Object-Oriented Software Construction

Bertrand Meyer

Lecture 10: Project Presentation

Ilinca Ciupa
Organizational matters

- **Project deadline**: 30 June 2005 (last day of the semester)
- You will work on the project in **groups of 2**
- **Project specification available at:**
  http://se.inf.ethz.ch/teaching/ss2005/0250/project/Project_Specification.pdf
Project development

- Implementation language: Eiffel
- Project should compile and run under Windows and Unix (at least)
Overview

- The project has 2 parts:
  - Object Spyglass
  - Testing framework
- Testing framework is a client of the Spyglass
- Spyglass must:
  - Expose its functionality through an interface
  - Remain independent of client implementation
Object Spyglass (1)

- Main task: display a snapshot of the state of a running Eiffel system at a particular point of its execution

- This snapshot will contain
  - The objects that its client (in this case the testing framework) passes to it
  - Any objects reachable from these
Object Spyglass (2)

- Additional functionality:
  - Allow the user to navigate the object structure by expanding and collapsing object fields
- Desirable features:
  - The ability to store user preferences about the layout of objects
  - Heuristics to apply these preferences to subsequent runs of the system
In the Eiffel method, contracts are:
- Routine preconditions – properties that must hold whenever the routine is called
- Routine postconditions – properties that the routine guarantees when it returns
- Class invariants – properties that the instances of a class satisfy at all “stable” times
- Loop variants and invariants – loop correctness constructs
- check instructions – express the software writer’s conviction that a certain property will be satisfied at certain stages of the computation
- Contracts are expressed with assertions (boolean expressions + old notation)
Testing framework

- Relies on contract violations to signal bugs
- Works based on:
  - **Test cases** – contain:
    - Test inputs
    - Conditions for execution (preconditions of the tested routines)
    - Calls to the tested routines
    - Expected results (assertions)
  - **Test driver** – runs the test cases
- When a contract violation occurs, the Spyglass helps the user understand the cause of the bug by providing a graphical display of the last saved system state. (It is the responsibility of the user to save state by inserting in his code calls to a state saving facility.)
Testing framework - Functionality

- Lets users write test cases
- Runs the test cases
- Provides a state saving facility which allows users to record system state at particular points in the execution
- If a contract is violated when running the test cases, the testing framework:
  - Displays information regarding the violated assertion (where it occurred, type and tag of contract)
  - Displays information regarding the location of the last performed state saving operation
  - Calls the Spyglass to display the last saved system state
class EMPLOYEE
...

feature -- Basic operation
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

State saving

Functionality provided by the testing framework

Routine body
Example – System under test (2)  
(all code written by user)

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
Example – Test case
(also written by user)

class RECEIVE_SALARY_TC_1
inherit TF_TEST_CASE
feature -- Basic operations
execute is
  local
  an_account: ACCOUNT
  an_employee: EMPLOYEE
  do
    create an_account.make (0)
    create an_employee.make (an_account, "John Doe", 30)
  end
  an_employee.receive_salary (100)
end

Deferred class provided by the testing framework

-- Implementation of the deferred routine from -- class `TF_TEST_CASE'.

Postcondition of tax violated!
Example – Root class
(also written by user)

class ROOT_CLASS
create
  make
feature -- Initialization
  make is
    -- Create and call the test driver.
    local
      test_driver: TF_TEST_DRIVER
      tcl: RECEIVE_SALARY_TC_1
    do
      create test_driver.make
      create tcl
      test_driver.add_test_case (tcl)
      test_driver.execute
    end
end

Initiates testing

Provided by the testing framework
Example – what the system (testing framework + Spyglass) should do

- When the postcondition of routine tax is violated, execution of tests stops and the testing framework:
  - Displays info regarding the contract violation:
    - Where it occurred: routine tax of class EMPLOYEE
    - Type of assertion: postcondition
    - Tag: tax_less_than_sum
  - Displays info about location of last performed state saving operation: routine tax of class EMPLOYEE
  - Calls the Spyglass passing it the last saved state
- The Spyglass will display the objects that are passed to it (the object on which tax was called and argument sum) and will allow navigation of the reference fields of these objects.
Required tasks

- Develop:
  - Object Spyglass
  - Testing framework

- You decide on:
  - How you display objects in Spyglass
  - What user preferences regarding the display you store (if any)
  - How you use these preferences
What you must deliver

- **Project source code:**
  - ace file + Eiffel classes (.e files)
  - No compiled code!
  - No absolute paths in the ace file!
- **A test suite** – a set of test cases that demonstrate the functionality of the system (also only as source code!)
- **Requirements document**
- **Documentation:**
  - **User guide** - how to use the tool
  - **Developer guide** - description of the architecture, main classes, limitations, how to extend the tool
- All documents as pdf’s!
Grading criteria

1. Correctness
   - Conformance to the specification provided by us
   - 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
References

- Eiffel
  - “Object-Oriented Software Construction, 2nd edition”, Bertrand Meyer
  - “Eiffel: The Language”, Bertrand Meyer
- Writing requirements specifications
  - IEEE standard: “IEEE Recommended Practice for Software Requirements Specifications” – available on the course web page
- Style guidelines
Tools

- EiffelStudio 5.5
  - Helpful material
    - Recommendations:
      - Use the Vision2 library for developing the Object Spyglass
      - Use class INTERNAL (part of the EiffelBase library) for runtime information about objects
      - Use class EXCEPTIONS (also part of EiffelBase) for information about assertion violations
    - An example for drawing in Vision2 provided on the web page (no obligation to use drawing though)
    - A guided tour of EiffelStudio available on the web page (also in EiffelStudio Help)
End of lecture 10