: ARRAY[EDGE[G]]

 *SORT* BUBBLE_SORTER DS_ARRAY_BUBBLE_SORTI DS_ARRAY_QUICK_SORTEI DS_ARRAY_SHELL_SORTEI DS_ARRAY_SORTER DS_BUBBLE_SORTER DS_HASH_TOPOLOGICAL_ DS_HEAP_SORTER DS_INDEXABLE_SORTER DS_QUICK_SORTER DS_SHELL_SORTER DS_SORTABLE DS_SORTER DS_TOPOLOGICAL_SORTEI PART_SORTED_LIST PART_SORTED_SET PART_SORTED_TWO_WAY_I QUICK_SORTER SHELL_SORTER SORTABLE_ARRAY SORTED_LIST	In the EiffelStudio IDE, use the regular expresssion `*SORT*` to obtain a list of classes that contain the string "SORT" and that sort based on a comparable generic parameter `G`. There are a number of classes that might be used. • Eiffel Base library: `SORTED_TWO_WAY_LIST`. This would have to be converted into an array. • Gobo library: `DS_ARRAY_QUICK_SORTER`. This would require the use of `KL_COMPARATOR`. • Eiffel Base extended library: `QUICK_SORTER`.

In the vertex class, queries `outgoing` and `incoming` return lists of edges for the current vertex that need not be sorted. However, query `outgoing_sorted` must return an array of outgoing edges that is sorted (by the *destination* of outgoing edges from the current vertex). We use this query in algorithms such as breadth first search to ensure that searches are done in a deterministic order, which also makes testing such algorithms easier.

You could write your own sort routine but it is safer to use libraries for this. Part of class `EDGE` is shown below.

```
class EDGE [G -> COMPARABLE] inherit
      ANY
            redefine
                  is_equal, out
            end
feature -- basic queries

      source: VERTEX [G]
                  -- source of edge

      destination: VERTEX [G]
                  -- destination of edge

      is_equal (other: like Current): BOOLEAN
                  -- Is the current edge equal to other?
            do
                  Result := source ~ other.source
                             and destination ~ other.destination
            end

end -- class EDGE
```

Two edges are equal if they have the same *source* and *destination*. The problem is that query `outgoing_sorted` must return an array of edges sorted by only the *destination* of an edge. So the following design will not work: specify that class `EDGE` also inherits from `COMPARABLE`, and effect {`EDGE[G]`} `is_less` (which is deferred in `COMPARABLE`).

```
 is_less alias "<" (other: like Current): BOOLEAN
          -- Is current object less than `other'?
      deferred
      ensure then
          asymmetric: Result implies not (other < Current)
      end
```

We can effect it by: `Result := e1.destination < e2.destination` (that uses *destination* only and not *source*). The problem is that this will violate the property of trichotomy inherited from `COMPARABLE`, which is:

```
is_equal (other: like Current): BOOLEAN
        -- Is `other' attached to an object of the same type
        -- as current object and identical to it?
    do
        Result := not (Current < other) and not (other < Current)
    ensure then
        trichotomy: Result = (not (Current < other) and not (other < Current))
    end
```

From the above postcondition we get:

$$\begin{aligned} e1.is\_equal(e2) &\equiv \neg(e1.destination < e2.destination) \wedge \neg(e2.destination < e1.destination) \\ &\equiv e1.destination = e2.destination \end{aligned}$$

which conflicts with query `{EDGE[G]}is_equal` which involves both the source and the destination.

A possible solution is to use a different comparator (e.g. using the Gobo or Eiffel Base Extended libraries). For example one might do something like:

```
class EDGE_COMPARATOR[G -> COMPARABLE] inherit
      KL_COMPARATOR[EDGE[G]]
feature  -- effect attached_less_than
```

```
-- in outgoing_sorted
l_comparator: EDGE_COMPARATOR [G]
l_sorter: DS_ARRAY_QUICK_SORTER [EDGE[G]]
…
create a_sorter.make (a_comparator)
…
l_sorter.sort
```

Hint: If you don't understand the above, write a unit test to help you explore these classes so that you understand how to use them. You may add classes of your own (such as `EDGE_COMPARATOR`) in cluster `model`.

2.5 Documenting Architecture: BON/UML Class Diagrams

A critical way to document a design (and the design decisions) is via a BON class diagram. Use the EiffelStudio IDE to generate BON (or UML) class diagrams. For our Lab, the IDE generates the following:

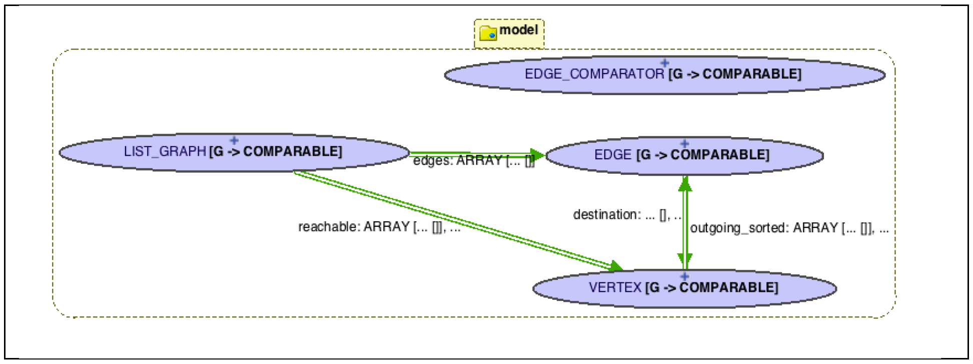

model
EDGE_COMPARATOR [G -> COMPARABLE]
LIST_GRAPH [G -> COMPARABLE]
edges: ARRAY [... []]
EDGE [G -> COMPARABLE]
reachable: ARRAY [... []], ...
destination: ... [], ...
outgoing_sorted: ARRAY [... []], ...
VERTEX [G -> COMPARABLE]

Figure 3 BON class diagram (IDE generated)

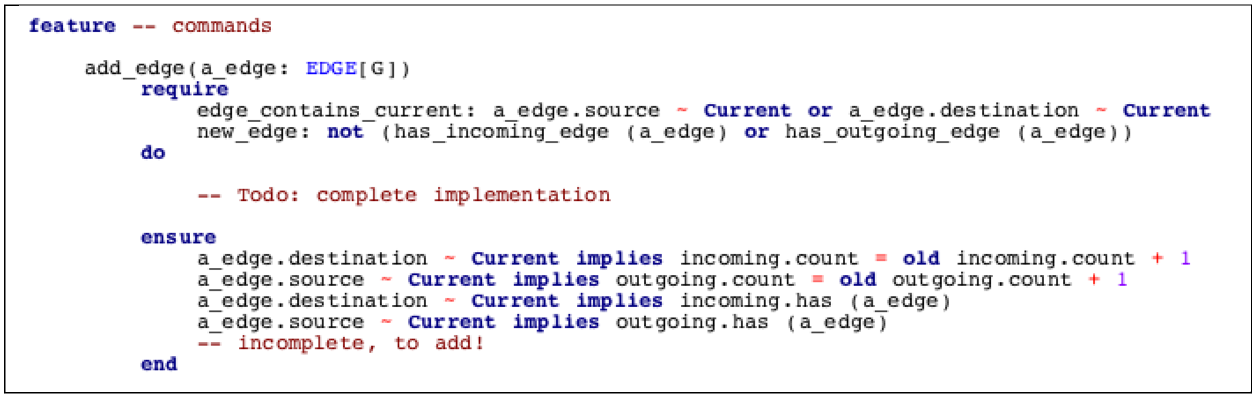
This diagram shows some important characteristics of the design:

- The diagram shows one clusters: *model*. Each cluster contains classes (shown as ellipses). The green double arrows denote *client-supplier* relationships. There are none in this diagram, but a red single arrow denotes an *inheritance* relationship between classes.
- A "*" decorator denotes *deferred* classes and the "+" decorator denotes *effective* classes. A deferred class has at least one *routine* (either a query or a command) that is deferred, i.e. has no implementation. Such a class cannot be instantiated at runtime, and thus does not have explicit constructors.
- Classes are always written using UPPER_CASE. *Features* (queries and commands) are written using lower case.
- Deferred class LIST_GRAPH [G] has one *generic* parameter, G for the type of element stored in the vertices. Generic parameter K is *constrained* to be COMPARABLE, needed for a sorted order.

As shown above, class LIST_GRAPH has an *edges* attribute containing all the edges in the graph, and a *reachable* query that returns all vertices that are reachable from the specified vertex. There is also a *vertices* attribute that contains all the vertices in the graph (not shown in the diagram).

The VERTEX class has an *outgoing_sorted* query that returns all of the outgoing edges from the specified vertex. VERTEX supplies the EDGE class when it is used in the *source* and *destination* attributes.

Throughout the implementations you are required to code, there are often contracts that the implementation must satisfy.



```
feature -- commands

    add_edge(a_edge: EDGE[G])
        require
            edge_contains_current: a_edge.source ~ Current or a_edge.destination ~ Current
            new_edge: not (has_incoming_edge (a_edge) or has_outgoing_edge (a_edge))
        do

            -- Todo: complete implementation

        ensure
            a_edge.destination ~ Current implies incoming.count = old incoming.count + 1
            a_edge.source ~ Current implies outgoing.count = old outgoing.count + 1
            a_edge.destination ~ Current implies incoming.has (a_edge)
            a_edge.source ~ Current implies outgoing.has (a_edge)
            -- incomplete, to add!
        end
```

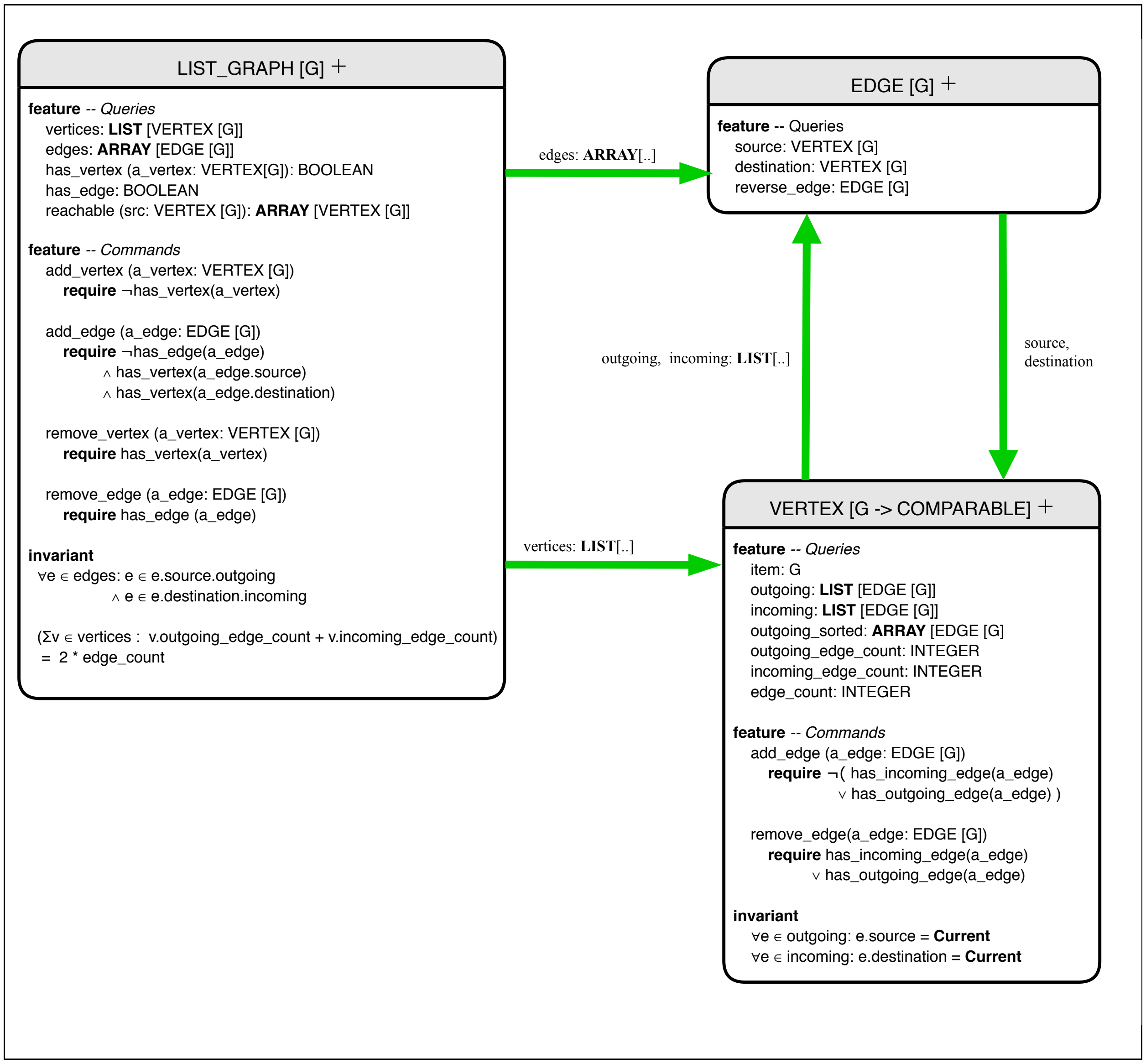
Figure 4 Contracts of {VERTEX}.add_edge

In this lab we have used classical contracts that only use available queries. With Mathmodels, we can provide complete contracts (which we will explore in lab 2). In fact, we may omit the classical contracts altogether.

2.6 Design architecture: BON diagram using the draw.io Template

The IDE Drawing tool is a good starting point for the BON class diagram. But we obtain a better view of the design using the draw.io tool.[3]
The draw.io diagram is constructed manually, which allows for the selective provision of classes, their relationships, their features, their signatures, their contracts and class invariants. The design architecture is thereby better described.

[3] http://seldoc.eecs.yorku.ca/doku.php/eiffel/faq/bon



LIST_GRAPH [G] +
feature -- Queries
vertices: LIST [VERTEX [G]]
edges: ARRAY [EDGE [G]]
has_vertex (a_vertex: VERTEX[G]): BOOLEAN
has_edge: BOOLEAN
reachable (src: VERTEX [G]): ARRAY [VERTEX [G]]
feature -- Commands
add_vertex (a_vertex: VERTEX [G])
require ¬has_vertex(a_vertex)
add_edge (a_edge: EDGE [G])
require ¬has_edge(a_edge)
∧ has_vertex(a_edge.source)
∧ has_vertex(a_edge.destination)
remove_vertex (a_vertex: VERTEX [G])
require has_vertex(a_vertex)
remove_edge (a_edge: EDGE [G])
require has_edge (a_edge)
invariant
∀e ∈ edges: e ∈ e.source.outgoing
∧ e ∈ e.destination.incoming
(Σv ∈ vertices : v.outgoing_edge_count + v.incoming_edge_count)
= 2 * edge_count
edges: ARRAY[..]
vertices: LIST[..]
EDGE [G] +
feature -- Queries
source: VERTEX [G]
destination: VERTEX [G]
reverse_edge: EDGE [G]
outgoing, incoming: LIST[..]
source, destination
VERTEX [G -> COMPARABLE] +
feature -- Queries
item: G
outgoing: LIST [EDGE [G]]
incoming: LIST [EDGE [G]]
outgoing_sorted: ARRAY [EDGE [G]
outgoing_edge_count: INTEGER
incoming_edge_count: INTEGER
edge_count: INTEGER
feature -- Commands
add_edge (a_edge: EDGE [G])
require ¬(has_incoming_edge(a_edge)
∨ has_outgoing_edge(a_edge))
remove_edge(a_edge: EDGE [G])
require has_incoming_edge(a_edge)
∨ has_outgoing_edge(a_edge)
invariant
∀e ∈ outgoing: e.source = Current
∀e ∈ incoming: e.destination = Current

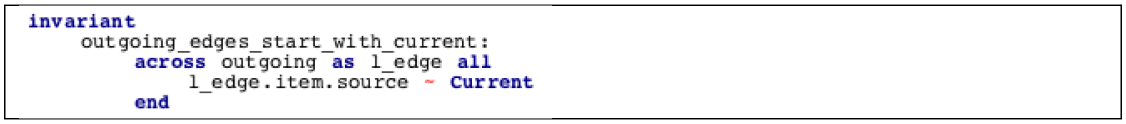
Figure 5 BON class diagram (draw.io template)

2.7 Design by Contract (DbC), Class invariants and Iterator Design Pattern

Figure 4 provides contracts for command *add_edge* of VERTEX. This class also has *invariants* that must be satisfied by all routines (whether function routines or command routines). For example, below a class invariant is:

```
∀e ∈ outgoing: e.source = Current
```

The invariant asserts that each outgoing edge has Current as the source vertex. In Eiffel, the invariant is written as shown below in Figure 6.



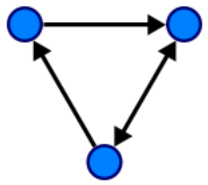
```
invariant
    outgoing_edges_start_with_current:
        across outgoing as l_edge all
            l_edge.item.source ~ Current
        end
```

Figure 6 Class invariant for VERTEX, described in Eiffel across notation

The universal quantifier ∀n is written using the **across** construct. This construct is based on the *iterator design pattern* (to be discussed in further detail in class).

3 List Graph Data Structures

You may recall graphs from your Data Structure and Algorithms courses. List graphs are graphs that store the graph information as an *adjacency list*. They allow for a dynamic kind of data representation and are used for finding shortest paths through networks, max flow through a network, and much more.



From Wikipedia

Graphs come in different varieties, but in this implementation, we will be using a directed graph. This means if vertex A is connected to vertex B, it is not necessarily the case that B is also connected to A. This kind of graph could be used for example to model course dependencies for a student enrolled in university. In order to take EECS 3311, they must first have taken EECS 2011.

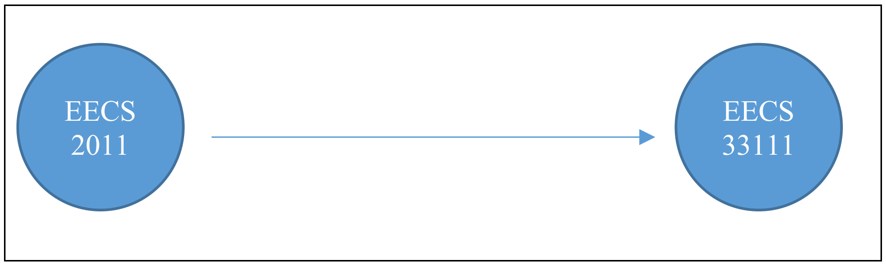
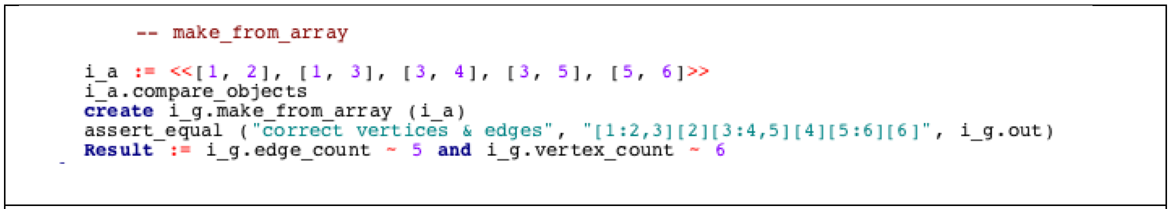


Figure 7 Course Dependency Graph

3.1 Testing class VERTEX and LIST_GRAPH

The test classes contain many different features that each test a different aspect of your implementation. In each case, class VERTEX is used in combination with EDGE to construct LIST_GRAPH structures.

Consider the following graph structure built using LIST_GRAPH [G]:



```
        -- make_from_array

    i_a := <<[1, 2], [1, 3], [3, 4], [3, 5], [5, 6]>>
    i_a.compare_objects
    create i_g.make_from_array (i_a)
    assert_equal ("correct vertices & edges", "[1:2,3][2][3:4,5][4][5:6][6]", i_g.out)
    Result := i_g.edge_count ~ 5 and i_g.vertex_count ~ 6
```

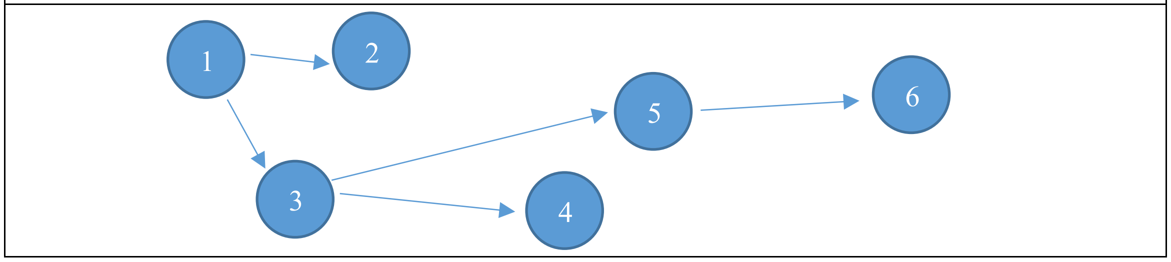
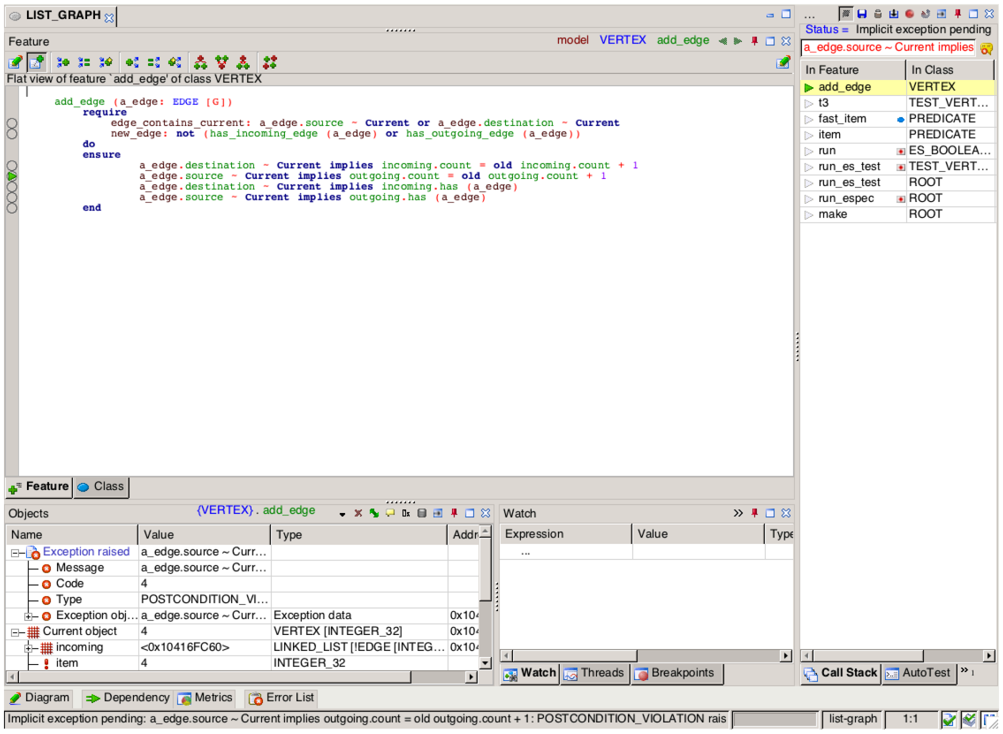


Figure 8 Test graph of integers

The test class uses the {LIST_GRAPH}.out query to ensure that the graph has inserted the vertices and edges into the graph and prints it out in the correct order.

3.2 Initially Tests Fail

If you execute the tests (F5) then the debugger will halt at a failing test as shown below. You must learn how to use the debugger (see Eiffel101).



LIST_GRAPH

Feature — model VERTEX add_edge

Flat view of feature `add_edge' of class VERTEX

```
add_edge (a_edge: EDGE [G])
    require
        edge_contains_current: a_edge.source ~ Current or a_edge.destination ~ Current
        new_edge: not (has_incoming_edge (a_edge) or has_outgoing_edge (a_edge))
    do
    ensure
            a_edge.destination ~ Current implies incoming.count = old incoming.count + 1
            a_edge.source ~ Current implies outgoing.count = old outgoing.count + 1
            a_edge.destination ~ Current implies incoming.has (a_edge)
            a_edge.source ~ Current implies outgoing.has (a_edge)
    end
```

Status = Implicit exception pending

a_edge.source ~ Current implies

In Feature	In Class
add_edge	VERTEX
t3	TEST_VERT…
fast_item	PREDICATE
item	PREDICATE
run	ES_BOOLEA…
run_es_test	TEST_VERT…
run_es_test	ROOT
run_espec	ROOT
make	ROOT

Objects — {VERTEX} . add_edge

Name	Value	Type	Addr
Exception raised	a_edge.source ~ Curr…		
Message	a_edge.source ~ Curr…		
Code	4		
Type	POSTCONDITION_VI…		
Exception obj…	a_edge.source ~ Curr…	Exception data	0x104
Current object	4	VERTEX [INTEGER_32]	0x104
incoming	<0x10416FC60>	LINKED_LIST [!EDGE [INTEG…	0x104
item	4	INTEGER_32	

Watch

Expression	Value	Type
…		

```
Implicit exception pending: a_edge.source ~ Current implies outgoing.count = old outgoing.count + 1: POSTCONDITION_VIOLATION rais    list-graph    1:1
```

Figure 9 Running the tests at the start

3.3 Void safety

You are required to read the section on Void Safety in Eiffel-101 to understand the keywords **attached** and **detachable**.

4 To Submit

1. Add correct implementations 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 5 tests of your own to STUDENT_TEST, 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 *list-graph* again to remove all EIFGENs.

Submit your Lab from the command line as follows:

```
submit 3311 Lab1 list-graph
```

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 *list-graph* **must** be a superset of Table 1.