EECS3311 Fall 2019
Lab Exercise 0
Practicing DbC (Design-by-Contract) and TDD (Test-Driven Development) in Eiffel Studio

Chen-Wei Wang

Abstract

There is no submission required for this Lab 0, but you should complete it as soon as possible, no later than the scheduled date of Lab 1 (Wednesday, September 11). All your lab submissions must be compilable on the department machines. It is then crucial that should you choose to work on your own machine, you are responsible for testing your Eiffel project before submitting it for grading. It is highly recommended that you complete this lab exercise using your EECS accounts (e.g., from machines located in LAS1006), so as to prepare yourself for the lab tests which will take place in the same working environment.

This lab is intended to help you get familiar with the programming/design environment of this course. We will learn about the most common features of Eiffel Studio (an IDE for the object-oriented Eiffel programming language, analogous to Eclipse for Java). Screen shots of this lab are taken from a Mac OS X environment, but the look and feel of Eiffel Studio on your lab accounts should be fairly similar.

Contents

1 Resources

Take some time exploring this wiki site here:

`http://seldoc.eecs.yorku.ca/doku.php/eiffel/start`

Feel free to reuse the code samples.

2 Important Tutorials for Introducing You to Eiffel Basics

Once you complete the simple exercises for Lab 0, you will be ready to finish studying this tutorial series on DbC and TDD:

`https://www.youtube.com/playlist?list=PL5dxAmCmjv_6r5VfzCQ5bTznoDDgh__KS`

3 Acronyms

- **EStudio** [Eiffel Studio]
- **DbC** [Design by Contract]
- **TDD** [Test-Driven Development]
- **eclean** [Prism-only Command for Cleaning Eiffel Projects]

4 Create a Workspace for Your Eiffel Projects

- Launch a terminal (right click on your desktop, then there should be an option for that).
- Type the following command to: **1)** change the current director to your desktop; and **2)** create an empty workspace for your all Eiffel projects (for labs, the end-of-semester project, and **more importantly, your own exercises**).

```
cd ~/Desktop
mkdir eecs3311_workspace
```

  where the symbol ~ is a shorthand for the path of your home directory (e.g., `/eecs/home/jackie`). Now, you have on your desktop an empty directory `eecs3311_workspace`.

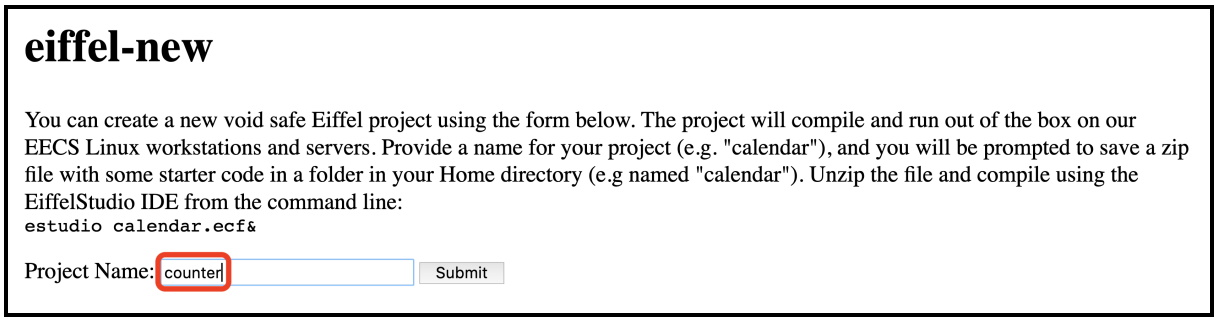
Advice. The best way to learning a new programming language (not just Eiffel!) is by trying out as many examples as you can. Whenever you find an idea or concept taught in class being puzzling or fascinating, do not hesitate to create your own Eiffel projects to do your own experiments!

5 Creating a Project

– Using your Passport York (not EECS) credentials, access this website:

`https://www.eecs.yorku.ca/~eiffel/eiffel-new/`

– In the `Project Name` textfield, enter the Eiffel project name for this Lab 0: `counter` (and later, any name of project which you wish to try out!).



Click on the `Submit` button, then choose to save the project arhicve file (i.e., `counter.zip`) into the workspace you just created: `~/Desktop/eecs3311_workspace`.

– Now on your terminal, go to the workspace and uncompress the project archive file:

```
cd ~/Desktop/eecs3311_workspace
unzip counter.zip
```

– Verify that the starter project for Lab 0 has now been in place for compilation, by typing the command:

```
tree counter
```

Then you should see the following project structure:

```
counter
├── counter.ecf
├── model
├── root
|   └── root.e
└── tests
    └── test_example.e

3 directories, 3 files
```

Now, proceed to Section 5.1 if you are working on your own machine (in which case you need to install and set up the MATHMODELS library). Otherwise, if you are working on a Prism lab computer, the MATHMODELS library has been installed and setup properly, so you can skip to Section 6

5.1 Download the MATHMODELS library (if you work on your own machine)

Skip this section if you are working on a Prism lab machine! If you are setting up your own machine, download from the following link:

`http://www.eecs.yorku.ca/~eiffel/zip/mathmodels.zip`

Assuming that you already stored the archive file `mathmodels.zip` on your desktop (of if other directory, then just set up accordingly), type the following command:

```
cd ~/Desktop
unzip mathmodels.zip
```

Now you need to set an *environment variable* pointing to the where the uncompressed `mathmodels` directory. If you work on a Windows machine, you can easily find out from the web how this can be done, e.g., click on **this link**. Otherwise, if you work on a mac (and similarly for Linux), then open the `.bash_profile` file:

```
nano ~/.bash_profile
```

Then, write the following line to the `.bash_profile` file (replace `jackie` by the user account name of your mac):

```
export MATHMODELS="/Users/jackie/Documents/svn/sel-open/mathmodels"
```

Type `Ctrl + x` to exit from the `nano` editor, and be sure to save the `.bash_profile` file.

Finally, type the following command to make the new environment variable effective:

```
source ~/.bash_profile
echo $MATHMODELS
```

Verify the output path denotes the exact location of the `mathmodels` directory.

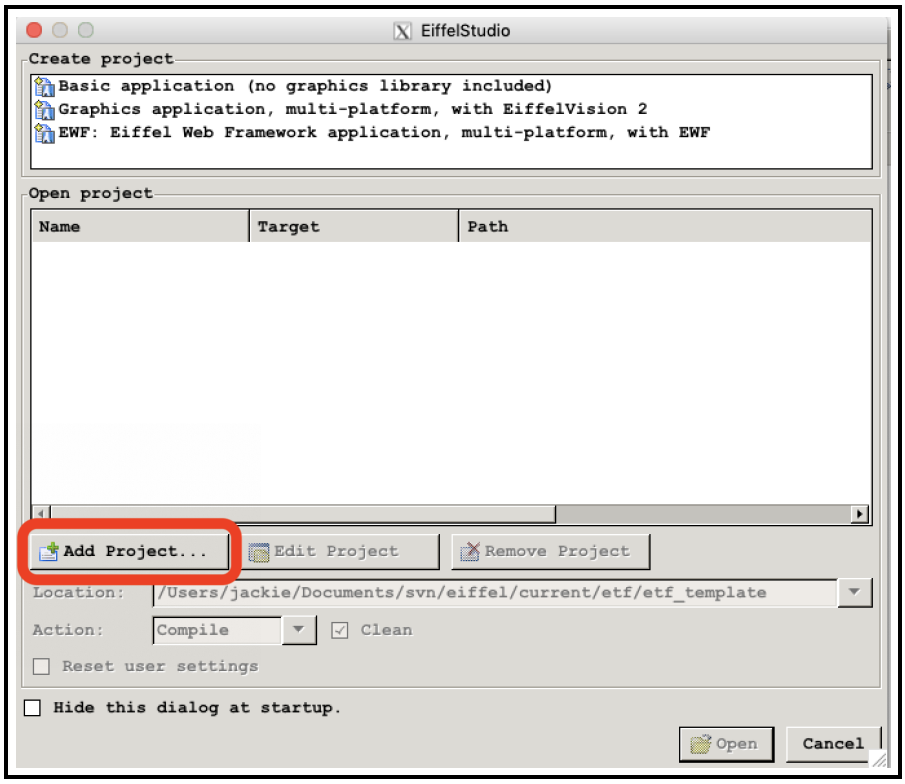
6 Compiling the Starter Project in Eiffel Studio

– On your terminal, type the following command to launch the latest version of Eiffel Studio (19.05):

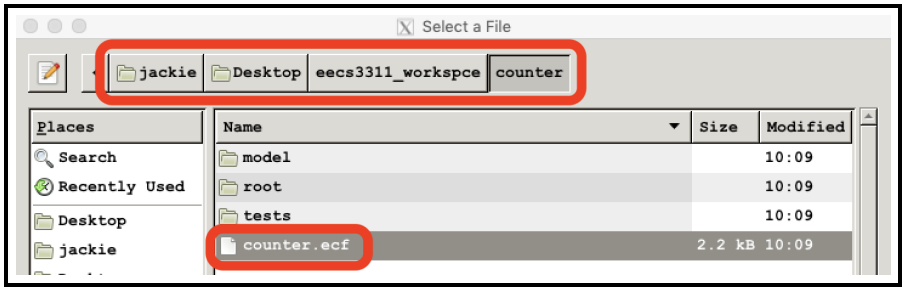
```
estudio19.05 &
```

where the `&` means, as you learned from your EECS2031, that the process will be executed at the background of the current terminal, and thus you do not need to open another terminal to execute other commands if needed.

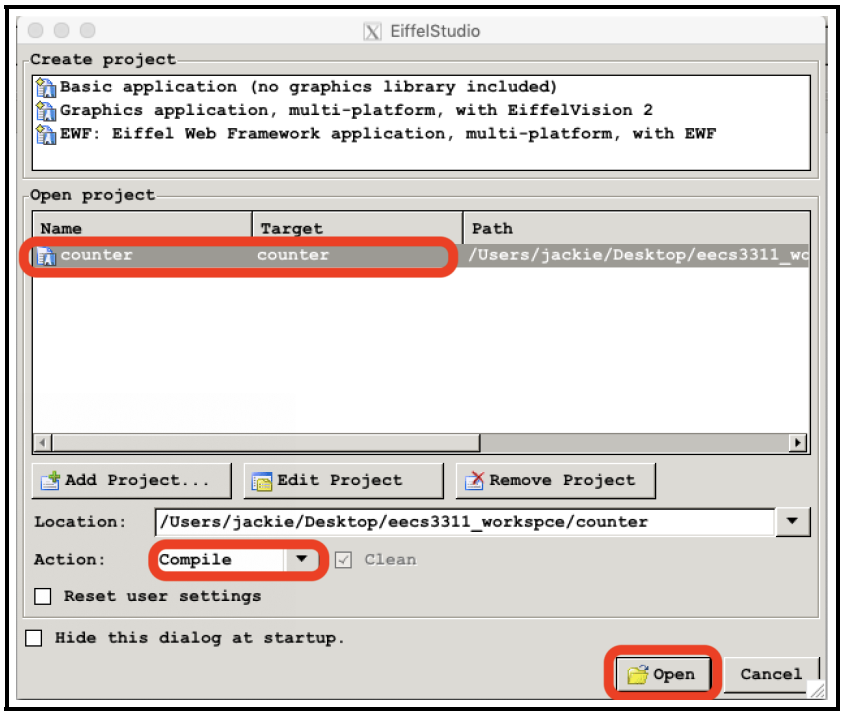
– Right after EStudio is launched, a window will pop up. Click on the `Add Project` button:



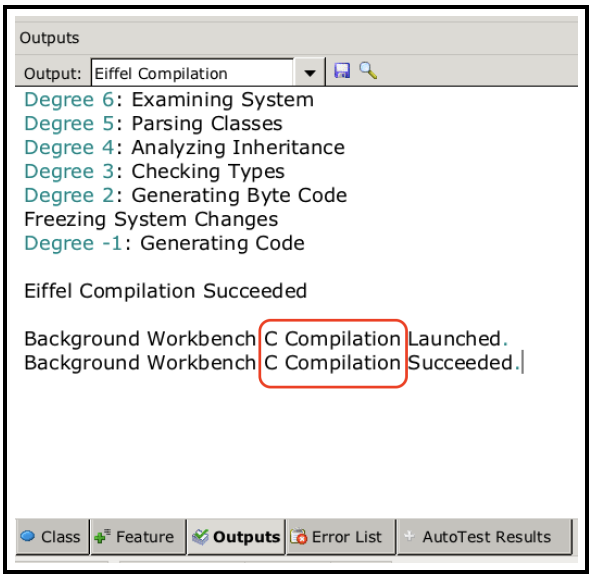
– Choose the ecf (Eiffel Configuration) file for the `counter` project, which should be stored in here: `~/Desktop/eecs3311_workspace/counter/counter.ecf`.



– Now you should see the `counter` project being remembered in the EStudio startup window. Click on `Open` to start compiling the project. Wait until you see from the `Outputs` panel that the compilation is successfully completed.



– A success compilation should look like this:

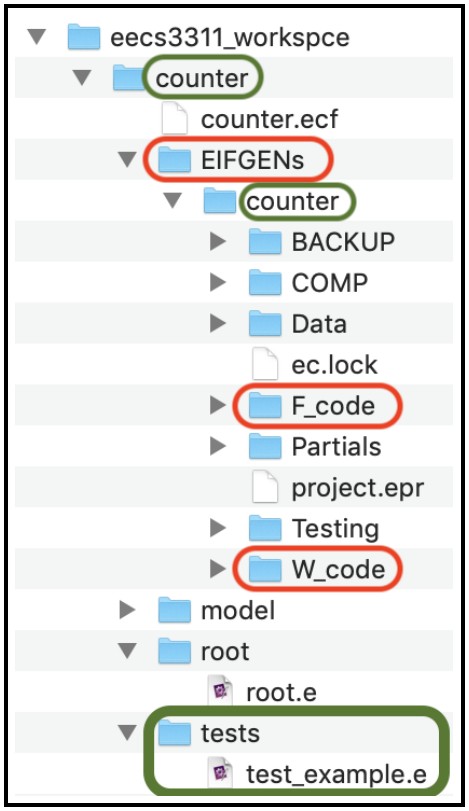


Remark: The output message above includes phrases of `C Compilation`. Keep in mind that code written in Eiffel is not directly executable; instead, they are compiled and, for efficiency reason, optimized into C code. All C code is stored in a subdirectory of your project named `EIFGENs` (standing for Eiffel Generations). We will explore this `EIFGENs` directory further.

This compilation makes sense: Eiffel is meant to be a **design language** for you to **think at a much higher level of abstraction**, whereas C is meant to be an **implementation language** for you to tweak the performance of your code, e.g., pointer arithmetic, dynamic memory (de)allocation.

6.1 Exploring Project Structure from the File System

– Using the GUI-based file explorer, understand that the `counter` project, automatically generated and successfully compiled, has the following directory structure in the file system:



– Alternatively, using the terminal, reaffirm yourself about the directory structure:

```
jackie:~$ cd ~/Desktop/eecs3311_workspce/
jackie:eecs3311_workspce$ ls
counter     counter.zip
jackie:eecs3311_workspce$ ls counter
EIFGENs     counter.ecf model       root        tests
jackie:eecs3311_workspce$ ls counter/EIFGENs/
counter
jackie:eecs3311_workspce$ ls counter/EIFGENs/counter
BACKUP      Data        Partials    W_code      project.epr
COMP        F_code      Testing     ec.lock
jackie:eecs3311_workspce$ ls counter/root/
root.e
jackie:eecs3311_workspce$ ls counter/tests/
test_example.e
```

6.2 Understanding the Critical Directories

– The directory `root` typically stores the `ROOT` class, which serves as the entry point of execution. This is analogous to the case in Eclipse: a Java class (e.g., `MyApplication`) with the `main` method can be run as a Java application. Just as in Eclipse you can have multiple Java classes with the `main` method and choose which one to run as a Java application, in EStudio you can set a class as the *root* class of the project. *See Section 9.2 for more details on setting the root class of your project.*

– The directory `tests` directory is meant to store all classes related to testing the correctness of the counter. *See Section 2 for the link to a tutorial series, which discusses how to write unit-tests in Eiffel.*

– The directory `EIFGENs` (standing for Eiffel Generations) stores all the C code this is compiled from the source Eiffel code in the current project. Since we chose the project name as `counter`, this results in the fact that there is a subdirectory named `counter` under the `EIFGENs` directory. There are two subdirectories under `EIFGENs/counter` that you should know about:

- **F_code**: This directory stores the finalized (i.e., optimized and ready for delivery/submission) executable of your project. When your new project is first created and compiled, the **default** option is that the project is **not** finalized, meaning that it is still subject to a number of intermediate revisions/recompilations. To see this, on your terminal type the following command:

```
ls ~/Desktop/eecs3311_workspace/counter/EIFGENs/counter/F_code
```

  You should see an empty output. Why? Because nothing has been finalized yet!

- **W_code**: As said, when your project is first created and compiled, it is still subject to a number of recompilations until you are satisfied so that you can finalize your project. Currently there is an intermediate (i.e., not optimized) executable that exists in this `W_code` directory. To see this, type the following command:

```
ls ~/Desktop/eecs3311_workspace/counter/EIFGENs/counter/W_code
```

  The output should be:

```
C1  Makefile     TRANSLAT   counter
E1  Makefile.SH  config.sh  counter.melted
```

  where the file `counter` is in fact an *executable*. Try it by typing:

```
~/Desktop/eecs3311_workspace/counter/EIFGENs/counter/W_code/counter
```

  You should see two outputs: one on the terminal and the other being a popped up HTML page. The terminal output looks like:

The above console output says that the `TEST_EXAMPLE` class (in the tests cluster) contains 4 tests, out of which 3 passed (i.e., `t1`, `t2`, and `t3`) and 1 failed (i.e., `t0`).

The equivalent, but more user-friendly HTML output looks like:

ROOT

Note: * indicates a violation test case

FAILED (1 failed & 3 passed out of 4)

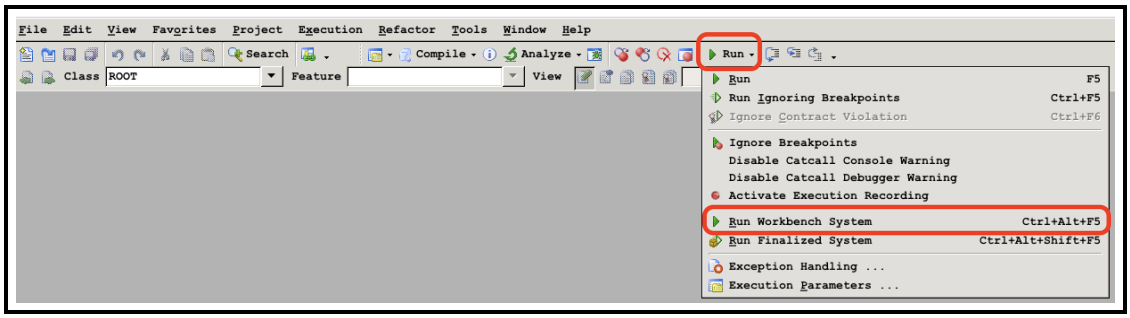
Case Type	Passed	Total
Violation	0	0
Boolean	3	4
All Cases	3	4

State	Contract Violation	Test Name
Test1		TEST_EXAMPLE
FAILED	NONE	t0: First test fails as Result = False
PASSED	NONE	t1: test array of chars
PASSED	NONE	t2: test regular expression ^[abc]*$
PASSED	NONE	t3: test regular expression groups ((.)\2) repeated letters

The top of the HTML test table has a **red bar**: this is bad, because that means not all tests passed (i.e., at least one test failed).

7 Launching Tests in EStudio

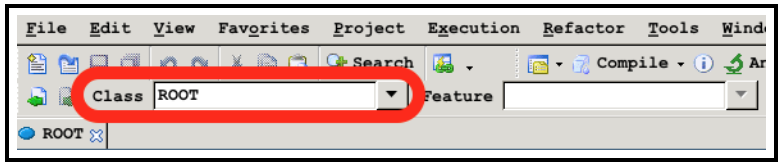
- You typically do not need to run tests and generate the test report from the terminal (recall Section 6.2). Instead, it is more advised that you (re-)run tests and (re-)generate the test report in EStudio. Now switch back to EStudio. Look for a small, downward arrow (▼) besides the `Run` button. Click on ▼ and select `Run Workbench System`.



You should then see the HTML report page popped up again (with a **red bar**).

- Now let's do something (useless) that would make all tests pass (and get a **green bar**).

  On the `Class` text box, type `ROOT` open that class:



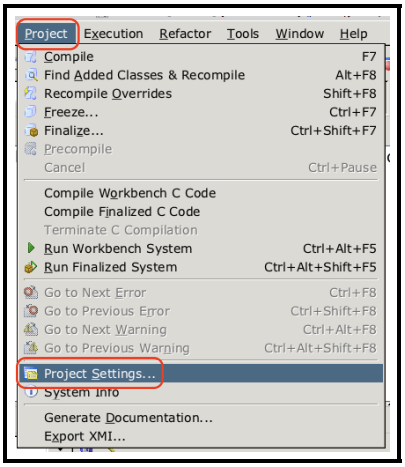
You should now see the ROOT class on an open tab in the EStudio editor:

```
1  class
2         ROOT
3  inherit
4         ARGUMENTS_32
5         ES_SUITE -- testing via ESpec
6  create
7         make
8  feature {NONE} -- Initialization
9         make
10                      -- Run app
11                do
12                      print ("Hello EECS: void safe Eiffel project for 19.05!%N")
13                      add_test (create {TEST_EXAMPLE}.make) --suite of tests
14                      show_browser
15                      run_espec
16                end
17 end
```

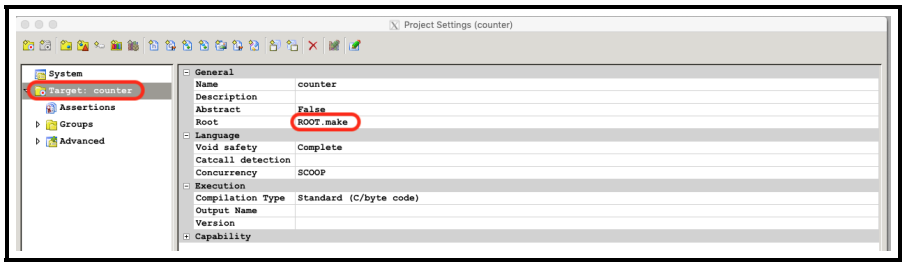
- ◇ Line 3 specifies the list of ancestor classes of ROOT.
- ◇ Line 4 specifies the ancestor class ARGUMENTS_32 that supports the building of a console application. For example, Line 12 calls the inherited feature print to write to the console.
- ◇ Line 5 specifies the ancestor class ES_SUITE that supports the building of a collection of unit-tests. For example, Line 13 adds all test cases implemented in the TEST_EXAMPLE class.

  Note. As you might have observed, Eiffel inherit allows you to have multiple ancestors, unlike Java extends. Also, Eiffel inherit allows all ancestors to contain implementations/code, unlike Java implements (where each interface contains method headers only).
- ◇ Line 6 declares the list of commands (mutators) that can be used as constructors to create instances of ROOT.
- ◇ Line 7 specifies that make currently is the only valid constructor. In general, you may declare as many constructors as you wish. And, unlike Java, Eiffel constructors need not have the same name as the residing class.

- Before we re-run the test, let's show you as to why the ROOT class happens to be the entry point of execution.
  - ◇ At the top of EStudio, go to Project, then select Project Settings.



  - ◇ On the left panel, click on Target: counter, and inspect the right panel. You will see that the Root of the project is set to ROOT.make.



Consequently, when we execute the project, the `make` constructor in the `ROOT` class will be executed. Hence `ROOT.make` being the entry point of execution of the counter project. You can easily change the root of project. See Section 9.2.

- Now that we know that the `ROOT.make` is the entry point of execution, let's now modify the test class which it depends on currently: `TEST_EXAMPLE`. On the `Class` text box, type `TEST_EXAMPLE` open that class and change the `t0` test that failed:

```
t0: BOOLEAN
      do
            comment ("t0: First test fails as Result = False")
            -- this test will fail because Result = False
            Result := True
      end
```

- Now look for the small, downward arrow (▼) besides the `Run` button. Click on ▼ and select `Run Workbench System` to generate a new test report. You should would then see this:

ROOT

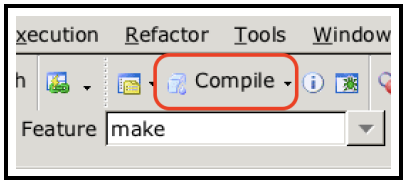
Note: * indicates a violation test case

FAILED (1 failed & 3 passed out of 4)

Case Type	Passed	Total
Violation	0	0
Boolean	3	4
All Cases	3	4

State	Contract Violation	Test Name
Test1		TEST_EXAMPLE
FAILED	NONE	t0: First test fails as Result = False
PASSED	NONE	t1: test array of chars
PASSED	NONE	t2: test regular expression ^[abc]*$
PASSED	NONE	t3: test regular expression groups ((.)\2) repeated letters

Why?! Didn't we just change the test `t0`? Always remember, to have your program exhibit the behaviour specified the latest version of your code, you need to **re-compile** it. In Eclipse your Java code is compiled when it is saved; in Eiffel, saving a class does not compile it. Now on the top of EStudio, click on the `Compile` button:



– After re-compilation, run the Workbench System again. Then you should see this:

ROOT

Note: * indicates a violation test case

PASSED (4 out of 4)

Case Type	Passed	Total
Violation	0	0
Boolean	4	4
All Cases	4	4

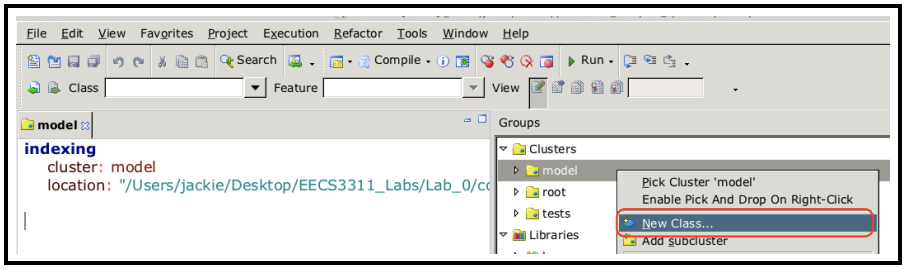
State	Contract Violation	Test Name
Test1		TEST_EXAMPLE
PASSED	NONE	t0: First test fails as Result = False
PASSED	NONE	t1: test array of chars
PASSED	NONE	t2: test regular expression ^[abc]*$
PASSED	NONE	t3: test regular expression groups ((.)\2) repeated letters

Hurrah! All tests passed! But we know that this is not very useful as we have not written any tests for counter yet. But this illustration is meant to remind you that you should always want all tests to pass and get a **green bar**.

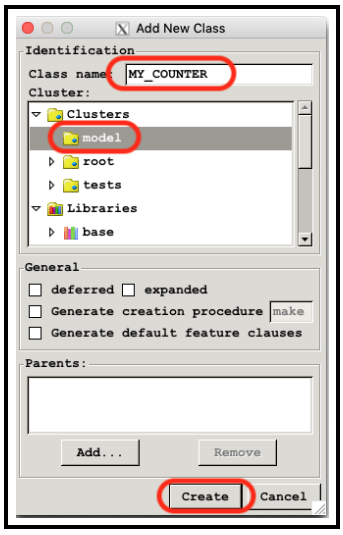
– To write meaningful tests, you need to learn about the ESpec library. See Section 2 for the link to a tutorial series, where you can find parts on writing unit-tests in Eiffel.

8 Creating a New Class for Counter

– In EStudio, on the right panel `Groups`, right click on cluster `model`, then select `New Class...`.



– Enter the `Class name` as `MY_COUNTER` and click on `Create`. In Eiffel convention, class names are all capitals, separated by underscores _ for compound words..



– Type in the following text for the `MY_COUNTER` class (**don't be lazy, type!**):

```
note
  description: "A counter always has its value between 0 and 10."
  author: ""
  date: "$Date$"
  revision: "$Revision$"

class
  MY_COUNTER
create -- We need to explicitly declare which feature is a constructor.
  make

feature -- Attribute: counter value
  value: INTEGER

feature -- Constructor
  make (v: INTEGER)
         -- Initialize counter with value 'v'.
    -- No require clause here means that there's no precondition.
    -- Any input value 'v' will be accepted and used to assign to 'value'.
    do
      value := v
    end

feature -- Commands (mutators in Java)
  increment_by(v: INTEGER)
      -- Increment the counter value by 'v' unless
      -- it causes its value to go above the max.
    require -- Precondition: what's assumed true by the supplier
      not_above_max: value + v <= 10
    do
      -- Implementation
      value := value + v
    ensure -- Postcondition: what's expected to be true, guaranteed by supplier
      value_incremented: value = old value + v
    end

  decrement_by (v: INTEGER)
      -- Decrement the counter value by 'v' unless
      -- it causes its value to go below the min.
    require
      not_below_min: value - v >= 0
    do
      value := value - v
    ensure
      value_decremented: value = old value - v
    end

invariant -- Class invariant: what a legitimate counter means.
  counter_in_range:
    0 <= value and value <= 10
end
```

– Notice that line comments in Eiffel are preceded by `--`.

– Once you have typed the above Eiffel code, compile and make sure everything is ok.

– Study this code via the comments provided to you. Try to understand what's going on, especially how contracts (i.e., preconditions, postconditions, and class invariants) are specified.

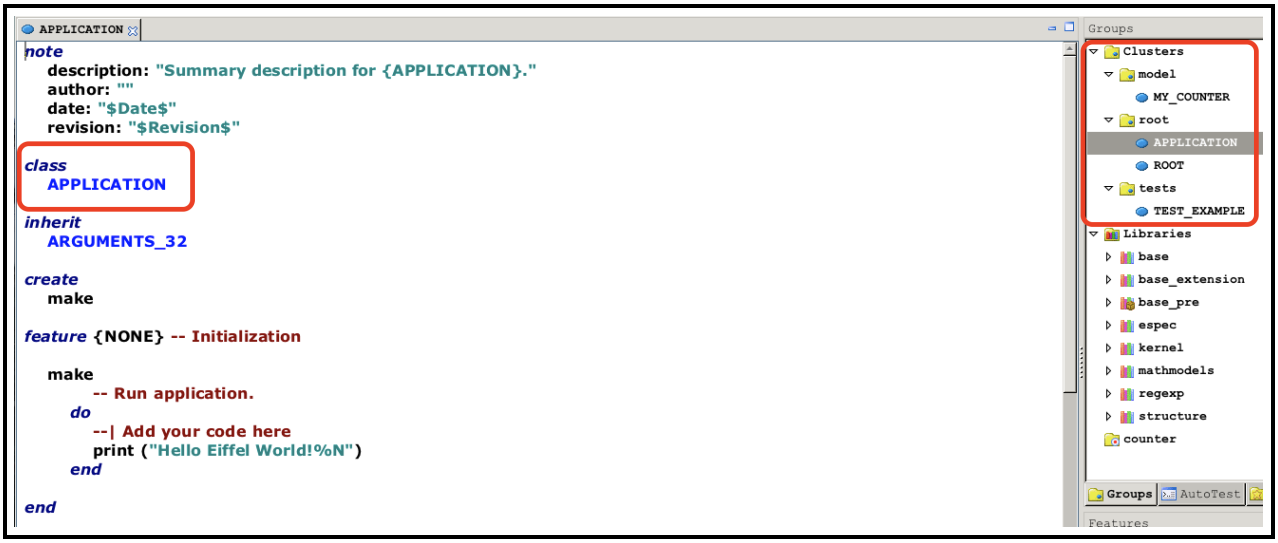
Hints: Under `Views`, switch between the `Basic text view` and `Contract view`:

What differences do you between these two views? Based on what we learned about Design by Contract, which view is **supplier**'s and which one is **client**'s?

9 Adding Another Root Class for Console Outputs Only

9.1 Adding the APPLICATION Class

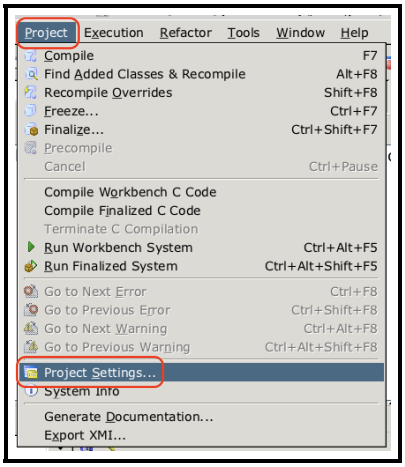
By following the same procedure for adding the `MY_COUNTER` class in Section 8, add a new class `APPLICATION` inside the `root` cluster. This is what you should see after the creation:



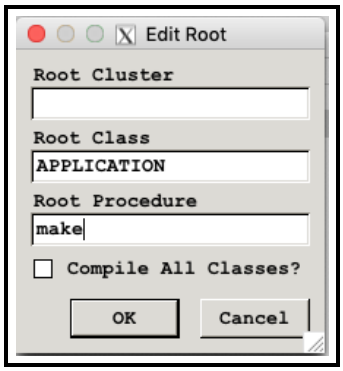
Up to now, remember the root of project has been set to `ROOT.make`. In order to print what's defined in `APPLICATION.make`, we need to change the root of project to `APPLICATION.make`.

9.2 Denoting APPLICATION as the Root

As shown previously, at the top of EStudio, go to `Project`, then select `Project Settings`.



On the left panel, click on `Target: counter`, and inspect the right panel. Now click on the existing **ROOT.make**, and in the popped up editor box, enter `APPLICATION` as the Root Class and `make` as the Root Procedure:



Click on OK and **make sure you compile**! Question: What would happen if you do not re-compile and run the workbench system?

9.3 Change of Root Class ⇒ Change of Behaviour

– The root of project (entry point of execution) has been changed to `APPLICATION.make`, a re-compilation was done to make sure that change is now effective. Now we can illustrate this on the terminal.

Find a line in the class `APPLICATION` that reads: `print ("Hello Eiffel World!%N")`. Notice that the percentage sign `%` there means the start of an escape sequence (whereas in Java, you start an escape sequence using a backward slash \). The Eiffel escape sequence `%N` here denotes the new-line character.

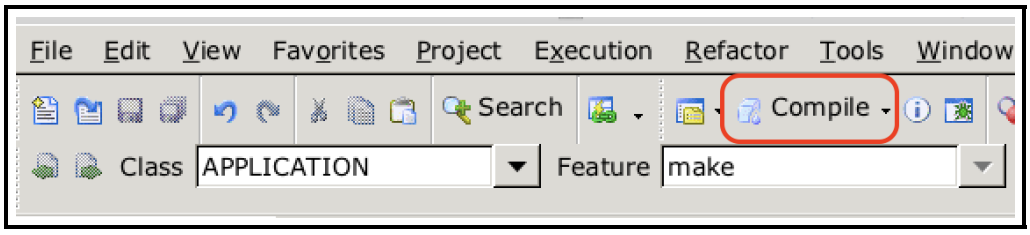
Now, modify that line to: `print ("Hello Eiffel World @ EECS3311!%N")` and save the file (`Ctrl + s`). Then, switch back to your terminal and run the (un-finalized) executable again from `W_code`:

```
~/Desktop/eecs3311_workspace/counter/EIFGENs/counter/W_code/counter
```

The output should be:

```
Hello Eiffel World!
```

Aah! Shouldn't the output be changed to `Hello Eiffel World @ EECS3311!%N`?! The reason for this is because, again, **we did not re-compile for the changed code to take effect**. We somehow have been spoiled by Eclipse in the previous Java courses, where a re-compilation is performed automatically as soon as you save your Java file. **In EStudio, saving a file does not trigger re-compilation automatically.** To fix this, re-compile your code from EStudio:



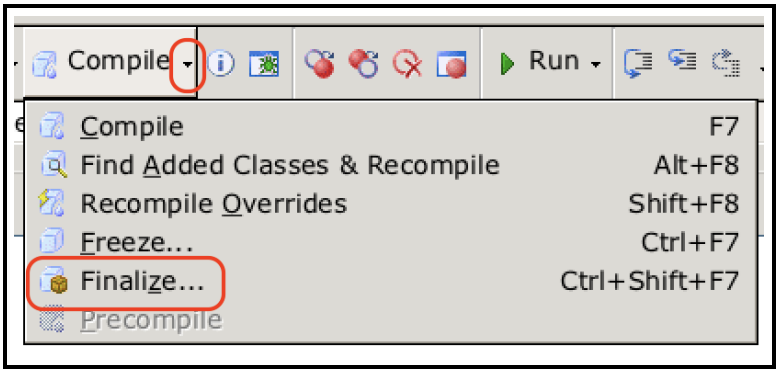
Try running the executable again:

```
~/Desktop/eecs3311_workspace/counter/EIFGENs/counter/W_code/counter
```

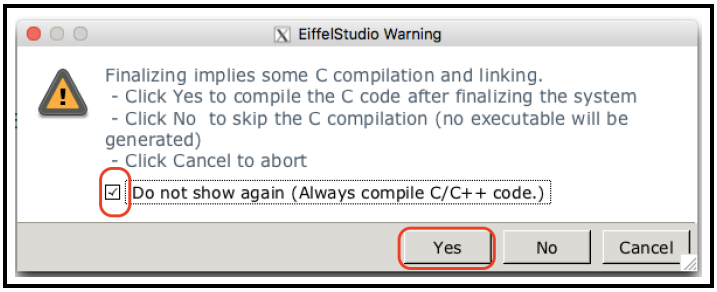
You should now get the expected output:

```
Hello Eiffel World @ EECS3311!
```

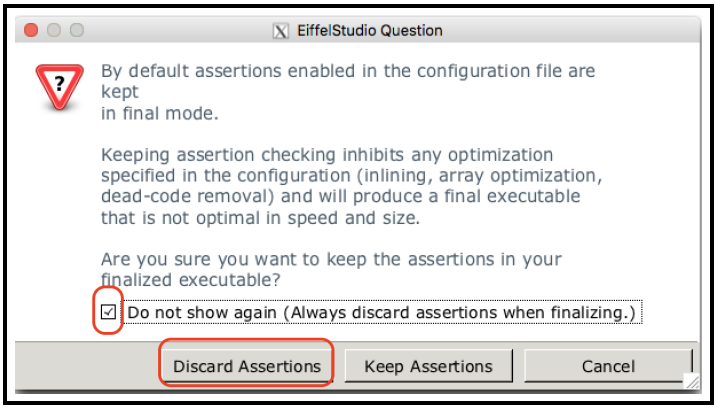
– The next question is: How do we finalize the project then? In EStudio, click on the tiny, up-side-down triangle symbol right beside `Compile`, and this will give you a drop-down menu list of options: click on `Finalize...`.



- Tick the check box `Do not show again (Always compile C/C++ code)` and click `Yes`.



- Tick the check box `Do not show again (Always discard assertions when finalizing)` and click `Discard Assetions`.



Then wait until the `Outputs` panel indicates that the compilation is completed.

- After finalizing the project, let's switch back to the terminal and type:

```
ls ~/Desktop/eecs3311_workspace/counter/EIFGENs/counter/F_code
```

You will see that there is, similar to the case of `W_code`, also an executable file called `counter`. Run this (finalized, optimized) executable by typing:

```
~/Desktop/eecs3311_workspace/counter/EIFGENs/counter/F_code/counter
```

Remark: The executables in `W_code` and `F_code` have no difference in terms of their behaviour. It is only that one (in `F_code`) has better performance than the other (in `W_code`). While your still developing your project, there is no need to finalize your code, as it takes time to finalize/optimize each time.

9.4 Redefining the APPLICATION Class

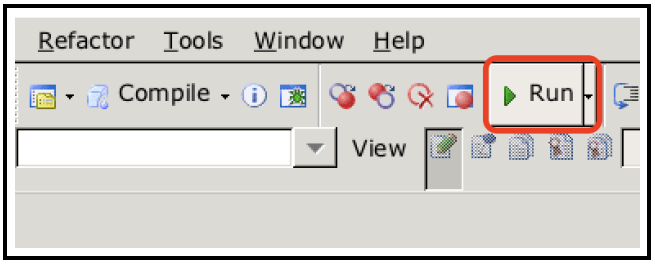
- Now we further modify the `APPLICATION` class so that it manipulates the `MY_COUNTER` class (Section 8)
- Go to the `APPLICATION` class that we modified before, change its `make` feature so that it looks like:

```
make
    -- Run application.
  local -- local variables
    c: MY_COUNTER
  do
    create {MY_COUNTER} c.make (-10)
    print (c.value)
  end
```

where

- In Eiffel, assignments are done using `:=`, whereas value comparisons are done using `=` (which corresponds more closely to math).
- The Eiffel syntax `c: MY_COUNTER` for declaring a variable corresponds to `MY_COUNTER c` that you write in Java.
- The Eiffel syntax `create {MY_COUNTER} c.make (-10)` for creating a new object of type `MY_COUNTER` corresponds to `MY_COUNTER c = new MY_COUNTER(-10)` that you write in Java.

- Equivalent to running the executable from the `W_code` or `F_code` directory, on EStudio click on `Run` to execute the code. Remember: when you wish to see the HTML test report, you need to **Run the Workbench System**; otherwise, just click on **Run**.



- Then you should run into this contract violation (i.e., the class invariant of `MY_COUNTER` is broken):

10 Your Tasks

- **Task 1**: Study this tutorial series on DbC (Design by Contract) and TDD (Test Driven Development):

  `https://www.youtube.com/playlist?list=PL5dxAmCmjv_6r5VfzCQ5bTznoDDgh__KS`

  Pay special attention to how to write unit-tests in Eiffel.

- **Task 2**: Can you explain why a violation of class invariant occurs in the above `APPLICATION` class?

  Hint: What does it mean to be a legitimate `MY_COUNTER` instance and where is that explicitly defined as a **contract** (i.e., precondition, postcondition, or class invariant) in the `MY_COUNTER` class.

- **Task 3**: There are two ways to fixing this class invariant violation (try both and verify that it does get rid of the contract violation):

  - From the supplier `MY_COUNTER` side, does the precondition (specified using `require`) have any missing cases of illegal values? If so, fix the current precondition.
  - From the client `APPLICATION` side, let them pass a legitimate value for initializing the counter.

  Modify either the `APPLICATION` class or the `MY_COUNTER` class, so that you can observe other kinds of violations:

  - **Precondition violation (caused by illegal inputs by client)**: When a feature call (or method call in Java) is passed with an input argument value that does not satisfy the Boolean condition under the **require** clause.
  - **Postcondition violation (caused by wrong implementation by supplier)**: When a feature call's input argument value satisfies its precondition, then after executing its implementation (i.e., what goes between **do** and **ensure**), the object state (i.e,. in this case the counter value) does not satisfy the Boolean condition under the **ensure** clause.

- **Task 4**: Switch the root of project back to `ROOT.make`. Then, based on what you learned about TDD in the above tutorial series, add 3 different, but **meaninful** tests to the `TEST_EXAMPLE` class and make sure they all pass (i.e., a **green bar**).

  Note. Remember: when you wish to see the HTML test report, you need to **Run the Workbench System**; otherwise, just click on **Run**.

- We will continue from here in the lectures.

11 Recompiling from Scratch

Whenever your Eiffel project is behaving weirdly, **re-compiling everything from scratch** is always the first thing to try. There are two ways to achieving this. For both ways, **close your Eiffel Studio first!**

Approach 1. On a terminal, run the following command (only available on your Prism account) on your project:

```
eclean ~/Desktop/eecs3311_workspace/counter
```

This will remove the EIFGENs folder and some other auxiliary files. You are now ready to launch EStudio and compile.

Approach 2. Without removing the EIFGENs, simply re-launch EStudio, and check the `clean` box before you start compiling.