3
Ferramenta Proposta

3.1.
Objetivos

O objetivo deste trabalho é a criação de um *framework* de testes que incorpore algumas das novas idéias encontradas na literatura. Sua principal característica deve ser o foco na especificação de sistemas. Como em metodologias ágeis muitas vezes a experimentação e os ciclos de desenvolvimento curtos são utilizados em detrimento da criação de documentos de especificação mais detalhados [19], a criação dos casos de teste antes do início da codificação é uma forma bastante interessante de se realizar um planejamento do que será desenvolvido. Portanto, criar um *framework* de testes que apresente evoluções na especificação de sistemas pode gerar bons resultados para o desenvolvimento de *software* utilizando métodos ágeis.

A aplicação das idéias de BDD na construção deste *framework* é um caminho interessante para se conseguir tais resultados. A principal idéia que surge das técnicas de BDD é a utilização de linguagens mais ricas e mais próximas do domínio da aplicação. Dos diversos *frameworks* pesquisados, alguns utilizavam a especificação dos testes em alguma linguagem de programação enquanto outros utilizavam arquivos textuais. Embora as descrições em arquivo de texto apresentem vantagens em relação às realizadas em código, estas ainda podem não utilizar a linguagem ideal para descrever os comportamentos do sistema. De fato é provável que não exista uma única linguagem ideal para representar todos os domínios de aplicações [16]. Portanto, o *framework* desenvolvido deve fornecer uma forma de definição de novas linguagens ou de evolução de linguagens já existentes de acordo com as necessidades de cada equipe.

O formato de apresentação dos resultados é outro ponto importante. Os resultados da execução dos testes devem ser separados de seu formato de apresentação, de forma que seja possível apresentar os mesmos resultados em diferentes formatos, inclusive utilizando diferentes recursos como e-mail,

resultados disponíveis na web, entre outros. Para tal, o *framework* deve fornecer meios para que seja possível definir novas formas de apresentação dos resultados.

Tornando possíveis as personalizações através de interfaces de entrada e saída do *framework*, passa a ser possível também realizar a sua integração com outras ferramentas, inclusive com a criação de linguagens e formatos de apresentação de resultados gráficos. A integração deste *framework* com ambientes de desenvolvimento já utilizados pelas organizações pode fornecer uma forma mais simples de se introduzir esta nova ferramenta.

3.2. Motivação

O interesse pela construção de um novo *framework* nasceu do contato com o desenvolvimento de sistemas de controle, com módulos reativos e muitas vezes com requisitos de tempo real, envolvendo interação com outros módulos, sistemas e recursos de hardware. Estes sistemas são muitas vezes de missão crítica, o que torna novas técnicas e ferramentas que resultem em redução na freqüência de ocorrência de falhas especialmente úteis.

Levando-se em consideração que em média oitenta por cento dos defeitos de um sistema são advindos de vinte por cento de seus módulos [1], é interessante identificar os módulos de maior probabilidade de defeitos, através de critérios como complexidade e grau de incerteza em suas especificações. Os módulos com essas características parecem ser os mais propensos a defeitos e, portanto são os que motivam a busca por novas soluções. Uma forma interessante de especificar os comportamentos de tais módulos é através de máquinas de estados [3]. Freqüentemente as etapas iniciais de especificação desses sistemas envolvem máquinas de estados, que posteriormente são utilizadas como documentação e transformadas em testes que utilizam outras linguagens. Este processo cria múltiplos artefatos, os quais possuem o mesmo conteúdo em diferentes linguagens, necessitando que sempre que um comportamento seja alterado, a alteração seja propagada para os diversos documentos, tarefa que, se realizada de forma manual, é altamente propensa a erros e omissões. Portanto a possibilidade de executar seqüências de eventos diretamente em uma máquina de estados verificando se as transições ocorrem corretamente é provavelmente uma forma mais eficiente de especificar e testar esse tipo de sistema.

Muitas idéias empregadas na construção deste *framework* surgiram da utilização do arcabouço para automação de testes redigidos em C [20], que possui uma arquitetura que torna bastante simples modificações para a inclusão de novos comandos de teste. O primeiro módulo de linguagem e de interpretação de comandos foi escrito utilizando uma estrutura bastante semelhante à encontrada nesta ferramenta. O *framework* foi desenvolvido utilizando a linguagem Java [18] versão 6, e utilizando técnicas de inversão de controle e reflexão computacional.

3.3. Arquitetura

A ferramenta proposta é um *framework* de testes composto pelos seguintes elementos:

- Módulo(s) sob teste, ou seja, o(s) artefato(s) que se deseja testar utilizando a ferramenta.
- Um módulo específico de testes, responsável por servir de interface entre o framework e o(s) módulo(s) sob teste.
- Um *script* de testes, que será interpretado pela ferramenta para exercitar o módulo específico de testes e conseqüentemente o(s) módulo(s) sob teste.

A figura 7 mostra um diagrama com a interação entre esses elementos.

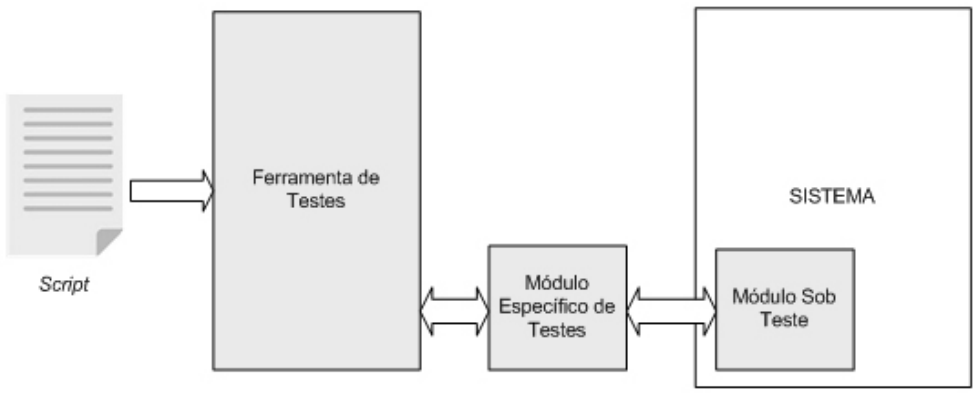

Figura 7: Diagrama de interação dos elementos da ferramenta

Essa arquitetura permite que os usuários da ferramenta escrevam apenas o código de interface entre o sistema e a ferramenta na linguagem nativa do sistema, através do módulo específico de teste, e depois escrevam seus casos de teste em uma linguagem de mais alto nível de abstração, nos *scripts*, permitindo manter o foco na lógica dos testes redigidos.

Internamente, a arquitetura do *framework* é composta por três camadas, como apresentado na figura 8:

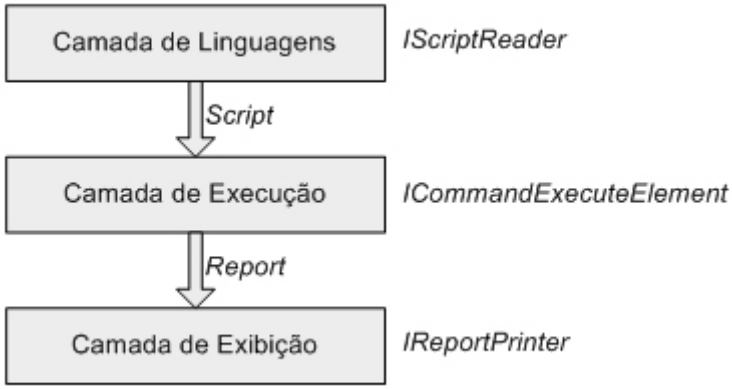

Figura 8: Arquitetura do *framework*

- **Camada de Linguagens**: é responsável por realizar a leitura das especificações de testes e criar um modelo que o represente, que será interpretado pela camada seguinte.
- **Camada de Execução**: É a camada na qual existem os interpretadores responsáveis pela execução de cada um dos comandos do sistema.
- **Camada de Exibição:** que é responsável por apresentar as informações para os usuários.

Para a comunicação entre as diversas camadas, dois modelos são utilizados:

- Modelo da especificação: Representa a especificação de testes lida pela camada de linguagem e que será interpretada pela camada de execução.
- Modelo dos resultados: É o modelo que representa os resultados gerados pela camada de execução por conta da execução dos testes, e que será exibido pela camada de exibição.

A figura 9 apresenta um diagrama de classes da ferramenta.

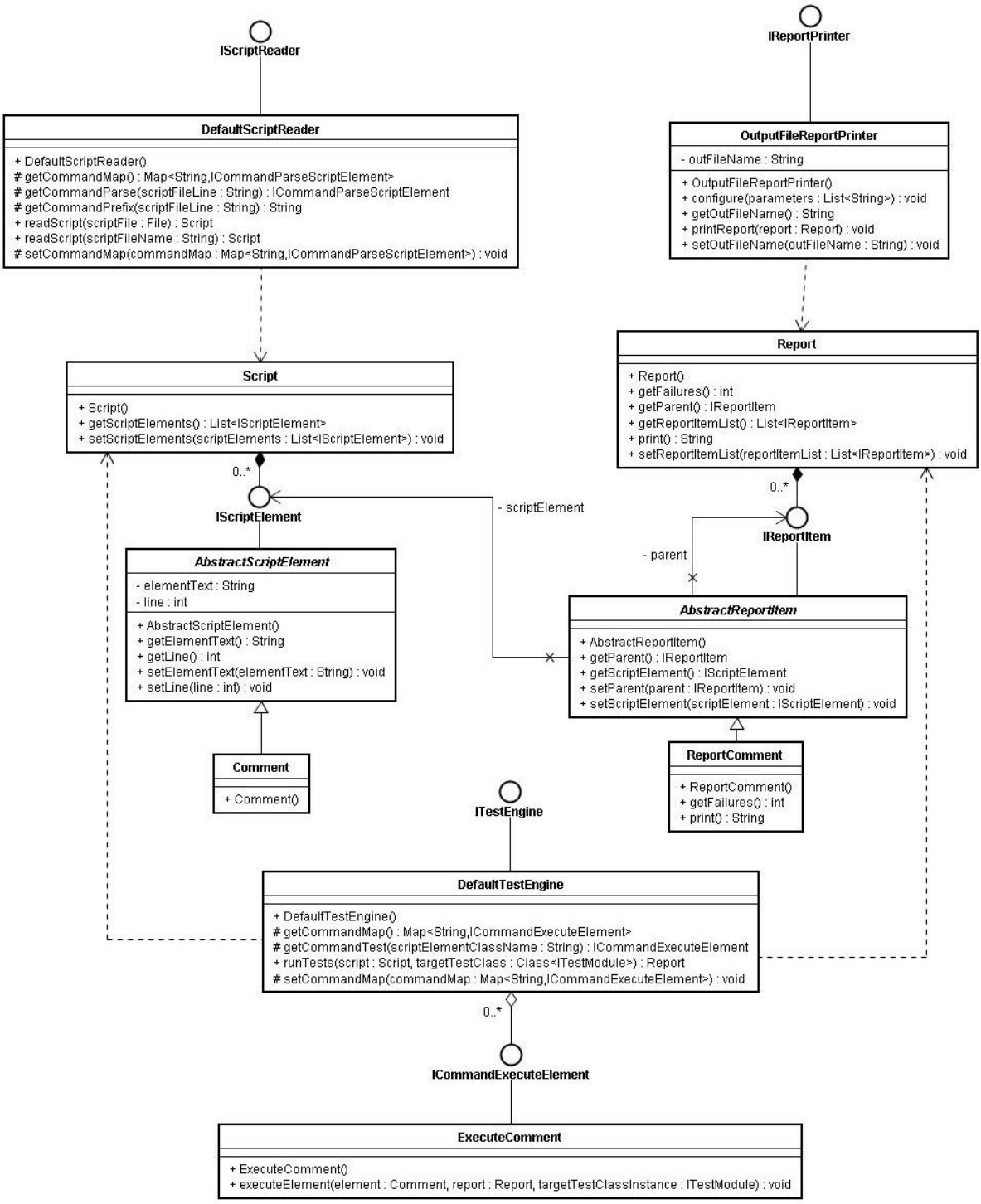

Figura 9: Diagrama de Classes da Ferramenta

Módulos de linguagens devem implementar a interface ***IScriptReader***, que possui o método ***readScript*** que retorna um objeto do tipo ***Script***. O objeto do tipo ***Script*** possui uma lista de ***IScriptElement*** que representam elementos da especificação de testes que foram lidos pelo módulo de linguagem utilizado,

possuindo também algumas meta informações relacionadas à linguagem como por exemplo, no caso de um arquivo texto, a linha e o texto do comando que gerou o elemento no ***Script***. Estes elementos constituintes do ***Script*** serão interpretados na camada de execução.

Na camada de execução existem interpretadores responsáveis pela execução de cada um dos elementos que devem implementar a interface ***ICommandExecuteElement***. Esta interface possui o método ***executeElement*** que recebe como entradas o módulo que será testado, o elemento ***IScriptElement*** que será executado, e um objeto do tipo ***Report***. Os objetos do tipo ***Report*** representam relatórios de resultados de execução das especificações e possuem uma lista de objetos do tipo ***IReportItem***. Cada objeto ***IReportItem*** representa os resultados da execução de um ***IScriptElement***, contendo informações como o sucesso do teste, mensagens de erro e outras informações relevantes sobre a execução. Portanto, ao definir em uma linguagem um novo elemento de linguagem devem ser definido um novo ***IScriptElement*** para representá-lo, um novo ***ICommandExecuteElement*** que será responsável pela sua execução e um novo ***IReportItem*** que apresentará os resultados do processo. Para redefinir sintaticamente um elemento já existente, entretanto, essas alterações não são necessárias, bastando as alterações na camada de linguagem.

Depois de gerado o relatório de resultados, este é passado para a camada de exibição. Módulos de apresentação de resultados devem implementar a interface ***IReportPrinter***, que possui o método ***printReport***. Este método é responsável por obter os dados dos elementos do relatório e exibi-los de forma adequada. Mais de um método de saída pode ser utilizado, uma vez que é possível que seja necessário exibir um mesmo relatório de diferentes formas.

A figura 10 mostra um diagrama de seqüência descrevendo o funcionamento da ferramenta.

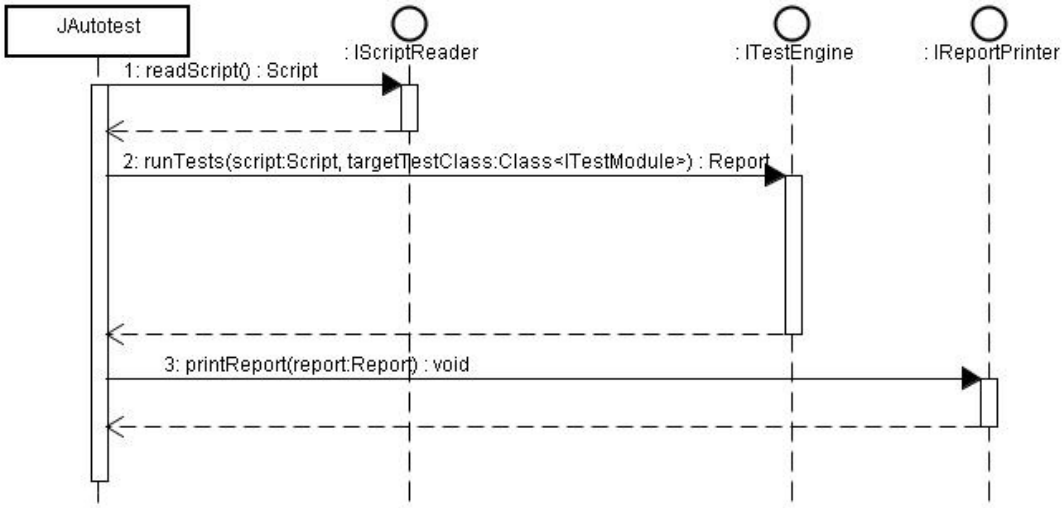

Figura 10: Diagrama de seqüência descrevendo o funcionamento da ferramenta.

A ligação dos diversos módulos é realizada em tempo de execução utilizando conceitos de injeção de dependências e reflexão computacional, através de um arquivo de configuração em que o usuário altera propriedades que possuem os nomes das classes que implementam as interfaces.

A figura 11 mostra um diagrama da ferramenta instanciada com os diversos elementos que a compõem.

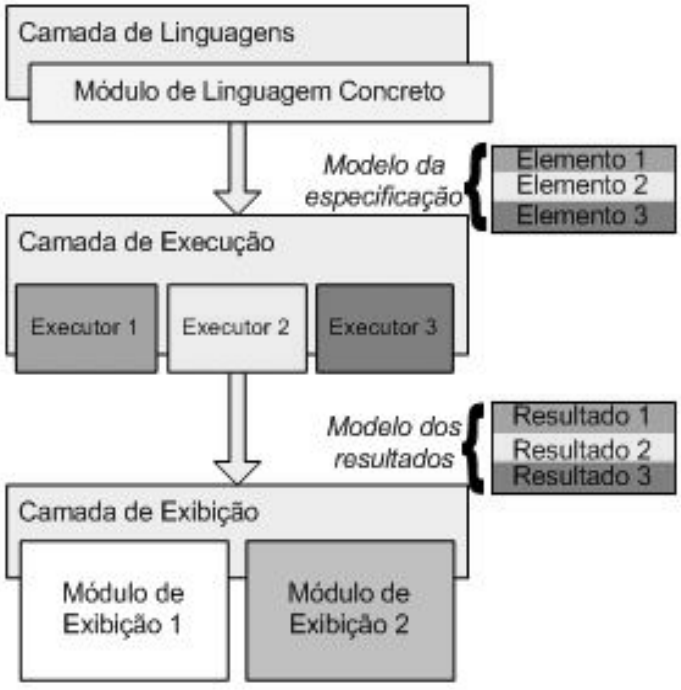

Figura 11: Diagrama da ferramenta após a instanciação.

Esta abordagem permite que os usuários alterem os módulos utilizados em diferentes instanciações do *framework* de forma bastante simples, diminuindo o custo de reuso dos módulos desenvolvidos por uma organização, ou até mesmo através de compartilhamentos entre diferentes organizações.

3.4. Instância de Exemplo da Ferramenta

3.4.1. A Linguagem

A figura 12 apresenta um exemplo de arquivo de configuração criando uma instância da ferramenta de testes. Esse exemplo utiliza apenas os módulos de linguagem, execução e exibição disponibilizados com a ferramenta, porém serve para ilustrar como ocorre a criação de uma instância, através da definição dos diversos componentes que a compõem.

```
1  SCRIPT_READER_CLASS_NAME=br.pucrio.inf.jautotest.reader.DefaultScriptReader
2  TEST_ENGINE_CLASS_NAME=br.pucrio.inf.jautotest.test.DefaultTestEngine
3  REPORT_PRINTER_CLASS_NAME=br.pucrio.inf.jautotest.report.OutputFileReportPrinter
4
5  ELEMENT_PREFIX_LIST=//->recupera->caso->executa->procedimento->chama
6
7  SCRIPT_ELEMENT_PARSE_CLASS_//=br.pucrio.inf.jautotest.reader.command.CommandParseComment
8  SCRIPT_ELEMENT_PARSE_CLASS_caso=br.pucrio.inf.jautotest.reader.command.CommandParseTestCase
9  SCRIPT_ELEMENT_PARSE_CLASS_executa=br.pucrio.inf.jautotest.reader.command.CommandParseTestCommand
10 SCRIPT_ELEMENT_PARSE_CLASS_recupera=br.pucrio.inf.jautotest.reader.command.CommandParseRecovery
11 SCRIPT_ELEMENT_PARSE_CLASS_procedimento=br.pucrio.inf.jautotest.reader.command.CommandParseProcedure
   Definition
12 SCRIPT_ELEMENT_PARSE_CLASS_chama=br.pucrio.inf.jautotest.reader.command.CommandParseProcedureCall
13
14 COMMAND_RUN_TEST_CLASS_Comment=br.pucrio.inf.jautotest.test.command.ExecuteComment
15 COMMAND_RUN_TEST_CLASS_TestCase=br.pucrio.inf.jautotest.test.command.ExecuteTestCase
16 COMMAND_RUN_TEST_CLASS_Procedure=br.pucrio.inf.jautotest.test.command.ExecuteProcedureDefinition
17
18 COMMAND_RUN_TEST_COMMAND_CLASS_TestCommand=br.pucrio.inf.jautotest.test.command.ExecuteTestCommand
19 COMMAND_RUN_TEST_COMMAND_CLASS_TestCommandRecovery=br.pucrio.inf.jautotest.test.command.ExecuteTestC
   ommandRecovery
20 COMMAND_RUN_TEST_COMMAND_CLASS_ProcedureCall=br.pucrio.inf.jautotest.test.command.ExecuteProcedureCa
   ll
```

Figura 12: Exemplo de arquivo de configuração

Na linha 1 do arquivo da figura 12 é realizada a declaração do módulo de linguagem que será utilizado, através do nome da classe que o implementa. A sua ligação ocorrerá em tempo de execução utilizando-se recursos de reflexão computacional. No exemplo em questão, o módulo de linguagem utilizado é o ***DefaultScriptReader***, que é um módulo distribuído junto à ferramenta e que por si só já possui diversas capacidades de extensão. Esse módulo permite que novos comandos sejam cadastrados, e também torna mais simples realizar pequenas extensões na linguagem, como alterar o texto ou o comportamento de um

comando existente. Na linha 5, encontramos os comandos disponíveis nesse módulo. Neste exemplo estão disponíveis os seguintes comandos:

- // <**comentário**>: Adiciona o comentário ao script.
- **Recupera**: Recupera um erro esperado na execução dos testes, reduzindo a contagem de erros em uma unidade.
- **Caso <nome do caso de testes>**: Cria um novo caso de teste. Um caso de teste é uma seqüência de comandos de teste, comentários e chamadas de procedimentos.
- **Executa <comando de teste> <parâmetros>**: Executa um comando de teste com os parâmetros passados. Equivale dizer que o método correspondente a ele será invocado no módulo específico de testes. Os comandos de teste podem possuir os seguintes tipos de parâmetros:
  - **Texto**: Deve aparecer no *script* entre aspas duplas (como por exemplo, “teste”) e será mapeado em Java para um parâmetro do tipo String.
  - **Caractere**: Deve aparecer no *script* entre aspas simples (exemplo, ‘a’) e será mapeado para um parâmetro do tipo char.
  - **Número**: Aparecem no script sem nenhum caractere delimitador e podem ser de ponto flutuante ou não. São mapeados para o tipo Double em Java.
- **Procedimento <nome do procedimento>**: Cria um procedimento com o nome passado como parâmetro. Um procedimento é uma seqüência de comandos de teste, comentários e chamadas de procedimentos.
- **Chama <nome do procedimento>**: Invoca o procedimento com o nome passado como parâmetro.

Nas linhas seguintes, de 7 até 12, são declarados os sub-elementos desse módulo de linguagem responsáveis por realizar a leitura de cada um dos comandos disponíveis. Os leitores de cada comando nessa linguagem devem implementar a interface ***ICommandParseScriptElement***. Através desse arquivo também é possível alterar o texto dos comandos de teste do script. Para isso, deve-

se cadastrar o comando com um nome diferente na linha 5 na propriedade **ELEMENT_PREFIX_LIST**. Como deve existir um elemento responsável pela leitura de cada comando cadastrado nessa propriedade, ao alterar um nome, deve-se alterar o cadastro do componente leitor do comando. Na figura 13, vemos outra possibilidade para a linha 5, com os textos dos comandos alterados e as respectivas mudanças nos leitores nas linhas 7 a 12.

```
5  ELEMENT_PREFIX_LIST=//→=recuperar→==→=→@=→@
6
7  SCRIPT_ELEMENT_PARSE_CLASS_//=br.pucrio.inf.jautotest.reader.command.CommandParseComment
8  SCRIPT_ELEMENT_PARSE_CLASS_eqeqsp=br.pucrio.inf.jautotest.reader.command.CommandParseTestCase
9  SCRIPT_ELEMENT_PARSE_CLASS_eq=br.pucrio.inf.jautotest.reader.command.CommandParseTestCommand
10 SCRIPT_ELEMENT_PARSE_CLASS_eqrecuperar=br.pucrio.inf.jautotest.reader.command.CommandParseRecovery
11 SCRIPT_ELEMENT_PARSE_CLASS_@eq=br.pucrio.inf.jautotest.reader.command.CommandParseProcedureDefinition
12 SCRIPT_ELEMENT_PARSE_CLASS_@=br.pucrio.inf.jautotest.reader.command.CommandParseProcedureCall
```

Figura 13: Exemplo de mudança nos comandos no arquivo de configuração

Na linha 2 do arquivo é definido o módulo utilizado como “motor” da ferramenta de testes. Esse módulo usualmente não precisa ser modificado para a maioria das extensões do *framework*, pois é possível redefinir os componentes responsáveis pela execução específica de cada comando registrado. Tal configuração é especialmente útil para o caso da inclusão de comandos que introduzem novos elementos de *script* ao sistema. Além do módulo que conhece a forma de ler o comando do arquivo de entrada, é necessário criar o módulo que sabe executá-lo e adicionar o resultado de sua execução ao relatório. Componente de execução de comandos devem implementar a interface ***ICommandExecuteElement***, que possui o método ***executeElement***, que recebe o elemento de script a ser executado, o relatório onde os resultados da execução devem ser adicionados e a instância da classe que implementa o módulo específico de teste. Para registrar um novo componente de teste criado deve-se configurar a propriedade relativa ao elemento o qual ele é responsável pela execução com o nome da classe que o implementa. Essa configuração se encontra nas linhas 14 até 16.

Na linha 3 está o parâmetro que configura o módulo utilizado na camada de exibição da ferramenta, **REPORT_PRINTER_CLASS_NAME**. O módulo inicialmente configurado é um módulo que imprime os dados do ***Report*** em um arquivo texto, o ***OutputFileReportPrinter***. Esse módulo pode ser redefinido para se guardar os resultados em banco de dados, realizar a integração com outras ferramentas, entre outras possíveis aplicações. Nesse parâmetro, mais de um

módulo de saída pode ser definido, e para tal é preciso valorar o parâmetro com os nomes das classes dos diversos módulos separados pelo caractere especial de quebra de linha, neste caso representado pelo símbolo **'\n'**. Cada módulo será executado na ordem em que aparece na lista. A figura 14 apresenta um exemplo de configuração com múltiplos módulos de saída.

```
REPORT_PRINTER_CLASS_NAME=br.pucrio.inf.jautotest.report.OutputFileReportPrinter\n\
br.pucrio.inf.jautotest.report.SendMailReportPrinter
```

Figura 14: Exemplo de configuração com dois módulos de saída.

Isso é especialmente importante para eliminar a necessidade de criar novos módulos quando se deseja uma funcionalidade que represente a união de dois módulos já existentes como, por exemplo, um módulo que gere um relatório local e outro módulo de integração com um sistema de integração contínua.

Além dessas configurações, o arquivo contém ainda configurações relativas à execução dos comandos subordinados a blocos de comandos. Nessa linguagem blocos de comandos podem ser procedimentos ou casos de teste. No momento em que comandos subordinados são lidos, eles são adicionados aos blocos que os contém, de maneira que no momento da execução dos testes tais elementos não aparecerão como elementos do ***Script***, mas sim como sub-elementos de outros elementos. No momento da execução, portanto, eles não precisam ser interpretados pelos componentes de execução registrados, e sim por sub-componentes de execução dos blocos de comandos. Esses sub componentes são registrados nas linhas 18 a 20 e devem implementar a interface ***ICommandExecuteTestCommand***. Nesta implementação de módulo de linguagem esses sub componentes de execução serão utilizadas pelos componentes de execução dos comandos **Caso** e **Chama**. Vale lembrar que esta infraestrutura de sub-comandos não faz parte da arquitetura básica do framework. Este esquema foi apenas uma forma de criar os componentes responsáveis pela execução dos blocos comandos que utilizou uma solução semelhante à que foi adotada para o cadastro dos componentes da ferramenta. Entretanto, esta solução ilustra de forma bastante interessante como a solução adotada pode ser aplicada recursivamente para criar componentes configuráveis em diversos níveis da implementação das instâncias.

3.4.2.
Scripts e Módulo Específico de Testes

Com a linguagem configurada na ferramenta, os scripts são escritos utilizando os comandos disponíveis. A figura 15 mostra um exemplo de script para a linguagem descrita anteriormente, para o teste de um módulo de lista encadeada.

```
1  procedimento preparaTestes
2  executa removeTodos
3
4  caso testa adicionar elementos no início
5  chama preparaTestes
6  executa adicionaInicio 3
7  executa obtemInicio 3
8  executa obtemInicio 4
9  recupera
```

Figura 15: Exemplo de *script* de testes

Para cada um dos diferentes comandos de teste executados pelo comando **executa <comando de teste> <parâmetros>** deve existir método no módulo específico de testes com o nome **<comando de teste>** e recebendo como parâmetros os parâmetros contidos em **<parâmetros>**. Por exemplo, para executar o comando **Executa exemplo "teste"**, deveria existir um método chamado **exemplo** recebendo um parâmetro do tipo *String*. Os tipos de retornos desses métodos deve ser sempre *void*. Para o script do exemplo acima, por exemplo, o módulo de testes deve conter os métodos da figura 16.

```
public void removeTodos();

public void adicionaInicio( int valor );

public void obtemInicio( int valorEsperado );
```

Figura 16: Assinaturas dos métodos do módulo específico de testes correspondente ao *script* da figura 15

Dentro desses métodos, o programador tem acesso a um objeto de contexto de execução, que deve ser utilizado para sinalizar os erros encontrados durante a execução, como no exemplo da linha 8 da figura 15, o valor obtido seria diferente do esperado. Exceções que ocorram durante a execução do método de teste são automaticamente capturadas pela ferramenta e também se tornam erros no relatório gerado.

Existe ainda na interface do módulo específico de testes os métodos **buildUp** e **tearDown**, que são executados respectivamente antes e depois da execução do script. Esses métodos são úteis para preparar as pré-condições necessárias para a execução dos testes. Outra forma de criar o ambiente necessário para os testes é através de procedimentos definidos nos *scripts*. A idéia original é que os métodos sejam utilizados para a inicialização de elementos que dizem respeito à estrutura interna do módulo específico de testes, como a inicialização de estruturas de dados utilizadas, abertura de conexão com banco de dados, entre outros, enquanto os procedimentos podem ser utilizados para a inicialização de elementos mais ligados aos comportamentos do sistema, como *login* de um usuário, por exemplo.