EECS-3311 – Lab – Bank Specification

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]

[1] https://wiki.eecs.yorku.ca/course_archive/2015-16/W/3311/honesty:start

Precondition

Before doing this Lab, ensure you have done Lab1 and read and understood its Appendices. You shall also have done all the preparatory work mentioned in the Appendix of Lab1 such as fixing bugs and getting tests in your project to work, and other preparatory readings mentioned in the Lab. You have attended the lectures and mastered the slides and done the suggested background reading.

In this Lab, you will design a small banking system that keeps a database of customers and allows customers to deposit, withdraw and transfer money. The arithmetic must be precise so you cannot use floats or REAL. Even INTEGER alone will not work as money is in dollars and cents, and INTEGER would be limited to 32 or 64 bits. It is essential that the bank system be correct and well-designed. This Lab emphasizes the *correctness* of the system using Design by Contract (DbC).

1 Can a good design be incorrect?

The answer is: “No”.

If a software system is properly decomposed into well-documented and efficient modules and satisfies other design goals, but does not satisfy the customer specification, it is not yet a good design. If software controlling a transportation system causes trains to crash, then the design is not good. If a medical system allows physicians to prescribe medications with dangerous interactions, without warning, then the design is not good.

Software correctness is a *sina qua non*.[2]

We describe and check software correctness using specifications, design by contracts, and testing.

2 Learning Outcomes

- *Specification*: Write contracts to help specify the system correctly free of implementation detail.
- Implementation: Code the system to satisfy the specification.
- Write tests and use test driven development to check correctness.
- Use the IDE to do the above and provide BON class diagrams that describe the *architecture* of the design.

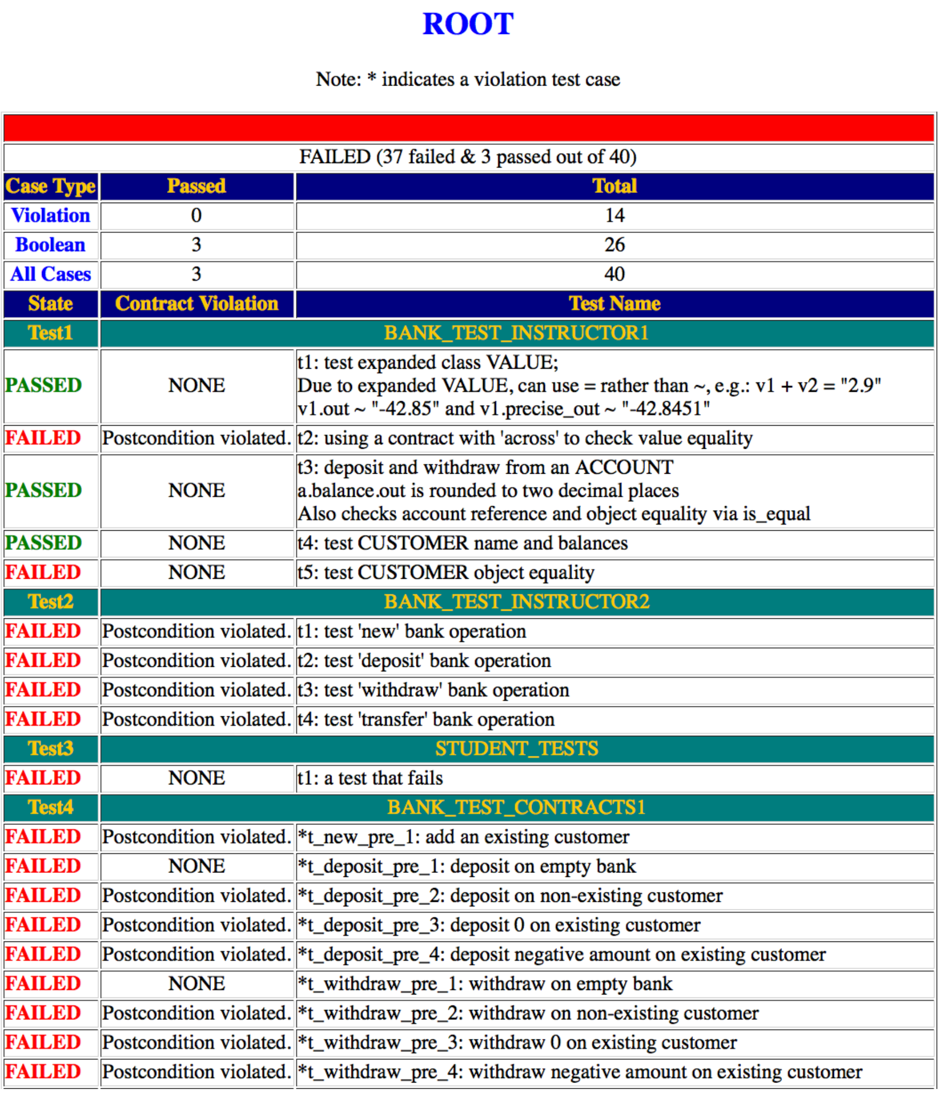
[2] Latin for something absolutely indispensable or essential, e.g. reliability of the software is a *sine qua non* for success.

3 Getting started

1. Login to your account on a Prism Linux workstation.
2. Then do: `~sel/3311/retrieve/lab2`
   a. This will download folder *bank*.
   b. `cd bank`
   c. Then do[3]: `estudioXX.YY bank.ecf &`

Now compile and run the ESpec tests (Ctl+Alt+F5).

When you first run ESpec you will see the red bar! Something like the illustration below.



ROOT

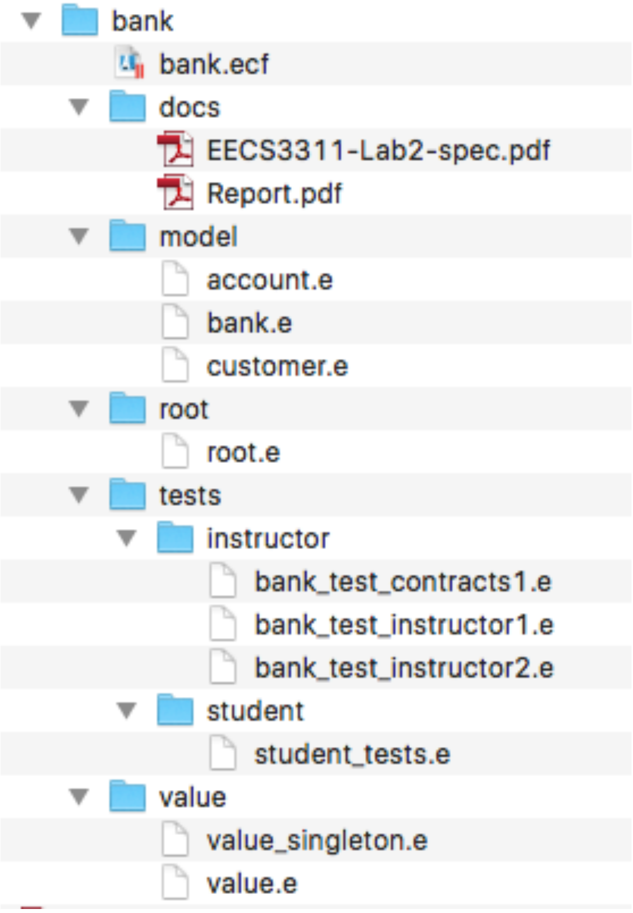
Note: * indicates a violation test case

🟥	🟥	🟥
FAILED (37 failed & 3 passed out of 40)		
Case Type	**Passed**	**Total**
Violation	0	14
Boolean	3	26
All Cases	3	40
State	**Contract Violation**	**Test Name**
Test1	**BANK_TEST_INSTRUCTOR1**	
PASSED	NONE	t1: test expanded class VALUE; Due to expanded VALUE, can use = rather than ~, e.g.: v1 + v2 = "2.9" v1.out ~ "-42.85" and v1.precise_out ~ "-42.8451"
FAILED	Postcondition violated.	t2: using a contract with 'across' to check value equality
PASSED	NONE	t3: deposit and withdraw from an ACCOUNT a.balance.out is rounded to two decimal places Also checks account reference and object equality via is_equal
PASSED	NONE	t4: test CUSTOMER name and balances
FAILED	NONE	t5: test CUSTOMER object equality
Test2	**BANK_TEST_INSTRUCTOR2**	
FAILED	Postcondition violated.	t1: test 'new' bank operation
FAILED	Postcondition violated.	t2: test 'deposit' bank operation
FAILED	Postcondition violated.	t3: test 'withdraw' bank operation
FAILED	Postcondition violated.	t4: test 'transfer' bank operation
Test3	**STUDENT_TESTS**	
FAILED	NONE	t1: a test that fails
Test4	**BANK_TEST_CONTRACTS1**	
FAILED	Postcondition violated.	*t_new_pre_1: add an existing customer
FAILED	NONE	*t_deposit_pre_1: deposit on empty bank
FAILED	Postcondition violated.	*t_deposit_pre_2: deposit on non-existing customer
FAILED	Postcondition violated.	*t_deposit_pre_3: deposit 0 on existing customer
FAILED	Postcondition violated.	*t_deposit_pre_4: deposit negative amount on existing customer
FAILED	NONE	*t_withdraw_pre_1: withdraw on empty bank
FAILED	Postcondition violated.	*t_withdraw_pre_2: withdraw on non-existing customer
FAILED	Postcondition violated.	*t_withdraw_pre_3: withdraw 0 on existing customer
FAILED	Postcondition violated.	*t_withdraw_pre_4: withdraw negative amount on existing customer

[3] Note: On the Prism (Linux) systems and the SEL Virtual Machine there is no *estudio*. Use the version of *estudio* required by your course director, e.g. *estudio15.12*. For the compiler “ec” it would be *ec15.12*.

Don’t panic! You are going to work step-by step and get all the tests to pass (eventually). You will also write some of your own tests which should must also pass.

The directory structure of the project looks as shown below. You will create *Report.pdf*. Other than that, do not add or delete any files or classes. You will, however, be editing some (but not all) of these files using the IDE.



```
▼ bank
      bank.ecf
  ▼ docs
        EECS3311-Lab2-spec.pdf
        Report.pdf
  ▼ model
        account.e
        bank.e
        customer.e
  ▼ root
        root.e
  ▼ tests
      ▼ instructor
            bank_test_contracts1.e
            bank_test_instructor1.e
            bank_test_instructor2.e
      ▼ student
            student_tests.e
  ▼ value
        value_singleton.e
        value.e
```

3.1 Requirements

You are to design a bank system with the following requirements.

(1) A Bank consists of many customers. There are *count* customers.
(2) New customers can be added to the bank by name. We never delete a customer record, even after they leave the bank.
(3) Each customer shall have a single account at the bank. Initially the balance in the account is zero.

(4) Money can be deposited to and withdrawn from customer accounts. Money is deposited as a dollar amount, perhaps with more than two decimal places.
(5) Money calculations shall be precise (e.g. adding, subtracting and multiplying money amounts must be done without losing pennies or parts of pennies).
(6) Money can also be transferred between two customer accounts.
(7) Balances in accounts shall never be negative.
(8) Customers are identified by name, so there cannot be two customers having the same name.
(9) Customers are stored in a list sorted alphabetically by name.
(11) The bank has an *attribute* `total` that stores the total of all the balances in the various customer accounts. This can be used to check for fraud.

3.2 The Design

A *design* is the *architecture* of the system (usually shown via a UML or BON class diagram) and a *specification* (usually shown via contracts, tests and informal specifications). The architecture describes how the system is decomposed into components and modules and how those modules relate to each other (e.g. via client-supplier or inheritance relationships).

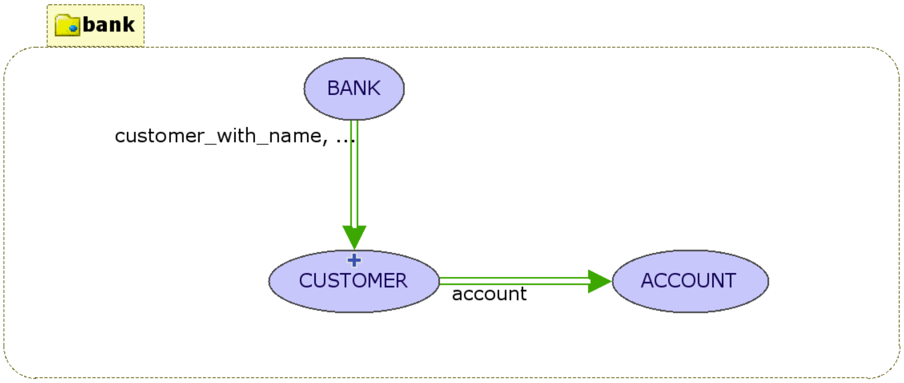
The EStudio IDE can be used to obtain the UML or BON diagrams. These are helpful, but usually incomplete or not sufficiently abstract. In the space below, provide a BON class diagram for the bank system. Give, yourself some time to think about it. What classes do we need? What is the relationship between the classes (client-supplier, inheritance, etc.)? What features should these classes have (i.e. what is their interface to the outside world)?

Use the space below to develop your design using a class diagram:

Having analyzed the problem statement, the following classes seem reasonable:

- ACCOUNT
- CUSTOMER
- BANK

The EStudio IDE diagramming tool can be used to obtain the following BON diagram:



We will need to make may design decisions as we develop the code to satisfy the requirements. For example, one design decision is how to to do arithmetic with dollars and cents without losing pennies (or more) along the way. Consider the Python code below:

```
>python
Python 2.7.6 (default, Sep 9 2014, 15:04:36)
>>> 0.1 + 0.1 + 0.1 + 0.1+ 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
0.9999999999999999
```

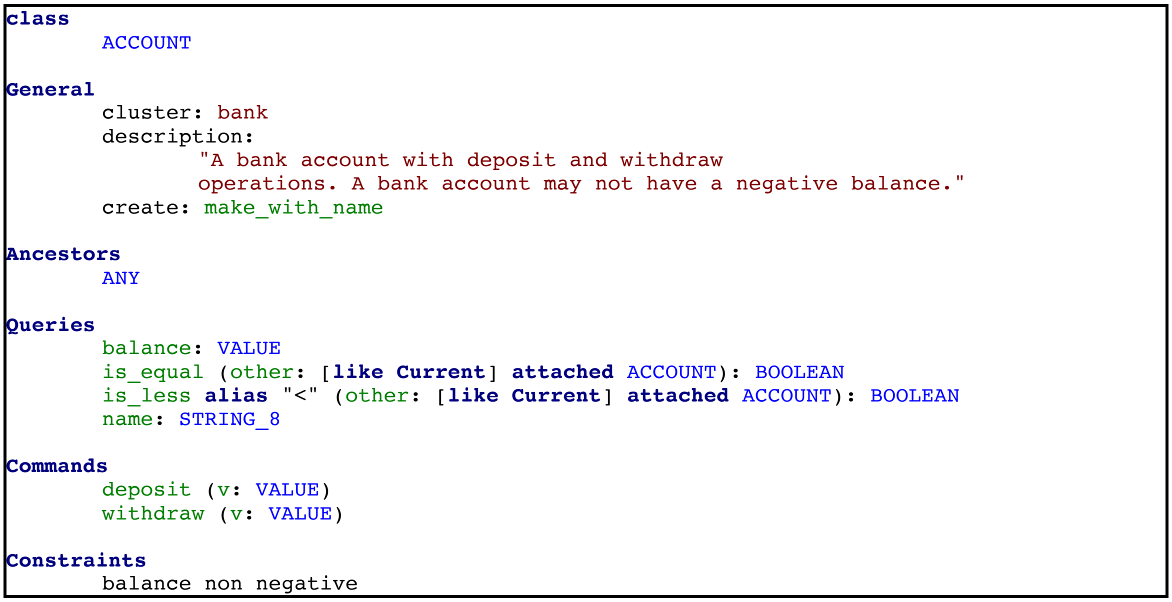
Adding the floating point number “0.1” ten times should give us “1.0”. But it does not! This is because the number “0.1” is not precisely represented. Using floating point numbers in C, Java, C#, Eiffel (where the relevant class is REAL) etc. would have the same problem.

So now we need to design a MONEY class that will allow us to do arbitrary precision arithmetic with currency. In this Lab, we will provide you with the expanded class VALUE that may be used to do arbitrary precision arithmetic. So in this case the design decision has already been made for you.

3.2.1 Clean interfaces (Chart View)

The BANK has a list of CUSTOMER. A CUSTOMER has an ACCOUNT. These are client-supplier relationships as shown in the BON diagram.

EStudio can automatically generate documentation from the code for a class. Here is the **chart view** for class ACCOUNT.

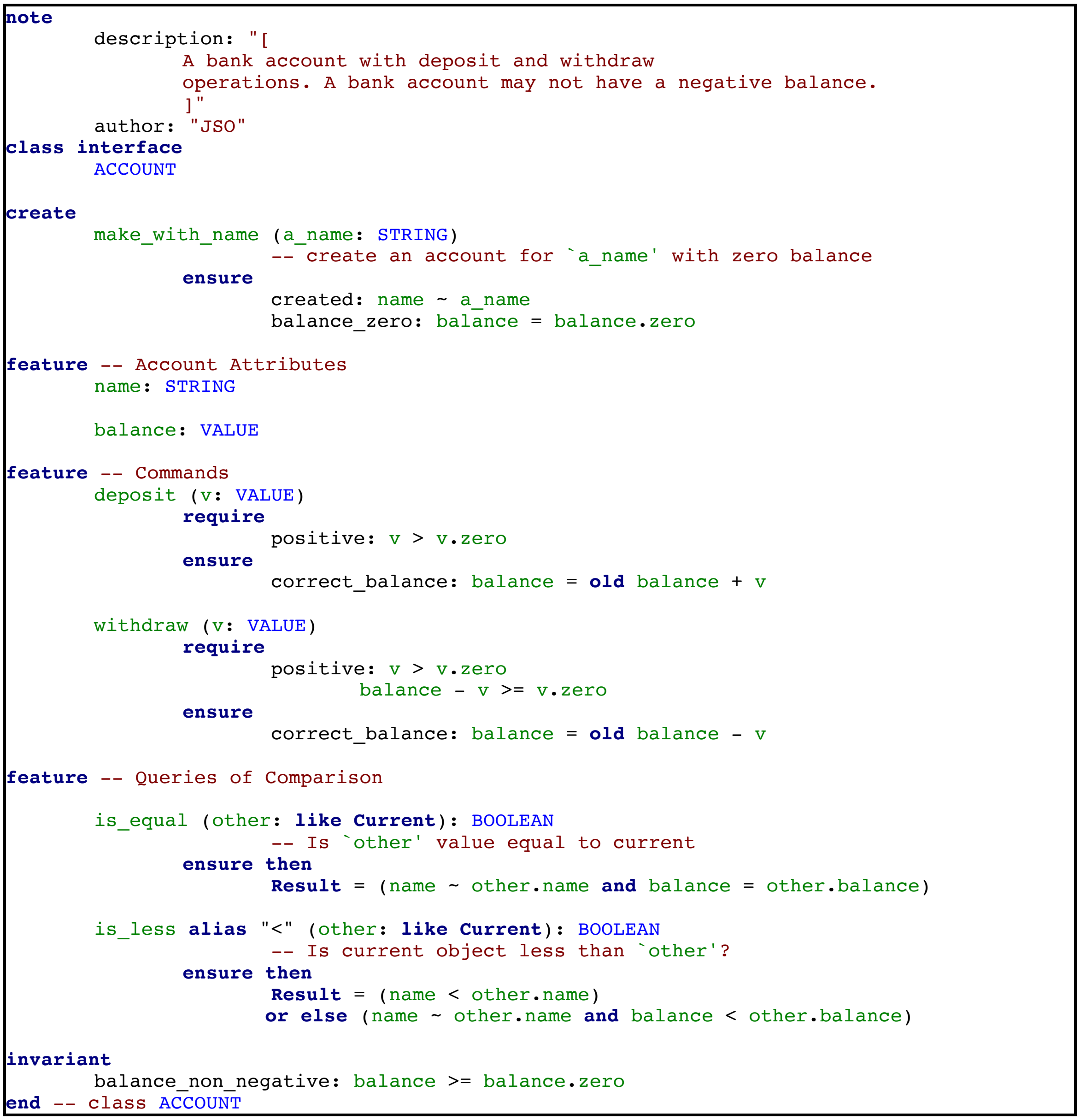


```
class
        ACCOUNT

General
        cluster: bank
        description:
                "A bank account with deposit and withdraw
                operations. A bank account may not have a negative balance."
        create: make_with_name

Ancestors
        ANY

Queries
        balance: VALUE
        is_equal (other: [like Current] attached ACCOUNT): BOOLEAN
        is_less alias "<" (other: [like Current] attached ACCOUNT): BOOLEAN
        name: STRING_8

Commands
        deposit (v: VALUE)
        withdraw (v: VALUE)

Constraints
        balance non negative
```

3.2.2 Correctness (Contract View)

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

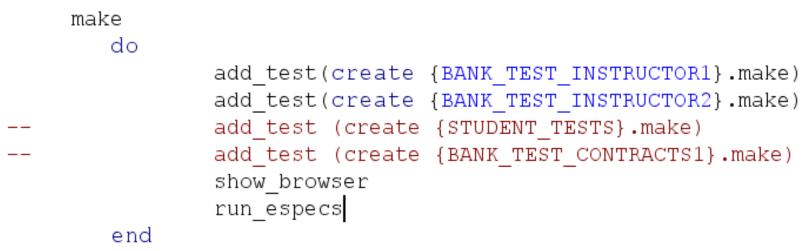
- We use contracts to specify the bank system, in a way that is free of implementation detail.
- We will want many tests to exercise the contracts to ensure that the implementations of the routines satisfy the specifications.
- Invariants are very important in constraining objects to remain safe. On 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.



```
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
                ensure 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
```

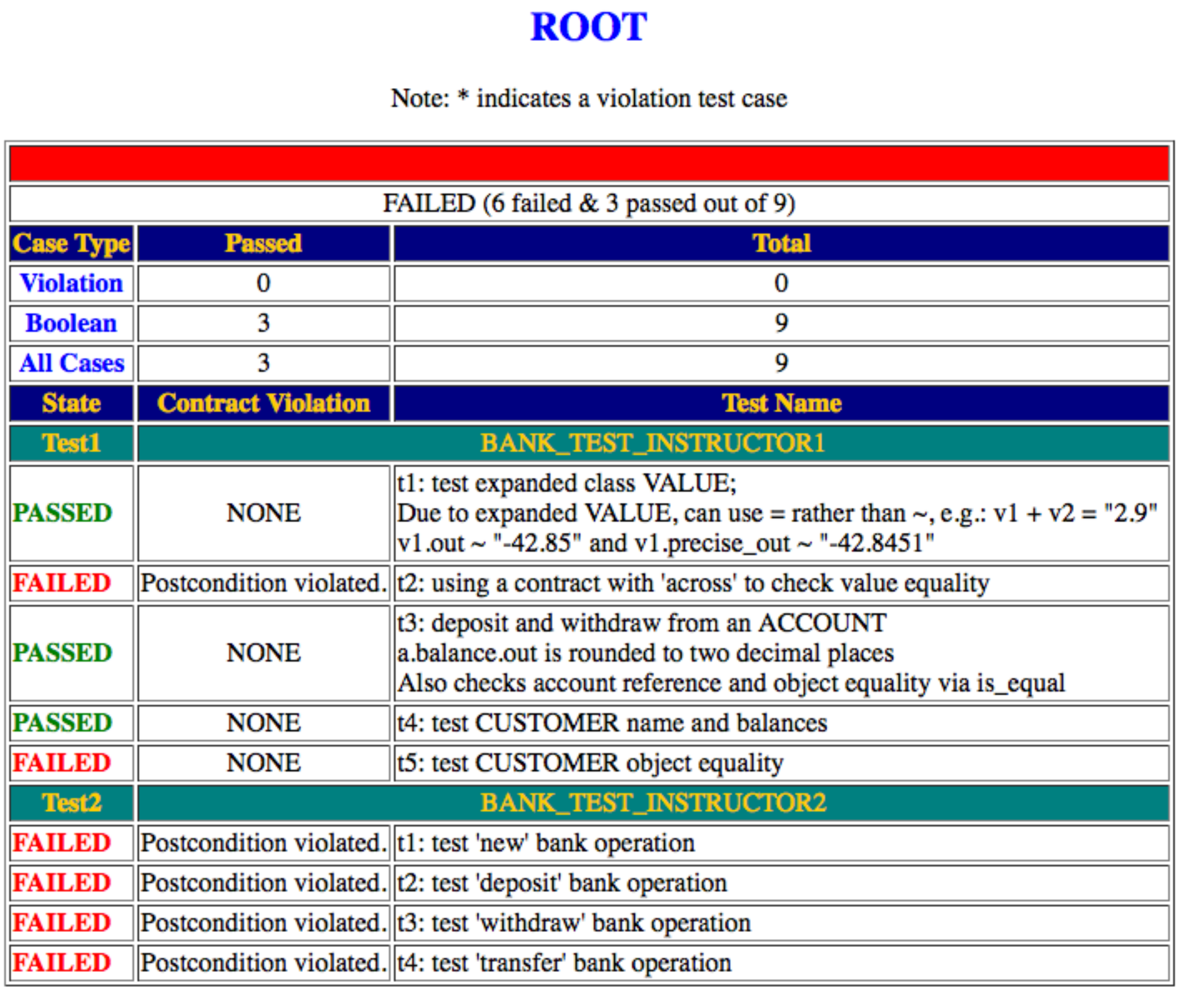
4 Getting Started on the Lab

In the ROOT class, comment out some of the tests like this:



```
      make
         do
                 add_test(create {BANK_TEST_INSTRUCTOR1}.make)
                 add_test(create {BANK_TEST_INSTRUCTOR2}.make)
--               add_test (create {STUDENT_TESTS}.make)
--               add_test (create {BANK_TEST_CONTRACTS1}.make)
                 show_browser
                 run_especs
         end
```

Eventually, you must get all the tests to run. But, in the beginning, focus on getting the instructor tests to run. As you can see, some of the tests already work.



ROOT

Note: * indicates a violation test case

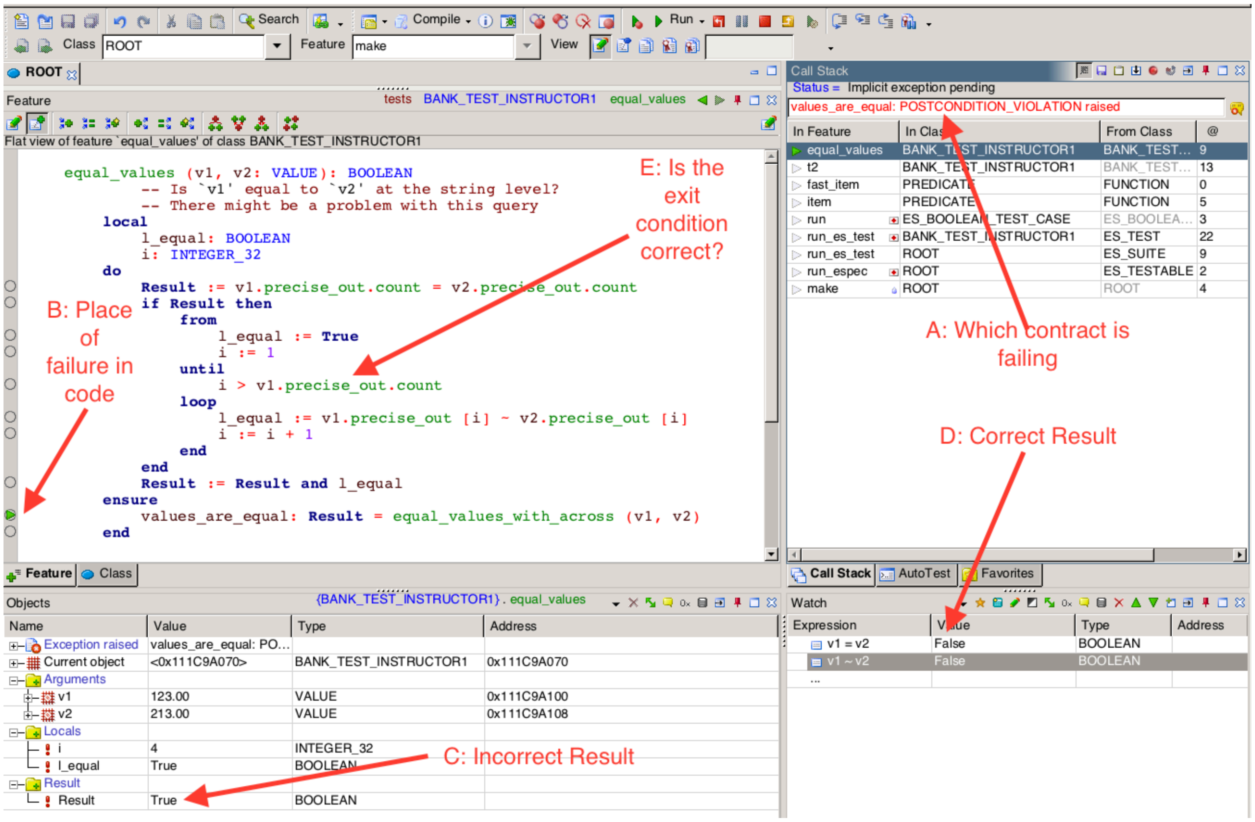
FAILED (6 failed & 3 passed out of 9)

Case Type	Passed	Total
Violation	0	0
Boolean	3	9
All Cases	3	9

State	Contract Violation	Test Name
Test1	BANK_TEST_INSTRUCTOR1	
PASSED	NONE	t1: test expanded class VALUE; Due to expanded VALUE, can use = rather than ~, e.g.: v1 + v2 = "2.9" v1.out ~ "-42.85" and v1.precise_out ~ "-42.8451"
FAILED	Postcondition violated.	t2: using a contract with 'across' to check value equality
PASSED	NONE	t3: deposit and withdraw from an ACCOUNT a.balance.out is rounded to two decimal places Also checks account reference and object equality via is_equal
PASSED	NONE	t4: test CUSTOMER name and balances
FAILED	NONE	t5: test CUSTOMER object equality
Test2	BANK_TEST_INSTRUCTOR2	
FAILED	Postcondition violated.	t1: test 'new' bank operation
FAILED	Postcondition violated.	t2: test 'deposit' bank operation
FAILED	Postcondition violated.	t3: test 'withdraw' bank operation
FAILED	Postcondition violated.	t4: test 'transfer' bank operation

Start by getting {BANK_TEST_INSTRUCTOR1}.*t2* and {BANK_TEST_INSTRUCTOR1}.*t5* to run.

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 on the next page in Figure 1. 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).



```
equal_values (v1, v2: VALUE): BOOLEAN
        -- Is `v1' equal to `v2' at the string level?
        -- There might be a problem with this query
    local
        l_equal: BOOLEAN
        i: INTEGER_32
    do
        Result := v1.precise_out.count = v2.precise_out.count
        if Result then
            from
                l_equal := True
                i := 1
            until
                i > v1.precise_out.count
            loop
                l_equal := v1.precise_out [i] ~ v2.precise_out [i]
                i := i + 1
            end
        end
        Result := Result and l_equal
    ensure
        values_are_equal: Result = equal_values_with_across (v1, v2)
    end
```

E: Is the exit condition correct?

B: Place of failure in code

Call Stack

Status = Implicit exception pending

values_are_equal: POSTCONDITION_VIOLATION raised

In Feature	In Class	From Class	@
equal_values	BANK_TEST_INSTRUCTOR1	BANK_TEST...	9
t2	BANK_TEST_INSTRUCTOR1	BANK_TEST...	13
fast_item	PREDICATE	FUNCTION	0
item	PREDICATE	FUNCTION	5
run	ES_BOOLEAN_TEST_CASE	ES_BOOLEA...	3
run_es_test	BANK_TEST_INSTRUCTOR1	ES_TEST	22
run_es_test	ROOT	ES_SUITE	9
run_espec	ROOT	ES_TESTABLE	2
make	ROOT	ROOT	4

A: Which contract is failing

D: Correct Result

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			
Result	True	BOOLEAN	

C: Incorrect Result

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*.

Test t5 fails because object equality for class CUSTOMER is not properly defined.

Hint: (1) You will want to redefine {CUSTOMER}.*is_equal* as is done in class ACCOUNT. (2) Eventually you may want to inherit from COMPARABLE so that you can sort customers alphabetically.

Start by understanding the first 5 tests and getting them to work. Do not change the tests. Rather change the class CUSTOMER as needed.

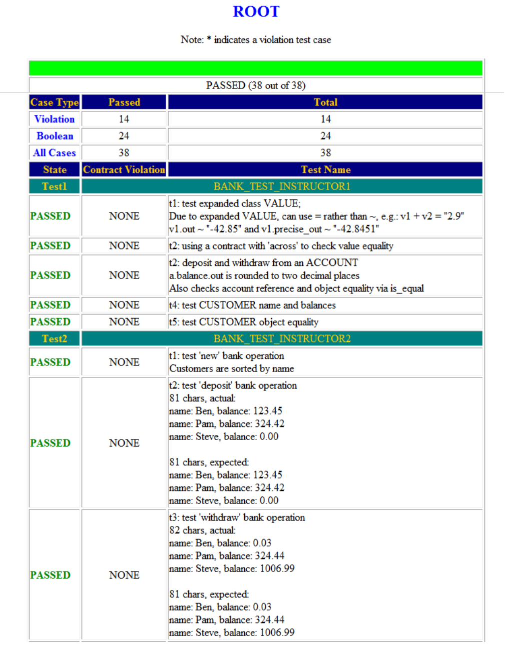
- Test t1 helps you to understand class VALUE for precise arithmetic. It uses Eiffel's design notation of *reference* vs. *expanded* semantics.[4] Class VALUE does precise arithmetic and can thus be be used for the bank system.
- Test t2 will help you understand and use the *across* notation for writing contracts.[5] It will also help you understand the VALUE semantics. Don't change this test. You may change the query {BANK_TEST_INSTRUCTOR1}.*equal_values*.
- Test t3 checks that class ACCOUNT is working correctly.
- Test t4 checks that the basic features of class CUSTOMER are working correctly.

Test t5 checks that object equality for CUSTOMER is working correctly

4.1 What we would eventually like to see

Eventually, you must revise classes CUSTOMER and BANK so that all the tests run and you obtain the green bar, as on the next page. To do this, you must uncomment the test cases in ROOT and get all the tests to work. You may only change the following:

- Query {BANK_TEST_INSTRUCTOR1}.*equal_values*.
- Class CUSTOMER as needed
- Class BANK as needed.
- You add your tests to class STUDENT_TESTS
- You may comment and uncomment tests in class ROOT.[6]
- Do not change anything else.



ROOT

Note: * indicates a violation test case

PASSED (38 out of 38)

Case Type	Passed	Total
Violation	14	14
Boolean	24	24
All Cases	38	38
State	Contract Violation	Test Name
Test1	BANK_TEST_INSTRUCTOR1	
PASSED	NONE	t1: test expanded class VALUE; Due to expanded VALUE, can use = rather than ~, e.g.: v1 + v2 = "2.9" v1.out ~ "-42.85" and v1.precise_out ~ "-42.8451"
PASSED	NONE	t2: using a contract with 'across' to check value equality
PASSED	NONE	t2: deposit and withdraw from an ACCOUNT a.balance.out is rounded to two decimal places Also checks account reference and object equality via is_equal
PASSED	NONE	t4: test CUSTOMER name and balances
PASSED	NONE	t5: test CUSTOMER object equality
Test2	BANK_TEST_INSTRUCTOR2	
PASSED	NONE	t1: test 'new' bank operation Customers are sorted by name
PASSED	NONE	t2: test 'deposit' bank operation 81 chars, actual: name: Ben, balance: 123.45 name: Pam, balance: 324.42 name: Steve, balance: 0.00 81 chars, expected: name: Ben, balance: 123.45 name: Pam, balance: 324.42 name: Steve, balance: 0.00
PASSED	NONE	t3: test 'withdraw' bank operation 82 chars, actual: name: Ben, balance: 0.03 name: Pam, balance: 324.44 name: Steve, balance: 1006.99 81 chars, expected: name: Ben, balance: 0.03 name: Pam, balance: 324.44 name: Steve, balance: 1006.99

[4] See INTEGER_REF_32 (reference semantics) and INTEGER_32 (expanded) for an illustration of the concepts. See also OOSC2.

[5] It is based on the iterator pattern that we will one day study in class.

[6] You may also uncomment tests such as *add_boolean_case (agent t1)* in order to temporarily simplify the ESpec output, but you must then eventually uncomment all tests.

When running ESpec, at the command line, you should see something like the following:

```
passing tests
> BANK_TEST_INSTRUCTOR1.t1
> BANK_TEST_INSTRUCTOR1.t2
> BANK_TEST_INSTRUCTOR1.t2
> BANK_TEST_INSTRUCTOR1.t4
> BANK_TEST_INSTRUCTOR1.t5
> BANK_TEST_INSTRUCTOR2.t1
> BANK_TEST_INSTRUCTOR2.t2
> BANK_TEST_INSTRUCTOR2.t3
> BANK_TEST_INSTRUCTOR2.t4
> BANK_TEST_CONTRACTS1.t_new_total_balance_unchanged
> BANK_TEST_CONTRACTS1.t_new_pre_1
> BANK_TEST_CONTRACTS1.t_deposit_pre_1
> BANK_TEST_CONTRACTS1.t_deposit_pre_2
> BANK_TEST_CONTRACTS1.t_deposit_pre_3
> BANK_TEST_CONTRACTS1.t_deposit_pre_4
> BANK_TEST_CONTRACTS1.t_withdraw_pre_1
> BANK_TEST_CONTRACTS1.t_withdraw_pre_2
> BANK_TEST_CONTRACTS1.t_withdraw_pre_3
> BANK_TEST_CONTRACTS1.t_withdraw_pre_4
> BANK_TEST_CONTRACTS1.t_withdraw_pre_5
> BANK_TEST_CONTRACTS1.t_transfer_precond_1
> BANK_TEST_CONTRACTS1.t_transfer_precond_2
> BANK_TEST_CONTRACTS1.t_transfer_precond_3
> BANK_TEST_CONTRACTS1.t_transfer_precond_4
> BANK_TEST_CONTRACTS1.t_new_total_balance_unchanged
> BANK_TEST_CONTRACTS1.t_new_num_customers_increased
> BANK_TEST_CONTRACTS1.t_new_customer_added_to_the_end
> BANK_TEST_CONTRACTS1.t_deposit_total_balance_increased
> BANK_TEST_CONTRACTS1.t_deposit_num_customers_unchanged
> BANK_TEST_CONTRACTS1.t_deposit_customer_balance_increased
> BANK_TEST_CONTRACTS1.t_withdraw_total_balance_decreased
> BANK_TEST_CONTRACTS1.t_withdraw_num_customers_unchanged
> BANK_TEST_CONTRACTS1.t_withdraw_customer_balance_decreased
> BANK_TEST_CONTRACTS1.t_withdraw_other_customers_unchanged
> BANK_TEST_CONTRACTS1.t_transfer_total_balance_unchanged
> BANK_TEST_CONTRACTS1.transfer_customer_balance_decreased
> BANK_TEST_CONTRACTS1.transfer_customer_balance_increased
> BANK_TEST_CONTRACTS1.transfer_other_customers_unchanged
failing tests
38/38 passed
passed
```

In addition, you must write at least 3 additional interesting student tests.

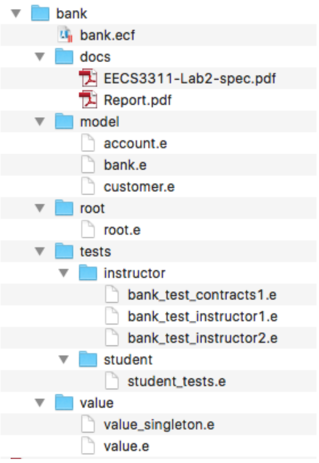
4.2 What you must do

Appendix 5.1 describes what must be in your report *Report.pdf* (about 3 pages). You must print your report and place it in the course DropBox. The report is also part of your electronic submission described below.

1. Uncomment all the tests in the ROOT class
2. Get all the tests to work by making appropriate changes to class BANK and CUSTOMER.
    - **Do not** change the contract *tags* (in preconditions, postconditions, etc.) in classes CUSTOMER and BANK; however, you must complete the contract bodies appropriately to describe the bank specification.
    - You must revise or complete all the routine implementations (usually with empty bodies) so as to satisfy the contracts.
    - Usually there is a `--ToDo` signal where you make the changes.
3. Write at least 3 reasonable tests of your own in STUDENT_TESTS.
4. On Prism (Linux), *eclean* your system, freeze it, and re-run all the tests to ensure that you get the green bar.
5. *eclean* your directory *bank* again to remove all EIFGENs.
6. Submit your Lab as follows: `submit 3311 Lab2 bank`

Remember

- Your code must compile and execute on the departmental Linux system (Prism).
- 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 *bank* is as follows:



```
bank
    bank.ecf
    docs
        EECS3311-Lab2-spec.pdf
        Report.pdf
    model
        account.e
        bank.e
        customer.e
    root
        root.e
    tests
        instructor
            bank_test_contracts1.e
            bank_test_instructor1.e
            bank_test_instructor2.e
        student
            student_tests.e
    value
        value_singleton.e
        value.e
```

5 Appendix

5.1 The Report

You must produce a report that is printed and handed in at the DropBox. It is also submitted electronically as *Report.pdf*.

The Report must have the following components

1. *Title*: EECS3311-F16 – Lab2 – Bank
2. *Name of Student*
3. *Prism login* of student submission
4. Statement describing which parts of the work was not completed
    a. Comment why it was not completed
5. Chart View of class BANK (using RTF)
6. Contract View of class BANK only for features *deposit*, *withdraw*, and *transfer* and *class invariants*
    a. In the contracts, use mathematical notation $\forall$, $\exists$, etc. rather than the more wordy **across** notation.
7. Use the BON/Visio stencil to construct a BON class diagram consisting of ACCOUNT, CUSTOMER and BANK. In this diagram ACCOUNT and CUSTOMER are displayed in compressed form, but BANK is displayed in un-compressed form showing the signatures of deposit, withdraw and transfer and the invariants (again using shorthand mathematical notation).[7]

5.2 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.*[8]

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.

[7] https://wiki.eecs.yorku.ca/project/eiffel/bon:start. The Visio templates for BON are at this URL. The graphical shapes of BON can be drawn using Microsoft Office Visio (available on Windows workstations in the Prism Lab LAS1002, or for your Laptop via Dreamspark).

[8] *null* in other languages.

```
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).

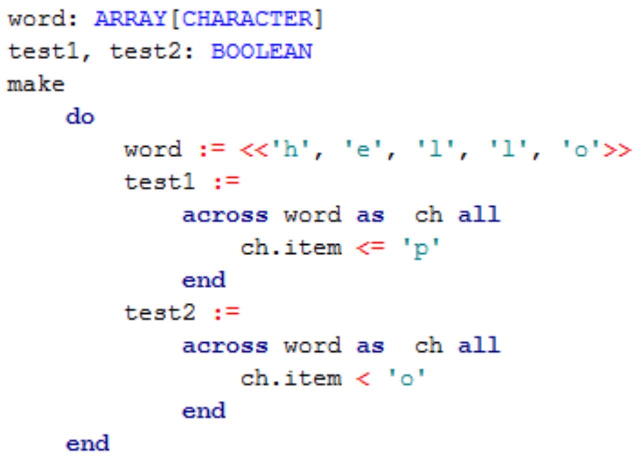
You can read more about reference and expanded types in OOSC2 sections 8.1 to 8.8. These sections also explain copy (*twin*) and deep copy (*deep_twin*). Many of the notions in these sections should already be familiar to you from earlier courses. Expanded types are discussed in Section 8.7.

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 /.

5.3 A2 Using *across* in contracts and debugger breakpoints

The failing test t2 detects an error in the query {BANK_TEST_INSTRUCTOR1}.*equal_values* which you must correct. 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:

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

which is true, and *test2* asserts that

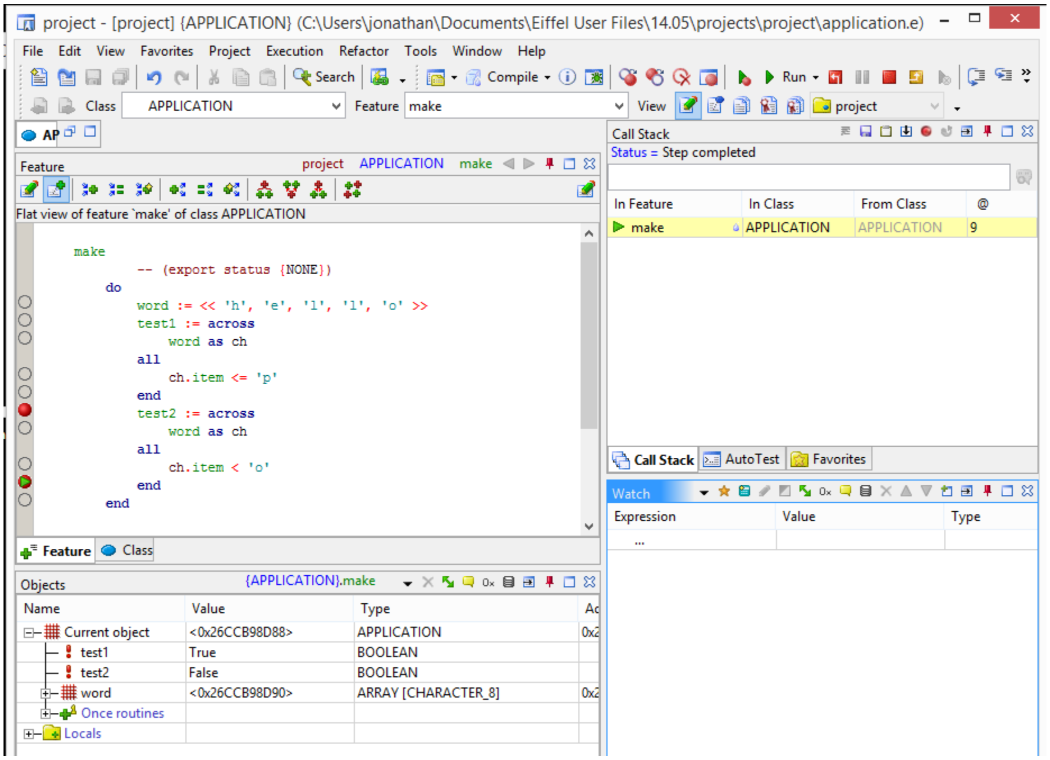
$$\forall\ ch \in word: ch < \text{‘o’}$$

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*.



project - [project] {APPLICATION} (C:\Users\jonathan\Documents\Eiffel User Files\14.05\projects\project\application.e)

Flat view of feature `make' of class APPLICATION

```
make
        -- (export status {NONE})
    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
```

Status = Step completed

In Feature	In Class	From Class	@
make	APPLICATION	APPLICATION	9

Expression	Value	Type
...		

{APPLICATION}.make

Name	Value	Type	Ac
Current object	<0x26CCB98D88>	APPLICATION	0x2
test1	True	BOOLEAN	
test2	False	BOOLEAN	
word	<0x26CCB98D90>	ARRAY [CHARACTER_8]	0x2
Once routines			
Locals			

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

5.4 A2 Using the debugger

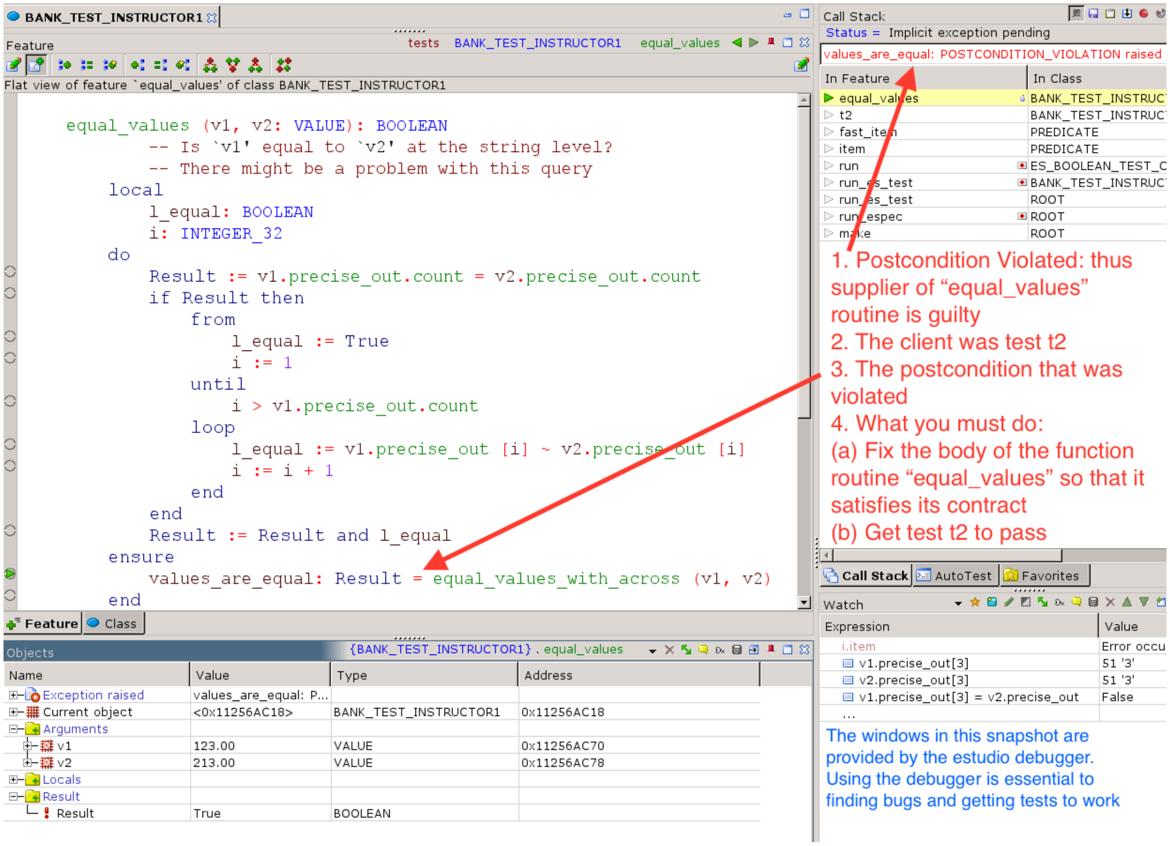
The first two tests *t1* and *t2* help you to understand how to use the expanded class VALUE.

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.



Flat view of feature `equal_values' of class BANK_TEST_INSTRUCTOR1

```
equal_values (v1, v2: VALUE): BOOLEAN
        -- Is `v1' equal to `v2' at the string level?
        -- There might be a problem with this query
    local
        l_equal: BOOLEAN
        i: INTEGER_32
    do
        Result := v1.precise_out.count = v2.precise_out.count
        if Result then
            from
                l_equal := True
                i := 1
            until
                i > v1.precise_out.count
            loop
                l_equal := v1.precise_out [i] ~ v2.precise_out [i]
                i := i + 1
            end
        end
        Result := Result and l_equal
    ensure
        values_are_equal: Result = equal_values_with_across (v1, v2)
    end
```

Status = Implicit exception pending

values_are_equal: POSTCONDITION_VIOLATION raised

In Feature	In Class
equal_values	BANK_TEST_INSTRUC
t2	BANK_TEST_INSTRUC
fast_item	PREDICATE
item	PREDICATE
run	ES_BOOLEAN_TEST_C
run_es_test	BANK_TEST_INSTRUC
run_es_test	ROOT
run_espec	ROOT
make	ROOT

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

{BANK_TEST_INSTRUCTOR1} . equal_values

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			
Result	True	BOOLEAN	

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
...	

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

5.5 Object vs. Reference

See OOSC2

5.6 Class LIST[G] with generic parameter G

See OOSC2