Rules:

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 an Integrated Testing Framework (ITF). The framework is to be used to generate and perform unit tests for programs. ITF 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 floating point division:

Division		
numerator	denominator	quotient()
float	float	float
300	3	100
-300	3	-100
4195835	3145729	1.3338196

The first row tells ITF 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 quotient of 300 and 3 is 100.

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 Division extends ColumnFixture {
    public float numerator;
    public float denominator;
    public float quotient() {
      return numerator / denominator;
    }
   public boolean check(Row row) { ... }
  }
```

ITF 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:

Division		
numerator	denominator	quotient()
float	float	float
300	3	100
-300	3	-100
4195835	3145729	1.8383832

Exercise 1

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

Show code using the pattern that adds the property of color gray to all the cells of a table.

Exercise 2

Implement a *recursive descent parser* for HTML files containing (just) fixture tables producing a representation with the classes of Exercise 1.

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 the fixture for the testing, with an empty body to be filled by the programmer;
2. the code for performing the tests and generating the output in HTML.

The generated code should not use Reflection.

Provide the code generated for the example in the Introduction.

Exercise 4

Extend ITF 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 average of a set of numbers: a sequence of invocations is used first to provide the values to average, followed by an invocation of Division to obtain the average.

Action		
start	Accumulator	
call	add	772
call	add	228
call	divide	3
check	average	500

More precisely:

- action `start` creates an instance of the named class
- action `call` invokes a method with the arguments provided in the following columns
- action `check` invokes a method with no argument and checks whether the results corresponds to the supplied value.

Provide the code generated for the example above.

Exercise 5

Describe a possible technique for implementing exceptions, in particular the mechanism that allows unwinding the stack, including performing any necessary cleanup before leaving a frame and returning to the frame where the exception is handled.

Exercise 6 (for students of the Laurea Specialistica)

Extend the syntax of action fixtures in order to allow action `check` to have arguments.