Um mecanismo de captura de informações contextuais em um Ambiente de Desenvolvimento Distribuído de Software

Yoji Massago[1], Renato Balancieri[1,] Raqueline Ritter de Moura Penteado[1], Elisa Hatsue Moriya Huzita[1,] Rafael Leonardo Vivian[2]

1. Universidade Estadual de Maringá – UEM, Maringá, Paraná, Brasil
yojimassago@gmail.com, {rbalancieri2, rrmpenteado, ehmhuzita}@uem.br
[2].Instituto Federal do Rio Grande do Sul – IFRS, Sertão, Rio Grande do Sul, Brasil
rafavivian@gmail.com

Abstract. The Distributed Software Development (DSD) has been increasingly adopted by software development companies since it provides support for a better use of their material, human and time resources. However, it presents challenges from geographical distance, time and cultural differences. So, different teams are faced to communication problems which affect directly the quality of the product generated. During software development several artifacts are generated, and they may change during their life. Therefore it is important that such information can be captured and properly dealt in order to be disseminated and used. The purpose of this paper is to present a mechanism to capture contextual information of Java source code files, from a repository for distributed version control. Then, they are stored in a XML file, making they more flexible to be used by other tools. It will also contribute to improve production in DSD.

Keywords: Contextual Information, Data Repository, Distributed Software Development.

1 Introdução

Em busca de vantagens competitivas e cooperativas, diversas organizações adotaram atividades multilocais, multiculturais e globalmente distribuídas, aumentando a produtividade, melhorando a qualidade e reduzindo custos de suas tarefas ([11][5][1]).

O Desenvolvimento Distribuído de Software (DDS) surgiu para tentar resolver vários problemas que, frequentemente, existiam no desenvolvimento de software tradicional, principalmente no que se refere a alocação de recursos e o aproveitamento do tempo. Entretanto, a dispersão geográfica, a distância temporal e as diferenças sociais, elementos inerentes a DDS, ampliaram alguns dos desafios existentes no desenvolvimento de software e, principalmente, adicionaram novas exigências acerca da comunicação entre os indivíduos participantes de um trabalho cooperativo [11].

Um dos desafios é a captura e a disseminação de informações sobre o processo de desenvolvimento entre os integrantes de uma equipe.

Para tentar resolver o problema da captura e disseminação de informação contextual, uma solução seria fazer uma constante verificação dos dados e, então, capturar informações consideradas importantes para serem repassadas aos membros de uma equipe com o intuito de aumentar a compreensão do contexto. Porém, se os dados capturados forem armazenados de forma não padronizada, o uso posterior dos mesmos pode se tornar inviável. Logo, é necessário algum meio para capturar as informações relevantes e armazená-las em um formato que pode ser usado como padrão. Esta padronização pode ser alcançada utilizando-se por exemplo arquivos no formato XML *(Extensible Markup Language)* e XMI (XML *Metadata Interchange*).

Durante o desenvolvimento de um software são criados muitos artefatos, tais como documento de requisitos, diagramas, códigos, relatórios, entre outros. Uma abordagem para apoiar a percepção sobre a criação, a evolução e a manutenção do código fonte, dos diagramas de classes e dos relatórios de erros que contêm justificativas de mudanças efetuadas nestes artefatos, foi proposta em [11]. Nesta abordagem, estes artefatos são armazenados em um repositório de dados central, sendo os relatórios armazenados em um banco de dados, os diagramas em formato XMI, e os códigos fonte em formato texto.

Os arquivos de código fonte, normalmente, estão armazenados em formato tipo texto de acordo com a gramática da linguagem de programação, adotada para o desenvolvimento de uma determinada aplicação. Dependendo da gramática e do tamanho do código, para se obter informações do mesmo pode demandar muito tempo e poder computacional. No caso destas mesmas informações serem usadas mais de uma vez, como nas informações necessárias a todos os membros de uma equipe distribuída, refazer tudo, a cada vez que a informação for necessária, pode ser muito custoso.

Assim, este artigo tem por objetivo apresentar um mecanismo que capture informações a partir de arquivos de código fonte Java armazenados em um repositório de Sistema de Versionamento Distribuído e armazenados em formato XML, para possibilitar maior facilidade de manipulação destas informações por parte de outros sistemas/programas. Este mecanismo será útil para os casos em que existem muitos dados nos arquivos de código fonte Java, dos quais se necessita capturar apenas uma parte das informações contidas para serem utilizadas por outros programas, e, também, serem acessadas e manipuladas facilmente por uma equipe de DDS.

O texto encontra-se dividido em mais quatro seções além da introdução. A seção 2 descreve os conceitos relacionados ao desenvolvimento do mecanismo de captura de informação. A seção 3 mostra como foi desenvolvido do mecanismo: os passos executados, as funcionalidades desenvolvidas. A seção 4 apresenta os resultados obtidos, bem como a avaliação destes. Por último, a seção 5 apresenta as conclusões.

2 Conceitos envolvidos no projeto do mecanismo

Durante muitos anos, o desenvolvimento de software ocorria de forma centralizada, por uma equipe local. Mas, dependendo das empresas e do software a ser

desenvolvido, trabalhar utilizando apenas os recursos locais (humanos e/ou materiais) era difícil e custoso. Assim, buscando melhorar o desenvolvimento, otimizar a alocação de recursos, diminuir o custo, entre outros fatores, surgiu a ideia de DDS.

"*O DDS tem sido caracterizado principalmente pela colaboração e cooperação entre departamentos de organizações e pela criação de grupos de pessoas que trabalham em conjunto, mas estão localizados em cidades ou países diferentes, distantes temporal e fisicamente.*"[8]. Resumidamente, DDS é o desenvolvimento de um software por uma equipe cujos membros estão trabalhando em locais geograficamente dispersos.

"*Para a execução de um trabalho colaborativo por um grupo de pessoas, é importante que os indivíduos compreendam as atividades dos outros para que isso seja relevante para a realização de suas próprias tarefas e, assim, otimizar o andamento dos trabalhos. Percepção ou Awareness, é uma compreensão das atividades dos outros, que oferece um contexto para a própria atividade do indivíduo.*" [4] apud [11].

Contexto é qualquer informação que pode ser usada para caracterizar uma situação de uma entidade. Uma entidade é uma pessoa, um lugar, ou um objeto que é considerado relevante para a interação entre um usuário e uma aplicação, incluindo o próprio usuário e a própria aplicação [3] apud [11].

No desenvolvimento de software de forma colaborativa, faz-se necessário a percepção dos eventos que estão ocorrendo, a fim de melhorar o desempenho do desenvolvimento, seja evitando trabalhos repetidos/duplicados, seja melhorando na compreensão conjunta da equipe, ou através de outros fatores. No DDS, isso se torna mais importante, considerando-se os desafios decorrentes desta abordagem de desenvolvimento (dispersão geográfica, diferença de fuso horário,...).

Para evitar perdas de dados importantes, muitos desenvolvedores de software utilizam algum Sistema de Controle de Versões (SVM) para o armazenamento de seus dados. Um SVM "*... é responsável por manter o histórico de diversas revisões de uma mesma unidade de informação. Ele é comumente utilizado em desenvolvimento de software para gerenciar o código fonte de um projeto*."[2].

Uma das variações dos SVM é o Sistema de Versões Concorrentes (CVS – *Concurrent Version System*), o qual permite o armazenamento dos dados em um repositório local ou remoto. Neste repositório são armazenadas as versões antigas e os logs daqueles que manipularam os arquivos e quando isto foi feito. Como exemplos de CVS podem ser citados o Subversion[1], o Perforce[2], entre outros.

Porém, um CVS comum, principalmente pelo fato de utilizar um repositório central, não possui suporte adequado para a sua utilização no Desenvolvimento Distribuído de Software [6]. Assim surgiram os DCVS *(Distributed Concurrent Versions System)* que fornecem o suporte adequado para que os programadores que estão em locais geograficamente dispersos possam desenvolver software colaborativamente. Estes sistemas utilizam vários repositórios espalhados pela rede, e também possuem um mecanismo com o qual os repositórios filhos possam

1 http://subversion.apache.org/

2 http://www.perforce.com/products/perforce

compartilhar informações entre si de modo paralelo [6]. Alguns dos DCVS existentes atualmente são o Git[3], e o Mercurial[4].

Dentre estes, para o desenvolvimento do mecanismo apresentado neste artigo foi utilizado DCVS Mercurial, que é uma ferramenta multiplataforma que pode ser utilizada em conjunto com algumas IDEs *(Integrated Development Environment)* como NetBeans[5] e Eclipse[6]. Algumas outras características desta ferramenta são: é um programa de linha de comando, possui repositório descentralizado, é escalável, suporta trabalho distribuído e colaborativo, e suporta arquivos texto e binário.

3 Detalhes do Projeto do Mecanismo

Uma abordagem para a percepção de informações contextuais sobre os artefatos de software no ambiente de desenvolvimento distribuído de software chamado DiSEN é proposta em [11]. Em [12] é apresentada a abordagem, da qual faz parte um mecanismo para exibir as relações existentes entre os arquivos de código fonte (em Java) e os diagramas de classe correspondente. Além destas relações foi prevista em [12] a necessidade de oferecer apoio adequado à captura de informações em arquivos de código fonte Java que pudessem proporcionar percepção de contexto aos membros de equipes DDS.

Assim, o mecanismo apresentado no presente artigo faz parte do trabalho apresentado em [12] e tem como foco obter informações contextuais a partir de arquivos de código fonte escritos em Java e armazenados no repositório Mercurial. Para tal, faz se necessário verificar a localização destes códigos, para, posteriormente, realizar uma varredura através destes a fim de conseguir as informações desejadas, além de manipular os métodos/comandos existentes no sistema Mercurial para a manipulação/obtenção dos dados contidos no repositório, tais como versões anteriores, quem executou estas mudanças e datas e horários das modificações.

3.1 Desenvolvimento

As principais funcionalidades do mecanismo ora apresentado são: (i) capturar informações do sistema Mercurial, (ii) verificar os programas Java existentes e, após as verificações, (iii) criar os arquivos XML contendo as informações capturadas.

Assim, o desenvolvimento do mecanismo, obedeceu-se as seguintes etapas:

1. *Captura de informação no Mercurial* - Para isso, foi criado um arquivo de *shell script* que executa os comando do Mercurial e chama o mecanismo criado, passando os dados relevantes como parâmetros de entrada. Ele irá capturar informações sobre os arquivos modificados, a versão dos arquivos, o autor do *commit*, a data da modificação, a mensagem enviada junto ao *commit* e a versão do Mercurial;

3 http://git-scm.com/
4 http://mercurial.selenic.com/
5 http://netbeans.org/
6 http://www.eclipse.org/

2. *Verificação dos arquivos Java existentes* - Para esta etapa, foram criadas funções que fazem a varredura de um diretório, previamente especificado pelo usuário do sistema (no caso, o diretório do repositório Mercurial), bem como os seus subdiretórios, em busca de arquivos de código fonte Java;
3. *Verificação dos arquivos de código fonte Java* - Para a varredura dos arquivos de código fonte Java, foram implementadas funções baseadas em um compilador: mais especificamente, as funcionalidades de análise léxica e sintática, responsáveis por determinar se os arquivos estão de acordo com a gramática específica da linguagem de programação. Simultaneamente, é utilizado a biblioteca Java *jdom-2.0.1*[7], para manipular arquivos XML, a fim de armazenar as informações obtidas;
4. *Execução junto ao commit* - Também foi modificado um dos arquivos do Mercurial (o arquivo *hgrc*), para que execute o *shell script* acima mencionado, toda vez que um *commit* é executado.

3.2 Visão geral do projeto

Para facilitar a compreensão, são apresentados a seguir dois diagramas do projeto do mecanismo. A Fig. 1 exibe os pacotes e as classes existentes, bem como as relações destas classes. Nela pode-se observar que existem três pacotes:

- Pacote *criarxml*, que contém a classe CriacaoArquivoXML, responsável pela criação e armazenamento de dados em arquivos XML;
- Pacote *verificarepositório*, que contêm as classes *main*, Parametros e ManipulacaoDiretorio. A classe Parametros é um objeto que armazena os dados que a classe main recebeu como entrada (os dados obtidos pelo shell script), a fim de serem repassados ao Parser, onde serão armazenados nos arquivos XML. A classe ManipulacaoDiretorio é responsável pela varredura dos diretórios em busca de arquivos .java.
- Pacote *comp*, que contêm as partes do "compilador": Lexer e Parser, além do objeto Token. O Lexer é o analisador léxico, responsável por analisar o arquivo e criar tokens, com os dados necessários. No Parser ou analisador sintático, que existe a "varredura" das informações contidas nos arquivos Java, bem como o seu armazenamento em formato XML, utilizando-se a classe CriacaoArquivoXML.

A Fig. 2 mostra um diagrama de sequência do funcionamento geral do mecanismo proposto. Nela pode-se observar que a classe *main* (VerificacaoRepositorio) inicializa uma nova instância da classe ManipulacaoDiretorio e inicializa-o por meio do método ImprimirConteudo. Nesta classe ManipulacaoDiretorio existe um *loop* que fica ativo até que não exista mais arquivo .java a ser verificado. Dentro deste *loop*, sempre que um arquivo Java é encontrado, é criado um novo Parser e inicializa-o. Este Parser utiliza o Lexer para capturar os dados (*Tokens*) dos arquivos e armazena os dados em XML, com a utilização da classe CriacaoArquivoXML.

7 http://www.jdom.org/

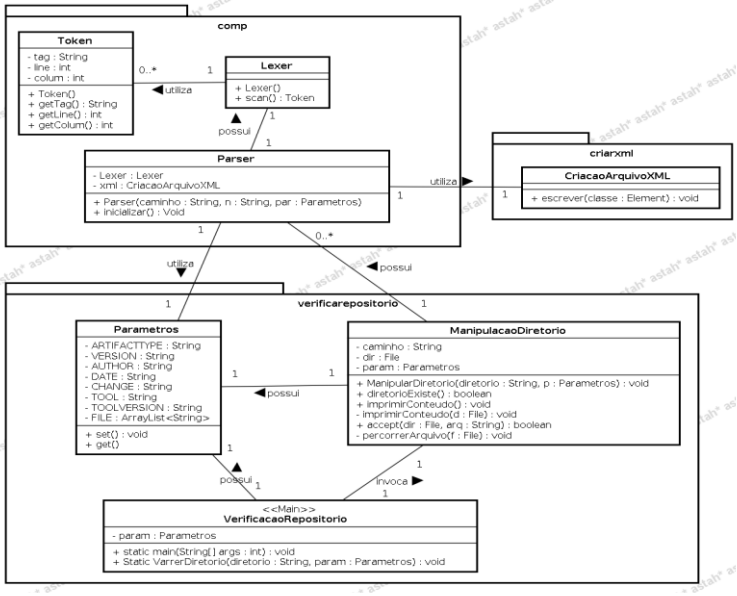

Fig. 1 Diagrama de pacotes do mecanismo

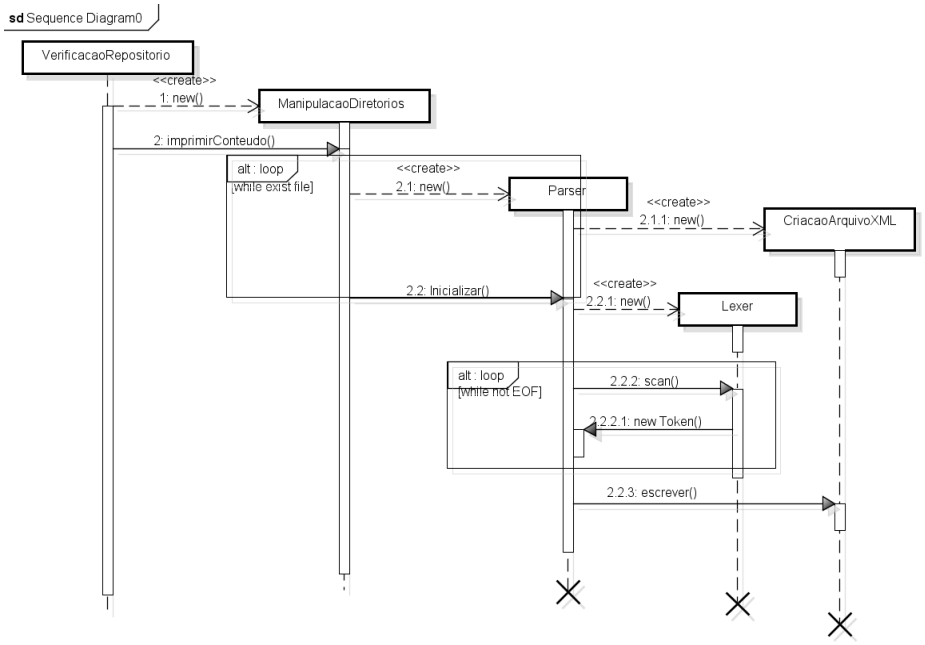

Fig. 2 Diagrama de sequência mostrando o fluxo de execução do mecanismo

4 Avaliação dos resultados

Para a validação do mecanismo desenvolvido, foram definidos casos de teste, para mostrar exemplos de uso deste. Para tanto, executa-se o mecanismo criado, com diversas entradas, e observam-se as saídas a fim de verificar se estas estão de acordo com o esperado. Este método de avaliação foi escolhido para verificar a corretude do mecanismo.

Para os testes, foram escolhidos vários casos, entre eles: um código simples, um código mais complexo, comparação dos XML após algumas modificações. Destes, será apresentado o caso de um código simples, sendo os outros possíveis de serem visualizados em [7].

Para exemplificar a geração de um arquivo XML a partir de uma classe Java, a seguir será mostrado o caso de um código simples. A Fig. 3 mostra o arquivo texto de um código fonte Java de um arquivo chamado *PersistenceAcessoPolicy.java*. Conforme pode ser observado na linha 5, este arquivo está contido no pacote *disen.supernode*. Pelas linhas 7 a 11, pode-se observar que utiliza cinco *imports*, dos quais três são do próprio DiSEN, e os outros dois são bibliotecas do Java. Outros dois pontos a serem considerados são: este arquivo é uma classe (linha 17), com o nome *PersistenceAcessoPolicy* e ela possui um único método identificado como *validarUsuario* (linha 20).

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  package disen.supernode;                                          <- Pacote
6
7  import br.uem.din.disen.core.comunicacao.acesso.AcessoPolicy;
8  import br.uem.din.disen.core.comunicacao.acesso.TipoAcesso;
9  import disen.recurso.usuario.bean.Usuario;                        Importações
10 import java.util.logging.Level;
11 import java.util.logging.Logger;
12
13 /**
14  *
15  * @author will
16  */
17 public class PersistenceAcessoPolicy implements AcessoPolicy {    <- Classe
18
19     @Override
20     public TipoAcesso validarUsuario(String login, String senha) { <- Método
21         try {
22             Usuario u = new Usuario();
23             u = u.getUsuarioByLogin(login);
24             if(u!=null) {
25                 if(u.getSenha().equals(senha)) {
26                     return TipoAcesso.NODE;
27                 }
28             }
29         } catch (Exception ex) {
30             Logger.getLogger(PersistenceAcessoPolicy.class.getName()).log(Level.SEVERE, null, ex);
31         }
32         return TipoAcesso.NEGADO;
33     }
34 }
```

Fig. 3 Arquivo de código fonte java

A Fig. 4 mostra parte do arquivo XML gerado pelo mecanismo, que é um arquivo de nome *PersistenceAcessoPolicy.xml*, que possui os campos *artifacttype*, *version*,

author, *date*, *change*, *tool*, *toolversion*, *package*, *Imports* e *type*. A primeira *tag* se refere ao tipo de artefato armazenado (no caso um arquivo de código fonte). Os seis atributos seguintes são referentes aos dados obtidos do Mercurial: i) versão do artefato, ii) autor do *commit*, iii) a data da execução do *commit*, iv) a mensagem de *commit*, v) o nome do repositório e vi) a versão deste repositório. Abaixo destas *tags*, existem as *tags package*, *Imports* e *type*. Destes, a primeira se refere ao Pacote no qual o arquivo Java está inserido; a segunda é referente aos *imports* utilizados e, finalmente, o *type* (no caso, *classe*), no qual os dados da classe estão inseridos.

```
- <PersistenceAcessoPolicy>
    <artifacttype>SourceCode</artifacttype>
    <version>51</version>
    <author>Yoji Massago <massagoy@gmail.com></author>
    <date>2012-11-05 20:50 -0200</date>
    <change>teste</change>
    <tool>Mercurial</tool>
    <toolversion>Mercurial Distributed SCM (version 2.0.2)</toolversion>
    <package>disen.supernode</package>
  - <Imports>
    - <import1>
        br.uem.din.disen.core.comunicacao.acesso.AcessoPolicy
      </import1>
    - <import2>
        br.uem.din.disen.core.comunicacao.acesso.TipoAcesso
      </import2>
      <import3>disen.recurso.usuario.bean.Usuario</import3>
      <import4>java.util.logging.Level</import4>
      <import5>java.util.logging.Logger</import5>
    </Imports>
  - <type>
      <type>classe</type>
    - <classe>
        <modifier>public</modifier>
        <NomeClasse>PersistenceAcessoPolicy</NomeClasse>
        <implements>AcessoPolicy</implements>
      - <method>
          <modifier>public</modifier>
          <retorno>TipoAcesso</retorno>
          <nome>validarUsuario</nome>
        - <parametros>
          - <parametro>
              <tipo>String</tipo>
              <identificador>login</identificador>
            </parametro>
          - <parametro>
              <tipo>String</tipo>
              <identificador>senha</identificador>
            </parametro>
          </parametros>
```

Fig. 4 Parte do XML gerado para o arquivo PersistenceAcessoPolicy.java

Como resultado dos testes foi verificado que os arquivos que foram testados e verificados (o que inclui uma versão antiga do DiSEN, além do próprio mecanismo),

retornam os resultados esperados, capturando as informações e armazenando corretamente em arquivos XML.

Devido ao grande número de arquivos gerados ao utilizar este mecanismo nos arquivos de código fonte Java do DiSEN, não foram verificados todos os arquivos, um por um, para garantir que todos foram varridos e verificados de forma correta. Todavia, aqueles que foram verificados estavam de acordo com o esperado, da mesma forma que os exemplos mostrados anteriormente.

5 Conclusão

Este artigo apresenta um mecanismo para captura de informações contextuais a partir de um repositório de controle de versão, mais especificamente do Mercurial. Para tal, realizou-se, inicialmente, uma pesquisa para entender os conceitos relevantes, a serem utilizados, e a partir de tais informações, optou-se pelo uso dos analisadores léxicos e sintáticos de um compilador.

O analisador léxico-sintático é um mecanismo para verificar se um arquivo de código fonte possui a sintaxe correta. Também, existem vários comandos no Mercurial, que podem ser utilizados, como no caso deste mecanismo, em conjunto, por meio de um *script*, para obter informações, solicitar alguma alteração, entre outros tipos de comandos.

Além dos testes apresentados na seção 4 deste artigo foram também realizados outros que por limitações de espaço não estão aqui ilustrados. Detalhes de tais testes podem ser encontrados em [7].

Ao final da implementação, e tendo como base os testes realizados, verificou-se que este mecanismo consegue obter as informações necessárias para o nosso contexto. No caso desta versão do mecanismo, adotou-se a estratégia de capturar a maioria das informações do código fonte e colocá-las nos arquivos XML. Isso foi feito pelo simples fato de que, como se utiliza os analisadores, será percorrido o arquivo inteiro e, consequentemente, obter uma pequena parte ou uma grande parte do código não faria muita diferença, além de que, no dado momento, não se sabe quais as informações que outras ferramentas irão necessitar. Futuramente, no caso de não necessitar destes dados "extras", basta retirar as linhas do código referentes à inserção, destes mesmos dados, no arquivo XML.

Assim, espera-se que este mecanismo possa ajudar os desenvolvedores e principalmente outras ferramentas, como o mecanismo desenvolvido por [11], na captura de informações sobre o desenvolvimento dos arquivos de código fonte escritos em Java e, consequentemente, possa ser útil para que outros mecanismos e/ou pessoas consigam obter as informações de contexto necessárias ao bom desempenho das equipes geograficamente distribuídas.

Uma das limitações existentes neste mecanismo está no uso dos analisadores, o que implica no fato deste mecanismo ser específico à gramática escolhida na criação destes analisadores. Assim, no caso de utilizar para outras linguagens ou versões, serão necessários ajustes nestes analisadores.

Portanto, um possível trabalho futuro será modificar os analisadores a fim de poder reconhecer outras gramáticas de outras versões ou até de outras linguagens. Também,

seria interessante a existência de alguma funcionalidade que permitisse o mecanismo identificar, de forma automática, a linguagem utilizada nos arquivos a serem verificados. Com a implementação destes dois casos, o mecanismo poderá fazer a varredura e transformação em XML de arquivos de várias Linguagens de Programação, de forma automática e sem a necessidade do usuário informar, a cada vez, a linguagem utilizada, ou utilizar apenas uma linguagem específica para programar o sistema inteiro. Outro trabalho a ser desenvolvido no futuro será a integração deste mecanismo com o projetado pelo [11][12] com intuito de melhorar o desempenho deste último.

Referencias

1. AUDY, J.: PRIKLADNICKI, R.: Desenvolvimento Distribuído de Software: desenvolvimento de software com equipes distribuídas. Rio de Janeiro: Elsevier, (2008)
2. AUVRAY, S.: Melhores da InfoQ em 07: Sistemas de Controle de Versão Distribuído: Um Guia não tão rápido. InfoQ, (2007)
3. DEY, A. K.: Providing Architectural Support for Building Context-Aware Applications. Ph.D. Thesis, Georgia Institute of Technology, (2000)
4. DOURISH, P., BELLOTTI, V.: Awareness and Coordination in Shared Workspaces. In: The 1992 ACM Conference on Computer-Supported Cooperative Work, *Proceedings*…, pp. 107--114, (1992)
5. HUZITA, E.H.M., SILVA, C.A., WIESE, I.S., TAIT, T.F.C., QUINAIA, M., SCHIAVONE, F.L.: Um conjunto de soluções para apoiar o desenvolvimento distribuído de software. In: II Workshop de Desenvolvimento Distribuído de Software, Campinas, pp. 101--110, (2008)
6. GIT.: Reference Manual. Disponível em: <http://git-scm.com/documentation>. Acesso em: 11/09/2013, (2013)
7. MASSAGO, Y.: Um mecanismo para captura de informações contextuais em um ambiente de desenvolvimento distribuído de software, Departamento de Informática, Universidade Estadual de Maringá, Trabalho de Conclusão de Curso, (2012)
8. PRIKLADNICKI, R., LOPES, L., AUDY, J. L. N., EVARISTO, R.: Desenvolvimento Distribuído de Software: um Modelo de Classificação dos Níveis de Dispersão dos Stakeholders. University of Illinois at Chicago - College of Business Administration 601 S. Morgan Street MC 294, Chicago, IL 60607, United States, (2004)
9. VIEIRA, V.: Gerenciamento de contexto em sistemas colaborativos. Tese de Doutoramento, CIn - Universidade Federal de Pernambuco, Recife - Pernambuco, Monografia de Qualificação, (2006)
10. VIEIRA, V.: CEManTIKA: A Domain-Independent Framework for Designing Context- Sensitive System. Tese de Doutorado, Recife: Cin – UFPE, (2008)
11. VIVIAN, R. L.: Uma Abordagem Context-Awareness sobre Artefatos de Software no Desenvolvimento Distribuído de Software. 29f. Projeto de Dissertação (Mestrado) - Programa de Pós-Graduação em Ciência da Computação, Universidade Estadual de Maringá, Maringá, (2011)
12. VIVIAN, R. L., HUZITA, E. H. M., LEAL., G. C. L.: Supporting distributed software development through context awareness on software artifacts: the DiSEN-CollaborAR approach. In: 28th Annual ACM Symposium on Applied Computing *(SAC '13), Proceedings…*, New York, NY, USA, pp. 765-770, (2013)