A Caminho da Manutenção de Software Baseado em Componentes via Técnicas de Gerência de Configuração de Software[♣]

Leonardo Murta, Cristine Dantas, Hamilton Oliveira,
Luiz Gustavo Lopes, Cláudia Werner

COPPE/UFRJ – Programa de Engenharia de Sistemas e Computação
Universidade Federal do Rio de Janeiro – Caixa Postal 68511
CEP 21945-970 – Rio de Janeiro – RJ – Brasil

{murta, cristine, hamilton, luizgus, werner}@cos.ufrj.br

Resumo

A manutenção é a fase mais cara do ciclo de vida do software. Essa constatação toma dimensões desastrosas quando ferramentas e processos não são capazes de apoiar a execução das atividades de manutenção em um dado paradigma, como, por exemplo, a engenharia de software baseada em componentes. Este artigo apresenta uma abordagem que visa apoiar a manutenção de sistemas baseados em componentes através de técnicas de gerência de configuração de software. Essas técnicas fornecem apoio para a execução das diferentes atividades de manutenção, desde a requisição de manutenção até a implementação e integração das modificações. Além disso, a abordagem proposta nesse artigo faz uso de mineração de dados em repositórios de gerência de configuração para a detecção de relacionamentos não explícitos entre os artefatos utilizados no desenvolvimento baseado em componentes.

Abstract

Maintenance is the most expensive phase of the software life-cycle. This is especially true when there is a lack of supporting tools and processes to perform the maintenance activities in a given paradigm, such as component-based software engineering. This paper presents an approach that aims to assist maintenance of component-based systems by means of software configuration management techniques. These techniques support different activities of software maintenance, from the maintenance request up to implementation and integration. Moreover, some feedback about the relationship of component-based artifacts is provided by applying data mining techniques over configuration management repositories.

Palavras-chave: *Manutenção de Software, Gerência de Configuração de Software, Engenharia de Software Baseada em Componentes.*

1. Introdução

Durante os primeiros anos de pesquisa nas áreas de Gerência de Configuração de Software (GCS) e de Reutilização de Software, o foco era sobre código-fonte, armazenado em arquivos do sistema operacional. Uma grande infra-estrutura foi criada nas últimas décadas para apoiar a evolução de código-fonte utilizando técnicas de GCS. Durante esse mesmo período, a

[♣] Este artigo é uma versão estendida do artigo "Towards Component-based Software Maintenance via Software Configuration Management Techniques", publicado no primeiro Workshop de Manutenção de Software Moderna do Simpósio Brasileiro de Engenharia de Software, Brasília, outubro de 2004.

Reutilização de Software foi aplicada ao nível de código-fonte através de linguagens orientadas a objetos, idiomas, bibliotecas de ligação dinâmica, dentre outras.

Com o passar do tempo, novos paradigmas foram definidos para possibilitar a uniformização dos conceitos usados para representar as entidades do mundo real, passando pelos diversos níveis de abstração abordados pelas atividades de análise, projeto e codificação. Apesar da orientação a objetos atender de certa forma a essa demanda, a granularidade de encapsulamento utilizada é muito fina, pois os objetos não são autocontidos, dificultando a sistematização da substituição ou reutilização dessas partes que constituem os sistemas.

O paradigma de Desenvolvimento Baseado em Componentes (DBC) contorna algumas das deficiências detectadas na orientação a objetos no que tange a reutilização de software. Além disso, o DBC fornece uma estrutura bem definida, composta por componentes, interfaces e conectores, que visa permitir o tratamento da complexidade crescente do desenvolvimento de software.

No contexto de DBC, podem existir, dentro de uma mesma organização, diversas equipes de desenvolvimento de componentes e diversas equipes de desenvolvimento de sistemas que utilizam esses componentes. Além disso, as equipes de desenvolvimento de componentes podem fazer uso de componentes providos por outras equipes de desenvolvimento de componentes, caracterizando equipes híbridas, que atuam concomitantemente nos dois papéis.

Essas diferentes equipes estão intimamente relacionadas através do processo de reutilização, onde as equipes de desenvolvimento de componentes atuam como produtoras e as equipes de desenvolvimento de sistemas que utilizam esses componentes atuam como consumidoras. Nesse processo, as equipes produtoras constroem e mantêm componentes reutilizáveis, enquanto as equipes consumidoras adaptam e reutilizam esses componentes para construir aplicações específicas.

Devido à necessidade intransponível de manter o software após a sua liberação (*release*), a reutilização deve ocorrer de forma controlada, possibilitando que a evolução dos componentes reutilizáveis não dificulte o trabalho já complexo de desenvolvimento de componentes e de desenvolvimento com componentes.

O problema em questão está relacionado com a complexidade da evolução de artefatos reutilizáveis. Essa complexidade ocorre em virtude da necessidade freqüente de adaptar o artefato antes da sua efetiva reutilização. Contudo, as várias instâncias adaptadas de um artefato são profundamente semelhantes e, possivelmente, os mesmos defeitos ou necessidades de melhorias acontecerão em todas essas instâncias. O retrabalho de manutenção pode ser evitado se forem definidas técnicas de GCS para apoiar a evolução desses artefatos de forma controlada.

Essas técnicas podem apoiar tanto a etapa de desenvolvimento quanto a etapa de manutenção. Durante a evolução do software, os sistemas de GCS são fundamentais para prover controle sobre os artefatos modificados por diferentes desenvolvedores. Além disso, esses sistemas permitem um acompanhamento minucioso do andamento das tarefas de manutenção, possibilitando que diversas métricas sejam coletadas e analisadas.

Contudo, os sistemas de GCS atuais não são capazes de lidar com os artefatos complexos utilizados no DBC (Sowrirajan e Hoek, 2003; Murta, 2004). Desta forma, este artigo apresenta uma abordagem para a manutenção de software baseado em componentes que integra as funcionalidades de GCS ao ambiente de DBC. Mais especificamente, este artigo apresenta soluções inovadoras que envolvem: (1) a manutenção de um mapa de reutilização que guarda informações sobre componentes reutilizados e seus contratos de reuso, (2) o versionamento de modelos UML em granularidade fina com definição flexível do grão de versionamento, e (3) a mineração de rastros de modificação em elementos de modelo UML, possibilitando a detecção de dependências implícitas entre esses elementos.

O restante desse artigo está organizado em cinco seções. Na Seção 2 são apresentados os conceitos básicos da área de GCS. A Seção 3 descreve a abordagem proposta para manutenção de software baseado em componentes. A Seção 4 apresenta um exemplo de utilização da abordagem. A Seção 5 apresenta outros trabalhos relacionados com o tema. Finalmente, a Seção 6 apresenta as considerações finais e descreve os trabalhos futuros.

2. Gerência de Configuração de Software

A Gerência de Configuração (GC) surgiu nos anos 50 devido à necessidade da indústria aeroespacial norte-americana em controlar as modificações na documentação referente à produção de aviões de guerra e naves espaciais (Leon, 2000; Estublier, Leblang *et al.*, 2002; Hass, 2003). Posteriormente, nos anos 60 e 70, a GC passou a abranger artefatos de software, indo além dos artefatos de hardware já estabelecidos, e desencadeando o surgimento da GCS (Christensen e Thayer, 2002).

Apesar do surgimento da GCS nos anos 70, o seu foco era muito restrito às aplicações militares e aeroespaciais, e somente no final dos anos 80, com o surgimento do padrão IEEE Std 1042 (Ieee, 1987), e em meados dos anos 90, com o surgimento da norma ISO 10007 (Iso, 1995), a GCS foi finalmente assimilada no processo de desenvolvimento de software de organizações não militares (Leon, 2000).

Como em toda nova área de pesquisa, existem diversas definições para GCS. Contudo, a definição mais aceita e utilizada caracteriza a GCS como "uma disciplina que visa identificar e documentar as características de Itens de Configuração (IC), controlar as suas alterações, armazenar e relatar as modificações aos interessados e garantir que foi feito o que deveria ter sido feito" (Ieee, 1990). O termo IC representa a agregação de hardware, software ou ambos, tratada pela GCS como um elemento único. Desta forma, a GCS não se propõe a definir quando e como devem ser executadas as modificações nos artefatos de software, papel este reservado ao próprio processo de desenvolvimento de software. A sua atuação ocorre como processo auxiliar de controle e acompanhamento dessas atividades.

A GCS pode ser tratada sob diferentes perspectivas em função do papel exercido pelo participante do processo de desenvolvimento de software (Asklund e Bendix, 2002). Na perspectiva gerencial, a GCS é dividida em quatro funções, que são (Ieee, 1990; Iso, 1995): identificação da configuração, controle da configuração, relato da situação da configuração e auditoria da configuração.

A função de *identificação da configuração* tem por objetivo possibilitar: (1) a seleção dos ICs que são os elementos passíveis de GCS; (2) a definição do esquema de nomes e números, que possibilite a identificação inequívoca dos ICs no grafo de versões e variantes; e (3) a descrição dos ICs, tanto física quanto funcionalmente.

A função de *controle da configuração* é designada para o acompanhamento da evolução dos ICs selecionados e descritos pela função de identificação. Para que os ICs possam evoluir de forma controlada, esta função estabelece as seguintes atividades: (1) requisição de modificação, iniciando um ciclo da função de controle dado um pedido de manutenção, que pode ser corretiva, evolutiva, adaptativa ou preventiva (Pressman, 1997); (2) classificação da modificação, que estabelece a prioridade do pedido em relação aos demais pedidos efetuados anteriormente; (3) análise da modificação, que visa relatar os impactos em esforço, cronograma e custo e definir uma proposta de implementação da manutenção; (4) avaliação da modificação pelo Comitê de Controle da Configuração (CCC), que estabelece se o pedido será implementado, rejeitado ou postergado, em função do laudo fornecido pela análise da modificação; (5) implementação da modificação, utilizando processos formais de *check-out* (requisição, aprovação e cópia de ICs do repositório para o espaço de trabalho do desenvolvedor (Leon, 2000)) e *check-in* (revisão, aprovação e cópia de ICs do espaço de trabalho do desenvolvedor para o repositório (Leon, 2000)), caso o pedido tenha sido aprovado pela avaliação da modificação; (6) verificação da modificação, aplicando uma bateria de testes de sistema e validando com a

proposta de implementação levantada na análise da modificação; e (7) geração da configuração de referência (*baseline*), que pode ou não ser liberada para o cliente em função da sua importância e questões de marketing associadas.

A função de *relato da situação da configuração* visa: (1) armazenar as informações geradas pelas demais funções; e (2) permitir o acesso a essas informações em função de necessidades específicas. Essas necessidades específicas abrangem o uso de métricas para a melhoria do processo, a estimativa de custos futuros e a geração de relatórios gerenciais.

A função de *auditoria da configuração* ocorre quando a configuração de referência, gerada na função de controle da configuração, é selecionada para ser liberada para o cliente. Suas atividades compreendem: (1) verificação funcional da configuração de referência, através da revisão dos planos, dados, metodologia e resultados dos testes, assegurando que a mesma cumpra corretamente o que foi especificado; e (2) verificação física da configuração de referência, com o objetivo de certificar que a mesma é completa em relação ao que foi acertado em cláusulas contratuais.

Contudo, sob a perspectiva de desenvolvimento, a GCS é dividida em três sistemas principais, que são: controle de modificações, controle de versões e controle de construções (*building*) e liberações.

O sistema de *controle de modificações* é encarregado de executar a função de controle da configuração de forma sistemática, armazenando todas as informações geradas durante o andamento das requisições de modificação e relatando essas informações aos participantes interessados e autorizados, assim como estabelecido pela função de relato da situação da configuração.

O sistema de *controle de versões* permite que os ICs sejam identificados, segundo estabelecido pela função de identificação da configuração, e que eles evoluam de forma distribuída e concorrente, porém disciplinada. Essa característica é necessária para que diversas requisições de modificação efetuadas através da função de controle da configuração possam ser tratadas em paralelo, sem corromper o sistema de GCS como um todo.

O sistema de *controle de construções e liberações* automatiza o complexo processo de transformação dos diversos artefatos de software que compõem um projeto no sistema executável propriamente dito, de forma aderente aos processos, normas, procedimentos, políticas e padrões definidos para o projeto. Além disso, esse sistema estrutura as configurações de referência selecionadas para liberação, conforme necessário para a execução da função de auditoria da configuração.

As técnicas de GCS são comumente utilizadas para controlar a evolução de sistemas de software, fornecendo controle de versões e apoiando atividades relacionadas à gerência da mudança. Com sistemas de GCS, engenheiros de software são capazes de construir produtos de software consistentes e se comunicar e coordenar notificando uns aos outros sobre tarefas requeridas e finalizadas.

3. Manutenção de Software Baseado em Componentes via GCS

Em um cenário complexo de evolução de componentes, os componentes são, usualmente, adaptados antes de serem reutilizados. Apesar de a GCS apoiar os paradigmas convencionais de desenvolvimento de software, é reconhecido na literatura que ela não atente as demandas específicas do DBC (Sowrirajan e Hoek, 2003). Como os sistemas atuais de GCS não estão preparados para lidar com os artefatos complexos utilizados no DBC, é necessário adaptá-los para atender a esse novo cenário. Essa nova geração de sistemas de GCS deve ajudar na gerencia da evolução consistente de componentes, integrando funcionalidades tradicionais de GCS no DBC.

As próximas subseções apresentam os três tópicos principais relacionados à abordagem proposta: automação do processo de manutenção, evolução controlada de artefatos complexos e detecção de rastros de modificação entre artefatos.

3.1. Automação do processo de manutenção

O DBC difere de outras abordagens de engenharia de software em vários aspectos (Larsson, 2000). Primeiramente, o ciclo de vida no DBC possui atividades diferentes, tais como: procurar componentes, selecionar aqueles que satisfazem os requisitos, adaptá-los quando necessário, instalá-los em uma infra-estrutura de componentes e substituir aqueles que apresentam defeito.

Além disso, há uma diferença com relação aos atores envolvidos nestas abordagens. Na engenharia de software convencional, existem equipes de engenharia e usuários finais, enquanto que no DBC há equipes produtoras de componentes, que desenvolvem componentes, equipes consumidoras de componentes, que desenvolvem software reutilizando componentes, equipes híbridas, que atuam tanto como produtoras quanto como consumidoras de componentes, e usuários finais.

Assim, as modificações sobre componentes demandam algumas adequações aos processos de GCS para que eles se adaptem a este novo cenário. Porém, estes processos adaptados encontram-se em um estágio imaturo, e as principais normas de GCS definidas atualmente (Ieee, 1987; Iso, 1995; Ieee, 1998) não cobrem as questões específicas do DBC. Dessa forma, os Sistemas de Controle de Modificações (do inglês, CCS – *Change Control System*) devem ser flexíveis o suficiente para permitir a customização dos processos, incluindo processos em execução. Provendo esta característica, torna-se possível utilizar a experiência adquirida com a execução dos processos para melhorá-los, mesmo em tempo de execução, sem danificá-los e sem perder informações.

Além disso, novas informações precisam ser coletadas nos processos de GCS para DBC, como, por exemplo, quem são os produtores dos componentes e como localizá-los. Porém, não há consenso com relação às informações que devem ser coletadas. Por isso, um CCS deve permitir a definição de quais informações serão coletadas nos processos de GCS para DBC.

Para resolver os referidos problemas, propomos a definição e implementação de um sistema denominado Odyssey-CCS (Murta, 2004), que permite a customização dos processos e das informações a serem coletadas. A solução define processos primitivos e compostos. Um processo composto está associado a uma definição de processo, que descreve seus subprocessos e seus relacionamentos. Por exemplo, o processo de controle de modificações seria modelado como um processo composto. Processos primitivos representam atividades de GCS, e permitem a interação dos usuários finais com o sistema. Para a definição dos processos, utilizamos o metamodelo SPEM (*Software Process Engineering Metamodel*) (Omg, 2005), próprio para a modelagem de processos de engenharia de software.

Ao modelar um processo, o gerente de configuração associa processos primitivos a formulários, que podem ser customizados, permitindo a definição das informações a serem requisitadas e quais tipos de campos devem ser utilizados para a coleta de cada tipo de informação. Atualmente, os formulários modelados utilizados na execução dos processos são criados em HTML (W3c, 1999), mas está sendo considerada a utilização de outras abordagens, como a especificação de formulários para Web XForms (W3c, 2003), o *framework* CForms (Moczar e Aston, 2002) e a máquina de modelos (*template engine*) Velocity (Cole e Gradecki, 2003).

Cada ciclo completo de um processo gera um novo objeto de modificação, que agrega um conjunto de documentos contendo as informações inseridas pelos usuários finais. Portanto, eles agregam todas as informações coletadas nas atividades de GCS. Estas modificações podem ser agrupadas em um conjunto de modificações (*change set*), o que facilita a troca de informações entre as equipes produtoras e consumidoras de componentes.

Outras questões também precisam ser tratadas com relação ao controle de modificações para DBC, como o problema da cadeia de responsabilidades de manutenção. Vamos supor o cenário onde os componentes são construídos reutilizando-se outros componentes, possivel-

mente criados por equipes diferentes. Neste caso, uma equipe pode receber uma requisição de modificação e detectar que a modificação afeta um componente reutilizado, mas produzido por terceiros. Portanto, a equipe consumidora deverá encaminhar esta requisição para a equipe produtora.

Para isso, é necessário conhecer os detalhes do contrato de aquisição deste componente e os dados da equipe produtora. Caso o tipo de modificação não esteja garantido em contrato, a equipe consumidora deverá negociar sua manutenção com a equipe produtora. A equipe consumidora ainda possui as alternativas de realizar a modificação ela própria, caso possua o código-fonte do componente, e de procurar componentes equivalentes de outros fornecedores. Uma terceira opção, para o caso de componente sem código-fonte disponível, seria a utilização de invólucros (*wrappers*), permitindo a adaptação das interfaces de forma externa ao componente. Contudo, essa opção pode ser inviável em algumas situações, especialmente referentes a manutenções corretivas. Por exemplo, caso um componente para cálculo de PI com grande precisão estivesse com defeito exatamente na rotina principal de cálculo, seria muito complexo ou até impossível adaptar essa rotina de forma externa, sem acesso ao código-fonte. Por outro lado, para um componente de conversão monetária, no caso de uma manutenção adaptativa devido à desvalorização da moeda, seria viável adicionar externamente o código responsável por adequar os resultados providos pelo componente à nova realidade.

A situação se complica quando há reutilização de componentes em vários níveis, por exemplo, quando um componente A reutiliza um componente B, que reutiliza um componente C, e a requisição de modificação recebida pela equipe responsável pelo componente A implica em modificações no componente C. Assim, a equipe responsável pelo componente A encaminhará a requisição de modificação para a equipe responsável pelo componente B, que, por sua vez, encaminhará a requisição para a equipe responsável pelo componente C.

Alem disso, ao modificar um componente, a equipe produtora precisa informar aos consumidores do componente que está disponível uma nova versão. Portanto, ela precisa manter as informações dos consumidores dos seus componentes, bem como os dados dos seus contratos de aquisição.

A solução proposta engloba o desenvolvimento de um "mapa de reutilização", que mantém as informações sobre a composição das versões dos componentes e sobre seus produtores e consumidores, assim como dados contratuais, facilitando o processo de manutenção no paradigma de DBC. Esse mapa de reutilização permite o controle sobre as responsabilidades dos produtores e consumidores, evitando que liberações incompatíveis substituam componentes existentes. Por exemplo, evoluções no componente A de uma determinada liberação não podem invalidar os contratos existentes para essa liberação. Desta forma, todas essas evoluções devem implicar na geração de uma nova liberação, com o possível estabelecimento de novos contratos. Somente manutenções corretivas, que preservam as condições contratuais existentes, poderiam ser executadas sobre a liberação corrente do componente A na forma de revisões.

3.2. Evolução controlada de artefatos complexos

Os Sistemas de Controle de Versões (do inglês, VCS – *Version Control System*) combinam procedimentos e ferramentas para gerenciar as diferentes versões de artefatos que são criados e modificados durante o ciclo de vida do software (Pressman, 1997). O objetivo principal do controle de versões é auxiliar nas funções de identificação e controle da configuração. Os VCSs atuais utilizam um modelo de dados simples, baseado em sistemas de arquivos. Este modelo de dados não é apropriado para ambientes de modelagem que utilizam estruturas de dados complexas, como, por exemplo, componentes.

Dois tipos de artefatos podem ser identificados em GCS: IC básico e IC composto (Pressman, 1997). Um IC composto é uma coleção de ICs básicos e ICs compostos. Um componente é essencialmente um IC composto formado por outros ICs, como classes, casos de

uso, pacotes, código-fonte e código-binário. Este relacionamento entre um IC composto e as suas partes é denominado neste artigo como composição. Por outro lado, quando um componente usa os serviços fornecidos por um outro componente, surge um relacionamento de dependência. Por exemplo, o componente "Engenharia Reversa", na versão 2, usando os serviços fornecidos pelo componente "Parser", na versão 5, constitui um relacionamento de dependência. Qualquer modificação nos elementos que descrevem o projeto interno do componente "Parser" (e.g.: modelos, código, etc.) demanda necessariamente a geração de uma nova versão deste componente. Contudo, esta nova versão de "Parser" pode ou não motivar uma nova versão do componente "Engenharia Reversa".

Como discutido anteriormente, o projeto interno de um componente é descrito através de diferentes tipos de artefatos. No entanto, somente o código-fonte pode ter sua evolução controlada pelos VCSs atuais de modo satisfatório. Esse problema ocorre devido ao grão de versionamento (GV) adotado por esses VCSs, conseqüência do uso de um modelo de dados simples, baseado em sistema de arquivos. O termo GV representa os artefatos que serão sujeitos ao controle de versões. Por exemplo, o GV utilizado nos VCSs atuais é arquivo do sistema operacional. Neste contexto, um modelo UML (Omg, 2003a) contendo, por exemplo, dezenas de componentes, centenas de casos de uso e milhares de classes, seria persistido em um único arquivo e versionado como um elemento indivisível. Esse cenário inviabilizaria o acompanhamento individual da evolução de cada componente, classe ou caso de uso.

No contexto desta abordagem, foi construído uma ferramenta denominada Odyssey-VCS (Oliveira, Murta *et al.*, 2005) que utiliza um modelo de dados orientado a objetos para versionar individualmente os elementos da UML. Diferentemente das demais abordagens existentes para controle de versões, que utilizam meta-modelos baseados em arquivos ou utilizam meta-modelos proprietários, a abordagem proposta segue um meta-modelo padrão, denominado MOF (*Meta-Object Facility*) (Omg, 2002). Devido a essa característica, é possível a comunicação entre o Odyssey-VCS e ferramentas CASE compatíveis com o padrão XMI (Omg, 2003b), que é uma especificação que possibilita a geração de modelos MOF em XML.

Para que esse controle individual sobre um determinado elemento da UML possa ocorrer, o mesmo deve ser definido como GV. Este elemento será considerado IC, e sua história de evolução será registrada pelo Odyssey-VCS. É importante ressaltar que a definição de quais elementos serão IC é feita com base nas características particulares de cada projeto. Desta forma, é possível que um dado projeto controle versões de casos de uso e classes, enquanto um outro projeto atue de forma mais abrangente, chegando até o nível de métodos e atributos.

Apesar da versão atual da UML não possibilitar a definição explicita de componentes, o protótipo de controle de versões está sendo utilizado para versionar modelos de componentes através do emprego de estereótipos UML. Os estereótipos permitem que as classes da UML representem os elementos existentes no DBC. Com o lançamento da UML 2.0, o protótipo será capaz de atuar diretamente no nível dos componentes.

3.3. Detecção de rastros de modificação entre artefatos

As atividades relacionadas à manutenção e construção do software impõem a modificação de artefatos de software produzidos em diferentes fases. Ao modificar um requisito, modificam-se todos os artefatos relacionados a ele, desde a análise até sua implementação em código-fonte. No entanto, apesar do processo impor a modificação em artefatos de software situados em diferentes níveis de abstração e fases do processo, conforme dito anteriormente, as ferramentas de apoio à manutenção do software estão voltadas basicamente para o código-fonte.

Considerando qualquer diagrama ou modelo presente na UML, percebe-se que durante a manutenção, as modificações realizadas em determinado modelo podem levar a modificações subseqüentes em outros artefatos do mesmo modelo UML ou em outros modelos UML relacionados. A propagação de uma modificação nos modelos tem os seguintes propósitos: (1)

mantê-los atualizados e consistentes com o software, (2) estimar o que é necessário modificar, ajudando a análise de impacto na definição dos custos e da complexidade da modificação, e (3) determinar os artefatos que devem ser alterados anteriormente à implementação, o que permite que o engenheiro de software avalie qual é a forma mais adequada de implementar a modificação. Ao considerar os efeitos de uma modificação durante a fase de análise e projeto, falhas que poderiam ser encontradas tardiamente no processo de desenvolvimento são detectadas antecipadamente, o que é uma vantagem em relação às abordagens que atuam apenas em código-fonte (Shirabad, Lethbridge *et al.*, 2003; Ying, 2003; Zimmermann, Weisgerber *et al.*, 2004).

Com o uso de DBC e a partir da UML 2.0, os artefatos produzidos durante o processo de desenvolvimento de software, situados em diferentes níveis de abstração, estarão intimamente relacionados através do conceito de componente. Devido à característica do paradigma de DBC e ao fato do componente evoluir em relação à sua funcionalidade e à sua interface, dois tipos de relações se destacam: relações entre componentes (dependências) e relações entre artefatos presentes no projeto interno de cada componente (composições), como discutido na Seção 3.2. Quando a interface é modificada durante o processo de desenvolvimento, o engenheiro de software verifica as relações de dependência entre os componentes. Quando a funcionalidade de um ou mais componentes é alterada, ele verifica as relações entre os diferentes artefatos UML que compõem o componente.

Como os componentes se relacionam via interfaces, algumas dependências lógicas entre componentes podem não ser facilmente identificadas pelo engenheiro de software durante a manutenção. Por exemplo, se a classe de um componente estiver sendo freqüentemente modificada em conjunto com a classe de um segundo componente, provavelmente existe uma dependência lógica entre dois artefatos UML que pertencem a componentes diferentes. Essa relação, por não estar documentada, não é facilmente identificada. Além disso, ela pode estar interferindo na coesão dos componentes.

Como no processo de desenvolvimento de software a complexidade de alteração de modelos de análise e projeto é alta (Briand, Labiche *et al.*, 2003), quando o engenheiro de software modifica um modelo UML, ou realiza alguma alteração em código que influencia a reorganização do modelo, ele se questiona: "Quais outros artefatos do modelo devem ser alterados em virtude dessa modificação?". Para responder a esse questionamento, é necessário encontrar relações entre artefatos UML que indiquem quais artefatos devem ser modificados em conjunto.

A rastreabilidade define relações que existem entre os diversos artefatos de um processo de desenvolvimento de software (Cleland-Huang e Chang, 2003). Nesta definição, dois tipos diferentes de rastros se destacam (Kowalczykiewicz e Weiss, 2002): (1) rastros intermodelos, que são rastros de um mesmo elemento conceitual em diversos níveis de abstração e (2) rastros intramodelo, que são rastros entre diferentes elementos conceituais, porém, em um mesmo nível de abstração.

Com o intuito de prover um instrumento de apoio para a detecção desses rastros, esta abordagem propõe correlacionar as informações existentes no Odyssey-CCS com os artefatos versionados no Odyssey-VCS (Dantas, Murta *et al.*, 2005). As técnicas de mineração de dados podem ser aplicadas com o objetivo de extrair informações relevantes dos repositórios através de pesquisa e da determinação de padrões, classificações e associações entre elas (Goebel e Gruenwald, 1999). O uso dessas técnicas no repositório dos sistemas da GCS apóia a detecção de rastros, encontrando regras do tipo: "Desenvolvedores que modificam esses artefatos também modificam esses outros artefatos". Considerando os artefatos UML, encontra-se, por exemplo, relações entre classes e casos de uso. Dentre as técnicas e algoritmos de mineração de dados, utilizamos a técnica de regra de associação e o algoritmo *Apriori* (Agrawal e Srikant, 1994).

O algoritmo *Apriori* retorna todos os elementos associados ao artefato que será modificado, calculando para cada associação o suporte e a confiança. A medida de suporte indica a co-ocorrência de elementos. Isto significa dizer que, do total de modificações realizadas no repositório dos sistemas da GCS, x% contém os elementos associados. A medida de confiança indica o quão forte é a presença de um elemento em função de outro, o que equivale a dizer que em y% das vezes que um elemento é modificado, outro também é modificado. As medidas de suporte e confiança indicam a relevância do rastro para a modificação que será efetuada.

No contexto desta abordagem, os rastros são denominados rastros de modificação visto que surgiram da evolução do software e a partir dos dados armazenados no repositório dos sistemas da GCS. No entanto, percebemos que estes rastros não apresentam significado semântico quando analisados pelo engenheiro de software. A semântica ou significado atribuído ao rastro existe em função do raciocínio empregado pelo engenheiro de software na definição do rastro. Para aumentar o entendimento e facilitar a análise dos rastros, torna-se necessário coletar e organizar informações dos sistemas de GCS, de forma a obter um indício de como o rastro surgiu. A descrição detalhada presente nos repositórios dos sistemas da GCS fornece uma grande oportunidade para a análise de estudos empíricos sobre a evolução do software. O uso destes repositórios tem a vantagem de não impor a adição de custos extras ao projeto, uma vez que os sistemas de GCS estão cada vez mais difundidos na indústria, já fazendo parte do processo de desenvolvimento de software (Hassan e Holt, 2003).

Através do processo de controle de modificações, é possível detectar informações referentes ao por quê uma modificação é necessária, como pode ser implementada, onde já foi necessário alterar para implementar, quem já implementou, quando foi implementada e o quê já foi realmente feito. As questões "o que"(*What*), "quem" (*Who*), "quando" (*When*), "onde" (*Where*), "como" (*How*) e "por quê" (*Why*) compõem a estrutura 5W+1H (Dourish e Bellotti, 1992) que caracteriza o que é necessário saber para que determinada atividade ou informação seja compreendida.

O por quê informa o motivo da modificação e pode ser obtido do documento de requisição da modificação, que é a base para o prosseguimento do processo de controle de modificações. A informação de como fazer a modificação, presente na análise de impacto realizada durante o processo, indica a forma que a modificação deve ser atendida. O autor e a data da implementação da modificação podem ser encontrados nos registros de *check-in* dos artefatos versionados na atividade de implementação. Durante o processo e após a implementação, a atividade de verificação da modificação relata o que realmente foi feito e deve ser aprovado para que a modificação faça parte da configuração de referência.

Embora correlacionar as informações de ambos os sistemas não indique propriamente o motivo de um rastro, as ocorrências capturadas formam todo o histórico dos prováveis rastros.

4. Exemplo de utilização

Nesta seção, apresentaremos como a abordagem proposta pode ser aplicada para apoiar a manutenção de software baseado em componentes. O exemplo utilizado trata-se de um sistema voltado para o cadastro de clientes, hotéis, quartos e tipos de quartos, disponível na literatura (Teixeira, 2003). Os principais casos de uso são ilustrados na Figura 1, onde dois tipos de usuários são apresentados: administradores e hóspedes. A função do administrador é manter os dados atualizados e consistentes para que consultas e reservas possam ser realizadas posteriormente pelos hóspedes.

Neste exemplo, podem ser identificados quatro componentes: "Gerente de Reserva", "Gerente de Tipos de Quarto", "Gerente de Hotéis" e "Gerente de Hóspedes", listados na Figura 2.a. A geração destes componentes é realizada com base nas classes definidas no modelo da Figura 2.b.

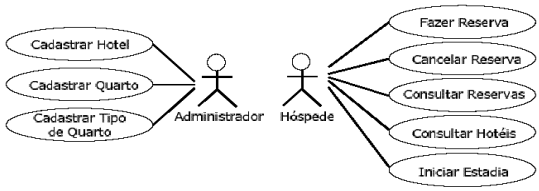

Figura 1. Casos de Uso do exemplo de hotelaria

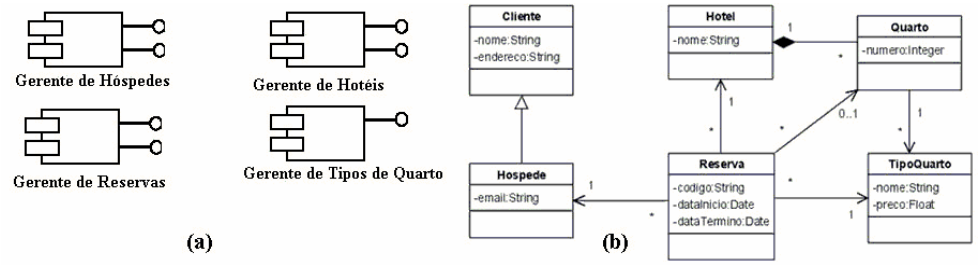

Figura 2. Componentes (a) e Classes (b) do exemplo de hotelaria

Para demonstrar o funcionamento da abordagem, seis modificações serão efetuadas sobre o sistema exemplo. Essas modificações estão listadas na Tabela 1.

Tabela 1. Descrição das modificações realizadas no exemplo hotelaria

N°	Descrição
1	Ao efetuar a reserva, verificar a disponibilidade de quartos no hotel para o tipo de quarto selecionado no período determinado para a reserva.
2	Ao consultar a reserva de um hóspede, visualizar o tipo de quarto reservado, assim como sua característica e preço na temporada.
3	Não existindo quarto disponível durante a efetuação da reserva, verificar se esse tipo de quarto se encontra disponível em outro hotel (cadastrado no sistema) da região.
4	Ao efetuar a reserva de um hóspede, não disponibilizar o quarto reservado para futuras reservas (considerar o período reservado).
5	Ao cancelar a reserva de um hóspede, disponibilizar o quarto para nova reserva (considerar o período reservado).
6	Relacionar todos os hotéis da rede hoteleira que já teve o cliente como hóspede, e apresentar como consulta para o cliente.

Inicialmente, é necessário definir o processo de controle de modificações e, para cada atividade do processo, um formulário com um conjunto de campos para a coleta de informações. A Figura 3.a apresenta a janela do ambiente de modelagem de processos, que segue a notação SPEM. Essa notação permite a modelagem de diagramas de atividades representando fluxos de trabalho (*workflows*). Nesse ambiente, é possível especificar as atividades que comporão o processo (representadas pelo ícone no formato de seta) e os produtos que devem ser manipulados por essas atividades (representados pelo ícone que contem o agrupamento de triangulo, retângulo e circulo). As transições contínuas representam fluxos de controle entre duas atividades e as transições pontilhadas representam fluxos de controle e dados em que algum produto participe. Na Figura 3.b é apresentada uma outra janela onde os formulários podem ser criados, especificando cada tipo de campo necessário.

A partir do processo modelado e dos formulários definidos, é possível determinar quais atividades fazem uso de quais formulários para geração dos produtos. Com essa associação entre atividades e formulários, o sistema é capaz de apoiar a coleta de informações durante o ciclo de vida de uma modificação. Por exemplo, a primeira atividade cadastrada no processo é "Requisição", que foi associada ao formulário "Requisição de Modificação", apresentados respectivamente na Figura 3.a e Figura 3.b. As informações coletadas nesta atividade são: nome do sistema ou projeto, classificação, prioridade e descrição da modificação. Depois da execução dessa atividade, o produto "Pedido de Modificação" é criado e o controle passa para

a atividade “Classificação”. Após esta etapa inicial de configuração, o processo é disponibilizado para execução.

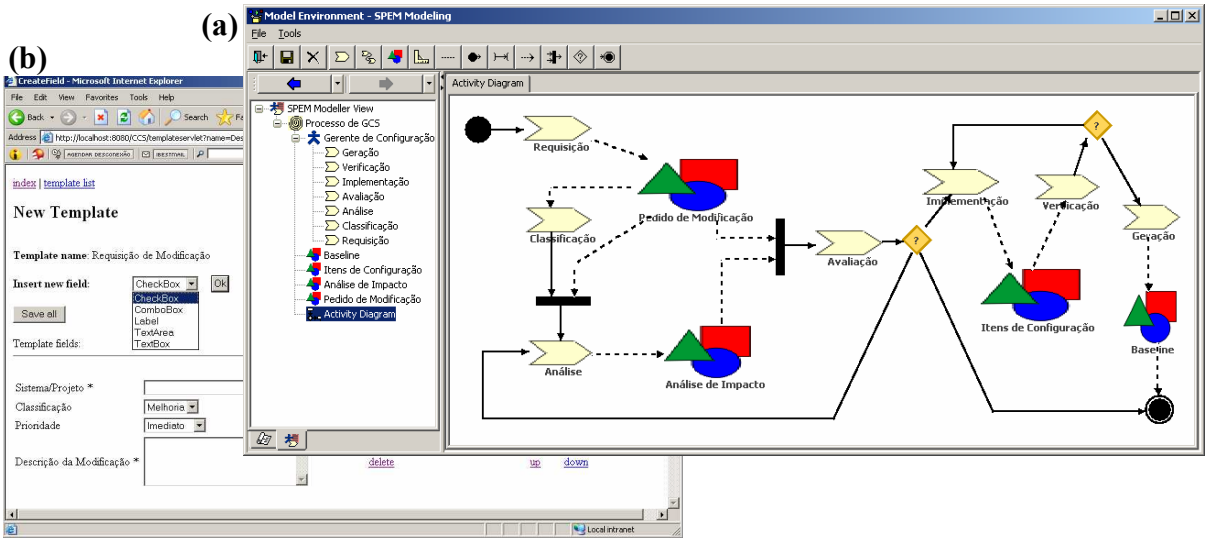

Figura 3: Configuração do formulário "Requisição de Modificação"

Durante a atividade de “Implementação”, vários desenvolvedores devem modificar os modelos em paralelo para atender as requisições de modificação listadas na Tabela 1. Para que isso seja possível, é necessário que eles interajam com o Odyssey-VCS. Essa interação ocorre através de uma ferramenta cliente, apresentada na Figura 4.

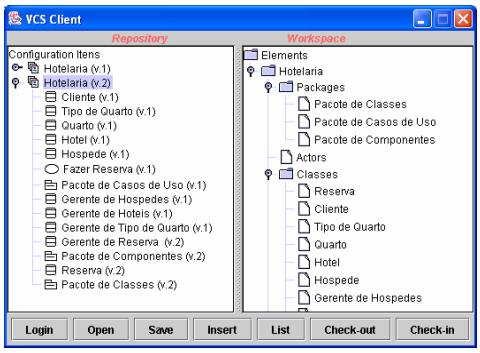

Figura 4: Visualização do repositório versionado

A ferramenta cliente viabiliza a comunicação entre ferramentas CASE e o Odyssey-VCS. Esta comunicação ocorre através da transferência de modelos UML descritos segundo o padrão XMI. O lado esquerdo da janela da ferramenta cliente exibe todos os ICs existentes no repositório (*Repository*). O lado direito da janela apresenta os elementos que se encontram na área de trabalho do desenvolvedor (*Workspace*).

Cada modificação, quando implementada, gera um conjunto de versões de elementos da UML. O Odyssey-VCS permite que estes artefatos sejam obtidos do repositório através do processo de *check-out*, modificados dentro do espaço de trabalho do desenvolvedor, e depois retornados ao repositório através do processo de *check-in*.

Como os sistemas de controle de modificações e de controle de versões estão integrados, em um único repositório, cada versão está obrigatoriamente associada à modificação que a originou. Seguindo o processo e após completar a implementação das seis modificações listadas na Tabela 1, o desenvolvedor gerou novas versões de artefatos em cada uma das modificações, como mostra a Tabela 2. Por exemplo, no caso apresentado na Figura 4, o desenvolvedor fez *check-out* sobre “Hotelaria (v.2)” para implementar a “Modificação 2”, apresentada

na Tabela 1. Esta implementação, após a execução de *check-in*, resulta na criação dos artefatos apresentados na modificação nº 2 da Tabela 2.

Tabela 2. Modificações no repositório dos sistemas de GCS

N°	Elementos Modificados			
	Pacotes	**Casos de Uso**	**Classes**	**Componentes**
1	Pacote de Casos de Uso (v 1), Pacote de Classes (v 2), Pacote de Componentes (v 2).	Fazer Reserva (v 1).	Hóspede (v 1), Tipo de Quarto (v 1), Cliente (v 1), Quarto (v 1), Hotel (v 1), Reserva (v 2).	Gerente de Reserva (v 2), Gerente de Hotéis (v 1), Gerente de Hóspedes (v 1), Gerente de Tipo de Quarto (v 1).
2	Pacote de Casos de Uso (v 3), Pacote de Classes (v 3), Pacote de Componentes (v 3).	Consultar Reserva (v 3).	Tipo de Quarto (v 3), Reserva (v 3).	Gerente de Tipo de Quarto (v 3), Gerente de Reserva (v 3).
3	Pacote de Casos de Uso (v 4), Pacote de Classes (v 4), Pacote de Componentes (v 4).	Fazer Reserva (v 4).	Reserva (v 4), Hotel (v 4).	Gerente de Reserva (v 4), Gerente de Hotéis (v 4).
4	Pacote de Casos de Uso (v 5), Pacote de Classes (v 5), Pacote de Componentes (v 5).	Fazer Reserva (v 5).	Reserva (v 5), Hotel (v 5).	Gerente de Reserva (v 5), Gerente de Hotéis (v 5).
5	Pacote de Casos de Uso (v 6), Pacote de Classes (v 7), Pacote de Componentes (v 7).	Cancelar Reserva (v 6).	Reserva (v 6), Hotel (v 7).	Gerente de Reserva (v 6), Gerente de Hotéis (v 7).
6	Pacote de Casos de Uso (v 8), Pacote de Classes (v 8), Pacote de Componentes (v 8).	Consultar Hotéis (v 8).	Hotel (v 8), Hóspede (v 8).	Gerente de Hotéis (v 8), Gerente de Hóspedes (v 8).

A cada vez que o desenvolvedor desejar saber qual artefato modificar em função da alteração de um artefato em particular, ele poderá solicitar o serviço de detecção dos rastros. Neste exemplo, após a implementação de todas as seis modificações, assumimos que existe uma sétima modificação que afetou a classe "Reserva". O desenvolvedor deseja saber quais outros artefatos devem ser alterados. A Figura 5 ilustra o resultado da mineração de dados sobre a classe "Reserva" com base nas modificações exibidas na Tabela 2.

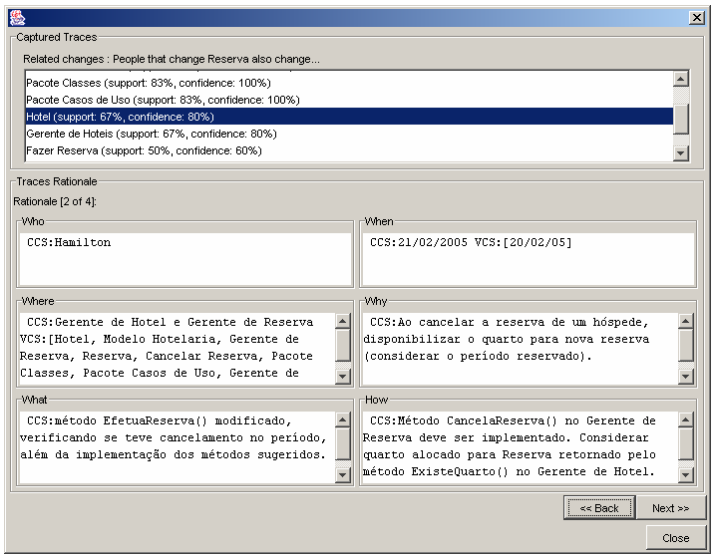

Figura 5: Resultado da detecção dos rastros de modificação

Na parte superior da janela são apresentados os artefatos que devem ser analisados quando a classe "Reserva" é modificada. Juntamente com esses artefatos, são fornecidos os valores de suporte e confiança, para que o desenvolvedor possa priorizar a análise de forma criteriosa. Na parte inferior da janela são apresentadas informações relativas às modificações passadas, que podem fornecer indícios de como proceder nas modificações futuras. Neste exemplo, o

desenvolvedor “Hamilton” poderia ser consultado antes que a modificação fosse implementada, visto que ele já modificou esses artefatos no passado e pode dar sugestões importantes para a modificação atual.

5. Trabalhos relacionados

Os trabalhos relacionados a esse artigo podem ser divididos em três áreas principais, que são: controle de modificações, controle de versões e rastreabilidade.

Dentre os trabalhos relacionados a *controle de modificações* para DBC, podemos destacar dois principais: o TERRA (Kwon, Shin *et al.*, 1999) e o KobrA (Atkinson, Bayer *et al.*, 2001). O TERRA é um protótipo que fornece apoio ao modelo de processo MwR, definido para apoiar a manutenção de um sistema legado e uma biblioteca de componentes através da reutilização de software e de funcionalidades de GCS. Entre suas principais funcionalidades estão a busca e registro de componentes, o registro de requisições de modificação, o registro de relatório de reutilização e o registro de aprovação de modificações. Existem formulários próprios para cada uma delas, onde informações são coletadas e armazenadas no sistema. Ao requisitar uma modificação, é possível incluir informações como o identificador e a versão do componente relacionado. Porém, ele disponibiliza um processo de GCS fixo, que pode ser adaptado somente via modificações no código fonte. Além disso, ele não permite a customização dos formulários de coleta de dados, sendo difícil adaptar o sistema para receber outras informações. Essas limitações não são encontradas no Odyssey-CCS.

O método KobrA é uma abordagem para engenharia de linha de produtos que envolve todo o ciclo de vida do software. Uma das áreas abrangidas pelo método é a gerência de configuração, sendo que ele utiliza um metamodelo para representar seu sistema de controle de modificações. Este método possui características interessantes, como a propagação de modificações baseada em conjunto de modificações e o relacionamento de causa entre modificações. Porém, diferentemente do Odyssey-CCS, não é de conhecimento dos autores que exista uma implementação computacional que forneça apoio à utilização do KobrA até o momento.

Na literatura de *controle de versões* existe uma ampla gama de abordagens para versionamento de arquivos. O CVS (Fogel e Bar, 2001) é um VCS de código-aberto bastante utilizado atualmente. Com o intuito de corrigir algumas das suas limitações, como o não versionamento de diretórios e a inabilidade para tratar reestruturação de código (*refactorings*), o projeto Subversion (Collins-Sussman, Fitzpatrick *et al.*, 2004) foi criado. Esses sistemas utilizam um modelo de dados baseado em sistemas de arquivos, inviabilizando uma análise mais detalhada sobre a evolução de elementos granulares, como, por exemplo, componentes, interfaces, classes e casos de uso.

Outros autores propuseram abordagens que utilizam modelos de dados mais sofisticados, como orientado a objetos (Render e Campbell, 1991), ou ainda modelo baseado na árvore sintática de uma linguagem específica (Habermann e Notkin, 1986). No entanto, estes sistemas sofrem de problemas relacionados à aplicabilidade limitada, pois usualmente atuam sobre meta-modelos proprietários, diferentemente do Odyssey-VCS, que atua sobre um meta-modelo padrão (i.e. MOF).

Os trabalhos de CHRISTENSEN (1999) e ATKINSON et al. (2001) tratam mais especificamente do problema de versionamento no contexto de DBC. Relacionamentos de dependência, conectores, interfaces e elementos arquiteturais fazem parte das preocupações dos autores. Todavia, diferentemente do Odyssey-VCS, estes sistemas não controlam a evolução dos elementos de modelo individualmente.

Finalmente, na área de *rastreabilidade* podem ser destacados os trabalhos de BALL et al. (1997) e ZIMMERMANN et al. (2004), que visam detectar automaticamente relações semânticas entre artefatos através da análise das modificações. Ao modificar artefatos do software tais como funções, classes e componentes, o engenheiro de software deve estar atento à cobertura da modificação. Neste contexto, alguns autores (Shirabad, Lethbridge *et al.*, 2003; Ying,

2003; Zimmermann, Weisgerber *et al.*, 2004) questionam quais arquivos ou rotinas seriam relevantes a uma determinada modificação. Mesmo utilizando técnicas diferentes na identificação de dependências, algumas abordagens compartilham propósitos, mas em diferentes níveis de granularidade quanto aos artefatos relacionados. ZIMMERMANN et al. (2003) relacionam variáveis ou métodos localizados no código-fonte e, a partir das dependências encontradas, analisam a arquitetura do software. A análise da arquitetura envolve verificar o grau de modularidade com base no acoplamento inter e intra-elemento. No entanto, estas abordagens ignoram os artefatos de análise e projeto, que são tratados na abordagem discutida nesse artigo, concentrando seus focos apenas em código-fonte.

6. Conclusão

Este trabalho apresentou uma abordagem para a manutenção de software baseado em componentes. Essa abordagem faz uso de técnicas de GCS, possibilitando a evolução controlada de artefatos complexos, seguindo processos não convencionais de desenvolvimento. Os principais aspectos discutidos neste artigo foram: automação do processo de manutenção, evolução controlada de artefatos complexos e detecção de rastros de modificação entre artefatos.

A contribuição principal dessa abordagem está relacionada com a integração de diferentes técnicas de GCS para apoiar a manutenção de software baseado em componentes. Outras abordagens encontradas na literatura não apóiam as características específicas da manutenção de software baseado em componentes, ou focam em aspectos muito específicos, não beneficiando as diversas etapas da manutenção. Outras contribuições importantes dessa abordagem podem ser destacadas:

- Possibilidade de modelar e instanciar o processo de controle de modificações e os formulários de coleta das informações, adaptando-os às necessidades específicas do projeto em questão;
- Controle de versões de elementos de modelo UML, utilizando uma granularidade fina e permitindo a definição do GV, de acordo com as peculiaridades de cada projeto; e
- Auxílio à manutenção dos modelos de análise e projeto no decorrer do processo de desenvolvimento do software através da detecção automática de rastros de modificação (intramodelos e intermodelos). O rastro auxilia o engenheiro de software a estimar quais artefatos deverão ser alterados em função de uma modificação, minimizando complicações como a introdução de novos erros e alterações incompletas durante a modificação, o que torna o trabalho de manutenção menos propenso à falhas.

A abordagem apresentada nesse artigo, concretizada através de protótipos implementados em Java, faz parte de um projeto mais amplo, denominado Odyssey-SCM (Murta, Dantas *et al.*, 2005), que visa prover uma infra-estrutura de GCS para o DBC. Tanto o Odyssey-VCS, quanto o protótipo para detecção de rastros de modificação entre artefatos já tem uma primeira versão operacional disponível. Contudo, o Odyssey-CCS ainda está parcialmente operacional, demandando esforço adicional de implementação. Além disso, outros trabalhos futuros também são vislumbrados:

- Obtenção de informações relevantes ao Mapa de Reutilização através da integração com uma biblioteca de componentes. Pode-se detectar, por exemplo, quais consumidores atualizaram seus componentes, e para qual versão, monitorando a cópia (*download*) de componentes da biblioteca;
- Estender o Odyssey-VCS para controlar versões de outros tipos de artefatos, como, por exemplo, documentos em XML e código-fonte Java, além da UML. Uma vantagem dessa extensão é que um único VCS e, conseqüentemente, um único repositório seria utilizado para gerenciar todo o ciclo de vida do software, desde a especificação de requisitos

até o código-fonte, o que evitaria problemas de integração de dados localizados em diferentes repositórios;

- Sumário das informações coletadas nos sistemas da GCS que dizem respeito à origem do rastro. Seria interessante trazer a informação de uma forma mais resumida para análise do engenheiro de software, ao invés de apresentar as informações cruas dos sistemas de controle de modificações e versões. Por exemplo, dados estatísticos poderiam ser fornecidos sobre quais partes do sistema foram mais afetadas por modificações equivalentes no passado (e.g.: 30% no Pacote X, 40% no Pacote Y, etc.);
- Aumento da qualidade dos rastros identificados pelo protótipo. Isto significa encontrar um número maior de rastros válidos através da comparação entre os rastros identificados manualmente pelo engenheiro de software (rastros previstos) e os rastros detectados pelo protótipo; e
- Execução de estudos experimentais, que visem medir a eficácia da abordagem proposta em cenários reais de manutenção de software baseado em componentes. Esses estudos estão em fase de elaboração e serão executados em seguida.

Agradecimentos

Os autores desejam agradecer à CAPES, ao CNPq e ao Banco Central do Brasil pelo apoio financeiro.

Referências

Agrawal, R. e R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. International Conference on Very Large Data Bases. Santiago de Chile, Chile: Morgan Kaufmann. September, 1994. 487-499 p.

Asklund, U. e L. Bendix. A Study of Configuration Management in Open Source Software. IEE Proceedings - Software, v.149, n.1, February, p.40-46. 2002.

Atkinson, C., J. Bayer*, et al.* Component-Based Product Line Engineering with UML: Addison-Wesley. 2001

Ball, T., J. Kim*, et al.* If your version control system could talk. ICSE '97 Workshop on Process Modelling and Empirical Studies of Software Engineering. Boston, MA. May, 1997.

Briand, L. C., Y. Labiche*, et al.* Impact Analysis and Change Management of UML Models. International Conference on Software Maintenance. Amsterdam, Netherlands. September, 2003. 256-265 p.

Christensen, H. B. The Ragnarok Architectural Software Configuration Management Model. Annual Hawaii International Conference on System Sciences. Maui, Hawaii. January, 1999.

Christensen, M. J. e R. H. Thayer. The Project Manager's Guide to Software Engineering's Best Practices: IEEE Computer Society Press and John Wiley & Sons. 2002

Cleland-Huang, J. e C. K. Chang. Event-Based Traceability for Managing Evolutionary Change. IEEE Transactions on Software Engineering, v.29, n.9, September, p.796-810. 2003.

Cole, J. e J. D. Gradecki. Mastering Apache Velocity: Wiley Computer Publishing. 2003

Collins-Sussman, B., B. W. Fitzpatrick*, et al.* Version Control with Subversion: O'Reilly. 2004

Dantas, C. R., L. G. P. Murta, *et al.* Consistent Evolution of UML Models by Automatic Detection of Change Traces. International Workshop on Principles of Software Evolution (IWPSE). Lisbon, Portugal. September, 2005.

Dourish, P. e V. Bellotti. Awareness and Coordination in Shared Workspaces. Conference on Computer Supported Cooperative Work. Toronto, Canada: ACM Press. October, 1992. 107-114 p.

Estublier, J., D. Leblang, *et al.* Impact of the research community on the field of software configuration management: summary of an impact project report. ACM SIGSOFT Software Engineering Notes, v.27, n.5, September, p.31-39. 2002.

Fogel, K. e M. Bar. Open Source Development with CVS. Scottsdale, Arizona: The Coriolis Group. 2001

Goebel, M. e L. Gruenwald. A Survey of Data Mining and Knowledge Discovery Software Tools. SIGKDD Explorations, v.1, n.1, June, p.20-33. 1999.

Habermann, A. N. e D. Notkin. Gandalf: Software Development Environments. Transactions on Software Engineering, v.12, n.12, December, p.1117-1127. 1986.

Hass, A. M. J. Configuration Management Principles and Practices. Boston, MA: Pearson Education, Inc. 2003

Hassan, A. E. e R. C. Holt. ADG: Annotated Dependency Graphs for Software Understanding. Visualizing Software For Understanding And Analysis (VISSOFT). Amsterdam, Netherlands. September, 2003.

Ieee. Std 1042 - IEEE Guide to Software Configuration Management. Institute of Electrical and Electronics Engineers. 1987

_____. Std 610.12 - IEEE Standard Glossary of Software Engineering Terminology. Institute of Electrical and Electronics Engineers. 1990

_____. Std 828 - IEEE Standard for Software Configuration Management Plans. Institute of Electrical and Electronics Engineers. 1998

Iso. ISO 10007, Quality Management - Guidelines for Configuration Management. International Organization for Standardization. 1995

Kowalczykiewicz, K. e D. Weiss. Traceability: Taming uncontrolled change in software development. National Software Engineering Conference. Tarnowo Podgorne, Poland, 2002.

Kwon, O., G. Shin, *et al.* Maintenance with Reuse: An Integrated Approach Based on Software Configuration Management. Asia Pacific Software Engineering Conference. Takamatsu, Japan. December, 1999. 507-515 p.

Larsson, M. Applying Configuration Management Techniques to Component-Based Systems. (Licentiate Thesis). Deparment of Information Technology, Uppsala University, Sweden, 2000.

Leon, A. A Guide to Software Configuration Management. Norwood, MA: Artech House Publishers. 2000

Moczar, L. e J. Aston. Cocoon Developer's Handbook: Sams. 2002

Murta, L. G. P. Odyssey-SCM: Uma Abordagem de Gerência de Configuração de Software para o Desenvolvimento Baseado em Componentes. (Exame de Qualificação). COPPE, UFRJ, Rio de Janeiro, Brasil, 2004.

Murta, L. G. P., C. R. Dantas, *et al.* Odyssey-SCM. 2005 2005.

Oliveira, H. L. R., L. G. P. Murta, *et al.* Odyssey-VCS: a Flexible Version Control System for UML Model Elements. International Workshop on Software Configuration Management (SCM). Lisbon, Portugal. September, 2005.

Omg. Meta Object Facility (MOF) Specification, version 1.4. Object Management Group. 2002

_____. Unified Modeling Language (UML) Specification, Version 1.5. Object Management Group. 2003a

_____. XML Metadata Interchange (XMI) Specification, Version 2.0. Object Management Group. 2003b

_____. Software Process Engineering Metamodel (SPEM), Version 1.1. Object Management Group. 2005

Pressman, R. S. Software Engineering: A Practitioner's Approach: McGraw-Hill. 1997

Render, H. e R. Campbell. An Object-oriented Model of Software Configuration Management. International Workshop on Software Configuration Management. Trondheim, Norway. June, 1991. 127-139 p.

Shirabad, J. S., T. Lethbridge, *et al.* Mining the Maintenance History of a Legacy Software System. International Conference on Software Maintenance. Amsterdam, Netherlands: IEEE Computer Society, 2003. 95-104 p.

Sowrirajan, S. e A. Hoek. Managing the Evolution of Distributed and Interrelated Components. International Workshop on Software Configuration Management. Portland, USA. May, 2003. 217-230 p.

Teixeira, H. V. Geração de Componentes de Negócio a Partir de Modelos de Análise. (M.Sc.). COPPE, UFRJ, Rio de Janeiro, Brasil, 2003.

W3c. HTML 4.01 Specification. World Wide Web Consortium. 1999

_____. XForms 1.0, W3C Recommendation. World Wide Web Consortium. 2003

Ying, A. T. Predicting Source Code Changes by Mining Revision History. (M.Sc. dissertation). University of British Columbia, 2003.

Zimmermann, T., S. Diehl, *et al.* How history justifies system architecture (or not). International Workshop on Principles of Software Evolution (IWPSE). Helsinki, Finland. September, 2003. 73-83 p.

Zimmermann, T., P. Weisgerber, *et al.* Mining version histories to guide software changes. International Conference on Software Engineering (ICSE). Edinburgh, Scotland, UK. May, 2004. 563-572 p.