Ostra: uma abordagem para análise da qualidade de software por meio de regras de associação de métricas

Daniel D. C. Ribeiro, Alexandre Plastino, Leonardo Murta

Instituto de Computação – Universidade Federal Fluminense (UFF)
Rua Passos da Pátria 156, 24.220-240 – Niterói – RJ – Brazil

danielcastellani@id.uff.br, {plastino, leomurta}@ic.uff.br

Abstract. *A key element to control the software quality, not only reacting to its variation, is the understanding of which factors influence the quality attributes and how they influence each other. In this work, we present the Ostra approach, which allows the historical analysis of software by mining association rules of its metrics, extracted from the historic information stored in version control systems. The Ostra's goal is to provide information to decision-making process. We also present experiments in which the Ostra approach is applied to real projects. With these experiments, it was possible to find evidences that the proposed approach can achieve its goals.*

Resumo. *Para controlar a qualidade de software e não apenas reagir à sua variação, deve-se entender quais os fatores que influenciam os atributos de qualidade e qual a influência deles entre si. Neste trabalho, é apresentada a abordagem Ostra, que permite a análise do software através da mineração de regras de associação de métricas, extraídas do histórico do software armazenado no sistema de controle de versão. O objetivo da Ostra é fornecer informações relevantes sobre a qualidade do software para o processo de tomada de decisão. Para avaliar esse objetivo, são apresentados experimentos nos quais a abordagem é aplicada em projetos reais. Com os experimentos, foi possível obter indícios que a proposta consegue alcançar seu objetivo.*

1. Introdução

O ciclo de vida de um software pode ser dividido em três fases, de acordo com o seu foco [Pressman 2001]: definição, desenvolvimento e manutenção. A manutenção consome cerca de 60% do tempo [Pressman 2001] e 90% do custo do projeto [Erlikh 2000]. Analisando-se superficialmente, para reduzir o tempo e o custo do projeto, o objetivo torna-se investir na melhoria da manutenção, já que esta detém os maiores gastos de tempo e custo. Um empecilho para tal investimento consiste no fato da flexibilidade e manutenibilidade do software – atributos de qualidade que definem a facilidade de se estender ou consertar o software – serem estabelecidos durante a fase de desenvolvimento [Wiegers 2003]. Isso acontece, pois quanto maior a qualidade do produto desenvolvido, menos manutenções corretivas serão necessárias e mais fácil será a evolução do software. Consequentemente, é necessário que se invista na qualidade desde o início da fase de desenvolvimento para minimizar os custos e o tempo gastos na manutenção durante todo o projeto.

Métricas são utilizadas para descrever constantemente a qualidade de objetos. As métricas são importantes para qualquer disciplina de engenharia para descrever seus objetos de estudo e são fundamentais para um engenheiro de software [Pressman 2001]. A compreensão do relacionamento entre as métricas e seu comportamento ao longo da evolução do software permite um melhor entendimento do software em desenvolvimento. Esse entendimento permite o aprendizado de como as métricas e, consequentemente, a qualidade do software varia com as modificações ao longo do tempo.

Para que o gerente do projeto possa tomar decisões mais acertadas e agir de forma a maximizar os atributos de qualidade de seu interesse, ele deve ser capaz de monitorar a qualidade do software e identificar o momento de intervir. Dessa forma, é necessário entender como os atributos de qualidade variam isoladamente e entre si, pois nem sempre é possível maximizar todos ao mesmo tempo [Henderson-Sellers 1995]. Consequentemente, são fundamentais informações sobre a evolução do software e sobre como a qualidade é afetada pelas modificações que ocorreram ao longo de sua evolução. Assim, para que o gerente de projeto possa controlar a evolução do software, e não somente reagir a ela, ele deve ser capaz de monitorar a qualidade e possuir informações sobre quem afetou a qualidade do software, como e quando.

A partir da necessidade de prover condições para que um gerente de projetos acompanhe e controle a evolução do software, este trabalho estuda o processo de evolução de software e o relacionamento entre métricas. O objetivo principal é fornecer informações relevantes que aumentem o conhecimento sobre a evolução do software e auxiliem no processo de tomada de decisão. Na abordagem proposta, a evolução do software é analisada através das modificações realizadas ao longo do seu desenvolvimento, armazenadas em um Sistema de Controle de Versão. Essas modificações são descritas em termos da variação de métricas, que posteriormente são utilizadas para extrair regras de associação e gerar gráficos. O nome da abordagem proposta é Ostra e ela se baseia em três disciplinas para alcançar seu objetivo: métricas de software, gerência de configuração e mineração de dados.

Para avaliar a Ostra, as seguintes questões de pesquisa foram consideradas: (1) "a mineração de dados é capaz de obter informações relevantes sobre a evolução do projeto?" e (2) "a mineração de regras de associação é capaz de obter informações relevantes sobre o relacionamento entre métricas de software?". A primeira questão de pesquisa está relacionada ao objetivo da Ostra de fornecer informações que possam auxiliar no processo de tomada de decisão. Já a segunda questão de pesquisa está relacionada à identificação de padrões evolutivos de um ou mais projetos.

Este artigo está organizado em cinco seções. Na Seção 2, são apresentados conceitos fundamentais ao entendimento de mineração de repositórios de software e trabalhos relacionados. Na Seção 3, é apresentada a abordagem Ostra. Na Seção 4, são apresentados experimentos com o objetivo de responder às questões de pesquisa e, na Seção 5, é apresentada a conclusão do artigo.

2. Mineração de Repositórios de Software

A Gerência de Configuração é uma disciplina que torna possível a evolução controlada do software [Dart 1991]. Conforme o desenvolvimento do software amadurece, torna-se necessário controlar sua evolução almejando maior produtividade. A Gerência de

Configuração auxilia as atividades de desenvolvimento de software à medida que possibilita que a evolução ocorra de forma concorrente e distribuída.

Durante a evolução do software, são geradas grandes quantidades de dados pelas ferramentas de Gerência de Configuração que podem ser explorados por técnicas de Mineração de Dados [Han et al. 2011 Witten et al. 2011] para descobrir informações valiosas. Técnicas de Mineração de Dados têm sido utilizadas para encontrar mudanças que introduzam erros no software [Kim et al. 2006], para descobrir qual a melhor forma de ordenar *releases* [Colares et al. 2009], e até para descobrir quais artefatos de um software também devem ser alterados quando ocorrer uma alteração em um determinado artefato [Júnior et al. 2009 Zimmermann, T. et al. 2004]. Outros trabalhos analisam *releases* para entender mais sobre a evolução do software [Robles et al. 2006 Wermelinger and Yu 2008]

Outra forma de estudar a evolução do software é através da mineração de métricas de software. As métricas de software são utilizadas para caracterizar objetos quantificando suas características [IEEE 1990 Pressman 2001]. Assim, ao estudar a relação entre as métricas pode-se melhorar o entendimento do software e de como é sua evolução. Essa ideia já inspirou trabalhos com objetivo de aumentar o entendimento sobre as métricas [Dick et al. 2004 Nagappan et al. 2006].

Apesar de existirem trabalhos que abordam mineração de repositórios de software e métricas de software, existem lacunas ainda não abordadas nesses estudos. Essas lacunas são discutidas a seguir.

Wermelinger e Yu [2008] apontam como um ponto forte da sua abordagem a independência do Sistema de Controle de Versão, porém a obtenção das versões do software é realizada de forma manual, o que impossibilita a análise de muitas versões. O mesmo acontece no trabalho de Robles et al. [2006], que apesar de analisarem sete anos de releases, eles consideram apenas cinco versões do Debian Linux.

Um dos trabalhos relacionados utiliza regras de associação para identificar quais arquivos são alterados conjuntamente [Júnior et al. 2009 Zimmermann, T. et al. 2004]. Entretanto, não foram encontrados trabalhos que minerem regras a partir de métricas de software para descrever as alterações que ocorreram durante a evolução.

Outra questão interessante é que, dentre os trabalhos encontrados até o momento, o único que consegue definir valores para atributos de qualidade automaticamente do código fonte do software é o de Bansyia e Davis [2002]. As métricas que são utilizadas nos outros trabalhos estão no nível dos componentes de projeto, ou seja, nem sempre são capazes de avaliar a qualidade do software por meio de atributos de qualidade, que são elementos de mais alto nível e mais fáceis de serem entendidos.

3. Abordagem Ostra

Com base na pesquisa realizada, foi identificada uma possibilidade de pesquisa que: (1) considere as modificações realizadas pelos desenvolvedores através dos *commits* como elementos primários de análise; (2) consiga avaliar as diversas versões do software automaticamente, utilizando gerência de configuração e métricas de software; (3) tenha avaliações realizadas de forma objetiva com métricas de software que descrevem as alterações realizadas durante a evolução do software; e (4) aplique técnicas de mineração de dados a uma base com transações referentes às modificações realizadas em cada *commit* para obter informações relevantes sobre a evolução da qualidade do software.

Nesta seção, é apresentada a abordagem Ostra, que contempla essas oportunidades. O funcionamento da Ostra é dividido em três etapas principais: (1) medição das revisões do software, (2) mineração de dados, e (3) apresentação das informações descobertas através de regras, gráficos e tabelas. O restante dessa seção detalha cada uma dessas três etapas.

3.1. Fase de Medição

A abordagem Ostra se inicia com a medição das revisões do software. Esse passo está subdividido em obtenção e construção das revisões e extração das métricas. Na primeira parte, cada revisão do software é obtida do Sistema de Controle de Versão e construída utilizando o Sistema de Controle de Construção, pois existem métricas que são extraídas do código compilado. Finalmente, são realizadas medições no projeto. A implementação atual da Ostra utiliza o Subversion [Collins-Sussman et al. 2008] como Sistema de Controle de Versão e o Maven [Apache Software Foundation 2011] como Sistema de Gerenciamento de Construção. Mas, esses são pontos de extensão da abordagem.

Nesta seção, primeiramente, é apresentada a arquitetura orientada a objetos utilizada para representar e armazenar as informações sobre a evolução do software. Em seguida, são apresentados os tipos de métricas que a Ostra é capaz de coletar: simples, compostas e de delta. Finalmente, são apresentadas as métricas utilizadas na abordagem.

Os conceitos referentes à evolução do software são mapeados em um modelo Orientado a Objetos (OO), persistido em banco de dados. Isso possibilita a utilização dessas informações nas próximas etapas da abordagem. Detalhes da modelagem utilizada podem ser encontrados na dissertação [Ribeiro 2012].

As métricas de software e os atributos de qualidade são mapeados respectivamente como métricas simples e compostas. As métricas simples são aquelas que podem ser extraídas diretamente do código fonte do software. Já as métricas compostas são calculadas a partir de expressões definidas com base em métricas simples ou outras métricas compostas. O objetivo do suporte às métricas compostas é poder criar facilmente novas métricas com a combinação de outras. Um exemplo da utilização de métricas simples e compostas é o cálculo da densidade de linhas de código por método. Nesse caso, são utilizadas duas métricas simples: número de linhas de código (LOC) e número de métodos (NOM). Com elas, é criada a métrica composta, definida pela divisão de LOC por NOM. Para se extrair as métricas compostas, é necessário a extração prévia das métricas simples. Consequentemente, o passo de extração das métricas é iniciado com a extração das métricas simples e, após todas serem extraídas, é iniciado o cálculo das métricas compostas.

Existem métricas que são extraídas do código fonte e outras do software executável. Isso varia de acordo com a implementação da medição ou concepção da métrica. Para compilar versões diferentes de um software, com dependências diferentes e em constante mudança, deve-se delegar a construção do software para o Sistema de Gerenciamento de Construção. A abordagem foi concebida assim, pois um dos seus objetivos é viabilizar a análise do histórico de um projeto automaticamente.

O armazenamento das medições respeita a granularidade de cada métrica. Assim, medidas de métricas de projetos são armazenadas de maneira diferente das medidas de métricas de pacote e classe. Dessa forma, as medidas podem ser acessadas posteriormente sem perda de informação, respeitando a granularidade utilizada na medição.

As métricas apresentadas até o momento são medidas absolutas das características do software ou de seus artefatos (para métricas de pacote ou classe). Isoladamente, elas não medem as alterações realizadas ao longo da evolução, mas apenas como o software está em um determinado momento. Consequentemente, em um contexto evolutivo, falta uma medida que avalie como o software evoluiu com as mudanças ao longo do tempo. Para isso é utilizado o delta (diferença) das métricas, que visa mostrar como a qualidade do software variou em cada *commit* de acordo com a métrica considerada.

Utilizar o delta é interessante, pois no contexto evolutivo, o objeto de estudo não é o projeto propriamente dito, mas as mudanças realizadas a ele que o transformam a cada versão. Assim, é necessário saber o impacto da mudança entre essas versões, medindo suas alterações. Nesse cenário, o gerente de projetos deveria saber como um determinado *commit* afetou o software. Em outras palavras, se a mudança está seguindo o comportamento esperado para esse projeto, de acordo com os dados históricos, ou se é um valor anormal que precisa ser investigado. Para exemplificar, suponha um projeto com 260.000 linhas de código que possui uma variação média em torno de 100 linhas por *commit*, com desvio padrão de 50 linhas, e que um desenvolvedor adiciona em um único *commit* 1.000 LOC, levando o projeto para 261.000 LOC. O número de linhas de código do projeto se manteve aproximadamente o mesmo, pois variou menos de 1%. Porém, a variação do *commit* foi muito além da variação padrão. Consequentemente, esse *commit* é anormal (considerando a variação de LOC) e deveria ser verificado para evitar problemas futuros. Esse problema é ainda pior se as 1.000 linhas de código foram adicionadas à mesma classe, que tinha, antes da alteração, apenas 20 linhas. Analisando esse cenário, adicionar as 1.000 linhas de código a apenas uma classe é provavelmente um indicativo de problema arquitetural.

A abordagem Ostra não é dependente de um conjunto de métricas específico. Ao contrário, podem ser utilizadas quaisquer métricas que sejam interessantes para quem estiver analisando o projeto. Contudo, neste trabalho é utilizado o conjunto de métricas de QMOOD [Bansiya and Davis 2002], que pode ser diretamente mapeado para atributos de qualidade através de equações já existentes. Com esse conjunto de métricas, pode-se medir cada revisão e transformar os valores das métricas em atributos de qualidade.

Junto ao conjunto de métricas de QMOOD, foram escolhidas ainda outras métricas que podem ser utilizadas como indicadores. Essas outras métricas são: Complexidade Ciclomática de McCabe [1976], Linhas de Código e Número de Métodos, e cada uma delas complementa as métricas de QMOOD com outras informações. A Complexidade Ciclomática tem relação direta com a facilidade de testar o código desenvolvido e avalia a quantidade de caminhos possíveis que um código possui. A métrica LOC avalia o tamanho do software em linhas de código executáveis e pode ser utilizado junto à Complexidade Ciclomática para avaliar a relação entre complexidade e tamanho do software. A métrica NOM, assim como LOC, avalia o tamanho do software, contabilizando a quantidade de métodos do projeto. Essa métrica, combinada à Complexidade Ciclomática, é capaz de informar a complexidade média por métodos, informação relevante para equipes que se preocupam com a capacidade de se testar o software.

As métricas utilizadas na Ostra estão listadas na Tabela 1, na qual é indicado se a métrica é proveniente do modelo QMOOD. A coluna alvo indica se a métrica é extraída

de classe, pacote ou do software como um todo. Na Tabela 2, são apresentados os atributos de qualidade de QMOOD e as equações para calculá-los em função das métricas.

Tabela 1: Métricas utilizadas na abordagem Ostra.

Métrica	Sigla	QMOOD	Alvo
Número Médio de Ancestrais	ANA	Sim	Projeto
Tamanho do Projeto em Classes	DSC	Sim	Projeto
Abstração Funcional	MFA	Sim	Projeto
Número de Métodos Polimórficos	NOP	Sim	Projeto
Número de Hierarquias	NOH	Sim	Projeto
Medida de Agregação	MOA	Sim	Projeto
Complexidade Ciclomática Total	TCC	Não	Projeto
Abstração	RMA	Sim	Pacote
Complexidade Ciclomática de	ACC	Não	Classe
Coesão entre os Métodos da Classe	CAM	Sim	Classe
Número de Métodos	NOM	Não	Classe
Linhas de Código	LOC	Não	Classe
Métrica de Acesso a Dados	DAM	Sim	Classe
Acoplamento Direto de Classes	DCC	Sim	Classe
Tamanho da Interface da Classe	CIS	Sim	Classe

Tabela 2: Atributos de Qualidade de QMOOD como métricas compostas.

Atributo de Qualidade	Sigla	Equação da métrica composta
Reusabilidade	Reu	-0.25 * DCC + 0.25 * CAM + 0.5 * CIS + 0.5 * DSC
Flexibilidade	Fle	0.5 * ANA - 0.5 * DCC + 0.5 * MFA + 0.5 * NOP
Entendimento	Ent	-0.33 * ANA + 0.33 * DAM - 0.33 * DCC + 0.33 * CAM - 0.33 * NOP - 0.33 * NOM - 0.33 * DSC
Funcionalidade	Fun	0.12 * CAM + 0.22 * NOP + 0.22 * CIS + 0.22 * DSC + 0.22 * NOM
Extensibilidade	Ext	0.5 * ANA - 0.5 * DCC + 0.5 * MFA + 0.5 * NOP
Efetividade	Efe	0.2 * ANA - 0.2 * DAM + 0.2 * MOA + 0.2 * MFA + 0.2 * NOP

3.2. Fase de Mineração

O segundo passo consiste em minerar a base de dados em busca de informações sobre o projeto. A mineração de dados busca por informações valiosas e desconhecidas sobre as métricas, através de regras de associação, melhorando o conhecimento sobre os relacionamentos entre métricas de software e sobre os desenvolvedores. Para aplicar a mineração de dados, é construída uma base de dados que reflete as mudanças realizadas. Essas mudanças são descritas através da variação de métricas de software. Nesse passo, é criada uma base de dados a partir das medidas coletadas na fase de medição, que é utilizada para extrair os padrões.

Knowledge Discovery in Databases (KDD) é o termo usado para descrever o macro processo de mineração de dados [Han et al. 2011 Witten et al. 2011]. O apoio da Ostra ao processo de KDD inicia ainda antes da fase de seleção: ao medir o projeto e extrair as métricas que descrevem a evolução do software. Depois disso, segue

desde a seleção dos dados, que formam a base de dados, até a fase final, que compreende a exibição dos padrões minerados e sua validação para construir o novo conhecimento.

Para realizar a mineração de dados propriamente dita, é utilizada uma base de dados onde cada transação é equivalente a um *commit* e os itens da transação são informações sobre ele. São utilizados os seguintes dados para criar as transações: o nome do projeto e o número da revisão (como um identificador único da transação); o momento no qual ocorreu o *commit* (dia da semana, turno de trabalho e hora), o desenvolvedor; o número de arquivos modificados; e o delta das métricas que estão sendo utilizadas para descrever as alterações.

Para facilitar a interpretação das regras que serão mineradas, são aplicadas duas discretizações sobre os dados originais: turno e variação do delta. O turno de trabalho é proveniente do momento do *commit*, que é dividido em quatro intervalos: madrugada, manhã, tarde e noite. Cada turno tem um período de seis horas iniciando com a madrugada às 00h00min. A variação do delta representa se a modificação aumentou, manteve ou diminuiu o valor da métrica em questão. Por exemplo, se LOC aumentou em 10 linhas em um *commit*, o delta do LOC na base de dados a ser minerada é positivo. Se a revisão não contém alguma das informações selecionadas, um símbolo específico é utilizado para identificar que existe um valor faltante. Isso é utilizado, por exemplo, quando a compilação falha, pois para realizar a medição de algumas métricas é necessário que o projeto seja previamente compilado.

Após ter-se a base de dados formada, são extraídas as regras de associação. A extração das regras de associação ocorre na fase de mineração de dados e é a principal forma de descoberta de conhecimento da abordagem Ostra. As regras relacionam os atributos da base de dados utilizada na mineração e mostram os padrões considerados interessantes de acordo com as medidas de interesse definidas (suporte, confiança ou *lift*) [Agrawal and Srikant 1994 Han et al. 2011].

Diversos tipos de informação podem ser descobertos com essas regras: padrões sobre as alterações envolvendo o desenvolvedor, o momento ou o tamanho do *commit*, além de relações entre as métricas e atributos de qualidade.

Os padrões envolvendo o momento de *commit* devem ser utilizados para descobrir informações sobre os dias da semana, turnos de trabalho ou horas. Exemplos desse tipo de padrão com dias da semana são: "na segunda-feira, a chance de o projeto não compilar é maior que nos outros dias" ou "na sexta-feira, a chance de o atributo de qualidade Entendimento diminuir é maior". Exemplos com turnos de trabalho são: "durante a tarde, a complexidade do código aumenta mais do que nos outros turnos" ou "60% dos *commits* da noite são realizados por um determinado desenvolvedor". Ainda, podem-se minerar regras envolvendo o momento do *commit*, como, por exemplo, "às 18h, a chance de o projeto compilar é 20% menor". Vale notar que o enriquecimento da informação do momento do *commit* permite a realização de análises em diferentes granularidades.

Os padrões envolvendo o desenvolvedor permitem descobrir quais os desenvolvedores que mais afetam o projeto e como o fazem. Alguns exemplos são: "quando Eddard realiza modificações, o entendimento diminui", "80% dos *commits* de Arya não compilam" ou "50% dos *commits* do turno da noite são realizados por Jon".

Os padrões envolvendo o tamanho do *commit* mostram a relação entre a quantidade de artefatos alterados e desenvolvedores, momento do *commit* ou atributos de qualidade e métricas. Um exemplo de informação que se pode descobrir é que

"*commits* grandes tendem a compilar menos que os pequenos".

Finalmente, as regras de métricas ou atributos de qualidade evidenciam informações sobre como eles se relacionam. Exemplos de regras desse tipo são: "quando o entendimento aumenta, a reusabilidade diminui" ou "quando a quantidade de linhas de código aumenta, a complexidade também aumenta".

3.3. Fase de Apresentação

O último passo é a apresentação dos padrões descobertos, por meio de regras, gráficos de controle, histogramas e tabelas. Após a mineração de dados, as regras devem ser avaliadas para verificar se fazem sentido *per se* e se podem auxiliar na gerência de projetos. Para auxiliar nessa tarefa, são utilizados filtros baseados em atributos e gráficos históricos das métricas. Os gráficos históricos e histogramas fornecem informações temporais, exibindo respectivamente o comportamento de cada métrica ao longo do tempo e a dispersão dos valores das métricas. Esses gráficos podem ser utilizados para monitorar a qualidade do software durante sua evolução e também para ajudar na interpretação das regras mineradas. Os gráficos históricos são apresentados de duas formas, uma com base nos valores absolutos das métricas e outra com base no delta. O gráfico histórico de delta é particularmente interessante para detectar *commits* que fogem ao padrão histórico.

A Ostra também fornece uma forma mais elaborada de apresentação das informações, chamada de tabela de comportamento. Essa tabela mostra como é a influência de uma métrica nas outras, de acordo com o que é evidenciado nas regras de associação mineradas. A influencia pode ser positiva, negativa ou neutra. O objetivo dessa tabela é facilitar o entendimento das regras de associação que envolvem duas métricas, simples ou compostas, sendo uma métrica no antecedente e outra no consequente.

O gerente de projetos pode utilizar os gráficos históricos e os histogramas para entender melhor sobre a evolução do produto sendo desenvolvido, visando controlar de forma mais eficiente sua qualidade *in time*. Ao analisar esses gráficos durante o desenvolvimento, podem-se identificar valores inesperados das métricas e reagir rapidamente para manter a qualidade do produto. Além disso, com as regras de associação é possível descobrir se existem padrões que envolvem os desenvolvedores ou dias da semana que motivem uma atuação na equipe de desenvolvimento com capacitação ou até mesmo removendo ou premiando alguns de seus membros.

4. Avaliação Experimental

A abordagem proposta neste trabalho tem como objetivo fornecer informações relevantes sobre a evolução do software para o processo de tomada de decisões, visando à qualidade do produto. Para alcançar esse objetivo, a abordagem oferece de três ferramentas: gráficos do histórico das métricas, regras de associação e tabelas de comportamento.

Essas ferramentas foram avaliadas com experimentos de acordo com os seus objetivos. Nesta seção, são apresentados três experimentos. O primeiro teve o objetivo de avaliar se as regras de associação trazem informações relevantes sobre a evolução do software. O segundo experimento teve o objetivo de avaliar a capacidade da Ostra em monitorar a qualidade de um projeto. No terceiro, foi avaliada a utilidade das tabelas de

comportamento na busca de padrões gerais à engenharia de software, utilizando-a para buscar padrões indicados na literatura. Neste artigo, são apresentadas as principais análises realizadas nos experimentos. Os experimentos completos, com seus planejamentos, análises e os resultados encontrados estão disponíveis na dissertação [Ribeiro 2012].

No primeiro experimento, são apresentadas dez regras positivas e dez regras negativas extraídas do histórico de quatro projetos: dois proprietários e dois de código aberto. Uma regra positiva é aquela que explicita um comportamento desejável encontrado durante o desenvolvimento do projeto. Analogamente, uma regra negativa é aquela que apresenta um comportamento que não é desejável e, consequentemente, ações devem ser tomadas para revertê-lo. Neste artigo serão apresentadas apenas análises realizadas com o projeto IdUFF, pois ele possui a maior quantidade de revisões dentre os projetos analisados.

O projeto IdUFF é o sistema acadêmico da Universidade Federal Fluminense e é do tipo cliente-servidor, codificado em Java com interface web. Esse projeto possuía quatro anos de desenvolvimento e até o momento da medição, tinha um total de 1.509 *commits* com alteração em arquivos Java, dos quais 1.355 compilavam. Esse projeto é composto de 1.068 artefatos e conta com 31 desenvolvedores. A última revisão medida foi a de número 22.695.

Para analisar o histórico dos projetos com regras de associação, foram utilizadas cinco métricas que pudessem ser entendidas mais facilmente, mesmo por pessoas que não fossem especialistas em engenharia de software: Complexidade Ciclomática de McCabe, Reusabilidade, Entendimento, Linhas de Código e Tamanho do Projeto em Classes. Junto às medidas coletadas para as cinco métricas, também foram utilizadas informações de cada *commit*: desenvolvedor, se a versão resultante compila ou não, o dia da semana, o turno de trabalho, a hora e a quantidade de artefatos alterados. O suporte mínimo para minerar as regras foi de 1% e o *lift* maior que 1,0. A confiança foi utilizada para ordenar as regras, mas não para descartá-las.

Para manter a confidencialidade em relação aos desenvolvedores, optou-se por não os identificar. Dessa forma, os nomes dos desenvolvedores foram substituídos por nomes de personagens de "As Crônicas de Gelo e Fogo" [Martin 1996].

As regras positivas encontradas com a mineração de dados do projeto IdUFF são apresentadas na Tabela 3. A quarta regra, por exemplo, mostra que quando o projeto cresce em número de classes, a reusabilidade também aumenta. Isso mostra que a preocupação da equipe com a reusabilidade do projeto teve bons resultados. A oitava regra, mostra que quando Stannis faz alterações, a complexidade do código diminui com uma chance 20% maior que o esperado. Essa regra pode ser explicada, pois Stannis é um desenvolvedor experiente e uma das suas responsabilidades era realizar refatorações no código. A nona regra mostra que, quando Tywin faz alterações, a chance de compilar é 11% maior que a chance geral de compilar. Isso acontece, pois esse desenvolvedor, apesar de inexperiente, se preocupava em verificar se o que tinha sido implementado estava correto e não iria atrapalhar os outros desenvolvedores desde quando entrou na equipe.

Enquanto as regras positivas indicam comportamentos benéficos ou neutros à manutenção do projeto, as regras negativas indicam comportamentos nocivos à sua qualidade. As regras negativas do projeto IDUFF são apresentadas na Tabela 4.

Tabela 3: Regras Positivas do Projeto IdUFF.

#	Regra	Sup.	Conf.	Lift
1	Se a quantidade de classes aumenta, então nove ou mais artefatos são alterados.	0,14	0,72	3,1
2	Se a quantidade de linhas de código diminui, então o entendimento aumenta.	0,03	0,20	2,8
3	Se a reusabilidade diminui, então o entendimento aumenta.	0,02	0,17	2,4
4	Se a quantidade de classes aumenta, então a reusabilidade aumenta.	0,15	0,77	2,2
5	Se Tywin faz alterações, então três ou quatro artefatos são alterados.	0,01	0,28	2,0
6	Se Stannis faz alterações, então nove ou mais artefatos são alterados.	0,01	0,39	1,7
7	Se Robb faz alterações, então a complexidade diminui.	0,01	0,43	1,4
8	Se Stannis faz alterações, então a complexidade diminui.	0,01	0,36	1,2
9	Se Tywin faz alterações, então compila.	0,03	1,00	1,1
10	Se Renly faz alterações, então compila.	0,02	1,00	1,1

Tabela 4: Regras negativas do Projeto IdUFF.

#	Regra	Sup.	Conf.	Lift
1	Se a quantidade de classes diminui, então a reusabilidade diminui.	0,01	0,75	4,6
2	Se Jofrey faz alterações, então não compila.	0,01	0,32	3,2
3	Se Stannis faz alterações, então a quantidade de linhas de código diminui.	0,01	0,41	2,6
4	Se Tyrion faz alterações, então a complexidade aumenta.	0,00	1,00	2,3
5	Se o commit é na sexta-feira, então a reusabilidade diminui.	0,02	0,30	2,0
6	Se a quantidade de classes aumenta, então o entendimento diminui.	0,18	0,94	2,0
7	Se o commit é na sexta-feira, então não compila.	0,15	0,11	1,5
8	Se Renly faz alterações, então a complexidade aumenta.	0,01	0,50	1,3
9	Se Tommem faz alterações, então a complexidade aumenta.	0,03	0,45	1,2
10	Se o commit é no turno da noite, então não compila.	0,02	0,12	1,2

Enquanto a oitava regra positiva mostra que quando Stannis faz alterações, a chance de compilar aumenta, a segunda regra negativa mostra que quando Jofrey faz alterações, a revisão resultante não compila em 32% das vezes. Apesar de esse desenvolvedor ser experiente, provavelmente, por um excesso de confiança, ele não verifica se as suas alterações mantêm o projeto compilando antes de realizar o *commit*, resultando assim no aumento da chance de não compilar em três vezes o esperado. A sexta regra negativa mostra que, à medida que o projeto aumenta em quantidade de classes, o entendimento diminui. É esperado que, à medida que a quantidade de classes aumente, o entendimento diminua, pois se torna mais difícil conhecer o projeto e entendê-lo [Bansiya and Davis 2002]. A sétima regra indica um padrão temporal, ressaltando que na sexta-feira a chance de compilar é menor. Neste projeto, sexta-feira é um dia marcado por reuniões com cliente e entre os membros da equipe de desenvolvimento. Provavelmente, *commits* nesse dia são realizados com pressa e com menos atenção, resultando em uma taxa de compilação menor.

Foi realizado outro experimento para avaliar a capacidade dos gráficos históricos

em fornecer informações sobre os *commits* realizados no projeto. Nesse experimento, foi possível encontrar indícios de que a Ostra auxilia no monitoramento da qualidade, pois facilita a avaliação da alteração da qualidade em cada *commit*. Na Figura 1.a, é apresentado o gráfico do histórico das alterações realizadas à métrica TCC no software IdUFF. Com esse gráfico, é possível perceber que alguns *commits* fogem ao padrão do projeto. Por exemplo, é possível observar que no *commit* 2.528, identificado pelo ponto 1, foi adicionada uma quantidade de linhas de código muito além do padrão desse software, o que afetou a complexidade total do projeto. Porém, apesar de adicionar tantas linhas de código, esse *commit* não afetou negativamente a reusabilidade, como mostrado no gráfico dessa métrica apresentado na Figura 1.b. No *commit* 3.336, indicado pelo ponto 2, foram adicionadas 18 classes e isso causou uma variação pequena no gráfico de TCC, além da linha amarela (acima de um desvio padrão), mas causou pouca variação no gráfico da reusabilidade. Por outro lado, no *commit* 4.201, indicado pelo ponto 3, foi feita uma refatoração que afetou 158 artefatos (arquivos e pastas) e removeu 57 classes. Isso causou uma diminuição muito grande na métrica TCC (além da linha vermelha) e teve um impacto grande e positivo na reusabilidade, como indicado na Figura 1. Ambos os gráficos apresentados na Figura 1 foram gerados com a abordagem proposta.

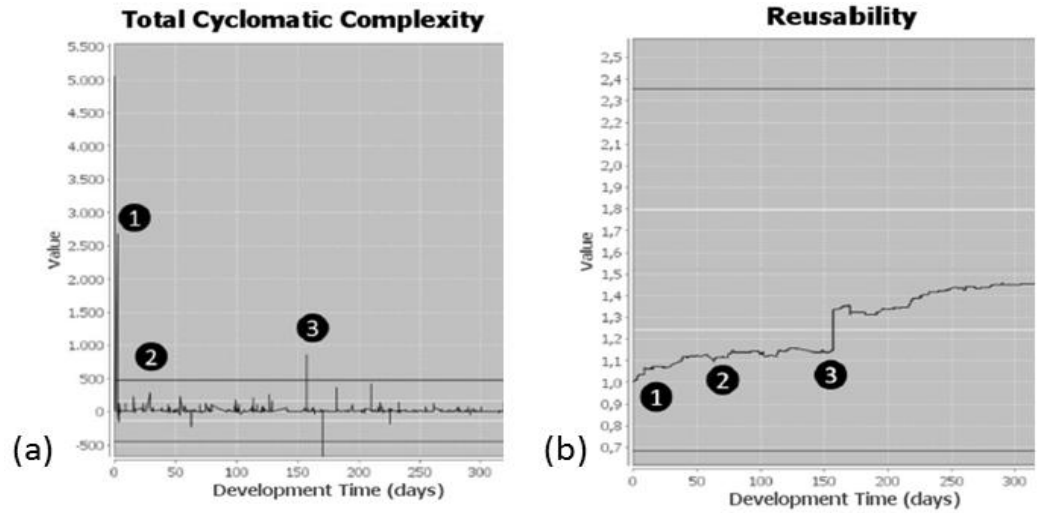

Figura 1: (a) Gráfico da variação de TCC, (b) gráfico absoluto da reusabilidade.

Foi realizado ainda um terceiro experimento, no qual a Ostra foi utilizada para buscar por padrões que correlacionam atributos de qualidade. Um exemplo de correlação é que a manutenibilidade afeta positivamente a flexibilidade [Wiegers 2003], ou seja, que quando um aumenta o outro também aumenta. Neste experimento, inicialmente, foram identificados os atributos de qualidade de Wiegers [2003] presentes na Ostra e, então, se buscou pelos comportamentos indicados por Wiegers [2003]. Foram realizadas duas análises: uma em que os projetos foram analisados de forma global e outra em que os projetos foram agrupados de acordo com características buscando-se por padrões que se aplicam apenas a determinados tipos de projetos. Inicialmente, foram geradas as tabelas de comportamento da Ostra com as seguintes métricas para 16 projetos: integridade, manutenibilidade, flexibilidade, reusabilidade e testabilidade. Na tabela de comportamento, a cor cinza claro indica que o atributo da linha afeta positivamente o

atributo da coluna e a cor cinza escuro indica que o atributo da linha afeta negativamente o da coluna. Para exemplificar, a tabela de comportamento gerada para o projeto *Maven GWT Plugin* é apresentada na Figura 2. Com base nessa tabela, pode-se perceber que o comportamento que indica que a manutenibilidade (Man) afeta positivamente a flexibilidade (Fle) foi confirmado nesse projeto.

	Int	**Man**	**Fle**	**Reu**	**1/Test**
Int		2,57	4,87	2,73	2,86
Man	2,57		3,73	2,71	2,93
Fle	4,87	3,73		3,27	2,24
Reu	2,73	2,71	3,27		1,88
1/Test	2,86	2,93	2,24	1,88	

Afeta Positivamente Afeta Negativamente

Figura 2: Tabela de Comportamento gerada pela abordagem Ostra.

Após a construção das tabelas de comportamento, a quantidade de vezes que o comportamento positivo e negativo apareceram nas tabelas de comportamento de cada projeto foi contabilizada. Na Figura 3, são apresentadas as tabelas que sumarizam esses dados. O significado das cores da tabela segue o indicado na Figura 2, onde o número na posição superior indica em quantos projetos o comportamento apareceu e o número inferior, entre parênteses, indica em quantos projetos foi encontrado um comportamento contrário ao mostrado na célula. Por exemplo, na Figura 3.a, é apresentado que o comportamento entre reusabilidade e flexibilidade é positivo, ou seja, quando a reusabilidade aumenta, a flexibilidade também aumenta. Esse comportamento foi confirmado por quatorze projetos e negado por apenas um.

(a) Indicado na literatura	**Int**	**Man**	**Fle**	**Reu**	**Test**
Int				2 (-13)	
Man			15 0		7 (-7)
Fle	0 (-15)	15 0			8 (-6)
Reu	2 (-13)	12 (-3)	14 (-1)		11 (-2)
Test	10 (-3)	7 (-7)	8 (-6)		

(b) Não indicado na literatura	**Int**	**Man**	**Fle**	**Reu**	**Test**
Int			15 0		10 (-3)
Man				12 (-3)	
Fle				14 (-1)	
Reu					
Test				11 (-2)	

Figura 3: Tabelas sumarizando os padrões encontrados nos 16 projetos.

Na segunda parte deste experimento, os projetos foram agrupados de acordo

com algumas características para tentar identificar se existem comportamentos que ocorrem apenas em alguns tipos de projetos. As seguintes características foram consideradas para agrupar os projetos: utilização, empacotamento, tipo, tamanho da equipe e tamanho do projeto em linhas de código. Ao analisar os comportamentos nesses grupos, foi possível encontrar indícios de que alguns padrões acontecem somente para determinados tipos de projetos. Um exemplo é que para todos os projetos com equipes grandes analisados, a flexibilidade afeta positivamente a reusabilidade. Mas, isso não pôde ser confirmado para projetos com equipes pequenas ou médias. Entretanto, apenas para equipes médias, foram encontrados indícios que a testabilidade afeta negativamente a reusabilidade. Na Figura 4, são mostrados os comportamentos encontrados de acordo com os tamanhos das equipes dos projetos. As equipes dos projetos foram classificadas considerando pequenas as que possuem até 10 desenvolvedores, médias as que possuem entre 10 e 15 desenvolvedores, e grandes as que possuem mais de 15 desenvolvedores.

Equipe Pequena	Int	Man	Fle	Reu	Test
Int			9 0	1 (-8)	
Man			9 0		2 (-6)
Fle	0 9	9 0			3 (-5)
Reu	1 (-8)	8 (-1)	8 (-1)		1 (-6)
Test	1 (-6)	2 (-6)	3 (-5)		

Equipe Média	Int	Man	Fle	Reu	Test
Int		3 0	3 0	1 (-2)	
Man	3 0		3 0		2 (-1)
Fle	0 (-3)	3 0			2 (-1)
Reu	1 (-2)	1 2	3 0		0 (-3)
Test	1 (-3)	2 (-1)	2 (-1)	3 0	

Equipe Grande	Int	Man	Fle	Reu	Test
Int			3 0	0 (-3)	
Man			3 0	3 0	3 0
Fle	0 (-3)	3 0		3 0	3 0
Reu	0 (-3)	3 0	3 0		1 (-2)
Test	1 (-1)	3 0	3 0		

Figura 4: Tabelas sumarizando os padrões de acordo com o tamanho da equipe: pequena, média e grande.

5. Conclusão

Neste trabalho foi proposta uma abordagem que tem como objetivos: (1) fornecer informações, no contexto de evolução de software, relevantes à tomada de decisões gerenciais, (2) auxiliar no monitoramento da qualidade do projeto e (3) identificar padrões evolutivos em relação a um conjunto de projetos. Para alcançar esses objetivos a abordagem se baseia nas disciplinas de Gerência de Configuração, Métricas de Software e Mineração de Dados. Foi desenvolvido um protótipo que implementa a abordagem proposta e foram realizados experimentos para avaliar a abordagem.

Com os experimentos, foi possível encontrar indícios que a abordagem Ostra é capaz de obter informações relevantes ao processo de tomada de decisão, fornecendo informações sobre a evolução da qualidade do software realizada em cada *commit*. Essas informações podem ser usadas para evidenciar comportamentos de desenvolvedores ou mostrar como os atributos de qualidade ou métricas variam em relação a alguns fatores

durante a evolução do software. Dessa forma, pôde-se apresentar a utilização da abordagem e responder positivamente às duas questões de pesquisa identificadas: (1) “a mineração de dados é capaz de obter informações relevantes sobre a evolução do projeto?" e (2) "a mineração de regras de associação é capaz de obter informações relevantes sobre o relacionamento entre métricas de software?".

Consequentemente, acredita-se que a abordagem proposta possa auxiliar no entendimento da evolução do software, fornecendo informações que podem ser utilizadas por equipes de desenvolvimento, gerentes de projeto, escritórios de projetos ou pesquisadores de engenharia de software. As aplicações vislumbradas podem ser divididas em três perspectivas: do gerente do projeto, objetivando controlar melhor o projeto e entender sobre ele para gerenciá-lo de forma mais eficiente; do escritório de projetos, que deseja realizar medições e descobrir os padrões dos seus projetos; e de um pesquisador de engenharia de software, cujo objetivo seja descobrir padrões gerais de desenvolvimento de software de forma a contribuir para a sua área de estudo.

Em resumo, as principais contribuições deste trabalho são: (1) a definição de uma abordagem que utiliza bases de dados de modificações de software, representada pela variação de métricas, para realizar mineração de regras de associação com a finalidade de encontrar padrões que auxiliem no processo de tomada de decisões referentes ao desenvolvimento de software; (2) a representação através da tabela de comportamento de como as métricas se relacionam; (3) a utilização de gráficos de controle e histogramas, visando acompanhar o histórico das métricas e analisar as modificações realizadas ao longo da evolução do software; e (4) a disponibilização de uma infraestrutura para a realização de trabalhos futuros envolvendo: acesso a repositórios de software, construção com sistema de gerenciamento de construção, medição automática e mineração de regras de associação.

Agradecimentos

Agradecemos à CAPES, CNPQ e FAPERJ pelo apoio financeiro, à STI/UFF por permitir que seus projetos de software fossem utilizados nos experimentos, e também ao Wallace Ribeiro, por ter contribuído na implementação da coleta e análise de métricas.

Referências

Agrawal, R. and Srikant, R. (1994). Fast Algorithms for Mining Association Rules in Large Databases. In *International Conference on Very Large Data Bases*. VLDB 1994. Morgan Kaufmann.

Apache Software Foundation (2011). *Apache Maven*. Apache Software Foundation.

Bansiya, J. and Davis, C. G. (2002). A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, v. 28, p. 4–17.

Colares, F., Souza, J., Carmo, R., Pádua, C. and Mateus, G. R. (2009). A New Approach to the Software Release Planning. In *Proceedings of the 2009 XXIII Brazilian Symposium on Software Engineering*. , SBES 2009. IEEE Computer Society.

Collins-Sussman, B., Fitzpatrick, B. W. and Pilato, C. M. (2008). *Version Control with Subversion*. Sebastpol, CA, USA: O’Reilly Media. v. 2

Dart, S. (1991). Concepts in configuration management systems. In *Proceedings of the 3rd international workshop on Software configuration management*. SCM ’91. ACM.

Dick, S., Meeks, A., Last, M., Bunke, H. and Kandel, A. (2004). Data mining in software metrics databases. *Fuzzy Sets and Systems*, v. 145, n. 1, p. 81–110.

Erlikh, L. (2000). Leveraging Legacy System Dollars for E-Business. *IT Professional*, v. 2, p. 17–23.

Han, J., Kamber, M. and Pei, J. (2011). *Data Mining: Concepts and Techniques, Third Edition*. 3. ed. San Francisco, CA, USA: Morgan Kaufmann.

Henderson-Sellers, B. (1995). *Object-oriented metrics: measures of complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

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

Júnior, M. C., Mendonça, M. and Rodrigues, F. (2009). Mining Software Change History in an Industrial Environment. In *Proceedings of the 2009 XXIII Brazilian Symposium on Software Engineering*. SBES 2009. IEEE Computer Society.

Kim, S., Zimmermann, Thomas, Pan, K. and Whitehead, E. J. (2006). Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2006. IEEE Computer Society Press.

Martin, G. R. R. (1996). *A Game of Thrones*. 1. ed. New York, NY, USA: Bantam.

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, v. 2, p. 308–320.

Nagappan, N., Ball, T. and Zeller, Andreas (2006). Mining metrics to predict component failures. In *Proceedings of the International Conference on Software Engineering Advances*. ICSE 2006. ACM.

Pressman, R. (2001). *Software Engineering - A Practitioner's Approach*. 5. ed. New York, NY, USA: McGraw-Hill Higher Education.

Ribeiro, D. D. C. (2012). Ostra: Um Estudo do Histórico da Qualidade do Software Através de Regras de Associação de Métricas. Universidade Federal Fluminense - UFF.

Robles, G., Gonzalez-Barahona, J. M., Michlmayr, M. and Amor, J. J. (2006). Mining large software compilations over time: another perspective of software evolution. In *Proceedings of the 2006 International Workshop Conference on Mining Software Repositories*. MSR 2006. ACM.

Wermelinger, M. and Yu, Y. (2008). Analyzing the evolution of eclipse plugins. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. MSR 2008. ACM.

Wiegers, K. (2003). *Software Requirements*. 2. ed. Redmond, Washington: Microsoft Press.

Witten, I. H., Frank, E. and Hall, M. A. (2011). *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*. 3. ed. Morgan Kaufmann.

Zimmermann, T., Weisgerber, P., Diehl, S. and Zeller, A. (2004). Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*. ICSE 2004. IEEE Computer Society.