L1

EECS-3311 Lab – Word Analyzer

Contents

Not for distribution. The student retrieving this document agree that this document and Lab may only be used by EECS3311 students in EECS at York University. Registered EECS3311 students may download the Lab for their private use, but may not communicate it to anyone else. Each student must submit their own work thus complying with York academic integrity guidelines. See course wiki for details.

1 Learning outcomes – purpose of Lab

- Develop the ability to use the EiffelStudio IDE for browsing, editing, compiling, executing and debugging code
- Use unit tests together with the debugger to track down and eliminate errors
- Write new unit tests, and use regression testing and test driven development to specify and verify the correctness of the code
- Document classes both textually and with BON/UML diagrams
- Some Design by Contract

All of the concepts you should have seen in prior courses. If not, this is the place where you need to ensure that you have mastered the concepts as they will be used repeatedly in the course.

The use of the tools (IDE for code browsing, editing, unit-testing, debugger, BON/UML diagrams, documentation, etc.) is new. You should master the use of the tools within the first two weeks of the course.

As mentioned in the course description, this course is work intensive (like many software development courses) and it is expected that you will be doing 10 hours of work per week.

What you must do is described in *Section 5*, and what you must **submit** is described in *Section 6*. The rest of this write-up is to help you understand and implement the learning outcomes.

2 Preparation before the Lab

Some students like to learn as they do, others like to read ahead. Here is some material to get you started before you do the lab, if you like to read ahead. If you decide to jump in head first, be sure to refer back to this material if you get stuck along the way; also, you should read master this material sometime during the first two weeks of the course.

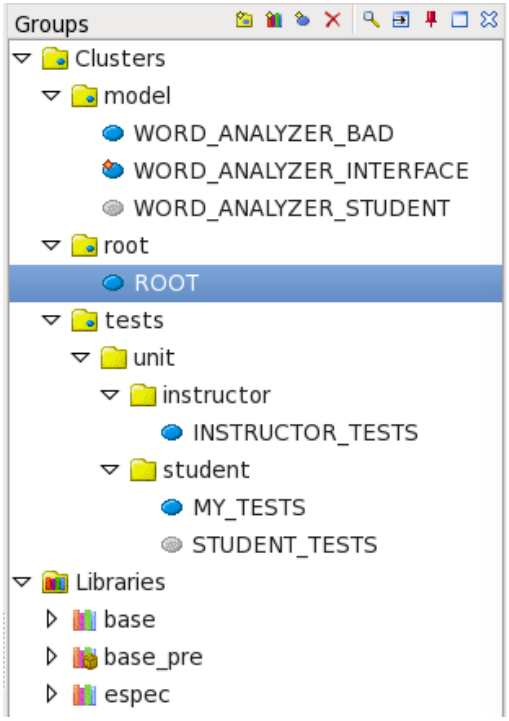
- Read https://wiki.eecs.yorku.ca/project/eiffel/getting_started.
  - Basic reading: *Eiffel Syntax* and *Eiffel Essentials* https://wiki.eecs.yorku.ca/project/eiffel/eiffel-language-basics
  - Try out the unit testing framework Espec
- Watch the EiffelStudio Overview Video (21 minutes) at: https://www.youtube.com/watch?v=nxt65skoH4o
- Read EiffelStudio Guided Tour up to (but not including) AutoTest: https://docs.eiffel.com/book/eiffelstudio/eiffelstudio-guided-tour
- Make sure that you know how to start a project, edit files (classes), create clusters, and use the debugger

3 Getting Started with the Lab

- Login to a Prism Linux workstation and do the following at the console:
  - **red%** *~sel/retrieve/3311/lab1*
  - This will download the directory *word-analyzer*. You do all your work in this directory.
- Unless instructed explicitly, you must make sure that you do not add files to -- or delete files from -- the *word-analyzer* directory. Once you submit, a check will first be performed to see if the expected file structure is still present: if not, then you will not receive a grade for this submission.
- Now do the following to compile the starter code (see footnote[1]):
  - *cd word-analyzer*
  - *estudioXX.YY word-analyzer.ecf &*
  - This will invoke the EStudio IDE and your lab will be compiled. In Eiffel, the code is ultimately compiled into C, which is stored in directory EIFGENs. There must be sufficient space (at least 300MB) in your account to store the EIFGENs.[2]

The organization of the project (and its clusters) is as follows:

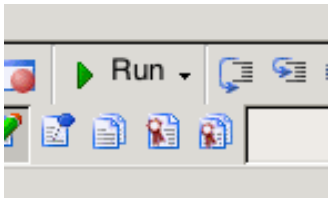
```
class
    ROOT
inherit
    ARGUMENTS
    ES_SUITE
create
    make

feature {NONE} -- Initialization

    make
            -- run student and instuctor tests
            -- instructor tests succeed
        do
            add_test (create {INSTRUCTOR_TESTS}.make)
            add_test (create {MY_TESTS}.make)
            show_browser
            run_espec
        end

end
```



[1] **Note**: On the Prism (Linux) systems and the SEL Virtual Machine there is no plain *estudio*. Use the version of *estudio* required by your course director, e.g. *estudio15.12*. For the command line compiler "ec" it would be *ec15.12*. If you install EStudio natively on your Windows or mac laptop from *eiffel.com*, then the IDE is invoked as "estudio" and the command line compiler as "ec".

[2] The command "quota -v" at the console will display your quota. You can also direct the compiler to store the EIFGENs folder in /tmp/$USER. First create /tmp/$USER, i.e. "mkdir /tmp/$USER". When your first invoke the IDE, it asks for a location for the EIFGENs folder" you enter */tmp/student* assuming your are user "student".

After a successful compile, if you press the green run button (keyboard shortcut F5)



then a browser window will display with the result of running all the Espec tests:

Test Run:12/22/2015 2:44:51.707 PM

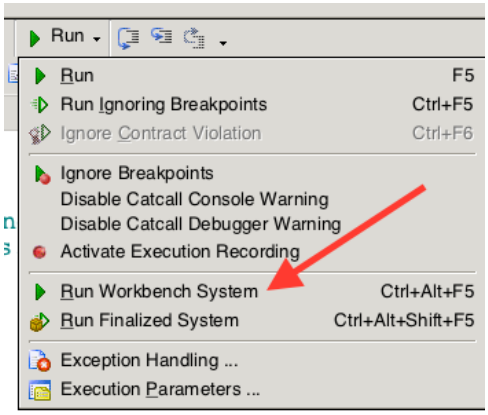
ROOT

Note: * indicates a violation test case

FAILED (10 failed & 3 passed out of 13)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	3	13
All Cases	3	13
State	**Contract Violation**	**Test Name**
Test1	**INSTRUCTOR_TESTS**	
PASSED	NONE	t_frc1: first repeated character for 'hollow' is 'l'
PASSED	NONE	t_fmc1: first multiple character in 'bob' is 'b'
PASSED	NONE	t_crc1: test count_repeated_character 'mississippi!!!' has 4 groups: ss, ss, pp and !!!.
Test2	**MY_TESTS**	
FAILED	NONE	t1: t1
FAILED	NONE	t2: t2
FAILED	NONE	t3: t3
FAILED	NONE	t4: t4
FAILED	NONE	t5: t5
FAILED	NONE	t6: t6
FAILED	NONE	t7: t7
FAILED	NONE	t8: t8
FAILED	NONE	t9: t9
FAILED	NONE	t10: t10

As you see from the red bar, not all the tests succeed. You will be writing tests, incrementally, and getting them to work one by one until all tests succeed and you get a green bar.

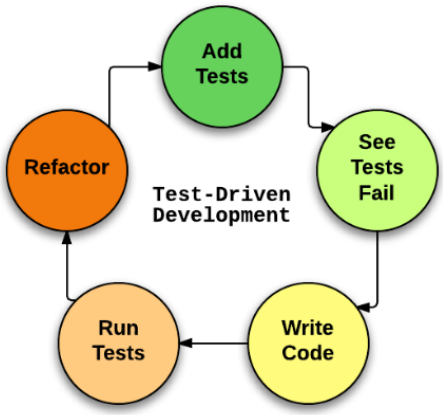
To run the tests, you normally press the “Run Workbench System” button (Ctl+Alt+F5) — rather than the “Run” button(F5).



Question: What is the difference between “Run Workbench System” button (Ctl+Alt+F5) “Run” (F5)?

When you invoke “Run Workbench System” — you do *Regression Testing* —which is a way of testing your code that seeks to uncover new software bugs after you make changes to the software. Experience has shown that as software is corrected, new faults emerge or old faults re-emerge. The purpose of regression testing is to ensure that any new change that you make has not introduced new faults. You thus re-run all previously completed tests (even though they tested green) to ensure that no new bugs have been introduced, especially in other parts of the code that depend on the changes you have introduced.

3.1 Test Driven Development and Regression Testing



1. The developer writes a test that specifies some new feature.
2. The developer compiles (F7) and runs the test (Ctl+Alt+F5). Obviously the test fails (red bar) because the new feature is not yet actually implemented in code.
3. The developer writes code to satisfy the test, compiles (F7) and re-runs all the tests (Ctl+Alt+F5).
4. If the developer writes the code correctly, then all the tests will pass (green bar). If not, go back to step (3) and fix the code.
5. If all the tests succeed, then go to (1) to add new functionality. Alternatively, at this point, a developer may refactor their code to remove code "smells".[3] No new features are added. Instead, the developer cleans up the design, adds comments and other documentation, knowing that if the new re-factored code breaks something, the tests will sound an alert.

4 Introduction to the word analyzer code

Look at class *WORD_ANALYZER_INTERFACE* under cluster *word*.

To do this, setup the IDE as shown below. If you are having trouble, ask a TA or lab instructor to help you obtain this view:

[3] https://en.wikipedia.org/wiki/Code_smell, e.g. code duplication, superman classes that do too much, classes that do too little, features that have too many parameters making it hard to read, class or feature names that are too-long/too-short/not-meaningful, and missing documentation.

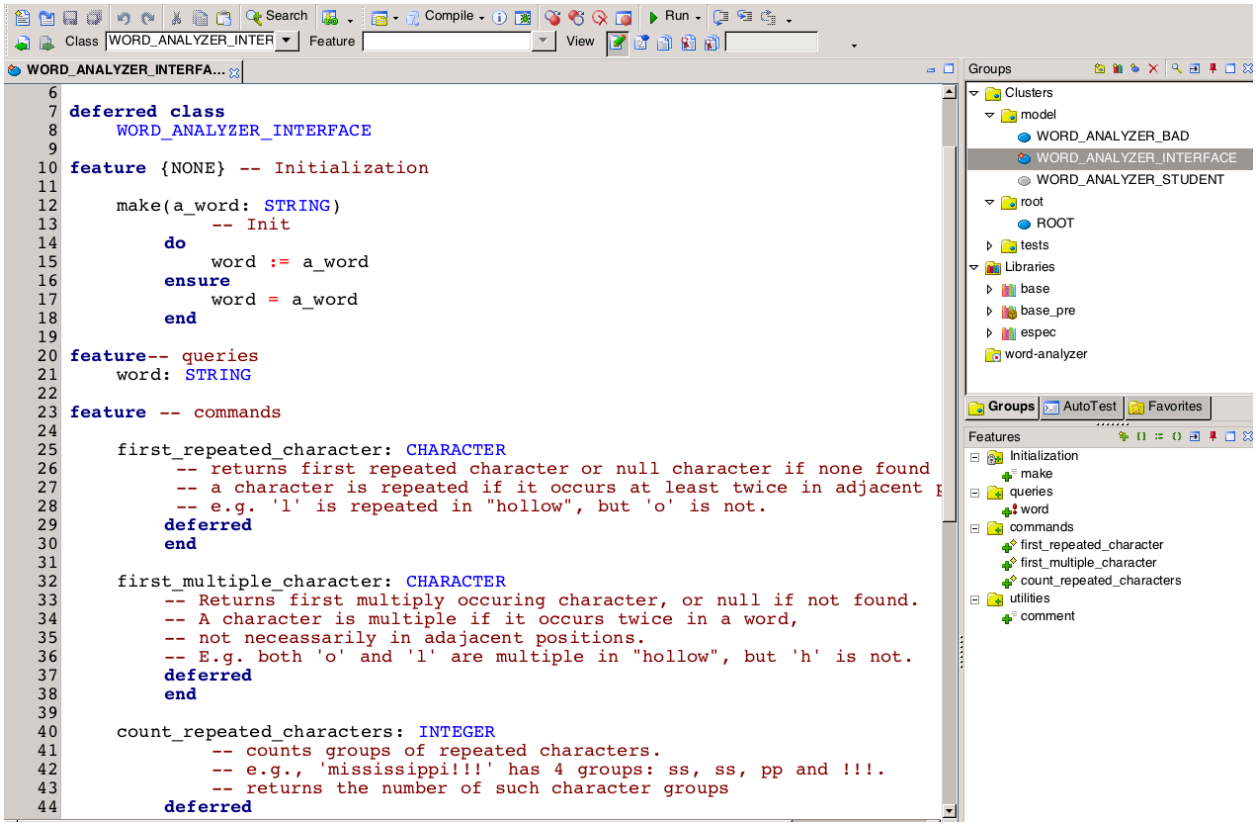


Figure 1 IDE view of class WORD_ANALYZER_INTERFACE

Study the above IDE view of class *WORD_ANALYZER_INTERFACE*, and ensure that you understand its syntax, semantics. On the right of the class – you have the cluster and class browsing views (right/top) and the feature browsing view (right/bottom).

Question: In Eiffel, there is no need for the limited notion of a *Java interface*.[4] Although the class is called *WORD_ANALYZER_INTERFACE*, it is still a complete class. Why does Eiffel not need the limited notion of a Java interface?[5]

Any class may have routines[6] that have implementations (i.e. they are not just feature signatures). A *deferred* class (i.e. an abstract class) has at least one feature that does not have an implementation; however, it may also have features that are effected (i.e. have implementations).

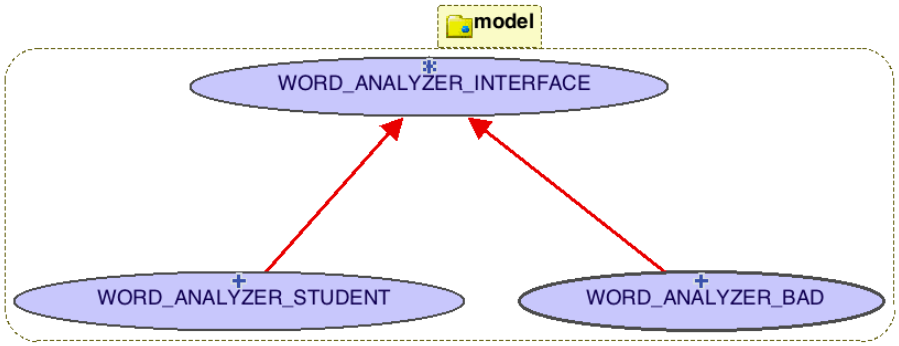
[4] A Java *interface* is a bit like a class, but not quite. A Java interface can only contain method signatures and fields, but not implementation of the methods, only the signature (name, parameters and exceptions) of the method.
[5] Hint: It has to do with the design notion of *multiple inheritance* (supported by the UML standard, but not Java or C#).
[6] In Java we would talk about *methods*.

Thus, class *WORD_ANALYZER_INTERFACE* may also have the routine *comment* which has some implementation (between the **do** and the **end**).[7]

```
47 feature -- utilities
48     comment(s:READABLE_STRING_32): BOOLEAN
49         do
50             Result := true
51         end
52
53 end
```

In the IDE, deferred classes are shown with a little red marker (see Figure 1). On the right hand of Figure 1, we can browse the project by cluster and class, and given a class we can also browse by feature (command or query).[8]

The following BON[9] class diagram describes how classes *WORD_ANALYZER_BAD* and *WORD_ANALYZER_STUDENT* inherit from *WORD_ANALYZER_INTERFACE.*



The EStudio/IDE can produce BON or UML class diagrams from the text of the class. See Appendix 8.5 for more on BON/UML. These class diagrams are important for describing the architecture (i.e. the design) of your software.

The two descendants of the class *WORD_ANALYZER_INTERFACE* implement the deferred routines. However, the bad descendant has some errors. In this Lab:

- You will write some tests that find those errors in the bad descendant.
- You will then fix the errors by implementing the features in class *WORD_ANALYZER_STUDENT*.

There are three *queries* that we are interested in for this lab:

---

[7] **Question**: What might be the use of the query *comment*?
[8] **Question**: What is the difference between a *method* and a *feature*? What is a routine? What is a command and a query? What is an attribute?
[9] You will need to document your design with BON diagrams such as the one shown above. For this, study the documentation for the IDE the diagraming tool. For BON, see https://wiki.eecs.yorku.ca/project/eiffel/bon:start.

1. *first_repeated_character* : CHARACTER
2. *first_multiple_character* : CHARACTER
3. *count_repeated_characters*: INTEGER

Their intended functionalities are specified informally as comments (i.e., strings that follow the comment delimiter "--"). Read the comments. Are they sufficiently complete and precise? How would you implement the three queries?

As an example, consider a bad implementation for query *first_repeated_character* in class *WORD_ANALYZER_BAD* under the same cluster. Note the strict convention:

- Classes are always in capital letters
- Features (commands and queries) are lower case
- Names use the underscore notation to separate words (rather than Camel case)

Important: Always follow the language conventions (in this case Eiffel, see OOSC2, chapter 26, *A sense of Style*).

Figure 2 on the next page, provides the text of query *first_repeated_character* in class *WORD_ANALYZER_BAD*.

Read the comments that describe this query – which provides an informal specification of *what* the query must do, but not *how* to do it.[10]

Meaningful comments for each feature provide important documentation of what the feature does from the point of view of a client who would like to use that feature. You are expected to document your own work with meaningful comments.

[10] **Note**: you will be required to produce documentation of your classes as shown above. The EStudio IDE can produce documentation in *html*, *rtf* and other formats. The text in the figure was generated using *rtf*, which is easy to paste into a Word document. The advantage of textual documentation (rather than a screenshot) is that you can produce just the parts of the code needed to illustrate the point you wish to make. The textual representation is easy to change and displays more clearly than a screenshot. In the IDE, invoke *Project* → *Generate Documentation* (read the manual).

```
word: STRING

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.
      local
            i: INTEGER
            ch: CHARACTER
            stop: BOOLEAN
      do
            from
                  i := 1
                  Result := '%U' -- not needed, why?
            until
                  i > word.count or stop
            loop
                  ch := word [i]
                  if ch = word [i + 1] then
                        Result := ch
                        stop := True
                  end
                  i := i + 1
            end
      end
```

Figure 2: Text of query first_repeated_character

Before proceeding, inspect the body of the code. Does the implementation of the query satisfy its informal specification (documented in the query comments)?

At first glance, the query looks respectable. In fact, the following test defined in class *INSTRUCTOR_TESTS* (under cluster *instructor*) succeeds (green bar):

```
t_frc1: BOOLEAN
        -- succeeds in bad analyzer
    local
        wa: WORD_ANALYZER_BAD
    do
        comment("t_frc1: first repeated character for 'hollow' is 'l'")
        wa.make("hollow")
        Result := wa.first_repeated_character = 'l'
        assert ("'o' is not repeated",
            wa.first_repeated_character /= 'o',
            wa.first_repeated_character)
    end
```

Test1		INSTRUCTOR_TESTS
PASSED	NONE	t_frc1: first repeated character for 'hollow' is 'l'

Note the following about the above Espec test:

- The test *t_frc1* has a return type BOOLEAN (i.e., it is a boolean test case).
- To succeed, it must (a) return *True* and (b) pass all *checks* and *asserts*.
- A failing *assert* is shown below:

```
t_frc1: BOOLEAN
        -- succeeds in bad analyzer
    local
        wa: WORD_ANALYZER_INTERFACE
    do
        comment("t_frc1: first repeated character for 'hollow' is 'l'")
        create {WORD_ANALYZER_BAD} wa.make ("hollow")
        Result := wa.first_repeated_character = 'l'
        assert ("'l' is not repeated",
            wa.first_repeated_character /= 'l',
            wa.first_repeated_character)
    end
```

Test1		INSTRUCTOR_TESTS
FAILED	Check assertion violated.	t_frc1: first repeated character for 'hollow' is 'l' Assert Violation: 'l' is not repeated Object: l

We could also have written (using **check** which is an Eiffel key word):

```
t_frc1: BOOLEAN
        -- succeeds in bad analyzer
    local
        wa: WORD_ANALYZER_INTERFACE
    do
        comment("t_frc1: first repeated character for 'hollow' is 'l'")
        create {WORD_ANALYZER_BAD} wa.make ("hollow")
        Result := wa.first_repeated_character = 'l'
        check
            wa.first_repeated_character /= 'o
        end
    end
```

Figure 3: using check

However, *assert* returns more diagnostic information than a check. **check** is an Eiffel key word and may thus be used to test a condition in any routine, whereas *assert* may only be used in a test as it is an ESpec library routine. We could also have written:

```
t_frc1: BOOLEAN
        -- succeeds in bad analyzer
    local
        wa: WORD_ANALYZER_INTERFACE
    do
        comment("t_frc1: first repeated character for 'hollow' is 'l'")
        create {WORD_ANALYZER_BAD} wa.make ("hollow")
        Result := wa.first_repeated_character = 'l'
        check Result end
        Result := wa.first_repeated_character /= 'o
    end
```

Figure 4: using check on Result

Question: In Figure 4, why must there be a **check** assertion directly after the first assignment to *Result*?

Feature *t_frc1* and the other two (i.e., *t_fmc* and *t_crc*) make use of the ESpec unit testing framework:

- Class *INSTRUCTOR_TESTS* inherits from **ES_TEST**, which gives access to testing facility features such as *add_boolean_case* and *comment*.

- However, this does not necessarily mean that all features whose names start with "t_" automatically become test cases.
  Instead, in the *make* routine of *INSTRUCTOR_TESTS*, you have to explicitly add, via the **add_boolean_case** feature, a list of Boolean queries.
  When executed, if a query from the list returns *TRUE* (observed from the value of its *Result*), is considered as a passing test case; otherwise, it is a failing test case.
- **For each test query *q*, you must have a *comment ("q: …")* clause to document its intended function. Note that the colon (":") inside the argument of *comment* is mandatory in all work you do in this course. To the left of the colon must be the name of the query, and to its right an informal description of the test case. Otherwise, the test case cannot be interpreted properly by the ESpec testing framework and our grading scripts.**
  For example, in query *t_frc1*, we have *comment ("t_frc1: first repeated …")*.
- Observe that class *STUDENT_TESTS* has a similar testing configuration as does *INSTRUCTOR_TESTS*.
- To combine test cases into a test suite, the root class *APPLICATION*
  - inherits from **ES_SUITE**
  - in its *make* routine explicitly includes, via the **add_test** feature, the collections of test cases (e.g., *STUDENT_TESTS*, *INSTRUCTOR_TESTS*, etc.)

Although the above test *t_frc1* succeeds, this does not mean that the query is correct for all possible words. In fact, the query fails (badly) for some words.

In this Lab exercise you are required find bugs in this query (and the other two) by providing tests that fail on the bad code (in class *WORD_ANALYZER_BAD*) and succeed on the correct code (in class *WORD_ANALYZER_STUDENT*).

The purpose of testing 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!" Albert Einstein is reputed to have said: "No amount of experimentation can ever prove me right; a single experiment can prove me wrong."

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. This is an important start to putting your software development skills on a more scientific basis. Your tests are like the critical experiments that a scientist does in the laboratory to check the validity of a scientific theory.

Before proceeding, list below what you know about testing, and how you use tests in software that you develop:

Make sure to read the Appendix to learn more about Testing.

The tests in *INSTRUCTOR_TESTS* all pass. So these queries work some of the time. However, each of *first_repeated_character*, *first_multiple_character*, and *count_repeated_characters* has bug(s), and you must fix them all in class *WORD_ANALYZER_STUDENT*.

4.1 Contracts for the find routine

```
find(c: CHARACTER; pos: INTEGER): INTEGER
        -- ToDo: put some meaningful comment here
    require
        -- ToDo: put some assertion here
    local
        i: INTEGER
        stop: BOOLEAN
    do
        from
            i := pos
            Result := 0 -- default
        until
            i > word.count or stop
        loop
            if c = word[i] then
                Result := i
                stop := true
            end
            i := i + 1
        end
    ensure
        -- ToDo: put some assertion here
    end
```

You are required to develop contracts (a precondition and postcondition) for the the *find* query in class *WORD_ANALYZER_STUDENT* (whether you use it or not). The precondition must assert that the argument *pos* is a valid index into the attribute *word*. The postcondition must assert that the query returns the index of a character *c* in *word* in range *pos .. word.count*, or zero if there is no such character. You must also document the query with a meaningful comment.

Hint: See the appendix Section 8.2 for how to use the **across** notation (which can also be used to iterate over a string). This might be useful for the postcondition.

In addition to an electronic submission, you must also print out a report, and place it in the course dropbox by the submit date. The report is one page as shown below, except that you must document the query *find*. Use the documentation feature of the IDE to generate the documentation of *find* in RTF format.

EECS331 – Lab 1 – Word Analyzer

Student Name:

Prism Login:

I completed and submitted the whole Lab: YES

Comments:

```
balance: INTEGER_32

deposit (v: INTEGER_32)
          -- deposit `v' dollars in the account
     require
          amount_is_positive: v > 0
     do
          balance := balance + v
     ensure
          correct_balance: balance = old balance + v
     end
```

5 What you must do

- You must print the one page report as described in Section 4.1 and submit it at the course dropbox.
- You must also do an electronic submission as described below.

1. In class *STUDENT_TESTS* (see the Figure 5 on the next page):
    a. Complete test *t_frc1* that reveals the bug in *query first_repeated_character*. (This test should fail by using the version of *query* in *WORD_ANALYZER_BAD*.)
    b. Similarly, complete test *t_fmc1* that reveals the bug in *first_multiple_character*.
    c. Similarly, complete test *t_crc1* that reveals the bug in *count_repeated_characters*.
2. In class *WORD_ANALYZER_STUDENT*, provide correct versions of the three *queries*.
3. In class STUDENT_TESTS:
    a. Complete test *t_frc2* that passes for the corrected *query first_repeated_character* (This test succeeds by using the **same** test word string as of *t_frc1*, but using your version of *query* in *WORD_ANALYZER_STUDENT*)
    b. Similarly, complete *t_fmc2* that passes for the fixed *first_multiple_character*.
    c. Similarly, complete *t_crc2* that passes for the fixed *count_repeated_characters*.
4. In class *MY_TESTS* in cluster *student*, add additional tests in this class to gain more assurance that your fixes to queries in *WORD_ANALYZER_STUDENT* are correct. You must have **at least 10 test cases** (≥ 10) in MY_TESTS and all of them must pass (otherwise, you will receive 0 without being further evaluated).

    Remember: you must equip each test query *q* with a *comment ("q: …")* clause in order for the ESpec testing framework to process that properly. (Note that the colon ":" is mandatory.)

5. You must equip query *find* in class *WORD_ANALYZER_STUDENT* with a meaningful comment and a precise precondition and postcondition.

Therefore, you are only allowed to make changes to the following classes:

- STUDENT_TESTS
- WORD_ANALYZER_STUDENT
- MY_TESTS
- *ROOT* (you will add *MY_TEST* to its make *routine*)

```
feature -- tests

    good: BOOLEAN = true
    bad: BOOLEAN = false

    make_wa(a_word:STRING; a_good: BOOLEAN): WORD_ANALYZER_INTERFACE
        do
            if a_good then
                create {WORD_ANALYZER_STUDENT} Result.make (a_word)
            else
                create {WORD_ANALYZER_BAD} Result.make (a_word)
            end
        end

feature -- first repeated character tests
    frc_test_string: STRING = "your-test-string"

    t_frc1: BOOLEAN
        local
            wa: WORD_ANALYZER_INTERFACE
        do
            comment("t_frc1: fail on WORD_ANALYZER_BAD")
            wa := make_wa (frc_test_string, bad)
            -- TODO
            -- complete this test
        end

    t_frc2: BOOLEAN
        local
            wa: WORD_ANALYZER_INTERFACE
        do
            comment("t_frc2: succeed on WORD_ANALYZER_STUDENT")
            wa := make_wa (frc_test_string, good)
            -- TODO
            -- complete this test
        end
```

Figure 5: class STUDENT_TESTS

In class ROOT, run the tests in the following order:

```
    make
            -- Run application.
        do
            )
            add_test (create {STUDENT_TESTS}.make)
            add_test (create {MY_TESTS}.make)
            add_test (create {INSTRUCTOR_TESTS}.make
--          show_errors
            show_browser
            run_espec
        end
```

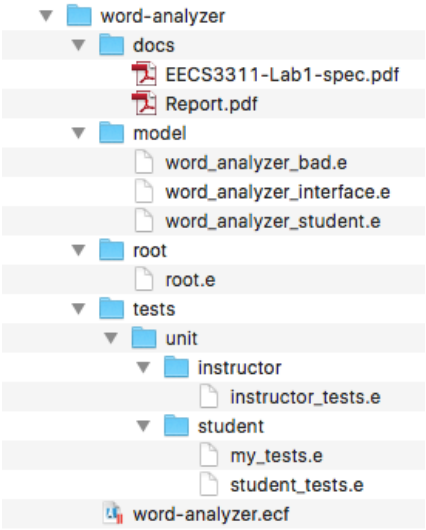
Use "*show_errors*" if you want to see a report of the complete exception stack. Comment it out for submission. When you submit the project, comment out the call to *show_browser*.

You should see the relevant tests fail/succeed as specified in class *STUDENT_TESTS*. On the other hand, all tests in *MY_TESTS* and *INSTRUCTOR_TESTS* should succeed (i.e., both files MY_TESTS.html and INSTRUCTOR_TESTS.html show a green bar).

6 Submission

Once you have completed the Lab, carefully do each of the following steps:

1. Freeze your project and run it to ensure that the appropriate results are generated in the web browser.
2. Before submission, make sure that:
    a. In classes *STUDENT_TESTS* and *WORD_ANALYZER_STUDENT*, you did not change the original signatures (i.e., names, parameters, and types) of any features.
    b. For the body of each test query that you defined in class *MY_TESTS*, you included a *comment* clause.
    c. You provided the comment and contracts for the query *find*.
    d. Your directory structure is as shown below. *Report.pdf* is you one page report.



3. At the console do: "**eclean** *word-analyzer*", so as to remove all EIFGENs. Then submit your project as follows:
4. **submit 3311 lab1 word-analyzer**
5. You may submit multiple times before the deadline. When directory **word-analyzer** is submitted, your project will be compiled and you will be provided with some feedback on errors, if any. If it is before the submission deadline, and there are errors in your submission, you may want to fix them and re-submit.

7 Grading

You will not receive a passing grade for the Lab if:

- Your submitted folder does not have the expected directory structure.
- Your submitted project cannot be compiled and executed.
- You do not have **at least ten** (≥ 10) test cases in MY_TESTS that **all** pass.

8 Appendix

8.1 Testing

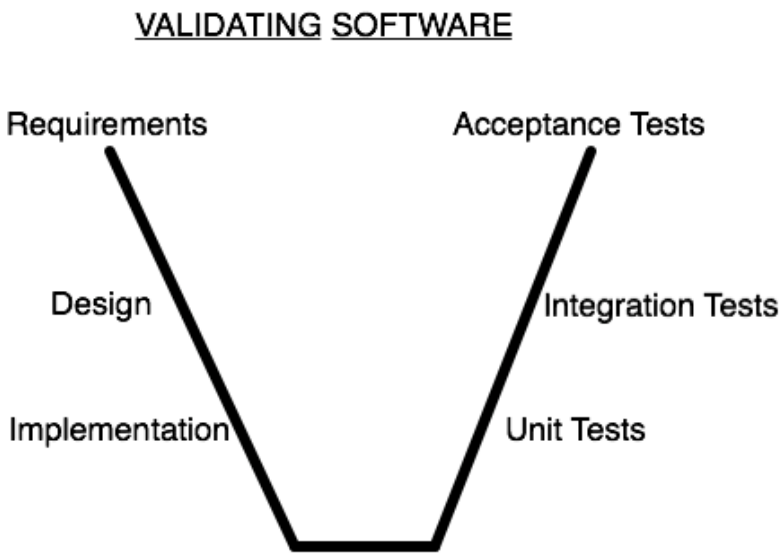
Much of this material comes from John Guttag's introductory text on Python. The ideas are applicable to testing code written in any language, not just Python.[11] Some information has been added, and the discussion is adapted to Eiffel.

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 tests classes (modules) and their routines, but we also need to test sub-systems (clusters of classes) and the overall system (acceptance tests).

[11] John V Guttag. Introduction to Computation and Programming Using Python, revised and expanded edition, MIT Press 2013.



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

8.1.1 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.

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

8.1.2 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* ≡ x < y

Before proceeding, provide below an informal English description of the specification[12]:

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-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.

[12] Answer: Assume x and y are integers. The query returns *True* if x is less than y and *False* otherwise.

8.2 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
```

So the string s is a function 1..3 → 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 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 ∀ and ∃. Thus *test1* asserts:

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

which is true, and *test2* asserts that

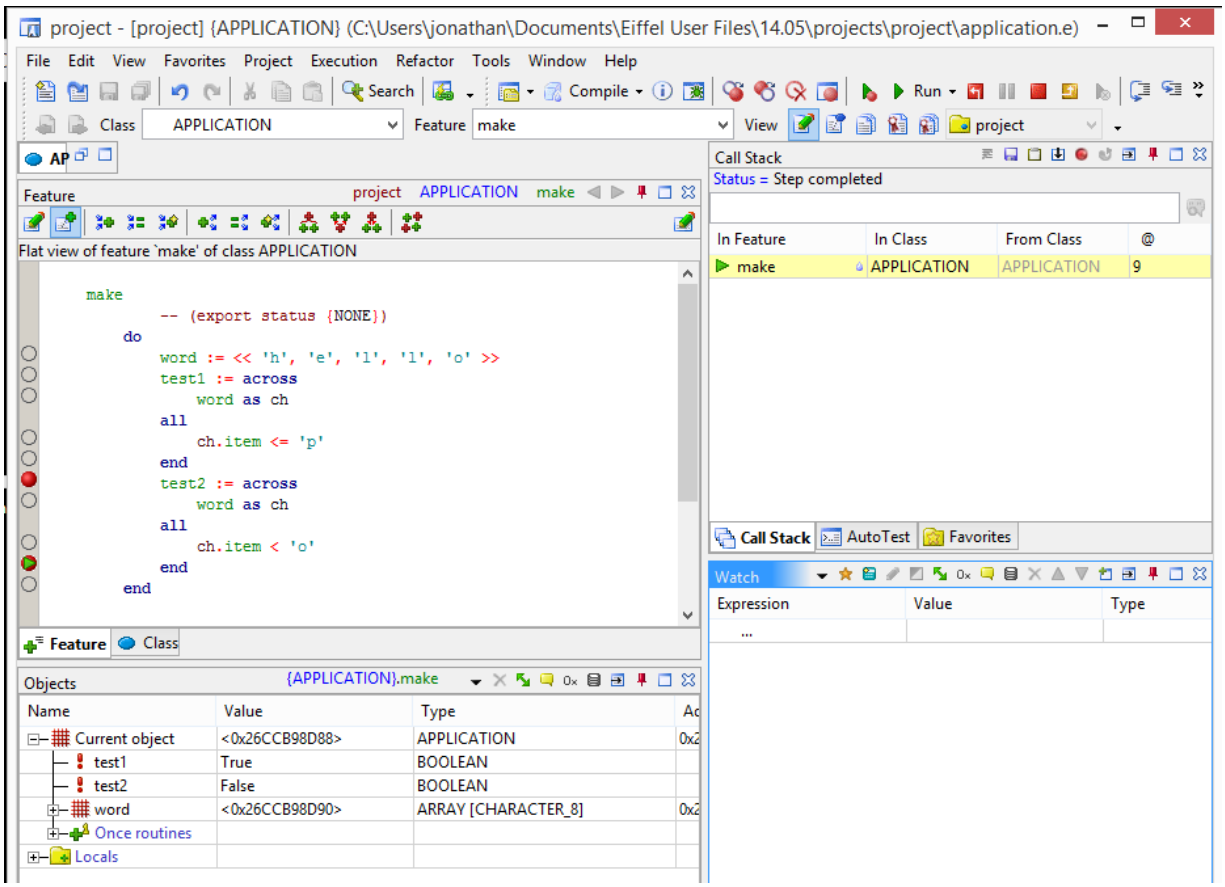
$$\forall \text{ ch } \in \text{ word: ch} < \text{'o'}$$

which is false. The comparison (≤) 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 ∀ and **some** for ∃. 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.

8.3 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 <= 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, than the hypothesis that awhile 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.

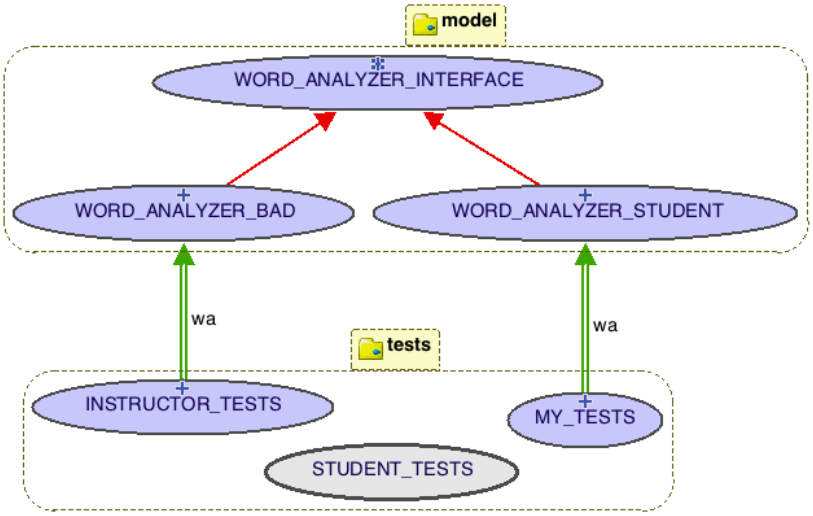
Finally, keep a record of what experiments you have run. This is particularly important. If you aren't careful, it is easy to waste countless hours trying the same experiment (or more likely an experiment that looks different but will give you the same information) over and over again. Remember, as many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

8.4 Using unit tests and the debugger

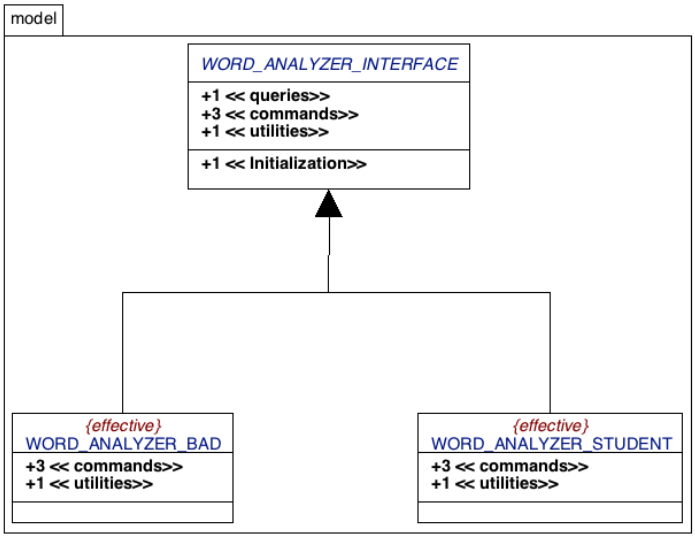
See https://docs.eiffel.com/book/eiffelstudio/debugger.

8.5 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 toll of the IDE, then you get



See the footnote for more information.[13] Please familiarize yourself with these notations.

[13] You might try to produce the BON diagram and also the UML diagram. Why do we use BON diagrams rather the the more standard UML notation?
(Hint: see the video: 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.

8.6 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[14]?

[14] 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?