Object Spyglass Developer's Guide

Peter Brandt Martin Geisler

Object Oriented Software Construction
Summer Semester 2005

Contents

1 Introduction

This guide explains how the Testing Framework and the accompanying Object Spyglass tool are used from the viewpoint of developers. Normal users are referred to the Object Spyglass User's Guide.

The target audience of developers can be reasonably divided into two groups: reusable component developers and software testers. The component developers only have to deal with a subset of the functionality in the Testing Framework (the State Manager to be precise), while the software testers make use of the Testing Framework for writing Test Cases and the Object Spyglass tool for inspecting the results of running such tests.

This guide has the following layout:

- First the overall architecture of the system and the main concepts used in the rest of the guide are introduced. Both component developers and software testers should be familiarize themselves with this section.
- The Testing Framework is then described in detail. The functionality is divided into two parts of interest to different audiences:
  - The Test Driver and Test Cases: this is the domain of the software testers. How to write Test Cases and how to integrate them using the Test Driver is explained.
  - The State Manager: the component developers should use it in their components to facilitate better debugging by the software testers. This section demonstrate how libraries can be instrumented with calls to the State Manager in an unobtrusive way.
- The Object Spyglass component is then presented, along with a description of how it works and how it is used.
- The Testing Framework and Object Spyglass component are already quite functional as presented here, but one might still wish to change and extend them. This section discusses some possible extensions.
- A conclusion which describes the highlights and limitation of our system.

2 Architectural Overview

This section introduces the main concepts and classes used in the Testing Framework and in the Object Spyglass. The definitions given here are the basis for the rest of this document.

2.1 The Testing Framework

The first major part in our system is the Testing Framework. Software testers will be particularly interested in the functionality of the Test Cases and Test Diver, whereas component developers will be interested in the facility for saving state through the State Manager.

2.1.1 The Test Driver and Test Cases

The most basic component in the Testing Framework is the Test Case, realized by the *TEST_CASE* class. This is also the most important class for the software testers.

The job of a test case is to exercise some code in a controlled environment, seeking to trigger bugs. For our purposes a bug is defined as a contract violation.

The execution of the test cases is supervised by a Test Driver — an instance of the *TEST_DRIVER* class. Test Cases are added to an instance of this class and when executed the Test Driver keeps track of the number of passed and failed tests.

2.1.2 The State Manager

The component developers can aid the software testers by having the libraries and components save their state at regular intervals. This is done with by calling the State Manager which maintain a list of Saved States.

After having collected a number of Saved States one can examine them using the Object Spyglass. This functionality is implemented by the Saved State Inspector which is a client of the Object Spyglass.

2.2 The Object Spyglass

The Object Spyglass is the second major component in our system. It has the task of showing the structure of an object — it must be able to deal with arbitrarily complex objects, including objects with cycles and self-references.

The structure of an object given to the Object Spyglass is shown as a tree, with the object at the root. The nodes in the tree correspond to the fields of the objects.

The Object Spyglass is a separate component that can be reused in many different contexts, and it does not depend on the Testing Framework in any way.

3 The Testing Framework

This section describes the classes in the Testing Framework and their use. As explained in the introduction, the first section about Test Cases and the Test Driver targets software testers, while the second section about the State Manager targets component developers.

3.1 The Test Driver and Test Cases

The most basic part of the Testing Framework is the Test Case. Below is a discussion of how to write Test Cases, and how to create a test suite using the Test Driver.

3.1.1 Writing Test Cases

Writing test cases with our Testing Framework is meant to be as easy as possible, requiring a minimal amount of "glue code". Writing a test case consists of two steps:

1. Making a class which inherits from *TEST_CASE*.
2. Effecting the deferred procedure *execute_test*.

The *execute_test* procedure is called whenever the test case is executed, and should contain the actual test code. The test code is free to do whatever is needed for initializing a proper test environment. The test case should create instances of the class under test, and test its functionality thoroughly. A simple example testing an *ACCOUNT* class could look like this:

```
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)
            account.withdraw (42)
        end

end -- class ACCOUNT_WITHDRAW_TC
```

Here the *execute_test* procedure simply declares a local variable of type *ACCOUNT*, creates the object with an initial balance of 42 and then tries to withdraw 42 from it. Because the Testing Framework relies on contract violations to signal bugs this is all the code that is necessary in the test case.

3.1.2 Executing Test Cases

A Test Case can be executed by calling its *execute* procedure. Note that one cannot call the *execute_test* directly — the *execute* procedure is there to wrap the *execute_test* procedure, in particular it deals with any exception that might be produced by running *execute_test*.

After a Test Case has been executed one can obtain information about the result using the *has_failed* query. If set, then the execution of *execute_test* caused an assertion violation, which is stored for inspection in the *exception* query. Other information about the exception can be obtained by the *class_name* and *routine_name* queries contain the name of the class and routine in which the exception was raised. The *tag_name* query gives the tag associated with the violated assertion clause.

3.1.3 Test Driver

Normally one does not just execute a single Test Case, but a number of them in conjunction with a Test Driver. The *TEST_DRIVER* class itself does not produce any output, but contains the core functionality of a Test Driver. The *GRAPHICAL_TEST_DRIVER* and *CONSOLE_TEST_DRIVER* classes are two examples that use facility inheritence from *TEST_DRIVER*, and implement different output channels for testing information.

The *TEST_DRIVER* class has a simple interface with a *add_test_case* procedure used to register a *TEST_CASE* object and an *execute* procedure used to execute all registered Test Cases. During the execution of the tests the Test Driver provides the number of passed and failed tests through the queries *passed* and *failed*.

The easiest way to use a Test Driver is construct a class that inherits from *TEST_DRIVER* and implements a creation procedure that does the following:

- Calls the `make` creation procedure of *TEST_DRIVER*,
- Adds Test Cases to run using *add_test_case*,
- And finally calls *execute*.

The *TEST_DRIVER* class also has support for calling an agent after each *TEST_CASE* has been executed — this is used by the *GRAPHICAL_TEST_DRIVER* class to ensure that the user interfaces is updated while the tests are being executed, and by the *CONSOLE_TEST_DRIVER* to output textual information about the tests. This functionality is implemented using an *ACTION_SEQUENCE*, available through the *test_executed_actions* feature.

3.1.4 Implementation of the Test Case and Test Driver

Test Cases need access to internal object information and to information about assertion violations, and thus *TEST_CASE* use facility inheritance from *EXCEPTIONS* and *INTERNAL* to gain access to those features.

The *TEST_CASE* class is itself a deferred class that, at its most basic level, is a wrapper for the code under test which is to be implemented in the deferred procedure *execute_test*.

The *execute_test* procedure is called when the Test Case is executed by a call to *execute*. Before the call to *execute_test* the *Saved_states* list in *STATE_MANAGER* is wiped out so that it will only contain Saved States related to the current Test Case when the call to *execute_test* returns.

The *execute_test* call might produce an exception, and if so this is recorded in the **rescue** clause. A **retry** sends the execution back in the body, where the execution of the test code will be skipped, and any Saved States are stored away for later inspection.

TEST_DRIVER is implemented as primarily a list of *TEST_CASE*s *test_cases*, which the procedure *add_test_case* adds to. To trigger the execution of all added Test Cases, simply call the procedure *execute*.

There are also *ACTION_SEQUENCE*s *execute_started_actions*, *execute_finished_actions*, and *test_executed_actions*. They are run before the tests are executed, after the tests are executed, and after each executed Test Case respectively. In the *TEST_DRIVER* class these lists are empty, and thus *TEST_DRIVER* itself produces no output; it simply provides a facility for running tests. The intention is that classes that inherit from *TEST_DRIVER* such as *GRAPHICAL_TEST_DRIVER* be constructed, that create output by adding to the *ACTION_SEQUENCE*s. In this way,

the core functionality that a Test Driver should exhibit is separated from implementing output channels.

3.2 The State Manager

To aid debugging one should have libraries save their state regularly during execution. An excellent way to implement this is to save the state of execution at the beginning of every routine. Saving state is accomplished with a State Manager, implemented with the *STATE_MANAGER* class.

The state of the system is defined as the following:

- The **Current** object. This is the active object at the point in time when the State Manager is invoked to save the state.
- The name of the current routine. This is just a label (a *STRING*) supplied by the writer of the code.
- Any routine arguments. This is a *TUPLE* of variables.

To store these three things one makes a call to *save_state* in *STATE_MANAGER*. The interface in *STATE_MANAGER* has been designed to be as simple as possible, so this is all that is needed: a single call to a single routine. This can be done in a single line:

```
(create {STATE_MANAGER}).save_state (Current, "name", [a, b])
```

Note that it is possible to use the *STATE_MANAGER* class to store other data than the current object, routine name, and routine arguments. This is beyond the scope of the project, but deserves mention because it may be useful to store intermediate state when debugging very complicated algorithms.

3.2.1 Saving the State Conditionally

Having each call to a routine result in a *SAVED_STATE* object being created with a copy of the arguments and the current object would be unnecessary and probably much too expensive for an application once it is being put into production use. Luckily, Eiffel provides a simple solution: wrap the calls to the State Manager in **debug** clauses. The call to the State Manager thus becomes:

```
debug ("foo_save_states")
    (create {STATE_MANAGER}).save_state (Current, "name", [a, b])
end
```

One then has to enable the debug clauses in the `.ace` file by inserting the following line in the **default** clause:

```
debug ("foo_save_states")
```

When the library is used in a production system one only has to disable the debug statements in the `.ace` file. By consistently using the same tag throughout a library (like "foo_save_states" for a library called "foo") the users of the library can choose to disable just the debug statements they want. However, even if all debug statements involving the State Manager are disabled, the Testing Framework still needs to be included in the project for compilation.

So, by using this technique one can equip a library with calls to the State Manager without having to fear any run-time consequences in the production system, and there is no need for multiple versions of the same code. Creating and using the *STATE_MANAGER* object all in one line one also avoids having to introduce a new local variable — those three lines are fully self-contained and their presence cannot introduce any new bugs.

3.2.2 The Saved State Inspector

The saved states are explored using the Saved State Inspector, which in turn uses the Object Spyglass to recursively display the routine arguments and objects in the saved states. The Saved State Inspector allow the user to browse the full history of Saved States for a *TEST_CASE*. In addition, multiple instances are possible so that the user can view multiple Saved States side-by-side.

This is implemented in the *SAVED_STATE_INSPECTOR* class, which is instantiated with a list of *SAVED_STATE* objects, and calls the *SPYGLASS_TREE_ITEM* class to generate a tree for display.

3.2.3 Implementation of the State Manager

The State Manager is implemented in the class *STATE_MANAGER*. The purpose of the State Manager is to provide the code under test a simply way to save state, that is, to create and store away new *SAVED_STATE* objects. Remember that the code under test and its Saved States are both a part of the *TEST_CASE* class.

STATE_MANAGER contains a list of *SAVED_STATE*s in the **once** feature *Saved_states*. *Saved_states* is visible to *TEST_CASE*, because *TEST_CASE* needs to call wipeout to clear the list at the beginning of the execution of the test case. The code under test saves state by calling the procedure *save_state* in the *STATE_MANAGER* provided by *TEST_CASE*. *save_state* then creates a new *SAVED_STATE* from the arguments and adds it to *Saved_states*, so that the code under test only needs to know about *STATE_MANAGER* and not *SAVED_STATE*.

4 The Object Spyglass

The Object Spyglass is an EiffelVision2 component that can be used to display the structure of an arbitrary object clearly in tree form. An object is represented as a tree with the object under investigation as the root node. The Object Spyglass creates a tree using the objects referenced in the fields of an object. It then recurses through the fields creating new trees as the user browses deeper into the tree. It is a very versatile tool for visualizing object state, and could easily be used in other contexts than software testing.

4.1 Output of the Object Spyglass

The text in a *SPYGLASS_TREE_ITEM* mimics the Eiffel syntax for declaring manifest constants, meaning that it consists of three parts for basic types: "⟨name⟩: ⟨type⟩ = ⟨value⟩". Here the ⟨name⟩ part is the field name, ⟨type⟩ is the dynamic type of the object attached to the field, and ⟨value⟩ is the value of the field. Basic types are defined as types conforming to *NUMERIC*, *BOOLEAN_REF*, *CHARACTER_REF*, *WIDE_CHARACTER_REF*, and *STRING*.

Basic types values are displayed in an appropriate way, for instance values for the *STRING* class are surrounded by double quotes and values for the *CHARACTER_REF* class are surrounded by single quotes. These types are not expanded further and thus make up the leaf nodes in the tree.

For objects of non-basic types the ⟨value⟩ part of the node text is empty and the equal sign is omitted. These nodes have zero or more child nodes, one for each field in the object. Each of these nodes is itself a *SPYGLASS_TREE_ITEM* object, and the fields of the object can thus be explored in exactly the same way as the object itself.

A special case is objects that conform to *CHAIN* or *TUPLE*, which have their items given names such as "chain_item_1" or "tuple_item_1" and recursively created as *SPYGLASS_TREE_ITEM* objects.

4.2 Implementation of the Object Spyglass

The Object Spyglass is implemented by having a class *SPYGLASS_TREE_ITEM* inherit from the EiffelVision2 *EV_TREE_ITEM* class. Clients of the Object Spyglass thus only have to create a normal *EV_TREE* and then insert one or more *SPYGLASS_TREE_ITEM* nodes into it. These nodes automatically create a suitable text and create child nodes as needed.

There are two different creation routines, make and make_internal. Both routines accept an object of *ANY* type and the name of that particular instance of the object as a *STRING*. The difference is that make

should be called by the client on the initial *SPYGLASS_TREE_ITEM*, and make_internal is only called recursively by other instances of *SPYGLASS_TREE_ITEM*. The only difference is that the make routine calls the *create_childrenroutine*, which creates the children of the current object recursively.

This is necessary because each *SPYGLASS_TREE_ITEM* assumes its children have already been created. The recursion is accomplished by attaching the routine *create_grand_children* as an agent to each node's expand action. So, each visible node already has children, and when expanded each node creates its grand children.

In this way the *SPYGLASS_TREE_ITEM* is able to handle cyclic and self-referencing objects without problems, and not waste resources creating parts of the tree the user is not interested in. The children are always stay one step ahead of what the user can see, and the user is free to browse as deeply as he or she wants until there are nothing but basic types.

5 Possible Extentions

Even though the Testing Framework and the Object Spyglass are quite functional already, there is room for extensions. Three possible extensions will be discussed next.

5.1 Hierarchical Grouping of Test Cases

In the current implementation the Test Driver contains a list of Test Cases to be executed. One could easily imagine treating the Test Driver itself as a Test Case.

Upon execution it would then execute its own Test Cases. The result of the *has_failed* query could be defined as the disjunction of the *has_failed* queries of the Test Cases stored. This would mean that a Test Driver is considered to have failed when one of its tests have failed.

With such a scheme the Test Driver Window would need to be adapted accordingly, perhaps using a tree widget to show the tree structure of the tests. This would be very practical when the system is large for then each division could have their own Test Cases, driven by their own Test Driver. Those Test Drivers could then be added as Test Cases and all Test Cases could be run with just a single invocation of this "Super Test Driver".

5.2 Interactive Graphical Object Spyglass

Displaying the object graph as a tree is not optimal since the graph is normally *not* a tree! Displaying it in a graphical way similar to what the professional edition of EiffelStudio provides would be better. Here each object is represented by an oval, with arrows connecting ovals to illustrate object attachments. In such a system a cyclic structure would be nicely represented as a cycle in the drawing, and not as an ever-expanding tree.

Implementing this would require changes in the Object Spyglass and in the Saved State Inspector which is a client of the Object Spyglass. The Testing Framework would not be affected.

5.3 Multiple Concurrent Output Channels

With the current design one uses exactly one Test Driver at a time. But having a design where one could attach multiple Test Driver Monitors to a single Test Driver could be interesting. One Monitor could implement a graphical GUI like *GRAPHICAL_TEST_DRIVER* does whereas another Monitor could store the results about the test run in a database, publish it on a web page or something similar.

We have actually had such an implementation at one point in the development, but we removed it since it was unclear how one could keep the current simple code style (inherit from or create a Test Driver, add Test Cases, call *execute* on the Test Driver) without resorting to what can best be described as a gross hack.

In particular, a Test Driver Monitor which uses the Vision2 library has to somehow "hook into" the call to *execute* on the Test Driver so that it can call *launch* in *EV_APPLICATION* to enter the Vision2 event loop. And when the event loop terminates, then the Monitor would have to make sure that the application terminates too, something which again would require extra hooks in *TEST_DRIVER* just for this special case.

6 Conclusion

We believe that the Testing Framework and Object Spyglass presented here live up to the Project Specification given and the Project Requirements. Some highlights and limitations will now be discussed.

6.1 Highlights

- The Saved State Inspector allows for browsing the list of saved states, as well as comparing saved states side-by-side.

- Very unobtrusive code for saving state and writing test cases, which greatly increases user-friendliness (or perhaps more importantly to the reader, developer-friendliness).
- Intuitive Object Spyglass that displays data fields in a pretty Eiffel-style way.
- Extensible code, allowing multiple output channels for testing data and smooth integration of yet to be developed components.

6.2 Limitations

- Cyclic and self-referencing objects are not detected, leaving the possibility of browsing endlessly deep in the Object Spyglass.
- While the tree visualization is an excellent way to see data fields, it would be nice to also have a more robust graphical output showing the relationships in the class heirarchy