Eiffel 101: Language, Method and Tools

A basic and elementary introduction for students in EECS3311 • Develop competence in the methods and tools —in the first 3 weeks of the term • Learn by doing all the examples • Compile, run, test, debug, document and understand.

Resources

- http://eiffel.eecs.yorku.ca
- https://www.eiffel.org/documentation

Available Online in the Steacie Library

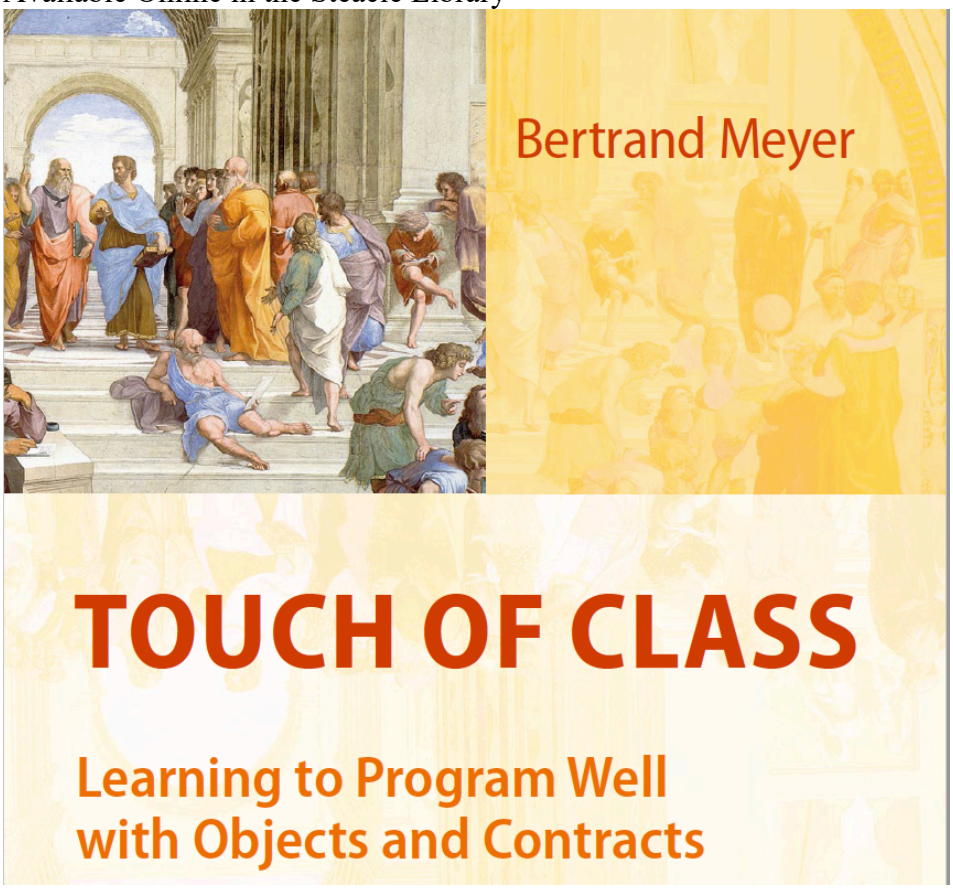


Table of Contents

1 Overview 3
2 Eiffel Syntax 4
3 Start a New Project 4
4 Launch EiffelStudio IDE and compile 6
5 Run ESpec Unit Tests 8
5.1 Unit Tests succeeds with a Green Bar 10
6 Contract violations and the debugger 12
7 Adding a new class NODE [G] 15
8 Void safety and class NODE [G] 17
8.1 VEVI compile time error: 19
8.2 VUAR error 20
9 Unit Tests for class NODE [G] 21
10 Object equality and is_equal 23
11 Reference vs. Expanded Types 27
12 Using the debugger 28
13 Iteration using the “across” notation in the debugger 28
14 Using the IDE to generate Documentation 31
15 Advice for writing comprehensive tests 32
15.1 Compile Time Errors 33
15.2 Bugs 33
15.3 Debugging 34
15.4 Using unit tests and the debugger 35
16 BON/UML class diagrams 36
16.1 Architecture: design structure 37
17 Abstraction, DbC and Information Hiding 38
17.1 Contract View 38
18 Mathmodels 40
18.1 Specifying stacks with Mathmodels 42

1 Overview

We use Eiffel in EECS3311 because programs in this language have both *implementation* and *design* constructs. And this is a course about design—requiring that you think above the implementation level. This document is an elementary introduction to the method and tools.

The Eiffel Method:

- Is Based on a small number of powerful ideas from computer science and software engineering
- One example is Design by Contract:
  - Defines a software system as a set of components interacting through precisely specified contracts. Contracts are active and enforceable throughout the life-cycle
  - Promotes precise software specification and software reliability
  - Uses a "single-product" model. All life-cycle phases are supported by a single notation. Less need to switch, say, from "specification language" to "design language"
  - Seamlessness: turns software construction into a single continuous process from specifications, analysis, design and implementation.
- Another example is true multiple inheritance as expressed in UML
- BON class diagrams document the architecture of the Design

The Eiffel Programming Language

- Exists to express the products of the Eiffel Method
- Contracts and contract monitoring
- Exception handling based on software specification (versus ad hoc try/catch)
- Void-safety: calls on void (null) references are eliminated at compile time
- Multiple Inheritance: includes multiple and repeated inheritance
- Genericity, constrained genericity and functional programming constructs
- Platform independent concurrency (SCOOP)

To make use of these software design ideas, you must become comfortable using the language and tool basics in the first three weeks of the term. This is also a time when you do not yet have many assignments and tests.

- This introduction is for students in EECS3311, working on EECS Linux workstations or the SEL Virtual Machine (SEL-VM)[1].
- See http://eiffel.eecs.yorku.ca for more information; it also contains information if you wish to install Eiffel on your own Laptop.[2]
- We assume that you have done Lab0: http://seldoc.eecs.yorku.ca/doku.php/eiffel/hello/hello2/start

[1] http://seldoc.eecs.yorku.ca/doku.php/eiffel/virtualbox/start
[2] https://wiki.eecs.yorku.ca/course_archive/2018-19/W/3311/_media/wiki:eecs3311-starter-guide.pdf

2 Eiffel Syntax

It is assumed that you know basic Eiffel Syntax.[3]

```
class
    PERSON
feature
    age: INTEGER assign set_age
    set_age ( new_age: INTEGER )
        do
            age := new_age
        end
end
```

Now `person.age := 21` is simply a shortcut for `person.set_age(21).` If you are not yet familiar with the syntax, review: https://www.eecs.yorku.ca/~eiffel/eiffel-guide/

By convention, classes are written upper-case and features (attributes, function routines and command routines are written in lower case.[4]

Exercise: How would you add a new query for the name of a person? How do you write public queries and how do you write private queries?

3 Start a New Project

At the command line, invoke *eiffel-new*: [5]

`> eiffel-new` `New Eiffel void-safe project name:` node-demo The directory and file structure is:	A new project called *node-demo* is created. By default, the project is *Void Safe*; more on this later. Three clusters are created (*model* which is empty, *root* and *tests*). New classes will be created in cluster *model*. The ECF file is `node-demo.ecf` The root class is in a file called `root.e`

[3] For more detail, see: https://www.eiffel.org/doc/eiffel/Eiffel_programming_language_syntax
[4] For style guidelines see: https://www.eiffel.org/doc/eiffel/Style_Guidelines
[5] Introduced in Lab0: http://seldoc.eecs.yorku.ca/doku.php/eiffel/hello/hello2/start.

```
node-demo/
├── model
├── node-demo.ecf
├── root
│   └── root.e
└── tests
    └── tests.e
```

Optional:

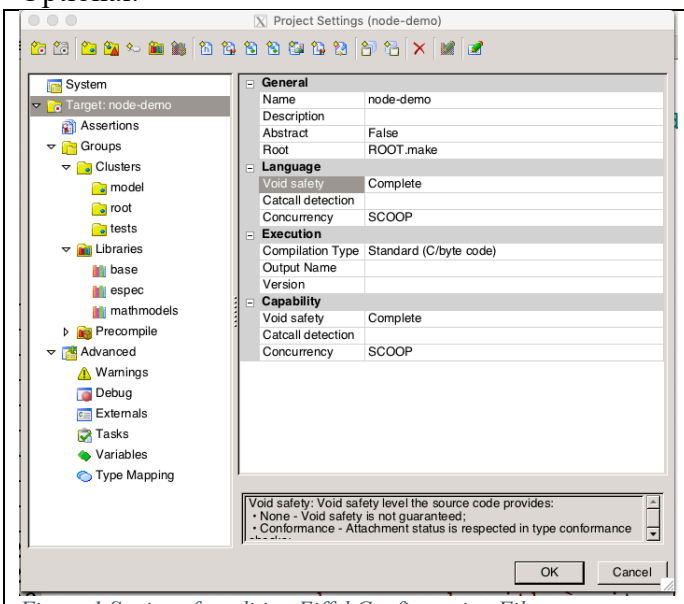


Figure 1 Settings for editing Eiffel Configuration Files

- Although it is not necessary to do so at this stage —you may optionally open the ECF (Eiffel Configuration file) `node-demo.ecf` in a text editor. It is in XML.
- To edit the ECF file, it is easier to use *Settings* in the IDE as in Figure 1.
- *Settings* shows the various clusters (*model*, *root*, *tests*) and libraries (*base*, *espec*, *mathmodels*) in use.
- There are many other configurations in the ECF file accessible via Settings.

Ensure that you have enough quota before launching the IDE and compiling:

```
> quota -v
/cs/home/student 667.9 of 976.6 MB used
```

Your Eiffel program text files (e.g. `root.e`) do not take up much space.

But, Eiffel programing text is translated into C and compiled in a directory called EIFGENs[6]. You need at least 300Mb of temporary memory free, especially if your EIFGENs folder will be in your home folder and not in */tmp/student* (where *student* is your EECS login).

[6] https://www.eiffel.org/doc/eiffelstudio/A_Look_at_the_Project_Directory: Every project has a project directory which will contain the files generated and managed by EiffelStudio. The project directory may also host some of the source files containing your Eiffel classes, the ECF (eiffel configurationl file), and external software written in other languages. You will notice a subdirectory called EIFGENs, for "EIFfel GENerations". EIFGENs is created and maintained by the compiler to store information about your project, including generated code (in our case in C) for execution. EiffelStudio manages your project in such a way that EIFGENs can always be re-generated if need be; this means that if things go wrong for any reason and you want to make a fresh start you can always delete this directory and recompile your system. This also means that you should not add any files into this directory, or modify any of its files, since a later compilation is free to change or regenerate whatever it chooses in EIFGENs

You might want to create a temporary folder for your EIFGENs like this:

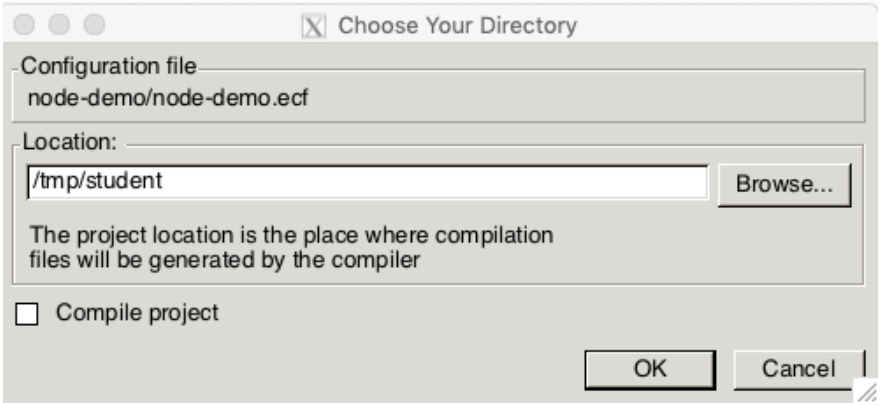
```
> mkdir /tmp/$USER
```

For me, $USER is `student`. You can also do: `mkdir /tmp/student`.

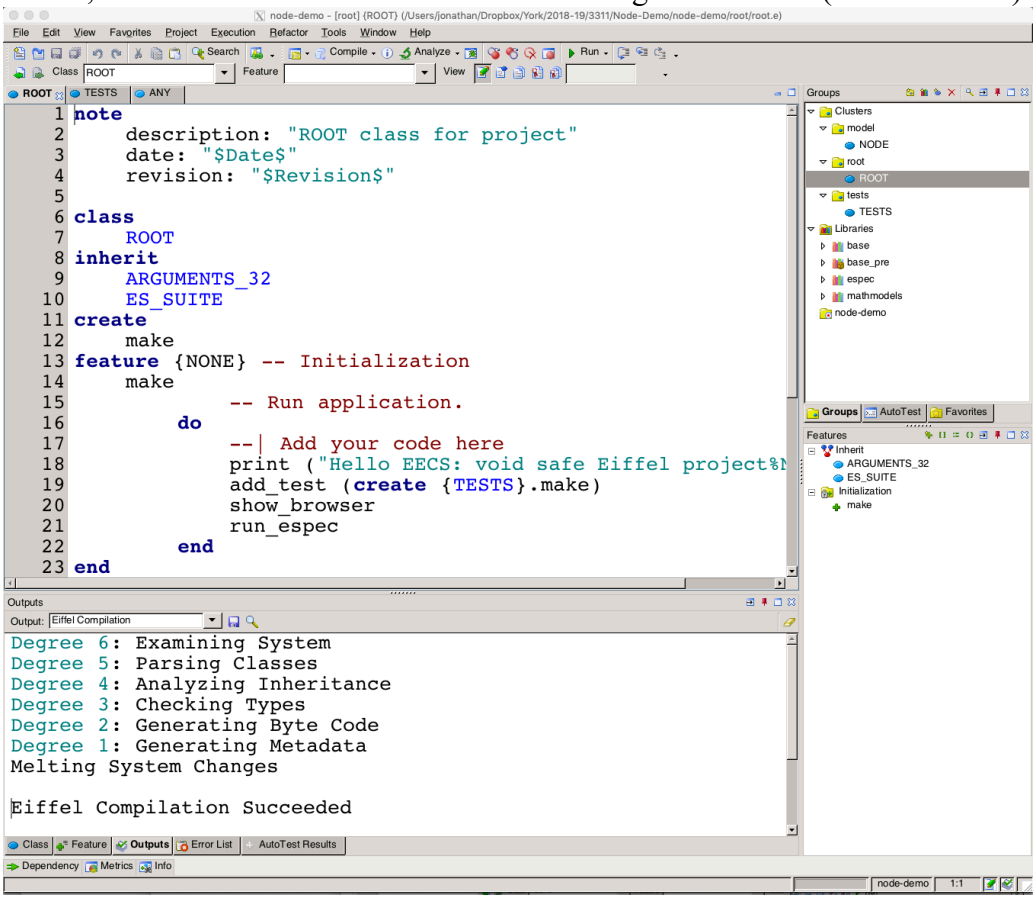
4 Launch EiffelStudio IDE and compile

Now launch your project in the IDE and compile: `>estudio node-demo/node-demo.ecf &`

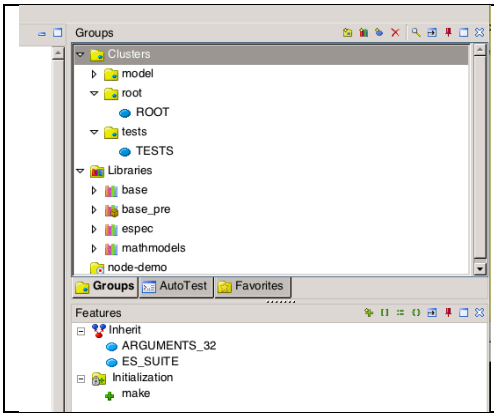
A dialogue box asks you where to compile the EIFGENs for your project (you may choose the default if you have enough quota). Below, I choose the temporary directory that I created earlier:

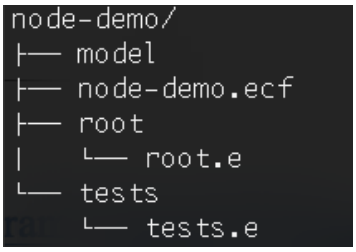


Below, we show a screen shot of the IDE showing class ROOT (in file `root.e`).

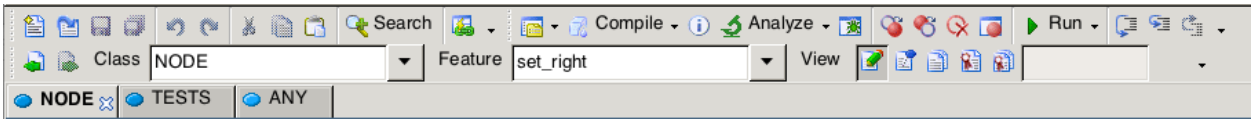


The *make* routine of the root class adds a class TESTS where you may start writing unit tests. The cluster and class structure in the top right IDE window pane —mirrors the project directory structure.





A user may browse classes and *feature* (attributes, function routines, command routines) of classes:



To learn more about browsing the classes in the project and libraries, see
https://www.eiffel.org/doc/eiffelstudio/Starting_To_Browse

The default method of using the IDE is pick and drop.
https://www.eiffel.org/doc/eiffelstudio/Retargeting_Through_Pick-and-Drop

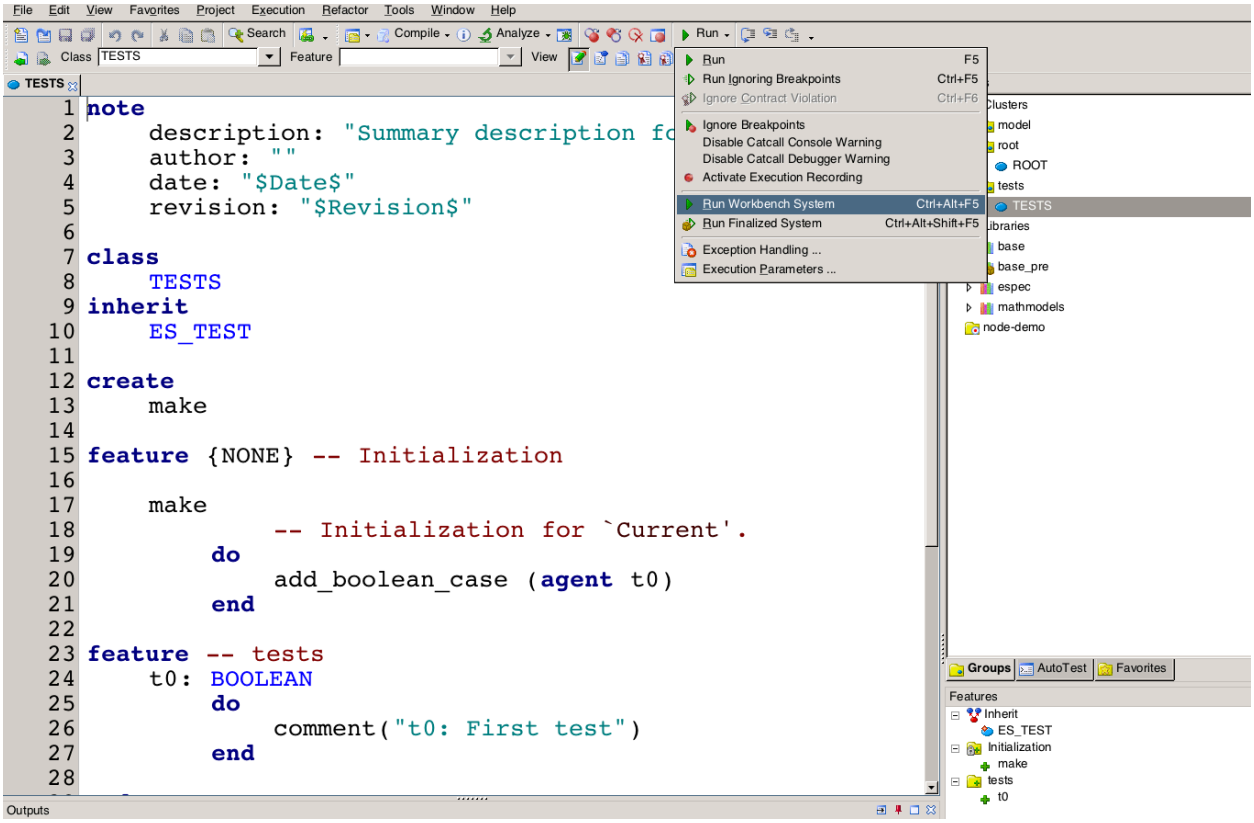
Pick-and-Drop consists of three steps:

- Pick step: find the development object (e.g. a class or a feature) and pick it: either through the context menu, or by right-click, depending upon EiffelStudio's Pick and Drop Mode.
- Move step: move the mouse to the desired drop point, without pressing any button.
- Drop step: right-click (again releasing the button immediately) at the drop position.

The Pick-and-Drop mechanism relies on the metaphor of *pebbles* and *holes*. When you pick a development object, the cursor changes into a *pebble* whose shape represents the type of the development object: cluster, class, feature, run-time object ... You may then drop it into a *hole*, which can be a window, a tree view entry, or a hole-shaped icon. This performs the appropriate action such as retargeting a tool.

5 Run ESpec Unit Tests

The *ESpec* library is already part of the project. Choose class TESTS to see what a unit test looks like. Below you can see test `t0`. We also show how to run unit testing.



The default unit test `t0` is a function routine with return type BOOLEAN. The return value of a function routine is set by assigning it to the `Result` variable. Unlike other languages, the *return* statement does not exist. Since `Result` for this test is never set to *True*, the test will fail.

You may use any name for your tests. It is essential that every ESpec test, say *testfoo*, starts with a *comment* taking an argument `"testfoo: …"`.

For more on unit testing using *ESpec*, see[7]:
http://seldoc.eecs.yorku.ca/doku.php/eiffel/espec/start

To run all the unit tests, invoke *Run Workbench System* (shortcut Control-Alt-F5). The unit tests are executed and the browser displays the result with a red bar indicating that at least one test failed:

[7] EiffelStudio IDE also has a built-in unit testing facility (similar to Junit). In this course for a variety of erasons we use ESpec (designed by our team). For more on the IDE unit testing see: https://www.eiffel.org/doc/eiffelstudio/Create_a_manual_test.

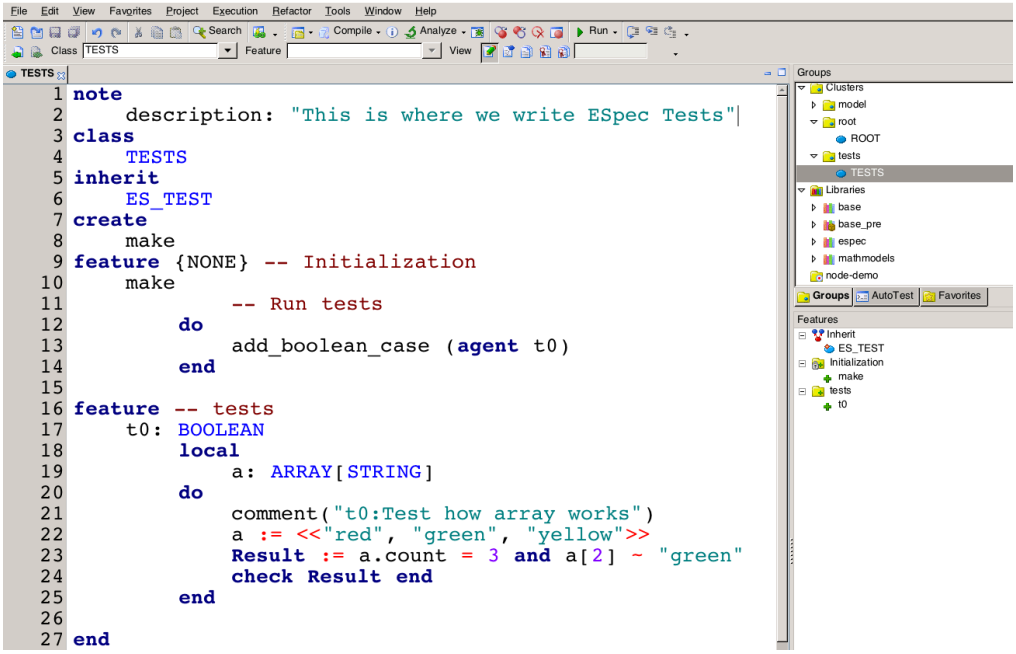
ROOT

Note: * indicates a violation test case

FAILED (1 failed & 0 passed out of 1)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	0	1
All Cases	0	1
State	**Contract Violation**	**Test Name**
Test1	TESTS	
FAILED	NONE	t0: First test

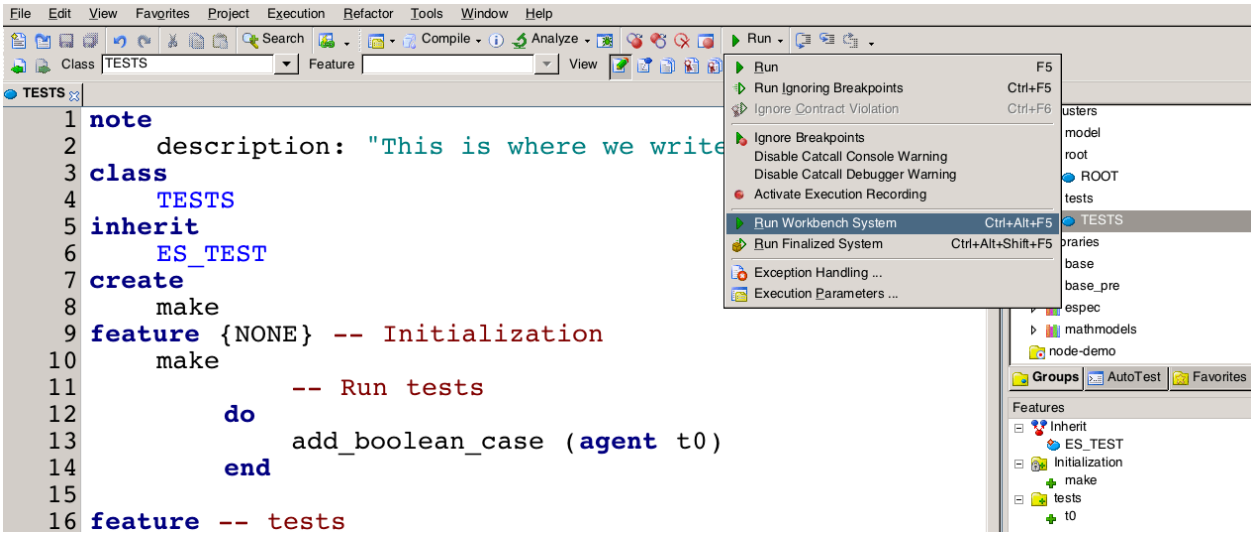
5.1 Unit Tests succeeds with a Green Bar

Let's write a test to check if we understand how arrays work in Eiffel. The revised unit test `t0` (below), creates an array of three strings and then checks that there are three entries in the array.

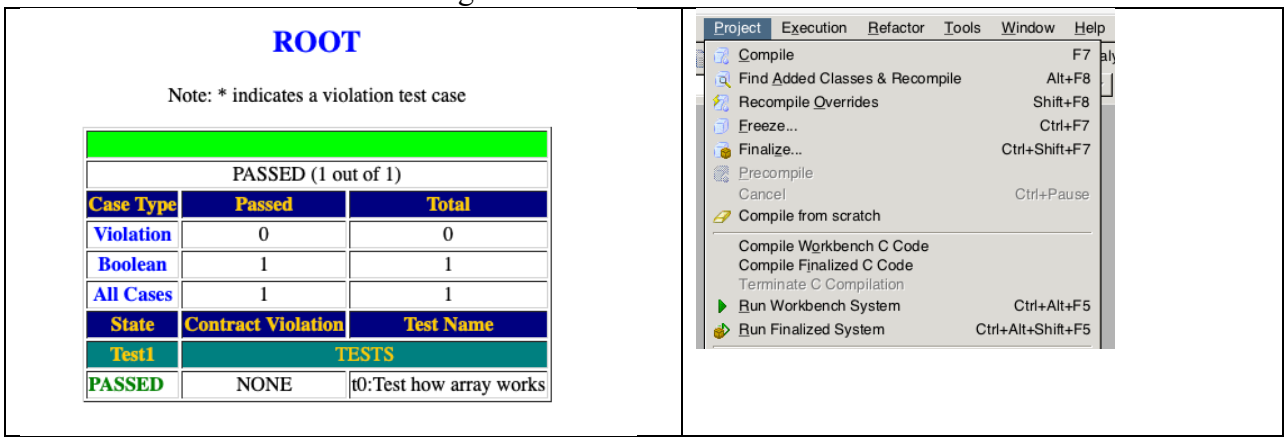



```
1  note
2      description: "This is where we write ESpec Tests"
3  class
4      TESTS
5  inherit
6      ES_TEST
7  create
8      make
9  feature {NONE} -- Initialization
10     make
11             -- Run tests
12         do
13             add_boolean_case (agent t0)
14         end
15
16 feature -- tests
17     t0: BOOLEAN
18         local
19             a: ARRAY[STRING]
20         do
21             comment("t0:Test how array works")
22             a := <<"red", "green", "yellow">>
23             Result := a.count = 3 and a[2] ~ "green"
24             check Result end
25         end
26
27 end
```

Compile and re-run the unit (Control-Alt-F5):



This time the tests succeed with a green bar:



Compile (F7)	Melt: quick incremental recompilation, doesn't optimize code for changed parts. Use this to check syntax and compile errors.
Compile/Freeze (Ctrl-F7)	Freeze: incremental recompilation, not as fast as Melt. but generates more efficient code for changed parts. Use this before running ESpec.
Compile/Finalize	
Run (F5)	EIFGENs/W_code (Workbench bytecode). Use this to debug your code, e.g. when contracts fail.
Run Workbench System (Ctrl-Alt-F5)	Use this to obtain a browser report (red/green bar).
Run Finalized System	EIFGENS/F_Code. For deploying your software product.

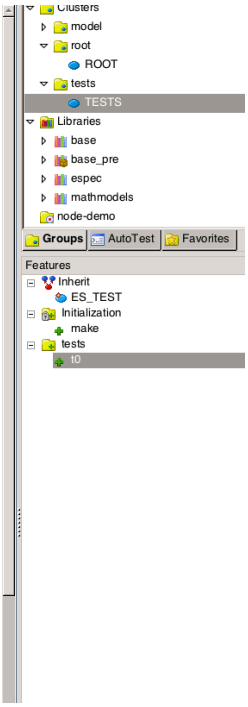
The result of melting is compiled into bytecode in the W_code directory for further C processing. Freezing does a full translation into C.[8]

[8] https://www.eiffel.org/doc/eiffelstudio/How_EiffelStudio_Compiles

6 Contract violations and the debugger

Suppose we extend test t0 with `Result := a[4] ~ "yellow"` (see below):

```
note
    description: "This is where we write ESpec Tests"
class
    TESTS
inherit
    ES_TEST
create
    make
feature {NONE} -- Initialization
    make
            -- Run tests
        do
            add_boolean_case (agent t0)
        end

feature -- tests
    t0: BOOLEAN
        local
            a: ARRAY[STRING]
        do
            comment("t0:Test how array works")
            a := <<"red", "green", "yellow">>
            Result := a.count = 3 and a[2] ~ "green"
            check Result end
            Result := a[4] ~ "yellow"
            check Result end
        end

end
```



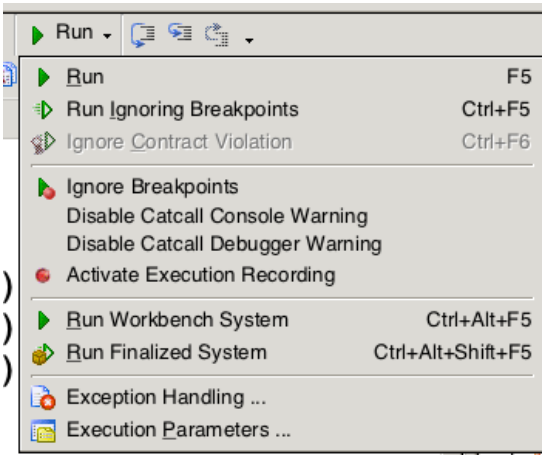
By default, arrays in Eiffel start at index 1. There is no entry at index 4. Thus, we now obtain a red bar indicating that one or more tests have failed. The failure is at a precondition violation.

ROOT

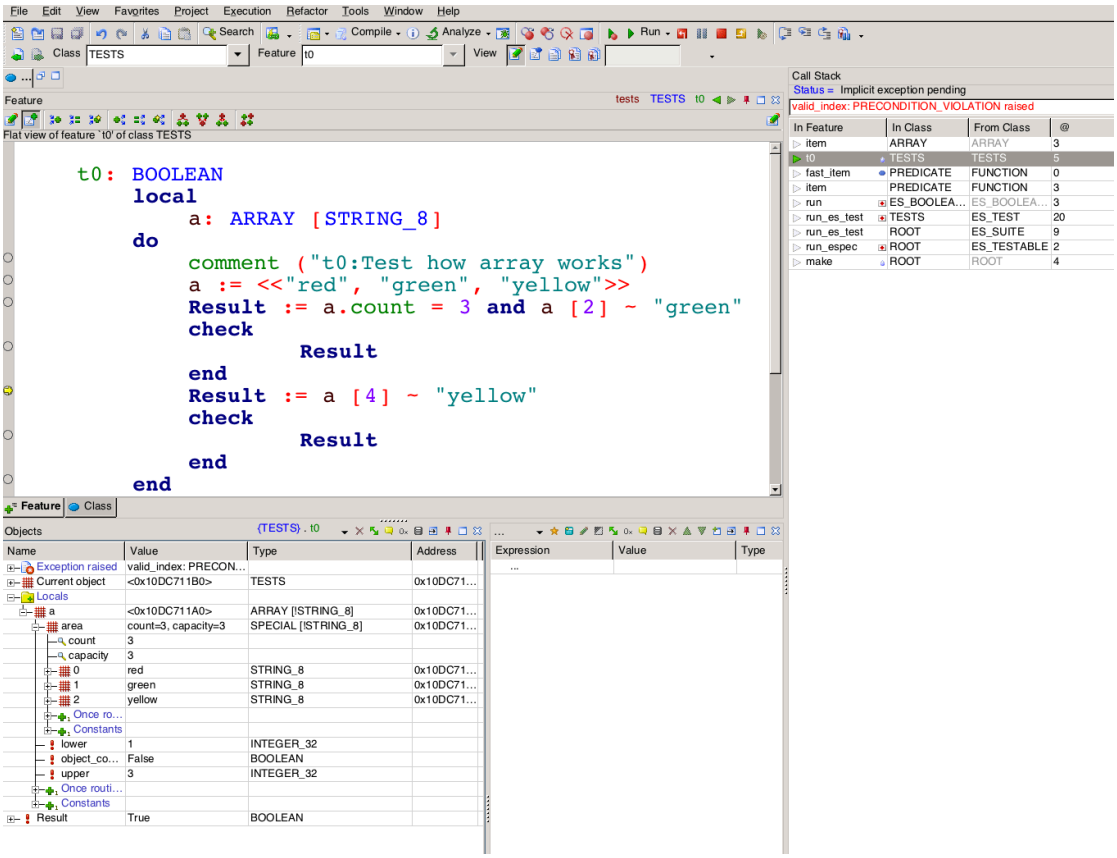
Note: * indicates a violation test case

FAILED (1 failed & 0 passed out of 1)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	0	1
All Cases	0	1
State	**Contract Violation**	**Test Name**
Test1	**TESTS**	
FAILED	Precondition violated.	t0:Test how array works

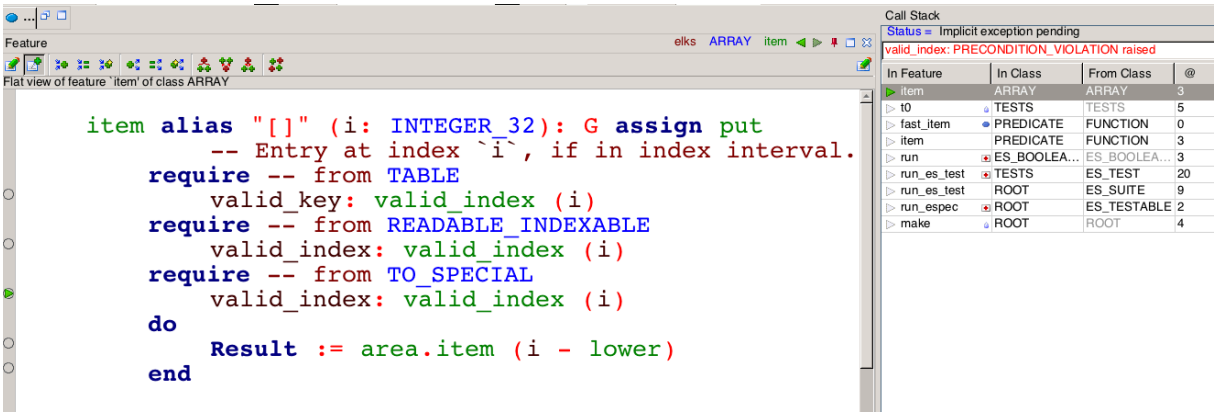
Run (shortcut F5) the system:



Given that there is a precondition violation, the IDE drops into debugger mode. A yellow arrow points to the line in the code that fails. In the top right hand window, we can see that this line of code is in test `t0`.



If we select *item* (from ARRAY) in the *Call Stack* pane, then the debugger indicates that the failure is in the precondition with tag `valid_index`. The debugger will also show that the argument `i` is 4 which is not a valid index into the array.



We can make the test succeed, by inserting a line: `a.force ("yellow", 4).` Arrays in Eiffel may be resized, and *force* will ensure that there is an index at 4.

```
t0: BOOLEAN
    local
        a: ARRAY[STRING]
    do
        comment("t0:Test how array works")
        a := <<"red", "green", "yellow">>
        Result := a.count = 3 and a[2] ~ "green"
        check Result end
        a.force ("yellow", 4)
        Result := a[4] ~ "yellow"
        check Result end
    end
```

It is vital that you become skilled in using the debugger. For more see https://www.eiffel.org/doc/eiffelstudio/Debugging_and_Run-time_Monitoring

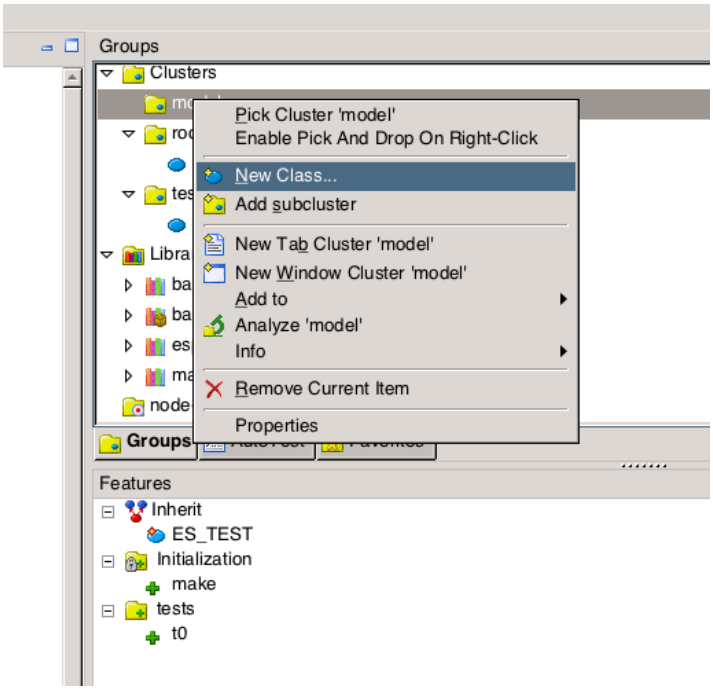
Exercises: Try the above array test `t0`, use the debugger and write more tests of your own. Write and test class PERSON (see Section 2).

Exercises: Writing a test is a smart way to check that you understand how to use a class. Write tests to check your understanding of some collection classes in the base library, e.g.

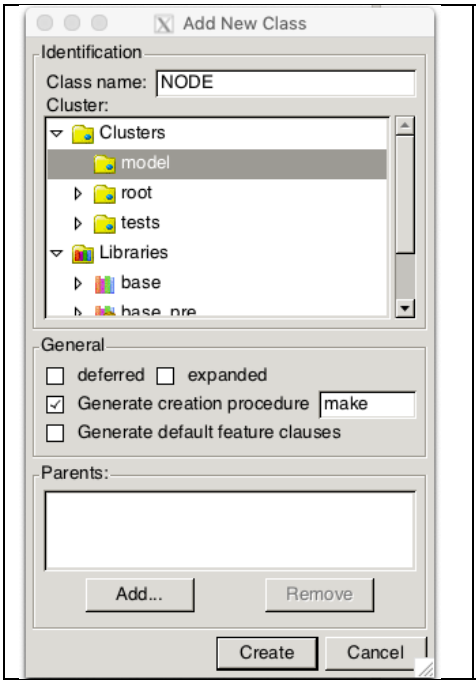
- ARRAY[G]
- List implementations: LINKED_LIST [G] and ARRAYED_LIST [G]. The abstract data type is LIST [G]
- HASH_TABLE [G, H]

7 Adding a new class NODE [G]

In the IDE, select the relevant cluster (*model*) and click on *New Class*.



A dialogue box displays:



A new class NODE is created to which we add a generic parameter *G*:

```
 7 class
 8     NODE[G]
 9
10 create
11     make
12
13 feature {NONE} -- Initialization
14
15     make
16             -- Initialization for `Current'.
17         do
18
19         end
20
21 end
```

We will work with a class NODE as shown in Figure 2 below

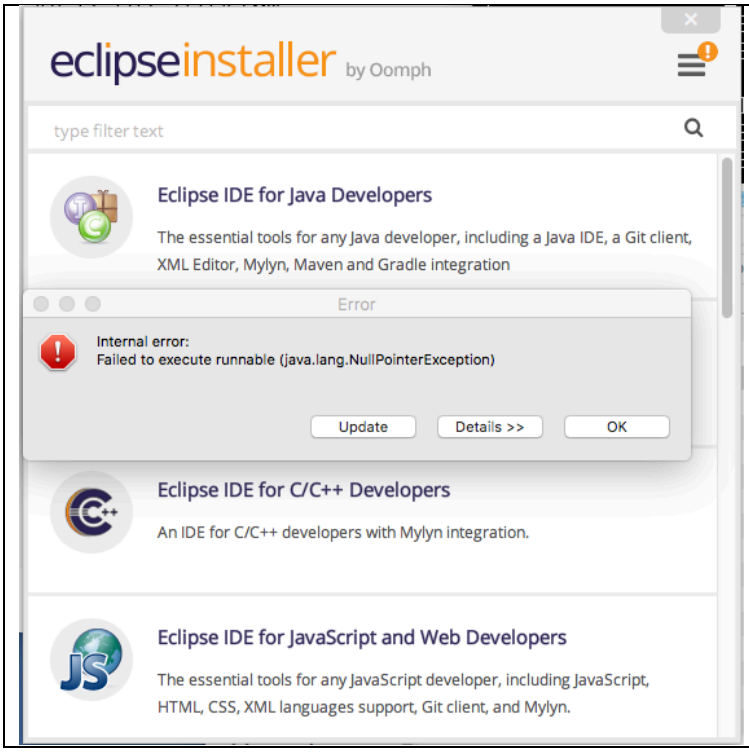
```
note
	description: "[
		A node has left and right nodes and possibly a parent
	]"
class  NODE [G] create
	make

feature {NONE} -- constructor
	make (a_item: G)
			-- makes a node with a_item
		do
			item := a_item
		end

feature -- attributes
	item: G assign set_item
			-- returns the current item

	left: detachable NODE [G] assign set_left
			-- reference to left child

	right: detachable NODE [G] assign set_right
			-- reference to right child

	parent: detachable NODE [G] assign set_parent
			-- reference to parent

feature -- commands
	set_item (a_item: G)
			-- sets Current item to a_item
		do
			item := a_item
		end

	set_left (a_node: detachable NODE [G])
			-- updates left child to a_node
			-- updates left child's parent to Current if attached
		do
			left := a_node
			if attached left as l then
				l.parent := Current
			end
		ensure
			node_set: left = a_node
			child_has_correct_parent:
				attached left as l implies l.parent = Current
		end

	set_right (a_node: detachable NODE [G])
			-- updates right child to a_node
			-- updates right child's parent to Current if attached
		do
			right := a_node
			if attached right as r then
				r.parent := Current
			end
		ensure
			node_set: right = a_node
			child_has_correct_parent:
				attached right as r implies r.parent = Current
		end

	set_parent (a_node: detachable NODE [G])
			-- updates parent to a_node
		do
			parent := a_node
		end
invariant
	left_parent: attached left as l_left implies l_left.parent = Current
	right_parent: attached right as l_right implies l_right.parent = Current
end
```

Figure 2 Program text for class NODE[G]

Note the class invariants in Figure 2. All routines must preserve these invariants. Class invariants ensure the consistency and safety of the business logic.[9] For more on contracting see https://www.eiffel.org/doc/solutions/Design_by_Contract_and_Assertions.

8 Void safety and class NODE [G]

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). [10]

In class NODE [G] we have:

[9] For more on the importance of invariants for correctness see: https://bertrandmeyer.com/2018/05/24/not-program-right/

[10] https://en.wikipedia.org/wiki/Eiffel_(programming_language) - Void-safety.
This approach has now been adopted by other languages such as C# and Go.

```
item: G assign set_item
        -- returns the current item

left: detachable NODE [G] assign set_left
        -- reference to left child
```

A query (attribute or function routine) has a return type. The return type of attribute *item* is the generic parameter G, and the return type of attribute *left* is NODE [G].

In Void-safety, the compiler gives assurance, through a static analysis of the code, that at run time whenever a feature is applied to a reference, that the reference in question will have an object attached. This means that the feature call

```
x.f (a)
```

is valid (at compile time before the code is ever executed) only if we are assured that `x` will be attached to an object when the call executes.
The *Target rule* (validity code VUTA) is the primary compiler rule for void-safety (but there are others).

The *left* and *right* child of a node may be *Void*. We must thus declare *left* to be **detachable** meaning that it may initially or at other times be *Void* (not attached to a node object).

By default, all entities are **attached**. Thus, attribute *item* is (and must) always attached to an object of type G. For example, if G is type STRING, then we may always write *item.count* (without the worry that there will be a null pointer exception at runtime).

By contrast, before writing *left.item* we must first check that left is attached, e.g.

```
set_left (a_node: detachable NODE [G])
        -- updates left child to `a_node'
        -- updates left child's parent to `Current' if attached
    do
        left := a_node
        if attached left as l_left then
            l_left.parent := Current
        end
    ensure
        node_set: left = a_node
        child_has_correct_parent:
            attached left as l implies l.parent = Current
    end
```

The expression `attached left` is a Boolean assertion that is either *True* or *False*. The clause `as l_left` creates a local copy of *left* with type `attached NODE[G]` (the same return type as *left*, except attachment is guaranteed). We may thus safely write: `l_left.parent := Current`.

For more on Void safety, see
https://www.eiffel.org/doc/eiffel/Void-safe_programming_in_Eiffel.

Figure 3 below provides two *compile time* Void Safety errors.

8.1 VEVI compile time error:

By default, in the declaration

```
n: NODE [STRING]
```

entity `n` is **attached** and has type NODE [STRING]. The problem is that the creation routine *make* in the class does not initialize `n`. Thus, it is possible that *n.item*, for example, might generate a null exception at runtime.

This problem can be fixed by initializing `n` in the creation routine. Alternatively, `n` may be declared local in `some_test`.

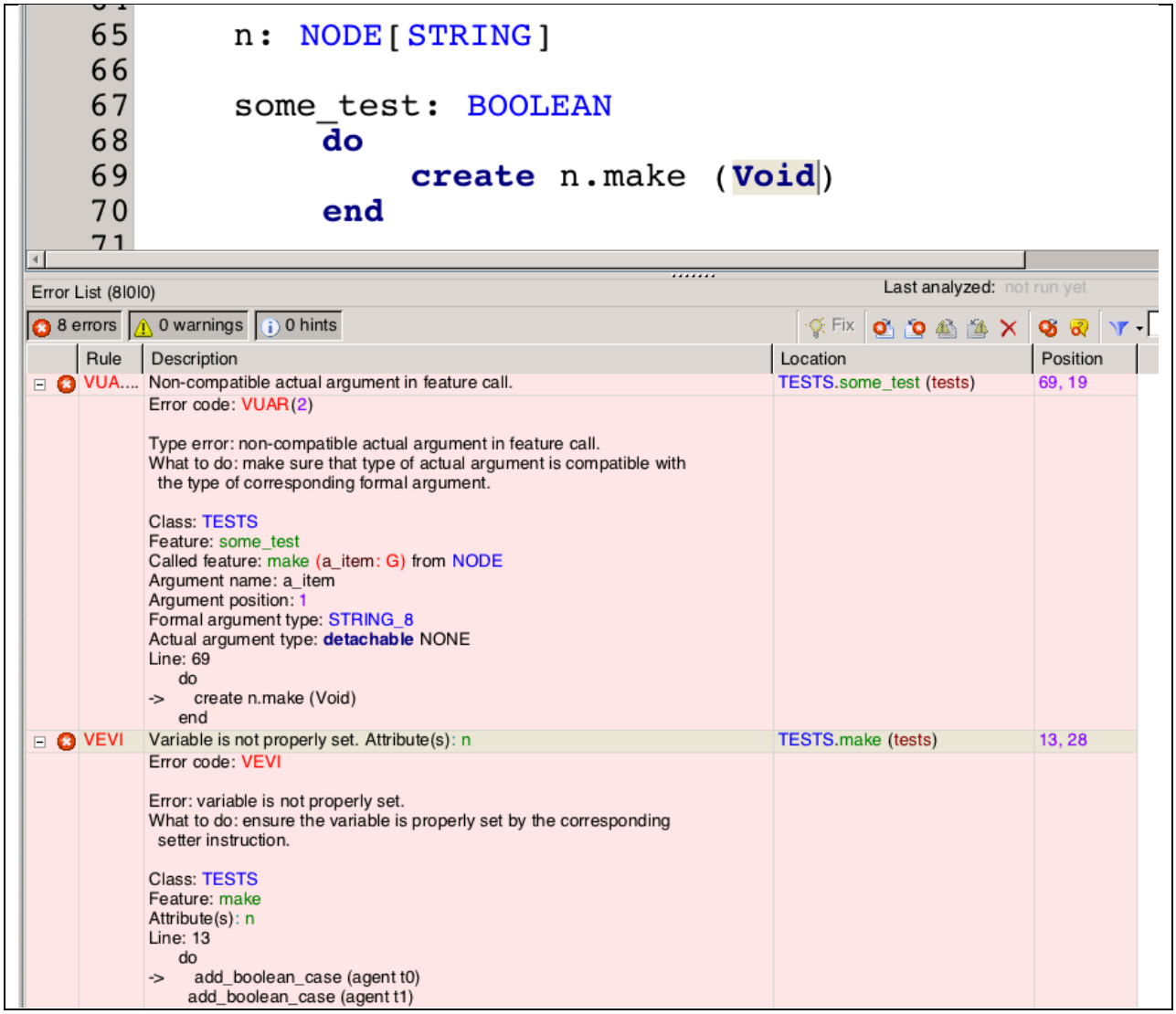


Figure 3 Void Safe Compile errors VUTA and VEVI

8.2 VUAR error

As before, by default, in the declaration `n: NODE [STRING]`, entity `n` is **attached** and has type NODE [STRING]. Now class STRING in Eiffel is mutable[11]. In addition, the declaration is not

`n: NODE [detachable STRING]`.

Thus, in `some_test`, create *n.make* (`Void`) is not allowed, as it might result in a null runtime exception.

[11] For immutable strings, we use class IMMUTABLE_STRING_8 or IMMUTABLE_STRING_32.

9 Unit Tests for class NODE [G]

Now write a test for class NODE setting the node item to “Noah”

```
t1: BOOLEAN
    local
        node, left, right: NODE[STRING]
    do
        comment("t1: create a node")
        create node.make ("Noah")
        Result := node.item ~ "Noah" and node.left = Void
        check Result end
    end
```

Test `t1` compiles and succeeds with a green bar. Try it.

However, extending test `t1` with a left child “Shem” results in a compile time error.

```
30
31      t1: BOOLEAN
32          local
33              node, left, right: NODE[STRING]
34          do
35              comment("t1: create a node")
36              create node.make ("Noah")
37              Result := node.item ~ "Noah" and node.left = Void
38              check Result end
39              create left.make ("Shem")
40              node.set_left (left)
41              Result := node.left.item ~ "Shem"
42          end
43
44  end
```

Error List (1|0|0)

1 error | 0 warnings | 0 hints

Rule	Description
VUT....	Target of the Object_call might be void.

Error code: VUTA(2)

Error: target of the Object_call might be void.
What to do: ensure target of the call is attached.

Class: TESTS
Feature: t1
Type: **detachable** NODE [STRING_8]
Line: 41

```
        node.set_left (left)
->      Result := node.left.item ~ "Shem"
    end
```

We may repair test `t1` with left child Shem as follows:

```
t1: BOOLEAN
    local
        node, left, right: NODE[STRING]
    do
        comment("t1: create a node")
        create node.make ("Noah")
        Result := node.item ~ "Noah" and node.left = Void
        check Result end
        create left.make ("Shem")
        node.set_left (left)
        check attached node.left as l_left then
            Result := l_left.item ~ "Shem"
        end

    end
```

Initially, *node.left* is Void. We use *node.set_left* (*left*) to create left child of the node *Shem*. This means that we are guaranteed that *node.left* is now attached to an object. Thus instead of using a conditional test for attachment, we may write a **check** assertion as follows:

```
check attached node.left as l_left then
    Result := l_left.item ~ "Shem"
    check Result end
end
```

Assertions (check, preconditions, postconditions, invariants etc.) can be turned off in finalized projects to be deployed in the field. The check for attachment (as above) cannot be turned off. This is because the software engineer has guaranteed that *node.left* is attached. If the guarantee is wrong the code will fail with a **check attached** exception. So, do not use **check attached** unless you have a proof that indeed the target is attached.

Extending test `t1`, with a right child *Japhet*, we obtain:

```
t1: BOOLEAN
    local
        node, left, right: NODE[STRING]
    do
        comment("t1: create a node and left and right children")
        create node.make ("Noah")
        Result := node.item ~ "Noah" and node.left = Void
        check Result end
        create left.make ("Shem")
        node.set_left (left)
        check attached node.left as l_left then
            Result := l_left.item ~ "Shem"
            check Result end
        end
        create right.make ("Japhet")
        node.right := right -- uses `set_right`
        check attached node.right as l_right then
            Result := l_right.item ~ "Japhet"
            check Result end
        end
    end
```

In the above case, we have used the assignment alias: `node.right := right`.

10 Object equality and is_equal

In Eiffel, as in Mathematics, the symbol "=" stands for equality, not assignment. For assignment, Eiffel uses the symbol ":=".

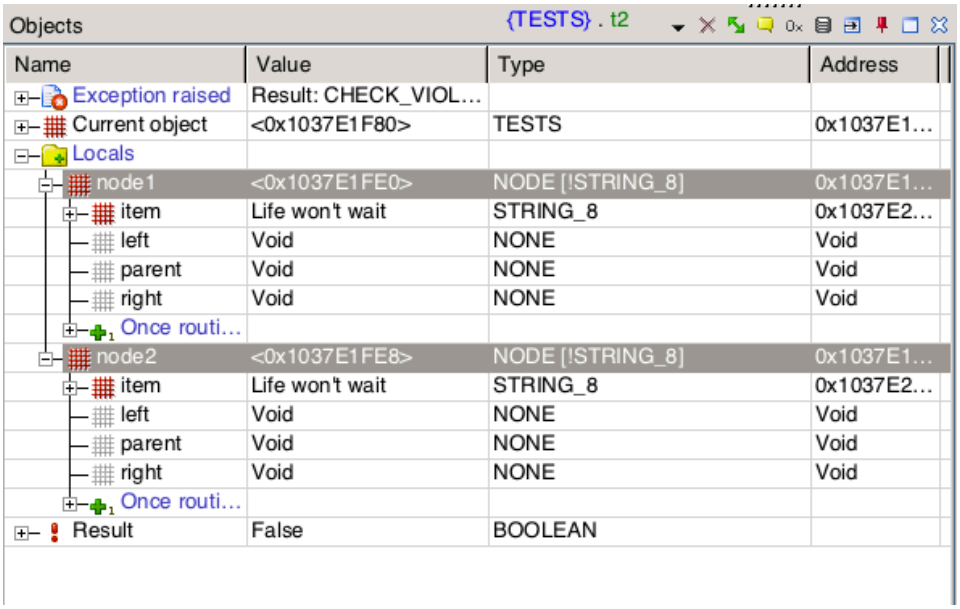
The following test fails because variables *node1* and *node2* do not satisfy reference equality, i.e. they do not refer to the same object:

```
t2: BOOLEAN
    local
        node1, node2: NODE[STRING]
    do
        comment("t2: test node equality")
        create node1.make ("Life won't wait")
        create node2.make ("Life won't wait")
        Result := node1 = node2
        check Result end
    end
```

Test `t2` thus fails.

FAILED (1 failed & 2 passed out of 3)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	2	3
All Cases	2	3
State	**Contract Violation**	**Test Name**
Test1	**TESTS**	
PASSED	NONE	t0:Test how array works
PASSED	NONE	t1: create a node and left and right children
FAILED	NONE	t2: test node equality

Class STRING is mutable in Eiffel. As can be seen in the debugger view (below), *node1* references one string object and *node2* a totally different string object; these objects have different locations in memory.



Both string objects have the same values (*Life won't wait*), but they are not the same object. We must thus use object comparison "~" as shown below:

```
t2: BOOLEAN
    local
        node1, node2: NODE[STRING]
    do
        comment("t2: test node equality")
        create node1.make ("Life won't wait")
        create node2.make ("Life won't wait")
        Result := node1 ~ node2
        check Result end
    end
```

PASSED (3 out of 3)

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

State	Contract Violation	Test Name
Test1	TESTS	
PASSED	NONE	t0:Test how array works
PASSED	NONE	t1: create a node and left and right children
PASSED	NONE	t2: test node equality

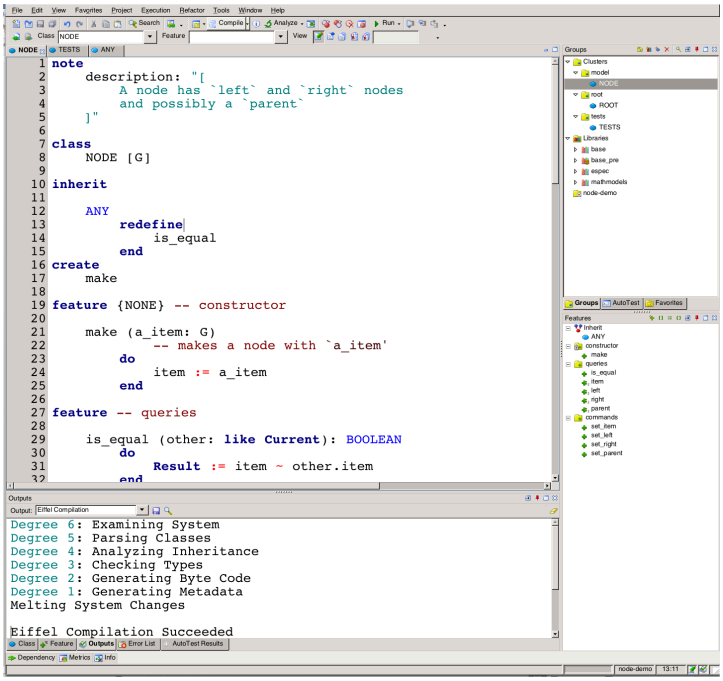
For the assertion `node1 ~ node2` to work, we must redefine *is_equal* in class NODE [G] as shown below:

```
class
    NODE [G]
inherit
    ANY
        redefine is_equal end

...

feature -- queries

    is_equal(other: like Current): BOOLEAN
        do
            Result := item ~ other.item
        end
```

The equality infix symbol "~" in the expression `node1 ~ node2` is aliased to {ANY} *equal* as shown below. Thus `node1 ~ node2` is the same as `equal(node1, node2)`:

```
class ANY …
   is_equal (other: like Current): BOOLEAN …

 frozen equal (a: detachable ANY; b: like a): BOOLEAN
          -- Are `a' and `b' either both void or attached
          -- to objects considered equal?
      do
          if a = Void then
               Result := b = Void
          else
               Result := b /= Void and then
                         a.is_equal (b)
          end
      ensure
          instance_free: class
          definition: Result = (a = Void and b = Void) or else
                       ((a /= Void and b /= Void) and then
                       a.is_equal (b))
      end
```

All classes inherit from class ANY. It is the root of the class hierarchy. Below we show a snapshot of the IDE. Ensure you know how to use it to browse the code, edit the code, compile, unit test, use the debugger, the documentation facility and the BON diagram utility.



11 Reference vs. Expanded Types

Suppose 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 reference semantics, it is possible that a variable (with its type declared detachable) may not refer to an object but be *Void.*[12]

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** type.

```
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.

For example, in the Mathmodels library, expanded class VALUE which does precise arithmetic needed in banking and other systems. Eiffel also provides an infix notation so that we can use the regular arithmetic operators such as +, -, * and /.

---

[12] *null* in other languages.

12 Using the debugger

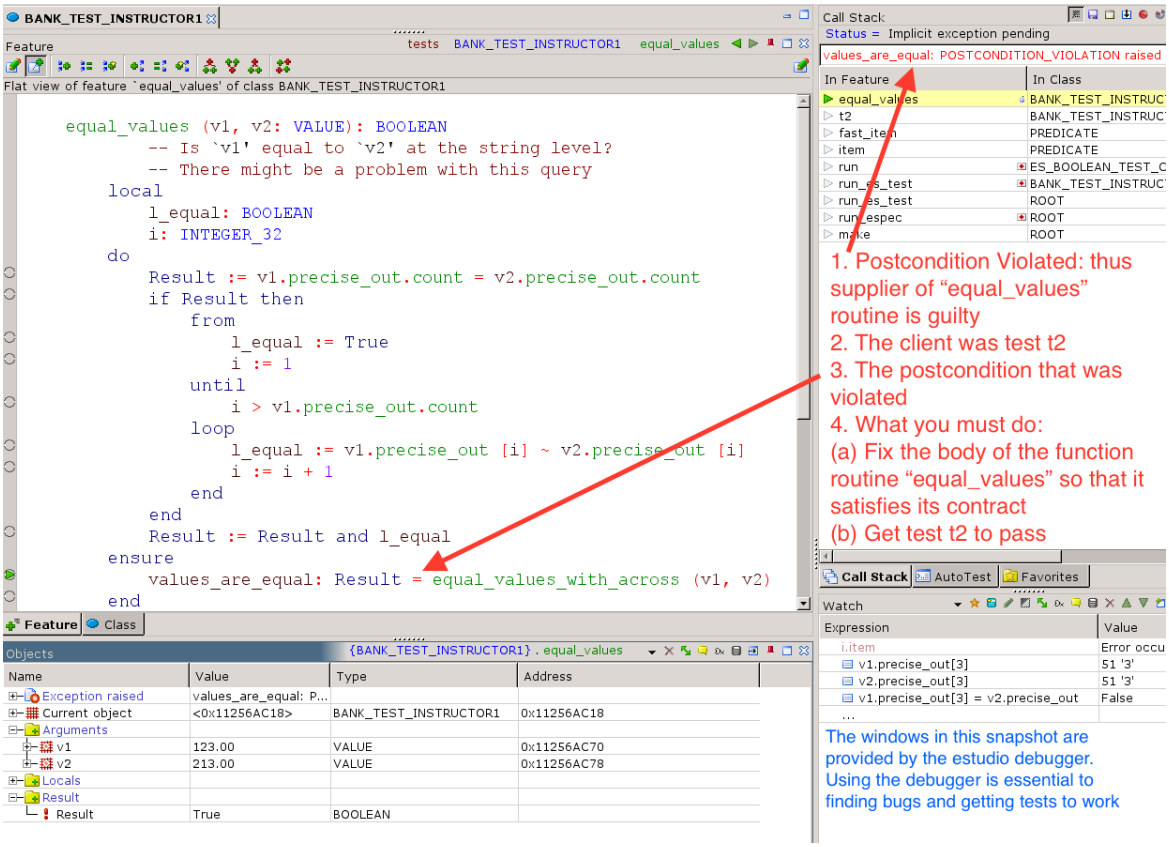
The first two tests *t1* and *t2* help you to understand how to use the expanded class VALUE.

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.



13 Iteration using the “across” notation in the debugger

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 https://www.eiffel.org/doc/eiffel/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

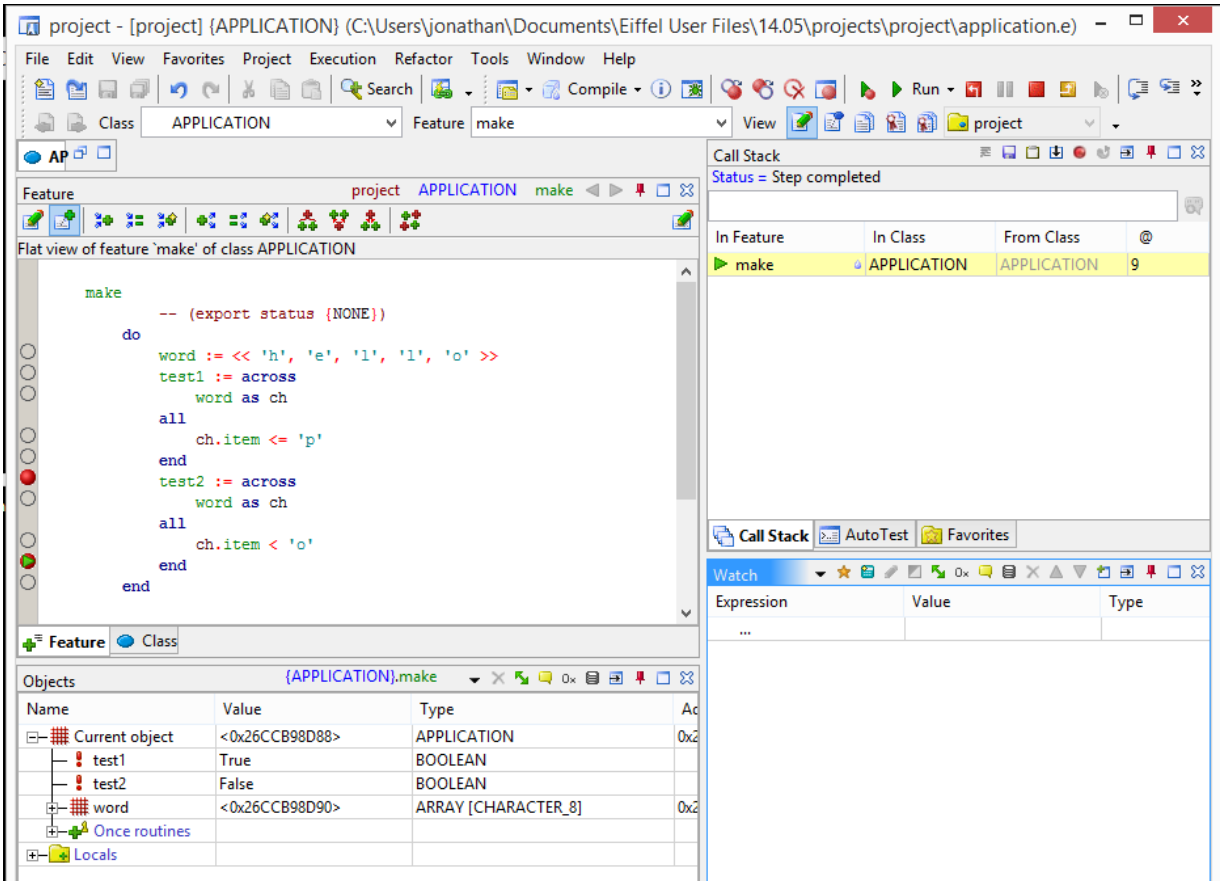
$$\forall \text{ ch} \in \text{word: ch} < \text{‘o’}$$

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.

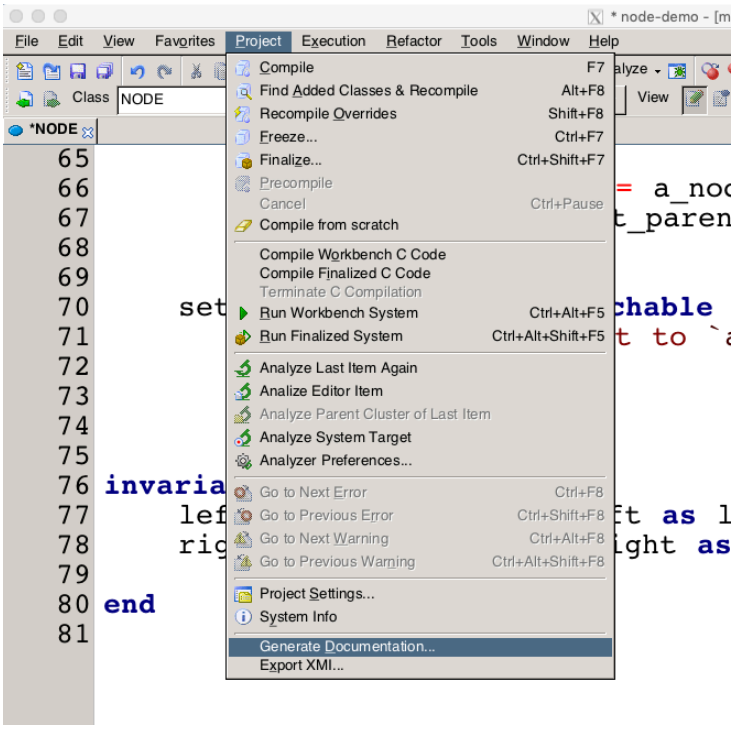
In the figure below, we have placed breakpoints shown with red dots and we execute the code, using the debugging facilities to get to the breakpoints. After the debugger reaches the second breakpoint, the debugger shows that test1 is *true* and test2 is *false*.



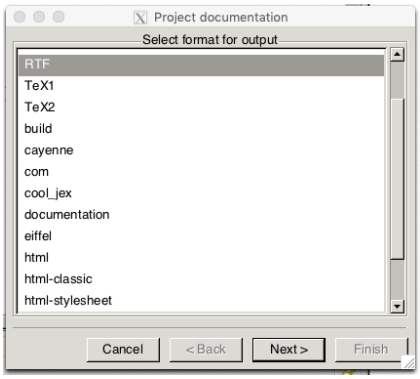
Make sure you know how to set breakpoints and how to execute to reach the breakpoints.

14 Using the IDE to generate Documentation

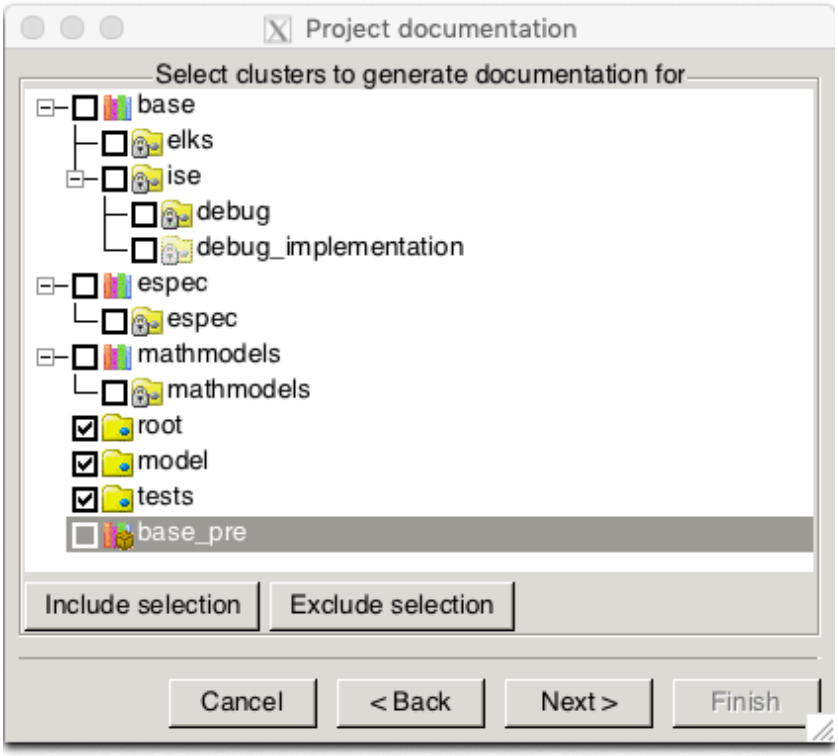
The IDE can be used to generate documentation in various formats as follows:



For the internet, *html-stylesheet* is a good choice. We choose *RTF* format, the one used in Figure 2.



In the next dialogue box, we select classes in clusters *root*, *model* and *tests*, and generate the documentation. By default, the documentation is in the EIFGENs directory.



15 Advice for writing comprehensive tests

John Guttag's introductory text on Python has some ideas are applicable to testing code written in any language, not just Python.[13] Some information has been added, and the discussion is adapted to Eiffel.

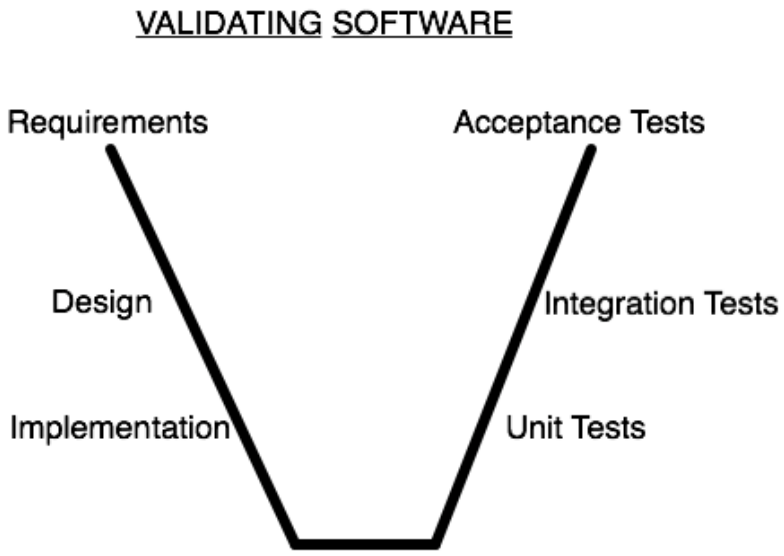
Our programs don't always function properly the first time we run them. Books have been written about how to deal with this last problem, and there is a lot to be learned from reading these books. However, in the interest of providing you with some hints that might help you get that next problem set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

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

(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.

15.1 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.

15.2 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[14]:

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 ≠ 0
- x ≠ 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.

15.3 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

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

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."

15.4 Using unit tests and the debugger

See https://www.eiffel.org/doc/eiffelstudio/Debugger

16 BON/UML class diagrams

A critical way to document a design (and the design decisions) is via a BON class diagram. Use the EiffelStudio IDE to generate BON (or UML) class diagrams. As an example, the IDE might genererate the following:

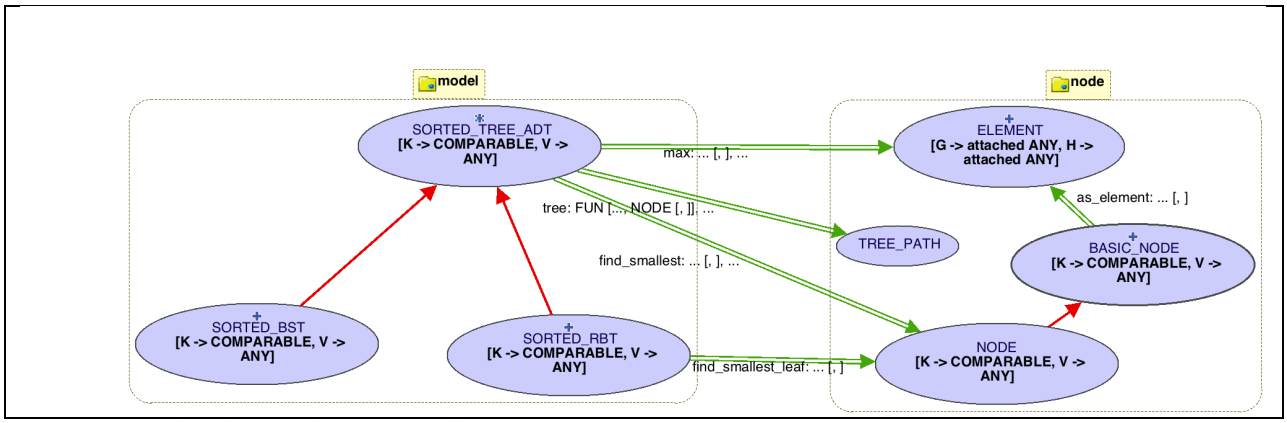


Figure 4 BON class diagram (IDE generated)

This diagram shows some important characteristics of the design:

- The diagram shows two clusters: *model* and *node*. Each cluster contains classes (shown as ellipses). The green double arrows denote *client-supplier* relationships and the red single arrow denotes an *inheritance* relationship between classes.
- The "*" decorator denotes *deferred* classes and the "+" decorator denotes *effective* classes. A deferred class has at least one *routine* (either a query or a command) that is deferred, i.e. has no implementation. Such a class cannot be instantiated at runtime, and thus does not have explicit constructors.
- Classes are always written using UPPER_CASE. *Features* (queries and commands) are written using lower case.
- Deferred class SORTED_TREE_ADT [K, V] has two *generic* parameters, K for keys and V for values. Generic parameter K is *constrained* to be COMPARABLE, needed for a sorted order.
- Most of the features of deferred class SORTED_TREE_ADT [K, V] are effected (implemented) using class NODE. Some examples are shown below:

The IDE Drawing tool is a good starting point for the BON class diagram. But we obtain a better view of the design using the draw.io tool.[15]

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

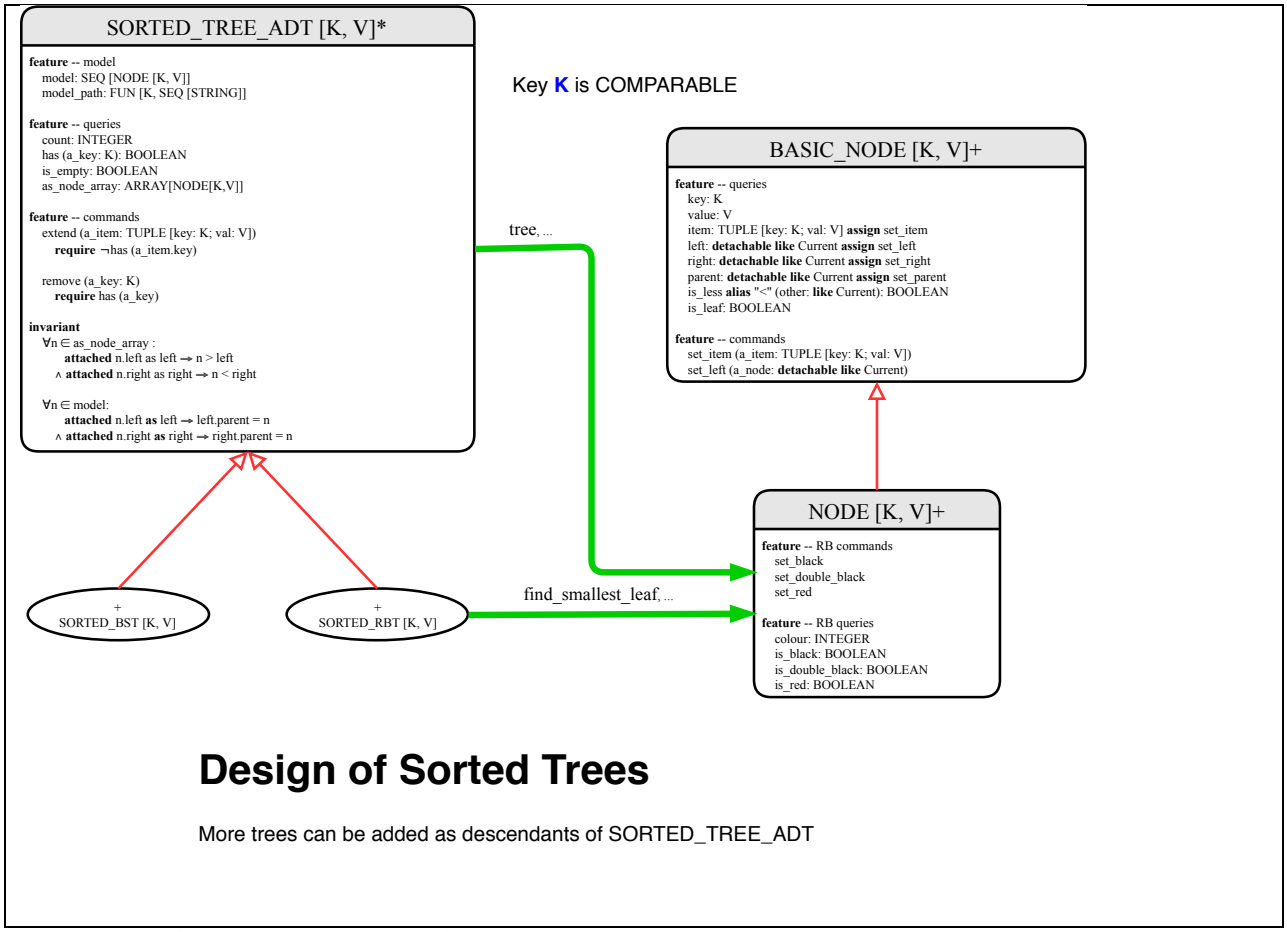


Figure 5 BON class diagram (draw.io template)

The *draw.io* diagram is constructed manually, which allows for the selective provision of classes, their relationships, their features, their signatures, their contracts and class invariants. The design architecture is thereby better described. See the footnote for more information.[16] Please familiarize yourself with these notations.

16.1 Architecture: design structure

BON (or UML) class diagrams are important for documenting the design of the system architecture, i.e. how the different parts (or modules) of the system *structure* are related to each other. A software system's structure is a division of that system into a set of parts and the relations between those parts.

[16] You might try to produce the BON diagram and 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.

> If all software experts agree on anything, it is that software shouldn't be a monolith (a large system that is, for all practical purposes, indivisible). In the half century since Edsger Dijkstra published his groundbreaking paper, "The Structure of the 'THE'-Multiprogramming System," it has become clear that the ability to design a software system's structure is at least as important as the ability to design efficient algorithms or to write code in a particular programming language. …
>
> Dijkstra's papers and presentations made it clear that he and his team designed the structures before any code was written. The structure guided the coders; the result was a design that was "cleaner" than other systems of that time.
>
> Dijkstra's team was small, highly motivated, and very talented. Dijkstra, who described himself as the team's captain, was very hands-on. They didn't create precise documentation of the structure. With a larger team, one that was managed rather than "captained," the lack of documentation would have led to miscommunication. That would have lengthened the development time and might have introduced errors.
>
> The lack of documentation became evident after Dijkstra's team dispersed. The system was their legacy and was used for some time after they left. One member of the original team frequently received phone requests for help when a problem occurred. In that team member's words, "The structure was clean and simple, but it existed only in our minds." The lesson is clear: structure is vital, but unless you plan to discard the software when its authors move on, it must be documented.[17]

17 Abstraction, DbC and Information Hiding

There is also *abstraction by specification* where we ignore implementation details, and agree to treat as acceptable any implementation that adheres to the specification.

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.

17.1 Contract View

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

- We use contracts to specify 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.

[17] "Software Structures: A Careful Look", David Lorne Parnas, *IEEE Software*, Nov-Dec 2018.

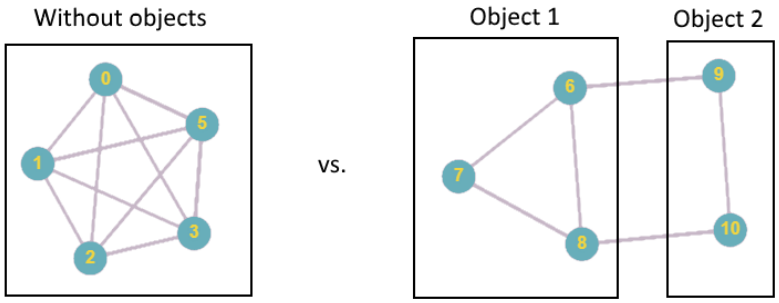
- Note that the class is self-documenting. There is an indexing clause to explain the purpose of the class and each feature has a meaningful comment.

```
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
```

A comment on Quora is interesting[18]. How do we explain to a beginner why we use object oriented programming?

[18] https://www.quora.com/Can-you-explain-to-a-beginner-why-we-use-OOP. Of course there are good reasons why programmers adopt alternative styles such as functional programming.

It is difficult to understand why we use OOP until we write large programs that don't fit in our brains at one time. Beginners don't normally write such large programs. Here is an image that may one day make sense.



Each node represents a module (e.g. a class, cluster or other parts of the code) of your program, and each connection represents an interaction. Both programs have the same number of modules, but in the second code is hidden and encapsulated so that a module only has limited interaction with the outside world (through its interface or API). For example, Object 2 is not even aware that module 7 exists; it's a private feature of Object 1.
The non-OO program has 10 possible interactions. The OO program has only has 6 interactions.

The formula for number of interactions in the non-OOP case is (n*(n-1))/2, so it grows *very* fast. As more modules are added it's easy to lose control over the complexity of your program without modules that have good interfaces. Each module should hide implementation or design decisions from users of the module. This is called information hiding.

This is assuming that we have already broken the program into modules (separation of concerns). Without modularity, the complexity is exacerbated.

18 Mathmodels

The Mathmodels library is used to write high level specifications beyond classical contracts. The library contains mathematical classes such as:

- SEQ [G]
- SET [G]
- PAIR [G, H]
- FUN [G, H]
- REL [G, H]
- VALUE

As an example, consider a chart view of some of the features of class SEQ[G].

```
class
        SEQ [G -> attached ANY]

General
        cluster: mathmodels
        description:
                "Finite sequences of some elements of type G.
                A valid index is 1..count.
                Array notation can be used, as well as iteration (across).
                Empty sequences can be created, or creation can be from an array.
                Sequences have a first item (the head), a tail and last item.
                Infix notation for prepended_by where x is of generic type G:
                        seq1 |< x
                Infix notation for appended_by where x is of generic type G:
                        seq1 |> x
                For concatenation we use infix: seq1 |++| seq2
                For queries, to assert that the state is not changed,
                the postcondition is
                        Current ~ old Current.deep_twin
                Class also has an inefficient implementation:"
        create: make_empty, make_from_array

Ancestors
        ITERABLE* [G]

Queries
        appended alias "|->" (v: G): SEQ [G]
        as_array: ARRAY [G]
        as_function: FUN [INTEGER, G]
        concatenated alias "|++|" (other: SEQ [G]): SEQ [G]
        count alias "#": INTEGER
        debug_output: STRING_8
        first: G
        front: SEQ [G]
        has (v: G): BOOLEAN
        hold_count (exp: PREDICATE [PAIR [INTEGER, G]]): INTEGER
        inserted (v: G; i: INTEGER): SEQ [G]
        is_empty: BOOLEAN
        is_subsequence_of alias "|<:" (other: SEQ [G]): BOOLEAN
        item alias "[]" (i: INTEGER): G
        last: G
        overriden (v: G; i: INTEGER): SEQ [G]
        prepended alias "|<-" (v: G): SEQ [G]
        removed (i: INTEGER): SEQ [G]
        reversed: SEQ [G]
        slice (a_start, a_end, a_step: INTEGER): SEQ [G]
        subsequenced (i, j: INTEGER): SEQ [G]
        tail: SEQ [G]
        twin2: SEQ [G]
        upper: INTEGER
        valid_position (pos: INTEGER): BOOLEAN

Commands
        append (v: G)
        concatenate (other: SEQ [G])
        insert (v: G; i: INTEGER)
        make_empty
        make_from_array (a: ARRAY [G])
        override (v: G; i: INTEGER)
        prepend (v: G)
        remove (i: INTEGER)
        reverse
        subsequence (i, j: INTEGER)
```

Mathmodels extends the classical Eiffel contracting notation with the use of mathematical models (based on sets, sequences, relations, functions, bags). The Mathmodels library has immutable queries (for specifications) as well as relatively efficient mutable commands.

For example, class SEQ[G] has an immutable query

```
appended alias "|->" (v: G): SEQ [G]
```

and a corresponding mutable command

```
append (v: G)
```

The query takes an argument `v`, and returns a brand-new sequence which is the same as `Current`, except that `v` is appended to `Current`.

For more on Mathmodels, see

https://www.eecs.yorku.ca/~jonathan/publications/2018/MoDRE18.pdf

You can explore FUN [G, H] at

https://www.eecs.yorku.ca/course_archive/2016-17/W/3311/eiffel-docs/mathmodels/fun_chart.html

Exercise: Write a test to use FUN [G, H]. For example, each PERSON has a unique ID so you may want to write FUN [PERSON, ID].

18.1 Specifying stacks with Mathmodels

You are familiar with stacks from your basic computing courses. You have studied the concept of an Abstract Data Type (ADT). If you are not sure what that is, read chapter 6 of OOSC2. Here is the stack ADT:

ADT specification of stacks

TYPES

- *STACK* [*G*]

FUNCTIONS

- *put*: *STACK* [*G*] × *G* → *STACK* [*G*]
- *remove*: *STACK* [*G*] ↛ *STACK* [*G*]
- *item*: *STACK* [*G*] ↛ *G*
- *empty*: *STACK* [*G*] → *BOOLEAN*
- *new*: *STACK* [*G*]

AXIOMS

For any *x*: *G*, *s*: *STACK* [*G*]

A1 • *item* (*put* (*s*, *x*)) = *x*

A2 • *remove* (*put* (*s*, *x*)) = *s*

A3 • *empty* (*new*)

A4 • **not** *empty* (*put* (*s*, *x*))

PRECONDITIONS

- *remove* (*s*: *STACK* [*G*]) **require not** *empty* (*s*)
- *item* (*s*: *STACK* [*G*]) **require not** *empty* (*s*)

Note that an ADT is more general than a Java Interface (which provides only the signatures of the attributes and methods). An ADT is more than just the signatures. It also specifies the preconditions and axioms of the ADT. Using the stack ADT we may write an expression such as

$$item\ (remove\ (put\ (remove\ (put\ (put\ (\\ remove\ (put\ (put\ (put\ (new, x1), x2), x3)),\\ item\ (remove\ (put\ (put\ (new, x4), x5)))), x6)), x7)))$$

and, from the axioms, we can prove that the expression is equal to x4.

WE will now use Mathmodels to provide an alternative complete specification of the stack ADT.

```
class
    STACK_ADT [G -> attached ANY]

General
    cluster: model
    description: "Abstract Data Type Signatures for a stack"
    create: make_empty

Queries
    count: INTEGER
    empty: BOOLEAN
    item: G
    model: SEQ [G]

Commands
    put (x: G)
    remove
```

Figure 6 Signatures for a generic stack (Chart View Generated by the IDE)

Figure 6 is a *chart* view of the stack ADT. In Figure 7 we provide a *contract* view of the stack ADT.

Without Mathmodels, we can document the preconditions of the ADT (see classical contracts). But we can document all the postconditions.

With Mathmodels, we can provide complete specifications as shown in the contract view (Figure 7). In fact, the classical contracts are no longer required. The complete contracts can be provided with reference only to the *model* which is:

```
    model: SEQ [G]
```

We could make class STACK_ADT a **deferred** class. Then all implementations might be a subclass of this ADT. For example, we might implement stacks with arrays, or alternatively with linked lists, etc. The subclasses inherit all the model contracts and must thus satisfy these model

contracts. Any descendant which does not completely implement the model specifications, will generate a contract violation. Thus all subclasses will completely implement the ADT.

In Figure 8, we choose to make the STACK_ADT an effective class. We implement this class with the commands of SEQ[G]. Thus, it is not necessarily so efficient, but we may already write tests to check the correctness of the specifications, and we may use the same tests to check the correctness of the subclass implementations.

In Figure 9, we show a test for the STACK_ADT and a breakpoint in the debugger to show what the model looks like.

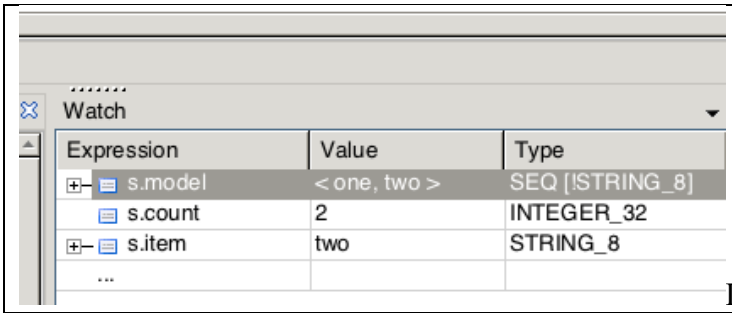
Exercises: Encode the STACK_ADT using the IDE. Write more unit tests and check that they pass. Design efficient stacks by sub-classing the ADT and using (a) arrays and (b) linked lists. Use the debugger to check the status of the model and to debug any problems.

```
class interface STACK_ADT [G -> attached ANY] create
    make_empty

feature -- model
    model: SEQ [G]
            -- abstraction function
            -- using SEQ queries

feature -- queries
    count: INTEGER
            -- number of items in stack
        ensure
            model_contract: Result = model.count

    item: G
            -- top element
        require
            classical_contract: not empty
            model_contract: not model.is_empty
        ensure
            model_contract: Result ~ model.last

    empty: BOOLEAN
            -- is the queue empty?
        ensure
            model_contract: Result = model.is_empty

feature -- commands
    put (x: G)
            -- push 'x' on top of stack ("push")
        ensure
            classical_contract1: count = old count + 1
            classical_contract2: item ~ x
            model_contract: model ~ (old model.deep_twin) |-> x

    remove
            -- pop top of stack, i.e. item
        require
            classical_contract: not empty
            model_contract: not model.is_empty
        ensure
            model_contract: model ~ (old model.deep_twin).front

end
```

Figure 7 Contract View for a generic stack using Mathmodels SEQ[G]

```
class STACK_ADT [G -> attached ANY] create
	make_empty

feature {NONE} -- Constructor
	make_empty
		do
			create model_imp.make_empty
		end

	model_imp: like model
			-- implementation using SEQ commands

feature -- model
	model: SEQ [G]
			-- abstraction function, using SEQ queries
		do
			Result := model_imp
		end

feature -- queries
	count: INTEGER
			-- number of items in stack;
		do
			Result := model_imp.count
		ensure
			model_contract: Result = model.count
		end

	item: G
			-- top element
		require
			classical_contract: not empty
			model_contract: not model.is_empty
		do
			Result := model_imp.last
		ensure
			model_contract: Result ~ model.last
		end

	empty: BOOLEAN
			-- is the queue empty?
		do
			Result := model_imp.is_empty
		ensure
			model_contract: Result = model.is_empty
		end

feature -- commands

	put (x: G)
			-- push 'x' on top of stack ("push")
		do
			model_imp.append (x)
		ensure
			classical_contract1: count = old count + 1
			classical_contract2: item ~ x
			model_contract: model ~ (old model.deep_twin) |-> x
		end

	remove
			-- pop top of stack, i.e. item
		require
			classical_contract: not empty
			model_contract: not model.is_empty
		do
			model_imp.remove (count)
		ensure
			model_contract: model ~ (old model.deep_twin).front
		end

end
```

Figure 8 Implementation of STACK_ADT[G] with commands from Mathmodels SEQ[G]



Debugger breakpoint is set in test `t1`

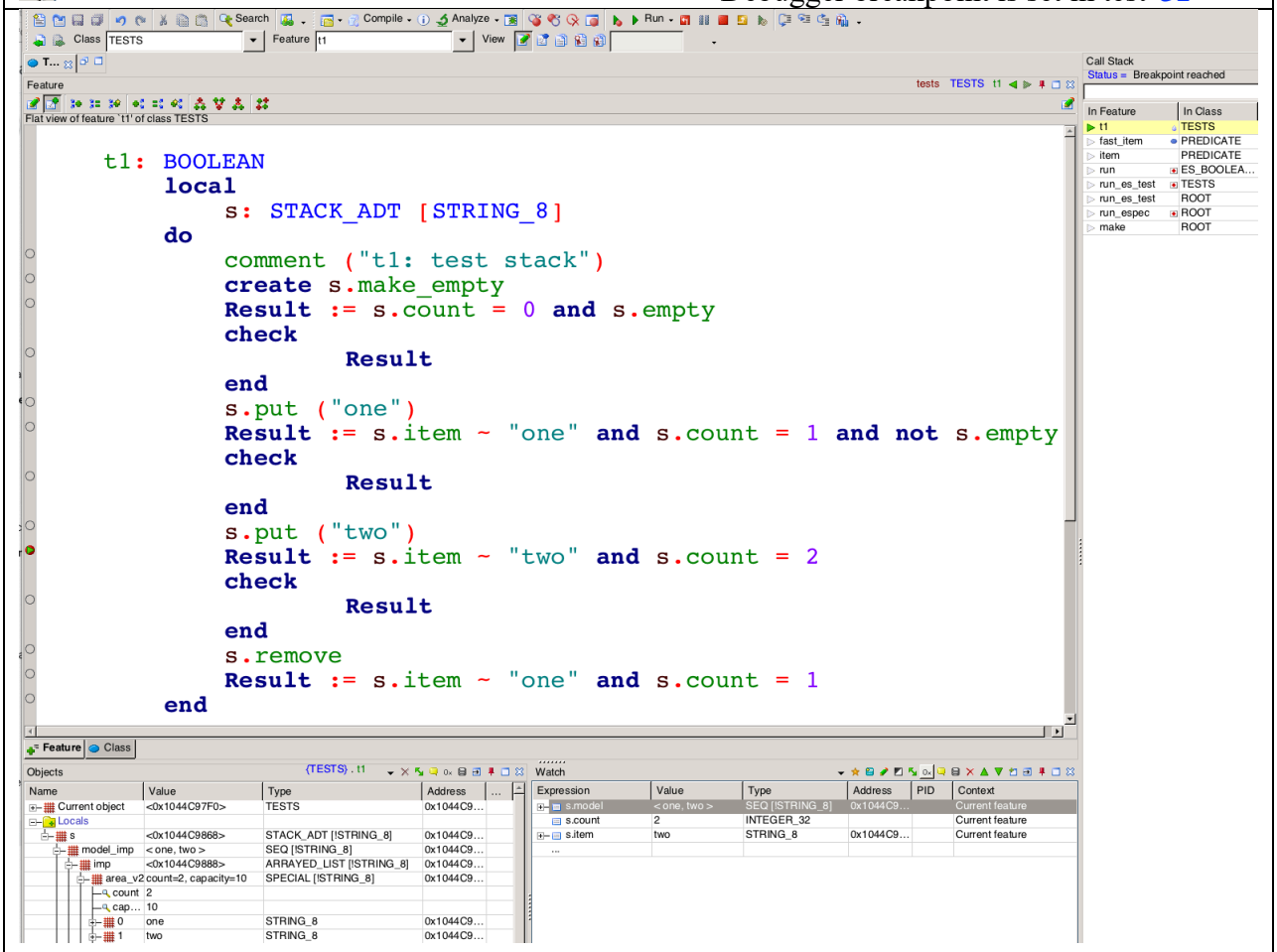


Figure 9 Debugger view of a test of class STACK_ADT