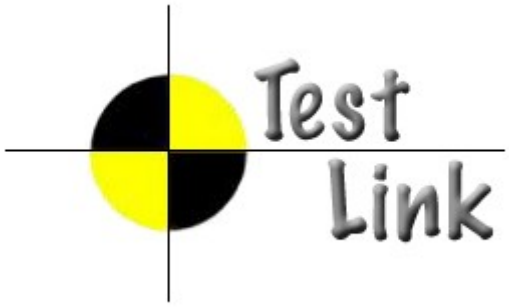


TestLink Developer's Guide

Martin Havlát, Andreas Morsing and Francisco Mancardi

Version: 1.16

1. General

TestLink is popular Open source Test management tool.

1.1. Purpose

This document describes project standards and practices for development, testing and publishing.

1.2. TestLink project target

The project is primary a tool for effective, centralized and automated verification and validation activities. High configurability should help to fit practices in particular companies.

1.3. TestLink Community goals

Testlink developers and supporters are open community. All work on project should be done with respect to other members. The next main requirements should help to be a world class project, enjoy the work and save our time.

Usability

The default interface for user must be maximally simple. High configurability of behavior should allow to reveal advance and customized functions.

Terminology and logic must conform standards. English language follows ISTB terminology.

Maintenability

Refactorization is a part of development cycle. New features should start with a review of a design document. Code should be readable and traceable via tracker report ID.

Effectivity

Project processes should help and not bore developers. On the other hand team should accept agreed rules to coordinate activities and project direction. These rules could support quality with respect to future work.

Solid documentation should brings better feedback from users.

Table of Contents

2. Project Processes

2.1. Team organization

"Testlink community" owns all rights of the project.

Roles within the team:

1. Team leaders (define goals, coordinate significant changes in project, plan a release content, approve members)

   The current core team: Francisco Mancardi, Martin Havlát and Andreas Morsing

2. Developers (has access to CVS; implement new functionality and do bug fixing)

3. Contributors (hasn't access to CVS; the code should be reviewed before add to repository). They contribute via tracker attachments.

4. Build manager (responsible for builds of course)

5. Infrastructure support (BTS, homepage, forum, demo)

6. Testers (coordined regular testing before announce of release, development of test automation)

2.2. Bug tracking policy

Community uses "Mantis BT" tool for work evidence. Each developer must trace changes by issue ID in CVS commit comment. Tracker doesn't allow anonymous access.

Each bug or new feature request should have the next phases:

- **New** - issue is reported with this initial status.
- **Feedback** - issue is not clear; reporter is requested to deliver clarification.
- **Acknowledged** – reporter is informed, that we accepted his report. Feature request kind should have target version (developer assignement is optional). A bug report must have assigned developer in this state.
- Confirmed – not used.
- **Assigned** – developer works on the issue. Developer assignment is mandatory.
- **Solved** – work is done.
- **Closed** – RC or official release was announced with solved the issue.

New features request must have severity "Feature request".

Tracker is used for generated change log (stored in testlink root directory).

~~1. Each main version has own project within BTS (for example 1.6, 1.7).~~

2.3. Contributions review

Reviewer could be assigned to a new contributor. Reviewer helps acquaint you with TestLink's development group processes, tools, and direction. Somebody who will make sure your contributions are aligned with the overall goals of the core team. Reviewer primarily review and commit his work. Reviewer suggest/agree/disagree contributor take an exact work. Reviewer could decide, this guy can obtain CVS access.

Rule: one contributor - one reviewer. So contributor knows who can contact at the first. Contributor of course (could) communicate with all. But reviewer keep in mind, that his responsibility is to to answer questions or suggestion.

2.4. Release policy

For TestLink there are mainly three phases with the next release categories: *beta, release candidate*, official major *release* and *hot-fix release.* The main policy is release often, so TestLink gets early feedback by users. One release per months is nice plan.

2.4.1. Types of Builds

Beta release - Main planned updates are done, when a first Beta is released. Minor enhancement is still ok. Users are welcome to report problems to tracker. For example: "TestLink 1.8 Beta1". No support for automated DB migration from Beta to RC and final version.

Release Candidate (RC) - focus is on bug fixing, exceptional minor enhancement is acceptable. Name should be for example: "TestLink 1.8 **RC2**". DB schema should not change, because it brings addition work for installer. But it could happens in reasonable cases. We support automated DB schema migration from previous version to the current RC and migration from the current RC to later RC or final release.

Official major release – is successfull RC. A hot-fix branch is created in repository. For example: "TestLink **1.8.0**". DB schema migration from the previous major versions (include hot-fixes) must be supported.

Hot-fixes - Bug fixing and localization is acceptable changes only. No development and DB schema are accepted. For example: "TestLink 1.8.**3**".

2.5. CVS Branching

New features development is executed on main branch HEAD only. A new branch is created for each official main release (1.x) to allow a bug fixing. A branch label example: "branch_testlink_1_6".

- All bug fixes with major (and higher) severity should be fixed on both HEAD and the latest maintenance branch.
- No database changes and official development are allowed on branch, because of maintenance effort increases too much.
- All development of new features is acceptable on HEAD. Do all development under

related number in our BTS (add number to commit comment; optionally to code comments).

- DB changes must be acknoledged by DB schema owner. Owner is responsible to verify that installation or migration for all supported types work properly before a release. Changes must be added for both testlink_create_tables.sql and alter tables (minimally to MySQL type) during development phase. Release candidates (and later) is supposed to have all supported DB types data current.

  Francisco is appointed to be DB schema owner.

2.6. Build Process

This chapter describes steps requires for build process. The process should have one or more rounds with dependence on results of preliminary testing. The list of task:

1. Notify about a planned build about a week before date.
2. Remind developers a day before (Ask if some unfihished work is must have).
3. Run testing scripts (Selenium + phpunit).
4. Update `README`. Especially section release notes.
5. Update `CHANGELOG` file with data generated by our bug tracker.
6. Verify/update TL version in const.inc.php.
7. Generate pdf/html files for Installation manual and User Guide. Generate phpdoc update. Upload into cvs:`<testlink_root>/documentation` Public it to homepage.
8. Process build:
   - Tag files in CVS:

     ```
     # cvs tag testlink_1_6_RC1
     ```

   - Export CVS:

     ```
     # cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/testlink login
     # cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/testlink export -r testlink_1_5_1 testlink
     ```

   - Update phpdoc - generate new documentation by phpDocumentator.
   - Build `testlink_<version>.zip` and `testlink_<version>.tar.gz` packages from tagged files. Do cvs export.
9. Load packages to SF (s`ftp://frs.sourceforge.net/uploads/`).
10. Bug tracker:
    - Add a new version title to Mantis.
    - Set the curent release as closed and correct release date.

- Move resolved Mantis issues to status "closed".

11. Update TL version in `const.inc.php` to a next version.

12. Publish the information about build (include download and documentation links). Official release should be also announced in public sites about testing.

    - Community email (also announce the next target release).
    - SF news
    - TestLink homepage news http://www.teamst.org/index.php?option=com_content&task=new&sectionid=1&Itemid=0,
    - http://opensourcetesting.org/,
    - http://freshmeat.net

13. Update roadmap info if relevant (main release). Edit http://www.teamst.org/index.php?option=com_content&task=view&id=6&Itemid=2.

3. Coding conventions

This chapter describes the coding conventions which should be confirmed by each developer of TestLink. Formal code description must be compliant with phpDocumentator standard.

3.1. Coding rules

There are some rules which all developers should follow

- Write your **code for humans** and not for computers!
- Don't re-invent the wheel! Take a look at already **existing code at first**![1]
- Initialize each variable, before using them!
- If you are not sure about the existence of indices within associative array, use isset to test.
- Avoid scripts with huge size, **use functions** instead (and put them into function only files)
- Avoid functions of huge size, use small to medium sized functions only
- Avoid magic numbers – define constants; so value will be readable
- Avoid unnecessary comments!
- Don't use fixed paths for accessing files
- Limit your access to $_GET, $_POST, $_FILES and similiar things, to one place in your script (preferrable at the beginning)!
- Don't access $_GET, $_POST, $_FILES directly, use wrappers instead!
- Document significant code changes! That means, at least **document your code change** in the change log. You may also drop a short mail to other developers (maybe the initial author), so the documentation can be also be updated (if relevant)
- Use TAB (instead of space characters)!

  Eclipse:

  Preferences->General->Editor->TextEditors->DisplayedTabWidth = 4
  PhpEclipse->PHP->Formatter->Style->Number of spaces ... = 4

- Set *error_reporting* to **E_ALL** and set *TL_LOG_LEVEL_DEFAULT* to **EXTENDED** to get the maximum error messages while developing!
- Use the shortest possible meaningful names for variables and functions!
- Don't bloat the code with debug output, unnecessary comments, unnecessary blank lines and so on. Use tLog function to get states and values info.

1 Prefer code with GPL license.

- Don't ignore bad code! If it's safe to refactor it, just refactor![2]
- Don't duplicate code! No copy/paste.

3.2. Code formatting

Always use braces and indent your code in the right way.

Condition statement should be enclosed by brackets (except one line body). Indent an inner code by tabulator.

```
if ($showFeature == 'editTc')
{
      // show edit test cases
      $treeframePath = 'lib/testcases/listTestCases.php?feature=tcEdit';
      $workframePath = 'gui/instructions/tcEdit.html';
}
```

3.3. Classes

Class definition should confirm the next rules:

- Class name should begin with acronym "tl" to differ from third party classes and feature name. For example "`tlPlatform.class.php`".

  This rule was introduced for 1.9 version. For the reason older classes should not fit this rule until refactoring.

- file-name must conform form: `<class_name>.class.php` . PHP "autoload" feature requires the same file and class title.
- Description must include @package definition and short description:

```
/**
 * TestLink wrapper for ADODB component
 * @package        TestLink
 */
class database
{
```

3.4. Functions

Function follows the formatting according the example below. Each function must have a description (see Error: Reference source not found).

```
/**
```

2 You should also add /** @TODO note */.

```
 * Function verifies if login exists
 *
 * @param string $login
 * @return integer number of founded records; i.e. 1 if account exists
 */
function existLogin($login)
{
  $sql = "SELECT * FROM user,rights WHERE user.rightsid = rights.id AND login='" .
        mysql_escape_string($login) . "'";

  $result = mysql_query($sql);
  $userExists = 0;
  if ($result)
  {
    if ($row = mysql_fetch_row($result))
      $userExists = $row;
  }

  return $userExists;
}
```

A function must return the same variable type. As seen in the example the function returns a number (0) or an array, which are two different types, so return null instead of 0.

3.5. SQL coding convention

The next point are mandatory for TestLink SQL requests and files (except a topic is marked as recommendation).

- Make SQL reserved words UPPERCASE.
- Make column and constraint names lower-case. Use underscore to connect terms if a name cannot be described by one term.
- The next abbreviations are used for field names:
  - <object>**_id** – refers to Identifier field of another table. For example testcase_id.
  - <action>**_ts** – time-stamp of actions. For example: create_ts, update_ts, etc.
- Enclose each table name with `{$tables['<table_name>']}` (this allows TestLink to prefix table names).[3]

```
SELECT id,name FROM {$tables['nodes_hierarchy']} WHERE ...
```

- Indent code to improve readability.
- Use single-line comment markers(–). Except a file header. Reserve multi-line comments (/*.. ..*/) for blocking out sections of code.

3 Since 1.9 version

- Use one blank line to separate code sections.
- Recommendation: Index names should begin with the name of the table they depend on, for example: `INDEX users_sid_idx`.
- Recommendation: use symbolic names via 'AS' if SQL request is difficult.

3.5.1. MySQL specific convention

The next parameter types are used:

- `int(10)` – integer with size 10 is default.
- `tinyint(1)` – boolean.

3.6. File naming conventions

This section describes the file naming conventions used in TestLink.

- Use **Camel case** or all lower case as default convention for all file-names. Underscore and other special characters are not acceptable.

  Warning: Unix/Linux file system differs upper and lower case.

- The name of a **Smarty Template** called/launched from a PHP page called xyz.php must be xyz.tpl, i.e. the file-names without considering the extension must be identical.
- Php include files should be named `<title>.inc.php`.
- Configuration php files should follows `<title>.cfg.php`.
- Class definition in php is indicated by file-name `<title>.class.php` . Autoload feature requires the same file and class title. For example "`tlPlatform.class.php`" file should contain "`tlPratform`" class.
- Separated parts of Smarty templates should be named `inc_<title>.tpl` .

3.7. Description and comments

To allow automatic creating of code documentation the phpDocumentor syntax will be applied to all code comments. Description for legacy code (without appropriate text) must be updated together with any code change.[4] All comments have to be in English.

3.7.1. Comments

Only BlockComment

```
/** ... */
```

and Line Comments

4 The current standard was introduced for 1.9 version

```
//
```

are allowed (# is not used).

Code should not be clustered with comments, but instead the code should be written in a readable way, so its meaning its clear and therefore it doesnt need to be commented. Only comment things where is absolutly needed and **avoid useless comments** like those bad examples:

```
//Initializing variables
$pass = 0;
```

or

```
// execute SQL request
$sqlTC = "SELECT id,title,summary,steps,exresult,version,keywords,"
```

or

```
$result = mysql_query($sql); //Execute query
```

or

```
//I had to write this code so that the loop before would
work.. I'm sure there is a better way to do it but hell if I know how to
figure it out..
if(count($testCaseArray) > 0){
```

These comments are quite unnecessary and useless (as they give no additional clues for the reader) and should be removed, before they bloat the code.

Example for a good comment:

```
// Chop the trailing comma off of the end of the keywords field
$myrowTC[6] = substr($myrowTC[6], 0, -1);
```

Here it isn't clear in the first view that there is always a trailing comma at the end of the keyword list, so its helpful to comment this intrinsic fact.

3.7.2. Standard file Header

Each file must contain the standard file header as seen in the example below below. Mandatory parameters are: @package, @copyright, @version. Other tags are optional. @TODO tag should be the last one.

```
/**
 * TestLink Open Source Project - http://testlink.sourceforge.net/
 * This script is distributed under the GNU General Public License 2 or later.
 *
 * There should be a description of file scope
```

```
*
* @package TestLink
* @author Andreas Morsing
* @copyright 2009, TestLink community
* @version CVS: $Id: inputparameter.class.php,v 1.11 2009/06/04 19:48:30 havlat Exp $
* @filesource http://testlink.cvs.sourceforge.net/viewvc/testlink/testlink/lib/f...
* @link http://www.teamst.org
* @since 1.9
*
**/
```

Common practice is also to store important changes in revision log. For example:

```
 * @internal Revision:
 *
 *   20090517 - franciscom - getTestersForHtmlOptions() interface changes
 *                           buildUserMap() added prefix to tag inactive users
 *   20081221 - franciscom - buildUserMap() interface changes
```

3.7.3. Class description

Class must include description and @package[5] in phpdoc format. Optional are using the next tags: @author, @see, @TODO. Do not use: @version or @license.

3.7.4. Function description

Each function be should be prefixed with a standard function header as seen in the example below.

```
/**
 * generates data for tree menu of Test Case Suite (in Test Plan)
 *
 * @author <nick>
 *
 * @param string $linkto path for generated URL
 * @param integer $hidetc [0: show TCs, 1: disable TCs ]
 * @param string $getArguments additional $_GET arguments
 *
 * @return string input string for layersmenu
 **/
```

This header consists of:

a comment: What does this function (mandatory)

@author tag(s) author nick or name (optional)

@param tag(s) for each parameter the function takes (mandatory if any)

@return tag describing the return value (mandatory if any)

@todo tag(s) describing the next work (optional)

Parameter types must correspond with official php types.[6] Note that DB handler type is

5 This is required by phpDocumentator :-(

6 http://www.php.net/manual/en/language.types.php

resource:

```
 * @param resource &$db reference to database handler
```

Do not use: @version or @licence.

3.7.5. Constants and variable description

Constants and defines should be prefixed by description without any tag.

Variables should be prefixed by **@var**, and optionally **@access** tag.

```
/** The root dir for the testlink installation with trailing slash */
define('TL_ABS_PATH', dirname(__FILE__) . DIRECTORY_SEPARATOR);

/**
 * @var integer current page number
 * @access private
 */
var $_currentPage = 1;
```

3.8. File structure

The file structure of TestLink should contain all sources of information which are needed by end-users to bring TestLink to work and also all possible information for developers. The infrastructure should be strongly categorized into the user related and the developer related sources of information, so end-user can easily identify documents which aren't needed for usage.

The infrastructure should contain documents, license information, sources and the website for TestLink. All items within the infrastructure must have version. This should be done per CVS so all items could be easily maintained and obtained.

All new file-names should contain only alphanumeric lower-case characters.

3.8.1. CVS directory structure

This section describes the directory structure on cvs.sourceforge.net.

- **Documents** (modul)
  - → **developers**
    - Developers Guide
  - → ***<major_release_vesion>*** (for example '1.7')
    - User manual *(obsolete)*
    - Installation manual
    - unsorted documents related to new features
- docs – includes obsolete www pages (for sf.net)
- **testlink** (source modul)
  - → **gui** (All user interface related material like templates, images, stylessheets should be inside this directory and the appropriate subdirectories. This directory

should also contains the locale specific stylesheets.)
    - javascript
    - templates
    - themes
        - css
        - images
    - templates_c
- → **lib** (All classes, function files should reside in this directory, no GUI related or stand-alone scripts! E.G. each scripts should contain only functions and/or classes Subdirectories as needed).
    - functions
    - general
    - testcases
    - ...
- → install (The install (and update)scripts, install (update) information are here).
    - sql (scripts with SQL requests needed by the installation)
- → thirdparty (Thirdparty component; as is fckeditor, etc.)
    - colorpicker
    - htmlarea
    - ...
- → config (configuration scripts (like config.inc.php) should be here)
- → locale (All localization related strings. Subdirectories are named like the appropriate locale like *en_GB*)
    - en_GB
    - de_DE
    - ...
- → docs
    - User manual (exported)
    - Installation and Configuration manual (exported)
    - Other user documentation
    - Examples (import files)

3.9. CVS

This section presents some short rules regarding to CVS handling. A good rule is to have two TestLink-source directories while developing. One which holds the current CVS contents and the other one in your web servers document root for developing. So its easy to revert any developing mistakes. Applying your changes to CVS is also easy, as the only thing you have to do is diff'ing your developing directory with the CVS directory and merge your changes and commit into CVS.

As you are not the only developing (-:, some rules should be applied while plaing with CVS.

- Update your CVS directory before applying your changes to it! As other developers may also change things on parts you are working on, updating avoids conflicts.
- Don't commit anything you are working on, which isn't complete! As CVS is accessible by all most people aren't happy if they checked out something, which isn't working at

all. So commit only when your changes (e.G. new feature) are working and complete. This doesn't meant that we expect only bugfree commits (-:. So don't commit your nearly 50%-ready changes, go in a two-weeks holiday and after that commit the rest.

- If you intend to work on a bigger feature it makes sense to communicate this to the other developers, so the amount of interferences are minimized.

- If you intend to work on a new feature which wasn't assigned to you, please contact the other developers, nothing is more frustrating and resource wasting when two different people are implementing same thing in parallel.

4. Architecture

TestLink common page consists from the next parts:

- Load configuration (include config.inc.php)
- Load core functions and classes (include common.php)
- Load page specific functions and classes (include scripts from `<testlink_root>/lib/functions/` directory)
- Initiate database connection and session

```
testlinkInitPage($db);
```

- Parse input variables
- Process the page logic
- Call smarty templates component (initiate, add the page specific variables and render with related template).

4.1. Classes

The next picture shows basic relation of core TestLink classes:

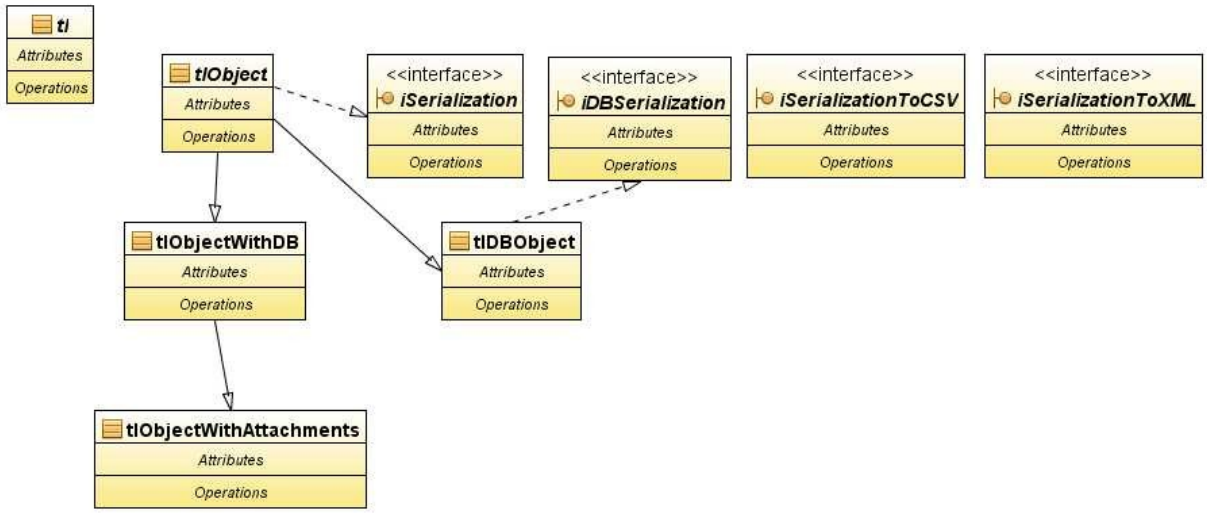


Illustration 1: Core classes

4.2. Debugging / Logging

TBD: Describe Audit functionality

See `config.inc.php` to set logging level and target file. Log rotation is not internally supported. Recommended user level is ERROR.

Now any log messages from the levels ERROR or INFO will be recorded. DEBUG messages will be ignored. We can have as many log entries as we like. They take the form:

```
tLog("testing level ERROR", 'ERROR');

tLog("testing level INFO", 'INFO');

tLog("testing level DEBUG");
```

This will add the following entries to the log:

```
[05/Jan/27 13:05:56][INFO][guest] - Login ok. (Timing: 0.000763)

[05/Jan/27 13:06:03][DEBUG][havlatm] - User id = 10, Rights = admin
```

`tLogSqlError($sql)` - function specified for unsuccesful database query.

4.2.1. Timing

Timing is available for performance optimalization. Use the next functions:

`tlTimingStart ($name),`

`tlTimingStop ($name)`

and `tlTimingCurrent ($name).`

The last one returns the measured time.

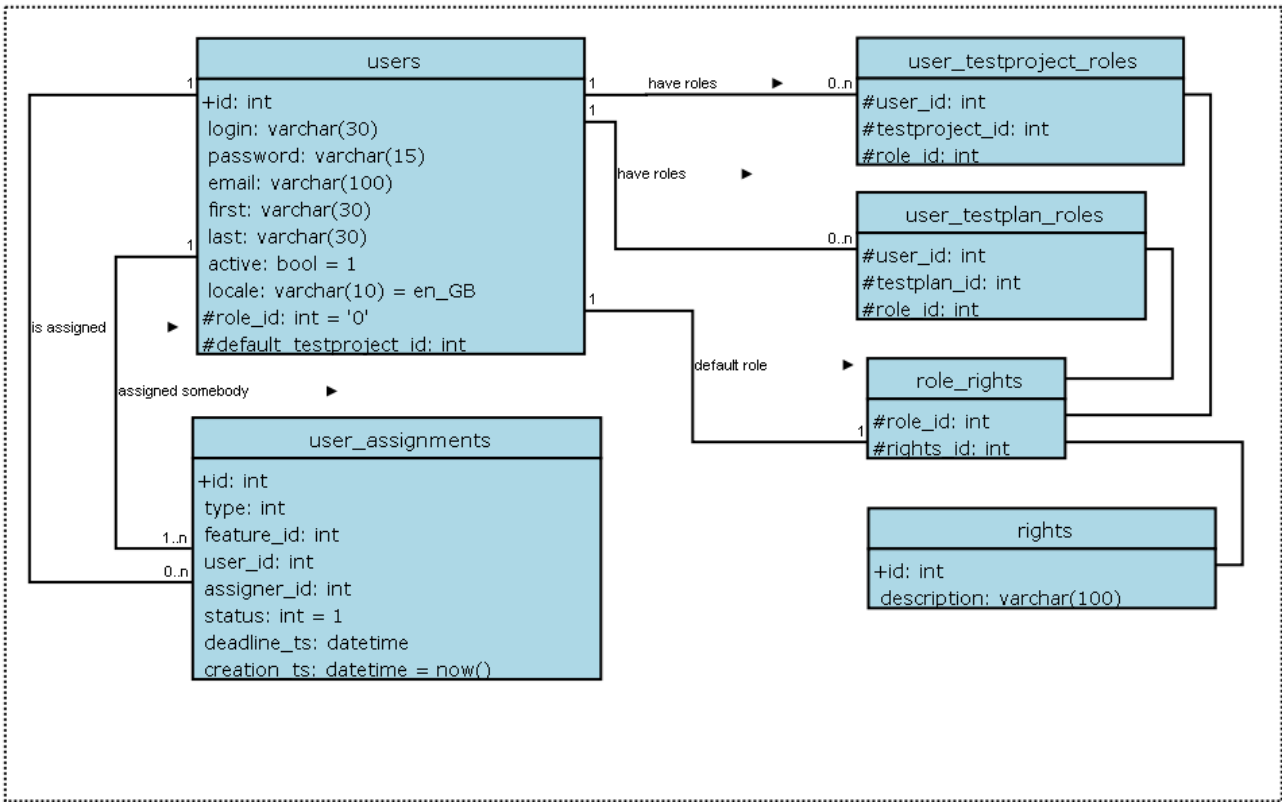
4.3. Database access

ADODB third party component should be used for any DB access. See wraper in database.class.php for more.

4.4. Users and roles

Any user has one generic role, none or more project related roles and none or more test plan roles. Generic role is valid if no specific role is set for the current Project or the current Test plan. This generic role is store together with user data. Test Project and Test plan specific roles are stored in extra tables.

TODO: Administrator role must be only generic role.



4.4.1. Rights Definitions

Particular features have defined **right** records. Common practice is to have view and edit rights for one feature. Keywords used for definition of role abilities. Naming convention of these keywords: test plan related keywords begins "testplan_"; "system_" are unrelated ones; test project related ones "mgt_" or "testproject_".[7]

There is six default roles defined in new installation process. Users can modify rights of these roles or create a new role definition via GUI. Migration must care that all new rights (developed in the release) are assigned to administrator, but do not change any other role definition.

7 "mgt_" is for historical reason and "testproject_" is task for future.

Right	Description
mgt_view_tc	View Test Specification (Test Suites and Test Cases)
mgt_modify_tc	Edit Test Specification (create, modify, delete, order, move, and copy - both Test Suites and Test Cases)
mgt_view_key	View keywords
mgt_modify_key	CRUD keywords
mgt_modify_product	Create,edit and delete Test Projects
mgt_view_req	View requirements
mgt_modify_req	Create,edit, associate and delete requirements
cfield_view	View defined custom fields
cfield_management	Manage custom fields
role_management	Modify definition of existing roles and add a new ones
user_role_assignment	Possibility to set a role for users
mgt_view_events	See audit/debug log

Table 1: Test Project related Rights

Right	Description
testplan_execute	Ability to execute Test Cases (insert Test Results)
testplan_create_build	Ability to manage Builds
testplan_metrics	View reports and metrics
testplan_planning	Create, edit, delete Test Plans, assign test urgency, assign ownership, milestones, edit Test Case Suite
testplan_user_role_assignment	Assign the rights to view projects

Table 2: Test Plan related Rights

Role	List of Rights
Guest	mgt_view_tc, mgt_view_key,
Browse data only.	testplan_metrics
Tester	mgt_view_tc, mgt_view_key,
Execute test.	testplan_execute, testplan_metrics
Test Analyst	
(Senior Tester)	mgt_view_tc, mgt_modify_tc, mgt_view_key, mgt_modify_key, mgt_view_req,
Edit Test Specification, execute tests, create Build.	testplan_execute, testplan_create_build, testplan_metrics
Test Designer	mgt_view_tc, mgt_modify_tc, mgt_view_key, mgt_modify_key, mgt_modify_req, mgt_view_req,
Edit Test Specification and Requirements.	testplan_metrics
Test Leader	mgt_view_tc, mgt_modify_tc, mgt_view_key, mgt_modify_key, mgt_view_req, mgt_modify_req, user_role_assignment,
All Test Plan permissions, edit Test Specification and Requirements.	mgt_testplan_create, testplan_execute, testplan_create_build, testplan_metrics, testplan_planning, testplan_assign_rights
Administrator	
Everything possible. Only this role can maintain Test Projects, Custom fields and users.	All existing roles

Table 3: Default Role attributes

4.4.2. Test Plan assignment to users

There is a table with Test Plan rights (i.e. which users can see which Test Plan). This table is made up of a combined user id and project id. The main page contains code which checks to see if the logged in user has the appropriate permissions (and then shows the allowed projects). It is not recommended that this be hacked with.

Administrator cannot be set to Test Plan.

4.5. EVENT-Log / AUDIT-Log

This document describe a new mechanism for event- and audit logging and should extend the current logging mechanism

TestLink already has some kind of logging mechanism and some DEBUG levels, the problem is now, that these infos are only visible within the logfiles. So if an error happens the user must inform it's admin to look into the logfiles. This process can be made easisier by given the user and admin more tools to detect, examine and be informed of Events.

4.5.1. Terms

TRANSACTION

Every Page has some purpose (-: Each purpose can be seen as some kind of transaction (at least an anonymous). So each page has at least one transaction which will be opened on start-up and closed on shut-down.

TRANSACTION-NUMBER

Each transaction has a unique reference number. This reference number can be used to gather of all events of a transaction

OPENED_DATE / CLOSED_DATE

Each Transaction has a opened-date and a closed-date

USERID:

Each transaction contains also the userID of the user which starts the transaction

ENTRYPOINT:

Each transaction begins a one entry point which is the name of the PHP script it started

EVENTS

Each transaction consists of one or more EVENTs. These EVENTS could be failed database statements, failed assertions or checks, or something else.

ERROR-LEVEL

Each EVENT has an Severity, this is similar to the already present logging concept in TestLink.

We have:

- WARNING (not really always an error, but maybe ...)
- ERROR (always an error)
- AUDIT (audit trail information)

DESCRIPTION:

Each Event has a description which fully describes the EVENT

SOURCE

Each event has a source, this can be the page itself, database or something else

TIMESTAMP:

Each Event has a timestamp. The timestamp is set to the time (UTC) when the event occurs

TRANSACTION-NUMBER

Each Event contains the transaction number to which it belongs

AUDIT-TRAIL:

For some reason (security and some more), Testlink should trace modification done by users. So for these special things AUDIT-Events are fired. Example are (failed) logins, logout, modifiying or deleting stuff ,...

4.5.2. Implementation notes

Database: All Transactions and events are stored within the database in two tables.

```
transactions (id, opened_date, closed_date, userid, entrypoint)
events (transaction_id error_level, source, description, timestamp)
```

The events can be seen within a new feature of TestLink: Eventviewer. This eventviewer is part of TestLink and displays the events of the system, offer search facilities, deletion of events.....

This is the place to look for errors and audit stuff. Because reviewing the EVENT-Log may be tedicious and boring if there are no reasons to do it. TestLink offers the possibility to send notifications via Email or something else or display them on the main page (like the security notes).

TL 1.8: The current TestLink logging mechanism is extended to support the transaction and event format.

4.5.3. User configuration

User configuration must be extended with two new rights: „***can see events***" and „***can manage events***".

Also Each user can configure how (eG Email...) It should be informed in case of new events and on which events he is interested (ERROR-LEVEL).

5. Security

This chapter rules for security requirements and recommendations.

5.1. Terms

SOFT requirements: that means, violation of this doesn't result in a call to exit(), or they are programming guidelines.

HARD requirements: Violation leads to an event entry and a call to exit() (aborting the script)

5.2. Input data

All script input parameters from GET,POST, or something else client-related MUST be:

- SOFT: accessed only once at the beginning of the script ONLY through tlInputParameter related access functions. Direct access to $_POST,$_GET,$_REQUEST is NO longer allowed

this can be done using init_args() structure, and adding all calls to sanity functions in a single place, then we can continue using $args->

- SOFT: trimmed of whitespaces (left and right)
- HARD: Min/Max-Length, Min/Max-Vals checked and enforced
- HARD: Type checked (int,string,bool,...)
- HARD: format checks where possible (IP, numbers, dates,....) => regexp check
- HARD: whitelist/blacklist-checks where possible
- SOFT: client side consistency checks must also be done on server-side

All output to the user MUST be escaped with htmlspecialchars() (PHP) or modified with escape-modifier (TPL).

No hidden fields should be used to transfer data from page to page. Use $_SESSION storage.

5.3. Rights

All scripts must be secured with checkRights() function. That means all scripts must check at the beginning if the user has the appropriate rights, to call the script. If the script contains multiple action then also for each action the rights must be checked.

5.4. Uploads

All uploads should check the uploaded files for malicious content:

- Whitelist of mime-types
- extension checking

- peek into content
- validity checks

6. Graphic User Interface

TestLink has independent GUI rendering by component Smarty templates.

All files (except generated documents) are expected to fulfill the **XHTML 1.0 Transitional** standard. A browser is instructed to not cache the pages. CSS 1.0 atttributes are prefered standard because of www client compatibility. Parameters from standard CSS 2.0 and later must be verified in both IE and Firefox before using.

6.1. GUI - Smarty templates

This section describe common practices using of some parts of templates. See more Smarty homepage to functional description and syntax. Also using of CSS is described in this chapter.

6.1.1. A page template structure

The pages are composed from a page template and included shared parts. Common page structure includes:

- Text Header with description and identification (See 6.1.3.)
- Included HTML header
- HTML body definition
    - Header 1 title
    - Action feedback
    - main with HTML content

6.1.2. File-name standard

All Smarty Templates are stored in the directorey `<tl_root>/gui/templates/`.

See related general chapter above for naming convention.

6.1.3. Template header

The simple header is used in each template. See the next example. The last line of the example informs about significant changes. No needs to describe changes in body of template more.

```
{* TestLink Open Source Project - http://testlink.sourceforge.net/ *}
{* $Id: execNavigator.tpl,v 1.3 2005/08/27 20:53:30 schlundus Exp $ *}
{* Purpose: smarty template - show test set tree *}
{* 20050828 - scs - added searching for tcID *}
```

6.1.4. HTML page header

Html header is defined in ***inc_header.tpl***. This template is included to each page template immidiatelly after file description. E.g. `{include file="inc_head.tpl" jsTree="yes"}`.

The header template can have the next parameters:

- $openHead="yes" - The head.tpl includes a closing of html header as default. This parameter cause that header remains open for another definition e.g. javascripts call. `</head>` must be defined in page then.
- $jsValidate="yes" - includes `validate.js` javascript library to the html header. It includes functions for input verificiation.
- $jsTree="yes" - includes `inc_jsTree.tpl` (javascript to the html header).

Functions in `testlink_library.js` are everytime loaded.

Included modificable parameters (via `$smarty->assign()`):

- $pageCharset - UTF-8 is default;
- $css - <tl_root>/gui/css/testlink.css is default for pages and tl_doc_basic.css for generated documents;
- $SP_html_help_file
- etc.

6.2. Tables

There are four basic types of style:

- class="common" - used to view data
- class="simple" - used to desing a forms (input data)
- class="invisible" - used where div/span formating is not sufficient
- class="smallGrey" - used to define options, filters (for example in left navigation pane)

6.3. Help reference

Use the next example of code to show help icon in title:

```
<h1> <img alt="{lang_get s='help'}: {lang_get s='your_string_identifier'}"
class="help" src="icons/sym_question.gif"
onclick="javascript:open_popup('{$helphref}requirementsCoverage.html');" />
{lang_get s='your_string_identifier'}: {$arrReq.title|escape} </h1>
```

Use the next example of code to call help by click a string:

```
<span class="help"
```

```
onclick="javascript:open_popup('../help/glosary.html#testspec');">{lang_get
s='your_string_identifier'}</span>
```

6.4. Buttons

Each button should be included in `<div class="groupBtn">`. Navigation buttons should be in a top left part of a page. Submit buttons should be also in a bottom part of a page. Both top and bottom button should be used if a page is long (for example the test execution page with tens of test cases).

6.4.1. Action feedback

The template `inc_update.tpl` is useful generalized template. See for more in.

Styles `div.error`, `div.info` should be also used for information about some result.

6.4.2. Combobox Menu

Use smarty component html_options. See more into Smarty documentation. E.g.

```
<select name="optReq">
     {html_options options=$option_yes_no selected=$reqs_default}
</select>
```

6.5. Delete Pop up

Use a pop-up window for confirmation of delete operations. This pop-up window should be generated using EXT JS javascript library.

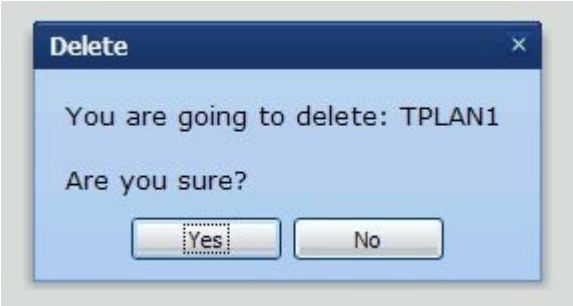


Illustration 2: Pop-up dialog example

Pop title will contain operation ('*Delete*').

Message body should contain as a minimun, following information:

- indication about will be done ('*You are going to delete:*')
- name of item on which user is working (for example *'Mantis release 1.2'*)
- confirmation question ('*Are you sure?*')

The following pieces must be present to add a pop-up window in a smarty template:

```
[1] {lang_get s='warning_delete_testplan' var="warning_msg" }
[2] {lang_get s='delete' var="del_msgbox_title" }
[3] {include file="inc_head.tpl" jsValidate="yes" openHead="yes"}
[4] {include file="inc_del_onclick.tpl"}
[5] <script type="text/javascript">
            /* All this stuff is needed for logic contained in
inc_del_onclick.tpl */
            var del_action=fRoot+'lib/plan/planEdit.php?
do_action=do_delete&tplan_id=';
    </script>
[6] <body {$body_onload}>
...
[7] <img style="cursor: pointer;"
            alt="{lang_get s='testplan_alt_delete_tp'}"
            title="{lang_get s='testplan_alt_delete_tp'}"
            onclick="delete_confirmation({$testplan.id},
                      '{$testplan.name|escape:'javascript'}',
                      '{$del_msgbox_title}','{$warning_msg}');"
            src="{$smarty.const.TL_THEME_IMG_DIR}/trash.png"/>
```

Explanation:

[1] gets the appropriate message from strings.txt. This message must have placeholder (%s) that will be replaced with name of item.

Example:

```
$TLS_warning_delete_testplan="You are going to delete: %s <br /><br /> Are you
sure?";
```

[2] will be pop up title.

[3] standard include that must have option **openHead="yes".**

[4] include delete on click logic (javascript functions)

[5] definition of callback function that will be called when user press yes button.

[6] initialization for ext js

[7] configuration of onclick event to call 'delete_confirmation()' javascript function.

```
/*
   function: delete_confirmation
   args: o_id: object id, id of object on with do_action will be done.
               is not a DOM id, but an specific application id.
         o_name: name of object, used to to give user feedback.
         title: pop up title
         msg: can contain a wildcard (%s), that will be replaced
              with o_name.
   returns:
 */
```

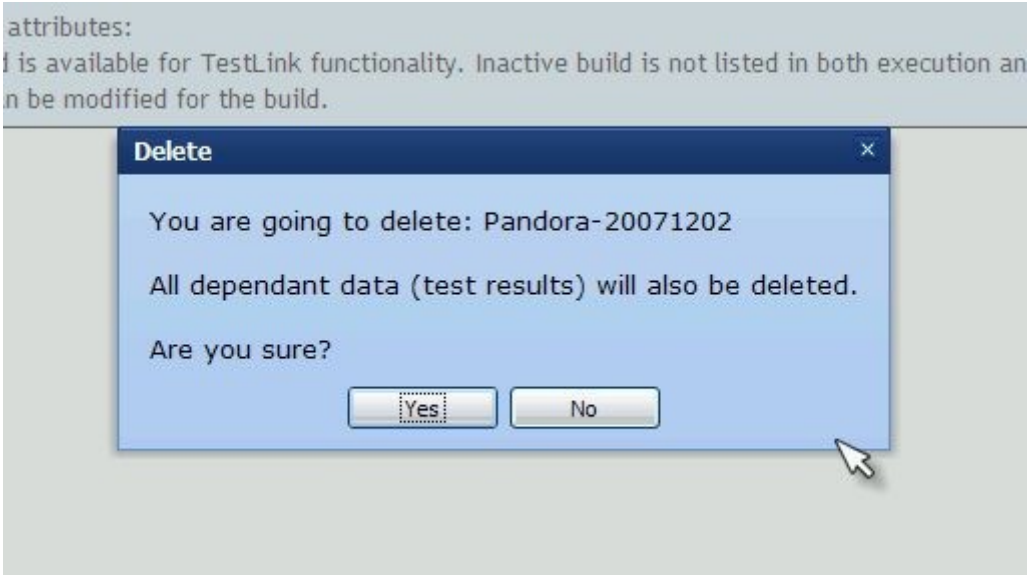


Illustration 3: Another pop up example with a more detailed message

6.6. Save GUI layout

Martin: We should implement EXT-JS ability to save states via cookie (stateId). Not implemented yet.

6.7. Smarty debug

Smarty allows to view all available variables and constants via extra window. Set-up:

```
$tlCfg->smarty_debug = TRUE;
```

7. How-to

7.1. How to write interface for Bug Tracking systems

This section shortly describes how to write a new interface for bug tracking systems. A bugtracking interface mainly consists of two files: First a config file located in the cfg-directory and a interface definition file located in the lib/bugtracking-folder.

7.1.1. Name the interface

First choose a name for your interface and make this name available in config.inc.php (search for TL_INTERFACE_BUGS). Just use some simple word like 'BUGZILLA' or 'MANTIS'.

7.1.2. Create the config file

As already mentioned the config file is located in the cfg-folder. The file should contain all the configuration options for the interface like dbhost and such stuff. For the already used options just look at the bugzilla.cfg.php file.

For simplicity just name the config file similiar to the name you've chosen in 1.1 like bugzilla.cfg.php for 'BUGZILLA'

7.1.3. Create the interface file

The interface file is located in lib/bugtracking and should be named similiar to the name chosen in 1.1 like int_bugzilla.php for 'BUGZILLA'.

Create a new interface which extends the base interface in the file int_bugtracking.php. It is important to made the interface name available via the define "BUG_INTERFACE_CLASSNAME". For an example of a bugtracking interface look at int_bugzilla.php file.

7.1.3.1 Description of the base class

The base class already contains code for some basic stuff like connecting. So mostly there is no need to overload these functions.

The only function which is directly called by TestLink is the buildViewBugLink-Function which retuns a link to the bugtracking system. The base already has a default implementation of it, so maybe you dont have to change it. The default implementation calls to helpers name buildViewBugURL and getBUGStatusString. The first one build the URL for accessing the bugtracking system and the second one returns a user friendly description of the bug like BugID and such stuff. So these two ones are the first to change. Another helper which may come in handy is getBugStatus which should be overloaded to get the bug status from the bugtracking-db. this status can be used in the two other helpers.

For an example how to implement these function just look at the mantis and bugzilla interfaces. For comments and the purposes of the different base interface functions just look at the int_bugtracking.php file. The code is well documented and straight forward.

7.1.4. Setup your bugtracking interface

After you named the interface and created the two files it is necessary to add these filenames to the int_bugtracking.php file. This can be done by adding the interface to the two associative array $configFiles and $interfaceFiles and the beginning of the file.

7.1.5. Testing the interface

If the connection to the bugtracking system is successfully done. You should test the interface. So enter a dummy bug in your bugtracking system and execute a testcase from a dummy-testplan within TestLink. You should now see a input field for entering bugs. So mark the testcase as failed and enter the id of the created dummy bug. Now switch to the reports and execute the report of failed testcases. The report should no show a link the bug. Clicking the link should open a new window which shows the bug in your bugtracking.

7.1.6. Some words

We would be very appreciated if you send your created and working interface(s) to us, so we can integrate them to our TestLink distribution.

8. Documentation

As already stated, all TestLink related documents should be made public and available, so no information gets lost or will be concentrated on a single person. Documentation is stored in CVS under module 'documents'.

Each document should have a single point of authority that means that this person is the only one person allowed to extend and change this document. This person is also responsible to collect possible changes, evaluate them and possibly put them into the document he owns.

All documents should be versioned and also all versions should be available (this is done by CVS). Each document should also have a form of change (or revision) history which shortly describes the changes between two versions. **Table of Revision history** is at the end of documents. Please, hold it current.

All documents should adhere the same format and underlying representation. To easily maintain the documents we choose to use **Open Office**. Output formats **HTML** and **PDF** are exported and published for users. Docbook system (used for TL1.5 is now obsolete).

Exception are README, LICENSE and CHANGELOG. This documents are not versioned and have simple text format. Update of this files is a part of Build process. See Error: Reference source not found.

The user related documents must be updated and check-in to CVS before build (i.e. README, CHANGELOG, User manual and Installation manual). All this documents must be available with build package and manuals also on web pages.

8.1. Developers documentation

The documents are stored in directory: `<testlink_root>/documents/developers/`. This directory holds any developer related sources of information like this documents, or documents related to organization, coding standard....

List of subdirectories:

- **Developers Guide** - This document is related to the developing process of TestLink which should be applied by all developers. It is describing the architecture, interfaces and database of TestLink.
- **Design documents** – this document should be designed before important changes for review and later understanding of feature.

8.2. User documentation

The documents are stored in CVS: `<testlink_root>/documents/<version>/`. This document contains the sources of the end-users related material.

- **User manual**
- **Installation and configuration manual**
- **readme** – stored in source root directory; updated during build process

- **Changes log** - This file holds the changes for TestLink versions. TXT type is used. Updates are generated by BTS (Mantis).

8.3. Source texts

The directory `<testlink_root>/testlink/documents/` contains the documentaion exported fo html format (created during build process).

There are also instructions and help files available (see `<testlink_root>/testlink/gui/help/`). This files are localized.

9. Third party components

9.1. ADOdb

ADOdb component is used as abstraction layer for connection to database. The component is accessible via class `database`.

Manual: http://phplens.com/lens/adodb/docs-adodb.htm

~~9.2. phplayersmenu~~

~~The component is used for tree menu with Test Suites and Test Cases.~~

~~Pages: http://phplayersmenu.sourceforge.net/~~

~~*Note: the projects seems dead; removed from 1.9 version*~~

9.3. FCK Editor

The component is used as default html editor.

http://www.fckeditor.net/

The component has amount of nice features that are not used in default configuration. Include pictures upload. We recommend to look at the component possibilities for all users.

9.4. Smarty templates

Smarty templates are used for GUI rendering.

Manual: http://smarty.php.net/manual/en/

9.5. Ext-JS

Tree menu component and dialogues rendering.

More: http://extjs.com/products/extjs/

9.6. phpDocumentator

The tool is used generated documentation of code description.

Reference: http://www.phpdoc.org/

#	Description	Date	Authors
0.1 - 0.5	Coding standard create	2005/3/13	SCS, FM, MHT
0.6	Initial compilation from the all related documentation to Developer's Guide	2005/07/15	Martin Havlat
0.7	Conversion to OO2, layout correction, doc section updated	2005/12/11	Martin Havlat
0.8	Added processes, build, CVS structure, documentation overview (all partly moved from obsolete architecture guide); created index	2006/01/13	Martin Havlat
0.9	Layout update; updated Documentation chapter; added Team organization chapter	2006/01/17	Martin Havlat
0.10	Update of BTS section	2007/02/26	Martin Havlat
0.11	Update styles	2007/09/27	Martin Havlat
0.12	Major update (edit obsolete texts); Chapters: filenames, releases	2008/02/02	Martin Havlat
0.13	Minor update and revision for TestLink 1.8	2008/08/18	Martin Havlat
0.14	Updates before TestLink 1.8.0	15/03/09	Martin Havlat
0.16	Updates for 1.8 hot-fixes; defined goals	03/06/09	Martin Havlat

Table 4: Revision History