Introdução

Este projeto consiste numa *Framework*[1] para testes unitários[2].

Os testes unitários podem ser vistos como um complemento ao compilador. Sendo uma responsabilidade do compilador a verificação da sintaxe do código, a responsabilidade dos testes unitários é a verificação da conformidade da sua execução segundo os critérios do programador.

Uma *Framework* para testes unitários deve proporcionar ao programador facilidade na realização e execução de testes unitários ao seu código.

Na Framework que se pretende desenvolver serão fornecidos os nomes das classes de teste. Estas classes deverão estar decoradas com uma anotação que explicita que a classe é uma classe de teste. Nessa classe estarão os métodos de teste, também decorados com uma anotação específica.

Para avaliar a execução dos métodos em teste usar-se-ão pares avaliador-anotação. Para determinar quais os métodos a avaliar, utiliza-se a referida anotação específica. De seguida são procuradas demais anotações conhecidas, para cada uma das quais é executado o avaliador correspondente.

Para gerir a interface com o utilizador existirão visualizadores dos resultados dos testes.

A referida Framework tem ainda o objectivo de ser extensível a novos tipos de avaliações sobre os métodos e a novos tipos de visualizadores dos resultados. Para tal, serão minimizadas as dependências tanto com os pares avaliador-anotação como com os visualizadores e os mesmos serão carregados dinamicamente conforme os parâmetros de arranque da Framework.

[1] *Framework* – Conjunto de módulos de código que colaboram para realizar uma responsabilidade para um domínio de um sistema.
[2] Testes unitários – Código que quando executado verifica a conformidade de uma unidade de código.

Avaliação

Para efeitos da verificação de conformidade de execução dos métodos de teste são utilizados pares avaliador-anotação. A anotação tem por objectivo assinalar determinada avaliação sobre o método anotado. O avaliador é responsável por realizar uma verificação específica sobre um método decorado com a sua anotação correspondente.

Devido às limitações da linguagem Java (na qual é implementada a Framework) quanto às anotações, não é possível responsabilizar as anotações por avaliar os métodos copulando a anotação com a avaliação. De forma a contornar esta limitação ***o avaliador é responsável por indicar qual a sua anotação correspondente***. Assim, é apenas necessário ter conhecimento dos avaliadores, uma vez que as anotações podem ser obtidas através dos mesmos.

De modo a minimizar a dependência com os avaliadores foi criada uma interface cuja implementação é exigida a todos os avaliadores. A Framework permanece agnóstica às implementações dos avaliadores conhecendo somente a interface.

A Figura 1 ilustra a interface de um avaliador.

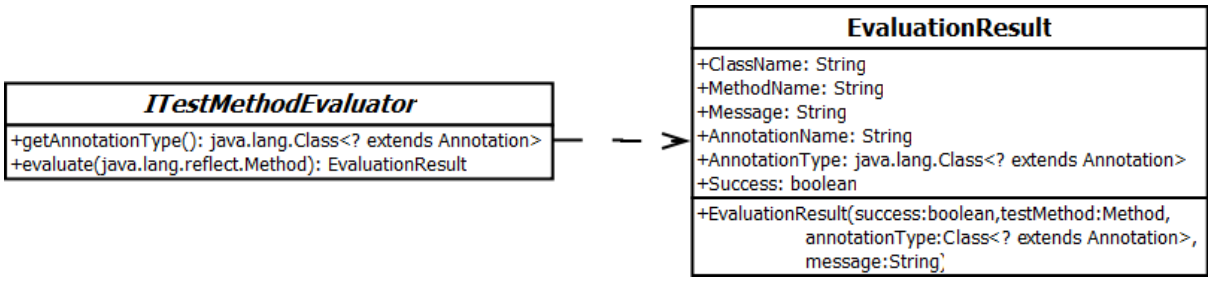

Figura 1 - Interface ITestMethodEvaluator

A um avaliador é exigido o tipo da sua anotação correspondente de forma permitir à Framework saber que anotação procurar nos métodos de teste para averiguar se uma avaliação é necessária. O método `evaluate` realiza a avaliação sobre um método passado por parâmetro e deve produzir uma instância de `EvaluationResult`. Este tipo funciona como um relatório de uma avaliação. Possuiu campos públicos inalteráveis com informação acerca da avaliação: o nome da classe, do método e da anotação, indicação de sucesso e uma mensagem.

O facto da invocação do método ser do cargo dos avaliadores pode ser entendido como uma ineficiência, visto que é provável que um método seja executado tantas vezes quantos forem os avaliadores cuja respectiva anotação decore o método. Isto porque admite-se que cada um dos avaliadores terá que invocar o método para realizar as suas verificações. É portanto uma decisão que prejudica a desempenho da Framework.

Por outro lado, a única forma de evitar este inconveniente seria recolher um conjunto restrito de informação sobre a execução de cada método de teste e os avaliadores fariam as suas verificações sobre esses dados recolhidos. Recolher todas as informações que possivelmente algum avaliador pretenderia verificar é simplesmente quimérico. É natural e complacente perceber que esta solução seria funesta para com todo o princípio de extensibilidade da Framework e portanto errónea.

Se nos lembrarmos de que nos testes unitários um tempo de execução mitigado pode ser valioso mas nunca constitutivo da sua essência, a escolha entre os dois paradigmas aproxima-se do incontestável. Sacrifica-se alguma eficiência pela conservação da extensibilidade.

Permite-se a extensibilidade da Framework a novos avaliadores aceitando como parâmetro de configuração os nomes das implementações de `ITestMethodEvaluator` a utilizar.

São oferecidas as seguintes implementações:

- `ReturnsEvaluator`
  - Confirma a representação do valor retornado pelo método.
- `MaximumTimeEvaluator`
  - Verifica o tempo gasto na execução de um método.
- `ExpectedExceptionEvaluator`
  - Examina o método quanto ao lançamento de excepções.

Para que as anotações estejam disponíveis em tempo de execução e consequentemente ao alcance dos serviços de reflexão. É necessário que as mesmas sejam decoradas com: "`@Retention(RetentionPolicy.RUNTIME)`".

ReturnsEvaluator

Os métodos decorados com a anotação `Returns` são avaliados com esta implementação de `ITestMethodEvaluator`. Um teste a um método decorado com este tipo de anotação é considerado bem sucedido quando o valor retornado pela sua execução tem a representação indicada na anotação. Se o valor retornado não possuir a representação esperada ou se uma excepção for lançada no método a avaliar e consequentemente nenhum valor for retornado o teste é considerado falhado.

MaximumTimeEvaluator

Este avaliador é utilizado para testar os métodos decorados com a anotação `MaximumTime`, onde se indica o tempo máximo aceitável em milissegundos para a execução do método.

O método é avaliado pelo tempo que demora a executar, independentemente do seu retorno ou de possíveis excepções lançadas. Se o tempo gasto for menor do que aquele especificado na anotação, então o teste é considerado bem sucedido.

ExpectedExceptionEvaluator

Na presença da anotação `ExpectedException` este avaliador é utilizado para verificar se a excepção lançada no método a ser testado é do tipo indicado na anotação. No caso de na anotação não ser referida nenhum tipo de excepção é esperado e o teste é considerado falhado se qualquer excepção for lançada.

AbstractEvaluator

Com vista em facilitar a implementação da interface `ITestMethodEvaluator` é oferecida a classe base `AbstractEvaluator`. Métodos auxiliares declarados nesta classe proporcionam uma forma simplificada de invocar o método de teste e de extrair a respectiva anotação.

Visualização

Para mostrar os resultados ao utilizador são utilizados visualizadores de resultados. Com o intuito de minimizar as dependências da Framework com estes visualizadores é utilizado um interface.

A Figura 2 mostra a interface que um visualizador de resultados necessita de respeitar.

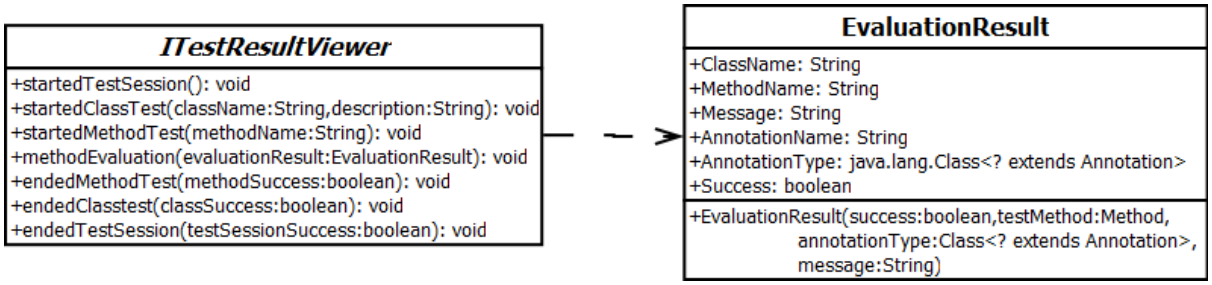

Figura 2 - Interface ITestResultViewer

A interface funciona por eventos, ou seja, os visualizadores são notificados à medida que sucedem as operações fundamentais. Utilizando este modelo de notificação é possível evitar-se acumular grandes porções de informação antes de a transferir para os visualizadores. As implementações do interface decidem quais os eventos que lhes interessam, decidindo para cada um deles o que fazer com a informação recebida.

Permite-se a extensibilidade da Framework a novos visualizadores de resultados de sessões de teste aceitando como parâmetro de configuração os nomes das implementações de `ITestResultViewer` a utilizar.

Para a implementação de visualizadores, cumpriu-se com a utilização do padrão *Observer*. No caso concreto desta Framework, utilizaram-se três visualizadores distintos, que permitem a consulta dos resultados quer em formato XML, através de uma lista e/ou através de tabela.

Há que ressalvar que na utilização destes componentes de Java, existe a utilização implícita de um *Model-View-Controller* (MVC).

Disto isto será explanado, em cada um dos seguintes componentes quais os tipos que suportam cada um dos intervenientes no MVC.

XmlTestResultViewer

Uma das implementações de visualizador de resultados é o `XmlTestResultViewer` que produz para um ficheiro XML[3] os resultados da sessão de teste.

Os resultados estarão descritos num idioma XML criado especificamente para este problema.

A Figura 3 mostra um exemplo de um resultado de uma sessão de testes no idioma referido.

[3] XML (Extensible Markup Language) - Recomendação da W3C para criar linguagens de *markup*.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE testSession (View Source for full doctype...)>
<testSession time="03-07-2009 17:29">
    <testClass name="MaximumTimeEvaluatorTest"
        description="A class that tests the MaximumTimeEvaluator">
        <testMehtod name="testMethod1">
              <methodEvaluation success="Failed"
                    annotationType="MaximumTime">
                Failed. Max time: 10ms. Time taken: 26ms.
            </methodEvaluation>
            Failed
        </testMehtod>
        <testMethod name="testMethod2">
            <methodEvaluation success="OK" annotationType="MaximumTime">
                OK. Time taken: 20ms
            </methodEvaluation>
            OK
        </testMethod>
        Failed
    </testClass>
    <testClass name="ExpectedExceptionEvaluatorTest"
        description="A class that tests the ExpectedExceptionEvaluator">
        <testMethod name="testMethod3">
          <methodEvaluation success="OK" annotationType="ExpectedException">
                OK.
          </methodEvaluation>
          OK
        </testMethod>
        <testMethod name="testMethod4">
          <methodEvaluation success="OK" annotationType="ExpectedException">
                OK.
          </methodEvaluation>
          OK
        </testMethod>
        OK
    </testClass>
    <testClass name="ReturnsEvaluatorTest" description="A class that tests the
    ReturnsEvaluator">
        <testMethod name="testMethod5">
            <methodEvaluation success="OK" annotationType="Returns">
                OK.
            </methodEvaluation>
            OK
        </testMethod>
        <testMethod name="testMethod6">
            <methodEvaluation success="Failed" annotationType="Returns">
                Failed. Expected: Green . Result: Red .
            </methodEvaluation>
            Failed
        </testMethod>
        Failed
    </testClass>
    Failed
</testSession>
```

Figura 3 - Exemplo de um ficheiro XML com os resultados de uma sessão de teste

O idioma segue o DTD[4] definido na Figura 4.

[4] DTD (*Document Type Definition*) – Documento que contém as regras que definem quais os elementos que podem ser utilizados num documento XML e quais os valores válidos.

```
<!ELEMENT testSession         (testClass*)>
<!ELEMENT testClass           (testMehtod*)>
<!ELEMENT testMehtod          (methodEvaluation*)>
<!ELEMENT methodEvaluation    (#PCDATA)>

<!ATTLIST testSession         time                  CDATA   #REQUIRED>
<!ATTLIST testClass           name                  CDATA   #REQUIRED>
<!ATTLIST testClass           description           CDATA   #REQUIRED>
<!ATTLIST testMehtod          name                  CDATA   #REQUIRED>
<!ATTLIST methodEvaluation    success               CDATA   #REQUIRED>
<!ATTLIST methodEvaluation    annotationType        CDATA   #REQUIRED>
```

Figura 4 - DTD do idioma XML utilizado

O idioma escolhido define uma hierarquia de elementos de teste e os seus respectivos resultados. Nomeadamente existe o conceito de sessão de testes que por sua vez é constituída por uma quantidade variável de classes submetidas a teste. Uma classe de teste é uma classe anotada com *TestClass* com uma quantidade variável de métodos de teste, ou seja, aqueles anotados com *TestMethod*. Por sua vez cada método de teste é anotado com uma ou mais anotações que especificam o tipo de avaliação que é feita ao método. Em cada um destes quatro níveis o insucesso é implícito pela falha de um elemento de nível inferior, ou seja, para que um método de teste seja considerado falhado basta que uma das avaliações que lhe foram feitas tenha insucesso. Da mesma forma, uma classe de teste é bem sucedida se, e só se, todos os seus métodos de teste forem considerados bem sucedidos. Sendo a granularidade de teste as avaliações, para cada uma delas existe uma mensagem associada que visa esclarecer a razão de insucesso.

Uma das decisões importantes quando se realiza um componente que processa ficheiros XML é escolher qual a API[5] a utilizar. Em Java as duas APIs mais relevantes para processar ficheiros XML são a SAX (*Simple API for XML*) e o DOM (*Document Object Model*). Cada uma possui vantagens e desvantagens.

O SAX é um *standard* para APIs XML. É uma das mais completas e correctas. É uma API baseada em eventos, em que existe um método para indicar a existência do início ou fim de um elemento XML. Este modelo torna a SAX bastante eficiente em termos de processamento e de memória ocupada (uma vez que não armazena todo o documento). Quando é possível realizar um processamento sequencial (*forward-only*) do documento e não é necessário ter informação global do documento a todo o momento utilizar SAX é habitualmente a melhor opção. Devido à sua eficiência é a única escolha para processar ficheiros realmente grandes (relativamente à memória disponível). O maior inconveniente da SAX reside na complexidade adicional ao código quando se necessita de criar estruturas de dados para armazenar partes do documento.

DOM é uma API relativamente complexa em que se modela um documento XML como uma árvore. Cada elemento do documento pode ser inquirido sobre o seu conteúdo, atributos e elementos descendentes. Enquanto os acessos aleatórios são promovidos existe a contrapartida da ocupação exaustiva de memória.

Considerando que o problema que pretendemos resolver com o uso de uma API para escrita de documentos XML não exige acessos aleatórios ao documento nem representação global do mesmo, e ainda cobiçando todas vantagens que a SAX proporciona, a alternativa escolhida foi a de utilizar a SAX.

[5] API (*Application Programming Interface*) – Interface através do qual uma aplicação solicita serviços de bibliotecas de software.

Este componente foi implementado na classe `XmlTestResultViewer`. Na sua instanciação é possível definir qual a *stream*[6] de saída. Por omissão, o resultado é encontrado no ficheiro de nome "testResults.xml".

GuiJListTestResultViewer

Este visualizador permite uma consulta simples aos resultados obtidos das avaliações utilizando uma interface gráfica mais amigável que uma escrita em consola.

Este visualizador permite a utilização de facilidades na representação gráfica que acelerem o processo de identificação dos resultados, nomeadamente é utilizada uma distinção entre sucesso e insucesso através de coloração de cada item da lista em concordância com o resultado da avaliação para esse item.

Neste componente foi utilizada coloração dos itens de lista que representam os resultados de avaliação recorrendo à implementação da interface *ListCellRenderer* onde é definido o método *getListCellRendererComponent* para este efeito. A sua tarefa consiste em transformar cada item da lista num componente *swing*.

GuiJTableTestResultViewer

Adicionalmente ao visualizador em lista, foi também implementado um visualizador em tabela.

Este visualizador não pode ser estendido e, internamente utiliza como modelo de suporte aos dados a apresentar uma instância de `DefaultTableModel`, cuja decisão para a utilização é sustentada pelo facto de suportar correctamente acesso concorrente entre várias tarefas.

Por outro lado, a fundamentação principal para utilizar um `JTable` em vez de `JList` consiste no facto de permitir ordenação de cada coluna presente na tabela individualmente, o que facilita a consulta dos resultados por parte do utilizador da Framework.

Relativamente aos papéis desempenhados por cada tipo que suporta o MVC (*Model-View-Controller*), existe o seguinte:

- O *Model* é definido por uma interface e qualquer implementação desta servirá consequentemente para fornecer os dados a apresentar.
- O *View* é o componente `JTable` propriamente dito no que diz respeito ao suporte dos métodos existentes na classe abstracta *Component*.
- O *Controller* está também implementado no `JTable` e pode ser visto como suportado pelos métodos `getValueAt`, `getColumnName`, etc.

Tal como acontece no componente `GuiJListTestResultViewer`, foi implementada coloração das linhas da tabela para auxiliar na identificação dos resultados das avaliações. Desta feita, isto foi conseguido recorrendo à implementação da interface `TableCellRenderer`, definindo o método `getTableCellRendererComponent` para o efeito. Da mesma forma, é da sua responsabilidade produzir um componente *swing* a partir de cada célula da tabela.

[6] *Stream* – Fluxo de dados de um sistema computacional.

Motor

O motor (`TestEngine`) forma o cerne da aplicação, uma vez que é da sua responsabilidade rentabilizar todos os recursos que lhe são entregues. Isto consiste em produzir o resultado das avaliações efectuadas sobre um conjunto de classes e notificar os visualizadores de resultados destes eventos e dos respectivos resultados. A Figura 5 esclarece.

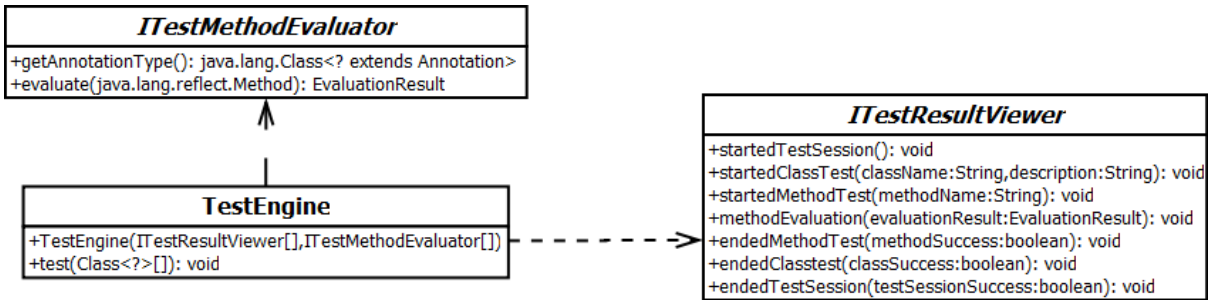

Figura 5 - Diagrama de classes do motor de testes

Nesta implementação foi utilizado o padrão *Observer*, funcionando o tipo `TestEngine` como o participante *ConcreteSubject*, a interface `ITestResultViewer` como o participante *Observer* e cada uma das implementações de `ITestResultViewer` recebidas no construtor de `TestEngine` como o participante *ConcreteObserver*. Nesta implementação não existe o participante *Subject* nem são disponibilizados métodos para registo e remoção de registo de *observers*, existindo apenas a possibilidade de registo através do construtor de `TestEngine`.

O método `test(Class<?>[] testClasses)` disponibilizado pelo tipo `TestEngine` realiza uma sessão de avaliação com base num conjunto de classes recebidas como parâmetro, e notifica todos os `ITestResultViewer` registados sobre o início e fim (com indicação do sucesso) da sessão. O sucesso de uma sessão de avaliação resulta do sucesso da avaliação realizada a todas as classes de teste, dependendo o sucesso da avaliação de cada classe do sucesso da avaliação realizada a cada um dos seus métodos em cada um dos parâmetros definidos pelas anotações que decoram os mesmos. Em conformidade com o que sucede com a avaliação de uma sessão, também existe notificação aos `ITestResultViewer` do início, fim e sucesso da execução da avaliação a cada classe e a cada método (neste caso o sucesso do teste é encapsulado num objecto do tipo `EvaluationResult`).

Na avaliação dos parâmetros definidos pelas anotações é utilizado o padrão *Strategy* de forma a garantir a extensibilidade das avaliações a realizar. O participante *Strategy* é representado pelo tipo `ITestMethodEvaluator`, o participante *ConcreteStrategy* é representado por todas as implementações de `ITestMethodEvaluator` e o participante *Context* é representado pelo tipo `TestEngine`.

Arranque

O motor de testes funciona com instâncias de visualizadores de resultados e com instancias de avaliadores. É através dos avaliadores que realiza os testes aos métodos das classes que lhe são também fornecidas. Resumindo, é necessário obter os seguintes recursos:

- Representação dos tipos das classes de teste.
- Instancias dos avaliadores de métodos de teste.
- Instâncias dos visualizadores de resultados.

No arranque da aplicação da Framework deverão ser fornecidos apenas os nomes destes tipos. Fundamenta-se pois a delegação das tarefas de recolha dos mesmos nomes e de instanciação das classes necessárias por duas entidades: `StartupConfigLoader` e `ResourceLoader` respectivamente. As informações recolhidas por StartupConfigLoader provindas dos parâmetros de arranque do programa e as informações oriundas do ficheiro de configuração são formatadas no tipo `StartupInfo` de modo a serem entregues ao `ResourceLoader`. O diagrama presente na Figura 6 realça estes pormenores.

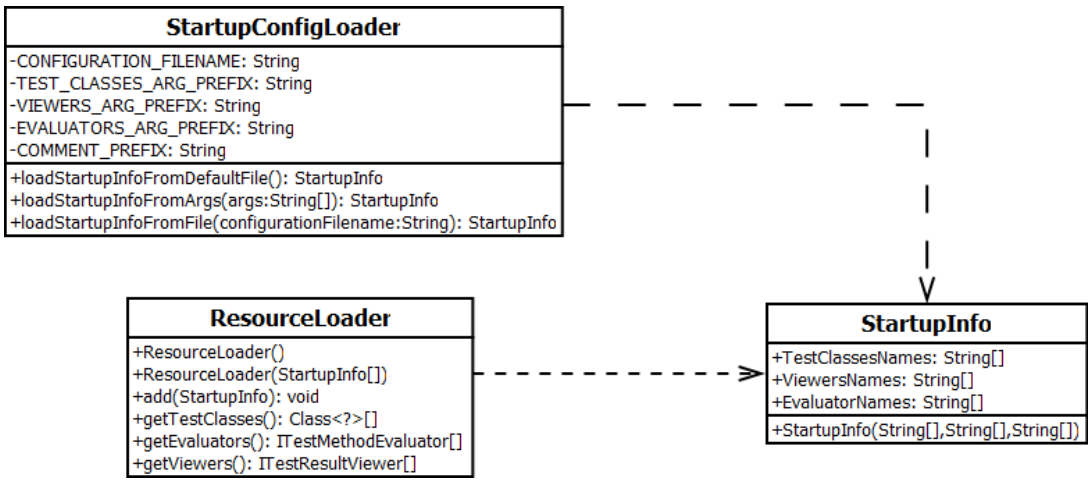

Figura 6 - Diagrama de classes que realizam o carregamento de rescursos

Interessa pois, definir quais os intervenientes e quais as suas responsabilidades neste processo.

StartupConfigLoader

Esta classe permite suportar os carregamentos das configurações para a realização dos testes. Utiliza os seguintes prefixos de argumento:

- '/c:' – Permite referir que o texto adjacente representa uma e só uma classe de teste a utilizar.
- '/e:' – Permite referir que o texto adjacente representa um e um só avaliador que será utilizado quando encontrada a respectiva anotação.
- '/v:' – Permite referir que o texto adjacente representa um e um só visualizador de resultados de avaliação a utilizar.

Estes prefixos podem ser utilizados mais que uma vez, quando necessário, desde que haja um símbolo de espaço que os separe.

Adicionalmente foi utilizado nesta Framework um conceito de junção de argumentos para quando estes são definidos recorrendo simultaneamente a entradas diferentes.

Como exemplo, caso o utilizador da Framework decida que irá criar um ficheiro de configuração para realizar determinadas configurações recorrentes, pode utilizar em simultâneo os argumentos em linha de comando para definir adicionalmente qualquer configuração pontual, uma vez que o carregamento do ficheiro de configuração não é negligenciado quando são utilizados argumentos em linha de comando.

StartupInfo

Esta classe armazena os resultados do carregamento da configuração, não tendo qualquer outra funcionalidade associada.

ResourceLoader

Esta classe realiza o carregamento das classes de teste, avaliadores e visualizadores e instanciação de avaliadores e visualizadores em função da informação reunida pelo `StartupConfigLoader`.

Foi incorporada a restrição de redefinir quais os recursos a carregar após a obtenção dos mesmos. Isto é, não faz sentido tentar redefinir quais os recursos que devem ser carregados após estes já terem sido carregados na Framework.

Esta classe utiliza *Lazy Initialization* nos métodos assessores dos recursos. Isto é, independentemente da quantidade de vezes que forem pedidos, por exemplo, as instancias de visualizadores de resultados, é apenas feita a instanciação das mesmas uma única vez.

Extensibilidade

E extensibilidade da Framework verifica-se fundamentalmente pelo suporte a novas implementações de `ITestMethodEvaluator` para criar novos avaliadores e pelo suporte a novas implementações de `ITestResultViewer`.

Após a Framework ser completamente implementada criaram-se um visualizador de resultados de teste adicional e um avaliador adicional. Para utilizar a Framework com estas novas funcionalidades basta indicar o nome das mesmas nos parâmetros de arranque da aplicação.

Adicionalmente a implementar a interface `ITestResultViewer`, um visualizador de resultados de teste necessita ainda de implementar um construtor sem parâmetros para que possa ser facilmente instanciado através de reflexão.

Existe ainda a possibilidade de modificar facilmente o modo como as configurações são carregadas ou como as classes são instanciadas antes de serem fornecidas ao `TestEngine`. Isto é permitido criando outro ponto de entrada do programa, carregando e instanciando os recursos a partir das fontes preferidas e da forma mais conveniente para depois delegar os mesmos ao motor de testes.

Anexos

Diagrama de classes