# Mineração de Rastros de Modificação de Modelos em Repositórios Versionados

Cristine Dantas, Leonardo Murta, Cláudia Werner
{cristine, murta, werner}@cos.ufrj.br

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

**Abstract**

The practical use of Component Based Development (CBD) methods depends on tool support. CBD uses components, interfaces and connectors as first-class elements for structuring systems. Components use contractually described interfaces to allow cohesion and reusability. Moreover, after the not yet released UML 2.0, the artifacts produced in different phases of the development process will be related through the component concept. This article presents an approach that aims to help the software engineer during the activities of CBD, indicating existing relationships among components and other model elements. These relationships are discovered via mining over software configuration management repositories.

**Resumo**

A utilização prática de muitos dos métodos de desenvolvimento baseado em componentes (DBC) depende de ferramental de suporte. DBC faz uso de componentes, interfaces e conectores como elementos de estruturação de sistemas. Componente, como módulo coeso e reutilizável de software, faz uso de interfaces descritas de forma contratual para definir seu comportamento com os demais elementos de software. Especialmente, a partir do lançamento da *UML* 2.0, os artefatos produzidos durante o processo de desenvolvimento, situados em diferentes níveis de abstração, estarão intimamente relacionados através do conceito de componente. Esse artigo apresenta uma abordagem que visa apoiar o engenheiro de software nas atividades de DBC indicando relacionamentos existentes entre os diversos elementos de modelagem. Esses relacionamentos podem ser obtidos através do uso de sistemas de gerência de configuração de software (GCS).

**Palavras Chave:** mineração de dados, rastreabilidade, gerência de configuração de software.

## 1 Introdução

Apesar de existirem vários métodos de Desenvolvimento Baseado em Componentes (DBC) como Catalysis [7], UML Components [4] e KobrA [2], a carência de ferramental de suporte ainda é grande. Com o uso de DBC e a partir da *UML* 2.0 [16], os artefatos produzidos durante o processo de desenvolvimento de software, situados em diferentes níveis de abstração, estarão intimamente relacionados através do conceito de componente.

No desenvolvimento de componentes, sempre que um componente é modificado, essa modificação deve ser propagada para os demais artefatos de modelagem visando manter a consistência. Se o modelo de análise pertencente a um componente for alterado, os modelos de projeto desse componente também deverão ser alterados. Este artigo apresenta uma abordagem para detectar gradativamente as dependências de manutenção entre artefatos *UML* indicando quais elementos precisam ser alterados em conjunto.

A análise das informações referentes à Gerência de Configuração de Software (GCS) fornece subsídio para a melhoria do processo que está por trás das atividades de Engenharia de Software. As informações coletadas pelos diferentes sistemas de GCS devem ser organizadas e relacionadas de modo a prover conhecimento indireto sobre o desenvolvimento de software [12]. Desta forma, elas podem ser utilizadas na descoberta de informações quantitativas e qualitativas sobre vários aspectos do processo de desenvolvimento de software, diagnosticando o estado do projeto sem sobrecarregar o engenheiro de software de forma significativa. Vários autores [3, 6, 17] exploram os repositórios de versões analisando propriedades do processo de desenvolvimento. O projeto Bloff [6], por exemplo, permite que métricas sejam estabelecidas e executadas sobre o histórico de versões do projeto.

Os sistemas de GCS contêm informações sobre quais artefatos foram modificados em conjunto e sobre como e porquê o sistema evoluiu [17]. O sistema de controle de versões

permite que elementos sejam identificados como itens de configuração e que estes evoluam de forma distribuída, concorrente e disciplinada [14]. A evolução dos elementos de software é coordenada pelas atividades executadas no sistema de controle de modificações [10]. Por armazenar, através das modificações, aspectos interessantes referentes à evolução do software, são considerados fonte de dados para técnicas de análise retrospectiva. Abordagens como Ball et al. [3] e Zimmermann et al. [17] visam detectar automaticamente relações semânticas entre artefatos através da análise das modificações. Neste contexto, o uso de mineração de dados aparenta ser promissor por ir além das informações explícitas existentes no repositório.

A descoberta de conhecimento em base de dados, ou mineração de dados, representa o processo de extração de relações implícitas, ou de alto nível a partir de um conjunto de informações relevantes [5]. O uso de técnicas de mineração de dados no repositório de versões pode apoiar a detecção das dependências ou rastros de modificação, encontrando regras do tipo: "desenvolvedores que modificam esses elementos também modificam esses outros elementos". Esse conhecimento apóia tanto atividades de GCS quanto de DBC, sendo útil, respectivamente, na análise de impacto das modificações e na detecção de falhas de projeto devido ao alto acoplamento entre artefatos não correlatos [17]. Descobrir e entender relações não triviais entre elementos que precisam ser modificados em conjunto favorece o entendimento do software, evita complicações como introdução de novos erros e alterações incompletas durante a modificação [17].

Este artigo está organizado da seguinte forma: a seção 2 apresenta uma breve discussão sobre os rastros de modificação. A seção 3 discute o projeto *Odyssey-SCM* [12] que provê uma abordagem de GCS para o desenvolvimento baseado em componentes. A seção 4 apresenta a abordagem em si e a seção 5 conclui o artigo.

## 2 Rastros de Modificação

O uso de modelos de análise e projeto baseados na *UML*, em grandes projetos, leva a um certo número de modelos interdependentes. A rastreabilidade, em uma visão mais ampla e no contexto desse artigo, pode ser vista como a habilidade de relacionar elementos, de forma que essas relações possam ser recuperadas entre dois elementos quaisquer a qualquer momento e de forma consistente.

Existem duas formas de rastreabilidade [8]: rastros de pré-especificação ou pré-rastros e rastros de pós-especificação ou pós-rastros. Na pré-rastreabilidade existe a necessidade de técnicas que armazenem e recuperem informações referentes à produção e revisão de requisitos, facilitando o entendimento de outros profissionais. Portanto, está relacionada com as origens dos requisitos, produção e refinamentos dos mesmos, anteriores à especificação.

Já a pós-rastreabilidade leva em consideração a evolução do requisito nas fases do ciclo de vida, considerando os demais níveis de abstração e representação até sua implementação em código. Dentre os rastros de pós-especificação, ou pós-rastros, dois tipos diferentes de rastros se destacam [9]: (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. A complexidade envolvida em descobrir os motivos que originaram o rastro torna a sua detecção um problema em aberto na Engenharia de Software.

Com o intuito de prover algum ferramental de apoio para a detecção desses rastros, é apresentada uma abordagem baseada no processo de GCS que correlaciona as informações existentes no sistema de controle de modificações com os artefatos versionados no sistema de controle de versões. Essa correlação pode apoiar gradativamente a geração automática de rastros intermodelos e intramodelo, como exibido na Figura 1.

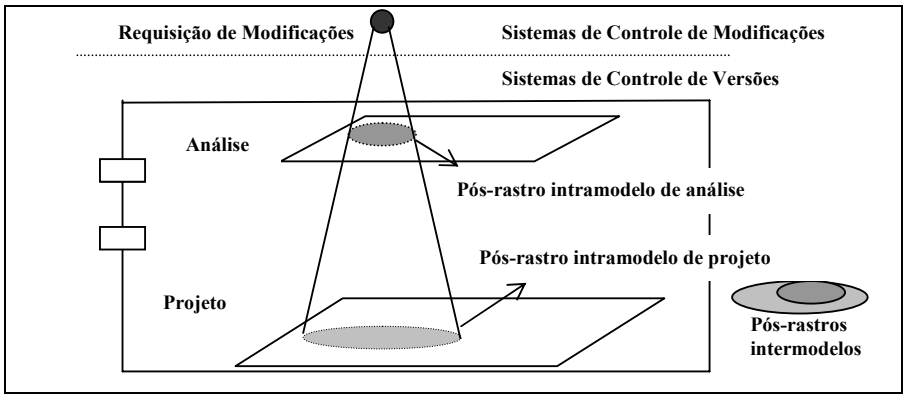

Figura 1 – Rastros de modificação no contexto de um componente

Um componente é caracterizado como um elemento coeso e composto de modelos de análise, projeto e código, entre outras representações. A Figura 1 apresenta apenas os modelos de análise e projeto como artefatos internos de um componente. Os rastros intramodelos e intermodelos aparecem no contexto de um componente.

Em um modelo de componentes, pode-se pensar em: (1) rastros intracomponentes, representando as relações de modificação entre elementos de um mesmo componente em diferentes níveis de abstração e (2) rastros intercomponentes, que como o próprio nome diz, identifica a relação de modificação entre componentes. Neste último caso, o rastro intercomponente pode fornecer um indício contrário à regra que diz que a ligação propriamente dita entre os componentes é obtida através das interfaces e dos conectores. Ao indicar, por exemplo, que um elemento do modelo de classes do "Componente 1" é sempre alterado quando se altera um elemento do modelo de classes do "Componente 2", verifica-se possíveis problemas de projeto, uma vez que interfere no conceito de coesão de componentes.

Através do processo de controle de modificações, para cada requisição de modificação, é possível detectar informações referentes ao por quê a modificação é necessária, como foi implementada, onde foi necessário alterar para implementar, quem implementou a modificação, quando foi implementada e o quê foi realmente feito.

O por quê informa o motivo da modificação e pode ser obtido do documento de requisição da modificação, que é 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*[1] 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 e apóiam o surgimento gradativo tanto de rastros intracomponentes quanto de rastros intercomponentes.

## 3 Odyssey-SCM

No projeto *Odyssey* [13], que visa apoiar a reutilização de software através de Engenharia de Domínio, Linha de Produto e DBC, a rastreabilidade tem como objetivo facilitar a identificação dos componentes do domínio em todos os níveis de abstração, ou seja, desde as

[1] O termo check-in representa o processo de revisão, aprovação e cópia de itens de configuração do espaço de trabalho do desenvolvedor para o repositório de armazenamento [10].

visões de modelos conceituais e funcionais do domínio ao longo de visões mais detalhadas para os componentes, como modelos de casos de uso, classes e interação.

No contexto do *Odyssey*, o projeto *Odyssey-SCM* [12] provê uma abordagem de GCS voltada à evolução controlada de artefatos de alto nível de abstração e às atividades de DBC, o que é, atualmente, pouco explorado pelas abordagens existentes. O *Odyssey-SCM* engloba, entre outras, as abordagens *Odyssey-CCS* e *Odyssey-VCS*. Neste projeto, a abordagem *Odyssey-CCS* [12] segue a idéia de uma abordagem configurável para o processo de controle de modificações fazendo a comunicação fluir entre os diferentes participantes do DBC.

No mesmo contexto, a abordagem *Odyssey-VCS* [14] controla versões de modelos baseados no *MOF* [15]. *MOF* é um padrão da *Object Management Group* (*OMG*) que especifica uma linguagem abstrata para descrever outras linguagens. As linguagens abstratas descritas a partir do *MOF* são chamadas de meta-modelos. O meta-modelo da *UML*, por exemplo, é descrito a partir do *MOF*. Nesse sentido, o projeto *Odyssey-SCM* faz uso do repositório *Netbeans MDR* [11] para persistir versões de modelos criados a partir de meta-modelos *MOF*, como, por exemplo, a *UML*.

Além de versionar, *Odyssey-VCS* permite modificar o grão de um artefato de modelo para mantê-lo sobre controle, adotando o conceito de grão de versionamento [12]. O grão de versionamento representa os elementos que serão versionados e, portanto, irão se tornar itens de configuração quando enviados para o repositório. Portanto, se diferencia das abordagens que procuram versionar arquivos como elementos únicos e indivisíveis.

## 4 Abordagem

A abordagem apresentada neste artigo visa aplicar técnicas de mineração de dados sobre o repositório de versões *Odyssey-VCS* e sobre o repositório de requisições de modificações *Odyssey-CCS* para gerar pós-rastros intracomponentes e intercomponentes, levando em consideração o grão definido como unidade de versão.

No *Odyssey-CCS* [12], o engenheiro de software informa a requisição de modificação que será implementada por ele. Ao iniciar a implementação, ele realiza *check-out*[2]*,* se o artefato a ser modificado existir no repositório do *Odyssey-VCS*, ou simplesmente cria o artefato*,* se ele não existir. Ao terminar, ele deverá fazer o registro de *check-in* dos artefatos criados ou modificados para que estes sejam versionados no *Odyssey-VCS*, segundo a política definida para o grão de versionamento. Cada *check-in* realizado em virtude de uma modificação executada por um engenheiro de software, dá origem a novas versões de itens de configuração. Como exemplo, utiliza-se o modelo de componentes da Figura 2.

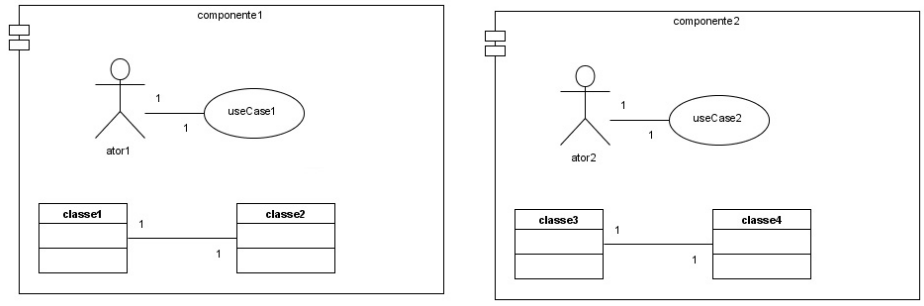


Figura 2 – Exemplo do modelo do usuário

A Figura 2 mostra a representação de dois componentes e seus respectivos artefatos internos. A implementação desse modelo ocorreu a partir das modificações descritas na Figura 3:

[2] O termo check-out representa o processo de requisição, aprovação e cópia de itens de configuração do repositório de armazenamento para o espaço de trabalho do desenvolvedor [10].

| |
|---|
| Modificação 1: implementação de uma nova funcionalidade 1 encapsulada no componente 1.<br>Modificação 2: implementação de uma nova funcionalidade 2 encapsulada no componente 2.<br>Modificação 3: incorporação do modelo de projeto e alterações no modelo de análise da funcionalidade 1.<br>Modificação 4: incorporação do modelo de projeto e alterações no modelo de análise da funcionalidade 2.<br>Modificação 5: alteração do modelo de projeto da funcionalidade 2. |

Figura 3 – Exemplo de modificações efetuadas no modelo

Como dito anteriormente, o *Netbeans MDR* é um meio de armazenamento capaz de carregar um meta-modelo *MOF* e armazenar instâncias desse meta-modelo. O *Odyssey-SCM* [12] utiliza um meta-modelo de versionamento para persistir versões de modelos criados a partir de meta-modelos *MOF*, como, por exemplo, a *UML*. Cada modificação inserida no repositório passa a estar disponível para que o engenheiro de software a implemente. Cada *check-in* gera uma versão do artefato armazenada no repositório. Recuperando as instâncias armazenadas no repositório, encontra-se um conjunto de registros onde cada registro é composto da modificação efetuada e de todos os itens de configuração versionados devido a ela. Para obter esse tipo de informação, é necessário navegar no meta-modelo e agrupar todos os itens de configuração que foram versionados segundo uma modificação. Os registros encontrados, para o exemplo da Figura 2, são apresentados na Figura 4.

| Representação: (Modificação, (ItemConfiguração (versão))) |
|---|
| (Modificação 1, (casoUso1(1.0), ator1(1.0), componente1(1.0)))<br>(Modificação 2, (casoUso2(1.0), ator2(1.0), componente2(1.0)))<br>(Modificação 3, (casoUso1(2.0), componente1(2.0), classe1(1.0),classe2(1.0)))<br>(Modificação 4, (ator2(2.0), componente2(2.0), classe3(1.0),classe4(1.0), classe1(2.0), componente1(3.0)))<br>(Modificação 5, (componente2(3.0), classe3(2.0), componente1(4.0), classe1(3.0))) |

Figura 4 – Exemplo de registros de modificação

Durante a implementação, caso necessite de orientação, o engenheiro pode consultar quais artefatos ele, provavelmente, deverá modificar. Para isso, ele seleciona a versão do artefato originalmente recuperado do repositório quando foi realizado *check-out*, e realiza a operação de mineração do repositório em busca de regras que indiquem quais elementos o engenheiro de software deve considerar ao alterar o elemento selecionado.

Em termos gerais, a mineração de dados por regras de associação, encontra relações entre elementos, ou regras, através da procura por conjuntos de elementos freqüentemente encontrados em ocorrências na base. A base é formada pelos registros de modificação. Uma regra de associação significa uma implicação na forma $X \rightarrow Y$, formada por um corpo (X) e uma parte conseqüente (Y). Pode-se ter interesse apenas em regras que tenham um elemento específico aparecendo na parte conseqüente ou no corpo da regra. A abordagem considera que o elemento selecionado pelo engenheiro de software no *Odyssey* será o corpo da regra. Os elementos presentes na parte conseqüente da regra são os elementos que deverão ser alterados.

O algoritmo de mineração de dados utilizado é o *Apriori* [1]. Esse algoritmo se baseia em duas medidas ao selecionar as regras de associação: suporte e confiança. As medidas de suporte e confiança podem indicar o quanto a regra é recomendada. Supondo que existe no repositório os elementos de modelo "a" e "b" dentre outros, a medida de suporte indica que x% de todas as ocorrências da base sob análise mostram que "a" e "b" foram alterados juntos, embora sejam elementos independentes, ou seja, corresponde à probabilidade de "a" união "b". A medida de confiança indica que em y% das ocorrências quando se modifica "a" também se modifica "b", ou seja, corresponde à probabilidade condicional de "b" em relação a "a".

Ao realizar a mineração dos dados, apenas as regras com medida de suporte igual ou maior que um valor determinado pelo engenheiro de software serão escolhidas. A abordagem mantém as medidas configuráveis para três tipos de precisão: alto, médio e baixo. Na Figura 5, o engenheiro de software escolheu a precisão baixa, com medidas de suporte e confiança fixadas em 20%, assim como definiu a "Classe 3" como corpo da regra. Deste exemplo, conclui-se que ao alterar a "Classe 3" do "Componente 2", pode ser necessário alterar a

"Classe 1" do "Componente 1". Esse tipo de comportamento que interfere no conceito de coesão de um componente deve ser avaliado pelo engenheiro de software.

| Representação na regra de associação: Corpo → Conseqüentes [suporte, confiança]. |
|---|
| classe3 → componente1, componente2, classe1 [40%,100%] |
| classe3 → componente2, classe4 [20%, 50%] |

Figura 5 – Exemplo de regras geradas para o modelo da Figura 2

Se for interessante para o engenheiro de software recuperar regras de mineração com baixa precisão, então um número maior de regras é gerado devido a configuração das medidas de suporte e confiança com valores baixos. Em contrapartida, gerar um maior número de regras pode indicar dependências incorretas representadas nas regras recuperadas.

## 5 Conclusões

Esse artigo apresentou uma abordagem que aplica técnicas de mineração de dados sobre repositórios *MOF* versionados para gerar pós-rastros intracomponentes e intercomponentes. A solução consiste em interferir o mínimo necessário nas atividades de DBC. Para esse fim, utiliza-se conhecimentos do ambiente de GCS como as informações do sistema de controle de versões e do sistema de controle de modificações. Portanto, é importante que ambos os sistemas de GCS sejam utilizados em conjunto durante a manutenção ou construção do software. Atualmente, um protótipo está sendo implementado na COPPE/UFRJ.

Embora algumas abordagens [3, 17] apresentem contribuições similares, como, por exemplo, apoio na detecção de modificações incompletas, na análise de impacto e na reengenharia de componentes, a abordagem apresentada nesse artigo se diferencia por estar no nível de abstração de componentes e modelos *UML*, e não no nível de código-fonte. Na abordagem proposta neste artigo, a detecção de rastros intercomponentes induz o engenheiro de software a reavaliar o modelo de componentes, caso sua detecção informe a quebra da coesão do componente.


## Agradecimentos

Os autores gostariam de agradecer o CNPq e a CAPES pelo investimento financeiro neste trabalho e aos demais integrantes do Projeto *Odyssey* pelo apoio durante o seu desenvolvimento.


## Referências


[1] *AGRAWAL, R., SRIKANT, R.,* 1994, Fast algorithms for mining association rules. In Proceedings of the 20th Very Large Databases Conference (VLDB), pages 487-499. Morgan Kaufmann.

[2] *ATKINSON*, *C. et al*., 2002, Component-based Product Line Engineering with UML, Addison Wesley.

[3] *BALL, T., KIM, J., PORTER, A.A., et al.,* 1997, If your version control system could talk. In: ICSE'97 Workshop on Process Modeling and Empirical Studies of Software Engineering, Boston, MA, May.

[4] *CHEESMAN, J., DANIELS, J.,* 2001, UML Components: A Simple Process for Specifying Component-Based Software 1, Upper Saddle River, NJ, Addison Wesley

[5] *CHEN, MING-SYAN; HAN, J.; YU, P.S.;* 1996, Data Mining: An Overview from Database Perspective, IEEE Transactions on Knowledge and Data Engineering, December, 8(6), pp: 866-883.

[6] *DRAHEIM, D., PEKACKI, L.,* 2003, Analytical Processing of Version Control Data: Towards a Process-Centric Viewpoint. Free University Berlin.

[7] *D'SOUZA, D., WILLS*, *A. C.,* 1998, Objects, Components and Frameworks with UML. 1, Addison Wesley

[8] *GOTEL, O.; FINKELSTEIN, A.;* 1994, An Analysis of the Requirements Traceability Problem, Proc. IEEE International Conference on Requirements Engineering, IEEE Computer Society Press, Colorado, Colorado Springs, pp.94-101.

[9] *KOWALCZYKIEWICZ, K.; WEISS, D.;* 2002, Traceability: Taming uncontrolled change in software development, Proceedings of IV National Software Engineering Conference, Tarnowo Podgorne, Poland, 10 pages.

[10] *LEON, A.,* 2000, A Guide to Software Configuration Management, Norwood, MA, Artech House Publishers.

[11] *MATULA, M.,* 2003, NetBeans Metadata Repository, in: http://mdr.netbeans.org/docs.html, accessed in 17/05/2004.

[12] *MURTA, L. G. P.,* 2004, Odyssey-SCM: Uma Abordagem de Gerência de Configuração de Software para o Desenvolvimento Baseado em Componentes, Exame de Qualificação, COPPE/UFRJ, Rio de Janeiro, RJ, Maio.

[13] *ODYSSEY SDE* (2004) Infra-estrutura de Reutilização Odyssey. Disponível em http: //www.cos.ufrj.br/~odyssey.

[14] *OLIVEIRA, H., MURTA, L.G.P., WERNER, C.M.L,* 2004, Odyssey-VCS: Um Sistema de Controle de Versões Para Modelos Baseados no MOF, submetido para XVIII Simpósio Brasileiro de Engenharia de Software, Caderno de Ferramentas.

[15] *OMG*, 2003a, Meta Object Facility (MOF) specification, version 1.4, Object Management Group.

[16] *UML* 2.0, http://www.uml.org/ accessed in 20/06/2004.

[17] *ZIMMERMANN, T.; WEISGERBER, P.; DIEHL, S.; ZELLER, A*.; 2003, Mining version histories to guide software changes, Proceeding International Conference on Software Engineering (ICSE 2004), EICC, Scotland, UK, May (23-28).