Projeto Persistência

A.v. Staa

LES - Laboratório de Engenharia de Software
Departamento de Informática
Pontifícia Universidade Católica
22453-900 Rio de Janeiro,
Brasil

Relatório LES

MCC-

Projeto Persistência

A.v. Staa
arndt@inf.puc-rio.br

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

Julho 2002

Relatório LES xx/xxxx

Resumo

Neste documento

Palavras chave:

Abstract

In this document

Keywords:

1. Introdução

Bases de dados relacionais são consideradas inadequadas para aplicações do gênero CAD e CAE [bases OO]. Nestes casos recorre-se a bases de dados orientadas a objetos. Embora exista uma proposta de padrão da OMG [sgbdoo] as implementações nem sempre aderem a estes padrões. Além disso, os sistemas de gerência de bases de dados OO tendem a ser caros e complexos, tornando elevados os custos de institucionalização e distribuição. Freqüentemente deseja-se simplesmente dispor de uma infra-estrutura de persistência local ao invés de utilizar um sistema de gerência de bases de dados capaz de atender simultaneamente diversos usuários. Torna-se interessante, então, desenvolver uma biblioteca de persistência simples e facilmente instanciável para diferentes aplicações. Embora uma tal biblioteca não implemente uma parte significativa das funcionalidades de um sistema de gerência de bases de dados, a sua virtude reside na simplicidade de sua interface e na simplificação dos contratos de distribuição do software que as usa.

2. Arquitetura

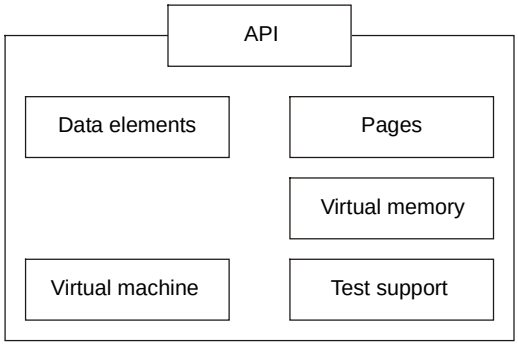

Figura 1. Arquitetura da biblioteca de persistência

A Figura 1 esboça a macro-arquitetura da biblioteca de persistência. A seguir descreveremos, em linhas gerais, cada um dos elementos.

2.1. *Virtual machine*

A ***Máquina Virtual*** estabelece a interface com a plataforma real. Algumas das classes que formam este componente podem ter que ser adaptados para assegurar que executem corretamente em uma dada plataforma.

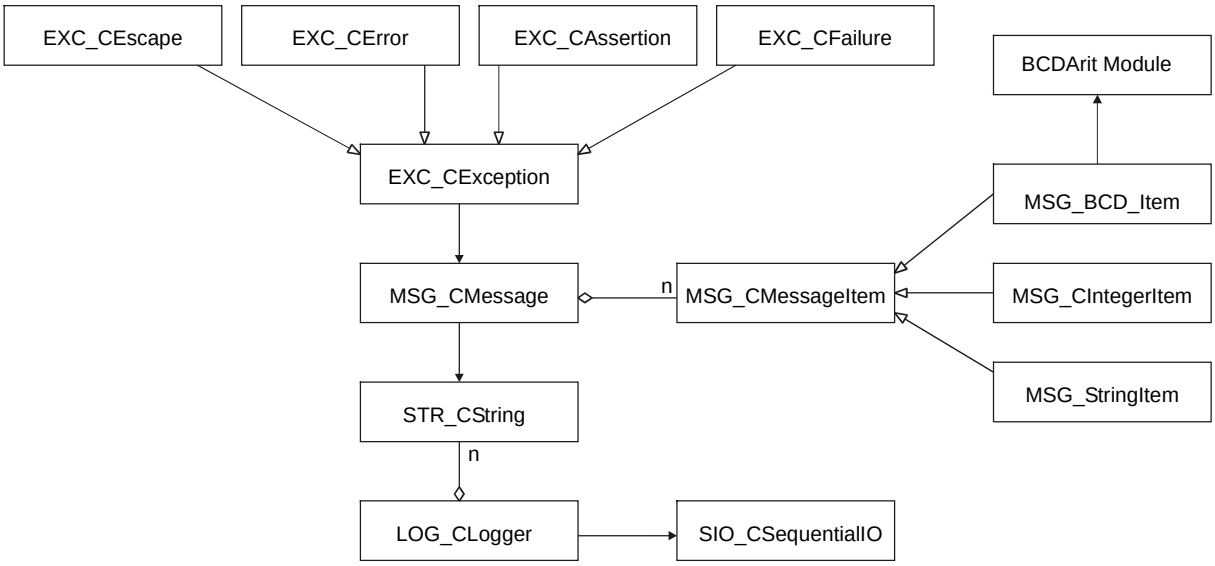

Figura 2. Diagrama de classes da máquina virtual Talisman

A Figura 2 esboça a estrutura de classes da máquina virtual Talisman.

Para assegurar portatilidade dos dados armazenados, todos os números são armazenados em formato BCD. Este formato codifica em cada 4 bits um dígito decimal. Os números negativos são armazenados na forma de complemento de dez. Cada número BCD pode ser tratado como uma seqüência de caracteres ASCII.

Mensagens identificam um *string* contido em uma tabela. Este *string* pode conter campos. Cada campo será preenchido com um item de mensagem. À medida que forem surgindo novos tipos, novas subclasses de `MSG_CStringItem` devem ser criadas.

A classe `STR_CString` implementa *strings* de tamanho variável armazenados de forma otimizada em memória dinâmica. A classe provê todos os operadores convencionais de *string*. Um dos construtores é capaz de buscar o valor *string* de uma tabela a partir de sua identificação. As tabelas de *string* e as tabelas contendo os as definições das chaves de acesso a estes *strings* são geradas por intermédio de ferramentas.

As exceções referenciam objetos `MSG_CMensagem` devidamente inicializados. Isto permite ao tratador de exceções determinar como proceder, caso seja sensível às condições que deram margem à sinalização da exceção. Estão definidas os tipos de exceção:

`EXC_CEscape` exceção interna que deve ser tratada e não gera interface com o usuário.

`EXC_CErro` erro devido a dados ou comandos errados fornecidos pelo usuário. Não impedem a biblioteca de continuar operando, mas requerem uma interação com o usuário. Advertências caem nesta categoria também.

`EXC_CAssertion` falhas de execução detectadas por alguma assertiva executável. Usualmente impedem a continuação da execução da biblioteca, porém não comprometem os dados armazenados.

`EXC_CFailure` falhas de execução detectadas por alguma assertiva executável. Sempre impedem a continuação da execução da biblioteca, e é provável que os dados armazenados estejam comprometidos.

Todas as linhas impressas são enviadas para a classe `LOG_CLogger`. Esta mantém uma lista das última *n* linhas impressas. Desta forma pode-se, sempre que desejado, rolar e redesenhar

janelas contendo mensagens do log. O *logger* imprime os resultados ou na console ou em um arquivo controlado pela classe `SIO_CSequentialIO`. Podem coexistir vários *loggers* ao mesmo tempo, possivelmente compartilhando um mesmo arquivo de saída físico.

2.2. *Test support*

O suporte aos testes provê serviços de apoio à depuração da biblioteca e, por extensão, das aplicações que dela fazem uso.

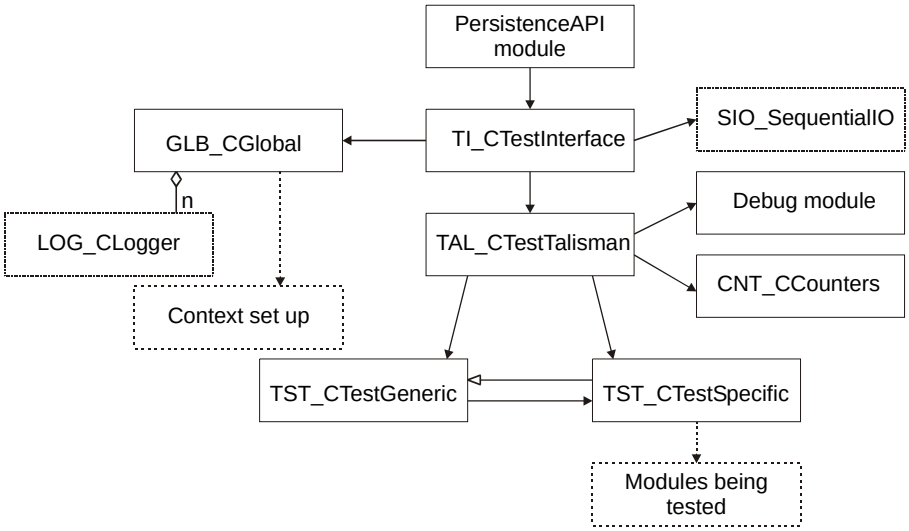

Figura 3. Diagrama de classes do suporte ao teste

A Figura 1 ilustra a estrutura de classes do ***suporte aos testes***. O desenvolvimento da biblioteca é incremental progredindo de construto para construto. Cada construto difere do anterior por adicionar poucos (em geral 1, algumas vezes até 3) módulos. Cada módulo é uma unidade compilável independentemente e contém uma ou mais classes fortemente interdependentes. Na presente arquitetura o suporte aos testes prevê o desenvolvimento de construtos formados estritamente por módulos redigidos em C++ e sem uma API bem definida.

O módulo `PersistenceAPI` implementa o programa principal. Futuramente será substituído pela API da biblioteca de persistência.

O classe `TI_TestInterface` gerencia o acesso aos arquivos de *script* de teste, bem como a geração dos *logs* de execução dos testes. O processo de teste será discutido em mais detalhe mais adiante.

A classe `GLB_CGlobal` cria e destrói o contexto necessário para a correta operação do construto. O contexto a ser estabelecido evolui à medida que a biblioteca for sendo desenvolvida.

O módulo `Debug` controla a alocação de memória dinâmica. O módulo registra todas as operações de alocação e desalocação de espaços de dados e possibilita o controle da extravasão de espaços. Facilita, desta forma, o controle e diagnóstico eventuais vazamentos de memória.

A classe `CNT_CCounters` fornece funções para a contagem de passagem da execução por pontos marcados no código. Ela tem por finalidade auxiliar na verificação da completeza dos testes

A classe TAL_TestTalisman estabelece o contexto para o teste de construtos e coordena a interpretação dos diversos arquivos contendo *script* de teste.

A classe TST_CTestGeneric cria a interface abstrata do módulo de teste específico do construto e interpreta os comandos de teste que se aplicam a todos os construtos.

A classe TST_CTestSpecific interpreta os comandos de teste específicos dos módulos a serem testados. Esta classe precisa ser adaptada para cada construto. Para simplificar esta adaptação existe uma ferramenta capaz de gerar um arcabouço específico para a função que interpreta os comandos de teste.

2.3. *Virtual memory*.

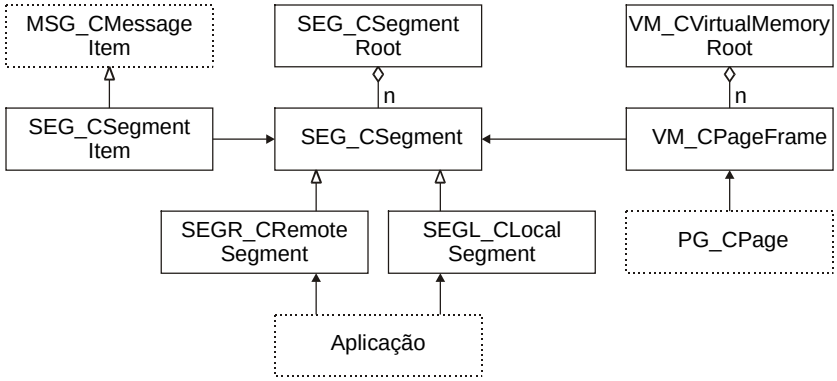

Figura 4. Estrutura de classes da memória virtual

A Figura 4 esboça a estrutura de classes da memória virtual. A memória virtual disponibiliza as classes básicas de interface entre arquivos e memória real. O esquema utilizado é similar ao de memória virtual segmentada descrita pela primeira vez no sistema operacional Multics [Denning 1970]. Neste esquema, ao invés de se ter um grande arquivo utilizado para paginação e compartilhado por todas as aplicações, tem-se um número indefinido de arquivos. Cada um destes arquivos implementa um segmento de memória virtual. Cada segmento persiste um conjunto coeso de dados. O espaço de páginas em memória real é compartilhado por todos os segmentos. Este esquema permite manter em memória persistente toda a estrutura de objetos, trazendo-os para memória sob demanda e sem precisar recorrer a maciças serializações. Isto é possível uma vez que os endereços virtuais internos a um segmento são invariáveis de uma instância de uso para outra.

A classe *singleton* SEG_CSegmentRoot coordena o acesso a todos os segmentos em uso pelo aplicação. Esta classe deve ser instanciada ao iniciar a execução.

A classe SEG_CSegment é uma classe abstrata que define a interface das classes de acesso aos dados contidos em arquivos específicos. Implementa, também, uma série de métodos comuns a todos os segmentos.

A classe SEG_CSegmentItem gera os nomes de segmentos utilizados em campos de mensagens.

A classe SEGL_CLocalSegment implementa o acesso a segmentos visíveis ao sistema de arquivos da estação de trabalhos. Objetos segmento local ou remoto devem ser criados diretamente pela aplicação. Na aplicação estes objetos são conhecidos por um identificador usualmente chamado idSeg.

A classe SEGR_CRemoteSegment implementa o acesso a segmentos via Internet.

A classe *singleton* `VM_CVirtualMemoryRoot` coordena a criação do conjunto de portadores de páginas residentes em memória real. O objeto deve ser criado ao iniciar a execução da aplicação.

A classe `VM_CPageFrame` implementa as operações de acesso a determinado portador de página. Uma página contida em um segmento é copiada para um portador. Este portador pode estar sendo acessado por um objeto página. Enquanto estiver em uso por um objeto página o conteúdo do portador não é removido da memória real. Caso um objeto página altere o conteúdo de um portador, deve sinalizar que a página foi alterada. Isto ativará o processo de gravação do conteúdo da página.

2.4. *Pages*

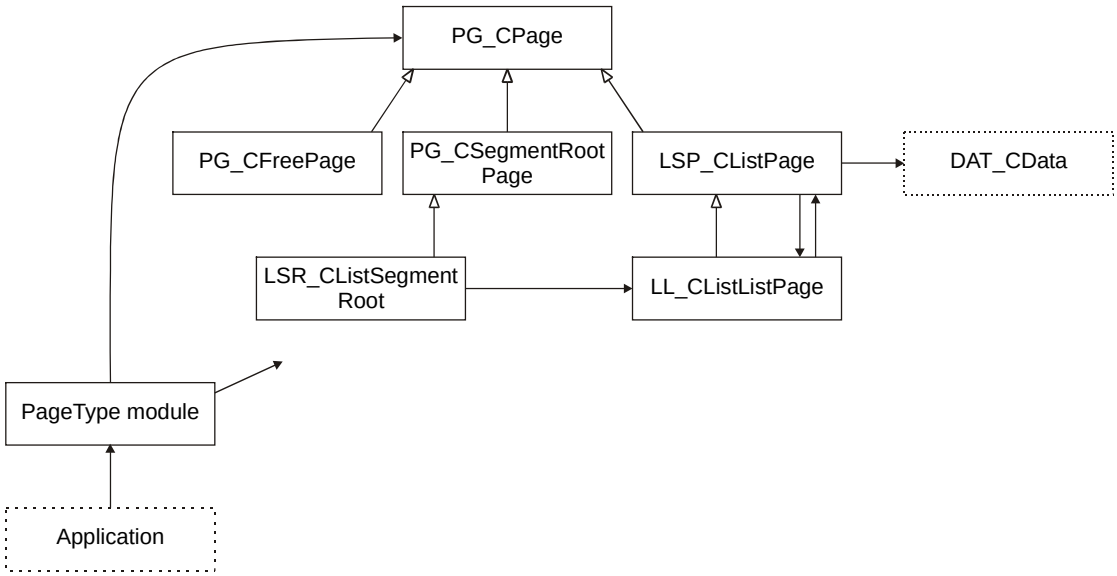

Figura 5. Estrutura de classes acesso a elementos

A Figura 5 Ilustra a estrutura de classes atual. Esta estrutura está parcialmente implementada. No entanto, pretendo modificá-la para que se aproxime mais dos requisitos de uma base dados orientada a objetos.

A classe `PG_CPage` instancia objetos referenciando páginas em memória real. Não é feita cópia do conteúdo, ao invés disso o objeto página referencia a página contida num portador de memória virtual. Para evitar ambigüidade utilizamos o termo ***objeto página*** para denotar objetos em memória real que referenciam alguma página contida em algum segmento. E utilizamos o termo ***valor página*** para denotar o vetor de caracteres que corresponde a uma página contida no segmento.

Objetos página podem estar preenchidos ou não. Quando preenchido, um objeto página referencia um valor página contido em um portador da memória virtual, sendo que, ao preencher o objeto (métodos `Build...`), é sempre verificado se o tipo do objeto é consistente com o tipo do valor, gerando uma exceção erro caso não o seja. Enquanto existir pelo menos um objeto página referenciando um determinado portador, este não será desalocado. Diversos objetos página podem referenciar um mesmo portador. Por enquanto não está disponível um sistema de bloqueio para evitar conflitos de atualização.

Do ponto de vista de um segmento, páginas são vetores de caracteres de `TAL_PageSize` caracteres. Todos os dados de um segmento são armazenadas em páginas. Para assegurar corretude de tipos, cada página possui um identificador de tipo. Estes tipos correspondem às

classes da estrutura de classes na Figura 5. Ou seja, a estrutura de classes evidencia tanto a organização dos objetos em memória real, como a organização do segmento.

O módulo `PageType` possui uma função que atua como fábrica de objetos, criando os objetos consistentes com o tipo do valor página a ser utilizada para preenche-lo. Internamente a uma página os dados podem ser organizados de qualquer forma, sempre observando que o primeiro par de bytes identifica o tipo do valor página.

A primeira página de um segmento é a raiz do segmento e contém informações de controle do segmento como um todo, além das referências para as estruturas de dados armazenadas no segmento. A raiz é, portanto, a cabeça de todas as estruturas armazenadas no segmento. Segmentos podem ser utilizados para diversas finalidades, consequentemente existe uma estrutura de herança com origem em `PG_CSegmentRootPage` e que identifica as possíveis raízes de segmento. Isto permite o desenvolvimento incremental da estrutura de segmentos. Cabe salientar que a estrutura de dados de uma determinada classe raiz é formada pela concatenação das estruturas de dados das raízes de que herda.

Um segmento pode ser entendido como sendo um vetor de páginas. Desta forma o identificador de página (`idPag`) nada mais é do que um índice deste vetor. Potencialmente um segmento pode crescer até `2**32*TAL_PageSize` bytes. Uma vez adicionada uma página a um segmento, ela não pode mais ser removida, já que não existe um ordenamento das operações de inserção e exclusão de páginas. Cada segmento mantém uma lista de páginas livres. Esta lista contém as zero ou mais páginas que foram desalocadas. Sempre que for solicitada a alocação, primeiro tenta-se retirar uma página da lista livre, somente é adicionada uma nova página ao segmento se e a lista estiver vazia. Evita-se, assim o crescimento descontrolado do segmento. No entanto, cabe à aplicação cliente desalocar as páginas que se tornarem desnecessárias. O controle da lista livre é realizado pela classe `PG_CFreePage`.

A classe `LST_CListPage` implementa listas genéricas. Cada lista possui um nome e deve ser registrada na classe `LL_CListListPage`. Desta forma cada lista poderá ser recuperada pelo seu nome simbólico. Uma lista poderá conter valores de qualquer tipo e de tamanho variável, sendo que cada elemento pode em princípio ter um tipo diferente dos demais elementos. A classe fornece diversos operadores necessários para a exploração seqüencial de uma lista. Cabe ao cliente identificar o tipo do objeto a ser acessado. O endereço virtual de um elemento de lista é `<segmento, idPag, inxElem>`, no qual:

`segmento` identifica o segmento que contém a lista;

`idPag` identifica a página que contém o elemento;

`inxElem` identifica o índice do elemento dentro da página.

Na atual implementação este endereço é efêmero pois poderá deixar de valer caso a lista seja alterada. Isto porque ao altera uma lista elementos podem ser movimentados entre páginas afim de assegurar suficiente espaço para acomodar o novo elemento.

2.5. *Data Elements*

A classe abstrata `DAT_CData` define a interface que todos os elementos devem satisfazer. Objetos devem ser armazenados em um formato neutro com relação à plataforma, tipicamente um *string* de caracteres ASCII codificado na forma `< Tamanho , string >`.

3. Processo de desenvolvimento e teste

3.1. The generic build programming process

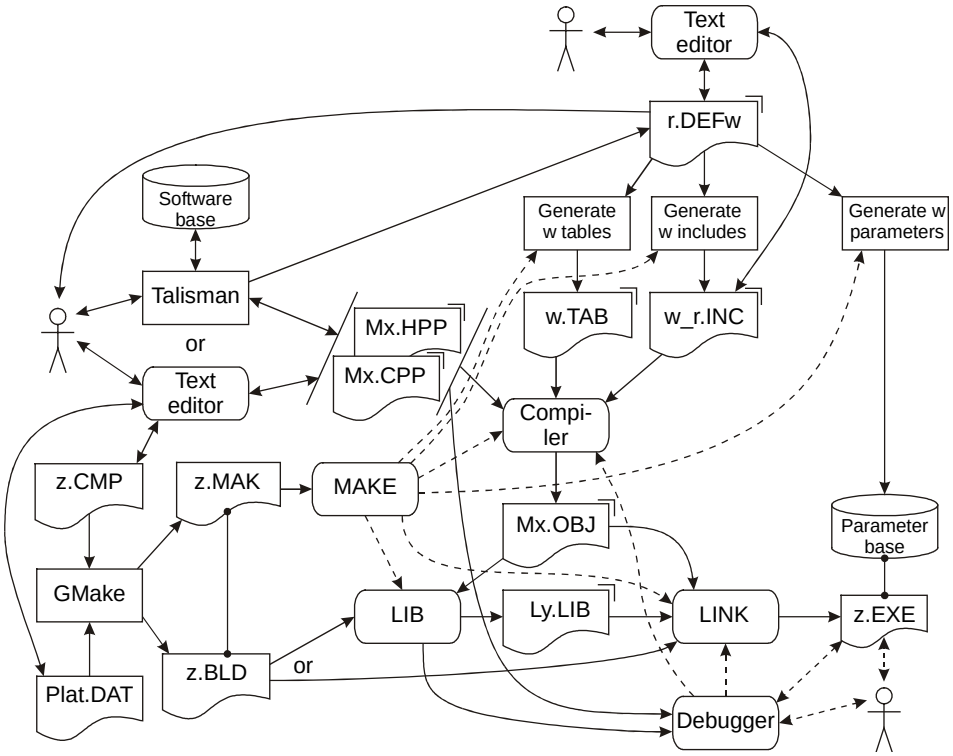

Figure 6. The build programming process

Figure 6 shows the standard build* development process. The process is iterative and progresses from build to build. Each build is a working program implementing a coherent set of the modules†. The aim of a build is assessing the quality of a very small number (usually one) of new modules within the context of a possibly large set of already accepted modules. In addition to correctness, several other execution properties of the modules under test are assessed, such as reliability, robustness, performance, absence of memory leaks, absence of resource leaks, absence of concurrency problems, and fitness for use.

The meaning of the elements is:

- *straight angled rectangle* represent tools or programs developed within the Talisman developing environment;
- *round angled boxes* represent COT‡ tools;
- *listing like boxes* represent sequential files;
- *disk drive like boxes* represent direct access files;
- *full line arrows* represent flow of data;
- *dashed arrows* represent flow of control;

---

* A ***build*** is a partial implementation of a system and is used to assess the quality of its components.

† A ***module*** is an independently compilable file. In some cases this file requires other files (e.g. #include), in other cases the file must be composed prior to compilation.

‡ ***COT*** – commercial of the shelf.

- *full line bubble ended lines* represent interdependent elements;
- *puppets* represent users interacting with some program.

Users develop modules using either Talisman* [Staa 1993] or some other tool (editor, IDE). In this paper we assume that Talisman will be used.

All data involving detailed design and code of one or more modules is kept in a software base. This is a special purpose database specifically designed to support Talisman. Due to restrictions of the old version, the project is composed of several independent software bases, each of which containing a small number of modules. Usually these are a production module and its specific test module. Once developed or modified a module is linearized†, generating implementation files, header files or script files. Talisman implements a programming language that can be used, among others, to program a large variety of linearizers and transformers. Rewriting linearization programs allows the composition of programs for a variety of languages, such as C, C++ and Java. A specific linearizer has been developed for the Talisman project itself.

Several modules interpret some form of byte-code, table, or symbolic text. The source text of the code or tables to be interpreted is kept in `.Dw` files, where "`Dw`" identifies the type of script file. Depending on the type of the script language, script files might be edited using some text editor or may be generated using Talisman. An attempt is made to keep the syntax of all script languages as simple as possible, usually in the form sequence of "command list-of-parameters". If more structure is required, a syntax similar to XML could be used.

Script files are transformed by tools into files to be used by a compiler or by the executable program. The `GeneratewTables` tools usually transform the script file into declarations that are compiled producing memory resident tables. However, some of these tools generate executable code fragments to be included by specific modules. The `GeneratewIncludes` tools transform the script into a file containing a list of constants. These constants are required by the interpreters in order to access specific elements of a memory resident table or parameter base. Finally, the `GeneratewParameters` tools transform the script file into a set of directly accessible elements contained in the parameter base. When script files are converted to data structures or byte code, specific modules must be developed to interpret them. Each of the script languages requires the development of specific instances of these tools.

The composition of a build is defined in a `.CMP` file. This script file describes the folder structure of the project and enumerates the script files and modules that compose the build. The `Plat.DAT` file contains data describing how to generate make script files for a given platform. A special tool, `GMake`, generates the make script file for the build. This script file assures that all modules are compiled and all required script files are correctly transformed into include files (`.INC`), memory resident table declaration files (`.TAB`) or added to the parameter base.

A minor problem arises when attempting to compile a build for debugging purposes. The make generators available in integrated development environments (IDE) usually do not support user provided tools. Furthermore, due to the heavy use of instrumentation and automatic testing, debuggers will seldom be needed. If necessary, the program might be compiled using the tools of the process and, then, recompiled for debugging purposes using

---

* The tool currently in use is an old embryonic MS-DOS version of Talisman. Once a workable version of the new tool becomes available, it will replace the old one, possibly entailing a slight change in the process.

† ***Linearization*** is a process that generates sequential ASCII files from the contents of the software base.

an appropriate IDE. This assures the generation of all non-code files required to correctly compile and use the build. Furthermore, it entails very little extra effort when setting up a build for debugging within the IDE.

3.2. Testing a build

Each build adds few new modules to the set of already accepted modules. These modules must be tested in order to accept the build. Only after accepting a build one may advance to a new one. The development process deploys a test framework that aids the automatic testing of builds.

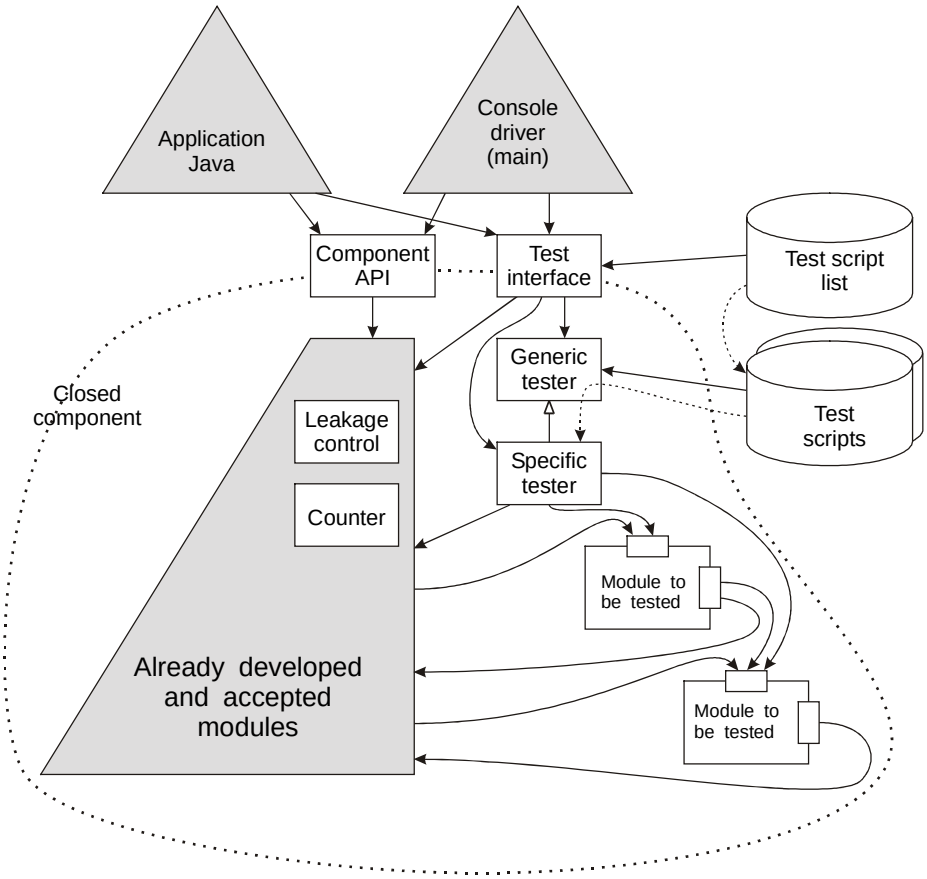

Figure 7. Architecture of the testing framework

Figure 7 shows the architecture of the test framework. Tests can be performed from two perspectives. The usual perspective is to test a build considering a single language implementation. However, Talisman will be developed as a hybrid program using Java and C++ modules. In order to verify correct working of all modules, tests must be performed in both contexts. The single language context eases the debugging of the modules under test. The hybrid language context (Java) contributes to assuring the proper functioning of the modules in the production context.

The modules to be tested may interact with the modules that have already been accepted. This reduces the amount of work setting up testing harnesses for modules under test. Among the accepted modules we find two modules specifically aimed at test support. The *leakage control* module permits the verification of memory leakage at specific points within a test script. Usually test scripts control leakage only at the end of a script file, although the test scripts may contain leakage tests in many places. The *counter* module counts the number of

times execution has passed a given point in a program. These points are identified by a name. Marking up the program with passage counters creates a simple mechanism for controlling test coverage in a variety of degrees of resolution.

Talisman is able to linearize code inserting counters for every function or method, and every pseudo-code block. Although this is quite a superficial counting scheme it has proved itself reasonably effective in verifying the completeness of test scripts.

4. Plano de desenvolvimento

Durante as conversas mantidas com Antonio e Casanova concluímos que a primeira versão poderia ser formada meramente pelo armazenamento e exploração de dados contidos em listas. Para viabilizar esta versão precisam ser feitas algumas adaptações no código existente.

4.1. Versão 1

- ajustar as classes da estrutura de acesso a listas de elementos contidos em páginas.
- criar uma versão simplificada da classe `DAT_CData` capaz de receber dados em uma codificação binária (`struct`), inserindo-os em uma dada lista, bem como recuperar dados nesta codificação. Nesta versão não será feita qualquer consideração quanto à portatilidade dos dados. Além disso cada linha corresponde a um `struct` físico, qualquer alteração na declaração ou em parâmetros de otimização de *layout* de dados levará ao mal funcionamento da biblioteca.
- assistir os programadores clientes no uso da biblioteca estática.
- desenvolver um exemplo de uso da biblioteca.
- avaliar a usabilidade e adequação da biblioteca.

Resultados a serem entregues

- documentação da macro-arquitetura.
- documentação técnica incluída no código. Está em desenvolvimento um sistema de geração de uma documentação técnica explorável remotamente. Diferente de JavaDoc, o sistema cria e explora um banco de dados contendo a documentação. A base de dados é povoada a partir do código fonte a ser documentado.
- *scripts* de teste.
- código fonte em C++.
- exemplo de uso.
- lista de problemas* a serem resolvidos em versões posteriores.

4.2. Versão 2:

- avaliar e resolver problemas.
- criar uma classe de interface (API) cuja finalidade é simplificar o uso da biblioteca e encapsular funções que o programador cliente não necessita ter.

---

* O termo ***problema*** é usado de forma abrangente, denotando coletivamente: solicitações de evolução, melhorias e adaptação, além de falhas e defeitos observados.

- encapsular o conjunto de módulos em uma DLL.
- desenvolver um exemplo de uso da DLL.
- avaliar a usabilidade e adequação da DLL.
- avaliação dos resultados desta versão e geração de uma lista de problemas a serem resolvidos em versões posteriores.

Resultados a serem entregues:

- Documentação de uso da API.
- biblioteca DLL implementando a biblioteca de persistência.
- documentação técnica incluída no código.
- Arcabouço de teste envolvendo a DLL.
- *scripts* de teste.
- código fonte em C++.
- lista de problemas a serem resolvidos em versões posteriores.

4.3. Versão 3:

- avaliar e resolver problemas pendentes.
- criar uma classe `STR_CStruct` capaz de definir simbolicamente a organização de uma linha de uma tabela, fornecendo funções para a inserção e recuperação de atributos a partir de seus nomes simbólicos.
- criar um interpretador para expressões lógicas capaz de selecionar elementos de uma tabela segundo uma expressão lógica simples. Os algoritmos não procurarão otimizar a operação `select`. O `select` será restrito a uma única tabela e produzirá uma lista de valores cuja estrutura é idêntica à estrutura da tabela sobre a qual trabalha.
- implantação do sistema de apoio ao desenvolvimento de software *open source*.
- ajustar a API para incorporar as novas funcionalidades.

Resultados a serem entregues

- documentação da API ajustada para a nova versão
- nova versão da biblioteca DLL
- *scripts* de teste.
- código fonte em C++.
- sistema de apoio ao desenvolvimento de software *open source*.
- lista de problemas a serem resolvidos em versões posteriores.

4.4. Versão 4:

- avaliar e resolver problemas pendentes.
- implantação do sistema de apoio ao desenvolvimento de software *open source*.

Resultados a serem entregues

- *scripts* de teste.
- código fonte em C++.
- sistema de apoio ao desenvolvimento de software *open source.*
- lista de problemas a serem resolvidos em versões posteriores.

Referências bibliográficas

basesOO
sgbdOO

[Bennet 1989] Bennet, K.H. (ed.); *Software Engineering Environments, Research and Practice*; Ellis Horwood; Chichester, England; 1989

[Brown 1989] Brown, A.W.; *Database Support for Software Engineering*; London, Kogan Page; 1989

[Cat 1994] Cattell R.G.G.; *Object Data Management, Object Oriented and Extended Relational Database Systems*; Addison Wesley; 1994

[Denning 1970] Denning, P.J.; "Virtual memory"; *ACM Computing Surveys* 2(3); New York; 1970; pags 153-189

[Deux 1991] Deux, O. et al.; "The O2 System", *Communications of the ACM* 34(10); Oct. 1991.

[Dittrich 1989] Dittrich, K.R.; "The DAMOKLES Database System for Design Applications: its Past, its Present, and its Future"; in [Bennet89]; pp 151-171

[ESW 1993] Emmerich, W.; Schäfer, W.; Welsh, J.; "Databases for Software Engineering Environments -- the Goal has not yet been attained"; in *ESEC'93 4th European Software Engineering Conference*; Sommerville, I.; Paul, M. eds; *Lecture Notes in Computer Science no. 717*; Springer; 1993; pp 145-162

[Kim 1990] Kim, W.; *Introduction to Object Oriented Databases*; Cambride, Massachusetts, MIT Press; 1990

[Staa 1993] *Ambiente de Engenharia de Software Talisman, Manual do Usuário*; Staa Informática; Rio de Janeiro; 1993