Além do Eclipse
-
Contribuindo com seus próprios Plugins

Alexandre Freire [alex@arca.ime.usp.br]

IME/USP

Objetivos

Após este curso você vai saber:

- Descrever a arquitetura extensível do Eclipse
- Descrever alguns pontos de extensão do Eclipse
- Descrever as possíveis relações entre plugins do Eclipse
- Desenvolver, depurar e testar plugins usando o PDE
- Desenvolver UIs usando as bibliotecas SWT e JFace
- Achar exemplos de funcionalidades no próprio Eclipse
- Listar as boas práticas de desenvolvimento de plugins
- Publicar seus plugins
- Prover ajuda para seus plugins

Oque é o Eclipse afinal?

Um IDE comum para todos (ou para o programador esquizofrênico...)

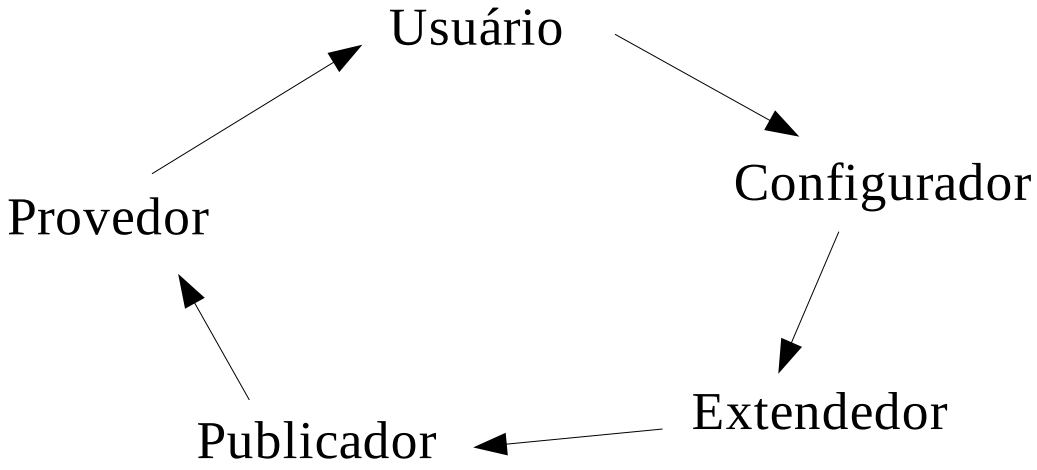
- O Eclipse não é uma IDE propriamente dita, é um arcabouço para desenvolvimento de ferramentas
- Estensível, aberto e portável. Através de *plugins*, diversas ferramentas podem ser combinadas criando um ambiente de desenvolvimento integrado
- Uma mesma plataforma para vários papéis de desenvolvimento: programador de componentes, integrador, responsável por testes, web designer...
  - O time inteiro usa a mesma aplicação
- Suporte para desenvolvimento de novas ferramentas
- O próprio Eclipse é desenvolvido usando o Eclipse

Arquitetura extensível

- Um plugin é a menor unidade extensível presente no Eclipse
  - pode conter código, recursos, ou ambos
- O próprio Eclipse é composto de diversos plugins
  - mais de 100 plugins
- Pontos de extensão
  - mecanismo que permite que um plugin adcione funcionalidade a outros plugins
- Interações de usuário são padronizadas
  - plugins de diferentes vendedores são integrados
  - reutilização de componentes – “IDE patterns”

Ciclo de contribuição

- O Eclipse também é uma comunidade de desenvolvedores
- A grande maioria dos usuários são programadores
  - Inclui um ambiente de aprendizado, uma arquitetura bem definida, e muitos exemplos e documentação
  - Aprender a usar as ferramentas de busca para ler código fonte do próprio Eclipse e de outros plugins é fundamental.

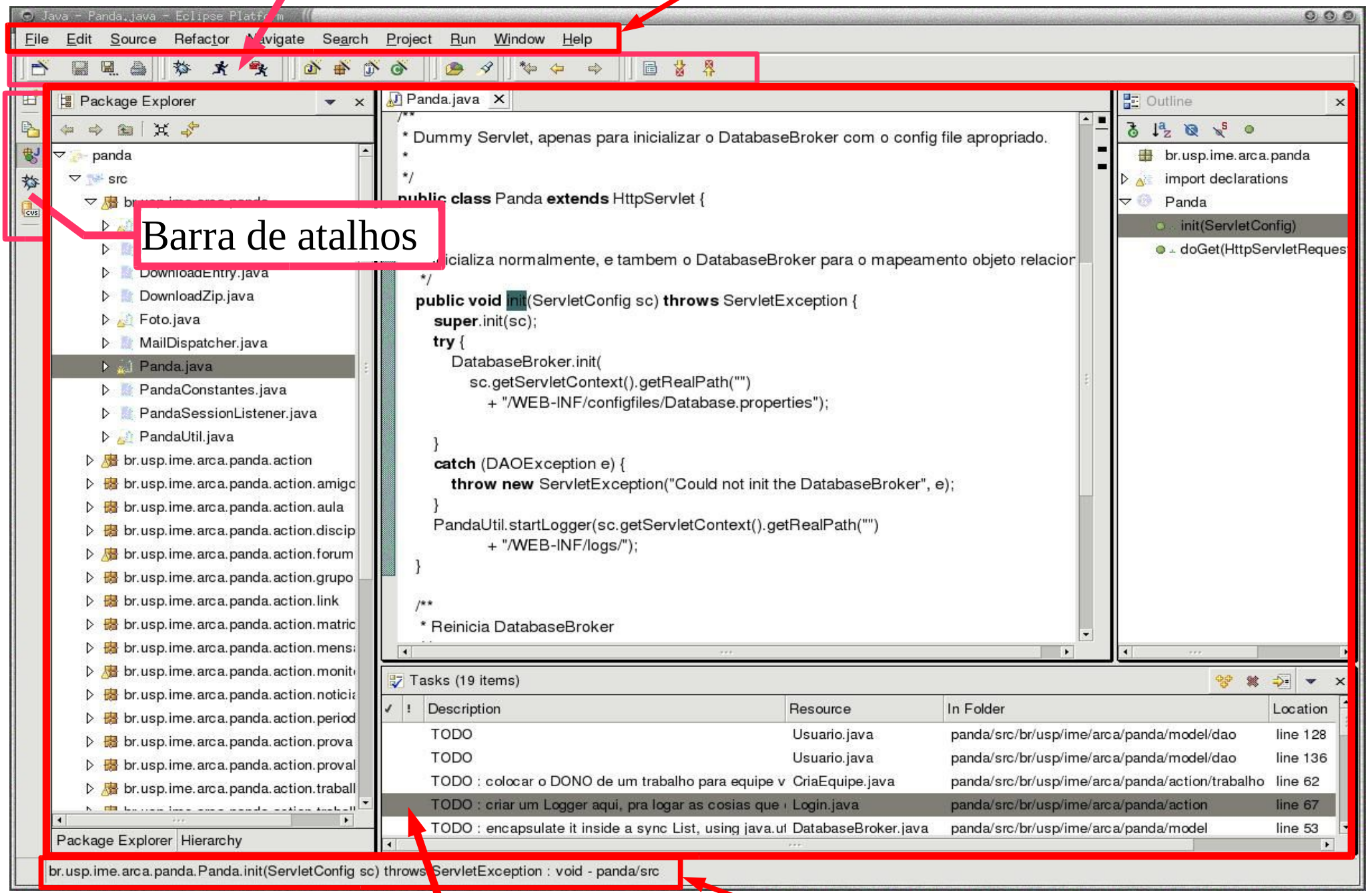


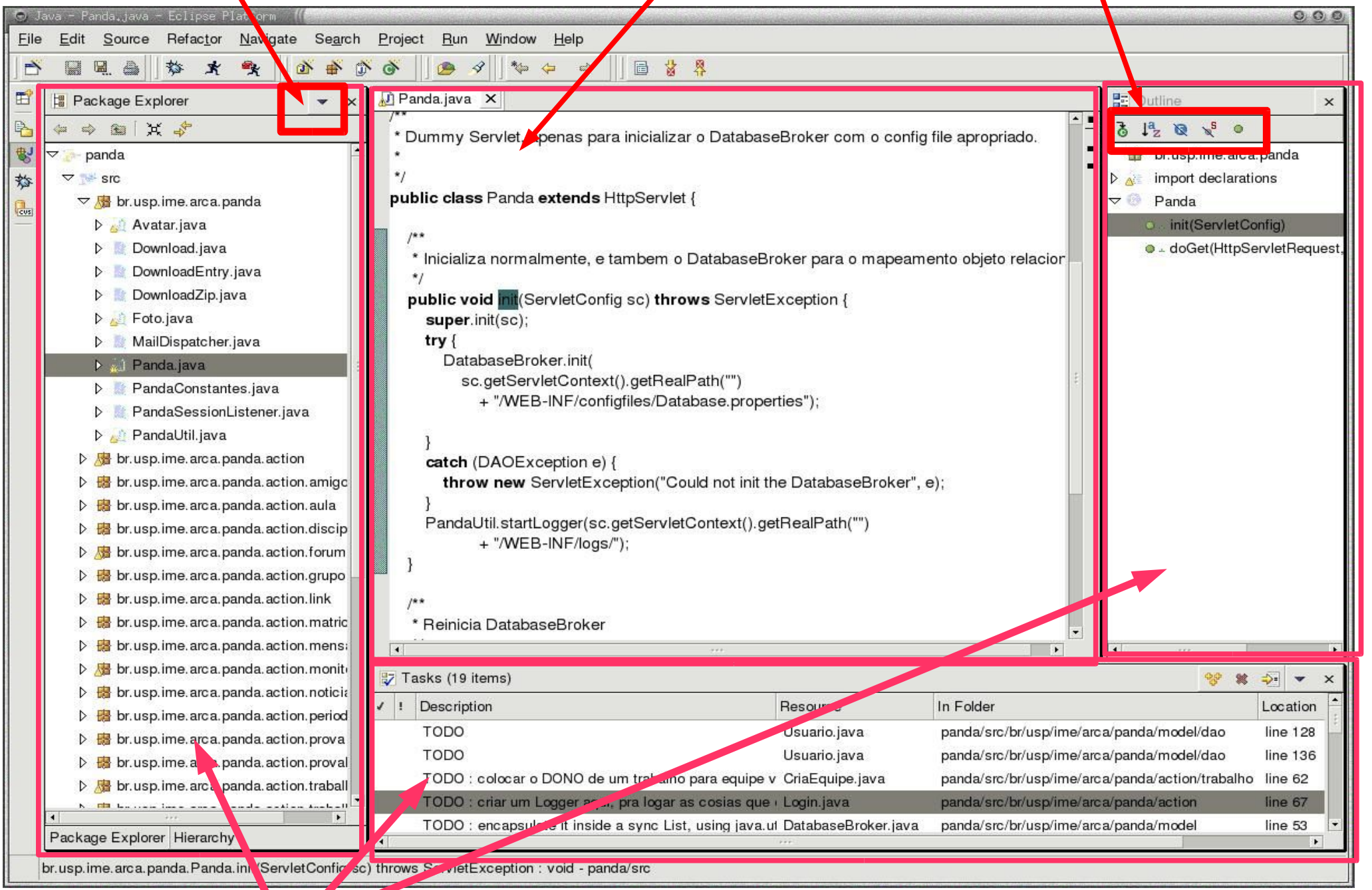
Papéis na comunidade

- Usuário
  - normalmente são programadores, usam, reportam bugs
- Configurador
  - Customizam sua experiência com o Eclipse, mudanças limitadas pela visão dos desenvolvedores
- Extendedor
  - Cria mudanças plugando novas funcionalidades
- Publicador
  - Cria pacotes com suas extensões e disponibiliza para a comunidade

Papéis na comunidade, cont.

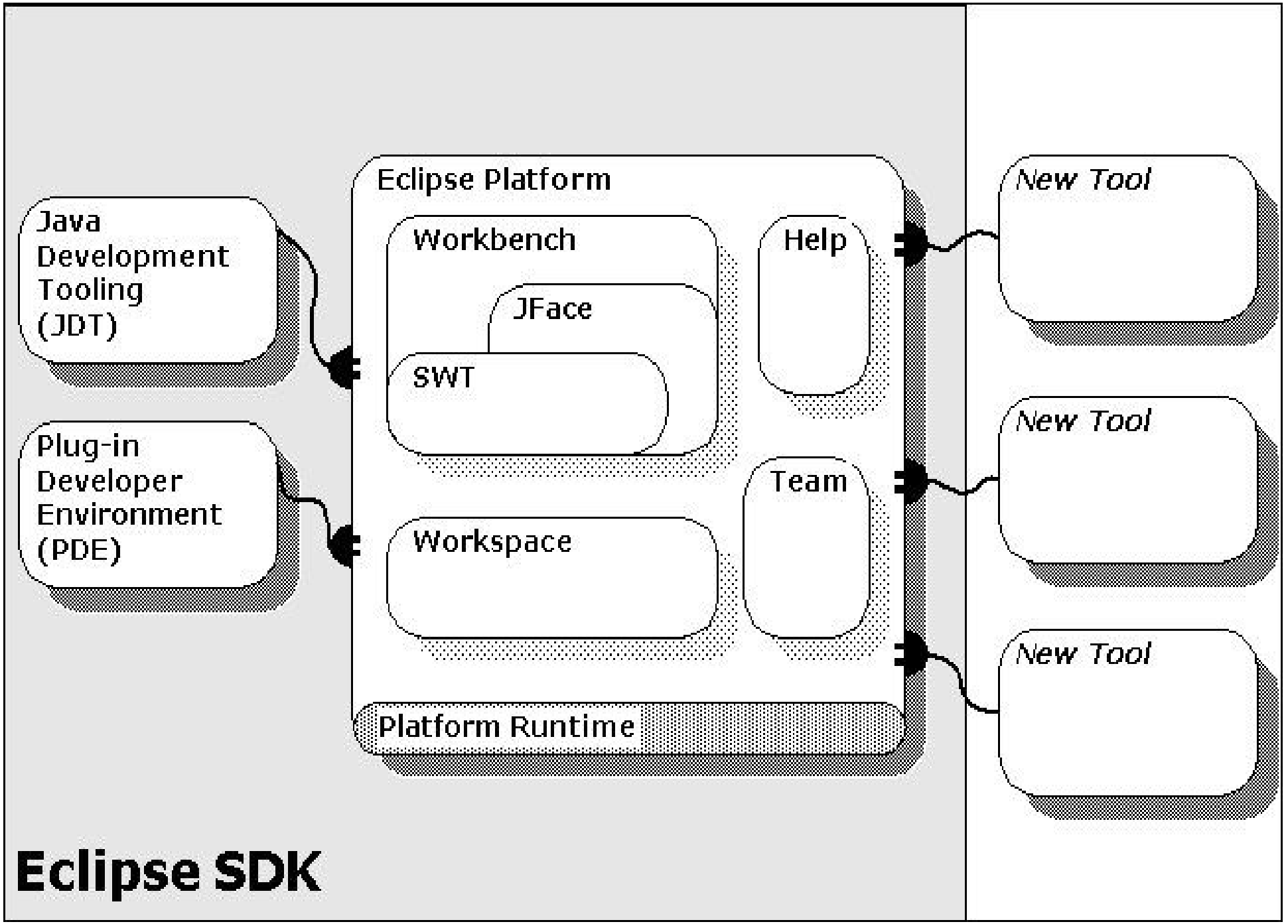
- Provedor
  - Prove pontos de extensão para que outros possam adcionar funcionalidades que não foram previstass à sua contribuição
- Commiter
  - O Eclipse é um projeto aberto, se existem mudanças que você gostaria de fazer e que não podem ser feitas usando plugins você pode mudar o próprio código.





A bancada de trabalho

- O Eclipse usa a abstração de bancada de trabalho para prover uma interface comun para seus usuários
- A bancada contém menus, barras de ferramentas, editores e visualizadores, todos contidos em uma janela
- Estes componentes são extensíveis
- A bancada pode ter várias perspectivas, cada perspectiva organiza diferentes editores e visualizadores e contribui com menus e barras de ferramentas, determinando a aparência da bancada
- Visualizadores oferecem diversas informações sobre os recursos de um projeto. Podem apresentar somente partes ou atributos internos de um recurso ou podem ser ligadas a um editor.


Eclipse Platform
Java Development Tooling (JDT)
Plug-in Developer Environment (PDE)
Workbench
JFace
SWT
Help
Team
Workspace
Platform Runtime
New Tool
New Tool
New Tool
Eclipse SDK

Plataforma de runtime

- Única parte do Eclipse que não é um plugin
- Responsável por gerenciar os plugins
- Cuida da inicialização do Eclipse, descobre os plugins instalados dinâmicamente e mantêm as informações em um registro.
- Define o modelo de plugins, e os pontos de extensão
- Liga extensões nos seus devidos pontos de extensão
- No Eclipse, tudo é uma contribuição
  - JDT é composto por mais de 60 plugins
  - WSAD contêm mais de 500 plugins

Workspace

- Gerência os recursos ligados a projetos do usuário
- Mantêm todas informações em um diretório comun
- Cada projeto é mapeado para um subdiretório deste diretório
- Define API para criar e gerenciar recursos:
  - Projetos, Pastas, Arquivos
- Mecanismo de notificação de mudanças
  - Mantêm histórico de mudanças
- Define os “Builders” e “Natures”, que podem ser usados para transformar recursos
- Cuida de “bookmarks”, “error markers” e “task items”

Workbench e UI

- Define interface comun para usuário lidar com recursos e ferramentas.
  - “Editor Part”, “View Part”, e “Perspective”
  - Cuida da navegação, layout e ativação destes componentes
- Define pontos de extensão para adcionar, por exemplo, visualizadores ou ações no menu.
- Contém toolkits para implementação de interfaces SWT e JFace
  - Standard Widget Toolkit, conjunto de widgets gráficos de baixo nível, genéricos e portáveis, construido com controles nativos do SO
  - Um arcabouço baseado em modelos, contêm componentes de alto nível

Eclipse UI guidelines

- http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html
- Contêm +- 125 regras que você deve seguir ao criar suas interfaces
- Exemplos:
  - If a programming error occurs in the product, communicate the occurrence with a dialog, and log it.
  - An action should only be enabled if it can be completed successfully.
  - Use a wizard for any task consisting of many steps, which must be completed in a specific order.
  - Let the user control the visible action set. Don't try to control it for them.
  - Start out with a single preference page. Then evolve to more if you need to.

Mais estrutura da plataforma

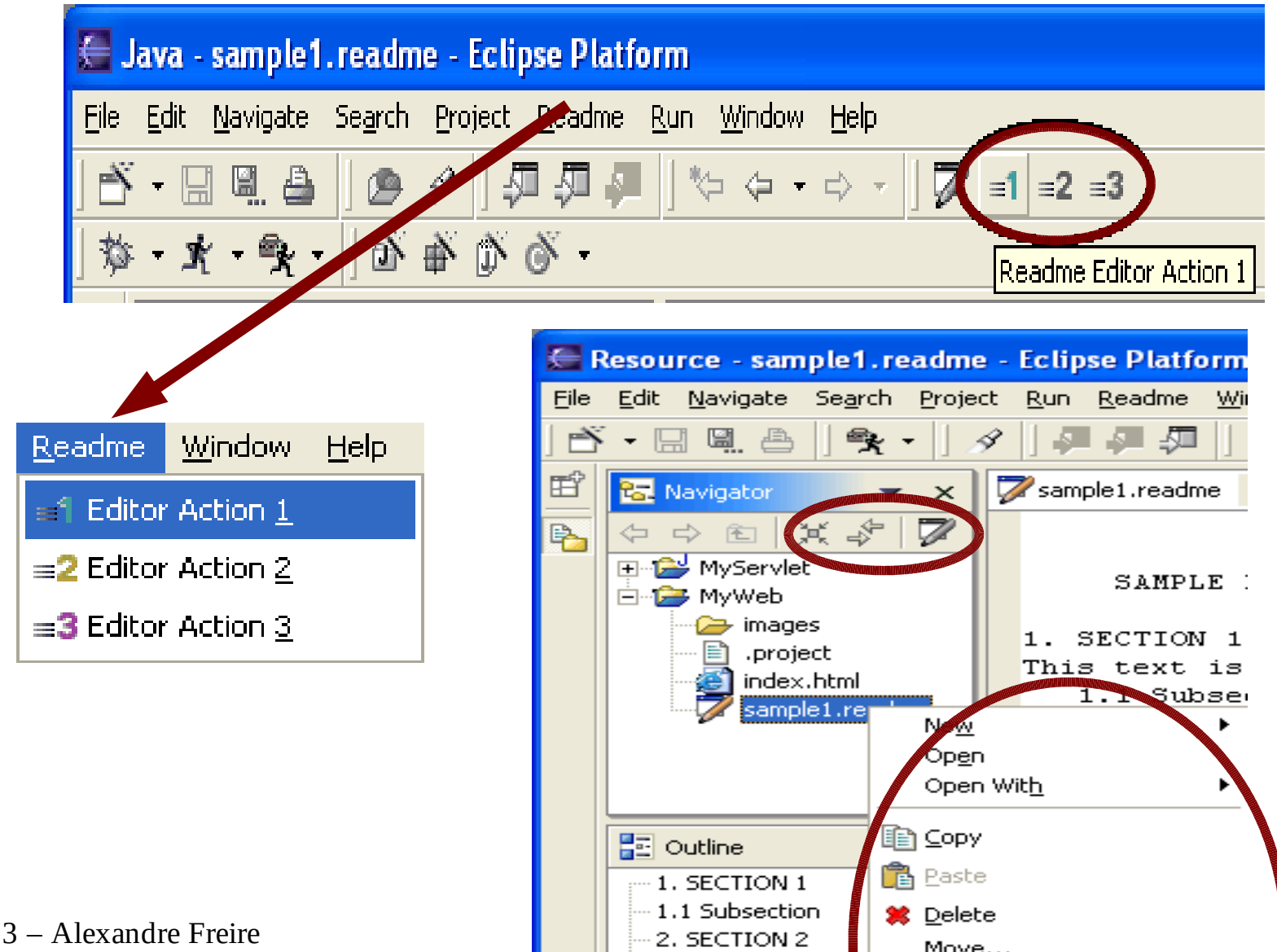
- Sistema de ajuda
  - Define pontos de extensão para plugins proverem ajuda ou documentação
- Suporte a times
  - Define um modelo de gerência e controle de versões dos recursos de um projeto para times
- JDT
  - Ferramentas para criação de um ambiente de desenvolvimento para Java. Conjunto de plugins que extendem a plataforma para visualizar, editar, compilar, depurar e rodar código em Java.
- PDE
  - Ambiente de desenvolvimento e execução de plugins. Define ferramentas que cuidam da criação, manipulação, depuração e implantação de plugins

Plugins são componentes

- Unidades de código reutilizáveis e estensíveis
- Normalmente representam um componente completo do ponto de vista do usuário final
- Definido em várias classes e pacotes
- Componentes com interface gráfica são normalmente divididos em dois plugins, uma para o modelo e outro para interface
- Plugins podem extender pontos de extensão de diversos outros plugins e definir seus próprios pontos de extensão
- São implantados no sub-diretório plugins do diretório de instalação do Eclipse e descobertos em tempo de execução

Pontos de extensão

- Ações
  - do workbench, de editores ou de visualizadores



Mais pontos de extensão

- Novos editores, visualizadores ou perspectivas
- Páginas de ajuda e preferências do plugin
- Wizards
  - criação de recursos
- Natures e Builders
  - compilar recursos
- Markers e Decorators
  - padronizar a visualização de recursos específicos do seu plugin
- Filtros de extensões
  - filtros podem indicar ou não a necessidade de adcionar uma extensão

Estrutura de um plugin

- $ECLIPSE_ROOT/plugins/br.ime.usp.arca.plugin_1.0.0/
  - plugin.xml -> Manifesto do plugin
  - plugin.properties -> i18n
  - about.html -> licensa
  - *.jar -> classes
  - /lib/*.jar -> + classes
  - /icons/*.gif ou *.png -> imagens
- Só a maior versão do plugin vai ser carregada
- Plugins só tem acesso a classes exportadas por outros plugins
- Cada plugin é carregado pelo seu próprio ClassLoader

Ciclo de vida de um plugin

- Durante a inicialização o Eclipse lê todos manifestos de plugins implantados no diretório
- As informações relativas a todos plugins encontrados são armazenadas em um registro de plugins que fica disponível para todos outros plugins em tempo de execução
- Os plugins não são carregados, permitindo que o tempo de abertura do Eclipse seja constante independente do número de plugins instalados, os plugins e suas respectivas classes só são carregados quando estritamente necessário (lazy-loading)
- Plugins ainda não podem ser descarregados

Exemplo de manifesto

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="br.ime.usp.arca.plugin"
   name="Plugin Demo"
   version="1.0.0"
    provider-name=”Curso do CEC”>
</plugin>
```

Exercicio 1 – criar o plugin mais básico que existe, só com um xml, no diretório de plugins, verificar que o plugin foi detectado pelo Eclipse (Help -> About Eclipse Plataform -> Plugin Details)

PDE – preparando o ambiente

- Vamos querer olhar (e copiar) exemplos de plugins, ainda bem que o Eclipse vem com o código dos plugins que o compoêm
- Selecione Preferences->Workbench->Label Decorations e ligue a opção “Binary Plug-in Projects Decoration”
- Selecione Preferences->Plug-in Development -> Java build Path Control e ligue a opção “Use Classpath Containers for Dependent Plug-ins”
- Mude para perspectiva Java e selecione File-> Import -> External Plug-ins and Fragments, selecione “binary projects”e não copie o conteúdo. Com projetos binários podemos trabalhar com o plug-in, mas não modificá-lo, verifique que vários projetos apareceram no seu workspace

Ainda preparando o ambiente

- Você pode verificar que as classes de todos plug-ins estão disponíveis no menu Navigate -> Open Type... e escrevendo “*” no campo de busca
- Você pode optar por filtrar os projetos binários para eles não atolarem sua lista de projetos, no menu do Package Explorer selecione “Filters” e marque a opçao “Binary Projects”
- Faça uma busca pelos manifestos de outros plug-ins, Search-> File Search e escreva plugin.xml no filtro de nomes de arquivo
- Faça uma busca por pontos de extensão, Search -> Plugin Search e escreva “org.eclipse.ui.views”
- Verifique a referência dos pontos de extensão no help on-line do Eclipse (Plataform Plugin Dev... -> Reference -> Extension Point Reference)

Exercício 1 – Hello World

- Vamos criar um plug-in que adciona uma ação que mostra uma menssagem ao Eclipse
- Vamos usar o wizard para issso
- Crie o plug-in e rode ele no run-time workbench
- Ligue o Tracing no run-time workbench para o org.eclipse.runtime plug-in em debug, loader/debug e loader/debug/activateplugin
- Verifique o lazy-loading do plugin
- Tour do código gerado

Tour da interface do PDE

- Vamos olhar o editor do manifesto de um plug-in e suas opcoes
- Olhar tambem o editor de propriedades
- Ver o plugin registry view no run-time workbench

Plugin ClassLoader

- O PluginClassLoader define quais classes são visíveis para cada plugin baseado nos tags <requires> e <runtime> do manifesto do plugin
- <runtime> define onde está o código e recursos do plugin, um atributo opcional do tag “library”, chamado “export” permite que outros plugins tenham acesso ao jar especificado
- <requires> define outros plugins necessários para carregar este plugin. Os plugins são referenciados por ID e não com referências diretas as suas bibliotecas
- Plugins não tem acesso ao classpath normal, por motivos de segurança, isto torna muito comun a criação de plugins que servem de “casca” ao redor de bibliotecas (eg, ver plugin do JUnit.org)

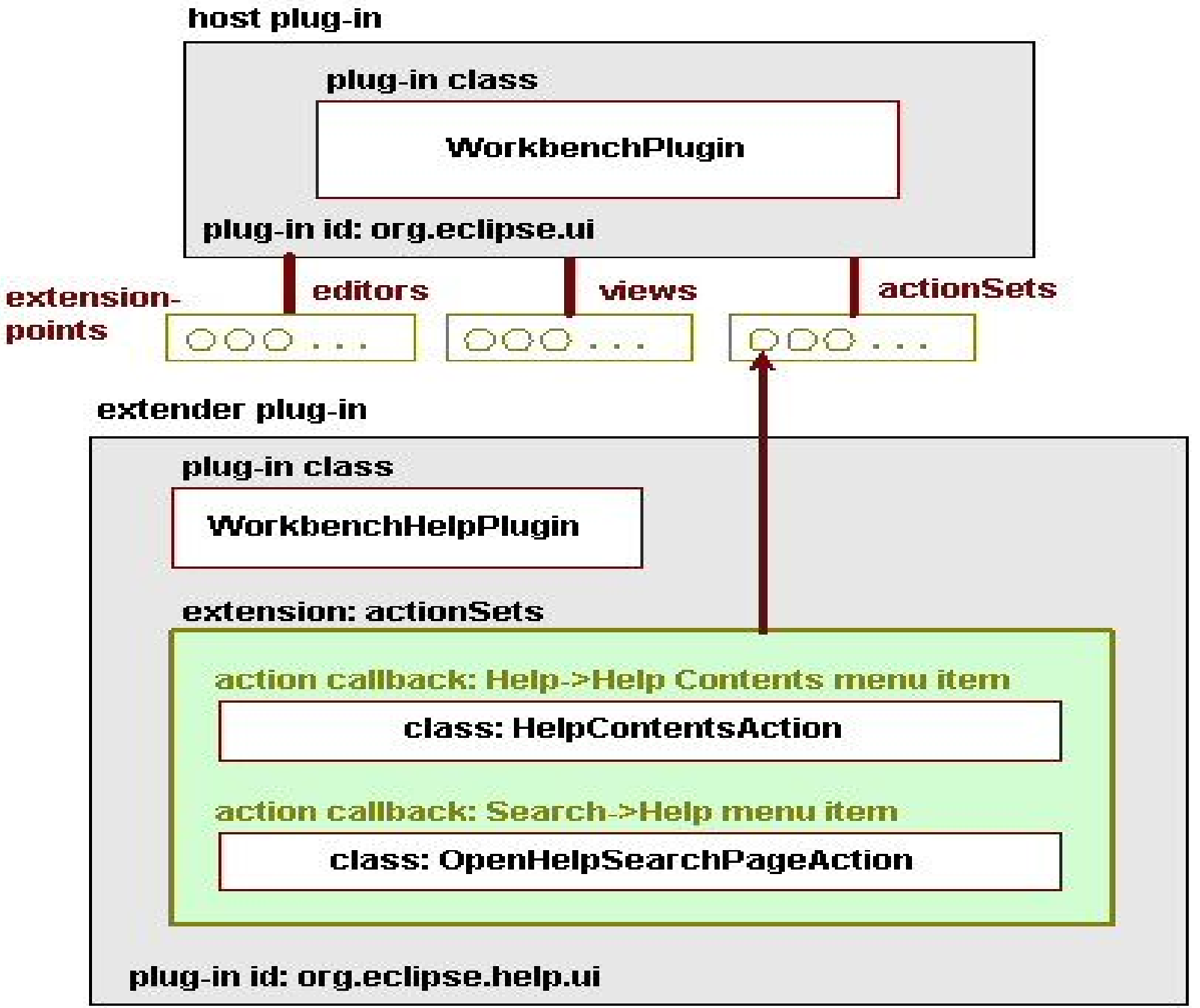
Extensões e inversão de controle

- A plataforma de responsabiliza por verificar quais pontos de extensão um plugin extende, se responsabilizando por construir e fazer chamadas as classes do plugin quando necessário

- Um plugin pode definir pontos de extensão (host plugin) no seu manifesto, ao fazer isto, normalmente define também um schema e uma interface de callback, que caracterizam a extensão e permitem que os plugins que o extendem sigam um mesmo padrão.

- O schema define elementos que são necessários para que o Eclipse possa criar componentes na sua interface sem necessariamente ter que carregar o plugin.

- Plugins extendem outros em seu manifesto (extender plugin) e definem propriedades da sua extensão



Tarefas do Host Plugin

- Define extension points
  - <extension-point id=”Action Sets” schema=”schema/actionSets.exsd”>
- Durante sua inicialização cria *Virtual Proxies* para cada plugin que o extende e cria componentes em sua interface que chamam estes proxies
- Ao ser chamado pela primeira vez, o Proxy delega suas tarefas para a classe definida pelo extender plugin, que somente neste momento é carregado pela plataforma
- Plugins podem ter pontos de extensão que não tem efeito em sua interface, neste caso é possível que um extender plugin nem precise definir classes para criar CallBack Objects
- Outro padrão utilizado para lazy-loading é o *Virtual Adapter*

Tarefas do Extender Plugin

```
<!-- Action Sets -->
    <extension
         point="org.eclipse.ui.actionSets">
        <actionSet
               label="Help"
               visible="true"
               id="org.eclipse.help.internal.ui.HelpActionSet">
         <action
               label="&Help Contents"
               icon="icons/view.gif"
               helpContextId="org.eclipse.help.ui.helpContentsMenu"
               tooltip="Open Help Contents"
               class="org.eclipse.help.ui.internal.HelpContentsAction"
               menubarPath="help/helpEnd"
               id="org.eclipse.help.internal.ui.HelpAction">
            </action>
            <!-- ... other actionSet elements -->
         <action
               label="&Help..."
               icon="icons/search_menu.gif"
               helpContextId="org.eclipse.help.ui.helpSearchMenu"
             class="org.eclipse.help.ui.internal.OpenHelpSearchPageAction"
               menubarPath="org.eclipse.search.menu/dialogGroup"
               id="org.eclipse.help.ui.OpenHelpSearchPage">
            </action>
        </actionSet>
    </extension>
```

Tarefas do Extender Plugin

- Define callback objects para cada extensão
- Pode extender mais de um extension point
- Pode extender mais de um host plugin
- Um único extension point pode ter mais de um callback object
- Define atributos necessários para que o host plugin modifique sua interface sem ter que carregar o plugin
- Pode prover seus próprios extension points
- Pode extender seus próprios extension points

Boas práticas

- Ao desenvolver plugins é uma boa idéia adotar padrões consagrados pela comunidade:
  - Tudo é uma contribuição
  - Contribuições só são carregadas quando extremamente necessário
  - Adcione, não substitua
  - Contribuições devem seguir as interfaces esperadas
  - Contribua somente quando pode fazer algo útil
  - Ao prover pontos de extensão e passar controle para código que não seja seu, proteja-se
  - Sempre que possível, deixe outros contribuirem com a sua contribuição

+ Boas práticas

- Todos clientes tem que seguir as regras do jogo, até você
- Declare explicitamente onde a sua contribuição pode ser extendida
- Pontos de extensão devem aceitar multiplas extensões
- Integre. não separe
- Identifique seu plug-in como a fonte de problema
- Quando muitas contribuições são aplicáveis, deixe o usuário escolher qual delas usar
- Separe a API de classes internas e funcionalidade da UI
- Depois que você convidou outros a contribuirem, não mude as regras
- Revele a API na qual você tem segurança, aos poucos revele o resto

Internacionalizando

- É importante adcionar suporte a internacionalização no seu plug-in
- Como vimos, algumas string são definidas no manifesto, elas também devem ser internacionalizadas
  - use o arquivo plugin.properties e crie alternativas (eg, plugin_pt.properties)
  - strings com o prefixo % no manifesto são procuradas neste arquivo de propriedades
- No código, use ResourceBundles, o Eclipse pode ajudar (Source -> Find Strings to Externalize)
  - strings já exportadas tem um comentário que marca o fato //$NON-NLS-1$
- Exercício 2 – suporte internacional no Hello World plug-in

Internacionalizando

- Normalmente você pode publicar o seu plug-in sem internacionalização e depois disponibilizar um fragmento
- “Fragments” provêm conteúdo adcional a um plug-in existente
- No Eclipse fragmentos são usados para prover pacotes de linguagem e implementações específicas de plataforma (SWT)
- Contêm um fragment.xml muito similar ao manifesto do plug-in

Provendo Ajuda

- Existe um ponto de extensão que permite que você adcione a ajuda do seu plug-in a página comum de ajuda, org.eclipse.help.toc
- Ao extender este ponto define um elemento <toc> (table of contents) que referência um arquivo toc.xml

```
<toc label="PDE Guide">
    <topic label="Introduction to PDE" href="guide/pde.htm"/>
    <topic label="Creating a plug-in" href="guide/pde_creating.htm"/>
    <topic label="Plug-in manifest editor" href="guide/pde_manifest.htm">
        <topic label="Welcome page" href="guide/pde_manifest_welcome.htm" />
        <topic label="Overview page" href="guide/pde_manifest_overview.htm" />
    </topic>
</toc>
```

Ajuda integrada

- TOCs tem pontos na árvore onde nova ajuda pode ser introduzida, são elementos “anchor”
  - para entrar neste ponto da ajuda use o atributo “link_to”
- Ajuda sensível a contexto (tecla F1)
  - você pode associar ajuda de contexto a views, editores, ações, diálogos, wizards, páginas de preferências e propriedades
  - o ponto de extensão é org.eclipse.help.contexts, um elemento <contexts> deve ser criado, referênciando um arquivo contexts.xml e o plugin no qual este arquivo reside
  - ao criar componentes SWT você deve associar o contexto de ajuda programaticamente: WorkbenchHelp.setHelp (viewer.getControl(), “nome_do_plugin.idDoContexto”);

+Ajuda integrada

```
<contexts>
  <context id="remove_all_action_context">
    <description>This command removes all buildfiles from the Ant view.</description>
    <topic label="Ant Support" href="concepts/concepts-antsupport.htm"/>
    <topic label="External Tools" href="concepts/concepts-exttools.htm"/>
  </context>
</contexts>
```

- Ações tem um atributo extra “helpContextId” que deve referênciar um contexto no arquivo contexts.xml

Lidando com erros

- Precisamos lidar com exceções geradas pelo plug-in, apresentar estes erros ao usuário se for o caso, e manter um log de erros
- Objeto CoreException é a raiz de todos erros no Eclipse
- Este objeto contêm um objeto IStatus que identifica o plug-in responsável, contêm o grau do erro, código de status, uma menssagen, e opcionalmente uma exceção
- Se o erro ocorre num contexto no qual você pode informá-lo ao usuário é sempre uma boa idéia fazê-lo usando um diálogo de erro, se não, imprima o erro no log
  - ErrorDialog.openError(parent.getShell(), title, message, coreException.getStatus());
  - getPluginDefault()getLog().log(title, coreException.getStatus());

Tracing

- É uma boa idéia instrumentar o seu plugin com código de debug que pode ser ligado pelo usuário
- As opções de tracing e seus valores são definidos em um arquivo .options na raiz do seu plug-in
  - br.usp.ime.arca.plugin/debug/interface=false
- Como vimos estas opções podem ser ligadas nas preferências do run-time workbench
- No seu código você verifica uma opção chamando:
  - String trueOrFalse = Platform.getDebugOption (“br.usp.ime.arca.plugin/debug/interface”);

Exercicio

- Vamos criar um plug-in que adciona uma entrada no pop-up menu quando uma classe é selecionada (Object Contribuition)
- Use o wizard de estrutura default para plug-in
- Ache um exemplo de plug-in que extenda org.eclipse.ui.popupMenus
- Adpte o exemplo, criando uma classe que implemente IObjectActionDelegate e extendendo o ponto correto no manifesto do seu plug-in
- Adcione ajuda de contexto ao menu
- Adcione tracing para imprimir algo quando a ação for chamada
- Rode seu plug-in e verifique que ele funciona.

Publicando o plug-in

- Queremos disponibilizar nosso plug-in para outros
- Em primeiro lugar precisamos criar um pacote com os arquivos do nosso plug-in, verifique o tag <library> no arquivo manifesto do plug-in Hello World que criamos e o arquivo build.properties. Este último define uma propriedade usada pelo ant na hora de empacotar nosso plug-in
- File -> Export... -> Deployable plug-ins and fragments
  - cria um arquivo zip com o seu plug-in pronto para ser instalado
- Para facilitar ainda mais este processo o Eclipse define o conceito de “Features”
- Com features o Eclipse pode gerênciar a instalação de plug-ins para você

Criando uma Feature

- Junto com o plug-in, uma feature tem informações como a licensa do plug-in, cada feature pode conter diversos plug-ins
- Features são um diretório dentro da pasta $ECLIPSE_ROOT/features/
- E advinhe, existe um arquivo feature.xml

```
<feature
    id="br.usp.ime.arca.minhaFeature"
    label="Feature da Arca"
    version="1.0.0"
    provider-name="Curso do CEC"
    image="arca.jpg"
     <requires><import plugin=”org.junit”/></requires>
     <license> GPL </license>
     <plugin id =”br.usp.ime.arca.plugin” version=”1.0.0”/>
```

Features e update sites

- Ótima notícia, o PDE cria a feature automáticamente para nós (New Project -> Plug-in Development-> Feature Project)
- As features podem então ser disponibilizadas em um update site, para que clientes possam usar o Update Manager do Eclipses para instalar os nossos plug-ins
- Um update site tem dois diretórios, “plugins” e “features” e um arquivo site.xml
- Dentro do diretorio plugins um jar com o conteudo do plugin e a versao do mesmo no final do nome do arquivo deve existir
- Dentro do diretorio features um jar com o feature e a versao do mesmo no final do nome do arquivo deve existir

SWT

- Standard Widget Toolkit é equivalente ao AWT, mas usa componentes nativos do SO
- Se um componente não está disponível em alguma plataforma ele é emulado em Java
- SWT não depende do Eclipse, você pode escrever uma aplicação stand-alone usando-o
- São 4 os componentes básicos:
  - A biblioteca nativa (JNI)
  - Uma classe Display, uma interface com a plataforma GUI
  - Uma classe Shell, a janela raiz da aplicação
  - Uma série de widgets que oferecem componentes básicos

SWT - componentes

- Shell
  - Janela visível para o usuário controlada pelo WM
  - Usada para janela raíz da sua aplicação, onde o resto da GUI deve ser construida, neste caso é shell é filha de Display
  - Usada para criar janelas filhas da sua aplicação principal, como diálogos
- Widgets
  - Widgest seguem o padrão Composite, podem ser incluidos dentro de outros widgets (determinado no construtor
  - Tem Style Bits, que determinam o look-n-feel do widget, o botão por exemplo tem um style bit para indicar se é radio ou checkbox

SWT – lidando com recursos

- Precisamos nos preocupar com a coleta dos recursos usados por componentes SWT!
- Quando um objeto baseado em recursos é criado por nós na aplicação, devemos nos preocupar em coletá-lo
  - Color, Cursor, Font, Images...
- Exemplo:
  - Color blue = new Color (display, 0, 0, 255);
  - blue.dispose();

SWT – lidando com recursos

- Precisamos nos preocupar com a coleta dos recursos usados por componentes SWT!
- Quando um objeto baseado em recursos é criado por nós na aplicação, devemos nos preocupar em coletá-lo
  - Color, Cursor, Font, Images...
- Exemplo:
  - Color blue = new Color (display, 0, 0, 255);
  - blue.dispose();

SWT – organizando widgets

- Para organizar os widgets na tela usamos layouts
- Layouts em SWT são similares ao padrão strategy
- Os layouts podem mudar de aparência dependendo da plataforma nativa
- Layouts tem usam objetos data para determinar onde cada componente entra
- Exemplos:
  - FillLayOut
  - GridLayout e GridData
  - RowLayout e RowData
  - FormLayout e FormData

SWT – eventos

- event dispatch loop

```
while (!shell.isDisposed())
{
  if(!display.readAndDispatch() )
        display.sleep();
}
```

- recebe eventos do SO (eg, mouse) e manda os eventos pro widget apropriado

- Quando escrevemos plug-ins não precisamos nos preocupar, o Workbench cuida do event dispatch loop para nós

- Os eventos são tratados usando o clássico padrão Observer

```
Button b = new Button(display, SWT.PUSH);
b.addSelectionListener(new SelectListener{
                        public void widgetSelected(SelectedEvent e){
                            //botão acionado}
                        });
```

SWT – Threads

- Existe uma thread que cuida de eventos de UI, nenhum operação custosa deve ocorrer nesta thread
- Para isso Display nos oferece dois métodos ajudantes:
  - asyncExec(Runnable) – usado quando vc quer atualizar a interface mas não se importa quando exatamente
  - syncExec(Runnable) – para a thread de UI até este evento conseguir atualizar a interface e depois continua

```
Display.getDefault().asyncExec(new Runnable() {
    public void run(){
        button.setText(new Date().toString());
    }
});
```

SWT – Usando-o

- Para usar SWT você precisa adcionar o jar específico para sua plataforma, no linux usando gtk:
  - swt.jar e swt-pi.jar
  - localizados em $ECLIPSE_ROOT/plugins/org.eclipse.swt.gtk_x.x.x/ws/gtk
- Na hora de executar o programa você tem que mandar um path para onde está localizado o código em C correspondente
  - -Djava.library.path = $ECLIPSE_ROOT/plugins/org.eclipse.swt.gtk_x.x.x/os/linux/x86/

JFace

- Toolkit independente de plataforma construido em cima do SWT, sem porém esconder o SWT
- Provê componentes de alto nível para tarefas comuns de interface
  - Viewers – usados para popular, ordenar, filtrar e atualizar widgets
  - Actions – cria comportamentos e pode ligá-los a menus, toolbars, botões... a mesma ação pode ser compartilhada por widgets
  - Registires – guardam imagens e fontes, cuida de recursos limitados

JFace

- Wizards e Dialogs
  - Usados para prover um arcabouço de interações complexas com o usuário
  - Wizards para tarefas que precisam ser feitas na ordem, com vários passos
  - Diálogos para colher entrada do usuário ou informá-lo sobre algo
- JFace Text
  - alta funcionalidade para editores de texto, com coloração e assistência
  - provê um modelo de documentos para um texto

JFace - Viewers

- Viewers são semelhantes ao padrão Pluggable Adapter
- Permite com que você se concentre no domínio do seu modelo e não se preocupe com os widgets que o representam
- Egs, Trees, Tables, Lists
- Estas classes oferecem métodos que delegam o acesso ao domínio (adaptando o acesso) para objetos separados
  - IContentProvider – dado um objeto de entrada, retorna os objetos de domínio correspondentes e mantêm a interface em sincronia com o modelo (se registra como um observer do modelo)
  - ILabelProvider – retorna um ícone e uma string que representam o objeto do domínio

JFace - + Viewers

- Viewers podem ser customizados sem subclasses, vemos de novo o padrão Strategy
- métodos que delegam operações de ordenação e filtragem para outros objetos
  - setSorter()
  - addFilter()
- Viewers são componentes balck-box, você pode reutilizá-los sem ter que criar subclasses
  - Um mesmo Label Provider pode ser usado em vários viewers
  - O mesmo pode ser dito de filters e sorters

JFace - usando-o

- Não precisamos nos preocupar com o loop de tratamento de eventos, podemos usar org.eclipse.jface.window.ApplicationWindow
- Além dos jars do SWT o JFace depende de:
  - jface.jar
  - runtime.jar (plugin core.runtime)
  - workbench.jar (plugin ui.workbench)
  - boot.jar (plugin core.boot)

Exercícios

- Vamos ver duas aplicações, uma pura SWT e outra pura JFace
- Baixe ambas do cvs:
  - host: sol
  - path: /var/lib/cvs
  - módulos swt e jface
- Verifique as dependências de bibliotecas dos projetos
- Adcione a variável de run-time com as bibliotecas em C para executar os programas
- Tour do código

PDE JUnit – testar plug-ins

- Testes são importantes
- Testes são importantes
- Testes são importantes
- Testes são importantes
- Testes são importantes
- Testes são importantes
- Testes são importantes
- Testes são importantes
- Testes são importantes
- Testes são importantes

PDE JUnit – testar plug-ins

- Precisamos de uma maneira de testar nossos plug-ins, o PDE JUnit oferece um arcabouço para que possamos rodar um run-time workbench, executar testes de JUnit e verificar se esta tudo funcionando corretamente

- Não faz parte do Eclipse 2.x, mas vem incluso no 3.x, depois de instalar o plug-in um novo item no aparece no menu Run-> JUnit Plug-in Test

- Como funciona:
  - Uma instância de run-time do Eclipse começa em outra VM
  - O Controle é passado para o JUnit que roda os testes dentro do mesmo workspace
  - O Eclipse desliga quando os testes acabaram

PDE JUnit – set-up

- Crie um outro plug-in com os testes, ele deve depender do seu plug-in e do plug-in do JUnit
- Usando Project Fixtures
  - É uma boa idéia usar um projeto criado só para testes no workspace de run-time e colocar neste projeto coisas que o seu plug-in precisa para funcionar (eg, criar um projeto python para testar o seu builder com ele)
  - crie o seu projeto mock no setUp() do teste e remova-o no tearDown()
  - O pacote org.eclipse.core.resources contêm classes que você pode usar para criar projetos e arquivos e pastas no seu programa

PDE JUnit – exercicio

- Vamos criar um plug-in que adciona um view ao Eclipse, queremos testar o nosso novo view!
- Crie o plug-in que vai contribuir com a nova view, mas não faça nada no momento
- Crie outro plug-in que vai testar o seu plug-in, adcione o JUnit plug-in e o seu plug-in como dependências, crie uma classe ViewTest que extende TestCase
- Vamos testar se a nossa view pode ser criada
  - Para isso precisamos pegar o página ativa do workbench e pedir para ela mostrar a nossa view (vamos referência-la pelo id)
  - Rode o teste, deixe-o falhar, e depois escreva o código necessário no plug-in para o teste passar

PDE JUnit – exercicio cont.

```
//método para pegar a página ativa
private IWorkbenchPage getPage(){
   IWorkbech wb = PlataformUI.getWorkbench();
   IWorkbenchWindow window = workbench.getActiveWorkbenchWindow();
   return window.getActivePage();
}

public void testShowAndHideView() throws PartInitException {
      //br.usp.arca.plugin.view é o id que vamos dar ao view no nosso plugin
   IViewPart view = getPage().showView(“br.usp.ime.arca.plugin.view”);
   getPage().hideView(view);
}
```

contribuindo ações

- Pontos de extensão
  - org.eclipse.ui.(actionSets, viewActions, editorActions, actionSetPartAssociations, popupMenus)
- Precisamos passar uma classe que implementa IActionDelegate ou uma subclasse desta interface (ver a interface no type hierarchy)
- Ações podem ter filtros (de tipo, nome, ou filtros complexos)

```
public void selectionChanged(IAction a, ISelection s){
    //pega a selection, faz um cast para oq realmente esperamos pegar, pode ter
    //mais de um item selecionado, guarda em uma variável
}

public void run(IAction a){
    //executa ação nos items selecionados
}
```

diálogos e wizards

- Extension points
  - org.eclipse.ui.(preferencePages, propertyPages, newWizards, importWizards, exportWizards)
- Para criar uma página de preferências:
  - Defina a extensão de preferencePages
  - Crie uma PreferenceStore para seu plug-in
  - Implemente a página de preferência usando FieldEditorPreferencePage e associe a PreferenceStore, no método createFieldEditors() adcione os FieldEditors para as chaves que você criou para suas preferências
  - FieldEditors prontos existem para valores booleanos, strings, números, cores, fontes, diretórios, arquivos, classes, etc...

diálogos e wizards

- Páginas de propriedades servem para armazenar propriedades de um dado recurso (eg. projeto)
- Para criar uma página de propriedades:
  - Defina a extensão de propertyPages, indicando qual o tipo de recurso a qual ela se aplica (eg. Ifile)
  - Recursos tem métodos para armazenar e pegar propriedades
    - set/get SessionProperty(QualifiedName key, Object value)
    - set/get PersistentProperty(QualifiedName k, Object v)
  - QualifiedName é um nome composto, normalmente usa um qualificador que é o nome completo do plug-in e um nome local

diálogos e wizards

- Wizards são usados para criar, importar, ou exportar recursos
- Ao extender um ponto para criar um Wizard você deve adcionar WizardPages no método addPages()
- WizardPages criam a interface no método createControl() e tem também um método boolean isPageComplete() que indica se podemos ir para próxima página ou não
- O Wizard tem um método boolean canFinish() que indica se podemos acabar o wizard ou devemos ver ainda mais páginas
- Você pode reutilizar WizardPages que vem com o Eclipse como:
  - WizardNewFileCreationPage, WizardExportPage, WizardResourcePage, NewPackageWizardPage, NewClassWizardPage

Editores

- Para criar um editor primeiro você decide como deve ser sua interface, para editores de texto uma classe completa TextEditor, com coloração e assistência está disponível
- Se seu editor é especial você precisa definir qual o modelo que ele está editando, junto com um ContentProvider e um LabelProvider pro modelo
- Opcionalmente você pode ligar um editor com algum visualizador
- Você pode também adcionar ações ao editor (ao toolbar, menubar, e menu de contexto)

Perspectivas

- Para criar uma nova perspectiva você pode simplismente extender o ponto org.eclipse.ui.perspectives e definir uma classe que obedeça a interface IperspectiveFactory
  - Usae o método createInitialLayout para adcionar views, atalhos para wizards de criação, atalhos para outras perspectivas, e ações
  - Você pode usar a interface IFolderLayout para empilhar views
- Você também pode extender outras perspectivas adcionando views, wizards e atalhos diretamente no seu manifesto, sem ter que escrever nenhum código adcional

Pegando as extensões

- Para que isso seja possível a perspectiva definiu pontos de extensão e quando está sendo construida ela deve pegar seus extenders e fazer oque for necessário

- Para isso um trecho de código muito comun é usado em plug-ins que definem pontos de extensão:

```
IPluginRegistry r = Platform.get|PluginRegistry();
IExtensionPoint ep = r.getExtensionPoint("br.usp.arca.meu_id");
IExtension [] exts = extensionPoint.getExtensions();
for( int i=0; i<exts.length; i++){
    IConfigurationElement [] els = extensions.getConfigurationElements();
    for( int j=0; j<els.length; j++){
        try{
            Object o = configElements[j].createExecutableExtension("class");
            if (o instanceof IMinhaInterface) return (IminhaInterface) o;
        }catch(CoreException e){}
    }
}
```

Muito obrigado!

gsd.ime.usp.br
arca.ime.usp.br

alex@arca.ime.usp.br