Metodologia para Adição de Plug-in ao Ambiente Eclipse

Felipe Silva Loredo

15 de dezembro de 2008

Projeto Financiado pela FAPEMIG
Processo Número: CEX 1484/06

Sumário

1 Pré-requisitos **3**

2 IDE - Integrated Development Environment **3**

3 O Ambiente Eclipse **4**

4 Recursos Abordados neste Texto **5**
4.1 Compilador 5
4.2 Editor de Código 6
4.3 Execução 6
4.4 Gerência de Projetos 6
4.5 Estruturas Auxiliares 6

5 Hello World - O Básico dos Plug-ins **6**
5.1 O Básico dos Plug-ins 7
5.2 Criando um Plug-in Hello World 7
5.3 Explorando o Plug-in Hello World 9
5.3.1 Pontos de extensão 9
5.3.2 Manifestos 9
5.3.3 Visitando os Manifestos do Hello World 10

6 Criando um Editor de Código **11**
6.1 Um primeiro editor: a criação 11
6.2 Um segundo editor: detectando e colorindo comentários 13
6.3 Detectando e colorindo palavras chaves e *strings* de caracteres 18

7 Criando projetos **22**
7.1 Extensões e afins 22
7.2 Criando a classe responsável pelo assistente 24

8 Compilando Código **35**
8.1 Exibindo mensagens 35
8.2 Compilando código 46

9 Executando um Programa Compilado **55**

1 Pré-requisitos

Para acompanhar este texto espera-se que o leitor tenha um conhecimento básico a respeito de compiladores, da linguagem Java e que saiba operar o ambiente Eclipse para escrita de código em geral.

2 IDE - Integrated Development Environment

Antigamente, e ainda hoje tem quem faça dessa maneira, a codificação costumava ser realizada em editores de textos simples e os arquivos produzidos dessa forma eram salvos e submetidos a compilação. Caso um erro fosse apontado, digamos na linha 36, era necessário voltar ao editor de texto e consertar a tal linha 36. Não é pouco comum que erronamente o usuário verificasse uma linha errada, no nosso caso, a 32, por exemplo. Depois de algum tempo, pensando e olhando, o programador chegava a dramática conclusão de que a linha 32 estava correta. Submetendo novamente seu código ao compilador, o usuário descobria que era na verdade 36 e não 32. Pior, descobria que na linha 36 tinha um erro de digitação apenas, simples, fácil de ser corrigido.

O parágrafo anterior expõe, não apenas como o processo de compilação ocorria, mas também um grave problema presente naquela abordagem. Além de dispendioso no quesito tempo, a depuração dos programas era difícil. Caso fosse necessário saber porque o programa executou a linha 36, quando ele não deveria se quer ter entrado no comando *if* que a contém, uma maneira usual era inundar o código com vários comandos para realização de saída de dados, como *write*, *printf*, ou seja lá qual a for linguagem de programação. Funcionava, e apesar de antiquado, funciona até hoje, mas conhecido o benefício dos depuradores, certamente essa não é a melhor forma de remover erros de um programa.

Para abstrair-se desse nível de trabalho de programação, alguns editores passaram a fornecer alguma interoperabilidade com compiladores e depuradores, além de fornecer *syntax highlighting*, *code completion* e *syntax indentation*. Note que ainda eram ferramentas separadas que precisavam de alguma configuração para trabalhar como uma unidade. Nesses casos ao solicitar a compilação, o editor de textos invocava o compilador com o código editado e podia também chamar um depurador.

A evolução não parou por aí, dessas idéias surgiram as IDEs, do inglês *Integrated Development Environment* – Ambiente Integrado para Desenvolvimento. As IDEs basicamente reúnem um editor, um compilador e o depurador, abstraindo a configuração necessária para que essas partes trabalhem

como uma unidade. A figura 1 mostra uma das IDEs mais utilizadas no mundo [2], a do Borland Delphi. Como indicado, nela há recursos para compilação, depuração, execução e *syntax highlighting*, se prestando muito bem como exemplo de nosso assunto.

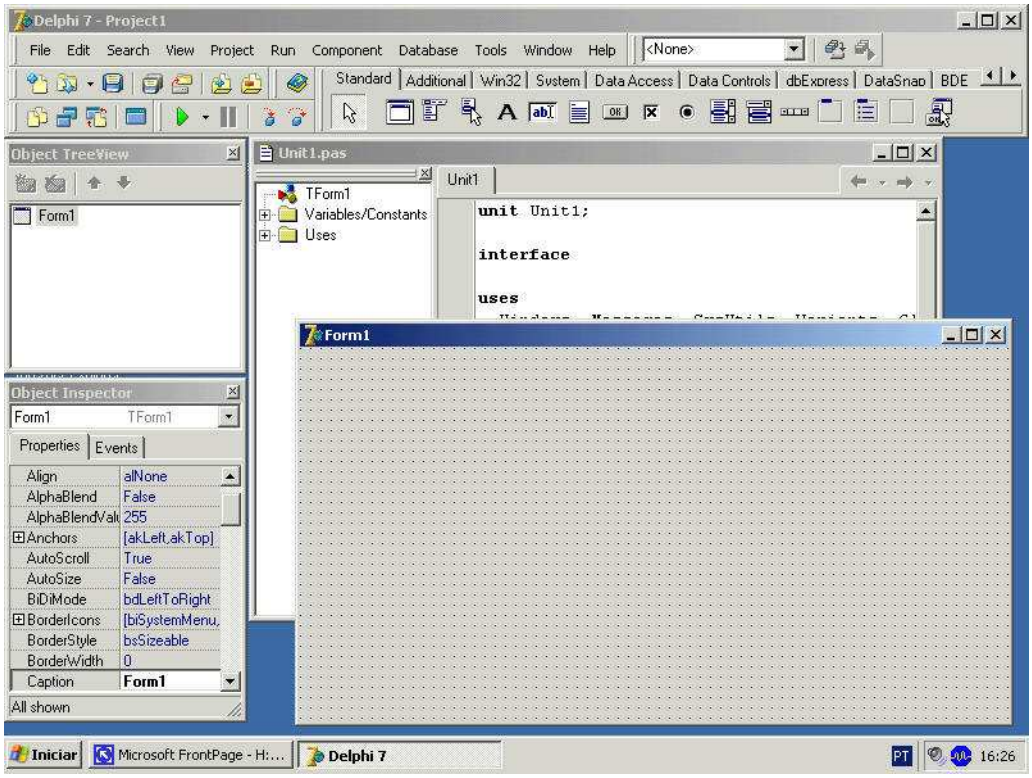

Figura 1: IDE do Borland Delphi 7

3 O Ambiente Eclipse

Nosso foco neste texto é o estudo de como produzir um ambiente integrado para trabalhar com linguagens de programação. Para facilitar o acompanhamento utilizaremos a linguagem Notus [3]. Para não perdermos o foco, em poucas palavras, segundo seus autores, trata-se de uma linguagem de uso específico para definição de linguagens de programação de forma compreensível e modular.

Construir uma IDE do zero é uma tarefa mais árdua do que a planejada para este trabalho, dessa forma utilizaremos uma plataforma livre que possa ser personalizada e possua ampla documentação: o Ambiente Eclipse. Segundo o site oficial do projeto [4], "Eclipse significa muitas coisas diferentes

para pessoas diferentes. Para alguns Eclipse é um ambiente de desenvolvimento Java grátis. Para outros, Eclipse é um ambiente flexível para experimentação com novas linguagens de programação ou com extensões de linguagens existentes...". Naturalmente, estamos interessados na última parte desse excerto. A escolha do projeto Eclipse não foi aleatória, como pode parecer, ele foi selecionado porque trata-se de um ambiente *aberto, livre* projetado para permitir extensões sem muita dificuldade.

Para adicionarmos o suporte a uma linguagem no ambiente Eclipse, é necessária uma instalação do Eclipse – no nosso caso utilizamos a versão 3.2 –, e o desenvolvimento de um *plug-in* que efetivamente adicionará o suporte a linguagem desejada ao ambiente Eclipse.

Falaremos disso mais tarde, mas por enquanto, basta saber que um plug-in é basicamente um código compilado que pode ser utilizado pelo Eclipse para estender suas funcionalidades. Como é de se esperar, a IDE possui um conjunto de recursos pré-disponibilizados que também são construídos através da idéia de plug-ins. Há ainda um conjunto de funcionalidades básicas que são fornecidas por uma espécie de *kernel*, sendo todo resto fornecido sob a forma de plug-ins.

4 Recursos Abordados neste Texto

A seguir listamos os recursos que serão adicionados ao ambiente Eclipse para que ele suporte a linguagem objetivo, no nosso caso, como já dito, a linguagem Notus.

4.1 Compilador

Não abordaremos aqui a produção de um compilador através de um plug-in. Suporemos que o compilador já existe e atua de forma independente do Eclipse. Dessa forma, quando o usuário solicitar uma compilação, o ambiente invocará o compilador, servindo de interface entre a ferramenta de compilação e o desenvolvedor. Na verdade, nosso exemplo será um pouco mais abrangente que isso. O compilador Notus produz arquivos de saída para três outros programas que devem ser chamados, o Alex [5], o Happy [6] e o GHC [7]. Dessa forma, chamaremos, o compilador Notus para o código do usuário, a seguir o Alex para a saída do compilador Notus, depois o Happy para a saída do Alex e do compilador Notus, e finalmente, o GHC, para a saída dos anteriores.

4.2 Editor de Código

O editor de código adicionado ao ambiente Eclipse basicamente fará a *syntax highlighting* do código do usuário. Serão distinguidos palavras chave, *strings* de caracteres e comentários.

4.3 Execução

Após compilado, o usuário pode querer executar seu código para conferência, ou por outro motivo qualquer. A parte de execução tratará justamente disso, quando solicitado ao ambiente, o programa compilado será executado e as saídas e entradas de dados dele serão exibidas e recebidas por intermédio do Eclipse.

4.4 Gerência de Projetos

Basicamente para que possamos compilar e executar os programas do usuário é necessário que tenhamos um tipo de projeto para a linguagem que será suportada pelo ambiente. Trata-se de uma exigência coerente do Eclipse. Para que a IDE saíba como proceder com um programa de uma determinada linguagem, uma organização mínima de configurações é necessária, como por exemplo, onde os arquivos fonte encontram-se, quais são os arquivos a serem compilados e qual o nome do executável gerado – os projetos fornecem essa organização. No nosso exemplo, criamos um "Notus Project", um tipo projeto que permitirá a compilação, a execução e a gerência dos arquivos que compõem um projeto da linguagem Notus.

4.5 Estruturas Auxiliares

Para que possamos desenvolver os recursos citados acima será necessária a produção de um outro conjunto de extensões para o ambiente Eclipse. Essas extensões servirão de suporte para que as demais funcionalidades tornem-se possíveis. Por exemplo, para produção do plug-in de um compilador é desejável que as mensagens emitidas durante a compilação sejam repassadas ao usuário. Assim, é necessário produzir uma janela que permita a exibição dessas mensagens.

5 Hello World - O Básico dos Plug-ins

Nesta seção estudamos através de um primeiro plug-in introdutório alguns conceitos básicos relacionados aos plug-ins. Mais especificamente, discutire-

mos os conceitos necessários ao progresso do nosso trabalho e criaremos um plug-in do tipo "Hello World", para vermos o funcionamento dos conceitos discutidos.

5.1 O Básico dos Plug-ins

Cada plug-in é composto de uma biblioteca escrita em linguagem Java e um arquivo chamado manifesto (do original *manifest*) que o descreve. Durante a sua abertura, o Eclipse carrega todos os manifestos disponíveis e compila um registro – uma espécie de tabela que contém a lista dos plug-ins disponíveis. Por questões de eficiência os plug-ins em si não são carregados, o que só ocorre quando eles realmente são necessários. Isso nos deixa uma nova questão: como saber quando um plug-in é necessário? A resposta encontra-se noutro conceito, o de pontos de extensão (*extension points*).

Basicamente um ponto de extensão informa a plataforma Eclipse quais aspectos são personalizados por um plug-in. Quando um evento que requer um ponto de extensão ocorre, o plug-in registrado para esse ponto é, então, respondendo à questão anterior, carregado.

Finalmente, para que o Eclipse possa gerenciar plug-ins e pontos de extensão é necessário que ambos possuam identificadores. Nesse contexto, identificadores são strings de caracteres únicas construídas com as mesmas regras para nomeação de pacotes da linguagem Java. Isso pode causar alguma confusão, portanto para evitar algum problema, vamos a um exemplo: para criar um editor de textos é necessário estender um ponto de extensão chamado *org.eclipse.ui.views*. Por outro lado, como o Eclipse é escrito em Java, esse ponto de extensão é codificado através de uma *interface* chamada *org.eclipse.ui.IViewPart*. É importante, senão essencial, perceber, compreender e manter em mente que são nomes parecidos para entidades distintas.

5.2 Criando um Plug-in Hello World

Para criarmos esse plug-in introdutório não escreveremos uma linha de código sequer. Nosso objetivo nesse ponto não é compreender como codificar para plug-ins e muito menos como programar em linguagem Java. Produziremos o Hello World como experimento para objeto de estudo e compreensão do modo como a arquitetura de plug-ins do Eclipse funciona. Sem mais delongas, utilizaremos, portanto, o assistente de criação de plug-ins do Eclipse.

1. Estando com o Eclipse carregado, selecione o menu **File**, opção **New** e a seguir **Project...**.

2. Na tela “New Project” etapa “Select a wizard”, selecione a opção **Plug-in Project** que se encontra na pasta **Plug-in Development** e clique no botão **Next**.

3. Em **Project name** (nome do projeto) digite Hello e clique no botão **Next**.

4. Na próxima etapa há duas opções que nos interessam:

   (a) **Plug-in ID**: identificador único do plug-in que será utilizado pelo Eclipse para identificá-lo.

   (b) **Plug-in Name**: nome para identificação do plug-in do ponto de vista do usuário.

   Observe que essas opções já estão preenchidas, e por isso não precisamos alterá-las: clique no botão **Next**.

5. Nessa etapa uma série de modelos de plug-ins são exibidos, dentre eles o que queremos: selecione a opção “Hello, World” da lista “Available Templates”, clique em **Next** e a seguir em Finish.

Como você já deve ter percebido, o plug-in esta pronto. Agora queremos, portanto, vê-lo em ação.

1. Selecione o menu **Run**, depois a opção **Run...**.

2. Clique duas vezes sobre a opção **Eclipse Application**.

3. Altere a caixa “Name” de alguma coisa como “New_configuration” para “HelloPlugin”.

4. Clique em **Run**.

Observe que ao solicitar a execução do plug-in um nova instância do Eclipse foi aberta. Trata-se de uma cópia do Eclipse em que seu plug-in encontra-se em ação. Após carregado, note que há um novo menu no Eclipse, “Sample Menu”. Se ainda não houve curiosidade suficiente, clique nesse menu e a seguir na única opção disponível. Feito, “Hello, Eclipse world”.

5.3 Explorando o Plug-in Hello World

Terminado o processo de criação do Hello World, chegou a hora de o explorarmos, único fato, além da sorte, é claro, que justifica sua existência.

Ao abrir o projeto do plug-in no "Package Explorer" há três componentes que são de nosso interesse: a pasta "src", onde, é fácil de deduzir, encontra-se o código fonte do plug-in, o arquivo "plugin.xml" e a pasta chamada "META-INF" que tratam, digamos de "meta informações" muito importantes para nossos propósitos.

5.3.1 Pontos de extensão

A plataforma Eclipse possui um motor (*engine*) de execução que inicia a base da plataforma e dinamicamente descobre e executa plug-ins. Esse motor codifica o modelo básico de plug-ins e a infra-estrutura básica usada pela plataforma, além de manter uma lista de todos os plug-ins instalados com as funcionalidades que eles provêem, ativando-os somente quando são necessários. As contribuições de um plug-in são definidas por seus pontos de extensão – pontos de função bem definidos que podem ser estendidos por outros plug-ins.

O uso do modelo de pontos de extensão provê uma maneira estruturada para que os plug-ins possam descrever as formas como eles podem ser estendidos, e para que os plug-ins clientes possam descrever os seus pontos de extensão alvo. Trata-se de algo bastante semelhante à definição de outras APIs. A única diferença é que o ponto de extensão é declarado usando XML ao invés de uma assinatura de código.

Uma grande vantagem deste modelo de execução é que o usuário final não sofre com penalizações de memória ou desempenho devidas a plug-ins que encontram-se instalados, mas não usados. A natureza declarativa do modelo de extensão permite que o motor de execução determine quais extensões e pontos de extensão são alvo de um plug-in sem nunca executá-lo. Isso significa, que muitos plug-ins podem ser instalados, mas não serem ativados até que a função que eles provêem seja solicitada de acordo com a atividade do usuário. Trata-se de um recurso importante quando o objetivo, como é o caso do Projeto Eclipse, o provimento de uma plataforma robusta e escalável.

5.3.2 Manifestos

A extensões e pontos de extensão de um plug-in são declarados nos arquivos de manifesto dos plug-ins. Um plug-in possui dois manifestos, um OSGi ("MANIFEST.MF") e outro para o manifesto do plug-in ("plugin.xml"). O primeiro (segundo o consórcio OSGi [8]), deve prover o conjunto de primitivas

que permitam a construção de aplicações a partir de pequenos componentes, reusáveis e colaborativos – no nosso caso os plug-ins. Já o segundo define os pontos que serão estendidos pelo plug-in através de quais extensões.

5.3.3 Visitando os Manifestos do Hello World

A seguir daremos uma breve olhada no conteúdo dos manifestos do plug-in Hello World.

1. Abra o arquivo "MANIFEST.MF" que encontra-se dentro do projeto "Hello" na pasta (sugestivamente) chamada "META-INF".

2. Ao fazê-lo o Eclipse mostrará uma janela com uma série de abas a respeito do plug-in. Trata-se de uma interface para manipular, além de outras informações, as extensões e os pontos de extensão do plug-in.

3. Ao selecionar a aba **plugin.xml** os pontos estendidos pelo plug-in serão mostrados.

A tag XML <*extension point="org.eclipse.ui.actionSets"*> mostra o único ponto de extensão do plug-in. Trata-se de um ponto de extensão utilizado quando o objetivo do plug-in é contribuir com menus, itens de menus e botões de barras de ferramentas das áreas comuns do ambiente de trabalho da IDE [9]. Observe a existência de uma opção dentro da tag <*action*> chamada *class*. Trata-se da definição de qual classe Java do plug-in efetivamente codifica o ponto de extensão. Já opção *id* da mesma tag, identifica unicamente a extensão realizada pelo plug-in. As demais tags tratam das opções de configuração do ponto de extensão, fornecendo, por exemplo, o texto que será exibido na opção de um menu.

Escrever arquivos XML a mão pode ser algo demorado, difícil e tedioso (talvez principalmente frustrante quando depois de todo o esforço o plug-in, sujeito a falhas humanas, não funcionar). Provavelmente, pensando nisso o Eclipse possui uma interface para manipular a extensões dos plug-ins. Para acessá-la efetue os passos a seguir (supondo que os passos para visualização do arquivo "MANIFEST.MF" foram realizados):

1. Selecione a aba **Extensions**.

2. Selecione o item "org.eclipse.ui.actionSets" que encontra-se na seção "All Extensions".

Note a existência de um botão **Add** que permite a adição de novas extensões e na seção "Extension details" opções que permitem configurar as

opções disponíveis para a extensão selecionada. Algo interessante que será deixado a cargo do leitor é descobrir para que serve o item "Sample Action" contido no item "actionSet" da extensão do plug-in Hello.

Para realizar algumas extensões outros plug-ins podem ser necessários, daí deve-se dizer ao Eclipse quais plug-ins pré-existentes serão utilizados pelo plug-in em desenvolvimento. Disso trata a guia **Dependencies**, nela deve-se informar ao Eclipse a lista de dependências a outros plug-ins. Por exemplo, para que um plug-in forneça funcionalidades para compilação há um ponto de extensão chamado "org.eclipse.core.resources.builders", mas se você tentar adicioná-lo a lista de extensões da guia **Extensions** ele não aparecerá. Para fazê-lo é necessário dizer ao Eclipse que o seu plug-in é dependente do plug-in que fornece essa funcionalidade. Dessa forma, antes de adicionar o ponto de extensão em **Extensions** é necessário ir a guia **Dependencies** e adicionar a dependência de seu plug-in a "org.eclipse.core.resources".

Finalmente, visitaremos uma classe muito importante. Na guia **Overview**, na seção "General Information" há uma opção chamada **Activator**. Nela deve-se definir a classe cuja ocorrência será criada quando o plug-in for ativado. No caso do plug-in Hello World, a classe de ativação ("Activator.java") encontra-se dentro do pacote hello da pasta "src" do projeto.

6 Criando um Editor de Código

Criaremos agora um editor de código simples que permitirá a edição de código com *syntax highlighting*, isto é, com formatação de texto. No nosso caso, palavras chave, comentários e strings serão diferenciadas do código corrente. Para facilitar a criação do editor, separamos o processo em aproximadamente três partes, onde cada uma delas adiciona funcionalidades ao editor obtido na etapa anterior, até que na última teremos o resultado desejado.

6.1 Um primeiro editor: a criação

A seguir indicamos os passos necessários para a criação do plug-in que inicialmente suportará as funcionalidades do editor de código e posteriormente as demais funcionalidades propostas neste texto. Observe, portanto, que as etapas que seguem na construção do plug-in são diretamente dependentes desta subseção.

Estando no ambiente Eclipse:

1. Crie um projeto através do assistente de criação de plug-ins (*wizard*) conforme abaixo:

(a) Na etapa "Plug-in Project" altere **Project name** para Notus.

(b) Em "Plug-in Content" altere **Plug-in name** para Notus Plug-in; Observe que em "Plug-in Options" a opção para geração da classe Activator esta marcada indicado que o Eclipse a produzirá automaticamente. Além dessa, outra opção importante, é a que indica que o plug-in contribuirá para interface de usuário. Isso significa que o Eclipse automaticamente incluirá a dependência aos plug-ins de interface.

(c) Em "Templates" desmarque a opção **Create a plug-in using one of the templates** – não utilizaremos nenhum modelo pré-concebido.

2. O editor que faremos será dependente dos plug-ins de edição do eclipse, dessa forma será necessário adicionar a dependência org.eclipse.ui.editors na guia **Dependencies** do arquivo MANIFEST.MF.

3. Agora adicionaremos o ponto de extensão do editor de código ao plug-in. Para fazê-lo na guia **Extensions** do arquivo MANIFEST.MF adicione a extensão org.eclipse.ui.editors.

4. Com o botão direito do mouse clique sobre o ponto de extensão adicionado e no menu **new** clique na opção **editor**.

5. Altere o editor adicionado acima conforme abaixo:

   (a) **Id**: notus.editor.NotusEditor

   (b) **Name**: Notus Editor

   (c) Defina o ícone que será utilizado no editor através da opção **icon**.

   (d) Em **Extensions** deve-se definir qual a extensão dos arquivos que serão utilizados pelo editor, no nosso caso "nts".

   (e) Devemos informar também, qual classe contribuirá com recursos básicos de edição para o seu plug-in (não é nosso intento reinventar a roda, por isso utilizaremos uma classe pré-concebida disponível no Eclipse), assim na opção, **contributorClass** adicione a classe "org.eclipse.ui.texteditor.BasicTextEditorActionContributor".

6. Crie um pacote chamado editor dentro do pacote notus da pasta src do projeto. Observe que nesse caso o pacote será nomeado como *notus.editor*.

7. Crie uma classe dentro do pacote notus.editor chamada NotusEditor que estende a classe org.eclipse.ui.editors.text.TextEditor.

8. Volte a guia **Extensions** do arquivo MANIFEST.MF e defina a opção **class** com o nome da classe criada acima, isto é, notus.editor.NotusEditor. Isso diz ao Eclipse qual é a classe responsável pela extensão.

Pronto, a criação do "editor básico" esta encerrada, agora é possível compilar o plug-in e executá-lo. Durante a execução, pode parecer que nada aconteceu, mas ao editar um arquivo com extensão "nts", o editor criado é executado. Caso haja algum problema durante a compilação, pode ser porque é necessário incluir ao projeto os pacotes *org.eclipse.jface.text* e *org.eclipse.ui.texteditor* na opção **Imported Packages** da guia **Dependencies** do arquivo MANIFEST.MF.

6.2 Um segundo editor: detectando e colorindo comentários

Nosso objetivo agora será incrementar o editor criado para que ele detecte comentários e os colora. Porém antes de prosseguirmos, alguns conceitos relativos a forma como a coloração sintática (*syntax coloring*, ou *syntax highlighting*) serão necessários. A coloração sintática fornecida pelo *framework* de texto da plataforma utiliza o modelo de invalidação, reparo e reconciliação (*damage*, *repair* e *reconciling*). Para cada alteração em um documento, um conciliador de apresentação determina qual região da apresentação visual deve ser invalidada e como repará-la. Conforme o usuário modifica o texto em um editor, partes do editor precisam ser redesenhadas para mostrar as mudanças. A computação do texto que precisa ser re-desenhado é conhecida como invalidação. O reparo trata da recuperação de todas as informações necessárias para descrever como reparar um texto invalidado.

Como "bons" programadores, criaremos uma interface que conterá as constantes de cor utilizadas pela parte de edição do plug-in. A interface deverá ser criada dentro do pacote notus.editor e se chamar INotusColorConstants. Veja a o código completo a seguir:

```
package notus.editor;

import org.eclipse.swt.graphics.RGB;

public interface INotusColorConstants {
    RGB COMMENT = new RGB(63,127,95);
}
```

Agora criaremos uma espécie de administrador das cores utilizadas no editor. Basicamente quando algum usuário desse administrador solicitar uma

cor (através do método *getColor*), dado um objeto do tipo RGB, ele deverá fornecer o objeto Color correspondente àquela cor. Faremos isso para evitar que a cada requisição de uma cor, um novo objeto do tipo Color seja criado. Na primeira requisição de uma cor a criaremos e a partir daí todos os demais pedidos deverão compartilhar o mesmo objeto. Além disso, precisaremos criar um método para liberar os recursos do sistema operacional utilizados por cada ocorrência do objeto Color, que apesar de ser programado em Java o manual do Elipse [9] nos atribui essa responsabilidade, isto é, no método *Dispose* do administrador de cores, para cada ocorrência de Color, deveremos chamar o método dispose correspondente. Veja o código:

```
package notus.editor;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

import org.eclipse.swt.graphics.Color;
import org.eclipse.swt.graphics.RGB;
import org.eclipse.swt.widgets.Display;

public class ColorManager {

    protected Map<RGB, Color> fColorTable = new HashMap<RGB, Color>(10);

    public void dispose() {
        Iterator e = fColorTable.values().iterator();
        while (e.hasNext())
             ((Color) e.next()).dispose();
    }

    public Color getColor(RGB rgb) {
        Color color = (Color) fColorTable.get(rgb);
        if (color == null) {
            color = new Color(Display.getCurrent(), rgb);
            fColorTable.put(rgb, color);
        }
        return color;
    }
}
```

Até agora trabalhamos apenas com cores, mas como já era de se esperar,

mais cedo ou mais tarde precisaríamos de um detector (*scanner*) de palavras chave, comentários, etc. O Eclipse fornece um ferramental básico para essa tarefa *não* tão simples. No nosso caso utilizaremos uma classe conhecida como *RuleBasedScanner*, segundo o manual do Eclipse [9] trata-se de um *scanner* que pode ser "programado" com uma sequência de regras. O *scanner* deve ser usado para obter o próximo *token* que identifica o texto a ser formatado. Isso ocorre através da avaliação de suas regras em sequência até que uma delas tenha sucesso. Como veremos a diante, os objetos usuários desse *scanner* esperarão que os *tokens* retornados possuam a formatação das palavras-chave em seu interior, o que nos fará utilizar o gerenciador cores e a interface de constantes criados anteriormente. Finalmente, antes de apresentarmos o código do *scanner*, será necessário adicionar a dependência aos plug-ins do pacote org.eclipse.jface.text (para fazê-lo, adicione o nome do pacote na guia **Dependencies** do arquivo MANIFEST.MF). Depois disso estamos prontos para apresentar o código do *scanner*:

```
package notus.editor;

import org.eclipse.jface.text.TextAttribute;
import org.eclipse.jface.text.rules.IRule;
import org.eclipse.jface.text.rules.IToken;
import org.eclipse.jface.text.rules.MultiLineRule;
import org.eclipse.jface.text.rules.RuleBasedScanner;
import org.eclipse.jface.text.rules.SingleLineRule;
import org.eclipse.jface.text.rules.Token;

public class NotusScanner extends RuleBasedScanner {
    public final static int BOLD = 1 << 32;

    public NotusScanner(ColorManager colorManager) {
        IToken comment =
            new Token(
                new TextAttribute(
                   colorManager.getColor(INotusColorConstants.COMMENT)));

        IRule[] rules = new IRule[1];

        // Rule for one line comment detection
        rules[0] = new EndOfLineRule("//",comment);

        setRules(rules);
```

```
    }
}
```

Na classe acima, os imports do tipo "org.eclipse.jface.text" obtêm funcionalidades de um framework fornecido pelo Eclipse para manipulação e criação de documentos de texto (os interessados nos detalhes do pacote devem consultar [9]). Como havíamos dito o *scanner* é baseado em regras que permitem a detecção de tokens, motivação para que a classe *NotusScanner* estenda a classe *RuleBasedScanner* fornecida pelo Eclipse. No início da classe definimos uma constante para identificar o tipo de conteúdo tratado pelo editor (*NOTUS_CONTENT*) e outra para a formatação de textos em negrito (*BOLD*). Na construtora da classe criamos o token que representará os comentários que desejamos detectar e formatar. Também conforme dito anteriormente, os atributos de formatação do texto devem ser colocados dentro do token para que os objetos clientes do *scanner* possam formatá-lo conforme o especificado nessa classe. Na sequência cria-se o vetor de regras de formatação e a regra para detecção de comentários. A construtora, e o código, terminam definindo que as regras criadas acima devem ser utilizadas pelo *scanner*.

Continuando rumo ao editor, precisaremos de uma classe de configuração. Ela fará as ligações entre as classes fornecedoras de um recurso e as classes utilizadoras desses recursos. No nosso caso será algo bem simples, veja a classe *NotusConfiguration*:

```
package notus.editor;

import org.eclipse.jface.text.IDocument;
import org.eclipse.jface.text.presentation.IPresentationReconciler;
import org.eclipse.jface.text.presentation.PresentationReconciler;
import org.eclipse.jface.text.rules.DefaultDamagerRepairer;
import org.eclipse.jface.text.source.ISourceViewer;
import org.eclipse.jface.text.source.SourceViewerConfiguration;

public class NotusConfiguration extends SourceViewerConfiguration {
    private NotusScanner scanner;
    private ColorManager cmanager;

    public NotusConfiguration(ColorManager cmanager) {
        this.cmanager = cmanager;
    }
```

```
    public String[] getConfiguredContentTypes(
                        ISourceViewer sourceViewer) {
        return new String[] {IDocument.DEFAULT_CONTENT_TYPE,
                             NotusScanner.NOTUS_CONTENT};
    }

    protected NotusScanner getNotusScanner() {
        if (scanner == null) {
            scanner = new NotusScanner(cmanager);
        }
        return scanner;
    }

    public IPresentationReconciler getPresentationReconciler(
                  ISourceViewer sourceViewer) {
        PresentationReconciler reconciler =
                          new PresentationReconciler();

        DefaultDamagerRepairer dr = new DefaultDamagerRepairer(
                                        getNotusScanner());
        reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE);
        reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE);

        return reconciler;
    }
}
```

Na construtora *NotusConfiguration* cria-se uma ocorrência do administrador de cores. Na sequência há um método utilizado pelo Eclipse para saber o tipo de conteúdo tratado pelo editor, outro para obter o *scanner* utilizado pelo editor – note que nessa função o *scanner* é criado se necessário ou apenas retornado caso já exista –, e finalmente, conforme requisitado pelo Eclipse, define-se quem é o responsável pela reconciliação (*reconciler*, note o uso do *scanner*), pela invalidação e pelo reparo. Para os dois últimos utilizou-se uma versão padrão fornecida pelo Eclipse. Além disso, para ambos define-se esse tratador padrão para o tipo de conteúdo padrão.

Finalmente, para concluir nosso primeiro passo, devemos alterar a classe *NotusEditor*, quem efetivamente representará o editor e reunirá tudo o que fizemos até agora, veja o código:

```
package notus.editor;
```

```
import org.eclipse.ui.editors.text.FileDocumentProvider;
import org.eclipse.ui.editors.text.TextEditor;

public class NotusEditor extends TextEditor {
    private ColorManager colorManager;

    public NotusEditor() {
        super();

        colorManager = new ColorManager();
        setSourceViewerConfiguration(
                  new NotusConfiguration(colorManager));

        setDocumentProvider(new FileDocumentProvider());
    }

    public void dispose() {
        colorManager.dispose();
        super.dispose();
    }
}
```

A classe *NotusEditor* cria um administrador de cores, depois cria a classe de configuração do editor configuração (*NotusConfiguration*) utilizando o administrador de cores e atribui a responsabilidade da configuração ao objeto criado. O método *dispose* é uma exigência do Eclipse para liberar, no caso, tanto o editor de código quanto o administrador de cores.

Pronto, agora já é possível testar o editor. Basta executar o plug-in, criar um arquivo do tipo nts e digitar, entre outras coisas, um comentário no formato //.

6.3 Detectando e colorindo palavras chaves e *strings* de caracteres

Agora que temos um editor inicial e "sabemos" como ele funciona, adicionaremos mais funcionalidades até que ele fique praticamente da forma desejada. Incluiremos recursos para a detecção de *strings* de caracteres delimitadas por aspas duplas e de palavras chave, como *if*, *import*, *public*, etc.

Como pontapé inicial dessa etapa, adicionaremos à interface *INotusColorConstants* as cores para as palavras chave, para as strings de caracteres e

para o restante do texto (não formatado). Veja:

```
package notus.editor;

import org.eclipse.swt.graphics.RGB;

public interface INotusColorConstants {
    RGB COMMENT = new RGB(63,127,95);
    RGB KEY_WORD = new RGB(127,0,85);
    RGB STRING = new RGB(4,0,170);
    RGB DEFAULT = new RGB(0,0,0);
}
```

Agora será necessário criar a base para a detecção das palavras chave. Inicialmente definiremos uma interface que conterá a lista das palavras chave:

```
package notus.editor;

public interface INotusKeyWords {
    public static final String[] KEY_WORDS = {
        "module", "input", "stdin", "output",
        "stdout", "append", "syntactic", "semantic",
        "token", "Bool", "Int", "Long",
        "Float", "Double", "Char", "String",
        "is", "extend", "with", "ignore",
        "element", "private", "public", "import",
        "end", "function", "if", "then",
        "else", "true", "false", "case",
        "of", "context", "match", "with",
        "in"
    };
}
```

Criaremos uma classe que será a responsável pela detecção dos espaços em branco, algo que é um requisito do Eclipse, e diga-se de passagem o que é bem plausível, veja:

```
package notus.editor;

import org.eclipse.jface.text.rules.IWhitespaceDetector;

public class WhitespaceDetector implements IWhitespaceDetector {
```

```
    public boolean isWhitespace(char c) {
        return (Character.isWhitespace(c));
    }

}
```

O código dessa classe é muito simples, ele usa a função da classe *Character* de Java para verificar se o caractere fornecido é um espaço em branco, retornando *true* em caso afirmativo e *false* caso contrário.

Além de detectar espaços em branco, como é de esperar, precisaremos detectar também as palavras chave, para isso podemos utilizar uma classe que serve como detector de palavras, veja o código a seguir:

```
package notus.editor;

import org.eclipse.jface.text.rules.IWordDetector;

public class WordDetector implements IWordDetector {
    public boolean isWordPart(char c) {
        return !(Character.isSpaceChar(c) ||
                 Character.isISOControl(c) ||
                 c == 65535 || c == '"');
    }

    public boolean isWordStart(char c) {
        return !(Character.isSpaceChar(c) ||
                 Character.isISOControl(c) ||
                 c == 65535 || c == '"');
    }

}
```

O método *isWordPart* informa ao eclipse se um caractere pode ser parte de uma palavra ou não. Já o método *isWordStart* serve para verificar se uma palavra pode iniciar com o caractere dado. No nosso caso essas palavras corresponderão as palavras chave da linguagem.

Criadas as classes necessárias para praticamente finalizar o editor de código, precisaremos agora fazer uma última alteração na classe *scanner* antes de concluirmos essa etapa. Lembre-se que essa classe tratava das regras de detecção dos comentários, assim, sem muita surpresa, adicionaremos as regras para detecção das palavras chave e das strings, a seguir veja o

código alterado do construtor da classe *NotusScanner* e adição do import *org.eclipse.jface.text.rules.WordRule*:

```
package notus.editor;

import org.eclipse.jface.text.TextAttribute;
import org.eclipse.jface.text.rules.EndOfLineRule;
import org.eclipse.jface.text.rules.IRule;
import org.eclipse.jface.text.rules.IToken;
import org.eclipse.jface.text.rules.RuleBasedScanner;
import org.eclipse.jface.text.rules.SingleLineRule;
import org.eclipse.jface.text.rules.Token;
import org.eclipse.jface.text.rules.WordRule;

public class NotusScanner extends RuleBasedScanner {
    public final static String NOTUS_CONTENT = "__notus_content";
    public final static int BOLD = 1 << 32;

    public NotusScanner(ColorManager colorManager) {
        super();
        IToken instr =
            new Token(
              new TextAttribute(
                colorManager.getColor(INotusColorConstants.KEY_WORD),
                        null,
                        BOLD));

        IToken other =
            new Token(
              new TextAttribute(
                colorManager.getColor(INotusColorConstants.DEFAULT)));

        IToken comment =
            new Token(
              new TextAttribute(
                colorManager.getColor(INotusColorConstants.COMMENT)));
        IToken text =
            new Token(
              new TextAttribute(
                colorManager.getColor(INotusColorConstants.STRING)));
```

```
        IRule[] rules = new IRule[3];

        // Rule for key word detection
        WordRule wrule = new WordRule(new WordDetector(), other);

        // Adds the key words
        for (String i : INotusKeyWords.KEY_WORDS)
            wrule.addWord(i, instr);
        rules[2] = wrule;

        // Rule for one line comment detection
        rules[1] = new EndOfLineRule("//",comment);

        // Rule for strings
        rules[0] = new SingleLineRule("\"","\"", text);

        setRules(rules);
    }
}
```

Nesse código inicialmente são criados os *tokens* de acordo com tipo de conteúdo a ser formatado (devemos retornar a formatação do texto dentro do token que o representa – por isso na criação de cada token há também a criação de um objeto que representa o atributo do texto).

Terminado o editor de código que, conforme o planejado, permite a escrita de código formatado, passaremos agora para a criação de projetos. A seguir veremos como criar um tipo de projeto para uma linguagem.

7 Criando projetos

Nosso foco agora será a produção de uma interface que permita a criação de projetos. Basicamente o usuário deverá ser auxiliado por um assistente (*wizard*) que efetue a configuração inicial do ambiente para que posteriormente possamos compilar e executar código.

7.1 Extensões e afins

A primeira tarefa a ser feita é a inclusão da extensão *org.eclipse.ui.newWizards*. Ela dirá ao Eclipse que criaremos um assistente para criação de projetos e atribuirá essa responsabilidade a uma dada classe. Mais especificamente a

extensão diz que o plug-in adicionará um assistente a opção **File** $\gg$ **New** do Eclipse. Nessa opção há uma série de categorias de assistentes, precisamos criar uma para o nosso plug-in ou ele será colocado na categoria **Other**, para isso,

1. clique com o botão direito do mouse sobre a extensão *org.eclipse.ui.newWizards*;
2. selecione a opção **New** e depois **category**.

Agora configuraremos a categoria criada, veja como ficam as opções:

1. **id**: notus.newWizards – essa opção deve fornecer um identificador único para a categoria criada;
2. **name**: Notus – corresponde ao texto que será exibido na lista de categorias do menu **New** do Eclipse.

Criada a categoria, precisamos de algo para colocar dentro dela, criaremos então, nosso assistente. Para fazê-lo siga os passos:

1. clique com o botão direito do mouse sobre a extensão *org.eclipse.ui.newWizards*;
2. selecione a opção **New** e depois **wizard**.

Agora precisamos configurar a opção **wizard** criada, veja como ficam suas opções abaixo:

1. **id**: notus.wizards.newproject – identificador único para o assistente;
2. **name**: Notus Project – texto que será mostrado para o usuário para identificar o assistente;
3. **icon**: adicione um ícone de sua preferência ao projeto e o utilize – esse ícone será mostrado nas opções do Eclipse que envolverem o assistente;
4. **category**: notus.newWizards – aqui conectam-se a categoria criada e o assistente. Devemos, portanto, preencher essa opção com o identificador da categoria criada acima;
5. **project**: true – quando definida para **true** indica que o assistente trata da criação de projetos, já o **false**, como é de se imaginar, é utilizado para os demais casos.

Dentre essas opções observe que **class** ficou em branco. A seguir, depois de criarmos a classe que cuidará do assistente retornaremos para preencher essa opção.

7.2 Criando a classe responsável pelo assistente

Criaremos agora a classe que será a responsável pelo provimento do assistente para criação de projetos. A primeira coisa que faremos é criar uma interface que conterá as constantes utilizadas no plug-in, por ser de uso comum, faremos isso dentro do pacote *notus*. Veja a seguir o código da interface:

```
package notus;

public interface PluginConstants
{

}
```

Por enquanto interface ficará vazia, pois a preencheremos a medida que for necessário. Precisamos agora criar um pacote que armazenará alguns recursos utilizados pelo plug-in. Mais especificamente, todo programa Notus deve possuir um módulo Main.nts que começa com *"module Main"* e termina com *"end"*.

O que queremos fazer então, é gerar automaticamente esse módulo no ato da criação do projeto. Além disso, também precisaremos de um pacote chamado *notus.resources* para conter outros recursos. Resumindo, dentro dele guardaremos uma cópia de um módulo Main genérico que será adicionado ao projeto do usuário e outros recursos que não são relevantes por hora. Precisamos importar para dentro desse pacote o tal Main genérico, por isso

1. crie o arquivo Main.nts utilizando um editor de textos;

2. importe o arquivo para dentro do pacote *notus.resources*.

Além disso, criaremos agora uma classe no pacote *notus.resources* que nos permita obter algumas informações locais em tempo de execução. Veja o código e a explicação a seguir:

```
package notus.resources;

import java.util.*;

public final class Resources {
   /**
    * the ResourceBundle for this project
```

```
    */
   public static final ResourceBundle resources =
      ResourceBundle.getBundle("notus.resources.Strings");

   /**
    * returns the resource string associated with the key
    */
   public static final String getString(String key)
      throws MissingResourceException {
      return resources.getString(key);
   }
}
```

Conforme já resumido acima, essa classe fornece funcionalidades básicas para recuperar informações locais armazenadas no plug-in. No nosso caso teremos um arquivo chamado "String.properties" que se encontrará no projeto conforme o caminho indicado pela chamada do método *getBundle*. Esse arquivo conterá as mensagens emitidas ao usuário durante o uso do plug-in e essas strings poderão ser obtidas através da operação *getString*. Para ficar mais claro, veja o conteúdo do arquivo String.properties:

```
# default (English) resource strings for Notus
notdirectory="{0}" is not a directory
cannotcreatedirectory=Cannot create "{0}" directory
progressdone=done: {0}
progressignored=ign.: {0}
suspiciouslink=Suspicious hyperlink, most likely a typo: {0}
nestedelement={0} cannot be nested.
dirrequired=dir attribute required in {0}
error=Error
fatal=Fatal
warning=Warning
info=Info
begin=Start processing of {0}
end=Done
templateproblem=Problem when loading the style sheet for {0}
# resources for the Eclipse plugin
eclipse.dialogtitle=Notus
eclipse.builderror=Build failed.
eclipse.projecterror=Could not create the new project.
eclipse.location.format={0}: {1}, {2}
```

```
eclipse.status=Status
eclipse.message=Message
eclipse.location=Location
eclipse.newprojectname=Notus Project Name
eclipse.newprojectdescription=Choose the name of the new project.
eclipse.creatingproject=Creates new Notus project.
eclipse.creatingdirectories=Creates file structure.
eclipse.creatingfiles=Creates basic files.
eclipse.launcherror=Could not run Notus on this project.
eclipse.builderror=Could not build the project with Notus.
eclipse.hasnature=Is Notus project.
eclipse.build="Run Notus" performs a build.
eclipse.src=Source directory:
eclipse.publish=Publish directory:
eclipse.rules=Style sheet directory:
eclipse.propertyerror=Error while initializing the property page.
eclipse.properties=Project Properties
eclipse.level=Display in the console:
eclipse.level.all=All messages
eclipse.level.info=Error, warning and information messages
eclipse.level.warning=Error and warning messages
eclipse.level.error=Error messages only
```

Nosso próximo passo será definir a natureza (do inglês *nature*) do projeto a que nosso assistente se associará. As naturezas permitem aos plug-ins definirem um tipo específico de projeto a que se relacionam. Isso pode ser utilizado, por exemplo, para atribuir um comportamento específico aos diferentes tipos de projeto. Vamos, então, a definição da natureza do projeto. Para isso, inicialmente criaremos a classe que será responsável pela gerência da natureza do projeto, devemos fazê-lo dentro do pacote notus, veja:

```
package notus;

import org.eclipse.core.runtime.*;
import org.eclipse.core.resources.*;

public class NotusProjectNature
   implements IProjectNature, PluginConstants {
   private IProject project;

   public void configure()
```

```
   throws CoreException {
      IProjectDescription description = project.getDescription();
      if(!hasBuildSpec(description.getBuildSpec())) {
         ICommand[] old = description.getBuildSpec(),
                    specs = new ICommand[old.length + 1];
          System.arraycopy(old,0,specs,0,old.length);
          ICommand command = description.newCommand();
          command.setBuilderName(BUILDER_ID);
          specs[old.length] = command;
          description.setBuildSpec(specs);
          project.setDescription(description,
                            new NullProgressMonitor());
      }
}

// should we erase the Notus properties, if any?
public void deconfigure()
   throws CoreException {
      IProjectDescription description = project.getDescription();
      int count = getBuildSpecCount(description.getBuildSpec());
      if(count != 0) {
         ICommand[] old = description.getBuildSpec(),
                    specs = new ICommand[old.length - count];
         int i = 0,
             j = 0;
         while(i < old.length) {
            if(!old[i].getBuilderName().equals(BUILDER_ID))
               specs[i] = old[j++];
            i++;
         }
         description.setBuildSpec(specs);
         project.setDescription(description,
                           new NullProgressMonitor());
      }
}

public IProject getProject() {
   return project;
}

public void setProject(IProject project) {
```

```
        this.project = project;
    }

    private boolean hasBuildSpec(ICommand[] commands) {
       return getBuildSpecCount(commands) != 0;
    }

    private int getBuildSpecCount(ICommand[] commands) {
        int count = 0;
        for(int i = 0;i < commands.length;i++)
           if(commands[i].getBuilderName().equals(BUILDER_ID))
              count++;
        return count;
    }
}
```

Rapidamente, o método *configure* configura a natureza para o projeto, sendo chamado quando a natureza é adicionada ao projeto. Em contrapartida o método *deconfigure* desconfigura, a natureza para o seu projeto. É invocado quando a natureza é removida do projeto. Finalmente, o método *getProject* obtém e *setProject* define o projeto a que a natureza se refere. Os dois métodos seguintes serão comentados adiante.

Para continuarmos é necessário definir uma constate BUILD_ID que foi utilizada acima, para isso à interface *PluginConstants* adicionamos o trecho de código a seguir. De interesse posterior adiaremos sua explicação até o momento apropriado.

```
/**
 * Notus Builder id from plugin.xml
 */
public static final String BUILDER_ID = "notus.builder";
```

1. adicione a extensão *org.eclipse.core.resources.natures*,
2. altere as opções da extensão como abaixo:
   (a) **ID**: notus.nature – identificador único para a natureza do projeto;
   (b) **Name**: Notus Nature – texto utilizado para identificar a natureza.
3. clique com o botão direito do mouse sobre a extensão e crie um item do tipo (runtime);

4. clique com o botão direito do mouse sobre o subitem (runtime) e crie um item do tipo run;

5. altere a classe do item run para notus.NotusProjectNature.

Rumo ao final dessa seção, será necessário definir três constantes na interface *PluginConstants*, veja como fica o código:

```
package notus;

public interface PluginConstants {
    /**
     * plugin id from plugin.xml
     */
    public static final String PLUGIN_ID = "Notus";

    /**
     * identifier for the project nature
     */
    public static final String NATURE_ID = "notus.nature";

    /**
     * default value for the source directory name
     */
    public static final String DEFAULT_SOURCE_DIR = "src";

    /**
     * default value for the rules directory name
     */
    public static final String DEFAULT_OUT_DIR = "out";
}
```

A constante *PLUGIN_ID* corresponde ao identificador do plug-in conforme o definido pelo arquivo MANIFEST.MF, já *NATURE_ID* corresponde ao identificador da natureza do projeto. Apesar de ambos encontrarem-se disponíveis no plug-in utilizou-se constantes simbólicas para que seu acesso fosse facilitado. O compilador da linguagem Notus exige que haja duas pastas para a compilação, uma para conter o código fonte da especificação semântica e outra para conter a saída gerada pelo compilador. Respectivamente, as constantes *DEFAULT_SOURCE_DIR* e *DEFAULT_OUT_DIR* tratam disso, elas são usadas para criar as pastas correspondentes dentro dos projetos do tipo Notus.

Finalmente, criaremos o pacote que conterá a classe que codifica o assistente, daremos a ele o nome de *notus.wizards*. Com o pacote criado devemos partir então, para a classe que fornecerá o assistente. Para criá-la precisaremos incluir a dependência *org.eclipse.core.resources* e importar os pacotes *org.eclipse.ui.actions* e *org.eclipse.ui.dialogs* (tanto a dependência quanto os pacotes podem ser incluídos na guia **Dependecies** do arquivo MANIFEST.MF).

Terminado tudo isso o código da classe *NewNotusProjectWizard*, responsável pelo assistente fica assim:

```
package notus.wizards;

import java.io.*;
import org.eclipse.ui.*;
import org.eclipse.core.runtime.*;
import org.eclipse.jface.dialogs.*;
import org.eclipse.core.resources.*;
import org.eclipse.jface.wizard.Wizard;
import org.eclipse.jface.resource.ImageDescriptor;
import java.lang.reflect.InvocationTargetException;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.ui.actions.WorkspaceModifyOperation;
import org.eclipse.ui.dialogs.WizardNewProjectCreationPage;

import notus.PluginConstants;
import notus.resources.Resources;

/**
 * Creates a new Notus project in the Eclipse.
 * Creates the default file (Main.nts).
 *
 */
public class NewNotusProjectWizard
             extends Wizard
             implements INewWizard, PluginConstants {
    /**
     * The main page on the wizard
     * obtain the project name and location.
     */
    private WizardNewProjectCreationPage namePage;
```

```
/**
 * @see org.eclipse.ui.IWorkbenchWizard#init(
        org.eclipse.ui.IWorkbench,
        org.eclipse.jface.viewers.IStructuredSelection)
 */
public void init(IWorkbench workbench,
                    IStructuredSelection selection) {
    setNeedsProgressMonitor(true);
}

/**
 * @see org.eclipse.jface.wizard.IWizard#addPages()
 */
public void addPages() {
    try {
        super.addPages();
        namePage = new WizardNewProjectCreationPage(
                                "NewNotusProjectWizard");
        namePage.setTitle(Resources.getString(
                                "eclipse.newprojectname"));
        namePage.setDescription(Resources.getString(
                                "eclipse.newprojectdescription"));
        namePage.setImageDescriptor(ImageDescriptor.createFromFile(
                  getClass(),
                  "/org/notus/eclipse/resources/newproject.gif"));

        addPage(namePage);
    }
    catch(Exception x) {
        reportError(x);
    }
}

/**
 * User has clicked "Finish", we need to create the project.
 * @see #createProject(IProjectMonitor)
 * @see org.eclipse.jface.wizard.IWizard#performFinish()
 */
public boolean performFinish() {
    try {
        getContainer().run(false,true,new
```

```
                        WorkspaceModifyOperation() {
            protected void execute(IProgressMonitor monitor)  {
                createProject(monitor != null ?
                                    monitor :
                                    new NullProgressMonitor());
            }
        });
    }
    catch(InvocationTargetException x) {
        reportError(x);
        return false;
    }
    catch(InterruptedException x) {
        reportError(x);
        return false;
    }
    return true;
}

/**
 * This is the actual implementation for project creation.
 * @param monitor reports progress on this object
 */
protected void createProject(IProgressMonitor monitor) {
    monitor.beginTask(
        Resources.getString("eclipse.creatingproject"),50);
    try {
        IWorkspaceRoot root =
            ResourcesPlugin.getWorkspace().getRoot();
        monitor.subTask(Resources.getString(
                    "eclipse.creatingdirectories"));
        IProject project = root.getProject(namePage.getProjectName());
        IProjectDescription description =
                    ResourcesPlugin.getWorkspace().
                            newProjectDescription(project.getName());
        if(!Platform.getLocation().equals(namePage.getLocationPath()))
            description.setLocation(namePage.getLocationPath());
        project.create(description,monitor);
        monitor.worked(10);
        project.open(monitor);
        description = project.getDescription();
```

```
        description.setNatureIds(new String[] { NATURE_ID });
        project.setDescription(description,
                        new SubProgressMonitor(monitor,10));
        IPath projectPath = project.getFullPath(),
        srcPath = projectPath.append(DEFAULT_SOURCE_DIR),
        outPath = projectPath.append(DEFAULT_OUT_DIR);

        IFolder srcFolder = root.getFolder(srcPath),
        outFolder = root.getFolder(outPath);
        createFolderHelper(srcFolder,monitor);
        createFolderHelper(outFolder, monitor);
        monitor.worked(10);
        monitor.subTask(Resources.getString("eclipse.creatingfiles"));
        IPath mainPath = srcPath.append("Main.nts");
        IFile indexFile = root.getFile(mainPath);
        Class clasz = getClass();

        // Create the main file
        InputStream mainIS = clasz.getResourceAsStream(
                                        "/notus/resources/Main.nts");
        indexFile.create(mainIS,false,
                                new SubProgressMonitor(monitor,10));
    }
    catch(CoreException x) {
        reportError(x);
    }
    finally {
        monitor.done();
    }
}

/**
 * Displays an error that occured during the project creation.
 * @param t details on the error
 */
private void reportError(Exception t) {
    IStatus status;
    if(t instanceof CoreException)
        status = ((CoreException)t).getStatus();
    else
        status = new Status(IStatus.ERROR,
```

```
                    PluginConstants.PLUGIN_ID,
                    IStatus.ERROR,
                    t.getMessage() != null ?
                         t.getMessage() : t.toString(), t);

        ErrorDialog.openError(getShell(),
                Resources.getString("eclipse.dialogtitle"),
                Resources.getString("eclipse.projecterror"),
                status);
    }

    /**
     * Helper method: it recursively creates a folder path.
     * @param folder
     * @param monitor
     * @throws CoreException
     * @see java.io.File#mkdirs()
     */
    private void createFolderHelper(IFolder folder,
                                    IProgressMonitor monitor)
    throws CoreException {
        if(!folder.exists()) {
            IContainer parent = folder.getParent();
            if(parent instanceof IFolder
                    && (!((IFolder)parent).exists()))
                createFolderHelper((IFolder)parent,monitor);
            folder.create(false,true,monitor);
        }
    }
}
```

Nesse código o método *addPages* é responsável pela criação da tela do assistente, no nosso caso ele cria a única etapa do assistente. O método *createProject*, trata da criação do projeto, ele define a natureza do projeto, cria duas pastas uma src e outra out, para respectivamente entrada e saída do compilador Notus, além de adicionar o arquivo Main.nts ao projeto – nesse caso observe que esse arquivo já foi criado e encontra-se no projeto do plug-in. O que na verdade é feito na criação do projeto é uma cópia do arquivo pré-concebido para o novo projeto. Os dois métodos restantes também são simples, *reportError* é o responsável pela exibição das mensagens de erro durante a criação do projeto e *createFolderHelper* é um auxiliar utilizado

para a criação de uma árvore de diretórios.

Como já dito, para podermos colocar o plug-in em funcionamento precisaremos revisitar a extensão *org.eclipse.ui.newWizards*, e lá atribuir a responsabilidade da geração do assistente à classe *NewNotusProjectWizard*. Para fazê-lo

1. na guia **Extensions** do arquivo MANIFEST.MF expanda a extensão *org.eclipse.ui.newWizards*;
2. selecione a opção **Notus Project (wizard)**;
3. altere o campo **class** adicionando a classe *NewNotusProjectWizard*.

Seção terminada, ao salvar e executar o projeto, no menu **File ¿ New** deverá existir a opção para criação de um projeto Notus.

8 Compilando Código

Começaremos agora uma parte grande. Para compilar código precisaremos mostrar mensagens, o que nos renderá algum trabalho extra além daquele que já seria dado apenas pela compilação. Primeiro produziremos a parte para exibição de mensagens e a seguir encerraremos a seção com a compilação propriamente dita.

8.1 Exibindo mensagens

Inicialmente faremos algumas adições às constantes que possuíamos antes,

```
/**
 * console (MessengerView) id from plugin.xml
 */
public static final String MESSENGER_ID = "notus.eclipse.views.console";

/**
 * preference name for the console level
 */
public static final String LEVEL_PREFERENCE_NAME = "level";

/**
 * preference value for the console level, show all messages
 * fatal, error, warning, info, progress
```

```
 */
public static final String LEVEL_ALL = "all";

/**
 * preference value for the console level,
 * show all progress messages
 * fatal, error, warning, info
*/
public static final String LEVEL_INFO = "info";

/**
 * preference value for the console level,
 * show error & warning messages
 * fatal, error, warning
 */
public static final String LEVEL_WARNING = "warning";

/**
 * preference value for the console level, show error messages
 * fatal, error
 */
public static final String LEVEL_ERROR = "error";
```

Agora faremos algumas adições ao arquivo de recursos que contém as strings do programa (Strings.properties),

```
eclipse.status=Status
eclipse.message=Message
eclipse.location=Location
error=Error
fatal=Fatal
warning=Warning
info=Info
progressdone=done: {0}
progressignored=ign.: {0}
end=Done
```

Criaremos, então, a classe que codificará uma janela dentro do Eclipse para que as mensagens de compilação possam ser exibidas. A classe, que foi criada dentro do pacote *notus.views*, é fornecida a seguir com uma breve descrição de seu código.

```
package notus.views;

import java.io.*;
import org.eclipse.ui.*;
import org.eclipse.swt.SWT;
import org.eclipse.ui.part.*;
import java.text.MessageFormat;
import org.eclipse.jface.util.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.jface.preference.*;

import notus.Activator;
import notus.PluginConstants;
import notus.resources.Resources;

/**
 * Implements an Notus messenger as an Eclipse part.
 * Shows Notus messages in the Eclipse workspace.
 */
public class MessengerView
    extends ViewPart
    implements PluginConstants {
   /**
    * On-screen control.
    */
   private Table table;

   /**
    * Level of messages to display.
    */
   private int level;

   /**
    * Constant to control the display levels.
    */
   private static final int ALL = 0,
                            INFO = 1,
                            WARNING = 2,
                            ERROR = 3;
```

```
/**
 * This class updates the table (add an item) in a separate thread
 * (typically used to synchronized with the UI thread).
 * It creates a new instance of this class for every update.
 */
private class AddItem
   implements Runnable {
   private String[] text;
   public AddItem(String type,String message,String location) {
      text = new String[] { type, message, location };
   }

   public void run() {
      TableItem item = new TableItem(table,SWT.NONE);
      item.setText(text);
   }
}

/**
 * This class adds the console (an instance of MessengerView)
 * to the workbench in a separate thread.
 * Typically used to synchronize
 * with the UI thread.
 */
private static class ShowConsole
   implements Runnable {
   private IWorkbench workbench;
   private MessengerView console;
   public ShowConsole(IWorkbench workbench) {
      this.workbench = workbench;
   }
   public MessengerView getConsole() {
      return console;
   }
   public void run() {
      try {
         IWorkbenchWindow window =
                  workbench.getActiveWorkbenchWindow();
         if(window == null) {
            IWorkbenchWindow[] windows =
                  workbench.getWorkbenchWindows();
```

```
            if(windows != null && windows.length > 0)
               window = windows[0];
            else
               return;
         }
         IWorkbenchPage page = window.getActivePage();
         if(page != null)
            console = (MessengerView)page.showView(MESSENGER_ID);
      }
      catch(PartInitException x) {
      }
   }
}

/**
 * Creates a new MessengerView
 */
public MessengerView() {
   super();
   IPreferenceStore store =
             Activator.getDefault().getPreferenceStore();
   store.addPropertyChangeListener(new IPropertyChangeListener() {
      public void propertyChange(PropertyChangeEvent e) {
         setLevel((String)e.getNewValue());
      }
   });
   setLevel(store.getString(LEVEL_PREFERENCE_NAME));
}

/**
 * Set the new level property.
 * @param l new value
 */
public void setLevel(String l) {
   if(l.equals(LEVEL_INFO))
      level = INFO;
   else if(l.equals(LEVEL_WARNING))
      level = WARNING;
   else if(l.equals(LEVEL_ERROR))
      level = ERROR;
   else
```

```
      level = ALL;
}

/**
 * Eclipse API: creates the control for this part.
 */
public void createPartControl(Composite parent) {

   table = new Table(parent,SWT.SINGLE);
   TableColumn column = new TableColumn(table,SWT.NONE);
   column.setText(Resources.getString("eclipse.status"));
   column.setWidth(50);
   column = new TableColumn(table,SWT.NONE);
   column.setText(Resources.getString("eclipse.message"));
   column.setWidth(500);
   column = new TableColumn(table,SWT.NONE);
   column.setText(Resources.getString("eclipse.location"));
   column.setWidth(75);
   table.setHeaderVisible(true);
   table.addSelectionListener(new SelectionListener() {
      public void widgetDefaultSelected(SelectionEvent e) {
            updateTooltip();
      }

      public void widgetSelected(SelectionEvent e) {
            updateTooltip();
      }
   });
}

/**
 * Changes the table tooltip to the current message (the
 * content of the first column). It's useful to read long messages.
 */
protected void updateTooltip() {
   TableItem[] items = table.getSelection();
   if(items != null)
      table.setToolTipText(items[0].getText(1));
   else
      table.setToolTipText(null);
}
```

```
/**
 * Eclipse API: sets the focus on this part.
 */
public void setFocus() {
   if(table != null)
      table.setFocus();
}

/**
 * Adds a new item (a line) to the message table.
 * It takes care to synchronize
 * with the UI thread.
 * @param type first column (message type)
 * @param message second column (message itself)
 * @param location last column (message location)
 * @throws NotusException
 */
private void addItem(String type, String message, String location) {
   if(table != null) {
      Display display = Display.getCurrent();
      if(display == null)
         display = Display.getDefault();
      display.asyncExec(new AddItem(type, message, location));
   }
   else
      throw new NullPointerException(
           "Table control has not been initialized in MessengerView");
}

/**
 * Utility method: adds an exception to the window.
 * @param type describes the type of error (fatal, error)
 * @param x describes the problem
 */
private void addItem(String type, String msg) {
   if(table != null) {
      addItem(type, msg, null);
   }
}
```

```
public void error(String msg) {
   addItem(Resources.getString("error"), msg);
}

public void fatal(String msg) {
   addItem(Resources.getString("fatal"), msg);
}

public void warning(String msg) {
      if(level > WARNING)
         return;
   addItem(Resources.getString("warning"), msg);
}

public boolean progress(File sourceFile,File resultFile) {
      if(level > ALL)
         return true;
   if(resultFile != null)
      info(Resources.getString("progressdone"),
           new Object[] { sourceFile.getPath(),
                                resultFile.getPath() });
   else
      info(Resources.getString("progressignored"),
           new Object[] { sourceFile.getPath() });
   return true;
}

public void info(String msg) {
      if(level > INFO)
         return;
   if(table != null)
      addItem(Resources.getString("info"),msg,null);
}

public void info(String pattern, Object[] arguments) {
   info(MessageFormat.format(pattern,arguments));
}

/**
 * Clears the console.
 */
```

```
public void begin(String source,String target) {
   clean();
}

public void end() {
   addItem(null,Resources.getString("end"),null);
}

/**
 * Clears all messages displayed so far.
 */
public void clean() {
   if(table != null) {
      Display display = Display.getCurrent();
      if(display == null)
         display = Display.getDefault();
      display.syncExec(new Runnable() {
         public void run() {
            table.removeAll();
         }
      });
   }
}

/**
 * Helper method: it attempts to extract location information from
 * a Throwable (e.g. typecast to SAXParseException and similar)
 * @param t exception to process
 * @return location as string if available, null otherwise
private static String locationAsString(Throwable t) {
      if(t instanceof NotusException) {
         NotusException x = (NotusException)t;
         if(x.getException() != null)
            return locationAsString(x);
      }

   return null;
}
 */

/**
```

```
     * Helper method: extracts the filename from the system id.
     * @param systemID system id to extract the filename from
     * @return filename, if possible
     */
    private static String extractFilename(String systemID) {
          if(systemID == null)
             return "?";
          char[] separators = new char[] {
             File.separatorChar, '/', '\\',
          };
          for(int i = 0;i < separators.length;i++) {
             int pos = systemID.lastIndexOf(separators[i]);
             if(pos != -1)
                return systemID.substring(pos + 1);
          }
          return systemID == null ? "?" : systemID;
    }

    /**
     * Returns the Notus Console, i.e.
     * the shared instance of MessengerView that
     * is currently on screen. If none is on-screen,
     * it brings one up.
     * This method synchronizes with the UI thread.
     * @param workbench the workbench in use
     * @return shared Notus console
     */
    public static MessengerView showConsole(IWorkbench workbench) {
       Display display = Display.getCurrent();
       if(display == null)
          display = Display.getDefault();
       ShowConsole showConsole = new ShowConsole(workbench);
       display.syncExec(showConsole);
       return showConsole.getConsole();
    }
}
```

Essa classe cria uma janela (ou visão numa tradução direta do inglês *view*) no Eclipse que exibirá uma espécie de tabela com as mensagens geradas durante o processo de compilação. Um pouco mais complexa que as anteriores, possui classes aninhadas para lidar com questões de concorrência, *AddItem*

e *ShowConsole*. Como é perceptível, a primeira é utilizada para a inserção de itens na tabela de mensagens e a outra para exibir a própria janela de mensagens. O método *createPartControl* é uma exigência da classe abstrata *ViewPart* que *MessengerView* estende. Ele trata da criação dos componentes de uma janela, no caso ele cria uma tabela com as colunas adequadas para exibição das mensagens vindas do processo de compilação. O método *updateTooltip* é utilizado para exibir a *tooltip* (dica de ferramenta?) quando usuário selecionar algum item da tabela. A segunda coluna da seleção do usuário é pega e utiliza para exibir a dica. Observe que o método referido é utilizado dentro de *createPartControl* no momento da adição do *listener* de eventos de seleção (no final do procedimento). O método *setFocus* é outra exigência da classe abstrata e deve estar disponível para que o Eclipse possa chamá-lo quando for necessário enviar o foco para a janela – isso é necessário porque não há como o Eclipse saber para qual dos componentes da janela o foco deve ser enviado. O método privado *addItem* adiciona uma mensagem à tabela, já *error*, *fatal*, *warning* e ambos os *info* mostram mensagens passadas pelo usuário da classe, variando apenas o tipo de acordo com o método utilizado – é relevante ressaltar o uso do arquivo de strings com recursos (*resources*). Como informado por seu nome, o método *clean* remove todas as mensagens da tabela, deixando-a limpa. Finalmente, *showMessenger* é utilizado para exibir a janela de mensagens na tela.

Agora precisamos informar ao Eclipse sobre a extensão que queremos fazer, para isso é necessário alterar o arquivo **MANIFEST.MF**,

1. na guia **Extensions**, clique no botão **Add** e adicione *org.eclipse.ui.views*;
2. os *views* do Eclipse são separados por categorias (por exemplo no menu que os contém), dessa forma criaremos uma categoria para que possamos encaixar nossa janela de mensagens. Para isso,
   (a) clique com o botão direito do mouse sobre *org.eclipse.ui.view*, selecione **New** e depois **category**;
   (b) selecione o subitem category (que acabou de ser criado);
   (c) altere o campo **id** para notus.views.category – identificador único da categoria dentro do Eclipse;
   (d) altere o campo **name** para Notus – texto que será exibido para o usuário identificar a categoria.
3. agora informaremos ao Eclipse qual classe codifica a extensão. Para isso,

(a) clique com o botão direito do mouse sobre *org.eclipse.ui.view*, selecione **New** e depois **view**;

(b) selecione o subitem view (que acabou de ser criado);

(c) altere o campo **id** para notus.views.messenger – identificador único do view no Eclipse;

(d) altere o campo **name** para Notus Messenger – texto que será exibido para o usuário identificar o view;

(e) altere o campo **class** para notus.views.MessengerView – classe que codifica a extensão;

(f) altere o campo **category** para notus.views.category – relaciona o view a categoria criada anteriormente.

Para facilitar a compreensão sobre o que acabou de ser feito é recomendável que coloquemos o plug-in em funcionamento. Compilado e rodando, podemos encontrar a janela (view) criada

1. no menu **Window**;
2. opção **Show View**;
3. **Other**;
4. categoria **Notus**;
5. item **Notus Messenger**.

8.2 Compilando código

Com o plug-in capacitado para mostrar mensagens, passaremos ao que realmente importa, a compilação de código. Apesar de julgar bem enfatizado, não é demais lembrar mais uma vez que não incluiremos código do compilador Notus dentro do plug-in, ao invés disso invocaremos um compilador existente para a compilação.

O compilador em questão recebe código Notus e gera código Haskell [**?**]. Mais especificamente o código Notus é transformado em arquivos de entrada para o gerador de analisadores léxicos Alex [5] e para o gerador de analisadores sintáticos Happy [6]. Dessa forma precisamos submeter os arquivos gerados pelo compilador Notus ao Alex, ao Happy e a saída de ambos ao compilador Haskell, no nosso caso utilizamos o GHC [7].

Como dissemos acima, utilizaremos o Alex, o Happy e o GHC. Isso significa que será necessário instalar e colocá-los no *path* para que o Eclipse possa encontra-los.

Depois de fazê-lo, a seguir mostramos o código para compilação, porém antes é importante discutirmos o conceito de "Compilação incremental de projetos" (do original *Incremental project builders*). De forma simplificada, trata-se da compilação apenas daquilo que é necessário. Por exemplo, em Java, se o programador alterou apenas o funcionamento de um método não é necessário compilar todas as classes do projeto novamente, apenas àquela que teve o método alterado. Dessa forma se o compilador o suportar, uma compilação pode ser

- completa (*full*) – todo o conteúdo do projeto é compilado. Geralmente acontece quando o projeto nunca foi compilado;
- incremental (*incremental*) – utiliza o "último estado de compilação", mantido internamente pelo compilador, para realizar uma compilação otimizada baseada nas alterações realizadas desde a última compilação.

Infelizmente, o compilador Notus disponível não possui suporte a compilação incremental. Isso significa que qualquer alteração efetuada sob o código Notus demanda uma compilação completa de todo o projeto. Como o Eclipse suporta "gratuitamente" o recurso de compilação incremental através da codificação da classe abstrata *IncrementalProjectBuilder*, a compilação do projeto só ocorrerá quando houver alguma alteração em um dos arquivos que o compõe. Para a compilação de código, criamos a classe *NotusBuilder* que foi sugestivamente alojada no pacote *notus.builder*. Veja o código a seguir,

```
package notus.builder;

import java.io.*;
import java.util.*;
import org.eclipse.ui.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.core.runtime.*;
import org.eclipse.core.resources.*;
import org.eclipse.core.runtime.IProgressMonitor;

import notus.views.MessengerView;


/**
```

```
 * Implements the builder interface for Notus.
 * Required for build from menu Project.
 */
public class NotusBuilder
   extends IncrementalProjectBuilder {
   /**
    * Retains a reference to the workbench
    * that we can use throughout the build.
    */
   private IWorkbench workbench;

   /**
    * Progress bar showed during the compilation process
    */
   private IProgressMonitor monitor;

   /**
    * Creates a new NotusBuilder.
    */
   public NotusBuilder() {
      workbench = PlatformUI.getWorkbench();
   }

   /**
    * Builds the project.
    */
   protected IProject[] build(int kind,Map args,
                                IProgressMonitor monitor) {
       try {
             IProject project = getProject();
             if(project != null && project.isAccessible()) {
                if(monitor == null)
                   monitor = new NullProgressMonitor();

                saveDirtyEditors();

                MessengerView messenger =
                     MessengerView.showMessenger(workbench);

                this.monitor = monitor;
                monitor.beginTask("Compiling Notus files",100);
```

```
            IResource publish = runNotus(messenger,
                                         kind == FULL_BUILD);
            publish.refreshLocal(IResource.DEPTH_INFINITE, monitor);

         }
   }
   catch(Exception x) {
        System.err.println("---");
        x.printStackTrace();
        System.err.println("---");
   }
   return new IProject[0];
}

/**
 * Runs Notus. Simply launches the Notus product to publish the site.
 * Note that Notus does not rely on Eclipse to decide which file has
 * been modified, etc.
 * @param messenger messenger to report errors on
 * @param build if true, does a full build
 * @throws IOException I/O error, e.g. accessing files
 * @return the folder we published to
 * @throws NotusException Notus reports an error
 */
private IResource runNotus(MessengerView messenger, boolean build)
   throws CoreException, IOException {
   IProject project = getProject();
   String sourcePath = "src", outPath = "out";

   sourcePath = project.getFolder(sourcePath).
                              getLocation().toOSString();
   outPath = project.getFolder(outPath).
                              getLocation().toOSString();

   IFolder outFolder = project.getFolder(outPath);

   int stage = 0;
   final int NOTUS = 1, ALEX = 2, HAPPY = 3,
               COMPILER = 4, LINKER = 5;
```

```
try {

    messenger.clean();

    // notus
    stage = NOTUS;
    messenger.info("Compiling notus files...");
    String[] notus = {"java","notus.Notus",
           "\"" + sourcePath + "\"", "\"" + outPath + "\""};
    execute(notus, messenger);
    monitor.worked(40);

    // alex
    stage = ALEX;
    messenger.info("Creating lexer files...");
    String[] alex = {"alex", "-o",
               "\"" + outPath + "\\Lexer.hs" + "\"",
               "\"" + outPath + "\\Lexer.x" + "\""};
    execute(alex, null);
    monitor.worked(50);

    // happy
    stage = HAPPY;
    messenger.info("Creating parser files...");
    String[] happy = {"happy", "-o",
           "\"" + outPath + "\\Syntactic.hs" + "\"",
           "\"" + outPath + "\\Syntactic.y" + "\""};
    execute(happy, null);
    monitor.worked(60);

    // compiler
    stage = COMPILER;
    messenger.info("Compiling...");
    String[] comp = {"ghc", "-c", "NotusDefault.hs",
                     "DataModule.hs", "NotusFunctions.hs",
                     "Lexer.hs", "Syntactic.hs", "Main.hs"};
    execute(comp, outPath, null);
    monitor.worked(80);


    // linker
```

```
        stage = LINKER;
        messenger.info("Linking...");
        String[] link = {"ghc", "--make", "-o", "compiler", "Main"};
        //String[] comp = {"java","hello"};
        execute(link, outPath, null);
        monitor.worked(100);

        messenger.info("Compilation finished");

    }
    catch (Exception err) {
        if (messenger != null) {
            switch (stage) {
                case ALEX:
                    messenger.error("An error occurred during " +
                                    "creation of the lexer files");
                    break;
                case HAPPY:
                    messenger.error("An error occurred during " +
                                    "creation of the parser files");
                    break;
                case COMPILER:
                    messenger.error("An error occurred during " +
                                   "the compiling");
                    break;
                case LINKER:
                    messenger.error("An error occurred while " +
                                   "linking the files");
                    break;
            }
        }

        err.printStackTrace();
    }

    return outFolder;
}

private void execute(String[] pname, MessengerView messenger)
              throws IOException, InterruptedException, Exception {
    execute(pname, null, messenger);
```

```
}

private void execute(String[] pname, String dir,
          MessengerView messenger)
          throws IOException, InterruptedException, Exception {
    ProcessBuilder builder = new ProcessBuilder(pname);
    if (dir != null)
        builder.directory(new File(dir));

    //builder.start();
    final Process p = builder.start();

    // Out
    // Show messages?
    if (messenger != null) {
        String line;
        BufferedReader input =
                   new BufferedReader(
                   new InputStreamReader(p.getInputStream()));

        while ((line = input.readLine()) != null) {
            // Is there some text?
            if (line.compareTo("") != 0) {
                showMessage(line, messenger);
            }
        }
        input.close();

        input = new BufferedReader(
                      new InputStreamReader(p.getErrorStream()));

        while ((line = input.readLine()) != null) {
            if (line.compareTo("") != 0) {
                // Is there some text?
                showMessage(line, messenger);
            }
        }
        input.close();

    }
```

```
    // Wait the end of process
    p.waitFor();

    if (p.exitValue() != 0)
        throw new Exception();

    p.destroy();
}

private void showMessage(String line, MessengerView messenger) {
    int index;
    // Warning?
    index = line.indexOf("Warning");
    if (index >= 0) {
        index += "Warning:".length();

        line = line.substring(index);
        messenger.warning(line);

        return;
    }

    // Error?
    index = line.indexOf("Error");
    if (index >= 0) {
        index += "Error:".length();

        line = line.substring(index);
        messenger.error(line);

        return;
    }

    messenger.info(line);
}

/**
 * Saves dirty editors (i.e. those editors that were modified but
 * not saved) in the workbench. It takes care to run the saving process
 * in the UI thread.
 */
```

```
   private void saveDirtyEditors() {
      Display display = Display.getCurrent();
      if(display == null)
         display = Display.getDefault();
      display.syncExec(new Runnable() {
         public void run() {
            IWorkbenchWindow[] windows = workbench.getWorkbenchWindows();
            for(int i = 0;i < windows.length;i++) {
               IWorkbenchPage[] pages = windows[i].getPages();
               for(int j = 0;j < pages.length;j++)
                   pages[j].saveAllEditors(false);
            }
         }
      });
   }
}
```

Nessa classe o construtor é utilizado para obter o *workbench*, isto é, o objeto que representa o ambiente de trabalho. O método *build* obtém as informações necessárias para a compilação, define o monitor – janela de diálogo que indica o progresso da tarefa de compilação – e marca o início da tarefa e a seguir invoca o método que de fato realiza o processo de compilação, *runNotus*. A função *runNotus* invoca os programas externos responsáveis pela compilação, como pode ser visto e já dito, inicialmente o compilador Notus, a seguir o Alex, o Happy, o GHC no modo de compilação (sem ligação) e finalmente, o GHC em modo de ligação. O método *execute* utilizado dentro de *runNotus* executa um programa externo lhe passado como parâmetro, podendo possivelmente, mostrar mensagens geradas por essas execuções. No caso do plug-in, como espera-se que o compilador Notus e os demais programas funcionem corretamente apenas as mensagens geradas pelo compilador Notus são exibidas. Fato que justifica o parâmetro *null* nas chamadas de *execute* que não envolvem o compilador Notus. O procedimento *showMessage* é utilizado para formatar as mensagens que serão exibidas pelo plug-in. Isso foi necessário porque a versão utilizada do compilador Notus não exibe mensagens de forma padronizada.

Finalmente, nosso esforço não terá servido de nada se não informarmos ao Eclipse que temos uma classe que serve para compilar conteúdo Notus. Para fazê-lo,

1. na guia **Extensions** do arquivo MANIFEST.MF adicione a extensão *org.eclipse.core.resources.builders*;

2. altere a extensão *org.eclipse.core.resources.builders*,
   (a) ID: notus.builder – identificador para o *builder* ante o Eclipse;
   (b) Name: Notus Builder – texto exibido pelo Eclipse para identificar o *builder* na interface gráfica;
3. clique com o botão direito do mouse sobre a extensão criada, selecione a opção **New** e a seguir **builder**;
4. altere a opção **hasNature** do item builder para **true** – nosso projeto possui uma natureza (criada anteriormente);
5. clique com o botão direito do mouse sobre o item **builder**, selecione **New** e depois **run**;
6. altere a opção **class** do item **run** para *notus.builder.NotusBuilder* – classe que codifica o *builder* do plug-in.

Nesse ponto, "onde fizemos a ligação do builder com a natureza do projeto?", seria uma boa pergunta. Quando criamos a natureza (veja a classe *NotusProjectNature*) fizemos prematuramente a ligação entre o *builder* e a natureza dentro do método *configure*. Mais especificamente, na linha

```
command.setBuilderName(BUILDER_ID);
```

9 Executando um Programa Compilado

Criamos projetos, formatamos e a seguir compilamos programas Notus, agora para terminarmos nossa caminhada executaremos os programas compilados. Como a execução de um programa pode precisar de entrada de dados, além é claro das esperadas saídas, precisaremos de uma espécie de console (*shell*) que atenda a ambos os requisitos. O Eclipse já fornece um console que se alinha a nossas necessidades, dessa forma precisaremos apenas integrá-lo ao nosso projeto.

Para executar os programas Notus teremos duas preocupações, a classe que de fato executa os programas, além da classe que criará a tela de execução (*run*) característica do Eclipse.

A primeira coisa que faremos será adicionar algumas constantes as já existentes,

```
/**
* namespace URI for the properties
*/
public static final String PROPERTY_NAMESPACE = "dcc.ufmg.br";

/**
* property qualified name for the out directory
*/
public static final QualifiedName OUT_PROPERTY_NAME =
   new QualifiedName(PROPERTY_NAMESPACE,DEFAULT_OUT_DIR);

public static final String NOTUS_LAUNCH_CONFIGURATION_TYPE =
   "notus.launch.NotusLaunchConfigurationType";

public static final String NOTUS_LAUNCH_FILES_TO_CONSULT =
   "notus.launch.NotusLaunchCommands";

public static final String NOTUS_LAUNCH_WORKING_DIRECTORY =
   "notus.launch.NotusLaunchWorkingDirectory";

public static final String NOTUS_LAUNCH_PROJECT =
   "notus.launch.NotusLaunchProject";

public static final String NOTUS_LAUNCH_STRING_FILES   =
   "notus.launch.NotusLaunchFiles";
```

e a seguir adicionar algumas dependências que serão necessárias para o projeto,

1. org.eclipse.debug.core;
2. org.eclipse.debug.ui;
3. org.eclipse.ui.console.

Depois disso criaremos um pacote para abrigar as classes responsáveis pela execução dos programas, demos a ele o nome sugestivo de *notus.launch*.

Ao pacote recém criado, adicionamos a classe *NotusConsoleLaunchTab* que codifica a tela do Eclipse responsável pela funcionalidade de execução,

```
package notus.launch;
```

```
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;
import org.eclipse.debug.core.ILaunchConfiguration;
import org.eclipse.debug.core.ILaunchConfigurationWorkingCopy;
import org.eclipse.debug.ui.AbstractLaunchConfigurationTab;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Text;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.graphics.Font;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.events.ModifyEvent;
import org.eclipse.swt.events.ModifyListener;

import org.eclipse.core.resources.ResourcesPlugin;
import org.eclipse.core.resources.IProjectDescription;
import org.eclipse.core.resources.IResource;
import org.eclipse.core.resources.IProject;
import org.eclipse.core.resources.IFile;
import org.eclipse.ui.PlatformUI;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.ui.IWorkbenchPage;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IPath;
import org.eclipse.ui.IEditorInput;
import org.eclipse.ui.IEditorPart;
import notus.Activator;
import notus.PluginConstants;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.StringTokenizer;

/**
 *
 * Tab-page to the Notus launch configuration.
 *
 */
```

```
public class NotusConsoleLaunchTab extends
       AbstractLaunchConfigurationTab implements PluginConstants {
    // Project UI widgets.
    private Label fProjLabel;
    private Text fProjText;

    // Files to consult UI widgets.
    private Label fFilesLabel;
    private Text fFilesText;

    private static final String EMPTY_STRING = ""; //$NON-NLS-1$


    /* (non-Javadoc)
     * @see org.eclipse.debug.ui.ILaunchConfigurationTab
     *       #createControl(org.eclipse.swt.widgets.Composite)
     */
    public void createControl(Composite parent) {
        Font font = parent.getFont();

        Composite comp = new Composite(parent, SWT.NONE);
        setControl(comp);
        GridLayout topLayout = new GridLayout();
        comp.setLayout(topLayout);
        GridData gd;

        createVerticalSpacer(comp);

        Composite projComp = new Composite(comp, SWT.NONE);
        GridLayout projLayout = new GridLayout();
        projLayout.numColumns = 3;
        projLayout.marginHeight = 0;
        projLayout.marginWidth = 0;
        projComp.setLayout(projLayout);
        gd = new GridData(GridData.FILL_HORIZONTAL);
        projComp.setLayoutData(gd);
        projComp.setFont(font);

        fProjLabel = new Label(projComp, SWT.NONE);
        fProjLabel.setText("Project: ");
        gd = new GridData();
```

```
        gd.horizontalSpan = 3;
        fProjLabel.setLayoutData(gd);
        fProjLabel.setFont(font);

        fProjText = new Text(projComp, SWT.SINGLE | SWT.BORDER);
        gd = new GridData(GridData.FILL_HORIZONTAL);
        gd.horizontalSpan = 2;
        fProjText.setLayoutData(gd);
        fProjText.setFont(font);
        this.fProjText.addModifyListener(new ModifyListener() {
            public void modifyText(ModifyEvent evt) {
                updateLaunchConfigurationDialog();
            }
        });


        Label spacer = createVerticalSpacer(projComp);
        gd = new GridData();
        gd.horizontalSpan = 3;
        spacer.setLayoutData(gd);

        fFilesLabel = new Label(projComp, SWT.NONE);
        fFilesLabel.setText("Files to consult:");
        gd = new GridData();
        gd.horizontalSpan = 3;
        fFilesLabel.setLayoutData(gd);
        fFilesLabel.setFont(font);

        fFilesText = new Text(projComp, SWT.SINGLE | SWT.BORDER);
        gd = new GridData(GridData.FILL_HORIZONTAL);
        gd.horizontalSpan = 2;
        fFilesText.setLayoutData(gd);
        fFilesText.setFont(font);
        this.fFilesText.addModifyListener(new ModifyListener() {
            public void modifyText(ModifyEvent evt) {
                updateLaunchConfigurationDialog();
            }
        });

    }
```

```
/**
 * Creates some empty space.
 */
private Label createVerticalSpacer(Composite comp) {
    return new Label(comp, SWT.NONE);
}


/* (non-Javadoc)
 * @see org.eclipse.debug.ui.ILaunchConfigurationTab
 * #setDefaults(org.eclipse.debug.core.ILaunchConfigurationWorkingCopy)
 */
public void setDefaults(ILaunchConfigurationWorkingCopy config) {
    IProject project = getSelectedProject(config);
    initializeProject(project, config);
    initializeFiles(project, config);
}

/**
 * Sets the 'project's name and its location as 'config'´s attributes.
 *
 * @param project
 * @param config
 */
private void initializeProject(IProject project,
          ILaunchConfigurationWorkingCopy config) {
    String projectName = EMPTY_STRING;
    if (project != null) {
        projectName = project.getName();
        setWorkingDirectory(project, config);
    }
    config.setAttribute(NOTUS_LAUNCH_PROJECT, projectName);
}

/**
 * Creates a string with all the names of Notus files in 'project' and
 * sets it as a 'config's attribute.
 *
 * @param project
 * @param config
 */
```

```
private void initializeFiles(IProject project,
         ILaunchConfigurationWorkingCopy config) {
    String fieldFiles = EMPTY_STRING;
    // final int NOTUS_EXTENSIONS_LENGTH = 3;
    ArrayList files = new ArrayList();
    if (project != null) {
        try {
            IResource[] resources = project.members();
            for (int i=0; i<resources.length; i++) {
                IResource r = resources[i];
                if (r instanceof IFile) {
                    IFile f = (IFile) r;
                    if (f.getFileExtension().equals("nts")) {
                        files.add("nts");
                    }
                }
            }
            fieldFiles = buildStringFiles(files);
        }
        catch (CoreException ce) {
            // Log the error to the Eclipse log.
            IStatus status = new Status(IStatus.ERROR,
                    Activator.getDefault().getBundle().
                                        getSymbolicName(),0,
               "Error in NotusConsoleLaunchTab.initializeFiles: " +
               ce.getMessage(), ce);
            //NotusPlugin.log(status);
        }
    }
    config.setAttribute(NOTUS_LAUNCH_STRING_FILES, fieldFiles);
}

/**
 * Builds a comma delimited String with de filenames in 'files'.
 *
 * @param files
 */
private String buildStringFiles(ArrayList files) {
    String field = EMPTY_STRING;
    Iterator it = files.iterator();
    while (it.hasNext()) {
```

```
        Object obj = it.next();
        if (obj instanceof IFile) {
            IFile file = (IFile) obj;
            field += "'" + file.getName() + "'";
            if (it.hasNext()) {
                field += ", ";
            }
        }
    }
    return field;
}

/**
 * Gets the current project selected in the workbench.
 *
 * @param config
 * @return the current project selected in the workbench.
 */
private IProject getSelectedProject(
               ILaunchConfigurationWorkingCopy config) {
    IWorkbenchWindow window =
           PlatformUI.getWorkbench().getActiveWorkbenchWindow();
    if (window != null) {
        IWorkbenchPage page = window.getActivePage();
        if (page != null) {
            ISelection selection = page.getSelection();
            if (selection instanceof IStructuredSelection) {
                IStructuredSelection ss =
                   (IStructuredSelection)selection;
                if (!ss.isEmpty()) {
                    Object obj = ss.getFirstElement();
                    if (obj instanceof IResource) {
                        IResource i = (IResource) obj;
                        IProject pro = i.getProject();
                        return pro;
                    }
                }
            }
            // If the editor has the focus...
            IEditorPart part = page.getActiveEditor();
            if (part != null) {
```

```
                IEditorInput input = part.getEditorInput();
                IFile file = (IFile) input.getAdapter(IFile.class);
                return file.getProject();
            }
        }
    }
    return null;
}

/* (non-Javadoc)
 * @see org.eclipse.debug.ui.ILaunchConfigurationTab#initializeFrom
 * (org.eclipse.debug.core.ILaunchConfiguration)
 */
public void initializeFrom(ILaunchConfiguration config) {
    updateProjectFromConfig(config);
    updateFilesFromConfig(config);
}

/**
 * Updates the project name field, acording to a launch
 * configuration 'config' attribute.
 *
 * @param config
 */
private void updateProjectFromConfig(ILaunchConfiguration config) {
    try {
        String projectName = config.getAttribute(
                            NOTUS_LAUNCH_PROJECT, EMPTY_STRING);
        fProjText.setText(projectName);
    } catch (CoreException ce) {
        // Log the error to the Eclipse log.
        IStatus status = new Status(IStatus.ERROR,
                Activator.getDefault().getBundle().
                                    getSymbolicName(),0,
           "Error in NotusConsoleLaunchTab.updateProjectFromConfig: " +
           ce.getMessage(), ce);
        // NotusPlugin.log(status);
    }
}
```

```
/**
 * Updates the 'Files to consult' field, according to a launch
 * configuration 'config' attribute.
 *
 * @param config
 */
private void updateFilesFromConfig(ILaunchConfiguration config) {
    try {
        String files = config.getAttribute(
                         NOTUS_LAUNCH_STRING_FILES, EMPTY_STRING);
        fFilesText.setText(files);
    } catch (CoreException ce) {
        // Log the error to the Eclipse log.
        IStatus status = new Status(IStatus.ERROR,
                Activator.getDefault().getBundle().getSymbolicName(),0,
           "Error in NotusConsoleLaunchTab.updateFilesFromConfig: " +
           ce.getMessage(), ce);
        // NotusPlugin.log(status);
    }
}


/* (non-Javadoc)
 * @see org.eclipse.debug.ui.ILaunchConfigurationTab#performApply
 * (org.eclipse.debug.core.ILaunchConfigurationWorkingCopy)
 */
public void performApply(ILaunchConfigurationWorkingCopy config) {
    config.setAttribute(NOTUS_LAUNCH_PROJECT, fProjText.getText());
    config.setAttribute(NOTUS_LAUNCH_STRING_FILES, fFilesText.getText());

    // TODO: Verificar se projeto e arquivos contidos nos campos da tab
    //       page ainda são válidos.

    setRunAttributes(config);

}

/**
 * Sets the launch configuration <code>config</code>
 * attributes used to run
 * the Notus console, according to other attributes
```

```
 * already defined.
 *
 * @param config
 */
private void setRunAttributes(ILaunchConfigurationWorkingCopy config) {
    try {
        String projectName =
                    config.getAttribute(NOTUS_LAUNCH_PROJECT,
                    EMPTY_STRING);

        if (projectName != EMPTY_STRING) {
            IProjectDescription p =
                 ResourcesPlugin.getWorkspace().newProjectDescription(
                 projectName);
            // IPath projectPath = p.getLocation();
            setFilesToConsult(config);
        }

    }
    catch (CoreException ce) {
        // Log the error to the Eclipse log.
        IStatus status = new Status(IStatus.ERROR,
                Activator.getDefault().getBundle().getSymbolicName(),0,
           "Error in NotusConsoleLaunchTab.setRunAttributes: " +
           ce.getMessage(), ce);
        // NotusPlugin.log(status);
    }
}

/**
 * Sets the 'INotusLaunchConstants.NOTUS_LAUNCH_FILES_TO_CONSULT'
 * <code>config</code>'s
 * attribute, used to consult Notus files into Notus console.
 * This attribute is defined according to the 'Files to consult'
 * field, whose
 * contents are stored in the
 * 'INotusLaunchConstants.NOTUS_LAUNCH_STRING_FILES'
 * 'config's attribute.
 *
 * @param config
 */
```

```
private void setFilesToConsult(ILaunchConfigurationWorkingCopy config) {
    try {
        ArrayList files = new ArrayList();
        String notusFiles = config.getAttribute(
                        NOTUS_LAUNCH_STRING_FILES, EMPTY_STRING);
        notusFiles = notusFiles.trim();

        StringTokenizer filesTokenizer =
                        new StringTokenizer(notusFiles,",");

        while (filesTokenizer.hasMoreTokens()) {
            String notusFile = filesTokenizer.nextToken().trim();
            if (notusFile.startsWith("'") &&
                notusFile.endsWith("'") &&
                notusFile.length() > 1) {
                notusFile = notusFile.substring(1,notusFile.length()-1);
            }
            files.add(notusFile);
        }
        config.setAttribute(NOTUS_LAUNCH_FILES_TO_CONSULT, files);
    }
    catch (CoreException ce) {
        // Log the error to the Eclipse log.
        IStatus status = new Status(IStatus.ERROR,
                Activator.getDefault().getBundle().getSymbolicName(),0,
           "Error in NotusConsoleLaunchTab.setFilesToConsult: " +
           ce.getMessage(), ce);
        // NotusPlugin.log(status);
    }
}


/**
 * Set the 'INotusLaunchConstants.NOTUS_LAUNCH_WORKING_DIRECTORY'
 * 'config's attribute as the 'project' location.
 *
 * @param project
 * @param config
 */
private void setWorkingDirectory(IProject project,
                ILaunchConfigurationWorkingCopy config) {
```

```
        String workingDirectory = project.getLocation().toString();
        if (workingDirectory != "" && workingDirectory != null) {
            workingDirectory += IPath.SEPARATOR;
        }
        config.setAttribute(NOTUS_LAUNCH_WORKING_DIRECTORY,
                                                workingDirectory);
    }


    /* (non-Javadoc)
     * @see org.eclipse.debug.ui.ILaunchConfigurationTab#getName()
     */
    public String getName() {
        return "Main";
    }

}
```

Criada a classe que codifica a interface gráfica para execução agora apresentamos a classe que de fato executa o programa compilado, ela naturalmente ela foi criada no mesmo pacote da anterior,

```
package notus.launch;

import org.eclipse.core.resources.*;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.NullProgressMonitor;
import org.eclipse.core.runtime.Plugin;

import org.eclipse.debug.core.ILaunch;
import org.eclipse.debug.core.ILaunchConfiguration;
import org.eclipse.debug.core.model.IProcess;
import org.eclipse.debug.core.model.ILaunchConfigurationDelegate;
import org.eclipse.debug.core.model.RuntimeProcess;
import org.eclipse.ui.IWorkbench;
import org.eclipse.ui.PlatformUI;
import org.eclipse.ui.console.ConsolePlugin;
import org.eclipse.ui.console.IConsole;
import org.eclipse.ui.console.IConsoleManager;
import org.eclipse.ui.console.IOConsole;
```

```
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;

import notus.Activator;
import notus.PluginConstants;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.util.List;
import java.util.ArrayList;

public class NotusLaunchConfigurationDelegate
       implements ILaunchConfigurationDelegate, PluginConstants {

    /**
     * Constructor
     *
     */
    public NotusLaunchConfigurationDelegate() {

    }

    /**
     *   This method actually launches the Notus.
     *
     * @param configuration the launch configuration.
     * @param mode The launch mode.
     * @param launch The launch event.
     * @param monitor The progress monitor.
     * @throws CoreException if the launch fails.
     */
    public final void launch(
        final ILaunchConfiguration configuration,
        final String mode,
        final ILaunch launch,
        final IProgressMonitor monitor)
        throws CoreException {
```

```
if (monitor.isCanceled())
    return;

try {
    String projectName =
        configuration.getAttribute(NOTUS_LAUNCH_PROJECT,
        "");

    if (projectName.compareTo("") == 0)
        return;
    IProject project = ResourcesPlugin.getWorkspace().
                            getRoot().getProject(projectName);
    if (project == null)
        return;

    String outPath = project.getPersistentProperty(
                            PluginConstants.OUT_PROPERTY_NAME);
    if (outPath == null)
          outPath = "out";
    outPath = project.getFolder(outPath).getLocation().toOSString();

    // Get file list from LaunchConfiguration.
    List files =
        configuration.getAttribute(
            NOTUS_LAUNCH_FILES_TO_CONSULT,
            new ArrayList());

    // fNotusRuntime = "compiler.exe";
    String[] fNotusRuntime = new String[files.size() + 1];
    fNotusRuntime[0] = outPath + "\\compiler";
    int j = 1;

    for (Object i : files) {
        fNotusRuntime[j++] = (String)i;
    }

    try {
        ProcessBuilder builder = new ProcessBuilder(fNotusRuntime);
        if (outPath != null)
```

```
                    builder.directory(new File(outPath));

                RuntimeProcess rProc = new RuntimeProcess(
                    launch, builder.start(), fNotusRuntime[0], null);
                launch.addProcess(rProc);

            }
            catch (IOException err) {
                return;
            }
              catch (Exception err) {
                  err.printStackTrace();
              }
        }
        catch (CoreException e) {
            // Log the error to the Eclipse log.
            IStatus status = new Status(
                   IStatus.ERROR,
                   Activator.getDefault().getBundle().getSymbolicName(),
                   0,"Problems executing notus.\n ", e);
            throw new CoreException(status);
        }
    }

}
```

Finalmente adicionaremos a classe que será a responsável pela criação da *NotusConcoleLaunchTab* sempre que o usuário requisitar, isto é, sempre que uma nova configuração de execução (*launch*) for solicitada,

```
package notus.launch;

import org.eclipse.debug.ui.AbstractLaunchConfigurationTabGroup;
import org.eclipse.debug.ui.ILaunchConfigurationDialog;
import org.eclipse.debug.ui.ILaunchConfigurationTab;
import org.eclipse.debug.ui.CommonTab;

public class NotusLaunchTabGroup
    extends AbstractLaunchConfigurationTabGroup {

    /* (non-Javadoc)
```

```
     * @see org.eclipse.debug.ui.ILaunchConfigurationTabGroup#createTabs
     * (org.eclipse.debug.ui.ILaunchConfigurationDialog, java.lang.String)
     */
    public void createTabs(ILaunchConfigurationDialog arg0, String arg1) {
        ILaunchConfigurationTab[] tabs = new ILaunchConfigurationTab[] {
            new NotusConsoleLaunchTab(),
            new CommonTab()
        };
        setTabs(tabs);
    }

}
```

Referências

[1] Oetiker, T., H., Partl, I., Hyna e E., Schlegl [1999]. "Uma não tão pequena introdução ao LaTeX". Pergamon Press, New York, 329-345.

[2] http://www.tiobe.com/, "TIOBE Programming Community Index".

[3] Tirelo, F. e Bigonha, R. S. [2006]. "Notus".

[4] http://www.eclipse.org, "Eclipse - an open development platform".

[5] http://www.haskell.org/alex/, "Alex: A lexical analyser generator for Haskell".

[6] http://www.haskell.org/happy/, "Happy: The Parser Generator for Haskell".

[7] http://www.haskell.org/ghc/, "The Glasgow Haskell Compiler".

[8] http://www.osgi.org/, "OSGi Alliance".

[9] http://help.eclipse.org/, "Eclipse documentation".