TestFarm Test Suite Reference Manual

Version 1r1

Table of Content

1 Structure of a Test Suite

A **Test Suite** is a collection of **Test Cases** gathered within a **Test Tree**.

The active nodes of the Test Tree are the Test Cases, which role is to verify a single functionality of the product under test. A Test Case embeds a **Test Script**, which is a small PERL program. This program may be written manually in **PERL** or generated automatically from a **WIZ-script**. The WIZ-scripts are written in a high-level simple macro-language dedicated to functional product testing.

A **Test Sequence** is a group of Test Cases, executed sequentially. A sequence may also contain other sequences, thus allowing to implement a well-defined hierarchy of tests when constructing the Test Tree. The head of the Test Tree is a sequence, called the Root Sequence.

A **Scenario** is Test Sequence that contains only Test Cases. This can be considered as a leaf group of Test Cases. This principle of grouping Test Cases allows to produce structured test reports, following a well-defined Test Suite specification process.

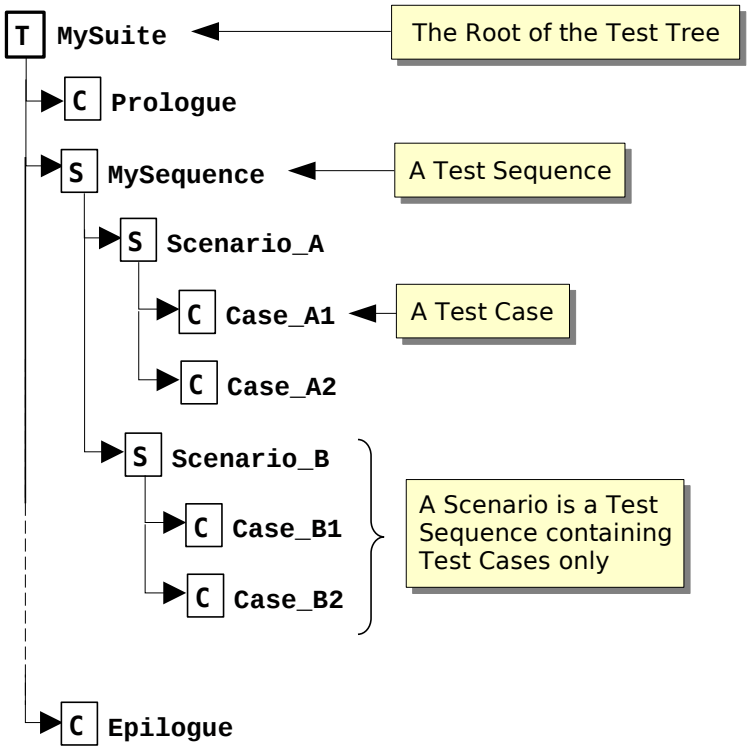
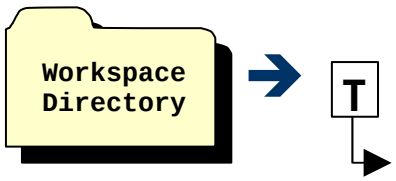
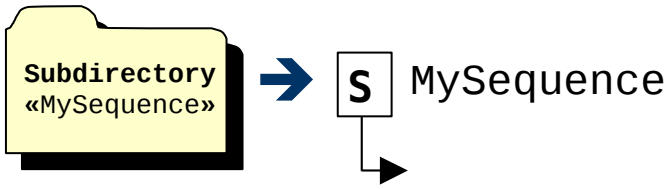
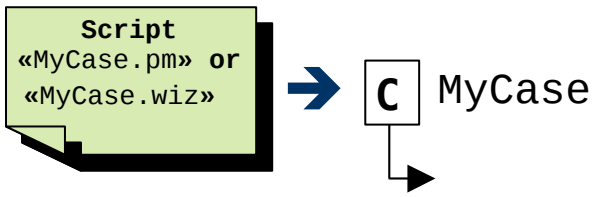


Figure 1.1: Example of Test Tree

2 The Test Suite Workspace

A Test Suite is stored in a directory of the computer file system. This directory is called a **Workspace**. The basic concept of a Test Suite consists in storing the Test Scripts in a hierarchy of subdirectories that have the same structure as the Test Tree, in respect of the following rules:

From Workspace Directory starts the root of the Test Tree	
A Subdirectory produces a Test Sequence	
A Test Script (written either in PERL or in WIZ macro language) produces a Test Case attached to this Test Script	

The job of the Test Suite developer is to construct a hierarchy of directories in which (s)he puts the Test Scripts. A compilation phase is then performed by the **TestFarm Build Tools** and the *GNU make* utility: a simple "`make`" command produces a set of files that are loadable by the Test Suite User Interface, in order to execute the Test Suite.

3 The Test Tree

3.1 The Tree Files

As explained in section Error: Reference source not found, the structure of the Test Tree is constructed from the directory hierarchy of the Test Suite Workspace.

Each directory of the workspace should contain a **Tree File**. A Tree File name has a suffix "`.tree`". It provides a list of the nodes attached to the Test Sequence implemented by the directory. The presence of a Tree File is required in the workspace directories for three reasons:

- The file system of the computer does not indicate in which order the nodes must be placed in the Test Sequence. For this reason, it is necessary to explicitly write an ordered list.
- A directory may contain some files that are not to be placed in the Test Sequence: if they are not listed in the Tree Branch File, they won't be taken into account in the Test Tree.
- By putting some directives into the Tree Branch File, it is possible to give additional information about the Test Sequence (a short description, a document reference, ...). When the Test Suite is executed, this information is put into the Test Report.

3.2 Naming the Tree Files

Two categories of Tree Files have to be considered: the **Tree Root File** is stored in the workspace root directory, and the **Tree Branch Files** are stored in the subdirectories. Both categories are processed in the same way, but they are handled differently when constructing the Test Tree.

3.2.1 The Tree Root File

A Tree Root File is stored in the Workspace root directory to declare a Test Suite. Several Test Suites may be declared in the same workspaces, and they may or may not contain common nodes. A Tree Root File "***MySuite*.tree**" is directly loaded by the Test Suite User Interface for execution.

The Test Sequence constructed from a Tree Root File is symbolized with **T** . It is also called the Root Sequence.

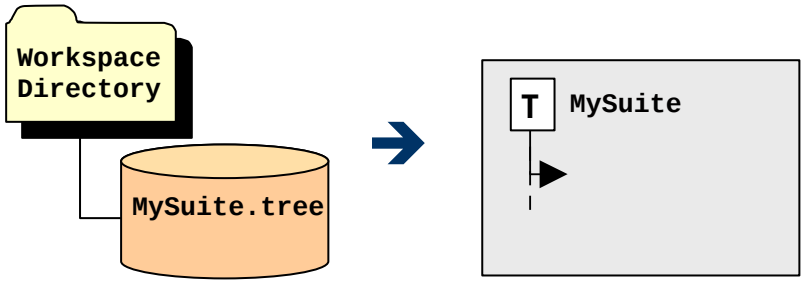


Figure 3.1: A Tree Root file defines a Test Suite and a Root Sequence

3.2.2 The Tree Branch Files

The Tree Branch Files are stored in the workspace subdirectories. There are two manners in which to name a Tree Branch File:

- When a Tree Branch File is named "***MySuite*.tree**", it is processed when constructing the Test Suite "*MySuite*" (i.e. the Test Suite constructed from the eponymous Tree Root File).
- If no file "*MySuite*.tree" is present when constructing the Test Suite "*MySuite*", the default file "**.tree**", is processed.

A Test Sequence constructed from a Tree Branch File is symbolized with **S** . When a sequence contains only Test Cases, it is called a Scenario.

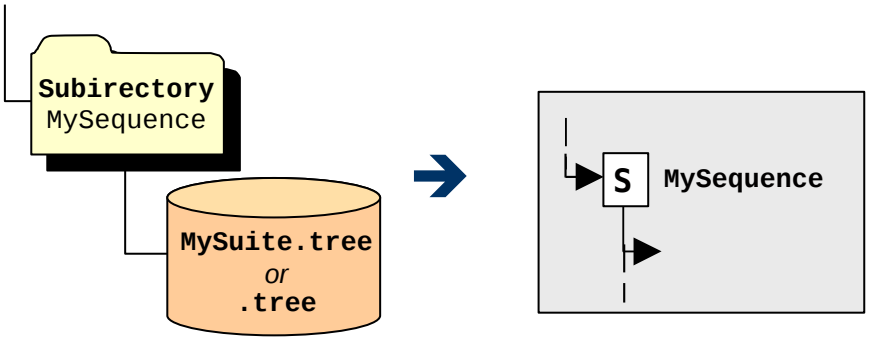


Figure 3.2: Creating a Test Sequence from a Tree Branch File

3.3 Writing a Tree File

A Tree File is a text file containing **Directives** and **Node Names**. Only one item should be written per line. A comment begins with "#" and finishes at the end of the line.

Figure 3.3 illustrates a workspace with it files, and how a Test Tree is constructed from them.

3.3.1 The Tree File Directives

#$DESCRIPTION **Tree File Directive**

SYNOPSIS

`#$DESCRIPTION` *Text*

DESCRIPTION

Gives a short description (one line) of the purpose of the Test Suite, Sequence or Scenario. This information is displayed in the result statistics tables of the Test Report.

EXAMPLE

```
#$DESCRIPTION ICC Terminal Type Approval
```

#$REFERENCE **Tree File Directive**

SYNOPSIS

#$REFERENCE *Text*

DESCRIPTION

Indicate the reference of the document in which the Test Suite, Sequence or Scenario is specified. This may contain a document title, chapter numbering, etc. The exact content of the information depends on the user process and environment.

When specified in a Tree Root File, the text provided by the directive is displayed in the header of the Test Report.

EXAMPLE

```
#$REFERENCE TBD/EXE/T01 2.04 - February 11th 2000
```

3.3.2 Naming the Tree File Nodes

A node name may designate a Test Script (without a suffix, like in ".pm" or ".wiz") or a subdirectory. By scanning the current directory (i.e. the directory the Tree File is stored in), the TestFarm Build Tools automatically detects whether the node corresponds to a Test Script or a subdirectory:

- If a node "*MyDir*" is present and the subdirectory "*MyDir*" exists, a child sequence "*MyDir*" is created. The subdirectory will in turn be processed recursively. A Test Sequence is symbolized with **S** in the Test Tree.
- If a node "*MyScript*" is present and a file "*MyScript*.pm" or "*MyScript*.wiz" exists, a Test Case is created using this script. A Test Case is symbolized with **C** in the Test Tree.
- If no subdirectory or valid Test Script is found, an error is reported by the TestFarm Build Tools.

The names of the Test Tree nodes are constructed with the following rules:

- A tree node (either a sequence or a test case) is given the full path name of the corresponding file object (respectively a directory or a script), considered from the workspace root directory.
- The directory separation character (which is "/" on the Unix family systems) is replaced with an underscore character "_".

The example shown in Figure 3.3 illustrates these naming rules.

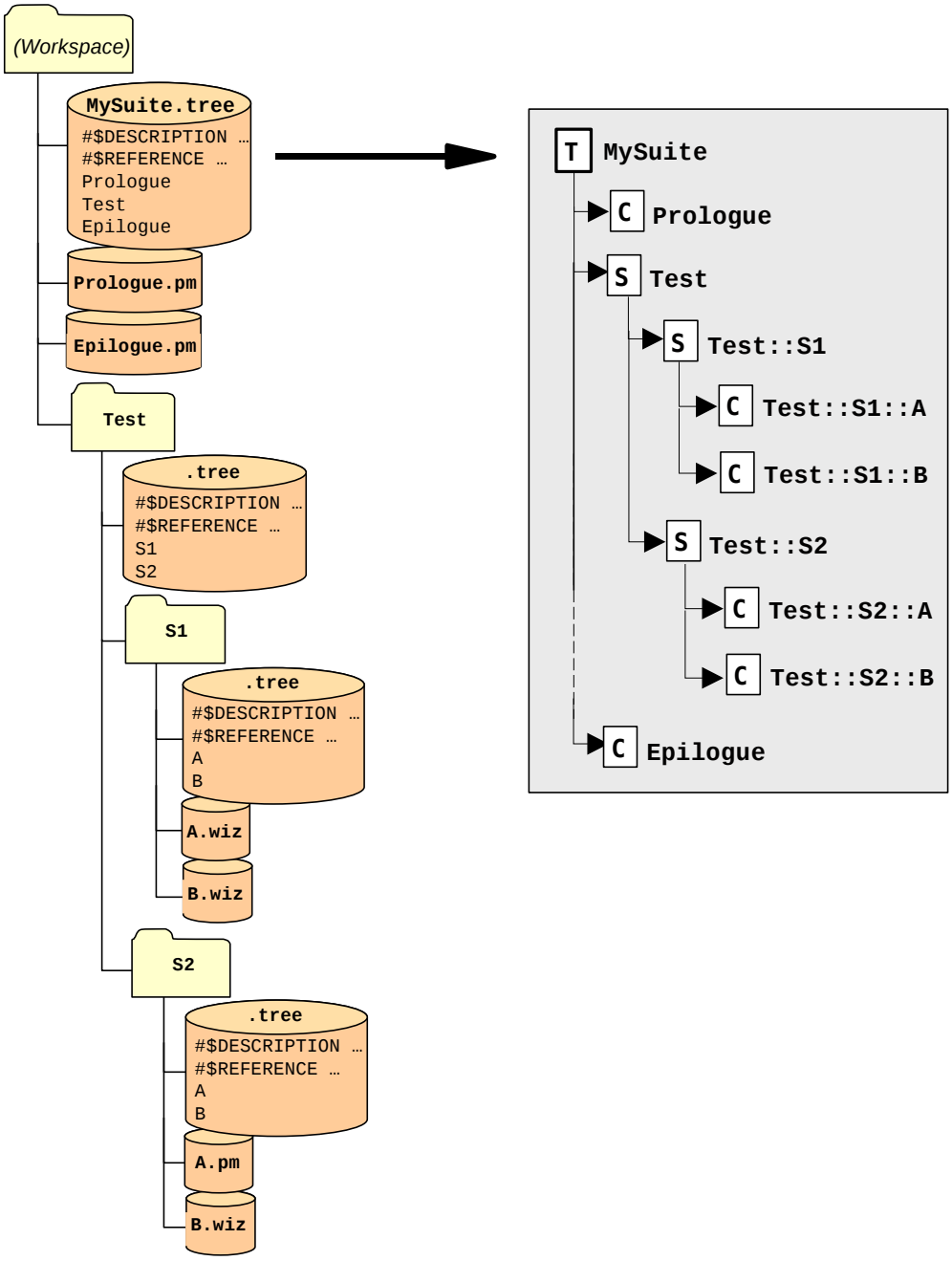


Figure 3.3: An example of Test Suite Workspace and its resulting Test Tree

4 The Test Cases

4.1 What is a Test Case

A Test Case is an active node of the Test Tree. Each Test Case is implemented with a Test Script written in PERL or in the TestFarm WIZ macro language. The content of a Test Script is detailed in section 5.

4.2 Verdicts

A Test Script executes a program and returns the **Verdict** of the Test Case. A verdict can take the following symbolic values:

Verdict Identifier	*Meaning*
PASSED	The Product Under Test behaves correctly
FAILED	A failure was detected in the Product Under Test
INCONCLUSIVE	The Product Under Test or the Automated Testing System has an abnormal behaviour: no significant verdict can be produced
SKIP	The test case is not relevant: it was not executed

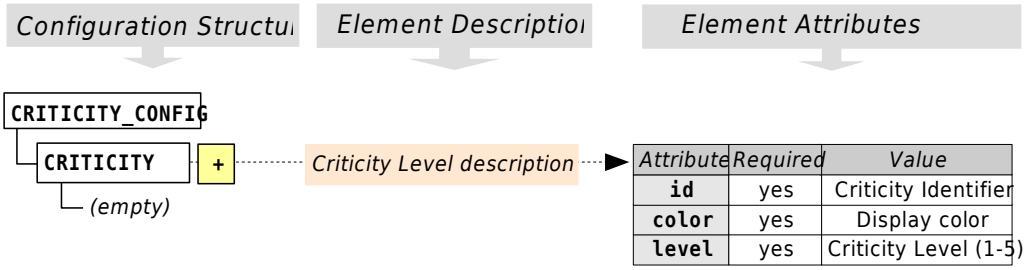
4.3 Criticity

A Test Case is also given a level of **Criticity**, which indicates how serious is the situation when a failure is detected in the Product Under Test. There are up to five criticity levels, to which you have to give a particular meaning. The definition of the criticity levels may take into account several considerations such as cost, delivery time, company reputation, etc.

The standard criticity levels are defined in the file "`/opt/testfarm/lib/criticity.xml`":

Criticity Identifier	*Definition*
Near	The lowest criticity level: the SUT implementation is close to the specification. The failure is not important, and a workaround exists.
Light	The SUT is under acceptable quality level but yet usable.
Serious	The SUT has a failure that make it unacceptable.
Major	The SUT has a failure that make it unacceptable and may have a significant impact on the cost, delivery time and/or relationships with your customer.
Catastrophic	The highest criticity level: the SUT has a failure that may cause the cancellation of your customer contract.

If needed, the system administrator may define some custom criticity levels in file "`/var/testfarm/lib/criticity.xml`", taking into account the quality management rules of the company.



Attribute	Required	Value
id	yes	Criticity Identifier
color	yes	Display color
level	yes	Criticity Level (1-5)

5 The Test Scripts

5.1 Two kinds of Test Scripts

In a Test Suite workspace, two kinds of test scripts may be used. As illustrated in Figure 3.3, these two kinds may be used conjointly within the same workspace. When compiling the Test Suite, the TestFarm Wizard Tools automatically detect the type of script, and launch the proper code generation process:

- **The scripts natively written in PERL: `*.pm` files**.
  A PERL Test Script (a.k.a. a **PERL-script**) is materialized with a PERL module (see 5.3).
  If a `*.pm` file is encountered, it is linked directly to the Test Suite.
- **The scripts written in the TestFarm WIZ macro-language: `*.wiz` files**.
  A WIZ Test Script (a.k.a. a **WIZ-script**) is a sequential assembly of PERL macros (see 5.4).
  If a `*.wiz` file is encountered, a PERL script is automatically generated from it (which is somehow a macro-expansion), and linked to the Test Suite.

5.2 Test Script Directives

A Test Script, either written in PERL or in the WIZ macro-language, may include some directives providing information for generating the Test Report. A directive have the "#$" prefix.

#$DESCRIPTION — Test Script Directive

Synopsis

```
#$DESCRIPTION Text
```

Description

Gives a short description (one line) of the purpose of the Test Script.

Example

```
#$DESCRIPTION Direct convention using T=0
```

#$REFERENCE — Test Script Directive

Synopsis

```
#$REFERENCE Text
```

Description

Indicate the reference of the document in which the Test Script is specified. This may contain a document title, chapter numbering, etc. The exact content of the information depends on the user process and environment.

Example

```
#$REFERENCE 1CE.060.00, 1CE.061.00
```

#$CRITICITY **Test Script Directive**

SYNOPSIS

#$CRITICITY *Level*

DESCRIPTION

Indicate the default criticity of the Test Case the Test Script is attached to. The criticity level is displayed in the Test Report. If needed, this value can be changed dynamically when the Test Script returns its verdict.

EXAMPLE

```
#$CRITICITY MAJOR
```

#$BREAK_IF_FAILED **Test Script Directive**

SYNOPSIS

#$BREAK_IF_FAILED

DESCRIPTION

Force the parent sequence to terminate if the verdict of the Test Case is FAILED or INCONCLUSIVE, as if the Test Case was the last of the sequence.

#$ABORT_IF_FAILED **Test Script Directive**

SYNOPSIS

#$ABORT_IF_FAILED

DESCRIPTION

Force the test suite to terminate if the verdict of the Test Case is FAILED or INCONCLUSIVE.

5.3 Writing a PERL Test Script

5.3.1 Test Script Layout

A Test Script is materialized with a PERL module providing some functions in which the program of the Test Case is written. In order to perform its job, a Test Script use the resources of the Test Execution Environment (the Test Engine Library and the Test Feature Library).

Figure 5.1Error: Reference source not found contains an explained example of PERL Test Scripts. It shows the different blocks that compose the Test Script module:

- In order to characterize the Test Case in the Test Report, some TestFarm-dedicated directives may be inserted in the script.
- The module header declaration must be given a package name. You can choose any valid name, provided it is unique among all the scripts of the Test Suite.
- Some “use” statements are needed to import the resources of the Test Engine library (packages `TestFarm::Engine` and/or `TestFarm::Trig` presented in the *TestFarm*

System Reference Manual [1]) and the Test Feature library. In the example below, the Test Feature Library is named `MyTestSystem`.

- You may have to put some other "use" statements to import other PERL packages, depending on your environment (package `MyTestLib` in the example below).
- The Test Case program resides in the `TEST()` function.
- The `TEST()` function returns the verdict of the Test Case, but it may return nothing if the `VERDICT()` function is present. As an option, the verdict may be accompanied with a criticity level that overrides the default criticity of the Test Case. See section 5.3.2 for details.
- As in any PERL package, in order to satisfy the package loading mechanism, the module file should be terminated with a "`1;`" statement.

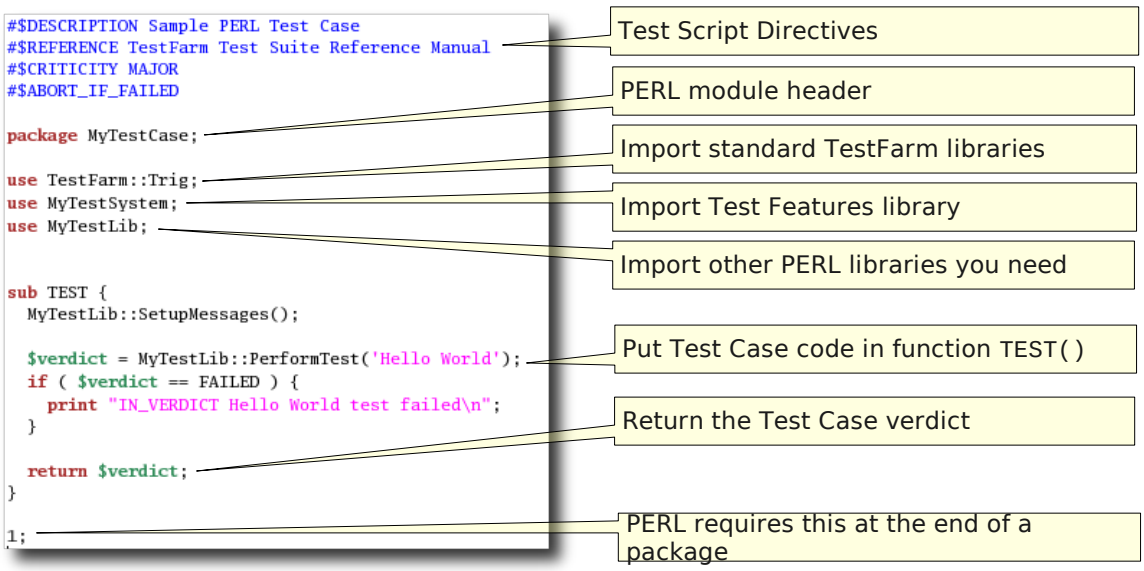


Figure 5.1: Layout of a PERL Test Script

5.3.2 Test Script Functions

This section describes the standard function provided by a Test Script PERL module. These function are recognized and invoked when the Test Suite is executed.

TEST — **Test Script Standard PERL function**

SYNOPSIS

```
TEST();
$verdict = TEST();
($verdict, $criticity) = TEST();
```

DESCRIPTION

The `TEST` function is mandatory: it contains the body of the Test Case program.

RETURNED VALUE

If the `VERDICT` function is present, the `TEST` function returns nothing.

If the VERDICT function is missing, the TEST function returns the Test Case verdict, accompanied if needed with the verdict criticity. If no criticity value is returned, the default Test Case criticity is taken into account. The default criticity is specified by the #$CRITICITY directive (see 5.2).

The standard values for verdict and criticity are imported from the Test Feature Library.

IMPLEMENTATION

```
sub TEST {
  … (Put the Test Case program here) …
  # Return Nothing if a VERDICT() function is present
}
```

or

```
sub VERDICT {
  … (Put the Test Case program here) …
  # Verdict:   PASSED,FAILED,INCONCLUSIVE,SKIP
  return $Verdict;
}
```

or

```
sub VERDICT {
  … (Put the Test Case program here) …
  # Verdict:   PASSED,FAILED,INCONCLUSIVE,SKIP
  # Criticity: NEAR,LIGHT,SERIOUS,MAJOR,CATASTROPHIC
  return ($Verdict, $Criticity);
}
```

VERDICT	**Test Script Standard PERL function**

SYNOPSIS

```
$verdict = VERDICT();
($verdict, $criticity) = VERDICT();
```

DESCRIPTION

The VERDICT function may be omitted if the TEST function returns a verdict. If the TEST function returns nothing, it is required to put a VERDICT function.

This function is intended to perform some post-processing after the Test Case program is completed. Typically, this consists in analysing the Local Log File or any other data produced during the test, in order to produce a verdict. If no post-processing is necessary, the verdict can be directly returned from the TEST function, and the VERDICT function can be omitted.

☞ If the VERDICT function is present, the verdict it returns overrides that returned by the TEST function.

RETURNED VALUE

The VERDICT function returns the Test Case verdict, accompanied if needed with the verdict criticity. If no criticity value is returned, the default Test Case criticity is taken into account. The default criticity is specified by the #$CRITICITY directive (see 5.2).

The standard values for verdict and criticity are imported from the Test Feature Library.

IMPLEMENTATION

```
sub VERDICT {
  … (Put Test Case post-processing program here) …
  # Verdict: PASSED,FAILED,INCONCLUSIVE,SKIP
  return $Verdict;
}
```

or

```
sub VERDICT {
  … (Put Test Case post-processing program here) …
  # Verdict:   PASSED,FAILED,INCONCLUSIVE,SKIP
  # Criticity: NEAR,LIGHT,SERIOUS,MAJOR,CATASTROPHIC
  return ($Verdict, $Criticity);
}
```

5.4 Writing a WIZ Test Script

5.4.1 The WIZ-script Philosophy

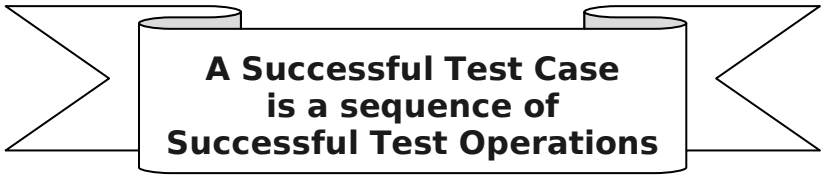
In order to speedup the development and maintenance of Functional Test Suites, the TestFarm development platform proposes the WIZ-script programming model. The TestFarm Wizard Tools provide an automatic code generator that produces PERL scripts from a dedicated macro-language.

The following highlights summarize the WIZ-script approach:

- Make high-quality **Functional Test Suites** using a well-specified and simple programming model.
- **Easy to Write**: People without high programming knowledge can write Test Scripts. More skilled people can focus on defining the Test Operations macros.
- **Easy to Read**: Test Scripts are easy to read and easy to maintain. They are easily understandable by the project management and quality assurance people.
- Bringing a high level of **Re-usability** by capitalizing elementary sequence items in a WIZ library.
- Test Scripts are way shorter than in their native PERL form.

5.4.2 The WIZ-script Programming Model

The WIZ-script programming model can be summarized as:



This means that a WIZ-script obeys the following rules:

- **The Sequence Rule**: A WIZ-script is a flat sequence Test Operations. No jumps or loops can be performed among the Test Operations. A Test Operation is materialized by a slice of PERL code stored in a WIZ-def file.
- **The Success Rule**: A Test Case is PASSED if all the Test Operations are PASSED.

- **The Failure Rule**: If a Test Operation fails (i.e. is not PASSED), the script is aborted and returns the non-successful verdict (which could be FAILED, INCONCLUSIVE or SKIP).

5.4.3 WIZ-script Layout

A WIZ-script is a list of Test Operations that are executed sequentially in respect of the WIZ-script programming model described in 5.4.2. Each Test Operation is implemented with a macro of PERL code defined in a WIZ-def file. The content of a WIZ-def file is described in section 5.5.

A WIZ-script is a text file named with the "`.wiz`" suffix.

Comments begin with a "`#`" character and finish at the end of a line.

Each line of this file contains a statement. Three types of statements may be put in a WIZ-script:

Type of Statement	***Syntax***	***Description***
Test Script Directive	`#$DESCRIPTION` *`text`* `#$REFERENCE` *`text`* `#$CRITICITY` *`level`*	Provide information for the Test Report: refer to section 5.2
Test Operation	*`keyword parameters`* …	Expands the macro defined in the WIZ-def file "*`keyword`*`.wizdef`", using the specified list of parameters. The parameter list is separated with spaces or tabs.
WIZ-script Inclusion	`INCLUDE` *`script`* …	Includes the Test Operations from the WIZ-script "*`script`*`.wiz`". Several files can be included from the same INCLUDE statement.

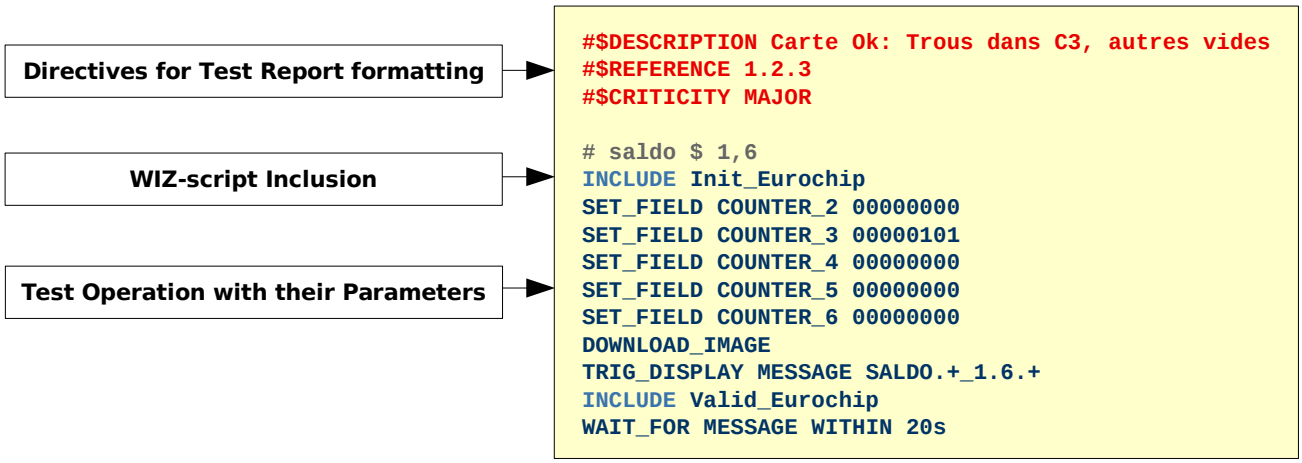


Figure 5.2: Layout of a WIZ-script file

5.5 Writing a WIZ-def macro

5.5.1 Building a WIZ-script from WIZ-def macro expansion

A WIZ-script is converted into a PERL-script module by expanding and assembling WIZ-def macros defined in a collection of WIZ-def files. A WIZ-def file can be considered as a slice of PERL script.

- The WIZ-def files are named with the "**`.wizdef`**" suffix.

- The WIZ-def files are stored in the **WIZ Definition Library**. By default, this library is stored in a workspace subdirectory named "**wiz**". This default organisation may be changed by setting the TESTFARM_WIZLIB environment variable with a list of directories in which the WIZ-def files can be found.

The generated PERL-script module is a concatenation of a PREAMBLE WIZ-def, followed by the WIZ-def macros referenced in the WIZ-script, and terminated by a POSTAMBLE WIZ-def. While concatenating the WIZ-def files into the resulting PERL-script, the macro expansion tool replaces the parameters passed from the WIZ-script with their actual values.

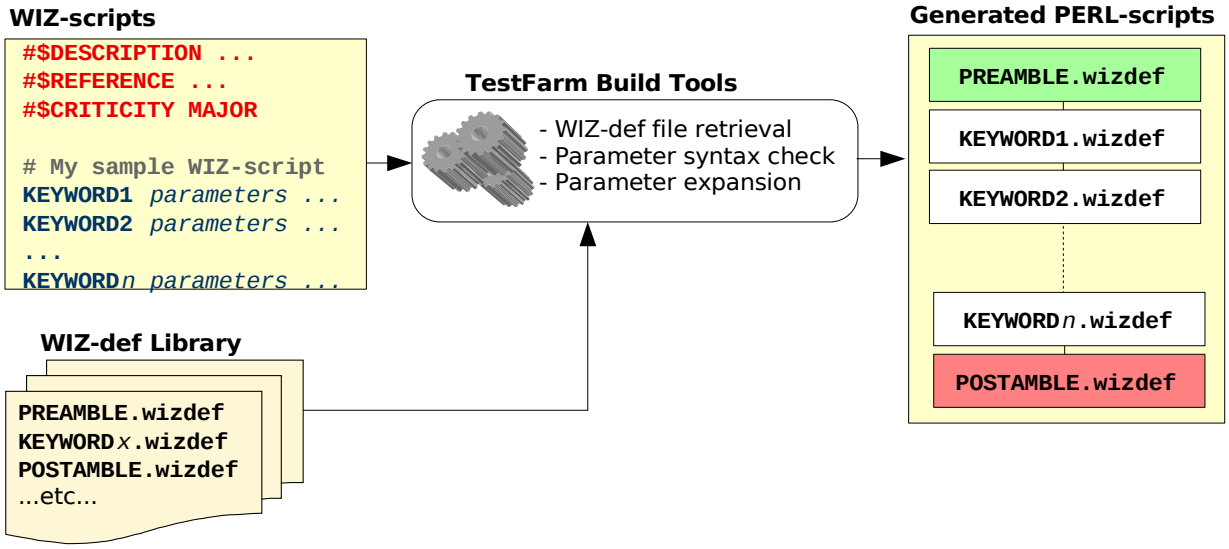


Figure 5.3: The WIZ-script macro-expansion process

5.5.2 What to put in a WIZ-def

The goal of a WIZ-def is to translate a Test Specification document into a Test Script in the most direct and readable manner.

A WIZ-def implements **a reusable Test Operation**, which may be specific to the context of the Product Under Test. Whereas the Test Feature Library provides simple actions which are bound to the structure of the Automated Testing System, the WIZ-def are aware of the basic features of the Product Under Test. A WIZ-def defines a high-level Test Operation that may combines several test features together.

A WIZ-def file contains a **PERL code pattern** that drives the Test Interface by invoking the Test Feature Library resources. It also manages the synchronization of the Test Interfaces by using the trigger features from the Test Engine Library. Please refer to the TestFarm System Reference Manual [1] for details about these libraries.

As a final role, a WIZ-def produces a verdict by setting the variable **$verdict** with values PASSED, FAILED, INCONCLUSIVE or SKIP. This verdict controls the execution of the Test Script in respect of the WIZ-script programming model presented in 5.4.2.

5.5.3 WIZ-def Layout

A WIZ-def is a text file named with the ".wizdef" suffix.

Comments begin with a "#" character and finish at the end of a line.

This file is contains three main sections:

- The declaration of the macro parameters;
- Some complementary information that helps generating a complete documentation automatically;
- The body, which is a PERL code pattern, optionally completed with some directives for implementing conditional sections, including WIZ-def files, or performing a dynamic code generation handled by a PERL function.

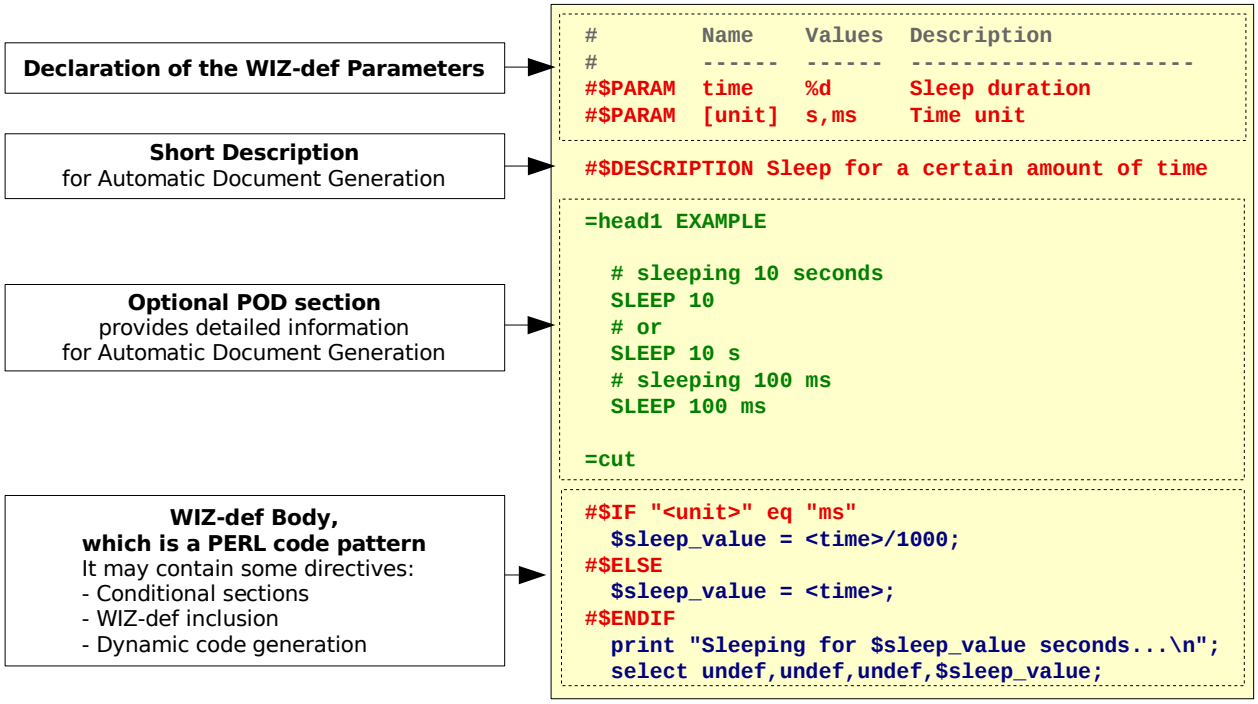


Figure 5.4: Layout of a WIZ-def file

5.5.4 Parameters Declaration

When referenced from a WIZ-script, a WIZ-def can be passed some parameters. This gives the possibility to define a macro that generates a variable code, depending on the values of the parameters. The parameters are declared using the "**#$PARAM**" directive. When processing the WIZ-script macro expansion, a syntax check is performed regarding the declaration, and a compilation error occurs if the value passed from the WIZ-script does not match the declaration. Moreover, the parameter declaration is used for automatically generating a User's Manual of the whole WIZ-def collection. Chapter Error: Reference source not found gives some details about the automatic WIZ-def document generation.

#$PARAM	**WIZ-def Directive**

SYNOPSIS

#$PARAM *Identifier* *Format* *Description*

DESCRIPTION

Declare a parameter *Identifier*. If the *Identifier* argument is in square brackets (like in "[MyId]"), the parameter is assumed to be optional.

The argument *Format* is a comma-separated list of possible values for the parameter. A value may be given in a discrete manner, or by specifying its data format.

Format	Meaning	Format Example	Value Example
value	Discrete value	`MANUAL,AUTO`	`MANUAL`
`%d` `%d:`*min*`-`*max*	Integer value, optionally ranging from *min* to *max*	`%d` `%d:0-100`	`34`
`%s`	The value is a character string, which may be passed within double quotes.	`%s`	`"Hello World"`
`/`*regex*`/`	The value should match the *regex* PERL-style regular expression	`/^\d+$/`	`45`
...	Collect all the trailing values of the parameter list into a single character string. This helps implementing a variable-length parameter list.	`...`	`Hello World`

☞ When a parameter is optional (i.e. when *Identifier* is specified within square brackets), its default value is the first item of the *Format* value list. Consequently, the first item of the format list must be a discrete value. If no explicit default value is given this way, an error message is displayed when compiling the WIZ-scripts using the TestFarm Wizard Tools.

☞ When declaring a variable-length parameter list, the last `#$PARAM` directive should be given the format "`...`".

The argument *Description* is short text line describing the purpose of the parameter. This text will be inserted in the automatically generated WIZ-def documentation.

Example 1: A List of Discrete Values

```
#$PARAM mode  ACCEPT,REJECT  What to do if an error occurs
```

In this configuration, the only values accepted for the parameter `mode` are "`ACCEPT`" or "`REJECT`". Any other values will display an error message when compiling the WIZ-script using the TestFarm Wizard Tools.

Example 2: Optional parameters

```
#$PARAM time    %d    Sleep duration
#$PARAM [unit]  s,ms  Time unit
```

In this configuration, the WIZ-def accepts one or two parameters. The first parameter `time` is an integer. The second parameter `unit` is optional: it may take the values "`s`" (default) or "`ms`".

Example 3: Variable-length parameter list

```
#$PARAM message  %s   Expected message to be displayed
#$PARAM others   ...  Other messages
```

In this configuration, the WIZ-def accepts two or more parameters. The first parameter `message` is a character string. All the other parameter values are gathered into a character string as parameter `others`.

5.5.5 Embedded Documentation

It is recommended to put some information in a WIZ-def file, in order for the TestFarm Wizard Tools to automatically generate a clear a complete documentation. This documentation constitutes the User's Manual of the whole WIZ-def collection available in a Test Suite workspace.

By default, the documentation includes a description of the WIZ-def parameters. It is also possible to declare a short description text using the **`#$DESCRIPTION`** directive.

Moreover, as an additional compatibility with the PERL culture, the document generator extracts all the **POD** sections from the WIZ-def file, and append them to the documentation. POD sections are delimited with the "**`=pod`**" and "**`=cut`**" markers. Please refer to the `perlpod` manual for a detailed description of the simple POD document format.

The TestFarm Wizard Tools automatically generate this documentation in two format:

- The document file `wizdef.man` is in the classical Unix *man* format. It can be displayed using the command "`man ./wizdef.man`".
- The document file `wizdef.html` is in HTML format, and can be viewed with any web browser.

#$DESCRIPTION	**WIZ-def Directive**

SYNOPSIS

`#$DESCRIPTION` *`Text`*

DESCRIPTION

Gives a short description (one line) of the purpose of the WIZ-def macro.

5.5.6 Static Code Pattern

Any parts of a WIZ-def file that are not a directive or a POD documentation section are considered to be the WIZ-def body. The body contains the PERL code pattern, in which it is possible to reference the WIZ-def parameters with the syntax "<*`Identifier`*>".

After parameter replacement and conditional section resolution, the TestFarm Wizard Tools dump the PERL code pattern in order to generate an executable PERL-script.

5.5.7 Conditional Sections

Some directives provide the possibility to implement conditional sections of code. These conditional section directives are intended to be used within WIZ-def body.

It is also possible to use them around `#$PARAM` directives, thus bringing the possibility to implement conditional parameter declarations. This feature must be used with care, as it makes the WIZ-def a bit more complicated to understand.

#$IF	**WIZ-def Directive**

SYNOPSIS

`#$IF` *`Expression`*

DESCRIPTION

Enter a conditional code section, which is enabled if *Expression* returns a TRUE (≠0) value. *Expression* should be in PERL syntax, and may reference parameter identifiers with the "<*Identifier*>" syntax.

The conditional code section terminates when another conditional section directive is encountered. It is possible to nest conditional sections.

EXAMPLE

```
#$IF <mode> eq "ACCEPT"
print "ACCEPT mode enabled\n";
#$ENDIF
```

#$ELSE — WIZ-def Directive

SYNOPSIS

#$ELSE

DESCRIPTION

Enter a conditional code section, which is enabled if the previous #$IF or #$ELSIF directive is FALSE (0).

EXAMPLE

```
#$IF <mode> eq "ACCEPT"
print "ACCEPT mode enabled\n";
#$ELSE
print "ACCEPT mode disabled\n";
#$ENDIF
```

#$ELSIF — WIZ-def Directive

SYNOPSIS

#$ELSIF *Expression*

DESCRIPTION

Enter a conditional code section, which is enabled if the previous #$IF or #$ELSIF is FALSE, and *Expression* returns a TRUE value. The syntax of *Expression* is similar to the #$IF directive.

EXAMPLE

```
#$IF <mode> eq "ACCEPT"
print "ACCEPT mode enabled\n";
#$ELSIF <mode> eq "REJECT"
print "REJECT mode disabled\n";
#$ELSE
print "Unknown mode\n";
#$ENDIF
```

#$ENDIF — WIZ-def Directive

Synopsis

```
#$ENDIF
```

Description

Ends a conditional section.

5.5.8 WIZ-def Inclusion

#$WIZCALL **WIZ-def Directive**

Synopsis

#$WIZCALL *Keyword* *Parameters* …

Description

Include the WIZ-def from file *Keyword*.wizdef, and give it the parameter values *Parameters*. The parameters are checked, and the body is expanded the same way as if the included WIZ-def was referenced from a WIZ-script.

Example

```
#$WIZCALL SLEEP 350 ms
```

5.5.9 Dynamic Code Generation

When a macro-expansion mechanism is not enough for generating a PERL Test Script from a WIZ-script, one may need to invoke a small program that performs some computation in order to check the parameters and generates some code dynamically.

This feature gives for instance the possibility to perform strengthened checks during the WIZ-script compilation phase, in order to detect potential error as soon as possible in the Test Suite development cycle. This may improve significantly the efficiency of the Test Script development teams, along with the overall quality of the Test Suite.

In order to display some error and warning messages during the compilation process, the code generation program can call the functions ERROR() and WARN() imported from the **TWiz** package, included in the TestFarm platform. These functions the file location information and display the error/warning messages with the standard format of the TestFarm Wizard Tools.

#$WIZGEN **WIZ-def Directive**

Synopsis

#$WIZGEN *Module*::*Function*(*Parameters*)

Description

This directive calls a dynamic code generation *Function* from a PERL *Module* during the WIZ-script compilation process. The PERL module file *Module*.pm must resides in the WIZ Definition Library, which by default is the “wiz” subdirectory of the workspace.

It is possible to pass some *Parameters* to the function, which may be references to the WIZ-def parameters passed with the “<*Identifier*>” syntax.

The value returned by the function can be retrieved in the "<__ret__>" built-in parameter (section 5.5.10).

WIZ-def File

```
#        Name    Format       Description
#        ------  -----------  ---------------------
#$PARAM  file    %s           UC parameter file

#$DESCRIPTION Sample WIZ-def with Dynamic Parameter Check

=head1 EXAMPLE
(Documentation section snipped)
=cut

#$WIZGEN Check::AcqConfig("<file>")

  print "Configuring acquisition from <__ret__>\n";
  SetAcqConfig("<__ret__>");
```

Check.pm PERL module

```
package Check;

use TestFarm::Wiz;

sub AcqConfig {
  my $file = "$dirname/$_[0]";
  if ( ! -e $file ) {
    WARN("Cannot find parameter file \"$file\"");
  }
  return $file;
}
```

Figure 5.5: Example of Dynamic Parameter Check implementation

5.5.10 Implicit Built-in Parameters

Some implicit built-in parameters can be referenced in the WIZ-def body using the "<*Identifier*>" syntax. These parameters provide some information about the Test Script that is calling the WIZ-def.

Identifier	*Content*
__packname__	The name of the generated PERL package
__filename__	The base name of the generated PERL-script file
__dirname__	The directory name of the generated PERL-script file
__ret__	The value returned by the last #$WIZGEN function call

6 Test Script Validation Management

6.1 Objectives and Principles

In order to have a strong Test Suite development process, it is necessary to follow a validation process that tracks whether the Test Scripts conform to the Testing Requirement they come from.

Validating a Test Script consists in checking its conformance to the Testing Requirements through code reviews and experimental Test Suite executions. It is then possible and to give it a **Validation State** indicating how mature and reliable it is. Afterwards, if the script is modified, the Validation State is cancelled, and the validation process must be re-applied, in order for the script to be considered validated again.

The Test Suite User Interface allows to edit the Validation Level for each Test Script. Refer to the TestFarm User's Manual for more information.

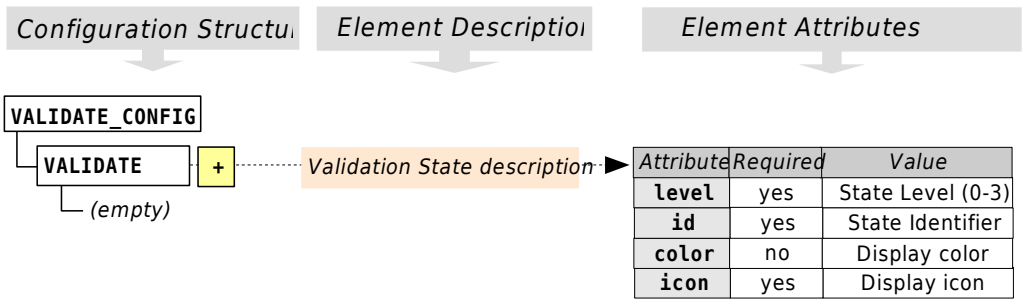
6.2 Validation States

The Validation State is shown in the Test Report, thus alerting the project manager about the reliability of the Test Cases.

The standard validation states are defined in the file "`/opt/testfarm/lib/validate.xml`":

Level/State		***Description***
0	**Unavailable**	The Test Script is not validated: development in progress
1	**Delivered**	The script developper considers that it is ready for execution
2	**Reviewed**	The script has been reviewed and experimented, and is accepted by the Test/Quality manager
3	**Approved**	The script has been sufficiently used to assumed it is reliable

If needed, the system administrator can redefine the validation states in file "`/var/testfarm/lib/validate.xml`", taking into account the quality management rules of the company.



6.3 Implementation

The Validation State is maintained for the PERL Test Scripts only, because they are the ultimate result of the Test Suite development chain. When a WIZ-script or a WIZ-def is modified, this will eventually impact the resulting PERL code.

When the Validation State of a Test Script is edited, the Test Suite User Interface updates a check-list file ".check" containing the state information and the MD5 checksum of the script being validated. There is one check-list file per (sub-)directory in the Test Suite workspace. The check-list is a text file containing one line per validated script of the directory it is stored in. Non-validated Test Scripts are removed from the check-list file.

Each line of the check-list file contains 4 fields separated by a space character:

MD5 sum	File	Level	Date	Operator
MD5 sum of the PERL script: 128-bit hexadecimal	Script file name	Validation level: 1, 2 or 3	Validation date: DDmmYYYY	Name of the person who edited the Validation State

6.4 Reporting the Validation State

The Validation State of a Test Case appears in two manners in the Test Report, depending on the activated Test Report layout options:

- The name of the Test Case is displayed in parenthesis if its validation level is not APPROVED.
  This can be disabled by unselecting the option "Names in parenthesis if test not validated" in the "General Options" page of the report configuration menu.
- The full validation state information are displayed in the "Output Dump" section of the Test Report. This can be disabled by unselecting the option "Show validation state" in the "Output Dump" page of the report configuration menu.

7 Glossary of Terms

Test Engine
Test Interface
Test Feature
Instrument
Command Interpreter
Interface Library
Test Suite
Test Tree
Test Case
Test Script
Test Sequence
Scenario
Tree Root File
Tree Branch File
Wizard Tools
SUT, System Under Test, Product Under Test
XML, eXtensible Markup Language
DTD, Document Type Definition
XSL, eXtensible Stylesheet Language

8 References

[1] IP070035-en : TestFarm System Reference Manual.

List of Figures