A framework for Web Applications Testing through Object-Oriented approach and XUnit tools

Alessandro Marchetto and Andrea Trentini
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano, Italy
www.dico.unimi.it

WMR 2006

Talk outline

Introduction and WAAT Project

Web Modeling

OTMW

- unit testing
- integration testing
- system testing

Case Study

Conclusions

Motivation and WAAT Project

Motivation:

There is an increasing need for: Web software (more strategic), **quality** and **reliabity**, **reusability**, support for legacy applications, documentation

Goal:

to be able to increase application **quality** through Web applications **analysis and testing**

Briefly:

reverse engineering techniques applied to Web applications to generate/extract models (UML Class and State diagrams); and application **analysis** and **testing** based on extracted model.

Definitions

Web application (WA): set of Web documents (Web pages), Web objects, and server components.

Web document can be: static, active or dynamic.

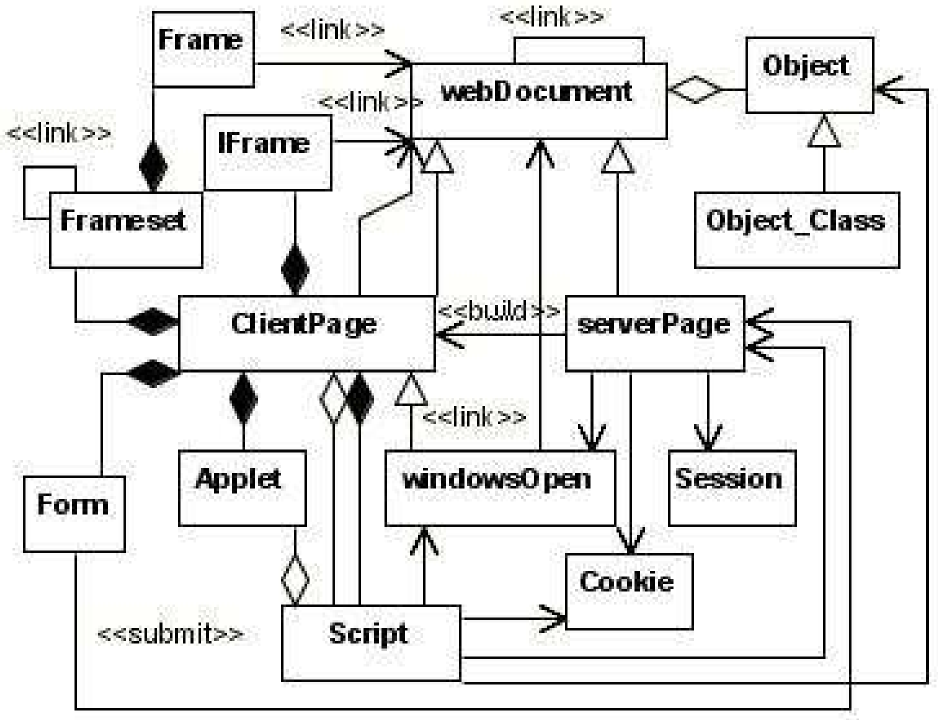
Application analysis are used to extract/generate model from existing Web applications, and can be: static and/or dynamic

Static analysis: traditional source code analysis used to extract software features (e.g., using scanner or parser)
Dynamic analysis: based on software execution, used to extract runtime features. (e.g., capture-replay, user controls, mutation analysis, and so on).

Web Modeling

Class diagrams define application structure (e.g. form,frames,script,...)

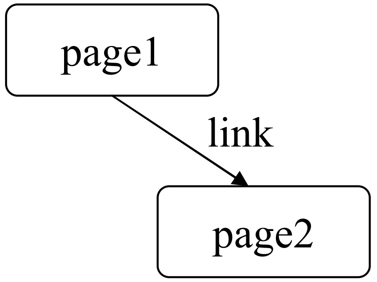
We defined a UML Web meta-model inspired by the models proposed by J.Conallen and F.Ricca-P.Tonella.



(OTMW) forward or reverse engineered models

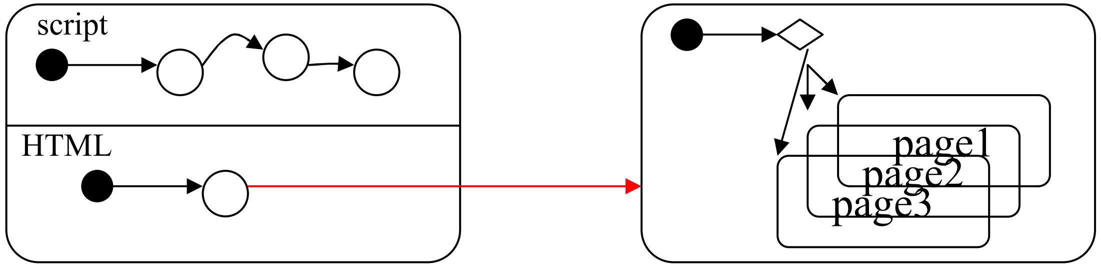
State diagrams represent application behaviour and navigational structure (e.g. pages, links, frames, scripting flow,...).

Static Web documents, e.g., link between pages



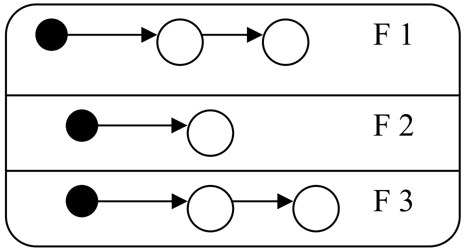
Active Web document, e.g., HTML page with client-side scripting

Dynamic Web document, e.g., server page that builds a set of client-side pages



Client-side page submits data to server-side page, that builds response pages

Active Web document, e.g., HTML page with frames



```
....<form id="F" action="server.asp"....>
<input type="button" id="B"...>
<input type="submit" id="S"....>
</form>......
```

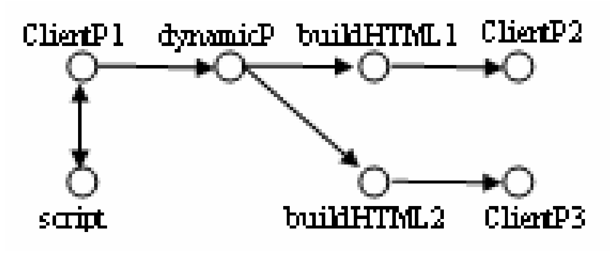
```
<%......request.form("B")....
if B= "" then error = 1
else response.write("Value=" & B)....
....%>
```

Extraction of several kinds of information from UML models

In particular, three different kinds of graph such as:

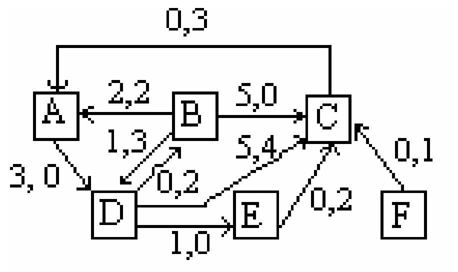
Class Diagram (high level) → Web Graph (WG)

* * nodes = web pages
* * edges = links between web pages
* * (B → A) = “there is a link between B and A pages”
* * used to represent the navigational system of applications



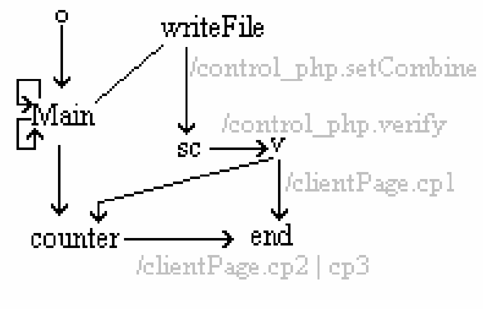
Class Diagram → Web Test Dependence Graph (wTDG)

- * nodes = model classes (pages, components, *etc.*)
- * edges = classes dependencies
- * every edge is weighted with used methods and attributes
- * (B→ A) = B uses A = "B is test dependent on A"
- * used to search the best integration orders



State Diagram → Extended Function Calls Graph (eFCG)

- * nodes = functions, state of objects, pages, components, *etc.*
- * edges = function calls, links, clicks, user gestures, *etc.*
- * used to represent system executions



OTMW -testing framework-

Functional testing inspired by equivalence classes testing (EC) and by the category partition method (CPm).

Steps:

* Unit testing: to verify every software component (client/server side pages, Web objects or components).
We may verify functionality and states of the units.

* Integration testing: to verify integrations/collaborations among components (cluster as a unique unit).
We may verify the messages exchanged among units.

* System testing: to perform the pages-based testing to verify the navigational system and structure of the application (i.e., in terms of sequences of pages).

Unit testing

Verification of:
- elements composing applications (e.g., pages, objects, and so on);
- the structure of every component
- the evolution of every component (in terms of function calls, etc.)
- the dynamically generated pages (e.g., HTML code generated by server pages).

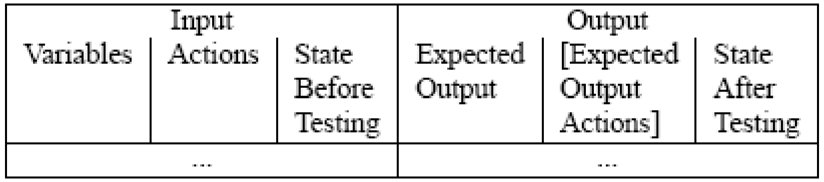
Main steps:
- identification of units to test in isolation (using wTDG of the AUT);
- identification of the needed drivers and stubs (using wTDG);
- test cases definition (using *testing tables* defined through the eFCG of the UUT);
- test scripts definition and tests execution (using XUnit tools);

To identify units we use a set of rules considering the kind of components and of dependencies among them.

Units may be:

- static HTML pages
- client side objects such as the scripting codes
- server side objects
- server pages written in PHP 4/5 and their set of dynamically generated HTML client-side pages
- client-side scripting code generating a set of HTML pages
- Web objects and components
- etc.

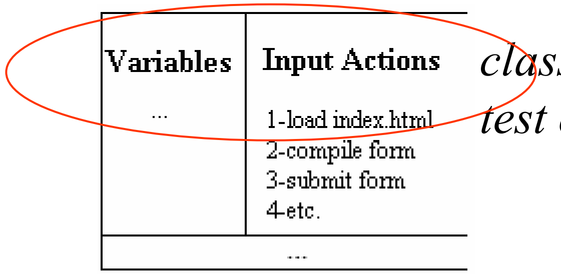
the dependencies type is used to decide if a relationship may be “breakable” (e.g., inheritances and compositions are considered not-breakable) and to define the stubs needed.



Input			Output		
Variables	Actions	State Before Testing	Expected Output	[Expected Output Actions]	State After Testing
...			...		

testing table

For every UUT we build its testing table, it represents the basic information to define test cases through a category partition method.



Variables	Input Actions
...	1-load index.html 2-compile form 3-submit form 4-etc.
...	

classes of test cases

to build a table for a given UUT:
* we identify the unit input variables
* we extract the eFCG (it describes unit executions in terms of function calls and actions performed)

From a given eFCG we may extract paths (=software executions) traversing the graph and using conventional coverage criteria.

Thus, every row of the table represents a class of test cases and may be converted in a set of test cases.

Integration testing

We test data or messages exchanged among units.
Web software may be written through more languages so we need to test interactions among some different kinds of elements (e.g., HTML-JS; JS-JS; HTML-PHP; PHP-PHP, and so on).

Steps:
- we identify sequences of units to test (using wTDG)
- then, we treat every integration cluster (group of units in the sequence) as a single “big-unit”
- we fill the cluster *testing table* (using eFCGs of the cluster units)
- we define and execute the set of test cases

* We define the best integration order using existing OO approach studying components dependencies and scaffolding complexity

* Often, a topological order of the elements composing the class-dependencies graph is defined

* Unfortunately, this graph may contain cycles of dependencies so may be impossible to define a topological order

* Existing deterministic/random strategies “break” dependencies in cycles to obtain acyclic graphs and then to find the best orders (e.g., minimizing number/complexity of needed stubs).

* When we have defined the testing order, we need to test every cluster composing the sequence.

* We use the same methods and tools used for unit testing

  Every cluster considered as a unique "big-unit" (black-box) with an interface composed of the sum of all units interfaces.

  Thus, we define several invocation sequences to stress methods of all units in cluster and verify every state reachable by cluster.

System testing

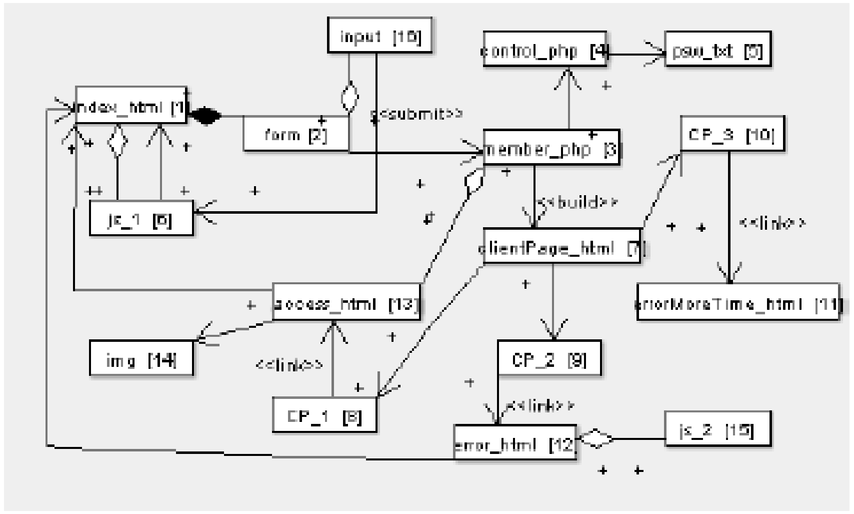
We use the Web Graph representing AUT to verify its navigational system.
A test case is a sequence of URLs requested to Web server with their input values (if needed).

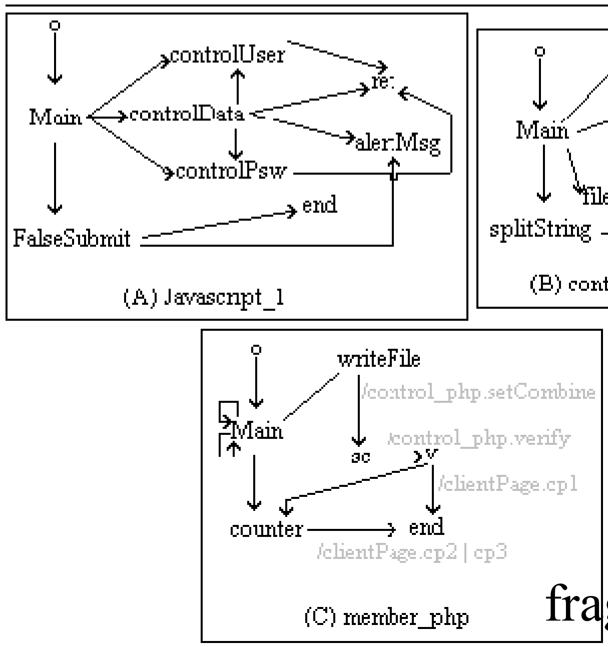
We traverse the AUT simulating user navigations and gestures:
* we extract several paths (URLs- sequences) from the graph through some graph-coverage measures
* these sequences (test cases) are completed with input values (using random, log-based or user manually definition)
* finally, test cases are executed (more time) performing requests to the Web server

Case Study

MiniLogin application:

* it is a simple Web application to control reserved area via login and password
* it is composed of few HTML, PHP and txt files



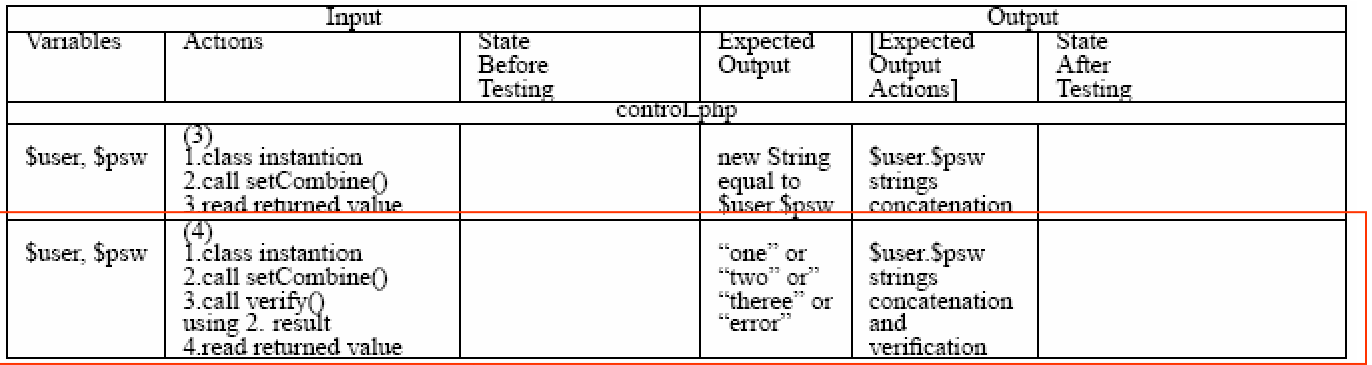


Testing layer:
- units
- integration
- system

fragments of eFCGs for three elements

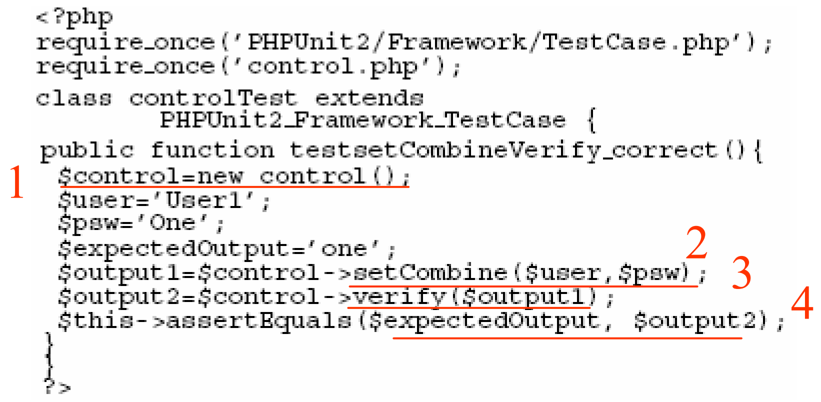
Unit

we identify units for everyone:



Input			Output		
Variables	Actions	State Before Testing	Expected Output	[Expected Output Actions]	State After Testing
control_php					
$user, $psw	(3) 1.class instantion 2.call setCombine() 3.read returned value		new String equal to $user.$psw	$user.$psw strings concatenation	
$user, $psw	(4) 1.class instantion 2.call setCombine() 3.call verify() using 2. result 4.read returned value		“one” or “two” or” “theree” or “error”	$user.$psw strings concatenation and verification	

* we extract methods, their input and used variables
* then (by the state diagram) we build its eFCG and we extract several paths (e.g., *control php* (3): “*Main, setCombine, verify, ret*”)
* we fill the testing its table (every row represents a path)
* we use this testing table to define test cases (rows) and write them using XUnit tools



```
<?php
require_once('PHPUnit2/Framework/TestCase.php');
require_once('control.php');
class controlTest extends
        PHPUnit2_Framework_TestCase {
public function testsetCombineVerify_correct(){
1 $control=new control();
  $user='User1';
  $psw='One';
  $expectedOutput='one';                                 2
  $output1=$control->setCombine($user,$psw);  3
  $output2=$control->verify($output1);               4
  $this->assertEquals($expectedOutput, $output2);
 }
}
?>
```

Integration

for a given AUT:
* we use wTDG to calculate coupling measures and we use a genetic-based tool to find the best integration order
* thus, we need to test every cluster defined in the integration order

for every cluster:
* we use eFCGs of its units to identify sequences of methods & variables used among units to collaborate.
  e.g., (index html.Main & Javacript 1.Main & input.Main & form.Main), formMain.user, input.onMouseOut, Javacript 1.controlUser, formMain.submit, input. onMouseClick, Javacript 1.controlUser; and so on.
* we fill *testing table* for cluster and we use them to define test cases (e.g., we may use HTMLUnit and PHPUnit to write testing script)
* we execute test cases repeating them using several input values.

Input			Output		
Variables	Actions	State Before Testing	Expected Output	[Expected Output Actions]	State After Testing
Cluster 1					

	(7.2)
	1.load index_html
	2.put user string
	3.onMouseOut activation
	4.call controlUsername()
	5.click submit
	6.onMouseClick activation
	7.call controlData()
		def(formMain.user)
		def(formMain.psw)
		def(index_html.ptagUsername)
		def(index_html.ptagPassword)
			ptagUsername==
			'is String'
			+alert(Number Sedning)
				login and password verification	ptagUsername,
					ptagUsername==
					"is String" and
					'ptagPassword==
					undef

Cluster 2					

$username
$password
	(8)
	1.load index_html
	2.put real username
	3.put real password
	4.click submit
	5.load access_html
	6.it contains gif and link
	7.click link
		def(formMain.user)
		def(ptagPassword)
			access OK +
			load access_html +
				username and password	

...					

- HTML page with Javascript

```
import com.gargoylesoftware.htmlunit.*;
import java.net.URL;
import com.gargoylesoftware.htmlunit.html.*;
import junit.framework.TestCase;
import java.util.*;
public class SimpleHtmlUnitTest extends junit.framework.TestCase {
public void testHomePage1() throws Exception {
 WebClient webClient = new WebClient();
 java.net.URL url = new  java.net.URL('http://localhost:8080/alex/logred2/index.html');   1
 HtmlPage page = (HtmlPage) webClient.getPage(url);
 assertEquals('Home Page', page.getTitleText());
 HtmlForm form = page.getFormByName('formMain');
 HtmlTextInput textField=(HtmlTextInput)form.getInputByName('username');
 textField.setValueAttribute('prova'); 2
  HtmlPage appWindow=(HtmlPage) page.executeJavaScriptIfPossible( 3,4
          textField.getOnMouseOutAttribute(),'testCU',false,textField).getNewPage();
  assertEquals('Home Page', appWindow.getTitleText());
  assertEquals('Username is String',
          appWindow.getHtmlElementById('ptagUsername').getFirstChild().asText() );
  HtmlSubmitInput button = (HtmlSubmitInput)form.getInputByName('Submit');
  List collectedAlerts = new ArrayList();
  webClient.setAlertHandler( new CollectingAlertHandler(collectedAlerts) );
  HtmlPage newPage = (HtmlPage)button.click();     5,6,7
        List expectedAlerts = Collections.singletonList('Number sending');  [or 'Numbero sending']
  assertEquals( expectedAlerts, collectedAlerts );
}
}
```

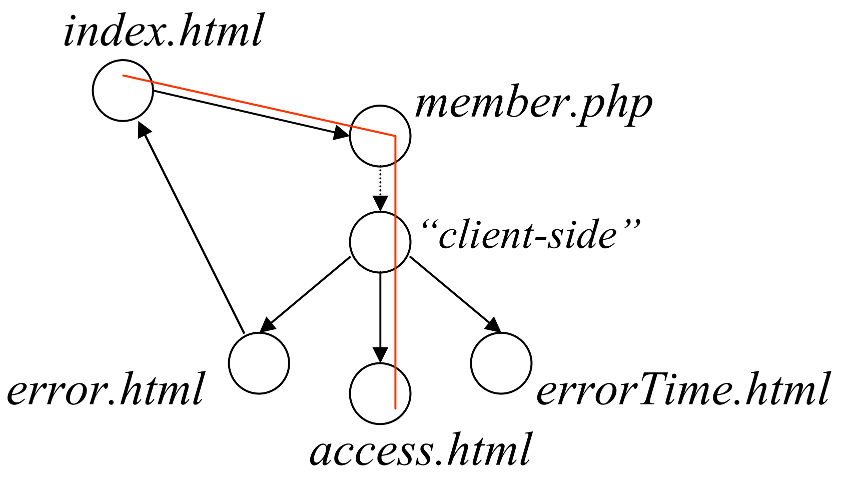
System

* A test case is a sequence of URLs and input values:

< page to load, [list of parameters values] >

* Possible test cases:

- (index.html), (member.php);
- ~~(index.html), (member.php, "*username*", "*password*"), (access.html);~~
- (member.php, "*user1*", "*psw1*"), (errorTime.html);
- (member.php,"*user2*", "*psw2*"), (error.html), (index.html);
- and so on.



Conclusions

In this presentation we have introduced our OTMW model
It may be usable to test WA through an OO approach and based on UML models and functional testing (i.e., category partition).

The future...
Complete the implementation of framework, currently there are:
- our WebUml (reverse engineering)
- our TestUml (system testing)
- our wJenInt (genetic-based integration orders)
- existing XUnit tools

So, we need to implement:
- eFCGs extraction tool
- a tool to help testers to fill the *testing tables*

Comments or questions

marchetto@dico.unimi.it