5.
A ferramenta SDiff

A ferramenta *SDiff* (Syntactical Diff) é uma ferramenta de comparação de documentos que implementa o mecanismo de comparação apresentado no último capítulo. A implementação realizada neste trabalho contém o componente que trata a sintaxe dos documentos escritos na linguagem de programação C++.

Ao longo deste capítulo apresentaremos detalhes de implementação da ferramenta, começando com algumas informações técnicas, arquitetura do software, seus pontos de extensão, estatísticas extraídas do código-fonte e uma breve explicação de como foi realizado o controle da qualidade ao longo do desenvolvimento.

5.1
Informações técnicas

Esta ferramenta foi implementada utilizando a linguagem de programação C++, com auxílio do framework de desenvolvimento Qt [TROLLTECH, 2008]. O ambiente de desenvolvimento utilizado foi o Microsoft Visual Studio 2008 [MICROSOFT, 2008] no ambiente Windows 7 [MICROSOFT, 2009], porém não existem restrições para utilizar outro ambiente. O código-fonte da ferramenta é multiplataforma, podendo ser compilado também nos ambientes Linux e Mac OSX, onde é mais comum o compilador GCC [STALLMAN, 1986]. O controle de versão utilizado foi o Mercurial [MERCURIAL, 2006], conhecido também como HG.

Durante a implementação do componente responsável pela interpretação da sintaxe da linguagem de programação C++ foi utilizada a biblioteca ANTLR [Parr, 2007], que, a partir de uma especificação da sintaxe escrita em uma linguagem proprietária, é capaz de gerar arquivos fonte na linguagem de programação C++, que fornecem primitivas para interpretar um conteúdo escrito na sintaxe especificada.

Também foi utilizado o *framework* CppUnit [FEATHERS, 2002] para auxiliar a criação dos testes de unidade.

PUC-Rio - Certificação Digital Nº 0812610/CA

5.2
O Framework DMF

Durante o desenvolvimento da ferramenta foi criado o *framework* DMF (*Document Management Framework*) que representa o núcleo da aplicação. Este *framework* é responsável pelas operações realizadas sobre a estrutura *Document*, que podem ser enumeradas como:

1. Carregar um documento a partir do conteúdo textual de um arquivo.
2. Linearizar um documento para a forma textual.
3. Comparar dois documentos.

O framework apresenta dois pontos de extensão: (1) representante de um tipo de documento específico; e (2) mecanismo de comparação de elementos sintáticos. Ambos os pontos de extensão serão apresentados nas subseções seguintes.

A arquitetura deste framework é apresentada na Figura 27 a seguir.

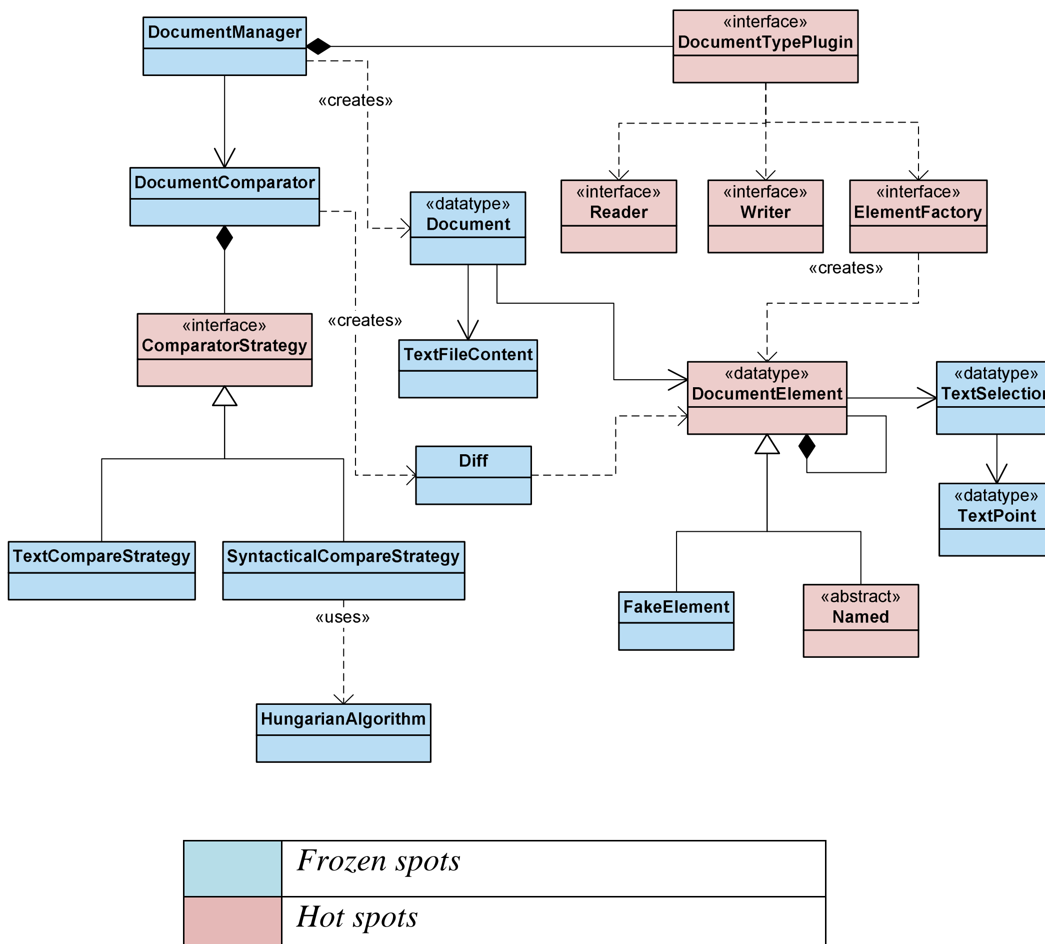

Figura 27 – Diagrama de classes da arquitetura do DMF.

PUC-Rio - Certificação Digital Nº 0812610/CA

A explicação deste diagrama seguirá a ordem do fluxo de dados em uma comparação de documentos. Inicialmente ressaltamos que para cada uma das estruturas apresentadas no capítulo anterior - *Document*, *DocumentElement* e *Diff* - foi implementada uma classe com seu respectivo nome, como pode ser conferido no diagrama apresentado. Também aproveitamos para comentar que a propriedade *seleção* dos elementos é representada pela classe *TextSelection*, composta por duas instâncias de *TextPoint*, representando os pontos inicial e final de uma *seleção*.

O procedimento de comparação recebe como entrada o caminho para os dois arquivos que serão comparados. A partir da extensão dos arquivos é identificado o tipo de documento, e, conseqüentemente, qual *plugin* será utilizado. Este mecanismo de identificação não é o ideal para usuários que utilizam extensões fora do padrão, porém, esta é uma limitação que pode ser facilmente eliminada com a criação de um recurso que permita o usuário cadastrar as extensões que utiliza.

Os *plugins* são os componentes que representam um tipo de documento, e são implementados por aplicações usuárias do *framework*. Cada *plugin* é uma fábrica de instâncias *Reader*, *Writer* e *ElementFactory*. Explicando brevemente: a classe *Reader* deve fornecer uma primitiva para converter um arquivo na sua representação canônica; a classe *Writer* fornece uma primitiva que realiza a operação inversa, transformando a forma canônica em uma representação textual; e a classe *ElementFactory* é uma fábrica de elementos sintáticos, derivados de *DocumentElement*.

Continuando, após encontrar o *plugin* que trata do tipo de documento geramos as estruturas que serão comparadas. Para isso criamos uma instância de *Reader*, a partir do *plugin* escolhido, que transformará o conteúdo de cada documento em sua forma canônica. Esta por sua vez, é processada gerando uma composição de objetos *DocumentElement*, que representa a estrutura hierárquica do documento a ser comparado. Uma observação importante é que a instância de *Reader* poderia retornar diretamente a composição de objetos, porém optamos por esta abordagem para que o resultado, uma estrutura XML, fosse genérico suficiente para ser entregue a diferentes tipos de ferramenta, não apenas as que trabalhassem especificamente com a composição de objetos, como é o caso do comparador de documentos.

PUC-Rio - Certificação Digital Nº 0812610/CA

Cada elemento desta composição de objetos é um *DocumentElement*, gerado pela instância de *ElementFactory* também criada pelo *plugin* escolhido. Esta fábrica fornece uma primitiva que recebe um nó XML e retorna a composição de objetos descrita por ele. Como exemplo, para um nó XML do tipo *Identifier* é criado um objeto da classe *Identifier*, que implementa os comportamentos esperados para este tipo de elemento. As subclasses para cada tipo de elemento devem ser fornecidas pelo *plugin*, porém sugere-se que sejam organizadas em uma linha de produto [CLEMENTS e NORTHROP, 2002], de forma que um tipo de elemento possa ser reutilizado em mais de um tipo de documento (Ex: *Class*, *Function*, *Identifier*, *Expression*, *VariableDeclaration*, etc.).

O framework implementa duas subclasses de *DocumentElement*. A primeira, *FakeElement*, foi apresentada no capítulo anterior e suas instâncias representam elementos *dummy*, criados a fim de representar o elemento modificado na resposta do algoritmo de comparação textual. A outra subclasse é a classe abstrata *Named*, que permite a criação de objetos nomeáveis, utilizados pela heurística de emparelhamento baseada em nomes.

Voltando ao processo de comparação, o próximo passo é construir a estrutura Document, que mantém uma referência para o primeiro nó da árvore de elementos construída no passo anterior. Esta estrutura também guarda uma instância da classe *TextFileContent*, que representa o conteúdo textual do documento em um formato onde cada caractere é indexado por [linha, coluna]. Isso facilita a extração de trechos de código pois geralmente os limites são definidos por estruturas *TextPoint*, obtidas em estruturas *TextSelection*.

O passo final é entregar as duas instâncias *Document* ao componente de comparação, cuja interface de comunicação é a classe *DocumentComparator*. Esta classe implementa o método de comparação recursivo apresentado na Figura 13, e, mantém uma lista de estratégias de comparação (interface *ComparatorStrategy*), que é um dos pontos de extensão deste *framework*. Internamente o *framework* implementa as duas estratégias de comparação descritas no capitulo anterior: comparação textual e comparação sintática, com as classes *TextCompareStrategy* e *SyntacticalCompareStrategy*, respectivamente.

As subseções seguintes descreverão os dois principais pontos de extensão deste *framework*.

PUC-Rio - Certificação Digital Nº 0812610/CA

5.2.1
Ponto de extensão: tipo de documento

O *framework* suporta diferentes tipos de documentos através da criação de *plugins*. Na prática, um *plugin* é uma implementação da interface *DocumentTypePlugin* para o tipo específico que se deseja prover suporte. Esta interface é apresentada na Figura 28.

```
1   class DocumentTypePlugin
2   {
3   public:
4           DocumentTypePlugin() {}
5           virtual ~DocumentTypePlugin() {}
6
7           virtual QString contentType() const = 0;
8           virtual QList<QString> parsableFileExtensions() const = 0;
9           virtual Reader * createReader() const = 0;
10          virtual Writer * createWriter() const = 0;
11          virtual ElementFactory * createElementFactory() const = 0;
12  };
13
```

Figura 28 – Interface *DocumentTypePlugin*.

O método *contentType* deve informar o tipo do conteúdo em um formato textual, como por exemplo “C/C++”. E, o método *parsableFileExtensions* deve informar a lista de extensões de arquivos que guardam este tipo de conteúdo. Como mencionado anteriormente, esta classe é uma fábrica de objetos *Reader*, *Writer* e *ElementFactory*, que, respectivamente, são criados com os métodos *createReader*, *createWriter* e *createElementFactory*. Porém, estes tipos também são interfaces e o usuário do *framework* deve criar classes que as implemente com o funcionamento esperado para o tipo de conteúdo tratado.

5.2.2
Ponto de extensão: algoritmo de comparação de elementos

Outro ponto de extensão é o algoritmo de comparação de elementos, que fornece uma interface flexível a fim de viabilizar a criação de novas estratégias de comparação de elementos por parte do usuário. A interface que deve ser implementada para criar uma nova estratégia de comparação é apresentada na Figura 29.

PUC-Rio - Certificação Digital Nº 0812610/CA

```
1	class ComparatorStrategy
2	{
3	public:
4		ComparatorStrategy() {}
5		virtual ~ComparatorStrategy() {}
6	
7		virtual const QString & name() const = 0;
8		virtual void compare(
9			const DocumentElement & oldElement,
10			const DocumentElement & newElement,
11			QList<Diff*> & diffList,
12			double & numDiffs,
13			double & numPossibleDiffs) = 0;
14	};
15	
```

Figura 29 – Interface *ComparatorStrategy*.

O método *name* deve ser implementado de forma que forneça um identificador único, que será referenciado pelos elementos que desejarem ser comparados por esta estratégia.

O método *compare*, deve ser implementado com o novo algoritmo de comparação de elementos, onde seus parâmetros *oldElement* e *newElement* são os elementos a serem comparados, a lista *diffList* é utilizada para retornar as diferenças identificadas, e as variáveis *numDiffs* e *numPossibleDiffs*, passadas como referência, são utilizadas para retornar, respectivamente, o número de diferenças identificadas e o número total de possíveis diferenças.

5.3 Visualização das diferenças

A fim de exibir para o usuário final as diferenças identificadas em uma comparação, criamos uma interface para a aplicação construída com auxílio das ferramentas de interface gráfica do *framework* Qt. Esta interface pode ser visualizada na Figura 30.

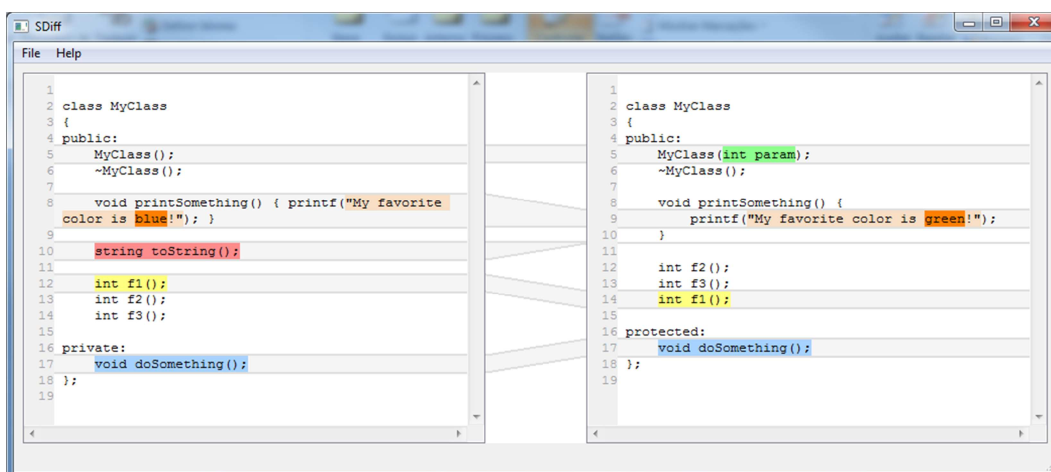

Figura 30 – Interface gráfica da ferramenta SDiff.

PUC-Rio - Certificação Digital Nº 0812610/CA

A interface é extremamente simples, permitindo que o usuário abra um par de arquivos, um ao lado do outro, e estes sejam comparados utilizando o mecanismo previamente apresentado. Em seguida o resultado da comparação, uma estrutura *Diff*, é percorrido recursivamente e notifica para cada painel as diferenças encontradas, que por sua vez, modifica a cor de fundo do elemento alterado, de acordo com a semântica da diferença. O painel é capaz de identificar os limites a serem marcados a partir da *seleção* de cada elemento. A semântica das cores utilizadas para realizar estas marcações é:

Cor	Tipo
(verde)	Inserção
(rosa)	Remoção
(laranja)	Modificação
(amarelo)	Movimentação
(azul)	Alteração no contexto
(bege)	Modificação no elemento

Cada cor equivale a um tipo de diferença. Os tipos *Inserção*, *Remoção*, *Modificação* e *Movimentação* são mapeadas diretamente da estrutura *Diff*. O tipo *Alteração no contexto* corresponde a uma modificação de uma das propriedades de contexto. E, como diferenças identificadas pelo mecanismo de comparação textual não respeitam necessariamente os limites sintáticos, criamos o recurso de visualização chamado *seleção estendida*, que, para cada diferença identificada, calcula os limites sintáticos do elemento no qual ela está imersa. Este limite encontrado é marcado com a cor representada por *Modificação no elemento*. Observe que em diferenças identificadas pelo mecanismo de comparação sintática esta *seleção estendida* é sempre idêntica a *seleção* do elemento modificado, logo, não aparece.

5.4
Estatísticas gerais do código-fonte

Esta seção apresentará estatísticas gerais extraídas do código-fonte da aplicação, a fim de expor mais informações sobre a aplicação criada. Toda medição apresentada foi realizada sobre a porção de código produzida pelo autor desta dissertação, descartando eventuais bibliotecas open-source utilizadas. Estas informações são apresentadas nas tabelas a seguir, e as que forem relevantes para o controle de qualidade do software serão referenciadas na seção seguinte.

PUC-Rio - Certificação Digital Nº 0812610/CA

Número de linhas total	13.7 KLOC
Número de linhas destinadas a código executável	9.1 KLOC
Número de linhas destinadas a comentários	2.1 KLOC
Número de linhas em branco	2.5 KLOC
Porcentagem de linhas destinadas a código executável	66 %
Porcentagem de linhas destinadas a comentários	16 %
Porcentagem de linhas em branco	18%

Tabela 1 – Estatísticas sobre o número de linhas do código-fonte.

Número de linhas executáveis destinadas ao controle da qualidade	2.3 KLOC
Número de linhas destinadas a assertivas executáveis	0.5 KLOC
Número de linhas executáveis destinadas a testes de unidade	1.8 KLOC
Porcentagem de linhas destinadas ao controle da qualidade	25 %
Porcentagem de linhas destinadas a assertivas executáveis	5 %
Porcentagem de linhas destinadas a testes de unidade	20 %
Número de módulos de teste	6
Número de casos de teste	102

Tabela 2 – Estatísticas sobre o controle da qualidade

Número de classes	108
Número de revisões no controle de versão	131

Tabela 3 – Outras estatísticas.

O número de linhas destinadas ao controle da qualidade é calculado somando o número de linhas destinadas a assertivas com o número de linhas de código executável dos módulos de teste de unidade. O número de revisões no controle de versão equivalem ao número de artefatos projetados, implementados e aprovados, com seus respectivos casos teste implementados.

PUC-Rio - Certificação Digital Nº 0812610/CA

5.5 Controle da qualidade

Antes de iniciar a fase de desenvolvimento a arquitetura do software foi cuidadosamente projetada, as fronteiras dos componentes foram bem definidas e as tecnologias candidatas a serem usadas foram testadas em protótipos.

Durante o desenvolvimento foi aplicada a técnica *Test-driven development* (TDD) [Beck, 2003]. Cada componente foi implementado separadamente com auxílio de testes de unidade. A ordem de implementação dos componentes foi *bottom up*:

1. Interpretador da linguagem de programação C++ com apoio da biblioteca ANTLR.
2. Fábrica de elementos sintáticos da linguagem de programação C++.
3. Serializador de estruturas representando documentos na linguagem de programação C++.
4. *Plugin* representante da linguagem de programação C++.
5. *Framework* DMF.
6. Estratégia de comparação textual.
7. Estratégia de comparação sintática.
8. Interface gráfica da aplicação *SDiff*.

Durante a construção destes componentes foram criados seis módulos de teste de unidade, totalizando 102 casos de teste, como foi verificado na Tabela 2. Além do controle da qualidade baseado em testes, o código foi produzido utilizando [STAA, 2000] contratos de software, assertivas executáveis e estruturas auto-verificáveis, representando 5% do número total de linhas executáveis.

Medindo o número de linhas destinadas ao controle a qualidade, ou seja, a soma do número de linhas destinadas a assertivas com o número total de linhas empregadas nos testes de unidade, temos que **um quarto** do código escrito foi dirigido para este propósito.

PUC-Rio - Certificação Digital Nº 0812610/CA