UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
ESCOLA DE INFORMÁTICA APLICADA

Desenvolvimento de um plug-in Eclipse para visualizar o histórico de alterações em arquivos utilizando o SVN

Amanda Aguiar da Motta

Orientador
Márcio de Oliveira Barros

RIO DE JANEIRO, RJ – BRASIL
DEZEMBRO DE 2013

Desenvolvimento de um plug-in Eclipse para visualizar o histórico de alterações em arquivos utilizando o SVN

Amanda Aguiar da Motta

Projeto de Graduação apresentado à Escola de Informática Aplicada da Universidade Federal do Estado do Rio de Janeiro (UNIRIO) para obtenção do título de Bacharel em Sistemas de Informação.

Aprovada por:

_____

Prof. Márcio de Oliveira Barros, DSc. (UNIRIO)

_____

Profa. Flávia Maria Santoro, DSc. (UNIRIO)

_____

Prof. Alexandre Luís Correa, DSc. (UNIRIO)

RIO DE JANEIRO, RJ – BRASIL.
DEZEMBRO DE 2013

Agradecimentos

Agradeço ao professor Márcio Barros pela dedicação, incentivo e paciência durante o desenvolvimento deste projeto.

Agradeço à minha família por estar sempre ao meu lado apoiando as minhas escolhas e incentivando o meu desenvolvimento como pessoa.

Agradeço aos amigos da Unirio, que estiveram comigo durante estes 4 anos, nos trabalhos, nos estudos, nas conversas e brincadeiras.

Agradeço também a todos os docentes da Unirio, que contribuíram para a minha formação profissional e acadêmica.

RESUMO

Este projeto apresenta a criação de um plug-in para o ambiente de desenvolvimento de software Eclipse. O plug-in tem como objetivo permitir a visualização de um histórico de *commits* dos arquivos componentes de um projeto de software através da comunicação com o sistema de controle de versões Subversion (SVN). Esta comunicação é feita através da API SVNKit.

Para implementar o plug-in foi necessário entender os conceitos de sistemas de controle de versões, seu funcionamento e objetivos. Também foi necessário aprender sobre o desenvolvimento de plug-ins para o Eclipse, além da utilização da API SVNKit. Este trabalho apresenta os conceitos envolvidos na resolução do problema, passando pela modelagem do plug-in até a apresentação das suas telas.

Palavras-chave: Plug-in, Eclipse, Subversion, Sistemas de Controle de Versões

ABSTRACT

This project introduces the creation of a plug-in for the Eclipse software development platform. The plug-in allows the visualization of a history of *commits* made to files comprising a software project. Such history is collected by communicating with the Subversion version control system. This communication is performed using the SVNKit API.

In order to develop the plug-in, we had to understand concepts related to version control systems, including their operation and goals. It was also necessary to learn about developing Eclipse plug-ins and using the SVNKit API. This work portrays the concepts involved in developing and using the plug-in, including the models that were built to support its implementation as well as screenshots.

Keywords: Plug-in, Eclipse, Subversion, Version Control Systems

Índice

Índice de Figuras

1 Introdução

1.1 Motivação

Trabalhar em equipe significa realizar um esforço coletivo para resolver um problema. No trabalho em equipe, cada membro sabe o que os outros estão fazendo e reconhece a importância de cada indivíduo para o sucesso da tarefa. Este tipo de trabalho envolve troca de conhecimento para maior agilidade no cumprimento de metas e objetivos compartilhados.

Em uma equipe de desenvolvimento de software, o compartilhamento de conhecimento e de informações pode prevenir problemas como retrabalho, baixa eficácia na produção, segregação de conhecimento, entre outros. É comum que vários programadores manipulem ao mesmo tempo determinado grupo de arquivos. Nestes casos, se faz necessário o uso de sistemas de controle de versões.

Os sistemas de controle de versões têm como finalidade gerenciar diferentes versões de um arquivo. Eles permitem que vários desenvolvedores trabalhem em paralelo sem que um sobrescreva o código do outro. Toda a evolução de um projeto fica registrada em seu repositório do sistema de controle de versões, incluindo cada alteração realizada sobre cada arquivo. Com essas informações, é possível saber quem fez que tipo de alteração, quando e por qual motivo.

Quando um desenvolvedor precisa alterar um arquivo, é necessário ter algum conhecimento sobre seu conteúdo: por exemplo, quem fez as últimas alterações, por

qual motivo e o que foi alterado. Um sistema de controle de versões, se utilizado da maneira correta, pode manter todas estas informações.

1.2 Objetivos

Tendo em vista que a disseminação das informações sobre as alterações realizadas nos arquivos componentes de um projeto de software, na maioria das vezes, não ocorre da melhor forma possível, buscou-se criar uma maneira simples de possibilitar a distribuição do conhecimento individual entre os desenvolvedores diretamente no seu ambiente de trabalho.

A proposta deste projeto é embutir estas informações no próprio ambiente de desenvolvimento de software, através do uso de plug-ins, a fim de facilitar a comunicação e a visualização das informações desejadas. Os *plug-ins* permitem estender as funcionalidades de um ambiente de desenvolvimento de software, como o Eclipse, que é utilizado neste projeto. Espera-se que, com o uso desta ferramenta, haja um aumento na qualidade dos produtos e diminuição dos problemas apresentados durante o processo de desenvolvimento dos mesmos.

1.3 Organização do texto

O presente trabalho está estruturado em capítulos e, além desta introdução, será desenvolvido da seguinte forma:

- Capítulo II: Conceitos – introduz o leitor aos conceitos relacionados ao tema do projeto, definindo-os e explicando de forma sucinta seus pontos principais;

- Capítulo III: Análise e solução técnica – trata do processo de análise do plug-in que foi desenvolvido neste trabalho. Apresenta os seus requisitos funcionais, seus

diagramas de classes e sequência, a modelagem dos seus casos de uso e as tecnologias utilizadas no seu desenvolvimento;

- Capítulo IV: O Plug-in – o início do capítulo traz uma breve explicação sobre como é feita a instalação do plug-in. Em seguida, são apresentadas as funcionalidades do mesmo, suas principais telas e as interações com o usuário que ele oferece;

- Capítulo V: Conclusões – Reúne as considerações finais, assinala as contribuições da pesquisa e sugere possibilidades de aprofundamento posterior.

2 **Conceitos**

Neste capítulo são apresentados os conceitos necessários para o bom entendimento deste projeto. O mesmo trata da construção de um plug-in para o ambiente de desenvolvimento Eclipse. O plug-in integra informações de um sistema de controle de versões para auxiliar os desenvolvedores na execução de suas tarefas no contexto da construção de um projeto de software.

A seção 2.1 traz uma explicação sobre os sistemas de controle de versões e seus benefícios para uma equipe de desenvolvimento de software. A seção 2.2 explica o funcionamento e a finalidade dos plug-ins para o ambiente Eclipse. A seção 2.3 descreve brevemente o funcionamento da API SVNKit, utilizada no desenvolvimento deste projeto. Por fim, na seção 2.4 são feitas as considerações finais sobre os assuntos tratados neste capítulo.

2.1 Sistemas de Controle de Versões

Alguns problemas comuns no desenvolvimento de software são causados por falta de controle de versões, como:

- Sobrescrita de código – Desenvolvedores realizam alterações sobre o mesmo arquivo e, por acidente, o código de um deles sobrescreve o do outro;
- Falta de controle de alterações – É difícil obter informações sobre modificações realizadas pelos desenvolvedores nos arquivos componentes de um sistema (quem fez o quê, o que foi feito, quando foi feito...);

- Dificuldade de recuperação de código – A falta de controle sobre os arquivos torna difícil a recuperação de versões anteriores do sistema.

Uma forma de evitar problemas e situações desse tipo é utilizando um Sistema de Controle de Versão (SCV) [Dias, 2011], que se torna essencial para o desenvolvimento em equipe. Esse sistema mantém um conjunto organizado de todas as versões de arquivos que são feitas ao longo do ciclo de desenvolvimento, permitindo também que os desenvolvedores possam voltar para revisões anteriores e comparar diferenças entre versões. Um SCV também possibilita rastrear cada mudança, o motivo pelo qual foi feita, reverter alterações, entre outros recursos. Estas funcionalidades podem fazer toda a diferença no sucesso de um projeto.

O controle de versão possibilita que vários desenvolvedores trabalhem em paralelo sobre os mesmos arquivos sem que um sobrescreva o código de outro. É possível que pessoas que estão geograficamente separadas trabalhem de forma conjunta, através da Internet ou rede privada, utilizando o mesmo repositório e contribuindo no desenvolvimento.

A estrutura de um SCV é composta de duas partes: o repositório e a área de trabalho. Os arquivos do projeto ficam armazenados no repositório e o histórico de suas versões é salvo nele. O desenvolvedor não trabalha diretamente nos arquivos do repositório. Ele usa uma área de trabalho pessoal, individual e isolada das demais áreas de trabalho, que contém uma cópia dos arquivos do repositório e que é monitorada para identificar as mudanças realizadas.

A comunicação e sincronização entre a área de trabalho e o repositório são feitas através de comandos que variam de acordo com a ferramenta de controle de versões. Basicamente, os comandos variam entre enviar as modificações feitas na área de trabalho para o repositório e recuperar as modificações presentes no repositório para a

área de trabalho. Toda vez que informações são enviadas para o repositório, uma versão nova do arquivo é criada. O conjunto destas versões é o histórico do projeto.

Conflitos ocorrem quando, por exemplo, dois desenvolvedores fazem alterações em sua área de trabalho sobre o mesmo arquivo e enviam para o repositório. Estes conflitos podem ser resolvidos através de uma operação conhecida como *merge* (*fusão*). Ela permite que duas versões do mesmo arquivo sejam incorporadas em uma versão combinada com todas as modificações realizadas. Quando o conflito é direto, ou seja, as alterações são no mesmo ponto do documento, a fusão pode ser feita manualmente. Caso contrário, ocorre a fusão automática das duas versões através da comparação entre elas.

Atualmente, os sistemas de controle de versões são classificados em dois tipos: centralizados e distribuídos [Dias, 2011]. Sistemas de controle de versões centralizados trabalham apenas com um servidor central e diversas áreas de trabalho, baseados na arquitetura cliente-servidor. Por ser centralizado, as áreas de trabalho precisam primeiro passar pelo servidor para poderem se comunicar, conforme mostra a Figura 1. Essa versão atende a maioria das equipes de desenvolvimento que não sejam muito grandes e trabalhem em uma rede local. Alguns dos sistemas centralizados são: Subversion[1], Microsoft Visual Source Safe[2] e Concurrent Version System[3].

[1] http://subversion.apache.org/download/
[2] http://msdn.microsoft.com/pt-br/library/ms181038%28v=vs.80%29.aspx
[3] http://cvs.nongnu.org/

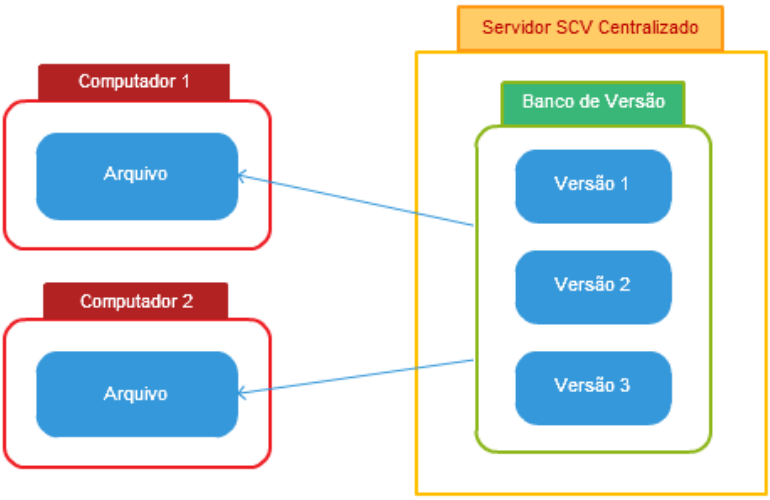

Figura 1 - Esquema de funcionamento de um SCV centralizado

Nos sistemas distribuídos, existem vários repositórios autônomos e independentes, sendo um para cada desenvolvedor. Estes podem se comunicar uns com os outros sem ter que passar por um servidor central, conforme ilustrado na Figura 2. Ou seja, o código que um desenvolvedor tem em seu computador pode ser totalmente diferente daquele que está no repositório central ou na área de trabalho de outro desenvolvedor.

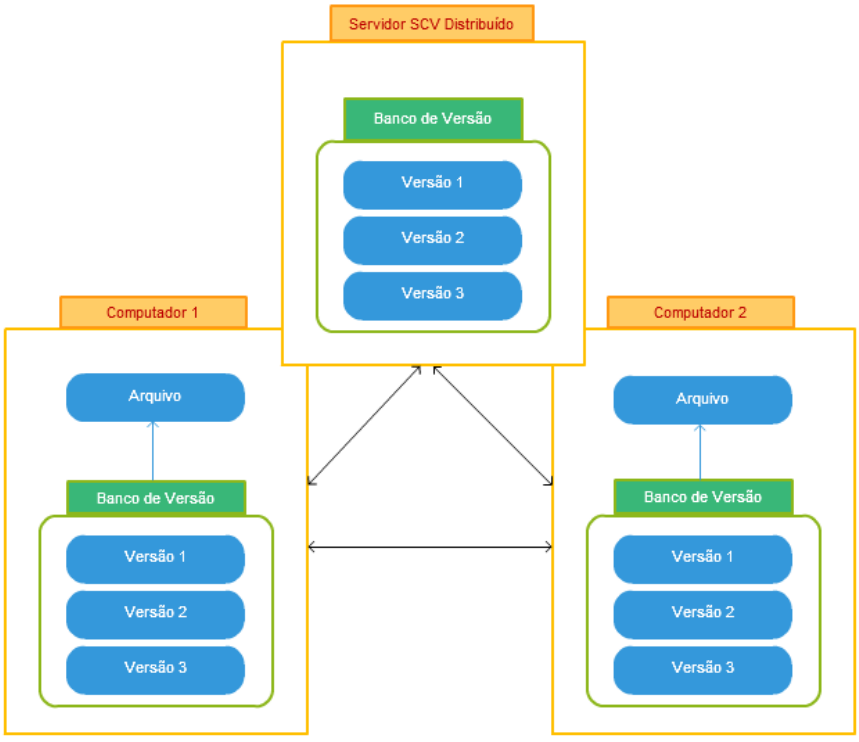

Figura 2 – Arquitetura de um SCV distribuído

A comunicação entre o servidor principal e as áreas de trabalho funciona com duas operações para atualizar e fundir (*merge*) o projeto, chamadas de *pull* e *push* (puxar e empurrar). Com a operação *pull* é possível atualizar o repositório local com as alterações feitas em outro repositório. A operação *push* faz o inverso, enviando para outro repositório as alterações realizadas no repositório local. Algumas das ferramentas de controle de versão distribuído são Git[4] e Mercurial[5].

A utilização dos Sistemas de Controle de Versões é comprovadamente uma prática eficaz na Engenharia de Software [Dias, 2011], pois resolve diversos problemas (conforme citado acima) e torna o desenvolvimento mais seguro e eficiente.

2.2 Desenvolvimento de plug-ins para o Eclipse

Um dos fatores para o grande sucesso do ambiente de desenvolvimento de software Eclipse[6] é a possibilidade de se construir funcionalidades que são integradas de forma transparente ao ambiente. A chave para esta integração são os plug-ins, pequenos sistemas que se integram à ferramenta, aumentando seu conjunto padrão de funcionalidades [Gallardo, 2002].

2.2.1 O plug-in

Os plug-ins para o ambiente Eclipse podem ser construídos através do *Plug-in Development Environment* (PDE)[7], que é fornecido como uma extensão padrão do Eclipse. Desta forma, é possível criar e testar um plug-in dentro do próprio ambiente de execução do Eclipse.

2.2.2 Os elementos do plug-in

[4] http://git-scm.com/
[5] http://mercurial.selenic.com/
[6] http://www.eclipse.org/
[7] http://www.eclipse.org/pde/

Um projeto de plug-in segue a forma básica de qualquer programa Java. Todo código-fonte fica encapsulado em pacotes e precisa implementar uma classe denominada *Activator*, de forma que o programa do plug-in possa ser chamado pela IDE. A Figura 3 ilustra a organização dos arquivos em um projeto de plug-in.

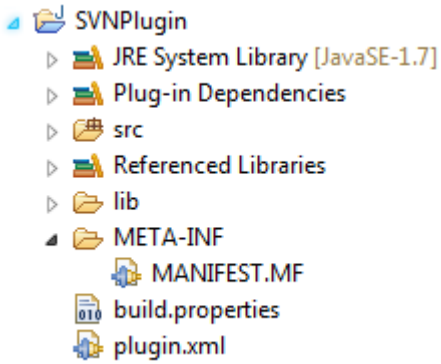

Figura 3 - Organização dos arquivos em um projeto de plug-in

O ambiente de desenvolvimento de plug-ins fornece uma página dividida em abas e baseada em formulário que permite a edição dos três arquivos gerados automaticamente pelo Eclipse no momento da criação do projeto: *MANIFEST.MF*, *plugin.xml* e *build.properties*. Esta página é usada como se apenas um arquivo estivesse sendo editado, enquanto o Eclipse lida com a tarefa de escrever os dados para os diferentes arquivos do projeto. As abas são as seguintes:

- *Plug-in Overview* – Possui seções que definem propriedades importantes do plug-in, como seu identificador, o nome, a versão e a classe que inicializará o plug-in. Possui também uma seção com breves referências de como desenvolver, testar e implantar plug-ins. Essa última seção provê links que direcionam para outras abas ou invocam comandos;
- *Plug-in Dependencies* – Página que define quais as dependências que o plug-in que está sendo desenvolvido possui com outros plug-ins. São listados todos os plug-ins que devem estar presente no *classpath* do projeto para que o mesmo possa compilar;

- *Plug-in Runtime* – Esta aba possui uma seção que lista todos os pacotes que o plug-in expõe para outros, podendo especificar a visibilidade para determinados plug-ins. Ainda nesta aba, temos uma seção denominada *classpath*, que lista a localização das bibliotecas necessárias para a execução do projeto do plug-in;
- *Plug-in Extensions* – O Eclipse possui os chamados "pontos de extensão". Quando um plug-in quer permitir que outros plug-ins estendam parte de suas funcionalidades, ele deve declarar um ponto de extensão. O ponto de extensão declara um contrato, normalmente uma combinação de arquivo XML e interfaces Java, que as extensões devem respeitar. Na aba *Extensions* são declarados todos os pontos de extensão que o plug-in em desenvolvimento irá estender. Alguns exemplos de pontos de extensão:
    - *org.eclipse.ui.menus* – Permite que um plug-in estenda a interface com o usuário com seleções de menus e botões da barra de ferramentas;
    - *org.eclipse.ui.preferencePages* – Utilizado para adicionar novas opções à página de preferências do Eclipse;
    - *org.eclipse.ui.perspective* – Adiciona uma nova perspectiva ao ambiente do Eclipse.
- *Plug-in Extension Points* – nesta página são declarados todos novos pontos de extensão que o plug-in fornecerá para os demais;
- *Plug-in Build* – Contém toda a informação necessária para que o plug-in seja construído, empacotado e exportado. Todas as modificações feitas nessa aba são salvas no arquivo *build.properties*, responsável por guiar o processo de construção do plug-in.

Grande parte das informações acima descritas são salvas no arquivo *plugin.xml*. Ele pode ser considerado o principal elemento do plug-in, sendo escrito como na Figura 4.

```
<plugin>
   <requires>
         <import plugin="org.eclipse.core.runtime"/>
         <import plugin="org.eclipse.ui"/>
   </requires>
   <runtime>
         <library name="svnplugin.jar"/>
   </runtime>
   <extension point="org.eclipse.ui.views">
        <view category="SVNPlugin"
            class="br.unirio.svnplugin.HistoricoView"
            id="SVNPlugin.HistoricoView"
            name="Histórico View"
            restorable="true">
         </view>
         <category id="SVNPlugin"
            name="SVNPlugin">
         </category>
   </extension>
   <extension point="org.eclipse.ui.preferencePages">
         <page class="br.unirio.svnplugin.SVNPreferencePage"
            id="br.unirio.svnplugin.SVNPreferencePage"
            name="SVN Preferences">
         </page>
   </extension>
   <extension point="org.eclipse.core.runtime.preferences">
         <initializer class="br.unirio.svnplugin.PreferenceInitializer">
         </initializer>
   </extension>
</plugin>
```

Figura 4 – Exemplo de um arquivo plugin.xml

As *tags* presente no arquivo *plugin.xml* têm significados específicos. A tag *runtime* define as configurações de tempo de execução do plug-in. Pode definir, por exemplo, o nome do arquivo JAR em que o plug-in será compilado e as bibliotecas externas que o plug-in utiliza. A tag *extension* define cada uma das extensões implementadas pelo plug-in. O atributo *point* dessa tag define qual ponto de extensão será utilizado. Dentro desta tag, o desenvolvedor pode colocar outras tags, para informar qual o pacote e a classe que contém a implementação daquela extensão, em quais circunstâncias aquela extensão estará habilitada, entre outras informações. Cada tipo de extensão deve

implementar uma classe diferente. Finalmente, a tag *requires* define que plug-ins são necessários para que o plug-in que está sendo desenvolvido funcione corretamente.

2.3 SVNKit

SVNKit[8] é uma biblioteca de código livre inteiramente desenvolvida em Java que permite a conexão de um programa Java com o Sistema de Controle de Versão SVN. A Figura 5 mostra a estrutura da API.

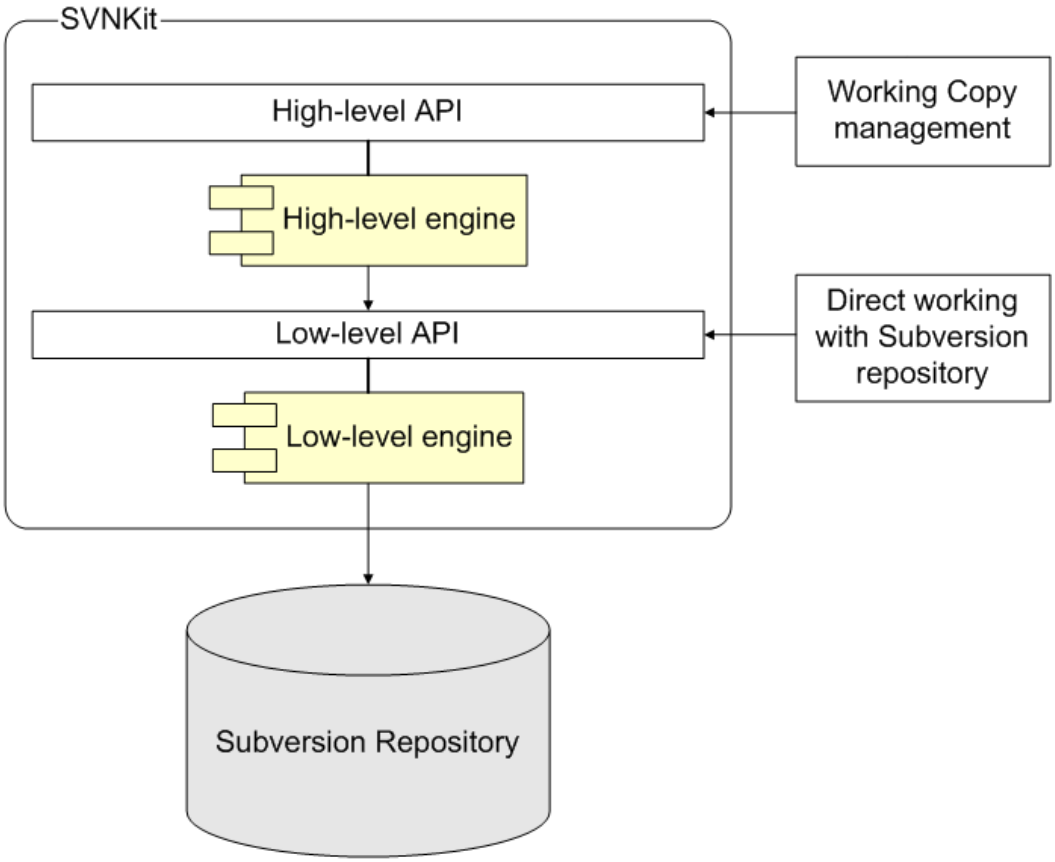

Figura 5 - Estrutura da API SVNKit

Existem duas camadas no SVNKit:

[8] http://svnkit.com/

- *High-level* - Camada de alto nível, utilizada para gerenciar áreas de trabalho (*checkout*, *update*, *switch*, etc.). Esta camada oferece comandos similares aos oferecidos por clientes de linha de comando do Subversion;
- *Low-level* - Camada de baixo nível, que representa um *driver* que permite o trabalho direto com um repositório Subversion. Esta camada se comunica com o repositório através de protocolos, como mostra a Figura 6. A versão atual da API disponibiliza três formas de comunicação com os repositórios.

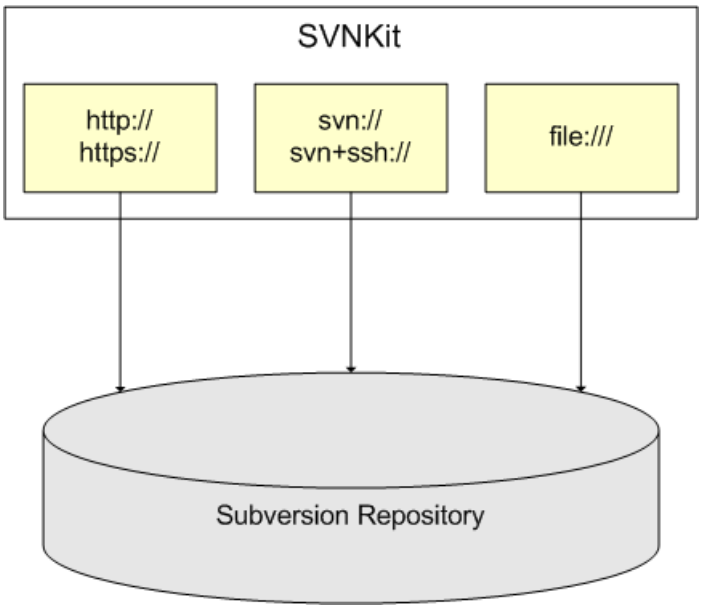

Figura 6 - Esquemas de acesso aos repositórios

A API é disponibilizada na forma de bibliotecas *.jar*. Para utilizá-las em um projeto desenvolvido no ambiente Eclipse, basta adicioná-las como bibliotecas referenciadas e incluí-las no *classpath* do projeto. Desta forma será possível utilizar as classes que compõem a API.

2.4 Considerações Finais

Nesse capitulo foi apresentado o conceito de plug-ins para o ambiente de desenvolvimento de software Eclipse, bem como sua utilização e os principais benefícios decorrentes do seu uso.

Viu-se que os sistemas de controle de versão são fundamentais no dia-a-dia das equipes de desenvolvimento de software, pois permitem que várias pessoas trabalhem simultaneamente nos mesmos arquivos sem que um desenvolvedor sobrescreva o código-fonte criado ou alterado outro. Possuem também outras funcionalidades, como o versionamento do projeto e a manutenção do histórico de alterações.

No próximo capitulo será detalhado o processo de levantamento de requisitos do plug-in, os casos de uso que ele irá apoiar e os diagramas de classes e sequência referentes à implementação destes casos de uso.

3 Análise e solução técnica

Este capítulo irá retratar o processo de análise do plug-in para o ambiente Eclipse que foi desenvolvido com o intuito de promover uma integração com o sistema de controle de versões SVN. Na seção 3.1 serão apresentados os requisitos funcionais do plug-in, juntamente com seus casos de uso. A seção 3.2 traz a modelagem conceitual do plug-in, diagramas de classes e sequência. A seção 3.3 trata das tecnologias que foram utilizadas para o desenvolvimento do mesmo. Por fim, a seção 3.4 contém as considerações finais referentes a este capítulo.

3.1 Requisitos do plug-in

Os requisitos de um sistema podem ser modelados e validados através de casos de uso. Os casos de uso foram propostos inicialmente pelo cientista da computação sueco Ivar Hjalmar Jacobson em sua metodologia de desenvolvimento de sistemas orientados a objetos. Ele define caso de uso como sendo uma "descrição de um conjunto de sequências de ações, inclusive variantes, que um sistema executa para produzir um resultado de valor observável por um ator" [Booch, 2000].

Um caso de uso envolve a interação dos atores com o sistema. Um ator representa a figura de um ser humano, de algum dispositivo de hardware ou de algum outro sistema que interaja com o sistema a ser construído. Como exemplos de atores podemos ter clientes, usuários, impressoras, computadores, entre outros.

Os casos de uso que o plug-in apoiará são:

- Inserir dados de conexão com repositório do SVN – neste caso de uso o usuário deve entrar na página de preferências do Eclipse (acessada através do menu *Window*) e registrar o endereço do repositório, um login e uma senha de acesso. Desta forma será possível obter os dados referentes às modificações feitas nos arquivos presentes no repositório;
- Visualizar log de *commits* – têm como objetivo exibir o log completo de *commits* feitos no arquivo que está aberto no editor selecionado no Eclipse naquele momento. Os dados exibidos são o desenvolvedor autor do *commit*, número da revisão, data do *commit* e a mensagem enviada junto ao *commit*;
- Visualizar log de *commits* por desenvolvedor – têm como objetivo exibir uma estatística de *commits* feitos no arquivo que está aberto no editor selecionado no Eclipse, agrupada por desenvolvedor. Os dados exibidos são o desenvolvedor autor do *commit* e sua porcentagem de *commits*;
- Visualizar log de *commits* por período – têm como objetivo exibir uma estatística de *commits* feitos no arquivo que está aberto no editor selecionado no Eclipse, agrupada por período. Os dados exibidos são o período dos *commits* (mês/ano) e porcentagem de *commits* naquele período.

A Figura 7 retrata o diagrama de casos de uso utilizado no desenvolvimento do plug-in. Nele, vemos o usuário do ambiente de desenvolvimento de software como ator principal e ligado aos quatro casos de uso. Temos também um ator que representa o sistema de controle de versões, com quem o sistema troca mensagens para atender aos três casos de uso de visualização das informações deste sistema.

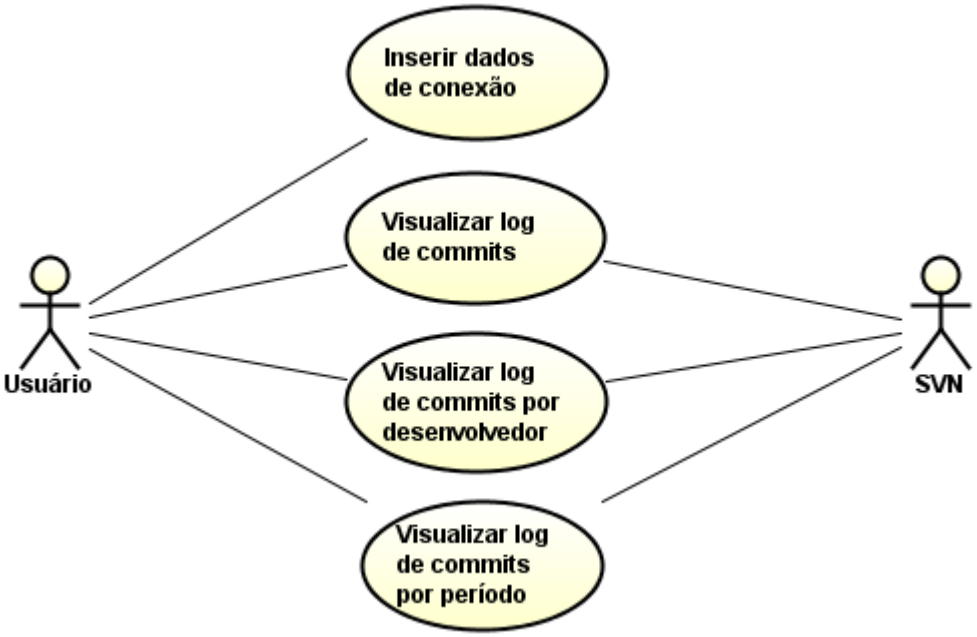

Figura 7 – Diagrama de Casos de Uso

3.2 Modelagem do plug-in

A partir da análise dos requisitos e da especificação dos casos de uso, deu-se início ao desenvolvimento do plug-in. A Figura 8 apresenta um diagrama com as principais classes do domínio do plug-in.

SVNPluginActivator é a classe que trata do ciclo de vida do plug-in. Ela é a primeira classe a ser notificada quando o plug-in inicializa, isto é, quando a *view* é carregada e a última a ser notificada quando o Eclipse é fechado e o plug-in finalizado. Ela é utilizada caso alguma ação precise ser executada nesses dois momentos, o que não é o caso deste plug-in. Porém ela se fez necessária, pois facilita o acesso aos dados referentes a janela de preferências do Eclipse, como será explicado mais a frente.

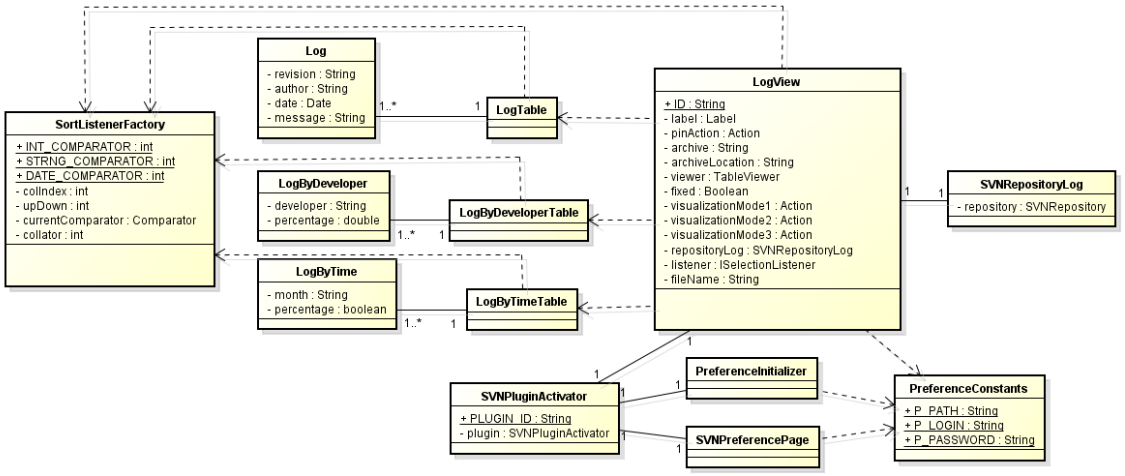

Figura 8 - Diagrama de classes do plug-in

O Eclipse possui uma classe abstrata, chamada *org.eclipse.ui.part.ViewPart* que implementa os comportamentos padrões da interface *org.eclipse.ui.IViewPart*. Esta interface define as responsabilidades de uma *view* no ambiente do Eclipse. Uma *view* é um dos tipos de componentes que formam uma janela do Eclipse. Elas podem exibir informações de forma hierárquica ou propriedades de um determinado objeto. Frequentemente são fornecidas informações para apoiar um usuário trabalhando no editor correspondente, como por exemplo, uma view que exiba as propriedades de um objeto que está sendo editado no momento. A classe *LogView* estende a classe abstrata *ViewPart*. É nela que são criados e preenchidos os componentes da *view* com o histórico de alterações dos arquivos versionados.

As classes *LogTable*, *LogByDeveloperTable* e *LogByTimeTable* são auxiliares para a criação das tabelas dos respectivos modos de visualização. O conteúdo de cada tabela é armazenado em objetos das classes *Log*, *LogByDeveloper* e *LogByTime*. Cada linha da tabela corresponde a uma instância de uma dessas classes. Uma lista desses objetos representa o conteúdo total de uma tabela. Essas listas de objetos são montadas na classe *LogView* e passadas para as classes responsáveis por criar as tabelas.

A classe *SVNRepositoryLog*, por sua vez, é responsável por fazer a comunicação

com o SVN. Utilizando as classes do SVNKit, ela estabelece a conexão com o repositório e obtém as informações sobre as modificações dos arquivos. A Figura 9 apresenta um diagrama com as classes que foram utilizadas da API.

Para que seja possível reordenar a tabela do log baseado em uma coluna, é necessário criar uma classe que implemente a interface *org.eclipse.swt.widgets.Listener*. Uma instância desta classe será adicionada a cada coluna da tabela, o que fará com que as mesmas "ouçam" quando um clique é dado sobre elas e executem uma ação. A classe *SortListenerFactory* possui este papel no plug-in.

As classes *PreferenceConstants*, *PreferenceInitializer* e *SVNPreferencePage* são utilizadas para criar uma seção na janela de preferências do Eclipse em que o usuário irá inserir os dados de conexão com o repositório SVN. Os dados necessários são o caminho do repositório em que os arquivos se encontram, seu login e senha de acesso.

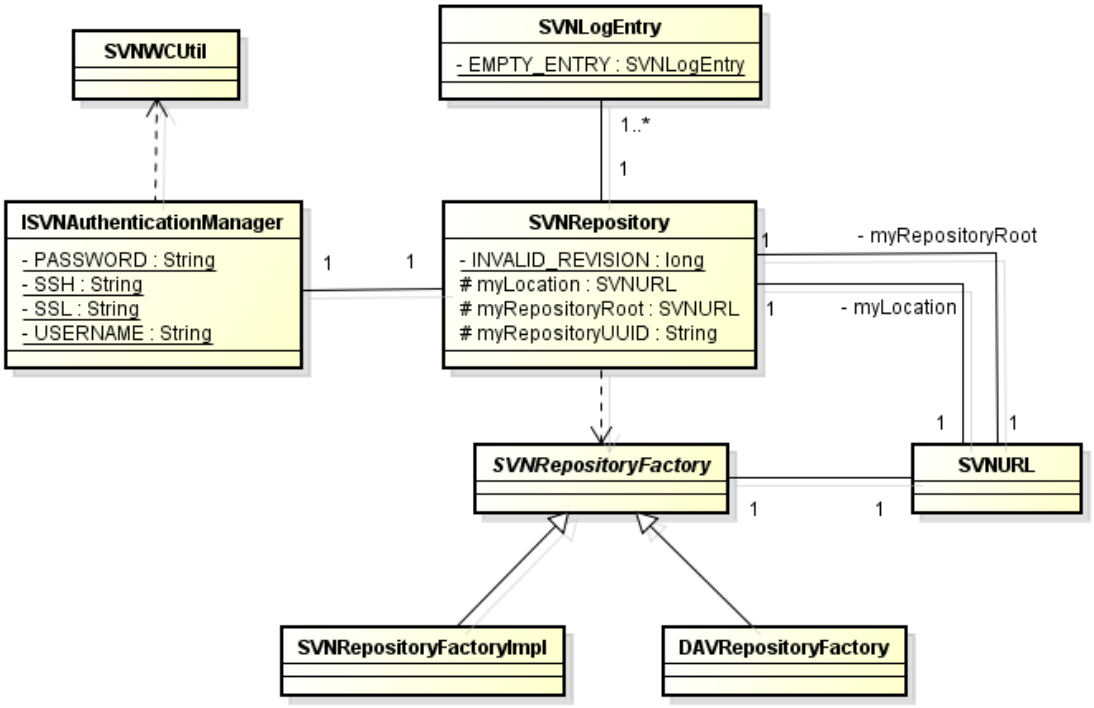

Figura 9 - Diagrama de classes da API SVNKit

A classe abstrata *SVNRepositoryFactory* é responsável por criar um driver *SVNRepository* apropriado para o protocolo de acesso que está sendo utilizado no plug-

in. Cada protocolo é representado por uma classe que estende esta classe abstrata, como por exemplo, as classes *DAVRepositoryFactory* e *SVNRepositoryFactoryImpl*, utilizadas para comunicação via protocolo *http* e protocolo *svn*, respectivamente.

A classe utilitária *SVNWCUtil* é utilizada para manusear a cópia de trabalho do usuário. Ela fornece alguns métodos para fins tais como a criação de configuração de diretório em tempo de execução, drivers de autenticação e alguns outros.

ISVNAuthenticationManager é uma interface implementada por classes de gerenciamento que são utilizadas pelo driver *SVNRepository* para fins de autenticação do usuário. Quando um driver é criado, um gerenciador de autenticação deve ser fornecido. A classe *SVNWCUtil* é responsável por criar este gerenciador, a partir do login e senha fornecidos pelo usuário.

Todos os métodos da API que necessitam da localização do repositório para a realização de uma operação recebem uma instância da classe *SVNURL*. Esta classe possui métodos que verificam a validade do repositório, codificam caminhos, realizam parses em URLs, entre outros. Ela é utilizada pela classe *SVNRepositoryFactory* no momento da criação do objeto *SVNRepository*.

SVNRepository é a classe que possui todos os métodos de baixo nível da API necessários para realizar operações com o repositório. As revisões enviadas ao repositório são representadas pela classe *SVNLogEntry*. Ela encapsula informações como o número da revisão, a data de quando a revisão foi enviada, o autor da revisão, uma mensagem de log do *commit* e todos os arquivos alterados naquela revisão.

O diagrama de sequência representado na Figura 10 mostra os acontecimentos decorrentes da seleção de um arquivo texto pelo usuário no editor.

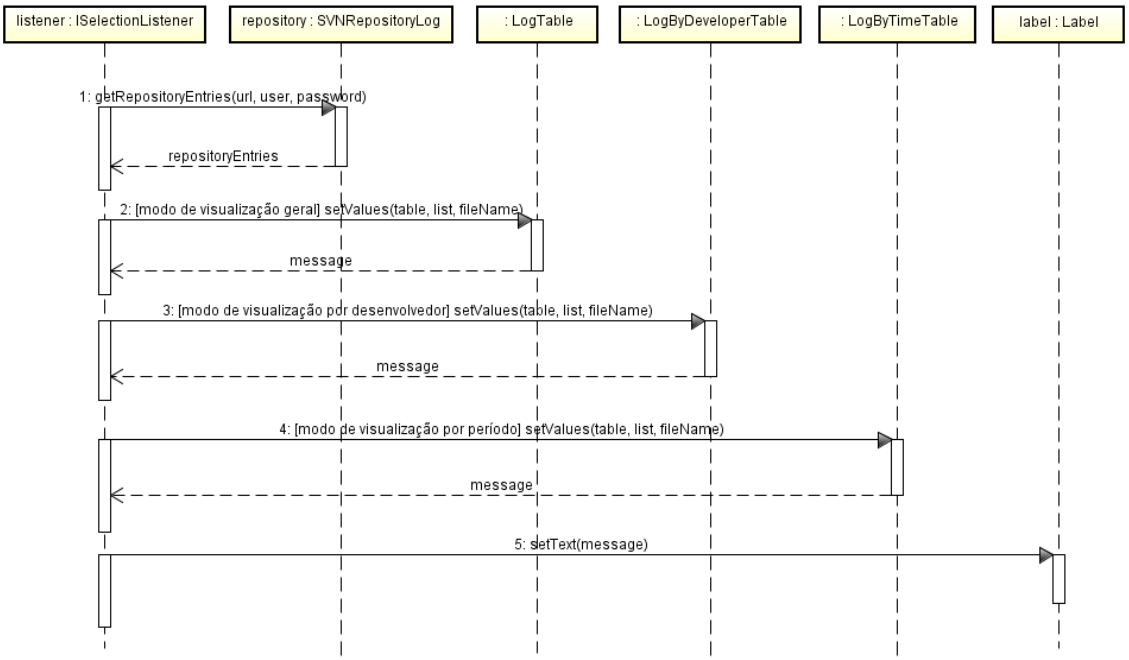

Figura 10 – Diagrama de sequência

O processo se inicia quando o *listener* é comunicado de que houve a seleção de um novo arquivo no editor. Neste momento, ele captura o nome do arquivo em questão e busca no repositório o seu histórico de *commits*. Em seguida, ele envia estes dados para as classes *LogTable*, *LogByDeveloperTable* ou *LogByTimeTable*, dependendo do modo de visualização em que o usuário se encontra, para que as mesmas preencham suas respectivas tabelas na interface com o usuário. Estas classes por sua vez, retornam para o *listener* uma mensagem que será exibida no topo da tabela (última mensagem apresentada no diagrama).

3.3 Linguagem e Ambiente de Programação

A linguagem utilizada para o desenvolvimento do plug-in foi Java, criada em 1995 por James Gosling e sua equipe na empresa Sun Microsystems [ORACLE, 2013]. Diferente de outras linguagens, que são compiladas em código nativo, a linguagem Java é compilada em *bytecodes* e executada nas chamadas máquinas virtuais Java, que transformam as instruções em linguagem de máquina. É essa característica que faz com

que os programas Java sejam independentes de plataforma.

O ambiente de desenvolvimento integrado (IDE) utilizado junto à linguagem Java foi o Eclipse[9]. Ele é um IDE multi-linguagem, com um sistema de extensão por plug-ins. Eclipse é um software livre (licenciado de forma a conceder o direito aos usuários de usar, modificar, estudar e melhorar sua concepção através da disponibilidade de seu código fonte) e de código aberto, desenvolvido pela comunidade. Também é um software multi-plataforma, pois permite a construção de programas simultaneamente em Linux, OS X, Solaris e Windows [IBM, 2005].

3.4 Considerações Finais

Este capítulo tratou dos requisitos técnicas que nortearam o desenvolvimento do plug-in para o ambiente de desenvolvimento de software Eclipse, passando pelo levantamento dos seus requisitos funcionais, sua modelagem e as tecnologias utilizadas na implementação.

O próximo capítulo apresentará o funcionamento do plug-in através de *screenshots* das suas principais telas, além de apresentar um guia para sua instalação no ambiente de desenvolvimento de software.

[9] http://www.eclipse.org/

4 O Plug-in

Neste capítulo apresentaremos o funcionamento e o visual do plug-in. A seção 4.1 traz uma breve explicação de como é feita a instalação do plug-in no Eclipse. Na seção 4.2 são apresentadas as telas do plug-in e suas funcionalidades. Por fim, na seção 4.3 são feitas as considerações finais referentes a este capítulo.

4.1 Instalando o plug-in no Eclipse

Os plug-ins para Eclipse são distribuídos para uso de terceiros na forma de arquivos *.jar*. Este arquivo pode ser facilmente importado para uma instalação do Eclipse para que as funcionalidades do plug-in possam ser utilizadas. O Eclipse possui uma pasta chamada *plugins* na raiz do seu diretório de instalação, conforme mostra a Figura 11.

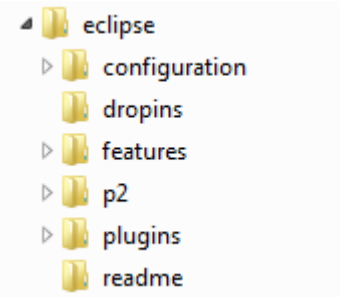

Figura 11 – Estrutura de diretórios do Eclipse

Esta pasta contém todos os plug-ins instalados no Eclipse. O arquivo *.jar* do projeto de plug-in desenvolvido neste trabalho deve ser copiado para esta pasta. Feito isto, basta abrir o Eclipse e o plug-in estará em funcionamento.

4.2 Apresentação do plug-in

Com o plug-in instalado no Eclipse, a primeira ação a ser tomada é exibir a *view* que apresenta o histórico de *commits* do arquivo que está aberto no editor. As figuras 12 e 13 mostram como este processo é feito.

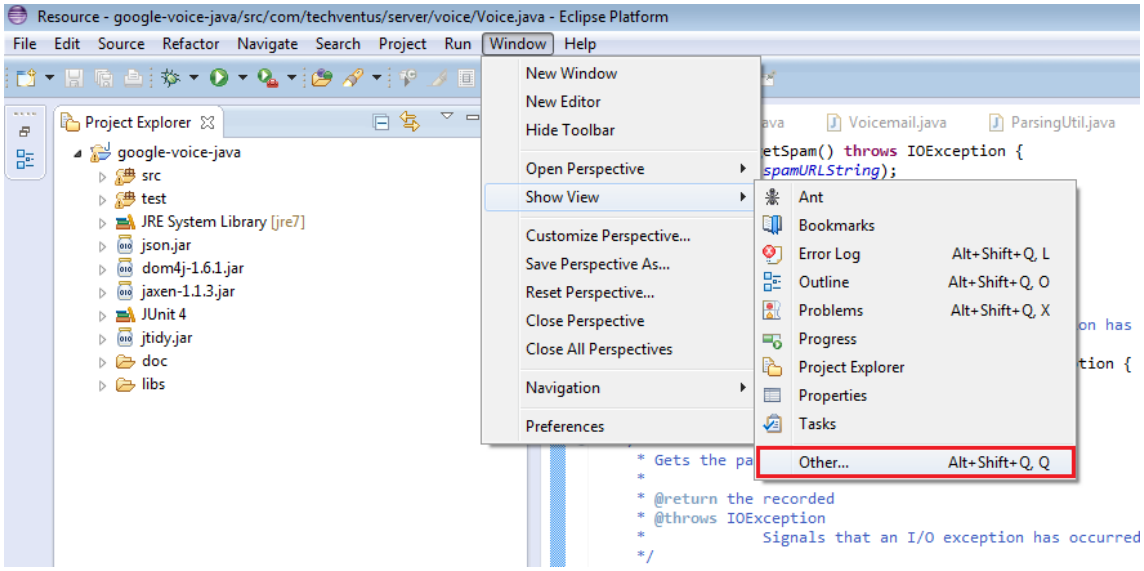

Figura 12 – Exibindo a *view* de histórico de *commits* – Passo 1

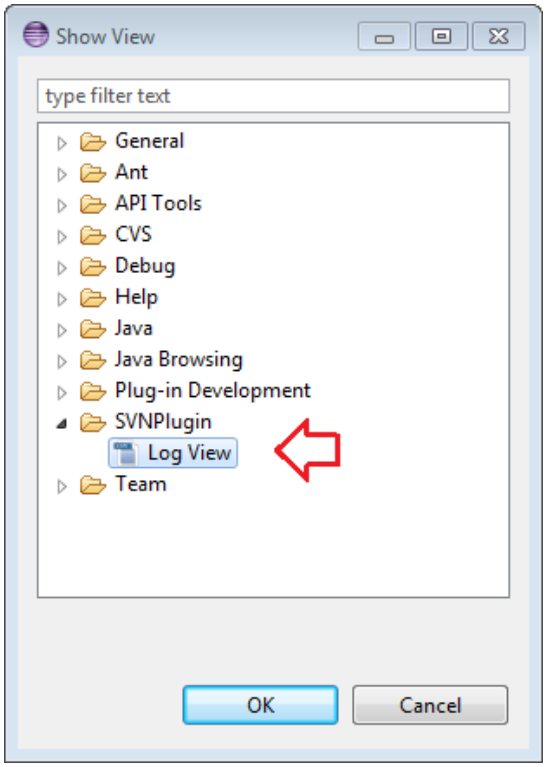

Figura 13 – Exibindo a *view* de histórico de *commits* – Passo 2

Ao selecionar a *view*, ela será incluída no ambiente do Eclipse e exibida conforme mostra a Figura 14.

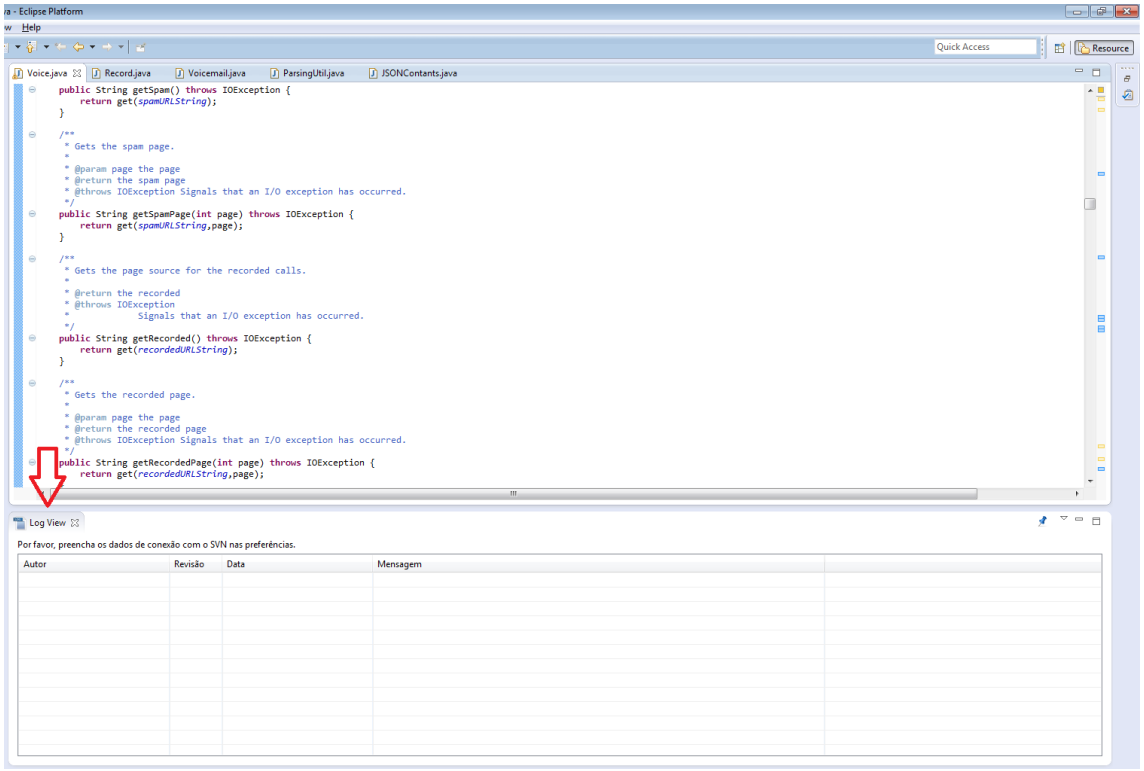

Figura 14 – View de histórico de *commits* sendo exibida

Conforme mostra a mensagem dentro da *view*, para que o histórico de *commits* do arquivo que está aberto no editor seja exibido, as informações de conexão com o repositório do sistema de controle de versões devem ser fornecidas. Essas informações são inseridas na janela de preferências do Eclipse, como ilustram as Figuras 15 e 16. O caminho do SVN deve seguir o padrão mostrado na figura, indo até a pasta raiz do repositório.

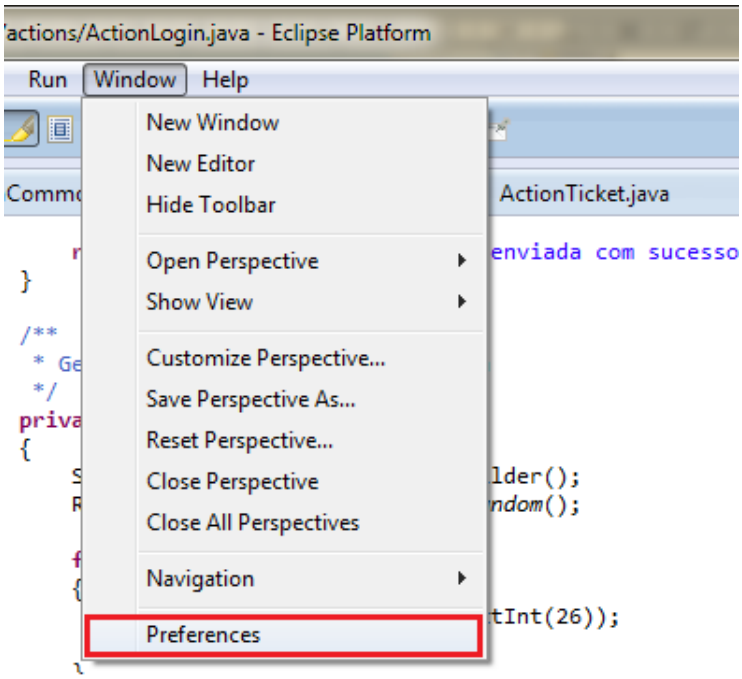

Figura 15 – Inserindo as informações de conexão com o repositório – Passo 1

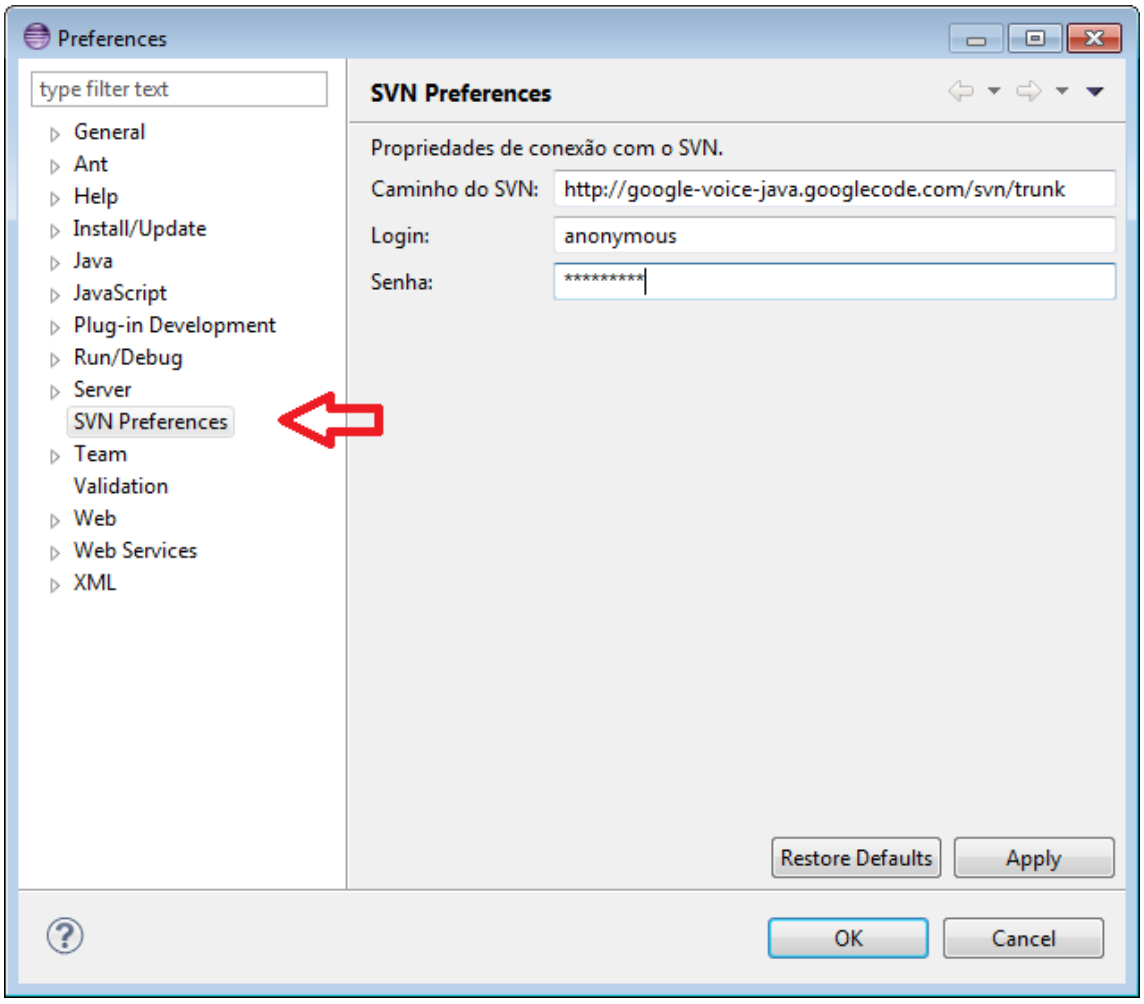

Figura 16 – Inserindo as informações de conexão com o repositório – Passo 2

Com os dados de conexão preenchidos corretamente, a *view* passará a exibir o histórico dos *commits*. Sempre que um novo arquivo for aberto no editor, o histórico do mesmo será exibido automaticamente na *view* do plug-in. Existem três modos de visualização deste log:

- Log de *commits* geral – mostra todos os *commits* feitos no arquivo;
- Log de *commits* por desenvolvedor – mostra a porcentagem de *commits* sobre aquele arquivo realizado por cada desenvolvedor;
- Log de *commits* por período – mostra a porcentagem de *commits* realizados sobre aquele arquivo em determinados períodos de tempo (agregados por meses).

Os botões que permitem alternar entre os modos de visualização ficam localizados no canto superior direito da *view*, conforme mostra a Figura 17.

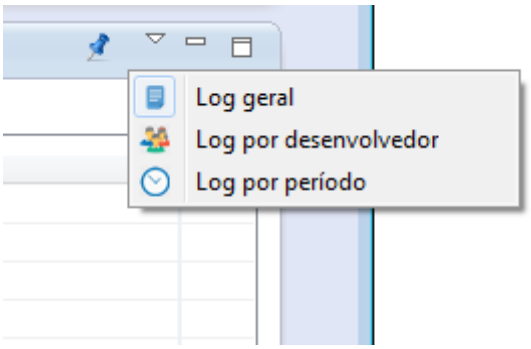

Figura 17 – Alternando entre os modos de visualização

Na Figura 18 temos um exemplo da *view* preenchida no modo de visualização geral. A tabela contendo o histórico de *commits* é referente ao arquivo *Voice.java*, que está aberto no editor. Foi utilizado como exemplo um projeto *open-source* disponível no Google Code[10]. Este projeto permite o controle programático do Google Voice[11] para desenvolvedores Java. Isto inclui fazer chamadas, envio de mensagens SMS,

[10] http://code.google.com/
[11] https://www.google.com/voice

visualização de mensagens de voz, entre outros recursos. O projeto contou com a contribuição de 13 desenvolvedores.

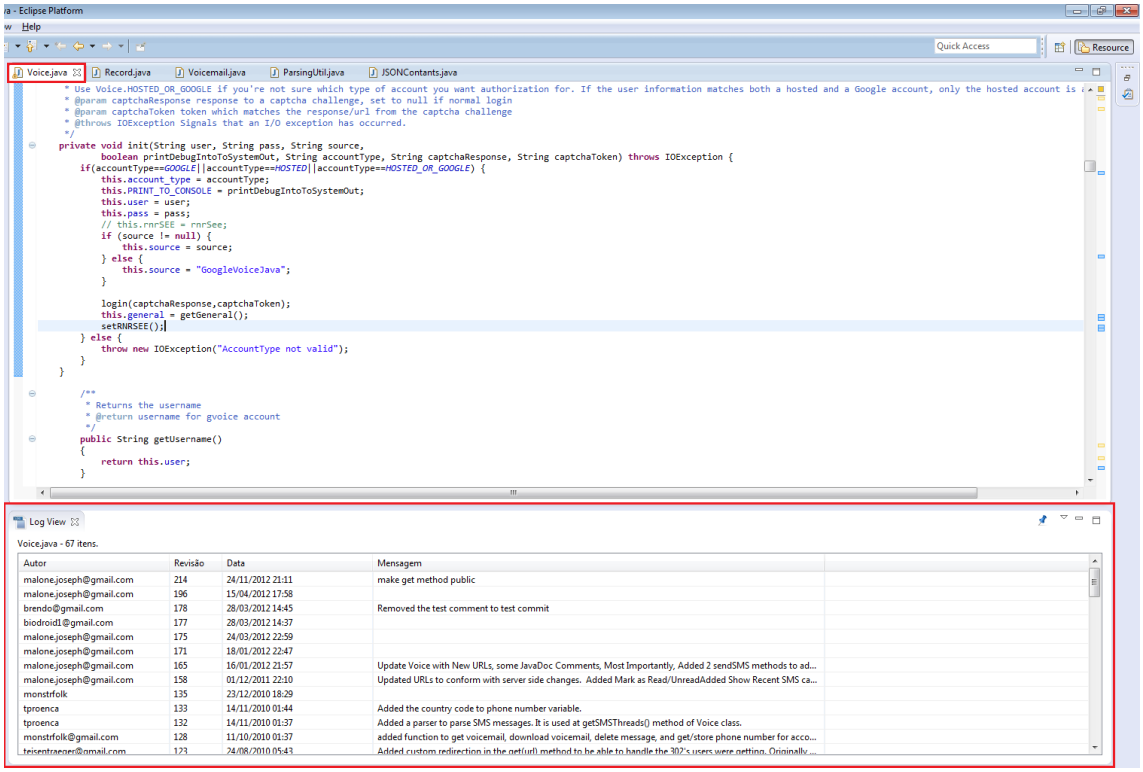

Figura 18 – Histórico de *commits* sendo exibido

O carregamento do histórico leva em torno de dois segundos, considerando um arquivo que tenha sido editado em média 60 vezes. Apesar deste carregamento ser realizado de forma assíncrona, durante estes dois segundos não é possível realizar nenhuma outra ação no Eclipse. Isto ocorre pois a linha de execução principal do Eclipse é a única responsável por atualizar a interface com o usuário. Portanto, o preenchimento da tabela deve aguardar o carregamento dos dados do repositório. A tabela é inicialmente ordenada pelo número da revisão atribuído a cada *commit* em ordem decrescente. Assim, os *commits* mais recentes ficam no topo e os mais antigos no final da tabela. A tabela pode ser reordenada por outros campos clicando no cabeçalho das colunas.

As Figuras 19 e 20 mostram os modos de visualização por desenvolvedor e por

período, respectivamente. Também é possível reordenar a tabela clicando nas colunas que compõem estas duas visualizações.

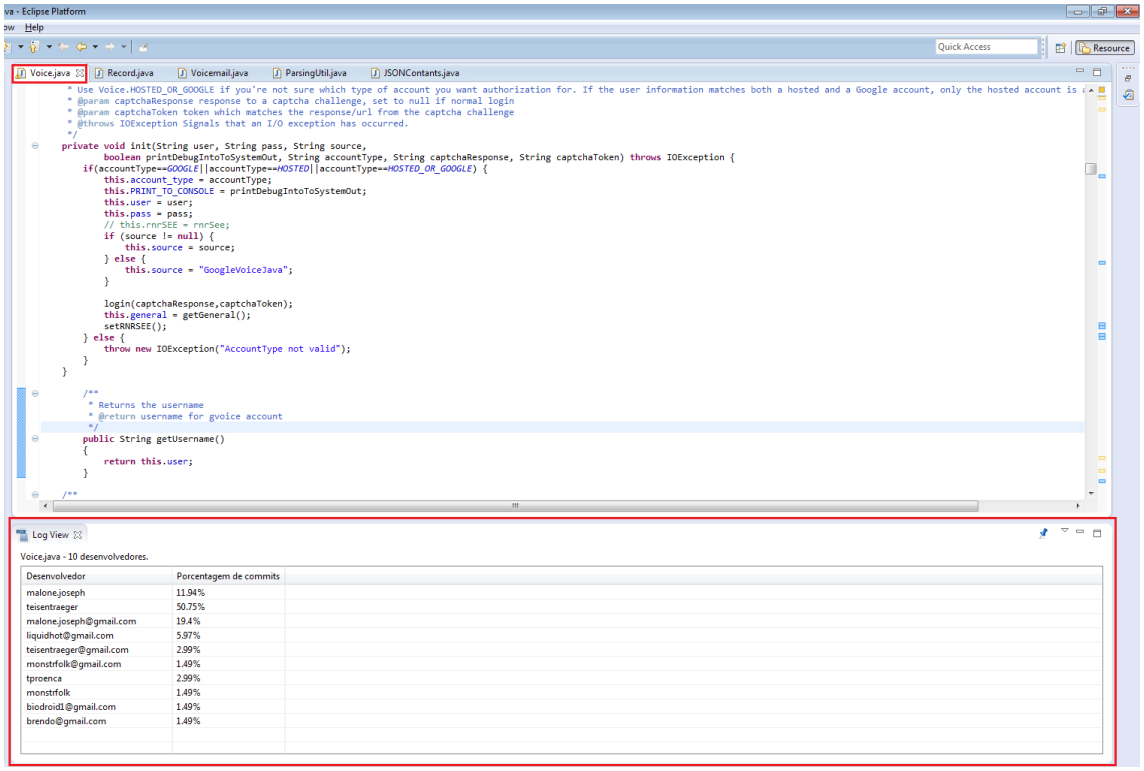

Figura 19 – Porcentagem de *commits* por desenvolvedor

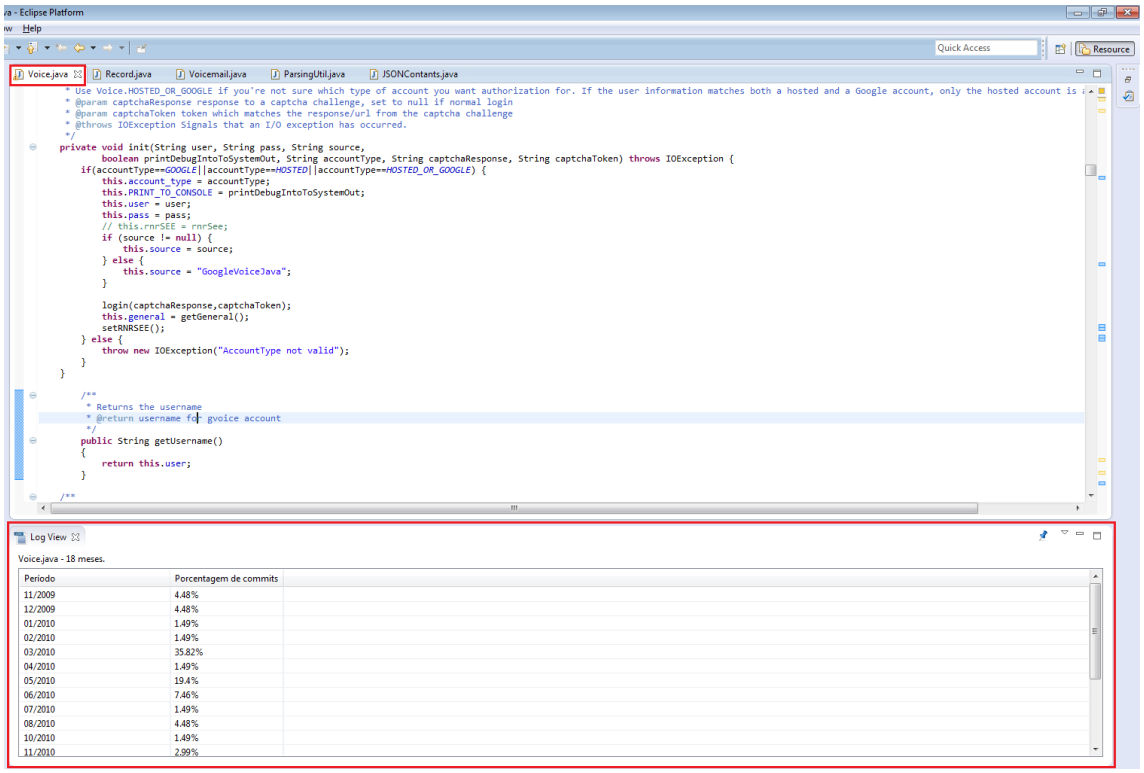

Figura 20 – Porcentagem de *commits* por período

Nas três visualizações o usuário tem a possibilidade de fixar o *log* que está sendo apresentado na *view* naquele momento. Com o *log* fixado, o usuário pode abrir outros arquivos no editor sem que o conteúdo da *view* se altere. A qualquer momento, o botão que trava o *log* pode ser pressionado novamente e o *log* voltará a ser preenchido com o arquivo atualmente selecionado no editor. O botão que executa esta ação se localiza no canto superior direito da *view*, como mostra a Figura 21.

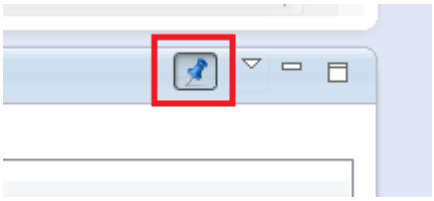

Figura 21 – Botão para fixar/desfixar o log na view

4.3 Considerações Finais

Neste capítulo apresentamos as telas do plug-in desenvolvido no contexto deste trabalho, explicando suas funcionalidades e seus comportamentos. O próximo capitulo apresentará nossas conclusões, limitações do plug-in e possíveis extensões que podem ser realizadas a partir deste ponto.

5 Conclusões

Este capítulo conclui o projeto de desenvolvimento de um plug-in para o ambiente de desenvolvimento de software Eclipse. Nele será apresentado um resumo do que foi observado durante o projeto, as melhorias que poderão ser feitas futuramente e algumas questões que não foram consideradas parte do escopo deste projeto.

5.1 Contribuições

Para dar início a este projeto, foi necessário buscar e entender os problemas enfrentados diariamente pelos desenvolvedores de sistemas. Feita esta busca, pensou-se em uma ferramenta que os auxiliaria na realização do seu trabalho. O desenvolvimento de um plug-in que integre no ambiente de desenvolvimento de software informações sobre o sistema de controle de versões visa auxiliar o desenvolvedor na execução de suas tarefas, de maneira rápida e objetiva. A implementação, junto com o apanhado sobre desenvolvimento de plug-ins relatado neste documento, são as principais contribuições deste trabalho.

5.2 Trabalhos Futuros

O plug-in implementado neste projeto é apenas a fase inicial de uma ferramenta que ainda pode evoluir muito, acrescentando funcionalidades para facilitar o dia-a-dia do trabalho dos desenvolvedores de software.

Entre os possíveis trabalhos futuros para este tema está a possibilidade de realizar *commits* e *updates* dos arquivos diretamente pela interface do Eclipse. Outras possibilidades de melhoria seriam criar uma visualização gráfica do log de *commits* dos arquivos, promover a integração destas informações com ferramentas de trouble ticket (como o BugZilla, JIRA e Trac) e com ferramentas de construção automática de sistemas (como o CruiseControl e o Jenkins) que tem por objetivo realizar de forma automatizada compilações parciais ou totais dos softwares que estão sendo desenvolvidos. Estas melhorias promoveriam um controle melhor aos desenvolvedores sobre o que foi alterado no projeto e otimizaria o seu tempo.

5.3 Limitações do Estudo

Sendo esta a sua primeira versão, o plug-in encontra-se longe de estar completo. Entre as limitações que podem ser apontadas aqui, incluímos: (i) não é possível visualizar versões anteriores dos arquivos; (ii) o plug-in não permite a visualização, de forma reunida, de todos os arquivos que foram alterados em uma única revisão; e (iii) não foi analisado o desempenho do plug-in utilizando um projeto com um número de informações muito grande.

Referências

GALLARDO, D.; "Desenvolvendo Plug-ins do Eclipse", 2002. Disponível na URL http://www.ibm.com/developerworks/br/library/os-ecplug/, acessado em 20 de setembro de 2013.

DIAS, A. F.; "Conceitos Básicos de Controle de Versão de Software", 2011. Disponível na URL http://www.pronus.eng.br/artigos_tutoriais/gerencia_configuracao/conceitos_basicos_controle_versao_centralizado_e_distribuido.php?pagNum=0, acessado em 20 de setembro de 2013.

BOOCH, G; RUMBAUGH, J e JACOBSON, I: "UML: Guia do Usuário", Editora Campus, Rio de Janeiro, 2000.

FOGEL, K.; Controle de Versão; Capítulo 3. Infra-estrutura técnica: http://producingoss.com/pt-br/vc.html, acessado em 13 de agosto de 2013.

SVNKIT, “The only pure Java™ Subversion library in the world!”, 2013. Disponível na URL http://svnkit.com/kb/index.html, acessado em 13 de agosto de 2013.

SVNKIT, "Documentação em formato JAVADOC", 2013; Disponível na URL http://svnkit.com/javadoc, acessado em 13 de agosto de 2013.

ORACLE, 2011. "Interactive Timeline". Disponível na URL: http://www.oracle.com/us/corporate/timeline/index.html, acessado em 13 de agosto de 2013.

ORACLE, 2013. "The History of Java Technology". Disponível na URL: http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html, acessado em 27 de outubro de 2013.

SOMMERVILLE, Ian; Engenharia de Software. 6ª edição, 2003.

IBM, 2005; A Brief History of Eclipse. Disponível (também) em: < http://www.ibm.com/developerworks/rational/library/nov05/cernosek/index.html>, acessado em 31 de outubro de 2013.