Università degli Studi di Pisa

Laurea Magistrale in Informatica

Advanced Programming

Final Term Paper

Start Date: 12/01/2015
Submission deadline: 1/02/2015 (send a single PDF file to attardi@di.unipi.it)

Rules:

The paper must be produced personally by the student, signed implicitly via his mail address.

You are allowed to discuss with others the general lines of the problems, provided that each student eventually formulates his own solution. Each student is expected to understand and to be able to explain his solution.

You are allowed to consult documentation from any source, provided that references are mentioned.

It is not considered acceptable:

- to develop code or pseudo-code with others
- to use code written by others
- to let others use someone's code
- to show or to examine the work of other students.

Violation of these rules will result in the cancellation of the exam and a report to the Presidente del Consiglio di Corso di Studio.

For the programming exercises you can choose a programming language among C++, C# and Java.

The paper must:

1. be in a single PDF file, formatted readably (font size ≥ 9pt with suitable margins), of **no more than 10 numbered pages**, including code: for each extra page one point will be subtracted from the score.
2. include **the student name**
3. **provide the solution and the code for each exercise separately**, referring to the code of other exercise if necessary.
4. cite references to literature or Web pages from where information was taken.

Introduction

The goal of the project is to develop a Simple Testing Framework (STF). The framework is to be used to generate and perform unit tests for programs. STF gets input from tables contained in an HTML file. Each table represents a "fixture" for checking the correctness of a program.

For example, the following fixture provides test cases for a program to compute the product of two numbers:

Product		
x	y	result()
float	float	float
7.5	42	315
42	-7.5	-315
28.7846	3.14159	90.4294

The first row tells STF the name of the fixture to use for testing. The second row gives headers for the examples, the third row provides the types of the arguments and result, and the remaining rows provide examples. For instance, the first example in the first table says that the product of 7.5 and 42 is 315.

The test is performed using a *Fixture* like this, which maps the columns in a table to variables and methods and provides a method `check()` to perform the test expressed in a row of a table:

```
public class Product extends ColumnFixture {
    public float x;
    public float y;
    public float result() {
      return x * y;
    }
   public boolean check(Row row) { ... }
  }
```

STF applies the `ColumnFixture` to each row in the table and produces a new table that displays the correct (green) and incorrect (red) outputs, like this:

Product		
x	y	result()
float	float	float
7.5	42	315
42	-7.5	-315
28.7846	3.14159	90.4283

Exercise 1

Design a set of classes suitable to represent the structure of fixture tables. The classes should provide *polymorphic methods* for implementing the framework. In particular design a generic `Fixture` class and its specialization `ColumnFixture`, that implements method `execute(Table table)`, that performs the tests expressed in a table.

Exercise 2

Implement a *recursive descent parser* for HTML files containing fixture tables (with no extra markup) producing a representation with the classes of Exercise 1. Hint: use a tokenizer that returns as individual token objects tags and text elements.

Exercise 3

Implement a code generator that takes a representation of a table from Exercise 2 and generates:

1. the code for a skeleton of a specialized `ColumnFixture` for the testing, with an empty body for method `result()` to be filled by the programmer;
2. the code for performing the tests, taking as input a `Table`, and generating the output in HTML, exploiting `Fixture.execute()`.

The generated code should use *polymorphism* and *should not use Reflection*.

Provide the code generated for the example in the Introduction.

Exercise 4

Extend STF with action fixtures, i.e. fixtures that represent a sequence of invocations. As an example of this, define an action fixture that will test a class for computing the root squares of a set of numbers: a sequence of invocations is used first to provide the values to square, followed by an invocation of `sqrt()` to obtain the final value to be checked.

Action				
start	Accumulator	acc		
call		product	12	12
result	acc	add		
call		product	7	7
result	acc	add		

result		sqrt		
check	13.8924			

More precisely:

- action `start` creates an instance of the named class, to be called with the name in the third column (e.g. `acc`).
- action `call` invokes the method named in the third column on the object named in the second column or a function if this is empty, with the arguments provided in the following columns.
- action `result` is similar to call, except that the first argument of the call will be the result of the previous call.
- action `check` invokes a function or method to check whether the result of the previous call corresponds to the supplied value and the value visualized in the output table should be the original value, colored in green or red depending on the correctness of the test, rather than the value returned performing the test.

Provide the code generated for the example above.

Exercise 5

Explain the so called “visitor design pattern”. Explain how it can be implemented in terms of object-oriented interfaces. In particular explain how it could be applied in the design of the above fixtures. Explain why in languages that provide multiple method dispatch, implementing the visitor pattern is simpler.