Test Execution Document

College Mart



23th Nov 2016

Software Engineering Project

NIIT UNIVERSITY

Course Software Engineering Project

Project Group

Anay yadav

Naved Ahamd

Suryakant Upadhyay

Customer NIIT UNIVERSITY STUDENTS

Project Masters

Amit Kumar (Supervisor)

Amit Kumar (Instructor)

Homepage

Change Log

Version Date Modifications

23/11/16 First version

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | i | |
| Contents | |  | |
| 1 | **Introduction** | 1 | |
| 2 | **Unit testing** | 1 | |
| 3 | **Integration testing** | 2 | |
|  |  |  | |
|  |  |  | |
|  |  |  | |
| **4** | **System testing** | 3 | |
|  |  |  | |
|  |  |  | |
|  |  |  | |
|  |  |  | |
|  |  |  | |
|  |  |  | |
|  |  |  | |
|  |  |  | |
|  |  |  | |
|  |  |  |
| **5 Found and bugs** | | 8 | |
| **References** | | 11 | |

1

1. **Introduction**

This is the Test Execution Document of the College Mart Software Engineering project at the Computer Science Department of NIIT UNIVERSITY. The main aim is to achieve use of reusable items and provide students with books and utilities at a lesser price as compared to the market price ,and all this is done through a proper frontend which is serviced by an efficient server at the backend

The document is organized as follows:

* Section 2 introduces the results of unit testing.
* Section 3 introduces the results of integration testing.
* Section 4 introduces the results of system testing.
* In section 5 the bugs found during test phase are listed.

The results of the test execution will be presented in subsection as follows:

* Id and name of the executed test.
* Description of the test, or collected results.

Errors found if any, and maybe a reference to the bug and fixes report. If error was found on certain step of Test Case, the step number is given.

All test cases were first executed once and found bugs were fixed. Then all test cases were executed again and no further bugs were found.

Test were run on following environment: Intel I5 2.0GHz processor,and Java 2 Runtime Environment.

1. **Unit testing**

**Introduction**

Unit testing, sometimes referred to as developer testing, focuses on testing small pieces of code, such as a class, that a developer is writing. These tests are critical for helping us ensure that the pieces we build work as expected and will operate correctly when combined with other parts of the application. Such testing helps support management of the application over time by ensuring that changes we make don't inadvertently affect other parts of the system.

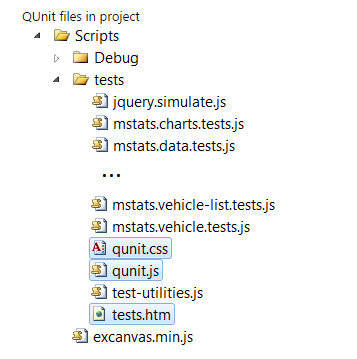
Unit testing of the client-side portion of a web application present different challenges than server-side testing presents. In addition to testing the structural layout of a page and basic application functionality, we may want to verify that animations execute properly, that a page with a large amount of JavaScript has no memory leaks, and that the application maintains its functional and performance expectations across multiple browsers.

Unit testing allows us to verify that individual pieces work as we expect and provides a way for us to verify that they continue to work as libraries or tools evolve. For example, we may build a jQuery UI widget to manage a piece of our UI. When the next version of jQuery is released, we can quickly and easily verify that our widget is still working by executing the unit tests using the new jQuery libraries.

Getting Started with Unit Testing

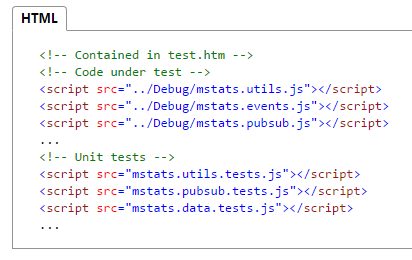
We decided to use the QUnit unit testing framework to do unit testing of our JavaScript components, which rely heavily on jQuery and the jQuery UI widget framework, which also use QUnit. The QUnit unit testing framework can be found on the jQuery website at <http://docs.jquery.com/QUnit>.

To set up QUnit, we will typically create an HTML page containing specific QUnit elements with certain class attributes specified, and add and reference the **qunit.js** and **qunit.css** files. In Mileage Stats, these were added to the tests folder under the Scripts folder.

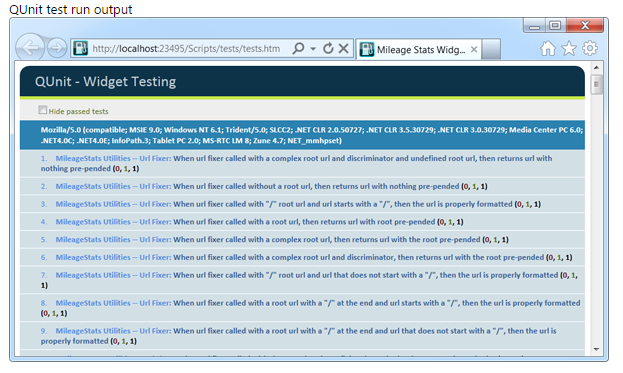


Once this structure is in place, we will create a test JavaScript file for each set of tests we want to run. A set is typically focused on a specific JavaScript object. For example, in Mileage Stats there is a JavaScript test file for each of the jQuery UI widgets that the application implements.

Each of these JavaScript test files and the JavaScript file of the item being tested are referenced from the test.htm file so the QUnit framework can locate and execute the tests.

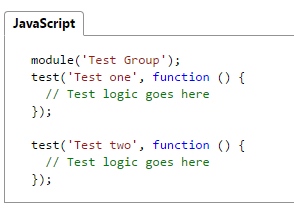


These unit tests can be run by viewing the test HTML file in a browser, we can right-click the test HTML file and select **View in Browser.** The output would look like this while executing the tests.

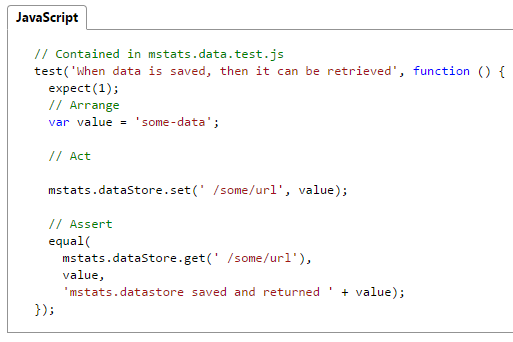


**Creating Unit Tests**

There are typically multiple unit tests in one file and they are often grouped around a particular topic or type of test. In QUnit, the **module** function is used to denote a group of tests.



Looking at the structure of a test. This is a test from mstats.data.test.js to test a data caching component within the solution.

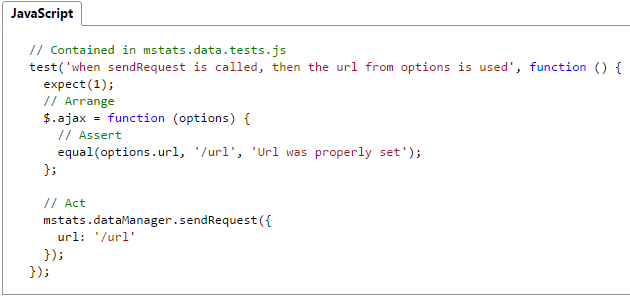


Most unit tests followed arrange-act-assert pattern, a common test pattern employed when unit testing. Its purpose is to make a clear distinction between the set up for the test, the action that is to be tested, and the evaluation of the results.

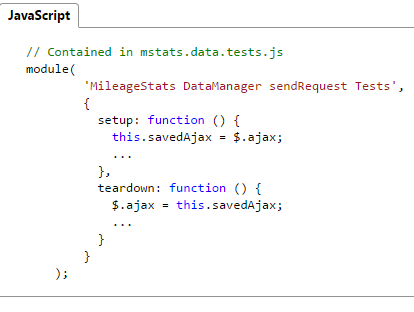
In the unit test we tried to verify the functionality of a relatively small component, such as a JavaScript object or a jQuery UI widget. Each test verified independent pieces such as whether a calculation happened correctly or whether the proper document object model (DOM) modification occurred.

**Isolating Tests**

When testing, we also wanted to isolate our component from other objects we built within our system. In Mileage Stats, many jQuery UI widgets relied on a publish-subscribe object for communication. When we tested objects with dependencies, we did not invoke the actual dependencies. If we did, we would be testing more than one thing at a time. Instead, it is important to test that the component which attempts to invoke a dependency. The typical strategy for isolating our component under test was to supply an alternative component or function that the component calls instead of the real component. These alternatives may be referred to as fakes, doubles, stubs, or mocks. As it turns out, the ability to isolate our component in this manner also helped the overall design of our application because we tend to create smaller, more focused components.



When providing these alternative functions or components, it is also imperative to capture and restore the original values to avoid interfering with any other test that may have relied on these values. In QUnit this can be done when defining the test module where the **setup** and **teardown** functions can be supplied.

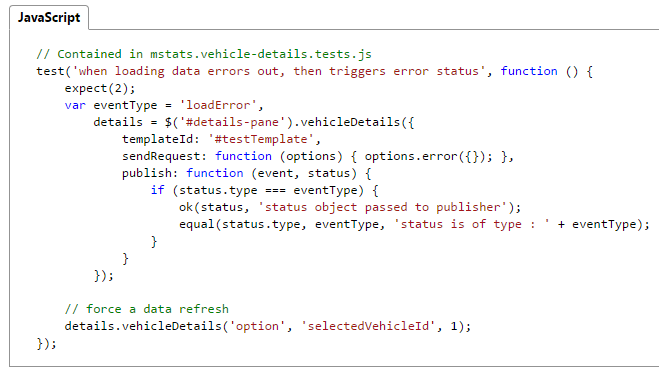


**jQuery UI Widget Testing**

When unit testing jQuery UI widgets, there are some additional considerations. Since a widget is attached to a DOM element, we will need to create these elements either in the test or, if they are more complicated, in the setup for a module. In Mileage Stats, because many of the widgets interact with a section of the DOM, some of that structure needs to be created during the test setup. For example, the setup for the header widget test recreates that portion of the DOM structure it manipulates.



When testing jQuery UI widgets we will also often need to supply an alternate implementation of dependent functions or objects. Because we don't control the creation of the jQuery UI widgets directly, we will typically do this as part of the **options** object passed into the widget.

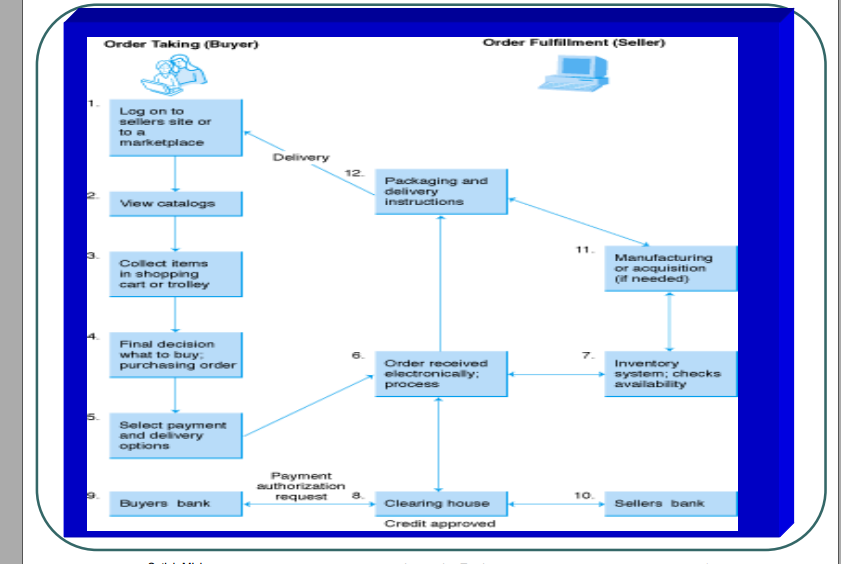


1. **Integration testing**

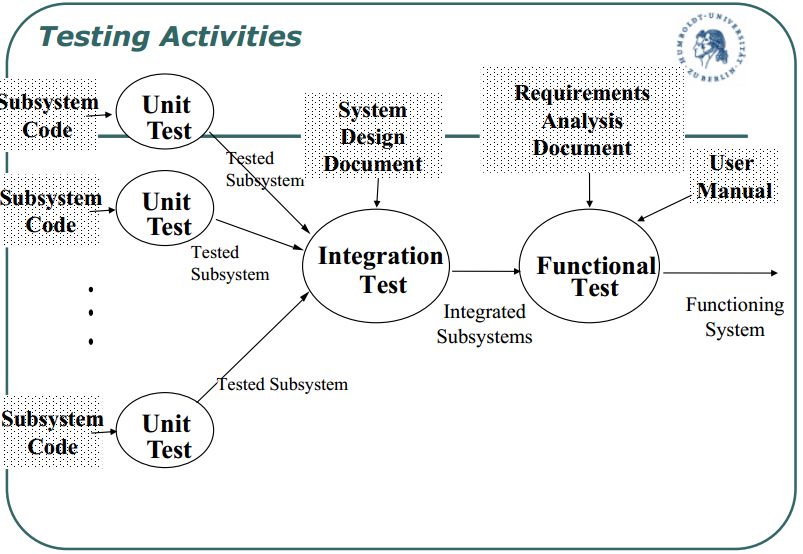
1.Testing in which software components, hardware  
components, or both together are combined and tested  
to evaluate interactions between them

2. Integration testing usually go through several realword  
business scenarios to see whether the system can  
successfully complete workflow tasks

3. Integration plan specifies the order of combining the  
modules into partial systems



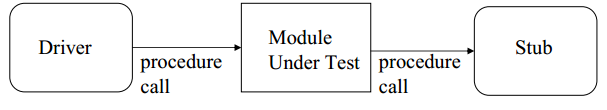
1. A unit is the smallest testable piece of software. A unit  
   is defined as a database trigger, stored procedure,  
   function, report, form, batch load program or PL/SQL  
   program.
2. Module /Component consists of units integrated into a  
   module that performs a specific business function



**Driver:** A program that calls the interface procedures  
of the module being tested and reports the results  
A driver simulates a module that calls the module currently being tested

**Stub:** A program that has the same interface  
procedures as a module that is being called by the  
module being tested but is simpler.

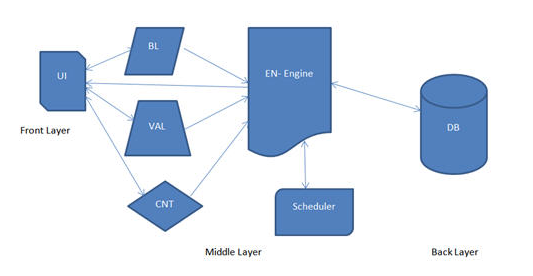
A stub simulates a module called by the module currently being  
tested



### ****Integration testing GUI application****

The integration testing in Black box technique.

We all understand that a web application is a multitier application. We have a front end which is visible to the user, we have a middle layer which has business logic, we have some more middle layer which does some validations, integrate some third party APIs etc., then we have the back layer which is the data base.



**UI** – User Interface module, which is visible to the end user, where all the inputs are given.  
**BL** – Is the Business Logic module, which has all the all the calculations and business specific methods.  
**VAL** – Is the Validation module, which has all the validations of the correctness of the input.  
**CNT** – Is the content module which has all the static contents, specific to the inputs entered by the user. These contents are displayed in the reports.  
**EN** – Is the Engine module, this module reads all the data that comes from BL, VAL and CNT module and extracts the SQL query and triggers it to the database.  
**Scheduler** – Is a module which schedules all the reports based on the user selection (monthly, quarterly, semiannually & annually)  
**DB** – Is the Database.

1. **System testing**

This section contains the largest part of testing College Mart testing. It contains all those tests that are needed to verify that all subsystems of the College Mart Web app work correctly

**Test Cases**

1. **Login/sign-up**

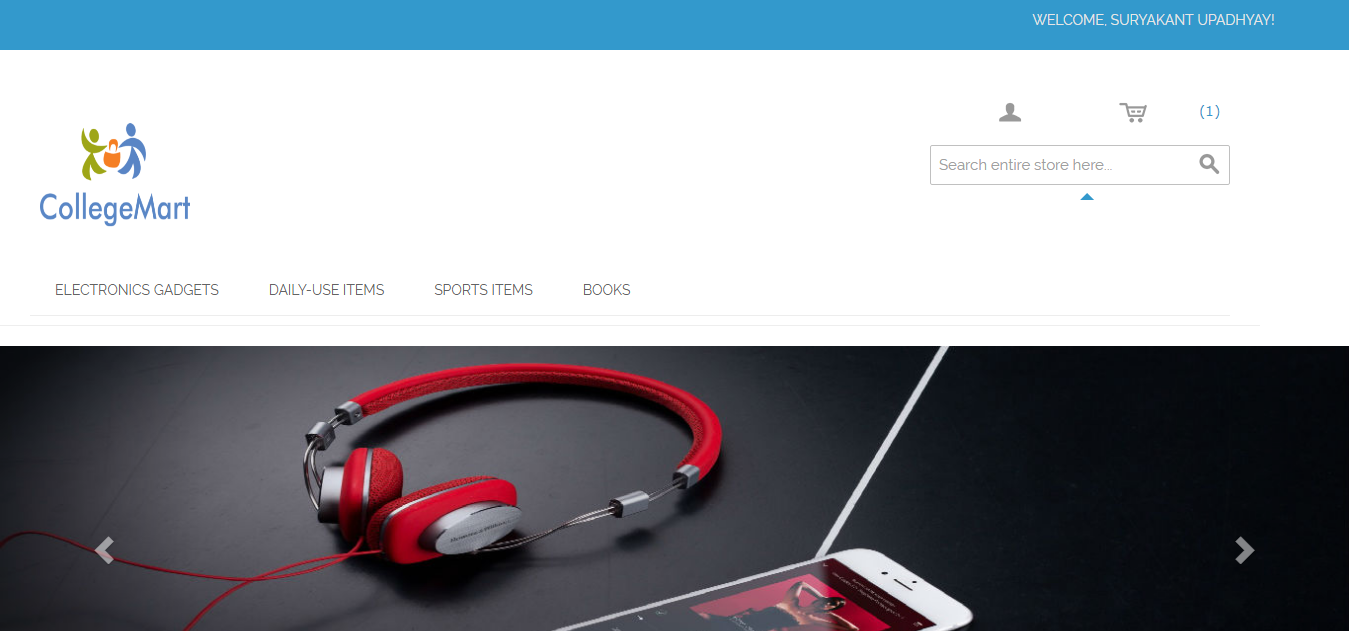
New user must sign-up to use College Mart by giving some necessary information.

* First name
* Last name
* E-mail Id

The registered user can use by just log-in.

ERROR FOUND(bug-1): Unable to validate the password properly. Later fixed using implementations of regular expression for the password.

1. **Home page**

****

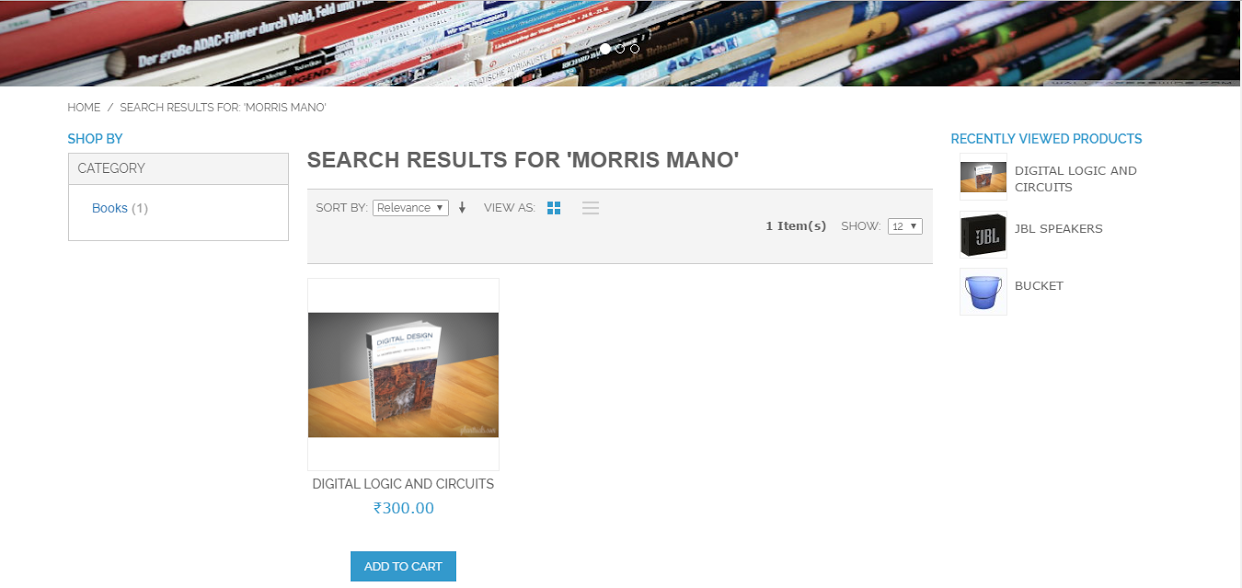
DESCRIPTION: Home page of the College Mart is busy. We have a lot going on but almost all of them are of product images like books, speakers etc.

Some special features

* Auto scroll
* The image will refreshed after 10 sec.
* When the user hovers over it, it will stop scrolling
* It is clickable
* It will take user to the right page and right deal.
* It renders the same way in different browser and different screen resolutions.

ERROS FOUND (bug-2): Problem was occurring with the carousal. Later fix by changing the delay.

1. **Search**

****

Search a very important part for the College Mart because the user can search any items by just entering the meaningful name.

* Search based on Product name, category. For example Camera, electronics.
* Search results are relevant.

Error Found (bug-3): By putting un-meaningful name in search will result some other items. Later we fixed this by some meaningful keywords like books, mobiles etc.

1. **Product details page**

Once the user finds a product either through search or by browsing or by clicking on it from the homepage, the user will be taken to the product details page.

* Image or images of the product.
* Price of the product.
* Product specifications.
* Contact information of the owner and owner’s room no with hostel name.
* Color and dimensions etc. will be shown.

ERROR FOUND (bug-4) Unable to retrieve the item’s complete information. Later we changed in coding and fixed it.

1. **Add Cart**

This is the penultimate stage before the user commits to purchase it later.

* Add items to the cart and continue shopping.
* A user can add more items to the cart.
* Remove the items from the cart.

ERROR FOUND (bug-5) If the user add the same item twice then nothing will happen. Later we fixed by showing a pop-up which shows that you have already added this item in your wish list.

1. **Add Items**

The user can add new item to sell by providing some necessary information about his items.

* The User must upload the photos of their item.
* User must upload necessary information like dimensions, color.
* User must add his information like hostel name and room no.
* The user must provide the specifications of their product.

ERROR FOUND (bug-6): The implementation was bug free.

1. **Delete Items**

It is the due duty of the users to delete their items when the item is being sold by just clicking on delete items.

ERROR FOUND(bug-6): The implementation was bug free.

Found and fixed bugs

In this section, the bugs we found during the first execution of test cases are listed. Also the fixes for bugs are listed. All bugs listed here are fixed.

BUG 1

Description: Problem was occurring with the carousal. Later fix by changing the delay.

BUG 2

Description: Problem was occurring with the carousal. Later fix by changing the delay.

BUG 3

Description: By putting un-meaningful name in search will result some other items. Later we fixed this by some meaningful keywords like books, mobiles etc.

9

BUG 4

Description: Unable to retrieve the item’s complete information. Later we changed in coding and fixed it.

BUG 5

Description: If the user add the same item twice then nothing will happen. Later we fixed by showing a pop-up which shows that you have already added this item in your wish list.

BUG 6

Description: The implementation was bug free.

**References**

* *[Kaner, Cem](https://en.wikipedia.org/wiki/Cem_Kaner" \o "Cem Kaner)*(November 17, 2006).[*"Exploratory Testing"*](http://www.kaner.com/pdfs/ETatQAI.pdf)(PDF). Florida Institute of Technology,Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL*. Retrieved November 22, 2014*.
* [Jump up^](https://en.wikipedia.org/wiki/Software_testing#cite_ref-2) [Software Testing](http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/) by Jiantao Pan, Carnegie Mellon University
* [Jump up^](https://en.wikipedia.org/wiki/Software_testing#cite_ref-3) Leitner, A., Ciupa, I., Oriol, M., Meyer, B., Fiva, A., ["Contract Driven Development = Test Driven Development – Writing Test Cases"](http://se.inf.ethz.ch/people/leitner/publications/cdd_leitner_esec_fse_2007.pdf), Proceedings of ESEC/FSE'07: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering 2007, (Dubrovnik, Croatia), September 2007
* ^ [Jump up to:*a*](https://en.wikipedia.org/wiki/Software_testing#cite_ref-Kaner2_4-0) *[Kaner, Cem](https://en.wikipedia.org/wiki/Cem_Kaner" \o "Cem Kaner)*; Falk, Jack; Nguyen, Hung Quoc (1999).Testing Computer Software, 2nd Ed. New York, et al: John Wiley and Sons, Inc. p. 480.[*ISBN*](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [*0-471-35846-0*](https://en.wikipedia.org/wiki/Special:BookSources/0-471-35846-0).
* [Jump up^](https://en.wikipedia.org/wiki/Software_testing#cite_ref-kolawa_5-0) Kolawa, Adam; Huizinga, Dorota (2007).[*Automated Defect Prevention: Best Practices in Software Management*](http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html). Wiley-IEEE Computer Society Press. pp. 41–43.[*ISBN*](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [*0-470-04212-5*](https://en.wikipedia.org/wiki/Special:BookSources/0-470-04212-5).