

# EECS-3311 – Lab – Doubly-Linked-List

**Not for distribution.** By retrieving this Lab and using this document you affirm that you are registered in EECS3311 at York University this term. Registered EECS3311 students may download the Lab for their private use, but may not communicate it to anyone else, even other students in the class. Each student must submit their own work thus complying with York academic integrity guidelines. See course wiki for details of academic integrity principles.

1	Prerequisite	2
2	Introduction	2
3	Design Decisions	3
3.1	BON (and UML)	3
3.2	A design decision	3
3.3	Information Hiding, Abstraction and Contracts	5
3.4	Void safety	8
4	Another design decision	9
4.1.1	Correctness (Contract View)	9
5	Getting Started on the Lab	11
5.1	Format for Unit Tests	13
6	What you must do	14
7	Appendix	15
7.1	A1 Reference vs. Expanded Types	15
7.2	A2 Using the debugger	16
7.3	Object vs. Reference	16
7.4	Class LIST[G] with generic parameter G	16
7.5	Testing	17
7.6	Compile Time Errors	17
7.7	Bugs	18
7.8	Iteration using the “across” notation	19
7.9	Debugging	20
7.10	Using unit tests and the debugger	21
7.11	BON/UML diagrams	22
7.12	Can you improve the comment?	23

## 1 Prerequisite

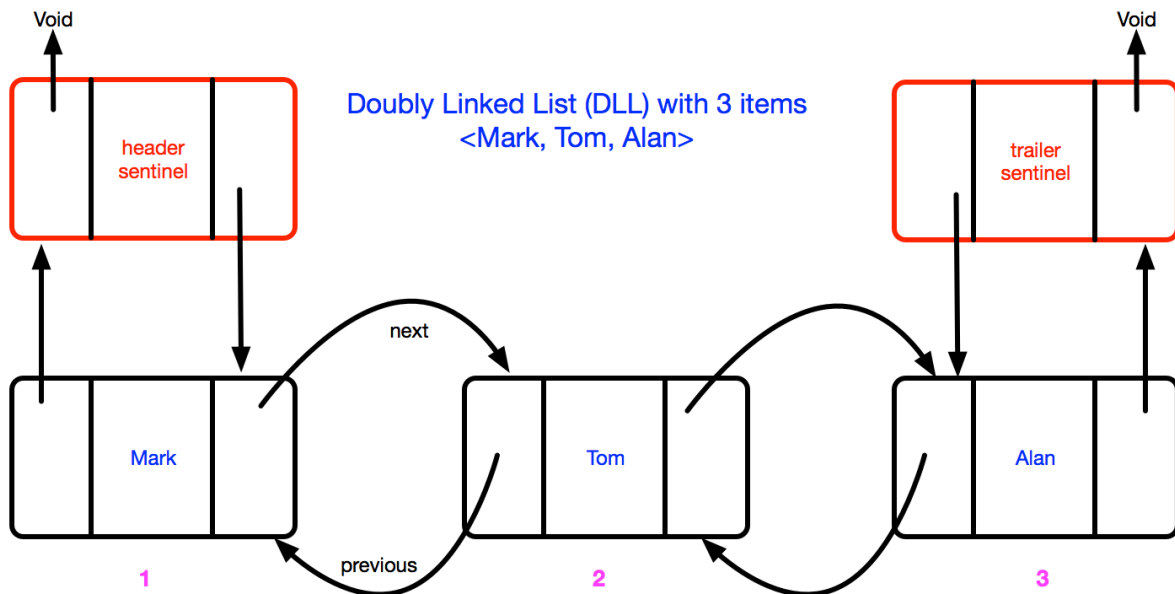
It is assumed that in the first week of the course, and during the Lab in the first week, you have:

- studied basic Eiffel syntax,
- used the EiffelStudio IDE and its debugger
- used the unit testing library ESPEC to regression test code

See *Getting Started* at: <http://eiffel.eecs.yorku.ca/>

## 2 Introduction

You recall from previous courses, that a doubly linked list (DLL) is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, that are references to the *previous* and to the *next* node in the sequence of nodes. A node also contains some element of a given type. The beginning and ending nodes' *previous* and *next* links, respectively, point to some kind of terminator, called a sentinel node (it might also be Void), to facilitate traversal of the list.



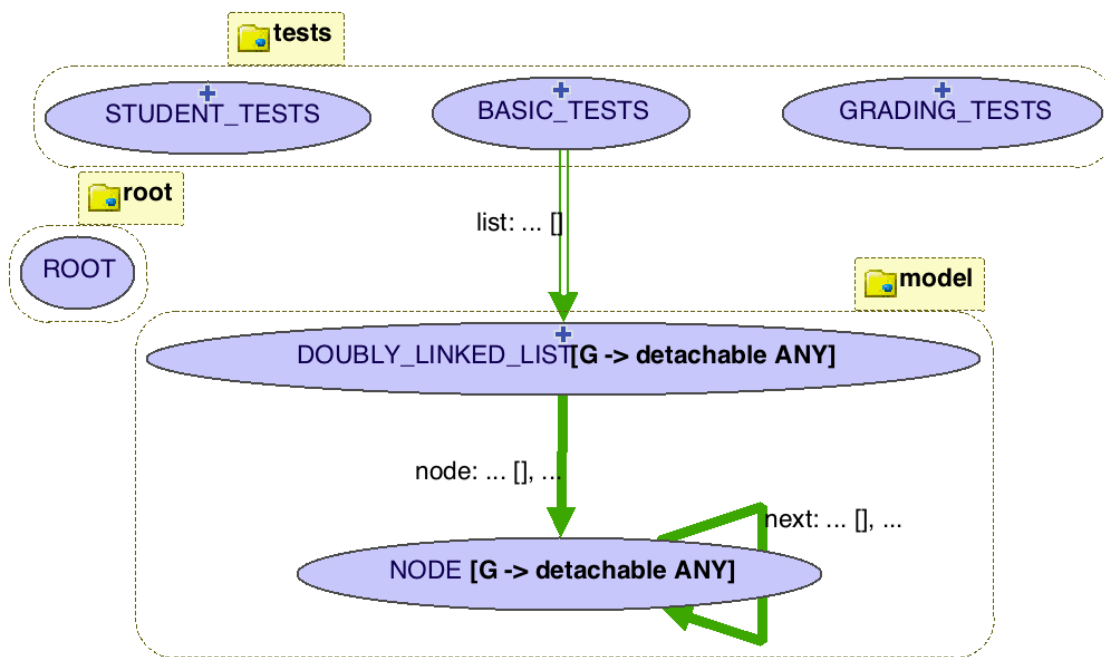
We will use doubly linked list to illustrate the following learning outcomes for this Lab:

- Use **genericity** and **void safe programming constructs (attached and detachable)** for abstraction and reliability
- Use **Design by Contract** (preconditions, postconditions and class invariants) for specifying API's and **Unit Testing** (test driven development and regression testing) for developing reliable code
- Understand **information hiding** and the concept of **making design decisions**

### 3 Design Decisions

#### 3.1 BON (and UML)

The EiffelStudio IDE can automatically generate a BON (or UML) class diagram from the code. The synchronization is useful as the architecture may keep changing. The significance of the diagram is that it describes the system *architecture*, a major aspect of design.



#### 3.2 A design decision

The two node links (*previous* and *next*) allow traversal of a list in either direction. While adding or removing a node in a doubly linked list (DLL) requires changing more links than the same operations on a singly linked list (SLL), the operations are simpler and potentially more efficient (for nodes other than first nodes) because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.

As an example consider removing the last node “Alan” (node 3) in the earlier diagram. In a singly linked list (SLL), this is an  $O(n)$  operation (if there are  $n$  nodes in the list). One must traverse the list from the *header* to the end to remove the node. In a DLL, this is an  $O(1)$  operation: one accesses the *trailer* directly and removes the previous node.

One may (1) have actual sentinel nodes as shown in the diagram or (2) allow the sentinel nodes to be *Void* (i.e. null). Consider commands such as *add\_after* to add a node, shown below. For choice #2, we would need extra logic for each such command to check if the next node is *Void*:

```

trailer: NODE[G]                -- attribute

add_after(node: NODE[G]; e: G)  -- command
  -- add node with element `e` after `node`
  local
    new_node: NODE[G]
  do
    create new_node
    new_node.previous := node    -- not allowed in Eiffel
    if node.next = Void then    -- needed if sentinel is Void
      new_node.next := Void     -- not allowed in Eiffel
    else
      new_node.next := node.next -- not allowed in Eiffel
      node.next.previous := new_node -- not allowed in Eiffel
    end
    new_node.element := e       -- not allowed in Eiffel
    node.next := new_node      -- breaks information hiding
  end
end

```

For choice #1 (there are actual header and trailer sentinel nodes) the check for *Void* is not needed and the code simplifies to:

```

trailer: NODE[G]                                -- attribute

add_after(node: NODE[G]; e: G)                  -- command
  -- add node with element `e` after `node`
  local
    new_node: NODE[G]
  do
    create new_node
    new_node.previous := node                    -- not allowed in Eiffel
    new_node.next := node.next                  -- not allowed in Eiffel
    node.next.previous := new_node              -- not allowed in Eiffel
    new_node.element := e                      -- not allowed in Eiffel
    node.next := new_node                      -- breaks information hiding
  end
end

```

The cost of choice #1 is only two extra nodes. Thus it is reasonable to choose #1 to obtain simpler logic in the code making it easier to understand and test (there are fewer branches in the code that need to be tested).

### 3.3 Information Hiding, Abstraction and Contracts

The red comments in the above code note that one cannot set attributes of an instance of class NODE (such as its *previous* and *next* links) outside of node. This is because class NODE is also a module and each module needs to control its own state (attributes) in order to ensure that the routine contracts and class invariants are not violated. If an external source can arbitrarily change the state of a module, then the consistency of the module state cannot be guaranteed. For this reason, an external source can read a public query (attribute or function routine) of a class but cannot write to the attribute (there must be a setter, but no need for a getter<sup>1</sup>).

Information hiding and contracts are shown for class/module NODE on the next page. Ensure that you understand the use and significance of generic parameter G, the attached keyword and construct and the contracts (preconditions, postconditions and class invariant).<sup>2</sup>

<sup>1</sup> In languages such as Java which allow arbitrary state change of public methods, we would need to make the attribute private and then add a setter and getter.

<sup>2</sup> Read the relevant chapters in the texts TOC and OOSC2 for more information. See the entry on Void safety at <http://seldoc.eecs.yorku.ca/doku.php/eiffel/faq/faq/void-safety/start>

```

note
  description: "[
    A node for a doubly-linked list stores:
    -- a reference to an element of a sequence; might be Void
    -- a reference to the next node; might be Void
    -- a reference to the previous node; might be Voids
  ]"

class
  NODE [G -> detachable ANY]
create
  make
feature
  element: detachable G
  previous: detachable NODE [G]
  next: detachable NODE [G]
feature {NODE} -- Constructor
  make (e: detachable G; p: detachable NODE [G]; n: detachable NODE [G])
    -- make a node with previous node p and next node n
  do
    element := e; previous := p; next := n
  ensure
    item_1: element = e and previous = p and next = n
  end
feature {DOUBLY_LINKED_LIST, ES_TEST}
  set_element (e: detachable G)
  do
    element := e
  ensure
    comment ("only element changes. Also, see invariant")
    element_changed: element = e
    previous_unchanged1:
      attached (old previous) as old_previous implies
      attached previous as new_previous
      and then old_previous.element = new_previous.element
    previous_unchanged2:
      not attached (old previous) implies previous = Void
    next_unchanged1: attached (old next) as old_next implies
      attached next as old_next
      and then old_next.element = new_next.element
    next_unchanged2: not attached (old next) implies next = Void
  end
  set_prev (p: NODE [G])
  require
    ok_p: p /= Void implies p /= next
  do
    previous := p
  ensure
    previous_set: previous = p
  end
  set_next (n: NODE [G])
  require
    ok_p: n /= Void implies n /= previous
  do
    next := n
  ensure
    next_set: n = next
  end
invariant
  if_empty: previous = next implies previous = Void and next = Void
end -- class NODE

```

- The use of generic parameter `G` provides a form of abstraction in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters (e.g. `G` can be an *expanded* type such as `INTEGER`, a *reference* type such as `STRING` or any other types constructed by the programmer). Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. The code for each type is virtually identical and thus we write the code once and re-use for all possible element types that may be needed.<sup>3</sup>
- The fields of a node need to be **detachable** as they may be `Void` (e.g. see the sentinel nodes in the diagram).
- We use contracts to specify the API and correctness of class `NODE`. For example:
  - The precondition of command `set_prev(p: NODE[G])` requires that a client cannot set the previous node to also be the next node (which would be circular).
  - The postcondition of command `set_element` ensures that only the element attribute is changed, while all other attributes (such as previous and next are changed). Ensure that you understand these contracts.
  - All commands must satisfy the class invariant which ensures that nodes are arranged sequentially and not circularly. The preconditions must be strong enough to ensure that the class safety invariants can be maintained.
  - Any class that inherits from `NODE` or `DOUBLY_LINKED_LIST` will also be required to satisfy the parent contracts (this is called **subcontracting**, discussed in OOSC2).
- The commands of `NODE` are *exported* only to class `DOUBLY_LINKED_LIST`, thus preventing other classes from changing the data.<sup>4</sup> The commands are also exported to testing classes. This together with the read-only approach to public attributes ensures *information hiding*.

<sup>3</sup> See [https://en.wikipedia.org/wiki/Generic\\_programming](https://en.wikipedia.org/wiki/Generic_programming).

<sup>4</sup> In Java-like languages one does not have selective export, only public, private and protected.

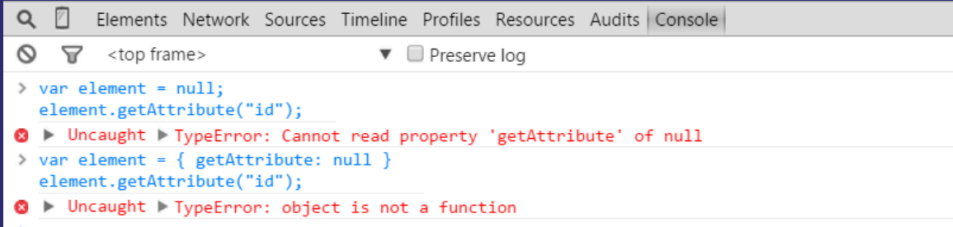
### 3.4 Void safety

**JavaScript**

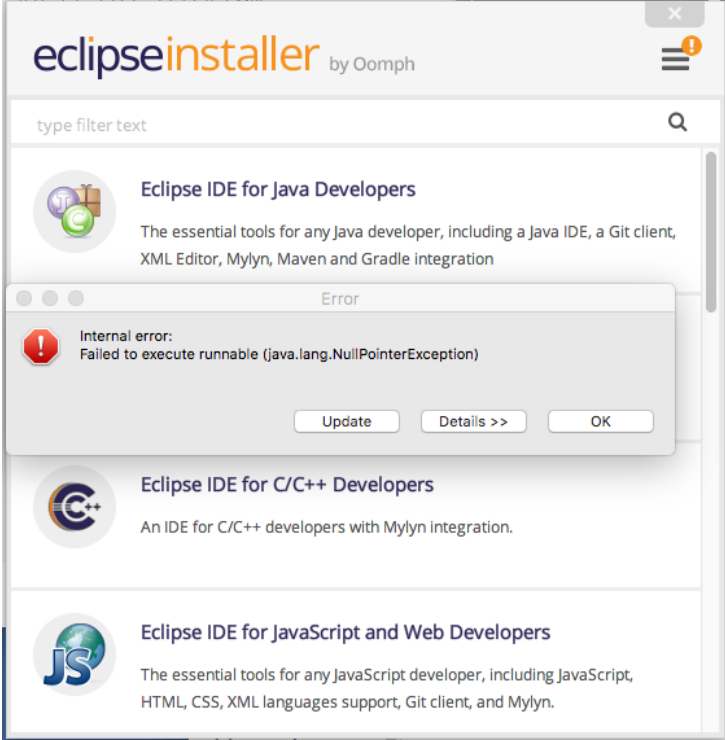
```
var element = null;
element.getAttribute("id");
// crash

var element = { getAttribute:null }
element.getAttribute("id");
// crash
```

**Errors in a browser console:**



**Void safety** is important for avoiding null pointer exceptions at *runtime*:



In a 2009 talk, Tony Hoare traced the invention of the *null* pointer to his design of the Algol W language and called it a "mistake":

“I call it my billion-dollar mistake. It was the invention of the *null* reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. *This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*”



Void-safety, like static typing, is another facility for improving software quality. **Void-safe software is protected from run time errors caused by calls to void references, and therefore will be more reliable than software in which calls to void targets can occur.** The analogy to static typing is a useful one. In fact, void-safe capability could be seen as an extension to the type system, or a step beyond static typing, because the mechanism for ensuring void-safety is integrated into the type system. The guard against void target calls can be seen by way of the notion of attachment and (by extension) detachment (e.g. **detachable** keyword).<sup>5</sup>

## 4 Another design decision

The use of a generic parameter *G* in `NODE [G]` is a form of *abstraction by parameterization*. We seek generality by allowing the same mechanism (algorithm) to be adapted to many different contexts by providing it with information on that context.

There is also *abstraction by specification* where we ignore implementation details, and agree to treat as acceptable any implementation that adheres to the specification. Using a procedure (routine, method) is just that type of abstraction in which we

A major benefit of these abstractions is re-use. Abstraction by specification helps lessen the work required when we need to modify a program. By choosing our abstractions carefully, we can gracefully handle anticipated changes to hide the details of things that we anticipate changing frequently. When the changes occur, we only need to modify the implementations of those abstractions. Clients are unchanged relying on the specification alone for their client code.

### 4.1.1 Correctness (Contract View)

What we are looking for are clean API's that make sense.

- We use contracts to specify the a system, in a way that is free of implementation detail.
- We will want many tests to exercise the contracts to ensure that the implementations of the routines satisfy the specifications.
- **Invariants are very important in constraining objects to remain safe.** On the next page we show the **contract view** generated automatically.
- Note that the class is self-documenting. There is an indexing clause to explain the purpose of the class and each feature has a meaningful comment.

---

<sup>5</sup> [https://en.wikipedia.org/wiki/Eiffel\\_\(programming\\_language\) - Void-safety](https://en.wikipedia.org/wiki/Eiffel_(programming_language) - Void-safety).

This approach has now been adopted by other languages such as C# and Go.

```

note
    description: "[
        A bank account with deposit and withdraw
        operations. A bank account may not have a negative balance.
    ]"
    author: "JSO"
class interface
    ACCOUNT
create
    make_with_name(a_name: STRING)
        -- create an account for `a_name` with zero balance
    ensure
        created: name == a_name
        balance_zero: balance = balance.zero
feature -- Account Attributes
    name: STRING

    balance: VALUE
feature -- Commands
    deposit (v: VALUE)
        require
            positive: v > v.zero
        ensure
            correct_balance: balance = old balance + v

    withdraw (v: VALUE)
        require
            positive: v > v.zero
            balance - v >= v.zero
        ensure
            correct_balance: balance = old balance - v
feature -- Queries of Comparison
    is_equal (other: like Current): BOOLEAN
        -- Is `other` value equal to current
    ens then
        Result (name ~ other.name and balance = other.balance)
    is_less alias < (other: like Current): BOOLEAN
        -- Is current object less than `other`?
    ensure then
        Result = (name < other.name)
        or else (name ~ other.name and balance < other.balance)
invariant
    balance_non_negative: balance >= balance.zero
end -- class ACCOUNT

```

## 5 Getting Started on the Lab

Given the starter project in the folder “dll”, there are only two classes that you will edit:

- DOUBLY\_LINKED\_LIST (add some implementations and some contracts)
- STUDENT\_TESTS (add at least 5 tests of your own)

When you **compile (F7)** and **workbench-execute (Control-Alt-F5)** the unit tests for the project, you will see something like this in your browser:

Note: \* indicates a violation test case

FAILED (10 failed & 9 passed out of 19)		
Case Type	Passed	Total
Violation	2	2
Boolean	7	17
All Cases	9	19
State	Contract Violation	Test Name
Test1	BASIC_TESTS	
PASSED	NONE	t0: test a doubly linked list with strings test make_empty, add_first, item, node and replace
PASSED	NONE	t1: test for node structure alan -> mark -> tom
PASSED	NONE	t2: test a doubly linked list with strings
PASSED	NONE	t3: test a doubly linked list with integers 6->8->10 check that all elements in list >= 6 and even use across notation
PASSED	NONE	t4: test replace at index 3 with Dan. list ==> . replace is also used by item.
PASSED	NONE	t5: test add_between two nodes with Alan and Mark. list ==>
PASSED	NONE	*t4v: remove_first empty list precondition violation
PASSED	NONE	*t6v: remove_last empty list precondition violation
Test2	STUDENT_TESTS	
FAILED	NONE	t1: describe test t1 here
Test3	GRADING_TESTS	
FAILED	Postcondition violated.	t4: remove first in Alan->Mark->Tom
FAILED	Postcondition violated.	t5: remove 8, 10, 6 in 6->8->10
FAILED	Postcondition violated.	t6: test add last Alan->Mark->Tom
FAILED	Postcondition violated.	t7: test add_before
FAILED	Postcondition violated.	t8: test add_before with Void
FAILED	Postcondition violated.	t9: test add_after and attribute node
FAILED	Postcondition violated.	t10: test add_at(i)
FAILED	Postcondition violated.	t11: test add_at(i)
PASSED	NONE	t12: test remove_at(i)
FAILED	Postcondition violated.	t13: test add last Alan->Mark->Void

The postconditions are violated because some (or all) of the implementation in class `DOUBLY_LINKED_LIST` is missing. Your job will be to complete the implementation to ensure that the implementation satisfies the specification. This is called *correctness* another important aspect of design.

Eventually, you must get all the tests to run and also add some tests of your own.

A quick way to get to the problem is to use the debugger. **Execute the code (F5) and then you should break into the debugger and see where the contract fails to hold.** This is called runtime assertion checking. The debugger view is shown below (for a different project). Notice the call stack (see note A). One level down is the test `t2` (if you click on `t2`, you are taken to the position in test `t2` where the contract failure occurred).

The screenshot shows the debugger interface with the following components:

- Feature View:** Displays the source code for the `equal_values` function of the `BANK_TEST_INSTRUCTOR1` class. The code includes a `local` block with `l_equal` and `i` variables, a `do` loop, and an `ensure` statement. Red arrows point to the `ensure` statement (B) and the `l_equal` variable (C).
- Call Stack:** Shows the sequence of function calls. The top frame is `equal_values` in `BANK_TEST_INSTRUCTOR1`. Below it is `t2` in `BANK_TEST_INSTRUCTOR1`. Red arrow A points to the `t2` frame.
- Objects:** A table showing the state of objects. The `Result` variable is `True`, which is labeled as an incorrect result (C).
 

Name	Value	Type	Address
Exception raised	values_are_equal: PO...		
Current object	<0x111C9A070>	BANK_TEST_INSTRUCTOR1	0x111C9A070
Arguments			
v1	123.00	VALUE	0x111C9A100
v2	213.00	VALUE	0x111C9A108
Locals			
i	4	INTEGER_32	
l_equal	True	BOOLEAN	
Result	True	BOOLEAN	
- Watch:** A table showing the values of expressions. The `v1 = v2` expression is `False`, which is labeled as the correct result (D).
 

Expression	Value	Type	Address
v1 = v2	False	BOOLEAN	
v1 ~ v2	False	BOOLEAN	

Figure 1: Using the Debugger to trace problem in test `t2`

Test `t2` fails because there is a bug in the query `{BANK_TEST_INSTRUCTOR1}.equal_values`.

**Before running the tests, always freeze first (Control-F7).** The browser will show a red or green bar and the console will show something like this:

```
passing tests
> BASIC_TESTS.t0
```

```

> BASIC_TESTS.t1
> BASIC_TESTS.t2
> BASIC_TESTS.t3
> BASIC_TESTS.t4
> BASIC_TESTS.t5
> BASIC_TESTS.t4v
> BASIC_TESTS.t6v
> GRADING_TESTS.t12
failing tests
> STUDENT_TESTS.t1
> GRADING_TESTS.t4
> GRADING_TESTS.t5
> GRADING_TESTS.t6
> GRADING_TESTS.t7
> GRADING_TESTS.t8
> GRADING_TESTS.t9
> GRADING_TESTS.t10
> GRADING_TESTS.t11
> GRADING_TESTS.t13
9/19 passed
failed

```

## 5.1 Format for Unit Tests

Unit tests must always have a comment as shown below that starts as follows:

"t4: some description ..."

```

t4: BOOLEAN
    -- use list from setup
    do
        comment("t4: test replace at index 3 with Dan.")
        sub_comment ("list <Alan, Mark, Tom> ==> <Alan, Mark, Dan>.")
        sub_comment ("replace is also used by item.")
        Result := list[3] ~ "Tom"
        check Result end
        list.replace("Dan", 3)
        assert_equal ("replace failed", list[3], "Dan")
    end

```

I used the latest GUI version of VIM to produce the above syntax highlighting.<sup>6</sup>

<sup>6</sup> In VIM, Syntax => Convert to HTML. Then copy from the browser and paste in MS Word. The IDE can generate larger scale documentation. You can setup *gvim* as your external editor in the EStudio IDE. You will need to do this in later documentation. Gedit also has Eiffel syntax highlighting.

## 6 What you must do

1. Add **correct implementations** and **additional contracts** to **DOUBLY\_LINKED\_LIST**.
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 TESTS**, i.e. don't just rely on our tests.
4. Don't make any changes to **NODE** or anything **g** else.
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 *dll* again to remove all EIFGENs.
3. Submit your Lab as follows: `submit 3311 Lab1 dll`

### 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 folder *dll* **must** contain the following<sup>7</sup>:

```

dll
├── doubly-linked-list.ecf
├── model
│   ├── doubly_linked_list.e
│   └── node.e
├── root
│   └── root.e
├── tests
│   ├── instructor
│   │   ├── basic_tests.e
│   │   └── grading_tests.e
│   └── student
│       └── student_tests.e

```

<sup>7</sup> Must be a superset of the following.

## 7 Appendix

### 7.1 A1 Reference vs. Expanded Types

If in some class we declare

```
a, b: ACCOUNT
...
create a.make_with_name("Steve")
...
b := a
```

then, after creation, variable `a` points to an object which is an instance of type `ACCOUNT`.

When the assignment `b := a` is done, then variable `b` also points to the same object that `a` points to. We now have *aliasing* because doing `a.deposit("420.10")` also changes `b`. We are using a *reference semantics*. In a reference semantics it is possible that a variable may not refer to an object but be *Void*.<sup>8</sup>

However, variables of the basic types `INTEGER`, `BOOLEAN`, `REAL` and `CHARACTER` do not follow a reference semantics. Rather they follow a *value semantics*. To obtain a value semantics Eiffel uses the notion of an **expanded** types.

```
i, j: INTEGER
...
i := 4
j := i
i := 5
```

For an expanded type, assignment does a copy not a reference. So, for `j := i` a copy of the value of `i` is provided for `j`. Thus, the subsequent assignment `i := 5` does not change the value of `j` (there is no aliasing). Also, there is no need to create `i` and `j` as they have default values 0. Expanded types must thus have a default creation procedure so that its value is always well-defined (and thus `i` and `j` will never have a value `Void`).

You can read more about reference and expanded types in OOSC2 sections 8.1 to 8.8. These sections also explain copy (*twin*) and deep copy (*deep\_twin*). Many of the notions in these sections should already be familiar to you from earlier courses. Expanded types are discussed in Section 8.7.

**In Java, C#, C++ etc. developers may not (directly) create their own expanded types. By contrast, Eiffel allows developers to create their own classes with a value semantics by using the expanded construct.**

In this assignment, you will be using expanded class `VALUE` which does precise arithmetic needed for the bank system. Eiffel also provides an infix notation so that we can use the regular arithmetic operators such as `+`, `-`, `*` and `/`.

---

<sup>8</sup> *null* in other languages.

## 7.2 A2 Using the debugger

The first two tests *t1* and *t2* help you to understand how to use the expanded class VALUE.

One way to try to understand a new class as a client of that class, is to write some tests to confirm that you know how to use its features.

Running ESpec (*Workbench Run*, i.e. Control-Alt-F5) we see that test *t2* fails. How should we use the IDE to explore why the test is failing. This is where the debugger is useful.

- Do a *Plain Run* (F5) and the runtime will halt at a Postcondition violation.
- You can then use the debugger to examine the state of the system

Use the debugger for finding bugs. Below is the debugger display for test *t2*.

The screenshot shows the IDE debugger with the following components:

- Feature View:** Displays the source code for the `equal_values` routine of class `BANK_TEST_INSTRUCTOR1`. The code includes comments and logic for comparing two values.
- Call Stack:** Shows the sequence of calls leading to the current state. The top entry is `equal_values` in `BANK_TEST_INSTRUCTOR1`, which is the source of the violation.
- Exception:** A red arrow points to the line `values_are_equal: Result = equal_values_with_across (v1, v2)` in the source code, indicating the point of failure.
- Objects:** A table showing the state of objects at the time of the error.
 

Name	Value	Type	Address
Exception raised	values_are_equal: P...		
Current object	<0x11256AC18>	BANK_TEST_INSTRUCTOR1	0x11256AC18
Arguments			
v1	123.00	VALUE	0x11256AC70
v2	213.00	VALUE	0x11256AC78
Locals			
Result	True	BOOLEAN	
- Watch:** A table showing the values of expressions being monitored.
 

Expression	Value
i.item	Error occu
v1.precise_out[3]	51 '3'
v2.precise_out[3]	51 '3'
v1.precise_out[3] = v2.precise_out	False

Red arrows highlight the `equal_values` routine in the Call Stack and the corresponding line in the source code.

**Annotations:**

1. Postcondition Violated: thus supplier of "equal\_values" routine is guilty
2. The client was test *t2*
3. The postcondition that was violated
4. What you must do:
  - (a) Fix the body of the function routine "equal\_values" so that it satisfies its contract
  - (b) Get test *t2* to pass

The windows in this snapshot are provided by the estudio debugger. Using the debugger is essential to finding bugs and getting tests to work

## 7.3 Object vs. Reference

See OOSC2

## 7.4 Class LIST[G] with generic parameter G

See OOSC2



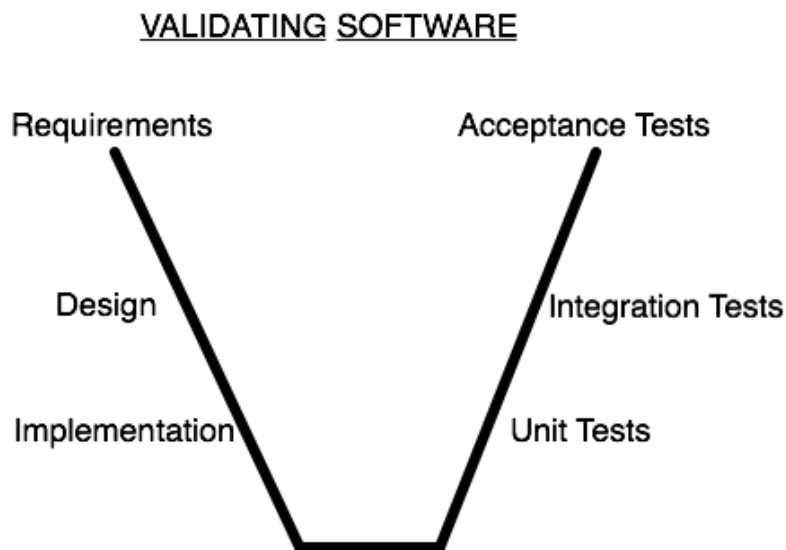
## 7.5 Testing

Much of this material comes from John Guttag's introductory text on Python. The ideas are applicable to testing code written in any language, not just Python.<sup>9</sup> Some information has been added, and the discussion is adapted to Eiffel.

Our programs don't always function properly the first time we run them. Books have been written about how to deal with this last problem, and there is a lot to be learned from reading these books. However, in the interest of providing you with some hints that might help you get that next problem set in on time, we provide a highly condensed discussion of the topic.

**Testing** is the process of running a program to try and ascertain whether or not it works as intended. **Debugging** is the process of trying to fix a program that you already know does not work as intended.

Testing and debugging are not processes that you should begin to think about after a program has been built. Good programmers design their programs in ways that make them easier to test and debug. The key to doing this is breaking the program up into components that can be implemented, tested, and debugged independently of each other. We need to test classes (modules) and their routines, but we also need to test sub-systems (clusters of classes) and the overall system (acceptance tests).



In the sequel, we will mostly be considering unit tests.

## 7.6 Compile Time Errors

The first step in getting a program to work is getting the language system to agree to run it—that is eliminating syntax errors and static semantic errors that can be detected without running the program. If you haven't gotten past that point in your programming, you're not ready for this appendix. Spend a bit more time working on small programs, and then come back.

---

<sup>9</sup>John V Guttag. Introduction to Computation and Programming Using Python, revised and expanded edition, MIT Press 2013.

The Eiffel compiler does a lot of checking at compile time, thus eliminating whole classes of errors before you run the program.

## 7.7 Bugs

The most important thing to say about testing is that its purpose is to show that bugs exist, not to show that a program is bug-free. To quote Edsger Dijkstra, “Program testing can be used to show the presence of bugs, but never to show their absence!” Or, as Albert Einstein reputedly once said, “No amount of experimentation can ever prove me right; a single experiment can prove me wrong.”

Why is this so? Even the simplest of programs has billions of possible inputs. Consider, for example, a program that purports to meet the specification:

*is\_bigger*(*x*,*y*: INTEGER): BOOLEAN

**ensure** *Result*  $\equiv x < y$

Before proceeding, provide below an informal English description of the specification<sup>10</sup>:

Running it on all pairs of integers would be, to say the least, tedious. The best we can do is to run it on pairs of integers that have a reasonable probability of producing the wrong answer if there is a bug in the program. The key to testing is finding a collection of inputs, called a test suite, that has a high likelihood of revealing bugs, yet does not take too long to run. The key to doing this is partitioning the space of all possible inputs into subsets that provide equivalent information about the correctness of the program, and then constructing a test suite that contains one input from each partition. (Usually, constructing such a test suite is not actually possible. Think of this as an unachievable ideal.)

A **partition** of a set divides that set into a collection of subsets such that each element of the original set belongs to exactly one of the subsets.

Consider, for example *is\_bigger*(*x*, *y*). The set of possible inputs is all pairwise combinations of integers. One way to partition this set is into these seven subsets:

- *x* positive and *y* positive
- *x* negative and *y* negative
- *x* positive, *y* negative
- *x* negative, *y* positive
- *x* = 0, *y* = 0
- *x* 0, *y*  $\neq$  0
- *x*  $\neq$  0, *y* = 0

If one tested the implementation on at least one value from each of these subsets, there would be reasonable probability (but no guarantee) of exposing a bug should it exist. For most programs, finding a good partitioning of the inputs is far easier said than done. Typically, people rely on heuristics based on exploring different paths through some combination of the code and the specifications. Heuristics based on exploring paths through the code fall into a class called glass-

<sup>10</sup> Answer: Assume *x* and *y* are integers. The query returns *True* if *x* is less than *y* and *False* otherwise.

box testing. Heuristics based on exploring paths through the specification fall into a class called black-box testing.

Please read Guttag's chapter on Testing for the rest.

## 7.8 Iteration using the “across” notation

Class STRING treats a string as a sequence of characters. So,

```
routine
  local
    s: STRING
  do
    s := "abc"
    check
      s[1] = 'a' and s[2] = 'b' and s[3] = 'c' and s.count = 3
    end
  end
end
```

So the string  $s$  is a function  $1..3 \rightarrow \text{CHARACTER}$ . The index  $i$  in  $s[i]$  must be a valid index so that  $i \in 1..3$ .

An alternative (but less efficient) way to have a string is to declare it as an array of characters. Generic classes such as `ARRAY[G]`, `LIST[G]`, `HASH_TABLE[G]` etc. all have iterators using the **across** notation. We may also use the across notation on `STRING` (given that it is a sequence of characters). Later we will see that we can equip our own collection classes with this form of iteration.

Please see <https://docs.eiffel.com/book/method/et-instructions> for how to use the **across** notation. Here is a simple example of using the across notation as a Boolean query.

The contract uses the **across** notation. Consider the following snippet of code:

```
word: ARRAY[CHARACTER]
test1, test2: BOOLEAN
make
do
  word := <<'h', 'e', 'l', 'l', 'o'>>
  test1 :=
    across word as ch all
      ch.item <= 'p'
    end
  test2 :=
    across word as ch all
      ch.item < 'o'
    end
end
```

In data structure collections such as `ARRAY [G]` and `LIST [G]`, we can use the **across** notation in contracts to represent quantifiers such as  $\forall$  and  $\exists$ . Thus *test1* asserts:

$$\forall ch \in \text{word}: ch \leq 'p'$$

which is true, and *test2* asserts that

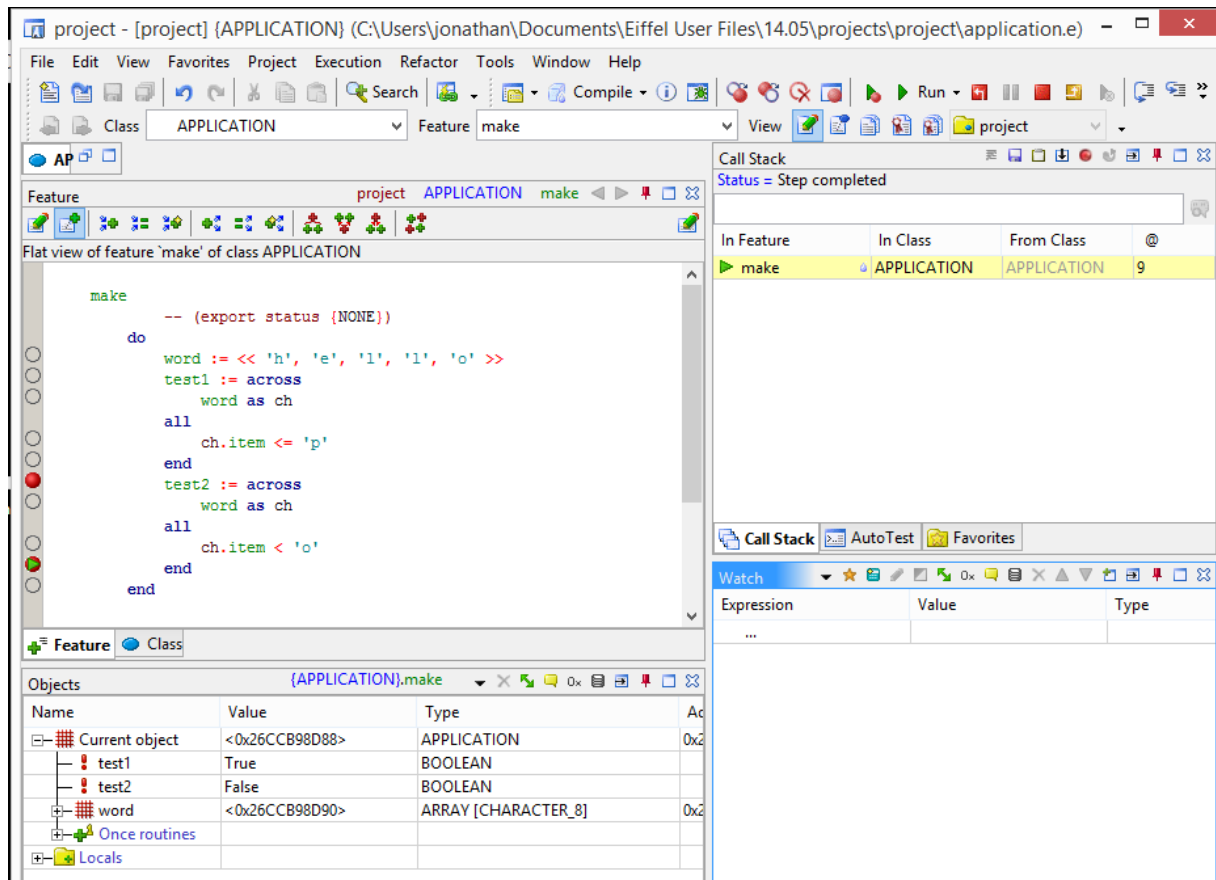
$$\forall ch \in \text{word}: ch < 'o'$$

which is false. The comparison ( $\leq$ ) is done using the ASCII codes of the character. Class CHARACTER inherits from COMPARABLE in order to allow the comparisons to be made.

We use the keyword **all** for  $\forall$  and **some** for  $\exists$ . Between **all** and **end** there must be an assertion (a predicate) that is either true or false.

We can also use the **across** notation for imperative code with the keyword **loop** instead of **all**. Between **loop** and **end** there can be regular implementation code including assignments.

In the figure below, we have placed breakpoints shown with red dots and we execute the code, using the debugging facilities to get to the breakpoints. After the debugger reaches the second breakpoint, the debugger shows that *test1* is *true* and *test2* is *false*.



Make sure you know how to set breakpoints and how to execute to reach the breakpoints.

## 7.9 Debugging

Debugging is a learned skill. Nobody does it well instinctively. The good news is that it's not hard to learn, and it is a transferable skill. The same skills used to debug software can be used to find out what is wrong with other complex systems, e.g., laboratory experiments or sick humans. For at least four decades people have been building tools called debuggers, and there are debugging tools built into *EiffelStudio*. These are supposed to help people find bugs in their programs. They can help, but they only take you part of the way. What's much more important is how you approach the problem. Some experienced programmers don't always bother with debugging tools, and they use only print statements. It is in your interest, though, to learn how to use the debugger and in most cases it is better than just using print statements.

Debugging starts when testing has demonstrated that the program behaves in undesirable ways. Debugging is the process of searching for an explanation of that behavior. The key to being consistently good at debugging is being systematic in conducting that search. Start by studying the available data. This includes the test results and the program text. Remember to study all of the test results. Examine not only the tests that revealed the presence of a problem, but also those tests that seemed to work perfectly. Trying to understand why one test worked and another did not is often illuminating. When looking at the program text, keep in mind that you don't completely understand it. If you did, there probably wouldn't be a bug.

Next, form a hypothesis that you believe to be consistent with all the data. The hypothesis could be as narrow as “if I change line 403 from  $x < y$  to  $x \leq y$ , the problem will go away” or as broad as “my program is not terminating because I have the wrong test in some while loop.” Next, design and run a repeatable experiment with the potential to refute the hypothesis. For example, you might put a print statement before and after each while loop. If these are always paired, then the hypothesis that a while loop is causing non-termination has been refuted. Decide before running the experiment how you would interpret various possible results. If you wait until after you run the experiment, you are more likely to fall prey to wishful thinking.

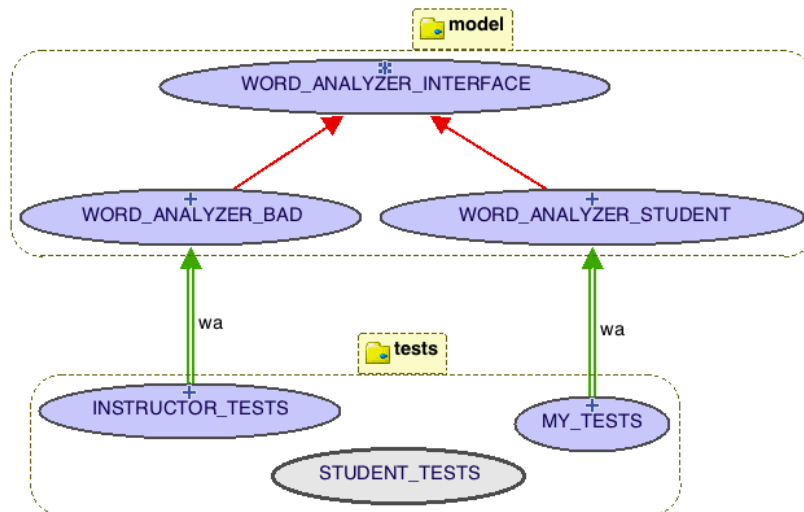
Finally, keep a record of what experiments you have run. This is particularly important. If you aren't careful, it is easy to waste countless hours trying the same experiment (or more likely an experiment that looks different but will give you the same information) over and over again. Remember, as many have said, “insanity is doing the same thing, over and over again, but expecting different results.”

## 7.10 Using unit tests and the debugger

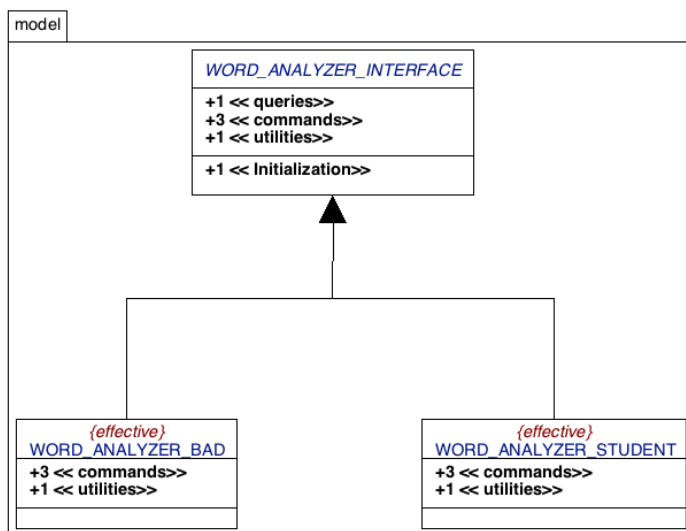
See <https://docs.eiffel.com/book/eiffelstudio/debugger>.

## 7.11 BON/UML diagrams

Here is a BON diagram (created with the EStudio IDE) showing both inheritance and client supplier relationships.



If you invoke the UML button in the diagram tool of the IDE, then you get



See the footnote for more information.<sup>11</sup> Please familiarize yourself with these notations.

<sup>11</sup> You might try to produce the BON diagram and also the UML diagram. Why do we use BON diagrams rather than the more standard UML notation?

(Hint: see the video: [https://wiki.eecs.yorku.ca/project/eiffel/start#eiffel\\_specifications\\_and\\_design](https://wiki.eecs.yorku.ca/project/eiffel/start#eiffel_specifications_and_design)).

Nevertheless, you should eventually familiarize yourself with UML – see <https://wiki.eecs.yorku.ca/project/eiffel/media/bon:uml.pdf>.

### 7.12 Can you improve the comment?

Consider the informal comment for the query below (acting as the query specification)

```
first_repeated_character: CHARACTER  
-- returns first repeated character or null character if none found  
-- a character is repeated if it occurs at least twice in adjacent positions  
-- e.g. 'l' is repeated in "hollow", but 'o' is not.
```

Before proceeding, in the space below, can you improve the comment<sup>12</sup>?

---

<sup>12</sup> Answer: “returns first repeated character of *word* or null character if none found”. Do you know how to enter “word” so that it shows up nicely in self-documentation mode?