Object Spyglass
Project for Object-Oriented Software Construction
Summer Semester 2005

Prof. Dr. Bertrand Meyer

Prepared by: Ilinca Ciupa

1. Overview

The goal of this project is to develop a visualizer for object state, called the "Object Spyglass", together with a system that uses the Spyglass's functionality. Both the Spyglass and its client system will be delivered as part of the project. The next two sections describe these two components. They are followed by an example of possible use of the system, and descriptions of the required tasks and of the criteria for evaluating the project.

2. The "Object Spyglass" Concept

The Spyglass should be able to display the state of a running Eiffel system at a particular point of its execution, as described in the "Intended Use" section. In other words, at whichever point the execution is interrupted, the Spyglass will show the user the objects that its client passes to it. (This client can be a testing framework, as described in the next section, or a debugger, etc.) In fact, the Spyglass has to display a forest of objects (a series of trees), whose roots are the objects it is passed. These objects can have reference fields leading to other objects, etc. This display must be interactive, so that users can navigate the object structure, and expand and collapse object fields.

Desirable features are the ability to store user preferences about the layout of objects, and heuristics to recognize that an object of a new execution matches an object of the previous execution and should be applied the corresponding stored user preferences.

3. Intended Use (The Client System of the Spyglass)

The Spyglass will be used in a testing framework which relies on contract violations to signal bugs. When a contract is violated, it is important for users to be able to reproduce the bug and to understand its cause.

The testing framework uses the concepts of test case and test driver. A **test case** specifies the test inputs and conditions and contains the calls to the tested routines. (In general, a test case can also specify the expected result, but in our approach this is the purpose of contracts.) A **test driver** runs the test cases on the system under test.

The testing framework provides the following facilities:

- It lets users write their own test code (in test cases).
- It runs the test cases.
- It implements a checkpoint mechanism which records state at various key points in the execution of a system.

- If a contract is violated while running the test cases, the testing framework:
  - Shows the user information regarding the contract violation:
    - What type of contract was violated
    - Where the violation occurred (name of class and routine)
    - The tag of the violated assertion, if any.
  - Shows the user information regarding the location where state was last saved (name of class and routine).
  - Calls the Spyglass to display the last saved state.

Implementation hints (it is not compulsory to follow them)

The state saving facility can be implemented as a class in the testing framework. When a user wants to save the system state, he will insert a call to the corresponding routine of this class in his code. You might want to assume that this call is performed at most once per routine and is inserted as the first instruction (or series of instructions) in the routine body. The minimal information to be saved contains:

- The name of the class and of the routine where state is saved;
- The **`Current`** object;
- The arguments of the routine.

The testing framework will thus build a sequence of snapshots of the system state.

You can use classes to describe the notions of test case and test driver. Class *`TEST_CASE`* can have a deferred routine *`execute`*, so that when users write their own test cases they inherit from this class and implement routine *`execute`* to contain their test code. The test driver can also be implemented as a class which runs all the test cases (instances of subclasses of *`TEST_CASE`*) by calling their *`execute`* routines. The test driver should also take care of wrapping the calls to *`execute`* in **`rescue`** clauses, so that it catches the exceptions resulting from contract violations and can then perform the required actions.

4. Example

Let's suppose a user wants to test a part of a system he has developed for managing employees of a company. Currently he wants to test procedure *`receive_salary`* of class *`EMPLOYEE`*, given in section 7. To do this, he may write class *`RECEIVE_SALARY_TC_1`* (see section 7).

As shown in section 7, in the creation procedure of the root class of the system, the user will create a test driver and an instance of his test case, will add this instance to the list of test cases that the test driver keeps, and then ask the test driver to execute the test cases in the list.

When the test case is executed, the postcondition of function *`tax`* will be violated (taxes are greater than the sum). When this violation occurs, execution stops and the testing framework:

- Displays information regarding the contract violation:
  - The type of assertion (postcondition)
  - It occurred in routine *`tax`* of class *`EMPLOYEE`*
  - Tag: *`tax_less_than_sum`*

- Displays information about the location of the last state saving operation performed:
  - Class: `EMPLOYEE`
  - Routine: `tax`
- Calls the Spyglass passing it the last saved system state.

The Spyglass will display the **`Current`** object (the object on which routine `tax` was called), the arguments that were passed to `tax` (in this case, only value `100` for `sum`) and will allow the user to navigate the reference fields of these objects.

5. Required Tasks

You are asked to develop both the Object Spyglass and its client, the testing framework, as described in sections 2 and 3. The Spyglass should be reusable and easily extendible. It should provide a program interface that any client can use to access its functionality and should be independent of the client implementation.

The way the Spyglass displays objects is left entirely up to you. Also, if you decide to implement this facility, you must find a way to store user preferences with regard to the display and use them in a subsequent run of the system. What you store and how you use it afterwards is for you to decide. Of course, complexity can vary, but your focus must be on making the Spyglass as user-friendly as possible.

Also try to think of optimizations you can bring to the testing framework and Spyglass. For instance, is it really necessary to save the class name when saving the state? Remember that you save the **`Current`** object anyway.

Because several points in the project specification are left open, you must also write the requirements document for the project, once you have decided on the open points. The requirements document, as well as the user guide and developer documentation, will be delivered together with the project code. The requirements document can reuse part of the present text (available at http://se.inf.ethz.ch/teaching/ss2005/0250/project/Project_Specification.pdf).

Deliverables

The project will be implemented in Eiffel. It should compile and run at least under Windows and Unix.

You are asked to deliver:

- The source code of the project – This means **only** the ace file and Eiffel classes (.e files). Do not use absolute paths in the ace file. Do not include any compilation output (compiled code, EiffelStudio project files, etc.) in the delivery.
- A test suite – a set of test cases that demonstrate the functionality of the system (also only as source code)
- Requirements document (in pdf format)
- Documentation (in pdf format):
  - User guide – describes how to use the tool
  - Developer guide – describes the architecture, main classes, limitations, how to extend the tool

6. Project evaluation

The project will be graded with respect to:

1. Correctness
   - Conformance to the specification provided in this document
   - Conformance to the requirements document that you deliver
2. Design
   - Interfaces between modules
   - Extendibility
   - Use of design patterns (if applicable)
3. Quality of contracts
4. Quality of code
   - Easy to understand
   - Style guidelines
5. Testing (delivery of a test suite)
6. Documentation
   - Requirements document
   - User guide
   - Developer guide

7. Appendix: Example code

```
class EMPLOYEE

...

create

      make

feature {NONE} -- Initialization

      make (an_account: ACCOUNT; a_name: STRING; an_age: INTEGER) is
                  -- Creation procedure
            require
                  an_account_not_void: an_account /= Void
                  a_name_not_void: a_name /= Void
                  an_age_positive: an_age > 0
            do
                  salary_account := an_account
                  name := a_name
                  age := an_age
            ensure
                  account_set: salary_account = an_account
                  name_set: name = a_name
                  age_set: age = an_age
            end

feature -- Basic operations

      receive_salary (sum: INTEGER) is
```

```
                -- Deposit `sum' in the employee's account after
deducing taxes.
            require
                sum_positive: sum > 0
            local
                a_list: LIST [ANY]
            do
                    -- Save state.
                create {ARRAYED_LIST [ANY]} a_list.make (0)
                a_list.extend (Current)
                a_list.extend (sum)
                state_manager.save_state             (generating_type,
"receive_salary", a_list)

                salary_account.deposit (sum - tax (sum))
            end

      tax (sum: INTEGER): DOUBLE is
                -- Taxes that the employee pays for `sum'.
            require
                sum_positive: sum > 0
            local
                pension, health_insurance: DOUBLE
                a_list: LIST [ANY]
            do
                    -- Save state.
                create {ARRAYED_LIST [ANY]} a_list.make (0)
                a_list.extend (Current)
                a_list.extend (sum)
                state_manager.save_state    (generating_type,    "tax",
a_list)

                pension := 0.1 * sum
                health_insurance  :=  200  --  Health  insurance  premium
is coded as a constant!
                Result := pension + health_insurance
            ensure
                tax_positive: Result >= 0
                tax_less_than_sum: Result <= sum
            end

feature {NONE} -- Implementation

      salary_account: ACCOUNT
                -- Account where employee receives salary

      name: STRING
                -- Employee name

      age: INTEGER
                -- Employee age

invariant
      salary_account_not_void: salary_account /= Void
      name_not_void: name /= Void
      age_positive: age > 0
```

```
end


class ACCOUNT

create

      make

feature {NONE} -- Initialization

      make (initial_balance: DOUBLE) is
                  -- Creation procedure
            require
                  initial_balance_positive: initial_balance >= 0
            do
                  balance := initial_balance
            ensure
                  initial_balance_set: balance = initial_balance
            end

feature -- Basic operations

      deposit (sum: DOUBLE) is
                  -- Add `sum' to `balance'.
            require
                  sum_positive: sum >= 0
            do
                  balance := balance + sum
            ensure
                  balance_updated: balance = old balance + sum
            end

feature {NONE} -- Implementation

      balance: DOUBLE
                  -- Money in the account

invariant
      balance_positive: balance >= 0

end

class RECEIVE_SALARY_TC_1

inherit

      TF_TEST_CASE

feature -- Basic operations

      execute is
                  -- Implementation of the deferred routine from class
`TF_TEST_CASE'.
            local
                  an_account: ACCOUNT
```

```
                an_employee: EMPLOYEE
            do
                    -- Test case 1
                create an_account.make (0)
                create an_employee.make (an_account, "John Doe", 30)
                an_employee.receive_salary (100)
            end
end

class ROOT_CLASS

create

      make

feature -- Initialization

      make is
                -- Create and call the test driver.
            local
                test_driver: TF_TEST_DRIVER
                tc1: RECEIVE_SALARY_TC_1
            do
                create test_driver.make
                create tc1
                test_driver.add_test_case (tc1)
                test_driver.execute
            end
end
```