CSPDT: Uma IDE para CSP

Trabalho de Conclusão de Curso

Engenharia da Computação

Victor Vilmarques Capistrano Pedrosa
Orientador: Prof. Joabe Bezerra de Jesus Júnior

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

VICTOR VILMARQUES CAPISTRANO PEDROSA

CSPDT: UMA IDE PARA CSP

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

Recife, Julho de 2016.

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

ESCOLA POLITÉCNICA
DE PERNAMBUCO

MONOGRAFIA DE FINAL DE CURSO

Avaliação Final (para o presidente da banca)*

No dia 14 de julho de 2016, às 14:00 horas, reuniu-se para deliberar a defesa da monografia de conclusão de curso do discente VICTOR VILMARQUES CAPISTRANO PEDROSA, orientado pelo professor Joabe Bezerra de Jesus Júnior, sob título CSPDT: uma IDE para CSP, a banca composta pelos professores:

Luis Carlos de Sousa Menezes

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: 8,0 (oito)

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

O discente terá 10 dias para entrega da versão final da monografia a contar da data deste documento.

Luis Carlos de Sousa Menezes

LUIS CARLOS DE SOUSA MENEZES

JOABE BEZERRA DE JESUS JÚNIOR

Dedico aos meus pais que me fizeram ser o homem que sou hoje.

Agradecimentos

Agradecimentos: ao professor Joabe por ter me acompanhado nessa longa e árdua jornada; aos meus colegas da POLI e a minha companheira, Jéssica, peça fundamental no meu amadurecimento acadêmico.

Resumo

Este trabalho tem como finalidade apresentar o *CSPDT*, uma proposta de ambiente de desenvolvimento integrado (IDE) para a linguagem de programação *Csp*M, sendo esta a versão computacional da linguagem formal *Csp*. Essa IDE têm como objetivo contemplar características fundamentais de um editor de desenvolvimento, tais como: edição sensível a linguagem, autocomplemento, marcação de erros, *outline*, coloração de sintaxe e navegação.

Sua construção foi feita utilizando a ferramenta Eclipse como base, tendo auxílio do PDE *(Plugin Development Enviroment)* que, por sua vez, é a parte do Eclipse responsável pela criação de *plugins*. Neste artigo será mostrado todo o processo de desenvolvimento do *CSPDT*, abordando detalhadamente cada passo de sua construção. Ao final, será realizado um teste com o ambiente já desenvolvido, criando um código fonte (na linguagem *CSP*M) e mostrando a maneira em que ele será processado pela IDE. Com isso, espera-se que o *CSPDT* seja uma IDE viável e com recursos fundamentais para o desenvolvimento de códigos em *CSP*M.

Abstract

This work aims to present the *CSPDT*, a proposal of integrated development environment (IDE) for the CSPM programming language, which is the computer version of the formal language Csp. This IDE aim to contemplate fundamental characteristics of a development editor, such as language sensitive editing, assistance writing (autocomplete), error marking, syntax tree viewer (outline), rich formatting (syntax highlight), browsing and documentation. It's construction was fully Eclipse based, with assistance from the PDE (Plugin Development Environment) which, in turn, is part of the Eclipse responsible for creating plugins. In this article it's shown the entire development process of the *CSPDT*, addressing each step in details. At the end, there will be a test with the already developed environment, creating a source code (in CSPM language) and showing the way in which it will be processed by the IDE. Thus, it is expected that the *CSPDT* be a viable IDE, with basic resources for the development of codes in CSPM.

Sumário

Índice de Figuras

Tabela de Símbolos e Siglas

AST - *Abstract Syntactic Tree*

CSP - *Communication Sequential Processes*

CSPDT - CSP Development Tools

CSPM - *Communication Sequential Processes (machine readable)*

CZT - *Community Z Tools*

FDR - *Failures-Divergence Refinement*

IDE - *Integrated Development Environment*

JDT - *JAVA Development Tools*

MF - *Métodos Formais*

MVC - *Model View Controller*

OSGI - *Open Services Gateway Initiative*

PAT - *Process Analysis Toolkit*

PDE - *Plug-in Development Environment*

ProBE - *Process Behaviour Explorer*

SDK - *Software Development Kit*

WTP - *Web Tools Plataform*

1. Introdução

Em Métodos Formais (MF), especificações e implementações são representadas em um sistema lógico, permitindo assim, o desenvolvimento de provas matemáticas de teoremas que mostram que a implementação satisfaz a especificação (Aho e Ullman, 1995). As principais atividades em MF são:

- Escrever uma especificação formal;
- Prova de propriedades sobre a especificação;
- Cálculo Construção de um programa através de manipulações/transformações da especificação;
- Verificação do programa usando argumentos matemáticos.

Em cada etapa do desenvolvimento usando MF as ferramentas (Editores, Verificadores e Provadores de Teorema) tem papéis fundamentais no auxílio à construção de um sistema. Uma das grandes dificuldades de adoção de Métodos Formais em escala industrial é o trabalho de efetuar uma verificação. Kurshan diz que uma abordagem integrada ao processo de desenvolvimento se faz necessária. O autor também define um teorema que afirma que a Automação de Design Eletrônico Comercial, por exemplo, será inevitavelmente baseada em Métodos Formais devido à complexidade da tarefa, que torna a simulação e a análise de testes tarefas impraticáveis, como no caso do bug que gerou um custo de meio bilhão de dólares.

Muitos consideram que as técnicas de Métodos Formais só serão amplamente adotadas no desenvolvimento de sistemas quando tivermos um alto grau de mecanização do processo de prova e há muitos mitos sobre o uso de métodos formais. Esse importante ponto é citado por Kurshan. Ele menciona que os usuários ficam mais que contentes de verificar propriedades locais “não importantes” do sistema, desde que isso possa ser realizado automaticamente, sem “educação”. Assim, os editores encontrados em ambientes integrados de desenvolvimento (IDE) deveriam ser capazes de permitir tais tarefas.

O *CSP* (Communicating Sequential Processes) é uma linguagem formal cujo objetivo é descrever padrões de interação em sistemas concorrentes, simplificando toda a sua complexidade e tornando-a aplicável a sistemas computacionais. Para isso, o *CSP* usa o conceito de eventos e processos para a representação do comportamento do sistema, modelando-o de forma que todos os passos sejam demonstrados (Hoare, 2015).

O *CSP* possui algumas ferramentas, sendo destacado o FDR (específico para a linguagem *CSP*M) e o PAT (específico para a linguagem *CSP#*). O FDR (Failures-Divergence Refinement) é um model checker que, além de comprovar se os processos estão livres de dead-lock, pode verificar as suas propriedades através da relação de refinamento entre o modelo do sistema e o processo de captura de propriedades. Já o PAT (Process Analysis Toolkit) é uma ferramenta que serve para compor, simular e compilar sistemas concorrentes e de outros domínios. Ela implementa várias técnicas de model checking e verifica algumas propriedades, tais como alcance, presença de dead-lock, presença de divergência, checkagem de refinamentos entre outros (Carvalho, 2012).

A falta de uma IDE para a linguagem *CSP* dificulta o trabalho de especificação da linguagem, tornando o processo menos produtivo e amigável. Uma IDE serve como uma ferramenta para auxiliar, explorar ao máximo a produtividade dos desenvolvedores e apoiar a construção de uma codificação mais fácil de ser escrita e entendida.

1.1 Objetivos da ferramenta proposta

A falta de suporte de um ambiente de desenvolvimento para a linguagem *CSP*, pelos motivos já expostos anteriormente, será o foco deste artigo. Baseado no problema exposto pretende-se, com esse trabalho, desenvolver plugins para o eclipse que permitam estender o editor de desenvolvimento dessa plataforma para dar suporte a linguagem *CSP*.

Um ambiente de desenvolvimento fornece ao usuário as ferramentas necessárias para o desenvolvimento eficaz e eficiente de projetos, de acordo com suas necessidades. Para isso, é necessário que esse ambiente possua uma série de requisitos definidos, sendo eles: coloração de sintaxe (*syntax highlight*), autocomplemento (*autocomplete*), *outline,* auxílio à navegação (*browsing*), marcação de erros (*error marking*), *folding, auto ident, double click, debugger (animator), model checker e theorem prover (*no caso de linguagens formais*).*

Neste trabalho abordaremos o desenvolvimento dos principais requisitos da IDE proposta, o suficiente para que ela se torne uma ferramenta fundamentalmente funcional*.* Sendo assim, ela se limitará a englobar os requisitos de *highligt, autocomplete, outline, error marker e browsing.*

1.2 Metodologia

1.2.1 Natureza e abordagem

Nossa pesquisa usará uma metodologia aplicada, na qual estaremos preocupados em solucionar problemas reais do uso da linguagem *CSP*, isto é, criação produtiva de uma especificação formal. 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 através de entrevistas com estudiosos da área.

1.2.3 Procedimentos técnicos

O nosso estudo reúne característica 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 como fonte de conhecimento.

1.3 Construção da ferramenta

O primeiro momento do trabalho envolve um esforço para desenvolver a ferramenta proposta. O Eclipse Plug-in Development Environment (PDE) é um plugin que facilita bastante a construção de novos plugins através do fornecimento de ferramentas de edição e wizards. Com o PDE será possível automatizar e abstrair boa parte do desenvolvimento da ferramenta (editor) tornando o trabalho de criação, desenvolvimento, debug e testes mais produtivos. A primeira etapa para o desenvolvimento do plugin é identificar os seus pontos de extensão, registrando-os no arquivo chamado de *MANIFEST* (plugin.xml), é neste arquivo que será definido e declarado todos esses pontos referentes ao editor. O próximo passo é implementar a interface esperada, de acordo com os pontos de extensões, e codificar as funcionalidades da ferramenta. Para simplificar o desenvolvimento e aumentar a produtividade o Eclipse já fornece um framework, JFace Text, que permite o desenvolvimento de editores de texto com as características necessárias para a ferramenta proposta no trabalho. Por questões de conveniência, nosso editor, ao invés de implementar uma interface, irá estender a subclasse TextEditor, que fornece uma implementação base para editores.

1.4 Resultados esperados

Inspirado no Xtext espera-se como resultado um plugin capaz de ser integrado à plataforma Eclipse. Tal plugin fornecerá um ambiente integrado de desenvolvimento para *CSP*. Os benefícios da ferramenta gerada no trabalho são análogos aos do CZT (Community Z Tools). Almeja-se, ainda, que a ferramenta

desenvolvida possa, futuramente, ser incubada no projeto Eclipse e estar disponível para toda a comunidade, assim como a integração a ferramentas de verificação *CSP*.

1.5 Organização do trabalho

Este trabalho está organizado em cinco capítulos. O primeiro é a introdução, onde explicamos sobre o *CSP* e a falta de ferramentas para utilizá-lo, bem como a metodologia utilizada neste artigo e os resultados esperados. No segundo, abordamos o estado da arte, onde damos detalhes do ambiente em que a ferramenta será criada. Já no terceiro, mostramos de forma abrangente o desenvolver da ferramenta propriamente dita, fazendo menção a todas as etapas e artifícios utilizados para a sua construção. No quarto capítulo, será apresentado um caso de uso da ferramenta, representando seu funcionamento. E finalmente, concluímos o trabalho no capítulo cinco, destacando a importância da ferramenta criada e deixando aberto a possibilidade de continuação do projeto, com trabalhos futuros nessa área.

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 ideia de metamodelagem no Eclipse e, finalmente, geradores de compiladores e as particularidades do *CSPDT*.

2.1 Editores de programas

Um editor de programa é a parte principal de uma IDE (Ambiente de Desenvolvimento Integrado), onde são construídos os códigos-fonte dos sistemas, ou seja, são editores de textos aperfeiçoados com funções que tornam o processo de codificação mais eficiente. Geralmente, essas funções são: edição sensível à linguagem (*syntax highlighting*), autocomplemento (*autocomplete*), suporte à navegação (*browsing*), marcação de erros (*error marking)* e visualização da árvore sintática (*outline*).

Existem outras ferramentas que, junto com o editor, conpoem a IDE, tais como: compilador, linker, depurador (debugging), modelador, entre outros. A integração dessas ferramentas auxilia a produção de um código mais inteligível e faz com que suas tarefas (criação, revisão, análise, transformação e execução) sejam otimizadas.

2.1.1 Edição sensível à linguagem (*syntax highlighting*)

Essa propriedade do editor o faz fortemente conectado à sintaxe da linguagem em questão, tornando possível o uso de recursos como coloração do código (syntax highlighting), verificação de erros ortográficos e organização sintática (formatação).

2.1.2 Autocomplemento (*autocomplete*)

O mecanismo de autocomplemento (do inglês Autocomplete) é um dos recursos de uma IDE. Sua função é auxiliar o programador a recordar de trechos de código (funções, palavras chave, procedimentos, etc), provendo uma lista de sugestões enquanto ele está sendo digitado. Em algumas IDEs, o autocomplemento é inteligente o suficiente para aprender novas sugestões enquanto o código é produzido, o tornando mais adaptativo e eficiente.

2.1.3 Suporte à navegação (*browsing*)

É uma funcionalidade que possibilita a navegação através do código. O browsing faz com que tenhamos fácil acesso a declarações de processos e variáveis, referências e outras definições, geralmente clicando nos trechos em que queremos obter a informação, tornando mais fácil o entendimento do código.

2.1.4 Visualizador da árvore sintática (*content outline*)

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

2.1.5 Marcação de erros (*error marking*)

O *error marking* é uma funcionalidade de extrema importância numa IDE, pois é através dela que serão detectados e mostrados os erros recorrentes no código fonte.

2.2 Eclipse

O Projeto Eclipse foi originalmente criado pela IBM em novembro de 2001 e apoiado por um consórcio de vendedores de software (Borland, Merant, QNX, Rational, RedHat, SuSE e TogetherSoft), cujo objetivo era focado em construir uma plataforma aberta de desenvolvimento composta por ferramentas e runtimes para a construção, implantação e gestão de software em todo seu ciclo de vida. Em 2004

foi criada a Eclipse Foundation, uma organização sem fins lucrativos independente, servindo para administrar e gerir a comunidade Eclipse como um todo, dando suporte a parceiros estratégicos e fornecedores de extensões/plugins. Ao longo do tempo, o Eclipse tornou-se um dos IDEs mais utilizados no mundo, especialmente por ter a propriedade de desenvolvimento baseado em plugins, fornecendo um grande suporte aos programadores para realizar diferentes projetos de formas diversas.

O Eclipse foi criado para desenvolver programas na linguagem JAVA, porém, por ser uma plataforma multilinguagem, ele suporta outras linguagens de programação tais como C, C++ e etc. Para que seja possível o desenvolvimento de softwares de outras linguagens no Eclipse, basta que instalemos seu respectivo plugin na plataforma.

A plataforma do Eclipse fornece vários pacotes de desenvolvimento, tais como Eclipse JDT (JAVA Development Tools). O JDT é uma ferramenta que implementa a IDE JAVA, dando suporte de qualquer aplicação nessa linguagem, incluindo também os plugins do Eclipse. Isso permite ao Eclipse se tornar por si só um ambiente de desenvolvimento completo. Além do JDT como um dos pacote de desenvolvimento, temos também o SDK (Software Development Kit), WTP (Web Tools Plataform) que serve para desenvolver linguagens voltadas à sistemas web e o compilador do JDT, que é o próprio compilador JAVA.

Podemos representar o Eclipse (IDE) como um pequeno núcleo e um imenso conjunto de plugins, que trabalham em conjunto para fornecer as diversas funcionalidades da IDE. Esses plugins compoem praticamente toda a plataforma, forneecendo todos as ferramentas necessárias para o desenvolvimento do sistema. Na figura 1 temos uma ilustração que representa a arquitetura da plataforma e dos seus principais componentes.

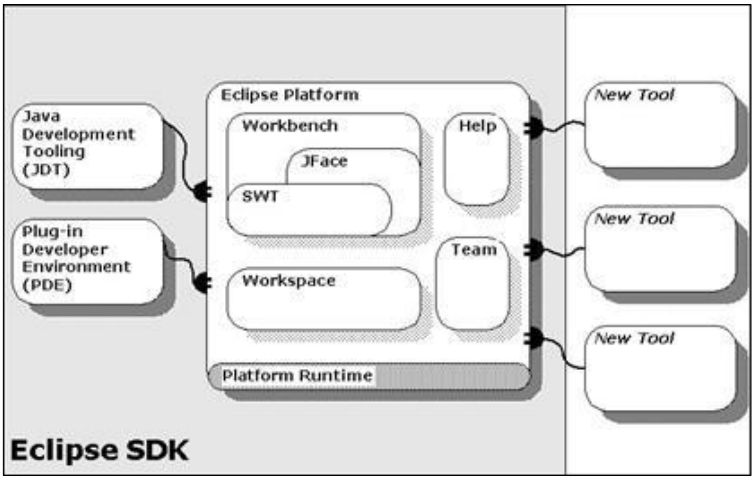

Figura 1. Arquitetura da plataforma eclipse.

2.2.1 Runtime da Plataforma

O Runtime é responsável pelo gerenciamento de todos os plugins disponíveis anexados à plataforma, isto é, cabe a ele a tarefa de preparar, integrar e executar todos os plugins instanciados, fazendo o registro de qualquer situação adversa que possa vir a ocorrer. Quando o Eclipse é inicializado, o Runtime acessa um arquivo chamado plugin.xml onde constam todas as informações necessárias para a execução dos plugins, ou seja, como eles estendem suas extensões, quais suas extensões e como são implementadas suas funcionalidades.

2.2.2 OSGI

O OSGI (Open Services Gateway Initiative) é um sistema modular que implementa um completo e dinâmico modelo de componentes, fazendo com que o software seja dividido em partes (módulos) mais independentes. Essa modularização faz com que o software tenha um menor grau de acoplamento e alto grau de coesão, reduzindo sua complexidade.

Os componentes gerados pelo OSGI são chamados *bundles,* sendo estes compostos por coleções de classes, *jars* (arquivo no formato JAVA), arquivos de configuração e outros recursos adicionais. Cada bundle é representado por um arquivo chamado MANIFEST.MF, onde constam informações como nome,

descrição, versão, ativador e etc. Para utilizar o modelo de OSGI no Eclipse temos que implementá-la através de uma ferramenta, sendo a mais famosa entre elas chamada de Equinox. Ele é o ambiente de execução modular no coração da IDE, implementando todas as funcionalidades necessárias e opcionais para o OSGI.

2.2.3 Workbench

Como o próprio nome sugere, o workbench é a “mesa de trabalho” do Eclipse, ou seja, onde todo o trabalho de desenvolvimento será feito e onde ficam localizadas as ferramentas que serão utilizadas no processo. Ele é a janela principal da IDE, contendo as barras de menu e de ferramentas, views e editores (instrumentos para manipular os dados e recursos) e uma ou mais *perspectivas*. Perspectivas são visões particulares de determinadas tarefas, onde cada uma tem seu conjunto de ferramentas e editores.

A workbench fornece meios para que possamos navegar por todos os projetos inicializados no Eclipse, tendo acesso rápido a suas classes, métodos e pacotes. Na figura 2 temos uma visão geral da workbench.

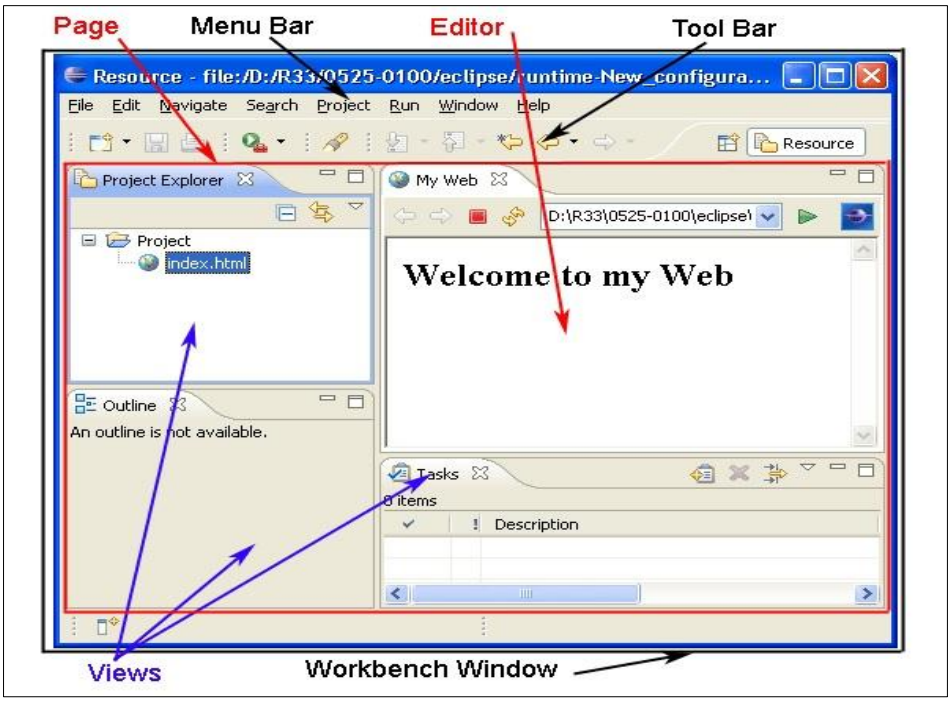

Figura 2. Workbench do Eclipse.

2.2.4 Workspace

A primeira coisa a ser feita ao executar o Eclipse é definir o local da workspace que, por sua vez, se trata do diretório onde ficam localizados todos os projetos que estão sendo desenvolvidos, bem como as preferências configuradas no programa (layout, por exemplo). O workspace é o espaço físico onde se está trabalhando, ou seja, é o espaço no disco que será destinado ao armazenamento dos arquivos referentes aos projetos. Ele organiza seu conteúdo de maneira hierárquica, isto é, os projetos ficam no topo e dentro dele você tem as pastas e arquivos. Vale ressaltar que é permitido criar mais de uma workspace no Eclipse, desde que seu novo diretório seja devidamente referenciado ao executar a IDE. Na figura 3 mostramos a seleção do diretório do workspace.

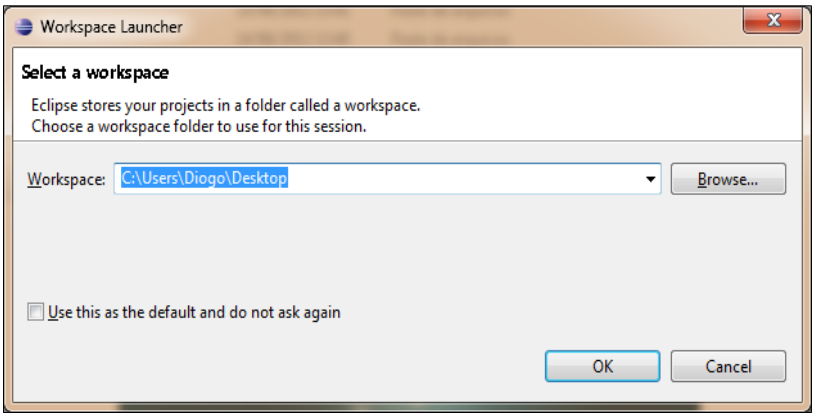

Figura 3. Selecionando o workspace.

2.2.5 Plugins

Por definição, entende-se como *plugin* ou módulo de extensão um tipo de ferramenta que, ao ser acoplado a um determinado sistema, provê novas funcionalidades adicionais e recursos ao mesmo. Graças a ele, não se faz necessário a obtenção de outros softwares para se conseguir uma determinada funcionalidade, basta apenas conectar seu respectivo plugin ao sistema atual. Os plugins são geralmente leves, de fácil instalação e manuseio, sendo amplamente usados por diversos tipos de aplicação, tais como:

- **Emails:** encriptação e desencriptação de emails;
- **Editores de áudio:** geração, processamento ou análise de sons;
- **Softwares gráficos:** suporte a novos formatos de arquivos e imagens;
- **Editores de texto:** suporte a novas linguages de programação.

O plugin pode utilizar serviços providos pela aplicação principal, incluindo um modo de efetuar seu registro no sistema e um protocolo para troca de dados com o mesmo. Ao contrário da aplicação principal, os plugins não funcionam de maneira totalmente independente, ou seja, precisam dos serviços disponibilizados por ela para funcionar. Podemos ter uma ideia da integração do plugin no sistema através da figura 4.

2.2.6 Editores de programa no Eclipse

O editor de programa corresponde a parte principal de desenvolvimento do código fonte. Sendo uma parte do *workbench*, o editor de programa assemelha-se a um editor de textos comum, servindo para codificar o sistema e fornecendo todas as ferramentas apropriadas para este processo. No Eclipse existem maneiras alternativas de se codificar um programa, uma delas é através da edição visual, atráves de plugins baseados em *swing*. Apenas um editor pode ser aberto para cada página do workbench.

O ponto de extensão do workbench *org.Eclipse.ui.editors* é usado por plugins para adicionar editores ao mesmo. No Eclipse, é possível criar novos editores para linguagens de codificação, utilizando determinados plugins. Para isso, esses plugins precisam registrar a extensão do editor no seu respectivo plugin.xml, bem como as suas informações de configuração, sendo estas composta por nome, rótulos, classes de implementação, ícones usados na workbench entre outros.

Existe uma ferramenta própria do Eclipse para criar editores de texto com funções básicas como copiar, recortar, colar, bem como funções avançadas como o *syntax highlight* e formatação de código, chamada de Jface text. Através dela é possível criar editores para qualquer linguagem de programação no Eclipse. Na figura 5 podemos ver um exemplo simples da sua utilização

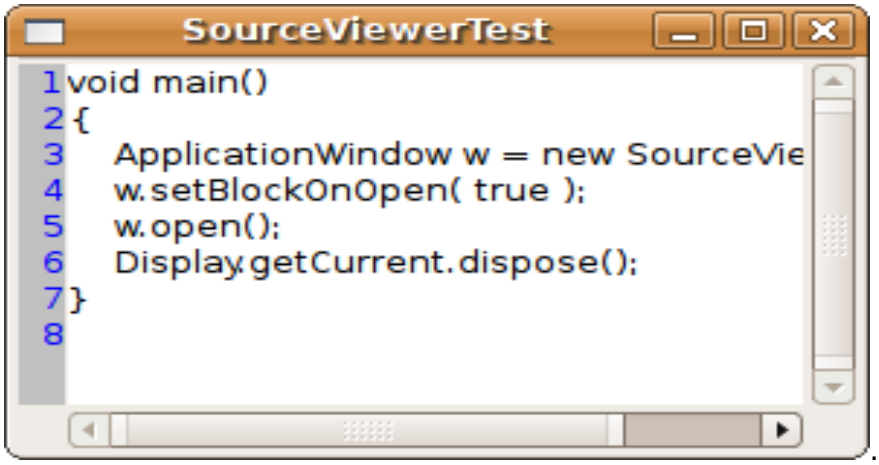

Figura 4. Exemplo Jface text.

2.3 CSP

Considerado como uma das principais teorias matemáticas de concorrência, o *CSP* é uma linguagem formal que tem como finalidade descrever padrões de interação em sistemas concorrentes, paralelos e distribuídos, baseado em trocas de mensagens através de canais. Essa linguagem tem como principal vantagem verificar e especificar precisamente os comportamentos dos processos envolvidos em cada estágio do programa, além de possuir um alto nível de abstração.

Para representar as relações entre os diferentes processos e a maneira com que eles interagem com o seu ambiente, o *CSP* utiliza operadores de processos algébricos[1], como mostra a figura 5. Dessa forma, sistemas complexos podem ser facilmente modelados usando alguns elementos primitivos, sendo eles classificados em duas classes:

- **Eventos:** são resultados de uma ação, representando comunicações ou interações no sistema. São indivisíveis e instantâneos;
- **Processos primitivos:** representam comportamentos fundamentais: por exemplo, STOP e SKIP.

[1] http://www.fsel.com/documentation/fdr2/html/fdr2manual_28.html#Syntax-Reference

As principais ferramentas para a análise das especificações do *CSP* são o FDR e o ProBE (Process Behaviour Explorer). Essas ferramentas analisam programas escritos em *CSP*M, sendo esta uma versão do *CSP* que combina os seus operadores algébricos com uma linguagem de programação funcional, o tornando *machine-readable*.

CSP	**CSP$_M$**	**CSP#**	**Description**
$STOP$	$STOP$	$Stop$	deadlock
$SKIP$	$SKIP$	$Skip$	termination
$CHAOS$	$CHAOS(A)$	-	chaotic process
$a \rightarrow P$	$a \rightarrow P$	$a \rightarrow P$	event prefixing
$c!e \rightarrow P$ $c?x \rightarrow P$	$c?x?x' : V!e \rightarrow P$	$c!e \rightarrow P$ $c?[b]x \rightarrow P$	channel communication
$P \,\square\, Q$	$P \,[]\, Q$	$P \,[\bullet]\, Q$	external choice
$P \sqcap Q$	$P \,\|\sim\|\, Q$	$P <> Q$	internal choice
$P;\ Q$	$P;\ Q$	$P;\ Q$	sequential composition
$P \setminus A$	$P \setminus A$	$P \setminus A$	hiding
$x := e$	-	$x := e$	assignment
$P \lhd b \rhd Q$	$if\ b\ then\ P\ else\ Q$	$if\ b\ then\ P\ else\ Q$	conditional choice
$P \parallel Q$	$P[\|\ A\ \|]Q$ $P[A \parallel A']Q$ $P[c <-> c']Q$	$P \parallel Q$	parallel composition
$P \,\|\|\|\, Q$	$P \,\|\|\|\, Q$	$P \,\|\|\|\, Q$	interleaving
$P \triangle Q$	$P /\!\!\setminus Q$	$P\ interrupt\ Q$	interrupt

Figura 5. Sintaxe similar entre CSP, CSPm e CSP#

3. *CSPDT*

Neste capítulo serão abordados todas as etapas da implementação da IDE proposta, bem como os detalhes de cada uma delas. Será mostrado como todas as suas funcionalidades foram desenvolvidas e como elas estão interconectadas entre si, de modo que a finalidade com a qual ela foi proposta seja alcançada. Ao final, será verificado se a IDE conseguiu englobar os conceitos e funcionalidades que um editor padrão possui, de acordo com o que foi apresentado no capítulo anterior.

3.1 Requisitos da IDE *CSPDT*

Como visto na seção 1.1, uma IDE eficiente tem uma série de requisitos definidos. Vimos também que neste trabalho serão desenvolvidos apenas os principais, o que tornaria o *CSPDT* (*CSP* Development Tools) uma IDE essencialmente funcional. Esses requisitos são representados no diagrama da figura 6.

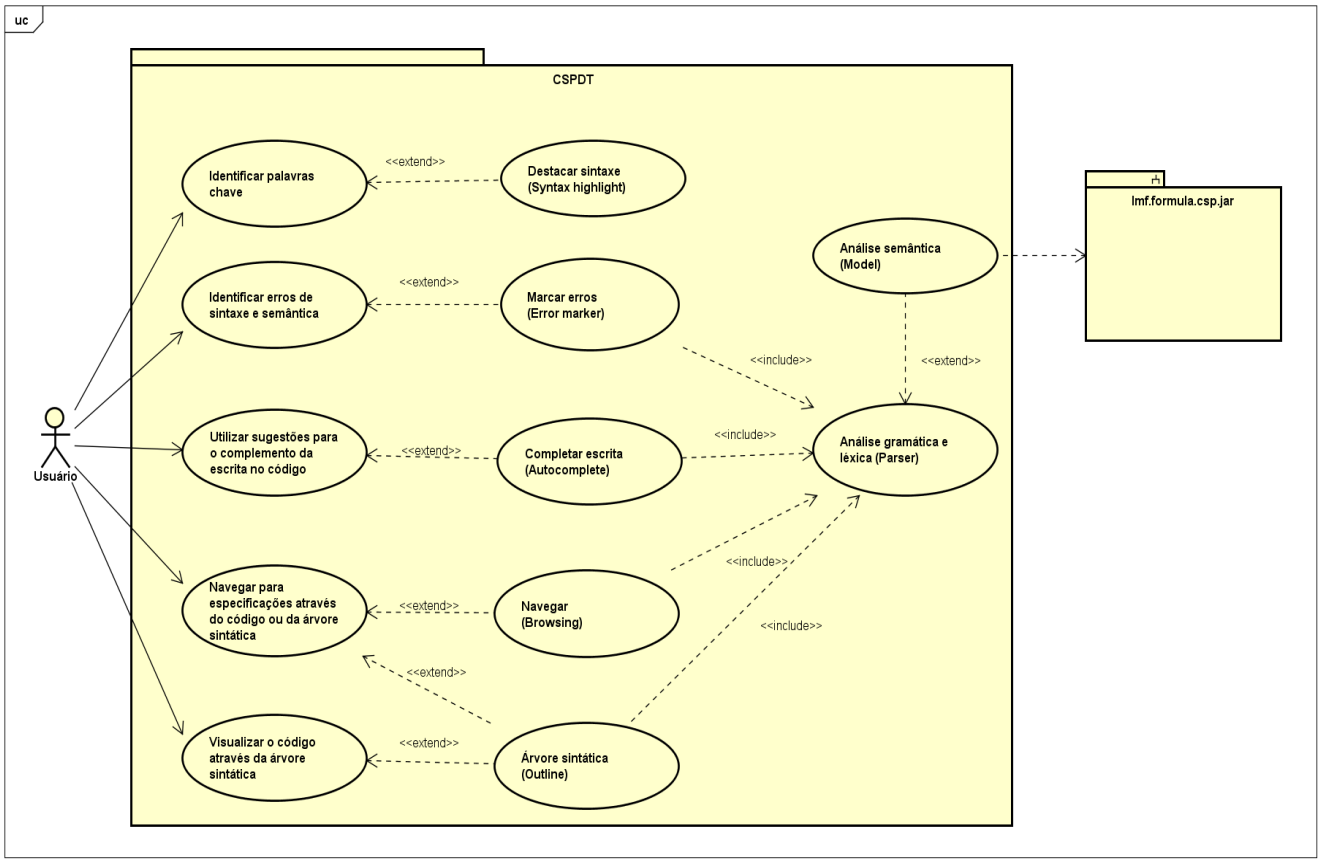

Figura 6. Diagrama de requisitos.

3.2 Arquitetura da IDE *CSPDT*

A IDE proposta é composta por uma estrutura de três módulos conectados de maneira hierárquica, são eles o Editor (*CspMEditor)*, Core (*CspMModel)* e Parser (*lmf.formula.csp.jar)*, onde cada um deles tem uma classe plugin associada. Na figura 7 é apresentada a estrutura dos módulos, bem como as classes plugin associadas, onde o Editor dependerá do Core que, por sua vez, fornecerá o acesso ao arquivo *jar* do Parser.

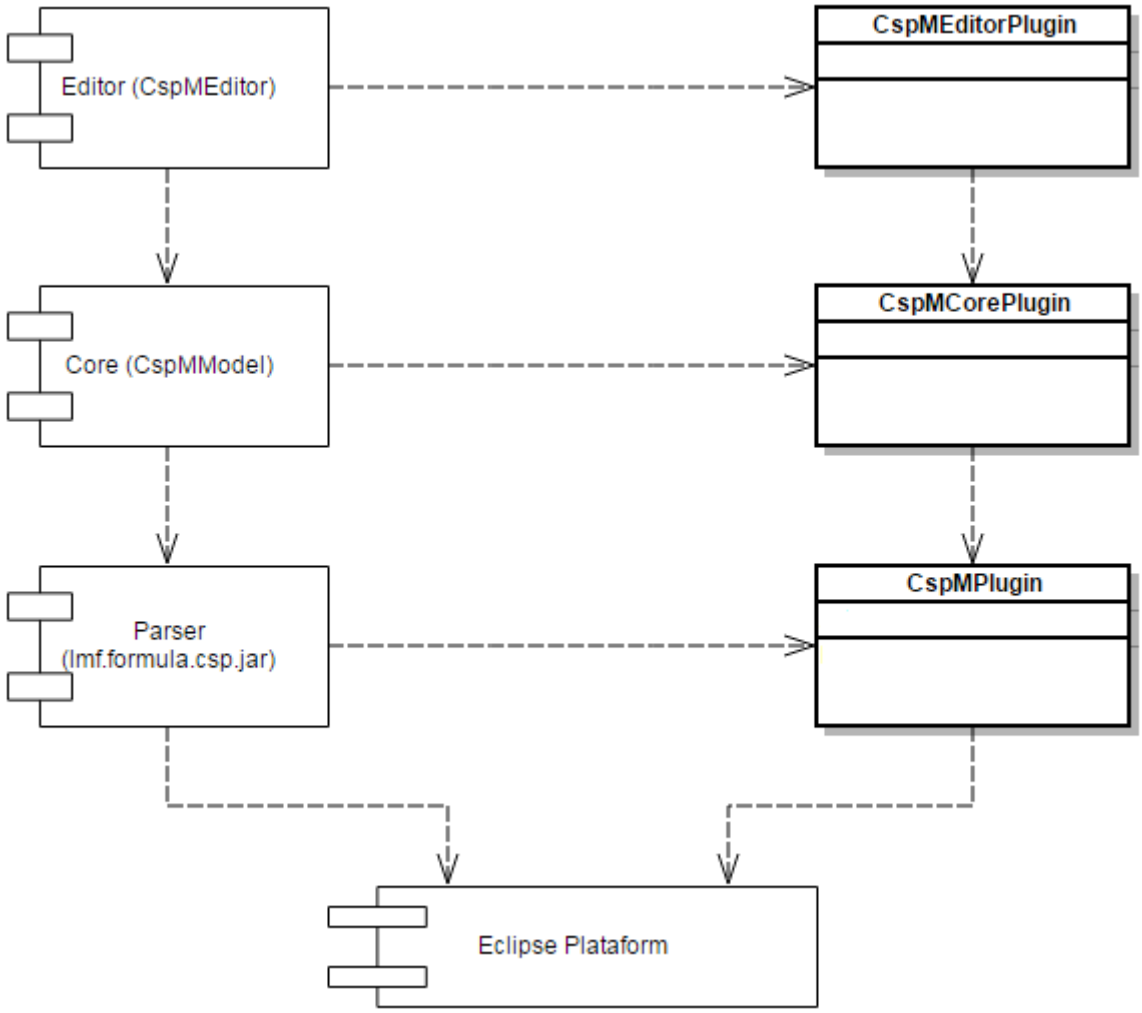

Figura 7. Estutura modular da IDE.

Todo o desenvolvimento da IDE será feito através da ferramenta integrada do Eclipse para criação de plugins, a PDE. Para desenvolver um plugin utilizando essa ferramenta, primeiramente deve-se criar os arquivos MANIFEST (*plugin.xml*), definindo e declarando os pontos de extensão necessários (Beck e Gamma, 2003). Em seguida, é necessário implementar a interface, de acordo com as extenções no *plugin.xml*, e por fim codificar as funcionalidades da ferramenta.

3.3 CSP Parser

O Parser (*lmf.formula.csp.jar*) será o módulo responsável pelo encapsulamento da implementação do *typechecker CSPM*. Esse módulo será chamado sempre que for necessária a utilização dos recursos de análise sintática (cuja sintaxe foi apresentada no capítulo 2) e semântica. Estão incluídos como serviços desse módulo a análise léxica, análise sintática, geração da AST (*Abstract Syntactic Tree*) e mecanismos para navegar através da árvore gerada.

3.4 CspMEditor Plugin

No *CspMEditor* definimos como será implementado o editor em si, ou seja, o ambiente onde os códigos fonte serão escritos, baseados na gramática do *CSP*. Nesta parte serão implementados todos os recursos básicos de um editor, tais como edição sensível à linguagem (*syntax highlighting*), autocomplemento (*autocomplete*), suporte à navegação (*browsing*), visualização da árvore sintática (*content outline*) e marcação de erros (*error marking*). Todas as informações referentes a esse plugin como nome, versão, extensões e autoria podem ser encontradas no seu arquivo MANIFEST.MF.

Para criar o editor, primeiramente devemos declará-lo no MANIFEST.MF (Beck e Gamma, 2003). O editor é definido no arquivo *plugin.xml* através do ponto de extensão *org.Eclipse.ui.editors* (Realsolve, 2006), sendo este o ponto principal no que diz respeito a criação de editores no Eclipse. O *plugin.xml* é composto por vários atributos, cada um deles fornece uma informação do editor. Um deles é o *point,* é uma referência para o ponto de extensão do editor do workbench. Temos também o *name*, que representa o nome do editor. Já o *extensions* especifica a extensão do arquivo e que consequentemente estará atrelado ao editor particular. O atributo *icon* representa o ícone que estára nos arquivos reconhecidos pelo editor. O *class* aponta para a classe principal que conterá a codificação do nosso editor e deverá estar em conformidade com uma interface esperada. E o *id*, por sua vez, é um identificador único do plugin desenvolvido (Beck, Gamma e Erick, 2003). Na figura 8, podemos

ver uma parte do *plugin.xml* onde são mostrados os atributos mencionados anteriormente.

```
4   <extension
5         point="org.eclipse.ui.editors">
6      <editor
7            name="CSPM Editor"
8            extensions="cspm"
9            icon="icons/csp.png"
10           contributorClass="com.fware.cspdt.cspm.editor.CspMFileEditorContributor"
11           class="com.fware.cspdt.cspm.editor.CspMEditor"
12           id="CspFileEditor">
13     </editor>
14  </extension>
```

Figura 8. 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 *CspMEditor* deveria implementar a interface *ItextEditor*. Porém, faremos com que o *CspMEditor* estenda de uma subclasse chamada *TextEditor,* que por sua vez já fornece uma implementação base para editores. Feito isso, podemos começar a desenvolver o editor através de um framework fornecido pelo Eclispe chamado *Jface Text (SWT)*, cuja funcionalidade é tornar mais simples e intuitivo a criação de novas ferramentas de edição, mantendo todas suas as características.

A classe *TextEditor, cuja* qual a classe *CspMEditor* 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 9 fará uma ilustração da estrutura dos componentes do *Csp*MEditor, utilizando diagramas de classe, cada um deles será explicado nas seções seguintes.

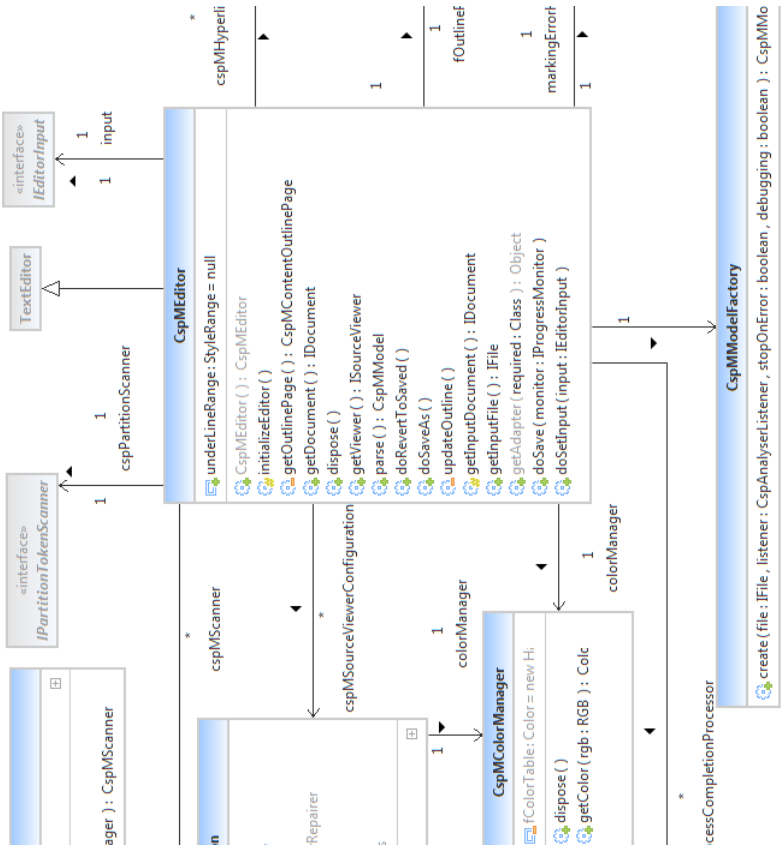

Figura 9. Definição e configuração do esquema de particionamento.

De acordo com a figura 9, podemos ver que o *Csp*MEditor é responsável por controlar todas as principais funcionalidades do editor: configuração do outline, marcação de erros, coloração do código entre outros. Nesta classe são definidos os procedimentos que devem ocorrer ao criar um novo arquivo, bem como o que deve acontecer quando um arquivo previamente criado é aberto ou salvo. Aqui também é definido quando o parser do *CSP* deve ser chamado, realizando a criação da sua respectiva *AST.*

3.4.1 Padrão MVC

O padrão arquitetural MVC (Model View Controller) é uma forma de quebrar uma aplicação, ou até mesmo um pedaço da interface de uma aplicação, em três partes: o modelo, a visão e o controlador (Dooley, 2011).

O *Model* é a parte que representa os dados, ou seja, todas as informações referentes à escrita, validação e leitura dos dados está dentro desta camada. O

Model sabe o que o aplicativo quer fazer e é a principal estrutura computacional da arquitetura, pois é ele quem modela o problema que está se tentando resolver.

Já o *Controller* é a camada intermediadora, sendo a parte que interpreta as informações de entrada (mouse ou teclado) fornecida pelo usuário, mapeando-as em comandos que são enviados para o *Model* e/ou para o *View*.

Por último temos o *View*, essa camada é a responsável pela apresentação das informações ao usuário, através de uma combinação de gráficos e textos. Sua função é limitada a receber as informações do *Controller* e imprimí-las na tela. Ela também se comunica de volta com o modelo e com o controlador para reportar o seu estado.

Sendo uma arquitatura amplamente usada, o MVC tem como principais vantagens a facilidade em reaproveitar o código, facilidade na adição e manutenção de recursos e em manter o código sempre limpo.

O *Jface Text* baseia-se no MVC. 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 view e o modelo. A Figura 10 ilustra o modelo MVC do *JFace Text* (Rich Client, 2006, apud BARACUHY, 2011, p. 23).

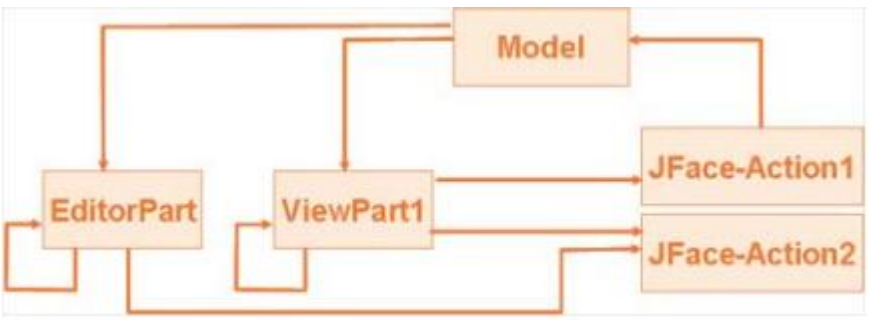

Figura 10. Comunicação entre componentes JFace no modelo MVC.

Conforme ilustrado pela figura 10, 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* representam views. Os controllers manipulam os models que por sua vez propagam as mudanças para as views.

3.4.2 Documents

O *Document* é uma entidade que tem como funcionalidade a manipulação e gerência dos textos armazenados, provendo vários métodos e serviços para este propósito. Através dela, é possível ter acesso a todo o texto de maneira rápida e detalhada, bem como controle total sobre o mesmo. Podemos obter informações sobre o que está escrito em determinada linha, editá-la, realizar pesquisas em seu conteúdo e particioná-lo. Para usufruir dos serviços oferecidos por essa entidade, é ncessário implementar a interface *org.Eclipse.jface.text.IDocument.*

Cada elemento do texto possui uma determinada posição na tela e toda vez que ele é mudado, essa posição é atualizada, através do *IPositionUpdater*. Esses atualizadores de posições são gerenciados numa lista que define a sequência na qual eles serão invocados. As posições são agrupadas em categorias que, por sua vez, são listas ordenadas de posições.

O *IDocument* permite que o usuário utilize delimitadores de texto, através do *TextUtilities.getDefaultLineDelimiter(IDocument).* Ele também fornece mecanismos de tratamento de erro, como o *BadLocationException*, que é chamado quando se tenta acessar conteúdo fora dos limites do documento. O propósito desse tipo de tratamento é, no geral, a preparação do documento para o acesso *multi-thread* (processamento paralelo em multi tarefa).

3.4.3 Document *Provider*

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* faz a representação textual de um arquivo armazenado no diretório do projeto, ou seja, ele tem as ferramentas necessárias para torná-lo um documento editável dentro do Eclipse. Através dele podemos abrir o arquivo no editor, realizar as edições no seu conteúdo e monitorar todas as modificações feitas desde seu carregamento. Para indicar que o conteúdo textual foi modificado, o Eclipse mostra um asterisco (*) na aba do arquivo atual, demonstrando que ele foi alterado mas ainda não foi devidamente salvo.

Para usufruir dos serviços do *DocumentProvider* é necessário implementar sua interface, a *IDocumentProvider*, que fica situada no pacote *org.Eclipse.ui.texteditor* do Eclipse. O pacote *org.Eclipse.ui.editors.text*, por sua vez, fornece a classe *FileDocumentProvider* que é uma implementação desta interface. Essa classe fornece mecanismos para carregar documentos a partir do sistema de arquivos.

Neste trabalho será utilizada a implementação fornecida pela classe *FileDocumentProvider*, que será estendida através da classe *CspMDocumentProvider,* com o intuito de fornecer os serviços de particionamento que serão discutidos com mais detalhes 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 CspMSourceViewerConfiguration

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. O *ITextViewer* também provê métodos de gerenciamento do conteúdo textual como auto identação, comportamentos em cliques duplos, 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 *Source Viewer* do Eclipse vem com uma configuração padrão que deverá ser modificada para o editor proposto neste trabalho, com o intuito de adequá-lo a linguagem *CSP*M. O *JFace* fornece meios para realizar essa mudança através da extensão da classe *SourceViewerConfiguration* e a redefinição dos métodos necessários. O editor deste trabalho será configurado na classe *CspMSourceViewerConfiguration* que herda, naturalmente, da classe *SourceViewerConfiguration* do Eclipse. Esta classe realiza um papel importante neste projeto, pois além de ser responsável por uma série de recursos da ferramenta, é também responsável pela configuração do editor. Funcionalidades como *syntax highlighting*, auxílio à escrita (autocomplete), auxílio à navegação e duplo clique também 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, *Csp*MEditor, pelo fato de ser uma subclasse de *AbstractTextEditor*.

3.4.5 A Classe CspMEditor

Como foi dito anteriormente, a subclasse *TextEditor* é a parte principal do framework de texto do Eclipse, ele que será responsável pelo modo em que o texto será impresso na tela, desde detalhes de baixo nível até a parte de coloração e formatação das partições do texto. A classe *CspMEditor* estenderá essa subclasse e será responsável por criar o *source viewer* e os outros recursos que são essenciais em um editor de texto. O *CspMEditor* implementa a interface *IEditorPart* que, por sua

vez, é a base abstrata de implementação de todos os editores no Eclipse, estando situado na *org.Eclipse.ui.IEditorPart*.

Ao abrir um arquivo na *workbench* que tem sua extensão já declarada, é criado e adicionado um componente *editor part*, baseado na respectiva extensão. O *workbench* do Eclipse representa uma instância de *IWorkbenchPage* que, por sua vez, é um conjunto de *IWorkbenchPart,* sendo esta a interface que proverá as funcionalidades dos componentes visuais nas páginas da *workbench.* Ela tem dois subtipos: *view* e *editor*, definidos como *IViewPart* e *IEditorPart.* 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 essenciais e de maior relevância da classe *CspMEditor* para a IDE proposta, destacam-se: o *initializeEditor,* que serve para criar o seu próprio *sourceViewer* do editor, no caso do padrão não ser suficiente para as necessidades do projeto; o *getAdapter,* que inicializa o outline e o *parse,* responsável pela geração da árvore sintática (*AST*) do código e de sua respectiva geração de marcadores de erros.

3.4.6 Particionamento

Um particionador é responsável por dividir o documento em regiões distintas e não sobrepostas de acordo com as regras da linguagem em questão, essas regiões são chamadas de *partições*. O editor, primeiramente, divide o documento em partições, logo em seguida ele analisa cada partição gerada e verifica onde a coloração de sintaxe ou outras funções devem ser utilizadas. Após o particionamento ficará claro para o analisador de script quais regiões são códigos funcionais ou simplesmente comentários, por exemplo, também conseguirá distinguir palavras chaves e strings. Uma partição é representada pela interface *ITypedRegion* que fornecerá informações relativas ao tipo e tamanho de cada região.

Na IDE proposta, o particionador será definido e configurado na classe *CspMDocumentProvider,* a partir do método *createDocument*. A Figura 11 e 12

exibem os trechos do código que configuram o esquema de particionamento do editor da IDE.

```
12     protected IDocument createDocument(Object element) throws CoreException {
13         IDocument document = super.createDocument(element);
14
15         if (document != null) {
16             IDocumentPartitioner partitioner = new CspMPartitioner();
17             partitioner.connect(document);
18             document.setDocumentPartitioner(partitioner);
19         }
20
21         return document;
22     }
23 }
```

Figura 11. Definição e configuração do esquema de particionamento

```
 7 public class CspMPartitioner extends FastPartitioner {
 8
 9     public CspMPartitioner() {
10         super(CspMEditor.getCspPartitionScanner(), CspMPartitionScanner.PARTITION_TYPES);
11     }
12 }
```

Figura 12. Definição e configuração do esquema de particionamento (2)

Assim que uma instância da interface *IDocumentPartitioner* é implementada, a IDE já pode utilizar os seus recursos de particionamento. Sua implementação deve ser feita através da subclasse *CspMPartitioner* que, por conveniência, estende a classe *FastPartitioner,* sendo esta fornecida pelo *Jface Text.* A *FastPartitioner* é uma implementação padrão de um particionador de documentos no Eclipse.

Conforme mostrado na Figura 11, o método *createDocument* cria uma instância do *CspMPartitioner* que precisa ser configurado (receber no construtor) um *IPartitionTokenScanner* (*CspMPartitionScanner*) e um array de string que contém os tipos de partições suportadas, o que acaba sendo feito conforme mostra a figura 12. O *IPartitionTokenScanner* escaneia o documento e determina todo o seu particionamento retornando *tokens* que, por sua vez, precisam retornar o tipo da sua respectiva partição. Para fazer esse escaneiamento, o Eclipse fornece um *scanner* baseado em regras chamado de *RuleBasedPartitionScanner*, que implementa a

interface *IPartitionTokenScanner*. Esse *scanner* vem com uma configuração padrão que deverá ser personalizada de acordo com as necessidades do projeto, através da subclasse *CspMPartitionScanner*. Também é necessário chamar o método *connect* para associar o particionador a uma instância de *IDocument*.

Na classe *CspMPartitionScanner*, que estende o *RuleBasedPartitionScanner*, são definidas constantes de strings que representam diferentes tipos de partições. A figura 13 mostra uma parte do código em que se faz a definição dos tipos de partições, são elas: CSPM_DEFAULT_CONTENT_TYPE (conteúdo padrão), CSPM_COMMENT_CONTENT_TYPE (conteúdo de comentários de uma linha única) e CSP_MULTILINE_COMMENT_CONTENT_TYPE (conteúdo de comentários de múltiplas linhas). É necessário fazer com que os tokens, que representam cada tipo de partição, sejam reconhecidos no decorrer da análise do documento. Para isso, a classe utilizada para o particionamento contém mecanismos de reconhecimento dos *tokens* através da definição de um conjunto de regras.

```
22    public CspMPartitionScanner() {
23        //IToken defaultToken = new Token(CSPM_DEFAULT_CONTENT_TYPE);
24        IToken commentToken = new Token(CSPM_COMMENT_CONTENT_TYPE);
25        IToken multiLineCommentToken = new Token(CSPM_MULTILINE_COMMENT_CONTENT_TYPE);
26
27        IPredicateRule[] predicateRules = new IPredicateRule[] {
28            // Add rule for single line comments.
29            new EndOfLineRule("--", commentToken),
30
31            // Add rules for multi-line comments and javadoc.
32            new MultiLineRule("{-", "-}", multiLineCommentToken) };
33
34        setPredicateRules(predicateRules);
35    }
36 }
```

Figura 13. Definição de tokens e regras de tipos de partições para o *CSP*M.

A Figura 13 mostra o construtor padrão sendo sobrescrito para definições das regras do editor da IDE proposta. Na função *predicateRules* são definidos as regras (rules) referentes aos códigos comentados. A *EndOfLineRule* define que o símbolo "**--**" delimitará uma linha de código comentado, já o *MultiLineRule* define o escopo de códico comentado usando os símbolos "**{-**" e "**-}**" como limitadores.

Cada vez que o documento é analisado uma verificação de regras é feita, com o intuito de constatar se alguma delas será utilizada. Caso ela tenha sucesso nessa verificação, uma instância de *token* do tipo da partição é retornado junto com informações da posição e tamanho dela, sendo em seguida 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 Syntax Highlighting

A *Syntax Highlighting* é um recurso muito importante em uma IDE, permitindo que determinadas palavras do código fonte tenham um destaque (mudança na cor e realce em negrito, por exemplo) baseado na sua respectiva linguagem de programação, o que torna o código muito mais fácil de ler e entender. Seu processo envolve mecanismos semelhantes aos procedimentos de criação de partições abordados anteriormente, pois também envolve ideias parecidas com as de *tokens* e regras. O Eclipse fornece o componente *presentation reconciler* (*IPresentationReconciler*), que define e mantém a representação do texto na presença de mudanças aplicadas ao documento, sempre monitorando-as. Esse recurso tem também a função de definir um conjunto de *tokens* que atribuem cores e estilos de fontes para uma determinada região de texto.

O *presentation reconciler* é semelhante ao particionador, porém ela divide as partições específicas em *tokens,* onde estes terão atributos associados que definirão os estilos de exibição. O *IPresentationReconciler* trabalha em conjunto com duas outras entidades: *IPresentationDamager* e o *IPresentationRepairer*. Através deles é possível manter a apresentação visual do documento enquanto ele está sendo editado pelo usuário.

Os *damagers (IPresentationDamager)* determinam a região do documento que deve ser reconstruída devido a alguma mudança sofrida, sendo cada um deles específicos para um determinado tipo de conteúdo no texto ou uma região específica dele. Ele tem como retorno uma *IRegion,* ou seja, uma região que sofreu a mudança (o “dano”) para o *repairer* (*IPresentationRepairer*). O *repairer,* por sua vez, será

responsável por fornecer todas as informações necessárias da região “danificada”, para que sejam descritos todos os reparos que deverão ser feitos.

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 IDE proposta 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.*

A classe *DefaultDamagerRepairer* gerencia todos os detalhes referentes aos danos e reparos no conteúdo do documento, de acordo com as regras definidas no *CspMPartitionScanner.* Ele 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. Em contrapartida, temos a classe *NonRuleBasedDamagerRepairer,* que é utilizada para aplicar a devida coloração no texto sem precisar de regras, ao contrário da *DefaultDamagerRepairer.*

Na classe *CspMScanner* são definidas as palavras chaves do *CSP*M através de uma classe Enum, chamada *KEYWORDS*. Em seu construtor são criados *tokens* que serão associados a alguma regra de identificação de sequência de caracteres, assim como serão também associados a alguma cor predefinida. A classe *CspMColorManager* é responsável por tratar as vinculações das cores, que são fornecidas através da tela de preferências da IDE. A Figura 14 exibe um trecho de código que ilustra a criação de *tokens*, associação a cores e definição de regras:

```
32    public CspMScanner(CspMColorManager colorManager) {
33        prefs = CspMEditorPlugin.getDefault().getPreferenceStore();
34        // A token that defines how to color normal text.
35        defaultToken = new Token(new TextAttribute(colorManager.getColor(PreferenceConverter.getColor(
36                prefs, CspMEditorPreferenceConstants.COLOR_DEFAULT))));
37
38        // A token that defines how to color numbers.
39        numberToken = new Token(new TextAttribute(colorManager.getColor(PreferenceConverter.getColor(
40                prefs, CspMEditorPreferenceConstants.COLOR_INT))));
41
42        // A token that defines how to color keywords.
43        keywordToken = new Token(new TextAttribute(colorManager.getColor(PreferenceConverter.getColor(
44                prefs, CspMEditorPreferenceConstants.COLOR_KEYWORD)), null, SWT.BOLD));
45
46        // A token that defines how to color predefined identifiers.
47        processesToken = new Token(new TextAttribute(colorManager.getColor(PreferenceConverter.getColor(
48                prefs, CspMEditorPreferenceConstants.COLOR_PROCESS)), null, SWT.ITALIC));
49
50        keywordRule = new WordRule(new CspMKeywordDetector());
51        Keywords[] values = Keywords.values();
52        for (int i = 0; i < values.length; i++) {
53            keywordRule.addWord(values[i].getValue(), keywordToken);
54        }
55
56        processRule = new WordRule(new CspMNameDetector());
57        for (int i = 0; i < PROCESSES.length; i++) {
58            processRule.addWord(PROCESSES[i], processesToken);
59        }
60
61        IRule[] rules = new IRule[] {
62                new WhitespaceRule(new CspWhitespaceDetector()),
63                keywordRule,
64                processRule,
65                new NumberRule(numberToken)
66        };
67
68        setDefaultReturnToken(defaultToken);
69        setRules(rules);
70    }
```

Figura 14. Definição de tokens e regras para o *CSP*M.

O ponto *org.Eclipse.ui.preferencePages* é estendido com a finalidade de definir uma página no menu de preferências do Eclipse. A classe *CspMPreferencePage* é responsável pela implementação da página que tem como objetivo permitir ao usuário personalizar as cores dos *tokens* de cada partição que, por sua vez, ficam armazenadas na classe *CspMEditorPreferenceConstants*.

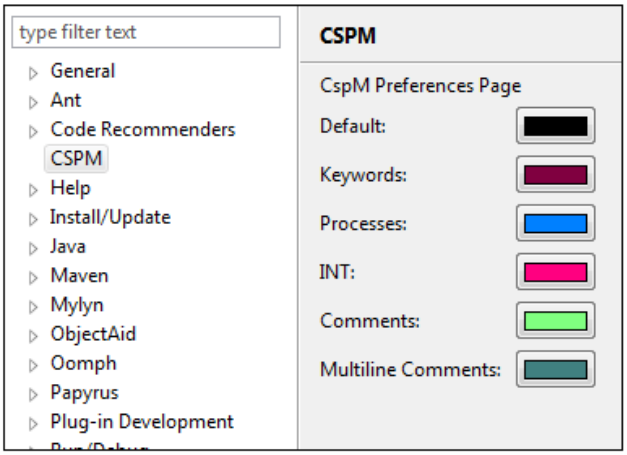

Figura 15. Página de preferência de cores do *Csp*MEditor.

A Figura 16 mostra um diagrama de classes que torna mais clara os relacionamentos existentes entre os componentes abordados e diretamente envolvidos com o recurso de *syntax highlight*.

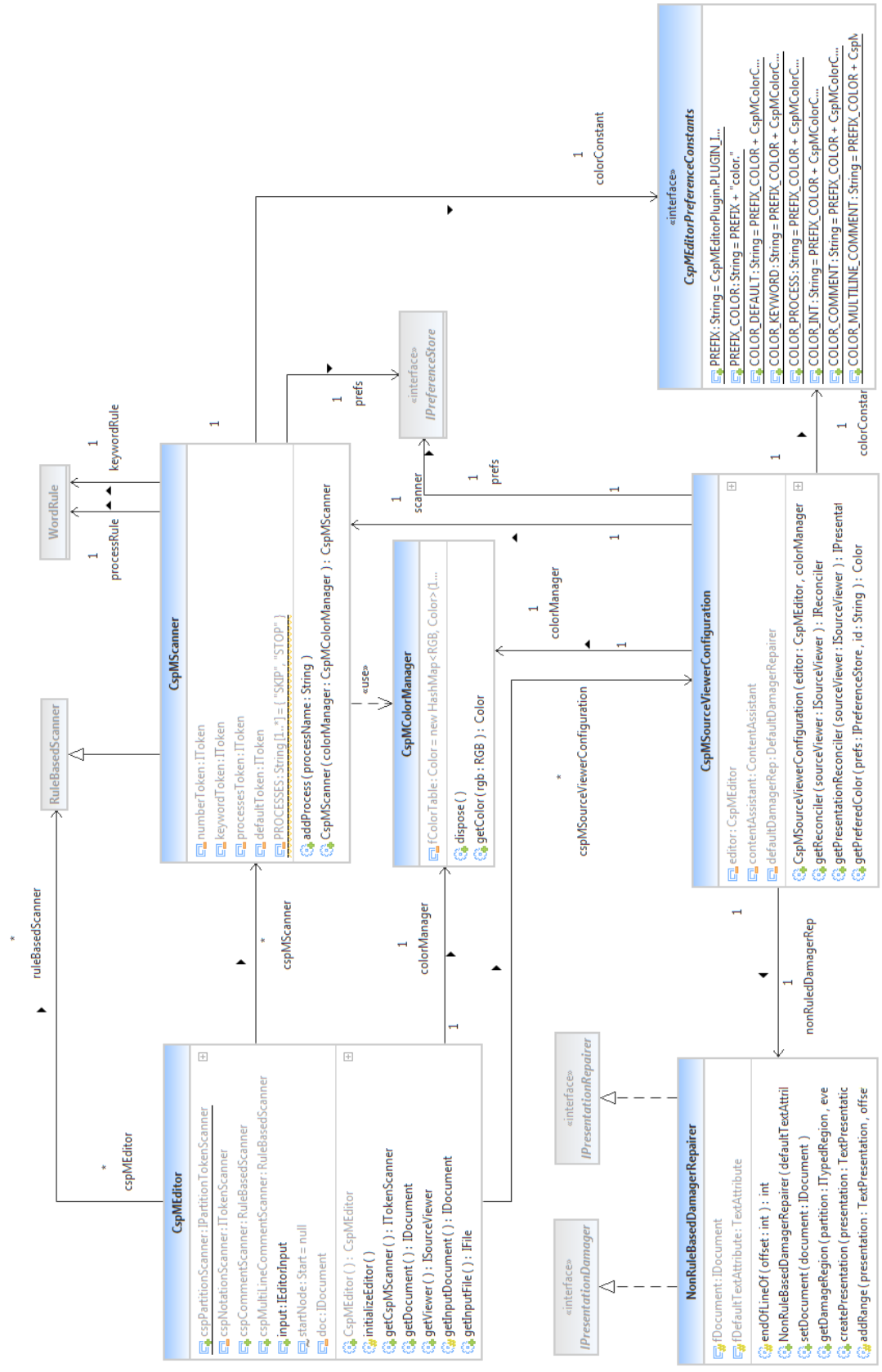

Figura 16. Diagrama de classes dos componentes do destaque de sintaxe.

3.4.8 Auto Complete

No *CspMEditor* 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. A configuração deste recurso será feita na classe *CspMSourceViewerConfiguration*, através do método *IContentAssistant getContentAssistant*(). É necessário instanciar um *IContentAssistant* e configurá-lo de acordo com as necessidades da IDE. Essa classe é definida no pacote *org.Eclipse.jface.text.contentassist* e implementa a interface *IContentAssistant*. O *ContentAssist* permite que o usuário forneça um conjunto de palavras que servirão de sugestões para o complemento do texto. Essas palavras serão mostradas dentro de uma janela *popup,* sendo possível escolher qualquer uma delas para completar a sentença que está sendo escrita no código.

O *IContentAssistProcessor* trabalha em conjunto com o *IContentAssistant,* definindo quando será viável o uso do auto complete. O método *computeCompletionProposals* é o responsável por fornecer o conjunto de sugestões de acordo com a posição do cursor. O primeiro passo desse método é gerar uma *AST* do código em questão através do *parsing* que*,* por sua vez, é uma instância da classe *CspMParser,* cuja configuração é feita em outra classe, a *CspMModel*. Após o *parsing* ser feito, o método *computeCompletionProposals* percorrerá toda a *AST* gerada, extraindo dela as referências que podem ser de dois tipos: *KEYWORD* ou *CspMRef*. A classe *KEYWORD* é do tipo Enum e armazena as palavras chave do *CSP*M previamente definidas como constantes em seu escopo. Já a *CspMRef* é a classe que representa as referências relacionadas aos demais tipos de dados (constante, função, etc) presentes no código. A classe responsável pelo acesso às referências do tipo *CspMRef* é a *CspMRefExtractor*, cuja finalidade é distinguir as referências pelos seus tipos. Após a *AST* ser gerada e todas as referências do código serem extraídas, elas são colocadas numa lista de sugestões (*completionProposals*) do tipo *IcompletionProposal* e, em seguida, ordenadas em ordem alfabética através do método *Collections.sort*. A figura 17 ilustra como a lista de sugestões é feita. Já a figura 18 mostra um diagrama de classes que representa a disposição dos componentes utilizados no recurso de autocomplemento.

```
36    public ICompletionProposal[] computeCompletionProposals(ITextViewer viewer, int offset) {
37        info = editor.parse();
38
39        if (info != null) {
40            try {
41                IDocument document = viewer.getDocument();
42                info.ast.apply(new CspMRefExtractor(document, info));
43                String prefix = lastWord(document, offset);
44                List<ICompletionProposal> completionProposals = new ArrayList<ICompletionProposal>();
45
46                for (CspMRef ref : info.tocRefs) {
47                    if (ref.getText().startsWith(prefix)) {
48                        completionProposals.add(new CompletionProposal(
49                                ref.getText(), (offset - prefix.length()), prefix.length(), ref.getText().length()));
50                    }
51                }
52
53                for (Keywords keyword : proposals) {
54                    if (keyword.getValue().startsWith(prefix)) {
55                        completionProposals.add(new CompletionProposal(
56                                keyword.getValue(), (offset - prefix.length()), prefix.length(), keyword.getValue().length()));
57                    }
58                }
59
60                Collections.sort(completionProposals, new KeywordsComparator());
61                return completionProposals.toArray(new ICompletionProposal[completionProposals.size()]);
62
63            } catch (Exception e) {
64                // ... log the exception ...
65                return NO_COMPLETIONS;
66            }
67        } else {
68            return NO_COMPLETIONS;
69        }
70    }
71
```

Figura 17. Representação do método *computeCompletionProposals*.

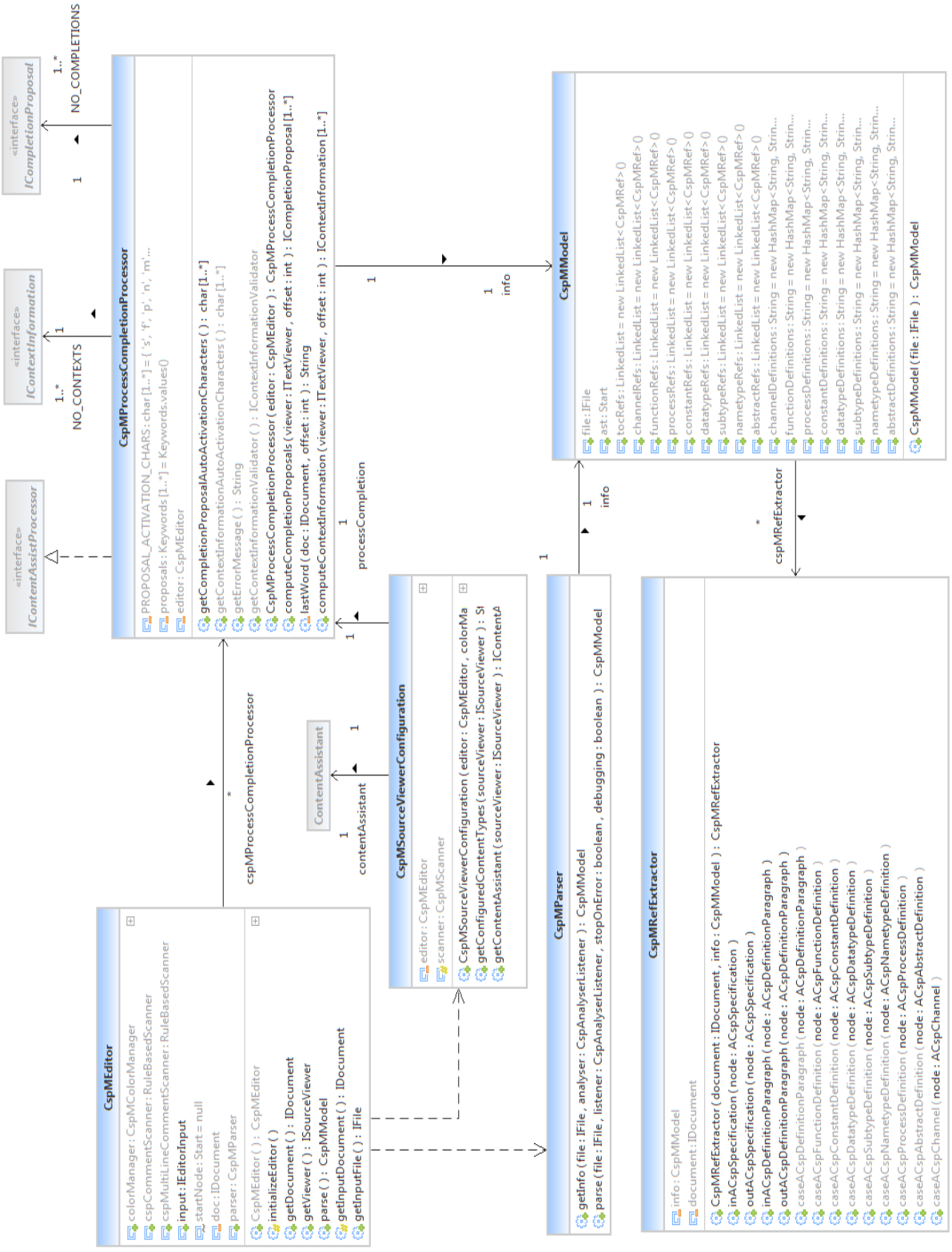

Figura 18. Diagrama de classes dos componentes da automação de escrita.

3.4.9 Browsing

O *Browsing* (auxílio à navegação) é um recurso de grande utilidade em um editor, tendo como finalidade permitir o usuário “navegar” pelo código. Basta dar um clique em um determinado termo (método ou variável, por exemplo) que será mostrado o local onde ele foi devidamente declarado. Esse mecanismo se mostra bastanté útil em códigos muito grandes, pois evita que o usuário perca tempo procurando os locais onde foram definidos ou declarados alguns termos, possibilitando acessá-los de maneira direta. Para usufruir deste recurso, é necessário criar um detector de links através da implementação da interface *IHyperlinkDetector*.

O *CspMHyperlinkDetector* implementa a interface *IHyperlinkDetector* e redefine apenas um método, o *detectHyperlinks*, cuja finalidade é definir regiões de texto clicáveis que funcionam, literalmente, como um link para sua respectiva declaração. Nessa classe também são definidos os pontos que servem de destino para a navegação, ou seja, o ponto para onde queremos navegar (definição da função, atribuição de um valor a uma constante, etc). Para que isso aconteça é necessário realizar o *parsing* no código, o que resultará numa lista de referências de navegação do tipo *CspMRef,* isto é, todos os pontos do código que são “navegáveis”. Também foi criado a classe *CspMHyperlink* que implementa a interface *IHyperlink*, onde iremos redefinir o método *open* que, como o próprio nome já diz, serve para “abrir” o local da referência de destino. Por último, temos o *CspMHyperlinkPresenter,* onde serão configuradas as características do *browsing,* tais como delimitação e sublinhamento da região que foi clicada e ativamento do *CURSOR_HAND* (quando o cursor se transforma numa mão). O diagrama de classes à seguir (figura 19), representa com maior precisão a estrutura do *browsing*.

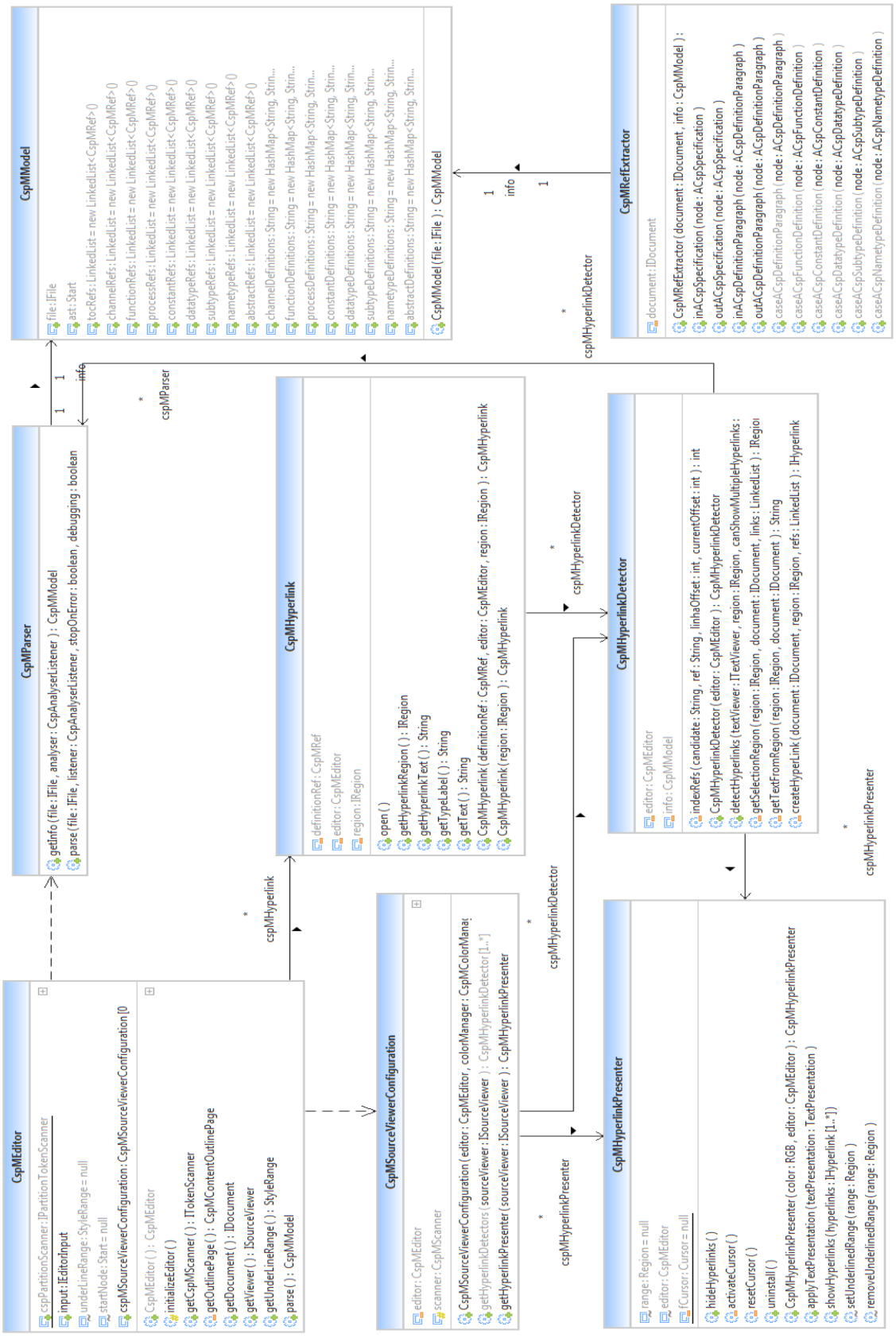

Figura 19. Diagrama de classes do recurso navegação (browsing).

3.4.10 Visualizador de árvore sintática (*content outline*)

O visualizador de árvore sintática fornece uma visão estrutural (*outline*) do conteúdo do editor, permitindo que o usuário navegue através dela. O *workbench* do Eclipse provê um *outline* padrão para este propósito, porém, podemos criar o nosso próprio, faremos isso através da classe *CspMContentOutlinePage*.

A classe *CspMContentOutlinePage* estende outra classe específica para esta função, a *ContentOutlinePage*. O papel dessa classe é definir um *Provider* que será responsável pelo modelo utilizado para definição do visualizador e também por propagar atualizações. Outra função pertinente a essa classe é a de navegação pelo contéudo do código através do *outline*. Para isso, ela contém um mecanismo responsável por detectar o nó em que o usuário acabou de clicar e, a partir da seleção deste nó, o editor consegue focar o posicionamento da estrutura selecionada.

O primeiro passo do *outline* é a criação da estrutura da "árvore" que será mostrada na tela, como podemos ver na figura 20:

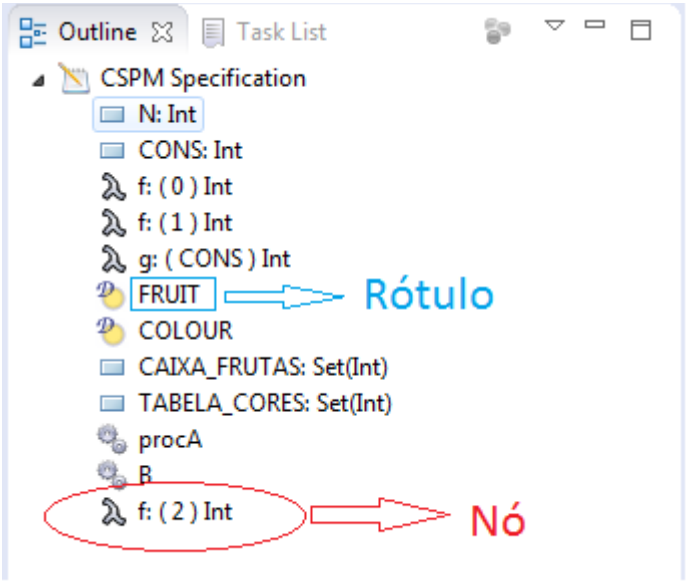

Figura 20. Conteúdo do *outline*.

A classe responsável por esta função é a *CspMOutlineContentProvider*. Aqui, através do *parser,* será gerado uma *AST* que servirá de base para o conteúdo do

outline. Logo após a *AST* ser gerada, é necessário definir os tipos dos nós de acordo com os padrões do *CSPM*, tarefa que será realizada pela *CspMAstTreeNodeGenerator*. Essa classe tem a função de percorrer toda a *AST* gerada, acessando todos os nós e atribuindo um determinado tipo (function, process, por exemplo) a cada um deles. Por último, é necessária a atribuição de rótulos aos nós da árvore, sendo essa função destinada à classe *CspMOutlineTreeLabelProvider*. A figura 21 mostra a estrutura do gerador de *outline*, através de um diagrama de classes.

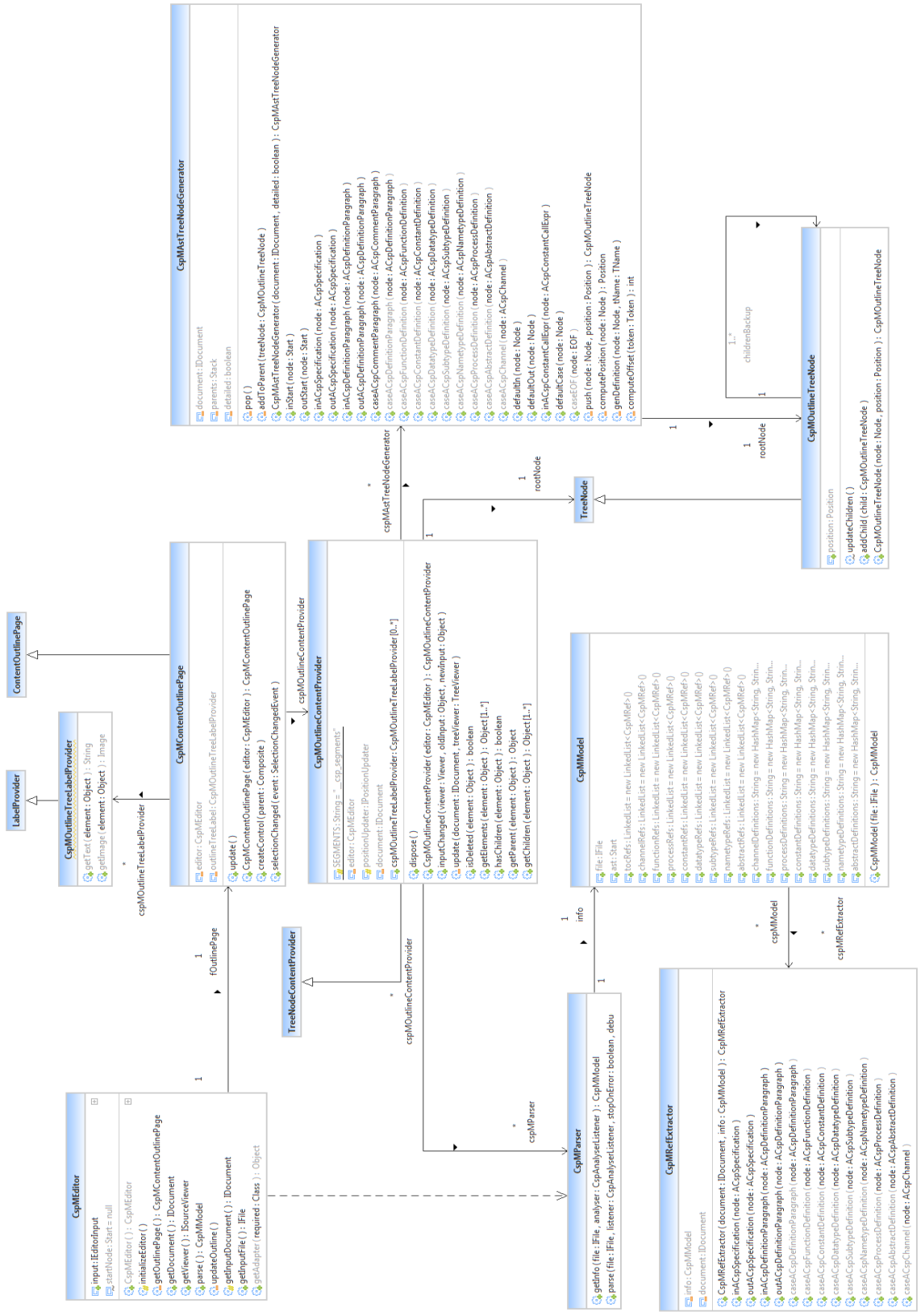

Figura 21. Diagrama de classes do recurso visualizador de árvore sintática (*outline*).

3.4.11 Error marker

O *error marker* (marcador de erro) é um recurso essencial numa IDE. Seu objetivo é detectar os erros no código de maneira dinâmica, geralmente destacando as expressões incorretas sublinhando-as com uma linha vermelha ou com um marcador que fica localizado na barra lateral do editor. Para que seja possível utilizar este recurso na IDE, é necessário estender o ponto *org.Eclipse.core.resources.markers* e declarar o tipo de marcador utilizado, que neste caso será o ERROR_MARKER.

Cada vez que o *parsing* é chamado no código (ao abrir um arquivo no editor ou utilizar o comando de salvar em um arquivo previamente aberto, por exemplo) uma instância da classe *CspMMarkingErrorHandler* é criada e ela, por sua vez, contém a configuração da marcação de erros no código. Essa classe possui o método *removeExistingMarkers,* cuja função é apagar todos os marcadores atuais no código, e também o *addProblemMarker*, responsável por criar, configurar e adicionar os marcadores de erro no arquivo em edição. O diagrama de classes presente na Figura 22 ilustra os componentes envolvidos na implementação dessa funcionalidade.

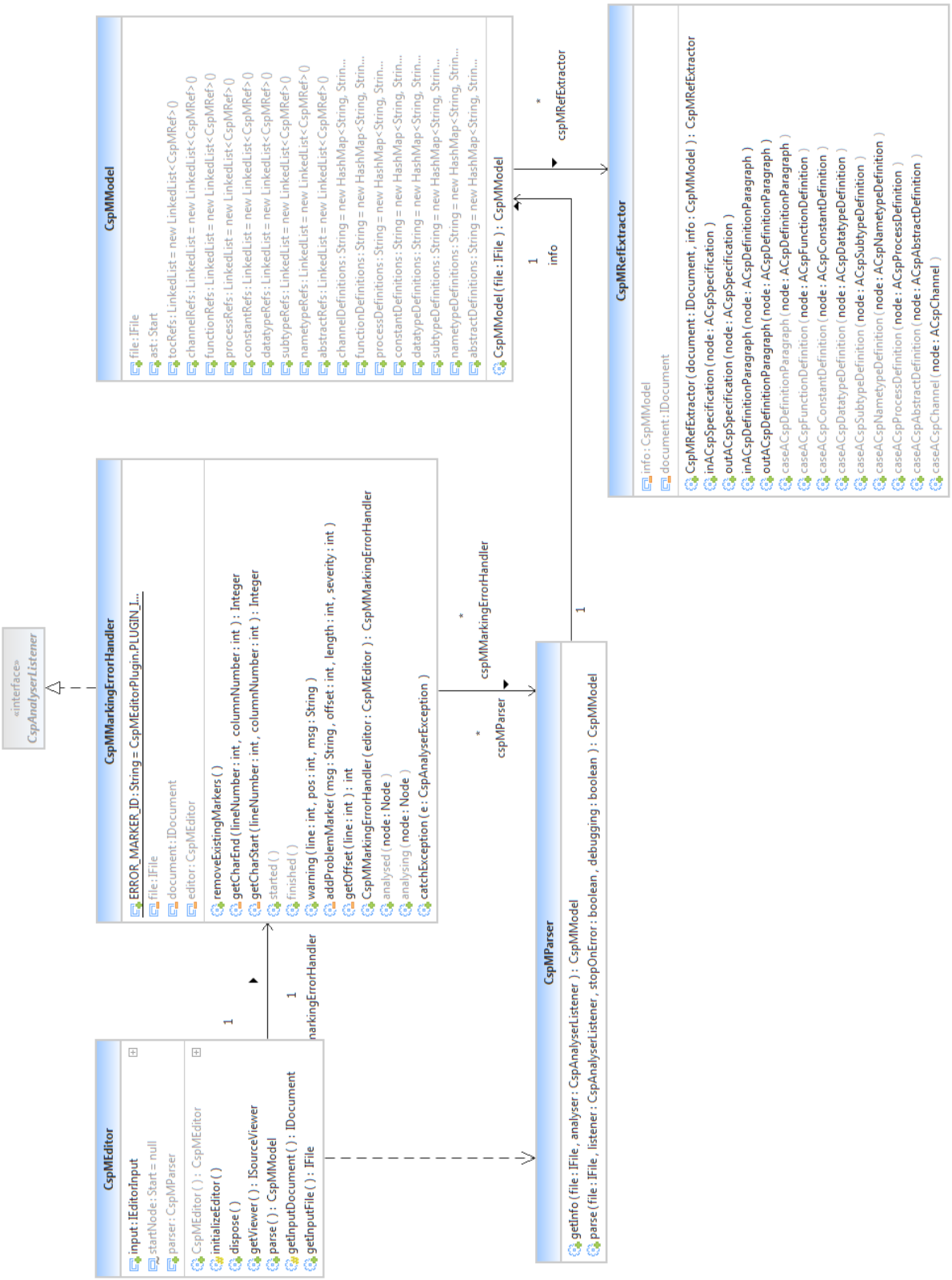

Figura 22. Diagrama de classes do recurso de marcação de erros (*error marker*).

4. Estudo de Caso

Neste capítulo será feita uma demonstração do uso da IDE proposta, onde serão mostradas todas as suas funcionalidades em atividade. Para isso, utilizaremos um código fonte como exemplo, chamado de Jantar dos Filósofos (*dining philosophers*). No decorrer deste capítulo será possível mostrar se a IDE atendeu aos requisitos propostos no início deste trabalho, expondo cada um deles detalhadamente no código que servirá de exemplo.

4.1 Jantar dos Filósofos

O jantar dos filósofos é o exemplo mais conhecido de problemas utilizados em algorítmos de concorrêcia, cujo objetivo é representar falhas de sincronização no sistema e as respectivas técnicas para resolvê-las (Ries, 2012). Este problema possui a seguinte situação:

- Cinco filósofos estão sentados ao redor de uma mesa circular para o jantar.
- Cada filósofo possui um prato para comer macarrão.
- Os filósofos dispõem de hashis e e cada um precisa de 2 hashis para comer.
- Entre cada par de pratos existe apenas um hashi: Hashis precisam ser compartilhados de forma sincronizada.
- Os filósofos comem e pensam, alternadamente. Eles não se atém a apenas uma das tarefas.
- Além disso, quando comem, pegam apenas um hashi por vez: Se conseguir pegar os dois come por alguns instantes e depois larga os hashis.

O problema é coordenar o uso dos hashi de maneira que nenhum filósofo fique com fome. A solução, escrita em CSPM, será utilizada para demonstrar o funcionamento da IDE. A figura 23 mostra a sua implementação num editor comum.

```
MAX = 2
RANGE = {0..MAX}

datatype EVENTS = picksup | putsdown | picksack | putsack | thinks | sits | eats | getsup

subtype LIFE = thinks | sits | eats | getsup
subtype PICK = picksup | putsdown | picksack | putsack

channel pfk: RANGE.RANGE.PICK
channel life: RANGE.LIFE
channel fk: RANGE.RANGE.PICK

PPH0 = life.0.thinks -> life.0.sits -> pfk.0.0.picksup -> pfk.0.0.picksack -> pfk.1.0.picksup -> pfk.1.0.picksack ->
       pfk.0.0.putsdown -> pfk.0.0.putsack -> pfk.1.0.putsdown -> pfk.1.0.putsack -> life.0.getsup -> PPH0
PPH1 = life.1.thinks -> life.1.sits -> pfk.1.1.picksup -> pfk.1.1.picksack -> pfk.2.1.picksup -> pfk.2.1.picksack ->
       pfk.1.1.putsdown -> pfk.1.1.putsack -> pfk.2.1.putsdown -> pfk.2.1.putsack -> life.1.getsup -> PPH1
PPH2 = life.2.thinks -> life.2.sits -> pfk.0.2.picksup -> pfk.0.2.picksack -> pfk.2.2.picksup -> pfk.2.2.picksack ->
       pfk.0.2.putsdown -> pfk.0.2.putsack -> pfk.2.2.putsdown -> pfk.2.2.putsack -> life.2.getsup -> PPH2

PFK0 = fk.0.2.picksup -> fk.0.2.picksack -> fk.0.2.putsdown -> fk.0.2.putsack -> PFK0
            [] fk.0.0.picksup -> fk.0.0.picksack -> fk.0.0.putsdown -> fk.0.0.putsack -> PFK0
PFK1 = fk.1.0.picksup -> fk.1.0.picksack -> fk.1.0.putsdown -> fk.1.0.putsack -> PFK1
            [] fk.1.1.picksup -> fk.1.1.picksack -> fk.1.1.putsdown -> fk.1.1.putsack -> PFK1
PFK2 = fk.2.1.picksup -> fk.2.1.picksack -> fk.2.1.putsdown -> fk.2.1.putsack -> PFK2
            [] fk.2.2.picksup -> fk.2.2.picksack -> fk.2.2.putsdown -> fk.2.2.putsack -> PFK2


PCOMP0 = PFK0 ||| PFK1
PFORKS = PCOMP0 ||| PFK2
PCOMP1 = PPH0 ||| PPH1
PPHILS = PCOMP1 ||| PPH2
```

Figura 23. Implementação do código em um editor comum.

4.2 Demonstração

O primeiro passo para a demonstração do exemplo consiste em criar um projeto na IDE (figura 24), em seguida, criaremos um arquivo com a extensão *.cspm* dentro deste projeto, este arquivo será chamado de *teste.cspm* (figura 25). O código será escrito neste arquivo, onde será possível mostrar as funcionalidades da IDE em atividade.

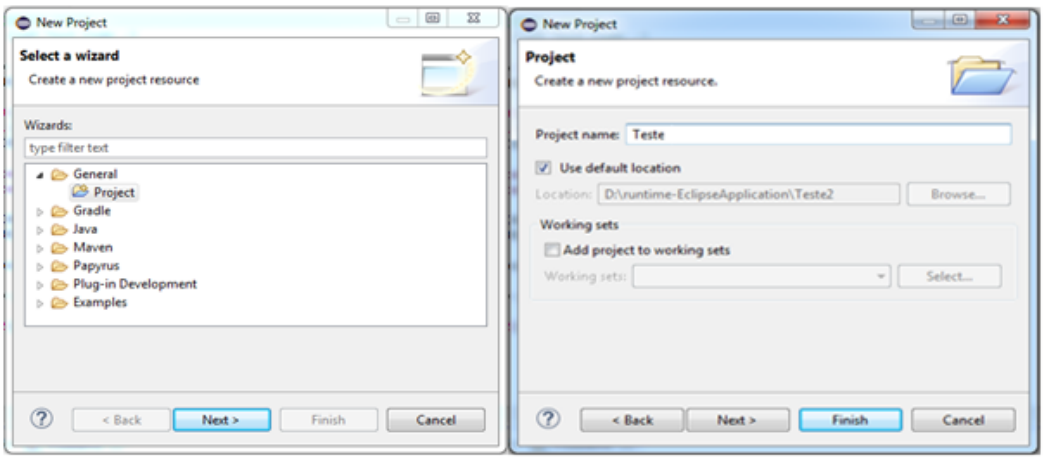

Figura 24. Criando o projeto de teste no eclipse.

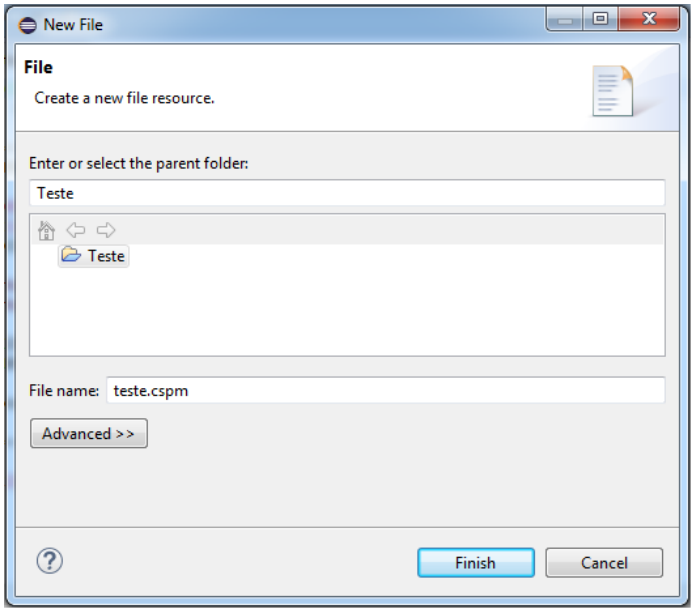

Figura 25. Criando o arquivo .cspm no eclipse.

4.2.1 Syntax Highlighting

Como visto na seção 2.1.1, o syntax highlighting serve para destacar as palavras chave do CSPM no código fonte, facilitando seu entendimento. Essas palavras são previamente definidas na IDE, bem como a maneira em que serão destacadas (cor da fonte, por exemplo). Na figura 26 é possível ver essa funcionalidade em atividade, destacando os termos *datatype*, *subtype* e *channel* que, por sua vez, são palavras chave do CSPM. Os números inteiros também foram colorados.

```
3 MAX = 2
4 RANGE = {0..MAX}
5
6 datatype EVENTS = picksup | putsdown | picksack | putsack | thinks | sits | eats | getsup
7
8
9 subtype LIFE = thinks | sits | eats | getsup
10 subtype PICK = picksup | putsdown | picksack | putsack
11
12
13 channel pfk: RANGE.RANGE.PICK
14 channel life: RANGE.LIFE
15 channel fk: RANGE.RANGE.PICK
```

Figura 26. Destaque de sintaxe do arquivo de teste.

4.2.2 Auto Complete

De acordo com a seção 2.1.2, foi visto que durante o processo de escrita do código poderá ser mostrado ao usuário uma lista de sugestões de termos válidos para o complemento de determinados trechos. Esta lista será exibida no local onde o cursor se encontra, pressionando-se *ctrl*+espaço. A figura 27 mostra a lista de sugestões sendo ativada mediante ao comando de solicitação. Na lista, são apresentadas as palavras chave do CSPM e os termos já utilizados no código.

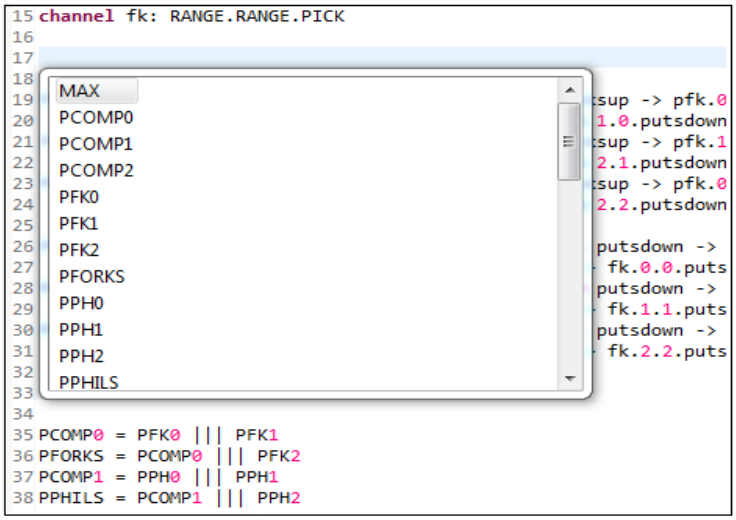

Figura 27. Auto complemento no arquivo de teste.

4.2.3 Browsing

Na seção 2.1.3, foi constatado que o Browsing proporciona um fácil acesso a declarações de processos e variáveis, referências e outras definições dentro do código. Para ativar essa função, deve-se pressionar o *ctrl* e clicar no termo desejado. A figura 28 representa essa funcionalidade em execução, onde clicamos no termo PPH2 e sua declaração é referenciada de imediato.

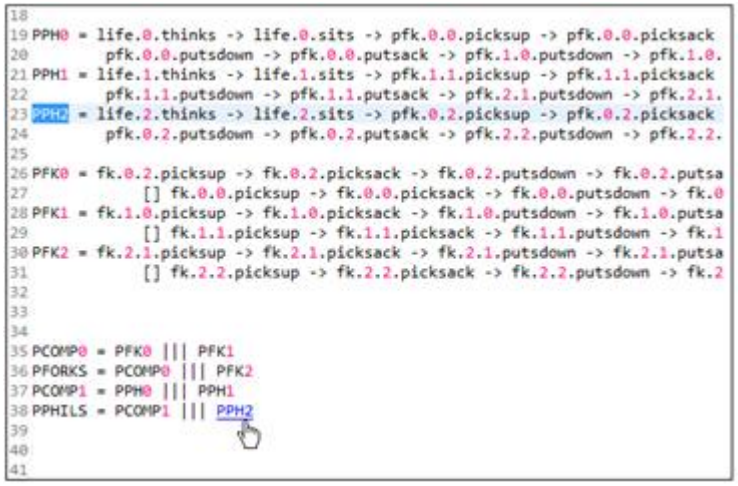

Figura 28. Navegação com hyperlink no arquivo de teste.

4.2.4 Content Outline

Já na seção 2.1.4 foi mostrado que o *content outline* cria uma árvore sintática com a finalidade de fornecer uma visão geral do código, organizado de maneira hierárquica. A figura 29 mostra o seu funcionamento, onde pode-se constatar a árvore e seu conteúdo devidamente organizado.

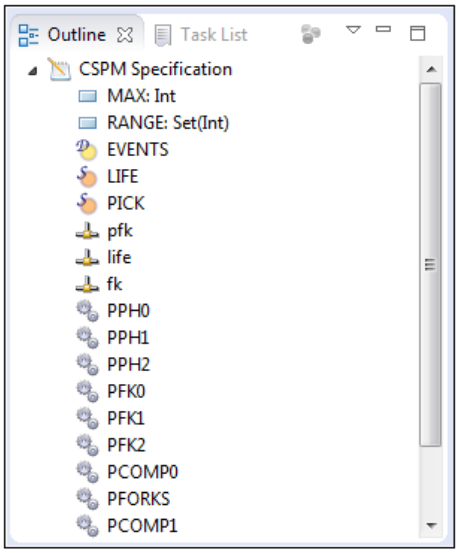

Figura 29. Visualização de estrutura da especificação de teste.

4.2.5 Error Marker

Por fim, na seção 2.1.5, foi visto que o *error marking* é fundamental numa IDE, uma vez que ele será responsável por detectar e mostrar possíveis erros de

implementação do código fonte. Na figura 30 podemos ver como um erro é ilustrado na IDE.

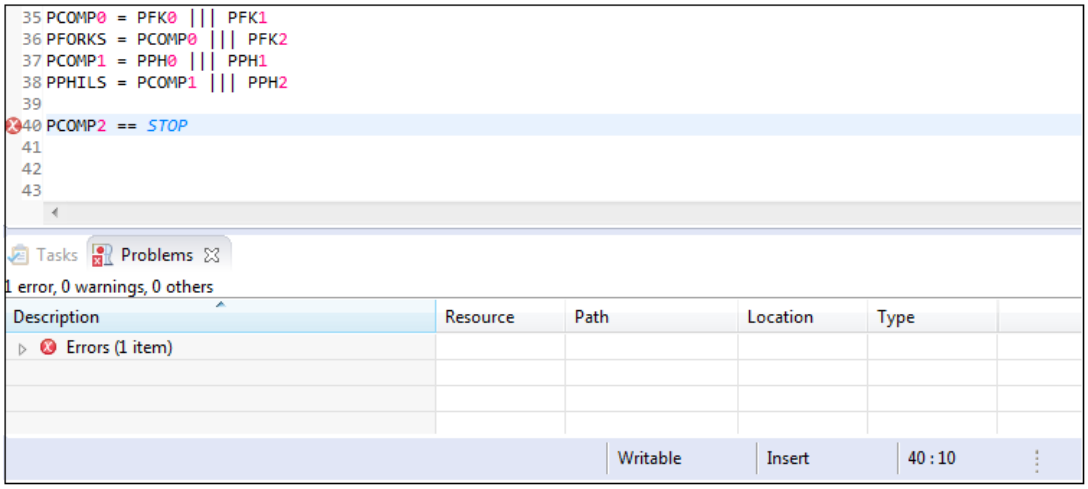

Figura 30. Identificação de erros na especificação de teste.

5. Conclusão

Atualmente a quantidade de ferramentas para se trabalhar com o CSPM é bastante limitada, o que dificulta a sua utilização em meios computacionais. Tendo em vista esse fato, este trabalho teve como objetivo criar um ambiente básico e amigável de desenvolvimento para a linguagem, com o objetivo de tornar seu uso mais compreensível e viável. Esta IDE conseguiu englobar as principais funções de um ambiente padrão (edição sensível a linguagem, autocomplemento, marcação de erros, outline, coloração de sintaxe e navegação), o tornando funcional para criação de códigos básicos em CSPM.

Sendo atualmente o ambiente de desenvolvimento integrado com uma das maiores popularidade do mercado, o Eclipse se mostrou como sendo uma excelente escolha para servir como ferramenta principal da criação do *CSPDT*, dispondo de todos os utensílios necessários para o seu desenvolvimento.

O Trabalho foi concluído mostrando a importância da criação de uma IDE simples e funcional para o *CSP*M, englobando apenas seus requisitos básicos e também dando sua contribuição para a comunidade de desenvolvedores dessa linguagem. Para trabalhos futuros, espera-se criar todos os outros requisitos que não foram contemplados neste artigo, tornando o *CSPDT* uma ferramenta aprimorada e mais completa.

Bibliografia

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

ANISZCZYK, C.; GALLARDO, D. **Introdução à Plataforma Eclipse.** Disponível em: <http://www.ibm.com/developerworks/br/library/os-eclipse-platform/>. Acesso em 15 abril 2016.

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

BEHRENS, H. **Xtext.** Disponível em: <http://www.eclipse.org/Xtext/>. Acesso em: 19 janeiro 2016.

BLEWITT, A. **10 Anos de Eclipse.** Disponível em: <https://www.infoq.com/br/news/2011/11/eclipse-10>. Acesso em 10 abril 2016.

Dos Santos, Alexandre K. **Os IDE's (Ambientes de Desenvolvimento Integrado) como ferramentas de trabalho em informática.** Disponível em: <http://www-usr.inf.ufsm.br/~alexks/elc1020/artigo-elc1020-alexks.pdf>. Acesso em 18 fevereiro 2016.

Eclipsepedia. **FAQ What are extensions and extension points?.** Disponível em: <http://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points/>. Acesso em: 15 abril 2016.

Eclipse Documentation**. org.eclipse.ui.editors.** Disponível em: <http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fworkbench_basicext_editors.htm>. Acesso em 20 Abril 2016.

Eclipse Foundation, Inc.. **Eclipse:** The Eclipse Foundation open source community website. Disponível em: <http://www.eclipse.org/>. Acesso em: 18 janeiro 2016.

FARIA, F.B.; LIMA, P. S. N.; DIAS, L. G.; SILVA, A. A.; DA COSTA, M. P.; BITTAR, T. J. **Evolução e Principais Características da IDE Eclipse.** Disponível em:

<http://www.enacomp.com.br/2010/cd/artigos/completos/enacomp2010_23.pdf>. Acesso em 10 abril 2016.

HOARE, C. A. R. **Communicating Sequential Processes.** Disponível em: <http://www.usingcsp.com/cspbook.pdf>. Acesso em: 28 abril 2015.

MADEIRA, M. **OSGI – Modularizando sua aplicação.** Disponível em: <https://celodemelo.wordpress.com/2009/11/12/osgi-modularizando-sua-aplicacao/>. Acesso em 19 março 2016.

MEDEIROS, H. **Introdução ao Padrão MVC.** Disponível em: <http://www.devmedia.com.br/introducao-ao-padrao-mvc/29308>. Acesso em: 12 março 2016.

Network Management Center of NTUA. **Text editors and JFace text.** Disponível em: <http://ftp.ntua.gr/eclipse/eclipse/downloads/documentation/2.0/html/plugins/org.eclipse.platform.doc.isv/guide/editors_jface.htm>. Acesso em 20 Abril 2016.

RIES, U. **Jantar dos Filósofos - Programação Paralela**. Disponível em: <https://www.vivaolinux.com.br/script/Jantar-dos-Filosofos-Programacao-Paralela/>. Acesso em: 14 abril 2016.

ROSCOE, Bill. **Introducing FDR3.0.** Disponível em: <http://www.cs.ox.ac.uk/projects/fdr/manual/examples/index.html>. Acesso em 28 abril 2016.

SHI, L., LIU, Y.; SUN, J.; DONG, J. S.; CARVALHO, Gustavo. **An Analytical and Experimental Comparison of CSP Extensions and Tools**. The 14th International Conference on Formal Engineering Methods (ICFEM 2012), pages 381-397, Kyoto, Japan, Novembro 12 - 16, 2012.

BEZERRA DE JESUS JÚNIOR, J.; **FormalDev: Tool support for generating UML-RT diagrams from CSP specifications.** Disponível em <http://www.cin.ufpe.br/~tg/2006-1/>. Acesso em 20 Junho 2016

Failures-Divergence Refinement; Disponível em <http://www.fsel.com/documentation/fdr2/fdr2manual.pdf>. Acesso em 20 Junho 2016.