UMA IDE PARA O SABLECC

Trabalho de Conclusão de Curso

Engenharia da Computação

Gabriel Baracuhy Macedo Melo
Orientador: Prof. Joabe Jesus

Universidade de Pernambuco
Escola Politécnica de Pernambuco
Graduação em Engenharia de Computação

GABRIEL BARACUHY MACEDO MELO

UMA IDE PARA O SABLECC

Monografia apresentada como requisito parcial para obtenção do diploma de Bacharel em Engenharia de Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Recife, Dezembro 2011.

Escola Politécnica de Pernambuco
Graduação em Engenharia de Computação
Projeto de Final de Curso

POLI
ESCOLA POLITÉCNICA
DE PERNAMBUCO

MONOGRAFIA DE FINAL DE CURSO

Avaliação Final (para o presidente da banca)*

No dia 21 de Dezembro de 2011, às 13:00 horas, reuniu-se para deliberar a defesa da monografia de conclusão de curso do discente GABRIEL BARACUHY MACEDO MELO, orientado pelo professor Joabe Bezerra de Jesus Júnior, sob título Uma IDE para o SABLECC, a banca composta pelos professores:

Gustavo Henrique Porto de Carvalho

Joabe Bezerra de Jesus Júnior

Após a apresentação da monografia e discussão entre os membros da Banca, a mesma foi considerada:

☒ Aprovada ☐ Aprovada com Restrições* ☐ Reprovada

e foi-lhe atribuída nota: 10 (DEZ)

*(Obrigatório o preenchimento do campo abaixo com comentários para o autor)

O discente terá 7 dias para entrega da versão final da monografia a contar da data deste documento.

Gustavo Henrique

GUSTAVO HENRIQUE PORTO DE CARVALHO

Joabe Bezerra de Jesus Júnior

JOABE BEZERRA DE JESUS JÚNIOR

Dedico esse trabalho aos meus pais.

Agradecimentos

Agradeço a Deus, primeiramente, por ter me abençoado, dado-me forças para superar todos os obstáculos e me capacitado a chegar até aqui.

Aos meus Pais e família, que tanto amo, por me apoiarem e me incentivarem incondicionalmente por todos esses anos.

Ao professor Joabe, demais professores e colegas que estiveram envolvidos no projeto e os quais tive o prazer de conviver e trabalhar.

Aos amigos, que sempre estiveram presente em todos esses momentos ao meu lado dando-me força e incentivo.

Resumo

A falta de um ambiente de desenvolvimento integrado (IDE), que forneça suporte ao gerador de parser SableCC, utilizado na especificação de linguagens de programação, torna o processo de desenvolvimento de uma nova linguagem difícil e improdutivo. A fim de auxiliar a explorar ao máximo a produtividade dos desenvolvedores que utilizam o SableCC e apoiar a construção de uma gramática mais fácil de ser escrita, entendida e mantida, esse trabalho propõe a construção de uma IDE para o SableCC a partir da extensão da plataforma Eclipse, ou seja, a construção de plugins. Além do desenvolvimento de um editor para o SableCC, o trabalho tem intenção de montar bases para implementar um processo de geração de metamodelo para uma linguagem especificada. A partir desse processo, seria possível que um ambiente de desenvolvimento integrado pudesse ser gerado, automaticamente, especificamente para a linguagem desenvolvida com o gerador de Parser. Portanto, a maior contribuição deste trabalho é a implementação do editor proposto que representa a IDE com recursos fundamentais para que vise uma rápida prototipagem. Dentre os recursos destacam-se a formatação rica (*syntax highlighting*), auxílio à escrita (*auto-complete*) e visualizador de árvore sintática (*outline*).

Abstract

The absence of an Integrated Development Environment (IDE), that supports the SableCC parser generator, to specify programming languages makes this process difficult and less productive. In order to help to maximize developers' productivity, that uses SableCC, and support the construction of a grammar easier to write, understand and maintain, this work proposes the construction of an IDE for SableCC from the extension of the Eclipse platform, in other words, the construction of a plugin. In addition to developing an editor for the SableCC, this work intends to build bases to implement a process for generating metamodel for a specified language. From this process it would be possible to automatically generate an integrated development environment, specifically for the language developed with the parser generator. Therefore, the major contribution of this work is the implementation of the editor proposed that represents the IDE with the critical resources that aims the quick prototyping of a new programming language. Among the resources we feature the rich formatting (syntax highlighting), assistance writing (auto-complete) and syntax tree viewer (outline).

Sumário

Capítulo 1 Introdução **1**

1.1 Objetivos da ferramenta proposta 3

1.2 Metodologia 3

1.2.1 Natureza e abordagem 3

1.2.2 Objetivos 3

1.2.3 Procedimentos técnicos 3

1.3 Construção da ferramenta 4

1.4 Resultados esperados 4

1.5 Organização do trabalho 4

Capítulo 2 Estado da arte **5**

2.1 Editores de programas 5

2.1.1 Edição sensível à linguagem 6

2.1.2 Automação de escrita 6

2.1.3 Suporte à navegação (*Browsing)* 7

2.1.4 Auxílio à documentação 7

2.1.5 Suporte à refatoramento 7

2.2 O Eclipse Plataform 8

2.2.1 The Platform Runtime 10

2.2.2 OSGI 10

2.2.3 Workbench 11

2.2.4 Workspace 12

2.2.5 Plugins 12

2.2.6 Editores de programas no Eclipse 13

2.3 Geradores de compiladores 14

2.3.1 Parser LL (*left-left)* 14

2.3.2 Parser LR, SLR e LALR 14

2.3.3 SableCC 15

Capítulo 3 SableCCDEV 18

3.1 Arquitetura da IDE 18

3.2 SableCC Plugin 19

3.3 SableCCBuilder Plugin 19

3.4 SableCCEditor Plugin 20

3.4.1 Padrão Model-View-Controller 22

3.4.2 Documents 23

3.4.3 DocumentProvider 24

3.4.4 SourceViewer e SableCCSourceViewerConfiguration 25

3.4.5 A classe SableCCEditor 26

3.4.6 Particionamento 27

3.4.7 Formatação rica (*Syntax Highlight*) 29

3.4.8 Automação de escrita (*auto complete*) 32

3.4.9 Folding 34

3.4.10 Auxílio à navegação (*browsing)* 35

3.4.11 Visualizador de árvore sintática (*content outline)* 36

3.4.12 Marcador de erro (*error marker)* 38

Capítulo 4 Estudo de caso 39

4.1 Mini Basic 39

4.1.1 Projeto Mini Basic na SableCC IDE 40

Capítulo 5 Conclusão 45

Referências 47

Lista de Figuras

Figura 1. Ambiente ANTLRWorks. 1

Figura 2. Edição sensível à linguagem. Visual Studio 2008 e Eclipse à direita. 6

Figura 3. Automatização de escrita. Visual Studio 2008 e Eclipse à direita. 6

Figura 4. Suporte à navegação. Visual Studio 2008 e Eclipse à direita. 7

Figura 5. Auxílio à documentação. Visual Studio 2008 e Eclipse à direita. 7

Figura 6. Suporte à refatoramento. Visual Studio 2008 e Eclipse à direita...............8

Figura 7. Visão geral da arquitetura do Eclipse 10

Figura 8. Workbench do Eclipse 11

Figura 9. Trecho de um arquivo plugin.xml...12

Figura 10. Família de parsers LR 15

Figura 11. Exemplo de gramática de uma linguagem descrita usando o SableCC. 17

Figura 12. Dependência entre os subsistemas...18

Figura 13. Tela da página de preferências do SableCCBuilder.............................20

Figura 14. Declaração do editor no arquivo plugin.xml..21

Figura 15. Diagrama de classes do SableCCEditor Plugin....................................22

Figura 16. Comunicação entre componentes JFace no modelo MVC...................23

Figura 17. Definição e configuração do esquema de particionamento 28

Figura 18. Definição de tokens e regras de tipos de partições para o SableCC....29

Figura 19. Definição de tokens e regras para o SableCC 31

Figura 20. Página de preferência de cores do SableCCEditor 31

Figura 21. Diagrama de classes dos componentes do destaque de sintaxe 32

Figura 22. Trecho de código do método *getContentAssistant()* definindo *processors*33

Figura 23. Diagrama de classes dos componentes do destaque de sintaxe 34

Figura 24. Diagrama de classes do recurso *folding*....35

Figura 25. Diagrama de classes do recurso navegação (*browsing)*....36

Figura 26. Diagrama de classes do recurso visualizador de árvore sintática (*outline)* 37

Figura 27. Diagrama de classes do recurso de marcação de erros (*error marker)*38

Figura 28. Trecho da gramática do Mini Basic 39

Figura 29. Tela de criação de uma gramática para o SableCC através de *wizard*40

Figura 30. Tela de definição da gramática....41

Figura 31. Tela de construção da seção *Lexer*....41

Figura 32. Demonstração do recurso de automação de escrita. 42

Figura 33. Demonstração do recurso de marcação de erro. 42

Figura 34. Demonstração do recurso de *folding.*....43

Figura 35. Demonstração do recurso de navegação (*browsing).* 43

Figura 36. Demonstração do recurso de visualizador da árvore sintática (*browsing).*44

Figura 37. Tela de preferência de definição de diretório de saída dos arquivos....44

Lista de Símbolos e Siglas

APIs – *Application Programming Interface*

BNF – Backus Naur *Form* (Formalismo de Backus-Naur)

DFA – *Deterministic Finite Automaton*

EBNF – *Extended Backus Normal Form*

EMF – Eclipse *Modeling Framework*

IDE – *Integrated Development Environment*

LALR – *Look Ahead Left Right*

LL – *Left-Left*

LR – *Left-Right*

MVC – *Model View Controller*

OSGI – *Open Services Gateway initiative framework*

PDE – *Plugin Development Environment*

RCP – *Rich Client Platform*

SLR – *Simple Left-Right*

SWT – *Standard Widget Toolkit*

UI – *User Interface* (Interface com o Usuário)

Capítulo 1

Introdução

Geradores de parsers são ferramentas capazes de criar parsers, ou analisadores sintáticos, a partir de uma especificação formal (comumente a notação BNF ou EBNF) de uma linguagem de programação. Dentre as opções de geradores de parsers disponíveis pode-se citar o ANTLR, JavaCC, Bison, Yacc, Packrat e SableCC (Gagnon, 1998). Cada um desses geradores implementam algum tipo de algoritmo de parser (Aho e Ulmann, 1995), como LL, LR, SLR e LALR, que irá determinar a capacidade de criar linguagens mais ou menos restritas.

Dentre todos os geradores citados, o SableCC (SableCC, 2011), é um compilador de compiladores (CC) que implementa o algoritmo de parser LALR (Aho e Ulmann, 1995), sendo mais poderoso que parsers que implementam o algoritmo SLR, LR(0) puro ou LL(k) como é o caso do ANTLR. Desde sua versão 3.2, o SableCC tem evoluído no suporte a gramáticas mais ricas e algoritmos mais elaborados como o suporte ao LALR(k), estando atualmente na versão 4.1 beta. Entretanto, o SableCC não possui uma IDE funcional e amigável como é o caso do AntLRWorks, (ANTLRWorks, 2011): uma IDE dedicada ao ANTLR, Figura 1 (ANTLR, 2011). Apenas iniciativas simples como a descrita em (Adam, 2004) podem ser encontradas.

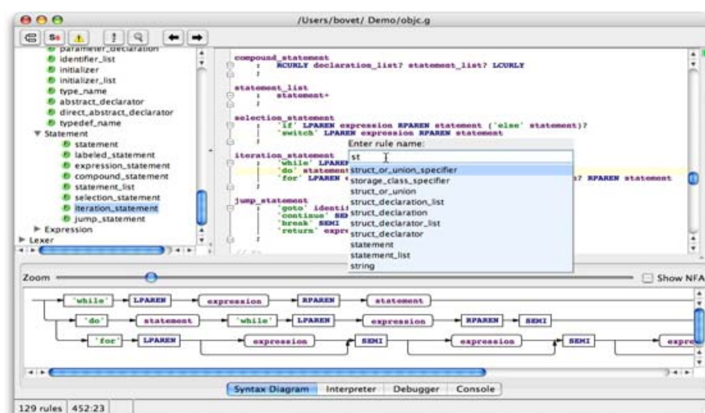

Figura 1. Ambiente ANTLRWorks.

O ANTLR possui ambientes completos como, por exemplo, o ANTLRWorks, que possui um editor visual rico e que fornece uma ampla quantidade de recursos (ANTLRWorks, 2011). Dentre esses recursos destacam-se: a formatação rica (*syntax highlighting*), o suporte à refatoramento (*refactoring*), a navegação (*browsing*), o visualizador de árvore sintática (*outline*) e o depurador (*debugger*).

A falta de um ambiente integrado de desenvolvimento (IDE) para o SabbleCC, ou qualquer outra linguagem, dificulta o trabalho de especificação da mesma, tornando o processo mais complexo e desgastante. Uma IDE serve não só como uma ferramenta para auxiliar e explorar ao máximo a produtividade dos desenvolvedores mas também auxiliar na construção de um código mais legível (Emmerich, 2008) e (Dart et al, 1987).

A IDE Eclipse (Beck e Gamma, 2003) é um ambiente integrado de desenvolvimento de código aberto e uma das ferramentas de desenvolvimento mais populares, sendo amplamente utilizada. Ela é uma plataforma extensível para construção de IDE baseada em plugins (Eclipse Foundation, 2011a).

O Eclipse Plug-in Development Environment (PDE) (Beck e Gamma, 2003) é um plugin que facilita bastante a construção de novos plugins através do fornecimento de ferramentas de edição e wizards. Com o PDE é possível automatizar e abstrair boa parte do desenvolvimento da ferramenta (editor) tornando o trabalho de criação, desenvolvimento, debug e testes mais produtivos. Para simplificar o desenvolvimento e aumentar a produtividade o Eclipse já fornece um framework, JFace Text (Guojie, 2005), que permite o desenvolvimento de editores de texto.

Motivado por este cenário, este trabalho propõe a construção de um ambiente (IDE) que seja capaz de lidar com gramáticas escritas para o SableCC e, também, almeja tratar as linguagens definidas pelos parsers gerados para essas gramáticas. Neste segundo caso, o processo de geração de metamodelos das linguagens permitiria a geração automática de IDEs para essas linguagens.

1.1 Objetivos da ferramenta proposta

O trabalho propõe desenvolver um plugin para o eclipse que permita estender o editor de desenvolvimento para dar suporte ao SableCC e se possível desenvolver a idéia de metamodelagem com o SableCC. A idéia é preparar uma base para que o ambiente possua uma ferramenta capaz de gerar um metamodelo da linguagem especificada através do SableCC e a partir desse metamodelo gerar um editor/IDE específico para a linguagem definida.

1.2 Metodologia

1.2.1 Natureza e abordagem

Esse trabalho fará uso de uma metodologia aplicada, na qual estaremos preocupados em solucionar problemas reais do uso do SableCC, isto é, criação produtiva de um compilador. Definimos que esse trabalho, quanto á abordagem, tem um caráter qualitativo, pois não há o interesse direto em quantificar os elementos da pesquisa, e sim, entender, interpretar fenômenos de forma clara e estabelecer comparações a fim de aplicar conhecimento e produzir uma ferramenta.

1.2.2 Objetivos

Quanto aos objetivos da metodologia adotada, o trabalho tem natureza exploratória, pois, envolve esforço de levantamento bibliográfico, além de ser fortemente orientado à análise de exemplos, visando ajudar o entendimento de fenômenos, e estar apoiado em estudos da área.

1.2.3 Procedimentos técnicos

O nosso estudo reúne características de pesquisa bibliográfica, documental e pesquisa na internet. Bibliográfica por ser constituída a partir de material como livros e artigos de periódicos. Documental por fazer uso de conhecimento oriundo de análise de relatório. Por último, pesquisa na internet pelo fato de utilizar portais de periódicos e páginas web como fonte de conhecimento.

1.3 Construção da ferramenta

Usando o PDE e o JFaceText será possível alcançar as características necessárias para a ferramenta proposta nesse trabalho. A ferramenta proposta será composta de três plugins e está organizada da seguinte maneira: o plugin SableCC, responsável pelo encapsulamento da implementação do framework SableCC; o SableCCEditor que é o editor que representa o ambiente de desenvolvimento propriamente dito; e o SableCCBuilder responsável por compilar (gerar código Java para) as gramáticas especificadas no SableCCEditor;

1.4 Resultados esperados

Inspirado na idéia de Adam (2004) e no ANTLRWorks, espera-se como resultado um plugin capaz de ser integrado à plataforma Eclipse. Tal plugin fornecerá um ambiente integrado de desenvolvimento para o SableCC com recursos semelhantes aos disponíveis no ANTLRWorks.

1.5 Organização do trabalho

Este trabalho está organizado e distribuído em cinco capítulos. O primeiro trata da introdução, ressaltando a motivação do seu desenvolvimento e abordando os objetivos propostos, assim como, a metodologia utilizada. O segundo capítulo descreve o estado da arte, fazendo um levantamento das principais idéias envolvidas no desenvolvimento da ferramenta proposta por este trabalho. O terceiro capítulo descreve os detalhes da construção da nossa ferramenta, tratando detalhes de arquitetura, implementação e bibliotecas envolvidas. O quarto capítulo descreve um exemplo de uso que apresenta a ferramenta e seus recursos sendo aplicados a partir da especificação de uma gramática para o SableCC. Por fim, o capítulo cinco conclui o trabalho ressaltando a importância da ferramenta e o que se espera como trabalhos futuros para dar continuidade ao projeto.

Capítulo 2
Estado da arte

Neste capítulo estão descritos os conceitos, elementos e estudos relacionados à compreensão e desenvolvimento desse trabalho. Aqui serão abordadas as idéias de editores de programa, a descrição e estrutura do Eclipse Platfom (Eclipse Object Technology International, 2003), assim como sua arquitetura de plugins e funcionamento de seus editores. Ainda será abordada a idéia de metamodelagem no Eclipse e, finalmente, geradores de compiladores e as particularidades do SableCC.

2.1 Editores de programas

Os editores de programas são os principais componentes dos ambientes de desenvolvimento integrado (IDE). Tais ambientes tem a finalidade de tornar mais produtivo o processo de desenvolvimento de software através de ferramentas que apóiem a construção do programa (Emmerich, 2008) e (Dart et al, 1987).

Editores podem ter como entrada variados tipos de conteúdo. O editor textual de uma IDE, por exemplo, edita o código fonte do programa e torna o processo de codificação mais eficiente devido aos seus requisitos essenciais. Normalmente esses requisitos são: edição sensível à linguagem, automatização de escrita (A*uto-complete*), suporte à navegação (*browser),* auxílio à documentação e suporte ao refatoramento.

A integração de ferramentas em um ambiente apóia o desenvolvedor nas atividades de construção e manutenção. A integração também auxilia a produção de um código mais inteligível e faz com que tarefas como criação, revisão, análise, transformação e execução de artefatos sejam otimizadas. Dependendo da complexidade do sistema sendo descrito realizar essas tarefas manualmente pode tornar-se inviável.

2.1.1 Edição sensível à linguagem

O requisito de edição sensível à linguagem faz com que o editor torne-se fortemente ligado à sintaxe da linguagem. Torna possíveis recursos como coloração do código, realçando erros sintáticos ou palavras chaves, relatórios detalhados de erros e organização (formatação) de código como identação automática.

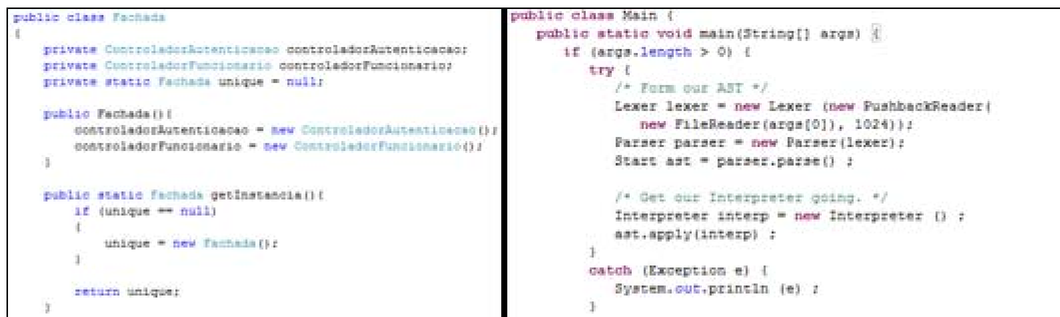

Figura 2. Edição sensível à linguagem. Visual Studio 2008 e Eclipse à direita.

2.1.2 Automação de escrita

A automação de escrita é bastante popular e presente nos editores mais modernos. É um recurso popularmente conhecido como *auto-complete* (smart) que apresenta possíveis sugestões de complementos para o usuário. As sugestões podem ser, por exemplo, referência a variáveis, métodos, palavras chaves ou tipos. A eficiência do auto-complete depende da inteligência implementada no seu desenvolvimento.

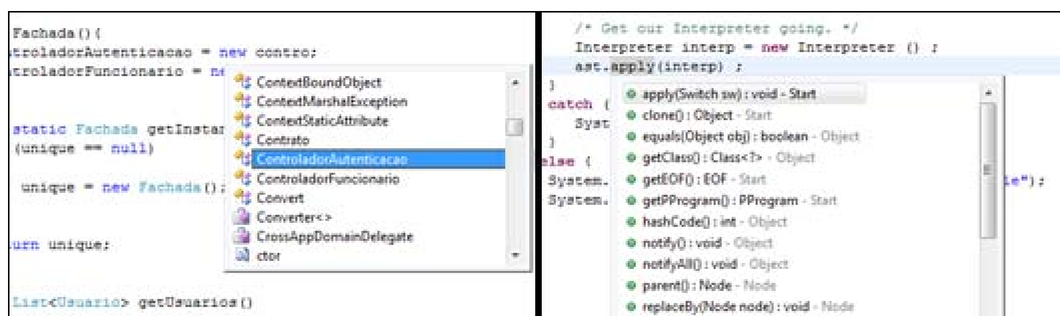

Figura 3. Automatização de escrita. Visual Studio 2008 e Eclipse à direita.

2.1.3 Suporte à navegação (*Browsing)*

É um recurso que permite o usuário navegar através do conteúdo que está sendo editado. O suporte a navegação, ou suporte a *browser*, auxilia a localizar declarações, referências e outras definições. Ajuda, portanto, o entendimento de um código mais complexo.

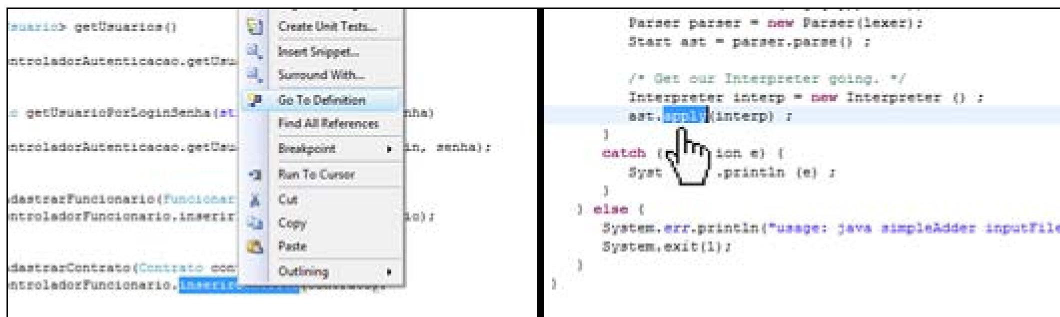

Figura 4. Suporte à navegação. Visual Studio 2008 e Eclipse à direita.

2.1.4 Auxílio à documentação

É um recurso que permite auxiliar na geração de documentação. O Javadoc, recurso de auxílio à documentação para a linguagem Java, permite criar documentação a partir de alguma marcação inserida em um formato específico acima de métodos e classes.

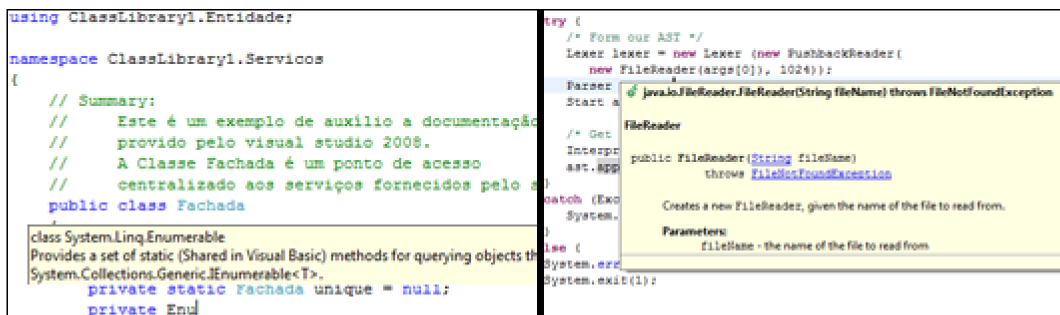

Figura 5. Auxílio à documentação. Visual Studio 2008 e Eclipse à direita.

2.1.5 Suporte à refatoramento

É um recurso que auxilia na automatização de atividades de manutenção de código. Normalmente suporta ações como reorganização e renomeação de termos/identificadores (variáveis).

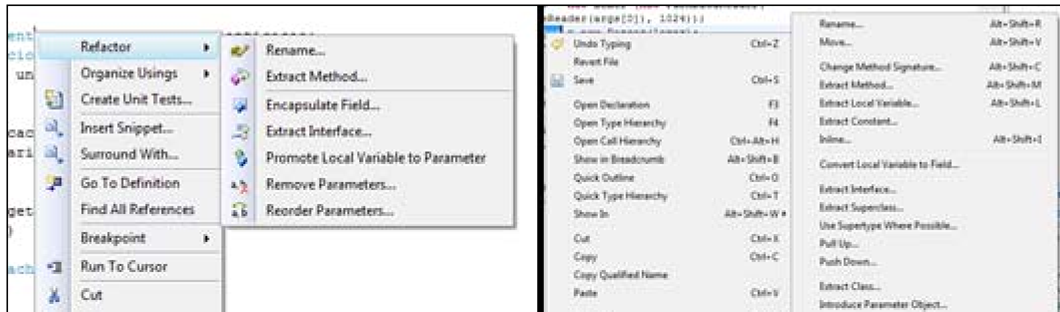

Figura 6. Suporte à refatoramento. Visual Studio 2008 e Eclipse à direita.

2.2 O Eclipse Plataform

A década de 90 foi marcada pelo crescimento fenomenal da Internet. Consequentemente, as aplicações sofreram impactos, evoluíram e tornaram-se mais sofisticadas. Essa complexidade, das aplicações, fez gerar mais artefatos e os desenvolvedores tiveram que lidar com variados arquivos e conteúdo. Por exemplo, as aplicações web são facilmente compostas por centenas de arquivos diferentes (HTML, Web Services, Php, Java). Para lidar com diferentes tipos de artefatos foi necessário recorrer a variadas ferramentas e editores, ou por falta de algum recurso específico ou pelo fato de outra ferramenta fornecer melhor o recurso. Logo se percebeu que dificilmente essas ferramentas trabalhariam integradas, além da dificuldade de superar a curva de aprendizado de cada uma delas. Essa situação trazia problemas de produtividade e desgaste para os desenvolvedores e empresas (Jim, Scott e Kehn et all, 2005).

A iniciativa do Eclipse surgiu em meio a uma pequena revolução com o intuito de fornecer um conjunto de ferramentas que trabalhem juntas em um ambiente produtivo e integrado. Para tanto, foi definida uma plataforma que pudesse integrar unidades funcionais chamadas plugins (Beck e Gamma, 2003). O Eclipse platform é essencialmente uma plataforma de integração de ferramentas inicialmente projetada para ser um ambiente de desenvolvimento para Java e, também, uma comunidade *open source*. Até novembro de 2001 o Eclipse era patenteado pela IBM (Erickson,

2001), depois teve seu código fonte aberto. Atualmente, está sob os cuidados de uma organização, sem fins lucrativos, chamada Eclipse Fundation.

A IDE Eclipse é um dos ambientes de desenvolvimento mais populares do mundo. É capaz de suportar uma ampla quantidade de linguagens de programação e ferramentas de suporte ao desenvolvimento. Grande parte dessa popularidade é justificada pelo fato da comunidade Eclipse contribuir para a constante atualização das ferramentas existentes e a criação de novos recursos a serem integrados à sua plataforma.

Tem-se como requisitos fundamentais da plataforma Eclipse a capacidade de fornecer um ambiente que suporte: a construção de ferramentas para apoiar as atividades de desenvolvimento, trabalhar com os mais diferentes tipos de arquivos, a integração de ferramentas que trabalhem com diferentes tipos de conteúdo, ambientes de desenvolvimento baseados em GUI (*Graphical User Interface*) e baseados em texto e a execução nos mais variados sistemas operacionais (Erickson, 2001).

Uma das atribuições da plataforma Eclipse é fornecer meios para que as suas ferramentas possam se integrar de forma correta. Para isso existem mecanismos que garantem que as ferramentas a serem integradas sigam algumas regras. Através de APIs, interfaces bem definidas, blocos de construção genéricos e pontos de extensão esses mecanismos atuam para facilitar a construção e integração de novas aplicações.

A plataforma é composta de uma estrutura base, conforme Figura 7, onde diferentes empresas e pessoas podem criar ferramentas que atendam às suas necessidades, integrá-las com ferramentas de terceiros e conseguir que elas trabalhem de forma correta juntas. Isso permite ao usuário customizar a IDE de acordo com as suas necessidades (Jim, Scott e Kehn et all, 2005) .

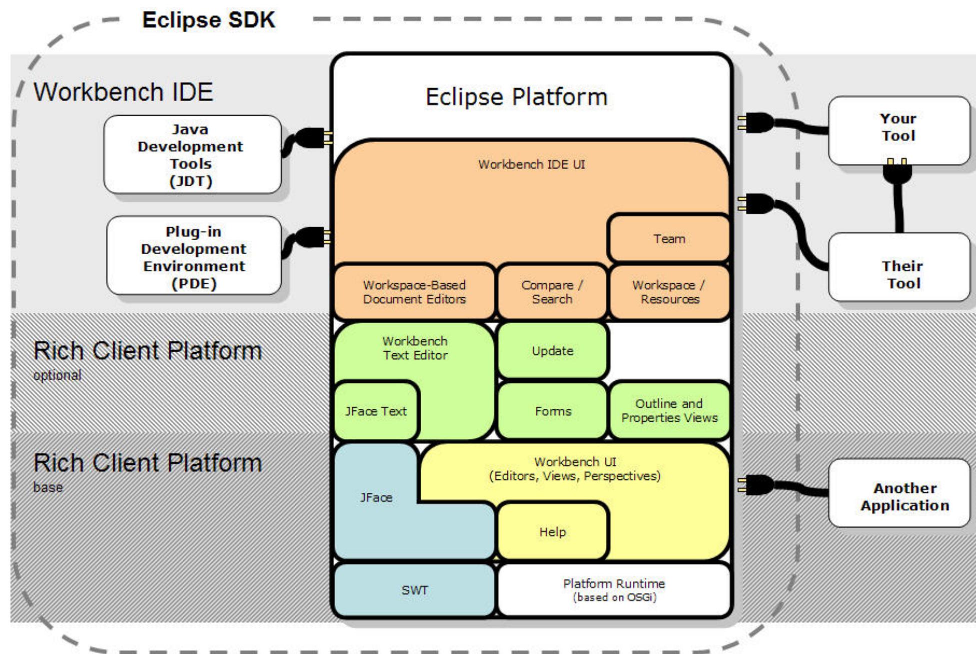

Figura 7. Visão geral da arquitetura do Eclipse

2.2.1 The Platform Runtime

O *platform runtime* (Eclipse Object Technology International, 2003) é o componente responsável por carregar, integrar e executar os plugins do Eclipse. Cada plugin possui um arquivo chamado *plugin.xml* que contém especificações e declaração de pontos de extensão. Ao iniciar o Eclipse, o núcleo *runtime* tenta descobrir os plugins disponíveis e analisa o conteúdo do arquivo *plugin.xml.* A análise das especificações do arquivo do plugin é armazenado em um repositório em memória chamado *plug-in registry*.

O *runtime é* responsável por gerenciar os plugins da plataforma e cria a instância de cada um deles a partir da API *plug-in registry*. O *plataform runtime* detecta e registra qualquer problema que possa vir a ocorrer durante o carregamento de algum plugin, como pontos de extensões inexistentes. As informações dos registros desses procedimentos podem ser acessadas a partir da API da plataforma.

2.2.2 OSGI

O OSGI (*Open Services Gateway initiative framework*) é um framework de especificação o qual o *plataform runtime* do Eclipse é baseado (Oliveira, 2011).

Mantido pelo OSGI Alliance, consórcio mundial de inovadores de tecnologia que desenvolve e mantém uma especificação eficiente para desenvolver softwares baseados em componentes utilizando a tecnologia Java (OSGi Alliance, 2011).

O OSGI defende a redução da complexidade de software a partir da modularização. Sua especificação facilita a modularização do software em componentes chamados *bundles* (plugins na linguagem do Eclipse). O Equinox é a implementação de todos os aspectos da especificação OSGI. É através do Equinox que o *runtime* do Eclipse faz uso do OSGI.

Atualmente o OSGI se encontra na versão *4.3,* lançado em abril de 2011, e faz parte do *service plataform release 4.* O Eclipse 3.0 é totalmente compatível com a especificação R3.0. Já a implementação OSGI na versão do Eclipse 3.1.2 e a partir da versão 3.2 é baseada na versão R4.0 do framework (OSGi Alliance, 2011).

2.2.3 Workbench

É o componente da plataforma onde o trabalho de desenvolvimento será realizado. O Workbench (Eclipse, 2005) torna possível a interação do usuário com as ferramentas da IDE Eclipse. É nesse ambiente que está concentrado a barra de menu, barra de ferramentas, perspectivas, *views* e o editor. É um framework de interface com usuário poderoso e que pode ser estendido. O Workbench é o objeto raiz da plataforma de interface do usuário do Eclipse e é formado pela API SWT e JFace (Guojie, 2005). O Workbench do Eclipse pode ser visualizado na Figura 8.

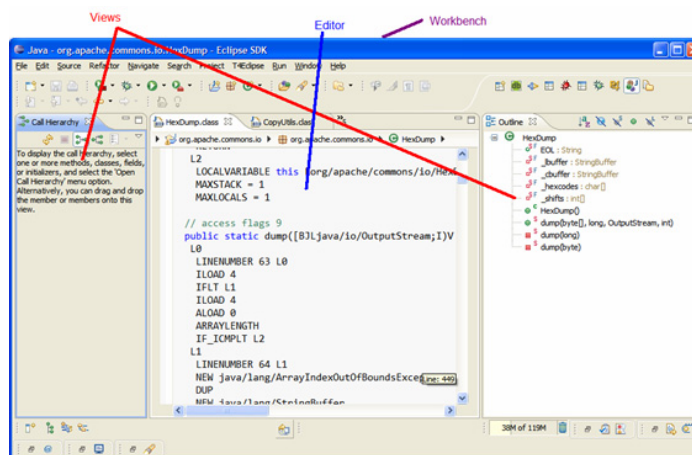

Figura 8. Workbench do Eclipse

2.2.4 Workspace

O Workspace (Eclipse Object Technology International, 2003) engloba um conjunto de projetos e armazena as configurações das ferramentas utilizadas para esses projetos. Ele possui todos os recursos necessários para construir e testar a aplicação.

2.2.5 Plugins

Um plugin é a menor unidade funcional que pode ser desenvolvido e distribuído individualmente. Uma funcionalidade do Eclipse pode ser fornecida por um único plugin simples ou ser composta por um conjunto de plugins, normalmente a situação de ferramentas complexas. Um exemplo de plugin é o framework JUnit que é utilizado para realizar automatização de testes (Beck e Gamma, 2003).

Os plugins são compostos de uma definição declarativa e uma implementação. A parte declarativa é constituída de um arquivo *manifest* baseado em XML. A implementação é codificada na linguagem Java. A estrutura básica e essencial que compõe um plugin é representada através de um diretório contendo o arquivo XML *manifest*, recursos (como imagens), e o código Java.

No arquivo *plugin.xml* é declarado os pontos de extensões necessários para o plugin. Esses pontos são interconexões com outros plugins e através deles podem-se realizar contribuições para a plataforma. Por exemplo, em nosso projeto declaramos o ponto de extensão *org.eclipse.ui.preferencePages* conforme se pode visualizar na Figura 9. Isso significa que está sendo realizada uma contribuição para a página de preferência do Eclipse.

```
<extension
      point="org.eclipse.ui.preferencePages">
   <page
         class="br.poli.ecomp.ads.sablecceditor.preferences.SableCCPreferencePage"
         id="br.poli.ecomp.ads.sablecceditor.preferences.SableCCPreferencePage"
         name="SableCCEditorPreferences">
   </page>
</extension>
```

Figura 9. Trecho de um arquivo plugin.xml

Após identificar os pontos de extensão necessários e criar o arquivo plugin.xml, inicia-se a etapa de implementação. É necessário codificar a funcionalidade através da implementação da interface esperada. Existe um plugin no Eclipse chamado Plug-in Development Environment que facilita bastante a construção de novos plugins através do fornecimento de ferramentas de edição e wizards.

2.2.6 Editores de programas no Eclipse

O Editor do Eclipse é a parte principal onde é realizado o desenvolvimento. Corresponde a uma parte do *workbench* o qual permite o usuário editar um objeto. É no editor textual que é realizado a atividade de codificação, mas no Eclipse encontra-se a disposição não só editores de texto. Existem vários plugins que fornecem ambientes de edição visual baseados em *swing*, por exemplo.

O workbench da plataforma Eclipse fornece pontos de extensão que torna possível construir novas funcionalidades e editores para a IDE. Plugins precisam utilizar o ponto de extensão *org.eclipse.ui.editors* (Eclipse Foundation, 2011b) para adicionar editores ao workbench. Na declaração do ponto de extensão, no arquivo XML, são especificados os tipos de arquivos reconhecidos pelo editor. É possível ainda definir o atributo contributorClass que determina uma classe que implementa ações para as barras de ferramentas quando o editor estiver ativo.

Os editores do Eclipse funcionam de maneira fortemente integrada ao workbench da plataforma e seus plugins. O objeto de entrada do editor é representado pela interface IEditorInput (Deva, 2006) e pode ser um documento ou qualquer outro tipo de conteúdo.

O Eclipse fornece um framework sofisticado, JFace Text, que permite criar editores de textos com recursos típicos do editor padrão da plataforma. A estrutura da plataforma permite estender o editor textual e através do framework é possível criar editores para qualquer linguagem de programação.

2.3 Geradores de compiladores

Geradores de compiladores são programas que a partir de uma especificação formal de uma linguagem, utilizada como entrada, irão gerar como saída compiladores que empregam algum algoritmo de *parsing*. Os algoritmos dos analisadores sintáticos (*parsers)* irão determinar a capacidade de expressão das construções sintáticas da linguagem a ser especificada.

O método *top-down* constrói árvores da raiz para as folhas e o método *bottom-up*, que ao contrário do anterior, constrói árvores começando das folhas até o topo. Ambos os métodos *top-down* e *bottom-up* são os mais comumente utilizados e os mais viáveis em termos de produção.

2.3.1 Parser LL (*left-left)*

O algoritmo de parsing LL é um analisador sintático descendente (*top-down)* que lê a entrada de texto da esquerda para a direita e produz derivações mais à esquerda. É um algoritmo fácil de implementar mas reconhece uma classe restrita de gramáticas. A gramática não pode ter recursões à esquerda, caso contrário, o parser entra em *loop* infinito. Há, entretanto, melhorias neste algoritmo como a usada pelo ANTLR, isto é, o LL(k), onde k é o lookahead (Gagnon, 1998).

2.3.2 Parser LR, SLR e LALR

São os algoritmos ascendentes ou *bottom-up.* Os *parsers* LR executam a leitura da esquerda para direita e realizam derivações reversas. A família de parser LR, Figura 10 são as mais eficientes do método bottom-up, pois conseguem alcançar um amplo subconjunto de classes de gramáticas livre de contexto (Puntambekar, 2010).

A popularidade dos parser LR está no fato de poderem reconhecer a maioria das linguagens de programação que podem ser especificadas a partir de gramáticas livres de contexto. Conseguem reconhecer um número de classes de gramáticas maior que analisadores preditivos. Além disso, conseguem detectar erros mais

eficientemente e utilizam técnicas de *shift-reduce* sem backtrack (Puntambekar, 2010).

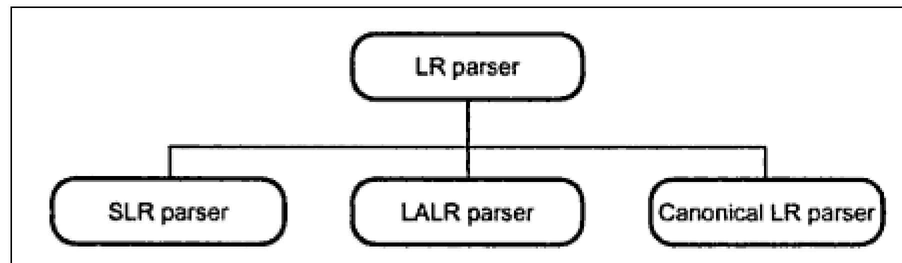

Figura 10. Família de parsers LR

A Figura 10 ilustra a família de parsers LR. O SLR significa *simple* LR (LR(0)). É o algoritmo menos poderoso da família. O LALR significa *look ahead* LR e é mais poderoso que o parser SLR. O LR canônico (LR(1)) é o mais poderoso de todos e consegue representar um maior número de classes que o SLR e LALR.

Dentre as opções de geradores de parsers disponíveis pode-se citar o ANTLR, JavaCC, Yacc e SableCC. O JavaCC e o SableCC, por exemplo, são implementações em Java que suportam EBNF. Ambos são de uso livre e são baseados em gramática LALR(k). Portanto, são capazes de analisar um número de linguagens muito maiores que o LL(1) (Gagnon, 1998).

O conjunto de linguagens que pode ser reconhecido por um *parser* LALR(1) é um super-conjunto das linguagens LL(K). Além disso, analisadores LALR(1) são, normalmente, mais rápidos do que os equivalentes analisadores LL(K) para a mesma linguagem, pois, estes últimos possuem um custo computacional maior para resolver conflitos. Os analisadores LALR(1) ainda podem lidar com recursividade à esquerda enquanto os analisadores LL(K) não podem.

2.3.3 SableCC

SableCC é um framework orientado à objetos para a construção de compiladores na linguagem de programação Java. O SableCC se beneficia de padrões de projetos para criar uma arquitetura mais modular, fácil de manter e gerar como produto final um compilador orientado à objetos (OO). Além disso, é um projeto *open source* (SABLECC, 2011).

O SableCC utiliza técnicas para separar, de forma clara, geração de código de máquina e código legível através de técnicas de orientação a objetos tornando a manutenção mais fácil. O SableCC utiliza uma extensão do padrão de projetos *visitor* que simplifica a manutenção da estrutura. É possível adicionar novos tipos de nós sem ter impacto nas classes existentes (Gagnon, 1998).

A implementação do SableCC é composta de um gerador de analisador léxico baseado em autômato finito determinístico (DFA), um analisador sintático capaz de analisar linguagens LALR(1) e um gerador de AST.

O processo de criação de um compilador utilizando o SableCC envolve uma série de procedimentos. Primeiro é necessário criar um arquivo de especificação onde estarão as definições léxicas e a gramática da linguagem. Após as especificações necessárias, utiliza-se o SableCC para gerar o framework. O passo seguinte é criar as *working classes* que geralmente herdam das classes geradas pelo SableCC. Normalmente, essas classes, possuem a função de análise semântica, geração de código e otimização (Gagnon, 1998). Ainda, é necessário criar uma classe principal que funcionará como o ativador do analisador léxico, analisador sintático e *working classes.* Por último, precisamos compilar o compilador com o compilador Java.

O compilador gerado pelo SableCC será capaz de reconhecer uma linguagem especificada em um arquivo texto que conterá definições léxicas e produções gramaticais. A estrutura do SableCC se beneficia do fato de não associar código de ação com *tokens* ou produções. Esse desacoplamento torna a manutenção mais eficiente e além de permitir a realização do debug, em uma IDE, diretamente nas *working classes.*

O SableCC recebe como entrada um arquivo que contém a especificação de uma gramática que reconhece uma linguagem e como saída gera arquivos (classes) que seguem uma determinada estrutura. Na versão 4.0 do SableCC é gerado quatro pacotes (*lexer, parser, node e analysis).* O pacote *lexer* conterá os arquivos necessários para realizar a análise léxica assim como as classes de *Exception.* O pacote *parser* segue a mesma analogia do pacote anterior. O pacote *node* contém

as classes que irão definir a AST. O pacote *analysis* contém as classes necessárias para caminhar através da AST (Gagnon, 2011).

```
1   Language exp;
2
3   Lexer
4     num = digit+;
5     digit = '0'..'9';
6
7     Ignored
8      ' ', #9, #10, #13;
9
10  Parser
11
12    program =
13      exp ';';
14
15    exp =
16      {add:} exp [op:]'+' exp |
17      {mul:} exp [op:]'*' exp |
18      {num:} num;
```

Figura 11. Exemplo de gramática de uma linguagem descrita usando o SableCC.

A Figura 11 descreve um exemplo de arquivo de especificação de uma linguagem para o SableCC versão 4.0. A declaração *Language* (linha1) define o nome da linguagem especificada e será o diretório raiz dos arquivos gerados pelo SableCC. A seção *Lexer* contém definições em expressões regulares, conforme Figura 11, para números (*num*) e dígitos (*digit*). Na seção *Ignored* (linha 7) são definidos os tokens que devem ser ignorados pelo parser. No caso do exemplo, espaço em branco, tabulação, LF *(Line Feed)* e CR tabulação (*Carriage Return*).

A seção *Parser* (linha 10) concentra as produções das gramáticas. A produção exp precede todas as alternativas das produções com um nome entre chaves que é responsável por tipificar uma alternativa.

Capítulo 3

SableCCDEV

Neste capítulo serão apresentadas as etapas e os detalhes da implementação do editor para o gerador de compiladores SableCC baseado na IDE Eclipse. Serão apresentados os procedimentos de desenvolvimento de cada recurso do editor bem como sua arquitetura e como eles estão interligados. Ao final será possível realizar uma analogia, dos recursos desenvolvidos para o editor proposto, com os conceitos já abordados no capítulo anterior acerca de editores de programas.

3.1 Arquitetura da IDE

A IDE proposta pelo trabalho está estruturada de forma modular seguindo um nível hierárquico. O ambiente é composto por três módulos na forma de plugins. São eles: SableCC, SableCCBuilder e o SableCCEditor.

A Figura 12, abaixo, descreve o nível hierárquico dos módulos. Quanto as dependências entre eles, o SableCCEditor depende do plugin SableCCBuilder que por sua vez depende do plugin SableCC. Este fornecerá os serviços inerentes ao gerador de compilador.

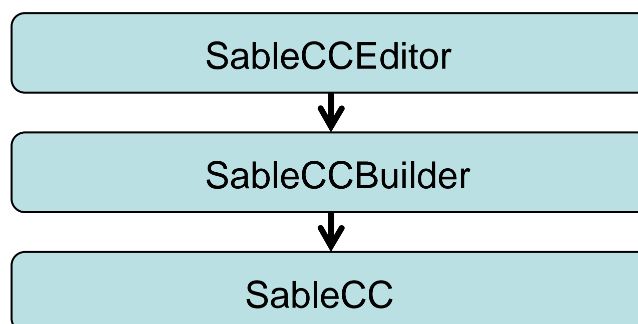

Figura 12. Dependência entre os subsistemas

Todo o desenvolvimento da IDE será realizado através do *Plug-in Development Environment* (PDE) incluso na plataforma Eclipse. A primeira etapa

para o desenvolvimento dos plugins, após identificar os pontos de extensão necessários, consiste na criação dos arquivos manifest (plugin.xml) (Beck e Gamma, 2003). São nesses arquivos que são definidos e declarados os pontos de extensão. O passo seguinte é implementar a interface esperada, de acordo com os pontos de extensões, e codificar as funcionalidades da ferramenta.

3.2 SableCC Plugin

O plugin SableCC será o módulo responsável pelo encapsulamento da implementação do framework SableCC, neste caso, a versão 4.0. Esse módulo será chamado sempre que for necessária a utilização dos recursos do gerador de compilador. Estão incluídos como serviços desse módulo a análise léxica, análise sintática, geração da AST e mecanismos para navegar através da árvore gerada.

3.3 SableCCBuilder Plugin

O SableCCBuilder é o segundo módulo e é responsável por gerar o código Java do parser a partir das gramáticas especificadas no SableCCEditor, ou seja, a idéia desse módulo é realizar a funcionalidade de *builder* descrita em (Arthorne, 2011).

O *SableCCBuilder* é a classe principal do módulo (plugin) de *builder.* Essa classe é responsável por verificar os erros da gramática escrita pelo usuário da IDE. Esses erros são produzidos pelo método *SableCC.generateJavaCode* fornecido pelo plugin SableCC, isto é, pelo framework SableCC que gera o conjunto de classes *Java* que serão utilizadas para compor o compilador da linguagem especificada pelo usuário.

Este módulo possui uma única configuração que pode ser personalizada pelo usuário da IDE. Esta configuração é o diretório de saída da geração das classes Java, que pode ser especificado a partir de uma página de preferências específica. A página de preferências do *builder* pode ser vista na Figura 13 a seguir.

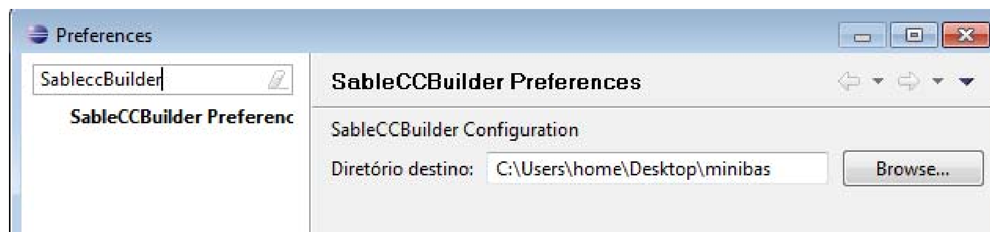

Figura 13. Tela da página de preferências do SableCCBuilder

3.4 SableCCEditor Plugin

O SableCCEditor, terceiro módulo, é a implementação do editor que representa o ambiente de desenvolvimento para especificação da gramática. É o subsistema que encapsulará a implementação de todos os recursos inerentes ao editor como edição sensível a linguagem, automação de escrita e suporte a navegação.

O arquivo *manifest* do SableCCEditor contém todas as definições declarativas necessárias para o funcionamento do plugin. É neste arquivo que estarão definidas as informações de propriedade, identificação, autoria, pontos de extensão, dependência e eventualmente informações de runtime.

O primeiro passo para dar prosseguimento ao projeto do editor é declará-lo no *manifest* (Beck e Gamma, 2003). O editor é definido no arquivo plugin.xml através do ponto de extensão *org.eclipse.ui.editors* (Realsolve, 2006). Considerado um dos principais pontos de extensão do workbench do Eclipse (Eclipse, 2005).

A Figura 1414 ilustra o procedimento de declaração do editor proposto, que será estendido. O atributo *point* (Eclipsepedia, 2011) é uma referência para o ponto de extensão do editor do workbench. O atributo *name* representa o nome do editor em um formato inteligível. *Extensions* é um atributo que especifica a extensão do arquivo e que consequentemente estará atrelado ao editor particular. *Icon* é a representação do ícone que estará atrelado aos arquivos reconhecidos pelo editor. O atributo *class* aponta para a classe principal que conterá a codificação do nosso editor e deverá estar em conformidade com uma interface esperada. *Id* é um identificador único do plugin desenvolvido (Beck, Gamma e Erick, 2003).

```
<extension
      point="org.eclipse.ui.editors">
   <editor
         name="SableCC Editor"
         extensions="grammar"
         icon="icons/sample.gif"
         class="br.poli.ecomp.ads.sablecceditor.editors.SableCCEditor"
         id="br.poli.ecomp.ads.sablecceditor.editors.SableCCEditor">
   </editor>
</extension>
```

Figura 14. Declaração do editor no arquivo plugin.xml

A classe especificada para a implementação do nosso editor, no arquivo manifest, deve estar em conformidade com uma interface, conforme mencionado anteriormente. Para isto a classe *SableCCEditor* deveria implementar a interface *ITextEditor.*

Para simplificar o desenvolvimento e aumentar a produtividade o Eclipse já fornece um framework, SWT / *JFace Text*, que permite o desenvolvimento de editores de texto com as características necessárias para a ferramenta proposta no trabalho (editor). Por questões de conveniência, nosso editor, ao invés de implementar a interface *ITextEditor* irá estender a subclasse *TextEditor*, que fornece uma implementação base para editores.

A classe *TextEditor* a qual nossa classe *SableCCEditor* irá estender, é uma subclasse de *AbstractDecoratedTextEditor.* Esta classe, por sua vez, representa um editor intermediário que compreende funcionalidades como número de linhas, impressão de margem e coloração da linha corrente (Realsolve, 2006).

A Figura 15 mostra um diagrama de classes com os principais componentes do SableCCEditor plugin. Cada uma dessas classes será discutida nas seções seguintes.

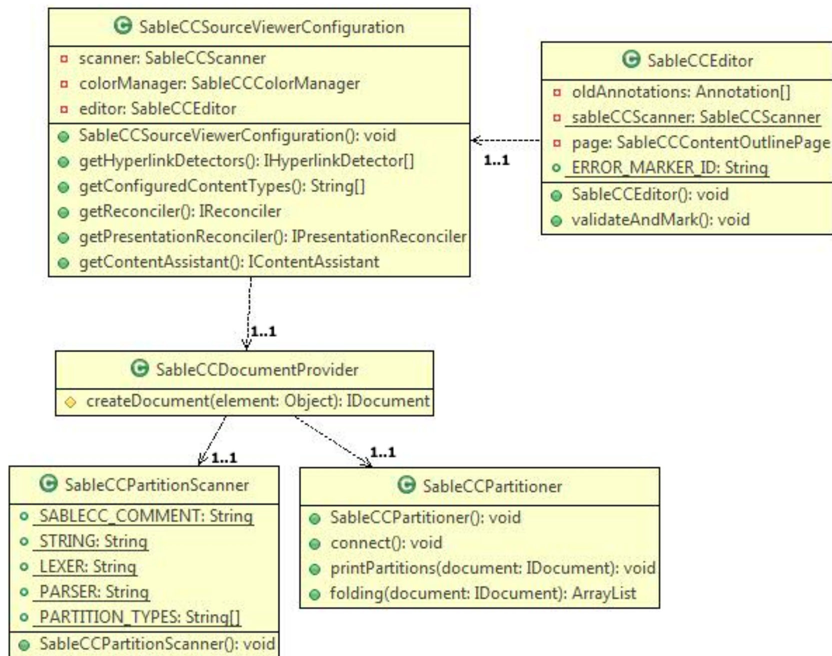

Figura 15. Diagrama de classes do SableCCEditor Plugin

3.4.1 Padrão Model-View-Controller

O padrão MVC (*model-view-controller*) (Guojie, 2005) é uma metodologia consagrada em projetos de software que tem como característica básica a capacidade de isolar a lógica de negócios da interface do usuário. O uso do MVC simplifica o desenvolvimento de aplicações fazendo com que diferentes camadas de interface gráfica consigam reutilizar a mesma lógica de negócios, ou seja, uma não irá interferir na outra.

O MVC fornece uma clara separação de interesses. A entidade *model* corresponde à representação dos dados. *View* será responsável por fornecer mecanismo para exibição dos dados. *Controllers* irão tratar os eventos capturados que irão afetar de alguma forma a *view* ou *model.*

As características mais importantes do padrão MVC são: diferentes *views* não se conhecem, *controllers* não referenciam *views*, *models* não referenciam *views,* as *views* não manipulam o *model* e somente elas sabem como reagir aos eventos.

O framework JFace Text do eclipse é uma aplicação baseada no padrão *model-view-controller.* A interface *IDocument* será responsável por representar o

model, neste caso, um documento de texto. Para realizar ações de visualização e edição do documento é necessário a figura do *controller* que é representado pela interface *ITextViewer.* Por fim um componente *StyledText* fará o papel da *view* (Guojie, 2005).

O framework fornece implementações abstratas e concretas das interfaces necessárias para o nosso editor. Os pacotes *org.eclipse.ui.texteditor* e *org.eclipse.ui.editors.text* contêm as classes que funcionam como *controller do editor* gerenciando o v*iew* e o modelo. A Figura 16 ilustra o modelo MVC do JFace Text (Rich Client, 2006).

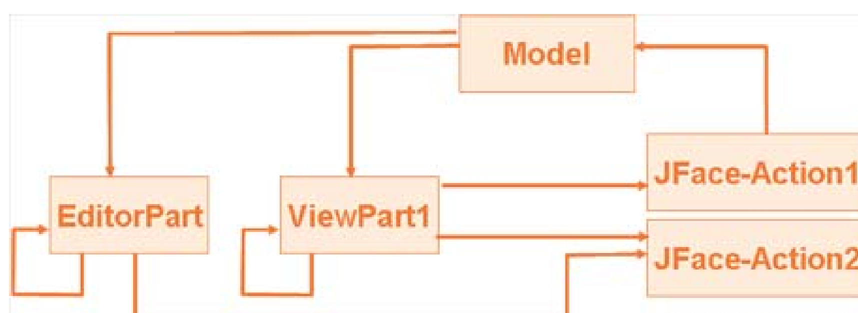

Figura 16. Comunicação entre componentes JFace no modelo MVC

Conforme ilustrado pela Figura 16, Os *controllers* são representados como *actions* (*JFace-Actions)* e podem ser embutidos em ferramentas que interagem com o usuário. O modelo é representado por *documents*, discutido mais adiante, e os *styledtext (EditorPart e ViewPart1)* (Vogel, 2011) representam *views*. Os *controllers* manipulam os *models* que por sua vez propagam as mudanças para as *views*.

3.4.2 Documents

Documents são elementos que armazenam texto e fornecem uma série de serviços para gerenciamento do conteúdo por ele armazenado. Através da entidade *Documents* é possível obter informações sobre determinada linha, manipular e detectar mudança do conteúdo do documento, realizar pesquisas e trabalhar com partições. As definições responsáveis por fornecer os mecanismos de manipulação do conteúdo estão disponíveis na interface *org.eclipse.jface.text.I*D*ocument.*

O *document* é a representação de um modelo (*model)*. Esse modelo contém apenas o conteúdo textual do editor. Em hipótese nenhuma é atribuído ao *model* informações de exibição do seu conteúdo.

Frequentemente estaremos utilizando em nossa implementação o termo *offset*. O *IDocument* utiliza o atributo *offset* para localizar determinada posição dentro do texto. Toda a complexidade envolvida em torno de posicionamento dos elementos do conteúdo é tratada por ela. Isso envolve, por exemplo, atualização automática de posicionamento do texto enquanto o documento está sendo editado.

O *IDocument* também fornece mecanismos de tratamento de erros. O *BadLocationException*, tipo de exceção da entidade, é lançado quando se tenta acessar conteúdo fora dos limites do documento (Deva, 2006).

3.4.3 DocumentProvider

A instância do *IDocument* não tem conhecimento de sua fonte de dados. Não se sabe, por exemplo, se ele será carregado a partir de um arquivo do sistema ou de um banco de dados. O papel do *IDocumentProvider* é criar uma instância do *IDocument* e iniciar processos necessários que irão definir o estado inicial do documento (Eclipse Foundation, 2011c).

O *DocumentProvider* cria uma ligação entre um arquivo armazenado em um disco e uma representação desse arquivo como um documento em memória. Além de ser responsável por prover meios para carregamento de arquivos ele também: cria e gerencia uma representação textual, cria e salva conteúdo e determina se existe modificação no documento desde o momento em que ele foi carregado através de uma notificação. Pode-se perceber visualmente esse mecanismo no Eclipse através de uma notificação na forma de um asterisco na aba do editor em que o documento está sendo editado.

O pacote do Eclipse *org.eclipse.ui.texteditor* define a interface *IDocumentProvider.* O Eclipse também fornece as subclasses necessárias, para nosso trabalho, que implementam essa interface. O pacote *org.eclipse.ui.editors.text* fornece a classe *FileDocumentProvider* que é uma implementação da interface

IDocumentProvider. Essa classe fornece mecanismos para carregar documentos a partir do sistema de arquivos’.

Para o nosso trabalho, iremos utilizar a implementação fornecida pela classe *FileDocumentProvider* No nosso caso, Iremos estendê-la através da classe *SableCCDocumentProvider* para que ela tome conhecimento do nosso esquema de particionamento que será discutido mais adiante. Será necessário redefinir o método *createDocument(Object element)*, que cria e configura um documento para uma entrada específica, define parâmetros necessários para o particionador e retorna como saída um objeto *IDocument*.

3.4.4 SourceViewer e SableCCSourceViewerConfiguration

O *ITextViewer* é o responsável por conectar um *widget (*elemento visual) de texto com o *IDocument*. Através dele o documento funcionará como o modelo do *widget* de texto, isso faz com que o código da aplicação não precise interagir com o *widget* diretamente. Além disso, fornece recursos como o gerenciamento de recursos como o desfazer (*undo)* e repetir (*redo)*.

O *ISourceViewer* (Deva, 2006) é um melhoramento do anterior, ela estende a interface *ITextViewer* provendo um maior número de recursos. Essa interface fornece suporte a *annotations* visuais que são estruturas “tipadas” que tem um texto associado e podem ser marcadas. Através das *annotations* o eclipse fornece funcionalidades como a marcação de erros (Eclipse Foundation, 2011a).

O nosso editor precisa modificar a configuração padrão do *source viewer*. O JFace fornece meios para realizar essa mudança através da extensão da classe *SourceViewerConfiguration* e a redefinição dos métodos necessários. O Nosso editor é configurado a partir da classe *SableCCSourceViewerConfiguration* que naturalmente herda de *SourceViewerConfiguration* (Deva, 2006).

A classe *SableCCSourceViewerConfiguration* do nosso editor realiza um papel importante em nosso projeto, pois, além de ser responsável por uma série de recursos da ferramenta é responsável pela configuração do editor. As

funcionalidades como formatação rica (*syntax highlighting),* auxílio à escrita (*auto-complete),* auxílio à navegação e duplo clique são configuradas a partir dela.

O *Source viewer* é criado pelo elemento *AbstractTextEditor.* Não é possível setar diretamente o *source*, porém, através do método *setSourceConfiguration* é possível adicionar uma classe de configuração e a partir dela configurar o *source viewer* utilizado pelo editor de texto.

O método *setSourceConfiguration* pode ser utilizado na classe principal do nosso editor, *SableCCEditor,* pelo fato de ser uma subclasse de *AbstractTextEditor* conforme consta na seção 3.2.

3.4.5 A classe SableCCEditor

Anteriormente foi explicado que a classe SableCCEditor estende a subclasse *TextEditor.* É o *SableCCEditor* que será responsável por criar o *sourceviewer* e outros recursos fundamentais do editor de texto. O *TextEditor* é parte do núcleo do framework de texto da interface de usuário do Eclipse.

O *SableCCEditor* relaciona o *viewer* e o *document (*model) juntos. Ela (*SableCCEditor)* é uma subclasse de *EditorPart* que é uma implementação de *org.eclipse.ui.IEditorPart.* O *IEditorPart* pode ser implementado diretamente, mas por questões de conveniência é mais interessante utilizar o *EditorPart* que é uma implementação base fornecida.

Um componente *editor part* é baseado na extensão criada e é adicionado ao *workbench* quando algum arquivo que tem uma das extensões declaradas é aberto pelo usuário. O *workbench* do eclipse representa uma instância de *IWorkbenchPage* que por sua vez é um conjunto de *IWorkbenchPart,* componentes visuais dentro da página de *workbench.* O *IEditorPart* é uma extensão de *IWorkbenchPart* e consequentemente representa o elemento *view* no modelo MVC.

Cada *IEditorPart* está relacionado com um *IEditorInput* que é um elemento que irá descrever a entrada do editor. O *IEditorInput* funcionará basicamente como uma descrição da fonte de modelo para um *IEditorPart* (Guojie, 2005).

Entre os métodos fundamentais e de maior importância para a nossa IDE da classe SableCCEditor destacam-se: o *createSourceViewer* que é responsável por criar, configurar e retornar o *sourceViewer;* o *createPartControl,* responsável por tornar o *EditorPart* visível no *workbench; initializeEditor* que define a classe de configuração do *sourceViewer*; o *getAdapter* que inicializa o *outline*; o *validateAndMark* responsável pela exibição das marcações de erros da IDE;

3.4.6 Particionamento

O framework JFace Text permite que um documento do Eclipse seja dividido em partições. Isto significa separar regiões de texto, dentro desse documento, e associá-lo a um tipo de conteúdo particular. Partições realizam um papel importante no framework, sendo responsável por recursos como folding, formatação rica e automação de escrita.

Cada partição proveniente do processo de particionamento do documento representa uma região de texto e tem um tipo associado ao mesmo. Uma partição pode ser, por exemplo, do tipo comentário, string ou não ter tipo especificado. Uma partição é representada pela interface *ITypedRegion* que fornecerá informações relativas ao tipo e tamanho de cada região. Por padrão, em nossa IDE, uma partição que não tem tipo especificado é do tipo *_dftl_partition_content_type.*

Através de partições é possível estabelecer diferentes comportamentos para cada região de acordo com o seu tipo. Através desse mecanismo é possível, por exemplo, distinguir a formatação de comentários, palavras chaves e strings, além de estabelecer critérios para automatização de escrita. O particionador da nossa IDE é definido e configurado na classe *SableCCDocumentProvider* a partir do método *createDocument.* A Figura 17 exibe o trecho de código que configura o esquema de particionamento do editor da nossa IDE.

```
protected IDocument createDocument(Object element) throws CoreException {
    IDocument document = super.createDocument(element);
    if (document != null) {
        IDocumentPartitioner partitioner = new SableCCPartitioner(
                new SableCCPartitionScanner(),
                SableCCPartitionScanner.PARTITION_TYPES);
        partitioner.connect(document);
        document.setDocumentPartitioner(partitioner);
    }
    return document;
}
```

Figura 17. Definição e configuração do esquema de particionamento

A nossa IDE habilita o mecanismo de particionamento a partir da criação de uma instância da interface *IDocumentPartitioner.* Essa interface é implementada através da subclasse *SableCCPartitioner* que, por conveniência, estende a classe *FastPartitioner* que é uma implementação fornecida pelo JfaceText.

Conforme mostrado na Figura 17, o método createDocument cria uma instância de SableCCPartitioner que precisa ser configurado (receber no construtor) um *IPartitionTokenScanner (SableCCPartitionScanner)* e um array de *string* que contém os tipos de partições suportadas. O *IPartitionTokenScanner* tem o objetivo de analisar o documento e associar um *IToken* a uma determinada partição do documento. O Eclipse fornece um *scanner* baseado em regras, *RuleBasedPartitionScanner*, que implementa a interface *IPartitionTokenScanner.* Customizamos o *scanner* para atender nossos interesses a partir da subclasse *SableCCPartitionScanner* que estende aquela classe. É necessário, também, chamar o método *connect* para associar o particionador a uma instância de *IDocument.*

Na classe *SableCCPartitionScanner,* que estende o *RuleBasedPartitionScanner,* são definidas constantes de *strings* que representam diferentes tipos de partições. Note no trecho de código da Figura 18 que definimos os tipos *SABLECC_COMMENT* (comentário), *STRING, LEXER* (seção lexer) e *PARSER* (seção parser). É necessário fazer com que os *tokens*, que representam cada tipo de partição, sejam reconhecidos ao passo que o documento for analisado. Para isso, a classe utilizada para o particionamento contém mecanismos de reconhecimento dos *tokens* através da definição de um conjunto de regras.

```
18      public SableCCPartitionScanner() {
19          String newLine = System.getProperty("line.separator");
20          IToken sableccComment = new Token(SABLECC_COMMENT);
21          IToken sableccString = new Token(STRING);
22          IToken sableccLexer = new Token(LEXER);
23          IToken sableccParser = new Token(PARSER);
24          List rules = new ArrayList();
25          // Add rule for single line comments
26          rules.add(new EndOfLineRule("//", sableccComment));
27          rules.add(new MultiLineRule("/*", "*/", sableccComment));
28          // Add rule for strings and character constants.
29          rules.add(new SingleLineRule("\"", "\"", sableccString, '\\'));
```

Figura 18. Definição de tokens e regras de tipos de partições para o SableCC

A Figura 18 mostra o construtor padrão sendo sobrescrito para definições das regras do editor da nossa IDE. Os *rules* analisam o documento em busca de sequencias de caracteres que correspondam a alguma regra especificada. A linha 26, da Figura 18, mostra a definição de uma regra para detectar comentários simples através da classe *EndOfLineRule* que cria uma regra para a sequencia inicial dada e retorna o *token sableccComment* em caso de sucesso.

O documento é analisado até que uma sequencia de texto corresponda a uma regra. Em caso de sucesso uma instancia de *token* do tipo da partição é retornado junto com informações da posição e tamanho dela e serão armazenados em um *IDocument*. Em caso de falha, um *token* do tipo *UNDEFINED* é retornado e o controle é passado para a próxima regra.

3.4.7 Formatação rica (*Syntax Highlight*)

A formatação rica de texto, mais conhecida como *Syntax Highlight,* remete à idéia abordada na seção 2.1.1. Trata-se de um recurso, bastante popular em IDE’s, que exibe uma porção de texto com uma coloração específica de acordo com o tipo de conteúdo que está associado àquela determinada região em questão.

O processo de formatação rica (destaque da sintaxe) envolve mecanismos semelhantes aos procedimentos de criação de partições abordados anteriormente, pois, também envolve idéias parecidas com as de *tokens* e regras. O Eclipse fornece o componente *presentation reconciler* (*IPresentationReconciler)* que é responsável pelo procedimento de destaque da sintaxe. Ele tem o papel de definir um conjunto

de *tokens* que atribuem cores e estilos de fontes para uma determinada região de texto.

Enquanto as partições definem um tipo de conteúdo particular de um documento, o *presentation reconciler* divide essas partições específicas em *tokens* que têm associado a eles atributos que definirão estilos de exibição. O *IPresentationReconciler* é responsável por detectar mudanças no documento e trabalha em conjunto com outras duas entidades: *IPresentationDamager* e o *IPresentationRepairer.* Através deles é possível manter a apresentação visual do documento enquanto o mesmo está sendo editado pelo usuário.

O d*amager* (*IPresentationDamager*) recebe uma notificação de cada partição de uma determinada área afetada devido a modificações do usuário no documento. O *damager* retorna um *IRegion* (região), indicando a área, que necessita ser reparada (reconstruída) por conta das modificações.

A região afetada e gerada como saída pelo *damager* é enviada para o *repairer (IPresentationRepairer)* que garantirá de reconstruí-la. O *repairer* será responsável por reparar a apresentação textual baseada nas regras que dividem a região em *tokens* com os devidos atributos de estilo associada cada um deles.

A classe *SourceViewerConfiguration* contém a implementação do método *IPresentationReconciler getPresentationReconciler*(*ISourceViewer sourceviewer*) que é responsável pelo recurso de destaque da sintaxe. O editor textual da nossa IDE utiliza a implementação *DefaultDamagerRepairer,* fornecida pelo JFace Text, que realiza o papel de *damager* e *repairer.* O editor também faz uso da classe *PresentationReconciler* que implementa a interface *IPresentationReconciler* (*reconciler).*

A classe *DefaultDamagerRepairer* espera em seu construtor uma instância de *ITokenScanner* que informará ao nosso editor as regras necessárias para identificação dos *tokens* dentro das partições e seus devidos atributos. *SableCCScanner* é a classe que possui esta responsabilidade. Ela estende de *RuleBasedScanner* e tem funcionamento semelhante ao particionador, abordado anteriormente, do nosso documento..

Na classe *SableCCScanner* são definidas as palavras chaves do SableCC através de um *array* de *Strings.* Em seu construtor são criados *tokens* que serão associados a alguma regra de identificação de sequência de caracteres assim como associá-los a alguma cor predefinida. É a classe *SableCCColorManager* a responsável por tratar as vinculações das cores. A Figura 19 exibe um trecho de código que ilustra a criação de *tokens,* associação a cores e definição de regras.

```
IToken comment = new Token(new TextAttribute(
        manager.getColor(PreferenceConverter.getColor(prefs,
                PreferenceConstants.COLOR_COMMENT))));
IToken operation = new Token(new TextAttribute(
        manager.getColor(PreferenceConverter.getColor(prefs,
                PreferenceConstants.COLOR_OPERATION))));

List rules = new ArrayList();
// Add rule for comments
rules.add(new MultiLineRule("/*", "*/", comment));
rules.add(new EndOfLineRule("//", comment));
```

Figura 19. Definição de tokens e regras para o SableCC

O ponto *org.eclipse.ui.preferencePages* é estendido com a finalidade de definir um página no menu de preferências do Eclipse. A classe SableCCPreferencePage é responsável pela implementação da página que tem como objetivo permitir ao usuário personalizar as cores dos *tokens* de cada partição.

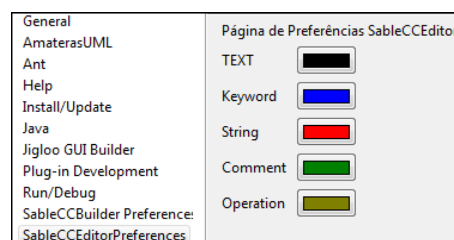

Figura 20. Página de preferência de cores do SableCCEditor

A Figura 21 mostra um diagrama de classes que torna mais clara os relacionamentos existentes entre os componentes abordados e diretamente envolvidos com o recurso de destaque da sintaxe (*syntax highlight).* O diagrama é uma representação parcial que aborda os principais atributos e métodos.

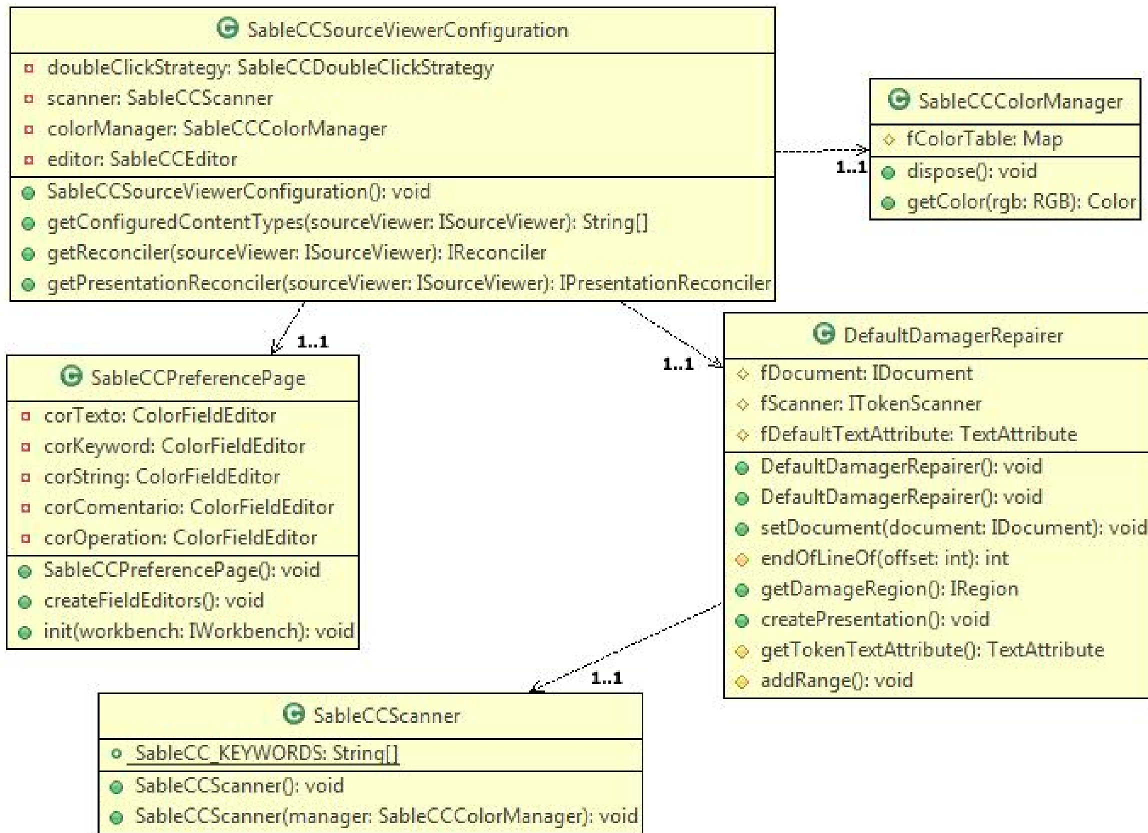

Figura 21. Diagrama de classes dos componentes do destaque de sintaxe

3.4.8 Automação de escrita (*auto complete*)

O recurso de automação de escrita é descrito na seção 2.1.2. No SableCCEditor a automação de escrita auxilia o usuário durante a etapa de codificação, fornecendo possíveis sugestões de termos válidos em função da posição atual do cursor.

É novamente na classe *SableCCSourceViewerConfiguration* que será configurado a automação de escrita através do método *IContentAssistant getContentAssistant().* É necessário instanciar um *IContentAssistant* e configurá-lo de acordo com as necessidades da IDE.

A classe, definida no pacote *org.eclipse.jface.text.contentassist, ContentAssistant* implementa a interface *IContentAssistant.* Essa classe fornece a implementação necessária para apresentação e inserção de sugestões de escrita dentro do documento além de prover meios para configurar a exibição do recurso

dentro do editor. Um elemento fundamental que trabalha em conjunto com ela é o *IContentAssistProcessor,* responsável por fornecer suporte a sugestões de escrita para uma determinada partição.

Naturalmente, cada partição do documento, que dará suporte a esse recurso, necessita definir um *IContentAssistProcessor.* No nosso caso, três implementações dessa interface são criadas:

- *SableCCContentAssistProcessor* que fornece suporte a partições sem tipo de conteúdo associado.
- *SableCCPaserCompletionProcessor* que fornece suporte à partição do tipo *Parser.*
- *SableCCLexerCompletionProcessor* que fornece suporte à partição do tipo *Lexer.*

```
ContentAssistant assistant = new ContentAssistant();
assistant.setContentAssistProcessor(new SableCCContentAssistProcessor(
        editor), IDocument.DEFAULT_CONTENT_TYPE);

assistant.setContentAssistProcessor(
        new SableCCPaserCompletionProcessor(file),
        SableCCPartitionScanner.PARSER);

assistant.setContentAssistProcessor(
        new SableCCLexerCompletionProcessor(file),
        SableCCPartitionScanner.LEXER);
```

Figura 22. Trecho de código do método *getContentAssistant()* definindo *processors*

O método *computeCompletionProposals* das implementações criadas é o responsável por fornecer o conjunto de sugestões de acordo com a posição do cursor. Esse método basicamente obtém uma lista de sugestões a partir da extração de elementos da árvore sintática gerada pelo SableCC. A classe *Echo* é a responsável por percorrer a árvore sintática e realizar o procedimento de extração dos nós que serão utilizados para gerar sugestões. As partições *lexer* e *parser* seguem esse processo enquanto a partição de conteúdo padrão (sem definição) obtém sugestões de palavras chaves pré-definidas na classe *SableCCScanner.* Um

diagrama de classes, Figura 23, ilustra a disposição dos componentes utilizados por esse recurso.

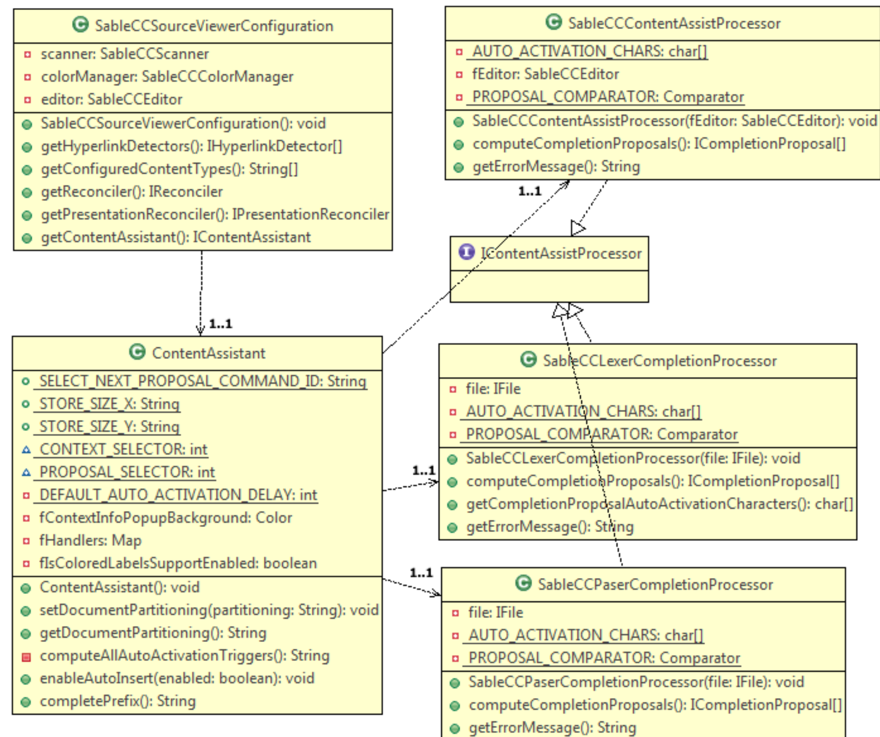

Figura 23. Diagrama de classes dos componentes da automação de escrita

3.4.9 Folding

O *folding* é um mecanismo que permite esconder e revelar blocos de códigos dentro do editor. Esse recurso permite uma melhor organização e visualização da especificação de uma gramática.

A classe *SableCCReconcilingStrategy* define a implementação que determina as regiões de código que suportam o mecanismo de expansão e ocultação dentro do editor. Essa classe implementa a interface *IReconcilingStrategy* que envolve a idéia análoga do destaque de sintaxe, onde um *reconciler* é utilizado para tratar partições particulares. Através do *reconciler,* configurado a partir do *IReconcilingStrategy,* uma *thread* separada verifica continuamente se houve modificações no documento e atualiza o posicionamento dos blocos de código (Eclipse corner article, 2005).

Para finalizar, o método *updateFoldingStructure* da classe *SableCCEditor* obtém a lista de posições dos blocos de código calculadas pelo *IReconcilingStrategy* e habilita o mecanismo de *folding.* Odiagrama de classes presente na Figura 23 ilustra a disposição das classes envolvidas neste recurso.

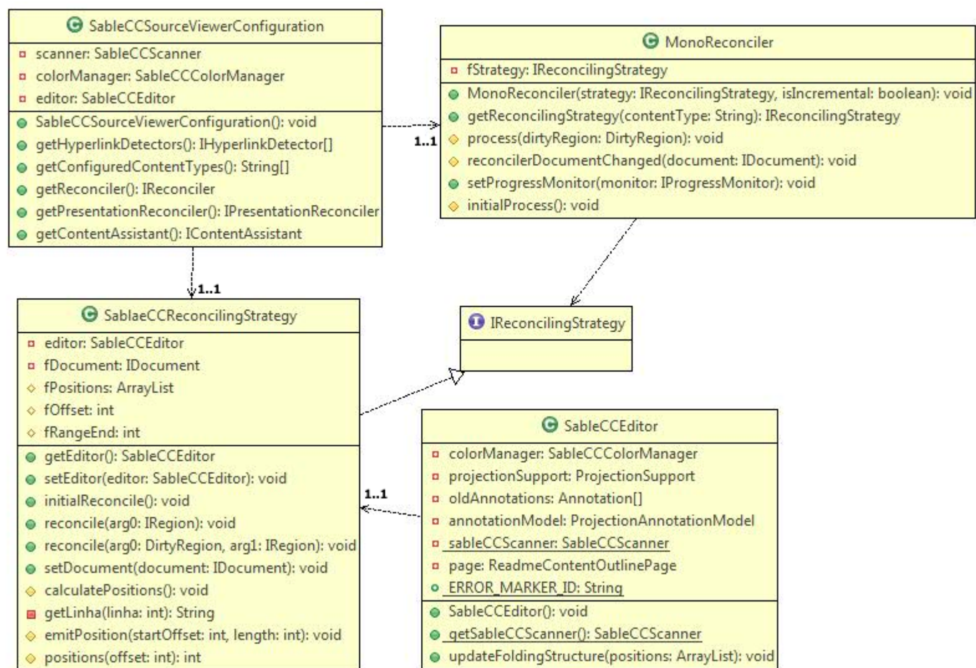

Figura 24. Diagrama de classes do recurso *folding*

3.4.10 Auxílio à navegação (*browsing)*

O auxílio à navegação é um recurso que permite ao usuário ir diretamente para a definição de um determinado termo dentro do arquivo de especificação da linguagem. Com apenas um clique simples em cima de um determinado termo, o usuário é levado diretamente para a definição de uma produção ou de um *token.* É uma ferramenta útil para tratar, normalmente, arquivos grandes.

A nossa ferramenta implementa esse recurso através da redefinição do método *IHyperlinkDetector*[] *getHyperlinkDetectors*(*ISourceViewer sourceViewer*) na classe *SableCCSourceViewerConfiguration.* É necessário criar um detector de links através da implementação da interface *IHyperlinkDetector.*

O *SableCCHyperlinkDetector* implementa a interface *IHyperlinkDetector* e redefine apenas um método, o *detectHyperlinks.* Nosso detector define regiões de

texto clicáveis que funcionam, literalmente, como um link para sua respectiva declaração. Através da classe *Echo,* que percorre a AST, é obtida uma lista de todos os termos do documento. A partir dessa lista são criados *IHyperlink* que armazenam as informações relativas ao destino do link, como identificação e posicionamento.

A classe *SableCCHyperLink* implementa a interface *IHyperlink* e na redefinição do método *open* implementa a funcionalidade de abrir o link, ou seja, levar até a definição de um determinado termo. O diagrama de classes presente na Figura 25 torna mais claro o entendimento das classes envolvidas neste recurso.

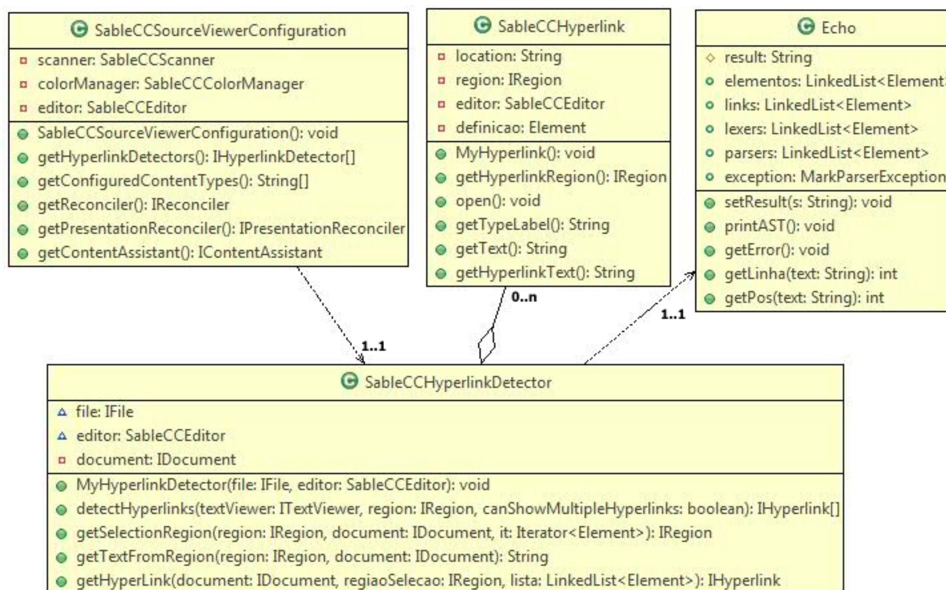

Figura 25. Diagrama de classes do recurso navegação (*browsing)*

3.4.11 Visualizador de árvore sintática (*content outline)*

O visualizador de árvore sintática sintetiza partes fundamentais da estrutura do documento editado em uma árvore visual localizada no *outline view* do Eclipse. Além de organizar a estrutura em uma hierarquia, dando uma visão geral dela ao usuário, o recurso também permite navegar no documento através da seleção dos nós da árvore (Realsolve, 2006).

O visualizador (*outline)* é acionado no Eclipse através do método *getAdapter()* da classe *SableCCEditor* que funciona como um mecanismo de extensibilidade.

Ainda é necessário passar uma implementação de *IContentOutlinePage* como argumento ao método.

A nossa subclasse SableCC*ContentOutlinePage* estende uma implementação do *IContentOutlinePage,* o *ContentOutlinePage.* O papel dessa classe é definir um *provider* que será responsável pelo modelo utilizado para definição do visualizador e também por propagar atualizações.

A classe *SableCCModelFactory,* através da classe *SableCCFileParser,* extrai elementos da árvore sintática que irão fazer parte do visualizador (*outline).* Tais elementos são representados pela classe *MarkElement* que define a identificação do que cada marcador representa no documento, assim como seu posicionamento e propriedades de estilo.

O SableCC*ContentOutlinePage* contém um *handler* responsável por detectar a seleção dos nós do *outline* realizada pelo usuário. É neste *handler* que é implementado o recurso de navegação, ou seja, a partir da seleção de um determinado nó o editor foca o posicionamento da estrutura selecionada. O diagrama de classes é apresentado para ilustrar na Figura 25 as classes necessárias para a implementação do recurso.

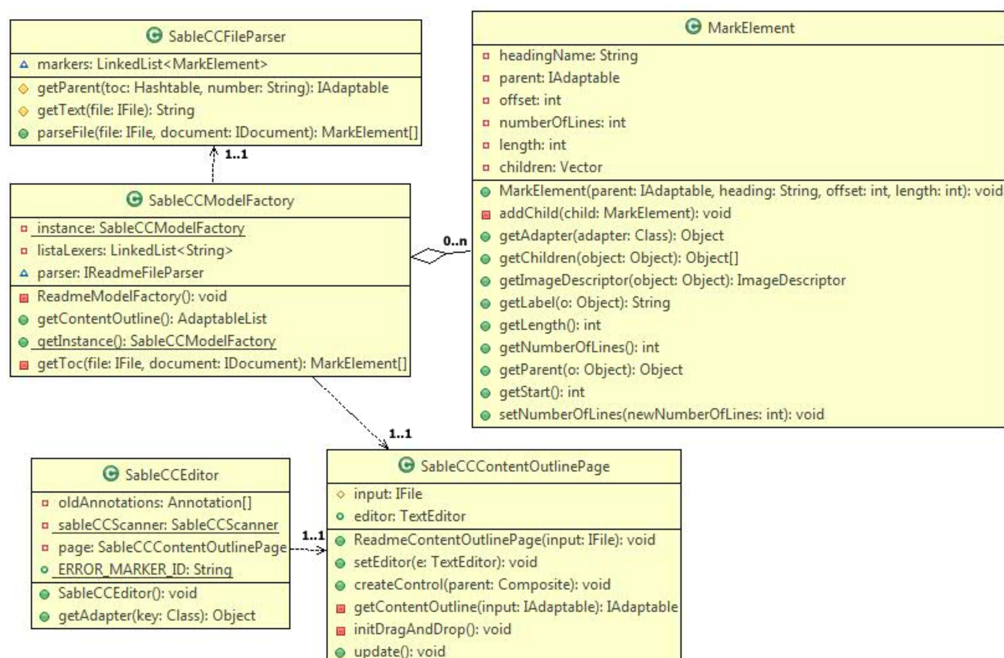

Figura 26. Diagrama de classes do recurso visualizador de árvore sintática (*outline)*

3.4.12 Marcador de erro (*error marker)*

O marcador de erro (*error marker)* é um recurso fundamental em um projeto de uma IDE. A partir dele é possível identificar e exibir erros ao usuário. Para implementar o marcador de erro da IDE para o SableCC, é necessário estender o ponto *org.eclipse.core.resources.markers* e declarar os tipos de marcadores utilizados, em nosso caso, o *problemmarker* e o *textmarker.*

O recurso de marcação é inicializado na classe *SableCCEditor* a partir do método *validateAndMark().* Basicamente, o método cria uma instância da classe *MarkingErrorHandler* passando um *IFile* e um documento como argumento. O *IFile* será utilizado pelo *parser* do SableCC para identificar erros sintáticos. Já o documento será utilizado para marcar as posições de erros baseado no resultado da análise do *parser.*

O método *markErrors* da classe *MarkingErrorHandler* contém a implementação da funcionalidade de marcação de erros. Ele chama o método *MarkerUtilities.createMarker(),* responsável por criar o marcador do arquivo em edição. O diagrama de classes presente na Figura 27 ilustra os componentes envolvidos na implementação dessa funcionalidade. A classe *Echo* é responsável por detectar os *Exceptions* lançados pelo analisador sintático e a classe *SableCCEditor,* a partir dos *Exceptions,* cria as marcações visuais de erros.

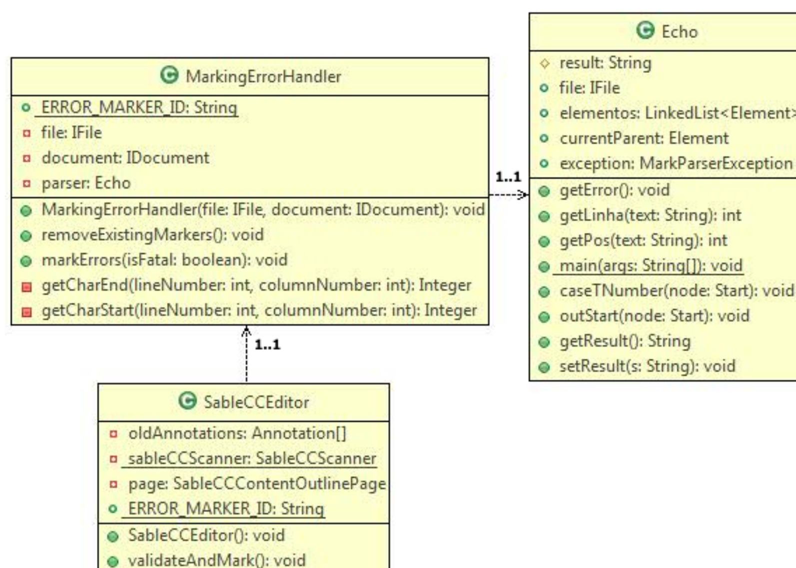

Figura 27. Diagrama de classes do recurso de marcação de erros (*error marker)*

Capítulo 4
Estudo de caso

Este capítulo tem o objetivo de apresentar um exemplo de uso que envolve a demonstração da utilização da IDE, desenvolvida por este trabalho, através da criação de uma gramática para uma simples linguagem imperativa. A linguagem especificada pela gramática chama-se *Mini Basic* e compreende um subconjunto da linguagem Basic (Wikipedia, 2011). A gramática do, *Mini Basic,* foi originalmente criada por Etienne Gagnon responsável por desenvolver o *SableCC* e adaptada para atender a nossa finalidade.

4.1 Mini Basic

O Basic é uma linguagem imperativa que surgiu nos anos 60 criada pelos professores Kemeny e Kurtz para fins didáticos. O Mini Basic é uma linguagem definida como um subconjunto da gramática especificada para o Basic. Conforme é possível visualizar na Figura 28, é uma linguagem simples composta por comandos facilmente compreensíveis na língua inglesa e que nos auxiliará neste exemplo.

```
9   Language arquivo;
10  Lexer
11    letter = 'A'..'Z';
12    digit = '0'..'9';
13    cr = #13;
14    lf = #10;
15    not_cr_lf = #32..#127;
16    if = 'IF';
17    then = 'THEN';
18    else = 'ELSE';
19    endif = 'ENDIF';
20    for = 'FOR';
21    to = 'TO';
22    next = 'NEXT';
23    read = 'READ';
24    print = 'PRINT';
25    println = 'PRINTLN';
26    assign = ':=';
27    less_than = '<';
28    greater_than = '>';
29    equal = '=';
30    plus = '+';
31    minus = '-';
32    mult = '*';
```

```
47  Parser
48
49    statements =
50      {list:}  statement statements |
51      {empty:} empty;
52
53    statement =
54      {if:} if condition then [nl1:] new_line
55                statements
56              optional_else
57              endif [nl2:] new_line |
58
59      {for:} for identifier assign [from_exp:] expression
60      to [to_exp:] expression [nl1:] new_line statements next
61      [nl2:] new_line |
62      {read:}       read identifier new_line |
63      {print_exp:}  print expression new_line |
64      {print_str:}  print string new_line |
65      {println:}    println new_line |
66      {assignment:} identifier assign expression new_line;
67
68    optional_else =
69      {else:}  else new_line statements |
70      {empty:} empty;
```

Figura 28. Trecho da gramática do Mini Basic

4.1.1 Projeto Mini Basic na SableCC IDE

O primeiro passo consiste na criação de um projeto no Eclipse que será responsável por conter a gramática do *Mini Basic*. Após a criação do projeto é criado o arquivo de especificação da gramática com o auxílio de um *wizard.* A Figura 29 abaixo ilustra o procedimento descrito.

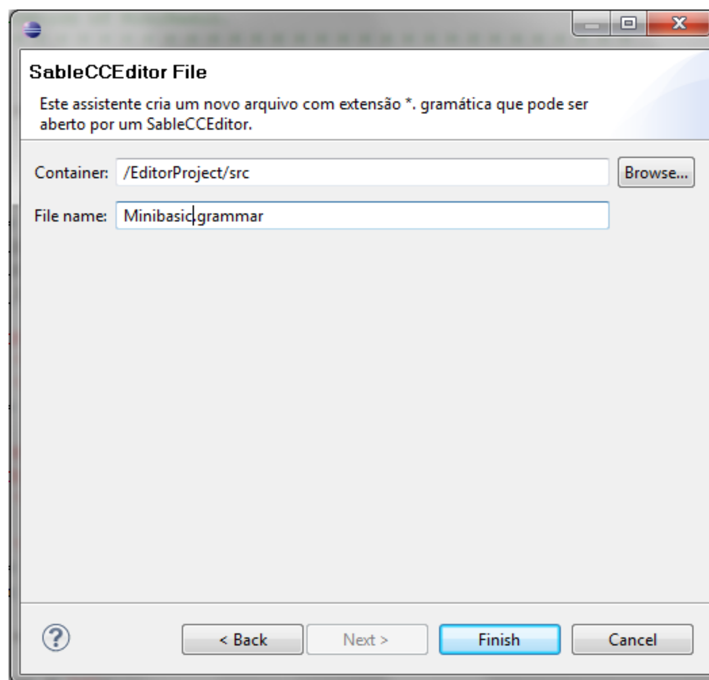

Figura 29. Tela de criação de uma gramática para o SableCC através de *wizard*

Uma das características fundamentais de uma IDE e que está presente em nosso editor é a capacidade de ser sensível a linguagem, propriedade descrita na seção 2.1.1. A partir dessa característica é possível usufruir de recursos como formatação rica. Conforme a Figura 30, começamos a especificação da gramática do *Mini Basic* com a inserção de um comentário de múltiplas linhas, compreendendo as linhas um a oito, que define a licença de utilização da gramática. Logo abaixo, na linha dez, através da palavra chave (*keyword*) Language é definido o nome da nossa gramática denominada *minibasic*. Neste momento já é possível visualizar (ver Figura 30) o trabalho do particionador do editor através do destaque da sintaxe.

```
1 /*
2   * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
3   * This file is part of MiniBasic.                                *
4   * See the file "MiniBasic-LICENSE" for Copyright information and *
5   * the terms and conditions for copying, distribution and         *
6   * modification of MiniBasic.                                     *
7   * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
8   */
9
10 Language minibasic;
```

Figura 30. Tela de definição da gramática

A etapa seguinte consiste na definição da seção *Lexer* (linha doze). Através de expressões regulares é definido um conjunto de *tokens* que serão utilizados para a especificação da gramática. A seção *Lexer* engloba as linhas doze a cinquenta e nove. É possível perceber que a partição foi reconhecida e visualizar na prática o trabalho realizado pelo *partitioner*, *damager* e o *repairer*. A Figura 31 abaixo exibe a construção da seção L*exer*.

```
12 Lexer
13
14    letter = 'A'..'Z';
15    digit = '0'..'9';
16    cr = #13;
17    lf = #10;
18    not_cr_lf = #32..#127;
19
20    if = 'IF';
21    then = 'THEN';
22    else = 'ELSE';
23    endif = 'ENDIF';
24
25    for = 'FOR';
26    to = 'TO';
27    next = '
```

Figura 31. Tela de construção da seção *Lexer.*

Na seção *Parser* são definidas as produções da gramática. Com o auxílio do recurso de automação de escrita, descrito na seção 2.1.2, é possível aumentar bastante a produtividade. Através da Figura 32, é possível ver na prática o trabalho realizado pelo *ContentAssistant* e *SableCCPaserCompletionProcessor.*

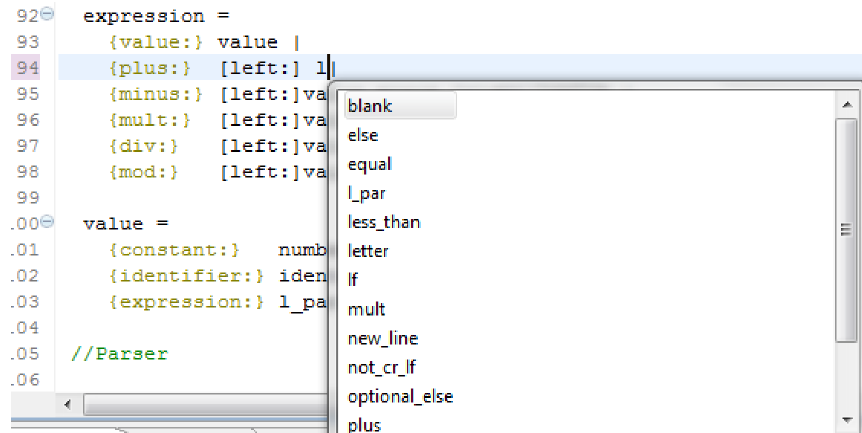

Figura 32. Demonstração do recurso de automação de escrita.

Se durante a construção da gramática ou durante alguma modificação o usuário cometer algum erro de sintaxe, um marcador aparecerá e o possível problema terá sua descrição mostrada na *view de* problemas, além da localização do mesmo. A Figura 33 mostra o trabalho realizado pela classe *MarkingErrorHandler,* responsável pela funcionalidade de marcação de erros, que detecta um erro na linha 39 informando a falta do elemento '='.

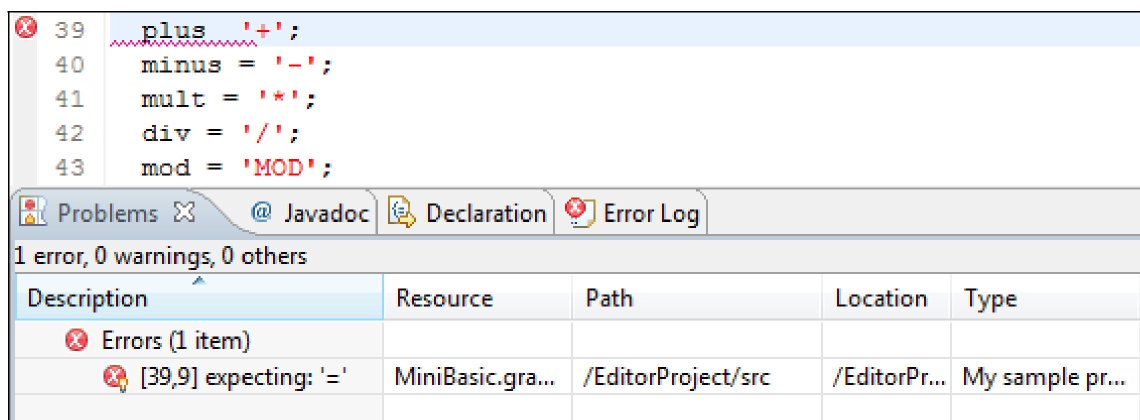

Figura 33. Demonstração do recurso de marcação de erro.

O recurso de *folding,* descrito na seção 3.2.9, permite ao desenvolvedor visualizar apenas partes específicas do documento. É possível através de esse recurso esconder determinados blocos de código permitindo ao usuário ter uma visão mais limpa da gramática. A Figura 34 exibe a seção *lexer* totalmente escondida e a visualização da seção *parser* com produções parcialmente escondidas.

```
12⊕ Lexer  [..]
22
23⊖ Parser
24
25⊖   statements =
26      {list:}   statement statements |
27      {empty:} empty;
28
29⊕   statement = [..]
41
42⊕   optional_else = [..]
```

Figura 34. Demonstração do recurso de *folding.*

A seção 2.1.3 descreve o recurso de navegação (*browsing)* utilizado em editores*.* No editor de nossa IDE esse recurso permite que o desenvolvedor encontre a definição de determinado elemento que está sendo utilizado para especificar uma produção ou *token.* Para encontrar a definição de um termo no editor da IDE do SableCC é necessário segurar a tecla *ctrl* do teclado e posicionar o cursor do mouse em cima de um determinado termo. Se existir uma definição para aquele termo, então ele se tornará uma região de texto "*clicável".* A Figura 35 demonstra a utilização do recurso, ao clicar no termo *value* na linha 91 o usuário é levado até a definição desse termo na linha 97.

```
89⊖  expression =
90     {value:}  value |
91     {plus:}   [left:]value plus  [right:]value |
92     {minus:}  [left:]v...  minus [right:]value |
93     {mult:}   [left:]v...  mult  [right:]value |
94     {div:}    [left:]v...  div   [right:]value |
95     {mod:}    [left:]v...  mod   [right:]value;

97⊖  value =
98     {constant:}   number |
99     {identifier:} identifier |
100    {expression:} l_par expression r_par;
```

Figura 35. Demonstração do recurso de navegação (*browsing).*

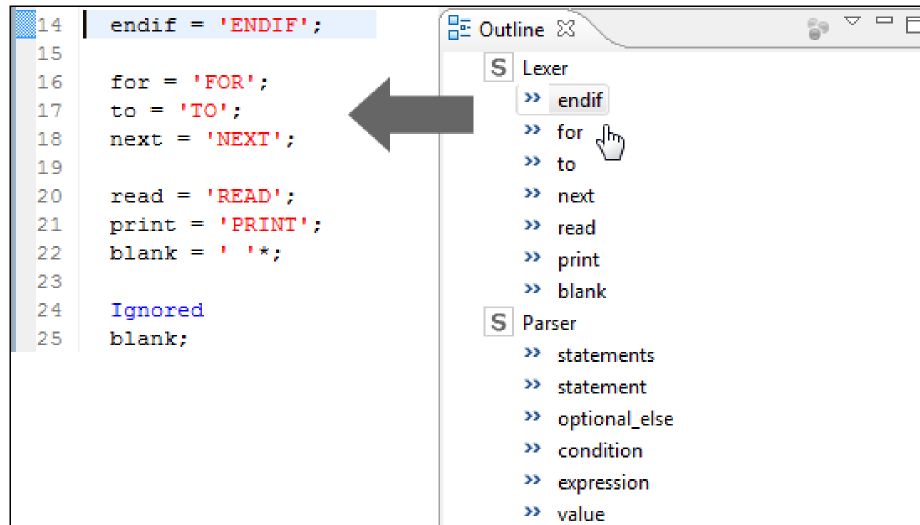

Figura 36. Demonstração do recurso de visualizador da árvore sintática (*browsing).*

O recurso de visualizador da árvore sintática, conhecido como *content outline,* é utilizado para fornecer uma visualização geral da estrutura do documento para o usuário. Além de uma melhor visualização é possível navegar no documento a partir de cliques nos nós correspondentes. A Figura 36 ilustra a utilização do recurso na gramática *Mini Basic.*

Para concluir, estando a especificação da gramática definida, a etapa final é gerar os arquivos em formato de classes *Java* para o compilador da linguagem correspondente à gramática. Para realizar esse procedimento é necessário informar um diretório válido, para geração dos arquivos, a partir do menu *Windows -> preferences -> SableCCBuilderPreferences.* A Figura 37 apresenta a tela de preferência de definição de diretório de saída dos arquivos.

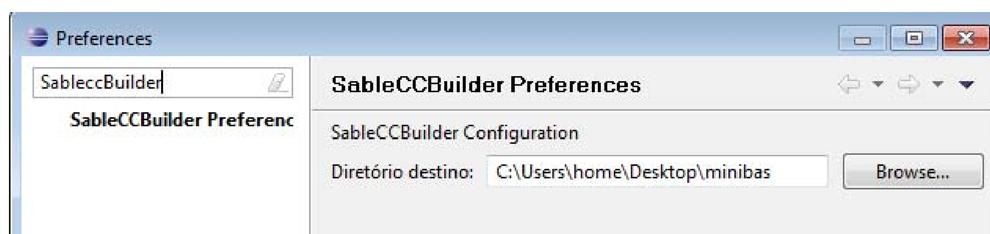

Figura 37. Tela de preferência de definição de diretório de saída dos arquivos

Capítulo 5
Conclusão

Concluímos nosso estudo apresentando nossos pontos de vista sobre a contribuição deste trabalho para a comunidade de desenvolvedores do SableCC, assim como, apresentar as possíveis contribuições futuras que poderão dar continuidade ao desenvolvimento desse projeto.

O ambiente de desenvolvimento integrado Eclipse continua com uma alta popularidade, sendo um dos ambientes mais utilizados no mundo. Boa parte dessa popularidade se deve ao fato de ser uma plataforma com uma arquitetura voltada para integração de ferramentas e possuir uma grande comunidade, disposta a trabalhar e contribuir com novas funcionalidades e recursos.

O plugin desenvolvido nesse trabalho contribui com a plataforma no sentido de apoiar o desenvolvimento de compiladores através do SableCC. Esse trabalho enxerga no framework SableCC um grande potencial, porém, havia a necessidade de um editor amigável e funcional para apoiar as atividades do desenvolvedor. É nesse contexto que o estudo entrou no ciclo de contribuição do Eclipse com a finalidade de prestar uma contribuição à comunidade SableCC e Eclipse.

Dentre os inúmeros plug-ins da plataforma Eclipse, o Xtext (Behrens, 2011) é um framework integrado a ele que permite criar uma linguagem de programação qualquer, de domínio específico, e gerar um metamodelo EMF (Flensburg e Voelter, 2006) dessa linguagem. A partir desse metamodelo o XText pode gerar um editor Eclipse específico para a linguagem criada. Esse editor específico conta com todos os recursos do editor nativo do Xtext.

Devido a problemas ocorridos durante o percurso do trabalho, principalmente devido ao tempo, as pretensões iniciais de se trabalhar e aplicar metamodelagem ao projeto foram comprometidas. Portanto essa idéia foi remanejada para os trabalhos futuros.

Inspirado no Xtext espera-se como trabalhos futuros a construção de um plugin capaz de ser integrado à plataforma Eclipse que forneça benefícios análogos aos do Xtext, poderá ser definida uma linguagem e um editor será gerado. A diferença está na vantagem do SableCC ser baseado em um algoritmo LALR (Aho, Sethi e Ullman, 1995), sendo capaz de especificar um maior número de classes de linguagens.

Além do ambiente de desenvolvimento integrado, é interessante realizar como trabalhos futuros um experimento controlado para avaliar critérios de produtividade no uso e não uso do editor proposto. Outros produtos finais esperados em trabalhos futuros envolvem o meta-modelo de uma linguagem formal CSP a partir de uma gramática escrita para o SableCC e a partir do método de bootstrapping (Terry, 1997), o metamodelo do próprio SableCC.

Referências

Aho, A.; Sethi, R.; Ullman, J. **Compiladores:** Princípios, Técnicas e Ferramentas. 2nd ed. Rio de Janeiro: Livros Técnicos e Científicos Editora S.A., 1995.

Adam, Sébastien (2004). **Visual Conflict Debugger Eclipse Plu-gin** Disponível em: <http://www.sable.mcgill.ca/publications/posters/sable-poster-2004-1.pdf>. Acesso em: 06 dezembro 2011.

ANTLRWorks. **ANTLRWorks:** The ANTLR GUI Development Environment. Disponível em: < http://www.antlr.org/works/index.html/>. Acesso em: 19 setembro 2011.

ANTLR (2011). **About The ANTLR Parser Generator**. Disponível em: <http://www.antlr.org/about.html>. Acesso em: 19 setembro 2011.

Arthorne, John. Project Builders and Natures. Disponível em: <http://www.eclipse.org/articles/Article-Builders/builders.html>. Acesso em: 15 nov. 2011.

Beck, Kent; Gamma, Erick. **Contributing to Eclipse**. Redwood City, CA, USA: Addison Wesley Longman Publishing Co, 2003.

Behrens, Heiko. **Xtext.** Disponível em: <http://www.eclipse.org/Xtext/>. Acesso em: 19 setembro 2011.

Dart, Susan et al. Software Development Environment. **IEEE Computer Society**, Pittsburgh, v. 20, n. 11, Nov. 1987.

Deva, Prashant. **Create a commercial-quality Eclipse IDE**. India: DeveloperWorks IBM Corporation, 2006.

Eclipse (2005). **Workbench UI Package** Disponível em: <http://help.eclipse.org/ganymede/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/package-summary.html/>. Acesso em: 8 outubro 2011.

Eclipsepedia. **FAQ What are extensions and extension points?.** Disponível em: <http://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points/>. Acesso em: 15 novembro 2011.

Eclipse Foundation, Inc.. **Eclipse:** The Eclipse Foundation open source community website. Disponível em: <http://www.eclipse.org/>. Acesso em: 18 setembro 2011a.

Eclipse Foundation, Inc. **Eclipse documentation - Current Release for package ui.editors**. Disponível em: <http://help.eclipse.org/indigo/index.jsp>. Acesso em: 21 outubro 2011b.

Eclipse Foundation, Inc. **Eclipse documentation - Current Release**. Disponível em: <http://help.eclipse.org/indigo/index.jsp>. Acesso em: 08 setembro 2011c.

Eclipse corner article (2005). **Folding in Eclipse Text Editors**. Disponível em: < http://www.eclipse.org/articles/Article-Folding-in-Eclipse-Text Editors/folding.html>. Acesso em: 15 setembro 2011.

Eclipse Object Technology International, Inc (2003). **Eclipe Platform Technical Overview.** Disponível em: <http://www.eclipse.org/whitepapers/eclipse-overview.pdf/>. Acesso em: 9 outubro 2011.

Emmerich, Wolfgang (2008). **Program Editors** Disponível em: <http://www.cs.ucl.ac.uk/staff/ucacwxe/lectures/GS04-0809/ProgramEditors.pdf>. Acesso em: 06 dezembro 2011.

Erickson, Marc (2001). **IBM - Working the Eclipse platform** Disponível em: <http://www.ibm.com/developerworks/library/os-plat/>. Acesso em: 8 outubro 2011.

Flensburg, Efftinge; Voelter, Markus. **oAW xText:** A framework for textual DSLs. In: Conference: Eclipse Summit 2006 Workshop: Modeling Symposium, 2006, Esslingen, Germany. 22 setembro 2006.

Gagnon, Ètienne. **SableCC, an Object Oriented Compiler Framework**. Março 1998. 107 f. Tese (Mestrado) – School of Computer Science, McGill University, Montreal.

Guojie, Jackwind. **Java Native Interfaces with SWT/JFace**. 1st ed. Canada: Wiley publishing, Inc., 2005. 508p.

Jim, D'anjou.; Scott, Fairbrother.; Dan, Kehn et al. **The Java Developer's guide to Eclipse**. 2nd ed. Boston: Pearson Education, Inc., 2005. 1083p.

Lars, Vogel. **Eclipse Editor Plugin Tutorial**. Disponível em: <http://www.vogella.de/articles/EclipseEditors/article.html>. Acesso em: 18 setembro 2011.

Oliveira, Hitalo. **Sistema de Gestão Corporativa de TI – SGCTI**. Junho 2011. 69 f. Monografia (TCC) – Escola politécnica de Pernambuco, Universidade de Pernambuco, Recife.

OSGi Alliance. OSGi Alliance | Specifications. Disponível em: < http://www.osgi.org/Specifications/HomePage?section=2#Release4>. Acesso em 21 outubro 2011.

Puntambekar, A.A. **Compiler Design**. 1st ed. India: Technical Publications Pune, 2010. 461p.

Realsolve (2006). **Realsolve - Building an Eclipse Text Editor with JFace Text**. Disponível em: <http://www.realsolve.co.uk/site/tech/jface-text.php>. Acesso em: 15 setembro 2011.

Rich Client (2006). **Using the MVC Pattern in Eclipse Applications**. Disponível em: < http://www.richclient2.de/2006_08_22/using-the-mvc-pattern-in-eclipse-applications/>. Acesso em: 15 novembro 2011.

SABLECC. **SableCC Project**. Disponível em: <http://sablecc.org/>. Acesso em: 19 agosto 2011.

Terry, P.D. **Compilers and Compiler Generators - an introduction with C++**. 1st ed. South Africa: Coriolis Group (Sd); Pap/Dsk edition., 1997. 512p.

Wikipedia (2011). **Basic**. Disponível em: < http://pt.wikipedia.org/wiki/BASIC />. Acesso em: 5 dezembro 2011.