DANIEL DRUMOND CASTELLANI RIBEIRO

OSTRA: UM ESTUDO DO HISTÓRICO DA QUALIDADE DO SOFTWARE ATRAVÉS DE REGRAS DE ASSOCIAÇÃO DE MÉTRICAS

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Engenharia de Software.

Orientador: Prof. Dr. LEONARDO GRESTA PAULINO MURTA
Co-orientador: Prof. Dr. ALEXANDRE PLASTINO

Niterói
2012

DANIEL DRUMOND CASTELLANI RIBEIRO

OSTRA: UM ESTUDO DO HISTÓRICO DA QUALIDADE DO SOFTWARE ATRAVÉS DE REGRAS DE ASSOCIAÇÃO DE MÉTRICAS

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Engenharia de Software.

Aprovada em Junho de 2012.

BANCA EXAMINADORA

_____

Prof. Dr. LEONARDO GRESTA PAULINO MURTA – Orientador
UFF

_____

Prof. Dr. ALEXANDRE PLASTINO – Co-orientador
UFF

_____

Prof. Dr. VIVIANE TORRES DA SILVA
UFF

_____

Prof. Dr. ALESSANDRO FABRICIO GARCIA
PUC - Rio

Niterói
2012

A minha avó, meus pais e orientadores.

AGRADECIMENTOS

Agradeço a Deus, em primeiro lugar, por tornar tudo possível, ao criar um mundo cheio de vida e cores que nos fazem seguir motivados e com a obrigação de sermos melhores a cada dia.

Aos amigos espirituais, enviados por Deus, que me auxiliaram a me manter no caminho do bem e não dispersar. Em muitos momentos da caminhada, vocês foram a força e o sentido que me trouxeram até aqui. Sem o esforço imensurável de vocês eu não teria conquistado o que consegui.

A minha família, que sempre me apoiou, incentivou e tornou possível que eu chegasse até a conclusão do mestrado. A minha avó, Jovelina, e meus tios, Paulo César e Eliana, que seguem sempre me dando bons conselhos. Em especial, aos meus pais, Sandra Rita e Josane, que sempre estiveram ao meu lado, me incentivando, dando forças e todo o necessário para que eu seguisse em frente. Sem vocês nada seria possível!

A Mônica Fabíola, minha namorada, que me apoiou e motivou em todos os momentos, sabendo me animar nas horas de desânimo e dando forças para que o mestrado se concretizasse. Você foi fundamental em mais essa conquista.

Aos meus orientadores, Alexandre Plastino e Leonardo Murta, que foram incansáveis na orientação, me ensinando que a qualidade, a honestidade, a seriedade e a ética não podem, nunca, ficar de lado. Podem ter certeza que vocês me orientaram não só no mestrado, mas também na vida. Obrigado por me incentivar e motivar nos momentos que pensei em desistir.

Aos colegas do GEMS, pelas conversas produtivas que auxiliaram na melhora da Ostra, em especial ao Heliomar Kann e Gleiph Ghiotto, sem os quais o Oceano não teria se concretizado.

Ao aluno de iniciação científica e colega de desenvolvimento, Wallace Ribeiro, que contribuiu significativamente não só nas métricas, mas na Ostra como um todo.

Aos companheiros de trabalho, da STI e do Evangelho nas Ruas, Thiago Diogo, Bruno Olímpio e Estevão Goulart, dentre outros, que acreditaram em mim e me apoiaram quando precisei me ausentar para dar foco ao mestrado.

Aos amigos que souberam compreender minhas ausências e não desistiram de mim, mas continuaram sempre me dando forças para continuar. E, até quando não entendiam do que se tratava minha pesquisa, diziam ser interessante.

A Superintendência de Tecnologia da Informação (STI) que permitiu que seus softwares fossem utilizados nos experimentos.

Aos professores e funcionários do Instituto de Computação.

A CAPES pelo apoio financeiro.

EPÍGRAFE

"Comece por fazer o que é necessário, depois o que é possível e de repente estará a fazer o impossível." - São Francisco de Assis

RESUMO

O ciclo de vida de um software pode ser dividido em três fases: definição, desenvolvimento e manutenção. Dessas três, a manutenção consome cerca de 60% do tempo e 90% do custo do projeto. Porém, os atributos de qualidade flexibilidade e manutenibilidade, que avaliam a capacidade de manutenções evolutivas e corretivas, são definidos desde a fase de desenvolvimento. Consequentemente, deve-se monitorar os atributos de qualidade desde a fase de desenvolvimento para garantir a qualidade do produto. Dessa forma, para controlar a qualidade do software e não apenas reagir à sua variação, deve-se entender sobre quais fatores influenciam os atributos de qualidade e como eles variam entre si. Neste trabalho, é proposta a abordagem Ostra, que permite a análise do histórico de um software, armazenado no sistema de controle de versão, através da variação de métricas de software. Os objetivos da Ostra são: (1) fornecer informações ao processo de tomada de decisões, (2) monitorar a qualidade do software e (3) buscar padrões entre métricas presentes em um ou vários projetos. Para alcançar esses objetivos, a Ostra se baseia em três disciplinas: métricas de software, gerência de configuração e mineração de dados. Para apresentar as informações encontradas, são utilizados: gráficos de controle, histogramas, regras de associação e uma tabela de comportamentos. Para avaliar a abordagem, foram utilizados 16 projetos em três experimentos distintos, com os quais foi possível obter indícios de que os objetivos da abordagem são alcançados para projetos de diferentes tamanhos e finalidades. As contribuições deste trabalho são: (1) uma abordagem que considera as alterações realizadas ao longo da evolução do software como elemento básico de análise, as quais são descritas como a variação de métricas, (2) experimentos avaliando as questões de pesquisa, (3) uma infraestrutura para futuras pesquisas sobre mineração de repositórios de software e (4) uma base de dados com a medição de cerca de 30 métricas sobre 150 projetos.

Palavras-chave: Gerência de Configuração, Métricas de Software, Qualidade de Software, Monitoramento da Qualidade, Mineração de Dados, Repositório de Software, Medição Automática e Processo de Tomada de Decisões.

ABSTRACT

The software life cycle can be divided into three phases: definition, development, and maintenance. The maintenance phase consumes 60% of the time and 90% of the cost of the whole life cycle. However, the flexibility and maintainability quality attributes, which evaluate the corrective and evolutive maintenance capability, are defined since the development phase. Consequently, it is expected to monitor the quality attributes since the development phase to guarantee the product quality. Thus, it is necessary understand which factors influence the quality attributes and how they vary between each other to control the software quality, not only reacting to its variation. In this work, is proposed the Ostra approach, that allows the historical analysis of a software, stored in a version control system, through the variation of its metrics. The Ostra's goals are: (1) provide information to decision-making process, (2) monitor software quality, and (3) search patterns between metrics in one or various software projects. To achieve these goals, the Ostra is based in three disciplines: software metrics, configuration management, and data mining. Ostra uses the following visual elements to support analysis: control graphs, histograms, association rules, and a behavior table. Our approach was evaluated over 16 projects in three different experiments, with which were possible find evidences that the Ostra's goals are achieved to projects of different sizes and purposes. The contributions of this work are: (1) an approach that considers the changes made during software evolution as first class elements of analysis, which are described as the variation of software metrics, (2) experiments evaluating the research questions, (3) an infrastructure to future research about mining in software repositories, and (4) a database with measurement of around 30 metrics over 150 projects.

Keywords: Configuration Management, Software Metrics, Software Quality, Quality Monitoring, Data Mining, Software Repository, Automatic Measurement, and Decision Making Process.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

DAO - *Data Access Object*

IDE - *Integrated Development Environment*

IdUFF - Sistema de Identificação Única da UFF

ISO - *International Organization for Standardization*

GC - Gerência de Configuração

GEMS - Grupo de Evolução e Manutenção de Software

GWT - *Google Web Toolkit*

JPA - *Java Persistent API*

QMOOD - *Quality Model for Object Oriented Design*

JSF - *Java Server Faces*

SCV - Sistema de Controle de Versões

SCM - Sistema de Controle de Mudanças

SGC - Sistema de Gerenciamento de Construção

STI - Superintendência de Tecnologia da Informação

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO

1.1 MOTIVAÇÃO

O ciclo de vida de um software pode ser dividido em três fases de acordo com seus focos (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 é a definição, durante a fase de desenvolvimento, da flexibilidade e da manutenibilidade do software, que são atributos de qualidade que definem a facilidade de se estender ou consertar o software (WIEGERS, 2003). Isso acontece, pois quanto maior a qualidade do produto desenvolvido, menos manutenções corretivas serão necessárias e mais facilmente o software será evoluído. 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.

Atualmente, tem-se dado muita atenção a processos de software, baseando-se na premissa de que processos de qualidade produzem produtos de qualidade. Dessa forma, verifica-se um aumento na adoção de modelos de maturidade (TRAVASSOS; KALINOWSKI, 2011). Porém, mesmo com esses processos, é fundamental que o produto seja monitorado para garantir sua qualidade. Por exemplo, mesmo em fábricas que possuem um processo de fabricação maduro, a qualidade do produto gerado é rigidamente monitorada. Com o desenvolvimento de software não pode ser diferente.

Métricas são utilizadas para descrever constantemente a qualidade de objetos, como, por exemplo, "um azeite extra virgem possui acidez de 0,4%". 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. Dessa forma, torna-se fundamental descobrir o que se esperar durante a evolução do software e como agir quando limiares forem ultrapassados para determinadas métricas.

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. Concluindo, para que o gerente do 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.

1.2 OBJETIVO

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, com o objetivo principal fornecer informações relevantes que aumentem o conhecimento sobre a evolução do software. A partir desse objetivo principal, se desdobram os seguintes objetivos: (1) fornecer informações relevantes à tomada de decisões gerenciais, (2) auxiliar no monitoramento da qualidade do projeto, (3) identificar padrões evolutivos em relação a um conjunto de projetos, por exemplo, projetos de uma organização, ou mesmo padrões gerais de desenvolvimento de software. A abordagem proposta neste trabalho se chama Ostra[1].

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.

Desta forma, para se alcançar os três objetivos citados anteriormente, a Ostra se baseia em três disciplinas: métricas de software, gerência de configuração e mineração de dados. As informações extraídas e mineradas são apresentadas e visualizadas através de: (1) regras de associação, (2) histogramas, (3) gráficos de controle para acompanhar o histórico das métricas e (4) tabelas de comportamento, mostrando como as métricas se comportam em relação à variação de outras métricas.

[1] As ostras são moluscos que crescem, em sua maioria, em água salgada. Elas têm um corpo mole que se protege dentro de uma concha calcificada (WIKIPÉDIA, 2010). A ostra tem uma forma curiosa de se defender: quando um parasita invade seu corpo, ela libera uma substância, a madrepérola, que não permite ao invasor se reproduzir e, mais tarde, cerca de três anos, este material se transforma em uma pérola. A ostra, dessa forma, consegue gerar uma pérola, algo extremamente valioso, de um parasita, que a ameaçava. O nome dado à abordagem proposta neste trabalho foi Ostra, pois se deseja gerar informações valiosas sobre as métricas de software.

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.

Para avaliar as questões de pesquisa e os objetivos da Ostra, foram realizados três experimentos. Com esses experimentos, ao menos para um conjunto de projetos, foi possível encontrar respostas positivas para as questões de pesquisa. Consequente, observou-se o êxito da Ostra em alcançar seus objetivos com projetos de diferentes equipes, tamanhos e tipos.

1.3 ORGANIZAÇÃO

Esta dissertação está organizada em seis capítulos. No Capítulo 2, é realizada uma revisão da literatura, visitando os conceitos básicos necessários ao entendimento do trabalho, tais como: mineração de dados, métricas de software e gerência de configuração. Além disso, também é apresento o estado da arte da área de mineração de repositórios de software. No Capítulo 3, é apresentada a abordagem proposta e, no Capítulo 4, a arquitetura do protótipo implementado. No Capítulo 5, são mostrados experimentos realizados para responder as questões de pesquisa discutidas: (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?". O experimento da Seção 5.2, responde a primeira questão de pesquisa, enquanto o experimento da Seção 5.4 responde a segunda questão de pesquisa. O experimento da Seção 5.3 avalia a capacidade da Ostra de monitorar a qualidade do software. No Capítulo 6, é apresentada a conclusão do trabalho, destacando as contribuições, limitações e trabalhos futuros.

CAPÍTULO 2 – MINERAÇÃO DE REPOSITÓRIOS DE SOFTWARE

2.1 INTRODUÇÃO

A Gerência de Configuração é uma disciplina que torna possível uma 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, ou seja, com desenvolvedores em locais diferentes e atuando ao mesmo tempo no mesmo projeto.

Durante a evolução do software, são geradas grandes quantidades de dados pelos Sistemas de Gerência de Configuração que podem ser explorados por técnicas de Mineração de Dados para descobrir informações valiosas. Técnicas de Mineração de Dados podem ser utilizadas para encontrar mudanças que introduzam erros no software (KIM *et al.*, 2006), para descobrir qual a melhor forma de ordenar os *releases* (COLARES *et al.*, 2009), e até para descobrir quais artefatos de um software também devem ser alterados quando ocorrer uma alteração num determinado artefato (ZIMMERMANN, T. *et al.*, 2004).

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. Assim, ao estudar a relação entre as métricas pode-se melhorar o entendimento do software e de sua evolução. Esta ideia já inspirou trabalhos com objetivo de aumentar o entendimento sobre as métricas (DICK *et al.*, 2004) e descobrir se um software possui defeitos (NAGAPPAN; BALL; MURPHY, 2006).

Neste capítulo, são apresentados os conceitos que dão base à mineração de repositórios de software e análise da qualidade através de métricas de software que se desdobram em atributos de qualidade. Primeiramente, são apresentados os conceitos de mineração de dados, gerência de configuração e métricas de software. Também são apresentados conjuntos de métricas e atributos de qualidade indicados na literatura. Após apresentar os conceitos necessários para o entendimento deste trabalho, são apresentados trabalhos relacionados, mostrando outros estudos de mineração de repositórios de software e análise de atributos de qualidade. Para finalizar, é apresentada uma lacuna deixada pelos trabalhos já realizados que é explorada neste trabalho.

2.2 MINERAÇÃO DE DADOS

O processo de Descoberta de Conhecimento em Bases de Dados, do inglês *Knowledge Discovery in Databases* (KDD), possui como principal objetivo extrair informações de bases de dados, viabilizando melhor conhecimento dos dados para dar suporte à tomada de decisões (HAN *et al.*, 2011; WITTEN; FRANK, 2005). O processo de KDD algumas vezes é confundido com Mineração de Dados (MD), sua fase principal, cujo objetivo é extrair informação útil e previamente desconhecida de grandes bases de dados na forma de regras ou padrões. Mineração de Dados é utilizada em diversas aplicações para obter vantagens estratégicas: econômicas, comerciais, de marketing, médicas, educacionais e biológicas.

O processo de KDD possui cinco fases (FAYYAD *et al.*, 1996): seleção, pré-processamento, transformação, mineração de dados, e avaliação e interpretação. Esse processo é ilustrado na Figura 1.

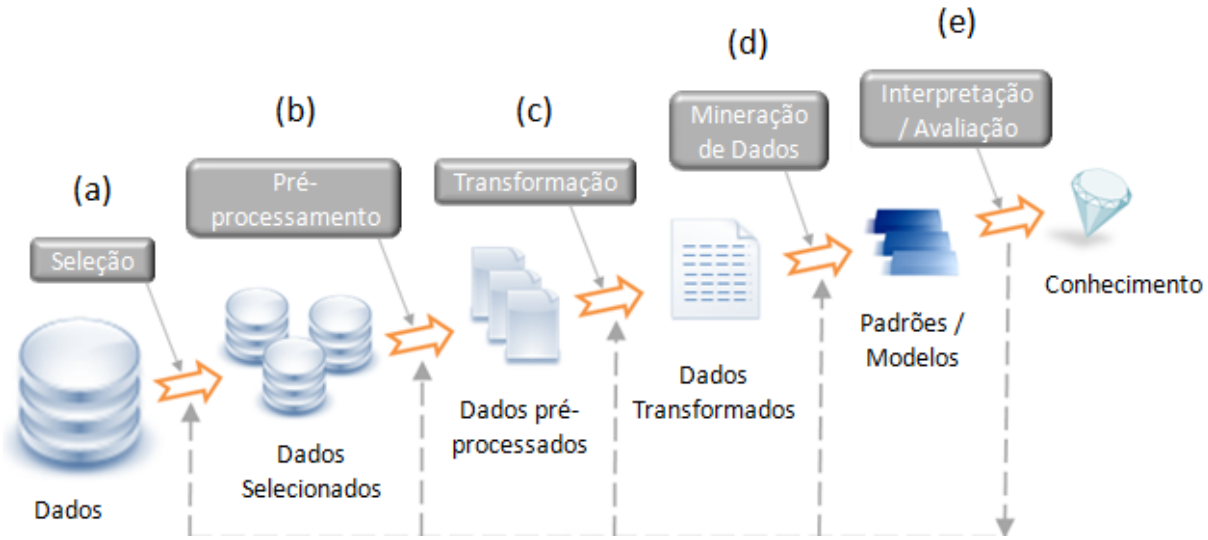


Figura 1: Fases do processo de KDD (FAYYAD *et al.*, 1996).

Na primeira fase, de seleção, exibida na Figura 1.a, os dados que serão analisados são selecionados, dando origem ao conjunto de dados que será utilizado nas próximas fases. Como bases de dados reais têm grande probabilidade de serem inconsistentes e com ruído (HAN *et al.*, 2011), na fase de pré-processamento, exibida na Figura 1.b, a qual é subdividida em limpeza e enriquecimento, os dados devem ser tratados, completando dados faltosos e corrigindo valores provenientes de ruído. Se a informação está representada de maneira que os algoritmos de mineração de dados não conseguem lidar[2], os dados devem ser transformados. Isto acontece na fase de transformação, exibida na Figura 1.c. Na quarta fase,

[2] Por exemplo, o algoritmo Apriori (AGRAWAL; SRIKANT, 1994) não consegue lidar com dados numéricos e contínuos. Consequentemente, se é necessário minerar uma base de dados que contém valores contínuos, seus valores devem ser discretizados.

com uma base de dados que já possui dados selecionados, limpos, enriquecidos e transformados, podem ser utilizadas técnicas de mineração de dados com a finalidade de extrair padrões ou regras. A fase de mineração de dados está representada na Figura 1.d. Após ser minerado, cada padrão (ou regra) extraído deve ser interpretado e avaliado, verificando se realmente é de interesse e valoroso. Assim, termina o processo de KDD na fase de avaliação e interpretação dos dados, exibida na Figura 1.e.

As técnicas de Mineração de Dados extraem diferentes tipos de informação, dentre os quais, destacam-se: regras de associação, padrões sequenciais, modelos de classificação e agrupamentos de dados (HAN *et al.*, 2011).

Regras de Associação (RA) mostram relacionamentos entre itens no domínio que possuem ocorrência significativa na base de dados. Exemplos comuns de RAs são produtos comprados juntos numa visita ao supermercado.

Padrões sequenciais são semelhantes às regras de associação, porém com um significado temporal. Para minerar este tipo de padrão, a base de dados deve ter informações de ordem ou temporais. Um exemplo deste tipo de padrão, numa base de dados de locações de filmes, poderia ser: se alguém aluga os filmes "O Senhor dos Anéis 1" e "O Senhor dos Anéis 2", então aluga "O Senhor dos Anéis 3", nesta ordem.

Quando é necessário inferir a classe de um objeto de acordo com seus atributos, técnicas de classificação são utilizadas. Consequentemente, esse tipo de técnica necessita de uma base de dados de exemplos, com classes conhecidas, que é utilizada para treinar um classificador. Um exemplo de aplicação da técnica de classificação é a detecção de SPAM. Um servidor de e-mails deve classificar todo e-mail recebido como SPAM ou não, de acordo com alguns dados, como remetente, destinatários, assunto, conjunto de palavras, etc.

Finalmente, técnicas de agrupamento são utilizadas para identificar grupos de elementos semelhantes na base de dados. A meta é que elementos similares pertençam a um mesmo grupo e elementos não semelhantes pertençam a grupos diferentes. Com este objetivo os algoritmos de agrupamento maximizam a similaridade entre elementos num mesmo grupo e a minimizam para elementos de grupos distintos. Um exemplo da utilização de agrupamento pode ser na identificação de grupos de clientes de acordo com seu perfil de consumo. De uma base de clientes de uma revendedora de carros, são criados grupos de clientes de acordo com a sua idade, sexo, modelo e marca do último carro comprado, etc. Esses grupos, depois de identificados, podem ser utilizados com o objetivo de se fazer uma campanha de marketing mais eficiente, com público alvo específico.

Neste trabalho, é utilizada a técnica de mineração de regras de associação. Dessa forma, essa técnica e as medidas de interesse suporte, confiança e *lift* serão melhor detalhadas. De maneira formal, regras de associação são definidas da seguinte forma. Sejam $I = \{i_1, i_2, .., i_m\}$ um conjunto de m itens distintos e D uma base de dados formada por um conjunto de transações, onde cada transação T é composta por um conjunto de itens (*item set*), tal que $T \subseteq I$. Uma regra de associação é uma expressão na forma $A \Rightarrow B$, onde $A \subset I$, $B \subset I$, $A \neq \emptyset$, $B \neq \emptyset$ e $A \cap B = \emptyset$. A é denominado precedente e B denominado consequente da regra. Tanto o precendete quanto o consequente de uma regra de associação podem ser formados por conjuntos contendo um ou mais itens. A quantidade de itens pertencentes a um conjunto de itens é chamada de comprimento do conjunto. Um conjunto de itens de comprimento k costuma ser referenciado como um *k-item set*.

O suporte de um conjunto de itens Z, Sup(Z), representa a porcentagem de transações da base de dados que contêm todos os itens de Z. O suporte de uma regra de associação $A \Rightarrow B$, $Sup(A \Rightarrow B)$, é dado por $Sup(A \cup B)$. Já a confiança desta regra, $Conf(A \Rightarrow B)$, representa, dentre as transações que contêm A, a porcentagem de transações que também contêm B, ou seja, $Conf(A \Rightarrow B) = Sup(A \cup B) \div Sup(A)$.

O *lift* permite avaliar a dependência entre o precedente e o consequente da regra. De maneira geral, expressa o quão mais frequente o consequente se torna quando o precedente ocorre. Para uma regra de associação $A \Rightarrow B$, o *lift* indica o quanto mais frequente torna-se B quando A ocorre. De maneira formal: seja D uma base de dados de transações. Seja $A \Rightarrow B$ uma regra de associação obtida a partir de D. O lift é definido como: $Lift(A \Rightarrow B) = Conf(A \Rightarrow B) \div Sup(B)$.

2.3 GERÊNCIA DE CONFIGURAÇÃO

Gerência de Configuração (GC) é uma disciplina que visa controlar a evolução do software (DART, 1991). Com este objetivo, GC aplica procedimentos técnicos e administrativos, normalmente auxiliados por softwares e processos específicos.

Um artefato sob Gerência de Configuração é chamado Item de Configuração (IC). Um projeto de software pode ser um IC, contendo o código fonte do software, documentação do sistema, manuais de usuário, ou documentos necessários à gerência do projeto. O que será definido como IC, os artefatos que fazem parte dele, e como a Gerência de Configuração auxilia a controlar a evolução do software dependem da necessidade de controle e definições de cada projeto.

Sob uma perspectiva de desenvolvimento, a GC possui três principais sistemas (ASKLUND; BENDIX, 2002): Sistema de Controle de Mudanças (SCM), Sistema de Controle de Versão (SCV) e Sistema de Gerenciamento de Construção (SGC). O Sistema de Controle de Mudanças armazena toda a informação de pedidos de mudança, enviando-os às partes interessadas e pessoas autorizadas, e mantém o status da configuração. Sistemas de Controle de Versão permitem uma evolução disciplinada à medida que podem ser realizadas modificações concorrentes e distribuídas do software. O SCV é necessário para manter os ICs íntegros durante a evolução, sem corromper o Sistema de Gerência de Configuração. O Sistema de Gerenciamento de Construção dá suporte à função de gerência de liberação e entrega do software, automatizando o processo de construção dos diferentes artefatos que tornam o software um programa executável.

Atualmente, a adoção de GC nas empresas tem crescido. Em 2005, cerca de 30% das empresas nacionais utilizavam CG (MCT, 2005).

O ciclo de trabalho de um desenvolvedor que utiliza GC começa com o SCV e segue utilizando o SGC, enquanto as mudanças requisitadas e o status do software são controlados pelo SCM. Primeiramente, o desenvolvedor copia o IC do repositório de código no SCV para uma área de trabalho de desenvolvimento (i.e., operação de *check out*). Em seguida, são realizadas modificações no software e, com a ajuda do SGC, o software executável é construído. Depois de realizadas as modificações, o IC retorna para o repositório (i.e., operação de *check in*). O desenvolvimento é finalizado ao indicar no Sistema de Controle de Mudanças que a nova funcionalidade ou correção já foi implementada como havia sido requisitada.

O Sistema de Gerenciamento de Construção pode ser utilizado para construir o programa executável com os artefatos que compõem o software e também para fazer o *release* do software, colocando-o nos diversos ambientes em que ele será utilizado. Para tornar a construção do software automatizada, o SGC utiliza configurações que indicam quais dependências o projeto possui, como seus artefatos devem ser combinados para construir a versão executável do software, como e onde deve ser feito o *release*, etc. Com o auxílio de um SGC é possível automatizar o processo de construção do software tornando-o independente dos desenvolvedores e menos propício a erros.

É importante notar que o SGC viabiliza a construção automatizada do software. Consequentemente, sem essa ferramenta, implementar um procedimento automatizado para construir diversas versões do software geradas durante seu desenvolvimento seria extremamente trabalhoso e complexo.

O SGC tem a responsabilidade de construção e o SCV mantém a configuração do software para cada versão, tornando possível a operação do SGC. À medida que o software evolui, o SCV mantém versionado, junto com o software, os descritores do processo de construção, ou seja, o SGC. O SCV mantém cada versão do software, as modificações realizadas em cada uma delas, quem fez e quando fez cada alteração.

2.4 MÉTRICAS DE SOFTWARE

As engenharias precisam de uma maneira de descrever objetivamente um objeto e, para tanto, utilizam-se métricas (PRESSMAN, 2001). Uma métrica provê indicação quantitativa da extensão, dimensão, capacidade, tamanho ou o quanto um objeto possui de um determinado atributo (PRESSMAN, 2001). Outra definição diz que a métrica define o método e a escala de medição para avaliar de maneira quantitativa o grau que um sistema, componente ou processo possui um atributo (IEEE, 1990). Desta forma, medição é o processo de determinar uma métrica, ou seja, extrair uma medida (IEEE, 1990).

Um atributo é uma propriedade física ou abstrata de um objeto que pode ser medida (IEEE, 1990). Atributos de software também podem ser referenciados como habilidades do software, atributos de qualidade, ou fatores de qualidade. Nas seções seguintes, são descritos alguns conjuntos de métricas e seus respectivos conjuntos de atributos de qualidade.

Através das métricas são construídos indicadores. Os indicadores podem ser construídos com uma ou mais métricas, e seu objetivo é fornecer conhecimento do objeto de estudo, seja ele o processo, o software ou qualquer produto analisado (PRESSMAN, 2001), através da predição ou estimativa de outras medições (IEEE, 1990). Indicadores podem ser utilizados como alarmes para sinalizar comportamentos indevidos que merecem atenção quando seus valores ultrapassam limites previamente estabilizados (HENDERSON-SELLERS, 1995).

De acordo com a ISO, existem métricas de processo e de produto (ISO, 2001). As métricas de produto podem ser classificadas segundo a forma pela qual são extraídas e de acordo com a fase de desenvolvimento, como indicado na Figura 2.

De acordo com Pressman (PRESSMAN, 2001), as métricas podem ser classificadas como de processo, de projeto ou de produto, de acordo com seu propósito, ou seja, de acordo com os indicadores que elas devem dar suporte. Métricas de produto são referentes a um indivíduo, métricas de projeto a uma equipe de projeto e métricas de processo a uma organização. As métricas de produto são, constantemente, combinadas para desenvolver

métricas de projeto, que por sua vez, são consolidadas para criar métricas de processo (PRESSMAN, 2001).

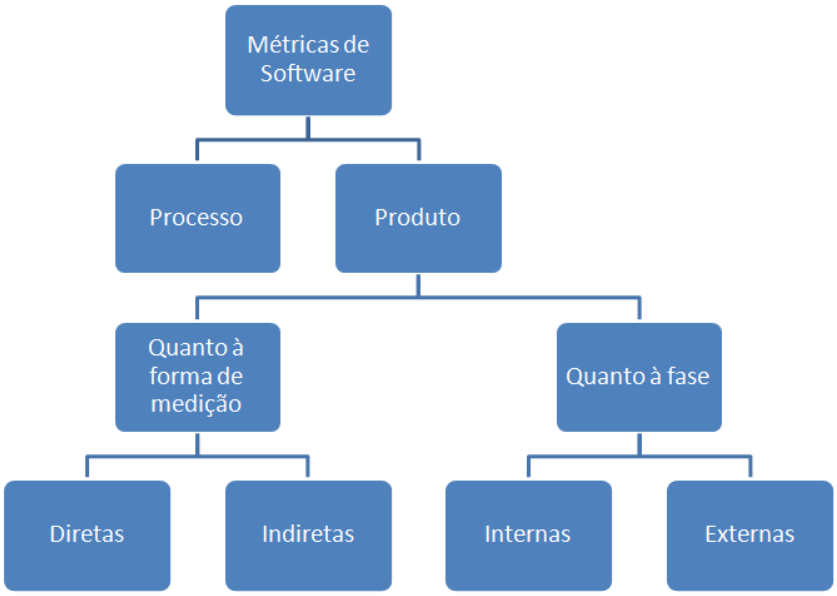


Figura 2: Classificação das métricas de software de produto (ISO, 2001).

Métricas de processo são aplicadas com um propósito estratégico e métricas de projeto, com um propósito tático (PRESSMAN, 2001). A estratégia indica o que deve ser feito para alcançar um objetivo com os recursos disponíveis focando em objetivos de longo prazo, enquanto a tática indica como deve ser feito, com objetivos de curto prazo (PRESSMAN, 2001). Dessa forma, as métricas de processo avaliam se o que está sendo feito segue o planejado, ou seja, se o processo está sendo executado como deveria e qual a qualidade que está sendo mantida em sua execução. Por outro lado, as métricas de produto avaliam o produto, verificando se ele foi feito conforme deveria, ou seja, com a qualidade desejada. Então, gerentes de projeto usam métricas para adaptar o processo de desenvolvimento de software e as atividades técnicas de acordo com os requisitos do projeto.

As métricas de processo são extraídas para dar suporte aos indicadores de processo. O seu objetivo é descrever o processo de desenvolvimento de software. Estas métricas são utilizadas para aumentar o entendimento e o controle sob o processo com a finalidade de aumentar a sua qualidade, visando assim, aumentar indiretamente a qualidade do produto produzido pelo processo.

Métricas de produto ganham significado à medida que o trabalho técnico avança. Enquanto o software evolui, métricas são extraídas para avaliar a qualidade e a arquitetura (do inglês, *design*), produzindo indicadores que serão utilizados durante as fases de codificação e

teste. Métricas de produto possuem duas aplicações (PRESSMAN, 2001). A primeira é minimizar o tempo de desenvolvimento, pois possibilitam que alguém atue na diminuição de riscos e a mitigação de problemas. A outra aplicação consiste em quantificar objetivamente a qualidade do produto sendo construído, permitindo que gerentes de projeto reajam com mudanças técnicas que melhoram a qualidade do produto final.

Métricas de produto são classificadas como diretas ou indiretas de acordo com a maneira como são coletadas e com o seu objetivo de medição. Métricas diretas verificam atributos que não dependem de outros atributos. Alguns exemplos são: linhas de código, número de interfaces ou número de métodos. Por outro lado, métricas indiretas avaliam atributos do software que são derivados de outros atributos do software (ISO, 2001). Exemplos de métricas indiretas são: entendimento, portabilidade e complexidade.

As métricas de projeto podem ainda ser classificadas como internas ou externas, de acordo com a fase do desenvolvimento na qual são extraídas (ISO, 2001), conforme ilustrado na Figura 2. Métricas internas são aplicadas ao produto não executável, durante as fases de projeto arquitetural e codificação. O objetivo primário dessas métricas é garantir que a qualidade seja alcançada já nas fases iniciais do projeto, antes de o produto do software estar pronto. Já as métricas externas medem o produto do software através do seu comportamento. O objetivo das métricas externas é informar aos usuários, avaliadores e desenvolvedores que a qualidade que eles buscam no software é alcançada durante a execução.

O propósito das métricas de software é descrever objetivamente a qualidade do software, porém, empresas, gerentes e clientes têm visões distintas de qualidade, impossibilitando assim que um único conjunto de métricas possa descrever qualquer software (NAGAPPAN; BALL; ZELLER, ANDREAS, 2006; PRESSMAN, 2001). Consequentemente, não existe um conjunto de métricas que descreva qualidade para qualquer projeto, resultando na criação de vários conjuntos distintos de métricas para mapear os atributos de qualidade. Nas seções seguintes, são descritos alguns desses conjuntos de métricas da literatura.

2.4.1 CONJUNTO DE MÉTRICAS DE MCCALL

McCall e Cavano criaram um conjunto de métricas baseado em fatores de qualidade (CAVANO; MCCALL, JAMES A, 1978; PRESSMAN, 2001). Os atributos de qualidade, ou fatores de qualidade, como são chamados pelos autores, se baseiam em três aspectos do produto: características operacionais, capacidade de sofrer modificações e capacidade de

adaptação a novos ambientes. Destes aspectos são derivados 11 fatores de qualidade, descritos nas Tabelas 1, 2 e 3.

Tabela 1: Fatores de qualidade de McCall relacionados a características operacionais.

Fator de qualidade	Descrição
Corretude	Grau de adequação do programa a especificação e aos objetivos do cliente.
Confiabilidade	Grau do quanto se pode esperar que o programa execute as devidas funções com a precisão requerida.
Eficiência	Grau do quanto de recursos computacionais e de código o programa precisa para executar suas funções.
Integridade	Grau do quanto é possível controlar os dados e o sistema de pessoas não autorizadas.
Usabilidade	Grau do esforço necessário para aprender, operar, preparar a entrada e entender a saída de dados do programa.

Tabela 2: Fatores de qualidade de McCall relacionados à capacidade de sofrer modificação.

Fator de qualidade	Descrição
Manutenibilidade	Grau do esforço necessário para localizar e corrigir erros no software.
Flexibilidade	Grau do esforço necessário para modificar um programa operacional.
Testabilidade	Grau do esforço necessário para garantir que um programa executa suas funções da maneira correta.

Tabela 3: Fatores de qualidade de McCall relacionados à capacidade de adaptação a novos ambientes.

Fator de qualidade	Descrição
Portabilidade	Grau do esforço necessário para transferir o programa de um hardware ou sistema operacional para outro.
Reusabilidade	Grau do quanto um programa pode ser utilizado em outras aplicações.
Interoperabilidade	Grau do esforço necessário para combinar um sistema com outro.

Com base nos fatores de qualidade, é descrito um conjunto de métricas, as quais possuem valores que variam numa escala que vai de baixo (0) a alto (10). Essas métricas mapeiam os fatores e aspectos de qualidade com uma expressão polinomial de diferentes coeficientes para cada métrica e fator, gerando assim uma expressão para cada fator de qualidade contendo todas as métricas. Na Tabela 4, são descritas as métricas propostas por McCall.

As expressões que mapeiam a relação entre as métricas e os fatores de qualidade seguem o modelo da equação $F_q = c_1 \times m_1 + c_2 \times m_2 + \cdots + c_n \times m_n$ (CAVANO; MCCALL, JAMES A, 1978), onde F_q representa algum fator de qualidade, c_i, $1 \leq i \leq n$, são

os coeficientes de regressão e m_i, $1 \leq i \leq n$, são as métricas descritas na Tabela 4. Com essas expressões é possível descobrir os valores dos fatores a partir dos valores das métricas.

Tabela 4: Métricas de McCall

Métrica	Descrição
Auditabilidade	Grau da facilidade de verificação da conformidade com os padrões.
Precisão	Grau da precisão computacional e controle.
Uniformização da comunicação	Grau da utilização dos padrões de interface, protocolo e banda.
Completude	Grau do quanto foi implementado no software e aceito pelo cliente das funções requeridas por ele.
Concisão	Grau do quão compacto está o programa em termos de linhas de código.
Uniformização dos dados	Grau da utilização de padrões de estruturas e tipos de dados pelo programa.
Tolerância a erro	Grau do dano que o sistema sofre quando ocorre um erro.
Eficiência da execução	Grau do desempenho do programa rodando.
Expansibilidade	Grau do quanto o projeto, dados ou procedimentos podem ser estendidos.
Generalidade	Grau da amplitude de potencial de aplicação de componentes do programa.
Independência do hardware	Grau de desacoplamento do software em relação ao hardware em que ele opera.
Instrumentação	Grau de monitoramento do programa das suas próprias operações e identificação da ocorrência dos seus erros.
Modularidade	Grau da independência dos componentes do programa.
Operabilidade	Grau da facilidade de operação do programa.
Segurança	Grau da disponibilidade de mecanismos que controlam e protegem os dados e o programa.
Documentação própria	Grau do quanto o código fonte provê documentação com significado.
Simplicidade	Grau do quanto o programa pode ser entendido sem dificuldade.
Independência do software	Grau do quanto o programa é independente de características da linguagem de programação fora do padrão, sistema operacional e outras limitações de ambiente.
Rastreabilidade	Grau do quanto é possível identificar quais componentes do programa estão relacionados a quais os requisitos.
Treinamento	Grau do quanto o software ajuda novos usuários na utilização do sistema.

Contudo, não existe uma formula geral para qualquer projeto, resultando em dificuldade de aplicação deste modelo para projetos com características diferentes. Isto se dá porque os pesos dos relacionamentos, ou seja, os coeficientes das expressões polinomiais dependem de cada projeto, seus objetivos e das preocupações dos clientes.

2.4.2 CONJUNTO DE MÉTRICAS DA ISO 9126

A Organização Internacional de Padronização, do inglês *International Organization for Standardization* (ISO), e a Comissão Internacional Eletrotécnica, do inglês *International Electrotechinical Commission* (IEC), criaram juntas a norma ISO 9126 para padronizar globalmente a descrição de atributos de qualidade (ISO, 2001). A ISO 9126 descreve um modelo de qualidade criado para identificar os principais atributos de qualidade dos softwares, no qual são identificadas seis características (atributos de qualidade) que são subdivididas em

subcaracterísticas. Além destes atributos, a ISO 9126 define conceitos de qualidade interna, qualidade externa e qualidade em uso, bem como um processo para avaliação da qualidade.

As seis características de qualidade identificadas são: funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade. A descrição de cada uma destas características está apresentada na Tabela 5.

Tabela 5: Características de qualidade da ISO 9126

Característica de qualidade	Descrição
Funcionalidade	Capacidade do software de prover as funções requeridas, alcançando as necessidades e estados requeridos, quando utilizado nas condições especificadas.
Confiabilidade	Capacidade do software de manter o desempenho, quando utilizado dentro das condições especificadas.
Usabilidade	Capacidade do software ser entendido, aprendido, utilizado e atrativo ao usuário, quando utilizado dentro das condições especificadas.
Eficiência	Capacidade do software de prover o desempenho apropriado de acordo com os recursos utilizados, quando utilizado dentro das condições especificadas.
Manutenibilidade	Capacidade do software de ser modificado. Modificações incluem: correções, melhorias ou adaptações de ambiente, requisitos e especificações funcionais.
Portabilidade	Capacidade do software de ser transferido de um ambiente para outro.

Das características de qualidade apresentadas na Tabela 5, são derivadas as subcaracterísticas apresentadas nas Tabelas 6 e 7. Estas subcaracterísticas podem ser medidas tanto por métricas internas quanto externas.

Porém, assim como os conjuntos de atributos descritos anteriormente, este conjunto não descreve métricas para medição direta. Contudo, provê um conjunto de atributos que pode descrever qualquer software e segue como guia para medições indiretas ou para criar conjuntos de medições diretas.

Tabela 6: Subcaracterísticas de qualidade da ISO 9126 da característica portabilidade.

Característica	Subcaracterística	Descrição
Portabilidade	Adaptabilidade	Capacidade do software de se adaptar a ambientes diferentes do especificado sem necessitar de ações ou meios não considerados para este propósito.
	Instalabilidade	Capacidade do software de ser instalado no ambiente especificado.
	Coexistência	Capacidade do software de coexistir em um ambiente com softwares independentes compartilhando recursos comuns.
	Capacidade de substituição	Capacidade do software de ser utilizado no lugar de outro software especificado para o mesmo objetivo e num mesmo ambiente.
	Cumprimento da portabilidade	Capacidade do software de aderir a padrões ou convenções relacionadas à portabilidade.

Tabela 7: Subcaracterísticas de qualidade da ISO 9126 das características manutenibilidade, eficiência, usabilidade, confiabilidade e funcionalidade.

Característica	Subcaracterística	Descrição
Manutenibilidade	Analisabilidade	Capacidade do software de ter diagnosticadas falhas ou deficiências no código, ou de ter identificada a parte que deve ser modificada.
	Mutabilidade	Capacidade do software de permitir que uma modificação seja efetuada.
	Estabilidade	Capacidade do software de evitar efeitos colaterais inesperados provenientes de modificações.
	Testabilidade	Capacidade do software de validar as modificações realizadas.
	Cumprimento da manutenibilidade	Capacidade do software de aderir a padrões ou convenções relacionadas à manutenção.
Eficiência	Comportamento temporal	Capacidade do software de prover tempos de resposta e processamento apropriados ao executar suas funções, quando utilizado nas condições especificadas.
	Utilização de recursos	Capacidade do software de utilizar os recursos apropriados, quando utilizado nas condições especificadas.
	Cumprimento da eficiência	Capacidade do software de aderir a padrões ou convenções relacionadas à eficiência.
Usabilidade	Compreensibilidade	Capacidade do software de permitir ao usuário que entenda quando o software deve ser utilizado, para quais tarefas e sob quais condições de uso.
	Capacidade de aprendizado	Capacidade do software de permitir que o usuário aprenda a utilizá-lo.
	Operabilidade	Capacidade do software de permitir que o usuário o utilize e o opere.
	Atratividade	Capacidade do software de ser atrativo ao usuário.
	Cumprimento de usabilidade	Capacidade do software de aderir a padrões, convenções, guias de estilo ou regulamentos relacionados à usabilidade.
Confiabilidade	Maturidade	Capacidade do software de evitar erros provenientes de falhas no próprio software.
	Tolerância a falhas	Capacidade do software de manter níveis especificados de desempenho em casos de falhas ou de violação da interface.
	Recuperabilidade	Capacidade do software de reestabilizar níveis especificados de desempenho e recuperar dados diretamente afetados em casos de falha.
	Cumprimento de confiabilidade	Capacidade do software de aderir a padrões, convenções ou regulamentos relativos à confiabilidade.
Funcionalidade	Adequação	Capacidade do software de prover um conjunto apropriado de funções para as tarefas especificadas de acordo com os objetivos do usuário.
	Precisão	Capacidade do software de prover os resultados corretos ou efeitos com o grau de precisão definido.
	Interoperabilidade	Capacidade do software de interagir com outros sistemas especificados.
	Cumprimento da funcionalidade	Capacidade do software de aderir aos padrões, convenções ou regulamentos de leis prescritos para a funcionalidade.

2.4.3 CONJUNTO DE MÉTRICAS DE QMOOD

QMOOD é o acrônimo de Modelo de Qualidade para Projeto Orientado a Objetos, do inglês *Quality Model for Object-Oriented Design* (BANSIYA; DAVIS, 2002). QMOOD define um conjunto de métricas que podem ser mapeadas indiretamente, porém de forma definida, para atributos de qualidade. Este conjunto de métricas e atributos de qualidade é voltado a projetos implementados com Orientação a Objetos (OO).

Para se identificar os atributos de qualidade, é utilizada uma abordagem *bottom-up*, apresentada na Figura 3. Essa abordagem possui quatro camadas: componentes, métricas, propriedade e atributos de qualidade de projeto Orientado a Objetos.

Figura 3: Níveis e ligações de QMOOD (BANSIYA; DAVIS, 2002).

A camada com os componentes de projeto (1) é formada por objetos de uma arquitetura OO: atributos, métodos, classes, relacionamentos e hierarquias de pacotes. Os componentes podem ser avaliados com as métricas de projeto (2), cujas definições são apresentadas na Tabela 8. A partir das métricas de projeto, são derivadas as propriedades de projeto (3), que são conceitos de qualidade que podem ser avaliados analisando-se a estrutura interna e externa do mesmo. Cada propriedade de projeto é avaliada através de uma métrica. Na Tabela 9, são mostradas as definições das propriedades de projeto e as métricas que as avaliam. Os atributos de qualidade mostrados na Tabela 10 qualificam o projeto com a finalidade de se avaliar a qualidade do produto derivado dela. Porém, esses atributos, como a própria qualidade, são conceitos abstratos que não podem ser avaliados diretamente. Dessa

forma, as propriedades de projeto são mapeadas para atributos de qualidade (4) conforme a influência que cada elemento do primeiro grupo exerce sobre os elementos do segundo.

Tabela 8: Métricas de projeto de QMOOD.

Acrônimo	Métrica	Descrição
DSC	Tamanho do Projeto em Classes	Número total de classes no projeto.
NOH	Número de Hierarquias	Número de hierarquias do projeto.
ANA	Número Médio de Ancestrais	Número médio de classes das quais uma classe herda informação. Esta métrica é determinada pelo número de classes pais, até a raiz, que uma classe possui.
DAM	Métrica de Acesso a Dados	Razão entre o número total de atributos privados (ou protegidos) pelo número total de atributos que uma classe possui.
DCC	Acoplamento Direto de Classes	Quantidade de classes diferentes com as quais uma classe se relaciona. Esta métrica considera atributos declarados ou parâmetros de métodos.
CAM	Coesão entre Métodos de Classes	Grau de parentesco entre os métodos de uma classe. Esta métrica é contabilizada a partir da interseção de parâmetros dos métodos de acordo com o total de parâmetros declarados pelos métodos da classe.
MOA	Medida de Agregação	Mede o quanto do relacionamento "parte do todo" uma classe possui, através de seus atributos. Ela é uma contagem da quantidade de atributos declarados na classe cujo tipo é declarado pelo usuário.
MFA	Medida de Abstração Funcional	Razão do número de métodos herdados pelo total de métodos acessíveis pela classe.
NOP	Número de Métodos Polimórficos	Número de métodos que possuem comportamento polimórfico.
CIS	Tamanho da Interface da Classe	Número de métodos públicos de uma classe.
NOM	Número de Métodos	Número de métodos definidos numa classe.

O modelo de qualidade de QMOOD é diferente dos demais apresentados, pois mapeia as métricas com exatidão para atributos de qualidade, de uma maneira bem definida. Consequentemente, este modelo permite algo que os outros não permitem: a obtenção de valores para atributos de qualidade a partir do código fonte de qualquer projeto. O mapeamento das propriedades de projeto para atributos de qualidade é feito através de fórmulas polinomiais, as quais indicam como cada propriedade de projeto influencia os atributos de qualidade. Estas fórmulas estão listadas na Tabela 11.

Tabela 9: Propriedades de projeto de QMOOD.

Propriedade do projeto	Definição	Métrica relacionada
Tamanho do projeto	Medida da quantidade de classes utilizadas no projeto.	DSC
Hierarquias	É uma contagem da quantidade de classes não herdadas que possuem especializações (classes filhas) no projeto.	NOH
Abstração	Medida da generalização-especialização do projeto. Conta a quantidade de classes no projeto que possuem uma ou mais classes descendentes.	ANA
Encapsulamento	Definido como o encapsulamento de dados e comportamento numa única estrutura. Em projetos OO, esta propriedade se refere às classes que previnem o acesso a métodos ou a dados, os tornando privados.	DAM
Acoplamento	Define a interdependência entre objetos do projeto. É o número de outros objetos com as quais um objeto se relaciona para desempenhar suas funções.	DCC
Coesão	Verifica o quão relacionados estão os métodos e atributos de uma classe. Grande sobreposição de parâmetros entre os métodos indica uma forte coesão.	CAM
Composição	Mede as relações "parte-de", "possui", "consiste de", ou "parde-do-todo", que representam agregações em um projeto orientado a objetos.	MOA
Herança	Mede a relação "é um" nos relacionamentos entre os objetos. Esta métrica indica a quantidade de classes aninhadas e mede a herança no projeto.	MFA
Polimorfismo	É a habilidade de substituir objetos por outros com a mesma interface durante a execução do código. É uma medida da quantidade de serviços de um objeto que podem ser definidos dinamicamente em tempo de execução.	NOP
Comunicação	É uma contagem da quantidade de métodos públicos que podem ser utilizados como serviços por outras classes. É uma medida da quantidade de serviços que uma classe provê.	CIS
Complexidade	É uma medida do grau de dificuldade de se entender a estrutura interna e externa de uma classe e seus relacionamentos.	NOM

Tabela 10: Atributos de qualidade de QMOOD.

Atributo de qualidade	Definição
Reusabilidade	Representa a presença de propriedades de projeto orientado a objetos que permite que o projeto seja reaproveitada em um novo problema sem grande esforço.
Flexibilidade	Característica que permite a incorporação de mudanças no projeto. É a habilidade do projeto de ser adaptado de maneira a prover capacidades relacionadas à funcionalidade.
Entendimento	Propriedades de projeto que permitem que este seja facilmente entendido e compreendido. Está diretamente relacionado à complexidade da estrutura.
Funcionalidade	As responsabilidades associadas a uma classe que são visíveis a outras classes do projeto através de suas interfaces públicas.
Extensibilidade	Refere-se à presença e utilização de propriedades no projeto que permitem a incorporação de novas funcionalidades.
Efetividade	Capacidade do projeto de alcançar a funcionalidade e o comportamento desejados utilizando conceitos e técnicas de orientação a objetos.

Tabela 11: Equações dos atributos de qualidade de QMOOD.

Atributo de Qualidade	Equação
Reusabilidade	- 0,25 * Acoplamento + 0,25 * Coesão + 0,5 * Comunicação + 0,5 * Tamanho do projeto
Flexibilidade	0,25 * Encapsulamento - 0,25 * Acoplamento + 0,5 * Composição + 0,5 * Polimorfismo
Entendimento	- 0,33 * Abstração + 0,33 * Encapsulamento - 0,33 * Acoplamento + 0,33 * Coesão - 0,33 * Polimorfismo - 0,33 * Complexidade - 0,33 * Tamanho do projeto
Funcionalidade	0,12 * Coesão + 0,22 * Polimorfismo + 0,22 * Comunicação + 0,22 * Tamanho do projeto + 0,22 * Hierarquias
Extensibilidade	0,5 * Abstração -0,5 * Acoplamento + 0,5 * Herança + 0,5 * Polimorfismo
Efetividade	0,2 * Abstração + 0,2 * Encapsulamento + 0,2 * Composição + 0,2 * Herança + 0,2 * Polimorfismo

2.4.4 CONJUNTO DE MÉTRICAS DE WIEGERS

Wiegers (2003) apresenta um estudo sobre atributos de qualidade e seus relacionamentos. Neste estudo, são considerados 12 atributos de qualidade: disponibilidade, eficiência, flexibilidade, integridade, interoperabilidade, manutenibilidade, portabilidade, confiabilidade, reusabilidade, robustez, testabilidade e usabilidade. Alguns deles são descritos pelos modelos anteriormente apresentados. Contudo, uma descrição completa de todos é mostrada na Tabela 12.

Estes atributos são classificados em dois grupos quanto ao seu principal interessado: cliente e desenvolvedor (WIEGERS, 2003). Os atributos de qualidade de interesse do cliente são: disponibilidade, eficiência, flexibilidade, integridade, interoperabilidade, confiabilidade, robustez, e usabilidade. Já os de interesse do desenvolvedor são: manutenibilidade, portabilidade, reusabilidade, e testabilidade.

O grande problema indicado pelo estudo é que não é possível maximizar todos estes atributos ao mesmo tempo, já que alguns deles possuem comportamentos inversos, ou seja, ao aumentar um atributo o outro diminui. Consequentemente, desenvolvedores e clientes precisam identificar, para cada software, quais atributos de qualidade devem ser priorizados. Um exemplo consiste na impossibilidade de aumentar a portabilidade de um software sem diminuir a manutenibilidade, pois à medida que o código é adaptável a mais ambientes ele se torna mais complexo.

A relação entre os atributos de qualidade, apresentada por Wiegers (2003), é mostrada na Tabela 13. Nesta tabela, o símbolo "+" indica que quando o atributo da linha aumenta o da coluna também aumenta. O símbolo "-" indica que quando o atributo da linha aumenta o atributo da coluna diminui. Já uma célula em branco indica que o atributo da linha afeta pouco ou não afeta o atributo da coluna.

Tabela 12: Atributos de Qualidade utilizados por Wiegers (2003).

Atributo de qualidade	Principal interessado	Descrição
Disponibilidade	Cliente	Avalia qual a relação entre o tempo que a aplicação está disponível para uso e o tempo que foi planejado que ela estivesse disponível.
Eficiência	Cliente	Avalia a utilização dos recursos disponíveis pelo software, como disco, processador ou banda de rede.
Flexibilidade	Cliente	Avalia quão fácil é adicionar novas funcionalidades ao software.
Integridade	Cliente	Avalia quão protegido está o software, ou seja, do quanto ele protege os dados e as funcionalidades de acessos não permitidos.
Interoperabilidade	Cliente	Avalia quão facilmente o software consegue trocar dados ou serviços com outros softwares.
Manutenibilidade	Desenvolvedor	Mede a facilidade de corrigir defeitos ou modificar o software.
Portabilidade	Desenvolvedor	Avalia a facilidade de migrar uma parte do software de um ambiente operacional para outro.
Confiabilidade	Cliente	Avalia a probabilidade do software executar sem erros durante um período específico de tempo.
Reusabilidade	Desenvolvedor	Avalia o esforço necessário para converter um componente do software para ser utilizado em outra aplicação.
Robustez	Cliente	Avalia a capacidade do software de manter o comportamento esperado das funções após a ocorrência de um erro.
Testabilidade	Desenvolvedor	Avalia a facilidade de verificar se a aplicação ou componentes dela estão funcionando corretamente com o objetivo de identificar erros.
Usabilidade	Cliente	Avalia o quão fácil é a utilização do software.

Tabela 13: Relacionamentos entre os atributos de qualidade estudados por Wiegers (2003).

	Disponibilidade	**Eficiência**	**Flexibilidade**	**Integridade**	**Interoperabilidade**	**Manutenibilidade**	**Portabilidade**	**Confiabilidade**	**Reusabilidade**	**Robustez**	**Testabilidade**	**Usabilidade**
Disponibilidade								+		+		
Eficiência			-		-	-	-	-		-	-	-
Flexibilidade		-		-		+	+	+			+	
Integridade		-			-				-		-	-
Interoperabilidade		-	+	-			+					
Manutenibilidade	+	-	+					+			+	
Portabilidade		-	+		+	-			+		+	-
Confiabilidade	+	-	+			+				+	+	+
Reusabilidade		-	+	-	+	+	+	-			+	
Robustez	+	-						+				+
Testabilidade	+	-	+			+		+				+
Usabilidade		-								+	-	

2.5 ABORDAGENS DE MINERAÇÃO DE REPOSITÓRIOS DE SOFTWARE

Alguns trabalhos mostram que técnicas de mineração de dados e métricas de software podem ser utilizadas conjuntamente para ajudar a engenharia de software a melhorar a

qualidade do produto desenvolvido. As aplicações são diversas, de detecção automática de *bugs* (KIM *et al.*, 2006; NAGAPPAN; BALL; ZELLER, ANDREAS, 2006) a construção de modelos preditivos para identificar falhas (KHOSHGOFTAAR; SELIYA, 2004; MERTIK *et al.*, 2006; NAGAPPAN; BALL; MURPHY, 2006), e planejamento de *releases* (COLARES *et al.*, 2009).

Para pesquisar os trabalhos já realizados sobre mineração de repositórios de software, foram realizadas buscas em ferramentas que indexam artigos acadêmicos, como *Google Scholar, CiteSeerX, Portal ACM,* e diretamente nos anais do *Mining Software Repositories* (MSR), que é uma conferência focada nesta área de pesquisa. As buscas utilizaram as seguintes palavras-chave e suas combinações: mineração de dados, engenharia de software, métricas de software, gerência de configuração, evolução de software e histórico de software. Assim como suas traduções em inglês: *data mining*, *software engineering*, *software metrics*, *configuration management*, *software evolution*, e *software history*. Os artigos encontrados foram analisados de acordo com o tipo de utilização de gerência de configuração, mineração de dados e métricas de software, priorizando os que lidavam com sistemas de controle de versão, evolução, utilização de mineração de dados com o histórico do projeto e a relação entre métricas de software.

Os trabalhos mais relevantes estão listados nesta seção. Inicialmente, são apresentados os trabalhos que utilizam métricas de software para tentar identificar falhas, *bugs* ou defeitos. Depois, são apresentados trabalhos que utilizam mineração de dados e, para finalizar, os trabalhos que estudam a evolução do software.

Métricas de complexidade foram utilizadas com dados históricos dos projetos para identificar defeitos pós-*release* (NAGAPPAN; BALL; ZELLER, ANDREAS, 2006). Neste trabalho, é indicado que não existe um único conjunto de métricas que possa ser aplicado a qualquer projeto, ou seja, um conjunto de métricas que consegue identificar corretamente defeitos em um projeto não necessariamente é eficiente em outros projetos. A estratégia para identificar quais métricas devem ser utilizadas na identificação de defeitos é baseada no histórico de defeitos pós-*release* do projeto. Este trabalho também identifica que métricas OO podem ser associadas com defeitos.

Um algoritmo que pode identificar automaticamente mudanças que introduzem *bugs* foi apresentado por Kim *et al.* (2006). A abordagem proposta nesse trabalho utilizada como entrada o SCM e o SCV do software. Para identificar os *bugs*, foram propostos os seguintes passos: utilizar anotações de grafos para prover informações detalhadas; ignorar comentários e linhas em branco no código fonte, mudanças de forma (endentação), *commits* que corrijam

bugs nos quais são alterados muitos arquivos; e verificar manualmente as dicas de correções de *bugs*. A estratégia proposta se mostrou eficiente para identificar mudanças que introduzem *bugs*, pois teve altas taxas de acerto, melhores que as outras abordagens encontradas na literatura.

Outras abordagens objetivaram construir uma ferramenta para predizer se o software possui ou não defeitos. *O Emerald Software Analysis Tool*, que possui métricas de grafos, de processo e de projeto, foi utilizado para construir modelos de classificação para predizer se um módulo possui defeitos ou não (KHOSHGOFTAAR; SELIYA, 2004). Outras abordagens para identificar se um software possui defeitos também utilizaram métricas de produto e técnicas mineração de dados, como regressão e aprendizado supervisionado (MERTIK *et al.*, 2006; NAGAPPAN; BALL; MURPHY, 2006).

Um estudo mostrou que classificadores baseados em regras de associação construídos para softwares proprietários possuem uma precisão maior do os construídos com softwares de código livre (JÚNIOR *et al.*, 2009). Essa hipótese foi verificada através da avaliação de classificadores construídos para 18 projetos da indústria brasileira de bebidas em contraste com outro estudo realizado com projetos de código aberto (ZIMMERMANN, T. *et al.*, 2004). As transações eram compostas pelas mudanças realizadas em módulos num mesmo *commit* e informações do desenvolvedor que realizou a modificação. As regras mineradas explicitaram informações valiosas que auxiliaram na fase de manutenção do software, identificando quais módulos, com frequência, eram alterados num mesmo *commit*.

Com o objetivo de monitorar e controlar a qualidade do software, três bases de dados foram mineradas com técnicas de agrupamento (DICK *et al.*, 2004) para aumentar a qualidade do software. O agrupamento foi utilizado para identificar grupos de módulos de acordo com os valores de medições. A abordagem indica que, nos módulos dos grupos com maior valor das métricas, deve ser investido mais esforço, já que esses módulos tendem a possuir mais defeitos. As três bases de dados utilizadas neste estudo possuíam mais de mil transações com 11 atributos, que representam métricas de software, tais como linhas de código e complexidade ciclomática de McCabe (MCCABE, 1976).

Classificadores foram construídos a partir de informações extraídas de repositórios de software, utilizados por alunos para realizar o trabalho de uma disciplina, para tentar predizer suas notas, mas baixas taxas de acerto foram obtidas (MIERLE *et al.*, 2005). Nesse trabalho, os atributos utilizados eram métricas de software e outras informações sobre os repositórios utilizados por alunos para implementar trabalhos de faculdade. Com essas informações, foram construídos três classificadores para tentar predizer a nota final dos alunos, porém não

obtiveram resultados satisfatórios, pois o erro associado às classificações foi grande. Esse trabalho conclui que, com os atributos utilizados e da forma com que foi realizado, não foi possível inferir as notas dos alunos. Entretanto, descobriram que o aprendizado dos alunos com os trabalhos é proporcional ao tempo gasto neles e não em quando é realizado: se perto do fim do prazo de entrega ou mais cedo.

Versões do software eclipse foram estudadas através de seus *plugins* com a finalidade de se entender mais sobre a evolução de um software grande (WERMELINGER; YU, 2008). Foram analisadas 47 versões geradas durante seis anos de desenvolvimento: *releases* corretivas, candidatas e finais. As datas do SCV não correspondiam às datas dos *releases* que constavam no site do Eclipse. Dessa forma, não foi possível utilizar o SCV para conseguir o código fonte. Consequentemente, cada versão teve que ser pega, manualmente, através do código fonte disponibilizado no site. As versões foram analisadas verificando quais *plugins* estavam disponíveis na versão inicial, e quais foram adicionados ou removidos. Além disso, foram extraídas métricas de produto entre as versões. As análises foram feitas guiadas por cinco questões de interesse técnico e gerencial: (1) quantas mudanças arquiteturais ocorrem entre *releases* importantes e qual a taxa de crescimento do sistema; (2) se ocorrem mudanças arquiteturais em *releases* de manutenção; (3) se existem diferenças entre *releases* candidatas e *milestones*, em termos de evolução arquitetural; (4) se existem deleções ou apenas adições; e (5) se existe alguma arquitetura central que se mantém estável ao longo dos *releases*. Neste trabalho, foi descoberto que a evolução do eclipse segue um processo sistematizado e que existem partes do código que estão estáveis desde a primeira versão e prevalecem até a ultima versão analisada.

Para entender mais sobre o desenvolvimento de software livre, foram analisadas as versões estáveis do Debian Linux desde 1998 (sete anos de releases) (ROBLES *et al.*, 2006). Das revisões foram extraídas as métricas LOC e quantidade de pacotes, e dos pacotes foram extraídas: LOC, quantidade de arquivos e quantidade de versões corretivas. Foram feitas análises da evolução do Debian ao longo dos anos com as informações coletadas: tamanho do Debian, tamanho dos pacotes, manutenção dos pacotes, qual a linguagem de programação adotada e tamanho dos arquivos. Com este estudo, foi identificado que: (1) o tamanho das versões estáveis, em número de linhas de código e número de pacotes, dobra a cada dois anos; e (2) por causa da complexidade associada a cada pacote, o número de desenvolvedores é proporcional à quantidade de pacotes e, consequentemente, o número de desenvolvedores deve crescer à medida que o software cresce, o que indica um agravante para gerência do projeto com o tempo.

A evolução do software também foi mapeada em uma modelagem orientada a objetos para facilitar a utilização de consultas objetivando a descoberta de novas informações (FISCHER *et al.*, 2003). Neste mapeamento, foram utilizados os dados evolutivos do Mozzila, contidos em seu SCV e SCM. O objetivo deste trabalho foi facilitar a análise do histórico do projeto através do mapeamento em modelagem OO. Este objetivo foi alcançado, ao possibilitar que sejam feitas consultas para buscar informações na base de dados relacional criada com o histórico do projeto seguindo a modelagem OO. Até onde os estudos da literatura foram realizados, este é o único trabalho que mapeia a evolução do software através de uma modelagem OO.

2.6 CONSIDERAÇÕES FINAIS

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

Foram apresentados trabalhos que utilizam métricas de software para predizer se o software contém ou não defeitos, mineração de dados para descobrir informações sobre o software, através de regras de associação ou agrupamentos, e até classificadores para tentar predizer notas de alunos. Além disso, foi realizado um mapeamento da evolução do software por Fischer *et al.* (2003), porém este mapeamento acabou perdendo abstração ao ficar muito próximo das entidades do CVS e Bugzzila, e se afastando dos conceitos de GC. 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. Apesar de Robles *et al.* (2006) analisarem sete anos de releases, eles consideram apenas cinco versões do Debian Linux.

Entretanto, 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 como atributos de qualidade. Ainda mais, a evolução do software é analisada através de *releases*, não representando cada modificação realizada no software individualmente. Ou seja, nenhum trabalho analisa as modificações realizadas pelos desenvolvedores ao longo do tempo, representadas pelos *commits*, que são a menor medida de modificação para o SCV, nem analisa todas as versões do software geradas durante sua evolução.

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) que consiga avaliar as diversas versões do software automaticamente, utilizando gerência de configuração e métricas de software; (3) que as avaliações realizadas sejam objetivas e consigam ser mapeadas para elementos de mais alto nível, como atributos de qualidade; e (4) que aplique técnicas de mineração de dados a uma base com transações referentes às modificações realizadas em cada *commit*.

CAPÍTULO 3 – ABORDAGEM OSTRA

3.1 INTRODUÇÃO

No Capítulo 2, foi realizada uma revisão da literatura, mostrando que existem diversos trabalhos na área de mineração de repositórios de software (COLARES *et al.*, 2009; DICK *et al.*, 2004; FISCHER *et al.*, 2003; KHOSHGOFTAAR; SELIYA, 2004; KIM *et al.*, 2006; MERTIK *et al.*, 2006; MIERLE *et al.*, 2005; NAGAPPAN; BALL; MURPHY, 2006; NAGAPPAN; BALL; ZELLER, ANDREAS, 2006; ROBLES *et al.*, 2006; WERMELINGER; YU, 2008). Porém, até onde foi pesquisado, existem oportunidades ainda não exploradas. Dentre as oportunidades identificadas, podem-se destacar: (1) considerar o *commit* como elemento principal de análise, identificando as modificações realizadas por um desenvolvedor em algum momento da evolução do software; (2) realizar medições automáticas das diversas revisões do software, armazenadas em um Sistema de Controle de Versão, utilizando o Sistema de Gerência de Construção para compilar o software quando necessário para viabilizar medições; (3) quantificar as mudanças realizadas entre as revisões em cada *commit* através da variação de métricas de software; (4) aplicar mineração de regras de associação numa base de dados onde cada transação consiste de informações de um *commit*, que é representado pela variação das métricas de software e outras informações.

Nesse capítulo, é apresentada a abordagem Ostra, que contempla as oportunidades citadas anteriormente. Na Figura 4, são ilustrados os três passos principais da abordagem proposta: (1) medição das revisões do software, (2) mineração de dados, e (3) apresentação das informações descobertas através de regras, gráficos e tabelas.

O primeiro passo dessa abordagem é a medição de todas as revisões do projeto. Esse passo se inicia quando o código fonte de cada revisão é obtido, ao se realizar o *check out* de cada revisão armazenada no Sistema de Controle de Versão do projeto. Depois disso, cada revisão é construída e medições são realizadas com um conjunto de métricas previamente estabelecido.

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. No Capítulo 2, foram apresentados alguns trabalhos relacionados que utilizaram mineração de dados. Dentre eles, um utiliza regras de associação para identificar quais arquivos são alterados conjuntamente

(JÚNIOR *et al.*, 2009), mas nenhum minera regras a partir de métricas de software para descrever as alterações que ocorreram durante a evolução.

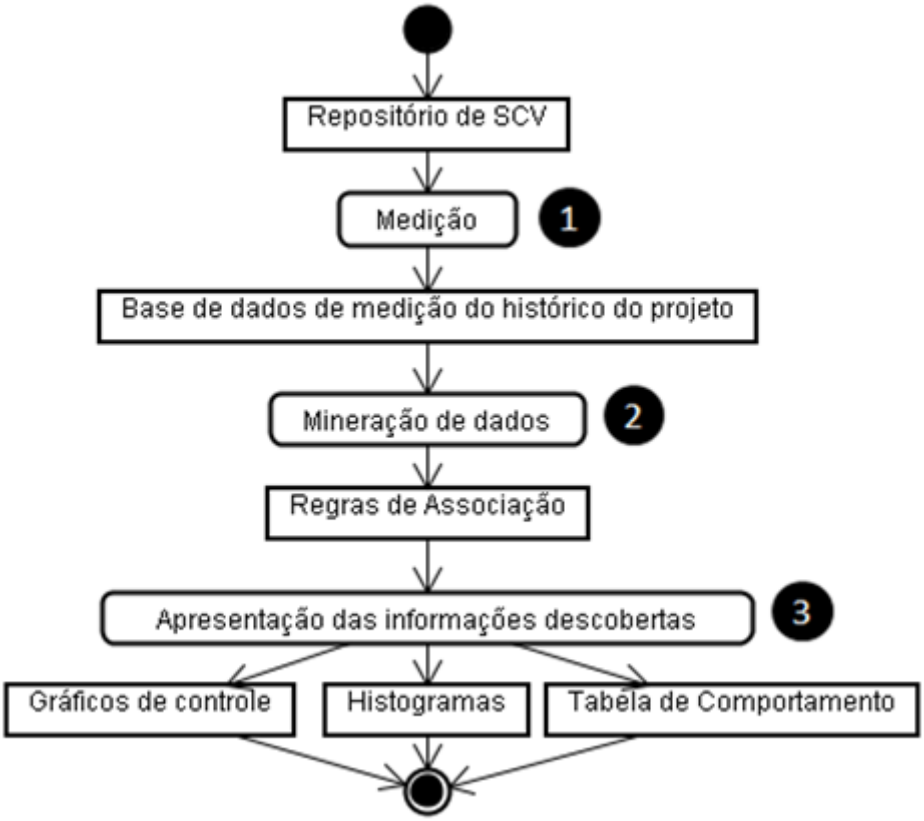


Figura 4: Os três passos principais da abordagem Ostra.

Para aplicar a Mineração de Dados com a finalidade de descobrir informações sobre a evolução do software, é construída uma base de dados que reflete as mudanças realizadas, as quais 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 padrões.

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 fornecidas algumas ferramentas: filtros baseados em atributos, gráficos históricos das métricas e uma tabela que mostra o comportamento das métricas, encontrado nas regras mineradas. Para fornecer informações temporais, são utilizados gráficos históricos, exibindo o comportamento de cada métrica ao longo do tempo e histogramas, que apresentam a dispersão dos valores das métricas.

Nesse capítulo, é apresentada a abordagem Ostra, explicando quais métricas são utilizadas, como são organizadas as informações da evolução do software no tempo, como é realizada a mineração de dados e quais são as formas de exibição do conhecimento descoberto. São apresentados os conceitos de métricas simples, compostas e delta. É apresentada a matriz de comportamento e os gráficos dos valores de métricas ao longo do tempo. A organização deste capítulo segue as três fases principais da abordagem: medição, mineração e apresentação. Na Seção 3.2, é apresentada a medição, na Seção 3.3, a mineração e, na Seção 3.4, a apresentação.

3.2 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 copiada 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.

Nesta seção, é apresentado como se realiza o passo de medição. Primeiramente, é apresentada a arquitetura orientada a objetos utilizada para representar a evolução do software. Em seguida, são apresentadas as métricas: simples, compostas e delta.

3.2.1 MODELO PARA REPRESENTAR A EVOLUÇÃO DO SOFTWARE

Para facilitar as próximas fases, os conceitos contidos na história do software são mapeados em um modelo OO. Como apresentado no Capítulo 2, sob o ponto de vista da Gerência de Configuração, o software pode ser considerado um item de configuração. Ao longo da evolução do software, ele sofre mudanças e cada mudança gera uma nova versão. Dessa forma, um item de configuração possui várias revisões durante sua evolução, que guardam as informações de cada versão e o estado dos artefatos que a formam.

Existem métricas que necessitam ser extraídas do software executável e outras do código fonte. Para compilar versões diferentes de um software, com dependências diferentes em constante mudança, deve-se delegar a construção do software para uma ferramenta. Essa ferramenta deve saber como juntar todos os artefatos do projeto, com suas respectivas dependências, e montar o software executável. Como apresentado anteriormente, no Capítulo 2, o Sistema de Gerenciamento de Construção é responsável por isso. Consequentemente, ele é fundamental para a abordagem Ostra, que faz medições das diversas versões do histórico do software automaticamente. Vale ressaltar que, para medir as diversas

versões de um software, é necessária a integração de dois Sistemas de Gerência de Configuração: Sistema de Controle de Versões e Sistema de Gerenciamento de Construção.

Na Figura 5, é exibido o diagrama de classes com a metamodelagem da evolução do software, ilustrando os conceitos de Gerência de Configuração utilizados nesse trabalho, os quais são mapeados em uma arquitetura OO. O item de configuração, classe *ConfigurationItem*, possui um nome que identifica o projeto e a URL base para acessá-lo no repositório. A linha de desenvolvimento, classe *DevelopmentLine*, possui a URL para fazer *check out* do projeto, informações sobre o SGC, a que item de configuração pertence, e as revisões que ela possui ao longo de sua evolução. Cada revisão, classe *Revision*, tem um número que a identifica e as informações do *commit*, como, por exemplo, o desenvolvedor que realizou as modificações, a data e hora da operação, e os artefatos que foram modificados. Cada artefato, classe *Artifact*, possui um caminho relativo que identifica o artefato representado por ele na estrutura de pastas do projeto. O artefato pode representar um arquivo ou uma pasta. Os artefatos e as revisões que os alteraram estão ligados pelo artefato versionado (AV), classe *VersionedArtifact*, que possui o tipo da mudança realizada: adicionado, alterado, removido ou movido[3]. O AV liga o artefato a cada revisão na qual ele foi modificado.

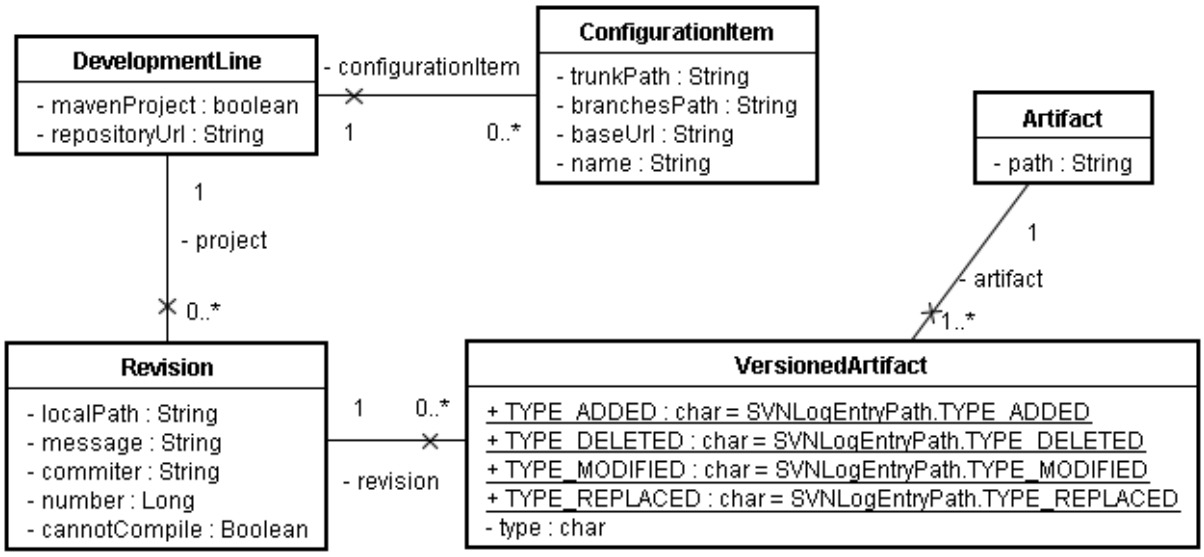


Figura 5: Modelo orientado a objetos para mapear a evolução do software através da história do item de configuração.

[3] Alguns SCV dão suporte a este tipo de operação. No Subversion, por exemplo, para mover um arquivo deve-se utilizar o comando "svn move". Porém, se a movimentação do arquivo for realizada sem utilizar o SCV, a abordagem não notará que ele foi movido, mas removido um arquivo e adicionado outro.

Para exemplificar, é mostrado um diagrama de objetos na Figura 6, que ilustra o *commit* identificado por #321 do projeto Oceano Core (explicado em mais detalhes no Capítulo 4). Nesse exemplo, o item de configuração é o Oceano Core e a linha de desenvolvimento sendo considerada pode ser encontrada na url *https://gems.ic.uff.br/svn/oceano/oceano-core/trunk*, como mostra a Figura 6. No *commit* #321, foi removido o arquivo persistence.xml, identificado pelo caminho */src/main/resources/META-INF/persistence.xml*. Esse commit foi realizado pelo desenvolvedor *daniel.*

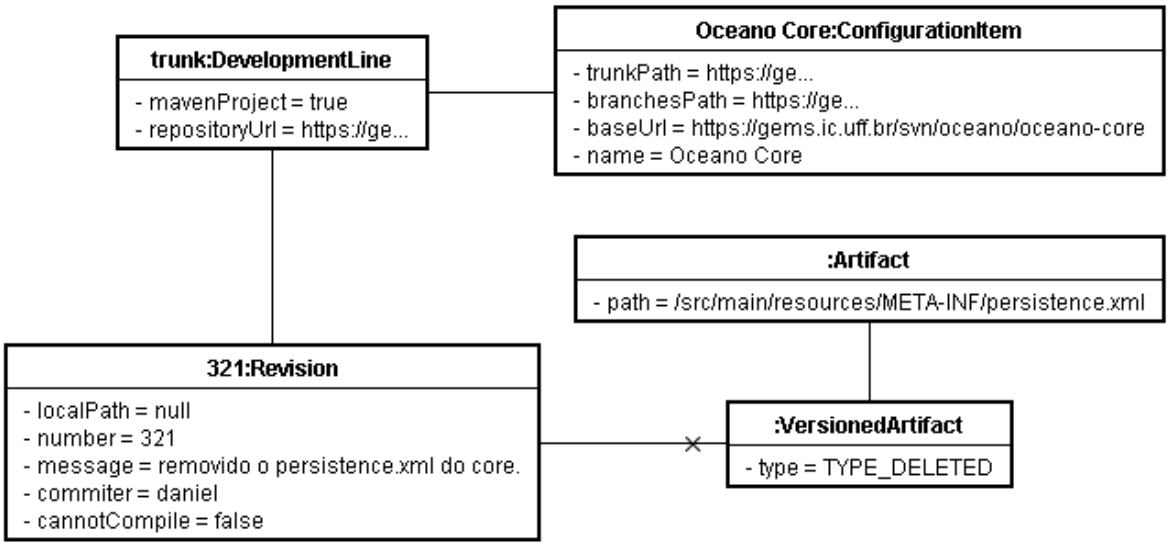


Figura 6: Diagrama de objetos mostrando um exemplo da modelagem com o *commit* #321 do projeto Oceano Core.

3.2.2 MÉTRICAS SIMPLES E COMPOSTAS

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 métricas simples ou compostas. O objetivo do suporte às métricas compostas é poder criar novas métricas facilmente com a combinação de outras métricas.

Um exemplo da utilização de métricas simples e compostas é para calcular a 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, definida pela seguinte expressão: $\frac{\text{LOC}}{\text{NOM}}$.

As métricas compostas permitem que sejam calculados os atributos de qualidade, apresentados no Capítulo 2, como qualquer outra métrica. Para extrair as métricas compostas,

é necessário já ter extraído as métricas simples, das quais ela depende. Consequentemente, o passo de extração das métricas é iniciado com a extração das métricas simples e, após todas elas serem extraídas, é iniciado o cálculo das métricas compostas.

3.2.3 CONJUNTO DE MÉTRICAS

O propósito das métricas de software é medir objetivamente a qualidade do software, como discutido anteriormente no Capítulo 2. As métricas que devem ser utilizadas dependem de quais características a empresa, os desenvolvedores, os gerentes, ou os clientes desejam avaliar no software. Além disso, os papéis envolvidos no desenvolvimento do software provavelmente desejam medir características diferentes (PRESSMAN, 2001). Consequentemente, um único conjunto de métricas não é capaz de avaliar todos os atributos de qualidade dos projetos (NAGAPPAN; BALL; ZELLER, ANDREAS, 2006). Contrastando com essa ideia, ao longo dos anos, alguns conjuntos de métricas e atributos de qualidade de uso geral surgiram (ISO, 2001; MCCALL, JIM A *et al.*, 1977; PRESSMAN, 2001).

A abordagem Ostra não é dependente de um conjunto de métricas específico. Contudo, nesse trabalho, é utilizado o conjunto de métricas de QMOOD, que pode ser diretamente mapeado para atributos de qualidade (BANSIYA; DAVIS, 2002). Apesar de ser utilizado esse conjunto de métricas e atributos de qualidade, a abordagem não está limitada a ele. Ao contrário, podem ser utilizadas quaisquer métricas que sejam interessantes para quem estiver analisando o projeto. Entretanto, foi escolhido o conjunto de atributos de qualidade de QMOOD, pois com esse conjunto as métricas podem ser mapeadas para atributos de qualidade com equações já definidas. Cada métrica de QMOOD é mapeada para uma propriedade de projeto (11 no total), que, por sua vez, é mapeada em atributos de qualidade de acordo com equações específicas, mostradas no Capítulo 2. Com esse conjunto de métricas, pode-se medir cada revisão e transformar os valores das métricas em atributos de qualidade, que são elementos de mais alto nível e podem ser compreendidos e relacionados com maior facilidade.

Junto ao conjunto de métricas de QMOOD, foram escolhidas ainda outras métricas que podem ser utilizadas como indicadores por gerentes de projeto ou pela equipe de desenvolvimento. As outras métricas são Complexidade Ciclomática de McCabe, Linhas de Código (LOC) e Número de Métodos e cada uma delas complementa as métricas de QMOOD com outras informações. A Complexidade Ciclomática de McCabe (1976), tem relação direta com a facilidade de testar o código desenvolvido e avalia a quantidade de caminhos possíveis um código possui, o que influencia diretamente na sua complexidade. A métrica LOC avalia o

tamanho do software em linhas de código executáveis e pode ser utilizado junto à Complexidade Ciclomática de McCabe para avaliar a relação entre complexidade e tamanho do software. A métrica Número de Métodos, 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 foram classificadas em três grupos, de acordo com o objeto alvo da medição: classe, pacote ou projeto. Essa classificação é baseada em como os valores das métricas são extraídos e armazenados. Cada arquivo ou pasta é a implementação física de classes e pacotes, compiladas ou não[4]. Como métricas de classe e pacote são extraídas numa forma mais granular que as de projeto, elas são armazenadas respeitando essa granularidade. Essa decisão foi tomada de forma que pudessem ser extraídas informações mais precisas, se necessário, em fases futuras, no processo de descoberta de conhecimento em bases de dados (KDD). As métricas são mostradas de acordo com sua classificação em relação ao alvo da medição: na Tabela 14 estão as métricas de projeto, na Tabela 15 está a métrica de pacote e na Tabela 16 estão as métricas de classe. Na Tabela 17 estão listados os atributos de qualidade de QMOOD e as equações utilizadas em seus mapeamentos como métricas compostas, explicadas anteriormente no Capítulo 2.

Tabela 14: Conjunto de métricas de Projeto da Ostra.

Métrica	Sigla	QMOOD
Número Médio de Ancestrais (Average Number of Ancestors)	ANA	✓
Tamanho do Projeto em Classes (Design Size in Class)	DSC	✓
Abstração Funcional (Measure of Functional Abstraction)	MFA	✓
Número de Métodos Polimórficos (Number of Polymorphic Methods)	NOP	✓
Número de Hierarquias (Number of Hierarchies)	NOH	✓
Medida de Agregação (Measure of Aggregation)	MOA	✓

Tabela 15: Conjunto de métricas de Pacote da Ostra.

Métrica	Sigla	QMOOD
Abstração (Abstractness)	RMA	✓

[4] Linguagens de programação orientadas a objetos, como Java ou C++, permitem que múltiplas classes sejam declaradas num único arquivo. Entretanto, após a compilação, cada classe torna-se um arquivo independente.

Tabela 16: Conjunto de métricas de Classe da Ostra.

Métrica	Sigla	QMOOD
Complexidade Ciclomática de McCabe (Average McCabe Cyclomatic Complexity)	ACC	
Coesão entre os Métodos da Classe (Cohesion Among Methods in Class)	CAM	✓
Número de Métodos (Number of Methods)	NOM	
Linhas de Código (Lines of Code)	LOC	
Métrica de Acesso a Dados (Data Access Metric)	DAM	✓
Acoplamento Direto de Classes (Direct Class Coupling)	DCC	✓
Tamanho da Interface da Classe (Class Interface Size)	CIS	✓

A descrição funcional das métricas utilizadas está no Apêndice A. Essa descrição é necessária para se definir exatamente como são contabilizadas as métricas.

Tabela 17: 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

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. Na Figura 7, a classe *MetricValue* armazena as medições de métricas de projeto, enquanto a classe *VersionedArtifactMetricValue* armazena as medições de pacote e classe. Dessa forma, as medidas podem ser acessadas posteriormente sem perda de informação, respeitando a granularidade utilizada na medição.

O atributo *isDelta* dessas classes indica se o valor armazenado é delta ou absoluto, como será apresentado na Seção 3.2.4. O atributo *expression*, da classe *Metric*, indica qual expressão define a métrica, se ela for composta.

Na Figura 8, é exibido um diagrama de objetos mostrando um exemplo da modelagem da Figura 7, com a medição da métrica LOC no *commit* #321. Nela, o valor para a métrica LOC para o artefato alterado foi zero e para a revisão também. Como é possível observar

nesse diagrama, a métrica LOC não é derivada e não possui expressão, e a classe que implementa sua medição é a *LinesOfCode*.

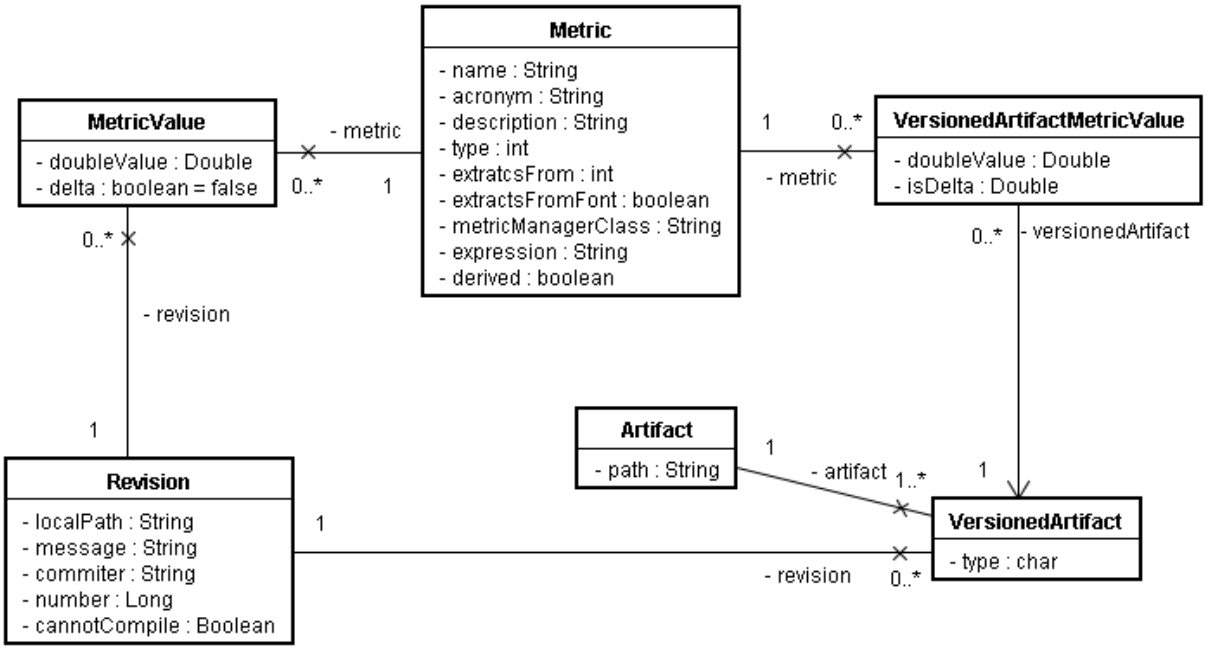


Figura 7: Modelagem Orientada a Objetos do armazenamento das medições.

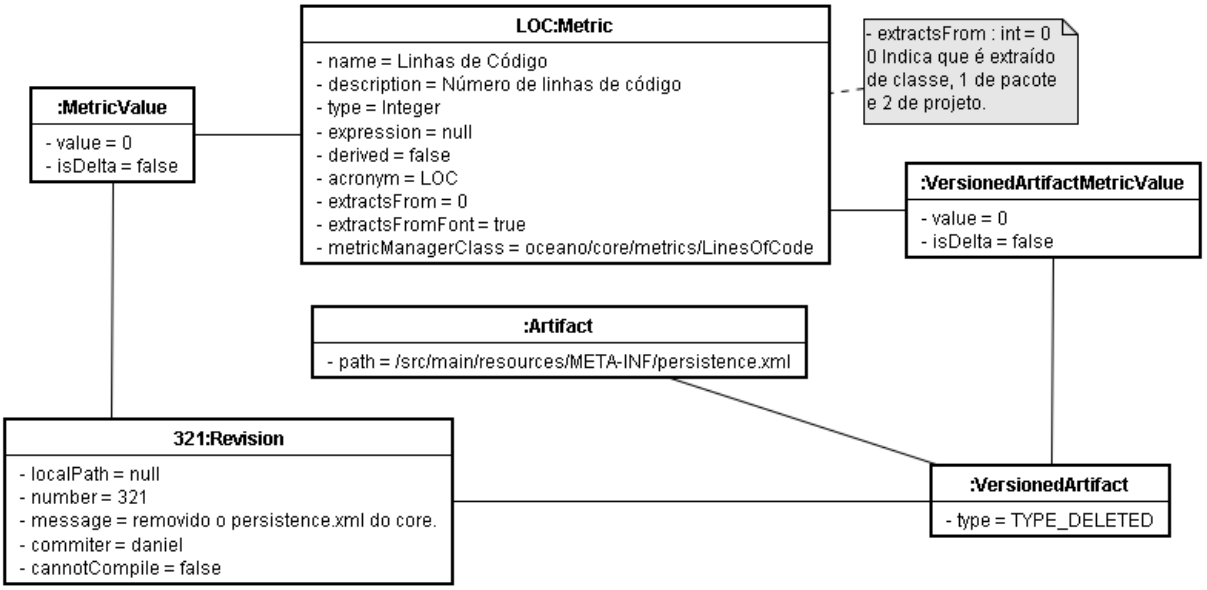


Figura 8: Diagrama de objetos mostrando um exemplo de medição realizada com o *commit* #321.

3.2.4 CÁLCULO DO DELTA

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 de sua evolução, mas apenas como ele está em um determinado momento. Consequentemente, num contexto evolutivo, falta uma medida que

avalie como o software evoluiu com as mudanças ao longo do tempo. Nessa seção é apresentado o cálculo do delta, que é a medida utilizada para mostrar como a qualidade do software variou em uma modificação de acordo com a métrica considerada.

Se um gerente quer saber qual o tamanho o software, a métrica LOC pode ser utilizada, ou se um desenvolvedor quer descobrir quão complexo está o código, a métrica Complexidade Ciclomática de McCabe pode ser utilizada. Portanto, essas métricas definem o valor absoluto das propriedades do projeto, i.e., LOC para um projeto é o somatório da quantidade de linhas de código de cada arquivo do projeto.

Entretanto, no contexto evolutivo, o objeto de estudo não é o projeto propriamente dito, mas as mudanças realizadas a ele que o transformam consecutivamente de uma versão na próxima. 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 é um valor anormal e deve-se atuar sobre ele. Para exemplificar: em um projeto com 260.000 linhas de código que possui uma variação média em torno de 100 linhas por *commit*, um desenvolvedor adiciona em um único *commit* 1.000 LOC, levando a métrica LOC do projeto para 261.000. 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 antecipar prováveis futuros problemas. Esse problema é ainda pior se as 1.000 linhas de código foram adicionadas a 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.

Para lidar com essa situação, é extraído de cada medição de classe outro valor: o delta. Ele representa a variação entre um artefato modificado e sua versão anterior, para todos os artefatos alterados no mesmo *commit*. Analogamente, o mesmo é feito para medidas de pacote.

Métricas cujo alvo é o projeto também possuem o valor delta, mas ele considera apenas o valor absoluto da revisão anterior, como um todo. Consequentemente, o valor do delta para métricas de projeto é igual à diferença entre a medição da revisão como um todo e sua versão anterior. Para métricas de pacote ou classe, a versão considerada não necessariamente é a anterior, mas sim o ultimo *commit* que modificou a classe ou pacote em

questão. Isso acontece, pois uma classe pode não sofrer alterações durante vários *commits*, mas após alguns dias voltar a ser alterada.

Na Figura 9, são ilustradas modificações em classes ao longo de quatro *commits*. As linhas representam a existência de cada classe e as setas (abaixo de cada número de revisão) representam modificações realizadas à classe, indicando um momento específico do tempo em que as modificações foram realizadas, durante um *commit*. A seta branca representa que a classe foi adicionada, a seta cinza representa modificação, e a seta preta representa a remoção da classe na revisão. Os valores à direta de cada seta representam o valor medido de uma métrica específica. A métrica do exemplo pode ser apenas uma cujo alvo é uma classe já que ela possui um valor para cada versão de cada classe. Para exemplificar, a métrica considerada é LOC. A seta preta não possui um valor ao seu lado, pois não existe um arquivo para ser medido, já que ele foi removido.

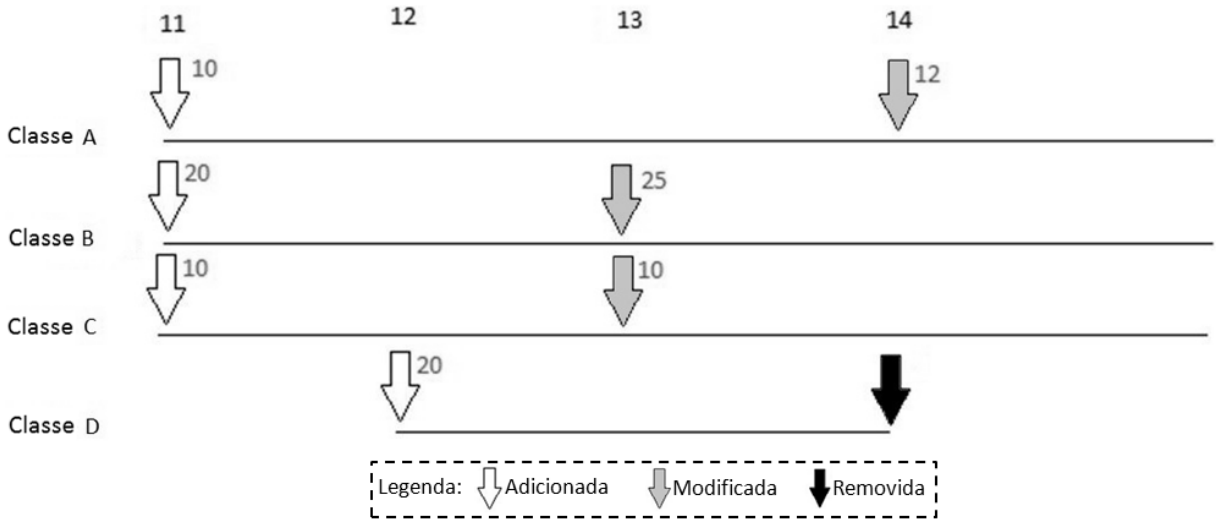


Figura 9: Exemplo de alterações ao longo de revisões.

Como mostrado na Figura 9, na revisão 11, foram adicionadas as classes A, B e C, e os três valores das métricas para elas são, respectivamente, 10, 20 e 10. Assim, o valor delta para cada classe é 10, 20 e 10, e o valor delta para a revisão é 40. Na revisão 12, a Classe D foi adicionada, então o delta é 20 para a revisão e para a classe. Na revisão 13, as classes B e C foram alteradas e os novos valores das métricas são 25 e 10, respectivamente. Assim, o valor delta para a Classe B é 5, para a Classe C é 0[5], e para a revisão é 5. Na revisão 14, a Classe A foi modificada e sua medição resultou em 12 e a Classe D foi removida. Portanto, o valor

[5] É importante notar que mesmo que a Classe C tenha sido modificada, não significa que o valor para a métrica precisa mudar. Por exemplo, se uma linha de código é removida e outra é adicionada, o valor do LOC continuará o mesmo para essa classe.

delta para a Classe A é 2, para a Classe D é -20, ainda que não tenha sido possível medi-la, e para a revisão 14 é -18.

Cada vez que a ação é adicionar uma classe, o delta é o mesmo que o valor da métrica. Por outro lado, se a classe é removida, o delta é o valor negativo da medição. Isso acontece porque a ação oposta a remover é adicionar, e o *commit* afeta o projeto proporcionalmente ao valor da métrica que a classe possuía. Quando a classe é modificada, o delta é a diferença entre os valores das versões nova e antiga.

3.3 FASE DE MINERAÇÃO

A abordagem Ostra possui muito em comum com o processo de KDD, mostrado no Capítulo 2. O apoio dessa abordagem 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.

Após o passo de medição, a Ostra implementa a fase de seleção do processo de KDD ao permitir que o usuário escolha quais métricas e projetos deseja utilizar para construir a base de dados. Desta forma, pode-se minerar o histórico de apenas um projeto, ou verificar os padrões que aparecem ao minerar vários projetos, considerando apenas as revisões que compilam ou todas elas.

Na fase de pré-processamento, os valores medidos são enriquecidos através do cálculo dos deltas. Depois de calculado o delta, as medições originais não são mais utilizadas na mineração de dados. Entretanto, esses valores ainda são úteis para mostrar, através de gráficos, a distribuição e a variação das métricas ao longo da evolução do software.

Na fase de transformação, algumas informações do *commit* são alteradas. Os valores das métricas e outras informações das revisões são transformados de três formas. A data do *commit*, a qual contém a informação dia e hora, é transformada em: dia da semana, turno de trabalho e hora. A segunda transformação é aplicada ao conjunto de arquivos que foram modificados no *commit*, dos quais é indicada apenas a quantidade de arquivos alterados. E a terceira transformação é aplicada aos valores delta, os transformando em alteração positiva, alteração negativa ou zero (sem alteração). Apesar de transformar os dados, essa fase não afeta as informações armazenadas sobre o histórico do projeto. Assim, a transformação afeta apenas a base de dados que é utilizada na mineração.

Para a mineração de dados o delta é transformado, pois são buscadas regras que mostrem o relacionamento entre as métricas, evidenciando a sua influencia sobre as outras. Por exemplo, deseja-se descobrir o que faz a complexidade ciclomática aumentar ou diminuir, e não em quanto à complexidade aumenta ou diminui. Ainda assim, como os valores primários medidos não são perdidos, pode-se utilizar outra transformação, que não esteja focada apenas no sinal da variação, mas agrupando em conjuntos de variação baixa, média ou alta ou da forma que for mais interessante para quem estiver analisando as regras.

Na base de dados a ser minerada, cada transação é equivalente a um *commit*. São utilizados os seguintes dados para criar as transações:

- O nome do projeto e o número da revisão (apenas 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.

Uma das informações provenientes do momento do *commit* é o turno de trabalho, que é dividido em quatro intervalos: madrugada, manhã, tarde e noite. Cada turno tem um período de seis horas. O turno da madrugada vai de 00h00min às 05h59min horas, a manhã vai de 06h00min às 11h59min, a tarde vai de 12h00min às 17h59min e a noite vai de 18h00min às 23h59min.

Se a revisão não contém alguma das informações utilizadas, um símbolo específico é utilizado para marcar e identificar que existe um valor faltante. Quando a compilação falha, a revisão pode ficar sem alguns valores de métricas, pois para realizar a medição, de algumas métricas, é necessário que o projeto seja compilado.

Finalmente, na fase de mineração de dados, são mineradas regras de associação, com o algoritmo Apriori (AGRAWAL; SRIKANT, 1994). Para calibrar a mineração, medidas de regras de associação podem ser utilizadas. As medidas mais utilizadas são suporte, confiança e *lift*. O suporte indica a frequência dos elementos envolvidos na regra na base de dados. A confiança indica qual a probabilidade de acontecer o consequente dado o precendete. Finalmente, o *lift* indica quanto o precendete aumenta a chance de ocorrer o consequente. A abordagem não estabelece quais valores ou medidas de mineração de dados devem ser utilizados nessa fase, mas optou-se por utilizar essas três medidas (HAN *et al.*, 2011; WITTEN; FRANK, 2005).

3.4 FASE DE APRESENTAÇÃO

Ao longo do processo descrito nas seções anteriores, muitos dados são coletados, e, ainda que não sejam utilizados na mineração de dados, podem transmitir informações relevantes à tomada de decisões gerenciais e auxiliar na avaliação das regras mineradas. Na mineração de dados, é utilizado o delta das métricas para descrever a alteração realizada. Mas, como discutido anteriormente, os valores absolutos das métricas, utilizados para calcular os deltas, também podem ser utilizados para fornecer informações relevantes a respeito de uma versão específica do software, contrastando-a ao histórico do projeto.

Nessa seção, serão apresentadas formas de apresentação dos resultados obtidos pela Ostra: gráficos históricos, histogramas, regras de associação e tabelas de comportamento. Inicialmente, os gráficos históricos são apresentados na Seção 3.4.1, o histograma na Seção 3.4.2, as regras de associação na Seção 3.4.3 e a Tabela de Comportamento na Seção 3.4.4.

3.4.1 GRÁFICO HISTÓRICO

Ao final da fase de medição, já é possível obter informações sobre o projeto: como está a qualidade atual do software, como foi a evolução das métricas (simples e compostas) e como foi cada *commit* individual. Essas informações podem ser visualizadas a partir de dois gráficos: o histograma dos valores medidos para cada métrica, apresentado na Seção 3.4.2, e o gráfico de controle (TAGUE, 2005), que mostra a evolução da métrica ao longo das revisões, ou seja, do tempo. O gráfico de controle mostra o histórico do projeto sob a perspectiva de uma métrica, e por isso é chamado de gráfico histórico.

O gráfico de controle (TAGUE, 2005), ilustrado na Figura 10, considera o desvio padrão das medições para identificar valores estranhos, que devem ser verificados. Nele, a linha verde representa a média, enquanto a linha amarela representa a variação de um desvio padrão e a linha vermelha representa a variação de três desvios padrões. As linhas vermelhas são chamadas de limite inferior e superior de controle. Os valores que estiverem acima ou abaixo da linha vermelha devem ser verificados, pois, provavelmente, representam uma anomalia.

Se os valores da métrica sendo analisada com o gráfico possuir uma distribuição normal, ele se torna particularmente interessante. Nesse caso, a chance de uma medição estar dentro do limite delimitado pela linha amarela é cerca de 70%, enquanto a chance de estar dentro dos limites da linha vermelha é de mais de 99%. Dessa forma, se algum valor estiver além dessas linhas, existe uma chance de ser uma anomalia. Para métricas cujos valores não seguem uma distribuição normal, fica a critério do usuário a utilização desses gráficos.

Por exemplo, o gráfico histórico da métrica Tamanho do Projeto em Classes está ilustrado na Figura 10. Nesse gráfico, é possível visualizar como a métrica se comporta ao longo da evolução do software. Pode-se reparar que, inicialmente, o projeto analisado possuía cerca de 50 classes, aumentando nos três primeiros meses de desenvolvimento (130 mil minutos iniciais). Após os três meses iniciais, a quantidade de classes seguiu uma tendência de diminuição, voltando a crescer somente após quase dois anos de desenvolvimento (1 milhão de minutos). Dois meses depois (100 mil minutos), se estabilizou e se manteve constante até a última versão analisada, tendo cerca de 60 classes.

Figura 10: Gráfico do histórico da métrica Tamanho do Projeto em Classes.

Dessa forma, são propostas duas utilizações do gráfico histórico: verificar a versão atual e entender como se comportou a evolução da métrica em questão. Para a primeira utilização, o valor atual da métrica pode ser utilizado para disparar um alarme (manual ou automático) de acordo com valores pré-estabelecidos ou através da análise do gráfico. Um exemplo de alarme seria indicar refatorações em classes que possuam medições acima de 10 para a métrica Complexidade Ciclomática de McCabe (HENDERSON-SELLERS, 1995). Na segunda utilização, o histórico da métrica pode servir para entender como foi a sua variação em função dos fatores externos ao desenvolvimento do projeto, como, por exemplo, decisões políticas que afetem o projeto, feriados e início ou final de mês, entre outros. Decisões do projeto, como, por exemplo, quais funcionalidades devem ser implementadas em quais momentos, implantação de técnicas ou ferramentas de desenvolvimento ou entrada e saída de membros no time de desenvolvimento também podem influenciar a variação das métricas e seus impactos podem ser observados nesse gráfico.

Com esse gráfico, um gerente pode notar como se comporta uma determinada métrica, o que auxilia a entender melhor como seu projeto evoluiu em relação às características avaliadas por ela. Também é possível verificar se uma versão é melhor ou pior que outras versões no que diz respeito a uma determinada métrica. Além disso, com as mudanças de comportamento do gráfico, é possível verificar se ações tomadas para melhorar a qualidade do produto estão surtindo efeito.

Outra utilidade desse gráfico é permitir identificar se a métrica atual do projeto está acima ou abaixo de um valor esperado. A Complexidade Ciclomática de McCabe, por exemplo, está diretamente relacionada à facilidade de testar o código. Essa métrica possui valor esperado abaixo de 10 por método (HENDERSON-SELLERS, 1995). Assim, é possível analisar, com o gráfico histórico, se algumas revisões possuem um valor que extrapola o valor esperado. Se esse gráfico for acompanhado diariamente, será possível perceber que a versão atual está fora do esperado (de acordo com alguma métrica) no momento em que isto acontecer, diminuindo assim o tempo de resposta e, provavelmente, diminuindo o custo de manutenção e tempo de correção do projeto.

3.4.2 HISTOGRAMA

Com o histograma, pode-se entender mais sobre os valores de uma métrica e mudanças que ocorreram no software. Na Figura 11, é mostrado o histograma da métrica Complexidade Ciclomática de McCabe. Nele, é possível notar que a maioria dos valores é menor que 5 e à medida que os valores aumentam, a ocorrência diminui. Dessa forma, se aparecerem muitos valores altos consecutivamente, algum problema deve ter ocorrido, seja arquitetural ou de programação.

Na Figura 11, é possível verificar que o intervalo entre 70 e 75 de Complexidade Ciclomática de McCabe possui algumas ocorrências (cerca de cinco). Porém, esses são valores altos para essa métrica, que não deveriam passar de 10, e não deveriam possuir ocorrências. A ocorrência de valores tão altos de Complexidade Ciclomática de McCabe neste gráfico pode alertar um possível problema. Esse gráfico pode ser utilizado para direcionar refatorações que objetivam facilitar a compreensão ou testes do software, iniciando o trabalho pelas classes afetadas por *commits* nos intervalos mais extremos. A utilização do histograma é para auxiliar a avaliação do gráfico histórico, já que ele mostra de forma mais clara a dispersão dos dados.

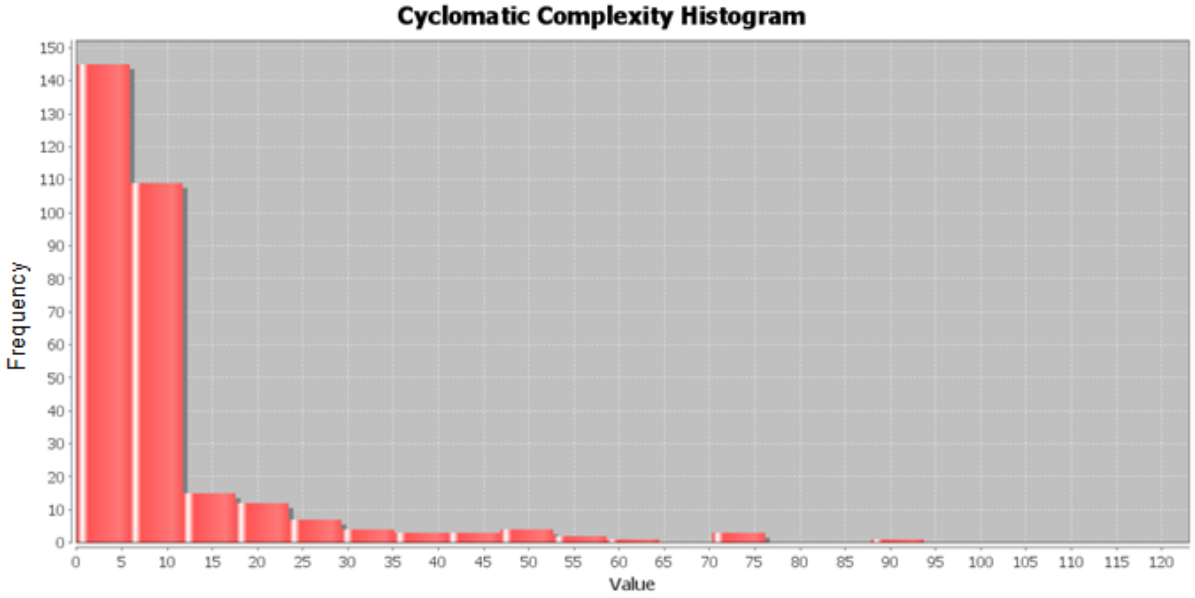


Figura 11: Histograma da Complexidade Ciclomática de McCabe.

3.4.3 REGRAS DE ASSOCIAÇÃO

As regras de associação são extraídas na fase mineração de dados e são a forma principal de descoberta de informação 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*).

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. Para auxiliar na sua análise, são disponibilizados filtros por atributo e filtros de tamanho do precedente ou consequente.

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 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 18 horas, 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 de tamanho 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 tentem a compilar menos que 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".

3.4.4 TABELA DE COMPORTAMENTO

A tabela de comportamento é uma ferramenta para auxiliar o estudo da relação entre os atributos de qualidade e métricas de software, através de uma exibição gráfica das regras de associação obtidas. Essas regras são mineradas considerando uma métrica no precedente e outra no consequente. Enquanto os dois gráficos mostrados anteriormente auxiliam a interpretar os dados obtidos na fase de medição, a tabela de comportamento auxilia a interpretar as regras encontradas na fase de mineração de dados.

A utilização dessa ferramenta é importante quando se deseja identificar padrões comportamentais que ressaltam a influência entre as métricas em um ou mais projetos. Por exemplo, quando o objetivo é saber como se comportam os atributos de qualidade em projetos web, deve-se medir um conjunto de projetos desse tipo, construir uma base de dados com eles e verificar quais comportamentos são identificados na tabela de comportamento gerada após a mineração de dados[6]. Na Figura 12, é mostrada uma tabela de comportamento relacionando os atributos de qualidade de QMOOD.

A tabela de comportamento é estruturada de forma que cada célula mostra a influência que a métrica da linha tem sobre a métrica da coluna, ou seja, a célula $C_{i,j}$ mostra a influência que a métrica da linha *i* exerce sobre a da coluna *j*. Ou ainda, a célula $C_{i,j}$ representa as regras de associação com a métrica da linha *i* no precendete e a métrica da coluna *j* no consequente.

Cada célula possui valores de suporte, confiança e *lift*, que são indicados pelas letras S, C e L, respectivamente. Esses valores são os da regra de associação com o maior *lift* dentre as

[6] Nesse caso, deve-se tomar cuidado com a interferência que projetos com quantidades de *commits* que desviam da maioria podem causar na base de dados. Por exemplo, se um projeto possui muito mais *commits* que outro na mesma base, as regras encontradas podem ser válidas por causa desse projeto em particular, e não por ocorrer em todos os projetos.

regras que embasam esse comportamento[7]. As regras de associação mineradas têm a seguinte estrutura: "comportamento da métrica *i*" → "comportamento da métrica *j*". Por exemplo: "diminuição da flexibilidade" → "diminuição do entendimento", indicando que quando a flexibilidade diminui, o entendimento também diminui. Porém pode ser que apareçam, ao mesmo tempo, mais de uma regra de associação envolvendo a métrica da linha i e a métrica da coluna *j* em função das calibragens de suporte, confiança e *lift*. Por exemplo, "diminuição da flexibilidade" → "diminuição do entendimento" e "aumento da flexibilidade" → "aumento do entendimento", simultaneamente. Dessa forma, os valores de S, C e L mostrados, são os da regra com o maior *lift*, dentre as regras encontradas para a métrica da linha *i* e a métrica da coluna *j*. Vale ressaltar que cada transação possui os dados apresentados na Seção 3.3.

	Efe	Ext	Fle	Fun	Reu	Ent
Efe		S:0.18 C:0.54 L:1.72	S:0.15 C:0.47 L:1.45	S:0.17 C:0.50 L:1.13	S:0.18 C:0.52 L:1.15	S:0.19 C:0.57 L:1.26
Ext	S:0.18 C:0.56 L:1.72		S:0.27 C:0.80 L:2.67	S:0.25 C:0.64 L:1.42	S:0.23 C:0.59 L:1.30	S:0.28 C:0.71 L:1.57
Fle	S:0.15 C:0.48 L:1.45	S:0.27 C:0.82 L:2.67		S:0.23 C:0.58 L:1.27	S:0.21 C:0.51 L:1.13	S:0.26 C:0.64 L:1.40
Fun	S:0.17 C:0.39 L:1.13	S:0.25 C:0.55 L:1.42	S:0.23 C:0.51 L:1.27		S:0.41 C:0.91 L:2.01	S:0.35 C:0.77 L:1.69
Reu	S:0.18 C:0.40 L:1.15	S:0.23 C:0.51 L:1.30	S:0.21 C:0.47 L:1.13	S:0.41 C:0.91 L:2.01		S:0.33 C:0.73 L:1.60
Ent	S:0.19 C:0.43 L:1.26	S:0.28 C:0.61 L:1.57	S:0.26 C:0.57 L:1.40	S:0.35 C:0.77 L:1.69	S:0.33 C:0.73 L:1.60	

Figura 12: Tabela de comportamento dos atributos de qualidade de QMOOD.

[7] A tabela de comportamento mostra a cor do comportamento de acordo com as regras encontradas para a métrica da linha e da coluna. Entretanto, quando é minerada mais de uma regra para a mesma célula, as regras devem ser observadas para se entender melhor o comportamento resultante. Por exemplo, o suporte, confiança e *lift* apresentados na célula são da regra com maior *lift*. Dessa forma, para saber as outras medidas de interesse, as outras regras devem ser analisadas. Na Seção 4.5.5, é mostrada a implementação da tabela de comportamento ilustrando o caso de múltiplas regras para a mesma célula, ou seja, para o mesmo par de métricas *i* e *j*. Nessa seção, é exemplificado como o protótipo auxilia na análise das regras que definem o comportamento de cada célula.

As cores das células indicam o comportamento identificado nas regras de associação envolvendo as métricas da linha e coluna. Na Figura 12, por exemplo, o fato de a célula $C_{3,6}$ ser verde indica que quando a flexibilidade diminui, o entendimento também diminui. Já a célula $C_{3,4}$, vermelha, indica que quando a flexibilidade aumenta, a funcionalidade diminui.

Na Figura 13, são definidas as cores básicas das células da tabela de comportamento e indicam a semântica do comportamento. Nas linhas e nas colunas, estão os valores "+", "0" e "-", representando, respectivamente, aumento do atributo, manutenção do seu valor ou diminuição. Por exemplo, a primeira célula ilustra que verde é a cor utilizada para representar que quando a linha diminui a coluna também diminui, enquanto o azul indica que quando a linha aumenta a coluna também aumenta. Na Tabela 18, são detalhados os comportamentos.

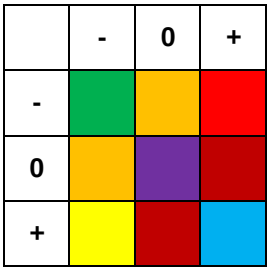


Figura 13: Cores básicas da tabela de comportamento de acordo com a regra encontrada.

Tabela 18: Significado das cores da tabela de comportamento.

Cor	Comportamento	Significado
Verde	Proporcional decrescente	Quando a métrica da linha diminui, a da coluna diminui.
Azul	Proporcional crescente	Quando a métrica da linha aumenta, a da coluna aumenta.
Roxo	Estável	Quando a métrica da linha se mantém estável, a da coluna se mantém estável.
Vermelho	Inversamente proporcional decrescente-crescente	Quando a métrica da linha diminui, a da coluna aumenta.
Amarelo	Inversamente proporcional Crescente-decrescente	Quando a métrica da linha aumenta, a da coluna diminui.
Mostarda	Inversamente proporcional estável-decrescente	Quando a métrica da linha se mantém estável, a da coluna diminui. Ou vice-versa.
Marrom-escuro	Inversamente proporcional estável-crescente	Quando a métrica da linha se mantém estável, a da coluna aumenta. Ou vice-versa.
Cinza	Não definido	Não é possível definir um comportamento, pois aparecem regras inversas e proporcionais ao mesmo tempo.

Cada célula da tabela de comportamento representa o relacionamento entre as métricas da linha e coluna. Porém, pode ser que ao minerar a base de dados em busca de regras, apareça mais de uma regra sobre duas métricas. Por exemplo, pode ser que apareçam ao mesmo tempo, as regras: "quando a reusabilidade diminui, o entendimento diminui" (cor

verde) e "quando a reusabilidade se mantém estável, o entendimento se mantém estável" (cor roxa). Dessa forma, faz-se necessária a definição das cores que representarão as possíveis combinações de comportamento. No exemplo, como os dois comportamentos são estáveis, um proporcional decrescente e outro estável, o comportamento resultante é proporcional decrescente, indicado pela cor verde.

Na Figura 14, são ilustradas as combinações de comportamentos básicos. No caso da combinação do comportamento estável com o inversamente-proporcional crescente-decrescente e inversamente-proporcional decrescente-crescente, os comportamentos resultantes são respectivamente: inversamente-proporcional, estável-decrescente e inversamente-proporcional estável-crescente.

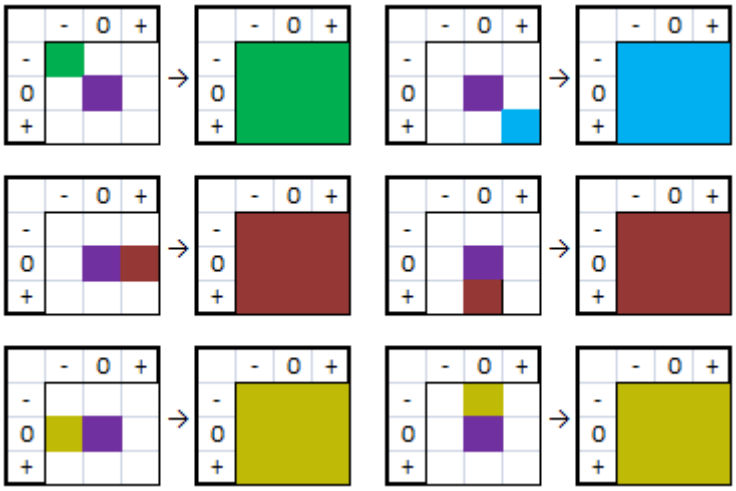


Figura 14: Combinações de comportamentos.

É importante ressaltar que podem existir várias regras de associação envolvendo duas métricas. Consequentemente, o comportamento resultante depende da combinação dos comportamentos individuais de cada regra. Se forem encontrados apenas comportamentos proporcionais ou não proporcionais, o comportamento resultante será proporcional ou não proporcional, respectivamente. Porém, se forem encontradas regras que evidenciem comportamentos proporcionais e não proporcionais ao mesmo tempo para um par de métricas, não é possível identificar com clareza o comportamento resultante. Assim, é utilizada a cor cinza, indicando que não se pode definir um comportamento. Na Figura 15, estão ilustradas as situações de conflito, que são representadas pela cor cinza na tabela de comportamento.

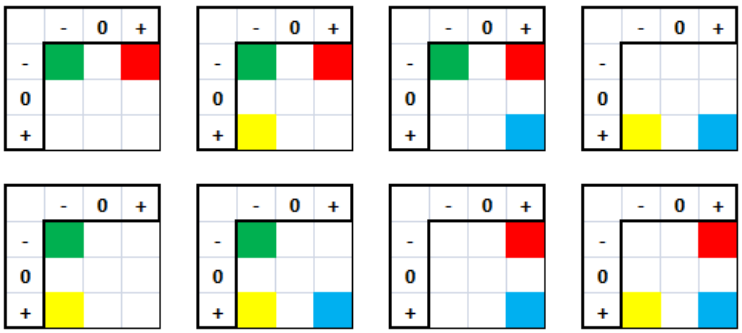


Figura 15: Conflitos de comportamentos que resultam na cor cinza.

3.5 USOS ESPERADOS

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[8] de empresas 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.

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. 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.

A utilização da Ostra por um escritório de projetos está voltada para descobrir padrões evolutivos dos seus projetos, os quais são importantes para auxiliar no controle da qualidade. Com esse propósito, o escritório de projetos pode utilizar a tabela de comportamento para descobrir como ocorre a variação dos atributos de qualidade em seus projetos e divulgar para os gerentes de projeto esses padrões. Posteriormente, quando um novo projeto for

[8] Escritório de projetos (do inglês *Project Management Office*, PMO) é uma entidade organizacional à qual são atribuídas responsabilidades relacionadas à gerência e coordenação centralizada de projetos dentro de uma empresa (PROJECT MANAGEMENT INSTITUTE, 2008).

desenvolvido, é possível notar, ao analisar a tabela de comportamento desse projeto, se ele possui algum comportamento anormal e atuar para identificar se estão ocorrendo problemas arquiteturais, débitos técnicos ou até descobrir algum padrão positivo que deve ser replicado para os outros projetos.

A terceira aplicação da Ostra é em pesquisas cujo objetivo seja descobrir padrões sobre a evolução de softwares de forma geral, contribuindo assim para a área de engenharia de software como um todo e não apenas para um projeto ou uma empresa. Com esse objetivo, podem ser analisados os gráficos evolutivos e histogramas de diversos softwares para identificar padrões, como, por exemplo, se ao longo da evolução a manutenibilidade tende a aumentar ou diminuir, ou a distribuição mais comum dos valores das métricas. Além disso, as regras de associação podem evidenciar informações interessantes sobre a relação entre as métricas, dias da semana e quantidade de arquivos modificados. Nesse cenário, a tabela de comportamento pode auxiliar a descoberta ou avaliação empírica do relacionamento entre atributos de qualidade.

3.6 CONSIDERAÇÕES FINAIS

O objetivo da abordagem da Ostra é descobrir informações que possam ajudar no entendimento da evolução do software. Acredita-se que, ao se entender melhor sobre a sua evolução, seja possível identificar com maior eficiência problemas que ocorrem durante o desenvolvimento e agir mais rapidamente, conseguindo assim controlar a qualidade do projeto.

Neste capítulo, foi apresentada a abordagem Ostra, que consiste de três passos principais: medição, mineração de dados e apresentação das regras e métricas. No primeiro passo, os projetos são medidos através do código fonte de cada revisão. No segundo passo, é construída uma base de dados com as revisões medidas para ser utilizada a técnica de mineração de regras de associação. No terceiro e último passo, as regras mineradas e as medições são apresentadas com o auxílio de gráficos históricos, histogramas e tabelas de comportamento.

No próximo capítulo, é apresentado o protótipo Ostra, que implementa essa abordagem. Nele, também é apresentado, com um exemplo real, como o protótipo deve ser utilizado para realizar as medições de um projeto e minerar a base de dados em busca de padrões.

CAPÍTULO 4 – O PROTÓTIPO IMPLEMENTADO

4.1 INTRODUÇÃO

No Capitulo 3, foi apresentada a abordagem Ostra, que tem como objetivo fornecer informações sobre a evolução do projeto de software para o processo de tomada de decisões. Essas informações são fortemente baseadas na variação dos atributos de qualidade, definidos como métricas compostas, que podem ser extraídas do histórico de modificações do projeto.

Para possibilitar a avaliação da abordagem, foi construído um protótipo que implementa todos os seus passos, desde o acesso às versões do software até a mineração de regras de associação, de forma a possibilitar a medição automática do histórico do software. Neste capítulo, é apresentado o protótipo implementado, suas funcionalidades e características arquiteturais e um exemplo de utilização. Na Seção 4.2, são apresentados os requisitos do protótipo desenvolvido, na Seção 4.3, a sua arquitetura, na Seção 4.4, os testes que garantem que ele está funcionando corretamente e, na Seção 4.5, a interface e um exemplo de utilização.

4.2 REQUISITOS

Como o objetivo do protótipo é implementar a abordagem Ostra, deve ser possível realizar três ações básicas: acessar cada versão do histórico de modificações de um projeto armazenado no sistema de controle de versão, extrair métricas dessas versões e construir uma base de dados na qual possam ser aplicadas técnicas de Mineração de Dados, aplicar mineração de dados e analisar os resultados. Consequentemente, dessas ações básicas foi construída uma lista de funcionalidades desejadas, implementadas no protótipo. Na Tabela 19, estão listadas as funcionalidades do protótipo Ostra. As ações do usuário estão indicadas com * (asterisco).

Além desses requisitos funcionais, dois requisitos não funcionais são desejados: desenvolver uma infraestrutura que possa ser facilmente reutilizada e também facilmente estendida. O primeiro requisito não funcional surgiu do interesse comum de pesquisas do grupo em acessar repositórios de gerência de configuração e avaliação das abordagens através de métricas de software. O segundo requisito não funcional surgiu da necessidade de evoluir o protótipo, com novas métricas e outros Sistemas de Gerência de Configuração, para ampliar as pesquisas.

Tabela 19: Requisitos do protótipo Ostra.

Grupo	Identificador	Requisitos
Projeto	1 *	Cadastrar projeto
	2 *	Visualizar projetos cadastrados
Métrica	3 *	Cadastrar métrica composta
	4 *	Visualizar métricas cadastradas
Gerência de Configuração	5	Obter log de *commits* do Sistema de Controle de Versões do projeto
	6	Fazer *check out* de revisão
	7	Fazer *update* de revisão
	8	Compilar projeto
Medição	9	Extrair métrica simples
	10	Extrair métrica composta
	11	Calcular valor de métrica de projeto com métricas de artefato
	12	Calcular delta
	13 *	Medir projeto (medir cada versão do projeto com as métricas selecionadas)
Configuração da Mineração de Dados	14	Construir base de dados para mineração
	15 *	Escolher projetos para construir a base de dados
	16 *	Escolher métricas para construir a base de dados
	14 *	Escolher se revisões que não compilam devem ser consideradas na construção da base de dados
	18 *	Definir suporte mínimo
	19 *	Escolher medida de interesse de regra de associação
	20 *	Definir medida de interesse mínima de regra de associação
Exibição das Regras Mineradas	21 *	Visualizar regras mineradas
	22 *	Filtrar regras de associação por atributos do precedente ou consequente
	23 *	Filtrar regras de associação por tamanho do precedente ou consequente
	24 *	Exportar base de dados em formato externo
Visualização	25 *	Visualizar tabela de comportamento
	26 *	Visualizar medições de um projeto
	27 *	Visualizar gráfico histórico da variação de métrica para projeto
	28 *	Visualizar gráfico histórico de métrica para projeto
	29 *	Visualizar histograma de métrica para projeto

4.3 ARQUITETURA

Como discutido anteriormente, os requisitos não funcionais que moldaram a arquitetura do protótipo são a facilidade de extensão e a facilidade de reutilização. Assim, a arquitetura foi planejada de forma a possibilitar o crescimento do projeto, com a adição de novas métricas, ferramentas de GC e técnicas de mineração de dados.

A arquitetura foi concebida de forma que esses três pilares da abordagem se mantivessem independentes e pudessem ser alterados sem afetar os outros. Isto facilita a extensão, pois são necessárias alterações pequenas para adicionar novas capacidades ao protótipo. Por exemplo, para adicionar uma nova métrica, apenas uma classe deve ser implementada, como será apresentado na Seção 4.3.2.

O protótipo Ostra foi implementado em dois módulos, ou seja, dois projetos: um módulo central que disponibiliza a infraestrutura e outro com a interface com o usuário, que

utiliza o primeiro. Dessa forma, as funcionalidades que são reutilizáveis e comuns a outras pesquisas do grupo estão nesse módulo central, enquanto a interface com o usuário está em outro módulo. O módulo central do protótipo, que é utilizado por outros protótipos, chama-se Oceano[9] Core, enquanto o módulo de interface com o usuário chama-se Oceano Web. Esses módulos estão ilustrados na Figura 16. No módulo Oceano Core, estão as funcionalidades de medição, Gerência de Configuração e Mineração de Dados. Por outro lado, as implementações dos outros requisitos da Tabela 19 estão no Oceano Web.

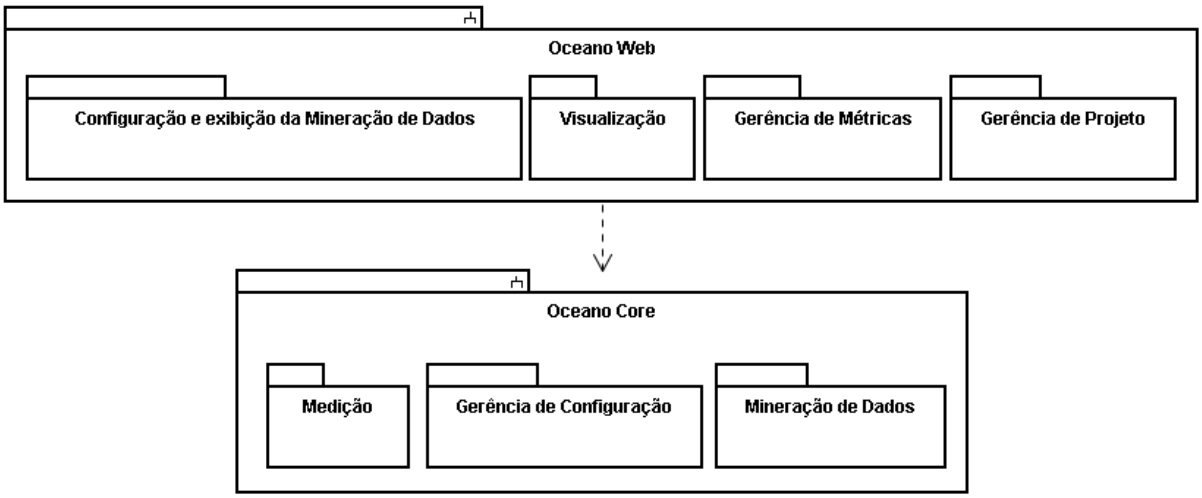


Figura 16: Módulos do protótipo Ostra.

A interface da Ostra (ou protótipo Ostra) está no Oceano Web, que é uma aplicação web, como o nome indica. A concepção do protótipo como uma interface web tem como objetivo viabilizar que a aplicação rode num servidor, com maior capacidade de processamento e armazenamento, enquanto o usuário pode analisar o histórico do projeto desejado e as regras mineradas de onde preferir.

A arquitetura do protótipo é dividida em camadas e tem como principais padrões de projeto: *Data Access Object* (DAO), *Model-View-Controller* (MVC), *Service* e *Factory* (ALUR *et al.*, 2003; GAMMA *et al.*, 1995). No Oceano Core, estão as camadas de modelo, serviço e *DAO*, enquanto a camada de visualização está no Oceano Web.

As três disciplinas, Métricas de Software (MS), Gerência de Configuração (GC) e Mineração de Dados (MD), permeiam as camadas da arquitetura, possuindo implementação em cada uma delas. Na Figura 16, é ilustrada a implementação das funcionalidades dessas

[9] Este nome é inspirado no oceano, habitat de vários animais, dentre eles a ostra. Como o objetivo é ser um módulo central e comum, este nome foi escolhido.

disciplinas nos pacotes *Medição*, *Gerência de Configuração* e *Mineração de Dados* respectivamente.

As informações de MS, GC e MD que são utilizadas e a forma que se combinam na abordagem Ostra foram apresentadas no Capítulo 3. Essas informações são modeladas como objetos persistentes (i.e., entidades) e mapeadas diretamente para tabelas do banco de dados em tabela homônimas. Para o mapeamento dos objetos, acesso a eles e persistência foi utilizado o framework *Hibernate* (JBOSS COMMUNITY, 2011) como implementação da especificação *Java Persistence API* (JPA) (ORACLE, 2011).

Nas próximas seções, serão mostradas as implementações da modelagem de dados, serviços e visualizações de cada disciplina. Na Seção 4.3.1, é detalhada a arquitetura da parte de GC, na Seção 4.3.2, a parte de MS e, na Seção 4.3.3, a parte de MD.

4.3.1 GERÊNCIA DE CONFIGURAÇÃO

Os conceitos de GC utilizados na abordagem foram apresentados no Capítulo 3. As informações contidas no histórico do software também são armazenadas na base de dados durante a medição, junto das métricas. Cada artefato que compõe o projeto e todas as suas versões são armazenadas. As classes que modelam as informações de GC e são persistidas em tabelas homônimas no banco de dados são mostradas no diagrama de classes da Figura 17, com fundo na cor cinza claro. Na Figura 17, existem classes da medição cuja explicação está na Seção 4.3.2. Na Figura 17, com fundo na cor cinza escuro, está a classe *OceanoUser* com os dados de *login* no protótipo Ostra. Essa classe se relaciona aos projetos cadastrados através de *ProjectUser*, que permite que cada usuário do protótipo possa ter acesso aos projetos com seu *login* e senha, quando necessários, pois alguns projetos aceitam acesso anônimo. Na interface do protótipo, apenas são exibidos projetos que o usuário tenha acesso, o que é registrado em *ProjectUser*.

O acesso aos repositórios de GC e à compilação automática são realizados através das classes *VCSService* e *CompilerService*, respectivamente. A Figura 18 apresenta um diagrama de classes com os serviços de GC. Com fundo cinza escuro estão os serviços de acesso ao repositório, com fundo cinza claro estão as classes relacionadas à compilação e de utilidade e com fundo branco estão os serviços de medição que utilizam GC. O único sistema de GC com acesso implementado no protótipo é o Subversion (COLLINS-SUSSMAN *et al.*, 2008), porém a Ostra foi criada tendo em mente esse ponto de extensão. O serviço de medição *OstraMetricService* utiliza o *VCSService* para acessar as versões do software sendo medido e o *MeasurementService* utiliza o *CompilerService* para compilar o software, pré-requisito para

a extração das métricas compiladas. *VCS* é a interface que deve ser implementada para disponibilizar um novo Sistema de Controle de Versão. As classes responsáveis pela implementação do Subversion são *SVN_By_CommandLineInterface* e *SVN_By_SVNKit*. A primeira utiliza chamadas diretas ao Subversion e necessita a sua instalação na máquina em que a Ostra está instalada, e a segunda utiliza a biblioteca SVN Kit (TMATE SOFTWARE, 2011) para acessar o Subversion sem precisar de ele estar instalado. O serviço *VCSService* é responsável por disponibilizar as funcionalidades relativas ao VCS, utilizando a implementação devida de acordo com o repositório adequado para o projeto.

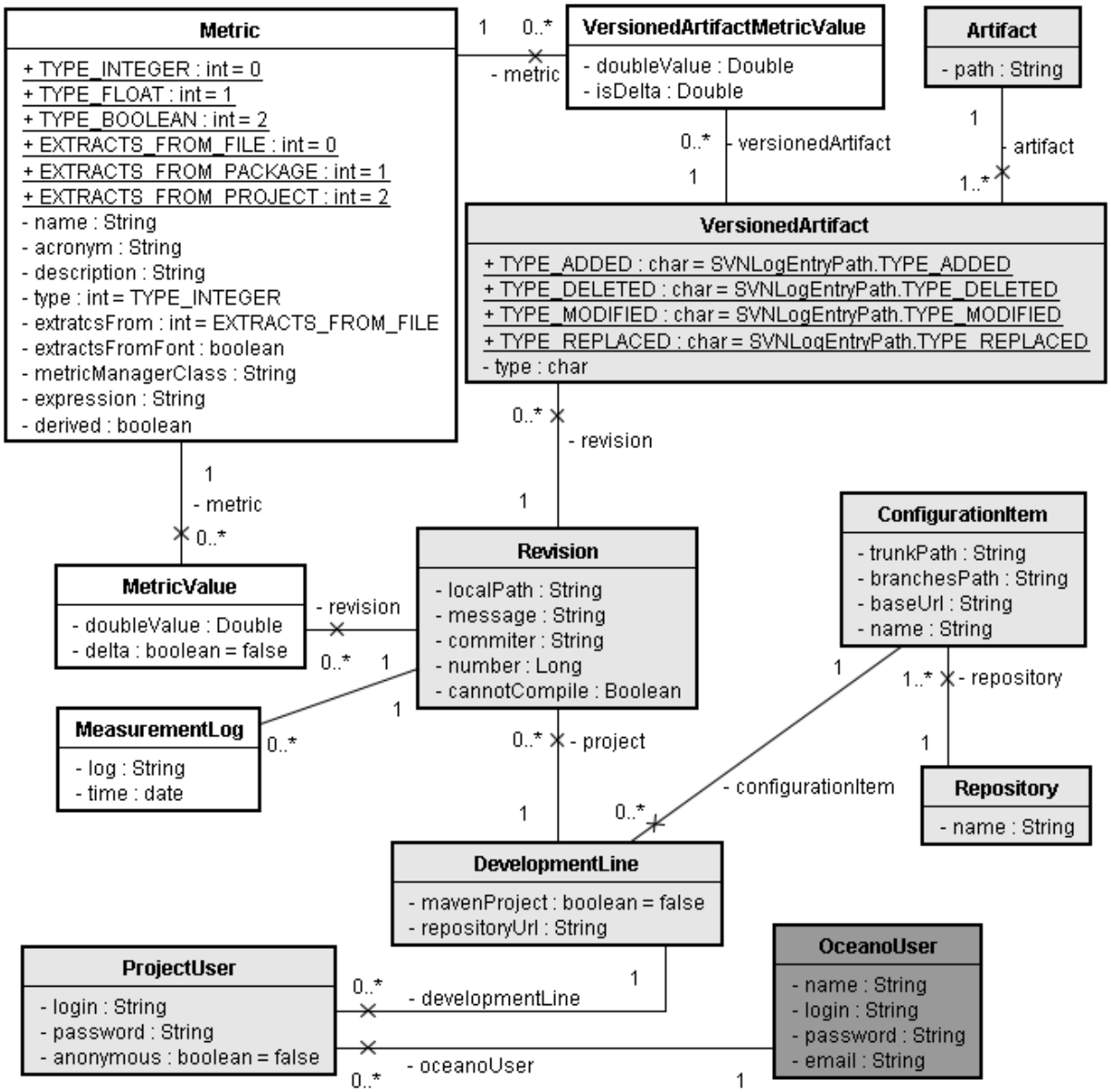


Figura 17: Diagrama de classes da parte persistente da medição e de Gerência de Configuração.

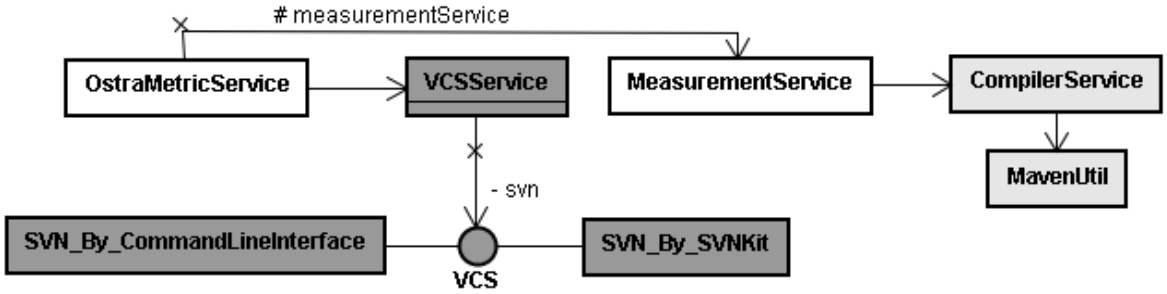


Figura 18: Diagrama de classe com os serviços de Gerência de Configuração.

Para incluir um repositório na Ostra é necessário: (1) implementar a interface VCS, (2) colocar na tabela *Repository* um registro para o novo repositório e (3) indicar no *VCSService* a nova implementação. Na Figura 19, é mostrado um diagrama de objetos que ilustra a implementação do Sistema de Controle de Versão Git (TORVALDS; HAMANO, 2006), que ainda não possui conector na versão atual da Ostra. Nesse exemplo: (1) a classe *Git_CLI* implementa a interface *VCS* utilizando linha de comando (CLI, do inglês *Command Line Interface*), (2) foi criada uma instância em *Repository* para o Git e (3) foi indicado no *VCSService*, através do atributo *git*, que a classe *Git_CLI* deve ser utilizada quando o projeto referenciar o repositório Git.

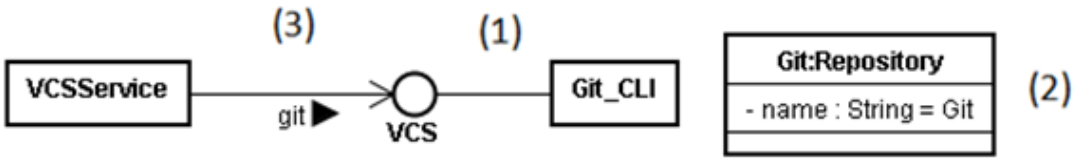


Figura 19: Diagrama de objetos mostrando um exemplo de implementação de VCS.

O serviço responsável pela compilação do projeto é o *CompilerService*, que utiliza a classe *MavenUtil*. O Sistema de Gerenciamento de Construção suportado é o Maven (O'BRIEN *et al.*, 2008).

4.3.2 MEDIÇÃO

As métricas viabilizam a avaliação do projeto sendo observado, pois permitem que as suas versões sejam descritas de forma quantitativa e objetiva. As métricas devem ser medidas e armazenadas, formando uma base de dados que é utilizada na mineração de dados.

A interface *MetricManager* representa o gerenciador de métricas e deve ser implementada sempre que uma nova métrica for criada. Ela define métodos para medir um artefato ou o projeto. Além de implementar esta interface, para disponibilizar a métrica na ferramenta, deve ser inserido um registro de métrica na base de dados, na tabela *Metric*,

indicando qual classe implementa essa métrica. Além das métricas apresentadas no Capítulo 3, o protótipo também conta com algumas outras. As métricas simples disponíveis na Ostra estão na Tabela 20 e, no diagrama de classes da Figura 20 são apresentadas as classes que implementam essas métricas. Na coluna alvo, dessa tabela, é indicado se a métrica é calculada de classe, pacote ou de projeto. A última coluna dessa tabela, compilada, indica se para medir a métrica é necessário compilar o projeto antes.

Tabela 20: Conjunto de métricas da Ostra, com a classificação do alvo da medição.

Sigla	Nome	Alvo	Compilada
RMA	*Abstractness*	Pacote	Sim
ANA	*Average Number Of Ancestors*	Projeto	Não
CIS	*Class Interface Size*	Classe	Não
DAM	*Data Access*	Classe	Não
DSC	*Design Size In Classes*	Projeto	Não
DCC	*Direct Class Coupling*	Classe	Não
LCOM	*Lack Of Cohesion Of Methods*	Classe	Não
MOA	*Measure Of Aggregation*	Projeto	Não
MFA	*Measure Of Functional Abstraction*	Projeto	Não
MLOC	*Method Lines Of Code*	Classe	Não
NOH	*Number Of Hierarchies*	Projeto	Não
NOI	*Number of Interfaces*	Pacote	Sim
NOM	*Number Of Methods*	Classe	Não
NORM	*Number of Overridden Methods*	Pacote	Não
NOP	*Number Of Polymorphic Methods*	Projeto	Não
NSF	*Number Of Static Attributes*	Classe	Não
NSM	*Number Of Static Methods*	Classe	Não
LOC	*Lines Of Code*	Classe	Sim
NOA	*Number Of Attributes*	Classe	Não
CAM	*Cohesion Among Methods In Class*	Classe	Não
TLOC	*Lines of Code Total*	Projeto	Sim
TCC	*Total Cyclomatic Complexity*	Classe	Sim

Parte da implementação da medição foi realizada pelo aluno de graduação Wallace Ribeiro, que participou da pesquisa durante a sua iniciação científica. Além da implementação das métricas simples e compostas, também foram implementados por ele o gráfico de controle e o histograma. Esta contribuição foi apresentada em seu projeto final (RIBEIRO, 2011). Vale ressaltar que o processo de medição, que acessa as versões do software e realiza as medições, não foi implementado por ele. As definições operacionais das métricas estão no Apêndice A.

Diferente das métricas simples, que necessitam ser codificadas para se tornarem disponíveis na medição, existem as métricas compostas, que necessitam apenas do cadastrado na interface da Ostra, como mostrado na Figura 21. As fórmulas das métricas compostas são armazenadas junto às métricas na base de dados e são calculadas após a medição das métricas simples. As métricas compostas existentes estão listadas na Tabela 21, juntamente com as suas expressões de cálculo.

Na Figura 21, é exibida a tela de cadastramento de métrica composta. Para criar uma métrica composta deve ser indicado seu nome, acrônimo, descrição e a expressão com a qual ela deve ser calculada.

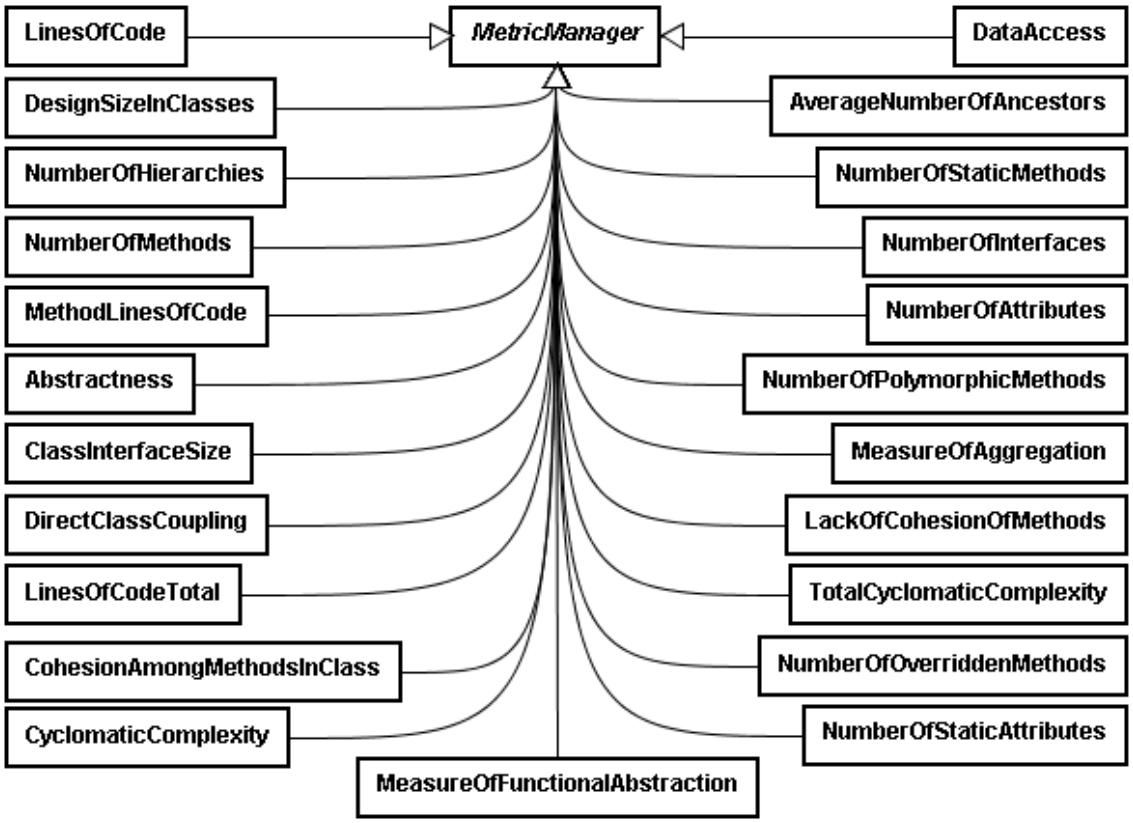


Figura 20: Diagrama de classes com as métricas simples.

Tabela 21: Métricas compostas presentes na Ostra.

Sigla	Nome	Alvo	Expressão
EFF	Effectiveness	Projeto	0,2 * ANA - 0,2 * DAM + 0,2 * MOA + 0,2 * MFA + 0,2 * NOP
EXT	Extendability	Projeto	0,5 * ANA - 0,5 * DCC + 0,5 * MFA + 0,5 * NOP
FLE	Flexibility	Projeto	0,25 * DAM - 0,25 * DCC + 0,5 * MOA + 0,5 * NOP
FUN	Functionality	Projeto	0,12 * CAM + 0,22 * NOP + 0,22 * CIS + 0,22 * DSC + 0,22 * NOH
MCD	Method Complexity Density	Projeto	TCC / NOM
REU	Reusability	Projeto	- 0,25 * DCC + 0,25 * CAM + 0,5 * CIS + 0,5 * DSC
UND	Understandability	Projeto	- 0,33 * ANA + 0,33 * DAM - 0,33 * DCC + 0,33 * CAM - 0,33 * NOP - 0,33 * NOM - 0,33 * DSC

As expressões das métricas compostas podem conter outras métricas e os seguintes operadores: adição, subtração, multiplicação, divisão, raiz quadrada e potência. Na expressão, quando são referenciadas outras métricas, seus acrônimos são utilizados. Por exemplo, para criar a métrica densidade de complexidade por métodos, a expressão deve ser: "*TCC / NOM*", como mostrado na Figura 21. As métricas compostas são calculadas pela classe

DerivedMetricService, que transforma suas expressões em objetos para depois as calcular (RIBEIRO, 2011).

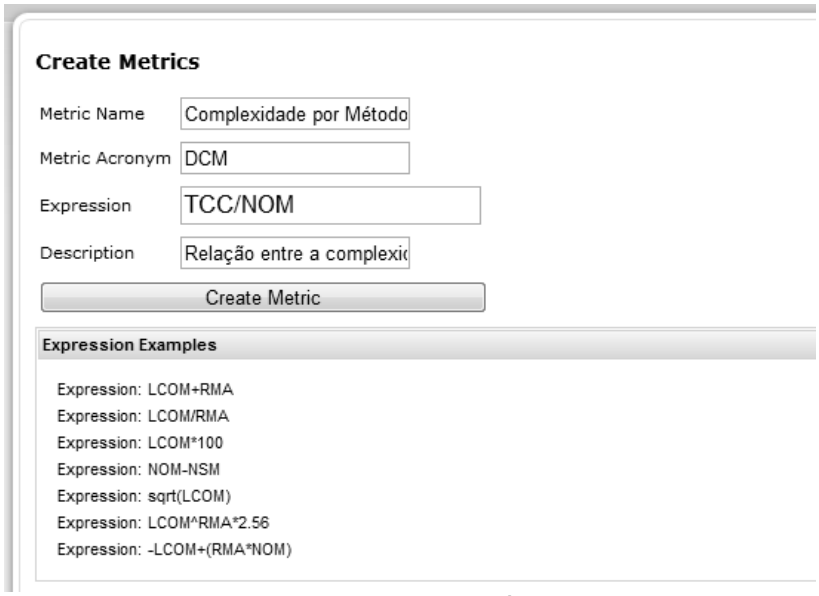


Figura 21: Tela de criação de métrica composta.

As informações sobre a medição que devem ser acessadas posteriormente são armazenadas em banco de dados. O diagrama de classes da Figura 17 também mostra as classes que mapeiam conceitos da medição e são armazenadas no banco de dados em tabelas homônimas. Na Figura 17, as classes que estão na cor branca pertencem à medição. A classe *VersionedItemMetricValue* armazena as medições de métricas de artefato, seja de classe ou pacote, identificando a métrica, o valor medido e a versão do artefato em questão. A classe *MetricValue* armazena as medições de projeto, identificando a métrica, o valor medido e a versão do projeto. A classe *Metric*, além de conter informações que identificam a métrica, como nome, acrônimo, descrição, alvo da medição, se é compilada ou não e a expressão que a define, se for uma métrica composta, possui a informação de qual *MetricManager* que automatiza a sua medição. Para ser utilizada a métrica, suas informações são armazenadas na classe *Metric* e é implementada a interface *MetricManager*. A classe *MeasurementLog* é utilizada para guardar informações sobre a medição. Com ela, é possível verificar, depois da medição, quando ocorrer um erro, qual sua causa em detalhes. Além disso, ela armazena o momento em que cada medição foi realizada, identificando o projeto e a versão. Para descobrir informações sobre a métrica, o atributo *log* dessa classe deve ser verificada.

Os valores delta, de variação das métricas, são armazenados em *MetricValue*. O que diferencia uma medição normal para uma medição de delta é o atributo *delta* desta classe: verdadeiro indica que é delta e falso que não é.

A orquestração das métricas e a inteligência do processo de medição estão nas classes de serviço: *OstraMetricValueService*, *OstraMetricService*, *OstraQualityAtributeService* e *MeasurementService*. O *OstraMetricService* é responsável pela medição das métricas simples. Essa classe verifica se existem métricas a serem extraídas e utiliza o *MeasurementService* para realizar a extração das métricas de uma revisão, que compila as revisões quando necessário. O *OstraQualityAttributesService* é responsável pelo cálculo das métricas compostas e é uma extensão do *OstraMetricService* para lidar com as métricas compostas.

O processo de medição é composto de várias atividades, como mostra o diagrama de atividades da Figura 22. A medição é iniciada com a extração das métricas simples e, em seguida, são inferidos valores para projeto com base nas medições de artefato. Isso é necessário, pois as métricas compostas são calculadas apenas na granularidade de projeto. Depois, são calculadas as métricas compostas e os valores de delta.

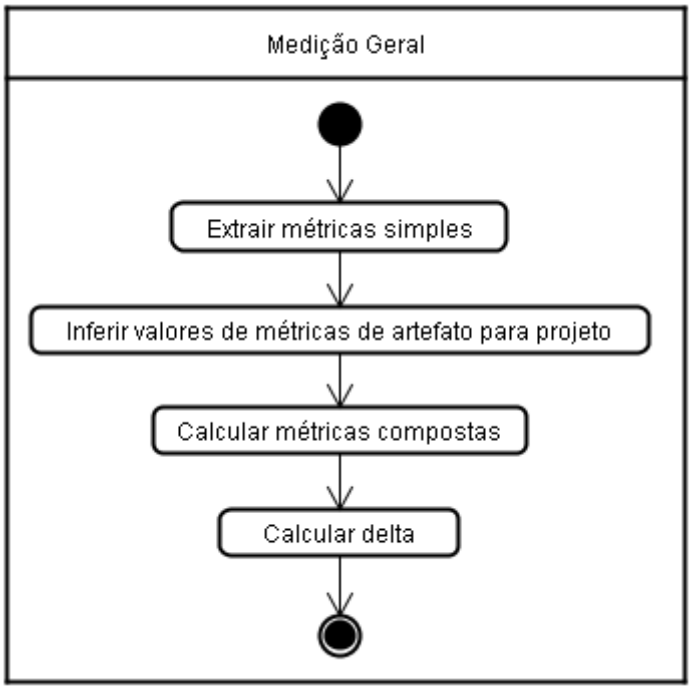


Figura 22: Diagrama de atividades mostrando as atividades da medição.

A primeira parte da medição, que extrai as métricas simples, é a mais complexa, pois acessa as versões do projeto compilando-as quando necessário. O diagrama de atividades da Figura 23, mostra em detalhes a medição das métricas simples. A classe *OstraMetricService* implementa esse processo e utiliza a classe *MeasurementService* como indicado na Figura 18.

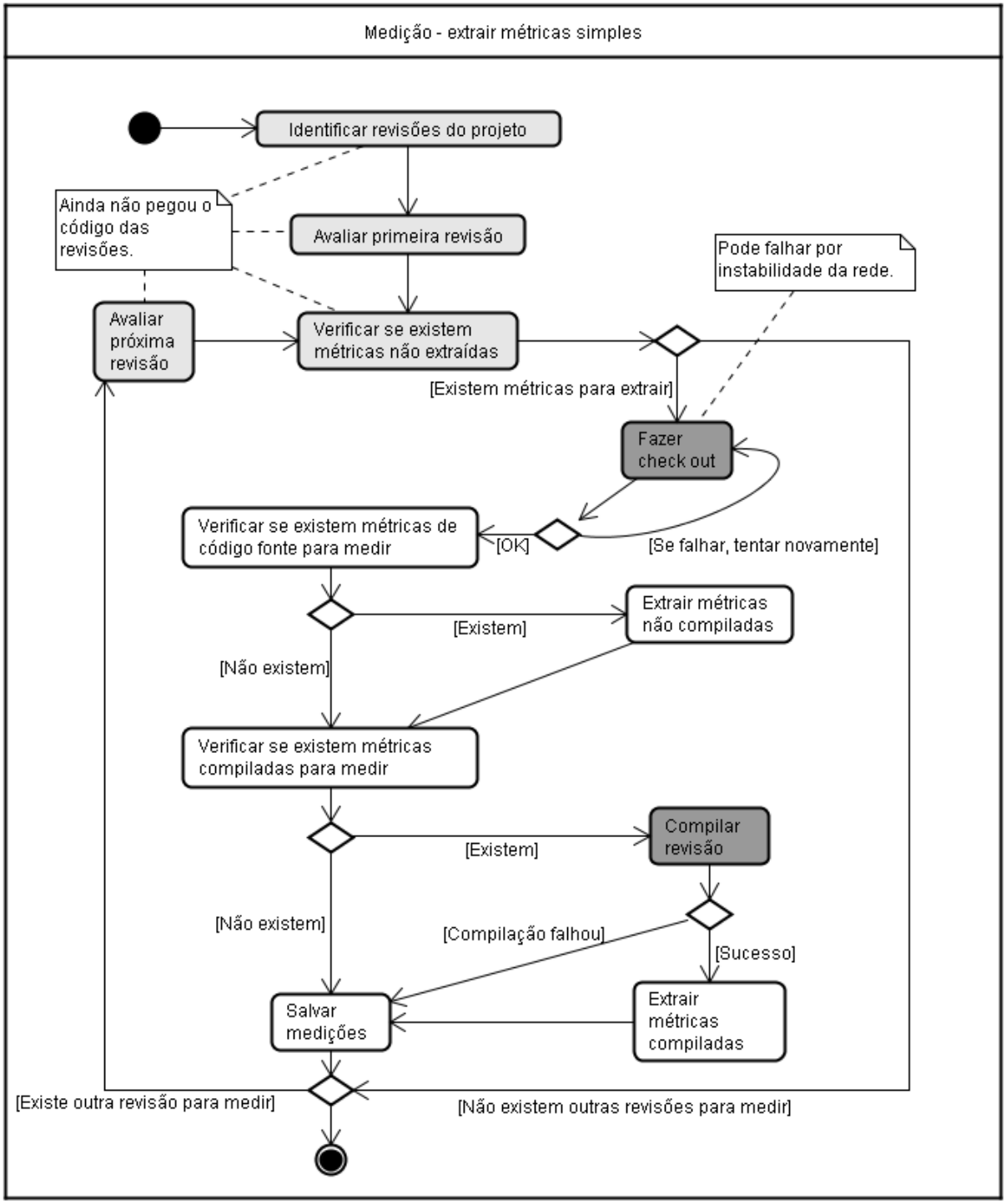


Figura 23: Diagrama de atividades da extração das métricas simples.

O processo da Figura 23 inicia ao identificar as revisões dos projetos utilizando um comando de log que retorna a lista de *commits* do projeto. Para cada *commit* são realizadas duas verificações: se há arquivos Java alterados e se há métricas não extraídas desse *commit*. Caso um *commit* não possua alterações em arquivos Java, ele é ignorado na medição. Por outro lado, se possuir arquivos Java que ainda não foram medidos, a medição da revisão

resultante desse *commit* é iniciada. Primeiramente, é feito o *check out* da revisão e, em seguida, são extraídas as métricas não compiladas e compiladas, nessa ordem. Após a medição das métricas não compiladas, se for necessário realizar medições de métricas compiladas, o projeto é compilado. Entretanto, a compilação pode falhar, principalmente, por erros de sintaxe ou falta de bibliotecas. Se a compilação falhar, o processo não é abortado, mas as métricas compiladas não são extraídas e a não compilação da revisão é indicada. A medição de cada revisão termina ao salvar as medições e o log da medição no *MeasurementLog*. O processo de medição de métricas simples continua com o próximo *commit.* Esse processo termina quando não existem mais *commits* para serem verificados.

4.3.3 MINERAÇÃO DE DADOS

A mineração de dados é responsável pela busca de regras e padrões, na base de dados de métricas, que descrevem a evolução do software. Para realizar a mineração de Regras de Associação, o algoritmo Apriori foi escolhido. A ferramenta WEKA (WITTEN; FRANK, 2005) foi utilizada, pois disponibiliza várias técnicas de mineração de dados através de sua biblioteca.

Para utilizar a WEKA, a base de dados tem que estar em um formato específico, denominado ARFF[10]. Dessa forma, a representação da base de dados que foi usada pela ferramenta WEKA é armazenada na base de dados como *String*, no campo *arff* da classe *DataMiningResult*. Esse e outros dados provenientes da mineração de dados são armazenados nos modelos persistentes do diagrama de classes da Figura 24.

A classe *DataMiningResult* armazena uma execução de mineração de dados e seu resultado, para que possa ser visualizado posteriormente. O resultado da mineração de regras de associação são regras, as quais são armazenadas em *DataMiningPattern*. Dessa forma, o resultado de uma mineração de regras de associação é formado por todos os *DataMiningPattern* que estão relacionados a *DataMiningResult* que se está observando. Os dados da mineração que são armazenados englobam tanto os parâmetros de suporte e confiança (ou outra medida de interesse para avaliar as regras de associação) utilizados na mineração quanto as medidas das regras encontradas e a base de dados utilizada na mineração.

[10] Existem outras formas de passar uma base de dados para a WEKA, e.g., arquivo CSS. Porém, foi escolhido o arquivo ARFF para facilitar os testes na ferramenta antes da implementação do protótipo ser finalizada e pela facilidade de sua construção.

Por outro lado, a classe *DataBaseSnapshot* não é persistida no banco de dados, mas é formada a partir do arquivo ARFF sempre que é carregado um resultado salvo previamente. Essa classe também é utilizada como intermediária entre o banco de dados da Ostra e a base de dados utilizada na mineração de dados. Ela é um espelho do arquivo ARFF. Essa classe é utilizada na preparação da mineração como um modelo intermediário entre a base de dados real com as medições e a base de dados no formato ARFF passado para a WEKA. Após a medição, a classe *DataBaseSnapshot* é utilizada em vez de uma String (representando o arquivo ARFF), para acessar mais facilmente as informações contidas na base de dados no momento da mineração.

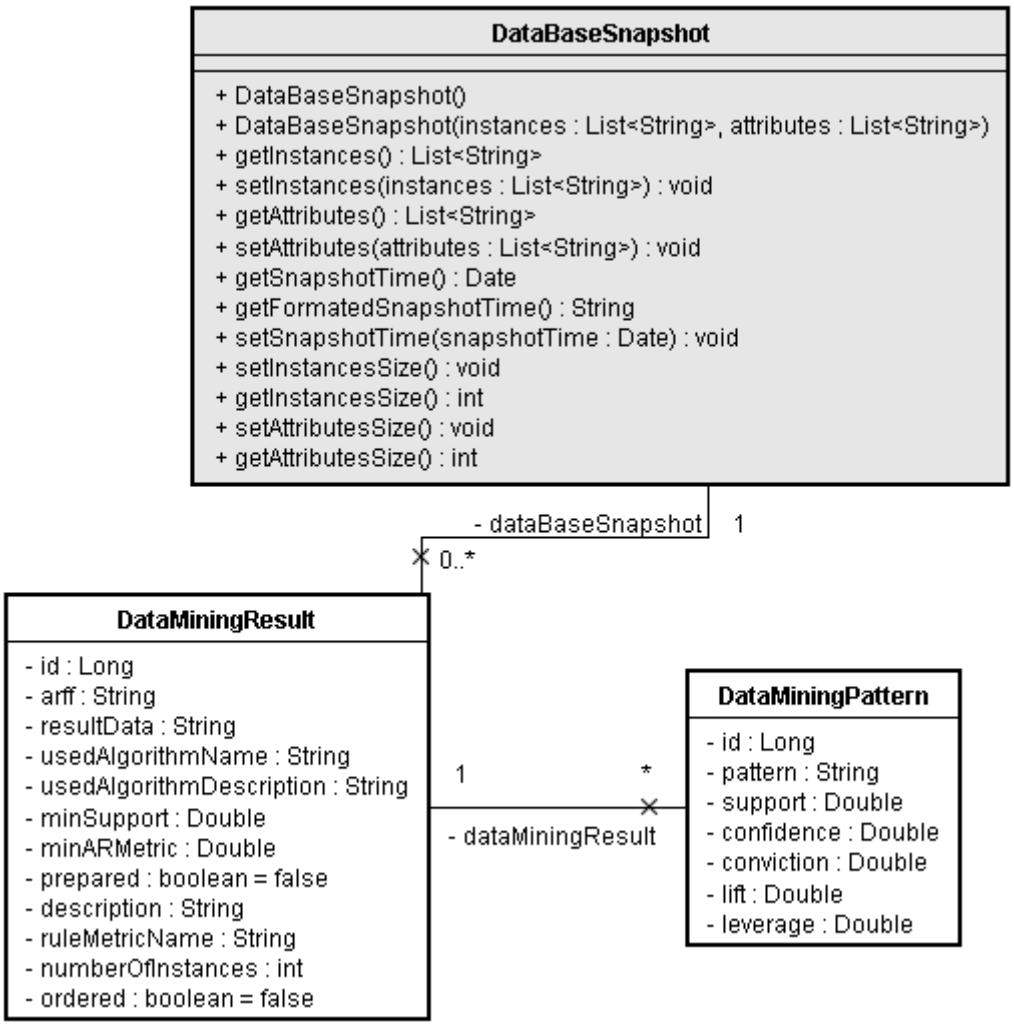


Figura 24: Diagrama de classes dos modelos de Mineração de Dados.

As técnicas de MD são mais complicadas de serem estendidas que as das duas outras disciplinas, pois os resultados que elas trazem são muito diferentes dependendo da técnica utilizada. Entretanto, se o objetivo for criar outra forma de minerar regras de associação, basta

implementar a interface *DataMiningTool*. Na atual versão da Ostra, a classe *AprioriTool* implementa essa interface e disponibiliza a mineração de regras de associação com o algoritmo Apriori (AGRAWAL; SRIKANT, 1994).

4.4 TESTES AUTOMATIZADOS

O protótipo Ostra possui mais de 400 *commits*. O módulo central Oceano Core, que agrega funcionalidades compartilhadas por outras pesquisas, recebeu colaboração de seis desenvolvedores. O módulo Oceano Core possui cerca de 260 classes, enquanto o Oceano Web possui cerca de 80 classes. Além disso, esses projetos possuem algoritmos complexos, como as implementações das métricas, a medição e a mineração de dados. As métricas que avaliam o projeto compilado dependem da gerência de configuração para a compilação. Consequentemente, se alguma dessas partes falharem ou não funcionarem corretamente, o funcionamento do protótipo fica comprometido.

Para garantir o funcionamento correto dos componentes do protótipo, foram desenvolvidos 93 testes automatizados. Os principais testes implementados são de métricas. Como entrada para esses testes, são utilizados três projetos, dos quais dois deles foram criados para os testes e um é um projeto de código aberto escolhido por ser multiprojeto, o *AnimalSniffer* (CODEHAUS, 2011). Um projeto multiprojeto é formado por vários módulos que podem ser construídos independentemente ou em conjunto.

4.5 INTERFACE E EXEMPLO DE UTILIZAÇÃO

O objetivo principal da Ostra é implementar a abordagem proposta e automatizar seu funcionamento, de forma a permitir a medição automática do histórico do projeto e mineração de dados. Nesta seção, é apresentada a utilização do protótipo, exemplificada com um projeto real, desde a medição à apresentação das regras mineradas.

A interface do protótipo é dividida em dois blocos: menu e conteúdo. No menu, estão dispostas as funcionalidades da ferramenta, enquanto o conteúdo é utilizado para a interação com o usuário e apresentação das informações requisitadas. Na implementação da interface, foram utilizadas as tecnologias: *Java Server Faces* (JSF), *RichFaces*, *PrimeFaces*, *Facelets* e *CSS*.

Para utilizar a Ostra, é necessário que o usuário faça *login* no sistema Oceano. A tela de *login* é apresentada na Figura 25. Na Figura 26, é apresentada a tela inicial do Oceano. Como pode ser observado, o Oceano é um sistema que centraliza várias ferramentas, desenvolvidas pelo Grupo de Evolução e Manutenção de Software (GEMS) da UFF. A Ostra é uma delas e foi implementada dentro desse ambiente maior chamado Oceano.

O Oceano fornece para as ferramentas algumas funcionalidades que são comuns a elas, como pode ser notado na Figura 26, no primeiro bloco do menu, com título Oceano. As funcionalidades comuns são referentes à criação, leitura, atualização e remoção de instâncias de usuário do Oceano, item de configuração, projeto e métricas. Essas funcionalidades são normalmente chamadas de CRUD, como acrônimo para *Create, Read, Update e Delete*.

Figura 25: Tela de *login* do Oceano.

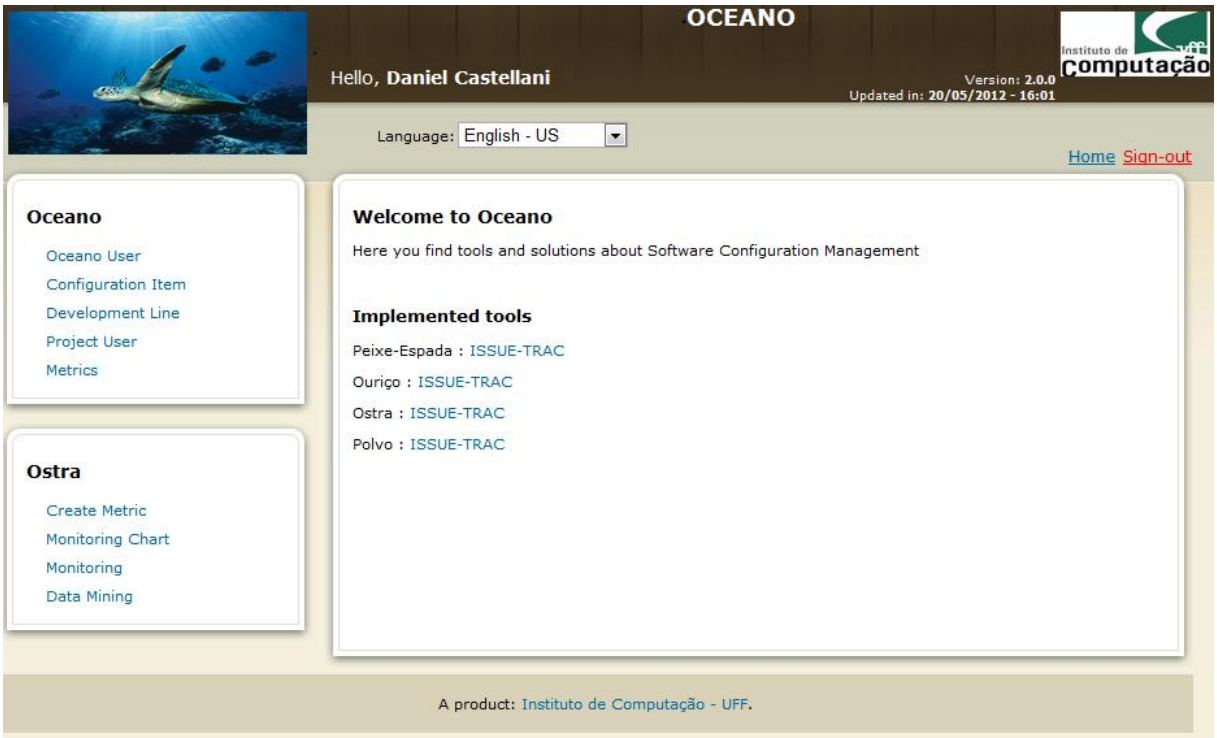


Figura 26: Tela inicial da Ostra.

O segundo bloco do menu é o da Ostra. Nele, podem ser acessadas as funcionalidades da Ostra: criação de métricas compostas (*Create Metric*), acesso aos gráficos de monitoramento (*Monitoring Chart*), acesso às medições (*Monitoring*) e mineração de dados (*Data Mining*).

As funcionalidades de CRUD das entidades e a maioria das funcionalidades do protótipo seguem a navegação ilustrada na Figura 27. Apesar de ser o padrão de navegação, algumas funcionalidades não o seguem. Um exemplo é referente às métricas compostas, que a criação é acessada através de um item no menu da Ostra, mas a exibição é na listagem de métricas do menu do Oceano.

Nas próximas seções é apresentada a utilização do protótipo, tendo como alvo de medição e análise o próprio projeto Oceano Core, mostrando como um usuário utiliza a Ostra para analisar este projeto. As próximas seções estão organizadas seguindo a utilização da Ostra como mostrado no diagrama de atividades da Figura 28.

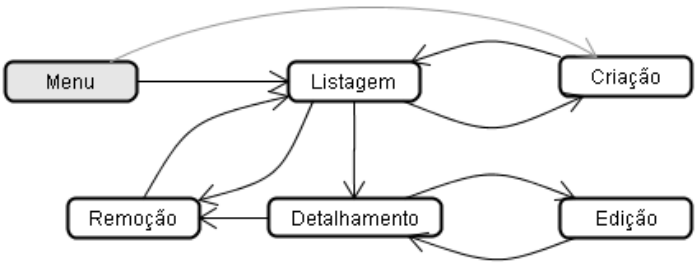


Figura 27: Padrão de navegação das telas do protótipo.

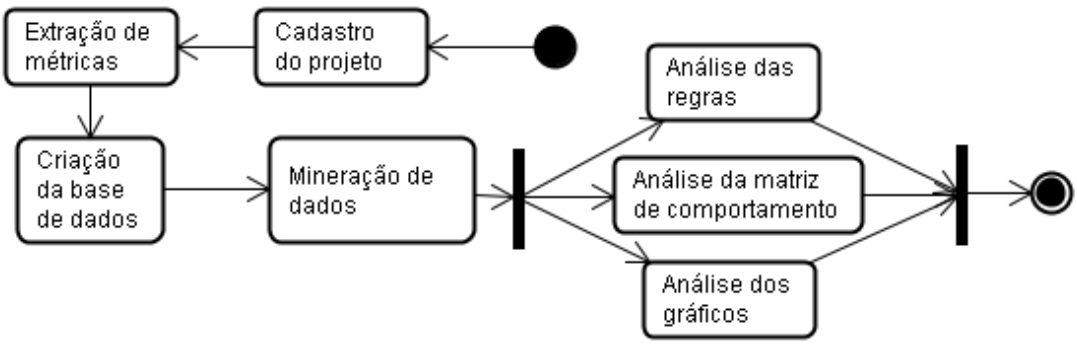


Figura 28: Fluxo básico de utilização da Ostra.

Inicialmente, é apresentado o cadastro do projeto e a medição na Seção 4.5.1. Em seguida, é apresentada a criação da base de dados na Seção 4.5.2 e a configuração da mineração de dados na Seção 4.5.3. As análises dos resultados através das regras de associação, da matriz de comportamento e dos gráficos são apresentadas, respectivamente,

nas Seções 4.5.4, 4.5.5 e 4.5.6. Apesar do sequenciamento das atividades ser apresentado nesta ordem, com a análise dos gráficos após a mineração de dados, é possível analisar o histórico do projeto através dos gráficos de controle e histogramas após a extração das métricas, sem a execução de mineração de dados.

A mineração de dados consiste em três passos: a criação da base de dados, a configuração das medidas de interesse e a extração e apresentação das regras de associação. Esses passos são apresentados nas Seções 4.5.2 e 4.5.3, respectivamente, denominadas configuração da base de dados e extração e apresentação das regras de associação.

4.5.1 MEDIÇÃO

Para realizar a medição de um projeto, algumas informações são necessárias, conforme explicado no Capítulo 3. Primeiro, deve ser criado o item de configuração com as informações que lhe dizem respeito. A criação dessa entidade é acessada pelo item "*Configuration Item*", no menu Oceano. Na Figura 29, é exibida a tela de criação do item de configuração, o qual deve ter um nome que o identifique, uma url base e url para o *trunk* e *branches*. Finalizando, deve ser indicado qual o repositório de código é utilizado, neste caso, o Subversion.

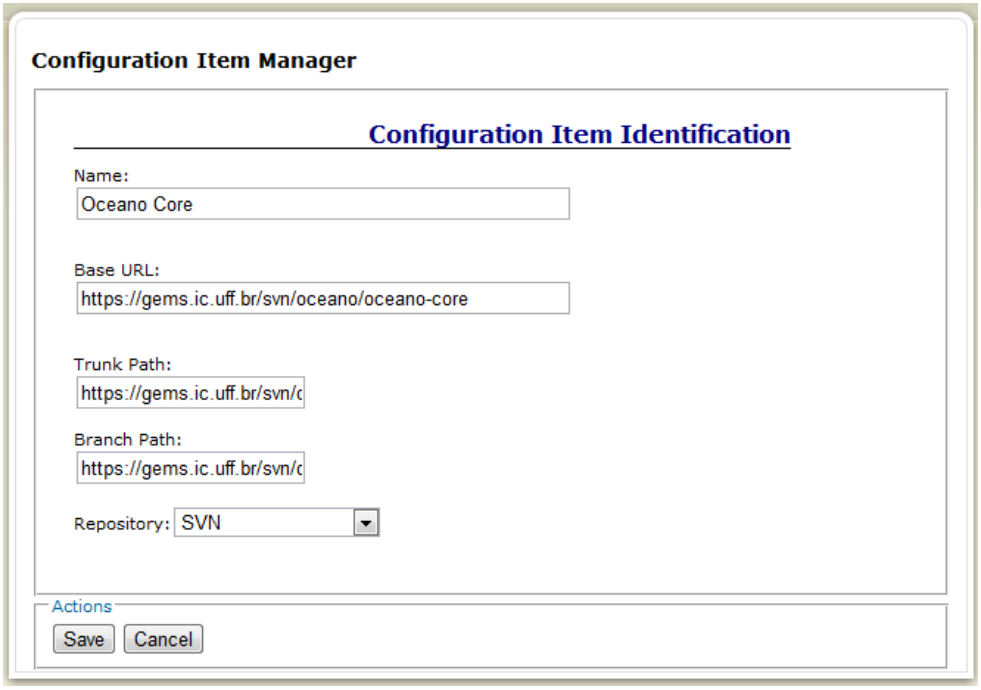


Figura 29: Tela de criação do Item de Configuração.

Após a criação do item de configuração, a linha de desenvolvimento pode ser criada. Para fazê-lo, o item "*Development Line*" deve ser acessado no menu Oceano. A Figura 30 mostra a tela de listagem de linhas de desenvolvimento, acessada através deste item do menu.

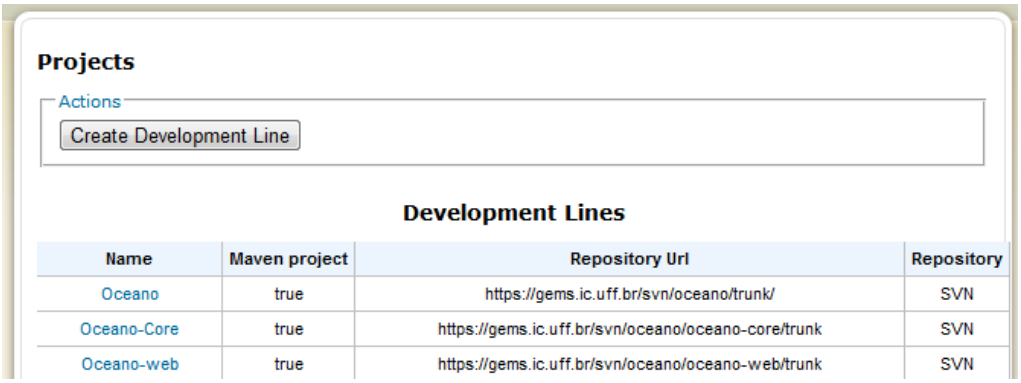


Figura 30: Tela de listagem de linhas de desenvolvimento cadastradas.

A tela de cadastro de projeto é acessada através do botão "*Create Project*", da tela de listagem. A Figura 31 mostra a tela de cadastro de projeto. Nela, deve ser selecionado o item de configuração, digitada a url do projeto que deve ser medido e indicado se ele é um projeto Maven ou não.

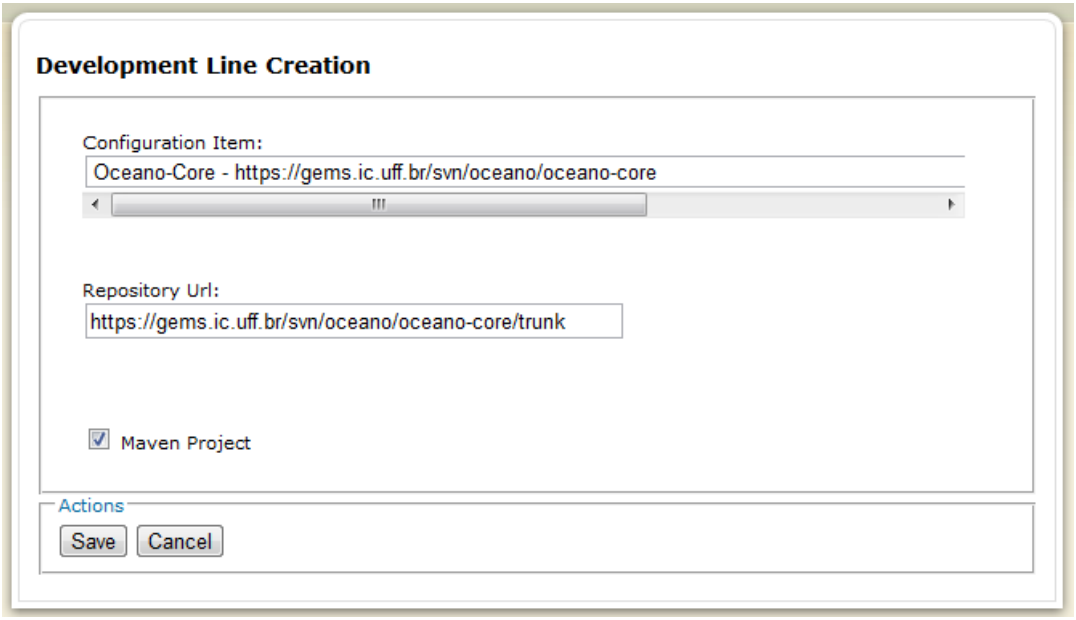


Figura 31: Tela de cadastro de linha de desenvolvimento.

Após a criação da linha de desenvolvimento, devem ser informados o *login* e senha do repositório que devem ser utilizados para acessar o histórico dela. Na Figura 32, é exibida a tela de listagem de acessos às linhas de desenvolvimento que um usuário possui. Para cada linha de desenvolvimento cadastrada no protótipo à qual o usuário possuir acesso, deve ser indicado o *login* e senha. Para cadastrar um acesso, o *link* "*Add*", deve ser utilizado, enquanto que, para alterar, o *link* "*Change*" deve ser utilizado. Para remover o acesso, "*Remove*" deve ser utilizado. Na Figura 33, é mostrada a tela de criação e edição de acesso ao repositório de código de uma linha de desenvolvimento.

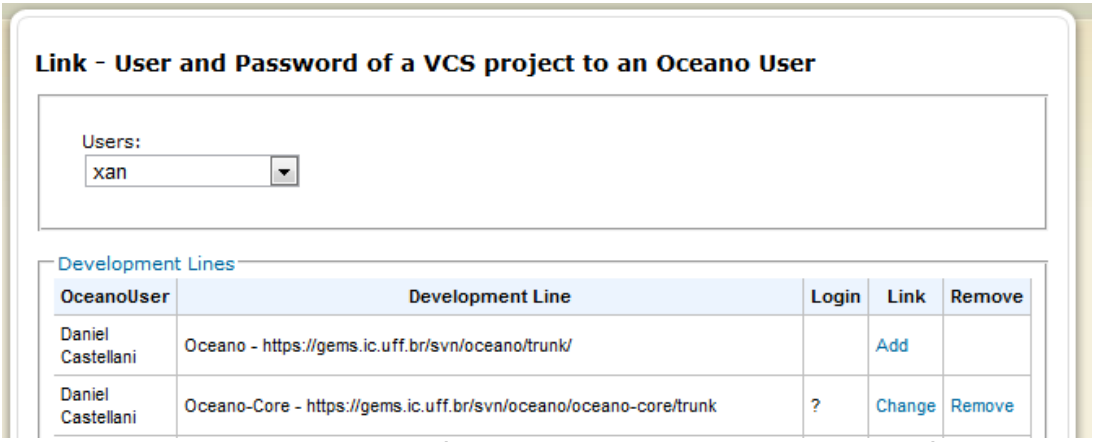


Figura 32: Tela de listagem de acesso às linhas de desenvolvimento de um usuário do oceano.

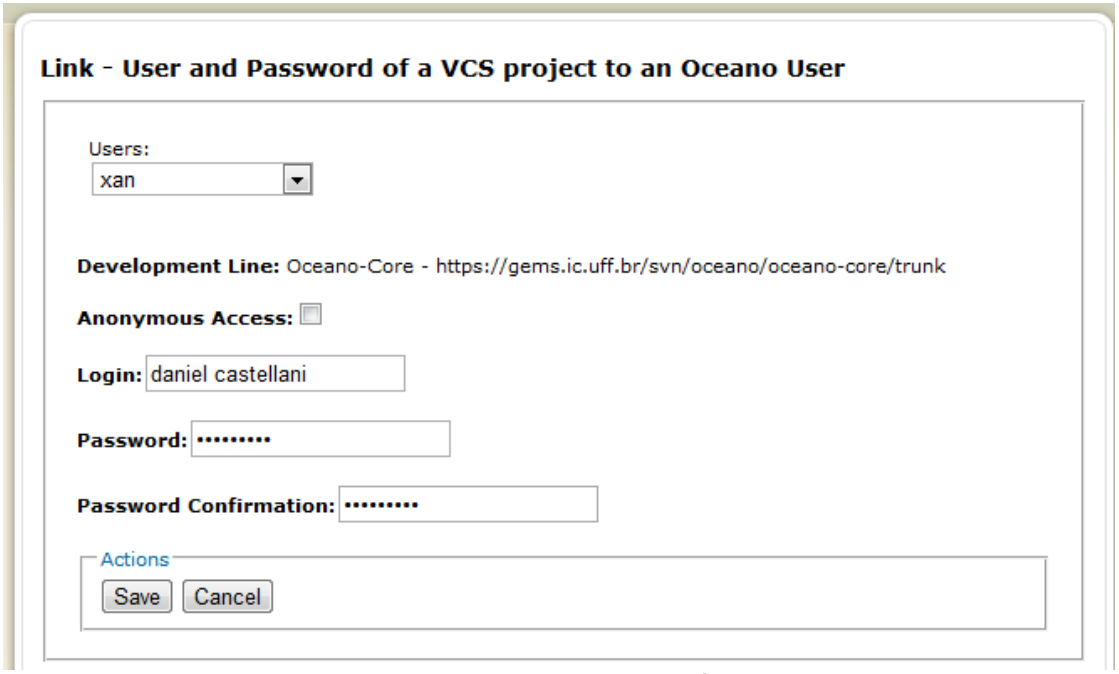


Figura 33: Tela de criação e edição de acesso ao repositório de linha de desenvolvimento.

Após o cadastro das informações de Gerência de Configuração pertinentes à linha de desenvolvimento e seu acesso, a medição pode ser executada. Não existe uma tela de medição, pois ela é executada por linha de comando como um processo *batch* do Oceano Web, que contém o código de medição da Ostra. Para a medição, a classe *OstraMeasurement* é utilizada. Essa classe possui uma lista de métricas e linhas de desenvolvimento que devem ser medidos e utiliza a classe *OstraMetricService*, mostrada na Figura 18. A execução da classe *OstraMeasurement* dispara o processo de medição explicado na seção 4.3.2, mostrado na Figura 22.

A classe *OstraMeasurement* busca por novas versões no repositório da linha de desenvolvimento e mede cada uma delas. Após a medição das linhas de desenvolvimento nela

indicadas, são calculadas as métricas de projeto a partir das métricas de artefato. Em sequencia, os valores dos deltas são calculados e armazenados. Consequentemente, para realizar o monitoramento constante de uma linha de desenvolvimento, basta indicar nesta classe de quanto em quanto tempo ela deve executar a medição com a variável *WAIT_UNTIL_NEXT_MEASUREMENT*.

A implementação da medição foi realizada desta forma, pois ela demora muito e teria que ter um controle complexo se implementada na interface. Por exemplo, o projeto IDUFF utilizado nos experimentos e mostrado em mais detalhes no Capítulo 5, precisou de aproximadamente 5 dias para ter cerca de 1.500 revisões medidas com as 29 métricas cadastradas.

4.5.2 CRIAÇÃO DA BASE DE DADOS PARA A MINERAÇÃO

Após a fase de medição, cada versão do projeto está medida e seus dados estão prontos para serem utilizados na Mineração de Dados para extrair informações sobre o seu histórico. Antes de minerar as regras de associação, deve-se formar uma base de dados com os projetos e métricas desejados. A criação da base de dados é acessada através do botão "*Mine Database*", da tela de listagem de resultados de Mineração de Dados, mostrada na Figura 34.

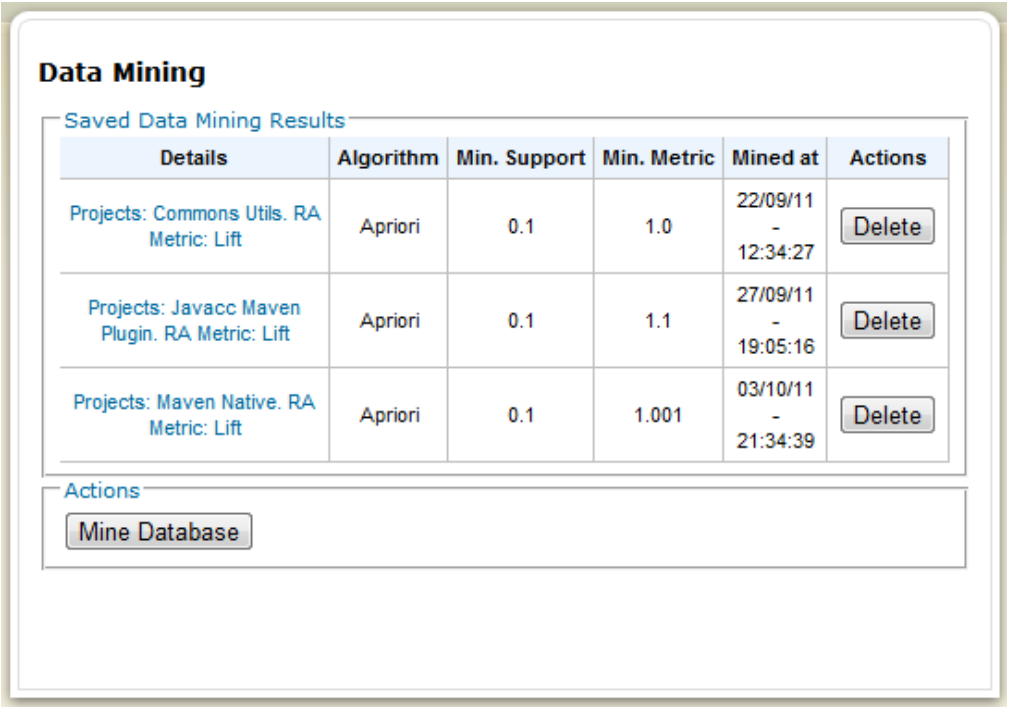


Figura 34: Tela de listagem de resultados da Mineração de Dados.

O processo de mineração de dados inicia na criação da base de dados, realizada na tela mostrada na Figura 35. Essa tela possui três painéis: "*Project Selection*", "*Attribute Selection*"

e "*Discretizer Selection*". Na primeira área, são selecionados os projetos que formarão a base de dados. Na segunda, as métricas que serão utilizadas e, na terceira, a forma que serão discretizados os dados para a mineração.

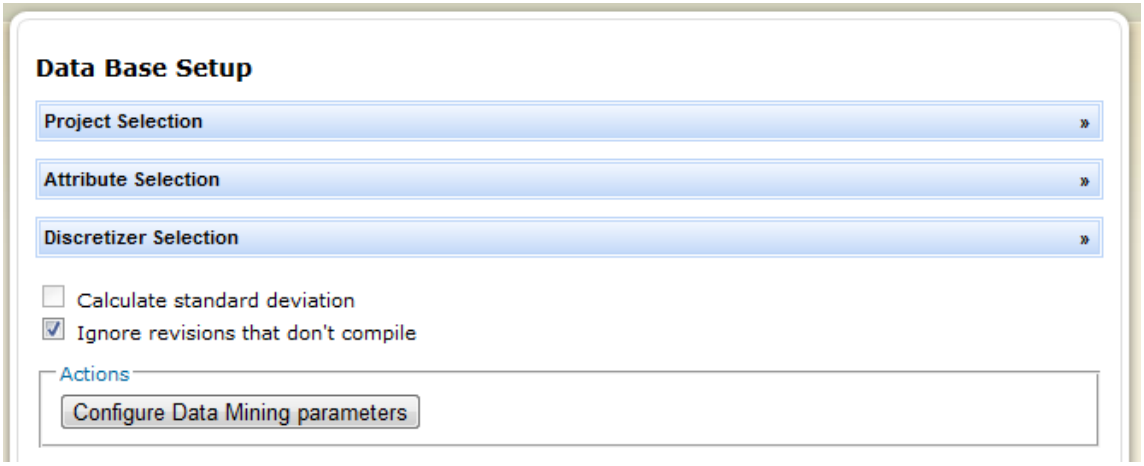


Figura 35: Tela de criação da base de dados.

Além dos painéis de seleção de projetos, métricas e discretizadores, na tela de criação da base de dados, existem outras duas opções: "*Calculate standard deviation*" e "*Ignore revisions that don't compile*". A primeira indica se o desvio padrão dos deltas de cada métrica para cada *commit* deve ser um atributo da base de dados. Se essa opção for selecionada, existirá um atributo a mais para cada métrica, com o valor do desvio padrão dessa métrica em cada *commit*. A outra opção indica se devem ser considerados ou não na base dados os *commits* que resultam em revisões que não compilam. Essa opção deve estar desmarcada quando são utilizadas, na mineração de dados, métricas extraídas de projetos compilados, pois as revisões que não compilam não possuem valores para essas métricas.

A Figura 36 mostra o painel de seleção de projetos. Esse painel exibe algumas informações sobre os projetos medidos: nome, quantidade de revisões total e quantidade de revisões que não compilam. O objetivo dessas informações é auxiliar na execução de experimentos, pois indica a quantidade de *commits* do projeto. Nele, está selecionado o projeto Oceano Core, que será minerado neste exemplo.

Após selecionar o projeto desejado, as métricas que descreverão esse projeto na base de dados devem ser selecionadas. Para isso, o painel de atributos exibido na Figura 37 deve ser utilizado. Nesse painel, são exibidos os atributos obrigatórios de cada instância da base de dados e as métricas, juntamente com a indicação de quais são simples ou derivadas (i.e., métricas compostas).

O terceiro painel, também da Figura 35, é para a seleção dos discretizadores. Apesar de já existir esse painel, eles não podem ser modificados. Atualmente, os discretizadores implementados e utilizados, obrigatoriamente, são os descritos no Capítulo 3: (1) positivo, zero ou negativo para indicar a variação das métricas e (2) dias da semana, (3) hora e (4) turno de trabalho para a data do *commit*. O objetivo deste painel é agrupar os discretizadores que venham a existir na ferramenta, tornando possível ao usuário escolher como discretizar os dados. Por exemplo, futuramente, pode ser implementado outro discretizador para a variação das métricas (delta), descrevendo-a como: zero, positiva baixa, negativa baixa, positiva média, negativa média, positiva alta e negativa alta. Dessa forma, o usuário poderia escolher entre duas formas diferentes de discretizar a variação das métricas nesse painel.

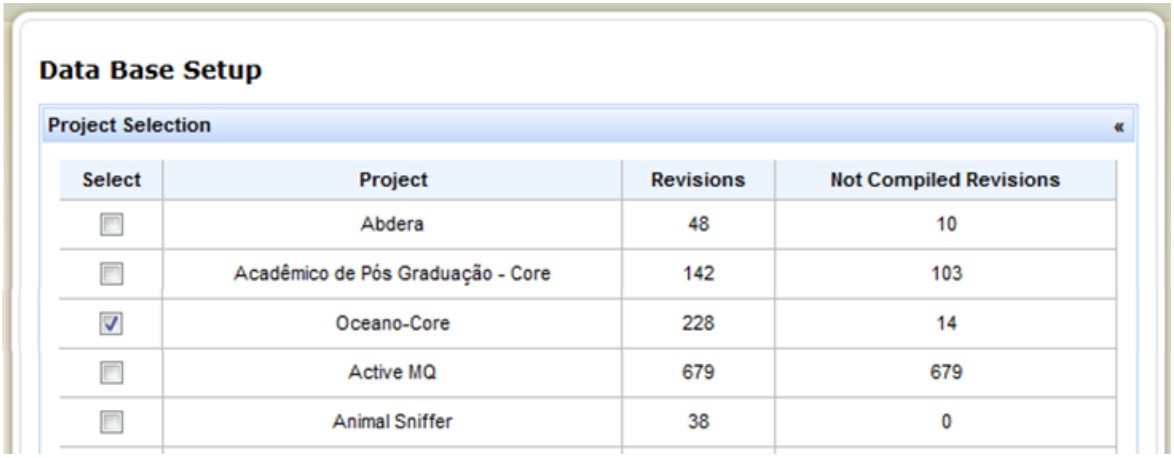


Figura 36: Painel de seleção de projetos da tela de criação de base de dados.

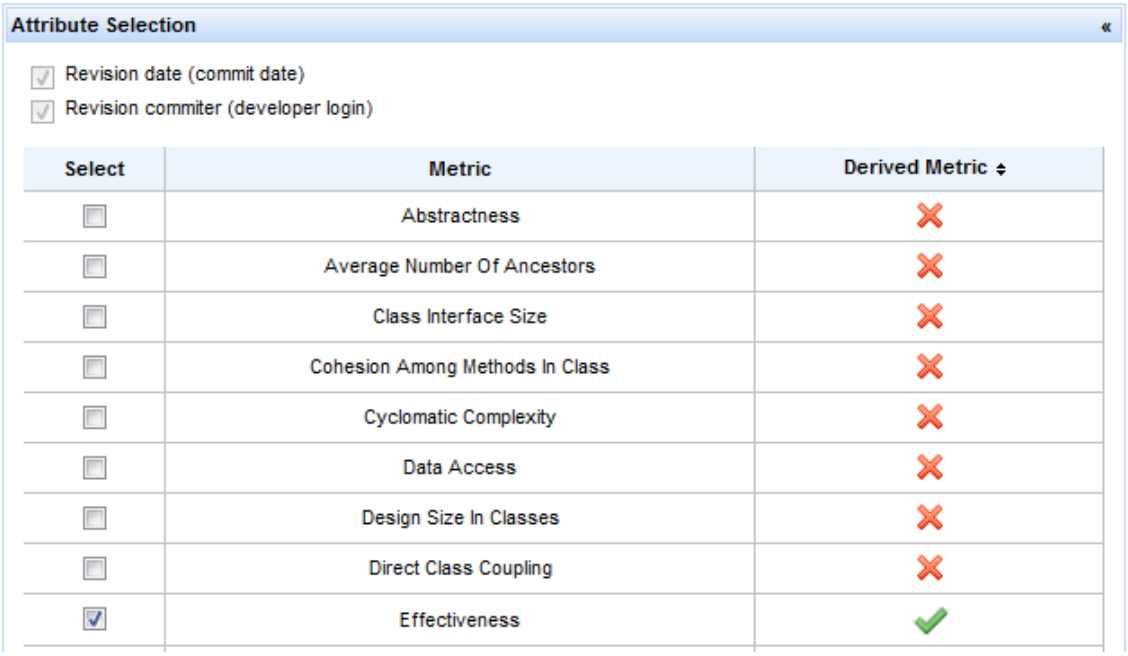


Figura 37: Painel de seleção de atributos da tela de criação de base de dados.

4.5.3 CONFIGURAÇÃO DA MINERAÇÃO DE DADOS

Após a criação da base de dados, a mineração de dados deve ser configurada. Devem ser definidas medidas de interesse para a avaliação das Regras de Associação. A Figura 38 mostra a tela de configuração da mineração de dados. Essa tela é dividida em duas partes: detalhes da base de dados e configuração da mineração de dados.

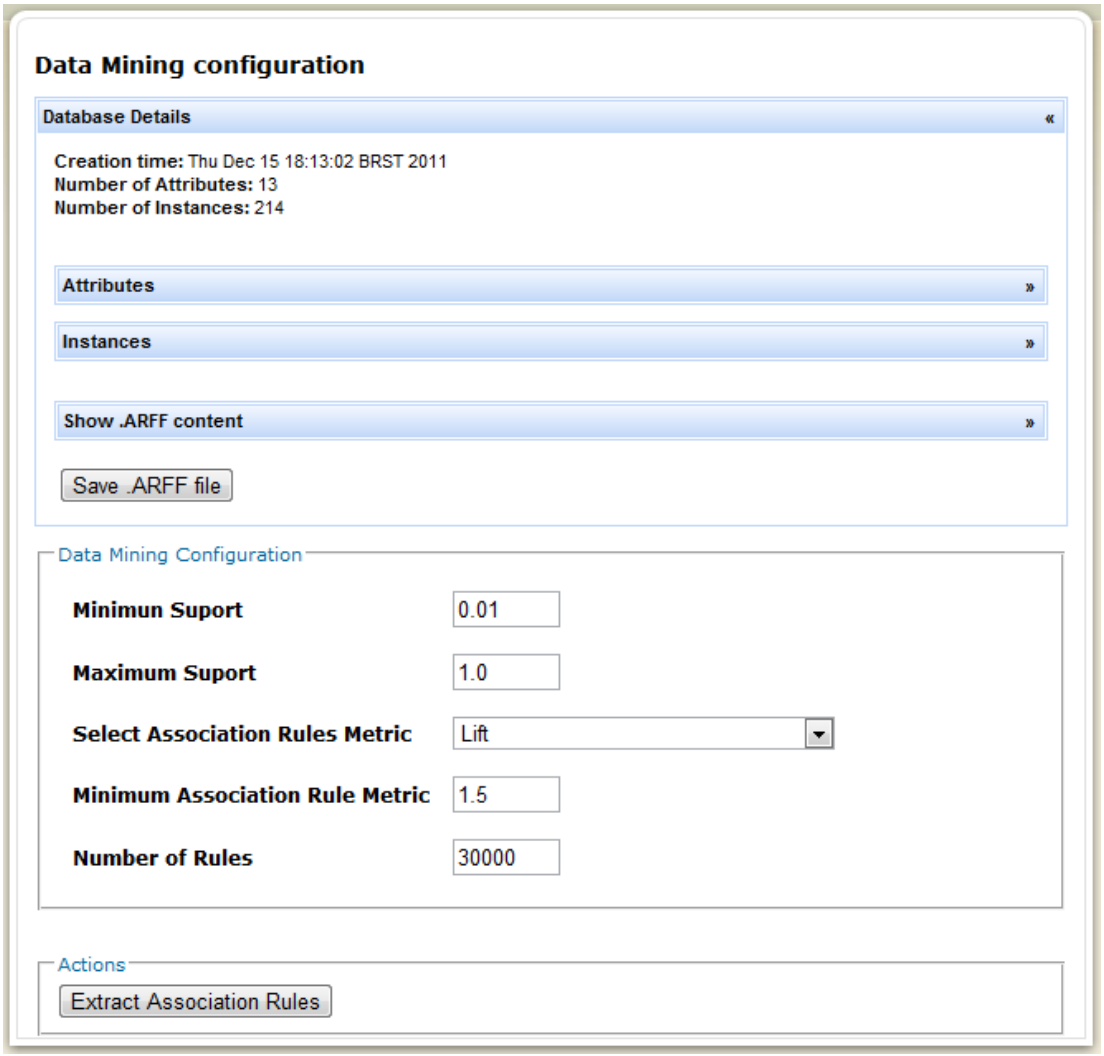


Figura 38: Tela de configuração da Mineração de Dados.

No painel "*Database Details*", são exibidas informações sobre a base de dados, como os atributos que foram selecionados e as instâncias. As Figuras 39-41 mostram os painéis com essas informações, respectivamente, atributos, instâncias e arquivo ARFF.

Na Figura 40, são exibidas as instâncias da base de dados em cada linha. Em cada uma delas, os atributos que serão utilizados na mineração de dados estão separados por barras verticais. O painel "*Instances"* exibe a mesma informação do painel "*Show .ARFF content"*, mas com formatação diferente.

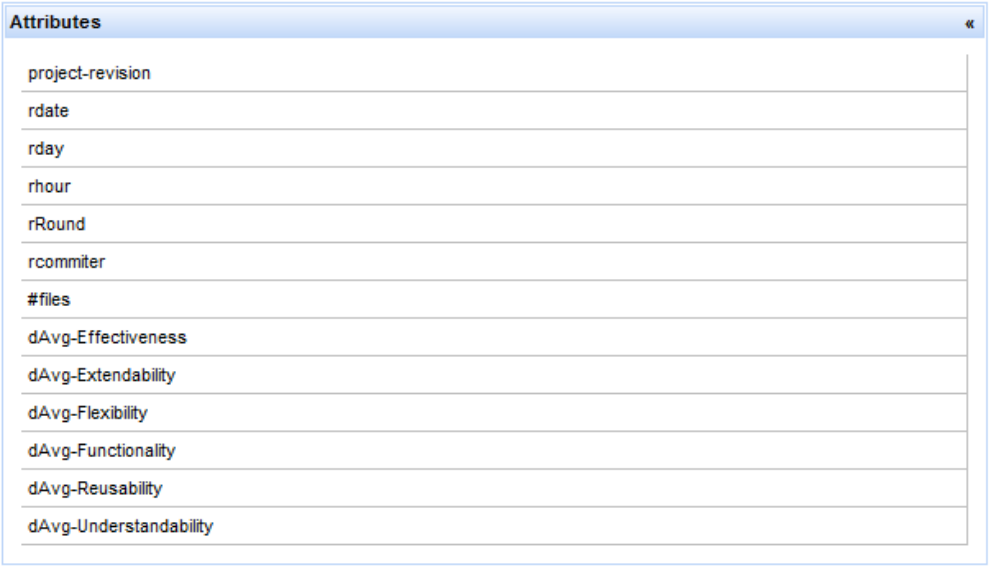


Figura 39: Atributos da base de dados.

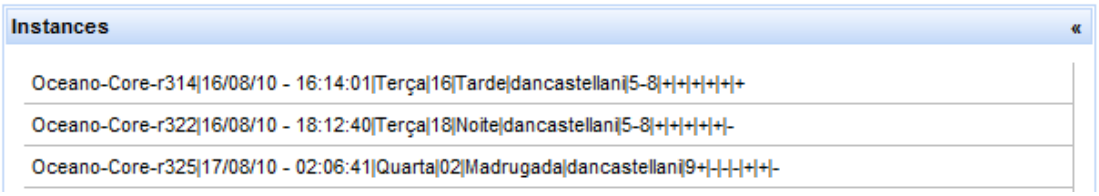


Figura 40: Instâncias da base de dados.

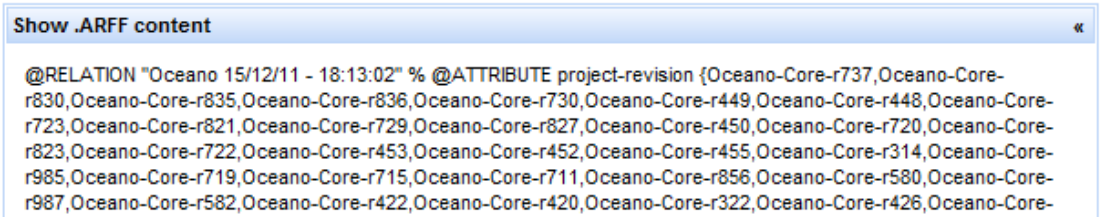


Figura 41: Exibição do arquivo ARFF da base de dados.

O suporte mínimo indica em quantas instâncias um padrão precisa ocorrer na base de dados para ser considerado interessante. Além dele, deve ser escolhida uma medida de interesse para avaliar as regras de associação. Assim, deve ser escolhida a medida e definido o valor mínimo para ela, como mostrado na Figura 38. Nessa tela, também devem ser definidas quantas regras devem ser exibidas, dentre aquelas com as maiores medidas de interesse indicadas.

Para seguir com a extração das regras de associação, o botão "*Extract Association Rules*", da tela mostrada na Figura 38, deve ser utilizado. Ao clicar nesse botão, a mineração de dados que foi configurada é executada. A tela de exibição dos resultados da mineração de

dados é mostrada na Figura 42. Ela possui cinco painéis principais: "*Data Mining Information*", "*Arff Parameter*", "*Data Mining Output*", "*Mined Rules*" e "*Behavior Table*".

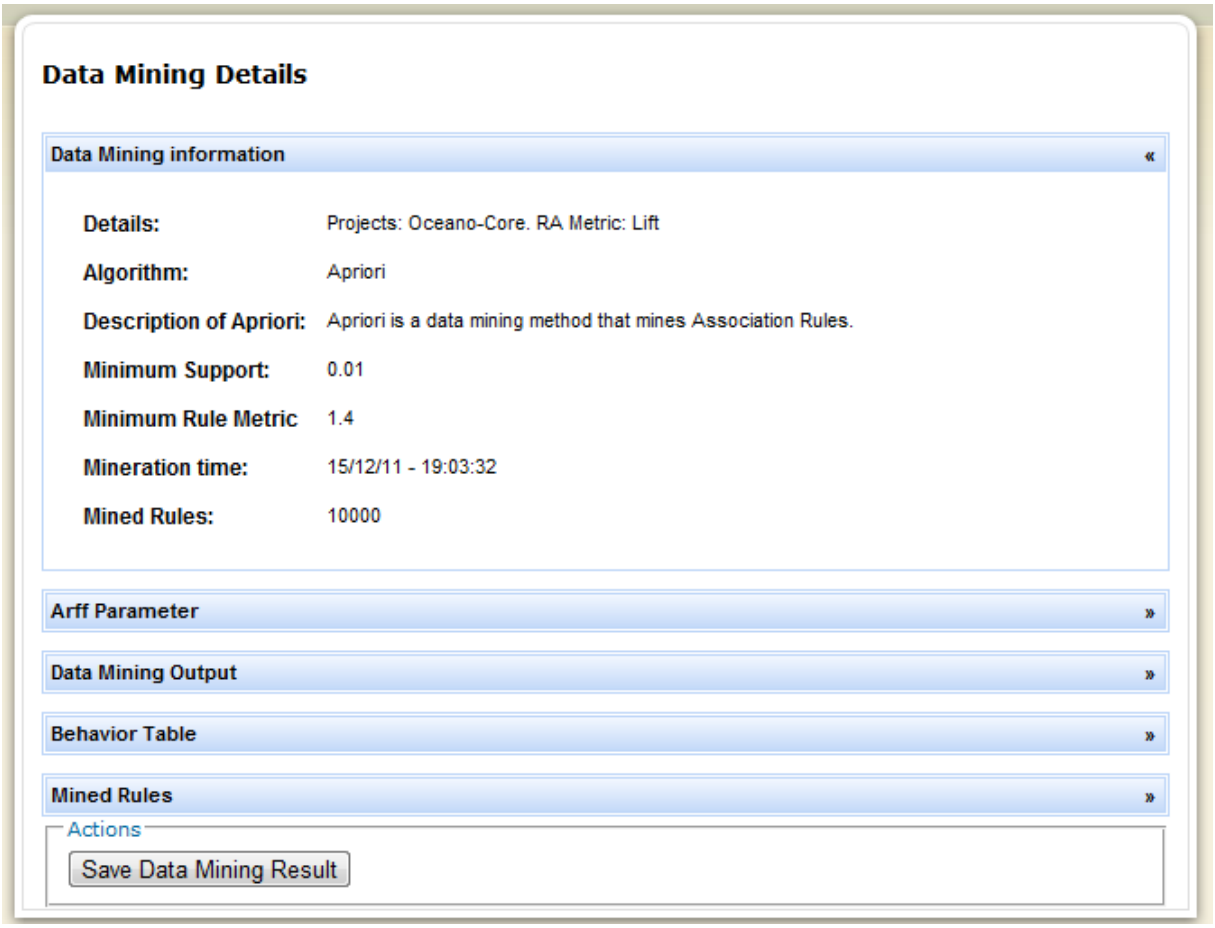


Figura 42: Tela de exibição de Mineração de Dados.

O painel "*Arff Parameter*" mostra o conteúdo do arquivo ARFF e permite que seja feito seu download para utilizar na ferramenta WEKA. A Figura 43 mostra esse painel.

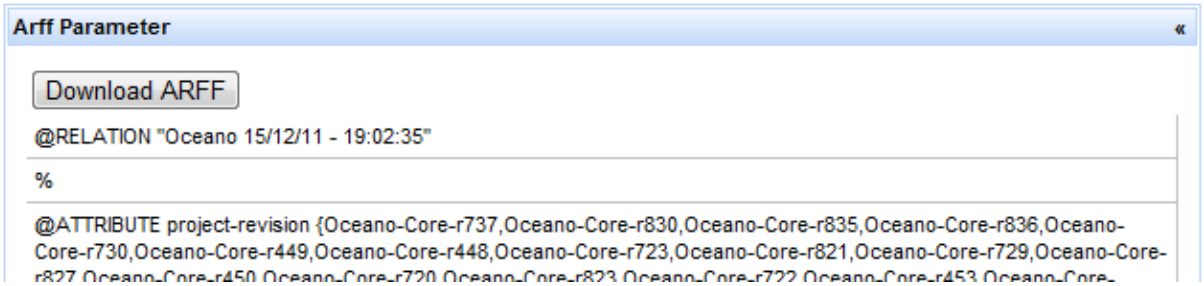


Figura 43: Painel *Arff Parameter* da tela de detalhamento de Mineração de Dados.

O Painel "*Data Mining Output*" mostra a saída da MD, da mesma forma utilizada pela ferramenta WEKA para apresentar os resultados. Ele é mostrado na Figura 44.

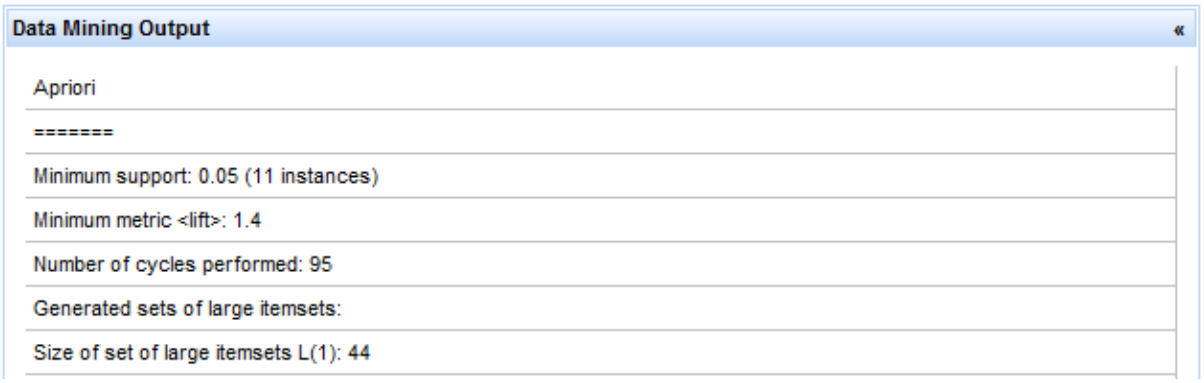


Figura 44: Painel Data Mining Output da tela de detalhamento da Mineração de Dados.

4.5.4 ANÁLISE DOS RESULTADOS VIA REGRAS DE ASSOCIAÇÃO

No painel "*Mined Rules*", exibido na Figura 45, são mostradas as regras extraídas na mineração de dados. Nessa tela são exibidas as medidas das regras interessantes considerando os limiares previamente estabelecidos na tela de configuração da mineração, exibida na Figura 38: Tela de configuração da Mineração de Dados..

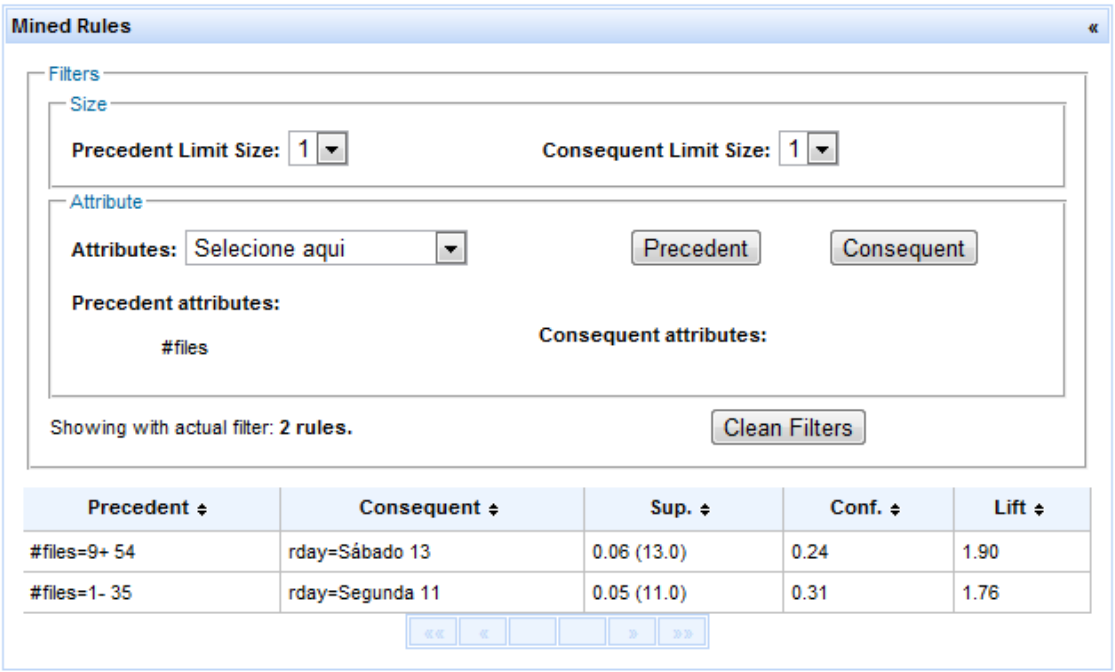


Figura 45: Painel de exibição das regras mineradas.

Na tela de exibição das métricas, é possível definir filtros que facilitam a análise das regras. Por exemplo, se o desejado é saber padrões derivados da quantidade de artefatos modificados em um *commit*, deve ser selecionado o atributo "*#Files*", no precedente. Com este filtro, foram encontradas duas regras, como mostra a Figura 45. Uma das regras indica que "quando são alterados nove ou mais artefatos, o *commit* é realizado no sábado" e a outra

indica que "quando é alterado um arquivo, o *commit* é realizado na segunda-feira". As regras também possuem valores de suporte, confiança e *lift* indicados na tabela desse painel.

Outro tipo de filtro que pode ser utilizado é o de tamanho do precedente ou consequente. Esse filtro é particularmente interessante quando o objetivo é identificar qual atributo utilizado na mineração influencia outro atributo, filtrando regras menores e de mais fácil compreensão. Por exemplo, se o objetivo é saber se algum dia da semana que afeta a extensibilidade do código, deve ser utilizado o filtro de tamanho um no consequente e no precedente, escolhendo *rday* no precedente e *Extendability* no consequente, como mostra a Figura 46. Com este filtro foi possível encontrar uma regra que indica que "na terça feira a extensibilidade aumenta", em 30% dos casos e com *lift* de 2,5.

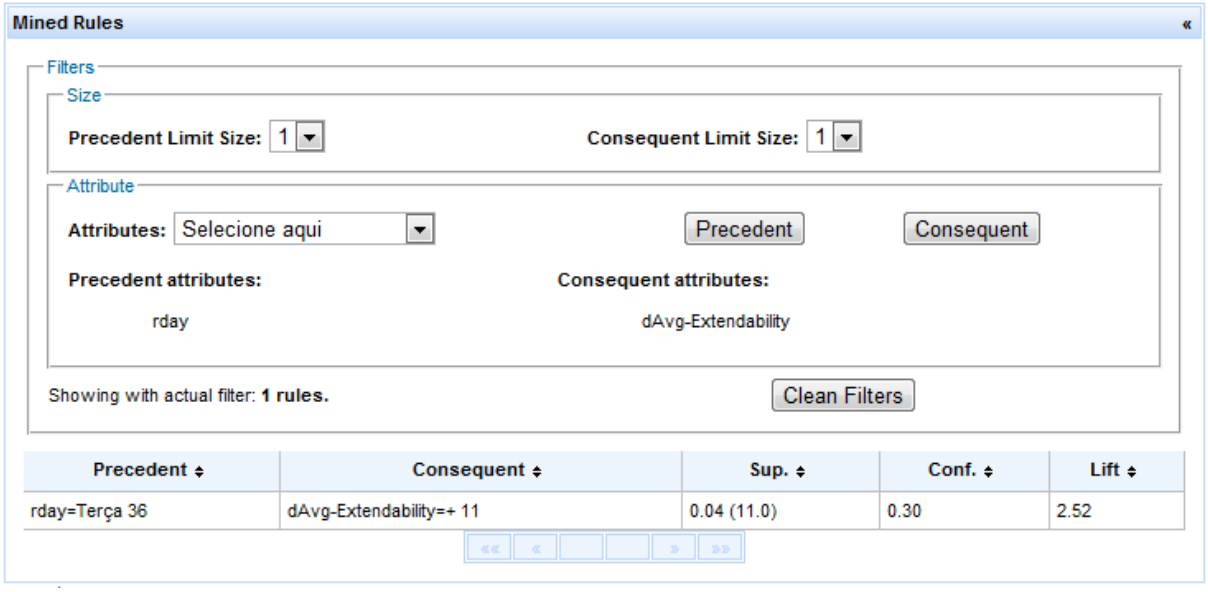


Figura 46: Exemplo de utilização dos filtros.

4.5.5 ANÁLISE DOS RESULTADOS VIA TABELA DE COMPORTAMENTO

Na tela de detalhamento do resultado da Mineração de Dados pode ser visualizada a matriz de comportamento no painel "*Behavior Table*". Esse painel é mostrado na Figura 47. A tabela de comportamento mostra como cada métrica varia em função das outras. Os comportamentos descritos nela são extraídos das RAs com essas métricas no precedente e consequente. Nessa figura, é possível ver as regras encontradas para a linha *Entendimento* (última linha) e coluna *Funcionalidade* (antepenúltima coluna), que possui a cor cinza. O quadro com as regras aparece ao colocar o *mouse* em cima da célula sobre a qual se deseja saber mais informações. Essa funcionalidade é particularmente interessante quando a cor do comportamento resultante é cinza, ou seja, indefinido, pois, nesse caso, as regras devem ser

analisadas para identificar quais comportamentos apareceram e se algum deles possui diferença significativa das medidas de interesse de mineração de dados.

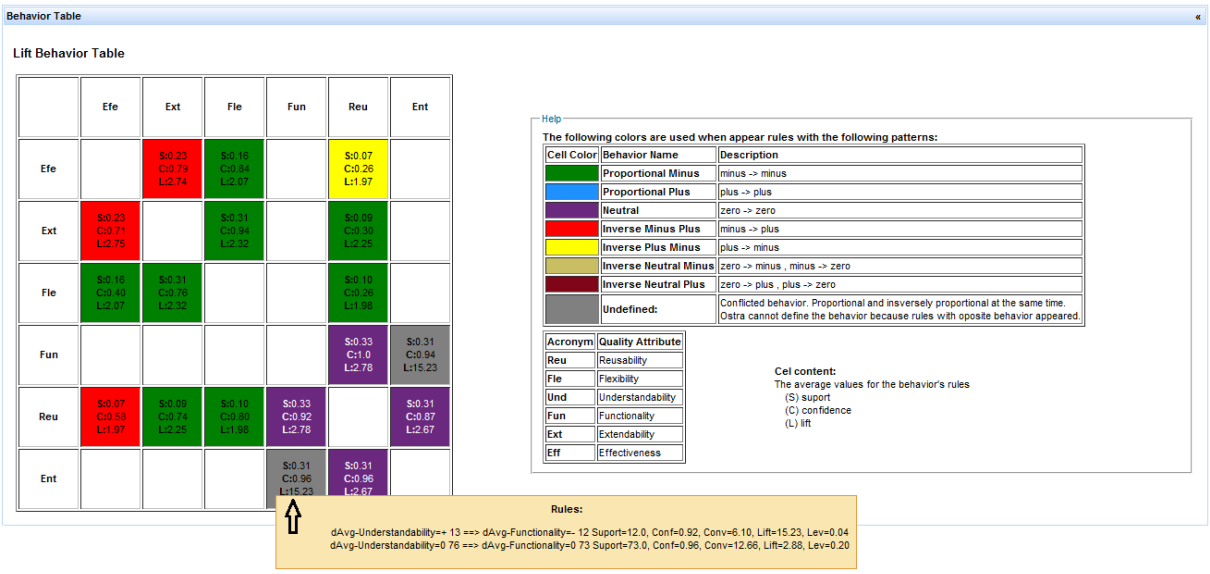


Figura 47: Painel Behavior Table da tela de detalhamento da Mineração de Dados.

4.5.6 ANÁLISE DOS RESULTADOS VIA GRÁFICOS

Os gráficos de controle e histogramas podem auxiliar a interpretar as regras mineradas e a monitorar a qualidade do projeto. A tela de monitoramento dos projetos é mostrada na Figura 48.

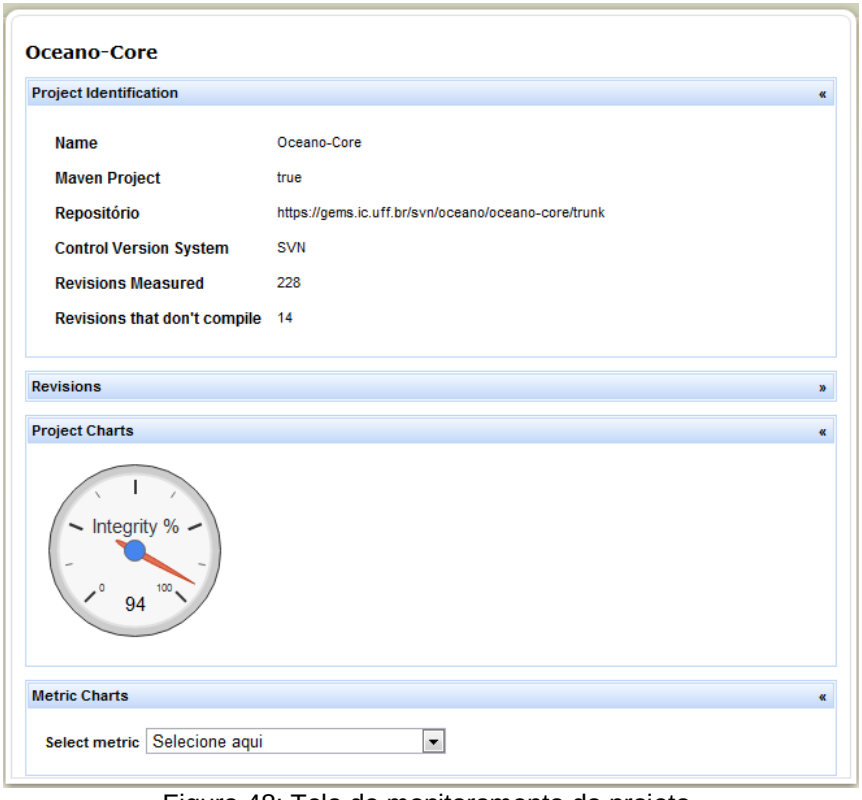


Figura 48: Tela de monitoramento do projeto.

Nessa tela, pode ser observada a integridade do projeto, que indica o percentual de versões analisadas que compilam de todas as revisões analisadas do projeto. Além da integridade do projeto, que é mostrada no formato de um conta-giros, outros gráficos podem ser acessados no painel "*Metric Charts*", presente na parte inferior da Figura 48. Quando selecionada a métrica que se deseja visualizar via gráficos, eles são exibidos. Na Figura 49, é exibido o gráfico de controle da métrica DSC para o projeto Oceano Core.

Na Figura 50, é exibido o histograma para essa métrica e projeto. No histograma, é possível definir a quantidade de intervalos que devem ser utilizados para caracterizar os valores e o limite de cada um deles.

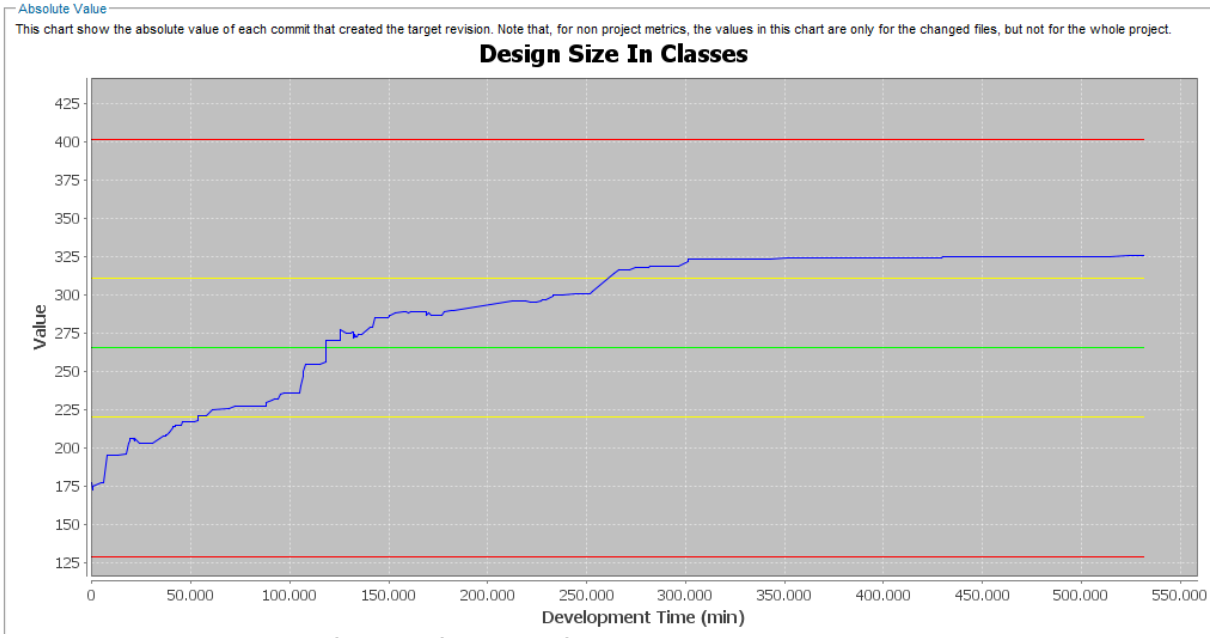


Figura 49: Gráfico histórico da métrica DSC para o projeto Oceano Core.

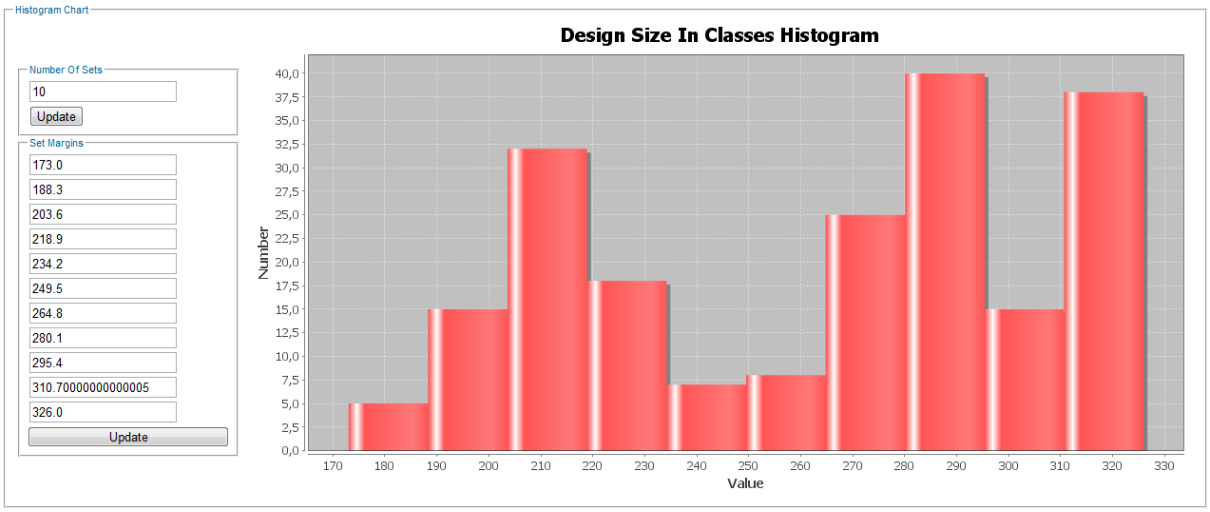


Figura 50: Histograma da métrica DSC para o projeto Oceano Core.

Parte da implementação dos gráficos foi realizada por Wallace (RIBEIRO, 2011) em seu projeto de conclusão de curso. Detalhes sobre a implementação dos gráficos que utilizou a biblioteca JFreeChart (GILBERT; MORGNER, 2011) podem ser encontrados nesse trabalho.

4.6 CONSIDERAÇÕES FINAIS

Neste capítulo, foi apresentado o protótipo que implementa a abordagem Ostra. Foram apresentadas as funcionalidades do protótipo, sua arquitetura, padrões de projeto utilizados e tecnologias. Também foi mostrado um exemplo da utilização do protótipo com o projeto Oceano Core, módulo de infraestrutura do protótipo Ostra. Foram apresentadas as funcionalidades e telas do protótipo indicando como ele materializa a abordagem proposta.

O protótipo implementado mapeia todas as fases da abordagem Ostra. A gerência das informações, a mineração de dados e a análise de regras e gráficos são realizadas pela interface gráfica. O único processo que deve ser iniciado de forma assíncrona, por linha de comando, é a execução da classe *OstraMeasurement* para a medição dos projetos.

Este capítulo teve como intuito evidenciar uma das contribuições desta pesquisa, que foi a construção de um ambiente que possibilite: (1) a medição automática do histórico de versões de um projeto, (2) mineração de dados, (3) filtragem e visualização das regras mineradas e (4) a geração de gráficos. Além disso, foi desenvolvida uma infraestrutura que disponibiliza ferramentas de gerência de configuração, medição e mineração de dados e pode ser utilizada por outros projetos do grupo de pesquisa.

Com o protótipo implementado, é possível analisar o histórico de projetos Java cujas versões são controladas pelo Subversion. No próximo capítulo, são apresentados experimentos realizados com o protótipo desenvolvido que evidenciam a capacidade da Ostra de obter informações relevantes sobre a evolução de um projeto.

CAPÍTULO 5 – AVALIAÇÃO EXPERIMENTAL DA OSTRA

5.1 INTRODUÇÃO

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

Acredita-se que, com as regras mineradas na última fase da abordagem Ostra, seja possível tomar decisões baseadas em informações sobre a evolução do software e reagir a problemas que podem acontecer durante o desenvolvimento. Com a Ostra, é possível tomar decisões com informações relevantes sobre os desenvolvedores, o momento em que ocorrem as modificações ou como elas afetam a qualidade do software. Consequentemente, as informações providas pela Ostra aumentam o conhecimento do gerente sobre a evolução do seu projeto.

Os gráficos do histórico mostram como o projeto evoluiu em relação a alguma métrica e auxiliam no monitoramento da qualidade do projeto. Acompanhando esses gráficos, o responsável pela qualidade do projeto pode monitorá-la e identificar os eventos que a afetam com mais informações relevantes do que se utilizando apenas a sua experiência. Dessa forma, com os gráficos do histórico, deve ser possível identificar alterações fora do padrão histórico e monitorar a qualidade do projeto, auxiliando a reagir rapidamente às variações não desejadas.

Finalmente, com as tabelas de comportamento, pode-se comparar o comportamento das métricas no projeto em desenvolvimento com padrões já conhecidos, indicados pela literatura ou pela empresa. Essa é uma forma de se identificar quais métricas têm influência sobre as outras e como é essa influência. Espera-se que, com essa tabela, seja possível visualizar as informações mais relevantes dos padrões comportamentais das métricas, auxiliando o pesquisador em busca de padrões gerais ou o gerente do projeto através da verificação se os comportamentos indicados pela literatura ou pela empresa são seguidos pelo seu projeto.

Neste capítulo, essas ferramentas são avaliadas com experimentos de acordo com os seus objetivos. Na Seção 5.2, são extraídas e analisadas regras de associação de quatro projetos, objetivando avaliar se as regras trazem informações relevantes sobre a evolução do projeto. Na Seção 5.3, o monitoramento da qualidade de um projeto é avaliado através da análise do histórico do IdUFF. Nesse experimento é avaliada a capacidade da tabela de

comportamento de verificar se o projeto segue os padrões indicados na literatura. Na Seção 5.4, é avaliada a utilidade das tabelas de comportamento na análise de um projeto e na busca de padrões gerais à engenharia de software. Na Seção 5.5, são apresentadas as ameaças à validade dos experimentos. Finalmente, na Seção 5.6, são realizadas considerações finais sobre os experimentos apresentados.

5.2 REGRAS PARTICULARES DE UM PROJETO

Para avaliar a utilidade da abordagem proposta em explicitar informações sobre a evolução do projeto que podem auxiliar o processo de tomada de decisões, são apresentadas dez regras positivas e dez regras negativas extraídas do histórico de quatro projetos: dois proprietários e dois de código aberto. Foram medidos com a abordagem proposta cerca de 150 projetos, dos quais apenas 16 possuem mais de 100 revisões que compilam. Neste experimento, foram utilizados os dois projetos que possuem maior quantidade de revisões que compilam dos projetos de código aberto e dos proprietários. Informações sobre os outros projetos medidos estão no Apêndice B.

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.

Os projetos *Maven Javadoc Plugin* e *Maven GWT Plugin* são de código aberto, enquanto o IdUFF e o Publico-core são proprietários. Na Tabela 22, são apresentadas algumas informações dos projetos apresentados neste estudo: quantidade de desenvolvedores, número de artefatos, quantidade de revisões que compilam, quantidade de revisões que não compilam e o total de modificações do projeto armazenadas pelo Sistema de Controle de Versão. As medições foram realizadas em janeiro de 2012 considerando as versões disponíveis até aquele momento. A última coluna da Tabela 22 indica qual foi versão mais recente considerada de cada projeto no experimento. Essa informação permite a reexecução do experimento a partir desse ponto.

Neste experimento, para analisar o histórico dos projetos com regras de associação, foram utilizadas cinco métricas que pudessem ser entendidas mais facilmente, mesmo para 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.

Tabela 22: Projetos utilizados no experimento para analisar as regras mineradas.

Projeto	Quantidade de Desenvolvedores	Quant. de Artefatos	Revisões que compilam	Revisões que não compilam	Total de revisões	Ultima revisão medida
Maven Javadoc Plugin	21	795	248	81	329	1232525
Maven GWT Plugin	6	463	252	60	312	14772
IdUFF	31	1068	1355	154	1509	22695
Publico-core	21	117	127	7	134	21633

Na mineração de dados, foram utilizadas duas bases de dados para cada projeto. A necessidade de se utilizar duas bases vem do cálculo do delta, pois ele compara sempre uma versão com a anterior: para as métricas compiladas e não compiladas, a revisão anterior pode ser diferente, já que as revisões que não compilam não são consideradas. Dessa forma, foram utilizadas duas bases para cada projeto analisado neste experimento: A e B, onde a base A continha todos os *commits* do projeto e a base B continha apenas com os *commits* que resultam em revisões que compilam.

As métricas que compõem essas duas bases são diferentes, mas possuem cinco atributos em comum: o desenvolvedor, a quantidade de artefatos alterados, o dia da semana que foi realizado o *commit*, o turno de trabalho e um identificador único para cada *commit*. Além desses dados, a base A possui um atributo indicando se a versão resultante do *commit* compila ou não e as métricas que não dependem de compilação. Por outro lado, a base B possui as informações do *commit* e as cinco métricas selecionadas para o experimento. Na apresentação das regras encontradas, é identificado de qual base foi extraída a regra. Nas duas bases de dados, foram utilizadas as mesmas métricas não compiladas: Linhas de Código e Complexidade Ciclomática de McCabe.

Para visualizar melhor a aplicabilidade da abordagem, algumas regras encontradas são apresentadas em detalhes. Apesar de serem mostradas as dez regras positivas e dez negativas com maior *lift* de cada projeto, para as mais interessantes serão apresentadas análises mais aprofundadas, justificando suas ocorrências e apresentando hipóteses para elas.

Para manter a confidencialidade em relação aos desenvolvedores, optou-se por não os identificar. Entretanto, acredita-se que utilizar apenas símbolos para os desenvolvedores (X, Y ou Z) tornaria as regras menos legíveis. Consequentemente, os identificadores dos

desenvolvedores foram substituídos aleatoriamente por nomes de personagens de "As Crônicas de Gelo e Fogo" (MARTIN, 1996).

5.2.1 PROJETO: IDUFF

O projeto IdUFF é o sistema acadêmico da Universidade Federal Fluminense e possui interface web. Esse projeto possui quatro anos de desenvolvimento e a maior quantidade de *commits* dentre os projetos analisados. Até o momento da medição, ele possuía 1509 *commits* com alteração em arquivos Java, dos quais 1355 compilavam.

Na Tabela 23, são apresentadas as regras positivas encontradas com a mineração de dados do projeto IdUFF utilizando as duas bases de dados. A quarta regra, por exemplo, indica que quando a quantidade de classes aumenta, a reusabilidade também aumenta, com 77% de confiança, ou seja, em 77% das vezes que o precedente acontece, o consequente também acontece. A quinta regra indica que, quando o desenvolvedor Tywin faz alterações, a chance de serem alterados três ou quatro artefatos é o dobro do esperado, como mostra o *lift* de 2. Já o suporte mostra em quantas instâncias (em valores percentuais) a regra ocorre na base de dados. Por exemplo, a primeira regra possui um suporte de 14%, ou seja, ocorre em cerca de 210 das instâncias da base de dados, ou seja, dos *commits*.

Nas tabelas que apresentam as regras encontradas, são marcadas as regras explicadas em detalhes. A marcação utilizada é um * (asterisco) ao lado do número que identifica a regra. Na Tabela 23, as regras quatro, oito e nove possuem * e serão apresentadas em detalhes.

A quarta regra mostra que quando o projeto cresce em número de classes, a reusabilidade também aumenta. Isso pode demonstrar uma preocupação da equipe com a reusabilidade do projeto. Para os criadores do modelo QMOOD, o aumento da reusabilidade ao longo da evolução do projeto é esperado (BANSIYA; DAVIS, 2002). Com o aumento da quantidade de classes, mais recursos desse projeto podem ser reutilizados, se forem disponibilizados métodos públicos. O commit #4828, embasa essa regra mostrando que foi criada uma classe chamada *IntegracaoRails*, com dois métodos públicos e um privado, como mostrado na Figura 51. Esse *commit* aumentou a reusabilidade afetando as métricas que a definem, pois aumentou as métricas DSC e NOM, enquanto CAM e DCC se mantiveram estáveis.

Para mostrar as alterações realizadas nos *commits*, é utilizada uma notação nos diagramas de classe e nos trechos de código para indicar adição, remoção e alteração. Nos trechos de código, as linhas adicionadas estão com o fundo na cor verde, enquanto as linhas

removidas estão com o fundo na cor vermelha. Para os diagramas de classe, os métodos ou atributos adicionados estão na cor azul escuro e sublinhados, os removidos, estão na cor cinza, e os modificados, estão sublinhados na cor verde. O nome das classes criadas está em azul, sem sublinhado, enquanto o das classes alteradas está em preto e o das classes removidas em cinza. Apesar de a UML já possuir um significado para o texto sublinhado, método ou atributo estático, este não deve ser considerado, prevalecendo o significado da notação indicada. Consequentemente, não é possível identificar nos diagramas de classe métodos ou atributos estáticos. Esta notação é utilizada para os diagramas e trechos de código mostrados neste capítulo.

Tabela 23: Regras positivas do projeto IdUFF.

#	Regra	Sup.	Conf.	Lift	Base
1	Se a quantidade de classes aumenta, Então nove ou mais artefatos alterados.	0,14	0,72	3,11	B
2	Se a quantidade de linhas de código diminui, Então o entendimento aumenta.	0,03	0,20	2,86	B
3	Se a reusabilidade diminui, Então o entendimento aumenta.	0,02	0,17	2,43	B
4 *	Se a quantidade de classes aumenta, Então a reusabilidade aumenta.	0,15	0,77	2,23	B
5	Se Tywin faz alterações, Então três ou quatro artefatos alterados.	0,01	0,28	2,00	B
6	Se Stannis faz alterações, Então nove ou mais artefatos alterados.	0,01	0,39	1,76	A
7	Se Robb faz alterações, Então complexidade diminui.	0,01	0,43	1,43	A
8 *	Se Stannis faz alterações, Então a complexidade diminui.	0,01	0,36	1,20	A
9 *	Se Tywin faz alterações, Então compila.	0,03	1,0	1,11	A
10	Se Renly faz alterações, Então compila.	0,02	1,0	1,11	A

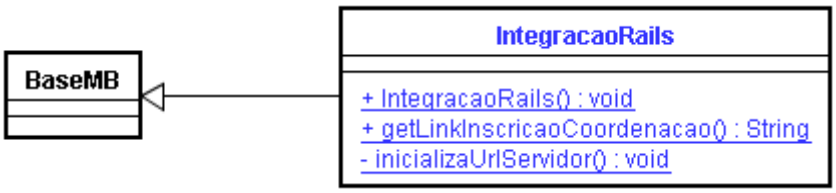


Figura 51: Classe *IntegraçãoRails*, adicionada no *commit* #4828.

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. Por exemplo, no *commit* #4622, foi alterado um arquivo, a classe *UnificarIdentificacaoMB*, que teve dois blocos de código, com duas estruturas condicionais

cada, removidos de dois métodos, o que diminuiu a complexidade do projeto. A alteração deste *commit* é mostrada na Figura 52.

```
...
@Begin(join = true)
    public void atrbuirIdentificacao1(Long idIdentificacao) {
        try {
            identificacao1 =
identificacaoService.getComIdentificadores(idIdentificacao);
            listIdentificadores1 = new
ArrayList<Identificador>(identificacao1.getIdentificadores());
            if (listIdentificadores2 != null && listIdentificadores1.size()
< listIdentificadores2.size()) {
                for (int i = 0; i <= listIdentificadores2.size() -
listIdentificadores1.size(); i++) {
                    listIdentificadores1.add(new Identificador());
                }
            }
            if (listIdentificadores2 != null && listIdentificadores2.size()
< listIdentificadores1.size()) {
                for (int i = 0; i <= listIdentificadores1.size() -
listIdentificadores2.size(); i++) {
                    listIdentificadores2.add(new Identificador());
                }
            }
        } catch (ObjetoRemovidoException ex) {
            error("Identificação não encontrada!");
        }
        comboEstadosIdentidadeUF1 = null;
    }

@Begin(join = true)
    public void atrbuirIdentificacao2(Long idIdentificacao) {
        try {
            identificacao2 =
identificacaoService.getComIdentificadores(idIdentificacao);
            listIdentificadores2 = new
ArrayList<Identificador>(identificacao2.getIdentificadores());
            if (listIdentificadores1 != null && listIdentificadores1.size()
< listIdentificadores2.size()) {
                for (int i = 0; i <= listIdentificadores2.size() -
listIdentificadores1.size(); i++) {
                    listIdentificadores1.add(new Identificador());
                }
            }
            if (listIdentificadores1 != null && listIdentificadores2.size()
< listIdentificadores1.size()) {
                for (int i = 0; i <= listIdentificadores1.size() -
listIdentificadores2.size(); i++) {
                    listIdentificadores2.add(new Identificador());
                }
            }
        } catch (ObjetoRemovidoException ex) {
            error("Identificação não encontrada!");
        }
        comboEstadosIdentidadeUF2 = null;
    }
...
```

Figura 52: Alteração realizada no *commit* # 4622.

A nona regra mostra que, quando Tywin faz alterações, a chance de compilar é 11% maior que a chance geral de compilar, sem considerar o desenvolvedor fazendo a alteração. Isso acontece, pois esse desenvolvedor, apesar de inexperiente, desde quando entrou na equipe, se preocupava em verificar se o que tinha sido implementado estava correto e não iria atrapalhar os outros desenvolvedores. O que é confirmado, pois todos os seus *commits* compilam. Para evidenciar melhor essa regra, na Tabela 24 são mostradas as maiores e menores taxas de compilação dos desenvolvedores deste projeto.

Tabela 24: Relação das maiores e menores taxas de compilação dos desenvolvedores do IdUFF.

Desenvolvedor	Taxa de compilação	Commits
Tywin	100,00%	52
Robert	100,00%	3
Renly	100,00%	44
Jofrey	67,11%	76
Jaime	50,00%	2
Jon	41,67%	12

É importante notar que a Tabela 24, apesar de parecer simples, possui uma confecção complexa, pois os dados nela contidos dependem de um processamento custoso realizado pela abordagem Ostra. Nessa tabela, estão sumarizadas as taxas de compilação de alguns desenvolvedores e para se chegar a esse dado deve-se verificar se cada versão do projeto, resultante de um *commit*, compila ou não. Com a base de dados gerada pela abordagem Ostra, após a medição do projeto, essa tabela pode ser facilmente montada. Consequentemente, essa abordagem não evidencia apenas informações com as regras mineradas, mas também disponibiliza informações sobre cada *commit* e desenvolvedor que não são triviais de se conseguir.

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 25.

Enquanto a oitava regra positiva mostra que quando um determinado desenvolvedor 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. Isto pode ser confirmado pela Tabela 24, construída com as informações extraídas da abordagem, como explicado anteriormente. Apesar de esse desenvolvedor ser experiente, provavelmente, por um excesso de confiança, ele não verifica se as suas alterações mantêm o projeto compilando, resultando assim no aumento da chance de não compilar em três vezes o esperado.

Tabela 25: Regras negativas do projeto IdUFF.

#	Regra	Sup.	Conf.	Lift	Base
1	Se a quantidade de classes diminui, Então a reusabilidade diminui.	0,01	0,75	4,61	B
2 *	Se Jofrey faz alterações, Então não compila.	0,01	0,32	3,22	A
3	Se Stannis faz alterações, Então a quantidade de linhas de código diminui.	0,01	0,41	2,65	B
4	Se Tyrion faz alterações, Então a complexidade aumenta.	0,00	1,0	2,30	B
5	Se o commit é na sexta-feira, então a reusabilidade diminui.	0,02	0,3	2,05	B
6 *	Se a quantidade de classes aumenta, Então o entendimento diminui.	0,18	0,94	2,04	B
7 *	Se o commit é na sexta-feira , Então não compila.	0,15	0,11	1,50	A
8	Se Renly faz alterações, Então a complexidade aumenta.	0,01	0,5	1,33	A
9	Se Tommem faz alterações, Então a complexidade aumenta.	0,03	0,45	1,21	A
10	Se o commit é no turno da noite, Então não compila.	0,02	0,12	1,20	A

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, ao longo do desenvolvimento de um projeto, o entendimento diminua, pois se torna mais difícil conhecer o projeto e entendê-lo (BANSIYA; DAVIS, 2002). O *commit* #3698 ilustra essa regra. Nele, foi adicionada uma nova classe *Pontuação*, que aumentou a quantidade de classes e de métodos, diminuindo assim o entendimento do projeto, não de acordo com a legibilidade do código escrito, mas de acordo com o tamanho do projeto.

A sétima regra indica um padrão temporal, ressaltando que na sexta-feira a chance de compilar é menor. Neste projeto, a sexta-feira é um dia marcado por reuniões com cliente e entre os membros da equipe de desenvolvimento: são realizadas reuniões de entrega e apresentação das funcionalidades desenvolvidas para o cliente, e reuniões de melhoria contínua do processo de desenvolvimento adotado pela equipe. Provavelmente, *commits* nesse dia são realizados com pressa e com menos atenção, resultando em uma taxa de compilação menor. Na Tabela 26, estão as taxas de compilação dos dias da semana, onde se pode observar que a sexta-feira possui a pior taxa de compilação dentre os dias da semana.

Tabela 26: Taxa de compilação dos dias da semana do projeto IdUFF.

Dia da semana	Total de *Commits*	*Commits* que Compilam	Taxa de compilação
Domingo	2	2	100,00%
Segunda-feira	252	231	91,67%
Terça-feira	358	318	88,83%
Quarta-feira	338	302	89,35%
Quinta-feira	328	298	90,85%
Sexta-feira	229	202	88,21%
Sábado	2	2	100,00%
Geral	1509	1355	92,70%

5.2.2 PROJETO: PÚBLICO CORE

O Publico-Core é um módulo do IdUFF que fornece serviços e entidades do domínio. O objetivo desse módulo é centralizar o acesso à parte comum a vários sistemas, por exemplo, informações sobre os usuários e alunos. Esse projeto possui quatro anos de desenvolvimento e uma parte da equipe do projeto IdUFF participou do seu desenvolvimento. Na Tabela 27, são exibidas as regras positivas desse projeto.

Tabela 27: Regras positivas do projeto Publico Core.

#	Regra	Sup.	Conf.	Lift	Base
1 *	Se aumenta a quantidade de classes, Então diminui a complexidade média dos métodos.	0,14	0,86	3,32	B
2	Se nove ou mais artefatos são *alterados,* Então aumenta a quantidade de classes.	0,11	0,55	3,20	B
3 *	Se Stannis *faz alterações,* então diminui a quantidade de linhas de código.	0,04	0,25	2,44	B
4	Se a reusabilidade aumenta, Então o entendimento diminui.	0,11	0,83	2,40	B
5	Se a reusabilidade aumenta, Então diminui a complexidade média dos métodos.	0,08	0,61	2,35	B
6 *	Se nove ou mais artefatos são *alterados*, Então diminui a complexidade média dos métodos	0,12	0,59	2,28	B
7	Se Theon *faz alterações,* Então o *commit* é na terça.	0,04	0,37	2,26	B
8	Se cinco a oito artefatos são *alterados,* Então a reusabilidade aumenta.	0,04	0,3	2,11	B
9	Se Bran *faz alterações*, Então a quantidade de linhas de código não se altera.	0,08	0,52	1,70	B
10	Se Theon *faz alterações*, Então o *commit* é no turno da noite.	0,04	0,37	1,64	B

A primeira regra positiva indica que, quando a quantidade de classes aumenta, a complexidade média dos métodos diminui. Essa regra possui uma confiança de 86%, ou seja, em 86% das vezes que aumenta a quantidade de classes, a complexidade média dos métodos diminui. Entretanto o mais importante é apresentado pelo *lift* de 3,32, indicando que a chance

do consequente acontecer é três vezes maior com esse precedente, do que se considerado isoladamente (26%).

O comportamento ressaltado na primeira regra indica sustentabilidade e melhoria contínua do projeto, associado à preocupação com a complexidade dos métodos. Uma arquitetura bem definida e seguida pela equipe de desenvolvimento justifica esse padrão: à medida que aumentam as classes, provavelmente, aumenta a quantidade de métodos, resultando em menor complexidade média. Quando são criados métodos de acesso ao banco de dados, os desenvolvedores respeitam as camadas do padrão de projetos MVC (Modelo, Visualização e Controle) e utilizam também as classes de acesso a dados (*Data Access Object*, DAO). Assim, quando devem ser apresentadas informações para o usuário em uma nova funcionalidade, uma hierarquia de métodos nas camadas MVC e DAO é criada. Isso resulta na diminuição da complexidade média por métodos, pois esses métodos são simples, normalmente realizando apenas uma consulta implementada em *Hibernate Query Language* (HQL), com Complexidade Ciclomática 1.

Um exemplo desse caso é o *commit* #10032, no qual, para se exibir na tela uma busca com parâmetros específicos de uma classe de modelo, quatro classes foram alteradas. As alterações realizadas são mostradas nas Figuras 53-56. Na Figura 53, pode-se notar a HQL incluída nesse *commit*: *Recurso.getListaPorNome*, com a anotação *@NamedQuery*. Na Figura 54, o método homônimo foi criado na interface *RecursoDAO*. Na Figura 55, o método que implementa essa consulta foi incluído na classe *RecursoHibernateDAO*, com a anotação *@MétodoRecuperaLista* que indica que a execução é realizada através de uma *NamedQuery*. Na Figura 56, o novo método de busca é disponibilizado no *RecursoService* para que seja utilizado na camada de interface com o usuário.

```
package br.uff.publico.core.model;
...
@NamedQueries({
...
   @NamedQuery(name = "Recurso.getListaPorNome",
   query = "select recurso from Recurso recurso " +
   "where upper(recurso.nome) like upper(?)"),
   ...
})
@Entity
...
public class Recurso implements Serializable {
...
}
```

Figura 53: Modificação realizada na classe Recurso no *commit* #10032.

```
package br.uff.publico.core.persistence;
...
public interface RecursoDAO extends DaoGenerico<Recurso, Long> {
   public Recurso getPorNome(String nome)  throws ObjetoNaoEncontradoException;
   public List<Recurso> getListaCompleta();
   public List<Recurso> getListaPorPapel(Papel papel);
   public List<Recurso> getListaPorNome(String nome);
}
```

Figura 54: Modificação realizada na classe RecursoDAO no *commit* #10032.

```
package br.uff.publico.core.persistence.hibernate;
...
@Repository(value="recursoDAO")
public class RecursoHibernateDAO extends HibernateDaoGenerico<Recurso, Long> implements
RecursoDAO {
...
   @MetodoRecuperaLista
   public List<Recurso> getListaPorNome(String nome) {
      throw new MetodoNaoInterceptadoException();
   }
}
```

Figura 55: Modificação realizada na classe RecursoHibernateDAO no *commit* #10032.

```
package br.uff.publico.core.service;
...
@Service
public class RecursoService extends BaseService {
...
   public List<Recurso> getListaPorNome(String nome) throws ObjetoNaoEncontradoException {
      return recursoDAO.getListaPorNome(nome + "%");
   }
...
}
```

Figura 56: Modificação realizada na classe RecursoService no *commit* #10032.

A terceira regra positiva indica que, quando Stannis faz alterações, a chance de diminuir as linhas de código (25%) é o maior que o dobro, se comparado a quando não se considera o desenvolvedor (10,2%). Essa regra evidencia um comportamento esperado, pois a principal tarefa desse desenvolvedor era fazer manutenções corretivas e perfectivas (refatorações) no projeto, consequentemente refatorações faziam parte do seu dia-a-dia. Cada *commit* tem uma mensagem associada que é utilizada para indicar o que foi realizado na modificação. Na Tabela 28, são mostradas mensagens dos seis commits que dão suporte a essa regra e que comprovam a hipótese apresentada. Observando-se a tabela, percebe-se que os seis *commits* foram de refatorações corretivas ou evolutivas, realizadas pelo desenvolvedor Stannis.

Tabela 28: Mensagens de commits de refatoração.

# Commit	Mensagem do commit
2769	Término refatoração dos modelos e *DAOs* para usar o *schema* (Produção, Homologação ou Teste)
2994	*Refatoração da classe de funcionário: Troquei o tipo primitivo *boolean* para a classe *Wrapper Boolean* para permitir melhor uso em conjunto com a classe professor * Criei o papel funcionário, ao qual todo funcionário passa a ser alocado. * Retirei a coluna tipo da classe papel, que não era usada.
2997	Revertendo modificações do órgão (feitas pelo *Rickon*).
3172	Término refatoração de órgão.
4228	*Correção na inclusão e alteração de identificação - parte de validação de cpf e pessoa estrangeira

Enquanto as duas regras apresentadas em detalhes anteriormente evidenciam informações sobre quem fez a alteração e como ela ocorreu, a sexta regra traz informações a respeito do tamanho das modificações. Essa regra mostra que, quando são alterados nove ou mais artefatos, a complexidade média dos métodos diminui com uma chance maior. Essa regra é ilustrada pelo diagrama de classes da Figura 57.

O comportamento evidenciado pela sexta regra positiva pode ser causado por uma refatoração que adicione pouca complexidade e que adicione ou altere muitos métodos e classes. Na Figura 57, é apresentado um diagrama de classes com as alterações realizadas no *commit* #2987, seguindo a notação definida anteriormente. Nesse *commit*, foram alteradas sete classes e criadas três classes e três interfaces. Como as classes novas e as alterações nas classes existentes não aumentaram consideravelmente a complexidade total, a complexidade média dos métodos diminuiu. Isso aconteceu, pois a maioria dos métodos criados tinha somente um caminho, sem decisões ou iterações, e consequentemente adicionou apenas uma unidade na Complexidade Ciclomática, enquanto contava como mais um método na quantidade total de métodos.

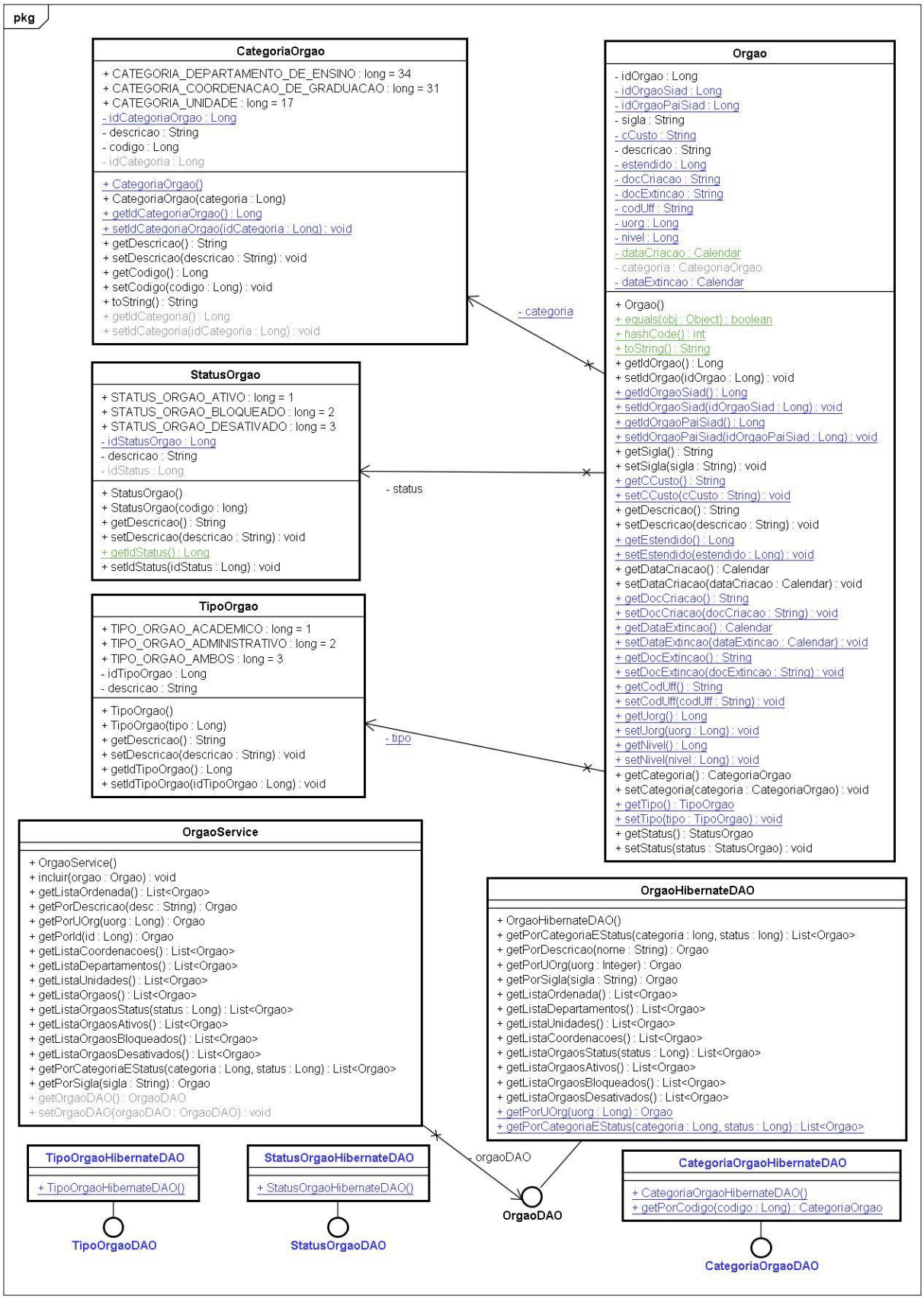


Figura 57: Diagrama de classes ilustrando alterações realizadas no commit #2987.

Na Tabela 29, são apresentadas as dez regras negativas do projeto Público Core. As regras mostradas em mais detalhes são: a segunda, a quarta e a nona.

Tabela 29: Regras negativas do projeto Público Core.

#	Regra	Sup.	Conf.	Lift	Base
1	Se Viserys *faz alterações,* Então não compila.	0,02	0,75	14,35	A
2 *	Se a quantidade de classes aumenta, Então a reusabilidade diminui.	0,13	0,77	3,63	B
3	Se a quantidade de classes aumenta, Então o entendimento diminui.	0,17	1,0	2,88	B
4 *	Se nove ou mais artefatos são *alterados,* Então Reusabilidade diminui.	0,11	0,55	2,61	B
5	Se Bran *faz alterações,* Então não compila.	0,02	0,12	2,39	A
6	Se a quantidade de linhas de código diminui, Então a reusabilidade diminui.	0,04	0,46	2,17	B
7	Se cinco a oito artefatos são *alterados,* Então a complexidade média dos métodos aumenta.	0,05	0,35	2,11	B
8	Se nove ou mais artefatos são *alterados,* Então não compila.	0,02	0,1	1,91	A
9 *	Se nove ou mais artefatos são *alterados,* Então o entendimento diminui.	0,13	0,62	1,81	B
10	Se Theon *faz alterações,* Então a reusabilidade diminui.	0,04	0,37	1,76	B

A segunda regra negativa indica que, à medida que a quantidade de classes do projeto aumenta, a reusabilidade diminui, com uma chance 3,63 vezes maior que se a variação de classes não fosse considerada. Essa regra deve ser investigada, pois apresenta um comportamento negativo e prejudicial ao projeto. Espera-se que, à medida que o projeto evolua, o atributo de qualidade reusabilidade aumente (BANSIYA; DAVIS, 2002). Além disso, a quantidade de classes influencia positivamente esse atributo de qualidade. Dessa forma, para a reusabilidade diminuir, com a quantidade de classes aumentando, a coesão ou a comunicação devem ter diminuído ou o acoplamento aumentando significativamente. Um dos *commits* em que a quantidade de classes aumentou, mas a reusabilidade diminuiu, é o #21338, no qual foi adicionado um novo modelo que mapeia uma entidade do banco de dados. Nesse *commit*, a coesão e a comunicação diminuíram, enquanto o acoplamento e quantidade de classes aumentaram. Nele, foram criadas quatro classes Java, como mostrado no diagrama de classes da Figura 58.

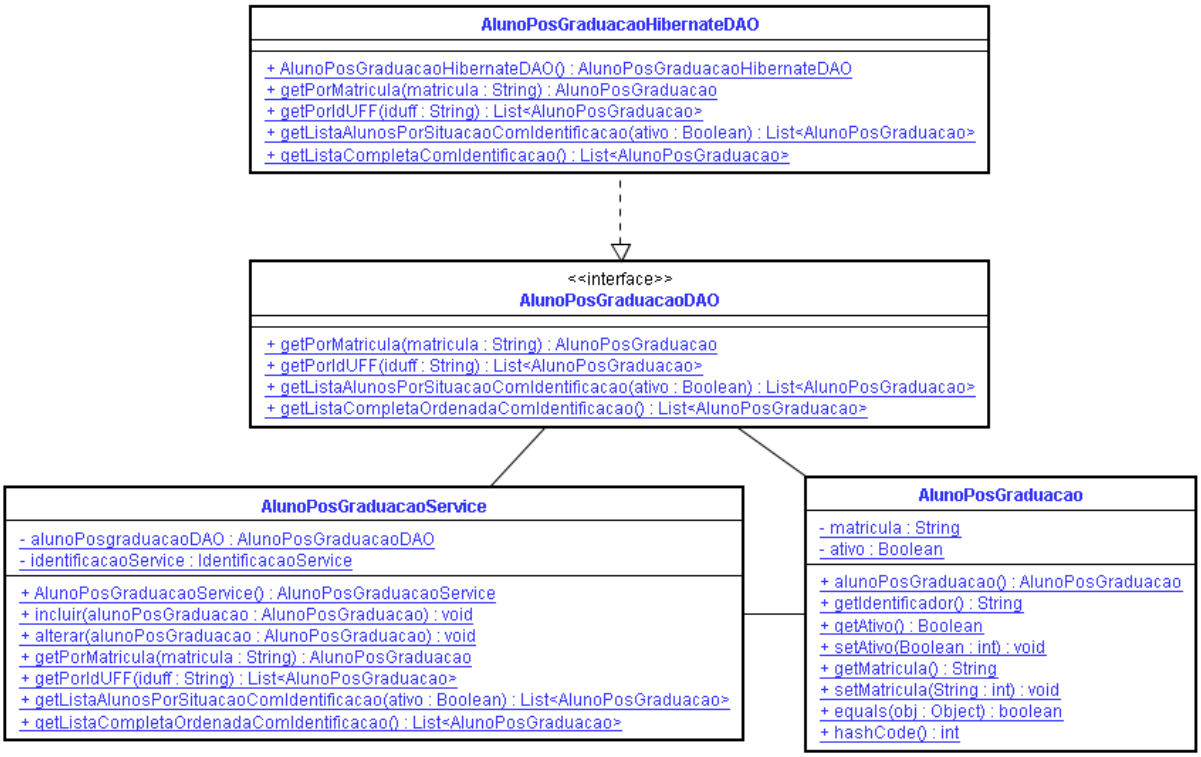


Figura 58: Diagrama de classes das modificações do *commit* #21338.

A quarta regra mostra que, quando são alterados nove ou mais artefatos, a reusabilidade diminui em mais de 50% dos casos. Provavelmente, *commits* grandes diminuem a coesão, implicando negativamente na reusabilidade. Um exemplo disso ocorre no *commit* #5847, ilustrado na Figura 59, no qual foram alterados seis classes e quatro pacotes: criando dois métodos, duas *named queries*, alterando interfaces e alterando outro método. Essas alterações resultaram na diminuição da coesão das classes, reduzindo assim a reusabilidade.

A nona regra negativa mostra que, em mais de 60% dos casos em que são alterados nove ou mais artefatos, o entendimento diminui. O entendimento pode ter diminuído, pois foram criados novos métodos ou classes e o tamanho do projeto influencia negativamente o entendimento o projeto. Isso pode ser comprovado, pois seis dos dezessete *commits* de nove ou mais arquivos que diminuíram o entendimento criaram classes e todos os dezessete criaram métodos.

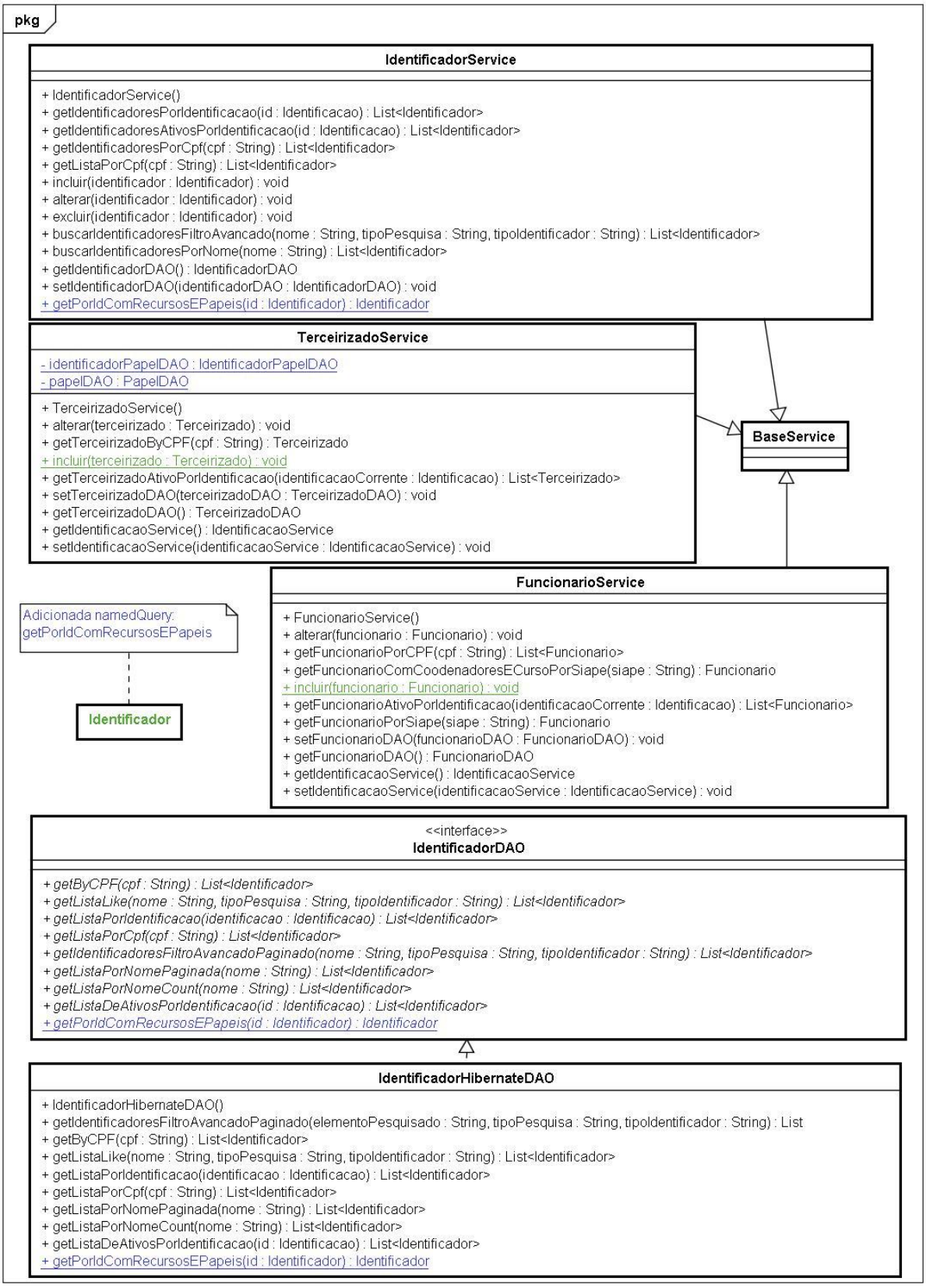


Figura 59: Diagrama de classes mostrando alterações do *commit* #5847.

5.2.3 PROJETO: MAVEN JAVADOC PLUGIN

O projeto *Maven Javadoc Plugin* [11] é um *plugin* do Maven que gera a documentação do código fonte do projeto utilizando a ferramenta homônima do Java. Esse projeto vem sendo desenvolvido há seis anos e conta com 248 *commits* feitos por 21 desenvolvedores. Na Tabela 30, são apresentadas as dez regras positivas desse projeto.

Tabela 30: Regras positivas do Projeto *Maven Javadoc Plugin*.

#	Regras	Sup.	Conf.	Lift	Base
1	Se a quantidade de classes aumenta, Então a reusabilidade aumenta.	0,08	0,90	7,51	B
2	Se a complexidade diminui, Então a complexidade média dos métodos diminui	0,07	0,90	4,00	B
3	Se a complexidade média dos métodos diminui, Então a reusabilidade aumenta	0,07	0,32	2,65	B
4 *	Se a reusabilidade aumenta, Então a complexidade média dos métodos diminui	0,07	0,6	2,65	B
5	Se a quantidade de classes aumenta, Então a complexidade média dos métodos diminui	0,04	0,54	2,41	B
6	Se a quantidade de linhas de código diminui, Então a complexidade média dos métodos diminui	0,07	0,51	2,27	B
7 *	Se o entendimento diminui, Então a complexidade média dos métodos diminui	0,16	0,46	2,05	B
8	Se o commit realizado durante a manhã, Então a quantidade de classes aumenta	0,05	0,11	1,33	B
9	Se Doran *faz alterações,* Então a reusabilidade não varia	0,10	0,92	1,28	B
10	Se três ou quatro artefatos são *alterados,* Então a complexidade média dos métodos diminui	0,05	0,27	1,22	B

Nessa tabela, pode-se observar que a quarta regra indica que, quando a reusabilidade aumenta, a complexidade média dos métodos diminui, com 60% de confiança. Uma das formas de se aumentar a reusabilidade é criando métodos públicos, já que a métrica CIS influencia positivamente esse atributo de qualidade e contabiliza justamente esses componentes. Aumentar a interface de uma classe tem esse resultado, pois se pode aumentar a quantidade de serviços que podem ser utilizados por outras classes. O *commit* #360590, que ilustra esse caso, é mostrado na Figura 60. Nele, a única alteração foi a criação de um método público.

A sétima regra positiva indica que a chance da complexidade média dos métodos diminuir é duas vezes maior quando o entendimento também diminui. Quando um algoritmo fica muito subdividido em muitos métodos, torna-se mais difícil compreender seu funcionamento. Assim, a complexidade média diminui, já que são criados novos métodos para dividir o seu comportamento, porém o entendimento também diminui. Um exemplo de

[11] *http://maven.apache.org/plugins/maven-javadoc-plugin/*

commit no qual isso ocorreu é o #752326, no qual foi criado um novo método, utilizado por outros da mesma classe, mostrado na Figura 61. Para essa figura caber em apenas uma página, apenas estão sendo exibidos os métodos, pois não houve alteração nos atributos.

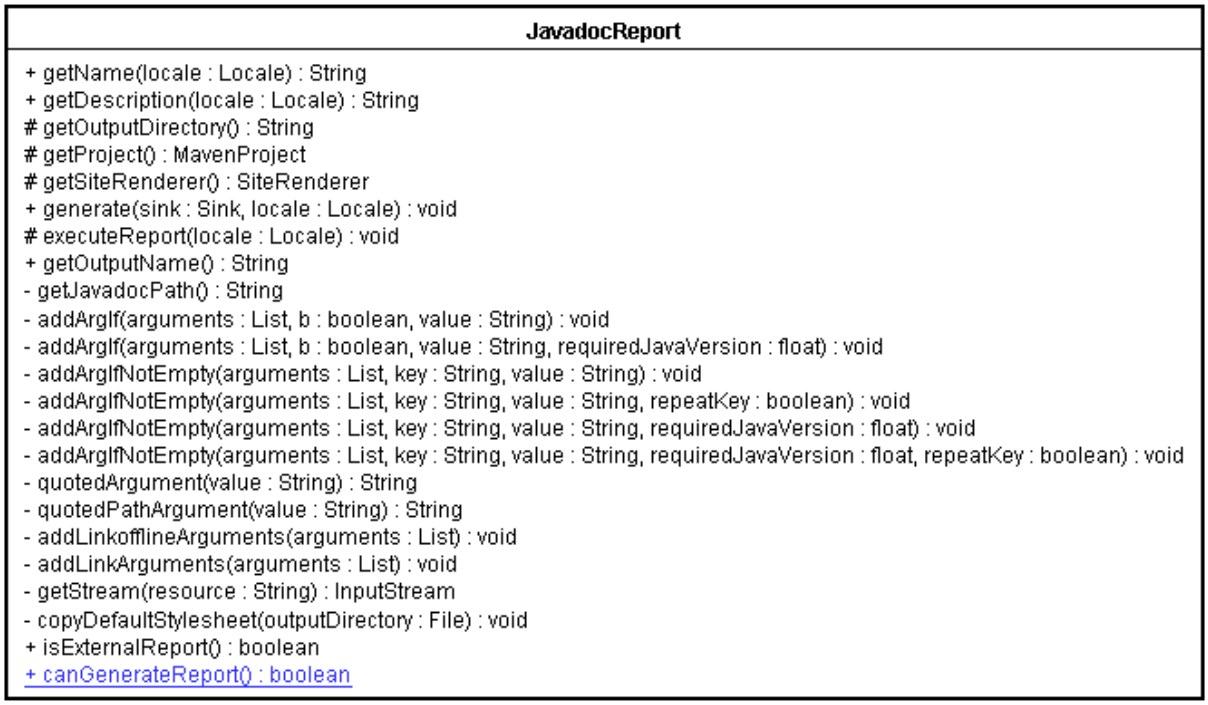


Figura 60: Diagrama de classes mostrando as alterações realizadas no *commit* #360590.

Na Tabela 31, são mostradas as regras negativas desse projeto. A primeira, sétima e nona regras serão apresentadas em mais detalhes.

Tabela 31: Regras negativas do projeto Maven Javadoc Plugin.

#	Regras	Sup.	Conf.	Lift	Base
1 *	Se a reusabilidade aumenta, Então o entendimento diminui.	0,11	0,96	2,78	B
2	Se o entendimento diminui, Então a reusabilidade aumenta.	0,14	0,40	2,58	B
3	Se a reusabilidade diminui, Então o entendimento diminui.	0,14	0,89	2,58	B
4	Se Arya *faz alterações,* Então não compila.	0,00	0,6	2,43	A
5	Se a complexidade média dos métodos diminui, Então o entendimento diminui.	0,16	0,71	2,05	B
6	Se cinco a oito artefatos são *alterados,* Então o entendimento diminui.	0,10	0,70	2,02	B
7 *	*Se o commit* é na terça-feira, Então não compila.	0,05	0,47	1,92	A
8	Se a quantidade de classes aumenta, Então a complexidade aumenta.	0,08	0,90	1,83	B
9 *	Se o commit é na sexta-feira, Então a complexidade média dos métodos aumenta.	0,07	0,57	1,50	B
10	Se cinco a oito artefatos são *alterados,* Então a complexidade média dos métodos aumenta.	0,08	0,56	1,48	B

Figura 61: Diagrama de classes mostrando[12] as modificações realizadas no *commit* #752326.

A primeira regra negativa mostra que em 96% dos casos em que aumenta a reusabilidade, diminui o entendimento. De acordo com Bansiya e Davis (2002), é esperado que à medida que um projeto evolui, a quantidade de classes aumente, resultando na

[12] Neste diagrama, a declaração de método "*...(...:...)...*" indica que existem outros métodos que não sofreram alterações neste *commit* e que foram removidos do diagrama para que ele coubesse na página.

diminuição do entendimento, enquanto os outros cinco atributos de qualidade aumentam. A comprovação dessa hipótese se dá ao observar que, em 69% (20 dos 29) dos *commits* que suportam essa regra, a quantidade de classes aumentou.

A sétima regra da Tabela 31 indica que na terça-feira, a chance de não compilar (47%) é quase o dobro da chance se o dia da semana for desconsiderado (24%). Essa regra levanta duas questões que devem ser investigadas para se tentar reverter este comportamento. A primeira é se acontece algum evento diferente nesse dia que implica na diminuição da chance de compilar, como muitas reuniões ou *commits* de desenvolvedores inexperientes de acordo com a distribuição de trabalho. Porém, esse projeto é de código aberto e não se tem conhecimento sobre os desenvolvedores ou sobre os turnos de trabalho, se é que algum foi definido. Entretanto, se as taxas de compilação globais dos desenvolvedores forem comparadas às da terça-feira, respectivamente, Tabelas 32 e 33, observa-se que elas são menores nesse dia para todos os desenvolvedores que a taxa variou. Dessa forma, não é apenas um desenvolvedor que causa a diminuição da chance de compilar nesse dia.

Tabela 32: Taxa de compilação dos desenvolvedores do projeto Maven Javadoc Plugin.

Desenvolvedor	Compila	Não Compila	Total	% Compila	% Não Compila
Myrcella	9	0	9	100	0
Balon	7	3	10	70	30
Aeron	14	1	15	93,33	6,67
Asha	1	0	1	100	0
Euron	6	0	6	100	0
Victarion	1	0	1	100	0
Kevan	1	0	1	100	0
Lancel	2	0	2	100	0
Doran	27	4	31	87,1	12,90
Arianne	9	0	9	100	0
Oberyn	2	3	5	40	60
Quentyn	1	0	1	100	0
Elia	1	0	1	100	0
Lyanna	3	0	3	100	0
Daenerys	6	2	8	75	25
Aemon	4	0	4	100	0
Aegon	1	0	1	100	0
Rhaegar	2	0	2	100	0
Hoster	1	0	1	100	0
Edmure	148	68	216	68,52	31,48
Brynden	2	0	2	100	0

Tabela 33: Taxa de compilação dos desenvolvedores do projeto Maven Javadoc Plugin na terça-feira.

Desenvolvedor	Compila	Não Compila	Total	% Compila	% Não Compila
Balon	1	2	3	33,33	66,67
Victaion	0	1	1	0	100
Doran	2	1	3	66,67	33,33
Arianne	2	0	2	100	0
Daenerys	2	1	3	66,67	33,33
Aemon	1	0	1	100	0
Aegon	1	0	1	100	0
Edmure	9	14	23	39,13	60,87
Brynden	1	0	1	100	0

Ainda, com o objetivo de explicar a sétima regra negativa, foi realizada uma busca nas listas de discussão deste projeto procurando por mensagens que evidenciassem alguma prática ou metodologia de desenvolvimento que pudesse justificar a diminuição da chance de compilar na terça-feira, mas sem sucesso. Consequentemente, outro questionamento torna-se interessante: "se nesse dia as chances diminuem, existe algum dia em que elas aumentam consideravelmente?" Ao observar a Figura 62, nota-se que não, pois na terça-feira realmente a taxa de compilação é menor que os outros dias, embora nenhum deles tenha uma taxa de compilação consideravelmente maior que os outros. Enquanto a chance média de compilação é quase 75%, na terça-feira ela cai para 52,63%. Conclui-se que, apesar de interessante, pela falta de informações, não se sabe ao certo o que ocasiona esse comportamento, embora a regra evidencie uma tendência que acontece na terça-feira e deva ser revertida. Vale ressaltar que as informações apresentadas nas Tabelas 32 e 33 e Figura 62 foram coletadas com a abordagem proposta.

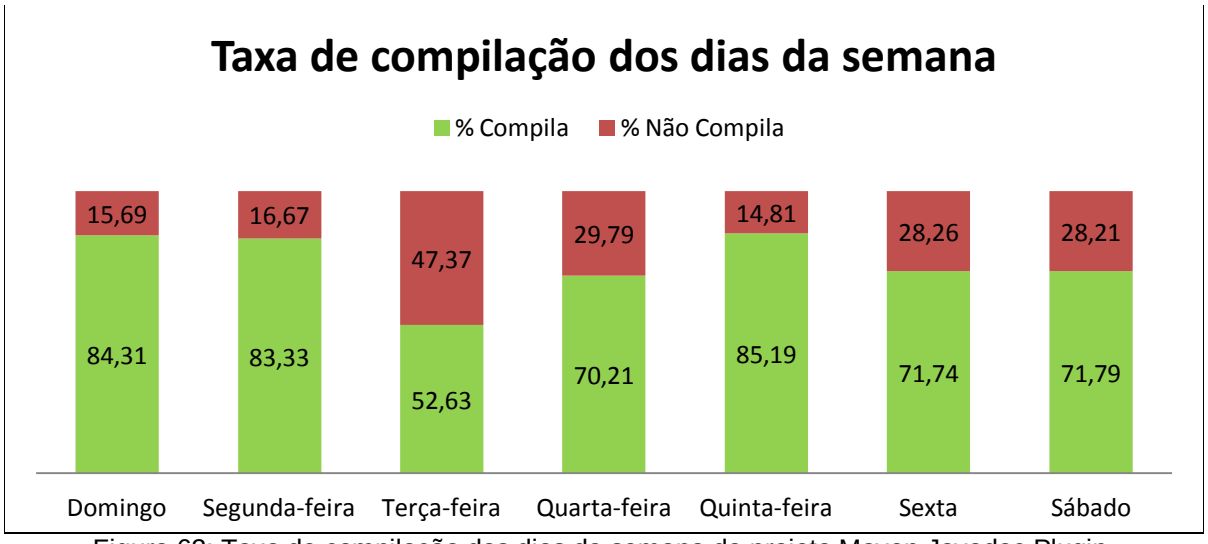


Figura 62: Taxa de compilação dos dias da semana do projeto Maven Javadoc Plugin.

5.2.4 PROJETO: MAVEN GWT PLUGIN

O projeto Maven GWT Plugin[13] também é um *plugin* do Maven e deve ser utilizado junto com a dependência do Google Web Toolkit, uma biblioteca para desenvolvimento WEB do Google. Esse projeto está em desenvolvimento desde agosto de 2006 e conta com seis desenvolvedores. As regras positivas desse projeto estão na Tabela 34.

Tabela 34: Regras positivas do projeto Maven GWT Plugin.

#	Regras	Sup.	Conf.	Lift	Base
1	Se a quantidade de classes aumenta, Então a reusabilidade aumenta.	0,08	0,87	3,93	B
2 *	Se a quantidade de linhas de código diminui, Então o entendimento aumenta.	0,09	0,5	3,15	B
3	Se a reusabilidade aumenta, Então a complexidade média dos métodos diminui.	0,15	0,67	2,75	B
4	Se a complexidade média dos métodos diminui, Então a reusabilidade aumenta.	0,15	0,61	2,75	B
5	Se o entendimento aumenta, Então a reusabilidade aumenta.	0,08	0,52	2,36	B
6	Se a reusabilidade aumenta, Então o entendimento aumenta.	0,08	0,37	2,36	B
7	Se a complexidade média dos métodos diminui, Então a quantidade de classes aumenta.	0,05	0,20	2,20	B
8	Se nove ou mais artefatos são alterados, Então a reusabilidade aumenta.	0,07	0,47	2,13	B
9	Se o commit é na quinta-feira, Então a complexidade média dos métodos diminui.	0,05	0,43	1,77	B
10	Se o commit é realizado no turno da noite, Então a complexidade média dos métodos não varia.	0,09	0,57	1,5	B

A segunda regra positiva mostra que a chance de o entendimento aumentar é três vezes maior quando a quantidade de linhas de código diminui. Esse comportamento acontece 24 vezes no histórico do projeto, como mostra o suporte de 9%. Ele pode ser explicado por refatorações que visam melhorar o entendimento do código, como o *commit* #13178, cujo objetivo foi remover código não utilizado, como indicado na mensagem de *commit*: *"remove unused code."*. As modificações realizadas nesse *commit* estão ilustradas na Figura 63.

Apesar de existirem outras regras interessantes, como a sétima e a nona regras positivas e a quarta, quinta e nona negativas, apenas a segunda regra positiva foi explicada em mais detalhes, pois as explicações das demais regras seriam muito parecidas com outras regras dos três projetos anteriores. As regras negativas do projeto Maven GWT Plugin estão na Tabela 35.

[13] *http://mojo.codehaus.org/gwt-maven-plugin/*

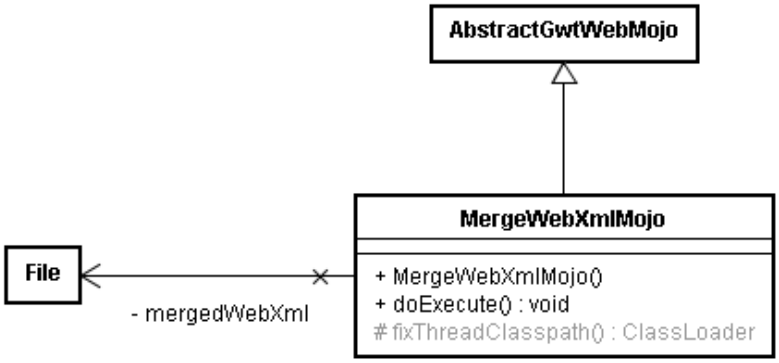


Figura 63: Diagrama de classes mostrando as alterações do *commit* #13178 do projeto Maven GWT Plugin.

Tabela 35: Regras negativas do projeto Maven GWT Plugin.

#	Regras	Sup.	Conf.	Lift	Base
1	Se a quantidade de classes aumenta, Então o entendimento diminui.	0,07	0,75	2,55	B
2	Se o entendimento diminui, Então a reusabilidade diminui.	0,13	0,45	2,41	B
3	Se a complexidade média dos métodos diminui, Então o entendimento diminui.	0,14	0,59	2,03	B
4	Se nove ou mais artefatos são *alterados,* Então a reusabilidade diminui.	0,05	0,36	1,93	B
5	Se nove ou mais artefatos são *alterados,* Então o entendimento diminui.	0,07	0,52	1,79	B
6	Se cinco a oito artefatos são *alterados,* Então o entendimento diminui.	0,06	0,51	1,75	B
7	*Se o commit* é no sábado e é relizado por Sansa, Então a complexidade média dos métodos aumenta.	0,07	0,60	1,62	B
8	Se três ou quatro artefatos são alterados, Então o entendimento diminui.	0,06	0,43	1,48	B
9	Se Sansa *faz alterações,* Então não compila.	0,18	0,27	1,43	A
10	Se a quantidade de linhas de código diminui, Então a reusabilidade diminui.	0,05	0,27	1,42	B

5.3 MONITORAMENTO DA EVOLUÇÃO DO PROJETO

Um dos objetivos da Ostra é auxiliar o monitoramento da qualidade do projeto através de gráficos. Para avaliar essa funcionalidade, a evolução de um projeto real é analisada por meio desses gráficos. O propósito desse experimento é avaliar a capacidade desses gráficos em auxiliar a identificação de ameaças à qualidade do projeto.

Nesse experimento, é analisada a evolução do projeto IdUFF. Esse projeto foi selecionado, pois o autor dessa dissertação participou de seu desenvolvimento, o que pode levar a melhores entendimento e justificativa dos resultados obtidos.

O monitoramento da evolução do IdUFF é considerado sob a perspectiva das métricas utilizadas no experimento anterior, apresentado na Seção 5.2. Assim, as métricas consideradas são: Complexidade Ciclomática de McCabe, Linhas de Código e Tamanho do Projeto em

Classes. Além dessas métricas, são considerados dois atributos de qualidade: Reusabilidade e Entendimento. Essas métricas foram escolhidas, pelo mesmo motivo do experimento anterior, pois são mais fáceis de serem entendidas e avaliadas.

Inicialmente, é realizada a análise da evolução da complexidade e do tamanho do projeto na Seção 5.3.1. Na Seção 5.3.2, é realizada a análise da evolução do entendimento e da reusabilidade.

5.3.1 ANÁLISE DA COMPLEXIDADE CICLOMÁTICA E DO TAMANHO

A complexidade ciclomática indica quantos caminhos de execução um código possui, ou seja, avalia sua estrutura. Essa métrica é útil, pois a complexidade do código está relacionada à facilidade de testar, entender e dar manutenção ao software. Quanto maior for a complexidade, mais difícil será compreender e testar o código.

Na Figura 64, é mostrado o gráfico de controle do histórico da complexidade total do projeto. Nesse gráfico, pode-se perceber que, à medida que o projeto evolui, essa métrica aumenta. Os números identificam pontos da evolução do software que serão discutidos nesta seção.

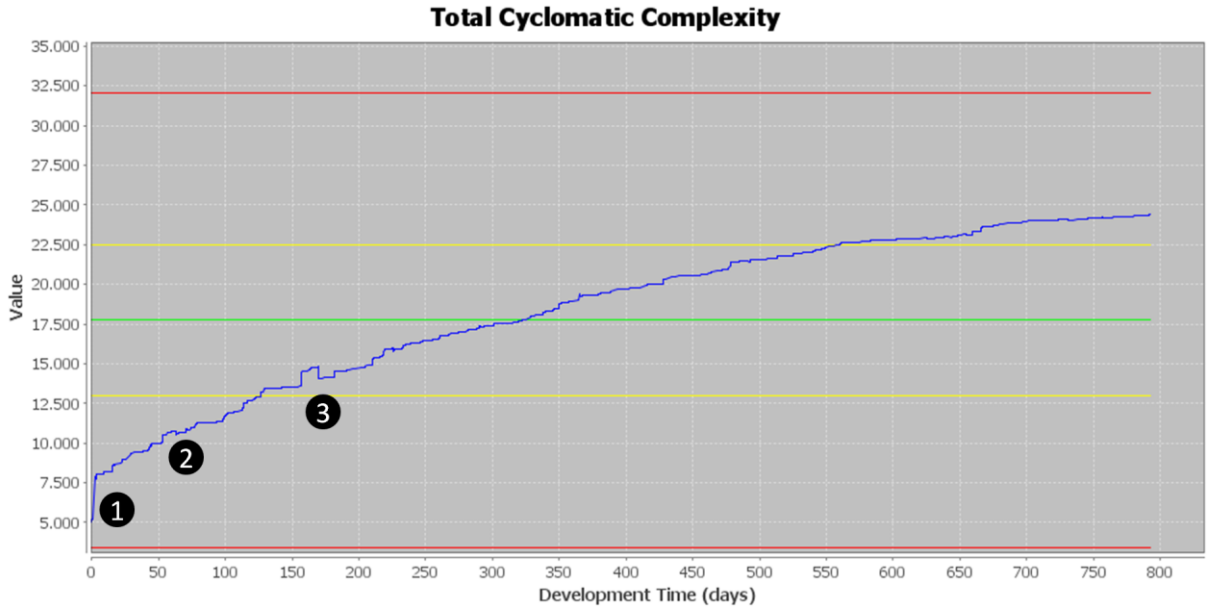


Figura 64: Gráfico de controle da métrica Complexidade Ciclomática de McCabe do projeto IdUFF.

A métrica Tamanho do Projeto em Classes (do inglês *Design Size in Classes*, DSC) contabiliza a quantidade de classes que o projeto possui. Analisando a evolução do projeto sob a perspectiva dessa métrica é possível identificar o crescimento do projeto. Na Figura 65, é mostrado o gráfico da evolução dessa métrica para o projeto IdUFF. A evolução da métrica DSC, indica que a quantidade de classes do projeto aumentou com o tempo. Esse

comportamento é esperado, pois o projeto está em desenvolvimento, sendo constantemente implementadas novas funcionalidades.

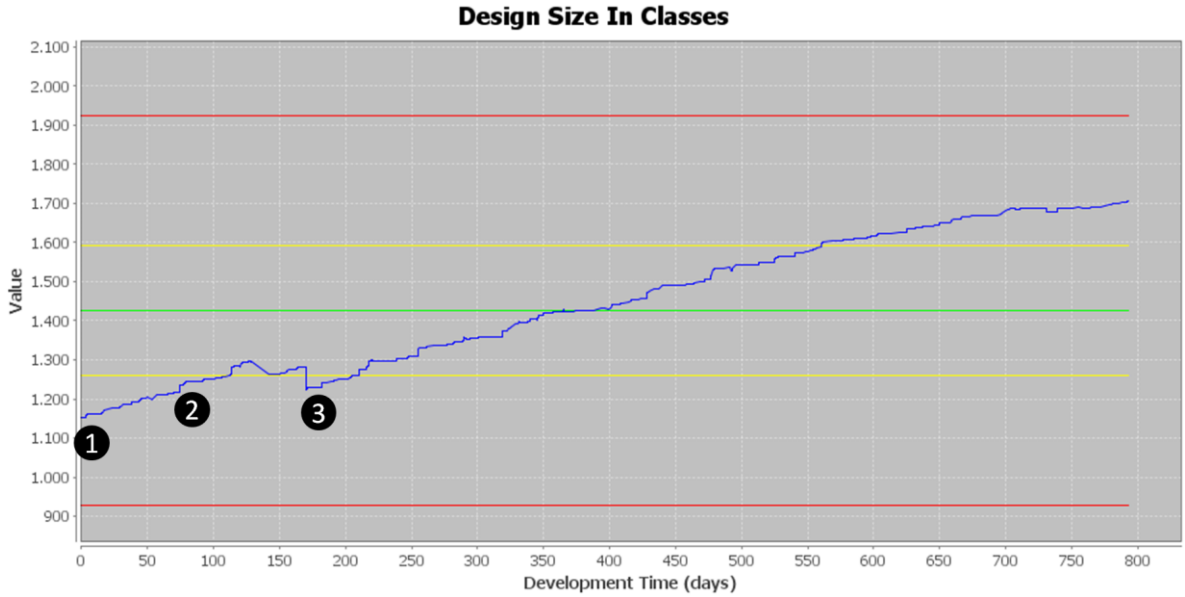


Figura 65: Gráfico de controle da métrica DSC no projeto IdUFF.

Observa-se que o projeto possui, inicialmente (rótulo "1" na Figura 65), cerca de 1.150 classes. Logicamente, o projeto não foi criado com essa quantidade de classes. O instante de tempo 0 no gráfico não indica a criação do projeto, mas a inclusão dele no repositório de controle de versões Subversion.

No rótulo "2" na Figura 65, pode-se observar um aumento significativo da quantidade de classes. O *commit* #3336 foi o responsável por essa variação repentina, na qual foram adicionadas 18 classes, pelo desenvolvedor Eddard. Como explicado anteriormente, apesar de a variação de classes ser grande para um *commit*, essa variação não mudou a reusabilidade ou entendimento de forma significativa, como é apresentado na Seção 5.3.2.

No rótulo "3", observa-se uma queda brusca na quantidade de classes, causada pelo *commit* #4201. Esse *commit* afetou 158 artefatos (arquivos e pacotes) e fez parte de uma refatoração realizada pelo desenvolvedor Stannis que continuou em outros *commits*. Nesse momento, foram removidas 57 classes. A diminuição da quantidade de classes, apesar de incomum, faz sentido se contextualizada: após a inscrição em disciplinas, que terminou em fevereiro de 2008, foi realizada uma grande refatoração no sistema que tinha como objetivo facilitar a navegação entre as entidades, na qual foi modificada a modelagem e algumas entidades foram removidas e suas informações realocadas.

Outra métrica que está relacionada ao tamanho do software é a quantidade de linhas de código. Ela contabiliza todas as linhas dos arquivos Java do software e sua evolução é muito

parecida com a evolução da métrica DSC. Na Figura 66, é mostrado o gráfico de controle da métrica LOC do projeto IdUFF.

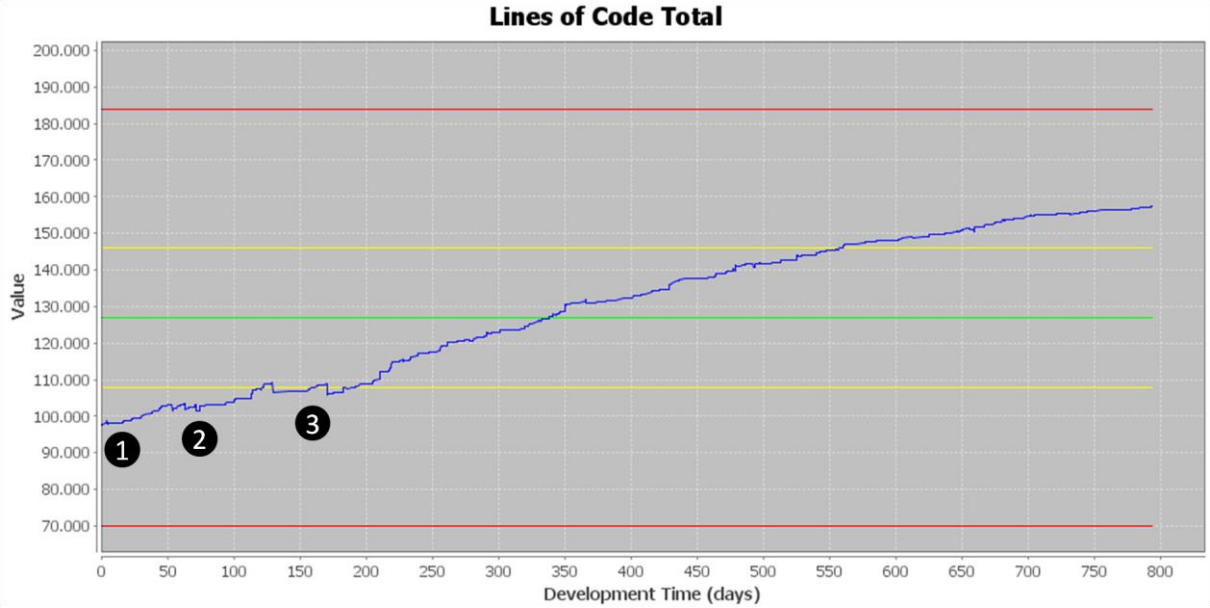


Figura 66: Gráfico de controle da métrica LOC do projeto IdUFF.

Ao observar as Figuras 65 e 66, pode-se notar que essas duas métricas tem gráficos quase idênticos, variando apenas na escala. Isso demonstra uma arquitetura equilibrada, na qual o projeto cresce em linhas de código e quantidade de classes de maneira proporcional.

Na Figura 64, pode-se observar que a complexidade ciclomática total cresce constantemente. Isso é normal já que o projeto está crescendo, como mostrado nas Figuras 65 e 66, e essa métrica aumenta proporcionalmente em relação à quantidade de funcionalidades.

Consequentemente, para analisar a complexidade do projeto, faz se necessário utilizar outra métrica, que também leve em consideração o tamanho do projeto. Essa métrica é a Densidade de Complexidade por Métodos, que indica a complexidade média dos métodos do projeto. Um valor menor que 10 para essa métrica é esperado por método (HENDERSON-SELLERS, 1995). Consequentemente, métodos com valores maiores que 10 devem ser refatorados com o objetivo de diminuir a sua complexidade. No gráfico da Figura 67, pode-se observar que os valores médios dessa métrica para o projeto IDUFF estão dentro do esperado. Inicialmente, a média era próxima de 3,5, mas com o crescimento do software e aumento da quantidade de classes e métodos, a média da complexidade diminuiu.

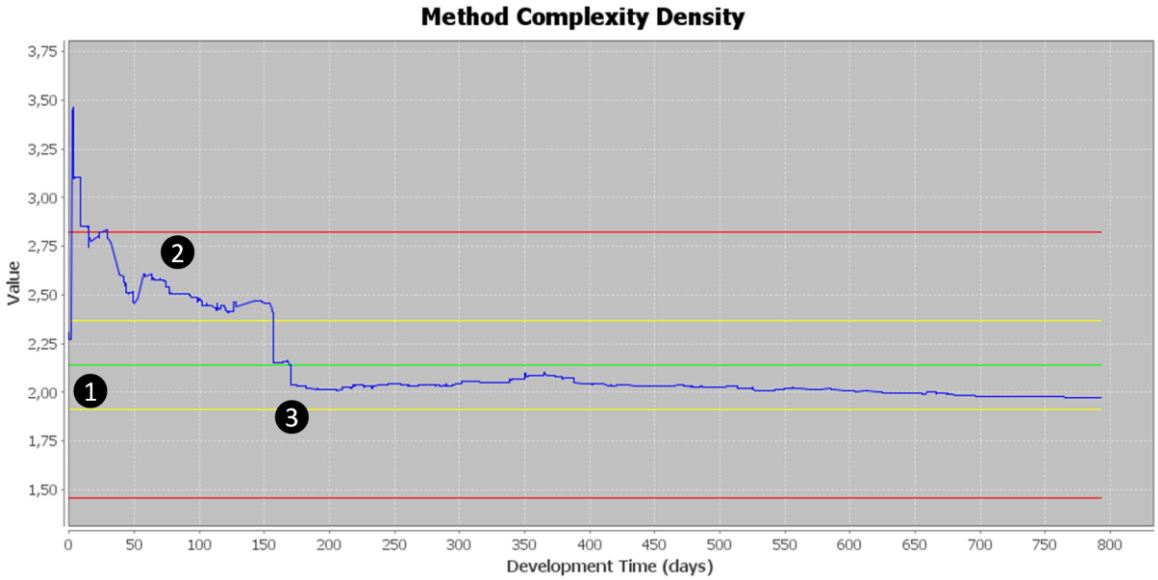


Figura 67: Gráfico de controle da métrica Densidade de Complexidade por Métodos.

Em um projeto em desenvolvimento, que está crescendo constantemente, a complexidade ciclomática total tende a aumentar, como mostra a Figura 64. Entretanto, apesar de possuir valores esperados, pode-se observar variações abruptas no gráfico da Figura 67, como os indicados pelos rótulos "1", "2" e "3".

Para facilitar o monitoramento, o gráfico de controle do histórico do delta da métrica deve ser utilizado. Na Figura 68, é mostrado o histórico do delta da métrica TCC, no qual se pode confirmar que as variações indicadas pelos rótulos “1” e “3” nas Figuras 54 e 67 realmente foram valores incomuns. Entretanto, a variação indicada pelo rótulo “2”, que também parecia suspeito, se mostrou dentro do aceitável, como mostra o gráfico da Figura 68, pois essa variação está dentro dos limites aceitáveis (abaixo da linha vermelha superior).

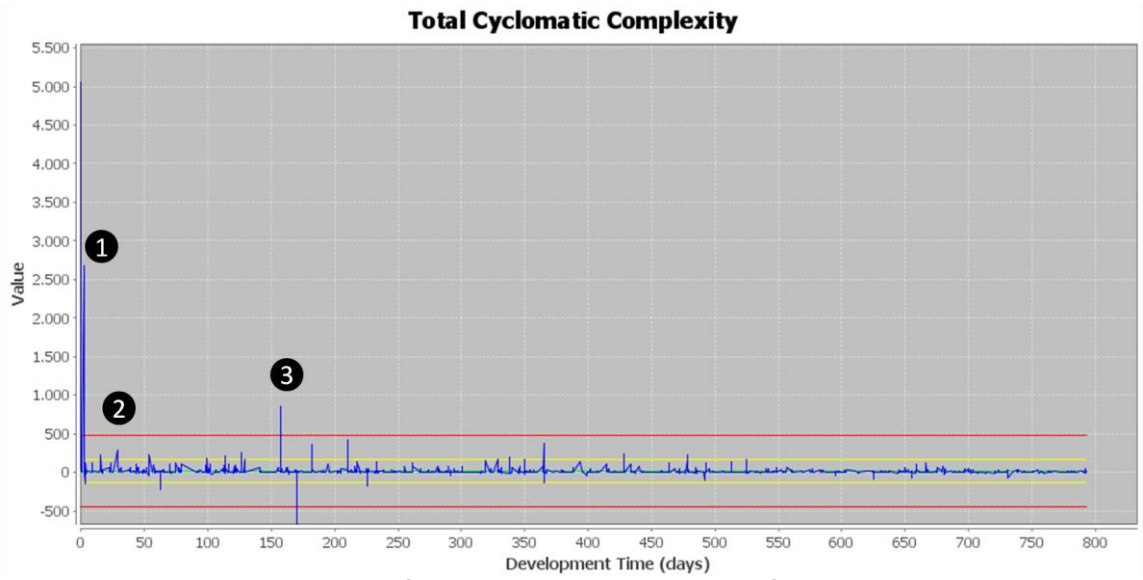


Figura 68: Gráfico de controle do delta da métrica TCC.

O gráfico de controle do histórico do delta da métrica TCC, exibido na Figura 68, evidencia variações com os rótulos “1”, “2” e “3”. Esses rótulos fazem referência às mesmas variações das figuras anteriores. Entretanto, a análise da variação dessa métrica é dificultada, pois no gráfico é exibido todo o histórico do projeto que contém variações muito grandes. Para se entender melhor a informação que esse gráfico transmite nos rótulos “1”, “2” e “3”, as Figuras 69 e 70 mostram gráficos de controle do histórico até os momentos em que ocorreram as variações identificadas pelos rótulos “1” e “3”. Na Figura 69, é exibido o histórico do projeto até a variação com rótulo “1” e, na Figura 70, é exibido o histórico do projeto até a variação com rótulo “3”. Nesses gráficos, foi utilizado no eixo das abscissas o número do *commit*, diferente dos outros gráficos, para deixar claro em qual *commit* as variações ocorreram.

No gráfico exibido na Figura 68, aparece, além das variações indicadas pelos rótulos "1", "2" e "3", outra variação além dos limites do gráfico de controle. Essa variação foi realizada pelo *commit* #4078, no qual a complexidade subiu cerca de 850. Porém sem afetar a complexidade média por métodos e as outras métricas do software significativamente, como mostram os gráficos anteriores.

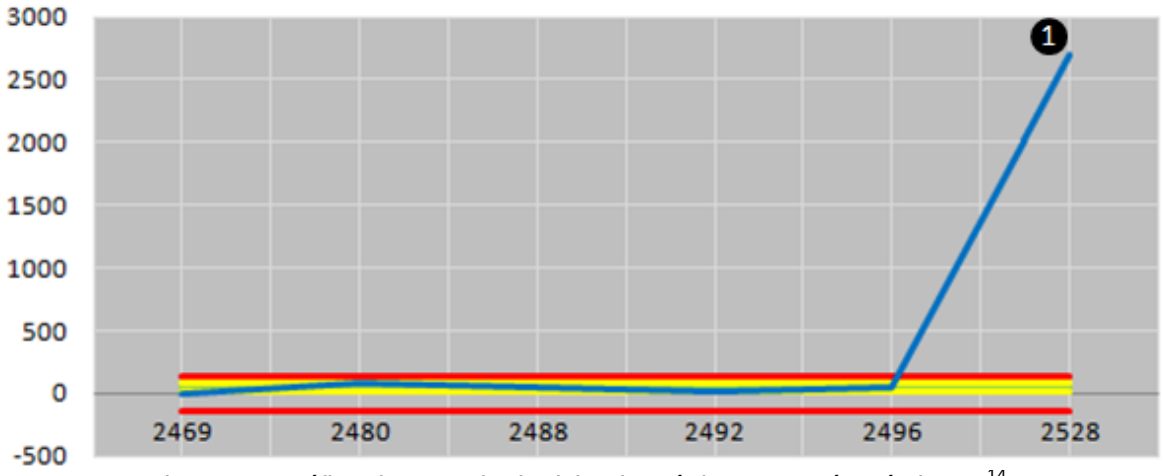


Figura 69: Gráfico de controle do delta da métrica TCC até o rótulo "1".[14]

Ao observar a Figura 69, pode-se notar que o *commit* #2528 (rótulo "1") introduziu uma complexidade muito maior que o esperado. Nesse *commit*, a variação da complexidade total foi maior que 2.500 e a variação da densidade de complexidade por métodos foi de 1,68 por método (151% em relação ao valor anterior), sendo o *commit* com maior complexidade até então, após o *commit* inicial. Esse *commit* possui características muito incomuns e deveria ter sido investigado.

[14] Nos gráficos das Figuras 69 e 70, é utilizado, no eixo das abscissas, o número do *commit* com o objetivo de destacar os pontos 1, 2 e 3.

Na Figura 70, é mostrado o gráfico de controle do histórico do delta da métrica TCC até o rótulo 3, que representa o *commit* #4201, no qual a alteração da complexidade foi de -748, afetando consideravelmente as métricas TCC e DCM, como mostrado na Figura 68. Após esse *commit*, a métrica TCC voltou ao seu comportamento crescente. Porém, a métrica DCM estabilizou e se manteve praticamente constante após este incidente, com uma diminuição muito menor ao longo do tempo.

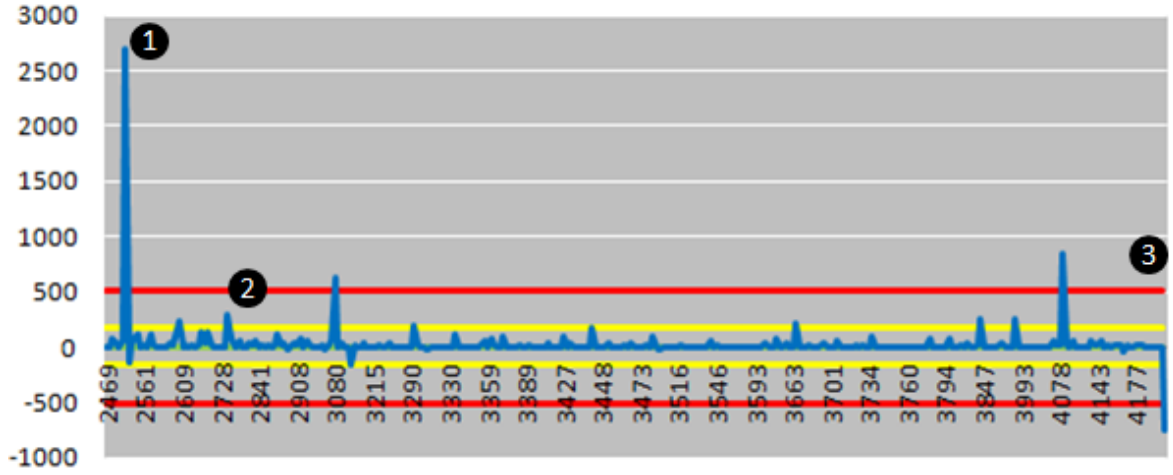


Figura 70: Gráfico de controle do delta da métrica TCC até o rótulo "3".

Conclui-se que os pontos evidenciados não afetaram negativamente a complexidade do software. Porém, surge uma questão: será que toda a complexidade removida afetou os atributos de qualidade e as outras métricas negativamente?

5.3.2 ANÁLISE DO ENTENDIMENTO E REUSABILIDADE

Na Figura 71, é apresentado o gráfico de controle da evolução do atributo de qualidade reusabilidade. Nele, nota-se que nos rótulos "1" e "2", a reusabilidade foi pouco influenciada, mas, em 3, esse atributo de qualidade aumentou significativamente. Dessa forma, conclui-se que o *commit* #4201 (ponto 3) foi muito positivo sob uma perspectiva de reusabilidade.

Entretanto, para o entendimento do projeto, os *commits* representados pelos pontos 1, 2 e 3 não foram positivos, pelo contrário, eles diminuíram esse atributo de qualidade. Na Figura 72, é mostrado o gráfico do histórico do entendimento para o projeto IdUFF, no qual se pode notar que o entendimento tende a diminuir ao longo do tempo. O entendimento diminuiu em 1, 2 e 3, o que era esperado. Nota-se que, no ponto 3, esse atributo de qualidade teve uma redução significativa, se observado seu comportamento geral, mas nem tanto, se observada a diferença em seu valor absoluto, -0,03 (de -2,67 para -2,64).

Um dos comportamentos evidenciados nas regras encontradas no experimento da Seção 5.2 indica que a reusabilidade e o entendimento evoluem de maneira contrária. Duas

regras evidenciaram esse comportamento: "quando a reusabilidade aumenta o entendimento diminui" e "quando o entendimento aumenta, a reusabilidade diminui". Apesar de terem amplitudes diferentes, nas Figuras 71 e 72, pode-se observar que quando um cresce o outro diminui. Isso se dá, segundo o modelo QMOOD, pois, à medida que o projeto cresce em quantidade de classes, torna-se mais difícil entendê-lo (BANSIYA; DAVIS, 2002). Por outro lado, à medida que a quantidade de classes aumenta, a reusabilidade do projeto também aumenta, pois mais classes estarão disponíveis para serem reutilizadas através de seus métodos públicos.

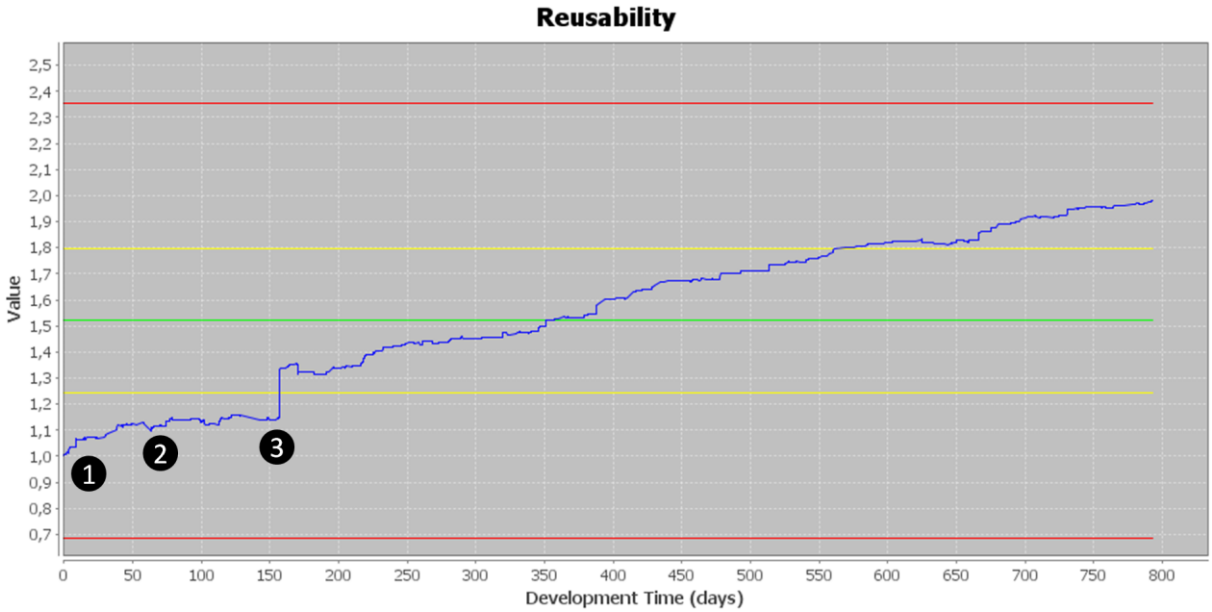


Figura 71: Gráfico de controle do atributo de qualidade reusabilidade do projeto IdUFF.

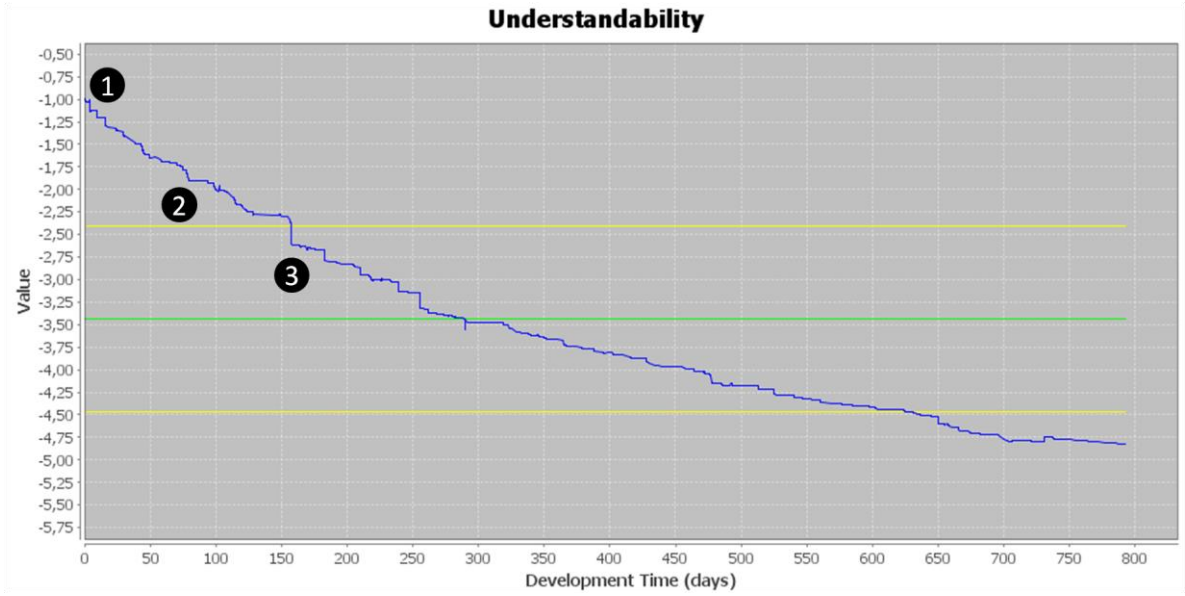


Figura 72: Gráfico de controle do atributo de qualidade entendimento do projeto IdUFF.

Concluindo, a complexidade total do projeto aumentou conforme a quantidade de linhas de código e quantidade de classes aumentou. Ainda assim, a complexidade média dos métodos não aumentou com o tempo. Além disso, a reusabilidade e o entendimento possuem comportamentos esperados para esses atributos de qualidade, que foram identificados nas regras do experimento apresentado na Seção 5.2. Consequentemente, no histórico do projeto analisado nesta seção, não é necessário se preocupar com o aumento da complexidade do projeto.

Entretanto, com base nas regras encontradas no experimento anterior e nos gráficos apresentados nesta seção, foi possível identificar padrões envolvendo a reusabilidade e o entendimento, e surgem duas questões: (i) se isso acontece para esses atributos nesse projeto, será que é possível encontrar padrões gerais e (ii) será que esses padrões estão de acordo com o que é indicado na literatura? O experimento apresentado na Seção 5.4 tem como objetivo explorar essas questões.

5.4 RELACIONAMENTOS GERAIS ENTRE MÉTRICAS

O terceiro objetivo da Ostra é auxiliar pesquisas em engenharia de software buscando padrões gerais envolvendo métricas. No experimento apresentado na Seção 5.3, ao observar as Figuras 71 e 72, fica claro que não é possível maximizar todos os atributos de qualidade do software durante seu desenvolvimento. Dessa forma, é importante para o gerente do projeto identificar quais métricas influenciam positivamente ou negativamente as outras, para, assim, escolher, com mais informações sobre a correlação entre elas, qual delas merece mais atenção e como a variação de uma influencia nas demais. A ferramenta disponibilizada pela Ostra para auxiliar na identificação do relacionamento entre as métricas e na priorização das métricas de acordo com a influência que umas exercem sobre as outras é a tabela de comportamento, explicada no Capítulo 3.

No experimento da Seção 5.2, foram apresentadas regras particulares de um projeto. Essa é uma forma de se entender como as métricas se relacionam e são influenciadas por outros fatores, como a quantidade de arquivos alterados, o momento do *commit* ou o desenvolvedor responsável. Entretanto, analisar vários projetos através de suas regras é muito trabalhoso e demanda muito tempo. Para auxiliar a análise dos padrões entre métricas, foi proposta a tabela de comportamento, que mostra de forma simplificada a relação entre pares de métricas encontradas nas regras mineradas. Essa proposta foi apresentada no Capítulo 3.

No Capítulo 2, foi mostrada, na Tabela 13, a influência que cada atributo de qualidade tem sobre os outros (WIEGERS, 2003). Nesta seção, é apresentado um experimento que

utiliza a tabela de comportamento da Ostra para avaliar empiricamente esses padrões apontados na literatura.

O experimento mostrado nesta seção tem como objetivo avaliar a capacidade da tabela de comportamento em auxiliar a identificação das variações das métricas em relação às demais. Esse objetivo é avaliado através da análise de um projeto, buscando padrões indicados na literatura e através da busca por padrões gerais à engenharia de software.

Esta seção está organizada em seis seções. Este experimento inicia apresentando na Seção 5.4.1 o mapeamento dos atributos de qualidade e métricas da Ostra para os utilizados na tabela de Wiegers (2003). Em seguida, na Seção 5.4.2, é mostrada a comparação do que é indicado na literatura e o encontrado no projeto IdUFF. Na Seção 5.4.3, é apresentado como foram construídas as tabelas de comportamento dos projetos com o conjunto de atributos de qualidade e métricas do mapeamento. Na Seção 5.4.5, é realizada uma análise quantitativa do que foi encontrado nos projetos e, na Seção 5.4.6, é apresentada uma análise dos padrões encontrados em grupos, de acordo com as características dos projetos.

5.4.1 MAPEAMENTO ENTRE WIEGERS E OSTRA

A Ostra e Wiegers (2003) trabalham com atributos de qualidade distintos. Dessa forma, foi necessário construir um mapeamento indicando quais atributos de qualidade são equivalentes. No Capítulo 2, foram apresentados os atributos de qualidade de Wiegers (2003) e de QMOOD (BANSIYA; DAVIS, 2002). Para identificar a equivalência entre o conjunto de métricas e atributos de qualidade de Wiegers (2003) e da Ostra, foram analisadas as definições de cada atributo de qualidade e métricas disponíveis na Ostra e comparadas com as definições dos atributos de qualidade de Wiegers (2003). Na Tabela 36, são apresentados os mapeamentos entres atributos de qualidade e métricas desses conjuntos. As siglas nessa tabela são as siglas referentes aos atributos de qualidade de QMOOD ou métricas que aparecem nas tabelas de comportamento criadas automaticamente pela Ostra.

Como mostrado na Tabela 36, três atributos de qualidade utilizados por Wiegers (2003) foram mapeados para atributos de qualidade de QMOOD, presentes na Ostra: flexibilidade, manutenibilidade e reusabilidade. A flexibilidade e a reusabilidade foram mapeadas para atributos de qualidade homônimos de QMOOD, pois suas definições são equivalentes. Entretanto, a manutebilidade foi mapeada parcialmente para extensibilidade, pois enquanto o atributo de qualidade de Wiegers (2003) aborda a correção de defeitos e modificação, o de QMOOD aborda apenas a incorporação de novas funcionalidades, ou seja, modificação.

Na Tabela 36, também são apresentados outros mapeamentos de atributos de qualidade de Wiegers (2003) para métricas presentes na Ostra. O atributo de qualidade integridade, por definição, é equivalente à propriedade de projeto encapsulamento de QMOOD, que é avaliada através da métrica *Data Access Metric*. Embora não exista uma métrica com definição igual à do atributo de qualidade testabilidade de Wiegers (2003), no conjunto de métricas da Ostra, a métrica Complexidade Ciclomática de McCabe está diretamente relacionada à testabilidade do código. Porém, a Complexidade Ciclomática de McCabe é inversamente proporcional à testabilidade, ou seja, quanto maior o valor dessa métrica, mais difícil é testar o código. Consequentemente, a testabilidade é mapeada como o inverso da Complexidade Ciclomática de McCabe.

Tabela 36: Mapeamento entre Wiegers (2003) e a Ostra.

Sigla	**Wiegers (2003)**		**Ostra**	
	Nome	Definição	Nome	Definição
Fle	Flexibilidade	Avalia quão fácil é adicionar novas funcionalidades ao software.	Flexibilidade	Característica que permite a incorporação de mudanças no projeto. É a habilidade do projeto de ser adaptada de maneira a prover capacidades relacionadas à funcionalidade.
Ext	Manutenibilidade	Mede a facilidade de corrigir defeitos ou modificar o software.	Extensibilidade	Refere-se à presença e utilização de propriedades no projeto que permitem a incorporação de novas funcionalidades.
Reu	Reusabilidade	Avalia o esforço necessário para converter um componente do software para ser utilizado em outra aplicação.	Reusabilidade	Representa a presença de propriedades de projeto orientado a objetos que permite que o projeto seja reaproveitado em um novo problema sem grande esforço.
DAM	Integridade	Avalia quão protegido está o software, ou seja, o quanto ele protege os dados e as funcionalidades de acessos não permitidos.	Propriedade de projeto: encapsulamento Métrica: *Data Access Metric*	Definido como o encapsulamento de dados e comportamento numa única estrutura. Em projetos OO, essa propriedade se refere às classes que previnem o acesso a métodos ou a dados, os tornando privados.
TCC	Testabilidade	Avalia a facilidade de verificar se a aplicação ou componentes dela estão funcionando corretamente com o objetivo de identificar erros.	Inverso da Complexidade Ciclomática de McCabe	Mede a quantidade de caminhos de execução independentes no código. Quanto maior essa métrica, mais difícil é testar o software (HENDERSON-SELLERS, 1995).

Com o mapeamento descrito anteriormente, foi construída uma tabela de comportamento modelo, representando os conceitos estabelecidos na literatura. Essa tabela é apresentada na Figura 73, com a cor verde indicando os comportamentos positivos (proporcionais), ou seja, aqueles em que a métrica da linha afeta positivamente a métrica da coluna, e a cor vermelha, os negativos (inversamente proporcionais), ou seja, aqueles em que a métrica da linha afeta negativamente a métrica da coluna.

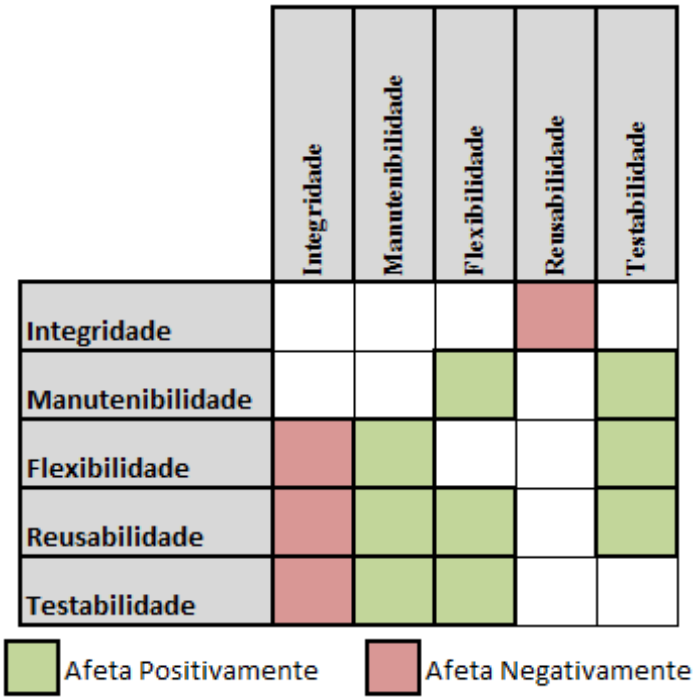


Figura 73: Tabela de comportamento indicada por Wiegers (2003).

5.4.2 COMPARAÇÃO COM UM PROJETO

Comparando a tabela de comportamento de um projeto, gerada pela Ostra, com os padrões indicados na tabela da Figura 73, um gerente de projetos pode verificar se seu projeto segue os padrões indicados pela literatura. Os pontos discordantes dessa comparação podem disparar investigações sobre os motivos e, eventualmente, ações corretivas: pode ser que o projeto não siga os padrões da literatura por causa de um problema ou porque eles não se aplicam a este projeto.

Embora a comparação com um gabarito seja a aplicação mostrada nessa seção, ela não é a única: pode-se utilizar a tabela de comportamento para se entender os efeitos colaterais da variação das métricas no projeto. Com a tabela de comportamento, o gerente do projeto pode identificar como as métricas variam em relação à variação das outras. Dessa forma, o gerente pode identificar, por exemplo, que o aumento do entendimento do projeto pode diminuir a

reusabilidade. Nesse exemplo, se o principal para o projeto é a reusabilidade, apesar de querer que o entendimento melhore, o gerente terá informações suficientes para entender que não poderá aumentar os dois atributos de qualidade e dará prioridade à reusabilidade, deixando de lado o entendimento. A tabela de comportamento pode ser interessante mesmo sem a utilização de um gabarito, pela sua capacidade de evidenciar o relacionamento entre as variações das métricas.

Para mostrar a utilização da tabela de comportamento no auxílio à busca de padrões indicados na literatura, é apresentada a comparação da tabela de comportamento do projeto IdUFF com a tabela gabarito gerada a partir do mapeamento da Ostra com Wiegers (2003). A tabela de comportamento é uma visualização das regras de associação encontradas na mineração de dados. Na mineração de dados, para construir a tabela de comportamento do IdUFF, foi utilizado suporte mínimo relativo de 0,1 (136 *commits*) e *lift* mínimo de 1,5. Na Figura 74, é mostrada a tabela de comportamento, que foi apresentada no Capítulo 3, gerada pela Ostra com esses parâmetros.

	DAM	Ext	Fle	Reu	TCC
DAM		S:0.10 C:0.69 L:2.66		S:0.09 C:0.64 L:1.86	
Ext	S:0.10 C:0.40 L:2.66		S:0.21 C:0.80 L:3.36	S:0.12 C:0.47 L:2.93	
Fle		S:0.21 C:0.88 L:3.36		S:0.11 C:0.47 L:2.89	
Reu	S:0.09 C:0.28 L:1.86	S:0.12 C:0.76 L:2.93	S:0.11 C:0.69 L:2.89		S:0.31 C:0.90 L:1.83
TCC				S:0.31 C:0.89 L:1.83	

Figura 74: Tabela de comportamento do IdUFF gerada pela Ostra.

Com a tabela de comportamento do IdUFF, mostrada na Figura 74, pode-se avaliar quais comportamentos indicados na literatura são encontrados neste projeto. Na Figura 75, é mostrada a comparação entre a tabela de comportamento do IdUFF e a tabela de comportamento gabarito. Nessa tabela, as células coloridas representam os comportamentos indicados pela literatura e as letras C e N indicam, respectivamente, para o projeto IdUFF, se

os comportamentos são confirmados ou negados. As células coloridas sem letras indicam que os comportamentos não foram encontrados na tabela gerada.

Pode-se reparar que alguns comportamentos indicados por Wiegers (2003) foram confirmados no IdUFF, mas outros foram negados. Quatro comportamentos esperados foram encontrados, três foram negados e seis não foram encontrados.

	Int	Man	Fle	Reu	1/ Test
Int				N	
Man			C		
Fle		C			
Reu	N	C	C		N
1/ Test					

Figura 75: Comparação da tabela de comportamento do IdUFF com o indicado por Wiegers (2003).

Os comportamentos que representam variações negativas, indicados por Wiegers (2003), que não foram encontrados no IdUFF, não são necessariamente indícios de problemas no projeto, pois nem sempre se pode maximizar todos os atributos de qualidade ao mesmo tempo. Os comportamentos negativos não encontrados (células na cor vermelha com a letra "N") representam que quando algumas métricas aumentam, outras devem diminuir. Porém, no IdUFF, foi encontrado que quando as métricas da linha aumentam as da coluna também aumentam, ou seja, comportamentos positivos. Um exemplo é a relação entre a integridade (Int) e a reusabilidade (Reu), que deveria ser negativa, mas foi encontrado indício de ser positiva, ou seja, quando a integridade aumenta, no IdUFF, a reusabilidade também aumenta.

Consequentemente, ao se analisar a tabela de comportamento gerada para o projeto IdUFF, surge uma questão: "os comportamentos esperados que foram negados são problemas ou o indicado por Wiegers (2003) não se aplica a projetos deste tipo?" Para responder, na próxima seção, são analisados alguns projetos em busca de padrões gerais.

5.4.3 FORMAÇÃO DAS TABELAS DE COMPORTAMENTO

Neste experimento, foram utilizados 16 projetos que possuem, cada um, mais de 100 *commits* compilando[15]. Informações sobre os 16 projetos são mostradas na Tabela 37.

[15] Foram medidos cerca de 150 projetos durante a preparação deste experimento, porém os projetos com menos de 100 *commits* compilando não foram considerados. Alguns dados sobre esses projetos são mostrados no Apêndice A.

Nesse experimento, apenas foram considerados os *commits* que resultam em revisões que compilam desses projetos. Assim como no experimento da Seção 5.2, a última coluna da Tabela 37 indica o número da revisão mais recente considerada no experimento. Essa informação é necessária para possibilitar a repetição do experimento.

Desses projetos, nove são de código aberto e os outros não. Os de código aberto estão marcados com um * (asterisco) ao lado do nome.

Diferentemente do processo de obtenção da tabela de comportamento do IdUFF, mostrada na Seção 5.4.2, com o objetivo de se encontrarem mais padrões, o processo de obtenção das tabelas de comportamento construídas para buscar padrões gerais, com os 16 projetos, foi menos restritivo. Dessa forma, para um comportamento ser relevante, ele deve aparecer pelo menos dez vezes na base de dados e o precedente deve aumentar em pelo menos 10% a chance de ocorrer o consequente. Assim, foram utilizados suporte mínimo absoluto de 10 instâncias e *lift* mínimo de 1,1.

Tabela 37: Projetos utilizados no experimento geral.

Projeto	Quantidade de Desenvolvedores	Quant. De Artefatos	LOC (Java)	Revisões que compilam	Total de revisões	Ultima revisão medida
Acadêmico Pós Graduação Core	9	221	10.413	102	142	4.196
IdUFF	31	1.068	151.621	1.355	1.509	22.695
Maven Changes Plugin*	10	686	8.188	230	259	1.140.265
Maven GWT Plugin*	6	463	10.816	252	312	14.772
Maven Javacc Plugin*	6	759	6.575	129	136	10.774
Maven Javadoc Plugin*	21	795	17.775	248	329	1.232.525
Maven Native Plugin*	4	1.305	9.403	148	163	13.690
Maven Nbm Plugin*	5	1.433	6.916	210	230	14.751
Maven PMD Plugin*	14	806	2.610	104	110	1.159.144
Maven Project Info Reports Plugin*	13	32	8.770	160	240	1.152.589
Maven Shade Plugin	12	84	6.650	102	104	1.300.217
Maven Versions Plugin*	6	143	17.537	182	198	13.376
Monitoria Core	6	143	10.246	133	445	23.628
Oceano Core	6	418	29.181	214	228	1.075
Oceano Web	6	117	10.984	117	179	1.077
Publico Core	21	117	9.183	127	134	21.633

5.4.4 SUMARIZAÇÃO DAS TABELAS DOS PROJETOS

Para cada projeto, foi construída uma tabela de comportamento com os parâmetros e atributos descritos anteriormente, na Seção 5.4.3. Em seguida, foi contabilizada a quantidade de vezes que apareceram os comportamentos em cada projeto. As tabelas de comportamento geradas para cada projeto estão registradas no Apêndice C.

O comportamento positivo (cor verde, azul ou roxo), significa que um atributo de qualidade influencia positivamente o outro. Analogamente, o comportamento negativo (cor vermelho, amarelo, marrom-escuro ou mostarda), significa que um atributo de qualidade influencia negativamente o outro. Entretanto, quando não é possível definir o comportamento (cor cinza), ou seja, concomitantemente ocorrem os dois comportamentos, foi considerado o comportamento com maior *lift*. Por exemplo, tem-se que para a integridade e a flexibilidade, foram encontradas duas regras: quando a integridade aumenta, a flexibilidade diminui, com *lift* 1,2, e quando a integridade aumenta, a flexibilidade aumenta, com *lift* 3,5. Nesse caso, será considerado o comportamento positivo, indicando que quando a integridade aumenta, a flexibilidade também aumenta, pois essa regra teve maior *lift* dentre as encontradas.

Após a construção da tabela de comportamento de cada projeto, a quantidade de vezes que o comportamento positivo e negativo apareceu nas tabelas de comportamento de cada projeto foi contabilizada. Na Figura 76, é apresentada a tabela que sumariza esses dados. O significado das cores da tabela segue o indicado na Figura 73, o número 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.

Para definir os comportamentos da tabela da Figura 76, quando houve indicação de dois tipos de comportamento pelos projetos, prevaleceu o comportamento indicado pela maioria dos projetos. Consequentemente, a maior quantidade de projetos definiu a cor da célula e o comportamento, enquanto a minoria (discordâncias) aparece entre parênteses. Por exemplo, na Figura 76, é mostrado que o comportamento entre integridade e manutenibilidade é positivo, ou seja, quando a integridade aumenta, a manutenibilidade também aumenta. Esse comportamento foi confirmado por oito projetos e negado por quatro.

Resultado GERAL	Integridade	Manutenibilidade	Flexibilidade	Reusabilidade	Testabilidade
Integridade		10 (-5)	15 0	13 (-2)	10 (-3)
Manuteniblidade	10 (-5)		15 0	12 (-3)	7 (-7)
Flexibilidade	15 0	15 0		14 (-1)	8 (-6)
Reusabilidade	13 (-2)	12 (-3)	14 (-1)		11 (-2)
Testabilidade	10 (-3)	7 (-7)	8 (-6)	11 (-2)	

Figura 76: Contabilização da ocorrência dos comportamentos nos 16 projetos.

5.4.5 ANÁLISE GERAL DO RESULTADO

A quantidade de vezes que cada comportamento indicado pela literatura apareceu nos 16 projetos é apresentada na Figura 77. O número positivo (superior) indica a quantidade de projetos que confirmam o comportamento, enquanto o número negativo (inferior) indica a quantidade de projetos que nega o comportamento. As células com contorno em negrito representam o comportamento apontado pela literatura. O significado das cores segue o indicado na Figura 73.

Observando a Figura 77, pode-se notar que alguns dos comportamentos apontados pela literatura foram identificados nos projetos, os quais são mostrados na Tabela 38. Entretanto, um dos padrões indicados na literatura não foi confirmado pelos projetos analisados e é mostrado na Tabela 39.

No entanto, outros comportamentos foram confirmados por alguns projetos e negados por outros com pouca diferença. Nesse caso, não é possível chegar a conclusões. Na Tabela 40, são mostrados esses comportamentos e a quantidade de projetos que os confirmam ou negam.

Comparação com a literatura	Integridade	Manutenibilidade	Flexibilidade	Reusabilidade	Testabilidade
Integridade				2 (-13)	
Manuteniblidade			15 0		7 (-7)
Flexibilidade	0 (-15)	15 0			8 (-6)
Reusabilidade	2 (-13)	12 (-3)	14 (-1)		11 (-2)
Testabilidade	10 (-3)	7 (-7)	8 (-6)		

Figura 77: Comportamentos indicados pela literatura e encontrados nos 16 projetos.

Tabela 38: Padrões da literatura não negados pelos projetos.

Comportamento	Confirmações	Negações
A manutenibilidade afeta positivamente a flexibilidade	15	0
A flexibilidade afeta positivamente a manutenibilidade	15	0

Tabela 39: Padrões da literatura não confirmados pelos projetos.

Comportamento	Confirmações	Negações
A flexibilidade afeta negativamente a integridade	0	15

Tabela 40: Padrões negados e confirmados pelos projetos.

Comportamento	Confirmações	Negações
A reusabilidade afeta positivamente a flexibilidade	14	1
A reusabilidade afeta positivamente a manutenibilidade	12	3
A testabilidade afeta positivamente a manutenibilidade	7	7
A manutenibilidade afeta positivamente a testabilidade	7	7
A flexibilidade afeta positivamente a testabilidade	8	6
A testabilidade afeta positivamente a flexibilidade	8	6
A reusabilidade afeta positivamente a testabilidade	11	2
A integridade afeta negativamente a reusabilidade	2	13
A reusabilidade afeta negativamente a integridade	2	13

Além dos comportamentos indicados por Wiegers (2003), foram encontrados outros comportamentos, mostrados na Figura 78. Dentre eles, se destacam os comportamentos apresentados na Tabela 41.

Os comportamentos apresentados na Tabela 38 foram apresentados pela literatura e não foram negados por nenhum dos projetos, pelo contrário, foram encontrados em 15 dos 16 projetos analisados. Analogamente, o comportamento apresentado na Tabela 39 não foi confirmado por nenhum dos projetos, mas foi negado por 15 dos 16 projetos analisados. Por outro lado, os comportamentos apresentados na Tabela 40 foram negados e confirmados pelos padrões encontrados nos projetos. Consequentemente, existem indícios de que os comportamentos apresentados nas Tabelas 38 e 39 são gerais.

Comportamentos não indicados pela literatura	Integridade	Manutenibilidade	Flexibilidade	Reusabilidade	Testabilidade
Integridade			15 0		10 (-3)
Manuteniblidade				12 (-3)	
Flexibilidade				14 (-1)	
Reusabilidade					
Testabilidade				11 (-2)	

Figura 78: Comportamentos encontrados nos projetos que não são indicados por Wiegers (2003).

Dos comportamentos identificados nos projetos, que não são indicados na literatura, o único que não foi negado por nenhum projeto, mas aprovado por 15 dos 16, foi: "a integridade afeta positivamente a flexibilidade". Os outros comportamentos apresentados na Tabela 41 foram negados e confirmados pelos projetos analisados. Dessa forma, o comportamento supracitado pode ser um padrão geral a qualquer projeto, mas os outros, provavelmente, se aplicam somente a projetos de determinados tipos.

Tabela 41: Padrões encontrados nos projetos que não são indicados por Wiegers (2003).

Comportamento	Confirmações	Negações
A integridade afeta positivamente a flexibilidade	15	0
A integridade afeta positivamente a testabilidade	10	3
A manutenibilidade afeta positivamente a reusabilidade	12	3
A flexibilidade afeta positivamente a reusabilidade	14	1
A testabilidade afeta positivamente a reusabilidade	11	2

5.4.6 ANÁLISE DO RESULTADO POR GRUPOS

Enquanto nos comportamentos das Tabelas 38 e 39 apenas foram encontrados indícios de negação ou confirmação nos projetos, os comportamentos da Tabela 40 foram negados e confirmados concomitantemente pelo conjunto de projetos analisados. Para diferenciar os projetos que confirmam e negam os comportamentos da Tabela 40, esses projetos foram divididos em grupos de acordo com as características mostradas na Tabela 42.

Tabela 42: Caracterização dos 16 projetos utilizados nos experimentos.

Projeto	Utilização	Empacotamento	Tipo	Tamanho da equipe	Tamanho do projeto
IdUFF	Interface Web	Jar ou War[16]	Proprietário	Grande	Grande
Oceano Core	Biblioteca	Jar	Acadêmico	Pequena	Médio
Público Core	Biblioteca	Jar	Proprietário	Grande	Pequeno
Maven GWT Plugin	Plugin Maven	Jar	Código aberto	Pequena	Médio
Maven Nbm Plugin	Plugin Maven	Jar	Código aberto	Pequena	Pequeno
Maven Project Info Reports Plugin	Plugin Maven	Jar	Código aberto	Média	Médio
Maven Versions Plugin	Plugin Maven	Jar	Código aberto	Pequena	Médio
Maven Changes Plugin	Plugin Maven	Jar	Código aberto	Média	Médio
Maven PMD Plugin	Plugin Maven	Jar	Código aberto	Média	Pequeno
Maven Native Plugin	Plugin Maven	Jar	Código aberto	Pequena	Pequeno
Oceano Web	Interface Web	War	Acadêmico	Pequena	Médio
Maven Javacc Plugin	Plugin Maven	Jar	Código aberto	Pequena	Pequeno
Maven Javadoc Plugin	Plugin Maven	Jar	Código aberto	Grande	Médio
Acadêmico Pós Graduação Core	Biblioteca	Jar	Proprietário	Pequena	Médio
Monitoria Core	Biblioteca	Jar	Proprietário	Pequena	Médio
Maven Shade Plugin	Plugin Maven	Jar	Código Aberto	Média	Pequeno

[16] O IdUFF é um projeto de interface web, mas que também é empacotado como jar para disponibilizar serviços para outros projetos acadêmicos. Porém, sua principal utilização e direcionamento no desenvolvimento é como um projeto de interface web e utilização direta pelo usuário.

Na Tabela 42, os projetos foram caracterizados de acordo com a utilização, empacotamento do software executável, tipo do projeto, tamanho da equipe e tamanho do projeto em linhas de código. A utilização do projeto foi caracterizada como: interface web, biblioteca ou *plugin* do Maven. O empacotamento é o formato do software compilado: jar ou war. O tipo pode ser proprietário, de código aberto ou acadêmico. A classificação de tamanho da equipe foi realizada considerando os intervalos descritos na Tabela 43 e a classificação do tamanho do projeto, considerando os intervalos da Tabela 44.

Tabela 43: Caracterização do tamanho da equipe.

Tamanho da equipe	Quantidade de pessoas
Pequena	Menos de 10 desenvolvedores
Média	Entre 10 e 15 desenvolvedores
Grande	Mais de 15 desenvolvedores

Tabela 44: Caracterização do tamanho do projeto.

Tamanho do projeto	Quantidade de linhas de código
Pequeno	Menos de 10 mil linhas de código
Médio	Entre 10 e 100 mil linhas de código
Grande	Mais de 100 mil linhas de código

A contagem dos comportamentos foi agrupada de acordo com a caracterização da Tabela 42. Nas Figuras 79-82, as células com contorno mais grosso são aquelas dos comportamentos indicados por Wiegers (2003), enquanto nas células coloridas sem contorno são aqueles apresentados os comportamentos identificados nos grupos. Quando houve conflito dentro do grupo, não é mostrado o comportamento, salvo os indicados por Wiegers (2003). O número superior indica quantos projetos confirmaram o comportamento indicado na célula e o número inferior, entre parênteses, os que negaram o comportamento.

5.4.6.1 ANÁLISE DE ACORDO COM O TAMANHO DA EQUIPE

Na Figura 79, são apresentadas três tabelas com comportamento, uma para cada tamanho de equipe: pequena, média e grande. Foram considerados nove projetos com equipes pequenas, quatro com equipes médias e três com equipes grandes.

Dos projetos de equipe pequena, quatro comportamentos indicados por Wiegers (2003) foram confirmados por unanimidade e dois foram negados por unanimidade. Dos projetos de equipes médias, três comportamentos foram apenas confirmados e um foi apenas negado. Dos projetos de equipe grande, oito comportamentos foram confirmados por unanimidade e três foram negados por unanimidade.

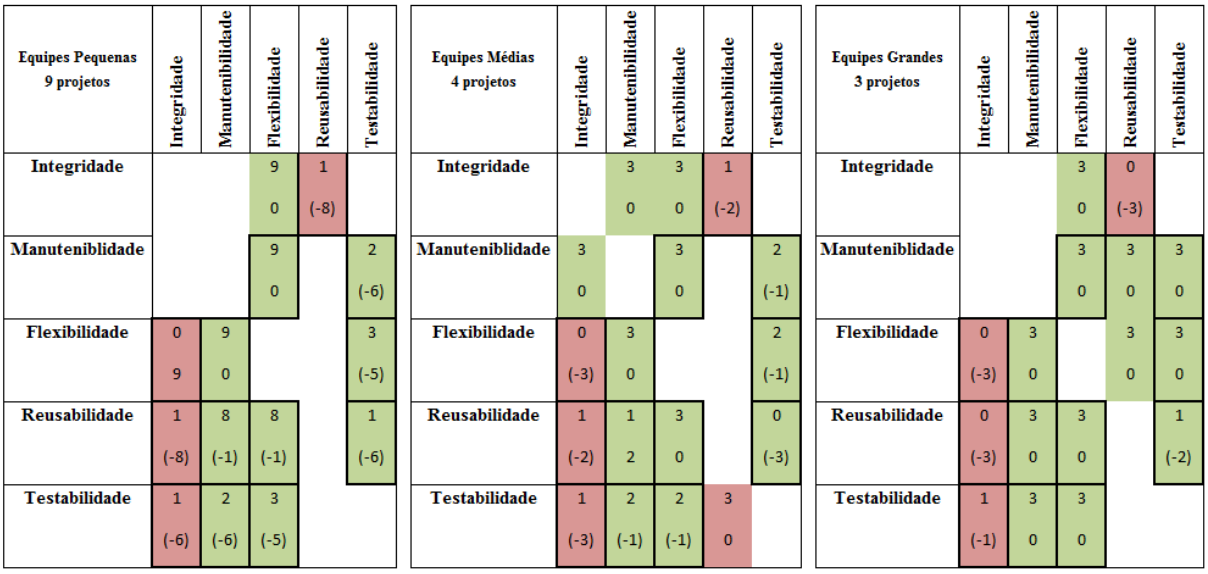


Figura 79: Tabelas de comportamento de acordo com o tamanho da equipe.

Além dos comportamentos indicados por Wiegers (2003), também se pôde ter indícios de outros comportamentos. Esses comportamentos são mostrados na Tabela 45.

Os comportamentos apresentados na Tabela 45 não foram negados em nenhum dos projetos analisados de acordo com os grupos. Por exemplo, foram encontrados indícios de que "a integridade afeta positivamente a flexibilidade" para projetos de todos os tamanhos. Porém, "a integridade afeta positivamente a manutenibilidade" verificou-se apenas em projetos com equipes de tamanho médio. Consequentemente, esses comportamentos podem ser gerais, mas aplicados a cada um dos grupos apenas.

Tabela 45: Comportamentos não indicados pela literatura que apareceram na análise por tamanho de equipe.

Comportamento	Encontrado em equipes
A integridade afeta positivamente a flexibilidade	Pequenas, Médias e Grandes
A integridade afeta positivamente a manutenibilidade	Médias
A manuteniblidade afeta positivamente a integridade	Médias
A testabilidade afeta negativamente a reusabilidade	Médias
A manuteniblidade afeta positivamente a reusabilidade	Grandes
A flexibilidade afeta positivamente a reusabilidade	Grandes

5.4.6.2 ANÁLISE DE ACORDO COM A UTILIZAÇÃO DO PROJETO

Na Figura 80, são apresentadas três tabelas com comportamentos, uma para cada grupo, de acordo com a utilização do projeto: projeto web, *plugin* do Maven e biblioteca. Foram considerados dois projetos web, nos quais três padrões de Wiegers (2003) foram confirmados por unanimidade e um negado por unanimidade. Foram considerados dez

projetos que são *plugins* do Maven, nos quais três comportamentos indicados por Wiegers (2003) foram confirmados e um foi negado. Finalmente, foram considerados quatro subprojetos utilizados como bibliotecas por outros projetos, nos quais oito comportamentos esperados foram confirmados por unanimidade e três foram negados por unanimidade.

Além dos comportamentos indicados por Wiegers (2003), também se pôde ter indícios de outros comportamentos. Esses comportamentos são mostrados na Tabela 46.

Os comportamentos apresentados na Tabela 46 não foram negados em nenhum dos projetos analisados de acordo com a utilização dos projetos. Por exemplo, foram encontrados indícios de que "a integridade afeta positivamente a flexibilidade" para projetos de todos os grupos de utilização. Porém, "a testabilidade afeta negativamente a reusabilidade" verificou-se apenas em projetos web e subprojetos. Consequentemente, esses comportamentos podem ser gerais a projetos desses grupos.

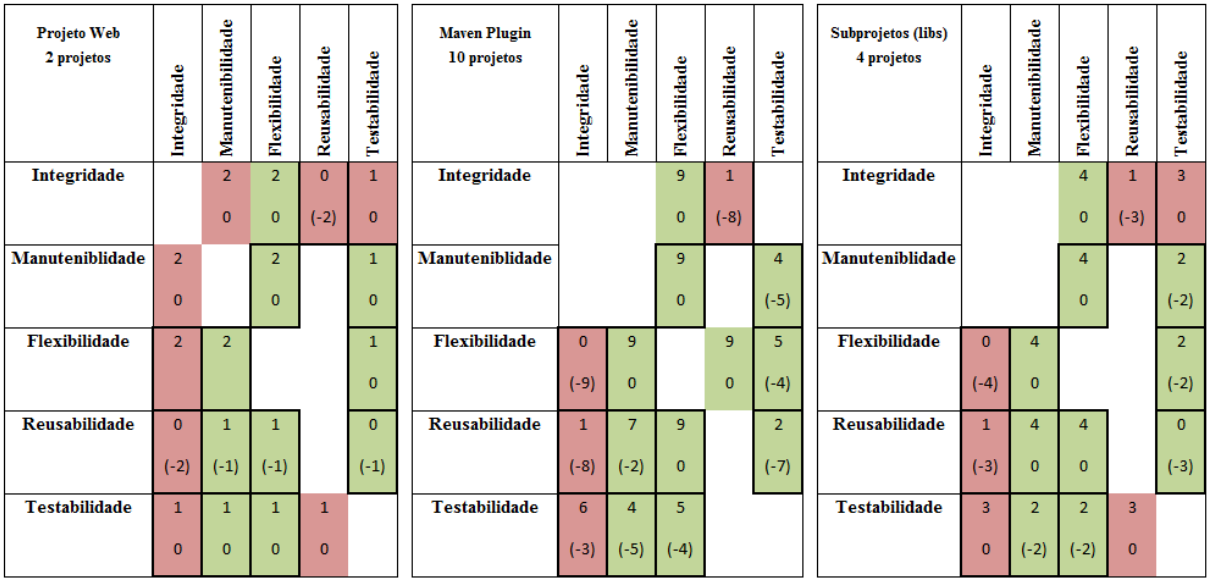


Figura 80: Tabelas de comportamento de acordo com a utilização.

Tabela 46: Comportamentos não indicados pela literatura que apareceram na análise por utilização do projeto.

Comportamento	Encontrado em projetos
A integridade afeta negativamente a manutenibilidade	Projetos web
A integridade afeta positivamente a flexibilidade	Projetos web, *Plugin* do Maven e Subprojetos
A integridade afeta negativamente a testabilidade	Projetos web e Subprojetos
A manutenibilidade afeta negativamente a integridade	Projetos web
A testabilidade afeta negativamente a reusabilidade	Projetos web e Subprojetos
A flexibilidade afeta positivamente a reusabilidade	*Plugin* do Maven

5.4.6.3 ANÁLISE DE ACORDO COM O TIPO DO PROJETO

Na Figura 81, são mostradas três tabelas com comportamentos identificados nos grupos de acordo com o tipo do projeto: proprietário, código aberto e acadêmico. Foram considerados quatro projetos proprietários, nos quais oito comportamentos de Wiegers (2003) foram confirmados e quatro foram negados. Foram considerados dez projetos de código aberto, nos quais três comportamentos foram confirmados e um foi negado. Nos dois projetos acadêmicos analisados, sete comportamentos de Wiegers (2003) foram confirmados e três negados.

Além dos comportamentos indicados por Wiegers (2003), também se pôde ter indícios de outros comportamentos. Esses comportamentos são mostrados na Tabela 47. Da mesma forma que os comportamentos encontrados para os grupos anteriores, para os comportamentos apresentados nessa tabela, foram encontrados indícios de que eles sejam gerais a projetos dos grupos analisados, pois os mesmos, não foram negados por nenhum dos projetos na analise por grupos.

Proprietários 4 projetos	Integridade	Manutenibilidade	Flexibilidade	Reusabilidade	Testabilidade
Integridade			4 0	1 (-3)	3 0
Manuteniblidade			4 0	4 0	2 (-2)
Flexibilidade	0 (-4)	4 0		4 0	2 (-2)
Reusabilidade	1 (-3)	4 0	4 0		0 (-4)
Testabilidade	3 0	2 (-2)	2 (-2)	4 0	

Código Aberto 10 projetos	Integridade	Manutenibilidade	Flexibilidade	Reusabilidade	Testabilidade
Integridade			9 0	1 (-8)	
Manuteniblidade			9 0		4 (-5)
Flexibilidade	0 (-9)	9 0		9 0	5 (-4)
Reusabilidade	1 (-8)	7 (-2)	9 0		2 (-7)
Testabilidade	6 (-3)	4 (-5)	5 (-4)		

Acadêmico 2 projetos	Integridade	Manutenibilidade	Flexibilidade	Reusabilidade	Testabilidade
Integridade		2 0	1 0	0 (-1)	1 0
Manuteniblidade	2 0		2 0		1 0
Flexibilidade	0 (-2)	2 0			1 0
Reusabilidade	0 (-2)	1 (-1)	1 (-1)		0 0
Testabilidade	1 0	1 0	1 0		

Figura 81: Tabelas de comportamento de acordo com o tipo do projeto.

Tabela 47: Comportamentos não indicados pela literatura que apareceram na análise por tipo do projeto.

Comportamento	Encontrado em projetos
A integridade afeta negativamente a manutenibilidade	Acadêmicos
A integridade afeta positivamente a flexibilidade	Proprietários, Código aberto e Acadêmicos
A integridade afeta negativamente a testabilidade	Proprietários e Acadêmicos
A manutenibilidade afeta negativamente a integridade	Acadêmicos
A testabilidade afeta negativamente a reusabilidade	Proprietários
A flexibilidade afeta positivamente a reusabilidade	Proprietários e Código Aberto

5.4.6.4 ANÁLISE DE ACORDO COM O TAMANHO DO PROJETO

Três grupos, definidos a partir do tamanho dos projetos, foram considerados: pequenos, médios e grandes. Com base nesses grupos foram geradas as tabelas apresentadas na Figura 82. Foram considerados seis projetos pequenos, nove projetos médios e apenas um projeto grande. No grupo de projetos pequenos, podem-se encontrar indícios que confirmam quatro comportamentos indicados por Wiegers (2003) e negam dois. Nos projetos de tamanho médio, três comportamentos foram confirmados e dois negados. No grupo de tamanho grande, nove comportamentos foram confirmados e quatro foram negados.

Projeto Pequeno 6 projetos	Integridade	Manutenibilidade	Flexibilidade	Reusabilidade	Testabilidade
Integridade		6 0	6 0	1 (-5)	
Manuteniblidade	6 0		6 0		3 (-3)
Flexibilidade	0 (-6)	6 0		6 0	4 (-2)
Reusabilidade	1 (-5)	4 (-2)	6 0		6 0
Testabilidade	3 (-1)	3 (-3)	4 (-2)	6 0	

Projetos Médios 9 projetos	Integridade	Manutenibilidade	Flexibilidade	Reusabilidade	Testabilidade
Integridade			8 0	1 (-7)	
Manuteniblidade			8 0		3 (-4)
Flexibilidade	0 (-8)	8 0			3 (-4)
Reusabilidade	1 (-7)	7 (-1)	7 (-1)		2 (-4)
Testabilidade	6 (-2)	3 (-4)	3 (-4)		

Projetos Grandes 1 projeto	Integridade	Manutenibilidade	Flexibilidade	Reusabilidade	Testabilidade
Integridade		1 0	1 0	0 (-1)	1 0
Manuteniblidade	1 0		1 0	1 0	1 0
Flexibilidade	0 (-1)	1 0		1 0	1 0
Reusabilidade	0 (-1)	1 0	1 0		0 (-1)
Testabilidade	1 0	1 0	1 0	1 0	

Figura 82: Tabelas de comportamento de acordo com o tamanho do projeto em LOC.

Além dos comportamentos indicados por Wiegers (2003), também se pôde ter indícios de outros comportamentos. Esses comportamentos são mostrados na Tabela 48.

Os comportamentos apresentados na Tabela 48 não foram negados em nenhum dos projetos analisados de acordo com o tamanho dos projetos. Por exemplo, foram encontrados

indícios de que "a integridade afeta positivamente a flexibilidade" para projetos de todos os grupos de acordo com os tamanhos dos projetos. Porém, "a manutenibilidade afeta negativamente a integridade" verificou-se apenas em projetos grandes. Consequentemente, esses comportamentos podem ser gerais a projetos desses grupos.

Tabela 48: Comportamentos não indicados pela literatura que apareceram na análise por tamanho do projeto.

Comportamento	Encontrado em projetos
A integridade afeta positivamente a manutenibilidade	Pequenos
A integridade afeta positivamente a flexibilidade	Pequenos, Médios e Grandes
A integridade afeta negativamente a manutenibilidade	Grandes
A integridade afeta negativamente a testabilidade	Grandes
A manutenibilidade afeta positivamente a integridade	Pequenos
A manutenibilidade afeta negativamente a integridade	Grandes
A manutenibilidade afeta positivamente a reusabilidade	Grandes
A testabilidade afeta negativamente a reusabilidade	Pequenos e Grandes
A flexibilidade afeta positivamente a reusabilidade	Pequenos e Grandes

5.5 AMEAÇAS À VALIDADE DOS EXPERIMENTOS

Neste estudo, devem ser consideradas algumas ameaças à validade dos experimentos. Ameaça à validade é aquilo que pode limitar ou invalidar as observações do estudo.

No experimento da Seção 5.2, a capacidade da abordagem em obter informações sobre as modificações que ocorreram durante a evolução do projeto foi avaliada com quatro projetos. No experimento da Seção 5.3, a capacidade de monitorar a qualidade do projeto foi avaliada com apenas um projeto. No experimento da Seção 5.4, a capacidade de identificar relacionamentos entre métricas foi avaliada com 16 projetos. A pequena quantidade de projetos utilizados nos experimentos impede que conclusões fortes sejam realizadas em relação ao sucesso da abordagem em alcançar seus objetivos, fornecendo apenas indícios do seu sucesso. Para que sejam realizadas conclusões com maior embasamento, são necessárias avaliações com uma maior quantidade e diferentes tipos de projetos.

A complexidade na definição dos valores mínimos para as medidas de interesse das regras de associação pode ter influenciado nos comportamentos identificados no experimento da Seção 5.4. Dessa forma, podem ter sido descartadas regras por não satisfazerem os valores mínimos para as medidas de interesse. Consequentemente, alguns comportamentos podem não ter aparecido, já que as regras que provem indícios para eles foram descartadas, o que influencia nas observações realizadas. A definição dos valores mínimos para essas medidas é

complexa, pois não se podem utilizar valores muito baixos, de forma que, nesse caso, praticamente, qualquer padrão se torna interessante. Por outro lado, valores muito altos descartam quase todos os padrões, à medida que estabelecem que, para serem interessantes, os padrões devem ocorrer em muitas instâncias. Para tentar reduzir esse efeito, o suporte utilizado foi absoluto e o lift mínimo foi pouco superior a 1 (valores maiores que 1 para o *lift*, indicam que o precedente possui influência positiva na chance do consequente ocorrer).

Se as dependências dos projetos não estiverem disponíveis no momento da medição, a compilação das versões falha, impossibilitando a medição de algumas métricas. Como as versões dos projetos foram compiladas e medidas automaticamente, não se sabe se as versões não compilaram por erros no projeto, falha na configuração das dependências ou não fornecimento das dependências. Para evitar esse problema, as revisões que não compilam foram descartadas do experimento da Seção 5.4 e da base B do experimento da Seção 5.2. Porém, no experimento da Seção 5.2, foram consideradas todas as versões na base A e isso pode ter influenciado os padrões envolvendo a não compilação do projeto.

A qualidade da análise da abordagem é limitada diretamente pelas métricas e atributos de qualidade utilizados. Dessa forma, as métricas e atributos de qualidade utilizados nos experimentos podem ter influenciado as análises realizadas. Foi utilizado apenas um modelo de mapeamento de métricas para atributos de qualidade (BANSIYA; DAVIS, 2002), pois, até onde foi pesquisado, é o único modelo que define através de fórmulas esse mapeamento. Seria interessante utilizar outros modelos para avaliar a capacidade da abordagem proposta em analisar a evolução do software.

No experimento da Seção 5.4, foi realizado um mapeamento entre Wiegers (2003) e o conjunto de métricas da Ostra. Esse mapeamento foi realizado de acordo com a definição das métricas e atributos de qualidade de forma subjetiva e apesar de o mapeamento ter sido validado pelo autor deste trabalho e pelos orientadores, ele pode conter falhas. Consequentemente, a busca por padrões da literatura nos projetos e as análises realizadas com base no mapeamento podem ter sido comprometidas. Entretanto, ainda que a comparação dos comportamentos encontrados nos projetos com a literatura esteja comprometida, a ocorrência dos padrões nos projetos é válida.

5.6 CONSIDERAÇÕES FINAIS

Com o experimento a partir de quatro projetos realizado neste capítulo, foram explorados os tipos de informações que a abordagem proposta é capaz de evidenciar. As informações descobertas na mineração de dados em forma de regras de associação podem

encontrar indícios de comportamentos benéficos ou maléficos à evolução do projeto. Apesar de encontrar indícios das informações históricas de forma simples, no formato de regras, a validação de algumas regras não é tão elementar como de outras, principalmente as regras grandes, com mais de três elementos.

Existem regras que podem ser facilmente entendidas e explicadas, como a nona regra negativa do projeto Maven GWT Plugin, que mostra que, "quando Sansa faz alterações, a chance de não compilar é de 27%". Essa é uma regra que mostra o comportamento de um desenvolvedor informando a chance de seus *commits* compilarem. Se essa chance for muito baixa em relação ao resto da equipe, alguma ação pode ser tomada, como oferecer um treinamento ou cobrar mais atenção.

Entretanto, existem regras que são mais difíceis de serem explicadas e entendidas, como, por exemplo, a sétima regra negativa do projeto Maven Javadoc Plugin, "se o *commit* é na terça-feira, então não compila", pois necessita de mais informações acerca da metodologia ou práticas do dia-a-dia do desenvolvimento para ser aproveitada em sua totalidade. Por outro lado, a sétima regra negativa do projeto IdUFF, "se o commit é na sexta-feira, então não compila", bem parecida com esta, mostra que "na sexta-feira, a chance de compilar diminui" e pôde ser explicada e entendida mais profundamente. Isso se deu, pois o autor deste trabalho participou do desenvolvimento desse projeto e tem conhecimento sobre a equipe e metodologia de desenvolvimento utilizada.

No segundo experimento, mostrado na Seção 5.3, foi avaliada a utilização dos gráficos históricos no monitoramento da qualidade do projeto. Com esses gráficos, pode-se identificar *commits* fora do padrão de variação das métricas que devem ser observados. Acompanhando esses gráficos, pode-se responder à variação da qualidade do projeto com informações quantitativas.

Além das regras e dos gráficos de controle, pode-se extrair informações do histórico do projeto através das tabelas de comportamento. Na Seção 5.4, foi mostrada a utilização dessas tabelas com um projeto, contrastando seu comportamento com o indicado pela literatura, e com vários projetos, em busca de padrões gerais. Com as comparações apresentadas, foi possível identificar indícios de confirmação e negação dos padrões da literatura. Com a análise dos projetos em grupos, foi possível identificar padrões que funcionam para alguns grupos, mas não para outros. Porém para atingir análises mais conclusivas, deve-se realizar esse experimento com um número maior de projetos.

Conclui-se que a abordagem é capaz de identificar informações interessantes sobre a evolução do software e que as informações descobertas na forma de regras podem aumentar o

conhecimento sobre a evolução do software. A abordagem proposta permite ainda acompanhar os atributos de qualidade que são interessantes para a equipe de desenvolvimento do projeto durante sua evolução, possibilitando reagir, ao identificar nos gráficos de controle ou tabelas de comportamento, padrões inesperados e indesejados.

CAPÍTULO 6 – CONCLUSÃO

6.1 EPÍLOGO

A qualidade do software muda constantemente durante a evolução de um projeto de software. Dessa forma, visando que a qualidade seja mantida, as métricas e os atributos de qualidade que descrevem o software devem ser monitorados. Entender sobre os fatores que afetam a qualidade do software é necessário para que se possa controlá-la.

Neste trabalho foi proposta uma abordagem que tem como objetivos: (1) fornecer informações 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. Foi desenvolvido um protótipo que implementa a abordagem proposta e foram realizados experimentos para avaliar a abordagem Ostra, como apresentados, respectivamente, nos Capítulos 4 e 5.

Neste capítulo, são apresentadas as contribuições, as limitações da proposta e trabalhos futuros. Na Seção 6.2, são apresentadas as contribuições e, na Seção 6.3, são apresentadas as limitações do trabalho. Finalmente, na Seção 6.4, são apresentadas sugestões de trabalhos futuros.

6.2 CONTRIBUIÇÕES

As principais contribuições deste trabalho são:

- Utilização de 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;
- Representação, através da tabela de comportamento, de como as métricas se relacionam;
- 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
- Disponibilização para o Grupo de Evolução e Manutenção de Software (GEMS) da UFF, assim como para grupos parceiros em projetos de pesquisa, uma infraestrutura para a realização de trabalhos futuros. O protótipo implementado fornece infraestrutura para: (1) acesso a repositórios de controle de versão,

(2) construção de software usando sistemas de gerenciamento de construção, (3) medição do histórico de um projeto armazenado no sistema de controle de versão e (4) mineração de regras de associação sobre os dados recuperados do repositório.

As principais iniciativas para a validação da abordagem proposta foram:

- Realização de experimentos, com quatro projetos, que levantam indícios sobre a capacidade da abordagem proposta em obter informações relevantes sobre a evolução do software;
- Realização de experimentos visando avaliar como a abordagem proposta pode auxiliar no monitoramento da qualidade do software ao longo do desenvolvimento;
- Realização de experimentos visando avaliar como a abordagem proposta pode ser utilizada pela equipe de um projeto que deseja verificar se ele segue os comportamentos indicados pela literatura;
- Identificação de padrões evolutivos em relação aos projetos de uma organização, ou mesmo padrões que possam transcender a organizações específicas, através da execução de experimentos que comparam os comportamentos encontrados em 16 projetos com os padrões indicados pela literatura; e
- Criação de uma base de dados com a medição do histórico de cerca de 150 projetos em relação a, aproximadamente, 30 métricas;

6.3 LIMITAÇÕES

Uma das limitações do protótipo desenvolvido se refere a realizar a medição apenas com projetos que utilizem Java como linguagem de programação, controle de versão realizado pelo Subversion e gerenciamento de construção com Maven. Desde o início, o protótipo foi projetado com pontos de extensão pensando na sua adaptação para novas nas métricas, sistemas de controle de versão e sistemas de gerenciamento de construção. A implementação de novos conectores para esses pontos de extensão deve ser realizada como detalhado no Capítulo 4.

Na versão da Ostra apresentada neste trabalho, a única linguagem de programação que pode ser medida é Java. Entretanto, para que mais projetos possam ser analisados, é fundamental que o protótipo seja evoluído de forma a avaliar novas linguagens. Atualmente,

já estão sendo desenvolvidos, por outros membros do GEMS, extratores de métricas para C++.

Durante o desenvolvimento deste trabalho, foram identificadas outras métricas, que não foram implementadas por restrições de tempo. Métricas de boas práticas são interessantes para identificar a relação entre atributos de qualidade e boas práticas de programação. Um exemplo de métrica de boas práticas é a quantidade de erros e avisos que programas como PMD (SOURCEFORGE.NET, 2012a) e Checkstyle (SOURCEFORGE.NET, 2012b) informam. Outra métrica interessante é a cobertura do código que pode ser descoberta com o *plugin* do Maven chamado Cobertura (SOURCEFORGE.NET, 2012c).

A capacidade de interagir com outros sistemas de gerência de configuração também é interessante para possibilitar que mais projetos possam ser analisados com a Ostra. O sistema de controle de versão Git (TORVALDS; HAMANO, 2006), por exemplo, tem sido utilizado por muitos projetos de código aberto que podem ser acessados no Sourceforge.net (GEEKNET, INC, 2012) ou no GitHub (GITHUB, INC., 2012). Conectores para outros sistemas de gerenciamento de construção, como Make (FELDMAN, 1979), Ant (BAILLIEZ, 2005) e Rake (WEIRICH, 2008), também seriam úteis.

Por fim, atualmente o protótipo fica constantemente verificando no sistema de controle de versão, de tempos em tempos, se existe uma nova versão para ser medida. A verificação de novas versões poderia ser reativa, disparada quando fosse realizado o *commit* de uma nova versão. Nesse cenário, o sistema de controle de versão poderia avisar à Ostra para realizar a medição quando um novo *commit* fosse identificado. Dessa forma, o custo de processamento do monitoramento poderia ser reduzido.

6.4 TRABALHOS FUTUROS

Enquanto este trabalho foi desenvolvido, foram identificados algumas extensões e trabalhos futuros, que são detalhados nas próximas seções.

6.4.1 MONITORAMENTO DA QUALIDADE DURANTE O DESENVOLVIMENTO

A Ostra faz um monitoramento do histórico do projeto, mas poderia fazer o monitoramento enquanto as versões são desenvolvidas. Atualmente, a Ostra verifica as versões armazenadas no sistema de controle de versões, mas poderia ser desenvolvido um *plugin* de IDE que monitorasse a qualidade do projeto enquanto ele é desenvolvido no espaço de trabalho dos desenvolvedores, antes de ser realizado o *commit*. Dessa forma, a

identificação de problemas poderia ser adiantada ainda mais, fornecendo informações para o desenvolvedor durante a codificação.

Um dos resultados esperados dessa utilização é prevenir a introdução de versões que afetem de maneira indesejada a qualidade do software. Por exemplo, antes de realizar o *commit*, o desenvolvedor poderia identificar que a complexidade ciclomática de algum dos métodos recém-desenvolvidos está acima de 10 ou que o entendimento do projeto diminuiu mais que o esperado, motivando uma possível refatoração.

6.4.2 PADRÕES GERAIS À ENGENHARIA DE SOFTWARE

Em um dos experimentos realizados, foi avaliada a ocorrência de padrões indicados pela literatura em 16 projetos. Nesse experimento, foi possível identificar que alguns padrões foram confirmados e outros negados. Porém, uma das ameaças à validade desse experimento foi a pequena quantidade de projetos considerados: apenas 16.

Uma extensão da pesquisa realizada neste trabalho é realizar novamente o experimento buscando padrões gerais à engenharia de software com mais projetos. Durante as discussões sobre os resultados encontrados nesse experimento, algumas questões de pesquisa foram identificadas. Dentre elas, destacam-se: (1) A linguagem de programação interfere na variação das métricas e atributos de qualidade? (2) Como as medidas de interesse de mineração de dados devem ser variadas para se obter melhores resultados? Além dessas questões de pesquisa, seria interessante realizar esse experimento com mais projetos com o objetivo de se conseguir avaliações mais representativas.

6.4.3 INTEGRAÇÃO ENTRE A OSTRA E O OURIÇO

Em um ciclo de trabalho comum de gerência de configuração, é realizado o *check out*, alterações e *commit*. Entretanto, dessa forma, podem ser inseridas configurações inconsistentes do software no repositório quando um desenvolvedor realiza o *commit* de uma versão que não compila ou não passa nos testes, por exemplo. Um efeito negativo disso é impedir que a equipe de desenvolvimento possa continuar com a implementação de novas funcionalidades, à medida que ao não compilar ou não passar nos testes, a verificação de outras partes do sistema fica comprometida.

O Ouriço é uma abordagem que amplia o conceito do ciclo de trabalho com gerência de configuração, inserindo verificações quando um *commit* é realizado (MENEZES, 2011). Essa abordagem inclui no ciclo de gerencia de configuração verificações assíncronas físicas, sintáticas e semânticas com o objetivo de manter a integridade do repositório.

Embora o Ouriço consiga manter a integridade do repositório de código, ele não protege o repositório de más práticas de desenvolvimento ou da diminuição da qualidade do software. Por outro lado, a Ostra tem a capacidade de monitorar a qualidade do software com métricas e atributos de qualidade de alto nível. Ao combinar essas duas abordagens, torna-se possível definir diretrizes de validação de *commits* de acordo com a qualidade. Um exemplo de diretriz de qualidade que poderia ser utilizada é: "nenhum *commit* pode elevar a complexidade ciclomática de qualquer método acima de 10".

6.4.4 PREDIÇÃO DE VALORES FUTUROS

Durante a realização dos experimentos, cerca de 150 projetos foram medidos com aproximadamente 30 métricas, gerando uma grande base de dados com informação sobre a evolução de vários projetos. Essa base de dados de medições de projetos pode ser utilizada para avaliar a capacidade de técnicas de mineração de dados na predição de valores de métricas.

Nos experimentos realizados neste trabalho, foram consideradas apenas as medições de projeto. Porém, as métricas que são extraídas de pacotes ou arquivos também têm suas medições armazenadas. Dessa forma, a predição poderia indicar qual deve ser a próxima variação de uma determinada métrica para o projeto como um todo e para cada artefato.

6.4.5 ALARMES AUTOMÁTICOS

No Capítulo 2, foram apresentadas diversas métricas. Contudo, somente para algumas delas, como a Complexidade Ciclomática de McCabe, são indicados valores esperados pela literatura. Uma forma alternativa para se trabalhar com métricas que não tem uma definição de valores esperados amplamente adotada é a utilização de alarmes automáticos de acordo com as medições anteriores. Por exemplo, a métrica LOC não possui um valor esperado que deva ser considerado para cada *commit*. Para esse tipo de métrica, os valores identificados no gráfico de controle poderiam ser considerados. Alarmes poderiam ser disparados quando o gráfico de controle identificasse valores de métricas fora do padrão, ou seja, valores além do limite superior de controle, disparando um e-mail para o gerente do projeto, tornando o monitoramento proativo.

6.4.6 DEFINIÇÃO DE INTERVALOS DE INTERESSE

O protótipo é capaz de gerar gráficos de controle para os valores absolutos das métricas e para os valores de delta das métricas. Entretanto, esses gráficos sempre consideram todo o histórico. A incapacidade de definir intervalos de interesse nas versões dos projetos

analisados gera uma limitação de escalabilidade de visualização. Dessa forma, seria interessante poder definir o intervalo que compreende as versões que devem ser consideradas no gráfico. Por exemplo, após algum evento, o comportamento do gráfico pode ser diferente e utilizar todo o histórico pode dificultar a visualização dos valores próximos ao evento, como mudança de equipe ou adoção de novas tecnologias.

Os intervalos de interesse poderiam ser definidos por data ou número de versão. Seria útil também pode fazer um gráfico apenas com as alterações de um determinado desenvolvedor, para verificar como são seus *commits* com o tempo e se ele obteve melhora na qualidade das alterações após uma capacitação, por exemplo.

Outra opção útil seria oferecer, na análise de projetos com diferentes tempos de desenvolvimento, diferentes granularidades para a representação do tempo no gráfico: minutos, dias, semanas, meses ou anos. Com essa funcionalidade, o gráfico se torna mais fácil de interpretar (MURTA, 2002).

A definição dos intervalos de interesse também se torna interessante quando se quer descobrir se um determinado padrão é encontrado em diferentes contextos temporais. Dessa forma é possível descobrir se alguns padrões podem ser encontrados em alguns meses ou durante o trabalho de diferentes equipes em um mesmo projeto.

Configurar automaticamente intervalos de interesse nos gráficos de controle e histogramas, de acordo com o tempo ou desenvolvedor, permitiria um acompanhamento mais fácil dessas ferramentas e, provavelmente, mais eficiente do que observar todo o histórico do projeto. Após alguns tipos de eventos, os gráficos de controle tendem a ser observados com novos intervalos, consequentemente, configurar intervalos de interesse se torna interessante.

6.4.7 DEFINIÇÃO DE MÉTRICAS COMPOSTAS COM OPERADORES LÓGICOS

No estado atual do protótipo é possível criar métricas compostas com operadores aritméticos: +, -, *, /, ^ (potência), *sqrt* (raiz quadrada). Esses operadores fazem sentido quando se quer utilizar métricas compostas para definir atributos de qualidade como os de QMOOD, que seguem equações matemáticas.

Entretanto, para tornar as métricas compostas mais úteis aumentando sua capacidade de relacionar métricas para extrair novas informações é interessante disponibilizar outros operadores. Operadores relacionais e lógicos são interessantes para automatizar a avaliação e tornar a capacidade de avaliação da Ostra maior. Exemplos de operadores relacionais são: > (maior), < (menor), >= (maior ou igual), <= (menor ou igual), = (igual) e <> (diferente). Exemplos de operadores lógicos são: AND e OR. Com esse novo conjunto de operadores, as

métricas compostas poderiam auxiliar na identificação de anomalias. Um exemplo é na identificação de *God Classes*, para as quais poderia ser definida uma métrica composta descrita como "LOC > 1.000", que retornaria verdadeiro caso a classe tenha mais de 1.000 linhas de código.

Para implementar esses novos operadores, a classe abstrata *MetricExpression* deveria ser estendida. A implementação dos operadores suportados atualmente na Ostra foi apresentada por Wallace (2011).

REFERÊNCIAS

AGRAWAL, R.; SRIKANT, R. Fast Algorithms for Mining Association Rules in Large Databases. International Conference on Very Large Data Bases, VLDB 1994. Santiago de Chile, Chile: Morgan Kaufmann. 1994.

ALUR, D.; MALKS, D.; CRUPI, J. Core J2EE Patterns: Best Practices and Design Strategies. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

APACHE SOFTWARE FOUNDATION. Apache Commons BCEL™ -. Disponível em: <http://commons.apache.org/bcel/>. Acesso em: 11 mar. 2012.

ASKLUND, U.; BENDIX, L. A study of configuration management in open source software projects. IEE Proceedings - Software, v. 149, n. 1, p. 40-46, 2002.

BAILLIEZ, S. Apache Ant 1.6.5 Manual. Disponível em: <http://ant.apache.org/manual/index.html>.

BANSIYA, J.; DAVIS, C. G. A Hierarchical Model for Object-Oriented Design Quality Assessment. IEEE Transactions on Software Engineering, v. 28, p. 4–17, 2002.

CAVANO, J. P.; MCCALL, JAMES A. A framework for the measurement of software quality. ACM SIGMETRICS Performance Evaluation Review. New York, NY, USA: ACM. 1978.

CHR. CLEMENS LEE. JavaNCSS - A Source Measurement Suite for Java. Disponível em: <http://javancss.codehaus.org/>. Acesso em: 11 mar. 2012.

CLARKWARE CONSULTING, INC. JDepend. Disponível em: <http://clarkware.com/software/JDepend.html>. Acesso em: 11 mar. 2012.

CODEHAUS. Animal Sniffer Maven Plugin - Introduction. Disponível em: <http://mojo.codehaus.org/animal-sniffer-maven-plugin/>. Acesso em: 16 dez. 2011.

COLARES, F.; SOUZA, J.; CARMO, R.; PÁDUA, C.; MATEUS, G. R. A New Approach to the Software Release Planning. Proceedings of the 2009 XXIII Brazilian Symposium on Software Engineering, SBES 2009. Washington, DC, USA: IEEE Computer Society. Disponível em: <http://dx.doi.org/10.1109/SBES.2009.23>. Acesso em: 3 abr. 2012. 2009.

COLLINS-SUSSMAN, B.; FITZPATRICK, B. W.; PILATO, C. M. Version Control with Subversion. Sebastpol, CA, USA: O'Reilly Media, 2008. v. 2

DART, S. Concepts in Configuration Management Systems. International Workshop on Software Configuration Management (SCM). Trondheim, Norway: ACM Press. 1991.

DICK, S.; MEEKS, A.; LAST, M.; BUNKE, H.; KANDEL, A. Data mining in software metrics databases. Fuzzy Sets and Systems, v. 145, n. 1, p. 81-110, 2004.

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

FAYYAD, U.; PIATETSKY-SHAPIRO, G.; SMYTH, P. From Data Mining to Knowledge Discovery in Databases. AI MAGAZINE, v. 17, p. 37-54, 1996.

FELDMAN, S. I. Make - A Program for Maintaining Computer Programs. Software - Practice and Experience, v. 9, n. 4, p. 255-265, 1979.

FISCHER, M.; PINZGER, M.; GALL, H. Populating a Release History Database from Version Control and Bug Tracking Systems. Proceedings of the International Conference on Software Maintenance, ICSM 2003. Washington, DC, USA: IEEE Computer Society. Disponível em: <http://portal.acm.org/citation.cfm?id=942800.943568>. Acesso em: 11 jul. 2011. 2003.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Design Patterns: Elements of Reusable Object-Oriented Software. 2. ed. Upper Saddle River, NJ, USA: Addison Wesley, 1995.

GEEKNET, INC. SourceForge - Download, Develop and Publish Free Open Source Software. Disponível em: <http://sourceforge.net/>. Acesso em: 26 fev. 2012.

GILBERT, D.; MORGNER, T. JFreeChart, a free Java class library for generating charts. Disponível em: <http://www.jfree.org/jfreechart>. Acesso em: 10 out. 2011.

GITHUB, INC. GitHub - Social Coding. Disponível em: <https://github.com/>. Acesso em: 26 fev. 2012.

HAN, J.; KAMBER, M.; PEI, J. Data Mining: Concepts and Techniques, Third Edition. 3. ed. San Francisco, CA, USA: Morgan Kaufmann, 2011.

HENDERSON-SELLERS, B. Object-oriented metrics: measures of complexity. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995.

IEEE. Std 610.12 - IEEE Standard Glossary of Software Engineering Terminology. . New York, NY, USA: Institute of Electrical and Electronics Engineers. 1990.

ISO. ISO/IEC 9126 - Software engineering - Product quality. . Geneva, Switzerland: International Organization for Standardization. 2001.

JBOSS COMMUNITY. Hibernate - JBoss Community. Disponível em: <http://www.hibernate.org/>. Acesso em: 16 dez. 2011.

JEAN TESSIER. Dependency Finder. Disponível em: <http://depfind.sourceforge.net/>. Acesso em: 11 mar. 2012.

JÚNIOR, M. C.; MENDONÇA, M.; RODRIGUES, F. Mining Software Change History in an Industrial Environment. Proceedings of the 2009 XXIII Brazilian Symposium on Software Engineering, SBES 2009. Washington, DC, USA: IEEE Computer Society. Disponível em: <http://dx.doi.org/10.1109/SBES.2009.8>. Acesso em: 23 jan. 2011. 2009.

KHOSHGOFTAAR, T. M.; SELIYA, N. Comparative Assessment of Software Quality Classification Techniques: An Empirical Case Study. Empirical Software Engineering, v. 9, p. 229–257, set 2004.

KIM, S.; ZIMMERMANN, THOMAS; PAN, K.; WHITEHEAD, E. J. Automatic identification of bug-introducing changes. Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006. Tokyo, Japan: IEEE Computer Society Press. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.89.5071>. Acesso em: 23 jan. 2011. 2006.

MARTIN, G. R. R. A Game of Thrones. 1. ed. New York, NY, USA: Bantam, 1996.

MCCABE, T. J. A Complexity Measure. IEEE Transactions on Software Engineering, v. 2, p. 308–320, jul 1976.

MCCALL, JIM A; RICHARDS, P. K.; WALTERS, G. F. Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality. . Rome, NY, USA: Rome Air Development Center. nov 1977.

MCT. Qualidade e Produtividade no Setor de Software Brasileiro. . Brasília, DF, Brasil: Ministério de Ciência e Tecnologia, Secretaria de Política de Informática. 2005.

MENEZES, G. Ouriço: Uma abordagem para manutenção da consistência em repositórios de gerência de configuração. Dissertação de M.Sc.. Niterói, RJ - Brasil: UFF, 2011.

MERTIK, M.; LENIC, M.; STIGLIC, G.; KOKOL, P. Estimating Software Quality with Advanced Data Mining Techniques. Proceedings of the International Conference on Software Engineering Advances. Washington, DC, USA: IEEE Computer Society. Disponível em: <http://portal.acm.org/citation.cfm?id=1193212.1193789>. Acesso em: 23 jan. 2011. 2006.

MIERLE, K.; LAVEN, K.; ROWEIS, S.; WILSON, G. Mining student CVS repositories for performance indicators. Proceedings of the 2005 International Workshop Conference on Mining Software Repositories, MSR 2005. St. Louis, Missouri: ACM. 2005.

MURTA, L. G. P. Charon: Uma Máquina de Processos Extensível Baseada em Agentes Inteligentes. Rio de Janeiro, Brasil: UFRJ, COPPE, 2002.

NAGAPPAN, N.; BALL, T.; MURPHY, B. Using Historical In-Process and Product Metrics for Early Estimation of Software Failures. Proceedings of the 17th International Symposium on Software Reliability Engineering, ISSRE 2006. Washington, DC, USA: IEEE Computer Society. Disponível em: <http://dx.doi.org/10.1109/ISSRE.2006.50>. Acesso em: 3 abr. 2012. 2006.

NAGAPPAN, N.; BALL, T.; ZELLER, ANDREAS. Mining metrics to predict component failures. Proceedings of the International Conference on Software Engineering Advances, ICSE 2006. New York, NY, USA: ACM. 2006.

O'BRIEN, T.; CASEY, J.; FOX, B. *et al*. Maven: The Definitive Guide. 1st. ed. Sebastopol, CA, USA: O'Reilly, 2008.

ORACLE. The Java Persistence API - A Simpler Programming Model for Entity Persistence. Disponível em: <http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html>. Acesso em: 16 dez. 2011.

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

PROJECT MANAGEMENT INSTITUTE. A Guide to the Project Management Body of Knowledge: 4. ed. Maryland, USA: Project Management Institute, 2008.

RIBEIRO, W. Monitor de Métricas de Software. Trabalho de Conclusão de Curso. Niterói, RJ, BR: UFF, 2011.

ROBLES, G.; GONZALEZ-BARAHONA, J. M.; MICHLMAYR, M.; AMOR, J. J. Mining large software compilations over time: another perspective of software evolution. Proceedings of the 2006 International Workshop Conference on Mining Software Repositories, MSR 2006. Shanghai, China: ACM. 2006.

SOURCEFORGE.NET. PMD - Welcome to PMD. Disponível em: <http://pmd.sourceforge.net/>. Acesso em: 26 fev. 2012a.

SOURCEFORGE.NET. Checkstyle. Disponível em: <http://checkstyle.sourceforge.net/>. Acesso em: 26 fev. 2012b.

SOURCEFORGE.NET. Cobertura. Disponível em: <http://cobertura.sourceforge.net/>. Acesso em: 26 fev. 2012c.

TAGUE, N. R. The quality toolbox. 2nd. ed. Milwaukee, Wisconsin: ASQ Quality Press, 2005.

TMATE SOFTWARE. SVNKit. Disponível em: <http://svnkit.com/>. Acesso em: 10 out. 2011.

TORVALDS, L.; HAMANO, J. C. GIT. Disponível em: <http://git.or.cz>. Acesso em: 20 mar. 2012.

TRAVASSOS, G. H.; KALINOWSKI, M. iMPS 2010 : desempenho das empresas que adotaram o modelo MPS de 2008 a 2010. . Campinas, SP: COPPE/UFRJ - Universidade Federal do Rio de Janeiro. Disponível em: <http://www.softex.br/mpsbr/_livros/arquivos/Softex%20iMPS%202010%20Portugues%20Baixa.pdf>. Acesso em: 7 abr. 2012. 2011.

WEIRICH, J. RAKE — Ruby Make. Disponível em: <http://rake.rubyforge.org/>. Acesso em: 20 mar. 2012.

WERMELINGER, M.; YU, Y. Analyzing the evolution of eclipse plugins. Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008. Leipzig, Germany: ACM. 2008.

WIEGERS, K. Software Requirements. 2. ed. Redmond, Washington: Microsoft Press, 2003.

WIKIPÉDIA. Ostra. Disponível em: <http://pt.wikipedia.org/wiki/Ostra>. Acesso em: 19 jun. 2011.

WITTEN, I. H.; FRANK, E. Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. 2. ed. San Francisco, CA, USA: Morgan Kaufmann Publisher, 2005.

ZIMMERMANN, T.; WEISGERBER, P.; DIEHL, S.; ZELLER, A. Mining version histories to guide software changes. Proceedings of the 26th International Conference on Software Engineering, ICSE 2004. Washington, DC, USA: IEEE Computer Society. Disponível em: <URL>. 2004.

APÊNDICE A – DEFINIÇÃO OPERACIONAL DAS MÉTRICAS

A descrição de como cada uma das métricas do protótipo é calculada é apresentada neste apêndice. A definição das métricas das Tabelas 14-16 está a seguir e também pode ser encontrada na monografia do aluno Wallace Ribeiro (2011).

A métrica Abstração contabiliza a quantidade de classes abstratas em um determinado pacote. Quando calculada para projeto, ela retorna a quantidade de classes abstratas em todo o projeto. A implementação desta métrica utilizou a ferramenta JDepend para identificar as classes abstratas (CLARKWARE CONSULTING, INC, 2011).

A métrica Acoplamento Direto de Classes contabiliza a quantidade de outras classes que são utilizados por uma determinada classe, seja na forma de declaração de atributos ou parâmetros de métodos. Assim, seu resultado é o somatório de diferentes classes que uma classe utiliza como tipo de atributo ou parâmetro de método. A implementaçãoda dessa métrica utilizou a ferramenta BCEL para identificar os tipos dos parâmetros e dos atributos (APACHE SOFTWARE FOUNDATION, 2011).

A métrica Coesão entre os Métodos da Classe avalia o grau de similaridade entre os métodos de uma classe. Para isso, inicialmente é calculado o somatório dos tipos parâmetros utilizados por cada método da classe alvo e em seguida, esse somatório é dividido pela multiplicação da quantidade total de tipos de parâmetros pela quantidade de métodos. Por exemplo, uma classe que possua dois métodos A e B. O método A recebe dois parâmetros: um inteiro (*Integer*) e um *String*. O método B possui como parâmetro apenas um *String*. O valor da métrica CAM para esta classe é 0,75, ou seja, dois parâmetros do método A mais um parâmetro do método B (2 + 1 = 3), dividido por dois parâmetros diferentes (*Integer* e *String*) multiplicado por dois métodos (A e B) resultando em 0,75 (3/4 = 0,75). A implementação dessa métrica utilizou a ferramenta BCEL para identificar os tipos dos parâmetros dos métodos (APACHE SOFTWARE FOUNDATION, 2011).

A métrica Complexidade Ciclomática de McCabe segue a definição de McCabe (MCCABE, 1976). Para calcular essa métrica foi utilizada a ferramenta JavaNCSS (CHR. CLEMENS LEE, 2011).

A métrica Linhas de Código foi implementada utilizando a ferramenta JavaNCSS (CHR. CLEMENS LEE, 2011). Essa implementação de LOC considera todas as linhas de código, mesmo comentários ou linhas em branco.

A Medida da Abstração Funcional contabiliza a relação entre a quantidade de métodos herdados e a quantidade total de métodos acessíveis de uma classe. Inicialmente é calculada a quantidade total de métodos que uma determinada classe possui, herdados e declarados. Para calcular a quantidade de métodos totais de uma classe, suas classes ancestrais são verificadas. A quantidade de métodos herdados considerada é a diferença entre o total de métodos e a quantidade de métodos declarados. Para contar a quantidade de métodos herdados e declarados é utilizada a ferramenta BCEL (APACHE SOFTWARE FOUNDATION, 2011). Por exemplo, se uma determinada classe possui dez métodos, dos quais três são declarados, ela possui MFA 0,7, ou seja sete (métodos herdados) dividido por dez (total de métodos acessíveis). O valor desta métrica para projeto é a média do valor de cada classe dele.

A Medida de Agregação mede a quantidade de atributos de uma classe que cujos tipos são declarados pelo usuário. Para calcular essa métrica, primeiro são identificadas todas as classes declaradas pelo usuário no software e então é verificada a quantidade de tipos de atributos de uma classe que são declarados pelo usuário. Esse procedimento é repetido para todas as classes do software e o valor dessa métrica para o projeto é o somatório de seu valor para cada classe do mesmo. Para identificar os tipos dos atributos foi utilizada a ferramenta BCEL (APACHE SOFTWARE FOUNDATION, 2011).

A Métrica de Acesso a Dados contabiliza a razão entre a quantidade de métodos privados e o total de métodos de uma classe. Para calcular essa métrica foi utilizada a ferramenta Dependency Finder (JEAN TESSIER, 2001). Para calcular essa métrica para projeto, é feita a média entre os valores dessa métrica para as classes de um software.

O Número de Hierarquias é uma métrica que contabiliza a quantidade de classes que estendem outras classes do projeto do usuário. Por exemplo, num software que contenha as classes Veículo, Carro, Moto, Combustível, Álcool e Gasolina. Carro e Moto estendem Veículo e Álcool e Gasolina estendem Combustível. O valor dessa métrica será de 4, ou seja, quatro classes estendem outras classes declaradas no software. Para identificar a quantidade de classes que estendem uma determinada classe é utilizada a ferramenta Dependency Finder (JEAN TESSIER, 2001).

O Número Médio de Ancestrais é calculado pela média de hierarquias das classes do software. Inicialmente é calculada a quantidade de classes que cada classe do software herda informação, até a raiz (em Java Object) e então é calculada a média para todo o software.

O Número de Métodos Polimórficos contabiliza a quantidade de métodos que possuem comportamento polimórfico no projeto. Para calcular essa métrica são identificados os

métodos declarados no projeto que são sobrescritos. Para calcular essa métrica foi utilizada a ferramenta BCEL (APACHE SOFTWARE FOUNDATION, 2011).

A métrica Número de Métodos contabiliza a quantidade de métodos do software. Para calculá-la é utilizada a ferramenta BCEL (APACHE SOFTWARE FOUNDATION, 2011).

A métrica Tamanho do Projeto em Classes contabiliza a quantidade de classes de um software. Essa métrica considera todas as classes do projeto, sejam elas públicas ou não. Para calculá-la, é somada a quantidade de classes do software após a compilação, pelos arquivos *.class*. No caso de projetos Maven, a quantidade de classes presentes na pasta */target/classes*.

O Tamanho da Interface da Classe é uma métrica que contabiliza a quantidade de métodos públicos de uma classe. Para calcular essa métrica foi utilizada a ferramenta Dependency Finder (JEAN TESSIER, 2001) para descobrir a quantidade de métodos públicos.

APÊNDICE B – PROJETOS MEDIDOS

No experimento descrito no Capítulo 5, foram utilizados os quatro projetos com maior quantidade de revisões e que tinham mais de 100 revisões que compilavam dentre os que foram medidos com a Ostra. Na Tabela 49, estão algumas informações sobre os projetos que não foram utilizados nos experimentos e suas medições.

Espera-se que, apesar de não terem sido utilizados nos experimentos apresentados neste trabalho, esses projetos possam ser utilizados em trabalhos futuros ou outros trabalhos do grupo. Dessa forma, essa listagem serve como referência para pesquisadores que necessitem realizar experimentos sobre o histórico de produtos de software.

Na Tabela 49, a segunda coluna mostra o tamanho da equipe (Tam. Equipe), a terceira coluna mostra a quantidade total de revisões do projeto (Rev. Total), a quarta coluna mostra a quantidade de artefatos do projeto (Art.), a quinta coluna mostra a quantidade de revisões que não compilam (Rev. Não Comp.), a sexta coluna mostra a quantidade de revisões que compilam (Rev. Comp.), a sétima coluna mostra a integridade (Int.) do projeto, ou seja o percentual de revisões que compilam e a última coluna mostra a identificação da última revisão (Últ. Rev.) considerada na medição. A informação da última revisão considerada na medição permite que o experimento seja reexecutado.

Tabela 49: Informações sobre os projetos medidos com a Ostra, que não foram utilizados no experimento da Seção 5.4.

Projeto	Tam. Equipe	Rev. Total	Art.	Rev. Não Comp.	Rev. Comp.	Int.	Últ. Rev.
Abdera	6	50	653	10	40	80,00%	1.210.126
Acr MP	1	3	8	0	3	100,00%	1.085.765
Active MQ	8	679	1423	679	0	0,00%	646.930
Animal Sniffer	5	38	36	0	38	100,00%	14.639
Ant MP	6	58	27	18	40	68,97%	833.678
Antlr Maven Plugin	5	8	22	0	8	100,00%	13.111
Antlr3 Maven Plugin	4	14	5	0	14	100,00%	9.039
AntRun MP	12	73	24	3	70	95,89%	1.190.734
App Assembler	14	112	114	14	98	87,50%	16.144
Apt Maven Plugin	3	10	48	0	10	100,00%	11.632
Archiva	4	479	321	479	0	0,00%	1.179.998
Aries	11	40	105	28	12	30,00%	1.069.799
AspectJ Maven Plugin	7	85	72	5	80	94,12%	14.602
Assembly MP	20	290	413	246	44	15,17%	1.214.956
Avro	7	122	186	31	91	74,59%	1.301.818
Axis 2 Java	5	32	55	32	0	0,00%	1.242.249

Projeto	Tam. Equipe	Rev. Total	Art.	Rev. Não Comp.	Rev. Comp.	Int.	Últ. Rev.
Core +							
Axis 2 Java Rampart	9	95	177	53	42	44,21%	1.303.198
Axis 2 Java Sandesha	5	14	32	13	1	7,14%	1.241.032
Axis Tools	7	47	26	5	42	89,36%	12.426
Build Helper Maven Plugin	7	55	17	2	53	96,36%	14.336
Build Number Maven Plugin	6	44	8	7	37	84,09%	15.918
Cassandra Maven plugin	2	22	15	2	20	90,91%	14.732
Castor Maven Plugin	7	43	13	4	39	90,70%	14.768
Changelog MP	6	42	13	12	30	71,43%	1.242.170
Checkstyle MP	7	53	34	13	40	75,47%	1.291.920
Clean MP	6	36	6	22	14	38,89%	1.026.638
Clirr Maven Plugin	12	41	13	1	40	97,56%	16.085
Cobertura Maven Plugin	13	69	62	3	66	95,65%	15.531
Commons Attributes Maven Plugin	2	8	16	0	8	100,00%	6.588
Commons BeanUtils	2	7	17	0	7	100,00%	1.101.749
Commons Utils	9	28	31	0	28	100,00%	6.599
Compiler MP	16	59	50	12	47	79,66%	1.162.951
Dashboard Maven Plugin	3	7	50	0	7	100,00%	12.079
DBUnit Maven Plugin	3	13	5	1	12	92,31%	10.657
DBUpgrade Maven Plugin	1	2	22	0	2	100,00%	14.773
Dependency MP	18	212	80	126	86	40,57%	1.231.549
Deploy MP	13	72	15	6	66	91,67%	1.160.164
Dita Maven Plugin	1	2	4	0	2	100,00%	15.662
Doap MP	5	70	5	4	66	94,29%	1.057.614
Docck MP	4	18	12	5	13	72,22%	948.872
Dock Book Maven Plugin	4	10	6	0	10	100,00%	8.492
Ear MP	11	94	44	20	74	78,72%	1.294.559
Eclipse MP	2	20	45	0	20	100,00%	1.213.573
Ejb MP	7	27	9	7	20	74,07%	1.235.931
Emma Maven Plugin	4	9	20	0	9	100,00%	10.271
Exec Maven Plugin	10	77	31	10	67	87,01%	15.590
Extra Enforcer maven plugin	1	12	3	0	12	100,00%	15.703
Find Bugs Maven Plugin	1	17	7	1	16	94,12%	5.014
Fitnesse Maven plugin	3	22	36	0	22	100,00%	12.224

Projeto	Tam. Equipe	Rev. Total	Art.	Rev. Não Comp.	Rev. Comp.	Int.	Últ. Rev.
GPG MP	9	45	8	1	44	97,78%	1.137.908
Help MP	11	84	14	21	63	75,00%	1.050.549
Hibernate 2 Maven Plugin	5	6	10	0	6	100,00%	6.588
hudson-plugin-svn-branch	1	1	2	0	1	100,00%	3
Idea UI Designer Maven Plugin	1	1	2	0	1	100,00%	6.588
Idlj Maven Plugin	7	25	17	0	25	100,00%	9.344
Install MP	12	41	11	12	29	70,73%	1.178.585
Invoker MP	9	169	26	92	77	45,56%	1.201.342
Jakarta Cactus	3	78	337	78	0	0,00%	741.091
Jakarta String Taglib	1	9	17	9	0	0,00%	793.278
Jalopy Maven Plugin	7	12	5	0	12	100,00%	7.135
Jar MP	15	40	59	17	23	57,50%	1.235.476
Jarsigner MP	2	18	21	1	17	94,44%	1.190.320
Jasper Reports Maven Plugin	3	8	3	0	8	100,00%	16.172
Javancss Maven Plugin	6	43	17	13	30	69,77%	12.176
Jaxb2 Maven Plugin	7	49	29	4	45	91,84%	15.920
JBoss Maven Plugin	6	80	19	7	73	91,25%	15.720
JBoss Packaging Maven Plugin	3	35	47	1	34	97,14%	14.368
JDepend Maven Plugin	8	16	17	7	9	56,25%	12.355
JDiff maven Plugin	4	33	18	0	33	100,00%	15.538
JPox Maven Plugin	6	21	10	20	1	4,76%	6.588
JRuby Maven Plugin	2	7	18	5	2	28,57%	6.588
JSLint Maven Plugin	1	3	5	0	3	100,00%	14.921
JSPC	2	2	8	0	2	100,00%	6.803
KeyTool maven Plugin	5	14	22	4	10	71,43%	15.047
L10n Maven Plugin	3	4	3	0	4	100,00%	10.364
Latex Maven Plugin	1	9	2	0	9	100,00%	13.767
License Maven Plugin	2	44	160	0	44	100,00%	14.832
Linkcheck MP	6	31	5	23	8	25,81%	1.021.298
Maven Extensions	3	6	3	0	6	100,00%	6.588
Mojo Archetypes	3	5	10	0	5	100,00%	14.385

Projeto	Tam. Equipe	Rev. Total	Art.	Rev. Não Comp.	Rev. Comp.	Int.	Últ. Rev.
Monitoria Web	6	363	64	350	13	3,58%	24.354
Native 2 ASCII Maven Plugin	4	5	3	0	5	100,00%	15.006
Open JPA Maven Plugin	5	32	22	0	32	100,00%	13.083
OSX App Bunbler Maven Plugin	3	21	6	16	5	23,81%	11.962
Ounce Maven Plugin	3	15	22	0	15	100,00%	14.255
Patch MP	3	11	2	1	10	90,91%	895.009
Pde Maven Plugin	4	15	25	9	6	40,00%	7.054
PDF MP	7	66	6	50	16	24,24%	1.070.092
Plugin Support	1	1	24	0	1	100,00%	6.588
Properties Maven Plugin	4	6	9	0	6	100,00%	12.895
Reactor MP	3	8	10	0	8	100,00%	735.952
Remote Resources MP	11	69	13	28	41	59,42%	1.300.228
Repository MP	8	30	8	5	25	83,33%	965.077
Resources MP	18	48	12	29	19	39,58%	1.292.992
Retrotranslator Maven Plugin	2	18	12	9	9	50,00%	6.588
RMic Maven Plugin	1	6	18	0	6	100,00%	8.127
RPM Maven Plugin	5	79	27	0	79	100,00%	16.150
Sablecc Maven Plugin	2	4	4	0	4	100,00%	6.232
SCM Changelog Maven Plugin	3	36	62	2	34	94,44%	10.760
Selenium Maven Plugin	3	10	4	10	0	0,00%	4.035
Ship Maven Plugin	1	9	13	0	9	100,00%	14.211
Site MP	5	31	14	7	24	77,42%	1.294.346
SMC Maven Plugin	2	2	9	0	2	100,00%	6.588
Solaris Maven Plugin	2	14	28	0	14	100,00%	7.323
Sonar Maven plugin	2	9	4	4	5	55,56%	14.935
Source MP	16	48	47	12	36	75,00%	1.000.029
SQL Maven Plugin	9	82	6	3	79	96,34%	15.103
SQLJ Maven Plugin	2	7	4	0	7	100,00%	9.039
Sysdeo Tomcat Maven Plugin	2	14	9	0	14	100,00%	10.709
Taglist Maven Plugin	8	57	22	3	54	94,74%	10.562
Toolchains	3	11	4	7	4	36,36%	802.617
Truezip Maven Plugin	2	21	18	0	21	100,00%	16.125

Projeto	Tam. Equipe	Rev. Total	Art.	Rev. Não Comp.	Rev. Comp.	Int.	Últ. Rev.
Unix	2	45	245	17	28	62,22%	14.303
Verifier MP	3	8	5	6	2	25,00%	900.472
Wagon Maven Plugin	3	13	27	2	11	84,62%	14.258
War MP	20	128	44	47	81	63,28%	1.235.053
Was6 Maven Plugin	3	56	24	1	55	98,21%	15.622
Weblogic Maven Plugin	3	23	17	23	0	0,00%	12.230
Webstart	7	47	93	9	38	80,85%	16.042
Webtest Maven Plugin	1	2	19	0	2	100,00%	14.339
Xdoclet Maven Plugin	4	10	10	1	9	90,00%	10.786
Xml Maven Plugin	5	33	13	11	22	66,67%	13.674
Xmlbeans Maven Plugin	7	48	14	10	38	79,17%	14.819

APÊNDICE C – TABELAS DE COMPORTAMENTO UTILIZADAS NO TERCEIRO EXPERIMENTO

As tabelas de comportamento geradas com os 16 projetos no experimento mostrado na Seção 5.4 estão apresentadas a seguir nas Figuras 84-91.

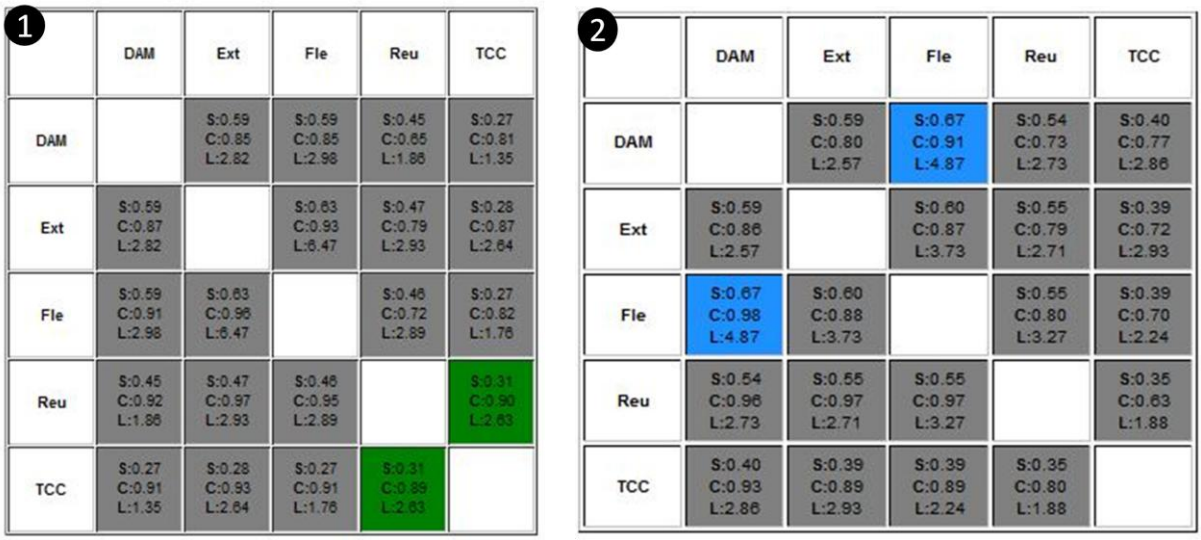


Figura 83: Tabelas de comportamento geradas pela Ostra para os projetos (1) IdUFF e (2) Maven GWT Plugin.

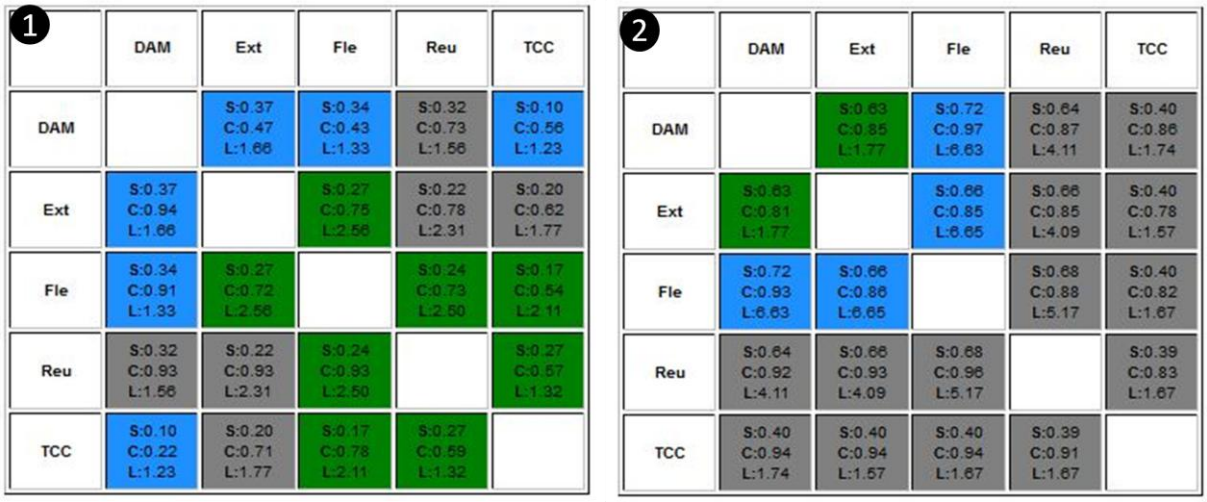
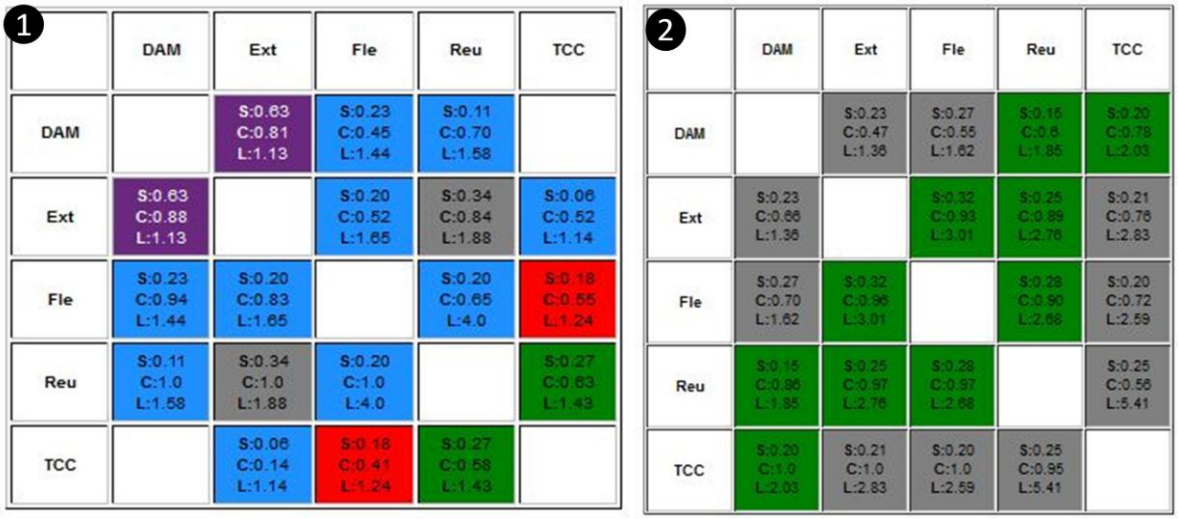


Figura 84: Tabelas de comportamento geradas pela Ostra para os projetos (1) Maven Javacc Plugin e (2) Maven Javadoc Plugin.



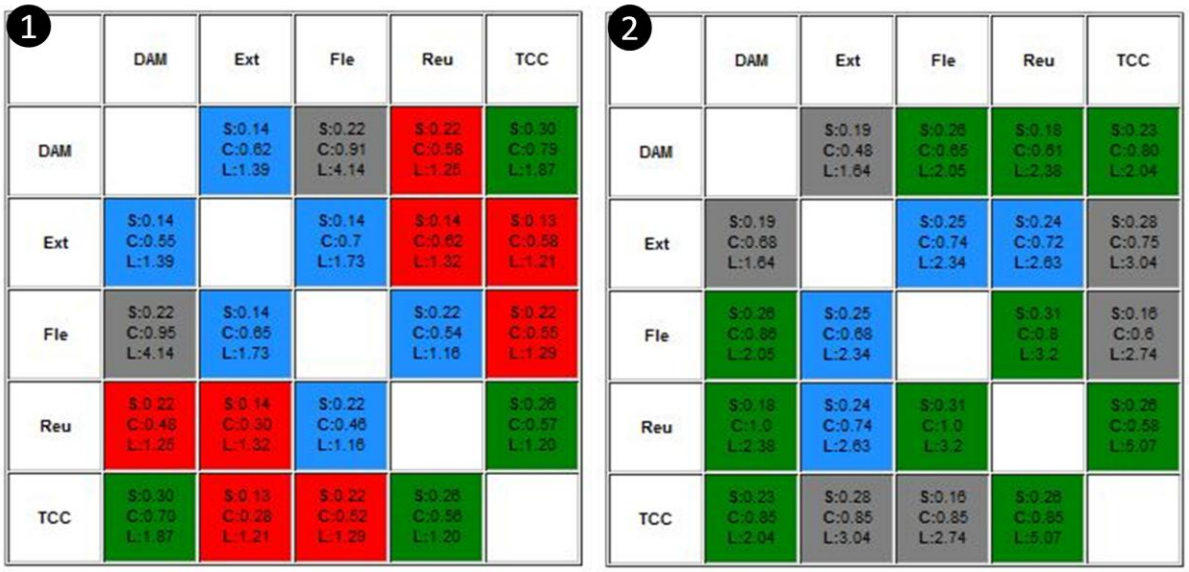
1

	DAM	Ext	Fle	Reu	TCC
DAM		S:0.63 C:0.81 L:1.13	S:0.23 C:0.45 L:1.44	S:0.11 C:0.70 L:1.58	
Ext	S:0.63 C:0.88 L:1.13		S:0.20 C:0.52 L:1.65	S:0.34 C:0.84 L:1.88	S:0.06 C:0.52 L:1.14
Fle	S:0.23 C:0.94 L:1.44	S:0.20 C:0.83 L:1.65		S:0.20 C:0.65 L:4.0	S:0.18 C:0.55 L:1.24
Reu	S:0.11 C:1.0 L:1.58	S:0.34 C:1.0 L:1.88	S:0.20 C:1.0 L:4.0		S:0.27 C:0.63 L:1.43
TCC		S:0.06 C:0.14 L:1.14	S:0.18 C:0.41 L:1.24	S:0.27 C:0.58 L:1.43	

2

	DAM	Ext	Fle	Reu	TCC
DAM		S:0.23 C:0.47 L:1.36	S:0.27 C:0.55 L:1.62	S:0.15 C:0.6 L:1.85	S:0.20 C:0.78 L:2.03
Ext	S:0.23 C:0.66 L:1.36		S:0.32 C:0.93 L:3.01	S:0.25 C:0.89 L:2.76	S:0.21 C:0.76 L:2.83
Fle	S:0.27 C:0.70 L:1.62	S:0.32 C:0.96 L:3.01		S:0.28 C:0.90 L:2.68	S:0.20 C:0.72 L:2.59
Reu	S:0.15 C:0.86 L:1.85	S:0.25 C:0.97 L:2.76	S:0.28 C:0.97 L:2.68		S:0.25 C:0.56 L:5.41
TCC	S:0.20 C:1.0 L:2.03	S:0.21 C:1.0 L:2.83	S:0.20 C:1.0 L:2.59	S:0.25 C:0.95 L:5.41	

Figura 85: Tabelas de comportamento geradas pela Ostra para os projetos (1) Maven Native e (2) Maven NBM Plugin.



1

	DAM	Ext	Fle	Reu	TCC
DAM		S:0.14 C:0.62 L:1.39	S:0.22 C:0.91 L:4.14	S:0.22 C:0.58 L:1.26	S:0.30 C:0.79 L:1.87
Ext	S:0.14 C:0.55 L:1.39		S:0.14 C:0.7 L:1.73	S:0.14 C:0.62 L:1.32	S:0.13 C:0.58 L:1.21
Fle	S:0.22 C:0.95 L:4.14	S:0.14 C:0.65 L:1.73		S:0.22 C:0.54 L:1.16	S:0.22 C:0.55 L:1.29
Reu	S:0.22 C:0.48 L:1.26	S:0.14 C:0.30 L:1.32	S:0.22 C:0.46 L:1.16		S:0.26 C:0.57 L:1.20
TCC	S:0.30 C:0.79 L:1.87	S:0.13 C:0.28 L:1.21	S:0.22 C:0.52 L:1.29	S:0.26 C:0.56 L:1.20	

2

	DAM	Ext	Fle	Reu	TCC
DAM		S:0.19 C:0.48 L:1.64	S:0.26 C:0.65 L:2.05	S:0.18 C:0.61 L:2.38	S:0.23 C:0.80 L:2.04
Ext	S:0.19 C:0.68 L:1.64		S:0.25 C:0.74 L:2.34	S:0.24 C:0.72 L:2.63	S:0.28 C:0.75 L:3.04
Fle	S:0.26 C:0.86 L:2.05	S:0.25 C:0.68 L:2.34		S:0.31 C:0.8 L:3.2	S:0.16 C:0.6 L:2.74
Reu	S:0.18 C:1.0 L:2.38	S:0.24 C:0.74 L:2.63	S:0.31 C:1.0 L:3.2		S:0.26 C:0.58 L:5.07
TCC	S:0.23 C:0.85 L:2.04	S:0.28 C:0.85 L:3.04	S:0.16 C:0.85 L:2.74	S:0.26 C:0.85 L:5.07	

Figura 86: Tabelas de comportamento geradas pela Ostra para os projetos (1) Maven PMD Plugin e (2) Maven Project Info Reports Plugin.

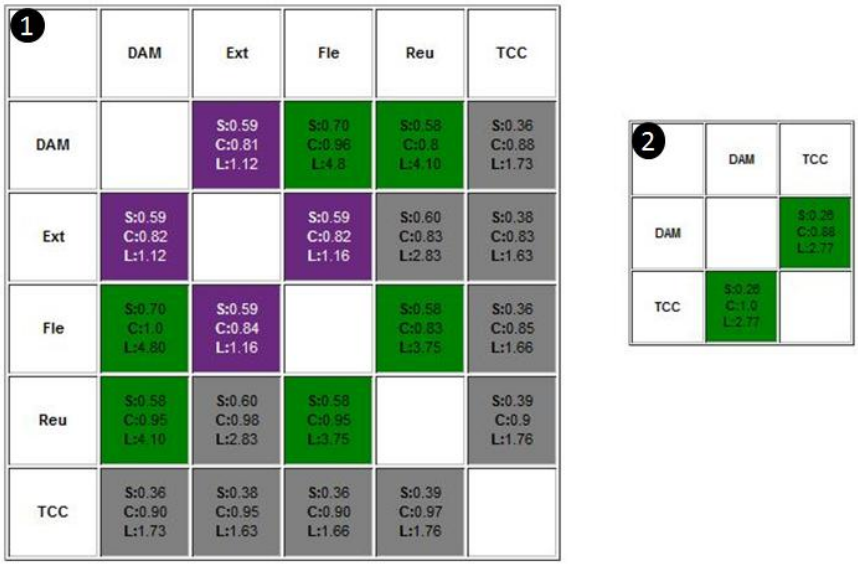


Figura 87: Tabelas de comportamento geradas pela Ostra para os projetos (1) Maven Shade Plugin e (2) Maven Changes Plugin.

A segunda tabela de comportamento da Figura 87 está num formato diferente, pois apenas foram encontradas regras envolvendo as métricas TCC e DAM. Optou-se por colocar a tabela de comportamento da forma que foi gerada pela Ostra, sem realizar alterações de estilo.

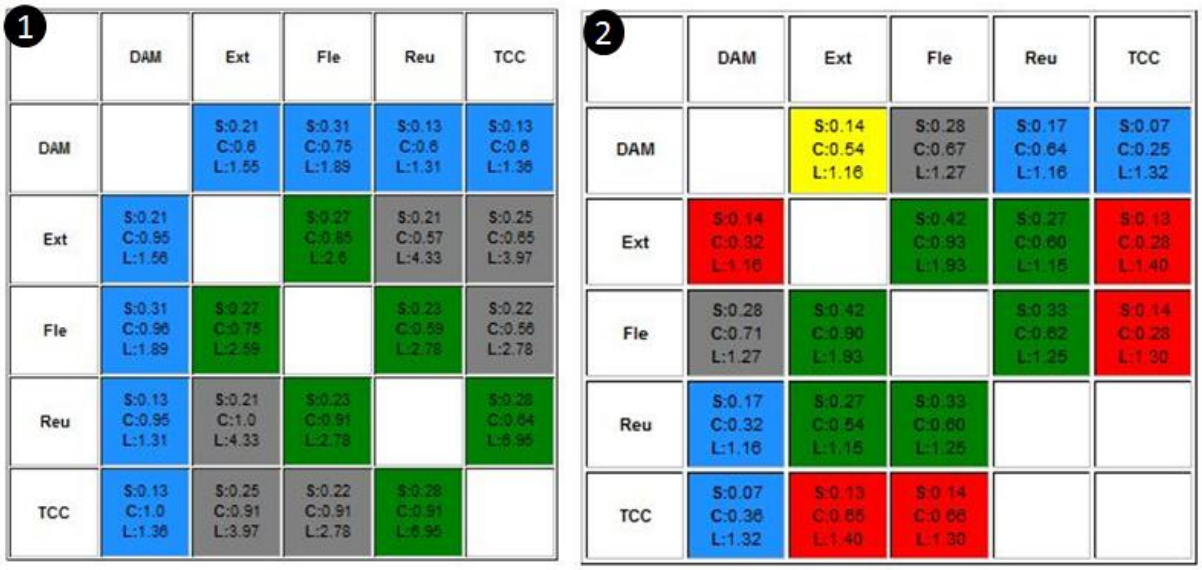


Figura 88: Tabelas de comportamento geradas pela Ostra para os projetos (1) Maven Versions Plugin e (2) Oceano Core.

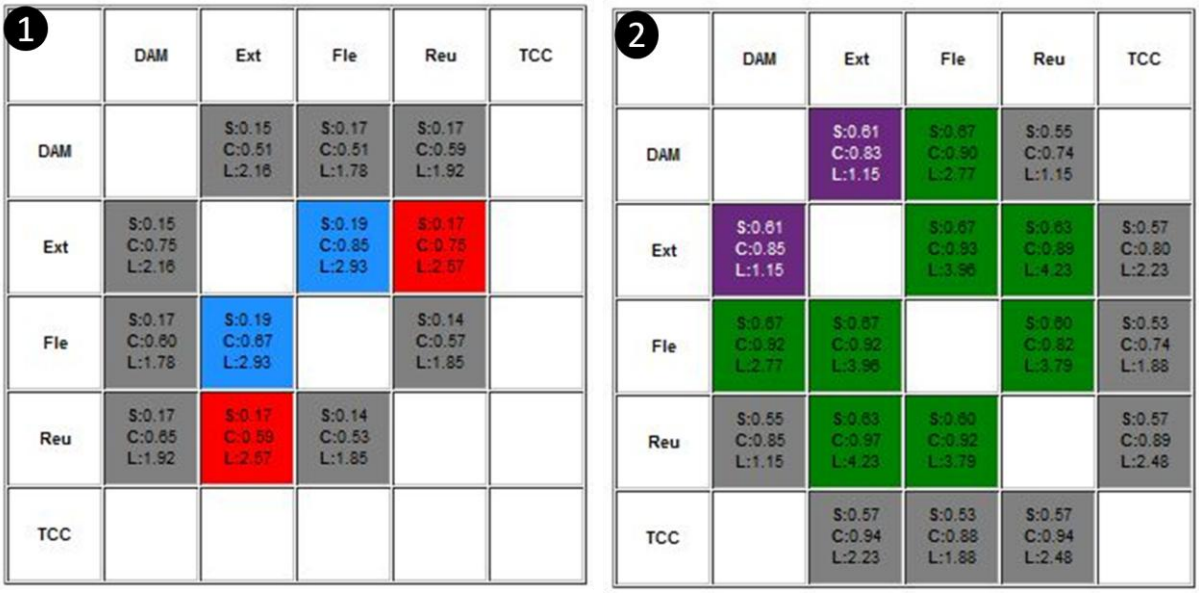


Figura 89: Tabelas de comportamento geradas pela Ostra para os projetos (1) Oceano Web e (2) Público Core.

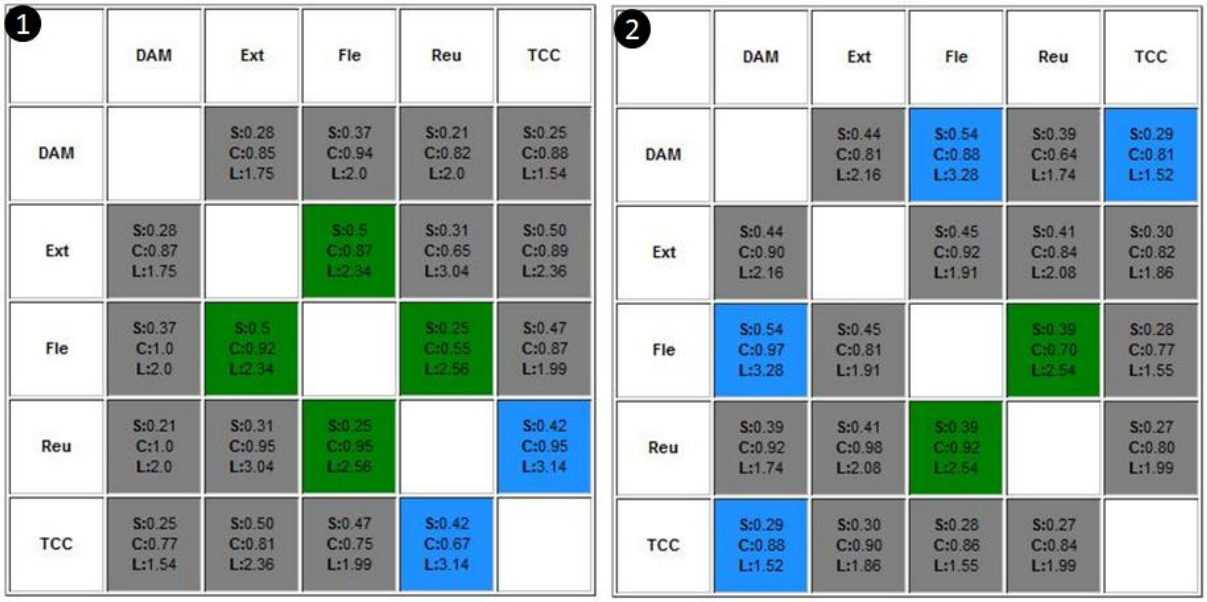


Figura 90: Tabelas de comportamento geradas pela Ostra para os projetos (1) Acadêmico Pós Graduação Core e (2) Monitoria Core.

Após a identificação do comportamento com maior *lift* nas células de cor cinza, foram geradas as Figuras 92 e 93. Na primeira célula de cada tabela está a quantidade aproximada de regras mineradas com o suporte e *lift* definidos, onde *k* indica mil. Para todos os projetos, foi utilizado *lift* igual a 1,1 e 10 como suporte absoluto. O suporte relativo, utilizado na mineração de cada projeto é mostrado acima das tabelas, à frente do nome do projeto. O valor indicado na célula de cada comportamento é o *lift* da regra que referência o comportamento em questão.

Público Core Sup= 0,079

29k	Int	Man	Fle	Reu	1/Test
Int		1,15	2,77	1,15	
Man	1,15		3,96	4,23	2,23
Fle	2,77	3,96		3,79	1,88
Reu	1,15	4,23	3,79		2,48
1/Test		2,23	1,88	2,48	

Oceano Core Sup= 0,047

3k	Int	Man	Fle	Reu	1/Test
Int		1,16	1,27	1,16	1,32
Man	1,16		1,93	1,15	1,4
Fle	1,27	1,93		1,25	1,3
Reu	1,16	1,15	1,25		
1/Test	1,32	1,4	1,3		

Oceano Web Sup= 0,085

264	Int	Man	Fle	Reu	1/Test
Int		2,16	1,78	1,92	
Man	2,16		2,93	2,57	
Fle	1,78	2,93		1,85	
Reu	1,92	2,57	1,85		
1/Test					

IdUFF Sup= 0,007

288k	Int	Man	Fle	Reu	1/Test
Int		2,82	2,98	1,86	1,35
Man	2,82		6,47	2,93	2,64
Fle	2,98	6,47		2,89	1,76
Reu	1,86	2,93	2,89		2,63
1/Test	1,35	2,64	1,76	2,63	

Maven GWT Plugin Sup= 0,040

51k	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	

Maven Javadoc Plugin Sup= 0,040

62k	Int	Man	Fle	Reu	1/Test
Int		1,77	6,63	4,11	1,74
Man	1,77		6,65	4,09	1,57
Fle	6,63	6,65		5,17	1,67
Reu	4,11	4,09	5,17		1,67
1/Test	1,74	1,57	1,67	1,67	

Maven Native Sup= 0,068

6k	Int	Man	Fle	Reu	1/Test
Int		1,13	1,44	1,58	
Man	1,13		1,65	1,88	1,14
Fle	1,44	1,65		4	1,24
Reu	1,58	1,88	4		1,43
1/Test		1,14	1,24	1,43	

Maven Nbm Plugin Sup= 0,048

17k	Int	Man	Fle	Reu	1/Test
Int		1,36	1,62	1,85	2,03
Man	1,36		3,01	2,76	2,83
Fle	1,62	3,01		2,68	2,59
Reu	1,85	2,76	2,68		5,41
1/Test	2,03	2,83	2,59	5,41	

Maven Versions Plugin Sup= 0,055

9k	Int	Man	Fle	Reu	1/Test
Int		1,55	1,89	1,31	1,36
Man	1,56		2,6	4,33	3,97
Fle	1,89	2,56		2,78	2,78
Reu	1,31	4,33	2,78		6,95
1/Test	1,36	3,97	2,78	6,95	

Maven Changes Plugin Sup= 0,043

1k	Int	Man	Fle	Reu	1/Test
Int					2,77
Man					
Fle					
Reu					
1/Test	2,77				

Maven Javacc Plugin Sup= 0,078

4k	Int	Man	Fle	Reu	1/Test
Int		1,66	1,33	1,56	1,23
Man	1,66		2,56	2,31	1,77
Fle	1,33	2,56		2,5	2,11
Reu	1,56	2,31	2,5		1,32
1/Test	1,23	1,77	2,11	1,32	

Maven PMD Plugin Sup= 0,096

824	Int	Man	Fle	Reu	1/Test
Int		1,39	4,14	1,25	1,87
Man	1,39		1,73	1,32	1,21
Fle	4,14	1,73		1,16	1,29
Reu	1,25	1,32	1,16		1,2
1/Test	1,87	1,21	1,29	1,2	

Figura 91: Tabelas de comportamento consideradas no terceiro experimento (Parte 1).

M. Project Info Reports P. Sup= 0,063

4k	Int	Man	Fle	Reu	1/Test
Int		1,64	2,05	2,38	2,04
Man	1,64		2,34	2,63	3,04
Fle	2,05	2,34		3,2	2,74
Reu	2,38	2,63	3,2		5,07
1/Test	2,04	3,04	2,74	5,07	

Academico Pos Grad. Core Sup= 0,098

4k	Int	Man	Fle	Reu	1/Test
Int		1,75	2	2	1,54
Man	1,75		2,34	3,04	2,36
Fle	2	2,34		2,56	1,99
Reu	2	3,04	2,56		3,14
1/Test	1,54	2,36	1,99	3,14	

Monitoria Core Sup= 0,075

7k	Int	Man	Fle	Reu	1/Test
Int		2,16	3,28	1,74	1,52
Man	2,16		1,91	2,08	1,86
Fle	3,28	1,91		2,54	1,55
Reu	1,74	2,08	2,54		1,99
1/Test	1,52	1,86	1,55	1,99	

Maven Shade Plugin Sup= 0,098

7k	Int	Man	Fle	Reu	1/Test
Int		1,12	4,8	4,1	1,763
Man	1,12		1,16	2,83	1,63
Fle	4,8	1,16		3,75	1,66
Reu	4,1	2,83	3,75		1,76
1/Test	1,73	1,63	1,66	1,76	

Figura 92: Tabelas de comportamento consideradas no terceiro experimento (Parte 2).