Object Spyglass User's Guide

Peter Brandt Martin Geisler

Object Oriented Software Construction
Summer Semester 2005

Contents

1 Introduction

A user-friendly GUI is provided for running the Test Cases on the code under test, and exploring last saved system state in the event of a test failure. In order to prepare code for testing, it must be adapted to make use of the state saving functionality. Test Cases must be written, and a Test Driver must be created to select the Test Cases to run.

The discussion of the Testing Framework in this document is intended to be brief and include examples to help users begin using it immediately. For a more detailed discussion of the underlying classes and architecture, please consult the Developer's Guide.

2 Using the GUI

The GUI as provided by the *GRAPHICAL_TEST_DRIVER* will now be described.

2.1 Program Startup

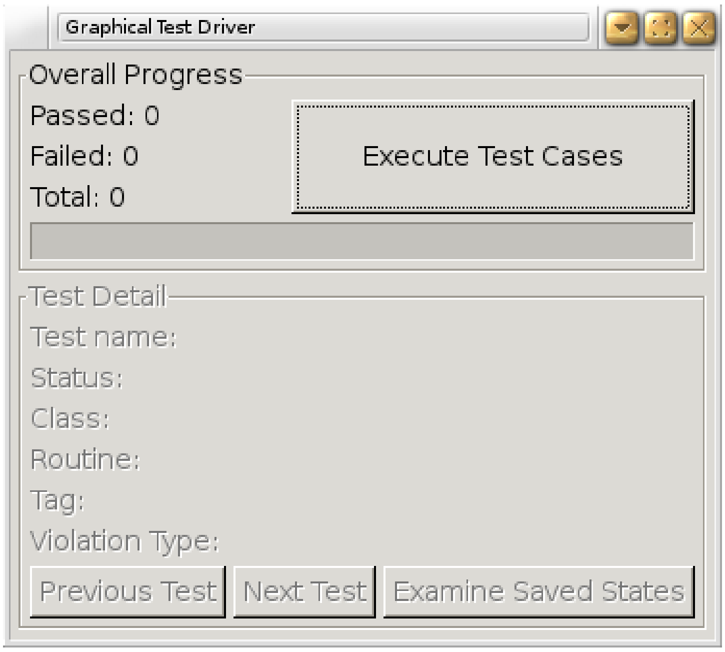


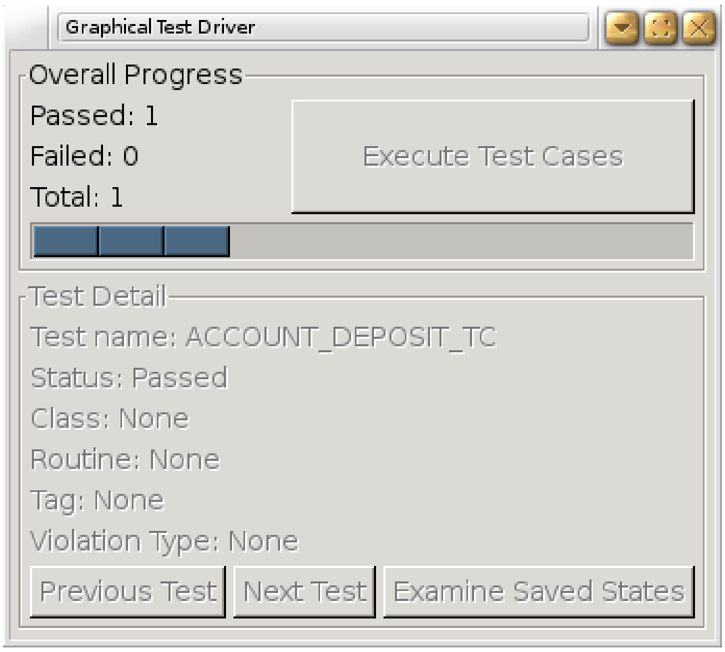
Figure 1: The program when it has just started.

When the program is first run, this is the window the user sees.

The window has two frames labelled "Overall Progress" and "Test Detail". When just started only the upper frame is active.

When "Execute Test Cases" is clicked, the test cases will be run on the code under test.

2.2 During Test Execution



Passed: 1
Failed: 0
Total: 1

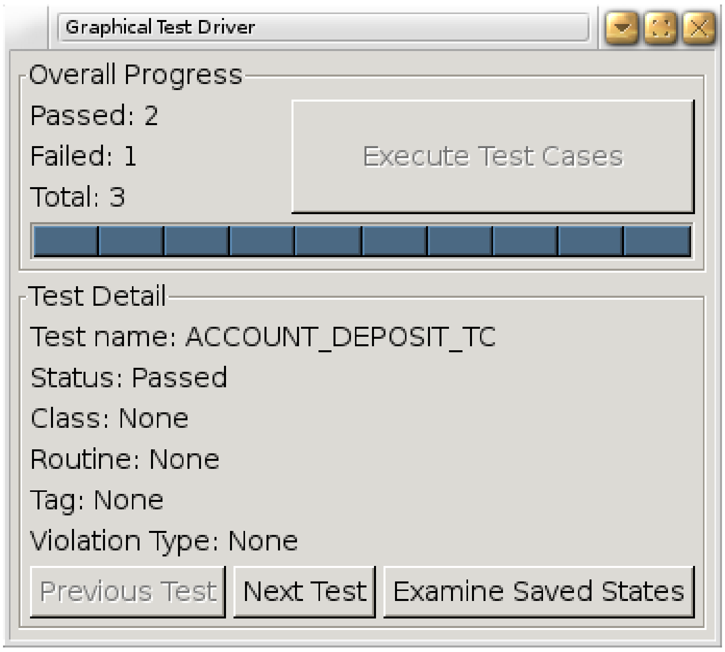
Test name: ACCOUNT_DEPOSIT_TC
Status: Passed
Class: None
Routine: None
Tag: None
Violation Type: None

Figure 2: Progress updates when tests are running.

As the test cases are running, a progress bar lets the user know the progress of the testing. At the same time the number of passed and failed tests as well as the total number of executed tests is displayed.

At the bottom frame the user sees more detail about the last executed test — this will most likely update too fast to be noticable, but the user can browser through all the information when the test run is completed.

2.3 Test Run Finished



Passed: 2
Failed: 1
Total: 3

Test name: ACCOUNT_DEPOSIT_TC
Status: Passed
Class: None
Routine: None
Tag: None
Violation Type: None

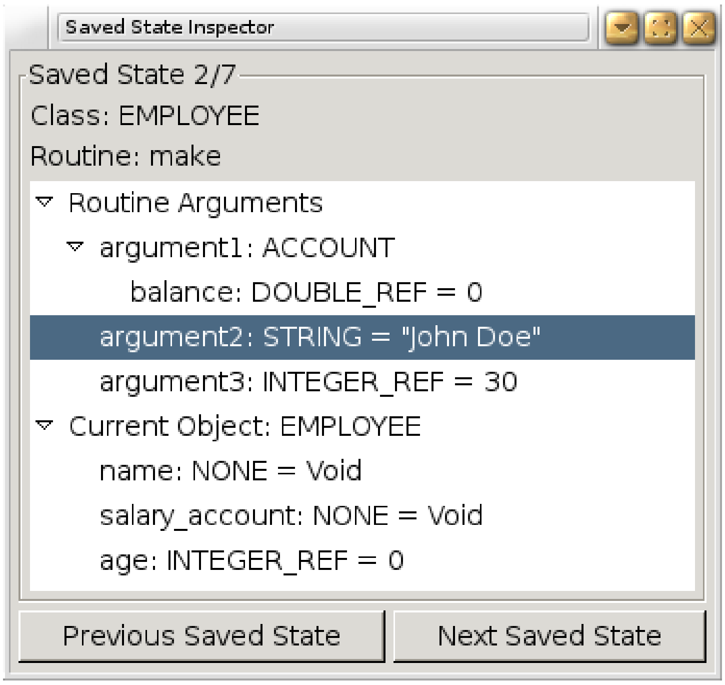
Figure 3: All tests have been executed.

The test run is finished when the progress bar is completely filed. The upper frame then shows the statistics for the test run.

The bottom frame is now active, and the user can browse through the tests with the “Previous Test” and “Next Test” buttons, and see information about the results of the individual test cases. For each test the frame displays the name and status (“Passed” or “Failed”) of the test. If the test has failed, then information about the failure will be displayed as well: the class and routine name in which the failure occured and any tag associated with the assertion violation, and finally the type.

The “Examine Saved States” button will open a new window that uses the Object Spyglass to display information about the states saved during the execution of the test case.

2.4 Examining a Saved State



```
Saved State 2/7
Class: EMPLOYEE
Routine: make
▽ Routine Arguments
    ▽ argument1: ACCOUNT
        balance: DOUBLE_REF = 0
      argument2: STRING = "John Doe"
      argument3: INTEGER_REF = 30
▽ Current Object: EMPLOYEE
      name: NONE = Void
      salary_account: NONE = Void
      age: INTEGER_REF = 0
```

Figure 4: Examining a Saved State.

After clicking the “Examine Saved States” button, the Saved State Inspector opens with information about the states for the test case in question. The window will automatically select last saved state since this is the one which normally contain the most interesting information.

The class name of the current object saved in the state as well as the routine name is displayed at the top, followed by a tree in which the routine arguments and current object can be explored using the Object Spyglass.

The Object Spyglass will recursively display object information, so that the user can browse as deep as he or she wants into the routine arguments or current object fields.

A single test case will likely produce more than one saved state. With the “Previous Saved State” and “Next Saved State” buttons the user can browse through the states saved during the execution of the test case.

3 Adapting Code for Testing

In order for the testing framework to work optimally, the state (defined as current object, routine name, and routine args) needs to be saved at the beginning of each routine. There is very lightweight code to accomplish this shown below, where the items in angle brackets should be replaced by the actual values. Remember that the code under test should be written using good design-by-contract techniques, because bugs are detected using contract violations only.

```
debug ("save_states")
     (create {STATE_MANAGER}).save_state
          (Current, ⟨routine name⟩, ⟨routine args⟩)
end
```

Here the ⟨routine name⟩ should be substituted with the name of the calling routine (a *STRING*) and the ⟨routine arguments⟩ should be substituted with the arguments of the routine (a *TUPLE*).

The code example makes use of a **debug** clause to enable conditional compilation. To enable the line the `.ace` file should be modified by adding the following line in the **default** group:

```
     debug ("save_states")
```

The benefits of this type of architecture are great, because there is no need for the code under test to be any different from the original code. If the line above is replaced with

```
     disabled_debug ("save_states")
```

then the code to save states will simply be skipped by the compiler and wont have any impact on the program.

4 Creating Test Cases

The role of Test Cases is to use the classes under test in such a way that tests their functionality and adherence to the built in contracts. All test cases must inherit from the *TEST_CASE* class and must implement the *execute_test* routine. The *execute_test* routine should create instances of a the class under test and test its functionality. An example Test Case class for testing a simple *ACCOUNT* class is shown below.

```
class
     ACCOUNT_WITHDRAW_TC

inherit
     TEST_CASE

feature -- Basic operations
```

```
execute_test is
        -- Test withdraw in ACCOUNT
    local
        account: ACCOUNT
    do
        create account.make (42)
        -- Test normal withdrawal
        account.withdraw (10)
        -- Test withdrawal to empty account
        account.withdraw (32)
    end

end -- class ACCOUNT_WITHDRAW_TC
```

5 The Test Driver

The Test Driver is a very simple class whose purpose is to determine which Test Cases to run and what to output. To easily construct a test driver, create a class that inherits from either *GRAPHICAL_TEST_DRIVER* or *CONSOLE_TEST_DRIVER*. A discussion of implementing other test drivers can be found in the Developer's Guide.

A Test Driver that inherits from *GRAPHICAL_TEST_DRIVER* will launch the GUI described above. A Test Driver that inherits from *CONSOLE_TEST_DRIVER* will output basic information about the test results to the console.

All custom Test Drivers must call the *make* routine to initialize the ancestor *TEST_DRIVER* class, and should use the *add_test_case* routine to add *TEST_CASE* objects to the list of Test Cases to run. An example Test Driver class based on the *GRAPHICAL_TEST_DRIVER* class is shown below.

```
class
    TEST_DEMO

inherit
    GRAPHICAL_TEST_DRIVER

create
    run

feature -- Initialization
    run is
            -- Create test cases
        do
            make
            add_test_case (create {ACCOUNT_DEPOSIT_TC})
            add_test_case (create {ACCOUNT_WITHDRAW_TC})
            add_test_case (create {EMPLOYEE_RECEIVE_SALARY_TC})
            execute
        end

end -- class TEST_DEMO
```

After everything has been coded, make sure that the `.ace` file points to the *run* creation procedure in the Test Driver class.

Two `.ace` files, `test-demo-linux.ace` and `test-demo-windows.ace`, are delivered along this guide. They are for use under Linux and Windows, respectively, and will execute the *run* creation procedure in the *TEST_DEMO* class described.