4
O Sistema de Críticas

Esse capítulo descreve o sistema de críticas que foi desenvolvido no decorrer deste trabalho. Inicia com uma descrição dos requisitos desejáveis de um sistema de críticas, seguido da especificação do sistema implementado, o NCL-Inspector. E finaliza com uma demonstração, através de exemplos, da funcionalidade-chave do sistema: a adição de novas regras. Neste capítulo usaremos as nomeclaturas: regras, regras de detecção, regras de inspeção ou simplesmente inspetores, intercambiavelmente.

4.1
Requisitos do Sistema

Os requisitos desejáveis, abordados nesta seção, estão descritos em um nível de abstração que permite aplicá-los na criação de sistemas de críticas de código de forma geral, ou seja, independentemente da linguagem que se deseja verificar. Conseguimos chegar neste conjunto de requistos a partir de estudo da literatura a respeito. A Tabela 6.1 na página 78 apresenta quais desses requisitos foram contemplados na implementação da ferramenta.

4.1.1
Sistema de críticas baseado em regras

Um conjunto de regras de produção é usado para representar o conhecimento de domínio do sistema. Regras de produção são compostas de: uma pré-condição e uma ação. Quando a pré-condição de uma dada produção é satisfeita, a ação daquela regra será executada.

O sistema de crítica deverá usar padrões de possíveis problemas como entrada. O sistema busca por um determinado padrão no código fonte de uma aplicação, e quando esse padrão é encontrado, uma crítica é apresentada sobre o padrão de código correspondente.

4.1.2 Possibilidade de escrever regras seguindo tanto o paradigma imperativo ou declarativo

Usar o paradigma declarativo para especificar regras é útil pois o programador especifica a tarefa que deve ser feita, ao contrário do paradigma imperativo, que necessita ao desenvolvedor especificar um algoritmo para descrever como a tarefa deve ser feita. Outra vantagem é que linguagens declarativas podem também ser linguagens de domínio específico (DSL – do Inglês *Domain Specific Languages*) (van Deursen et al. 2000). Uma DSL permite que regras possam ser especificadas com um nível de abstração do domínio do problema.

Como exemplo desse tipo de linguagem temos o Schematron, que é uma linguagem para fazer asserções (i.e. verificar a validade) sobre a presença ou ausência de padrões em documentos XML (SchematronSpec). A Organização Internacional para Padronização (ISO – do Inglês *International Organization for Standardisation*) definiu o Schematron como um padrão público.

Infelizmente, especificar regras apenas usando Schematron não é o suficiente. Um contraexemplo é que Schematron realiza apenas validações estruturais e não pode acessar o sistema de arquivos. É impossível para o Schematron, por exemplo, detectar se o atributo `src` de um elemento `<media>` referencia uma mídia inexistente. Por essa razão, é necessário oferecer suporte a uma linguagem de programação de propósito geral como, por exemplo, Java.

4.1.3 Seleção de regras em tempo de execução

O grande desafio em criar um sistema de críticas é dizer a "coisa certa" no "tempo certo" (Fischer 1990). Para encarar esse desafio, o sistema de críticas deve ser flexível o suficiente para permitir ao programador selecionar o conjunto de regras que ele acha mais relevante no contexto do seu projeto. Em algum outro momento, se o programador precisa avaliar seu código com outra perspectiva em mente, ele pode apenas trocar o conjunto de regras e o sistema irá avaliar seu código seguindo outro critério.

4.1.4
Possibilidade de adicionar regras em tempo de carga

É interessante que o sistema de críticas seja distribuído com um conjunto de regras pré-criadas, as quais acredita-se que serão úteis para a maioria das situações durante o desenvolvimento de aplicações. Entretanto, um desenvolvedor pode estar interessado em criar um padrão de codificação para ser usando especificamente por sua equipe de desenvolvimento.

Considera o seguinte exemplo: um desenvolvedor decide que em todas as aplicações NCL escritas por seu time, o atributo `id` de todas as tags `<media>` deve conter a letra "m" como prefixo. Deverá ser possível para ele adicionar uma regra que garanta que todos os desenvolvedores da sua equipe sigam essa convenção. Em outras palavras, o sistema deverá permitir que o desenvolvedor adicione uma nova regra sem a necessidade de recompilar todo código do NCL-Inspector. Para permitir isso, as regras deverão ser implementadas como *plug-ins*. O mecanismo de *plug-in* traz alguns benefícios adicionais. Se, após implementar essa regra, o mesmo desenvolvedor acha que a essa regra pode ser útil também a outros usuários ao redor do mundo, esse requisito também lhe permitirá facilmente compartilhar essa implementação com outros.

4.1.5
Integração com o ambiente de desenvolvimento

Para aumentar a produtividade, quando um programador comete um erro, o sistema apresenta a crítica imediatamente. Nesse momento o desenvolvedor está altamente concentrado em resolver o problema e irá entender a sugestão do sistema de críticas mais rapidamente.

Quando a notificação de uma solução problema ou solução sub-ótima é postergada, é necessário um esforço maior para o desenvolvedor lembrar o que ele estava pensando no momento da falha (Fischer 1990). Um sistema de críticas integrado ao ambiente de desenvolvimento é necessário para prover críticas em tempo real. Entretanto, deve-se ter cuidado para que a crítica não seja muito intrusiva e atrapalhe o usuário no desenvolvimento de suas tarefas.

4.1.6
Explicação do problema e sugestão de correção

Apresentar uma explicação e uma ou mais sugestões de correções é valioso para tornar o sistema de críticas um tipo de instrumento de aprendizado. Quando o sistema encontra um erro ou uma condição sub-ótima, ele irá não apenas apresentar aos usuários a sugestão para melhorar seu projeto, mas também uma explicação do erro e as possíveis causas daquele possível problema.

Fischer afirma (Fischer 1990) que, quando os usuários não entendem por que uma sugestão foi feita, eles normalmente seguem a sugestão dada pelo sistema, mesmo que ela seja ou não apropriada para aquela situação. Explicar cuidadosamente os problemas detectados irá ajudar os usuários a evitar esse problema, ajudando-lhes a jugarem quando a crítica é relevante ou não para a situação encontrada.

4.1.7
Correções de problemas automaticamente

É útil prover correções automáticas, elas reduzem o esforço de aplicá-las manualmente. Muitas IDEs como Eclipse[1], Netbeans[2] e IntelliJ IDEA[3] possuem uma ferramenta de refactoring para código Java, que aplicam automaticamente refatorações no código fonte.

A correção automática deve ser feita de forma que o usuário não se sinta sem o controle. O sistema deve tomar a iniciativa da correção, identificando um problema, então, propõe ao usuário alternativas para correção desse problema. A decisão sobre esse conjunto de alternativas apresentadas deve ser feita exclusivamente pelo usuário, para então, o sistema se encarregar de alterar o código de acordo com a opção escolhida. Podemos considerar essa correção automática como uma forma de Auto-adaptação Controlada pelo Usuário (Dieterich et al. 1993).

[1] http://www.eclipse.org/
[2] http://www.netbeans.org/
[3] http://www.jetbrains.com/idea/

4.1.8 Repositório central de regras

A ferramenta de gerenciamento de projetos Maven[4], que foi usada para auxiliar a construção do NCL-Inspector, possui um repositório central onde são disponibilizadas bibliotecas de código e diversos *plugins* para estender as funcionalidades da própria ferramenta.

De forma análoga ao repositório de plugins e bibliotecas do Maven, seria interessante se as regras criadas para o sistema de críticas pudessem ser disponibilizadas através de um repositório central. Dessa forma, uma regra criada para ser utilizada por uma equipe localmente poderia ser compartilhada com outras equipes, desde que seja útil em um contexto geral. O objetivo da criação deste repositório é possibilitar o desenvolvimento colaborativo e distribuído do sistema de críticas e elevar o reúso de código.

Entre as tecnologias candidatas para a implementação desse repositório estão protocolos como WebDAV, *Web Services* RESTful ou SOAP, NFS (do Inglês – *Network File System*), SMB (do Inglês – *Server Message Block*).

4.2 Especificação

NCL-Inspector é dividido em 4 módulos principais, que são: parser, mecanismo de regras (*rules engine*), biblioteca de regras (ou biblioteca de inspetores) (*rules library*) e construtor do mecanismo de regras (*rules engine builder*). A Figura 4.1 apresenta um esboço simplificado dos componentes principais da arquitetura do NCL-Inspector.

O módulo principal do sistema é o Mecanismo de Regras, que possui a Composição de Inspetores (*Inspectors Composite*). Essa composição de inspetores possui dois inspetores-chave: o `RootInspector` e o `TreeWalkerInpector`. O primeiro é o inspetor raiz da composição, o segundo é o responsável por percorrer a árvore sintática abstrata que representa o documento NCL.

A Figura 4.2 ilustra de forma simplificada os passos realizados pelo NCL-Inspector desde o carregamento das regras, até a exibição das violações encontradas em um documento NCL. O processo se inicia com a leitura do

[4]http://maven.apache.org/

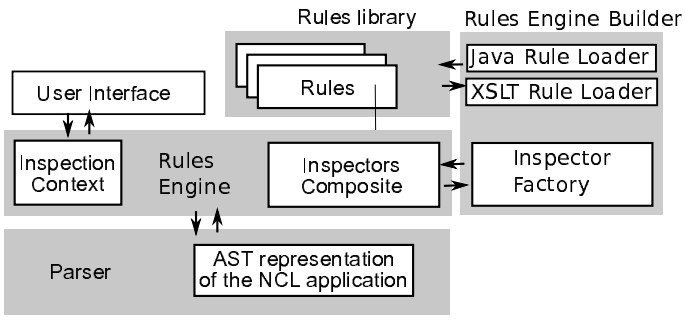

Figura 4.1: Esboço simplificado da arquitetura do NCL-Inspector

Arquivo de Configuração do NCL-Inspector pelo Construtor do Mecanismo de Regras, como é mostrado no passo 1. A partir desse arquivo, em tempo de carga, o Construtor é capaz de selecionar quais serão os inspetores que irão compor o Mecanismo de Regras.

No passo 2, são carregados da Biblioteca de Regras, os inspetores escritos em Java ou XSL que, em seguida, são compostos hierarquicamente, criando o Composite (Gamma et al. 1995) dos Inspetores, que inclui o `TreeWalkerInspector` (passo 3). Então o arquivo NCL fornecido como entrada é lido pelo *parser* e é feita uma validação automática baseada no XML Schema. Se a validação não retornar nenhum erro, o *parser* cria a AST que representa o arquivo NCL (passo 4) e o passo seguinte irá acontecer. Caso a validação falhe, a violação é adicionada ao contexto da inspeção e o processo vai para o passo 8.

Entre o passo 4 e o passo 5, para simplificação, foi omitido que o `RootInspector` é quem inicia a inspeção, delegando ao `TreeWalkerInspector` o arquivo NCL a ser inspecionado. Também foi omitido que entre os passos 7 e 8 (i.e. depois o `TreeWalkerInspector` finaliza o percurso na árvore) o `RootInspector` delega o arquivo para outros `MultipleFilesInspector` que possam existir no Composite.

Voltando ao passo 5, o `TreeWalkerInspector` caminha na AST delegando a inspeção para os `NodeInspector` contidos nele (passo 6). Essa delegação apenas ocorre se o inspetor contido informou que está interessado em inspecionar aquele elemento. Se houver inspetores interessados naquele vértice, a inspeção do vértice é delegada a esse inspetor. Assim, no passo 7, caso a inspeção detecte uma violação, ela é adicionada no contexto da inspeção. Então, o `TreeWalkerInspector` visita o próximo vértice da AST e retorna ao passo 5.

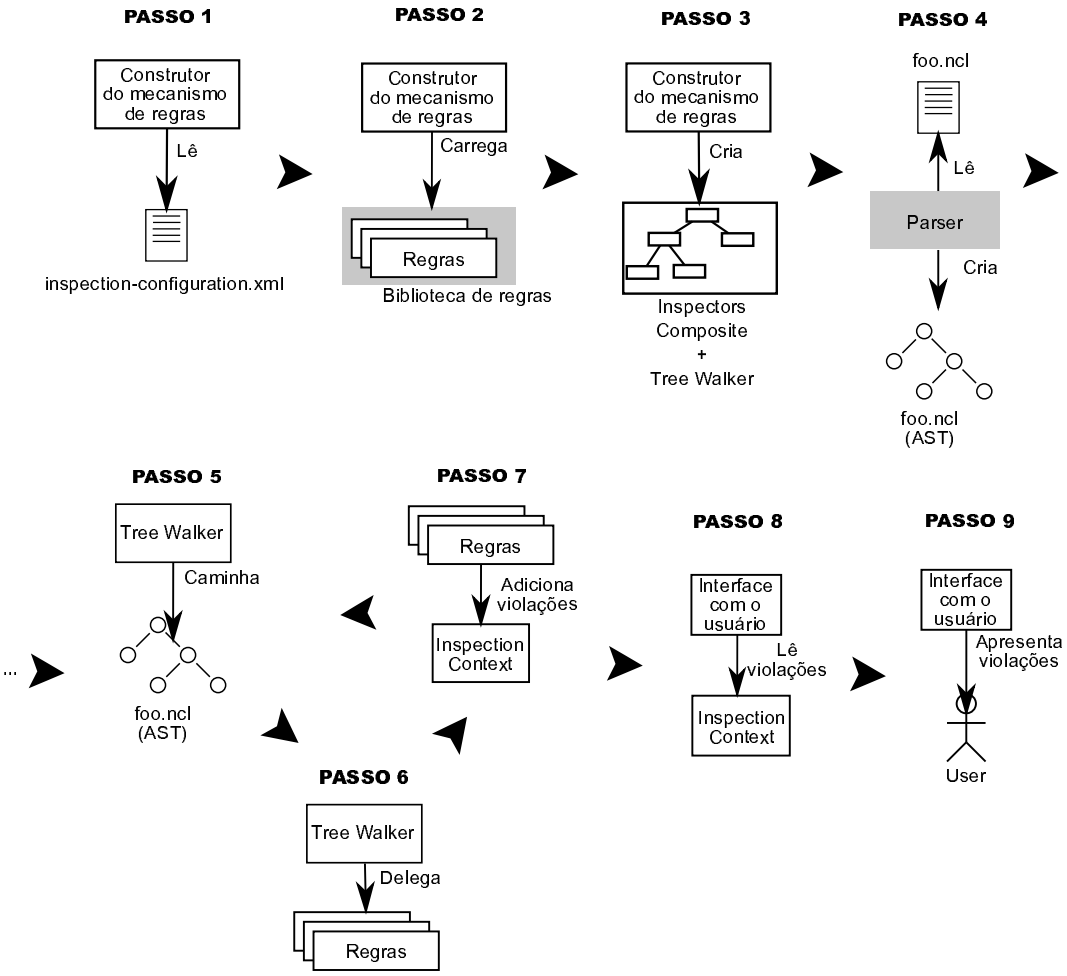

Figura 4.2: Diagrama simplificado do processo de inspeção do NCL-Inspector

Caso não haja mais vértices a serem visitados na AST, a interface com o usuário lê as violações contidas no contexto da inspeção (passo 8) e apresenta-as ao usuário (passo 9). O processo de inspeção está terminado.

Ao longo das próximas subseções, todos os quatro módulos serão apresentados em detalhes, bem como os principais componentes de cada um deles.

4.2.1 Parser

É responsável por criar a Árvore Sintática Abstrata (AST – do Inglês *Abstract Syntax Tree*) da representação textual de uma aplicação. O componente XML Beans[5] foi usado para gerar um parser automaticamente a partir de uma versão modificada do XML Schema da NCL.

[5] http://xmlbeans.apache.org/

A escolha pelo XML Beans foi devido ao fato de que era o único de nosso conhecimento com todos os requisitos necessários para a implementação, que são:

1. permitir a navegação pela AST tanto por uma interface fortemente tipada orientada a objetos, quanto pelo DOM;
2. permitir consultar a linha de código exata de um determinado objeto da AST. Isto é particularmente útil para as implementações das regras exibirem mensagens de erro relacionadas a um objeto da árvore;
3. possuir uma boa performance.

Como mencionado, a representação do código NCL utilizado pelo parser é grafo de objetos. Cada objeto desse grafo tem uma correspondência 1:1 com elementos da linguagem NCL. Para efeitos ilustrativos, observe a Figura 4.3 e veja um diagrama de classes simplificado das classes geradas pelo XML Beans, para representar os elementos `<media>` e `<property>` da NCL.

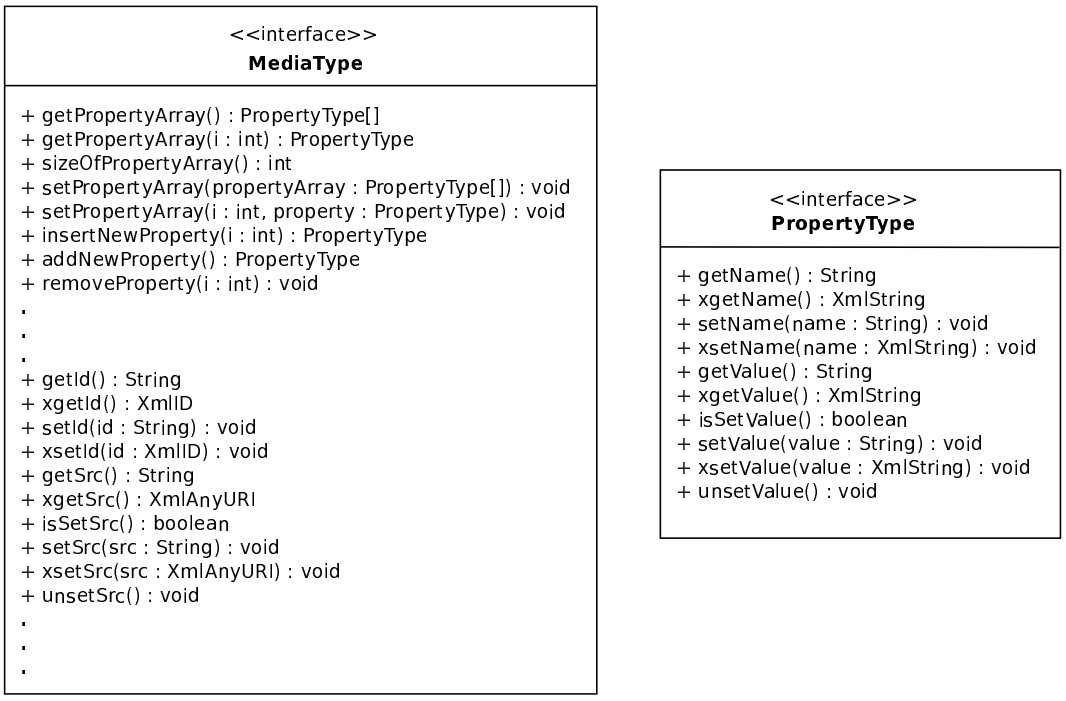

Figura 4.3: Interfaces que representam os elementos media e property da NCL.

A partir de um objeto do tipo `MediaType`, é possível descobrir quais são os valores dos atributos do elemento `<media>` que este objeto representa. Também é possível descobrir os elementos filhos desse `<media>`, isto é, os objetos do tipo `PropertyType` que representam os elementos `<property>`, filhos desse elemento `<media>`.

Ainda na Figura 4.3, os oito primeiros métodos de `MediaType` são exclusivos para manipulação dos objetos `PropertyType`. Para cada tipo que pode estar aninhado, existe um conjunto desses métodos para manipulação desses elementos. Por exemplo, o método `getPropertyArray()` retorna um *array* de `PropertyType` que representa os elementos `<property>` que possui. Como o elemento `<area>` também pode estar aninhado ao elemento `<media>`, logo, existem também oito métodos para manipular `AreaType` (e.g. `getAreaArray()`, `insertNewArea(int)`, etc.). Contudo, foram suprimidos no diagrama para economizar espaço.

Para manipulação dos atributos obrigatórios, existe um conjunto de quatro métodos para realização da tarefa, que no caso do atributo `id`, por exemplo, são: `getId()`, `xgetId()`, `setId(String)` e `xsetId(XmlString)`. No caso dos atributos não obrigatórios, existe uma coleção de seis métodos para manipulá-los. Tanto para atributos obrigatórios e não-obrigatórios, os métodos iniciados por `get` e `set` servem para, respectivamente, recuperar e modificar o valor do atributo. No diagrama, os seis útimos métodos pertencem a coleção de métodos para manipular o atributo `src`. Da mesma forma, apesar de ter sido omitido nesta figura, existe mais seis métodos para manipulação de cada atributo de `<media>`. Para mais informações, consulte a documentação do XML Beans em: `http://xmlbeans.apache.org`.

4.2.2 Mecanismo de Regras

É o núcleo da aplicação, pois concentra as funcionalidades principais do sistema. O mecanismo de regras é responsável por:

1. percorrer a AST;
2. avaliar qual regra é pertinente a qual elemento da AST;
3. disparar a regra pertinente ao objeto que está sendo visitado na árvore;
4. armazenar o contexto da inspeção, que é responsável por armazenar todas as violações detectadas durante o processo de inspeção, para posteriormente exibi-las ao usuário;
5. oferecer formas extensíveis para inclusão de novas regras.

O diagrama de classes simplificado, ilustrado na Figura 4.4, apresenta as classes e interfaces-chave do mecanismo de regras e como elas se relacionam entre si. A interface `Inspector` é a base de todas as classes. Na arquitetura do NCL-Inspector, tudo é um inspetor.

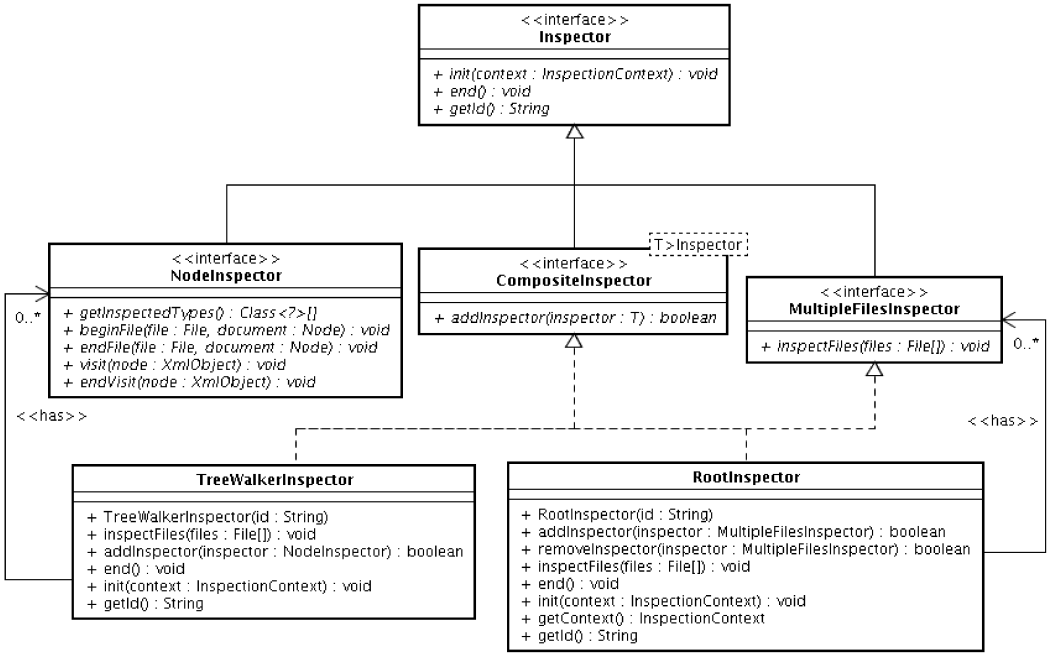

Figura 4.4: Classes e interfaces principais do Mecanismo de regras do NCL-Inspector e seus relacionamentos.

A interface `CompositeInspector<T extends Inspector>` indica que o Inspetor que a implementa tem a capacidade de possuir outros inspetores. Basicamente, fornece uma interface uniforme para a implementação do padrão Composite.

A classe `TreeWalkerInspector` é responsável por armazenar a árvore sintática abstrata que representa o documento (durante todo o processo de inspeção) e por isso, faz uso intensivo do parser. Por ser um `CompositeInspector`, implementa o padrão Composite. Ela é quem percorre a árvore sintática, delegando a inspeção de um determinado elemento NCL para os objetos que implementam `NodeInspector`. Os objetos podem ser adicionados no Composite através do método `addInspector(NodeInspector)`.

Acredita-se que em quase totalidade dos casos, uma regra de inspeção será implementada através de um `NodeInspector`, que fornece uma interface para representação do arquivo através de uma árvore abstrata. Entretanto, um inspetor pode não estar interessado em avaliar o código através da

árvore sintática abstrata. Por exemplo, muitos consideram tabular o código fonte usando o caracter de tabulação (`\t`) uma má prática. Através do `NodeInspector` esse tipo de problema não pode ser verificado, pois a interface oferecida por ele oferece um maior nível de abstração que descarta detalhes que são importantes para essa inspeção. Em casos como esse, devemos usar a interface `MultipleFilesInspector` pois ela trata o programa NCL com nível de abstração textual. Observando a hierarquia, o `TreeWalkerInspector` é uma especialização de um `MultipleFilesInspector`, pois ele precisa montar a árvore sintática a partir da representação textual do NCL.

Na verdade, todo o Mecanismo de Regras está implementado como um Composite, e a raiz dessa composição é o `RootInspector`, como mostra o Diagrama de Objetos, apresentado na Figura 4.5. Esse diagrama mostra um objeto da classe `TreeWalkerInspector` e outro da classe `TabulationInspector`, ambos implementando a interface `MultipleFilesInspector`. Os objetos filhos de `TreeWalkerInspector` implementam a interface `NodeInspector`.

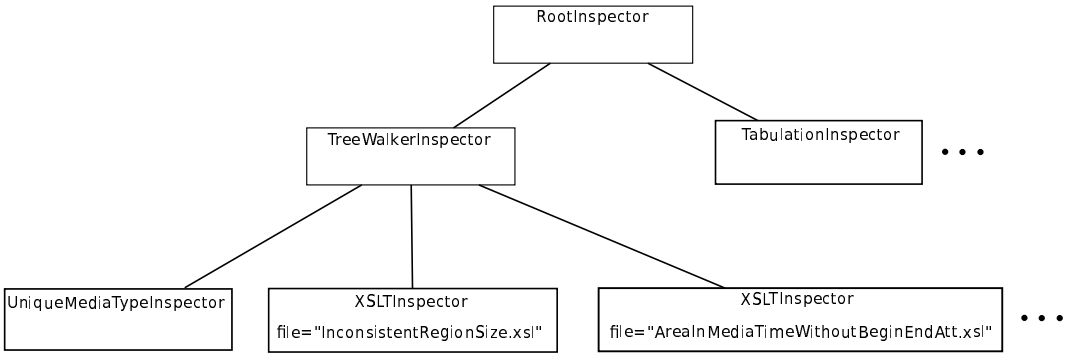

Figura 4.5: Diagrama de objetos do mecanismo de regras

Observe que as regras implementadas em XSL também fazem parte dessa composição. Um objeto do tipo `XSLTInspector`, que também implementa `NodeInspector`, será adicionado à composição para cada regra escrita em XSL. No exemplo em questão, podemos ver duas regras XSL dentro da composição, são elas `"InconsistentRegionSize.xsl"` e `"AreaInMediaTimeWithoutBeginEndAtt.xsl"`. As regras implementadas em XSL são implementadas como filhas de `TreeWalkerInspector`, pois o processador XSL aproveita a árvore sintática do documento, que é armazenada no `TreeWalkerInspector`. Assim não é preciso ler diretamente do arquivo para cada regra XSL, economizando operações custosas de entrada e saída.

Nesse exemplo, as regras representadas por `UniqueMediaTypeInspector` e `TabulationInspector` são regras implementadas em Java. A primeira, por ser filha de `TreeWalkerInspector`, implementa `NodeInspector` e por isso avalia a aplicação NCL a partir da árvore sintática. A segunda, por ser filha de `RootInspector`, implementa `MultipleFilesInspector` e avalia a aplicação NCL com o nível de abstração de um arquivo texto.

4.2.3 Biblioteca de regras (biblioteca de inspetores)

A biblioteca de regras funciona como um repositório e é onde se concentram todas as regras de detecção disponíveis para uso localmente. Uma regra de detecção é o que de fato realiza a inspeções no código, logo, por simplificação elas são chamadas apenas de inspetores.

O NCL-Inspector permite a definição de inspetores através do paradigma imperativo/orientado a objetos através da implementação de classes Java. Também permite a criação de regras declarativamente através da aplicação de transformações em folhas de estilo XML (XSLT – do Inglês *XML Stylesheet Transformation*).

4.2.4 Construtor do mecanismo de regras

O construtor do mecanismo de regras realiza o carregamento de regras em tempo de carga, como descrita na Seção 4.1.4. Na prática, ele funciona como um sistema de plugins. O propósito estabelecido para um plugin é definido por (Fowler et al. 2002) como:

Definição 4.1 *Ligar as classes por configuração ao invés de compilação*[6].

O construtor, ao inicializar a aplicação, procura pelos arquivos de declaração de inspetores (i.e. arquivos de declaração das regras) em um determinado diretório (e.g. "rulelib") e carrega as regras disponíveis à medida que são necessárias. Uma característica importante de se ressaltar é que o construtor utiliza Carregamento Preguiçoso (*Lazy Loading*), isto quer dizer que apenas as regras que estão configuradas pelo usuário para serem utilizadas serão carregadas, poupando assim recursos computacionais.

[6]Do Inglês: Links classes during configuration rather than compilation.

A configuração das regras que serão carregadas e utilizadas é fornecida através do arquivo de configuração do NCL-Inspector. O formato desse arquivo é definido pelo XML Schema apresentado da Listagem 4.1.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <schema xmlns="http://www.w3.org/2001/XMLSchema"
3    targetNamespace="http://www.serg.inf.puc-rio.br/NCLInspector/
       InspectorConfiguration"
4    xmlns:conf="http://www.serg.inf.puc-rio.br/NCLInspector/
       InspectorConfiguration"
5    xmlns:decl="http://www.serg.inf.puc-rio.br/NCLInspector/
       InspectorDeclaration"
6    elementFormDefault="qualified">
7    <import
8      namespace="http://www.serg.inf.puc-rio.br/NCLInspector/
         InspectorDeclaration"
9      schemaLocation="InspectorDeclaration.xsd" />
10
11   <element name="inspector-config" type="
       conf:inspectorConfigType" />
12   <complexType name="inspectorConfigType">
13     <sequence>
14       <element name="inspector" type="conf:inspectorType"
15         minOccurs="1" maxOccurs="unbounded" />
16     </sequence>
17   </complexType>
18   <complexType name="inspectorType">
19     <sequence>
20       <element name="properties" type="decl:propertiesType"
21         minOccurs="0" maxOccurs="1" />
22     </sequence>
23     <attribute name="id" type="ID" use="required" />
24   </complexType>
25 </schema>
```

Listagem 4.1: InspectorConfiguration.xsd – XML Schema do arquivo de configuração do NCL-Inspector.

A Listagem 4.2 mostra uma instância desse arquivo, usado para configurar uma regra que será apresentada na Seção 4.3.3. Esse arquivo possui um elemento raiz `<inspector-config>` (linha 2), aninhado a ele, uma lista de elementos `<inspector>` que possui um atributo `id` cada um (linha 4). Esse atributo é usado para identificar cada inspetor, que será usado pelo construtor do mecanismo de regras para saber qual regra deverá ser usada naquela inspeção. Além disso, pode ser usado para modificar valores das propriedades

das regras (linha 6), como é feito também na Seção 4.3.3.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <inspector-config xmlns="http://www.serg.inf.puc-rio.br/
       NCLInspector/InspectorConfiguration" xmlns:decl="http://www.
       serg.inf.puc-rio.br/NCLInspector/InspectorDeclaration"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.serg.inf.puc-rio.br/
       NCLInspector/InspectorConfiguration ../../rules-engine-
       builder/src/main/xsd/InspectorConfiguration.xsd ">
3
4    <inspector id="coreinspectors.reuse.
         TooManyChildrenInContextOrBody">
5      <properties>
6        <decl:property name="maxChildren" value="2" />
7      </properties>
8    </inspector>
9
10 </inspector-config>
```

Listagem 4.2: inspector-configuration.xml – Exemplo do Arquivo de Configuração do NCL-Inspector.

4.2.5 Mini-*framework* para elaboração de testes de inspetores

O mini-*framework* para elaboração de testes de inspetores possui apenas 3 classes. Apesar de um tamanho reduzido, o *framework* é fundamental para realizar testes das regras de inspeção pois possibilita:

- a padronização dos testes;
- a realização de testes funcionais usando apenas uma pequena porção de código.

Além disso, o *framework* utiliza interfaces fluentes (Fowler Bliki 2010) para especificação do resultado esperado do teste. Uma série de vantagens são obtidas com o uso das interfaces fluentes, algumas das mais importantes que podemos destacar são:

- possibilitar o aumento da legibilidade do código de teste;
- tornar o próprio teste uma documentação do sistema, pois exemplifica o comportamento de uma determinada regra;
- possibilitar que programadores menos experientes possam facilmente entender, modificar e desenvolver os testes.

Para ilustrar o funcionamento do *framework*, podemos utilizar o problema de regiões inconsistentes. Ao declarar um elemento `<region>` contendo os atributos `left`, `right` e `width` juntos, pode causar inconsistências quanto ao tamanho real da `<region>` que será exibida pelo formatador. A inconsistência pode ocorrer se o valor da diferença entre os atributos `left` e `right` não for o mesmo valor especificado em `width` (i.e. `left` − `right` ≠ `width`). O mesmo pode ocorrer se especificarmos os atributos `top`, `bottom` e `height` no mesmo elemento `<region>`. Pela norma, o atributo `right` (ou `bottom`) seria ignorado nesses casos.

No entanto, podemos considerar uma má-prática especificar os valores `left`, `right` e `width` ou `top`, `bottom` e `height` em uma `<region>`. Seria útil se existisse uma regra que avisasse ao desenvolvedor caso ele viesse a cometer esse erro.

Para testar uma regra que inspeciona as violações descritas, usaremos como caso de teste os dois arquivos mostrados na Listagem 4.3 e 4.4. O primeiro (TestCaseError.ncl) é um arquivo NCL contendo elementos `<region>` inconsistentes e o segundo (TestCaseSuccess.ncl) é um arquivo NCL válido.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ncl id="testCaseError">
3    <head>
4      <regionBase>
5        <region id="consistentRegion1">
6          <region id="consistentRegion2" zIndex="3" left="26" top
               ="42" width="250"
7            height="141" />
8          <region id="inconsistentRegion1" width="640" height="
               480" top="1" bottom="1" zIndex="1" />
9          <region id="consistentRegion3" zIndex="2">
10           <region id="inconsistentRegion2" top="1" bottom="2"
                 height="2">
11             <region id="inconsistentRegion3" left="12" right="
                   22" width="32"/>
12           </region>
13         </region>
14       </region>
15     </regionBase>
16   </head>
17
18   <body>
19   ...
20   </body>
```

```
21 </ncl>
```

Listagem 4.3: TestCaseError.ncl – Caso de teste que contém regiões inconsistentes.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ncl id="new_ncl_file">
3    <head>
4      <regionBase>
5        <region id="consistentRegion1">
6          <region id="consistentRegion2" zIndex="3" left="26" top
               ="42" width="250"
7            height="141" />
8          <region id="inconsistentRegion1" width="640" height="
               480" top="1" zIndex="1" />
9          <region id="consistentRegion3" zIndex="2">
10           <region id="inconsistentRegion2" top="1" height="2">
11             <region id="inconsistentRegion3" left="12" width="
                   32"/>
12           </region>
13         </region>
14       </region>
15     </regionBase>
16   </head>
17
18   <body>
19   ...
20   </body>
21 </ncl>
```

Listagem 4.4: TestCaseSuccess.ncl – Caso de teste válido.

Observando o arquivo TestCaseError.ncl (Listagem 4.3) é possível ver que este arquivo NCL possui três regiões inconsistentes: as regiões com `id inconsistentRegion1`, `inconsistentRegion2` e `inconsistentRegion3`. Uma regra que avalie corretamente esse problema deverá, ao receber esse arquivo como entrada, indicar apenas três, e somente três, erros nas linhas 8, 10 e 11. A Listagem 4.5 apresenta o teste que uma regra deverá passar para que seja considerada correta.

```
1 package coreinspectors.layout;
2
3 import java.io.IOException;
4
5 @RunWith(PowerMockRunner.class)
```

```
6  public class InconsistentRegionSizeTest extends
       InspectorTestCase {
7
8    public InconsistentRegionSizeTest() {
9      super("coreinspectors.layout.InconsistentRegionSize");
10   }
11
12   @Test
13   public void testInconsistentRegionSizeFail() throws
         IOException,
14        URISyntaxException {
15
16     inspectFiles("TestCaseError.ncl");
17
18     expectWarning().inLine(8).inColumn(5);
19     expectWarning().inLine(10).inColumn(6);
20     expectWarning().inLine(11).inColumn(7);
21
22     expectNoMoreErrors();
23   }
24
25   @Test
26   public void testInconsistentRegionSizeSuccess() throws
         URISyntaxException,
27        IOException {
28     inspectFiles("TestCaseSuccess.ncl");
29
30     expectNoMoreErrors();
31   }
32 }
```

Listagem 4.5: InconsistentRegionSizeTest.java – Implementação em Java de um caso de teste da regra de verificação de regiões inconsistentes.

Pessoas familiarizadas com JUnit, metodologia de Desenvolvimento Orientado a Testes (TDD - do inglês *Test Driven Development*), ou até mesmo programadores menos experientes, podem perceber que o código é bastante auto-explicativo.

Para utilizar as facilidades fornecidas pelo *framework* é necessário estender a classe `InspectorTestCase`, como visto na linha 6 da Listagem 4.5. Deve ser passado também ao construtor da classe `InspectorTestCase`, o identificador da regra que deseja testar (linha 9). Como o *framework* de teste usa o JUnit, cada caso de teste é representado por um método com a anota-

ção `@Test`, sendo assim, podemos perceber que temos dois casos de teste, representados pelos métodos `testInconsistentRegionSizeFail` (linha 13) e `testInconsistentRegionSizeSuccess` (linha 26).

O primeiro caso de teste, toma como entrada o arquivo `TestCaseError.ncl` apresentado na Listagem 4.3, através da utilização do método `inspectFiles(String ... filenames)` chamado na linha 16. Esse método permite a avaliação de vários arquivos de uma só vez. Isto é particularmente útil para testar uma regra que necessite de uma coleção de arquivos de entrada, ao invés de apenas um.

Como o arquivo possui três violações, a regra deve retornar três e somente três alertas. Isso é feito utilizando uma interface fluente, como mostram as linhas 18 a 20. Em seguida, na linha 22, é verificado se somente três violações foram reportadas. Desta forma, se outras violações forem reportadas por essa regra, ela estaria retornando falsos positivos, o que faria o caso de teste falhar.

Se essa regra informasse a violação como um erro, a validação deveria ser feita usando o método `expectError()` ao invés de `expectWarning()`. Por fim, se caso múltiplos arquivos forem usados em um caso de teste, é possível verficar em qual deles ocorreu a violação, como mostra a listagem 4.6.

```
1 expectWarning().inLine(8).inColumn(5).inFile("file1.ncl");
2 expectError().inLine(18).inColumn(23).inFile("file2.ncl");
```

Listagem 4.6: Verificando violações de diferentes arquivos em um mesmo caso de teste.

4.3 Adicionando novas regras ao NCL-Inspector

O NCL-Inspector foi projetado para ser facilmente extensível. Possui uma arquitetura de plugins que permite a inserção de novas regras sem a necessidade de recompilar todo o seu código. Isto é muito útil devido às razões mencionadas na Seção 4.1.4.

4.3.1 Componentes de uma regra

Uma regra de inspeção é composta basicamente de três arquivos. Um arquivo de declaração do inspetor, um arquivo de propriedades contendo textos que

serão internacionalizados e o arquivo de implementação do inspetor.

O arquivo de declaração do inspetor é um arquivo XML de metadados que possui informações que possibilitam ao construtor carregar um determinado inspetor. A Listagem 4.7 apresenta o XML Schema do formato das informações contidas neste arquivo.

Como observamos na linha 7, o elemento raiz do arquivo é `<inspector-declaration>` que possui elementos filhos obrigatórios `<type>` (linha 10) e `<implementation>` (linha 11); os elementos opcionais `<parent>` (linha 12) e `<properties>` (linha 13 e 14) e o atributo `id`.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <schema xmlns="http://www.w3.org/2001/XMLSchema"
3    targetNamespace="http://www.serg.inf.puc-rio.br/NCLInspector/
       InspectorDeclaration"
4    xmlns:decl="http://www.serg.inf.puc-rio.br/NCLInspector/
       InspectorDeclaration"
5    elementFormDefault="qualified">
6
7    <element name="inspector-declaration" type="
       decl:inspectorDeclarationType" />
8    <complexType name="inspectorDeclarationType">
9      <sequence>
10       <element name="type" type="string" />
11       <element name="implementation" type="string" />
12       <element name="parent" type="string" minOccurs="0"
           maxOccurs="1"/>
13       <element name="properties" type="decl:propertiesType"
14         minOccurs="0" maxOccurs="1" />
15     </sequence>
16     <attribute name="id" type="ID" use="required" />
17   </complexType>
18   <complexType name="propertiesType">
19     <sequence>
20       <element name="property" type="decl:propertyType"
           minOccurs="1"
21         maxOccurs="unbounded" />
22     </sequence>
23   </complexType>
24   <complexType name="propertyType">
25     <simpleContent>
26       <extension base="string">
27         <attribute name="name" type="string" use="required" />
28         <attribute name="value" type="string" use="required" />
29       </extension>
```

```
30      </simpleContent>
31    </complexType>
32  </schema>
```

Listagem 4.7: InspectorDeclaration.xsd – XML Schema do arquivo de declaração de inspetores.

O elemento <type> pode receber os valores "java" e "xsl". O construtor usa o valor desse atributo para descobrir qual o tipo da implementação do inspetor.

O valor do elemento <implementation> depende do valor atribuído ao elemento <type>. Caso <type> contenha o valor "java", o valor de <implementation> deverá ser o nome completamente qualificado de uma classe Java que implemente esse inspetor. Caso o valor de <type> seja "xsl", deverá ser informado o caminho do arquivo XSL que implementa o inspetor.

O elemento <parent> é opcional. Caso ele não esteja presente, é assumido que o core.RootInspector é o pai deste inspetor. Para inspetores XSL, o valor de <parent> é obrigatoriamente core.TreeWalker. Para inspetores Java que implementem a interface MultipleFileInspector, o elemento <parent> não deverá ser especificado. Para inspetores Java que implementem a interface NodeInspector, o valor obrigatório de <parent> é core.TreeWalker.

No futuro, é desejável que, nos casos em que o elemento <parent> não for especificado, ao invés de assumir o valor padrão core.RootInspector, o seu pai poderá ser detectado automaticamente.

O elemento <properties> possui elementos filhos <property> com atributos name e value. Com esses elementos é possível fornecer alguns parâmetros aos inspetores. Suponhamos que exista uma regra que verifique o número de elementos filhos aninhados ao elemento <context> da NCL. Se o número de elementos filhos for maior que um parâmetro maxChildren, esse <context> é considerado como grande ou que possivelmente venha ocorrer problemas de manutenibilidade. A parametrização poderá ser feita declarando algo como <property name="maxChildren" value="20"/>. Isso significa que se o elemento <context> da NCL possuir mais de 20 elementos filhos, ele estará violando essa regra.

Finalmente, o atributo `id` do elemento `<inspector-declaration>` de uma determinada regra deve ser único, pois ele é usado para identificar univocamente uma determinado inspetor dentre todas as presentes na biblioteca de inspetores. A formação do `id` segue uma notação similar a notação de pontos de Java. Caso uma regra possua, por exemplo, o `id="coreinspectors.layout.InconsistentRegionSize"`, o arquivo de declaração de regras deverá estar no diretório `coreinspectors/layout/InconsistentRegionSize.xml` dentro do `CLASSPATH`.

4.3.2 Implementação usando XSL

Para implementar a regra de inspeção que verifica inconsistências em `<region>` (veja Seção 4.2.5), primeiramente crie um projeto no Eclipse de acordo com o indicado na Apêndice B.

Dentro da pasta `src/main/resources` crie uma nova pasta com o nome `coreinspectors` e dentro desta crie uma outra com o nome `layout`. Dentro dessa pasta, crie o arquivo de declaração do inspetor com o nome de `InconsistentRegionSize.xml`. Então, preencha o conteúdo do arquivo de acordo com a Listagem 4.8.

A linha 2 deste arquivo declara um inspetor com seu respectivo `id`. Observe que todo `id` de uma regra do NCL-Inspector deve obedecer a convenção apresentada no último parágrafo da Seção 4.3.1. O elemento `<type>` (linha 6) está indicando que a implementação da regra será feita usando folhas de estilo XML. O elemento `<implementation>` (linha 7) indica a localização da implementação da regra, isto é, onde está o arquivo XSL que realiza de fato a inspeção. Por fim, o elemento `<parent>` (linha 8) com o valor `core.TreeWalker` é obrigatório para regras do tipo XSL.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <inspector-declaration id="coreinspectors.layout.
     InconsistentRegionSize"
3   xmlns="http://www.serg.inf.puc-rio.br/NCLInspector/
       InspectorDeclaration"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5
6   <type>xsl</type>
7   <implementation>coreinspectors/layout/InconsistentRegionSize.
       xsl</implementation>
```

```
8   <parent>core.TreeWalker</parent>
9
10 </inspector-declaration>
```

Listagem 4.8: InconsistentRegionSize.xml – Arquivo de declaração do inspetor de inconsistências de tamanho da região.

O próximo passo é criar um arquivo conforme descrito pelo atributo `<implementation>`, na linha 7, da Listagem 4.8 e então preenchê-lo de acordo com a Listagem 4.9. Este código XSL é o que de fato realiza o trabalho de inspeção de código.

As linhas de 1 a 5 são obrigatórias para especificar qualquer regra em XSL. A linha 2 e 3 declara a versão e o namespace da folha de estilo XML. A linha 4 é necessária pois ela inclui as funções que reportam a linha e a coluna de onde ocorreu uma violação. A linha 5 declara que o namespace do arquivo de entrada, isto é, o NCL, é o namespace padrão da transformação. A declaração do namespace padrão evita redundância ao referenciar elementos do NCL de entrada na folha de estilo, além de prevenir erros na execução da transformação.

As linhas 7 a 11 incluem o elemento raiz do XML de resultado. As transformações usadas pelo NCL-Inspector retornam também um XML contendo as violações e esse elemento raiz `<result>` é obrigatório.

As linhas 12 a 17 realizam de fato a inspeção. Essa regra XSL procura um padrão na AST do documento NCL contendo uma `<region>` possuindo os atributos `top`, `bottom` e `height`. Analogamente, as linhas 18 a 23 realizam a mesma tarefa para os atributos `left`, `right` e `width`. Caso haja o casamento do padrão, o XML informando a violação é impresso como resultado.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet version="2.0"
3    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4      xmlns:insp="java:br.pucrio.inf.serg.nclinspector.engine.
           core.xslt.Functions"
5    xpath-default-namespace="http://www.ncl.org.br/NCL3.0/
       EDTVProfile">
6
7    <xsl:template match="/ncl">
8      <result>
9        <xsl:apply-templates/>
10     </result>
```

```
11    </xsl:template>
12    <xsl:template match="region[@top][@bottom][@height]">
13      <violation line="{insp:line-number()}" column="{
            insp:column-number()}" msgKey="topBottomHeight" severity
            ="warning">
14        <arg><xsl:value-of select="./@id"/></arg>
15      </violation>
16      <xsl:apply-templates/>
17    </xsl:template>
18    <xsl:template match="region[@left][@right][@width]">
19      <violation line="{insp:line-number()}" column="{
            insp:column-number()}" msgKey="leftRightWidth" severity=
            "warning">
20        <arg><xsl:value-of select="./@id"/></arg>
21      </violation>
22      <xsl:apply-templates/>
23    </xsl:template>
24  </xsl:stylesheet>
```

Listagem 4.9: Implementação da regra como uma transformação de folha de estilo XML

O XML de resultado é bem simples. A Listagem 4.10 apresenta o XML Schema do XML de resultado. O elemento `<result>` (linha 7) é o elemento raiz. Ele pode conter zero ou inúmeros elementos `<violation>` (linha 13) que, por sua vez, representam cada violação que foi detectada por essa regra XSL.

Um elemento `<violation>` pode possuir elementos filhos `<arg>`. Esses elementos são importantes, pois permitem a passagem de argumentos do XSL para as mensagens de erro, permitindo assim a criação de mensagens de erro mais precisas, como visto nas linhas 14 e 20 da Listagem 4.9. O elemento `<violation>` também é composto por alguns atributos, que são:

- **severity:** informa a severidade da violação. Pode assumir os valores `warning`, `fatal` e `error` (linhas 17 a 25).
- **line:** informa a linha do documento de entrada onde ocorreu a violação (linha 27).
- **column:** informa a coluna do documento de entrada onde ocorreu a violação (linha 28).
- **msgKey:** informa a chave do arquivo `.properties` onde está definida a mensagem internacionalizada que será exibida para o usuário (linha 29).

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.serg.inf.puc-rio.br/NCLInspector/
     XSLTInspectorResult"
  xmlns:xres="http://www.serg.inf.puc-rio.br/NCLInspector/
     XSLTInspectorResult"
  elementFormDefault="qualified">
  <element name="result" type="xres:resultType" />
  <complexType name="resultType">
    <sequence>
      <element name="violation" type="xres:violationType"
        minOccurs="0" maxOccurs="unbounded" />
    </sequence>
  </complexType>
  <complexType name="violationType">
    <sequence>
      <element name="arg" type="string" minOccurs="0" maxOccurs
         ="unbounded" />
    </sequence>
    <attribute name="severity" use="required">
      <simpleType>
        <restriction base="string">
          <enumeration value="warning" />
          <enumeration value="fatal" />
          <enumeration value="error" />
        </restriction>
      </simpleType>
    </attribute>
    <attribute name="line" use="required" type="unsignedInt" />
    <attribute name="column" use="required" type="unsignedInt"
       />
    <attribute name="msgKey" use="required" type="string" />
  </complexType>
</schema>
```

Listagem 4.10: XSLTInspectorResult.xsd – XML Schema do XML de resultado da inspeção através de XSL.

Como mencionado na Seção 4.3.1, uma regra é composta de três arquivos. Na Listagem 4.8 foi apresentado o arquivo de declaração do inspetor. Na Listagem 4.9 foi apresentado o arquivo de implementação do inspetor. O último arquivo que compõe a regra, o arquivo de propriedades contendo os textos para internacionalização, é apresentado nas listagens 4.11 e 4.12. O primeiro apresenta o arquivo de mensagens de internacionalização para o idioma padrão

(i.e. Inglês) e o segundo apresenta mensagens de internacionalização para o idioma Português.

```
1 leftRightWidth=Specifying attributes left, right and width in
    region with id \"{0}\" could lead to inconsistencies.
2 topBottomHeight=Specifying attributes top, bottom and height in
    region with id \"{0}\" could lead to inconsistencies.
```

Listagem 4.11: InconsistentRegionSize.properties – Arquivo de mensagens de internacionalização para o idioma padrão.

```
1 leftRightWidth=Especificar os atributos left, right and width
    na região com o id \"{0}\" pode levar a inconsistências.
2 topBottomHeight=Especificar os atributos top, bottom and height
    na região com o id \"{0}\" pode levar a inconsistências.
```

Listagem 4.12: InconsistentRegionSize_pt_BR.properties – Arquivo de mensagens de internacionalização para o idioma Português do Brasil.

O arquivo de mensagens de internacionalização, no caso de regras escritas em XSL, deve sempre possuir o mesmo nome do arquivo de declaração do inspetor, porém com extensão `.properties`. De fato, esses arquivos são implementados como um Java Resource Bundle. Portanto, para criar versões em outras línguas para as mensagens, deve-se seguir a convenção adotada pelo Java Resource Bundle como sendo `<Nome do bundle>_<cód. língua>_<cód. país>.properties`. O código da língua e do país obedecem aos padrões ISO 639 e ISO 3166 respectivamente. Para mais detalhes sobre Java Resources Bundle consulte http://java.sun.com/javase/7/docs/api/java/util/ResourceBundle.html.

Os arquivos de properties são na verdade um mapeamento chave/valor (`<chave>=<valor>`). No caso da Listagem 4.11 o exemplo de uma chave é `leftRightWidth` e seu respectivo valor é a mensagem "`Specifying attributes left, right and width in region with id "{0}"could lead to inconsistencies.`".

De volta ao exemplo do arquivo de implementação do inspetor, o atributo `msgKey` do elemento `<violation>`, apresentado nas linhas 13 e 19 da listagem 4.9 sempre deverá fazer referência a alguma chave do arquivo de mensagens para internacionalização. As marcações `{0}`, `{1}`, `{2}` e assim sucessivamente, serão substituídas pelos argumentos passados através do elemento `<arg>`, como mostrado nas linhas 14 e 20 da listagem 4.8. Essa substituição irá ocorrer de

acordo com a ordem em que são declarados no arquivo de implementação do inspetor.

Com os três arquivos prontos, está finalizada a implementação da regra de inspeção. Agora basta seguir os passos descritos no Apêndice B.3 para realizar a implantação da regra diretamente na versão de linha de comando do NCL-Eclipse.

Para ilustrar como funciona em detalhes a implementação de inspetores através de XSL, considere o arquivo NCL apresentado na Listagem 4.13. Este arquivo contém três elementos `<region>` inconsistentes. O código XSL da Listagem 4.9, quando receber esse arquivo NCL, irá retornar o XML de resultado apresentado na Listagem 4.14. Então, o mecanismo de regras irá fazer o *parsing* deste XML, apresentando as violações em forma de uma API Java. Posteriormente a implementação em linha de NCL-Inspector exibe as violações em forma de caracteres no terminal de linha de comando, como mostra a Listagem 4.15.


```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ncl id="new_ncl_file">
3    <head>
4      <regionBase>
5        <region id="consistentRegion1">
6          <region id="consistentRegion2" zIndex="3" left="26" top
               ="42" width="250"
7            height="141" />
8          <region id="inconsistentRegion1" width="640" height="
               480" top="1" bottom="1" zIndex="1" />
9          <region id="consistentRegion3" zIndex="2">
10           <region id="inconsistentRegion2" top="1" bottom="2"
                 height="2">
11             <region id="inconsistentRegion3" left="12" right="
                   22" width="32"/>
12           </region>
13         </region>
14       </region>
15     </regionBase>
16   </head>
17
18   <body>
19     <port id="pIntro" component="intro" />
20     <media id="intro" src="pictures/inicio.gif" descriptor="
         dsIntro" />
21   </body>
```

```
22 </ncl>
```

Listagem 4.13: InconsistentRegionSizeTest1.ncl – Arquivo de entrada NCL contendo regiões inconsistentes.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <result xmlns:insp="java:br.pucrio.inf.serg.nclinspector.engine
       .core.xslt.Functions">
3    <violation line="8" column="5" msgKey="topBottomHeight"
         severity="warning">
4      <arg>inconsistentRegion1</arg>
5    </violation>
6    <violation line="10" column="6" msgKey="topBottomHeight"
         severity="warning">
7      <arg>inconsistentRegion2</arg>
8    </violation>
9    <violation line="11" column="7" msgKey="leftRightWidth"
         severity="warning">
10     <arg>inconsistentRegion3</arg>
11   </violation>
12 </result>
```

Listagem 4.14: Resultado retornado pela transformação XSL.

```
1  NCL-Inspector is starting...
2  Loading rules...
3  NCL-Inspector is inspecting files...
4
5  SUMMARY:
6
7  WARNING in file InconsistentRegionSizeTest1.ncl line 8 column 5
       - Especificar os atributos top, bottom and height na região
       com o id "inconsistentRegion1" pode levar a inconsistências
      .
8  WARNING in file InconsistentRegionSizeTest1.ncl line 10 column
      6 - Especificar os atributos top, bottom and height na
      região com o id "inconsistentRegion2" pode levar a
      inconsistências.
9  WARNING in file InconsistentRegionSizeTest1.ncl line 11 column
      7 - Especificar os atributos left, right and width na região
       com o id "inconsistentRegion3" pode levar a inconsistências
      .
10
11  3 potential problems found.
```

Listagem 4.15: Resultado retornado pela versão de linha de comando do NCL-Inspector.

4.3.3 Implementação usando Java

Suponhamos que desejemos criar uma regra que detecte quando um contexto está demasiadamente grande. O conceito de contextos, representado pelo elemento `<context>`, criado pelos projetistas da linguagem NCL serve para modularizar uma aplicação NCL. Portanto, é desejável que os elementos que tenham maior relação entre si estejam agrupados em contextos. Uma analogia que pode ser feita é que os contextos estão para a NCL, assim como as classes estão para uma linguagem orientada a objetos.

Pelos meios de computação atuais, é muito difícil criar uma regra que detecte quando um contexto está grande demais com exatidão. O conceito de contexto grande não é bem definido e depende muito do julgamento do desenvolvedor. Dessa forma, esta regra deve ser baseada em alguma métrica ou heurística. Podemos usar a heurística que já foi apresentada na Seção 4.3.1 para implementar essa regra, isto é, podemos contar a quantidade de elementos filhos diretos (exceto `<context>`) que um contexto possui.

O primeiro passo para se criar uma regra é criar o arquivo de declaração do inspetor. O formato das informações deste arquivo, bem como a utilização destas pelo NCL-Inspector foram apresentados na subseção 4.3.2. Em virtude disso, o foco será nas diferenças do arquivo de declaração do inspetor para Java em relação a implementação para uma regra em XSL.

A Listagem 4.16 apresenta o arquivo de declaração do inspetor implementado usando Java, explicitado pelo elemento `<type>`. Como a regra será implementada em Java, o valor do elemento `<implementation>` representa o nome completamente qualificado da classe Java que implementa essa regra (linha 7). Por fim, esse inspetor irá fazer o casamento de padrões na árvore XML, logo, o elemento `<parent>` deverá possuir o valor `core.TreeWalker` (linha 8).

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <inspector-declaration id="coreinspectors.reuse.
    TooManyChildrenInContextOrBody"
3   xmlns="http://www.serg.inf.puc-rio.br/NCLInspector/
      InspectorDeclaration"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5
6   <type>java</type>
```

```
7   <implementation>br.pucrio.inf.serg.nclinspector.
        coreinspectors.reuse.TooManyChildrenInContextOrBody</
        implementation>
8   <parent>core.TreeWalker</parent>
9
10 </inspector-declaration>
```

Listagem 4.16: TooManyChildrenInContextOrBody.xml - Arquivo de declaração de inspetor implementado em Java.

Para realizar o casamento de padrões na árvore XML, o inspetor deverá implementar a interface `NodeInspector`. Porém, a classe `AbstractNodeInspector` possui métodos úteis para a maioria das implementações de inspetores baseados em casamentos de padrões XML. Por isso, iremos estendê-la ao invés de implementar a interface `NodeInspector`, como podemos ver na linha 15 da listagem 4.17.

A implementação da regra se dá de fato no método `visit(XmlObject)`. Entretanto outras linhas de código também possuem informações relevantes. Por exemplo, o construtor da classe, implementado nas linhas 19 a 21, deve obrigatoriamente receber um parâmetro do tipo `String` contendo o `id` do inspetor. Além disso, chamar o construtor do pai também é obrigatório.

O método `getInspectedTypes()` (linhas 28 a 30) informa ao mecanismo de regras (mais especificamente ao `TreeWalkerInspector`) quais elementos o inspetor, implementado através desta classe, está interessado em inspecionar. Isso é feito retornando um *array* de classes `Class` do tipo Java, cada qual representando uma entidade NCL. Para facilitar descobrir qual tipo Java que representa uma determinada entidade NCL, a convenção usada pelo NCL-Inspector é sempre concatenar o nome da entidade NCL com "Type". Desta forma, para a entidade NCL Body teremos a classe `BodyType` e para a entidade NCL Context teremos a classe `ContextType`.

No caso deste inspetor, a tarefa de inspeção constitui em avaliar se um elemento `<context>` ou `<body>` é muito grande seguindo a heurística de contagem de elementos filhos. A contagem, porém, não deve levar em consideração os elementos filhos `<context>`, pois justamente o uso deste elemento é o que a regra quer disseminar.

Como mencionado, o método `visit(XmlObject)` (linhas 33 a 49) é o que de fato implementa o inspetor. Quando o `TreeWalkerInspector` encontrar um elemento `<context>` ou `<body>` na árvore, `visit(XmlObject)` será invocado. Esse método irá receber como parâmetro um objeto `ContextType` ou `BodyType`, assinado como um `XmlObject`. A classe `XmlObject` é a superclasse comum a todos os objetos Java usados para representar alguma entidade NCL.

A variável `nonContextChildNodesCount` conta os filhos que não são do tipo `<context>` do elemento representado pelo parâmetro `xmlObject`. Na linha 35, são recuperados todos os filhos de `xmlObject`. Os nós filhos recuperados incluem atributos, elementos, comentários, entre outros. Precisamos então separar os nós, pois a nossa heurística leva apenas em consideração os elementos XML diferentes de `<context>`. Isso é feito no `for` que se inicia na linha 37.

Neste `for` é recuperado cada nó filho de `xmlObject` e verficado se ele é um nó do tipo elemento XML (linha 40). Na linha 41, é testado também se o nó não é um elemento do tipo `<context>`. Caso essas duas condições sejam verdadeiras, é contabilizado que um elemento filho que não é contexto foi encontrado, incrementando a variável `nonContextChildNodesCount` (linha 42). Finalmente, se a quantidade de elementos filhos diferentes de contexto é maior (linha 44) do que o limite estabelecido por `maxChildren` (linha 17), uma violação é reportada (linha 45).

```
1  package br.pucrio.inf.serg.nclinspector.coreinspectors.reuse;
2
3  import java.text.MessageFormat;
4
5  import org.apache.xmlbeans.XmlBeans;
6  import org.apache.xmlbeans.XmlObject;
7  import org.w3c.dom.Node;
8  import org.w3c.dom.NodeList;
9
10 import br.org.ncl.ncl30.edtvProfile.BodyType;
11 import br.org.ncl.ncl30.edtvProfile.ContextType;
12 import br.pucrio.inf.serg.nclinspector.rulesengine.core.
       AbstractNodeInspector;
13 import br.pucrio.inf.serg.nclinspector.rulesengine.core.
       Violation.ViolationSeverity;
14
15 public class TooManyChildrenInContextOrBody extends
       AbstractNodeInspector {
16
17   private int maxChildren = 20;
```

```
18
19  public TooManyChildrenInContextOrBody(String id) {
20    super(id);
21  }
22
23  public void setMaxChildren(int value) {
24    this.maxChildren = value;
25  }
26
27  @Override
28  public Class<?>[] getInspectedTypes() {
29    return new Class<?>[] { BodyType.class, ContextType.class
        };
30  }
31
32  @Override
33  public void visit(XmlObject xmlObject) {
34    int nonContextChildNodesCount = 0;
35    NodeList childNodes = xmlObject.getDomNode().getChildNodes
        ()
36
37    for (int i = 0; i < childNodes.getLength(); i++) {
38      Node childNode = childNodes.item(i);
39
40      if (childNode.getNodeType() == Node.ELEMENT_NODE &&
41          !isContext(childNode)) {
42        nonContextChildNodesCount++;
43
44        if (nonContextChildNodesCount > maxChildren) {
45          addViolation(xmlObject, getMessage(xmlObject),
              ViolationSeverity.WARNING);
46          break;
47        }
48      }
49    }
50  }
51
52  private String getMessage(XmlObject xmlObject) {
53
54    if (xmlObject.schemaType().equals(ContextType.type))
55      return getContextViolationMessage((ContextType) xmlObject
          );
56    else if (xmlObject.schemaType().equals(BodyType.type))
57      return getBodyViolationMessage();
58    else
59      return "No message.";
60  }
```

```
61
62    private String getContextViolationMessage(ContextType context
          ) {
63
64      return MessageFormat.format(getString("
            contextViolationMessage"),
65          context.getId(), maxChildren);
66    }
67
68    private String getBodyViolationMessage() {
69      return MessageFormat.format(getString("bodyViolationMessage
            "),
70          maxChildren);
71    }
72
73    private boolean isContext(Node childNode) {
74      XmlObject child = XmlBeans.nodeToXmlObject(childNode);
75      return child != null && child.schemaType().equals(
            ContextType.type);
76    }
77
78  }
```

Listagem 4.17: TooManyChildrenInContextOrBody.java - Arquivo de implementação (Classe Java) do inspetor.

O inspetor emite mensagens de violação diferentes dependendo do tipo do elemento que ocorreu a violação. Isto é implementado pelo método `getMessage(XmlObject)` (linhas 52 a 60), que recupera mensagens de violação diferentes para os elementos `<context>` (linhas 54 e 55) ou `<body>` (linhas 56 e 57).

O método `getString(String)` pertence à classe `BaseInspector` e abstrai o mecanismo de internacionalização. Este método recupera a mensagem internacionalizada seguindo uma convenção idêntica a usada pelas regras implementadas em XSL, como mostrado na Seção 4.3.2. A Listagem 4.18 apresenta o arquivo de mensagens para internacionalização usado para implementar esse inspetor.

```
1 bodyViolationMessage=Creating a large <body> element would
    reduce the reusability and legibility of a NCL application.
    The <body> element should not have more than {0} non-context
     children.
2 contextViolationMessage=Creating large <context> elements would
    reduce the reusability and legibility of a NCL application.
```

```
The <context> with id \"{0}\" should not have more than {1}
non-context children.
```

Listagem 4.18: TooManyChildrenInContextOrBody.properties - Arquivo de mensagens para internacionalização.

4.4 Usando o NCL-Inspector como uma biblioteca

Além da interface de linha de comando, o NCL-Inspector possui uma API Java que permite o seu uso a partir de outros programas. A própria interface de linha de comando é implementada utilizando a API Java. No futuro, essa API poderá facilitar a implementação de uma integração com o NCL-Eclipse, por exemplo.

A interface Java é simples de ser usada e possui um bom nível de abstração. Isto permitiu implementar toda a interface de linha de comando em apenas uma classe. A listagem 4.19 apresenta essa classe.

```
1  package br.pucrio.inf.serg.nclinspector.ui.console;
2
3  import java.io.File;
4  import java.text.MessageFormat;
5  import java.util.Collection;
6
7  import br.pucrio.inf.serg.nclinspector.builder.
       XmlInspectorFactory;
8  import br.pucrio.inf.serg.nclinspector.engine.core.
       RootInspector;
9  import br.pucrio.inf.serg.nclinspector.engine.core.Violation;
10
11 public class Main {
12
13   private static final String REPORT_STRING = "{0} in file {1}
         line {2} column {3} - {4}";
14
15   public static void main(String [] args) {
16
17     if(args.length < 2 || args.length > 3) {
18       System.err.println("Usage: ncl-inspector-console <
             configFile> <inspectedFile> [rulelib path]");
19       System.exit(1);
20     }
21
22     System.out.println("NCL-Inspector is starting...");
```

```
23      System.out.println("Loading rules...");
24      XmlInspectorFactory factory = new XmlInspectorFactory(new
            File(args[0]));
25
26      if(args.length == 3)
27        factory.setRuleLibPath(args[2]);
28
29      RootInspector rootInspector = factory.createRootInspector()
            ;
30
31      System.out.println("NCL-Inspector is inspecting file...");
32      rootInspector.inspectFiles(new File[]{new File(args[1])});
33
34      Collection<Violation> violations = rootInspector.getContext
            ().getViolations();
35
36      System.out.println("");
37      System.out.println("SUMMARY:");
38      System.out.println("");
39
40      for (Violation violation : violations)
41        System.out.println(MessageFormat.format(REPORT_STRING,
              violation.getSeverity(), violation.getFile().getName()
              , violation.getLine(), violation.getColumn(),
              violation.getMessage()));
42
43      System.out.println("");
44
45      MessageFormat mf = new MessageFormat("{0,choice,0# No
            potential problem|1#One potential problem|1< {0,number,
            integer} potential problems} found.");
46      Object [] summaryArgs = new Object[] {new Integer(
            violations.size())};
47      System.out.println(mf.format(summaryArgs));
48
49    }
50
51  }
```

Listagem 4.19: Classe que implementa a versão de linha de comando do NCL-Inspector.

A linha 24 cria a fábrica responsável por compor os objetos da classe `Inspector` de forma hierárquica. Os arquivos contendo os inspetores, por convenção padrão, deverão estar no diretório "rulelib". Caso contrário, poderá ser especificado um outro diretório, como mostra a linha 27.

O RootInspector é criado a partir do método `createRootInspector()` da classe `XmlInspectorFactory` (linha 29). A linha 32 chama o método que irá realizar a inspeção nos arquivos passados como parâmetro.

As linhas a seguir são responsáveis por imprimir o relatório da inspeção na tela. Na linha 34 é recuperada a lista de violações que foram encontradas nessa inspeção. O laço das linhas 40 e 41 imprime de fato as violações encontradas na inspeção. As linhas 45 a 47 mostra uma mensagem informando quantas violações foram encontradas ao todo.