PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 01/12

The Talisman
C++ Unit Testing Framework

Arndt von Staa

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO – BRASIL

Monografias em Ciência da Computação, No. 01/12 **ISSN: 0103-9741**
Editor: Prof. Carlos José Pereira de Lucena January, 2012

The Talisman C++ Unit Testing Framework

Arndt von Staa[1]
arndt@inf.puc-rio.br

Departamento de Informática
Pontifícia Universidade Católica
22453-900 Rio de Janeiro, Brasil

Resumo

Neste documento descrevemos resumidamente a arquitetura do arcabouço de apoio ao teste de unidades redigidas em C++ utilizado no projeto Talisman 5. Detalhes podem ser encontrados na documentação de cada módulo (arquivos **`.h`**). O arcabouço foi construído sem fazer uso de bibliotecas ou arcabouços voltados para o ambiente Windows. Tampouco faz uso de *templates* do C++. O arcabouço pode ser integrado com aplicações, permitindo, assim, o teste de uma ou mais unidades ou funcionalidades no contexto de produção. O arcabouço utiliza uma linguagem script muito simples para redigir os testes. A linguagem assemelha-se a *assembler*, no sentido que cada comando corresponde a uma ação a realizar e um ou mais parâmetros. Em geral o comando corresponde a uma função a testar e os parâmetros dados para esta função e o resultado esperado ao executá-la come esses dados.

Palavras chave: Arcabouço de apoio a testes, C++, teste de unidades, teste de integração, script de teste, biblioteca de apoio.

Abstract

This document describes, in a very abridged way, the organization, tools, usage and conventions of the Talisman C++ unit test framework. Further details about the framework, as well as details about each module, class and method can be found in the corresponding header files. The present framework uses a very restricted set of C++ capabilities. It uses neither templates, nor MFC – Microsoft Foundation Classes or other class libraries. It may be integrated with an executable application, allowing to test modules in a production setting. It interprets a very simpleminded script language instead of coding the test cases directly in C++. The script language is similar to assembler, in the sense that it is line oriented and each executable or declarative line contains a command followed by a list of zero or more parameters. Usually the test command corresponds to the method to be tested and the parameters correspond to the parameters to be used when calling this method as well as values that will be compared with the outcome of the method execution.

Keywords: Test support framework, C++, unit test, integration test, test script language, support library.

[1] Trabalho apoiado por: CNPq, Bolsa de Produtividade 306802-2008-2, e Auxílio 479344-2010-8

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
e-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Table of Contents

1. Introduction

This document describes, in a very abridged way, the organization of the Talisman 5 test support framework. Details of each module and method can be found in the corresponding header files.

Talisman 5 [Staa, 2011] is a software engineering meta-environment. Its infra-structure is being developed in C++. The test environment requires a support library that will be part of the meta-environment. Talisman 5 is being designed and developed using Talisman 4.4 [Staa, 1993]. This is a rudimentary software engineering meta-environment written in C for MS-DOS and that allows, among others, structured editing as well as model based code generation and editing. It allows also the generation of code fragments that will be interspersed with written code. This feature provides means to selectively insert test support code into the modules under test. Furthermore, given the header file of the module to be tested, it allows generating the specific test command list, as well as the test script interpreter skeleton. Talisman 4.4 can be seen as a prototype of Talisman 5.

Parts of the code have been written before 1998, i.e. before the C++ standard was available. To assure portability considering different compilers, a very restricted subset of the C++ language has been used to implement the framework. The code contains no templates, makes no use of the Microsoft Foundation Classes and uses only the C language library. Hence, the framework has still very a strong C flavor. A new version is being planned to be implemented using the C++ 2011 [ISO/IEC 14882:2011] standard once compliant compilers show up.

Several unit test frameworks exist for C++ [see *Tools* in the reference section]. Almost all of them use the full set of capabilities of the C++ language. The present framework deliberately uses a restricted set of capabilities:

- Tests are written in a very simpleminded script language instead of coding the test cases in C++.
- The script language is similar to assembler, in the sense that it is line oriented and each executable or declarative line contains a command followed by a list of zero or more parameters. Usually a test command corresponds to the method to be tested or executed and the parameters correspond to the parameters to be used when calling this method as well as values that will be compared with the outcome of the method execution.
- It allows script commands to be extended by a list of one or more parameter commands. These commands allow the use of tabular data as part of a script command.
- It allows reading from **`#include`** files. **`#include`** files may refer to other **`#include`** files, up to a nesting depth of 10.
- It allows testing modules in the context of the component to which they belong. Hence, the test script language may use methods of classes that have already been accepted and are needed to provide context for the module under test.
- It allows test support modules to be integrated with the application. In other words, the framework does not impose the use of its own **`main`** module.

- The testing environment provides means to control recompilation and retesting of all modules that constitute a given component.
- The test framework is tested using itself. This provides maintenance support for the framework itself.
- The scripting language does not provide functions nor control structures. Such features should be implemented by the scripting language interpreter and triggered by a specific script command.

One of the goals of the testing environment is to provide means for developing a maintenance subsystem as a byproduct of the development.

1.1. Context, the Talisman 5 project

The Talisman 5 project aims at developing a software engineering meta-environment. It supports among others: model checking; transformations from one representation to another, even if written in different representation languages; various forms of generation; and maintenance involving models, code and scripts. Its main characteristics are:

- Talisman stores all elements as hyper-objects.
- Talisman hyper-objects contain several attributes (e.g. names, strings, text fragments, relations, text fragments containing references to other hyper-object attributes, record structures), each of which corresponds to an OO programming language object.
- Attributes are stored in a proprietary object oriented data base, called *attribute base*, storing very small granularity data elements.
- All references between objects are virtual addresses. The attribute base provides means to convert these addresses to real addresses when manipulating objects corresponding to attributes.
- Attributes and references may be versioned. Versioning is controlled directly by the attribute base.
- The attribute base may be partitioned into several different, distributed and interdependent bases. Each one of them may possibly be shared by several projects. Finally the collection of attribute bases may be geographically distributed.
- The set of attributes, including relations, defined for a given hyper-class may be changed dynamically, even while development of one or more target systems is in progress.
- Representations may share hyper-objects.
- Hyper-objects and attributes that compose graphical or textual representations may be generated, verified, validated and transformed into other hyper-objects and attributes.
- Code may be generated and/or composed directly from the contents of the attribute base.
- Representations, graphical or textual, are rendered each time they are accessed. No physical representation image is saved.
- The set of representations form a hyper-document.
- Maintenance and test support are part of the environment's software system.

- One of Talisman's 5 goals is to support reverse engineering as well as reengineering.
- Each software component (collection of modules each of which implements exactly one single purpose) of the meta-environment must have its own specific testing module Furthermore, a reconstruction sequence must have been defined that recompiles and tests incrementally each partial build of the component.

2. System architecture overview

We are studying the development of systems that implement not only their required functionality, but implement also a maintenance sub-system that supports complex evolution. The goal is to develop highly portable, maintainable[1] and dependable code. Furthermore, code should pay specific attention to testability, diagnosticability[2], and debuggability[3]. Hence, for the purpose of allowing the use of standard debuggers we require the ability to run tests in a strict C++ environment. We also require the ability to run tests in a mixed language environment, assuming that the C++ code is almost correct. The mixed language testing environment is necessary to assess the dependability of the system when operating in a hybrid environment. Finally, when evolving a system, it should be possible to test parts of it while using the system.

Another goal of our study is the development of self checking code. Self checking is achieved using specifically designed redundancy, as well as highly instrumented code [Deveaux et al, 1999; Staa, 2000]. Since instruments might interfere with performance, our aim is to develop self checkers that eventually run in a separate low priority thread that is unaware of the processing state of the system. This obliges programmers to follow a defined discipline. It also requires control over all allocated dynamic data spaces. A side effect of this goal is the ability to control memory and other resource leaks.

We assume that the application development follows an incremental path. Hence, tested modules are accreted to the set of accepted modules as they are readied. However, we do not test always module units in total isolation. Possibly, as shown in Figure 1, more than one module might be tested at a given time. Furthermore, they may interact with already accepted modules. That means that modules under test might interact with stubs or mock modules [Hunt and Thomas, 2003], as well as with already accepted modules and other modules under test. When retesting, the build used to test a given module must be reconstructed. Therefore, all required stubs and mock modules must be available when rebuilding a specific test build. However, when the system or component under test is fully developed, it is usually better to use production modules instead of mocks or stubs. This reduces the need to maintain development support modules. Obviously, after accepting a production module that replaces a stub or mock, all tests of builds that have used the stub or mock to be eliminated should be reviewed allowing the corresponding production module to participate in the build instead of the eliminated stubs or mocks. Thus only mock modules that perform test specific actions, as for example throwing exceptions whenever required by the test script, will have to remain available.

---

1 ***Maintainability*** – involves both the elimination of defects, as well as the evolution of the system. When evolving a system sometimes a major restructuring may be needed to remove design defects due to structural deterioration.

2 ***Diagnosticability*** – is the ability of providing effective support to identify the causes of a detected failure.

3 ***Debuggability*** – is the ability of providing effective support to completely remove the causes of a detected failure.

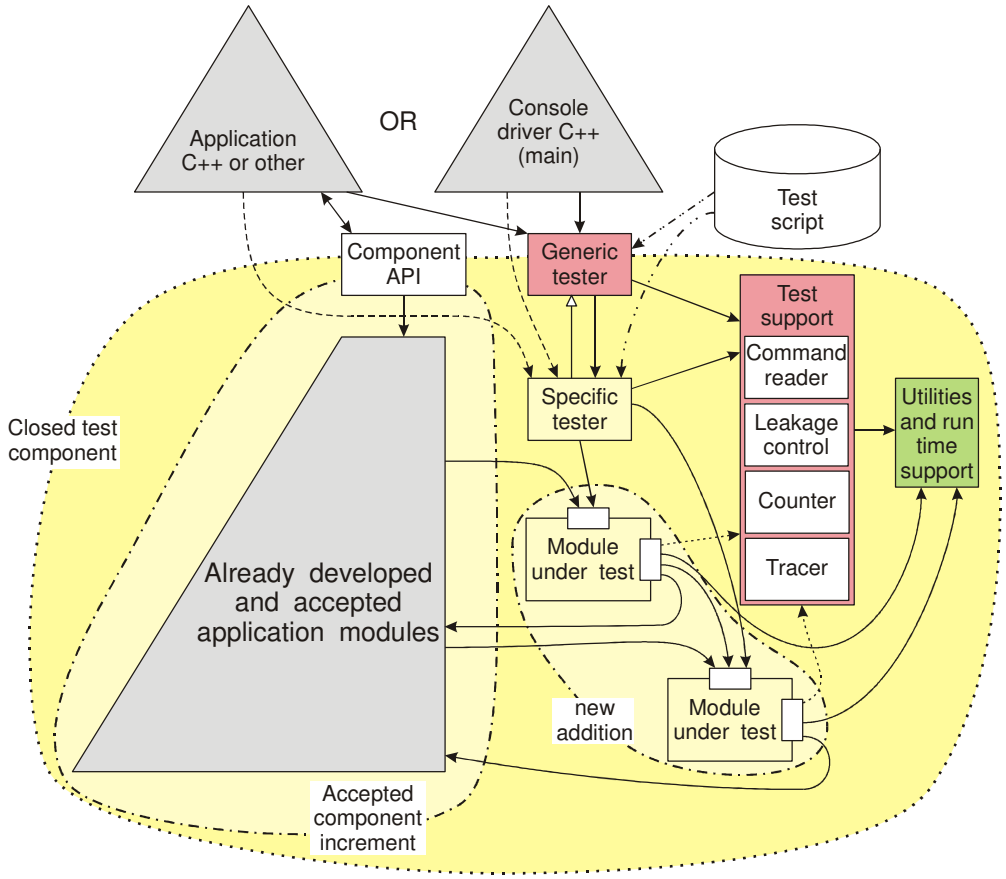

Figure 1. Overview of the test support framework

The goal of this approach is to create a maintenance environment as a complementary component while developing the production component. Hence incremental integration control tools and scripts, as well as the testing tools and scripts are part of the final delivered product. Detailed testing may be performed using code compiled for debugging, and less detailed tests may be used to test optimized production code. Of course, in several cases mock objects will still be needed to establish a faster or more precise testing context.

An example of test support created together with the production component is the test environment which now is tested using itself. Initially, while developing the bootstrap framework, it was tested using ad hoc testing methods. After the bootstrap version had been finished, specific test modules and much more accurate test scripts were developed. Currently, the test builds encompass the collection of modules that compose the testing framework and specific test modules, one for each of the modules comprising the framework. In some cases, though, the module under test must be factored out, since tests might interfere with the working of the framework if the module under test contains a defect. For example, modifying the hash table module must be performed on a factored out version of it, since if the new version is part of the framework and contains a defect, the framework would cease to work rendering all testing useless.

The standard test run driver is a simple command line C++ main program that reads test control command line arguments, constructs the specific test control object and triggers the start of the test. Thus it might easily be substituted by other modules as complex as whished. The test framework does not depend on the test run driver; it requires only that the specific test control object is created before the test begins.

The generic and the specific test modules, as well as the test support modules compose the test framework. The generic tester is an abstract class and, among other functions, acts as a façade for the test framework. To be able to use the generic tester, the specific tester must inherit from it and must be constructed before actual testing begins. The generic tester provides the usual comparison methods (assertions as they are called in CPPUNIT, and other *Unit test frameworks). The test framework provides dynamic memory control, passage counting and script command reader.

The script command reader reads test script lines and decomposes them into their lexical elements. The lexical rules of the test command tokens are similar to the C++ lexical syntax.

Dynamic memory control is used to control memory leakage as well as incorrect usage and deletion sequences. This module redefines the **`new`** and the **`delete`** operators. Furthermore, if specific instrumentation is added, it can verify the type correctness data space type and the implied type of pointers that refer to it.

Passage counting is used to perform a very simpleminded form of test coverage, and might be used to control path execution. To perform true test coverage control, more enhanced tools should be used.

The tracer provides several tools to generate execution traces and also runtime stack evolution traces. These tools depend on instrumentation inserted into the code, which is easily performed when developing with the aid of Talisman 4.4.

The test framework requires several utilities and support modules. These establish a virtual machine that provides a standard way of handling exceptions, strings, messages, BCD arithmetic, and simple auto-verifiable containers such as hash tables and doubly linked lists. If desired, this virtual machine might be used by an application that does not require the presence of the test framework.

3. Run-time support, utilities

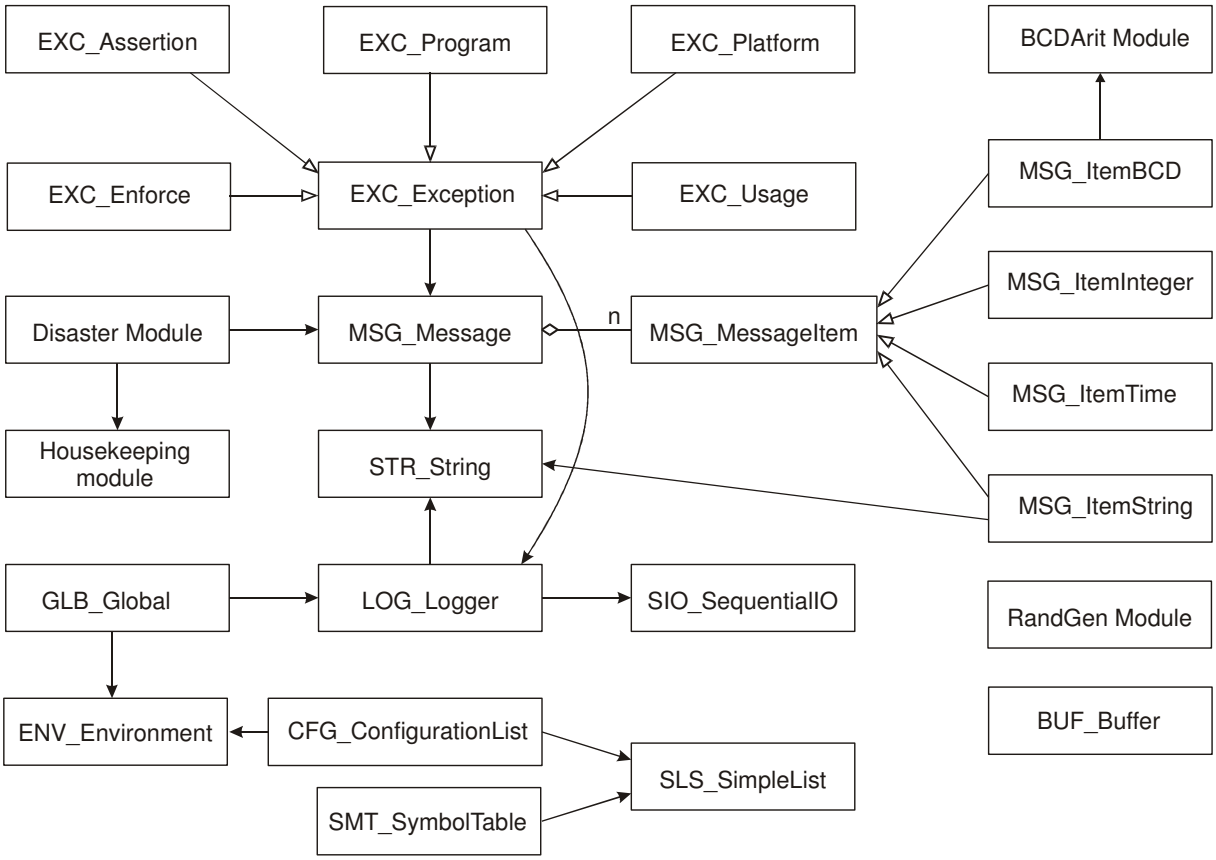

Figure 2. Architecture of the run-time support classes and modules

In this section we describe the run-time support library required by the test framework. This library may be used independently of the test framework. It has been designed to provide support to the Talisman 5 meta-environment. It establishes a run-time environment that shields applications from platform details, as well as providing generic services that almost all applications will need. The organization of the run-time support is depicted in figure 2.

A quick note about the diagram. Modules might contain classes and/or global functions whenever these are convenient. Some modules contain only global functions. These modules are identified by the text fragment "Module" in the name. When a module contains classes, only the classes are displayed in the diagram.

3.1. Modules

`BCDArit` Module[1] - this module implements binary coded decimal (BCD) arithmetic operations. For portability purposes it does not use assembly code. A BCD number is essentially an ASCII number where the first 4 bits have been removed from the character. This allows packing two decimal digits into a single byte. BCD numbers are portable since they are machine and persistence layout independent. Furthermore, performing a few BCD arithmetic operations may prove less expensive than

[1] All tools and modules have been developed using Talisman 4.4, which is a MS-DOS application and hence restricts the module names to be in an 8.3 format.

converting an ASCII string to a binary integer, performing the operation in binary, and converting the result back to an ASCII string. The encoding of the BCD numbers is signed magnitude. The size, in bytes of a BCD number may be chosen from 1 byte (1 BCD digit) to up to 6 bytes (11 BCD digits). The size (number of bytes used) of the number is part of the encoding, allowing mixed size operations. The module controls arithmetic and conversion overflows, assuring size correctness of its operations, even when mixed sizes are being used.

`Disaster` Module - this module implements a standard disaster handling function. Disasters correspond to unrecoverable failures. When such failures are observed the application must be canceled.
To be implemented: a standard way of registering rollback actions to be performed when a disaster is observed. Disaster handling might be issued by a call to the disaster handling function, as well as by throwing an exception. The disaster module attempts to roll back to some stable state (if defined), cancelling execution afterwards.

`Housekeeping` Module - *To be implemented*: this module implements several functions that record and destruct actions to be performed when cancelling the execution of the program. For example, when rolling back to a previous correct state is needed these actions establish how to perform this.

`RandGen` Module - this module contains functions that generate random numbers in accordance with a few distribution functions. It also contains a function that computes random permutations. This module has been designed to aid generating random test data.

3.2. Classes

`MSG_Message` - This class implements standard message objects. Message objects contain string ids instead of pointers to the strings or literal string values. These string ids identify the message. Hence it is possible to determine the meaning of a message by examining its string id and, if necessary, change it, adapting low abstraction level messages to the higher abstraction level ones as needed by human users or recovery functions. The strings referred to by messages must be contained in a string table, see the **`STR_String`** class below. Messages may contain up to 20 parameters. Parameters, called *message items*, are inserted in fields contained in the message string. Fields are identified by the sequence **`%n`**, where **`n`** is a number between 0 and 19. Items may be added to a message object at any time and in any order. Items may be inserted more than once into a same message string, and the order of the items in the string is independent of the order of the items in the message object. Finally, there may be more items than are fields in a message. However, missing items will be signaled as errors when rendering the message. The association between items and message fields is established by the item index. The type and formatting of an item is defined by the item class. Implemented in module **`MESSAGE`**.

`MSG_MessageItem` - This is an abstract class defining a message item. Each message item contains an item-id, a value type id and a value. The item-id is the index

of the item in the item table of the corresponding message object. The index is used to indicate where it should be expanded in the message string, or to provide access by methods that examine the content of a specific message item. The value type id allows implementing a data driven object factory that always constructs objects that are consistent with the value part of the message item. Another method defines how the item value is converted to an ASCII string. More item types may be defined as needed. For example, when developing recovery oriented software, exceptions must carry adequate information to allow correct recovery. Actual message items must specialize this abstract class. Implemented in module **MESSAGE**.

MSG_ItemBCD - This class specializes a message item to a BCD integer item. Implemented in module **MSGBCD**.

MSG_ItemInteger - This class specializes a message item to a binary integer item. Implemented in module **MSGBIN**.

MSG_ItemString - This class specializes a message item to a character string item. Implemented in module **MSGSTR**.

MSG_ItemTime - This class specializes a message item to a date and time item. Date and time are displayed in a format similar to **2007/11/24-15:50:37**. Implemented in module **MSGTIME**.

STR_String - This class implements self-verifying ASCII character string objects. It also provides methods that allow accessing strings defined in a string table. String tables are generated by the **GENSTRTB** tool. The corresponding string identifiers (string access keys) are generated by the **GENSTRID** tool. The input data for both tools are **STR_xxx.STR** files related to module **xxx**, and whose format is described by comments contained in the tool's code. The string table is partitioned into a memory resident part and a file (segment) resident part. Using string tables consistently throughout the application instead of string literals is a simple way of constructing easily localizable programs. Such programs might be converted to different locales without the need of recompilation, except for the **STRING** module, since it includes the memory resident part of the table. Implemented in module **STRING**.

EXC_Exception - This abstract class defines the standard Talisman 5 exception classes. It must be specialized by specific exception types. The programming conventions used by the Talisman 5 project requires that exceptions should correspond only to abnormal events (i.e. when the occurrence probability of the event is zero or almost zero). However, exceptions may also be used when it is known that the distance (number of nodes in the call graph) from the detector (**throw**) to the handler (**catch**) will certainly be long (3 or more). The distance is measured in terms of the number of methods to be unraveled on the stack, starting at the method that detects the problem and ending at the method that handles it. If an event is expected, even if it occurs rather seldom (e.g. end of file, or illegal file name, incorrect data in an input field), return conditions should be used instead of exceptions. Every exception object points to a message object, which contains the message id and the set of message

items (parameters) that should explain the cause of the exception. Exception objects always contain the source code line number and the module name where the exception was thrown. Finally, exception objects contain a context id. An exception is identified by the triple <*exception object type*, *context id*, *message-id*>. The context id and the message id allow a catcher to determine whether it should handle the exception or not. In this way it is possible to establish a strict binding between the thrower and the handler. Use the **`MayBeHandled(…)`** method to verify if a given catcher should handle the caught exception or not. The **`EXC_Exception`** class contains also methods to access the message id and to change it if necessary. The **`Exceptn`** module contains the function **`EXC_LogError(…)`** used for logging problems without throwing an exception. Implemented in module **`EXCEPTN`**.

`EXC_Assertion` - This class implements the standard Talisman 5 assertion exception. Assertions exceptions are thrown by the **`EXC_ASSERT`** macro, which is similar to the C++ **`assert`** function, but generates an exception instead of canceling the execution. However, these exceptions do not provide an explanation of the detected error. Since assertions are debugging tools, they should not be part of the production code; hence **`EXC_Assert`** throwers should always be enclosed in an **`#ifdef _DEBUG`** block. If production code should contain an active assertion, use the **`EXC_Enforce`** exception instead, see below. Assertion exceptions inform only the failing expression, the module name and the source line number containing the **`EXC_ASSERT`** macro call. Assertion exceptions may be caught by a **`catch EXC_Assertion * pExc`** handler. Implemented in module **`EXCEPTN`**.

`EXC_Enforce` - This class serves a purpose similar to **`EXC_Assertion`**, the difference being that the code throwing **`EXC_Enforce`** exceptions should remain active in the production code. **`EXC_Enforce`** exceptions are thrown by the **`EXC_ENFORCE`** macro. **`EXC_Enforce`** exceptions allow triggering roll back functions or even recovery functions, in which case they should be caught by an appropriate **`try`** … **`catch`** block. **`EXC_Enforce`** exceptions may be caught by a **`catch EXC_Enforce * pExc`** handler. Use **`EXC_Enforce`** when controlling errors that, in principle, should never occur, even when considering that programs could still contain defects, or be subject to user error or platform malfunction. Such controls serve the purpose of preventing the propagation of errors to persistent structures. Enforce exceptions inform only the failing expression, the module name and the source line number containing the **`EXC_ENFORCE`** macro call, thus they provide little help for maintainers. If, for the purpose of helping to diagnose failures, detailed information should be provided, use one of the **`EXC_Program`**, **`EXC_Platform`** or **`EXC_Usage`** exceptions instead. Implemented in module **`EXCEPTN`**.

`EXC_Usage` - This class implements the standard Talisman 5 exceptions that are due to incorrect usage of the system. An **`EXC_Usage`** exception is an unusual or unexpected event that is usually due to some usage error. Different from **`EXC_Enforce`** handlers, **`EXC_Usage`** handlers (**`catch EXC_Usage * pExc`**) should provide adequate explanation to the user

and attempt to recover, possibly with the assistance of the user, and continue execution in a valid state. Implemented in module **EXCEPTN**.

EXC_Platform - This class implements the standard Talisman 5 exceptions that are due to platform malfunction, such as I/O errors, memory and message system errors. Different from **EXC_Enforce** handlers, **EXC_Platform** handlers (**catch EXC_Platform * pExc**) should provide adequate explanation to the user and attempt to recover and continue execution. Implemented in module **EXCEPTN**. Unfortunately, due to C++ standard restrictions, many platform exceptions can only be caught by a catch all catcher.

EXC_Program - This class implements the standard Talisman 5 implementation error exception. An **EXC_Program** exception is an event that corresponds to the detecting of an error generated by a software defect. In a sense it corresponds to a broken assertion. Different from **EXC_Assertion** and **EXC_Enforce** exceptions, failure exceptions must provide detailed data to help diagnosing and, if possible, recovering from the cause of the detected failure. Code that throws **EXC_Program** exceptions should remain active in production code. Otherwise, prefer throwing **EXC_Assert** exceptions instead. Implemented in module **EXCEPTN**.

ENV_Environment - This class provides access to environment variables. Environment variables may be used for several installation dependent purposes. For example, an environment variable may tell where the configuration files and workstation state files are stored. The default implementation requires the **TALISMAN** environment variable to be set. This variable should contain a parameter **-config:** <*configuration file name*> that informs where the main configuration file is stored. An environment variable may define several parameters. All parameter names must start with a hyphen ('**-**'). Implemented in module **ENVIRON**.

GLB_Global - This class implements the application context initialization. The class contains methods that assure establishing the necessary startup context. The class creates the main **ENV_Environment** object that accesses data contained (usually) in the **TALISMAN** environment variable. When additional context should be established for a specific application, a class that inherits from **GLB_Global** should be implemented. This class should specialize the **BuildGlobal(…)** method, but should still call the parent (super) **BuildGlobal(…)** method. The standard class starts the event logger used by the test support classes. It also creates a safety blanket to be used if memory overflow should occur. Finally, it saves the session starting time. Implemented in module **GLOBAL**.

LOG_Logger - This class implements the standard log file handler. Logging may be directed to the console or to a file. There may be several different loggers active at any given instant. Implemented in module **LOGGER**.

SIO_SequentialIO - This class implements a standard sequential file handler. It provides safe file reading and writing that assuredly do not provoke buffer overruns, while still following the line based approach of the C I/O handling library. Implemented in module **SIOCTRL**.

CFG_ConfigurationList - This class accesses configuration parameter files. Configuration parameters may save several workstation specific data, such as installation characteristics and platform state data that should be persisted from one execution instance to another. Examples of installation characteristics are names of parameter files. Examples of persistent parameters are: the list of the last *n* accessed files, and user preferences. Configuration files are ASCII files written in a representation language that is simpler than XML when considering human interaction with such files. Since it is not expected that these files be shared with other software, using a specific format does not imply restrictions. Implemented in module **CONFIGFL**.

SLS_SimpleList - This class implements a self-verifying generic doubly linked list container. Contents of the list are generic; however they must be homogeneous for a given list and must specialize the **SLSE_ListElementValue** class. Implemented respectively in module **SIMPLIST** and **SLS_ELEM**.

SMT_SymbolTable - This class implements a self-verifying generic hash container. The size of the hash table must be defined when constructing a table object. The collision resolution is performed by a collision list. Contents of a table entry are generic; however, they must be homogeneous for a given table and must specialize the **SMTE_SymbolTableElement** abstract class. Implemented in module **SYMBTAB**.

3.3. String table construction and access tools

String tables collect all strings needed by the application using the run-time support library, except strings that are used only by the test framework. Talisman's programming convention dictates that the code must not contain any literal string; all string literals (except those needed to tell that the string table is absent or mal formed) should be contained in the string table. This applies even to I/O format strings. To reduce string table pollution, the test framework uses string literals in its code. However, the run time support modules must strictly refer to strings contained in the string table. This rule is valid for all kinds of strings, such as messages, format descriptors, cultural aspects of the locale and default names.

When writing a module, a string definition file should be written too. This file defines string elements stating: a symbolic name, a numeric id, a table id (in which table should the string be stored) and the string itself. When using Talisman 4.4 to develop modules, strings may be edited at the point where they are used. A file generator (linearizer) tool generates the corresponding **.str** file. Figure 3 displays some string descriptors. The first is a format to be used by a **sprintf** or similar function. The second will be referred to by a message object.

```
<Offset>  525
LSP_Format02 4 m "%s Num elements: min %d max %d total %d"
LSP_ErrorAttr 7 m "ERROR - page. idPag: %1 Line: %2 File: %3"
```

Figure 3. Example of string descriptors.

```
pMsg = new MSG_Message( LSP_ErrorAttr ) ;
pMsg->AddItem( 2 , new MSG_ItemInteger( __LINE__ )) ;
pMsg->AddItem( 3 , new MSG_ItemString( __File__ )) ;
pMsg->AddItem( 1 , new MSG_ItemInteger( idPag )) ;
```

Figure 4. Example of message object creation.

As mentioned when describing the **MSG_Message** class, strings may contain parameters. Parameters are identified by **%n** entries in the string body, where **n** is a number between 0 and 19. These parameters will be filled by message items associated with the message that will display the string. The numeric order of the parameters in the string is irrelevant; furthermore a same parameter may be referenced more than one time in a string. Figure 4 shows how to construct a message object that contains a string with several fields.

String values are placed in a memory resident table or in a parameter file. String ids must be unique. For this purpose each module has a unique numeric id. In figure 3 its value is defined by the **<Offset>** parameter. Numeric module ids must be sufficiently separated one from the other to accommodate all strings defined in each module. According to the Talisman 5 programming standard, all modules have unique ASCII and numeric ids. Numeric module ids are used as an offset when computing the effective string id. Modules also possess an ASCII id, usually composed of two or three letters. This id is used to prefix external names (e.g. class names, global external functions, and global external constants) assuring name uniqueness considering the whole project[1]. The **TalismanModuleIds.xls** table contains the module names, ids and other information.

GENSTRID - This tool generates the string id table for a given string declaration file - **.inc** file. String declaration files have names similar to **STR_xxx.STR**, where **xxx** is the ASCII id of the corresponding module. String ids are used by **STR_String** and **MSG_Message** methods to access strings contained in a string table.

GENSTRTB - This tool generates the memory resident string table. It receives a **yyy.makeup** file that contains a reference to each of the string declaration files to be used - **#include STR_xxx.str**. In this case **yyy** is related to the component that uses the string table. This parameter file must be updated whenever a new string declaration file is developed.

1 Another convention would use name spaces instead of module id letters.

4. Test support framework

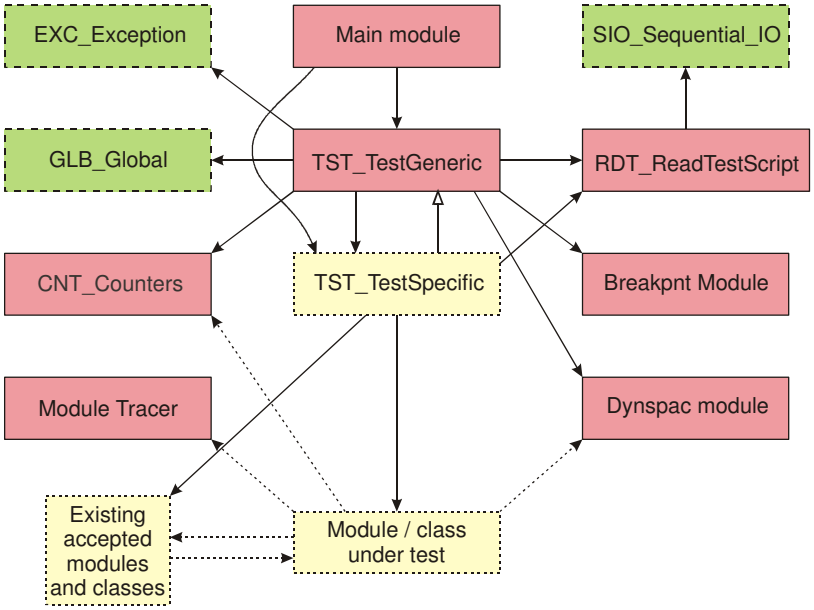

Figure 5. Test support framework class model

The test framework uses the utilities and run-time support classes described in the previous sections. Only the more relevant relations are shown in the diagram (figure 5).

The specific test class **`TST_TestSpecific`** specializes (inherits from) the **`TST_TestGeneric`** class. Usually a specific test class is geared towards testing a single module under test. A specific test class must be developed for each test build. Test builds usually test a single specific module; however, they may also be set up to test components, or several strongly interdependent modules. Thus specific test modules may interpret commands that exercise methods or functions of already accepted modules. Components and interdependent modules may be developed and tested incrementally; however, when finished the collection may be tested in detail using one single build capable of testing all modules. As mentioned before, the module under test might interact with already accepted modules, mock modules and stubs.

At test time the module under test might have been instrumented with tracing and test coverage control by means of passage counting. It might also interact with the dynamic space control module by means of global overwriting the **`new`** and **`delete`** operators.

4.1. Modules

`Main` Module - The main module receives and verifies the command line parameters. If correct, it constructs the test driver by constructing the specific test class that inherits from the generic test class. Each test build contains only one specific test class. Hence, when compiling a test build only the required specific test module must be included. See the *build composition script* described later on. Once the test driver object has been created, the main program activates the generic test control driver. At end of test it performs memory leakage control. The programming standard used requires the deletion of all data spaces allocated by the application and

by the test drivers before returning to the main function. Any remaining allocated data space will be considered being memory leakage.

Dynspac Module - The **Dynspac** module implements several function to help controlling dynamic memory usage. It replaces the standard **new** and the **delete** operators. The replacement occurs by simply including the **Dynspac.hpp** header file in the module to be controlled. No further coding is necessary. To allow replacement to occur only in debug mode, place the include command in an **#ifdef _DEBUG** conditional compilation block. Data spaces allocated with the use of the **Dynspac** module contain the name of the source module and the line number where the space has been allocated. These attributes are listed in the messages involving dynamically allocated spaces. This reduces the effort of discovering the code fragments responsible for the incorrect dynamic memory usage. The module permits also to simulate memory overflow events for the purpose of testing memory overflow exception handlers. Several test script commands directed to the **Dynspac** module are interpreted by the **INT_DSP.CPP** module.

Breakpnt Module - This module implements a function containing a dummy source command that might be marked by a debugger breakpoint. If this is done, when interpreting the test script command **=Breakpoint** the test interpreter yields control to the debugger. This allows starting the debugger only when execution of the script is close to the point where a failure has been observed during a previous run of the test.

Tracer Module - This module implements several functions that trace execution and/or the run-time stack usage. Tracing requires instrumenting the code with calls to trace functions. When using Talisman 4.4 to develop the program, trace function calls are placed under control of a **#ifdef _TRACE** block. Hence, to activate tracing modules must be compiled with this flag set. Tracing may be turned on or off, allowing tracing to occur only at the points of the test script that are involved in diagnosing failures found during testing.

4.2. Classes

TST_TestGeneric - This class is the standard test driver. It is an abstract class that must be extended by a **TST_TestSpecific** class. It controls the interpretation of the test script file. It also contains the comparison methods used to verify whether the computed result is equal to the expected one. In addition it contains several methods that display error and information messages in a standard way. The generic test class catches all exceptions. If an exception handling test script command has been provided, the unraveling process might be interrupted or not, depending on the command content. If unraveling is not interrupted, execution of the program will be canceled by the disaster handler. Implemented in the **GNRCTEST** module.

TST_TestSpecific - This is the specific test driver for the module under test. It extends the **TST_TestGeneric** class. It must contain the **PerformSpecificTest(…)** interpreter method that interprets the test

commands used in the test script. Usually a test command will be available for each of the methods and functions of the modules under test. It might also contain additional test commands that provide set up and tear down actions, as well as calling methods of already accepted classes. A **TST_TestSpecific** class must be specifically written for each test build. The build composition script provides means to including just the specific test module required by a given test build.

RDT_ReadTestScript - This class implements the test script command reader. Reading test command lines is similar to the C **scanf** function in that it is capable of reading **n > 0** parameters as defined in an input format string. Different from **scanf** though, the format string contains only an indication of the data types of the parameters to be read. Buffer overflow is controlled, and parameter types are controlled too. While reading, strong type checking of the fields being read is performed. Thus, if the type of the data field to be read does not match the expected type, an error message will be issued and the interpretation of the command is canceled. Furthermore, data elements read might be literals or declared constants. If the name of a constant is read, the value of the constant is returned as long as it is typewise correct with the parameter type definition. Finally, different from **scanf**, the syntax of a field to be read is the same as that of a literal in C/C++ code. For example, strings must be enclosed in double quotes and single characters in single quotes. Strings and characters may contain escape and hexadecimal character definitions, including **\x00** characters. String data is read into two parameters. The first will contain the length of the string considering only the characters contained in the string after conversion (i.e. no quotes, and a single character per special character input by means of an escape sequence), and the second will contain the converted string itself. This allows strings to contain special characters, including the null character (**'\0'**). Implemented in the **READTEST** module.

CNT_Counter - This class contains methods and functions that support a simpleminded test coverage control. This control is performed by means of passage counting. This requires that the module be marked up including calls to a passage counting function wherever necessary according to the measurement strategy used. Although inefficient, counters are identified by a symbolic name. This reduces the complexity of marking up the program since no other rule than name uniqueness is required. When using Talisman 4.4, counters may be inserted automatically into the code by the code composition tools. The **INT_CNT.CPP** module interprets several standard test script commands that are directed **CNT_Counter** objects. The **CNT_Counter** class is implemented in the **COUNTER** module.

5. Test script structure

```
// Script:    Tst-bcd-01.script
// Author:    Arndt von Staa
// Date:      15/set/2007
// Purpose:   Test bcd arithmetic

== Define constants
=DeclareParameter   less    int  0
=DeclareParameter   equal   int  1
=DeclareParameter   greater int  2

=DeclareParameter   num00   int  0
=DeclareParameter   num01   int  1

== Save status of the dynamic memory
=Reset                      // deletes all allocated test spaces
=AllocSaveNum          0    // saves the status of dynamic memory

== Add numbers of different size
=ConvertIntToBCD         num00  3  -49246
=GetBCDNumber            num00  "B49246"   // first hexa encodes size
=ConvertIntToBCD         num01  1      -3
=GetBCDNumber            num01  "93"       // gets the memory image
=Add                     num00  num01
=GetBCDNumber            num00  "B49249"

== Generate and add random numbers
=SetExceptionVerbose     .false  // prevents immediate throw messages
=RandAdd                 100000  // number of adds to perform
+Distribution            10 15 15 25 35   // percent, must add to 100

== Verify if the dynamic memory has been reset.
=Reset
=AllocCompareNum       0 // compares the status of dynamic memory
```

Figure 6. Example of a test script file

Test scripts are written in a simple language that resembles assembler. Figure 6 displays an example of a test script file. Each line corresponds to a statement, a blank line or a comment line. Comments start with a **"//"** and go until the end of the line. Blank lines may contain zero or more blank characters.

Each test case starts with a test case header statement. Test case header statements contain a **"=="** as the first characters of the line, the remaining characters are treated as comments.

After a test case header come one or more test command statements. Test command statements have the generic syntax:

=*command parameters*

where *command* identifies the test action to be performed. All command lines should start with a single '**=**' character followed immediately by the command name. Usually commands correspond to calling some method. The *parameters* field contains zero or more parameters that depend on how the interpreter expects parameters for that command. Parameters are either literals or symbolic names that refer to a literal. The syntax of literal parameters is similar to the syntax of C/C++ literals.

After a test command it is sometimes necessary to provide extra lines containing additional parameters. For example when describing how test data should be generated several parameters are usually needed. Parameter lines have the generic syntax:

+*parameterLineName parameters*

where *parameterLineName* identifies the set of parameters to be read. All parameter lines should start with a single '**+**' character followed immediately by the parameter line name. The parameters field follows the same syntax as the command line. If a varying number of parameter lines are to be read, the last useful line of each group should be followed by a line containing:

+ParameterListEnd

There are four test command interpreter classes:

generic tester controls the execution of the test.

specific tester is developed for testing the module to be tested. Each module to be tested has its own specific tester. Specific testers inherit from the generic tester.

dynamic space control this module intercepts new and delete commands. It aims at controlling correct use of dynamic spaces and of memory leaks.

passage counter control this module implements several operations that allow a simple minded form of measuring coverage. For more precise coverage measurements the IDE tools should be used.

```
else if ( strcmp( Command , Compare_028_CMD ) == 0 )
{
   /***** Function
      TAL_tpRetCompare Compare( const STR_String & Second ,
                                bool DoConvert = false )
   *****/
   int  inxObj   = -1 ;
   int  inxStr   = -1 ;
   int  expRet   = -1 ;
   bool Mode     = false ;

   int  numRead  = TST_pReader->ReadCommandLine( "iibi" ,
         &inxObj , &inxStr , &Mode , &expRet ) ;

   if ( ( numRead != 4 )
     || !VerifyInxElem( inxObj  , YES )
     || !VerifyInxElem( inxStr  , ANY ))
   {
      return TST_RetCodeParmError ;
   } /* if */
   return Compare( vtObj[ inxObj ]->Compare( vtStr[ inxStr ], Mode ),
             expRet , "String comparison error." ) ;
} // end selection: Test: STR !Compare string values
```

Figure 7. Example of a test interpreter fragment

The generic organization of an interpreter, see figure 7, is an **`if else if`** selector that selects the command to be interpreted. Each command interpreter fragment reads the

parameters, verifies whether they are valid, performs the desired action and compares the result with the expected result.

In this section we describe briefly the syntax of the test statements. Similar to assembler commands, test statements always start on a new line and use the whole line. There several kinds of test statements:

- Test commands, always start with a '**=**' character;
- Additional test command parameters, always start with a '**+**' character;
- Preprocessor statements, always start with a '**#**' character;
- Comment lines, always start with a "**//**" pair of characters;
- Blank lines, contain zero or more space characters. Blank lines do not appear in the generated log, the **=NewLine** command inserts a blank line into the generated log. This is useful for increasing readability.

Test command lines may contain comments in addition to the command and its parameters. By default everything beyond the last parameter to be read by a given command is a comment. However, anything starting with "**//**" will also be handled as a comment. This allows placing comments on fields that contain optional arguments.

Test commands contain the arguments to be passed to the function or method to be executed, as well as the expected results.

If required, the test command interpreter may read several lines containing command parameters, using the **ReadCommandLine()** method contained in the **RDT_ReadTestScript** class.

Generic test commands are interpreted by the generic tester.

Specific test commands are interpreted by the specific test module that interprets commands designed to test the module under test.

There are three vectors of saveable values:

clock saves clock values, see **=StartClock** command. These values are used to determine the elapsed time.

num saves the number of dynamic spaces, see the **=AllocSaveNum** command. These values are used to measure memory leakage.

bytes saves the total number of allocated bytes in dynamic spaces, see the **=AllocSaveBytes** command. These values are used to artificially limit the amount of available memory.

6. Generic test script commands

This section describes all generic test script commands. These commands are interpreted by modules that compose the test framework.

The test script commands interpreted by the generic test module are:

#include <string fileName> includes the text read from the file **fileName**. The standard extension of the file is the same as that of test script files ("**.script**"). Files to be include may contain any test statement, including even other **#include** statement. Nesting of **#include** statements is limited to 10 (see **dimReaderStack**) statements.

== beginning of a new test case. This command counts test cases. The string after the "**==**" characters is displayed in the log and is handled as a comment. The goal of these texts is to document the intention of the test case. Try to write a string that captures the gist of the test case.

=NewLine inserts a blank line in the test log.

=Cancel cancels the execution of the current script. It always increases the failure counter by one.

=Breakpoint <idBkpt> allows the interplay of the language processor debugger and the execution of the test. Usage:

- Close to the desired point in the test script insert a **=Breakpoint** command.
- If desired, use the parameter **<idBkpt>** to identify the break point.
- Using the debugging tool find the function **void BKPT_PerformBreakpoint(int idBreakpoint)** contained in module **BREAKPNT.cpp** . In the body this function find the line: **int id = idBreakpoint ;** // Dummy action. Set an IDE debugger breakpoint there. When running the program in debug mode, it will stop at this point. Now set the breakpoints in the code as you deem necessary to help diagnosing a failure. Using this approach allows uninterrupted run up to the breakpoint instruction in the test script.

=StartVerbose starts displaying each test command read. By default only the test case header comment is displayed.

=StopVerbose stops displaying the test commands read, returning to the default mode of displaying only the test case header.

=StartTracing starts displaying program execution traces. These traces are performed by the **TRC_TraceString** function. See module **TRACER** for details. When using Talisman to design and develop code, code blocks may be marked to be traced. Trace function calls will be generated if the block is marked. All generated trace commands are contained in a **#ifndef /D_TRACE #endif** conditional compilation code fragment. Tracing will occur only when the **/D_TRACE** switch is set *while compiling*.

=StopTracing stops displaying program execution traces.

=StartThrowTracing starts the displaying exception **throw**s at the moment they are issued. Since exceptions unravel the call stack, tracing throws provides information about all methods that have been canceled before reaching the **catch** command that handles the thrown exception. To see the unraveling the stack tracer should be active. See the **TRACER** module for more details.

=StopThrowTracing stops the tracing of throw commands.

=ExceptionProgram <type> <id> <char> when handling a program exception verifies if it is of type **type** and whether it refers to message **id**. contained in the exception object. If the correct exception has been caught and if **char** contains the character **'c'**, the exception is not propagated. In this case the corresponding failure is discarded, since it was expected.

=ExceptionSystem <char> catches any exception. It may be used to catch exceptions thrown in code that does not throw **EXC_Exception** objects, or is a system defined exception (e.g. ou of memory). If **char** contains **'c'**, the corresponding failure is discarded, since it was expected.

=Recover upon return from the specific test function and if a failure or a parameter error was reported, this command discards this failure, since it was expected. If this command is interpreted when no failure was detected by the command contained in the immediately preceding script line, it will be reported as a failure since it was not expected.

=FailureReset <count> resets the failure counter to zero if and only if it contains a value equal to **count**

=StartClock <inxClock> resets the **inxClock** clock to the current time. There are 10 clocks available. At beginning of the execution all of them are set to the starting time of the test session.

=DisplayElapsedTime <inxClock> displays the milliseconds elapsed with respect to the time contained in the **inxClock** clock.

6.1. Dynamic space control test commands

The next set of test commands are interpreted by the **INT_DSP** function which supervises the dynamic memory control module **DYNSPACE**.

=AllocSetTrace <bool> if **bool** contains **.true,** traces all **new** and **delete** operations. The default is **.false** .

=AllocSetTerse <bool> if .false only controlled space headers will be displayed. If .true header and hexadecimal content of the spaces will be displayed.

=AllocVerify <mode> <expBool> verifies the integrity of the allocated data spaces list. **mode** is one of the **TAL_tpVerifyMode** values, see the **Talisman_constants.inc** parameter file contained in the **..\Global\consts** directory. **expBool** is the expected return value, if it contains **.true** the list must be correct.

=AllocGetNum <expNumber> gets the current number of allocated data spaces. If **expNumber** is greater than 0, it compares the value and issues a failure if not equal. If 0 the value is displayed. The number of allocated spaces corresponds to the number of **new** and **delete** commands that are issued by the **DYNSPACE** module. This module will be used by all modules that contain a **#include "DYNSPACE.HPP"** statement.

=AllocSaveNum <inxNum> saves the current number of allocated spaces in the **inxNum** element, 0 <= **inxNum** < 10. **INT_DSP** allows saving up to 10 elements. These elements are changed only by test commands.

=AllocCompareNum <inxNum> <delta> compares the current number of allocated spaces with the sum of the value in the **inxNum** element added to **delta**. This allows verifying whether memory leakage occurred during the script fragment starting at the corresponding **=AllocSaveNum** command. **delta** is optional, if absent zero is used.

=AllocSetBytesLimit <inxNum> <newLimit> sets the limit of allocated bytes to the sum of the **inxNum** added to **newLimit**. If this limit is to be exceeded by a **new** operation, a memory overflow exception will be raised using the **GLB_HandleMemoryOverlow( )** function. If **NewLimit** == 0 sets the limit to the default absolute maximum (1 gigaByte)

=AllocGetBytes <expBytes> gets the current absolute total number of allocated bytes, allocated or deallocated by **DYNSPAC**. No overhead bytes contained in allocated data spaces are accounted, neither those used by the **DYNSPAC** module, nor those required by the dynamic memory management system. **expBytes** is the expected total. If greater than 0, the number will be compared. If equal to 0 it will be displayed.

=AllocSaveBytes <inxNum> saves the current total number of allocated bytes in the **inxNum** element.

=AllocCompareBytes <inxNum> <delta> compares the current total number of allocated bytes with sum of the **inxNum** element added to **delta**. **delta** is optional, if not given, zero will be used.

=AllocListSpaces <idSpace> if **idSpace** greater than or equal to 0, displays the data space corresponding to the given **idSpace**. If **idSpace** is less than 0, displays all allocated data spaces. If terse is set, only the data space header is displayed. Otherwise header and body are displayed.

6.2. Passage count test commands

The next set of test commands are used to control passage counting. They are directed to the **INT_CNT** test command interpreter of the **COUNTERS** module. Counters are used to measure the number of times that control passes a given point in the module's code. For this the module must be marked up with calls to the **CNT_Count(counter_name)** function. More details can be found in **COUNTERS.HPP**.

Count function calls may be generated using Talisman when composing the code using the "Linearizar programa com contadores" code composer action.

Previous to counting, a file containing all counter definitions contained in the controlled module must be read, see below. Counter definitions are strings.

This file can be generated using Talisman's "Linearizar contadores" code composer action.

Counting will only be performed if the test program is started with the "**c**" switch set.

=CounterVerifyZero <expNum> computes number of counters containing zero. This number must be equal to **expNum**

=CounterDisplaydisplays the contents of all counters.

=CounterDeleteAccumFile <fileName> deletes the **fileName** file. **fileName** must contain an extension name. Care should be taken when using this command, since no checking is performed to verify whether the file may or not be deleted.

=CounterCopyAccumFile <destinationName> <sourceName> copies the file **sourceName** to the file **destinationName**. The default extension is **.count** Care should be taken when using this command, since no checking is performed to verify whether the **destinationName** file may be overwritten or not.

=CounterReadAccumFile <fileName> reads the accumulated counters file **fileName** and places the counters read in the working counter symbol table. See module **Counters.hpp** for more details.

=CounterReadDeclaration <fileName> reads the counter declaration file **fileName** and places the counters read in the working counter symbol table. See module **Counters.hpp** for more details. If a counter name terminates by a **/: <count>** substring, count will be used to initialize the counter. If **count** is equal to -2, it is a forbidden counter. An error will be reported whenever control passes such a counter. If **count** is equal to -1, it is an optional counter; hence no error will be reported when verifying counters.

=CounterResetCounter <counterName> if the counter **counterName** contains a value greater than 0, it will be set to zero. Optional counters and forbidden counters remain unchanged.

=CounterGetCount <counterName> <expNum> verifies whether counter **counterName** contains **expNum**.

=CounterGetNum <expNum> verifies whether **expNum** counters are currently defined in the running instance.

=CounterResetAll resets all counters to 0. Optional and forbidden counters are not changed.

=CounterStart starts counting. Counting will be performed only if the '**c**' parameter has been given when starting the program and if counting has been started. The default is "stopped".

=CounterStop stops counting.

7. Tools

This section describes the tools used while developing modules to be tested using the Talisman C++ Test Framework.

7.1. Test build composition

GMake - This tool generates make script files from a file that defines the composition of a build. The script defines the directory structure to be used, and lists the components that make up the build. Components might be C++ modules, and also files that will be handled by specific user defined tools. It also tells whether the build should be linked, or whether a library, static or dynamic, should be composed instead. When defining a test build, the specific test module name must be assigned to the **SPECTEST** variable. This assures that the correct test object will be constructed by the **Main** module. The file **GMAKE-v6-00.pdf** contains the documentation of this tool (in Portuguese). The **GMAKE** tool requires a parameter file that establishes the rules to be used when generating the **MAKE** script file. The parameter file **MS-CPP.parm** is geared to generate script files to be used when compiling with the MS Visual Studio C++ compiler. It also establishes the specific rules required for the Talisman project conventions. By changing the parameter file, a variety of conventions and compilers can be properly addressed.

Generate - This tool generates a file from a string contained in a **MAKE** command line argument. When following the Talisman 5 project convention, the tool generates the include file (**Specific-test.hpp**) used by the **Main** module. This generated file contains an **#include** command referencing the specific test module header file. The header file to be used is identified by the **SPECTEST** variable defined in the build composition script (see **GMAKE** above). The **SPECTEST** variable, the composition parameter rules and the way **Main** has been coded, establish a variant of an object factory design pattern aiming at constructing the specific test object required by the build.

Compilebanner - This tool displays a separator line between successive compilation steps. This makes it easier to read the compilation log generated while recompiling the collection of all test builds.

RunTestSuite.lua - This **lua** program automates of the whole test process. It has been written for the **Lua 5.1.1** or more recent version. The **lua** program requires a suite file (as an example see **TestFramework.suite**) that describes how the test should be performed. The suite file defines the directory structure used by the project as well as which builds should be constructed and which script files should be submitted for a given build. Using the build composition files it regenerates all **make** script files. Afterwards all test builds and library builds are recompiled and then all test scripts are applied to the corresponding test builds. While performing these actions, the tool monitors the progress and generates log files about the overall process and also for each test script. It allows defining test runs that have been designed to end in expected failure conditions or disaster conditions. These test runs are considered to be correct iff the expected failure is reported.

TestFramework.suite - This test suite reconstructs and retests the whole test framework and all support modules it requires. It also regenerates the

TalismanTestDll.dll and **TalismanTestLib.lib** dynamic and static libraries. Use the **DoAll.bat** file to control the whole test framework reconstruction and retesting.

GenTestLibraries.suite - This test suite reconstructs and retests only the dynamic (**.dll**) and the static (**.lib**) test framework libraries. To perform this test suite, use the **TestSuite.bat** batch file with the **/A** command line parameter. It is assumed that the **GenDllExp.bat** batch file has been run prior to executing this suite generating the **TalismanTestDll** control (**.def**) file.

7.2. Configuration parameters

Configuration parameters store user preferences, platform state data and installation specific data. This allows giving the user the impression that the system memorizes its execution state from one usage instance to another. Usually configuration parameters are stored in XML files. Although such files are excellent means to interchange data between applications, they tend to be awkward for humans. Since these parameters tend to be application and user specific, we do not expect that they will be interchanged between different machines, hence we use the traditional section/parameter format instead. The configuration file to be used may be defined directly, by means of a parameter passed to the list builder, or may be defined in an environment parameter. Furthermore, the environment variable might be given by parameter or used by default. The default environment variable is **TALISMAN**. The value of the environment variable may contain several fields. Each field should start with a '**-**' and be separated from the preceding field by at least one blank character. The test framework defined field is:

-config:<configuration file name> - the name identifies the file that contains the configuration parameters. For automatic testing of the framework it is required to define:

```
set TALISMAN= -config:talisman
```

Without this environment variable declaration the testing configuration module using the test script **TST-CFG-01** will report testing failures.

7.3. Test run support tools

DISPLAYSTATS - This tool displays the contents of the accumulated test statistics file. When running several test scripts in a row for a same build, each one produces its own test statistic. These statistics may be accumulated (see line parameter **/a** of the main module) in an accumulated statistics file, which is then displayed by this tool.

7.4. DLL building tools

GRDLLEXP - This tool generates the **.def** file required when generating a dynamic library (**.dll**). As input it receives a text file generated by the **DUMPBIN** tool contained in MS Visual Studio. See the batch file **GenDllExp.bat** for the way of calling this tool.

7.5. Batch files

CleanAll.bat - This batch file deletes all files that are generated and which are contained in a Talisman 5 component project.

Compile.bat - This batch file compiles a component test build. It requires the name of the **.make** file as a parameter. Note: this is a generated file.

Delobj.bat - This batch file deletes all **.obj** files. It should be used before recompiling a build with different compilation parameters whenever the source code has not been changed.

DoAll.bat - This batch file reconstructs and retests the test framework, all utilities, all support modules and all libraries. When stated as **DoAll** it will recompile all programs for debugging, i.e. with the **_DEBUG** key set. Stated as **DoAll P** (upper case '**P**') it will recompile all programs for production, i.e. with maximum code optimization and without the **_DEBUG** key set.

GenDllExp.bat - This batch file controls the generation of the **.def** file required when building the dynamic library. It receives the name of the file to be generated.

GenMake.bat - This batch file generates the **.make** script file from a given build composition script (a **.comp** file). It requires the name of the composition build script file.

Test.bat - This batch file runs a given test build using a given test script. According to the Talisman 5 project file naming conventions used, all test builds have a name similar to **TST-xxx**, where **xxx** is the character string id of the module under test. Furthermore, all test script files have a name similar to **TST-xxx-nn**, where **xxx** is the id of the build under test and **nn** is the index of the test script file. The batch file requires the parameters **xxx** and **nn**. Example: **test rdt 01** tests the command reader module using the script file **tst-rdt-01.script**.

TestSuite.bat - This batch file runs a test suite file using the **RunTestSuite.lua** program. This batch file requires a suite script file. Use also the **/A** parameter to assure that compilation is performed even if errors were found while generating the **make** files. These errors occur due to string id files not having been found when the **make** file is being generated. More detailed documentation can be found in the source code of the **RunTestSuite.lua** program.

8. Process

This section describes briefly the recommended development process and standards to be used while developing.

8.1. Directory Structure

The system is being developed using the MS-Visual Studio compilers. The standard Windows directory structure is:

- **xxx** – the project directory, usually **\Talisman**
  - **Documents** – contains all documents regarding the Talisman 5 project.
  - **Drawings** contains pictures usually in **.cdr** (Corel Draw) and **.wmf** (Windows metafile) formats.
  - **Exec** – contains the Talisman 4.4 meta-environment instantiated for the Talisman 5 project.
  - **Global** – contains common files for several applications. Sub-directories:
    - **Comp** – contains common composition descriptors. More specifically, contains the module id control file **TalismanModuleIds.xls**. Use this file to keep track of the module name, location, and its ASCII and numeric ids.
    - **Consts** – contains global constants.
    - **Lib** – contains the library files.
    - **Tables** – contains common tables.
  - **Test** – contains the test framework, support and utility files.
    - **Batches**
    - **Bsw** – contains the Talisman 4.4 software bases used to develop the source code files of the test framework.
    - **Comp** – contains the build composition (**.comp**) files needed for testing the test framework, support and utility files. Contains also the generated **.make** and **.list** files. The latter contain statistics regarding the corresponding program. Finally, it contains the compilation logs generated while compiling the test suite.
    - **Docs** – contains the test framework documentation,
    - **Obj** – contains the object (**.obj**) and executable (**.exe**) files as well as generated libraries (**.lib** and **.dll**). It contains also the link script files (**.build**) that describe the composition of a given build. See the *GMake* tool documentation for details.
    - **Sources** – contains the **.cpp** and **.hpp** files as well as the **.count** files needed when counting passages. See the **COUNTERS** module for details. Finally, it contains also **.err** files that contain the error reports generated by the compiler.
    - **Tables** – contains the tables, especially the string tables, used by the support and utility modules.
    - **TestCase** – contains the test script files needed for testing the test framework and utility modules. Contains also the test log files.
  - **Component_XXX** – contains a structure similar to the Test directory. Each **component_XXX** should be kept in its own directory.
  - **Tools** – contains the tools used to develop Talisman components
    - **Batches**

- **Bsw** – contains the Talisman software bases used to develop the development support tools. It contains also the Talisman form programs that instantiate the Talisman 4.4 meta-environment for this project. Finally, it contains the Talisman 4.4 documentation.
- **Programs** – contains the source and executable code of the development support tools used, as well as parameter files and documentation files required by these tools.

8.2. Development process

When using Talisman 4.4 while developing a new module **mm$_i$** the name, ASCII id and numeric id should be registered in the **TalismanModuleIds.xls** file. Then the definition module **mm$_i$.hpp** (header file) should be designed and coded using Talisman 4.4. This file usually contains the interface specification, documentation and declarations of module **mm$_i$**. Afterwards the implementation module **mm$_i$.cpp** should be designed and coded too. If strings are required by the module, a **str_mm$_i$.str** file should be generated. This can be done using Talisman 4.4, writing the string definition in the *String list* field of the block code form. Later on the **str_mm$_i$.str** file must be generated (*linearized*). Do not forget to insert an "**#include "str_mm$_i$.str"**" command in the **.makeup** file that will be used when generating the string table for the test build.

When developing modules in an incremental way, the **mm$_i$.hpp** file should be designed and coded as completely as possible. However, considering the **mm$_i$.cpp** implementation file, only the functions and methods required by the increment should be designed and coded. The remaining may be left without code, or otherwise should be filled with stub code.

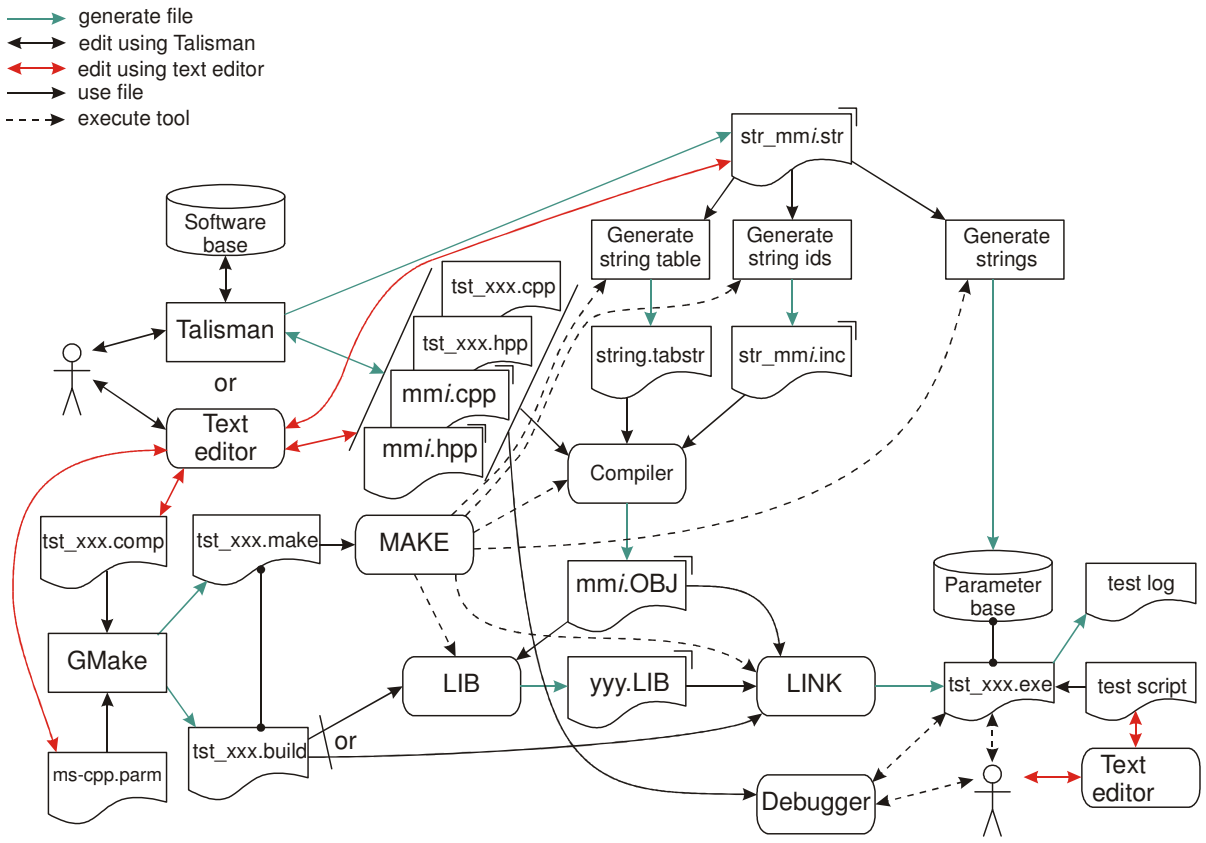

Figure 8. Talisman project programming process

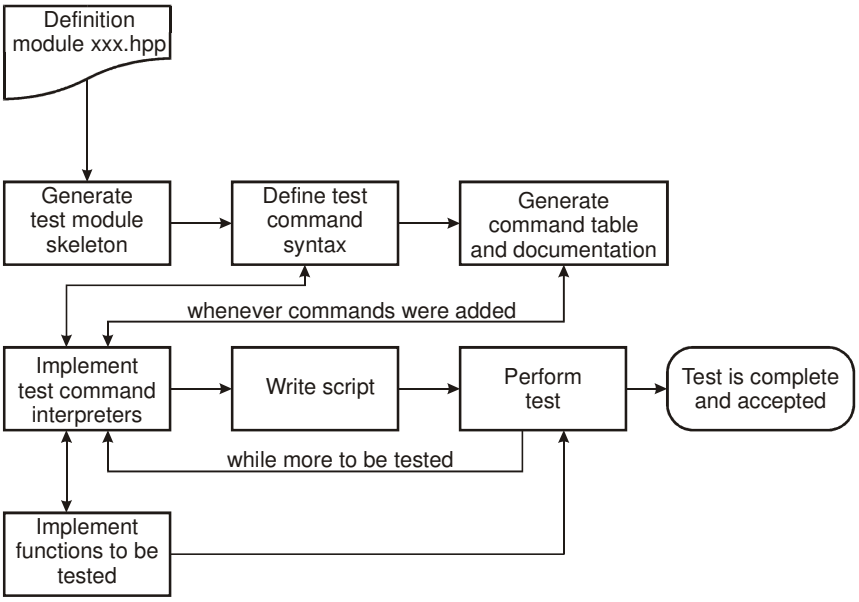

Figure 9. Specific test module development when using Talisman 4.4

Once the **mm$_i$.hpp** file exists, the **tst_xxx.hpp** and **tst_xxx.cpp** source code files may be generated. The **xxx** in the name correspond to the ASCII id letters of the module to be tested (usually module **mm$_i$**). The generated code implements the specific test module for module **xxx**. However, all command interpreter fragments are commented out with **#if 0** commands and result in *not implemented* errors.

If Talisman 4.4 is being used, a prototype of the specific test module may be generated once the header file (**.hpp**) exists. Use the "**Gerar modulo teste**" transformation when placed on the name of the module to be tested. This transformation generates a compilable initial version (stub) of the specific test module (**tst-xxx**). It also generates unique test script command names, which are not mnemonic. Edit the command name in the precondition field of each block of the interpreter method "**!P Perform specific test action**". While editing, establish the parameters required by the command and, if not obvious, document them.

Test script commands must be unique. Placing again the current object cursor on the name of the module to be tested, do the transformation "**Regerar tabela commandos teste**" to regenerate the test script command table contained in the **tst-xxx.cpp** file. Afterwards, do the transformation "**Regerar documentacao teste**" to generate the documentation of the test script commands. The documentation will be inserted in the **tst-xxx.hpp** file.

Create and compose the build script. The standard name format is **tst-xxx.comp**. Assure that the **[MacrosApos]** section of the **.comp** file contains "**SPECTEST = tst_xxx**". This will assure that the proper specific test control object will be created by the main program. Using **GMAKE** with the **.comp** file generate the corresponding **.make** and link (**.build**) scripts. You can now compile and start testing the build.

It is strongly recommended that the development of each module be performed in an incremental way. Implement some constructors and destructors and test, possibly using a simplified test. Implement a few more methods and test. Repeat until no more method, function, constructor or destructor needs to be implemented. If desired refactor the modules to get better engineering quality.

Once everything seems to be working, review, strengthen the test and measure coverage. Try to follow the recommendations of good test case selection criteria.

References

Tools

C++Test; ParaSoft Corporation; http://www.parasoft.com/

CTA++ (C++ Test Aider); http://www.testwell.fi/ctadesc.html

CppUnit; SourceForge; http://cppunit.sourceforge.net/

csUnit; "Complete Solution Unit Testing" for Microsoft .NET; csUnit.org; http://www.csunit.org/

CUT; SourceForge; http://sourceforge.net/projects/cut/

unit++; SourceForge; http://unitpp.sourceforge.net/

Cantata++; IPL; http://www.ipl.com/tools

.TEST; ParaSoft Corporation; http://www.parasoft.com/

Text

[Deveaux et al, 1999] DEVEAUX, D.; JÉZÉQUEL, J-M.; TRAON, Y.L.; "Self-testable components: from pragmatic tests to design-for-testability methodology"; *Proceedings of the (TOOLS 1999) Technology of Object Oriented Languages and Systems*; Los Alamitos, CA: IEEE Computer Society; 1999; pags 96-10

[Hunt and Thomas, 2003] HUNT, A.; THOMAS, D.; *Pragmatic Unit Test: in Java with JUnit*; Sebastopol, CA: O'Reilly; 2003; pags 65-78

[Staa, 1993] STAA, A.v.; *Ambiente de Engenharia de Software Assistido por Computador - TALISMAN*; versão 4.4; Rio de Janeiro, RJ: Staa Informática Ltda.; 1993 (in Portuguese)

[Staa, 2000] STAA, A.v.; *Programação Modular*; Rio de Janeiro, RJ: Elsevier/Campus; 2000 (in Portuguese)

[Staa, 2011] STAA, A.v.; *Overview of the Talisman Version 5 Software Engineering Meta-Environment*; Monografias em Ciência da Computação MCC, Rio de Janeiro: Departamento de Informática, PUC-Rio; 2011