5. Aspectos de Projeto e Implementação

Este capítulo apresenta aspectos relacionados ao projeto e a implementação da solução construída, indicando também algumas decisões tomadas em sua construção.

5.1. Aspectos de qualidade

O projeto foi desenvolvido com a preocupação de se manter a boa qualidade dos construtos gerados. Para isto, foram aplicadas diversas boas práticas de construção, como as citadas a seguir.

5.1.1. Práticas de projeto

Algumas práticas foram adotadas no projeto da ferramenta, como:

- **Controle de versão**: Foi realizado o controle de versão através do auxílio da ferramenta Apache Subversion.[32] O repositório dos artefatos do projeto foi mantido na nuvem, utilizando o serviço gratuito do site Assembla;[33]
- **Gerência de configuração**: A gerência de configuração sobre o projeto e suas dependências foi realizada com uso da ferramenta Apache Maven.[34] Com isso, as dependências entre versões de pacotes e bibliotecas foram controladas durante todo o processo de desenvolvimento;
- **Desenvolvimento Dirigido por Testes, com uso de modelos**: Diversas partes do projeto tiveram seu desenvolvimento guiado por testes, mas também se optou por criar diagramas UML,[35] para manter uma visão geral sobre o modelo em desenvolvimento e identificar possíveis problemas mais facilmente;
- **Planejamento de liberações**: As funcionalidades escolhidas para o escopo do projeto foram ordenadas por prioridade e distribuídas para cada liberação. Cada liberação procurou construir as funcionalidades na ordem da prioridade atribuída.

[32] http://subversion.apache.org/
[33] http://www.assembla.com/
[34] http://maven.apache.org/
[35] http://www.uml.org

5.1.2.Heurísticas para modelagem

Para a construção do modelo do projeto, teve-se em mente sua testabilidade.[36] Esta prática serve como guia e culmina na aplicação de outras, como:

- **Injeção de dependências**: Dependências de objetos de classes que realizam serviços são tornadas explícitas e passadas, preferencialmente, em seu construtor. Isto torna clara a ordem de construção e amarração dos objetos e propicia a substituição destes objetos por outros. Preferencialmente usam-se interfaces ao invés de classes, para que a implementação destes serviços possa variar de acordo com o propósito desejado (ex.: testes). Este princípio é abordado por Fowler (2004).
- **Inversão de dependências**: Classes passam a depender de interfaces ou classes abstratas ao invés de classes concretas, diminuindo o acoplamento. Este princípio é abordado por Martin (2000).

Para manter alta a coesão das classes, foi observado, dentre outros, o **Princípio da Responsabilidade Única** (MARTIN, 2000), que reforça a existência de uma única responsabilidade para classes e métodos.

Para manter o acoplamento baixo, foi observado, dentre outros, o **Princípio do Conhecimento Mínimo** (MARTIN, 2000), que reforça a comunicação de objetos com somente aqueles pertencentes à sua abstração, como os criados internamente ou os pertencentes a outros métodos da própria classe.

Também se procurou identificar situações em que o uso de **Padrões de Projeto** (GAMMA et al., 1994) trouxesse benefícios ao modelo construído, principalmente para manter o acoplamento baixo ou tornar a solução mais flexível para extensão ou manutenção futura.

5.1.3.Boas práticas de codificação

Diversas práticas foram adotadas para manter a boa qualidade do código produzido, dentre elas: a refatoração (FOWLER & BECK, 1999); a escolha de bons nomes para classes e métodos, o não uso de "valores mágicos"; o tratamento adequado de exceções (evitando, por exemplo, ignorá-las); a programação defensiva; o uso de comentários (estilo Doxygen)[37] no código; etc.

[36] Capacidade com que um artefato pode ser testado.
[37] http://www.stack.nl/~dimitri/doxygen/

Aliado a elas, foram construídos testes unitários e de integração, como o exemplificado[38] na Listagem 9, a seguir:

```
...
@DataProvider(name="validValueOptions")
public Object[][] validValueOptions() {
      return new Object[][] {
            { ValidValueOption.MIN },
            { ValidValueOption.MAX },
            { ValidValueOption.ZERO },
            { ValidValueOption.MIDDLE },
            { ValidValueOption.RIGHT_AFTER_MIN },
            { ValidValueOption.RIGHT_BEFORE_MAX },
            { ValidValueOption.RANDOM_INSIDE_RANGE }
      };
}

@Test(groups={"slow"}, dataProvider="validValueOptions")
public void test_valid_values_for_QueryBasedVC_with_NOT_ONE_OF(
            ValidValueOption vvo) throws Exception {
      QueryConfig qc1 = new QueryConfig();
      qc1.setDatabaseConfig( newDatabaseConfig() );
      qc1.setName( "All use cases" );
      qc1.setCommand( "SELECT name FROM usecase" );

      QueryBasedVC vc1 = new QueryBasedVC();
      vc1.setQueryConfig( qc1 );
      vc1.setTargetColumn( "name" );
      vc1.setTargetColumnType( ValueType.STRING );

      BusinessRule br1 = new BusinessRule(
            BusinessRuleType.NOT_ONE_OF, vc1, "a message" );

      EditableElement ee1 = elementWithValueType(
            ValueType.STRING );
      ee1.addBusinessRule( br1 );

      Object value = gen.generateValidValue( vvo, ee1,
            otherElementValues );
      List< Object > values = gen.valuesFromQueryConfiguration(
            vc1, otherElementValues );

      assertThat( value ).isNotNull();
      assertThat( value ).isNotIn( values );

}

...
```

Listagem 9 - Exemplo de teste de integração realizado no projeto

[38] O teste exemplificado realiza o teste de integração entre uma classe que especifica uma regra de negócio baseada em uma consulta a um banco de dados, e um gerador de valores baseado em regras de negócio. Detalhes dos *frameworks* de teste utilizados são discutidos na seção 5.2.2.

Espera-se que o uso destas práticas aliadas às anteriores tenha produzido um software de boa qualidade.

5.2.Ferramentas e tecnologias

Nesta seção são apresentadas as principais ferramentas e tecnologias utilizadas pelo presente trabalho. Dentre os principais critérios utilizados para a escolha delas estão:

1. Possuir licença gratuita ou código aberto;
2. Serem, preferencialmente, amplamente conhecidas, de modo a facilitar seu uso futuro por terceiros (uma vez que há a intenção em tornar o projeto publicamente disponível).

5.2.1.Principais ferramentas

A Tabela 10 apresenta as principais ferramentas utilizadas para a construção do projeto.

Tabela 10 - Principais ferramentas utilizadas

Nome:	**Eclipse Helios**
Descrição:	Ambiente de desenvolvimento integrado
Site:	http://eclipse.org

Nome:	**Astah Community**
Descrição:	Ferramenta de criação de diagramas UML
Site:	http://www.astah.net/editions/community

Nome:	**Apache Maven**
Descrição:	Ferramenta de gerenciamento de dependências, build, relatório e documentação do projeto.
Site:	http://maven.apache.org/

Nome:	**TortoiseSVN**
Descrição:	*Front-end* gráfico para o Subversion
Site:	http://tortoisesvn.net/

A Tabela 11, a seguir, apresenta os principais *plug-ins* usados no Eclipse.

Tabela 11 - Principais plug-ins para Eclipse utilizados

Nome:	**m2e**
Descrição:	Integração com Apache Maven
Site:	http://eclipse.org/m2e/

Nome:	**Google Window Builder Pro**
Descrição:	Construtor de janelas (Swing, etc.)
Site:	https://developers.google.com/java-dev-tools/wbpro/?hl=pt-BR

Nome:	**Google CodePro AnalytiX**
Descrição:	Gerador de métricas e analisador de código
Site:	https://developers.google.com/java-dev-tools/codepro/doc/?hl=pt-BR

Nome:	**TestNG Plugin**
Descrição:	Integração com TestNG
Site:	http://testng.org/doc/eclipse.html

Nome:	**Eclox**
Descrição:	Integração com Doxygen
Site:	http://home.gna.org/eclox/

A Figura 4 mostra as principais dependências de bibliotecas externas (pacotes Java) do projeto.[39]

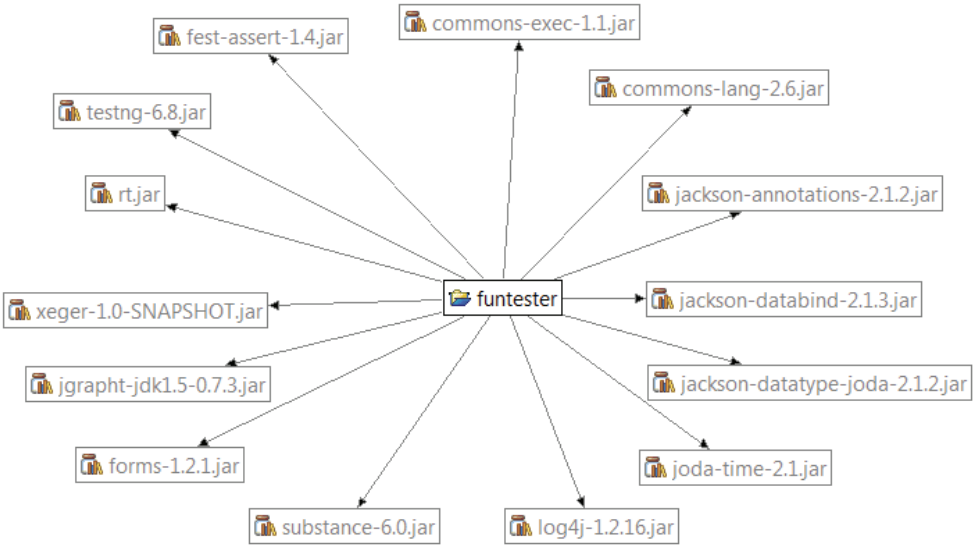

Figura 4 - Dependências de pacotes Java

O Apêndice A apresenta a listagem do arquivo de configuração *Maven* (`pom.xml`), que contém detalhes sobre os repositórios de código e a configuração das dependências externas do projeto.

[39] Figura gerada com o *plug-in* Google CodePro AnalytiX, anteriormente mencionado.

5.2.2.Frameworks de teste

Alguns *frameworks* de teste foram utilizados tanto na construção da **ferramenta** quanto no código de teste gerado pela **extensão**. A ferramenta faz uso do **TestNG** e do **FEST Assert**[40] em sua construção, enquanto o código de testes gerado pela extensão faz uso destas e do **FEST Swing**.

5.2.2.1.TestNG

O *framework* de testes unitários (FTU) TestNG possui algumas vantagens[41] sobre alguns FTUs populares, como o JUnit,[42] possibilitando, por exemplo, criar testes parametrizados, executar testes em múltiplas linhas de execução (*threads*) e criar grupos de teste com configurações personalizadas.

A Listagem 10, a seguir, apresenta um teste criado com estes recursos, que verifica os valores gerados pelo gerador de números inteiros, de acordo com cada regra apresentada na seção 4.6.1. Ao invés de se criar um oráculo (*assert*) para verificar cada combinação de valor e regra, foi possível criar várias combinações para um mesmo oráculo, parametrizando-as para o método de teste, através da especificação de um provedor de dados.

[40] https://github.com/alexruiz/fest-assert-2.x
[41] Listadas no site do *framework* (http://testng.org)
[42] http://junit.org/

```
...
/**
 * Tests the {@link LongValueGenerator}.
 *
 * @author Thiago Delgado Pinto
 *
 */
public class LongValueGeneratorTest {

...
@DataProvider(name="createValidValues")
public Object[][] createValidValues() {
      return new Object[][] {
      // ZERO (0 or Min as zero)
      { -10L, 10L, ValidValueOption.ZERO, 0L },
      { 10L, 20L, ValidValueOption.ZERO, 10L },
      { -20L, -10L, ValidValueOption.ZERO, -20L },
      // MIN
      { -10L, 10L, ValidValueOption.MIN, -10L },
      { 10L, 20L, ValidValueOption.MIN, 10L },
      { -20L, -10L, ValidValueOption.MIN, -20L },
      // MAX
      { -10L, 10L, ValidValueOption.MAX, 10L },
      { 10L, 20L, ValidValueOption.MAX, 20L },
      { -20L, -10L, ValidValueOption.MAX, -10L },
      // MIDDLE
      { -10L, 10L, ValidValueOption.MIDDLE, 0L },
      { 10L, 20L, ValidValueOption.MIDDLE, 15L },
      { -20L, -10L, ValidValueOption.MIDDLE, -15L },
      // RIGHT_AFTER_MIN (Min + 1)
      { -10L, 10L, ValidValueOption.RIGHT_AFTER_MIN, -9L },
      { 10L, 20L, ValidValueOption.RIGHT_AFTER_MIN, 11L },
      { -20L, -10L, ValidValueOption.RIGHT_AFTER_MIN, -19L },
      // RIGHT_BEFORE_MAX (Max - 1)
      { -10L, 10L, ValidValueOption.RIGHT_BEFORE_MAX, 9L },
      { 10L, 20L, ValidValueOption.RIGHT_BEFORE_MAX, 19L },
      { -20L, -10L, ValidValueOption.RIGHT_BEFORE_MAX, -11L }
      };
}

@Test(dataProvider="createValidValues")
public void testValidOptions(
 Long min, Long max, ValidValueOption option, Long expected) {
      ValueGenerator< Long > gen1 = new LongValueGenerator(
min, max );

      assertThat( gen1.validValue( option ) )
            .isEqualTo( expected );

}

...
```

Listagem 10 - Exemplo de teste parametrizável criado com TestNG

Outro uso interessante é a possibilidade de definir o número de vezes que o método de teste será invocado, o que foi utilizado no código da Listagem 11 (pertencente à mesma classe mostrada anteriormente), que gera valores aleatórios dentro da faixa definida.

```
...
@DataProvider(name="rangeForRandomValues")
public Object[][] rangeForRandomValues() {
      return new Object[][] {
            { -10L, 10L },
            { 10L, 20L },
            { -20L, -10L }
      };
}


@Test(dataProvider="rangeForRandomValues", invocationCount=5)
public void randomValidValueShouldBeInsideRange(Long min,
      Long max) {
      ValueGenerator< Long > gen1 = new LongValueGenerator(
            min, max );
      Long value = gen1.validValue(
            ValidValueOption.RANDOM_INSIDE_RANGE );
      assertThat( value )
            .isGreaterThanOrEqualTo( min )
            .isLessThanOrEqualTo( max );

}
...
```

Listagem 11 - Exemplo de teste com número de execuções parametrizável

O método de teste foi parametrizado para ser invocado cinco vezes, o que permite, ao longo do tempo, que se teste várias possíveis combinações de valores, uma vez que eles são gerados aleatoriamente, a cada execução.

5.2.2.2.FEST Assert

O *framework* FEST Assert, que possui integração com TestNG, possibilita a construção de oráculos mais legíveis, com uma sintaxe mais **fluida** (*fluent interface*) (FOWLER & EVANS, 2005), o que facilita sua compreensão. Ele também possui um conjunto de assertivas que atuam sobre várias bibliotecas padrão do Java, além de extensões para bibliotecas externas, permitindo também sua personalização. Alguns exemplos (bastante simples) de seu uso podem ser vistos nas duas listagens de código anteriores (construção `assertThat`).

Este *framework* não tem relação de dependência com FEST Swing, sendo apenas um projeto construído pelos mesmos autores.

5.2.2.3.FEST Swing

O *framework* FEST Swing possui uma sintaxe simples e compacta, que permite obter uma boa produtividade para a construção de testes funcionais para Swing. Ele também fornece informações relevantes quando um teste falha, o que facilita encontrar a causa de problemas mais rapidamente.

Um exemplo simples do uso do *framework* pode ser observado pelo trecho de código da Listagem 12, que entra com o nome de usuário "`root`" e a senha "`p4ssw0rd`" nas caixas de texto nomeadas, respectivamente, "`user`" e "`password`", presentes numa janela (`dialog`), clica num botão de nome "`ok`" e espera que seja exibida uma mensagem "`Please enter your .*`", que aceita expressões regulares.

```
dialog.textBox( "user" ).enterText( "root" );
dialog.textBox( "password" ).enterText( "p4ssw0rd" );
dialog.button( "ok" ).click();
// regular expression matching
dialog.optionPane().requireMessage( "Please enter your .*" );
```

Listagem 12 - Exemplo de código que usa o framework FEST Swing

5.3.Arquitetura

Nesta seção são expostos detalhes sobre a arquitetura da solução construída.

5.3.1.Visão geral

A Figura 5, a seguir, apresenta uma visão geral sobre os componentes envolvidos na solução construída. A ferramenta, chamada de **`FunTester`** (acrônimo para *Functional Tester*), e uma de suas possíveis extensões, chamada de **`FunTester Extension`**, são os componentes que executam o processo mostrado na seção 4.1. Através do **`FunTester`**, o usuário cria um arquivo com o projeto do software, representado por **`MySoftware.fun`**. Dele podem ser gerados os casos de teste semânticos valorados num arquivo, representado por

MySoftware.fst. O **FunTester** então usa a **FunTester Extension** para a geração do código-fonte dos testes, representado pelos arquivos **TestCase1**, **TestCase2** e **TestCaseN**. Estes arquivos geralmente farão uso de dois *frameworks* de teste, um para testes de interfaces gráficas, representado por **GUI Testing Framework**, e outro para a criação de testes unitários, representado por **Unit Testing Framework**. A extensão, então, executa os testes criados a partir da execução do **Unit Testing Framework**. Este, por sua vez, gera um arquivo com os resultados da execução (usualmente no formato XML), representado por **test-results.xml**. O referido arquivo é então lido pela extensão e transformado noutro arquivo com formato independente de *framework*, representado por **MySoftware.fer**. Por fim, a ferramenta o lê, para conhecer e analisar os resultados da execução.

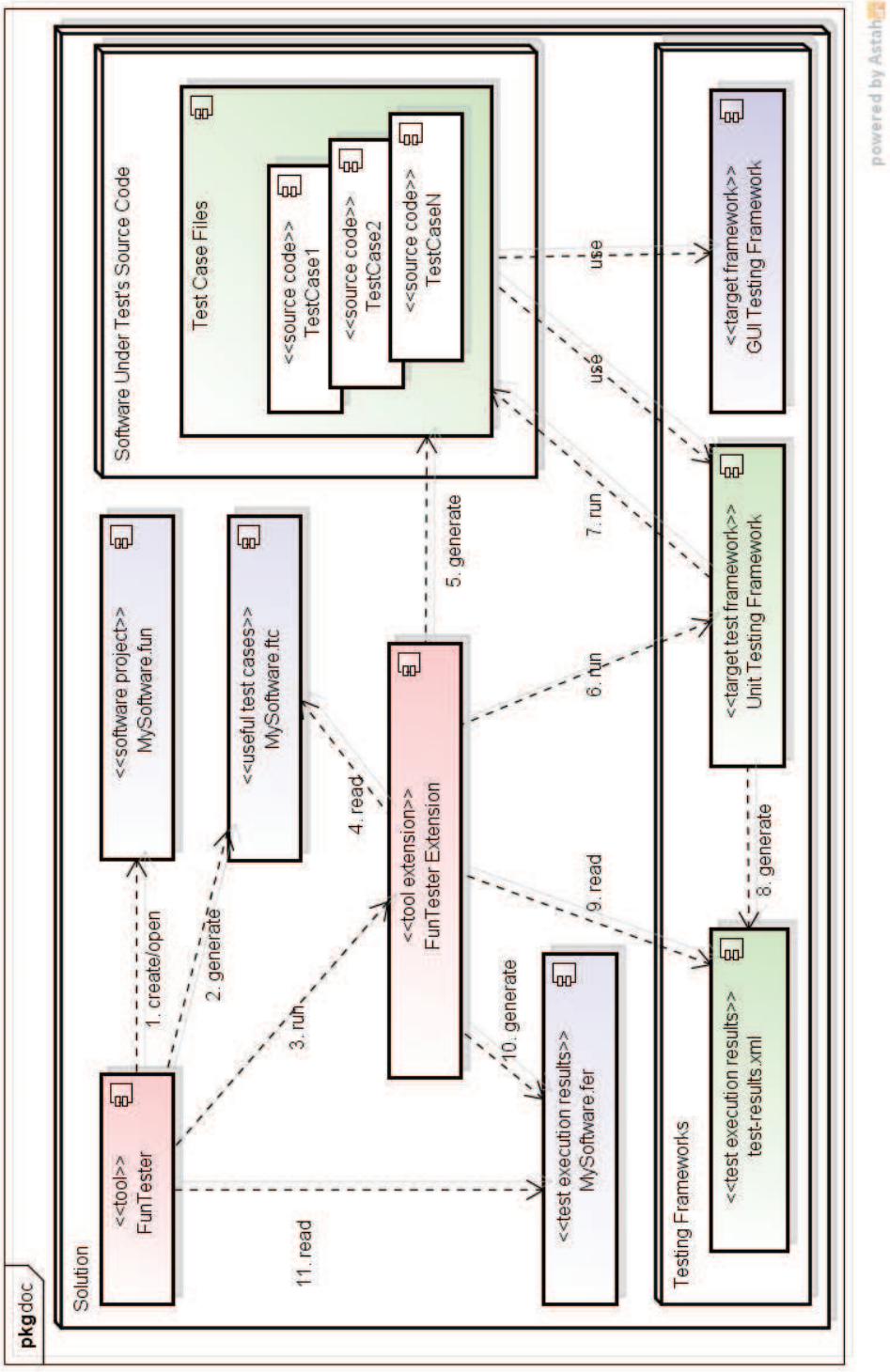

Figura 5 - Componentes envolvidos na solução

5.3.2.Arquitetura da ferramenta

A seguir serão apresentados detalhes sobre a arquitetura da ferramenta construída (FunTester).

5.3.2.1.Principais tipos de dados do modelo conceitual

A Tabela 12 apresenta os principais tipos de dados do modelo conceitual, no domínio da ferramenta, para servir como referência para a compreensão dos diagramas mostrados em seguida.

Tabela 12 - Principais tipos de dados de domínio do modelo conceitual da ferramenta

NOME	SIGNIFICADO/RESPONSABILIDADE
Action	Uma ação disparada em um passo de um fluxo.
ActionKind	O tipo da ação disparada em um passo de um fluxo.
ActionNickname	Um "apelido" para uma ação, para que esta possa ser descrita em outro idioma ou de outra forma.
ActionStep	Um passo que executa uma ação.
Actor	Um ator que interage com o sistema.
AlternateFlow	Um fluxo alternativo.
BasicFlow	O fluxo principal de um caso de uso.
BusinessRule	A regra de negócio associada ao elemento editável.
BusinessRuleType	O tipo de regra de negócio.
CancelatorFlow	Um fluxo cancelador.
Complexity	A complexidade de um fluxo
ConditionState	Um estado, podendo ser uma pré-condição ou uma pós-condição.
ConditionStateKind	O tipo de uma condição.
DatabaseConfig	A configuração da conexão com um banco de dados.
DatabaseScript	Um script de banco de dados a ser executado.

DocStep	Um passo que não realiza uma ação, servindo apenas como meio de documentar um processamento ou interação.
EditableElement	Um elemento que pode ter seu valor modificado pelo ator.
Element	Um elemento com o qual uma ação pode interagir.
ElementBasedVC	A configuração do valor de um elemento editável, baseada no valor de outro elemento.
ElementKind	A classificação do tipo de elemento.
ElementType	O tipo do elemento usado num passo, sendo geralmente o tipo de *widget* de interface.
Flow	Um fluxo de um caso de uso.
FlowType	O tipo de um fluxo.
Frequency	A frequência de um fluxo
InteractableElement	Um elemento de um passo de um fluxo, com o qual o ator pode interagir.
Locale	Uma localidade onde se encontra o usuário.
MultiVC	A configuração do valor de um elemento editável que pode receber mais de um valor.
ParameterConfig	A configuração dos parâmetros de uma consulta envolvida na configuração e um elemento editável.
Postcondition	Uma pós-condição de um caso de uso.
Precondition	Uma pré-condição de um caso de uso.
Priority	A prioridade de um fluxo.
QueryBasedVC	A configuração do valor de um elemento editável, baseada em uma consulta a um banco de dados.
QueryConfig	A configuração de uma consulta a um banco de dados.
RegEx	A configuração de uma expressão regular.
RegExBasedVC	A configuração do formato do valor de um elemento editável, baseada em uma expressão regular.
ReturnableFlow	Um fluxo retornável.
SemanticActionStep	Um passo semântico que executa uma ação.
SemanticElement	Elemento sobre o qual a ação é executada.
SemanticOracleStep	Um passo semântico que é um oráculo.

SemanticStep	Um passo semântico.
SemanticTest	Um teste semântico.
SemanticTestCase	Um caso de teste semântico.
SemanticTestMethod	Um método de teste semântico.
SemanticTestSuite	Uma suíte de testes semânticos.
SingleVC	A configuração do valor de um elemento editável que pode receber um único valor.
Software	O software que está sendo documentado.
Step	Um passo de um caso de uso.
TerminableFlow	Um fluxo terminável.
TerminatorFlow	Um fluxo terminador.
TestCaseExecution	O resultado da execução de um caso de teste.
TestExecution	Um resultado da execução dos testes.
TestExecutionReport	O resultado geral da execução dos testes.
TestExecutionStatus	O status da execução de um teste.
TestMethodExecution	O resultado da execução de um método de teste.
TestSuiteExecution	O resultado da execução de uma suíte de testes.
Trigger	O disparador de um passo de um fluxo.
UseCase	Um caso de uso do software.
ValueConfiguration	A configuração do valor de um elemento editável, de acordo com a regra de negócio.
ValueType	O tipo de valor que o elemento editável pode receber.

A Figura 6, a seguir, apresenta as classes que representam um software e suas partes. Um software (`Software`) pode ter vários casos de uso (`UseCase`) e pode interagir com vários atores (`Actor`). Ele também poder manter expressões regulares (`RegEx`), configurações de banco de dados (`DatabaseConfig`) e configurações de consultas (`QueryConfig`), que serão usadas na definição das regras de negócio de elementos de interface de um caso de uso, detalhadas adiante (Figura 11). Um caso de uso (`UseCase`) pode ter, como pré-condições, estados (`ConditionState`) necessários à sua execução (que, em geral, são pós-condições

geradas por outros casos de uso). Um caso de uso também possui fluxos (`Flow`) e mantém elementos de interface com o usuário com os quais um ator interage (`InteractableElement`) em um ou mais fluxos.

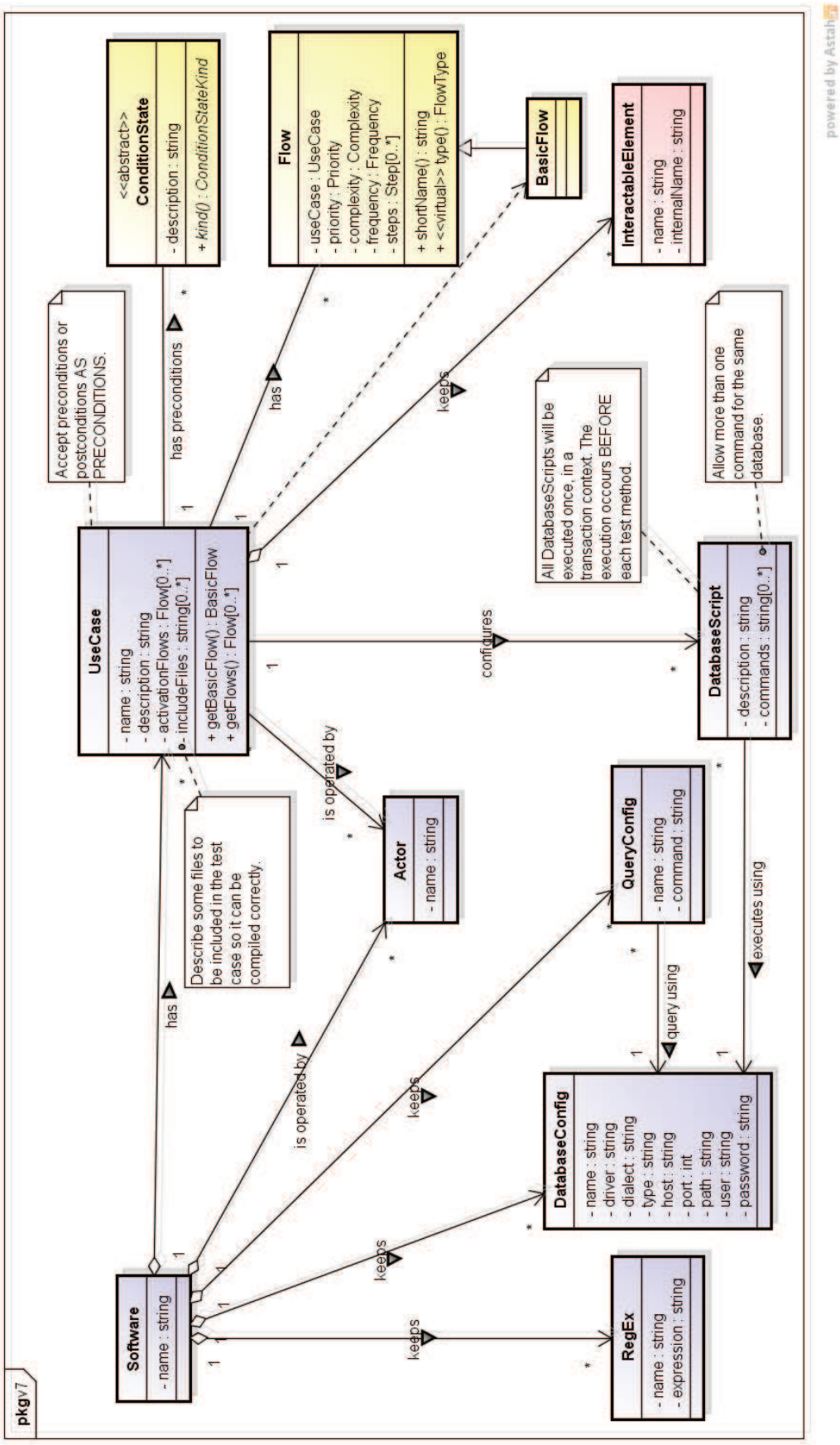

Figura 6 - Classes no contexto de um software

A Figura 7, a seguir, mostra a estrutura dos fluxos de um caso de uso. Um caso de uso (`UseCase`) pode ter vários fluxos (`Flow`), sendo que seu primeiro fluxo deve ser um fluxo básico (`BasicFlow`), que é um fluxo terminável (`TerminableFlow`), ou seja, que termina e gera uma pós-condição. Cada fluxo adicional deve ser um fluxo alternativo (`AlternateFlow`), podendo ser retornável (`ReturnableFlow`) – isto é, um fluxo alternativo que retorna para seu fluxo iniciador –, terminador (`TerminatorFlow`) – que permite terminar o caso de uso gerando uma pós-condição – ou cancelador (`CancelatorFlow`) – que permite terminar o caso de uso sem gerar uma pós-condição. Um fluxo (`Flow`) gera uma ou mais pós-condições (`Postcondition`), que poderão ser usadas como pré-condições (`Precondition`) em outros casos de uso. Um fluxo também possui uma prioridade (`Priority`), uma complexidade (`Complexity`), uma frequência de uso (`Frequency`) e é composto por vários passos (`Step`).

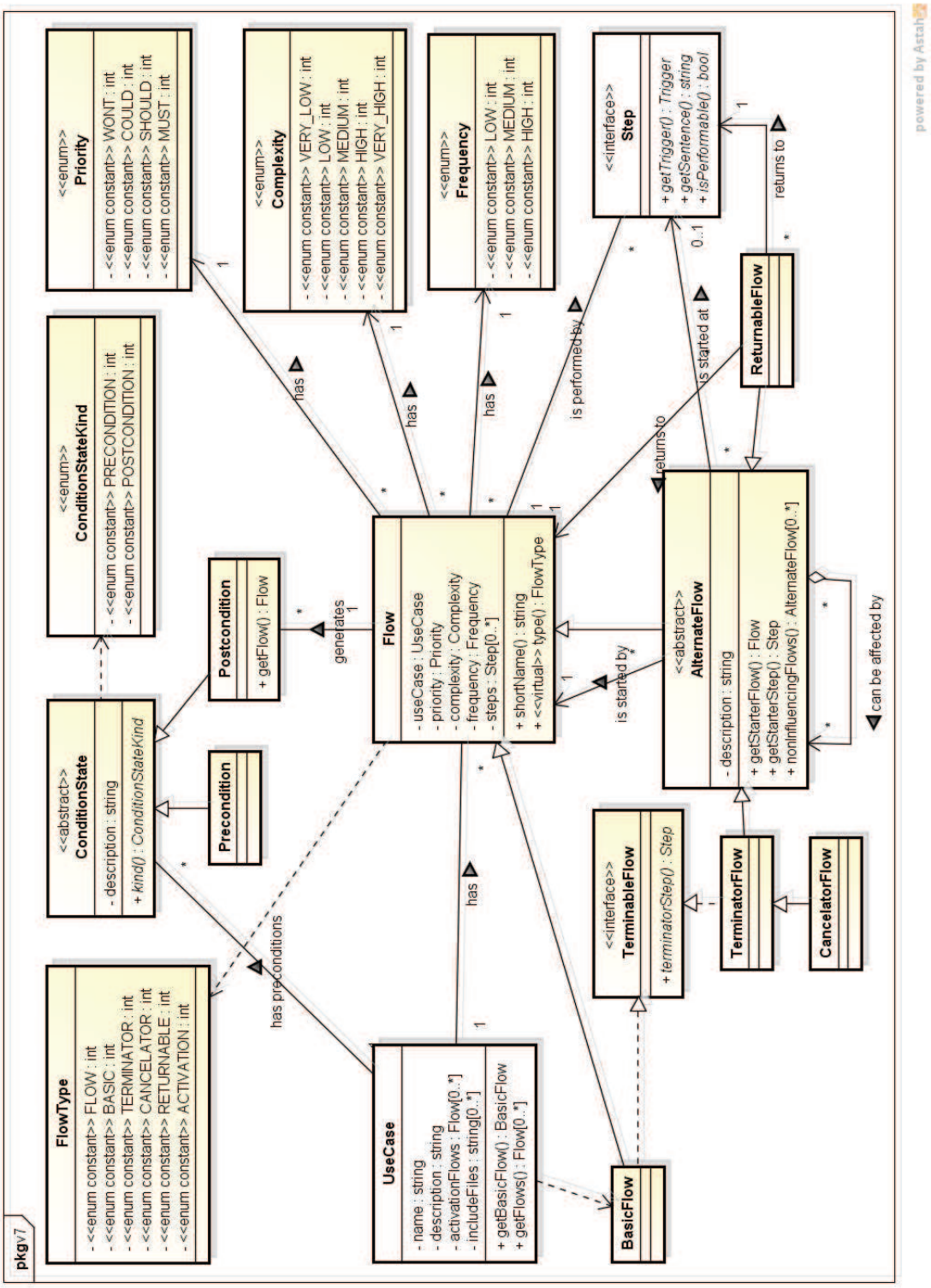

Figura 7 - Classes no contexto dos fluxos de um caso de uso

A Figura 8, a seguir, apresenta as classes relacionadas aos passos de um fluxo. Um passo (`Step`) é executado por um disparador (`Trigger`), que pode ser um ator ou o sistema. Um passo usado para documentação (`DocStep`) possibilita que

passos que não sejam relevantes para os testes, mas dão informação interessante para o entendimento do funcionamento do caso de uso, possam ser representados. A maioria dos passos, entretanto, é representada por um passo de ação (`ActionStep`), que permite descrever a ação executada (`ActionNickname`) e sob quais elementos (`Element`) ela é executada. Esta ação está relacionada a uma localidade (`Locale`) – que define informações do idioma utilizado – e uma ação real (`Action`), que será usada em substituição à ação do passo (`ActionNickname`) quando o passo precisar ser transformado em código de teste.

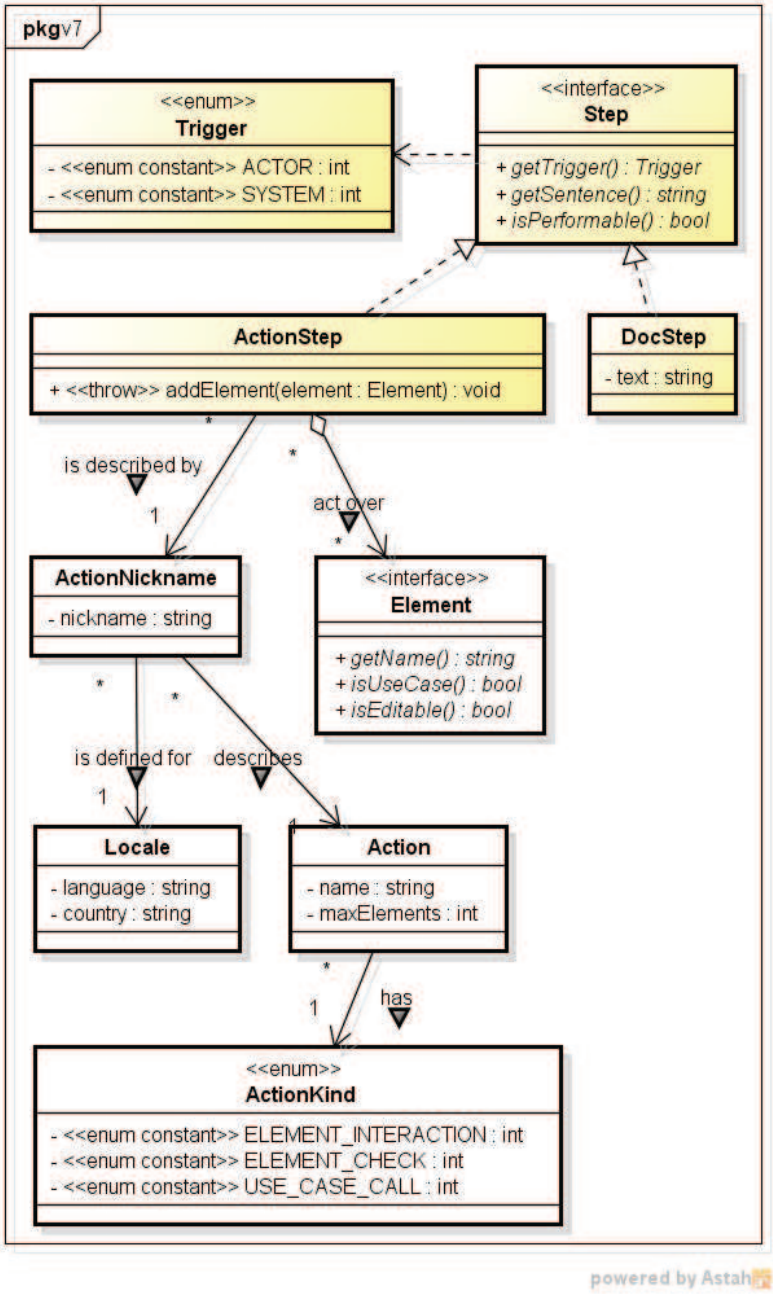

Figura 8 - Classes no contexto de um passo

A Figura 9, a seguir, apresenta as classes relacionadas com os elementos mantidos por um caso de uso. Um caso de uso (`UseCase`) mantém elementos (`Element`) que são referenciados durante a descrição dos passos de um fluxo. Esses elementos são geralmente elementos de interface com o usuário com o qual um ator pode interagir (`InteractableElement`) e possui um tipo (`ElementType`), que pode ser, por exemplo, uma janela, um botão ou uma caixa de texto. Cada tipo possui uma classificação (`ElementKind`), que indica se o elemento é um *widget*, uma URL, um comando, uma combinação de teclas ou um tempo de espera. Se um ator puder editar o valor de um elemento, esse elemento é considerado um elemento editável (`EditableElement`). O próprio caso de uso é considerado um elemento, podendo ser usado num passo de um fluxo, para indicar que ele deve ser executado (chamado) nesse passo.

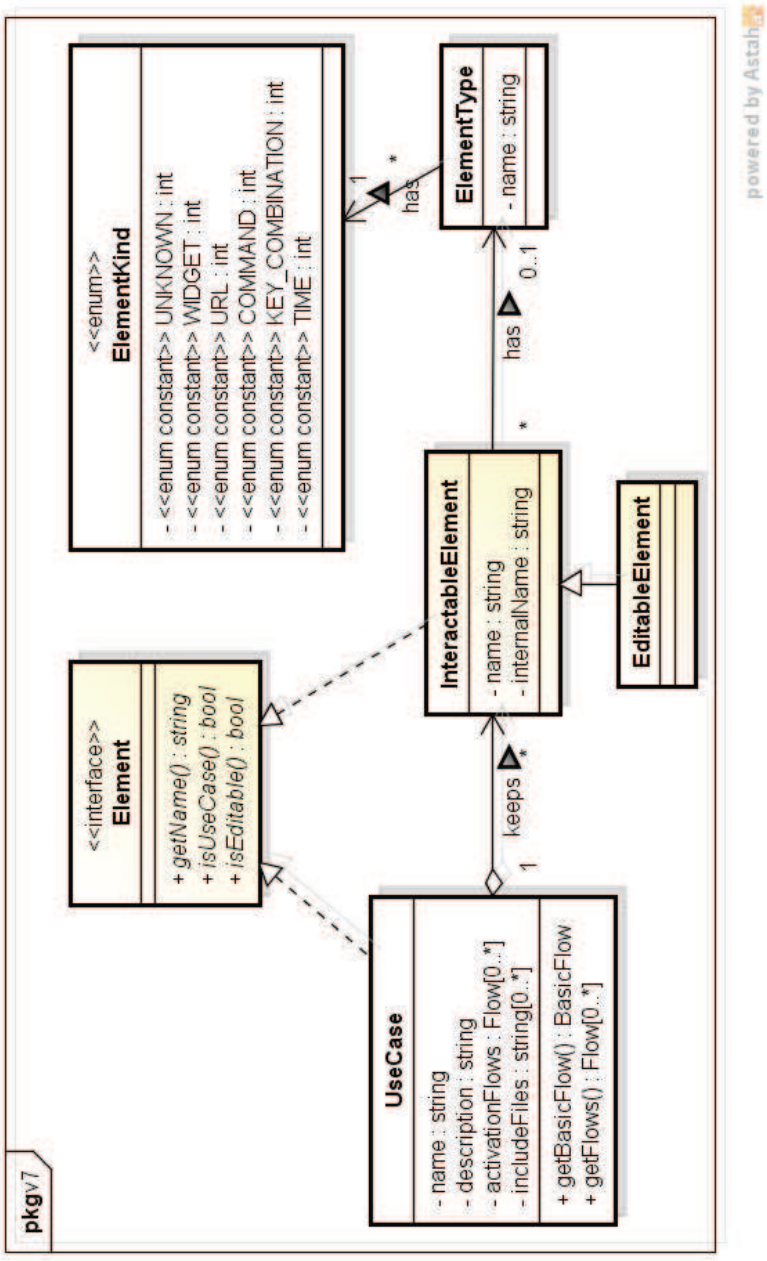

Figura 9 - Classes no contexto de elementos de um caso de uso

A Figura 10, a seguir, exibe as classes relacionadas a um perfil. Um perfil (`Profile`) representa uma configuração de vocabulário que pode ser usada por uma ou mais *extensões da ferramenta*, para transformar palavras reservadas em código-fonte. Estas palavras reservadas se referem a ações (`Action`) desempenhadas por usuários – como "clicar", "digitar", "escolher", etc. – e tipos de elemento de interface com o usuário (`ElementType`) – como "botão", "caixa de texto", "caixa de seleção", etc. Assim, um perfil "*Generic GUI*" poderia ser composto por palavras que são usadas por várias *extensões da ferramenta*, como a

que transforma essas palavras para o *framework* FEST, ou outra que poderia transformar para o *framework* Selenium, por exemplo. Para permitir usar as ações no idioma do usuário ou com variações na flexão das palavras, os perfis (`Profile`) mantêm apelidos (`ActionNickname`) para as ações. Cada apelido necessita informar a localização (`Locale`) usada, que contém informações do idioma adotado.

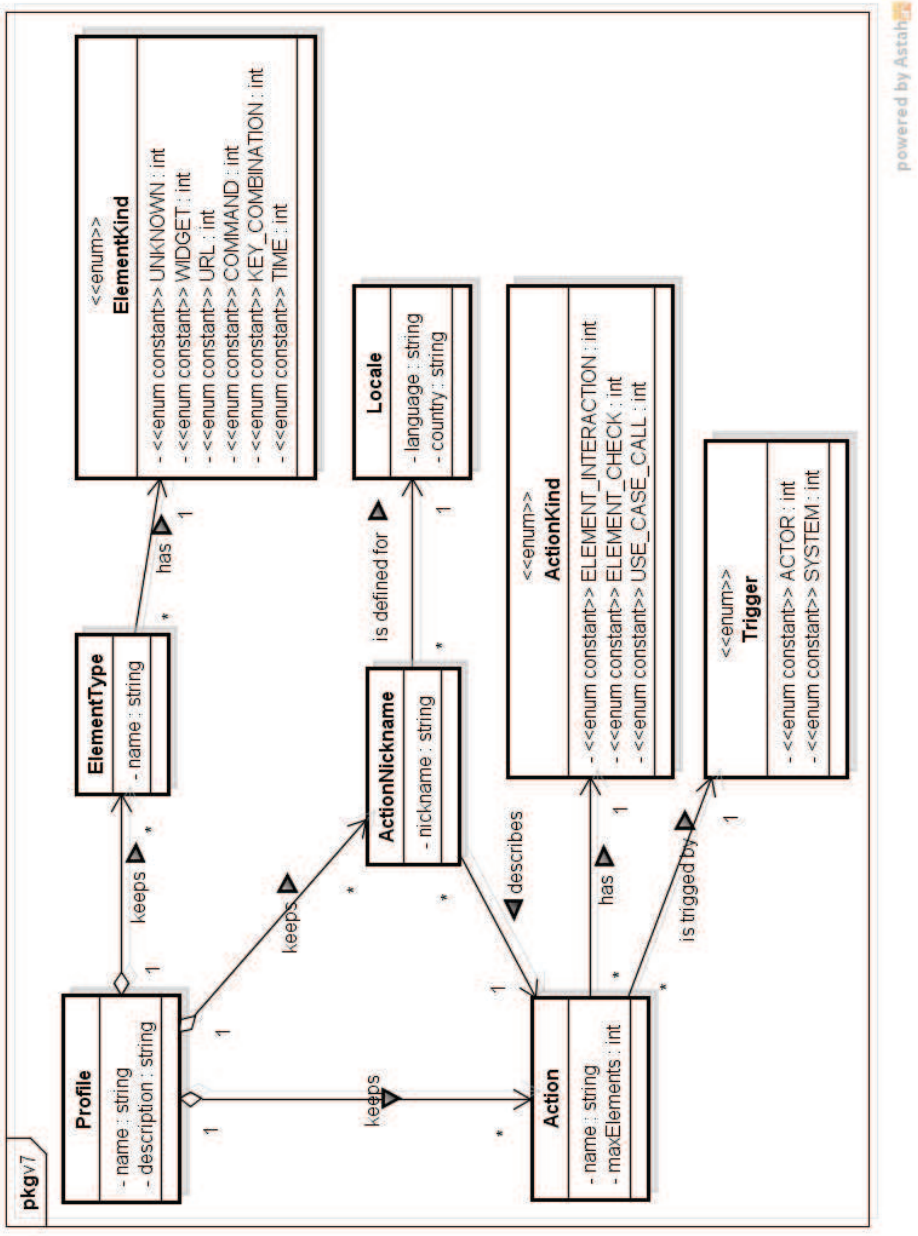

Figura 10 - Classes relacionadas a um Perfil

A Figura 11, a seguir, apresenta as classes relacionadas às regras de negócio de um elemento. Um elemento editável (`EditableElement`), isto é, que pode ter seu

valor editado pelo usuário, possui várias regras de negócio (`BusinessRule`). Cada regra de negócio possui um tipo (`BusinessRuleType`), uma mensagem (esperada que o sistema exiba quando o valor informado pelo usuário não esteja de acordo com a regra) e uma configuração de valor (`ValueConfiguration`). Uma configuração de valor é definida de acordo com o tipo da regra de negócio e especifica restrições sobre o valor do elemento editável. Na versão atual, o valor pode ser: manualmente definido e único (`SingleVC`); manualmente definido e restrito a uma lista de valores (`MultiVC`); gerado aleatoriamente, respeitando uma expressão regular (`RegExBasedVC`); baseado no valor de outro elemento editável (`ElementBasedVC`); ou ser consultado de um banco de dados (`QueryBasedVC`). Nesse último caso, a consulta pode ser parametrizada usando outra configuração de valor (`ValueConfiguration`), o que fornece flexibilidade para construir regras complexas (ex.: uma consulta que recebe como parâmetro o valor de outro elemento, que também possui sua configuração de valor).

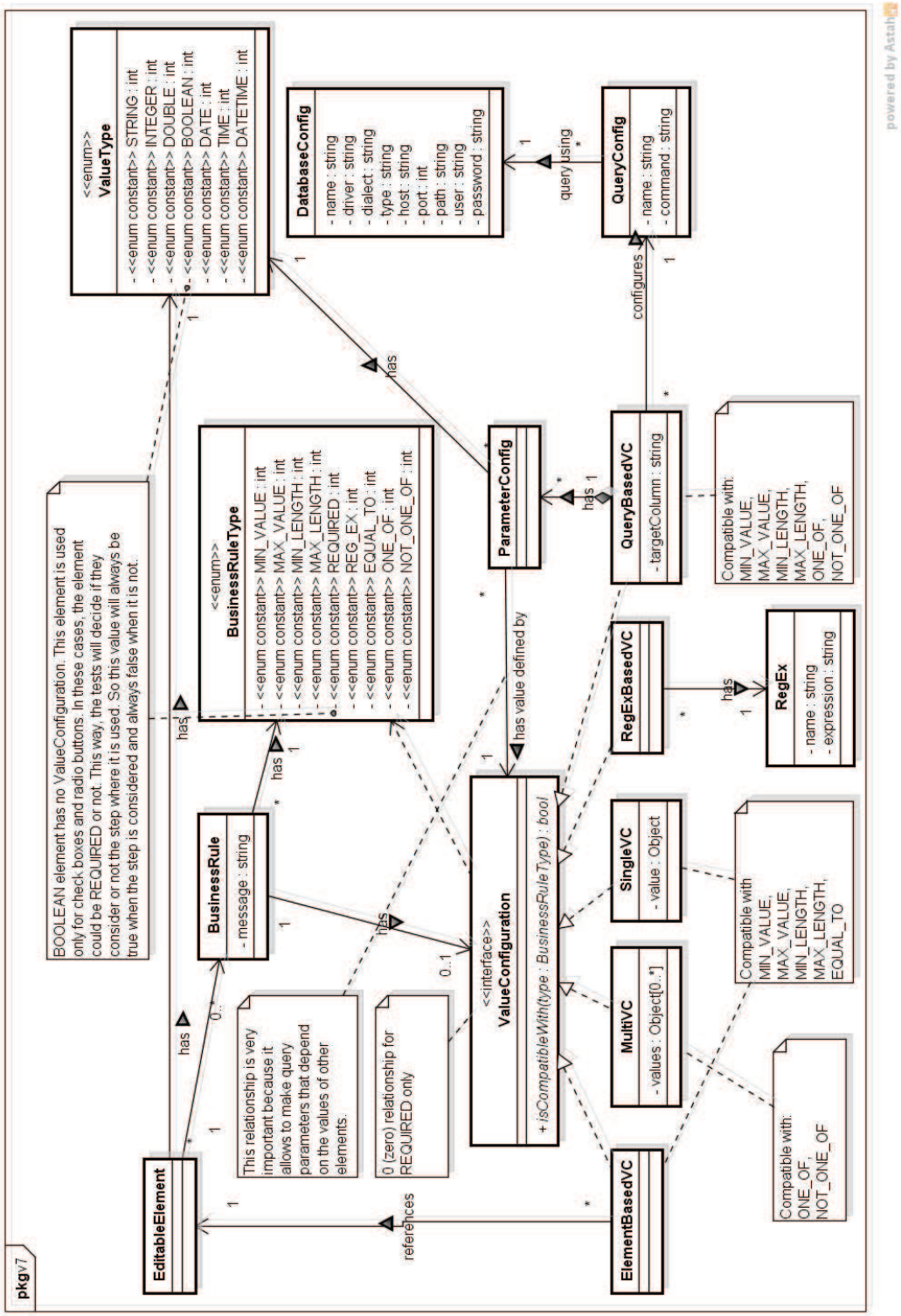

Figura 11 - Classes no contexto de uma Regra de Negócio

A Figura 12, a seguir, mostra as classes que fazem parte do contexto dos casos de teste semânticos valorados e com oráculo (CTSVO). Essas classes são resultado de uma fusão da especificação de casos de uso com a arquitetura básica

de *frameworks* de teste similares ao jUnit,[43] que possuem suítes de teste e casos de teste. Uma suíte de testes (`SemanticTestSuite`) é um conjunto de casos de teste (`SemanticTestCase`). Cada caso de teste testa um cenário de um caso de uso e possui um ou mais métodos de teste (`SemanticTestMethod`). Cada método de teste possui passos (`SemanticStep`), que equivalem aos comandos executados nos métodos de teste. Cada passo pode ser uma ação (`SemanticActionStep`) ou um oráculo (`SemanticOracleStep`). Quando for uma ação, ele pode atuar sobre vários elementos de interface com o usuário (`SemanticElement`).

Cada classe desse modelo mantém referências para as informações do modelo de casos de uso, possibilitando rastreá-las. Por exemplo, o passo semântico (`SemanticStep`) possui a identificação do caso de uso, do fluxo e do passo do fluxo correspondentes. Para permitir a transformação em código-fonte, o passo semântico possui o nome da ação esperada pela *extensão da ferramenta*, para que ela gere o respectivo comando, segundo o *framework* para o qual ela foi desenvolvida. Ela também deve usar a identificação do passo semântico (`id`) para instrumentar o código-fonte gerado, gerando uma correspondência entre cada linha de código e passo semântico. Assim, quando um teste obtiver falha na execução, é possível saber qual o passo semântico correspondente e dele obter as informações necessárias à identificação da especificação correspondente.

[43] http://junit.org

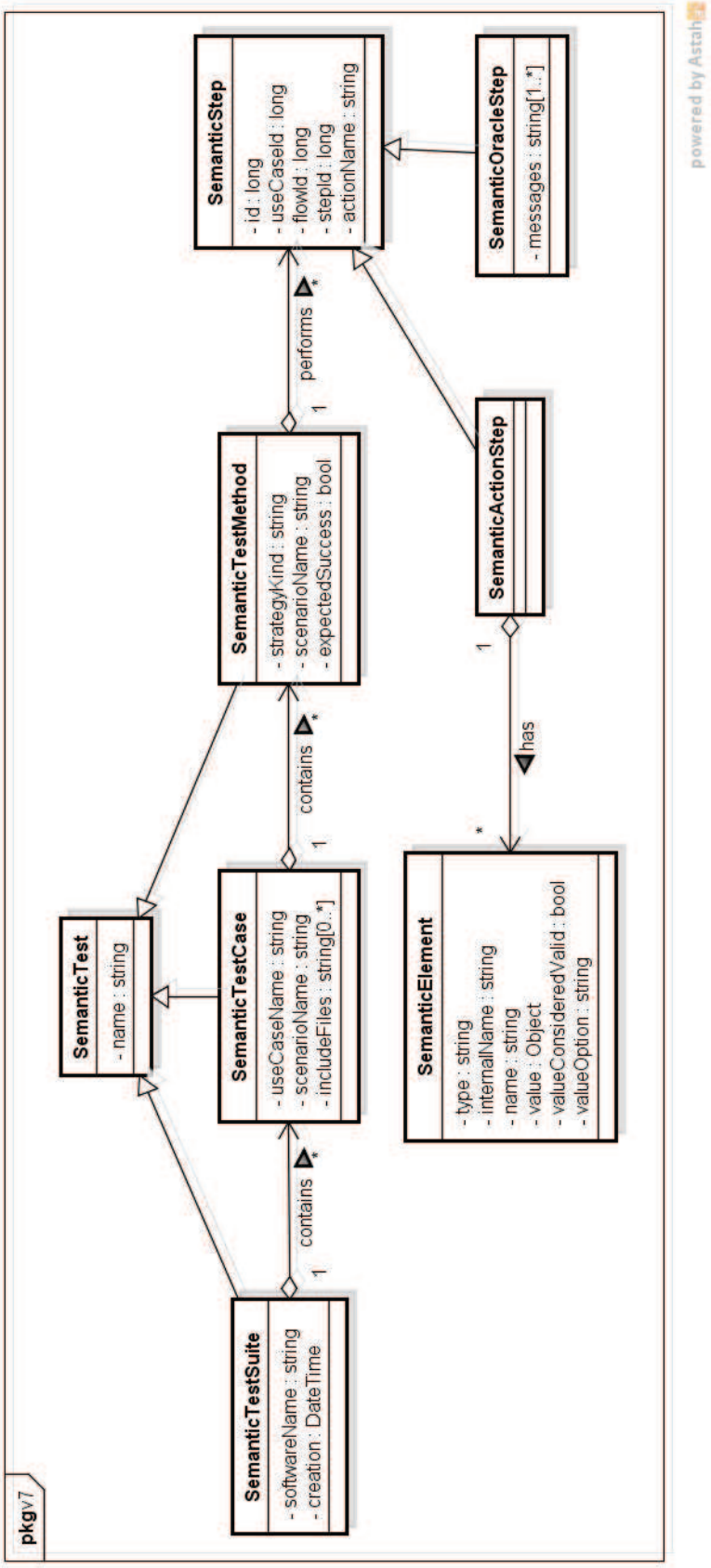

Figura 12 - Classes no contexto dos CTSVO

A Figura 13, a seguir, apresentada as classes no contexto do resultado da execução dos testes, realizado por uma *extensão da ferramenta*. Estas classes mapeiam a estrutura dos arquivos que relatam os resultados da execução dos testes em boa parte dos *frameworks* derivados do JUnit – usualmente referenciados como *xUnit* (MESZAROS, 2007), como CppUnit,[44] PhpUnit,[45] NUnit[46], etc. – e de algumas variações, como o TestNG.

[44] http://sourceforge.net/projects/cppunit/
[45] https://github.com/sebastianbergmann/phpunit/
[46] http://www.nunit.org/

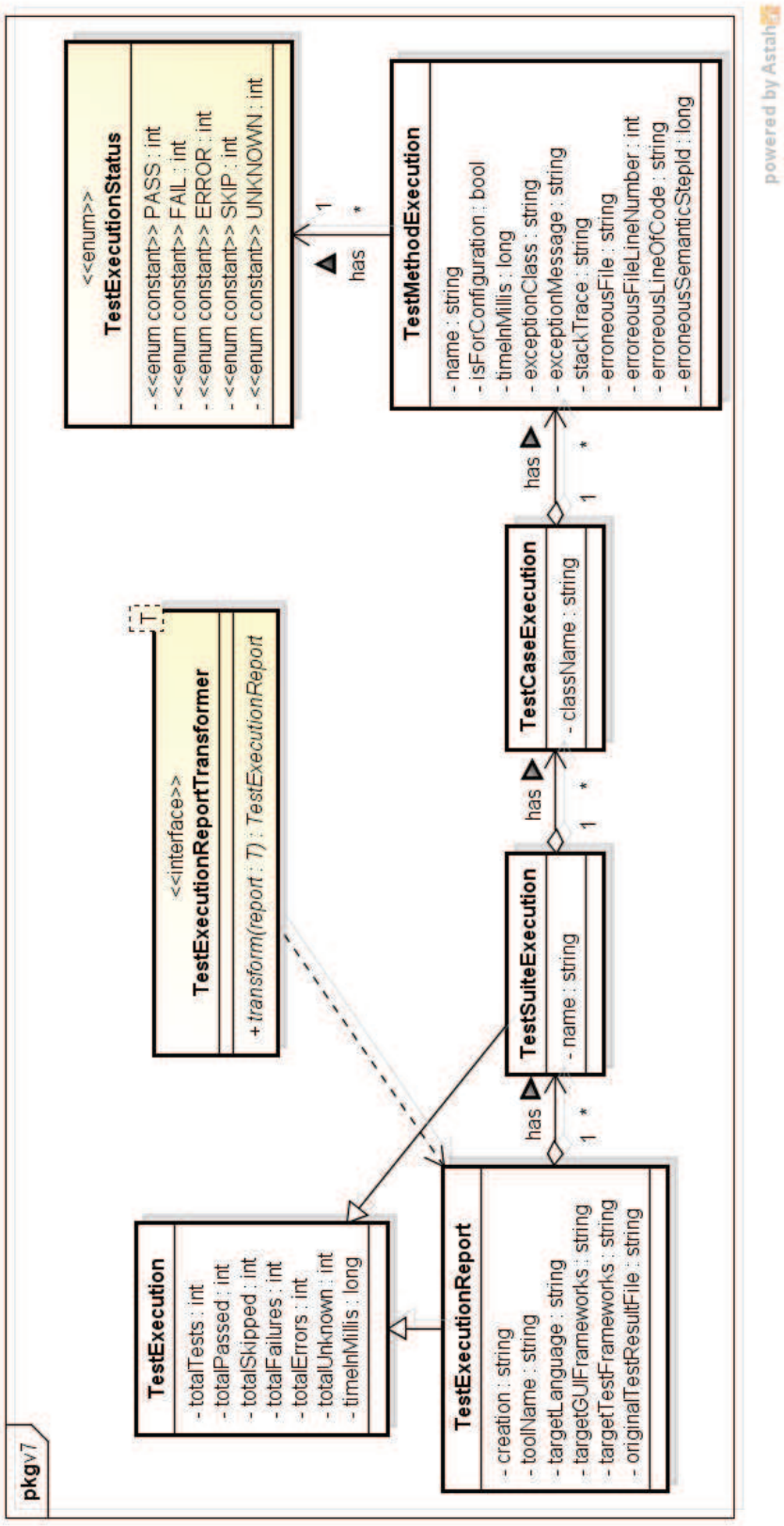

Figura 13 - Classes no contexto do resultado da execução dos testes

5.3.2.2.Principais tipos de dados relacionados ao funcionamento da solução

A Tabela 13, abaixo, apresenta os principais tipos de dados utilizados no funcionamento da ferramenta (adicionalmente aos tipos apresentados anteriormente).

Tabela 13 – Principais tipos de dados relacionados ao funcionamento da solução

NOME	SIGNIFICADO/RESPONSABILIDADE
`AllButOneTMGS`	Estratégia de geração de métodos de teste semânticos que usam todos os elementos, exceto um.
`AllButOneWithValidValuesTMGS`	Estratégia de geração de métodos de teste semânticos que usam todos os elementos com valores válidos, exceto um.
`AllRequiredButOneTMGS`	Estratégia de geração de métodos de teste semânticos que usam todos os elementos obrigatórios, exceto um.
`AllWithMaximumValuesTMGS`	Estratégia de geração de métodos de teste semânticos onde todos os elementos recebem o valor máximo definido para cada.
`AllWithMiddleValuesTMGS`	Estratégia de geração de métodos de teste semânticos onde todos os elementos recebem um valor intermediário entre o mínimo e o máximo definidos para cada.
`AllWithMinimumValuesTMGS`	Estratégia de geração de métodos de teste semânticos onde todos os elementos recebem o valor mínimo definido para cada.
`AllWithRandomValuesTMGS`	Estratégia de geração de métodos de teste semânticos onde todos os elementos recebem um valor aleatório válido.
`AllWithValidValuesButOneWithRandomValueAfterTheMaximumTMGS`	Estratégia de geração de métodos de teste semânticos que usam todos os elementos com valores válidos, exceto um, que receberá um valor aleatório maior que o valor máximo definido.

AllWithValidValuesButOneWithRandomValueBeforeTheMinimumTMGS	Estratégia de geração de métodos de teste semânticos que usam todos os elementos com valores válidos, exceto um, que receberá um valor aleatório menor que o valor mínimo definido.
AllWithValidValuesButOneWithValueRightAfterTheMaximumTMGS	Estratégia de geração de métodos de teste semânticos que usam todos os elementos com valores válidos, exceto um, que receberá um valor imediatamente posterior ao valor máximo definido.
AllWithValidValuesButOneWithValueRightBeforeTheMinimumTMGS	Estratégia de geração de métodos de teste semânticos que usam todos os elementos com valores válidos, exceto um, que receberá um valor imediatamente anterior ao valor mínimo definido.
AllWithValidValuesTMGS	Estratégia de geração de métodos de teste semânticos onde todos os elementos recebem valores válidos.
AllWithValuesRightAfterTheMinimumTMGS	Estratégia de geração de métodos de teste semânticos onde todos os elementos recebem um valor imediatamente posterior (maior) ao mínimo definido para cada.
AllWithValuesRightBeforeTheMaximumTMGS	Estratégia de geração de métodos de teste semânticos onde todos os elementos recebem um valor imediatamente anterior (menor) ao máximo definido para cada.
AllWithZeroValuesTMGS	Estratégia de geração de métodos de teste semânticos onde todos os elementos recebem zero ou o valor mínimo quando zero não está disponível.
AtomicIdGenerator	Gerador de identificação que suporta múltiplas linhas de execução (threads).
DateTimeGenerator	Gerador de valores usado para datas e horas.
DateValueGenerator	Gerador de valores usado para datas.
Default SemanticTestSuiteGenerator	Implementação padrão do gerador de suíte de testes semânticos valorados e com oráculo.

DefaultAbstractValueGenerator	Implementação comum a vários tipos de dados de um gerador de valores.
DefaultTMGS	Estratégia padrão para geração de métodos de teste semânticos.
DoubleValueGenerator	Gerador de valores usado para número de ponto flutuante.
ElementValueGenerator	Gerador de valores para um elemento editável
IdGenerator	Gerador de identificação usado nos passos semânticos.
InvalidValueOption	Opções para geração de valor inválido.
JustTheRequiredTMGS	Estratégia de geração de métodos de teste semânticos onde somente os elementos obrigatórios recebem um valor válido e os demais não recebem nenhum valor.
LongValueGenerator	Gerador de valores usado para números inteiros.
OracleMessageChooser	Auxilia a escolher a mensagem correta para o oráculo.
RegExValueGenerator	Um modelo para geração de valores para expressões regulares.
ResultSetCache	Cache usado para guardar o valor de consultas a bancos de dados.
SemanticTestSuiteGenerator	Gerador de suíte de testes semânticos valorados e com oráculo.
StringValueGenerator	Gerador de valores usado para strings.
TestMethodGenerationStrategy	Estratégia de geração de métodos de teste semânticos.
TimeValueGenerator	Gerador de valores usado para horas.
ValidValueOption	Opções para geração de valor válido.
ValueGenerator	Um modelo para um gerador de valores válidos e inválidos.
XegerRegExValueGenerator	Um gerador de expressões regulares baseado na biblioteca externa Xeger.

A Figura 14 apresenta as classes relacionadas à geração de valores conforme o tipo de dado e a opção de geração definidas, discutidos na seção 4.6.1. Um gerador de valor (`ValueGenerator`), pode gerar valores conforme a opção desejada. Há uma opção para valores válidos (`ValidValueOption`) e outra para valores inválidos (`InvalidValueOption`). Para cada tipo de dado básico há um gerador correspondente. Adicionalmente, há um gerador baseado em uma expressão regular (que gera valores aleatórios a partir dela).

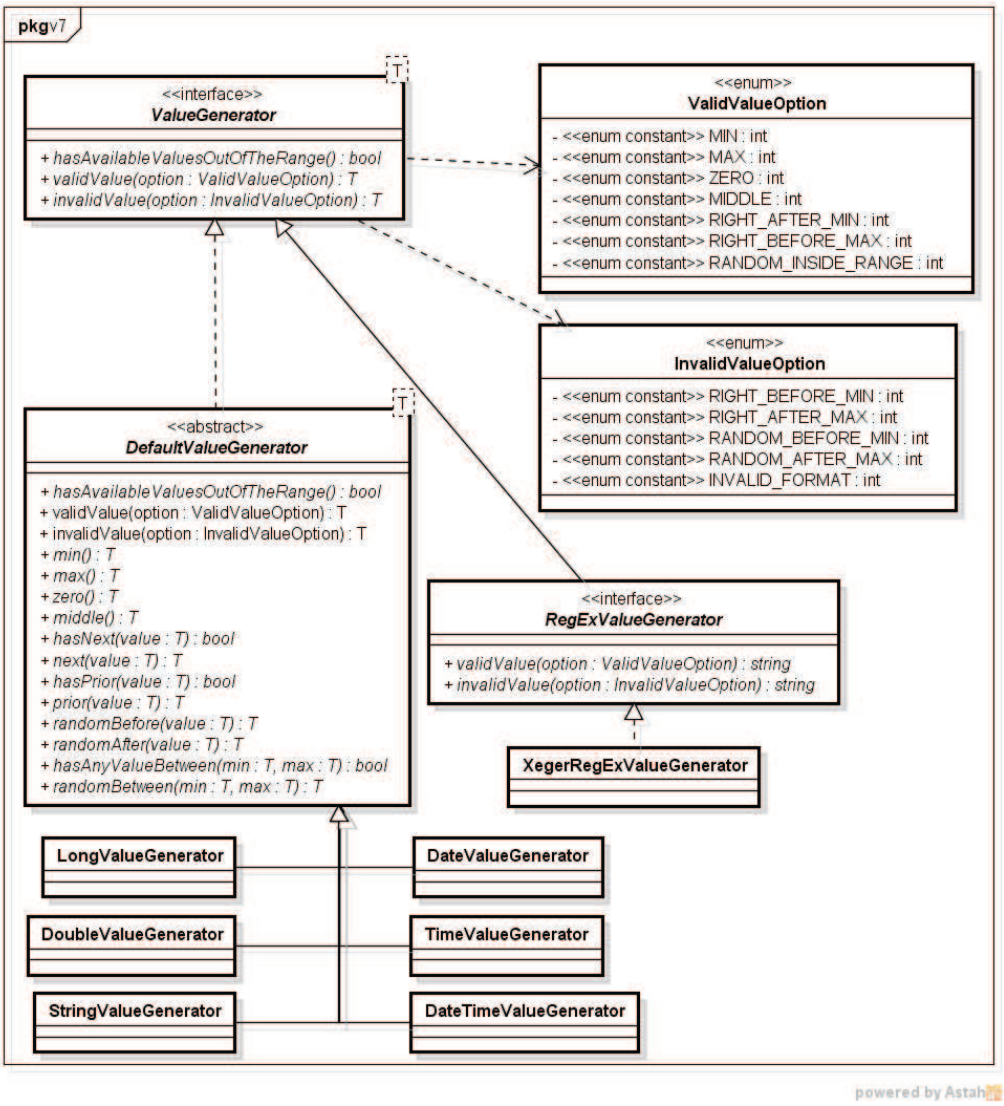

Figura 14 - Classes relacionadas à geração de valores

A Figura 15, a seguir, apresenta as classes relacionadas à geração de valores para elementos de interface com o usuário. Um gerador de valor de elementos (`ElementValueGenerator`) analisa as opções de geração de valor válido

(`ValidValueOption`) e inválido (`InvalidValueOption`), de acordo com as regras de negócio do elemento (`EditableElement`). Para evitar a repetição da geração de valores para regras de negócio que fazem referência a dados externos, armazenados em bancos de dados, é mantido um *cache* (`ResultSetCache`) desses valores. Como diferentes cenários podem necessitar de gerar dados com as mesmas regras, o *cache* permite tornar mais rápida a geração desses valores.

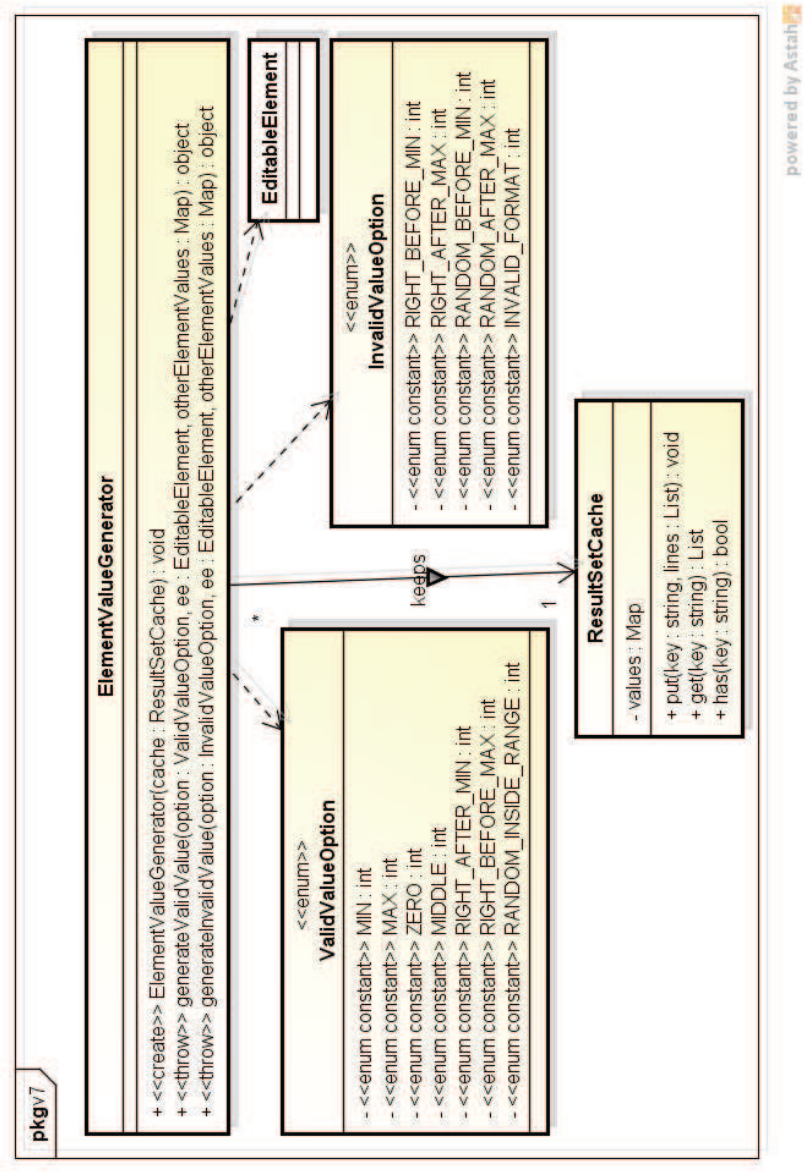

Figura 15 - Classes relacionadas à geração de valores para um elemento

A Figura 16, a seguir, apresenta as classes relacionadas à geração de uma suíte de testes semânticos. O gerador de suíte de testes semânticos (`SemanticTestSuiteGenerator`) provê um meio de gerar uma suíte de testes

(`SemanticTestSuite`) envolvendo todos os cenários (`Scenario`) gerados pela ferramenta, em etapas anteriores. Ele gera um caso de teste semântico (`SemanticTestCase`) para cada cenário. Cada caso de teste semântico é composto por métodos de teste semânticos (`SemanticTestMethod`), que são compostos por passos semânticos (`SemanticStep`) já valorados ou com oráculos. A geração de métodos de teste semânticos ocorre de acordo com uma **estratégia de geração de métodos de teste** (`TestMethodGenerationStrategy`). Cada estratégia gera um método de teste semântico capaz de testar um cenário sob algum aspecto, como informado na Tabela 9 da seção 4.6.3. Os valores para os passos semânticos são gerados de acordo com a(s) regra(s) de negócio a serem exploradas, com auxílio de um gerador de valores (`ElementValueGenerator`).

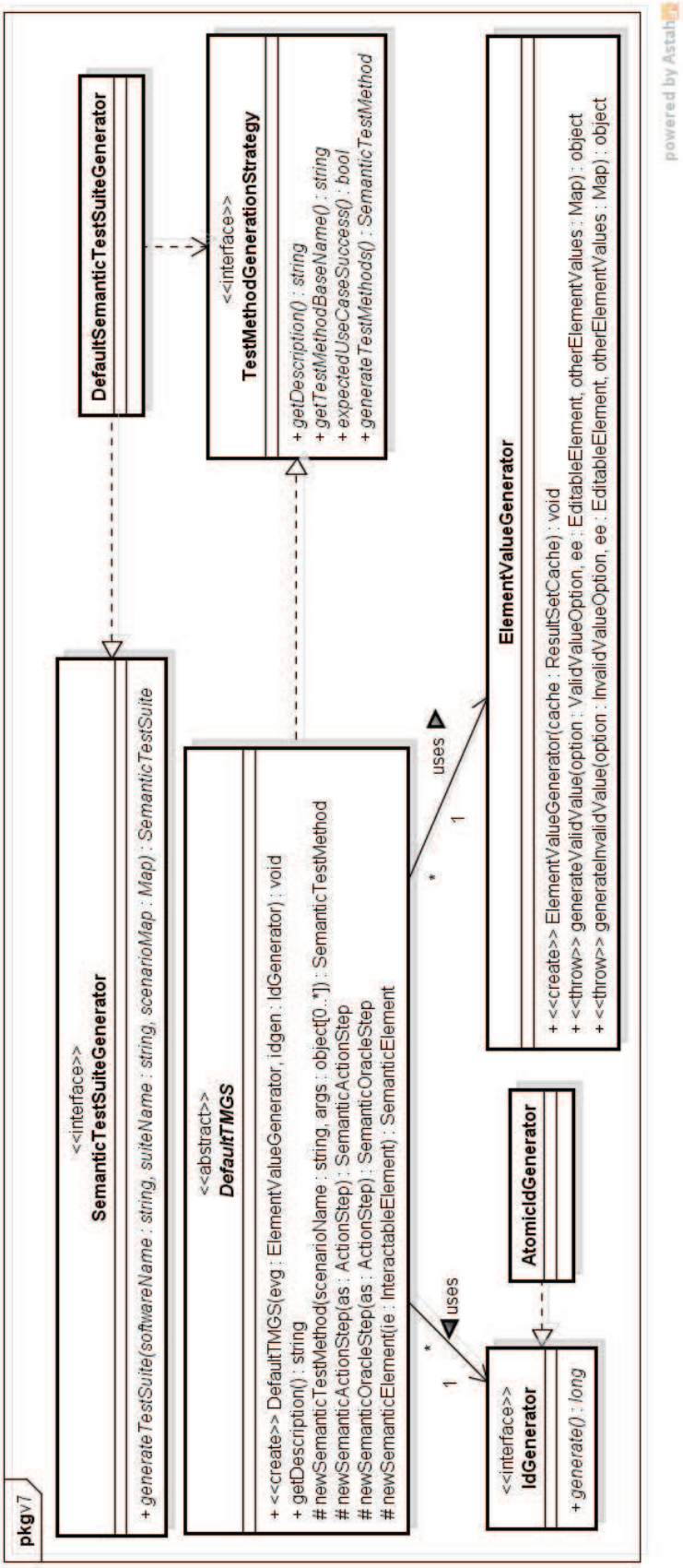

Figura 16 - Classes relacionadas à geração de uma suíte de testes

A Figura 17, a seguir, apresenta as classes relacionadas à geração de métodos de teste semânticos exploram as condições de funcionamento de um caso de uso quando valores obrigatórios não são informados ou valores considerados inválidos (de acordo com as regras de negócio) são informados.

Os tipos de métodos de teste gerados seguem o exposto na Tabela 9 (seção 4.6.3). Neles, todos os elementos de interface com o usuário, exceto um, recebem valores *válidos*. Esse elemento que não recebe valor válido pode ficar não preenchido (`AllRequiredButOneTGMS`), ter valor aleatório acima (`AllWithValidValueButOneWithRandomValueAfterTheMaximumTGMS`) ou abaixo (`AllWithValidValueButOneWithRandomValueBeforeTheMinimumTGMS`) do permitido, ou ter valor imediatamente acima (`AllWithValidValueButOneWithValueRightAfterTheMaximumTGMS`) ou imediatamente abaixo (`AllWithValidValueButOneWithValueRightBeforeTheMinimumTGMS`) do permitido. Em todos os casos, o teste gerado verifica se o SST apresenta a mensagem configurada em sua respectiva regra de negócio. É importante notar que um cenário pode conter mais de um caso de uso e somente será atribuído um valor inválido a um elemento do *caso de uso sob teste*. Os elementos de outros casos de uso receberão valores válidos *aleatórios*. O caso de uso sob teste é sempre o caso de uso para o qual o cenário foi gerado.

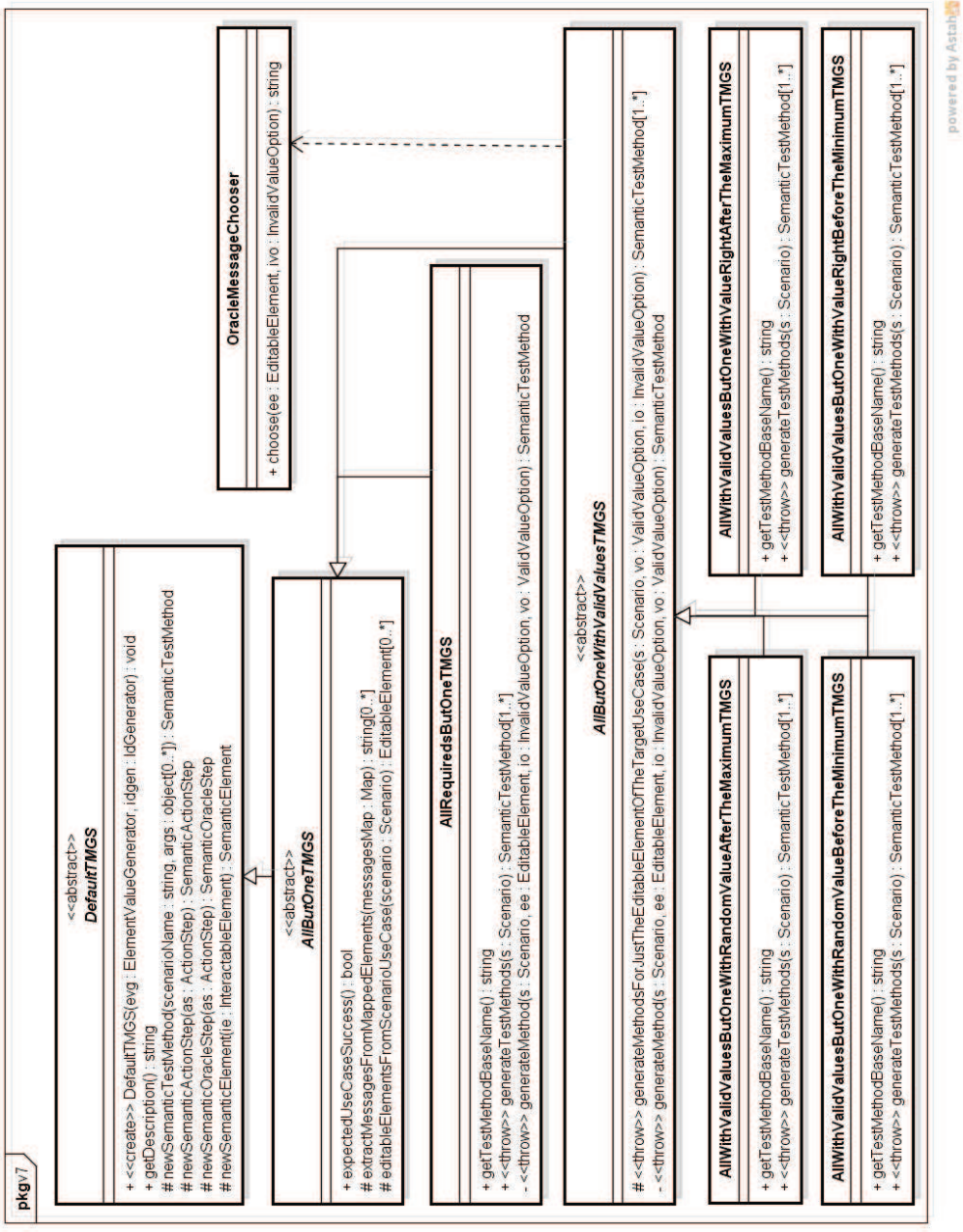

Figura 17 - Classes relacionadas à geração de métodos de teste semânticos com valores inválidos

As classes apresentadas na Figura 18, a seguir, são responsáveis por gerar métodos de teste semânticos que verificam se o caso de uso se comporta conforme esperado, quando fornecidos valores válidos aos seus elementos de interface com o usuário. Os tipos de métodos de teste gerados são apresentados na Tabela 9 da seção 4.6.3. Cada tipo (ou estratégia de geração) irá gerar valores de acordo com o exposto na seção 4.6.1, visando verificar se uma ou mais regras de negócio estão sendo obedecidas pelo *caso de uso sob teste*. Receberão valores *aleatórios* válidos

os elementos do caso de uso sob teste cuja regra de negócio a ser verificada não tiver sido definida e os elementos dos demais casos de uso envolvidos no cenário.

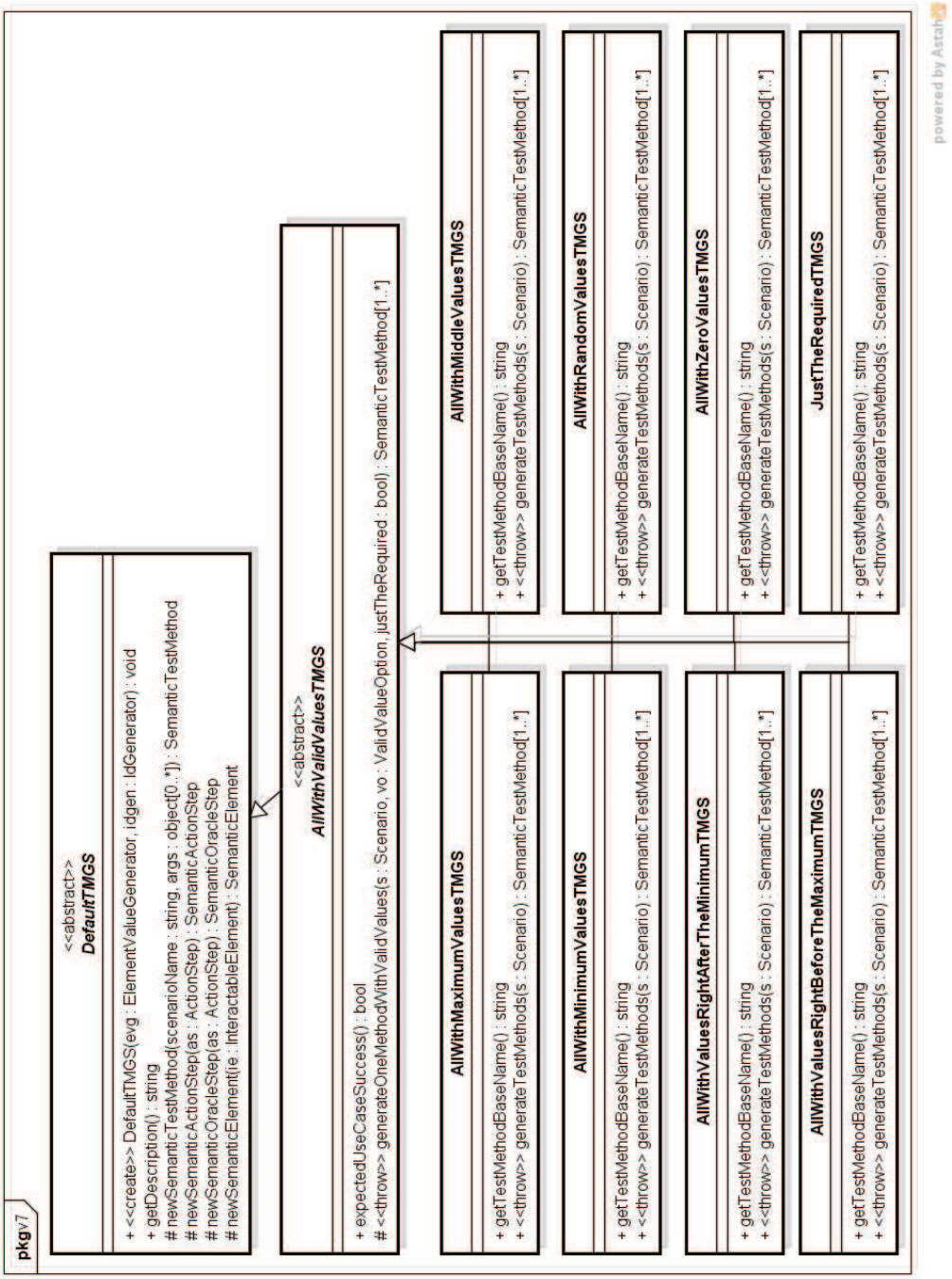

Figura 18 - Classes relacionadas à geração de métodos de teste semânticos com valores válidos

Assim, existem estratégias de geração de métodos de teste que, para os elementos de interface com o usuário geram: valores máximos (`AllWithMaximumValuesTMGS`); valores intermediários, entre os mínimos e os

máximos (`AllWithMiddleValuesTMGS`); valores mínimos (`AllWithMinimumValuesTMGS`); valores aleatórios, dentro da faixa válida (`AllWithRandomValuesTMGS`); valores imediatamente maiores que os mínimos (`AllWithValuesRightAfterTheMinimumTMGS`); valores imediatamente menores que os máximos (`AllWithValuesRightBeforeTheMaximumTMGS`); valores zero (`0`), quando aplicáveis (`AllWithZeroValuesTMGS`); e valores aleatórios somente para os elementos considerados obrigatórios (`JustTheRequiredTMGS`).

5.3.3.Arquitetura da extensão da ferramenta

A seguir são apresentadas as principais classes da **extensão da ferramenta**, construída para os *frameworks* TestNG e FEST (seção 5.2.2). Algumas delas foram apresentadas anteriormente, sobretudo na Figura 12 (que mostra as classes relacionadas à suíte de testes semânticos valorados e com oráculos) e na Figura 13 (que mostra as classes relacionadas ao resultado da execução dos testes).

A Figura 19, a seguir, apresenta as classes relacionadas ao relatório de execução dos testes, específicas para representar formato utilizado pelo *framework* TestNG. O conteúdo lido deste relatório é transformado para o formato interno da ferramenta (que é independente de *framework*), representado pelas classes da Figura 13. Um transformador de relatório de execução de testes (`TestExecutioReportTransformer`) foi definido para isto. Uma de suas possíveis implementações (`TestNGReportTransformer`) possibilita transformação a partir do formato usado pelo *framework* TestNG. Esse *framework* estrutura o relatório de execução com um resultado geral (`TestNGXmlReportResult`), uma suíte de testes (`TestNGXmlReportSuite`), um ou mais casos de teste (`TestNGXmlReportTest`), cada qual contendo classes de teste (`TestNGXmlReportClass`), métodos de teste (`TestNGXmlReportTestMethod`) e, possivelmente, exceções (`TestNGXmlReportException`).

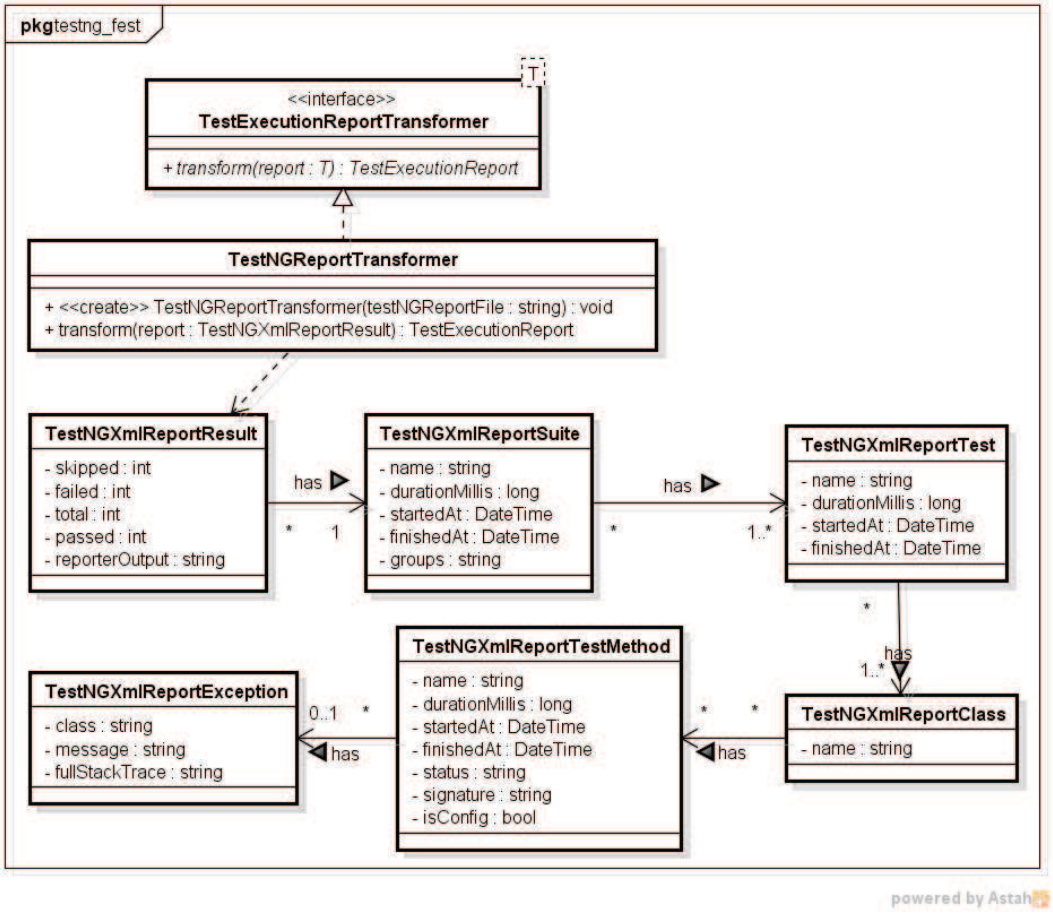

Figura 19 - Classes que representam a estrutura do relatório gerado pelo TestNG