Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Análise da Correlação entre Métricas de Evolução e Qualidade de *Design* de Software

Pablo Oliveira Antonino de Assis

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus Campina Grande, como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Dalton Dario Serey Guerrero
Jorge César Abrantes de Figueiredo
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

A848a
2009 Assis, Pablo Oliveira Antonino de.
Análise da correlação entre métricas de evolução e qualidade de design de software / Pablo Oliveira Antonino de Assis. — Campina Grande, 2009.
59f. : il.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Referências.
Orientadores: Profº. Dr. Dalton Dario Serey Guerreiro e Dr. Jorge César Abrantes de Figueiredo.

1. Qualidade de Software. 2. Evolução de Software. 3. Métricas de Complexidade. 4. Bad Smells. I. Título.

CDU 004.05(043)

"ANÁLISE DA CORRELAÇÃO ENTRE MÉTRICAS DE EVOLUÇÃO E QUALIDADE DE DESIGN DE SOFTWARE"

PABLO OLIVEIRA ANTONINO DE ASSIS

DISSERTAÇÃO APROVADA EM 13.03.2009

PROF. DALTON DARIO SEREY GUERRERO, D.Sc
Orientador

PROF. JORGE CÉSAR ABRANTES DE FIGUEIREDO, D.Sc
Orientador

PROF. ROHIT GHEYI, Dr.
Examinador

PROF. TIAGO LIMA MASSONI , Dr.
Examinador

PROFa. FRANCILENE PROCÓPIO GARCIA, D.Sc
Examinadora

CAMPINA GRANDE – PB

Para minha tia Evani e meu avô Chicão, que partiram no período de desenvolvimento deste trabalho, deixando muitas saudades.

Resumo

Nós investigamos a evolução de oito softwares *open source* e cinco proprietários, a fim de verificar a existência de correlações estatísticas entre complexidade e medidas de qualidade em termos de *bad smells* e *bugs*. Em todos os projetos, encontramos fortes correlações estatísticas entre medidas de complexidade (WMC) e qualidade. Todos os softwares proprietários e cinco *open source* apresentaram índices de correlação muito forte ($r > 0.9$). Surpreendentemente, em três dos softwares *open source*, a correlação encontrada foi forte, porém negativa. Isto é atribuído ao fato de que, nestes projetos, os *bad smells* foram removidos intencionalmente. Este resultado sugere que, apesar da correlação, não existe necessariamente relação de causa-efeito entre métricas de complexidade e de qualidade. Dessa maneira, concluímos que apenas eliminar *bad smells* não é uma boa estratégia a ser seguida se o objetivo for reduzir a complexidade do *design* e melhorar a qualidade nos termos associados à redução da complexidade.

Abstract

We have studied the evolution of eight open source projects and five proprietary ones, looking for statistical correlations between complexity and quality measures in terms of bad smells and bugs detected. In all projects, we found strong statistical correlations between complexity (WMC) and quality measures. In all the legacies softwares and five of open sources, the correlation can be considered very strong ($r > 0.9$). Surprisingly, in three of the open source, the correlation is strong, but negative. This has been attributed to the fact that, in these projects, designers have intentionally controlled the quality measures under study, by applying refactoring strategies. These results suggest that, despite the correlation, there is no necessary cause-effect relation between complexity and quality measures. We conclude that just eliminate *bad smells* is not a good strategy to be followed if the desired objective is to reduce software design complexity. Then also does not improve software quality in terms associated to software complexity reduction.

Agradecimentos

Agradeço primeiramente a Deus por ter me conduzido nessa jornada, a qual só foi possível porque Ele esteve sempre me orientando, dando força e discernimento em todas as situações. A Ele toda honra, glória e louvor para todo o sempre.

Aos meus pais Israel e Iolanda por serem meu exemplo de integridade, força e determinação. Obrigado por existirem e por serem anjos de Deus em minha vida. Amo vocês!

A Paloma, minha irmã amada, que, em suas poucas palavras, sempre demonstrou apoio. Amo você, maninha.

A Elisa, minha amiga, parceira, namorada, noiva, confidente... minha auxiliadora. Obrigado por fazer parte de minha vida e por deixá-la mais fácil de ser vivida. Amo você, galega.

De maneira bastante especial, agradeço a Côca, Gilson, Isabela e Katarina, por terem sido meu refúgio e ombro amigo nas semanas longe de casa. Vocês foram fundamentais.

Aos meus orientadores Dalton e Jorge pelos ensinamentos, orientações e conselhos. Obrigado pela amizade e atenção prestada.

Ao amigo Jemerson, pelo apoio no decorrer deste trabalho. Você foi fundamental para a realização deste trabalho, meu caro. Sucesso na sua jornada.

Aos amigos do GMF, pelo apoio e companheirismo. Em especial, a Roberto, pelos conselhos e ajuda no início dos trabalhos. Sua boa vontade e disposição serviram de exemplo.

A professora Michelli Silva do DME/UFCG e ao estatístico Hérico Gouveia pelo auxílio prestado na parte estatística do trabalho

Aos grande amigos que fiz na UFCG durante esse período, em especial a Fernando, Felipe, Gilson, Yuri, Guiga, Saulo, Marcos e Elmano. Obrigado pela amizade e companheirismo.

Aos colegas de morada, Jean, Gil, Tony, Hugo, Caio e Rodrigo, por terem sido amigos mais chegados que irmãos.

A COPIN, em especial Aninha e Vera por toda atenção e boa vontade em ajudar.

A CAPES, a FINEP e a CPMBraxis por terem acreditado e proporcionado o desenvolvimento deste trabalho.

A todos vocês, meu muito obrigado.

Conteúdo

Lista de Símbolos

WMC - *Weighted Methods per Class*

API - *Application Programming Interface*

JDK - *Java Development Toolkit*

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Evolução de software é o processo de mudanças realizado sobre um produto de software já desenvolvido, cujo propósito é incorporar melhorias e/ou corrigir bugs [LR01], [LR06].

A evolução de software, bem como suas causas e conseqüências, foram investigadas inicialmente por Meir M. Lehman [Leh69], que entendeu, após mais de 20 anos estudando os sistemas da IBM, que a atividade de evoluir um produto de software é necessária para que ele atenda novas necessidades dentro do contexto em que está inserido e consequentemente não caia em desuso.

Fred Brooks, em seu trabalho *No Silver Bullet: Essence and Accidents of Software Engineering* [FPB87], afirma que as características de produtos de software são agrupadas como Acidentais e Essenciais. Características acidentais são aquelas adquiridas por conta das circunstâncias impostas pelo contexto em que o software está inserido. Nessa categoria, se enquadram problemas criados pela equipe de desenvolvimento, como, por exemplo, *design* mal estruturado e módulos não condizentes com a especificação. Já características essenciais são inerentes à natureza de qualquer software. Nessa categoria se enquadram características como, por exemplo, necessidade de evolução. Além de entender que evolução de software é uma característica essencial, Brooks ainda afirma que, se um produto de software for realmente útil, então necessariamente vai passar por um processo de evolução [FPB87].

Parnas também considera a evolução como essencial e inevitável para produtos de software, uma vez que frequentemente se refere ao que ele denomina de envelhecimento de software (*software aging*) ao abordar o tema Evolução de Software [Par94]. Segundo Parnas, todo software precisa ser evoluído ao longo de sua vida útil porque a dinâmica dos

negócios em que os produtos de software estão inseridos impõe que eles sejam evoluídos. Nesse contexto, a expressão envelhecimento de software é usada para se referir ao fato de que, se não for evoluído, o software deixa gradativamente de atender às expectativas de seus usuários. Desse ponto de vista, a questão que se apresenta é: como retardar o envelhecimento de software? A hipótese amplamente aceita é que se o processo evolutivo for devidamente controlado, o software poderá evoluir bem e seu envelhecimento será retardado [LR06]. Por outro lado, a falta de controle sobre a evolução do software tende a reduzir a qualidade do código, do *design* e do produto como um todo, antecipando assim o envelhecimento do software [Par94].

É exatamente devido à necessidade de evoluir o software, possivelmente de forma não antecipada na arquitetura e *design* originais, que se estabelece uma relação entre o processo evolutivo do software e a qualidade do *design*.

O conceito de qualidade de software, por sua vez, é "onipresente"na Engenharia de Software e vem sendo exaustivamente investigado desde a década de setenta. O conceito engloba desde questões que tratam do processo de desenvolvimento até o código que está sendo gerado, passando pela documentação, arquitetura e *design*.

Naur e Randell [NR69] citam três pontos que sempre se destacam quando o tema qualidade de software é trazido à tona: 1) Produtos de software de baixa qualidade, 2) Código de difícil evolução e 3) Projetos de difícil gerenciamento. Códigos ruins caracterizam produtos de software de baixa qualidade e, consequentemente, de difícil evolução, o que torna o projeto difícil de ser gerenciado.

Questões que relacionam produtos de software, evolução e qualidade são temas que sempre remetem um ao outro e merecem ser analisados com cautela. Dentre as técnicas e ferramentas que vêm sendo estudadas e desenvolvidas com o objetivo de melhorar a qualidade dos softwares, três muito citadas são Métricas de Software [CK94], detecção de *Bad Smells* [Fow99] e informações relacionadas a *bugs* [HS91].

Métricas de software são informações geralmente coletadas automaticamente a partir das estruturas do código fonte de projetos de software. Elas têm por finalidade servir como indicadores quantitativos de uma série de aspectos do software que tenham relação com padrões de complexidade, coesão, acoplamento e tamanho do sistema como um todo ou de partes dele [CK94].

O termo *bad smell*, introduzido por Fowler [Fow99], refere-se a características presentes em trechos de código que, a longo prazo, podem vir a afetar a qualidade do sistema como um todo. A detecção de *bad smells* se baseia na experiência adquirida por engenheiros de software, que verificam características ruins em *designs* que devem ser removidas, de modo a garantir a qualidade do *design*. Além do próprio Fowler [Fow99], alguns pesquisadores como Mantyla [Man04], desenvolvem trabalhos que sugerem estratégias de refatoração para cada *bad smell* específico. Melhor eplicando, para cada *bad smell* indicado, sugere-se uma manipulação no código a fim de eliminá-lo.

Um *bug* é uma falha na sequência lógica de execução de um programa, que leva a não realização da plena execução da atividade fim do software. Vários estudos que tratam de qualidade de software mostram resultados gerados a partir dos *bugs* dos programas, como, por exemplo, densidade de bugs por classe, contagem de bugs por desenvolvedor e tempo de vida de um bug [HS91]. O propósito maior destes trabalhos é associar fatores de qualidade com informações desta natureza.

Dessa maneira, melhorar a qualidade de um software passou a ser uma atividade de estruturar o código de maneira que as métricas fossem minimizadas ou maximizadas [CFRH+07], [CFR05], eliminar *bad smells* dos *designs* e controlar o aparecimento de *bugs* através da manipulação de variáveis que influenciam de maneira direta o aparecimento deles [CK94] [Fow99], [HS91].

É senso comum entre os engenheiros de software que a qualidade está intimamente relacionada com a complexidade do software [ISO91]. Com base nisto, realizar atividades que reduzam a complexidade diretamente teria um impacto imediato na qualidade do software [CFRH+07]. Este trabalho mostra que esta hipótese não é verdadeira.

De modo a esclarecer e contribuir com a evolução do tema, este trabalho mostra o resultado de explorações de repositórios de software open source e proprietários que foram feitas buscando descobrir correlações entre métricas de complexidade e fatores-chave de qualidade como *bugs* e *bad smells*, verificando também a existência ou não de relações de causa-efeito entre elas. Estes dados podem servir de indicativos de quais características deveriam ser melhoradas, de modo que a complexidade do software fosse reduzida e consequentemente a qualidade do software fosse melhorada.

Nossos resultados indicaram a existência de forte correlação estatística entre:

1. WMC - *Weighted Methods per Class* [CK94]: Medida de complexidade de classe;

2. *Bad Smells*: Não apenas os clássicos definidos por Fowler [Fow99], mas também outras más práticas de programação definidas por especialistas em software; e

3. *Bugs:* Número de defeitos detectados por ferramentas de coleta automática.

Contudo, apesar da forte correlação estatística entre as variáveis, não foi identificada uma relação de causa-efeito entre elas. A pura e simples eliminação de *bad smells* não implica em diminuição da complexidade do software nem na melhora da qualidade associada à redução da complexidade. Isso mostrou fortes evidências de que realizar tarefas que reduzam a complexidade de um software, não necessariamente implica em melhora nas medidas de qualidade.

1.1 Organização do Trabalho

Esta dissertação está organizada da seguinte forma: No capítulo 2 são apresentados temas fundamentais para o entendimento da dissertação, incluindo evolução de software, qualidade de software, métricas de software, análise de correlação simples e testes de hiótese. O capítulo 3 apresenta o Evolution Miner, que é uma API desenvolvida no escopo deste trabalho e que propõe dar suporte ao estudo de evolução de software a partir da mineração de repositórios de software. No capítulo 4 é apresentada a análise de relação causa-efeito entre qualidade e e complexidade de *design* de software, realizada com dados providos pelo Evolution Miner. O capítulo 5 mostra possíveis trabalhos futuros e nossas conclusões e por fim, o Capítulo 6 discute os trabalhos relacionados.

Capítulo 2

Fundamentação Teórica

2.1 Evolução de Software

A evolução de software consiste em realizar mudanças em sistemas de software, tendo em vista novos requisitos gerados por mudanças em regras de negócio, assim como erros que impedem o funcionamento normal do sistema [Leh96].

As pesquisas nessa área tiveram início no final da década de 60 com Lehman [Leh69], que, por mais de vinte anos, analisou a evolução do OS/360 da IBM e formulou as oito leis da evolução de software, que são amplamente aceitas até hoje. Porém, apesar de todo esforço empregado na compreensão do processo evolutivo, evoluir software de maneira rápida e eficiente ainda é um dos maiores desafios da engenharia de software [MD08].

Um conceito bastante citado quando o tema evolução de software é trazido à tona é o de envelhecimento de software ou *software aging* [Par94]. Segundo Parnas, todo software precisará ser modificado em algum momento de sua existência. Ele afirma que se um software é bom, de fato, ele necessitará ser modificado em algum momento. Por outro lado, software ruim não necessita de alterações, mas sim ser descartado.

A analogia entre seres humanos e software é sempre feita quando se fala em envelhecimento de software, como um artificio didático para entendimento do tema: os seres humanos sofrem desgaste do corpo e necessitam de intervenções médicas para se manterem vivos e exercendo suas atividades do cotidiano. Da mesma maneira, com o passar do tempo, sistemas de software necessitam de intervenções de especialistas para continuarem desempenhando suas atividades [Par94].

Outro termo bastante utilizado quando se fala em evolução de software é decaimento de código ou *code decay* [EGK+01]. Decaimento de código não pode ser desassociado de envelhecimento de software, pois, essencialmente, ambos dizem respeito ao processo de degradação que o software sofre com tempo.

Segundo Eick [EGK+01], os principais indicadores que demonstram que um software necessita ser evoluido são:

- Código extremamente complexo;
- Histórico de mudanças frequentes;
- Mudanças dispersas no código;
- Soluções temporárias para um certo problema; e
- Interfaces numerosas.

Os fatores que certamente causarão os problemas citados estão ligadas não somente à escrita de código, mas também a outros artefatos produzidos ao longo do processo de software seguido. Eick [EGK+01] também elenca alguns desses artefatos e outros fatores:

- Arquitetura inadequada;
- Violação dos princípios de *design* originais;
- Requisitos imprecisos;
- Pressão de tempo para desenvolver o produto;
- Ferramentas de programação inadequadas;
- Ambiente organizacional inadequado;
- Diferentes níveis de programadores; e
- Processo de mudanças inadequado.

O tipo de mudança que está sendo feita em determinado software deve ser levada em conta ao analisar um processo evolutivo. O padrão ISO/IEC [ISO91] para manutenção de software propõe quatro categorias que uma mudança pode se enquadrar:

- **Mudança Perfectiva:** Tem o objetivo de melhorar performance ou manutenibilidade.
- **Mudança Corretiva:** Tem o objetivo de corrigir falhas de funcionamento.
- **Mudança Adaptativa:** Realizada visando manter o sistema funcionando no caso de mudanças de ambientes e/ou plataformas.
- **Mudança Preventiva:** Visa prevenir futuros problemas em potencial.

Um grande problema inerente a evolução de software, diz respeito a carência de técnicas e ferramentas que auxiliem no controle do processo evolutivo [Sca03]. Dessa maneira, diversos grupos de pesquisa vêm estudando o tema, a fim de encontrar padrões evolutivos, e assim identificar possíveis soluções para problemas ligados ao processo evolutivo em si, ou pelo menos ou alternativas para tornar os efeitos colaterais menos danosos [LR06], [GVG04].

2.2 Métricas de Software

Uma Métrica é composta por medidas que avaliam um atributo de um objeto. Métricas de software, em especial, servem para medir tanto caracterísicas internas de um software como externas referentes ao processo de desenvolvimento [FP08].

Métricas internas descrevem basicamente complexidade estrutural. Medidas comuns de complexidade estrutural são métricas de tamanho, de estruturação, de fluxo de informação inter e intra estruturas, e acoplamento intermódulos. Complexidade estrutural é um exemplo de característica interna que pode ser medida utilizando métricas internas.

No que diz respeito a métricas de *design* de software, Lanza e Marinescu [LM06] afirmam que tanto classes como funções, métodos e variáveis, devem ser medidas em um *design* orientado a objetos com métricas de tamanho, qualidade e complexidade, de modo a gerar informações relevantes do projeto para outros membros da equipe a fim de averiguar que partes do software precisam ser otimizadas ou consertadas a fim de melhorar a qualidade do *design* como um todo. Vários conjuntos de métricas, como, por exemplo, CK (Chidamber and Kemerer) [CK94], MOOD (Metrics for Object-Oriented Design) [eAM96] e QMOOD (Quality Metrics for Object-Oriented Design) [BD02] foram concebidas com esse objetivo.

Já as características externas não possuem um escopo tão bem definido. Aspectos como modificabilidade, testabilidade e compreensibilidade se enquadram nessa categoria.

Percebe-se que são características bem mais abstratas do que as internas, o que acaba por dificultar a definição de ferramentas que as avaliem de maneira objetiva.

As caraterísticas externas são influenciadas diretamente pela internas. A complexidade ciclomática, por exemplo, é uma medida que afeta diretamente a testabilidade de um sistema: A complexidade ciclomática é medida segundo o número de caminhos linearmente independentes de uma função de um programa. [McC76]. Logo, uma complexidade ciclomática alta indica muitos fluxos, o que consequentemente implica em mais casos de teste de alta complexidade a serem averiguados.

No entanto, apesar de todo esforço empregado com validações e aperfeiçoamento das métricas, tanto a comunidade científica como a indústria não aceitam um valor numérico como sendo suficiente na avaliação de qualidade de um produto de software.

Brian Handerson-Sellers [HS96] compara o uso isolado de métricas com o trabalho de Tycho Brahe (1546-1601) que fez uma extraordinária observação numérica que descrevia os movimentos do sistema solar. Contudo, Brahe simplesmente compilou uma grande quantidade de números, que possibilitaram a identificação de uma série de padrões que, apesar de tudo não forneciam nenhuma explicação real sobre o que ele investigava. Só mais tarde, de posse dos dados de Brahe, Johannes Kepler (1571-1630) e Isaac Newton (1642-1727) desenvolveram modelos matemáticos que forneciam informações que possibilitaram a partir daí, de fato, interpretações sobre gravitação e uma série de leis puderam então ser formuladas, gerando informações relevantes para a sociedade.

Boehm [Boe81], traçando o mesmo paralelo para âmbito da computação, diz que o software não poderá ter seu Kepler e/ou Newton a menos que exista um Brahe com dados bem organizados de onde observações mais profundas e cientificamente comprovadas possam se basear [HS96].

Tudo isso mostra que medidas e observações analíticas, quando tratadas de maneira conjunta, fornecem informações com muito mais fundamentação e diminuem a margem de erros na análise de qualidade. Como acontece mais comumente com as ciências naturais, a computação deve utilizar métricas como ferramenta de validação de observações analíticas, e não tentar utilizá-las de maneira desconexa.

2.2.1 Classificação de Métricas de software

As métricas de software podem ser classificadas de acordo com quatro categorias:

Objetiva

Uma métrica objetiva depende apenas do objeto em questão e não do ponto de vista que está sendo interpretado. Por exemplo, o número de *commits* em um repositório é uma métrica objetiva, pois é obtida diretamente de forma automatizada ou por meio de uma contagem simples.

Subjetiva

Uma métrica subjetiva depende do objeto em questão e do ponto de vista de quem a está interpretado. Por exemplo, em uma escala de 0 a 10, qual a nota que um especialista dá para determinado *design*. Apesar do resultado final ser um valor numérico, a natureza é subjetiva, pois depende de opiniões pessoais, que variam de pessoa para pessoa. Vale observar que uma métrica subjetiva não pode ser obtida de forma automatizada.

Quantitativa

O valor de uma métrica quantitativa é representado por um número e pertence a um intervalo de certa magnitude. Isso permite que métricas quantitativas sejam comparadas entre si.

Qualitativa

Uma métrica qualitativa é representada por uma palavra, símbolo ou figura. Por exemplo, mediante a análise de um especialista, determinado *design* pode estar **muito bom**, **bom**, **aceitável**, **ruim** ou **muito ruim** [BZ07].

Alguns estudos empíricos em engenharia de software usam combinações entre métodos quantitativos e qualitativos. Normalmente, dados quantitativos são extraídos a partir de dados qualitativos, de modo a permitir a realização de análises estatísticas que outrora seriam impraticáveis [Gol06]. Vale citar que a classificação de uma métrica como quantitativa ou qualitativa é ortogonal à classificação como objetiva ou subjetiva. Geralmente uma métrica quantitativa é objetiva e uma qualitativa é subjetiva, mas isso não é sempre verdade [Eva04].

Mediante os conceitos apresentados, pode-se dizer que este trabalho se detém a analisar métricas objetivas quantitativas.

2.2.2 Métricas de Complexidade

A complexidade de software é considerada um dos fatores mais preocupantes quando se fala em qualidade de software. Muito tem se falado a respeito de complexidade de software, em especial de métricas que sirvam como parâmetros de indicação da complexidade.

Complexidade Ciclomática

Thomas J. McCabe foi um matemático que até hoje tem forte influência sobre os estudos com complexidade de software, por propor uma métrica objetiva quantitativa muito bem fundamentada em conceitos matemáticos, denominada Complexidade Ciclomática ou Métrica de McCabe [McC76].

Segundo a definição de McCabe, a complexidade ciclomática é calculada a partir da contagem de arestas do grafo de fluxo de execução de uma função (ou métodos em linguagens orientada a objetos) [McC76].

A complexidade ciclomática é definida segundo a Equação 2.1.

$$CC = E - N + 2P \tag{2.1}$$

onde CC = Complexidade ciclomática, E = Número de arestas do grafo, N = Número de nodos do grafo e P = Número de componentes conectados.

Outra maneira de determinar complexidade ciclomática é contando o número de *loops* fechados no grafo de fluxo e incrementando esse valor a cada ponto de saída, segundo a equação 2.2.

$$CC = LF + PS \tag{2.2}$$

onde, CC = Complexidade ciclomática, LF = Número de *loops* fechados e PS = Número de pontos de saída.

Para um grafo conectado com um único ponto de saída, a complexidade ciclomática será simplesmente o número de *loops* fechados + 1.

Formalmente, a complexidade ciclomática pode ser definida segundo a Equação 2.3.

$$CC := b_1(G,t) := \text{rank } H_1(G,t) \tag{2.3}$$

A leitura da Equação 2.3 é: *a primeira homologia do grafo G, relativa ao nodo terminal t*. Esta é a maneira técnica de dizer: *o número de caminhos linearmente independentes através do gráfo a partir de um fluxo de entrada para uma saída* onde:

- **Linearmente independente** corresponde a homologia, e não uma dupla contagem de *backtracking*;
- **Caminhos** corresponde a primeira homologia: um caminho é um objeto unidimensional;
- **Relativo** significa que o caminho deve começar e terminar no ponto de entrada ou de saída.

Weighted Method Count - WMC

WMC corresponde a soma ponderada dos métodos implementados dentro de uma classe. Ela é parametrizada de modo a calcular o peso de cada método. Uma possível métrica de peso é a complexidade ciclomática [Lin04].

Dessa maneira, tomando a métrica de McCabe como métrica de peso, o WMC usa a complexidade ciclomática para calcular o peso de cada método. Originalmente ela foi definida por Chidamber e Kemerer [CK94] como uma métrica para software orientados a objeto. No entanto, ela pode ser facilmente adaptadas para softwares não-objetos-orientado ao computar a soma ponderada das funções implementadas dentro de um módulo [Lin04].

Formalmente, WMC é definido segundo a equação 2.4, onde, M(c) corresponde ao conjunto de todos os métodos de uma classe c e CC(m) a complexidade ciclomática de um método específico $m \in c$

$$WMC(c) = \sum_{m \in M(c)} CC(m) \tag{2.4}$$

2.3 *Bad Smells*

Bad Smells são padrões existentes nos *designs* que não quebram o fluxo de execução de um programa, no entanto, são potenciais causadores de problemas futuros. O termo foi usado pela primeira vez por Martin Fowler et al. [Fow99], e partiu de uma catalogação de características presentes em *designs* de software que causaram problemas a longo prazo. Os autores chamaram esses pedaços de software como pontos de refatoração imediata. Vale a pena citar que estas refatorações devem ser feitas de modo que a estrutura interna do sistema seja melhorada, porém, sem que qualquer alteração externa e conseqüentemente as funcionalidades do sistema sejam alteradas [Sli05].

Alguns dos *bad smells* mais comuns nos softwares, que foram catalogados por Fowler e abordados nesse trabalho estão descritos abaixo:

- Comando *Switch*

  Frequentemente encontra-se o mesmo comando *switch* retornando em várias partes do código. Adicionar ou removê-los significa fazer mudança em todas as suas ocorrências.

- Métodos Longos

  Métodos, procedimentos e funções muito grandes são estruturas de difícil entendimento. Quanto maiores são estas estruturas, maiores serão as listas de parâmetros e variáveis presentes. Consequentemente, a complexidade de cada método, procedimento ou função será muito alta, indicando classes difíceis de testar e de manter. Estruturas menores proporcionam seu melhor entendimento como um todo e, por conseqüência, menor complexidade do sistema e maior possibilidade de reuso.

- Lista longa de parâmetros

  Em linguagens orientadas a objeto, os dados podem ser obtidos dos objetos se eles forem visíveis aos métodos, ou podem ser derivados fazendo requisições em outros parâmetros. No entanto, a lista de parâmetros não deve ser longa, pois assim elas se tornam de difícil entendimento e, no caso de efetuar mudanças, todas as referências para os métodos envolvidos devem ser alteradas também.

- Encadeamento de Mensagens

  Em códigos orientados a objeto frequentemente são encontrados encadeamento de invocação de métodos tais como m1.getm2().getm3().getm4(). Apesar do uso de variáveis temporárias como alternativa de diminuir o encadeamento, a dependência permanece. Se a estrutura de pelo menos um dos métodos for alterada, isso implica em mudanças a serem realizadas onde ocorrem os encadeamentos.

- Classe Preguiçosa

  Classes preguiçosas se caracterizam por delegar todas as suas atribuições ou grande parte delas a outras classes. Segundo Fowler, são classes que não são suficientes [Fow99]. Classes demandam tempo e esforço para serem construídas e mantidas. Logo, classes com esse comportamento devem ser eliminadas do projeto.

- Tamanho de Classe

  Classes grandes possuem muitas responsabilidades. Elas são responsáveis por um volume muito grande de operações e dados. Elas são difíceis de entender e de manter por conta do seu tamanho. A solução para tal é encontrar porções de código que trabalham em comum e colocados em uma nova classe. Um cuidado que deve ser tomado é de não deixar a classe muito enxuta e praticamente sem responsabilidades, o que causa o *bad smell* classe preguiçosa.

2.4 Qualidade de Software

A chamada Crise do Software teve início no final da década de sessenta e início da década de setenta. O termo expressava as dificuldades do desenvolvimento de software frente ao rápido crescimento da demanda e da inexistência de técnicas e padrões para validar e avaliar os sistemas. Uma das primeiras e mais conhecidas referências ao termo foi feita por Edsger Dijkstra, na apresentação do trabalho *The Humble Programmer* [Dij72], feita em 1972 na Association for Computing Machinery Turing Award, publicada no periódico *Communications of the ACM*.

Com os processos de negócio das corporações ficando a cada vez mais complexos, existia e ainda existe uma necessidade fundamental em incrementar os softwares para que eles

acompanhem a evolução das regras de negócio. Esta situação tem colocado o software em posição de destaque, fazendo com que ele assuma papéis estratégicos e decisivos. Isto indica a necessidade de uma atenção especial ao desenvolvimento e evolução dos produtos de software, objetivando satisfazer quem os usa, sem deixar de lado o fator qualidade [RMW01].

Procurando seguir metodologias e padrões de desenvolvimento, os softwares vêm sendo desenvolvidos por grandes equipes ao mesmo tempo, o que exige um esforço elevado no que diz respeito, segundo Lanza e Marinescu, a comunicação entre os membros da equipe de desenvolvimento, compatibilidade de tecnologia utilizadas e, acima de tudo isto, aspectos de complexidade. Ainda mais que, em se tratando de software, não existe a opção de construir-se uma vez, colocar em produção e nunca mais modificar[LM06]

De fato, o software é uma entidade um tanto quanto complexa. Uma mudança mal planejada em determinado módulo pode comprometer todo o sistema. Isto não está relacionado apenas a más práticas de programação, mas sim a um problema de superdimensionamento e consequente complexidade elevada. Obviamente ninguém conseguirá ter conhecimento detalhado de um grande sistema. Ainda mais, com os requisitos feitos a cada dia pelos usuários, inevitavelmente o sistema evoluirá para atender a estes novos requisitos e terá sua complexidade aumentada [LB85].

Tendo em vista todo este cenário, a busca por qualidade de software se tornou algo fundamental. Os especialistas em software concordam que existem duas maneiras de alcançar este controle [ISO91]: (1) estabelecendo padrões a serem seguidos no desenvolvimento de sistemas e (2) desenvolvendo medidas para monitorar a complexidade do código que estava sendo produzido, fazendo com que trechos de programas com baixa qualidade fossem reescritos. Como falado anteriormente, o uso de métricas de software é fortemente indicado pela literatura, nesse contexto, uma vez que elas foram pensadas exatamente com o objetivo de mensurar a qualidade do software sob vários aspectos.

2.4.1 Fatores de Influência na Qualidade de *Design* de Software

Shepperd [M.S92], diz que a qualidade de um software é diretamente proporcional a manutenabilidade e segurança. Já Pfleeger [HS91] iguala qualidade a segurança + disponibilidade + manutenabilidade. À definição de Pfleeger, Fenton [FP08] adiciona usabilidade e

Henderson-Sellers [HS92] reusabilidade. Li e Cheung [LC87] focaram em manutenabilidade que, segundo os autores, é definida em termos de compreensibilidade, modificabilidade e testabilidade. Após todos estes estudos, Henderson-Sellers [HS96] se reposiciona quanto aos fatores que influenciam a qualidade de um software, na tentativa de unificar todas estas teorias. A Equação 2.5 mostra essas relações.

$$\begin{aligned} Qualidade &= Seguranca + Disponibilidade + Compreensibilidade + Modificabilidade + \\ &\quad Testabilidade + Usabilidade \end{aligned} \tag{2.5}$$

A Figura 2.1, proposta por Li e Cheung em [LC87], mostra as interconexões entre características externas, em especial manutenibilidade, os fatores que a influenciam (compreensibilidade, modificabilidade e testabilidade) e complexidade.

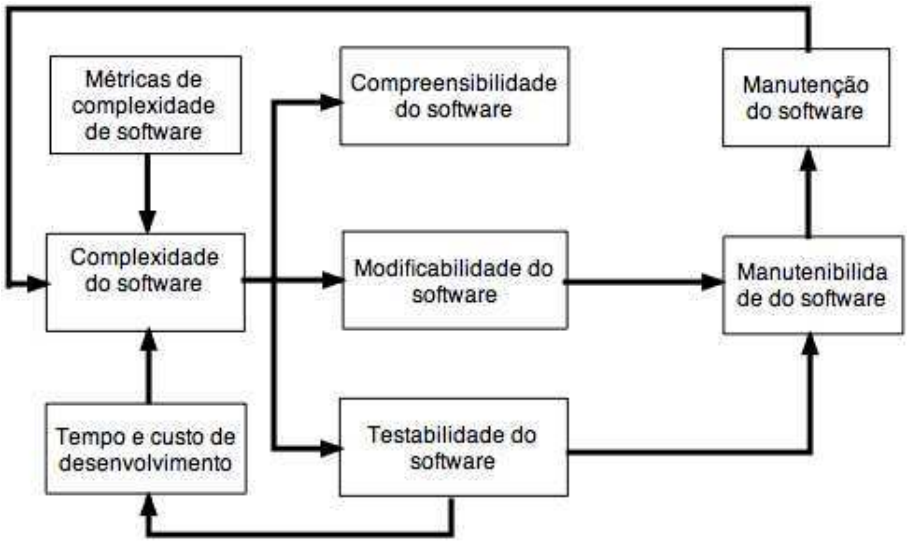

Figura 2.1: Interconexão entre características externas e internas de software. Adaptado de [LC87]

De maneira mais detalhada, pode ser observado que a complexidade do software é medida com métricas de complexidade. A complexidade influencia a compreensibilidade, a modificabilidade e a testabilidade do software, que, por sua vez, influencia tempo e custo de desenvolvimento, que também influencia a complexidade do software, formando assim um ciclo. Também pode ser observado que compreensibilidade e testabilidade influenciam diretamente a manutenibilidade do software. O quão mantível é um software irá determinar

todas as questões de manutenção propriamente dita, o que, juntamente com questões que influenciam testibilidade, afetam diretamente a complexidade do software. Isso demonstra que, indiretamente, a complexidade de um software influencia fatores que, de acordo com a Equação 2.5 afetam a qualidade do software.

De modo a sintetizar a relação entre métricas de complexidade e qualidade de software, a Figura 2.2 mostra como qualidade pode ser considerada uma combinação de segurança, disponibilidade, manutenibilidade e usabilidade. Por sua vez, manutenibilidade remete a compreensibilidade, modificabilidade e testabilidade. Como citado anteriormente, todas estas variáveis são influenciadas direta ou indiretamente pela complexidade do software.

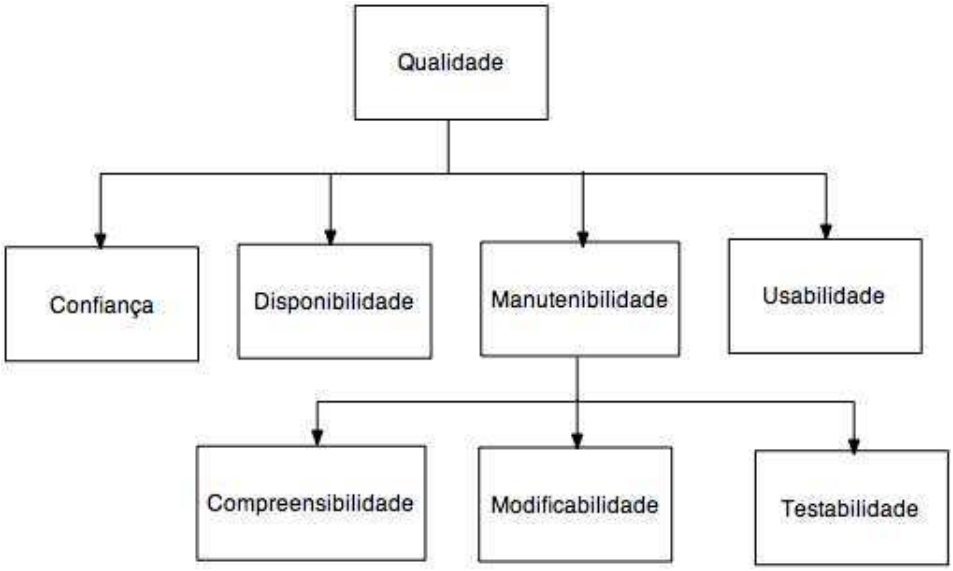

Figura 2.2: Características que afetam diretamente a qualidade de um software. Adaptado de [HS96]

2.4.2 Medindo a Qualidade de um *design*

Até o momento, no que diz respeito a características mensuráveis do software, o trabalho tratou sobre medir características internas e externas de um software, que podem ser de duas naturezas diferentes: (1) Provenientes do produto ou (2) do processo.

As medidas internas, que dizem respeito ao produto em si, corresponde à fotografia do software em um instante do tempo. Esta fotografia pode ser do *design* mais recente, ou de qualquer outro momento da história do software.

Já as medidas externas, que têm relação com o aspectos ligados principalmente com o processo, são bem mais complexas de serem definidas. Como exemplos de medidas que se enquadram nessa categoria, podem ser citados Pontos por Função, Medidas de custo por hora de trabalho das pessoas envolvidas em um processo de desenvolvimento, *bugs* por programadores, entre outras.

Como já mostrado anteriormente, existe uma relação, mesmo que indireta, muito forte entre métricas de produto e métricas de processo. Henderson-Sellers [HS96], propõem a Equação 2.6, trata deste ponto de maneira muito enfática quando trata de custo de produtos de software e as variáveis envolvidas.

$$\begin{aligned} Custo\,Total &= Custo\,com\,Ferramental + Custo\,de\,Produzir + \\ &\quad Custo\,para\,Manter + Custo\,de\,Executar\,o\,projeto \end{aligned} \tag{2.6}$$

onde,

$$\begin{aligned} Custo\,de\,Produzir &= Custo\,com\,Analise + Custo\,com\,\textbf{design} + Custo\,de\,Implementar + \\ &\quad Custo\,de\,Integrar + Custo\,com\,Testes \end{aligned} \tag{2.7}$$

Segundo a Equação 2.7, o custo de produção é influenciado por despesas com análise, *design*, implementação, integração e testes. O custo de produção por sua vez, como mostrado na Equação 2.6, influencia no custo total do projeto de software, juntamete com custo com ferramental, manutenção e execução do projeto.

Custo com *design* e custo com manutenção são influenciados, segundo o que foi mostrado até agora, com a complexidade estrutural do software. Assim, mediante um encadeamento lógico, percebe-se o quanto não dar a devida atenção a complexidade de software afeta o custo de produção, o custo total e a qualidade do software como um todo.

2.4.3 Discussão

Com toda essa necessidade de mensurar complexidade de software, não apenas McCabe [McC76], mas também Halstead [Hal77], Henry e Kafura [Kaf85], dentre outros, propuseram medidas que poderiam refletir esta característica que precisava ser controlada. No

entanto, várias críticas foram feitas a estas métricas. Curtis [Cur99], em especial, faz as seguintes observações neste sentido:

- Existe um número grande de métricas que propõe aferir complexidade;
- Os cientistas definem complexidade só depois de terem desenvolvido suas métricas;
- Muito tempo foi gasto no desenvolvimento de preditores de complexidade, quando, de fato, o maior esforço deveria ser empregado em critérios de desenvolvimento, como manutenibilidade, compreensibilidade e a operacionalização destes critérios;
- Para cada validação positiva existe uma negativa.

Zuse [Zus94] diz que as medidas de complexidade são confusas e não satisfazem o usuário, por não fornecerem informações de fácil interpretação.

Todo esse ceticismo partiu da indústria, que, com toda praticidade e imediatismo, entende que mensurar software é algo muito utópico. Essa idéia é apoiada por renomados pesquisadores em engenharia de software, como, Kent Beck e Martin Fowler, que se destacam como contrários a o uso de métricas como sendo suficientes na análise de qualidade de *design* de software. Eles são defensores da necessidade do papel do especialista na avaliação de qualidade, uma vez que números por si só não avaliam aspectos subjetivos de um *design* de software, os quais são influnciadores determinantes na estrutura geral do software. Foi a partir daí que, em conjunto com outros renomados na área de engenharia de software, Beck, Fowler e outros engenheiros de software escrevem sobre *bad smells* [Fow99].

O que se tem então são duas escolas que discordam da maneira adequada de avaliar a qualidade de um *design*. Isso acaba por dificultar a determinação padrões de qualidade de *design* de software.

2.5 Análise de Correlação Simples

A ferramenta estatística apresentada nesta seção, bem como a apresentada na Seção 2.6 são utilizadas na verificação de existência de correlação e relação de causa-efeito entre as variáveis relacionadas a complexidade, *bad smells* e *bugs*.

O coeficiente de correlação linear r ou $r(X, Y)$, proposto por Karl Pearson, é calculado a partir de uma amostra de *n* pares de observações de X e Y, e mede a quantidade de dispersão em torno da equação linear ajustada através do método dos mínimos quadrados, ou grau de correlação entre as variáveis, na amostra [Was04].

O coeficiente de correlação r é definido como a razão entre a covariação e a raiz quadrada do produto das variações de X e de Y, conforme mostra a equação 2.8.

$$r = \frac{\sum(X - X')(Y - Y')}{\sqrt{[\sum(X - X')^2][\sum(Y - Y')^2]}} \tag{2.8}$$

O valor de *r* é uma medida cujo valor se situa no intervalo compreendido pelos valores -1 e +1. As Figuras 2.3, 2.4, 2.5 e 2.6 mostram os diagramas de dispersão para quatro casos de possíveis valores de r.

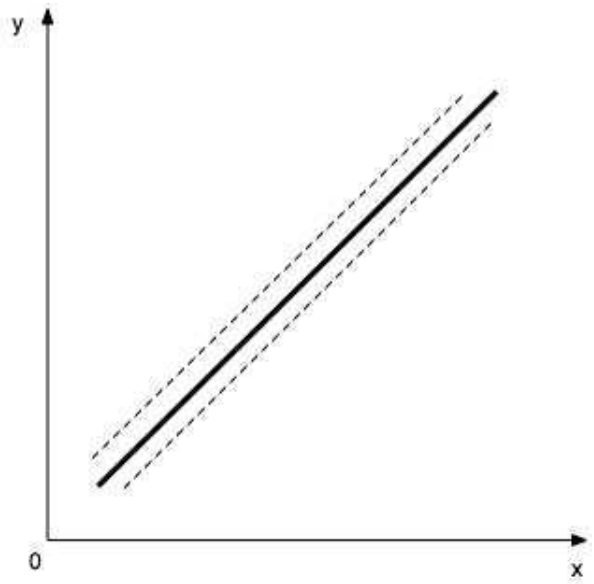

Figura 2.3: Diagrama de dispersão indicando forte correlação positiva

Quanto maior a qualidade do ajuste da reta proposta aos pontos do diagrama de dispersão, mais próximo de +1 ou -1 estará o valor de r. Conforme mostrado na tabela 2.1, +1 é o valor de máxima correlação direta e -1 o valor de máxima correlação inversa entre as variáveis. Não havendo relação linear alguma entre X e Y, $r = 0$.

Vale citar que r independe das unidades de medida das variáveis X e Y, ou seja, é um número adimensional e também que independe da origem à qual os valores que o compõem são calculados, ou seja, somando-se ou subtraindo-se um valor constante e arbitrário a cada

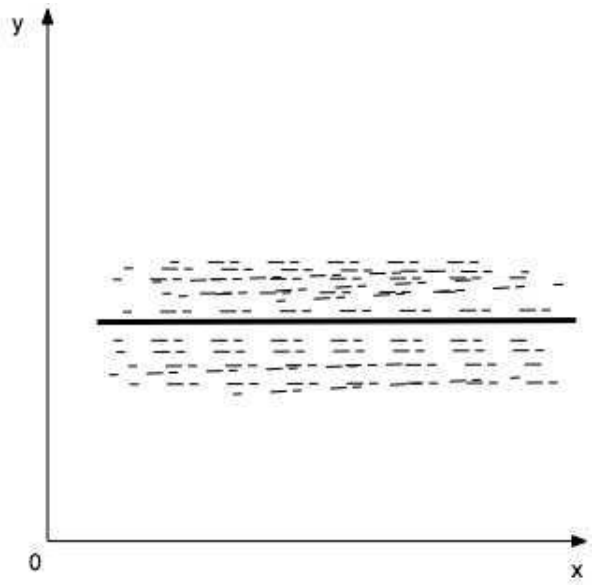

Figura 2.4: Diagrama de dispersão indicando ausência de correlação

Tabela 2.1: Tabela de significância de índice de Correlação.

Índice de correlação *r*	**Significância**
-1	Máxima correlação inversa
-0,99 a >0,7	Correlação inversa muito forte
-0,7 a <0,0	Correlação inversa fraca
0,0	Inexistência de correlação
>0,0 a 0,7	Correlação direta fraca
>0,7 a -0,9	Correlação direta muito forte
+1	Máxima correlação direta

valor de X ou Y, ou de ambas, o coeficiênte de correlação não se altera [Was04].

2.6 Teste de Hipótese e P-Valor

Quando um teste estatístico é aplicado a um conjunto de dados a fim de verificar determinado resultado, existe a possibilidade daquele teste não ter sido preciso o suficiente e existir a necessidade de uma comprovação de que o resultado, de fato, condiz ou não com o esperado. Para averiguar a veracidade de um teste estatístico, são feitos os chamados testes de hipótese.

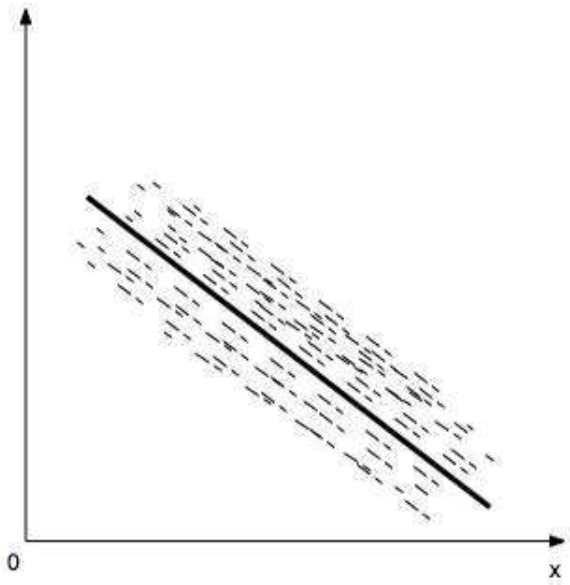

Figura 2.5: Diagrama de dispersão indicando forte relação negativa

Seja uma hipótese validada por um teste estatístico qualquer. Essa hipótese possui apenas duas alternativas: (1) ser aceita ou (2) rejeitada. Em outras palavras, estar correta ou errada. Para realizar a verificação com o p-valor da veracidade do resultado provido pelo teste, as duas hipóteses devem ser consideradas. Em especial, na consideração de H_0 como falsa, essa hipótese é chamada Hipótese Nula.

A hipótese nula H_0 é uma hipótese que é presumida verdadeira até que provas estatísticas sob a forma de testes de hipóteses indiquem o contrário. Por exemplo, no caso de querer saber se uma determinada moeda é equilibrada, ou seja, quando atirada ao ar a probabilidade de sair caras ou coroas é igual a 1/2. Inicialmente, não se tem nada que nos indique o contrário. Assim, a hipótese inicial - chamada Hipótese Nula - é que a moeda é equilibrada. Para testar essa hipótese, opta-se por fazer uma amostra de 100 lançamentos e com base no resultado decide-se se aceita-se ou rejeita-se a Hipótese Nula H_0.

A análise do p-valor, é uma das maneiras mais utilizadas para verificar a corretude de testes estatíticos [Was04].

P-valor, é a probabilidade de que uma amostra podia ter sido tirada de uma determinada população, assumindo que a hipótese nula seja verdadeira. Por exemplo, um valor de 0,05 por exemplo, indica que existe uma probabilidade de 5% de que a amostra a testar possa ser tirada, assumindo que a hipótese nula é verdadeira.

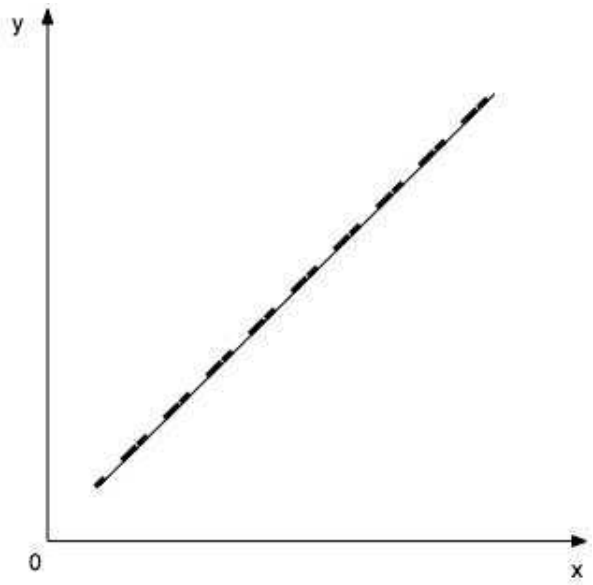

Figura 2.6: Diagrama de dispersão indicando relação inear perfeita

Informalmente, o p-valor é uma medida de evidência contra H_0: quanto menor for o p-valor, mais forte é a evidência contra H_0. A Tabela 2.2 apresenta a escala normalmente utilizada na análise do p-valor.

Tabela 2.2: Escala de evidência do p-valor

p-valor	**evidência**
< .01	Evidência muito forte contra H_0
.01 a .05	Evidência forte contra H_0
.05 a .10	Evidência fraca contra H_0
> .1	Nenhuma ou evidência muito fraca contra H_0

É importante rassaltar que um p-valor grande não é uma forte evidência em favor de H_0. Um p-valor grande pode ocorrer por duas razões: (1) H_0 é verdade ou (2) H_0 é falso mas o teste estatístico é fraco ou não é adequado para aquele conjunto de dados.

No contexto deste trabalho o p-valor foi utilizado na comprovação e validação das correlações encontradas entre as variáveis analisadas.

Capítulo 3

Evolution Miner

Cientistas da computação defendem que uma métrica por si só não é uma fonte confiável de informações relevantes sobre aspecto de qualidade de um software, uma vez que sistemas computacionais são compostos de várias partes como, por exemplo, projeto arquitetural, documentação e código fonte, e que apenas a análise conjunta de métricas que avaliem estes e outros aspectos, sob a supervisão de um especialistas, é que proporcionará diagnósticos mais eficientes e eficazes [LM06], [Fow99]. Outros cientistas afirmam que, mesmo que vários aspectos sejam analisados, uma análise mais completa de qualidade é feita quando várias versões de um software são avaliadas [GVG04]. A justificativa é que uma análise histórica das métricas revelam quais valores não estavam dentro dos padrões determinados como bons e que consequências isso trouxe ao produto e ao processo de software como, por exemplo, perda de usuários, dificuldade de refatoração de *features*, *bugs* difíceis de serem resolvidos, dentre outras. Contudo, a maior dificuldade encontrada por cientistas que trabalham com qualidade de software diz respeito ao suporte ferramental de extração de dados de sistemas de software [LR06].

De modo a viabilizar a pesquisa apresentada neste trabalho, bem como contribuir com o suporte ferramental para pesquisas com qualidade de software, desenvolvemos o Evolution Miner que é uma API feita em Java, que faz o papel de uma estrutura intermediária entre repositórios de software e aplicações-cliente de manipulação dos dados extraídos, a princípio com o propósito de avaliar a qualidade do software a partir de dados extraídos de várias partes do software.

De maneira geral, o funcionamento do Evolution Miner está representado na Figura 3.1

e pode ser entendido da seguinte maneira: Na aplicação cliente são especificadas as versões e as métricas que devem ser extraídas de um determinado software. Essas informações são passadas para o Evolution Miner (seta A), que por sua vez recupera do repositório as informações requeridas (setas B e C) e as retorna para o cliente (seta D).

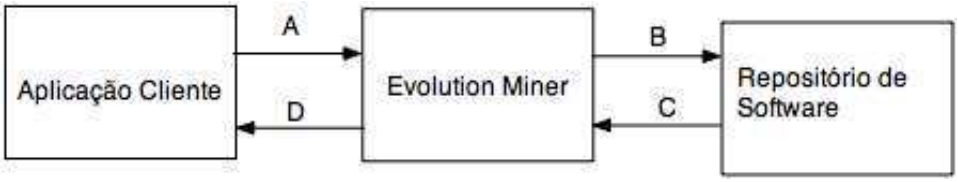

Figura 3.1: Visão geral do Evolution Miner.

3.1 Arquitetura

O Evolution Miner possui três camadas:

- **JavaCVS:** É uma ferramenta desenvolvida pelo NetBeans [Jav] e incorporada a nossa API, que é responsável por fazer a comunicação do Evolution Miner com os repositórios. Vale citar que o fato de utilizar o JavaCVS, limita a ferramenta a investigar apenas repositórios CVS de programas escritos em Java

- **Filtros:** O javaCVS não realiza filtragem por versões. Assim, o Evolution Miner possui uma estrutura que, como o nome sugere, é responsável por eliminar os dados vindos do repositório que não foram especificados na aplicação cliente; e

- **Extrator:** É o elo central da API que faz a intermediação entre a aplicação cliente, o repositório e os filtros;

Na Figura 3.2 pode ser observado o diagrama de componentes da ferramenta. Adicionalmente, podem ser vistos também a aplicação cliente, o repositório de software, bem como setas enumeradas que indicam a ordem de tráfego entre as estruturas, detalhados a seguir.

O Evolution Miner oferece uma interface para aplicação cliente, que por sua vez especifica o endereço do repositório do software a ser analisado, bem como os dados de acesso. Também é oferecido um esquema de filtros a serem utilizados na mineração dos dados. A

aplicação cliente pode personalizar a busca por meio de cinco opções de filtragem: Por autor(es), Data(s), Intervalos de tempo, Palavra(s) chave de *commits* e nomes e versões de arquivos. Esses filtros podem ser usados separadamente ou combinados. Uma vez especificados as informações básicas de acesso, o Evolution Miner estabelece a comunicação, através do JavaCVS [Jav], com o repositório especificado, recupera filtra o que é de interesse do cliente e retorna o que foi requisitado.

Uma vez que o extractor recebe as informações da aplicação cliente, ele repassa para o JavaCVS, que recebe as mesmas informações providas pelo cliente, e se comunica com o repositório especificado, conforme as setas 2 e 3 respectivamente. O repositório retorna para o JavaCVS uma lista com metadados dos arquivos requisitados (seta 4), a envia, juntamente com os dados de filtragem fornecidos pelo cliente, para o *extrator* (seta 5), que os envia para a camada de *Filtros* onde são realizadas as filtragens (seta 6). A opção por trazer apenas os metadados nesse instante é um diferencial da nossa ferramenta, uma vez que os arquivos reias só são minerados quando se sabe o que, de fato será minerado.

Após a filtragem dos metadados, restam apenas aqueles referentes ao que foi especificado como parâmetros na aplicação cliente. Uma vez que agora sabe-se, de fato, que arquivos devem ser retornados, a camada de *Filtros* reenvia essas informações para o *Extrator* (seta 7), que repete o processo de comunicação com o JavaCVS (seta 8), que por sua vez requisita do repositório apenas os arquivos que, de fato, são de interesse do cliente (seta 9). Vale citar que nesse ponto o repositório retorna não mais metadados, mas uma lista com os arquivos de fato (seta 10). essa lista é retornada para o extractor (seta 11), que a envia finalmente para a aplicação cliente (seta 12).

3.2 Estrutura do método de comunicação entre o Evolution Miner e a Aplicação Cliente

A comunicação entre a API e a aplicação cliente é feita através do método *getVersions*, que recebe como parâmetros uma lista de *retConfigs* e outra lista com *VCSConfigs* e retorna uma lista de *RemoteFiles*: **List<RemoteFiles> getVersions (<VCSConfigs>, <RetrieveConfigs>)**. De maneira detalhada, este método é estruturado da seguinte maneira:

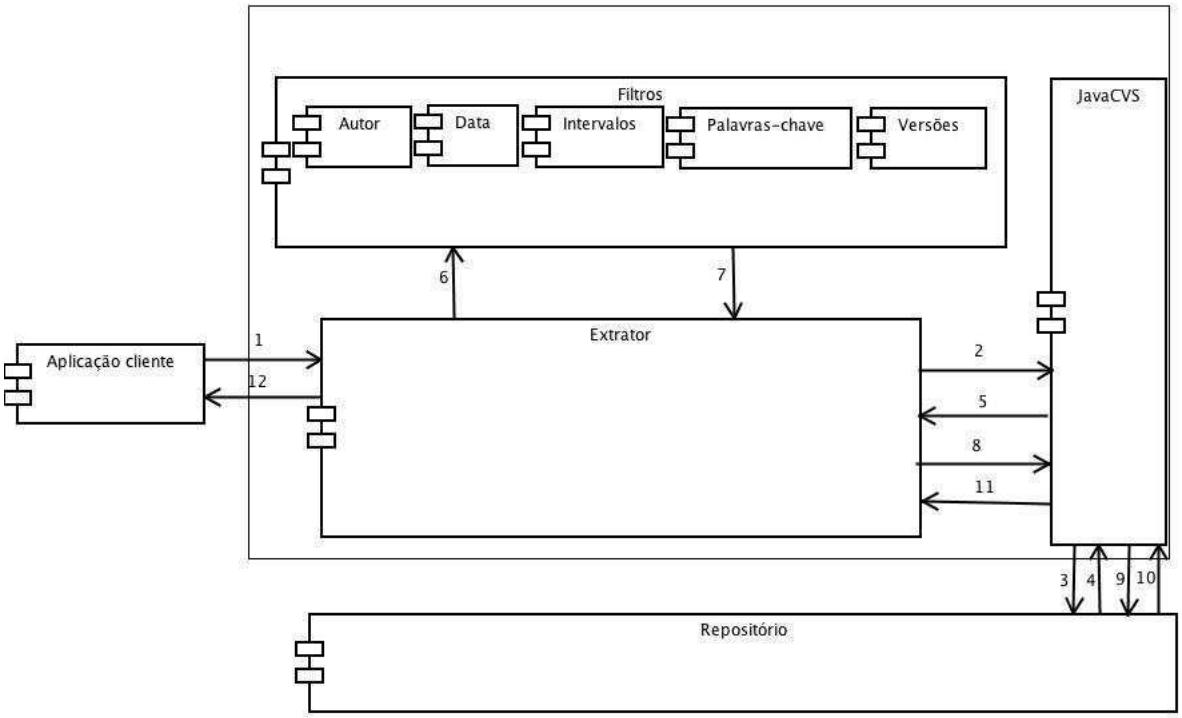

Figura 3.2: Diagrama de componentes do Evolution Miner.

- cada *RemoteFile* da lista retornada por *getVersions* corresponde a um arquivo específico: **RemoteFile1** corresponde a **Classe1**, **RemoteFile2** corresponde a **Classe2** e assim por diante.

- Cada *RemoteFile* por sua vez possui uma lista de *FileRevision*: Por exemplo, *RemoteFile1* contém *FileRevision1 = Classe1 v1* e *FileRevision2 = Classe1 v2*. RemoteFile2 contém *FileRevision1 = Classe2 v1* e *FileRevision2 = Classe2 v2*.

- A lista de *retConfigs* recebe como parâmetros as informações que serão utilizadas na filtragem do dados; já a lista de *VCSConfigs* recebe informações para acessar o repositório como endereço do repositório, *login* e senha do usuário.

É a partir da chamada ao método *getVersion* que a comunicação é iniciada e finalizada, através do envio de informações de requisição, no sentido da seta um da figura 3.2. O fato da aplicação cliente se comunicar com o Evolution Miner através de apenas um método é uma estratégia que facilita o uso da API, uma vez que, quem implementa as aplicações clientes, terá que tratar apenas de um ponto do sistema que trata tanto da solicitação como da recuperação dos dados.

3.3 Evolution Metrics Miner - Uma aplicação cliente do Evolution Miner

Uma vez que tínhamos uma API pronta, precisávamos de uma maneira para testá-la. Como o objetivo do trabalho era minerar repositórios a fim de investigar padrões de qualidade ao longo da evolução de software, desenvolvemos uma aplicação cliente que servisse como *front-end* para o Evolution Miner que minerasse a métricas de interesse para a pesquisa. Foi assim que surgiu o *Evolution Metrics Miner* ou simplesmente EMM. Essa era também uma maneira de validar a API.

O EMM tem por objetivo extrair métricas de tamanho (LOC), complexidade (WMC) e Bad Smells, do histórico de versões de projetos de software minerados por meio do Evolution Miner. A Figura 3.3 mostra o diagrama de componentes do EMM.

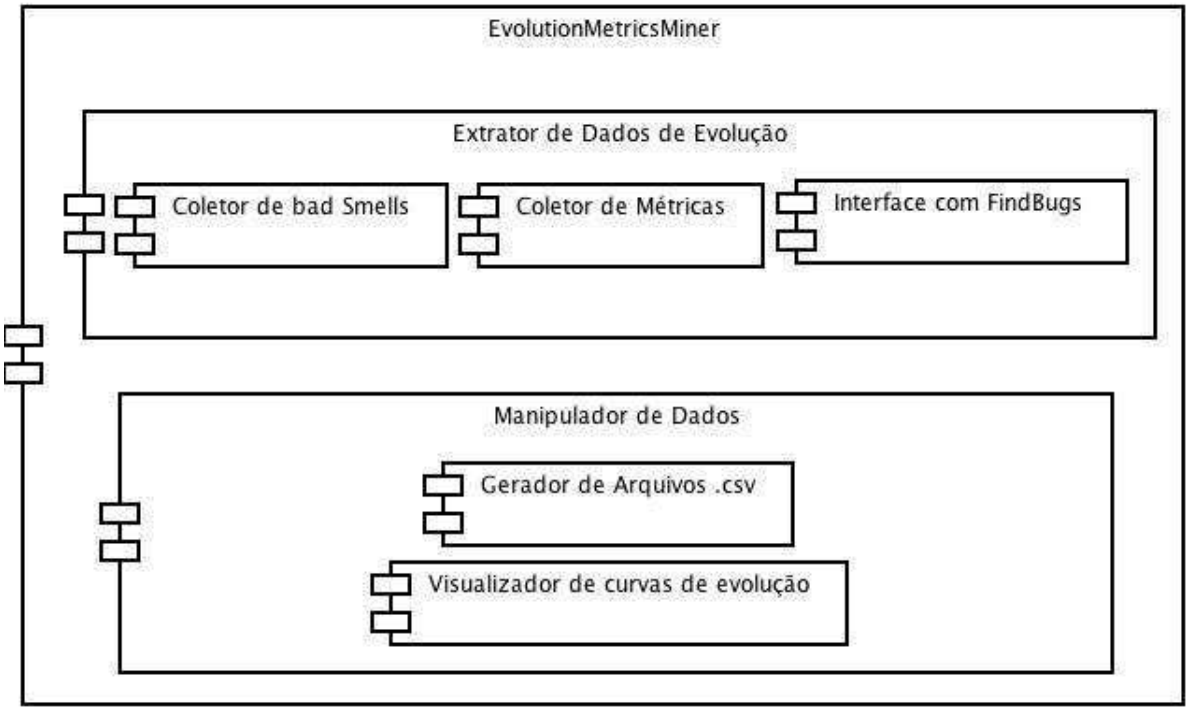

Figura 3.3: Diagrama de componentes do *Evolution Metrics Miner*.

Para coletar as métricas foram acopladas ao EMM as funcionalidades providas pelo plugin do Eclipse Eclipse-Metrics [oF]. Já para coletar os *Bad Smells*, foi utilizado o PMD [PMD]. Para coletar os dados providos pelo FindBugs, foi feita uma interface de comunicação que recebia os dados das versões de arquivos de interesse. Os módulos responsáveis

pela extração destes dados foram agregados em uma camada chamada *Extrator de Dados de Evolução*, conforme a Figura 3.3.

Uma vez que esses dados são extraídos do código fonte, eles são armazenados em um arquivo do tipo *comma separeted value (csv)*. Isso permite a manipulação dos dados e conseguinte análise deles sob várias perspectivas. Com os dados amazenados em arquivos .csv, utilizamos o JFreeChart para gerar os gráficos evolutivos das métricas a serem analisadas. Os módulos responsáveis por estas tarefas foram agregados em um módulo chamado *Manipulador de Dados*, conforme a Figura 3.3.

Tanto o Evolution Miner como o Evolution Metrics Miner estão em processo de melhoria estrutural e em breve estarão disponíveis na página do Grupo de Métodos Formais d UFCG, no *link* http://www.gmf.ufcg.edu.br .

Capítulo 4

Análise da relação de causa-efeito entre qualidade e complexidade de *design* de software

A evolução das pesquisas com qualidade de software vem mostrando que, em particular, o fator complexidade exerce uma forte influência na qualidade [ISO91]. Assim, grandes esforços têm sido empregados em pesquisas que tratam de identificar pontos-chave que sejam fontes de complexidade em software, bem como em elaborar estratégias de abordar esses pontos, com objetivo de reduzir a complexidade do software e, consequentemente, melhorar a qualidade.

De acordo com Robert L. Glass, editor da *Elsevier's Journal of Systems and Software* e da *The Software Practitioner newsletter*, a cada 10% de acréscimo de complexidade em um software, o esforço para solucionar os efeitos danosos causados por este aumento na complexidade é aumentado em 100% [Gla01].

Lanza e Marinescu destacam os esforços de muitos pesquisadores nas últimas duas décadas, que têm trabalhado na identificação e formulação de princípios de *design* [Mey88], [Lis87], [Mar02]; regras [CY91], [Mey88]; e heurísticas [Rie96], [JF88], [Lak96], [LR89] que poderiam ajudar desenvolvedores com o controle de complexidade e consequente melhoria da qualidade de *design* de software.

Fowler [Fow99] e toda a comunidade *anti-patterns* [BMHWMM98] trabalham em identificar pontos em que o *design* deve ser melhorado. Além dos *bad smells* elencados por

Fowler, muitas outras características que "cheiram mal"em um código devem ser listadas em uma análise dessa natureza, partindo sempre do olhar minuncioso de especialistas em *design*.

Mediante todas estas abordagens listadas, Lanza e Marinescu [HS96] afirmam que seguir a abordagem de Fowler, no que diz respeito a tratar qualidade de *design*, identificando e descrevendo os sintomas de um mau *design* é uma estratégia bastante viável na disseminação do tratamento de qualidade de software.

No entanto, caracterizar todo um sistema de software nos seus variados aspectos é um trabalho bastante oneroso e custoso. Consequentemente, seria falta de senso acreditar que algumas métricas serviriam como parâmetros de caracterização únicas de todo um sistema. Todavia, caracterizar um sistema é possível sim, se os meios de mensuração corretos forem utilizados, i.e., se as medidas em questão forem além das tradicionais medidas de tamanho e correlacionar esse resultados de maneira apropriada [LM06].

Apesar de existir certa concordância na comunidade científica de que a elimininação de *bad smells* e a redução da complexidade de software está diretamente relacionada com a melhoria da qualidade de sistemas computacionais, este capítulo apresenta experimentos realizados em treze sistemas de software, buscando encontrar relações de causa efeito entre número de *bad smells* e o valor da complexidade ciclomática ao longo da evolução deles. Os resultados encontrados indicam que, apesar da forte correlação entre estas variáveis, não existe relação de causa efeito entre elas. Isso fortalece ainda mais a idéia de que qualidade de software é uma questão bastante complexa e que não adianta controlar uma variável que a influencia, e esperar que outros fatores de influencia sejam alterados, em particular *bad smells* e complexidade em termos de complexidade ciclomática.

4.1 Artefatos e Métodos para Realização da Pesquisa

Para a realização dos experimentos mencionados, utilizamos um conjunto de versões de vários sistemas para coletar os dados. Optamos por selecionar tanto repositórios de software *open source* como proprietários, o que possibilitaria que os nossos resultados fossem os mais gerais possíveis.

Uma vez que têm-se os respositórios disponíveis, necessitavamos de uma ferramenta que minere os respositórios de software e então, sejam coletadas as métricas de cada versão

dos sistemas a serem analisadas no estudo. Como o estudo trata essencialmente do fator qualidade, coletar bugs cadastrados em bug reports é importante e necessário pois a melhoria na qualidade de software implicaria em redução de bugs.

Necessitávamos também uma ferramenta para eliminar os *bad smells* de versões dos sistemas escolhidos a fim de verificar se esta redução implica em diminuição da complexidade ciclomática do sistema.

No que diz respeito a análise dos dados, são necessárias ferramentas que gerem informações visuais que permitam a visualização das curvas de evolução de cada métrica minerada. No entanto esses gráficos servem apenas como fontes de identificação de padrões comportamentais de evolução, que são importantes em uma análise mais geral e preliminar. Já informações mais precisas e contundentes são conseguidas quando os dados são submetidos a análises estatísticas apropriadas.

As etapas seguidas na realização dos experimentos descritos neste capítulo são: Seleção dos repositórios *open source* e proprietários, extração de *bad smells* e métricas das versões dos repositórios de software, coleta de informações de *bugs* cadastrados em *bug reports*, geração das curvas de evolução, eliminação de *bad smells* visando controlar métricas de complexidade e, finalmente, análise estatística dos dados e verificação da não existência da reação de causa-efeito entre número de *bad smells* e complexidade ciclomática. Cada um desses passos serão detalhados nas subseções seguintes.

4.1.1 Seleção dos repositórios

Para ser escolhido, um software precisava obedecer os seguintes critérios:

1. **Ser largamente utilizado pela comunidade -** Mediante análise da média de *downloads* dos sistemas mais citados em fóruns, listas de discussões e nas próprias páginas dos projetos, para ser analisado, um software precisaria ter uma quantidade mínima de 15000 *downloads* (Esse número é baseado na média de *downloads* dos projetos mais populares do S*ourceforge - http://www.sourceforge.net e Apache - http://www.apache.org*, de onde foram baixados a maioria dos softwares analisados neste trabalho). Isso indicaria que o software atendia à necessidades em determinado segmento de atuação, e a existência de demanda de melhorias contínuas até que ele

caia em desuso. Tudo isso permitiria que as métricas de interesse fossem coletadas.

2. **Ser escrito em Java -** Grandes e importante sistemas da atualidade são escritos em Java. Trabalhar com essa linguagem possibilitaria um maior leque de opções no processo de escolha dos sistemas a serem analisados.

3. **Ter o código fonte compilado, bem como todos os outros arquivos que faziam parte do projeto -** O fato de que grande parte das ferramentas de extração de métricas exigiam códigos previamente compilados como entrada, foi determinante na escolha deste critério.

4. **Ter o *bug report* disponível -** Este critério é fundamental para coletar os *bugs* notificados. Para os softwares legados, não houve disponibilização dos *bug reports*. Assim, este critério era necessário apenas para os sistemas *open-source*.

5. **Ter pelo menos três anos de lançamento da primeira *release* -** Como nosso interesse é analisar métricas de qualidade no contexto de evolução de software, um sistema que tivesse pelo menos três anos de uso teria um número razoável de versões para uma análise mais detalhada, segundo a consulta prévia a estatísticos.

6. **Possuir pelo menos vinte e cinco versões -** Vinte e cinco versões de cada software seria uma quantidade mínima no que diz respeito a dados suficiente para realizar análises estatísticas válidas.

Uma descrição mais detalhada dos projetos *open source* selecionados, como nome, idade e informações estruturais de tamanho em termos de Linha de código (LOC), pacotes, classes e métodos estão na Tabela 4.1. As informações de tamanho mostram os valores máximos e mínimos que cada variável assume ao longo da vida do software. O software mais antigo analisado foi a JDK, com treze anos. Já os mais novos tinham cinco anos, que é o caso do Azureus, *FindBugs* e *Spring*. De maneira geral, no que diz respeito a tamanho, tanto em linhas de código, números de pacotes, classes e métodos, o maior software analisado foi o ZK, e menor, o Saxon.

No que diz respeito aos sistemas proprietários, investigamos cinco projetos de grande porte. Os critérios de seleção foram passados para a empresa que forneceu os sistemas.

Tabela 4.1: Dados dos projetos *open source*.

Software	Idade (anos)	LOC		Pacotes		Classes		Métodos	
—	—	min	max	min	max	min	max	min	max
Azureus	5	31629	423852	25	372	176	2494	1175	19371
FindBugs	5	42608	164183	9	41	354	1033	2408	7479
Hibernate	7	47292	151912	28	77	400	1242	3966	12742
JDK	13	152245	2034858	27	388	717	10151	5916	94516
Saxon	7	22402	144629	20	36	369	1309	3236	12475
Spring	5	40698	262808	52	204	401	1953	2357	13643
Tomcat	9	30020	211421	52	147	652	1833	5666	18108
ZK	7	2564751	12034858	371	1628	2499	10419	17569	67376

Por questões de sigilo, apenas algumas informações básicas sobre os sistemas, mostradas na Tabela 4.2, podem ser externadas. No entanto, pode ser observado que os sistemas são menores do que os *open-source*.

Tabela 4.2: Dados dos sistemas proprietários.

Software	Idade (anos)	LOC		Pacotes		Classes		Métodos	
—	—	min	max	min	max	min	max	min	max
Software Proprietário 1	> 4	11552	22064	30	45	112	157	584	945
Software Proprietário 2	> 3	48901	124123	35	51	323	667	2346	6122
Software Proprietário 3	> 6	20643	52988	18	23	119	174	1285	2133
Software Proprietário 4	>4	8230	219360	13	67	57	816	518	8896
Software Proprietário 5	>5	15094	44237	26	31	105	190	1149	2625

4.1.2 Coleta de Dados

Uma vez que os projetos haviam sido selecionados, utilizamos o EMM, apresentado no capítulo 3 para extrair os dados a serem analisados. Como estávamos interessados em analisar métricas de complexidade, mineramos WMC, Métrica de Halstead e Métrica de princípio de dependência acíclica.

No que diz respeito a *bad smells*, o conjunto de dados minerados inclui más práticas de programação, estilos de código, convenção de codificação, código morto, código duplicado, código "espaguete", dentre outras características indesejáveis a qualquer código, elencadas por Fowler et al. [Fow99], por outros especialistas em *design*.

Como mencionado anteriormente, este trabalho também inclui a análise dos *bugs* ao longo da vida dos softwares selecionados. Para tanto, foram coletados esses dados de duas fontes: (1) a partir dos *bug reports* dos projetos e (2) utilizando o *FindBugs*. Para coletar os dados dos *bug reports* dos sistemas *open source*, realizamos contagem dos *bugs* reportados pelos usuários dos sistemas e que possuiam notificação de confirmação da equipe de desenvolvimento do sistema de que o problema reportado era, de fato, um *bug*. Os experimentos com os dados dos*bugs reports* foram realizados apenas os sistemas *open-source*, uma vez que a empresa onde realizamos os experimentos com os softwares legados, por conta de sigilo, não deram acesso aos *bug reports* dos projetos.

A respeito do *FindBugs*, o seu uso foi estimulado pelo fato de ser este largamente utilizado tanto em ambientes de pesquisa como comercial, e por ser capaz de categorizar os dados como *bugs* ou *bad smells* seguindo a convenção adotada:

- *Bugs:* São erros, falhas, faltas ou qualquer outro problema de software que impeça o programa de se comportar conforme esperado.

- *Bad Smells*: Não quebram o fluxo de execução normal do software, mas são características indesejáveis a qualquer *design*, que, certamente, comprometerá performance e dificultará a refatoração.

A contagem de *bugs* fornecida pelo FindBugs era computada em uma categoria diferente daquela dos *bugs* coletados nos *bug reports*.

4.1.3 Controle de *Bad Smells*

Como tínhamos interesse em verificar se os *bad smells*, de fato, influenciam a complexidade do software, manipulamos os códigos-fonte dos projetos, de modo a remover os *bad smells* encontrados, tanto pelo PMD, como pelo FindBugs. Como esse trabalho era bastante custoso, nos limitamos a eliminar esse conjunto de dados apenas para o Hibernate e o Spring. Esse dois sistemas foram os escolhidos, pois eram aqueles com maior número de versões, o que proporcionaria posteriores análises estatísticas mais factíveis. Uma vez que removemos boa parte dos *bad smells* desses sistemas, executamos a ferramenta novamente, de modo a verificar se os indicativos dos *bad smells* removidos haviam desaparecido.

4.1.4 Análise do Conjunto de Dados

Os gráficos gerados pelo JFreeChart foram fundamentais na análise do conjunto de dados minerados pois, através deles, foram feitas as primeiras identificações da existencia de padrões nas métricas coletadas. Esta identificação prévia motivou o uso de ferramentas estatísticas e consequentes análises mais aprofundadas.

Além da análise gráfica, foi investigado também qual método estatístico seria relevante para verificar as relações entre os dados minerados. O mais apropriado foi a Análise de correlação simples. Como já citado, esta análise deve ser realizada entre duas variáveis e retorna um valor que indica a força ou grau de relação linear entre elas [KKNM07].

Para calcular esta correlação entre os dados minerados dos projetos, utilizou-se um software estatístico chamado R-Project ou simplesmente R. Este software, dentre outros formatos, lê arquivos do tipo .csv (*comma separetad value* ou valor separado por virgula) e, a partir daí, realiza análises estatísticas e gera artefatos como curvas de correlação, diagramas de dispersão, matrizes de correlação e o próprio índice de correlação, que são informações úteis na interpretação dos dados minerados.

4.2 Resultados

4.2.1 Comparação entre tamanho de código e métricas de complexidade

No que diz respeito a métricas de tamanho e de complexidade, foram encontradas relações muito fortes, especialmente entre LOC e WMC, que apresentaram índices de correlação em torno de 0,97. Métricas de Halsteas e métricas do princípio de dependência acíclica apresentaram apenas correlações em torno de 0,3. Assim, estas duas últimas foram descartadas da análise.

Este resultado não é nada inovador, uma vez que a segunda lei de evolução de Lehman afirma que se um software tem o tamanho aumentado, sua complexidade será aumentada também, ao menos que alguma estratégia de controle de complexidade seja seguida. Na Figura 4.1 podem ser vistas as curvas de evolução do servidor de aplicações Tomcat.

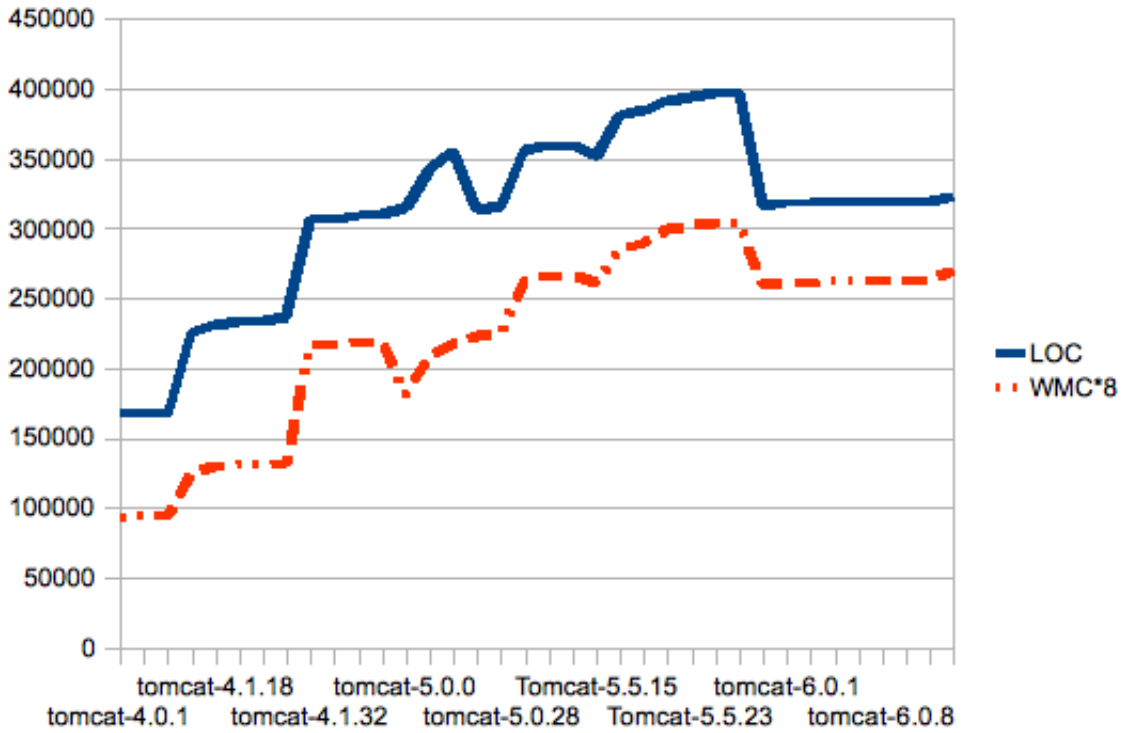

Figura 4.1: Similaridade entre as curvas de evolução de LOC e MWC do Tomcat.

A Figura 4.2 mostra o diagrama de dispersão, gerado pelo R, de LOC e WMC do Spring Framework. Podemos perceber que, neste caso, os pontos da dispersão estão muito próximos

da linha que indica máxima correlação direta. A correlação neste caso é de 0,998.

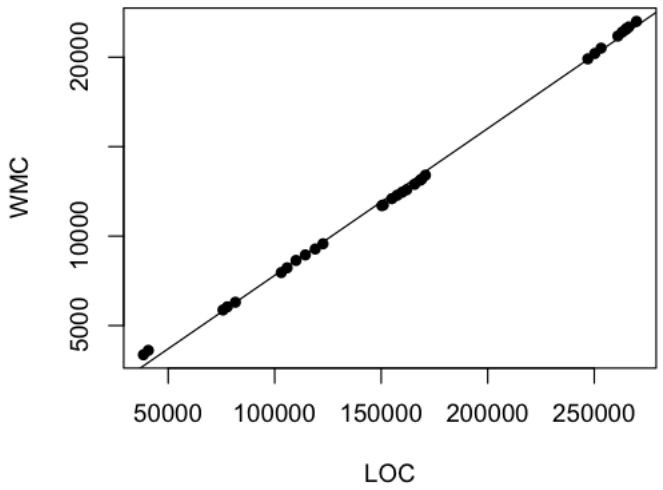

Figura 4.2: Diagrama de dispersão de LOC e WMC do Spring.

4.2.2 Considerando *Bugs* na análise

A análise dos dados dos *bug reports* revelaram uma correlação muito fraca quando analisadas com *bad smells* e com WMC. O índice de correlação encontrado foi em torno de 0,2. Estes valores estão em uma zona considerada ruim para fazer qualquer afirmação sobre a correlação, pois ela não é próxima dos limites inferiores ou superiores e fica a uma distância do zero, que não permite afirmar que não existe correlação. De forma resumida, o valor não e um bom indicador estatístico. O que esperavamos, de fato, era que, como a complexidade do software havia aumentado, o total de *bugs* reportados deveria ter aumentado, uma vez que a complexidade do software implica em decréscimo da qualidade.

Apesar do resultado não ter sido conforme o esperado, algumas considerações devem ser feitas:

- Ao considerar esta abordagem, intuitivamente se percebe que o aumento na complexidade iria refletir em mais bugs em um tempo desconhecido. Talvez o tempo que analisamos, apesar de considerável, não seja suficiente para para mostrar a influência esperada.

- Existe uma grande subjetividade na reportação de *bugs*, uma vez que a qualidade de reportação desses *bugs*, em geral depende absolutamente do usuário.

- *Bugs* corrigidos podem reaparecer, pelo fato da complexidade ter aumentado.

A Figura 4.3 mostra o diagrama de dispersão de *bugs* por LOC e WMC por LOC do Hibernate. Nós podemos observar que os pontos estão muito dispersos e distantes da reta de máxima correlação. O índice de correlação mostrado é de 0,0057.

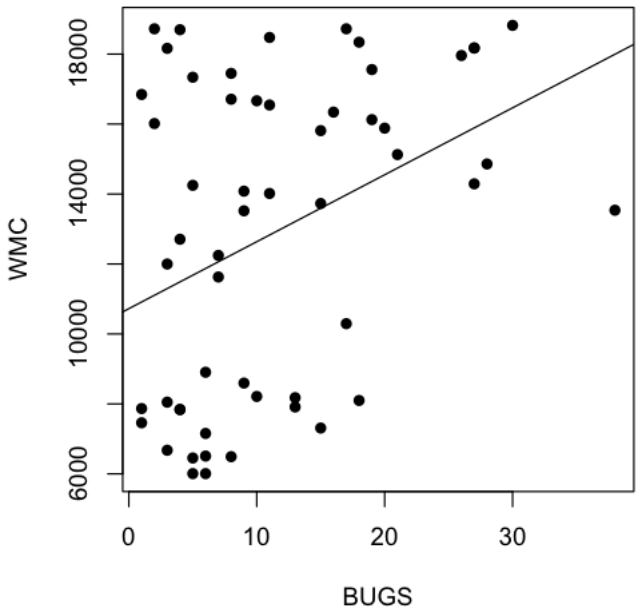

Figura 4.3: Diagrama de dispersão dos *bugs* reportados e WMC do Hibernate.

A segunda fase da investigação com *bugs* partiu dos dados fornecidos pelo *FindBugs*. Como citado anteriormente, o FindBugs categoriza *bugs* e *bad smells*. Como esta etapa do trabalho consistia em analisar *bugs*, contamos apenas o que ele categorizou como *bug*.

Os resultados revelaram uma correlação dos *bugs* coletados com o FindBugs e as demais variáveis em torno de 0,87, o que indica uma forte correlação entre elas. Nós também usamos os Bad Smells coletados pelo FindBugs para confrontar os coletados com o PMD. Como esperado, a correlação entre eles foi de 0.98.

4.2.3 Considerando *Bad Smells na Análise*

Um resultado interessante encontrado foi das relações entre LOC, WMC e o número de *Bad Smells*. A Figura 4.4 mostra a similaridade entre as curvas que representam essas variáveis

para a evolução do JDK. Neste caso, o índice de correlação entre *bad smells* e LOC foi de 0.97 e entre *bad smells* e WMC de 0.98.

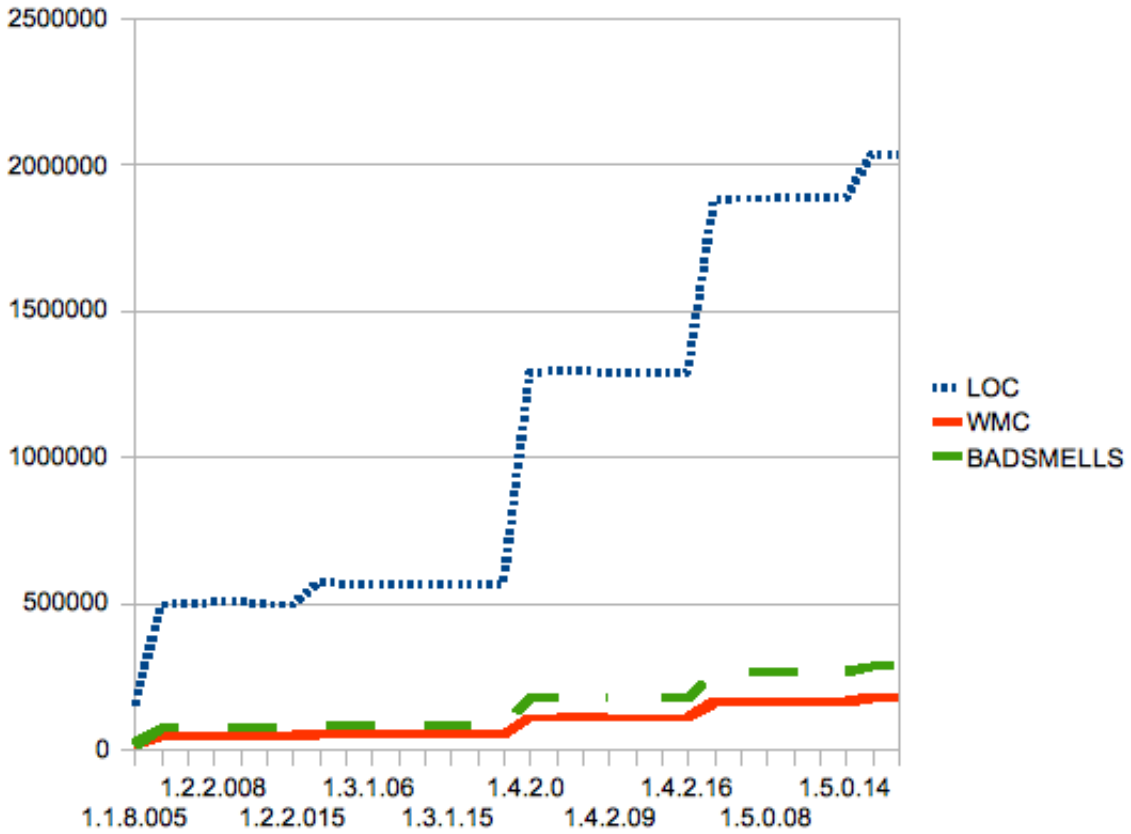

Figura 4.4: Curvas de evolução de LOC, WMC e *bad smells* do JDK.

A Figura 4.5 mostra o diagrama de dispersão para o WMC e os *bad smells* do Azureus, gerado pelo R. Neste caso, podemos observar também o quão próximo estão os pontos da reta de correlação direta máxima.

Um resultado interessante surgiu quando extraímos os dados do repositório do projeto *FindBugs*. Os desenvolvedores do *FindBugs*, se preocuparam de fato com os *bugs* e *bad smells* durante o desenvolvimento e procuraram eliminar os *bad smells* a cada nova versão lançada.

A Figura 4.6 mostra um gráfico de evolução do software FindBugs. Podemos perceber a curva de *bad smells* decrescendo ao longo de toda evolução do projeto e, a de WMC, crescendo, aparentemente sem nenhuma influência do decréscimo dos *bad smells*.

Conforme citado anteriormente, eliminamos manualmente os *bad smells* do Hibernate e Spring indicados pela ferramenta e o mesmo comportamento detectado nas curvas de evolu-

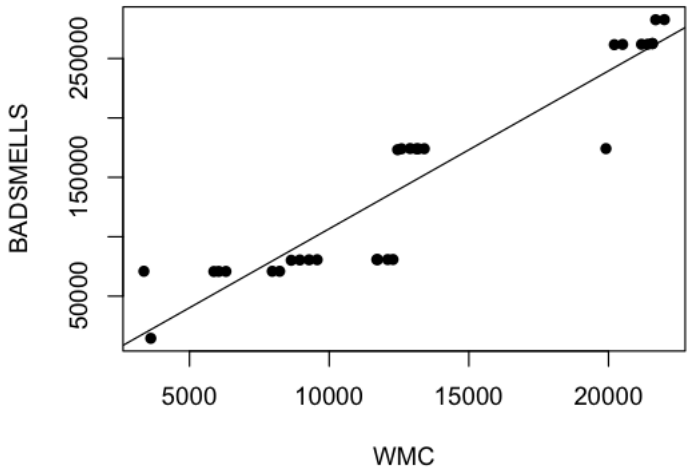

Figura 4.5: Diagrama de dispersão de WMC e *bad smells* do Azureus.

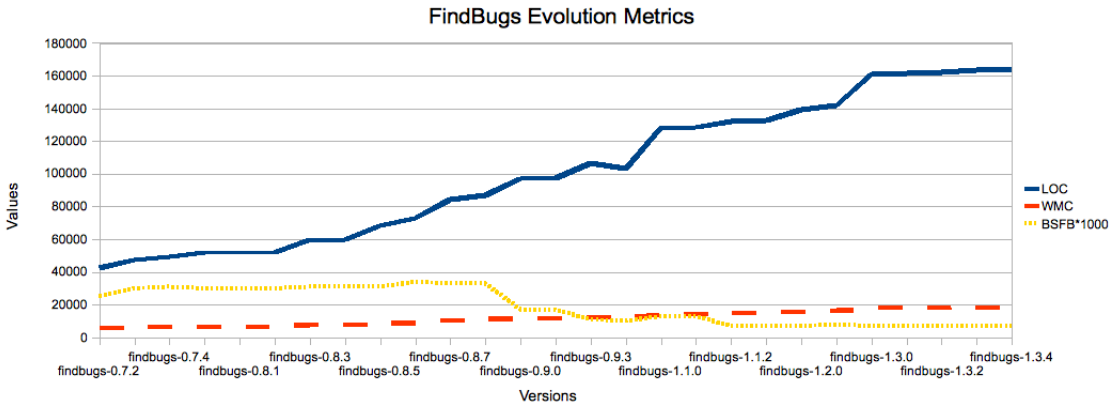

Figura 4.6: Curvas de Evolução do FindBugs.

ção do projeto FindBugs, foi detectado. As Figuras 4.7 e 4.8 mostram curvas de Evolução do Hibernate e Spring respectivamente após eliminação manual dos *bad smells*.

Esses experimentos mostraram que, apesar da correlação quase perfeita entre *bad smells* e WMC, não existe aparente relação de causa-efeito entre essas variáveis. Não tivemos acesso a fazer qualquer modificação no código fonte dos sistemas legados. Isso inviabilizou a generalização desse resultado.

Isto dá indícios que eliminar *bad smells* em sistemas *open source*, não é uma boa estratégia a ser seguida se o objetivo for reduzir a complexidade do *design*.

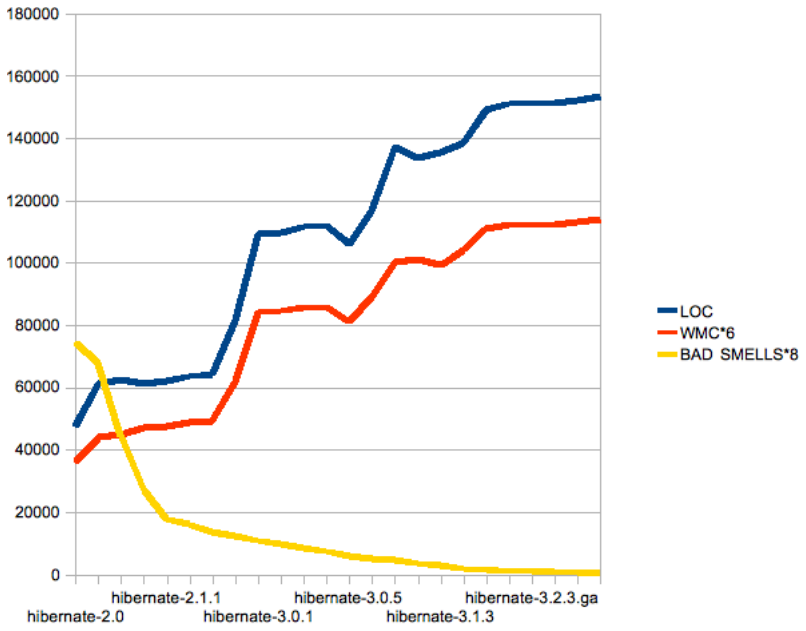

Figura 4.7: Curvas de Evolução do Hibernate após eliminação manual dos *bad smells*.

Teste Estatístico para Verificação das Correlações

As correlações entre as variáveis citadas anteriormente, como já citado foram calculadas utilizando o software R. Contudo é necessário informar como parâmetro da função que calcula a correlação, o método que será utilizado nesse cálculo. A escolha do método se baseia na normalização ou não dos dados em questão.

Realizou-se então teste de normalidade das variáveis e verificou-se a não normalização do conjunto de dados. Essa verificação foi feita utilizando a função *qqplot* do R. Assim, uma vez verificado que os dados não estavam normalizados, utilizou-se o método de Kendall, o qual não requer que as amostras a serem analisadas estejam normalizadas para cálculo dos índices de correlação e conseguinte estimativa da associação entre as variáveis.

As Figuras 4.9, 4.10, 4.11, 4.12 e 4.13 mostram respectivamente as curvas de verificação de normalidade dos dados de WMC e *bad smells* geradas pelo R para para o Tomcat, FindBugs, Hibernate e de dois dos cinco sistemas proprietário analisados. Já as Figuras 4.14, 4.15, 4.16, 4.17 e 4.18 mostram respectivamente as curvas de verificação de normalidade dos dados de LOC e WMC também geradas pelo R para o JDK, ZK, Spring e dois dos sistemas proprietários analisados. Se os dados estivessem normalizados, os pontos estariam rentes às retas de normalização, ao contrário do que se pode observar nas figuras.

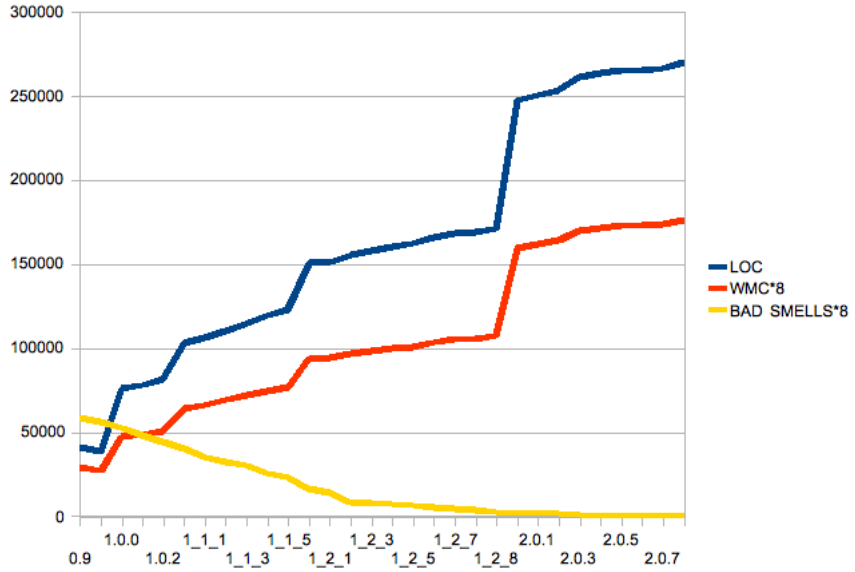

Figura 4.8: Curvas de Evolução do Spring após eliminação manual dos *bad smells*.

Uma vez calculados os índices de correlação, é necessário realizar testes de hipótese para verificar a veracidade das correlações. Para isso esse processo foi organizado da seguinte maneira:

1. **Determinação das hipóteses:** $H_0 = 0$ (Aceitação da hipótese nula) ou $H_1 \neq 1$ (Rejeição da hipótese nula), onde a hipótese nula em questão é a não veracidade das correlações.

2. **Fixação do nível de significância** α**:** Normalmente se utiliza $\alpha = 5\%$ ou $\alpha = 10\%$ [Was04]. Este α indica o quanto o teste de hipótese realizado pode estar errado. Para diminuir a margem de erro, neste trabalho, o α escolhido foi de 5%.

Para todos os índices de correlação encontrados a hipótese nula foi rejeitada, ou seja as correlações são verdadeiras.

A Tabela 4.3 mostra os índices de correlação, p-valores e os r-quadrados encontrados para os softwares analisados. Vale a pena citar que o r-quadrado mede a proporção da variabilidade em *Y* que é explicada por *X*, mediante cálculo prévio do índice de correlação entre estas duas variáveis. Por exemplo, para um índice de correlação r = 0,8547 entre duas variáveis *x* e *y*, tem-se um r-quadrado = 0.7305. Ou seja, 73,05% das variações em *Y* são explicadas por *X*

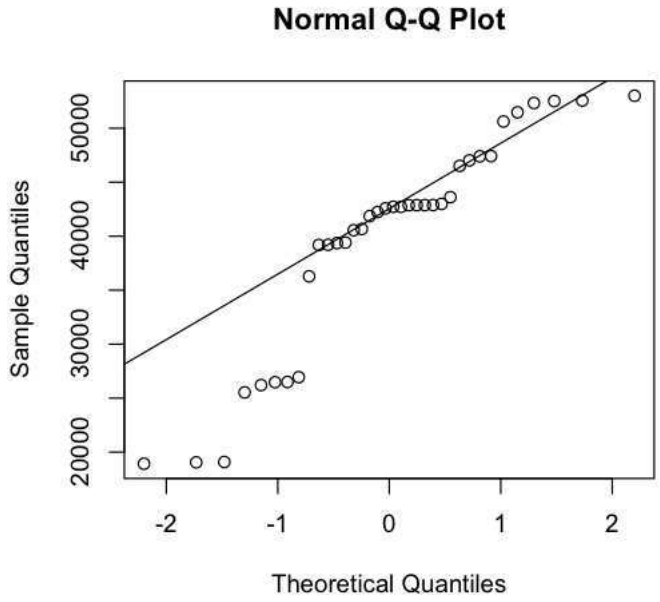

Figura 4.9: Gráfico indicador da não normalização dos dados de WMC e Bad Smells extraídos do Tomcat.

A Tabela 4.4 mostra os índices de correlação, p-valores e os r-quadrados encontrados após eliminar os *bad smells* do Hibernate e do Spring.

4.3 Discussão dos resultados

Nosso objetivo é tratar sobre aspectos chaves de qualidade de *design* como *bad smells* e suas relações com complexidade. Quando encontramos as correlações entre a evolução das curvas que representam estas variáveis ao longo dos projetos, fizemos a seguintes suposição:

- A respeito da forte correlação, não podemos inferir que a influência direta de WMC em *Bad smells* indica uma relação de causa-efeito (i.e. ao modificar uma variável, a outra será modificada no mesmo sentido). Nós podemos considerar isto porque, semanticamente, controlar aspectos que reduzem a complexidade de software (e.g. *loops*, aninhamento de *if's*) não irá resolver o problema com os *bad smells*. Isto iria apenas contribuir na melhoria de qualidade do *design* no que tange a redução de aspectos que influenciam a complexidade em sí. Entretanto, nosso interesse era manipular uma variável de maneira que a qualidade do software fosse melhorada não apenas no aspecto que tange àquela variável, mas que, ao manipular uma, a outra fosse contro-

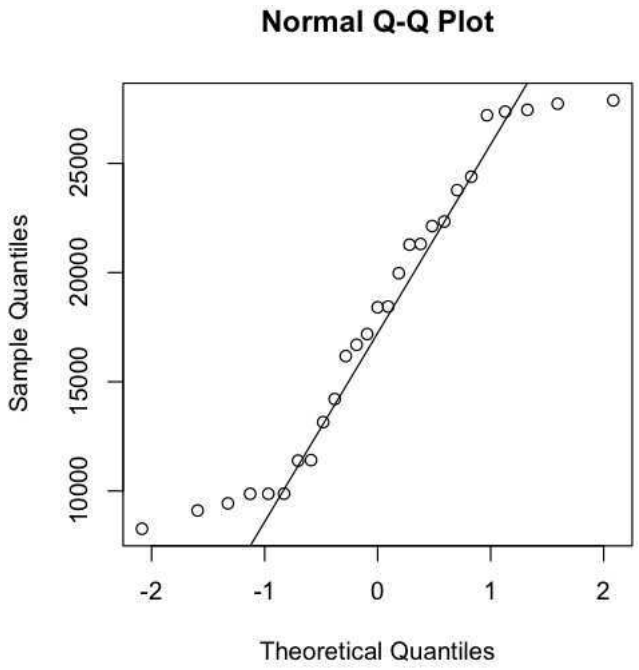

Figura 4.10: Gráfico indicador da não normalização dos dados de WMC e Bad Smells extraídos do FindBugs.

> lada também. Dessa maneira, a influência direta de *Bad smells* em WMC se mostra mais factível com o que nós estamos interessados. É razoável supor que a remoção de *bad smells* irá causar uma redução significativa na complexidade. Tal melhoria no código, de acordo com a literatura, implica em um *design* mais fácil de testar e refatorar [CK94]. Isto reflete uma clara melhoria do *design*.

No entanto, os resultados mostraram que, mesmo eliminando *bad smells*, a complexidade do software não sofre alterações consideráveis.

Os resultados permitem conclusões ainda mais interessantes, uma vez que eles confirmam o quão difícil é tratar qualidade de software, principalmente porque os fatores que a influenciam, não sofrem nem exercem influência direta uma na outra.

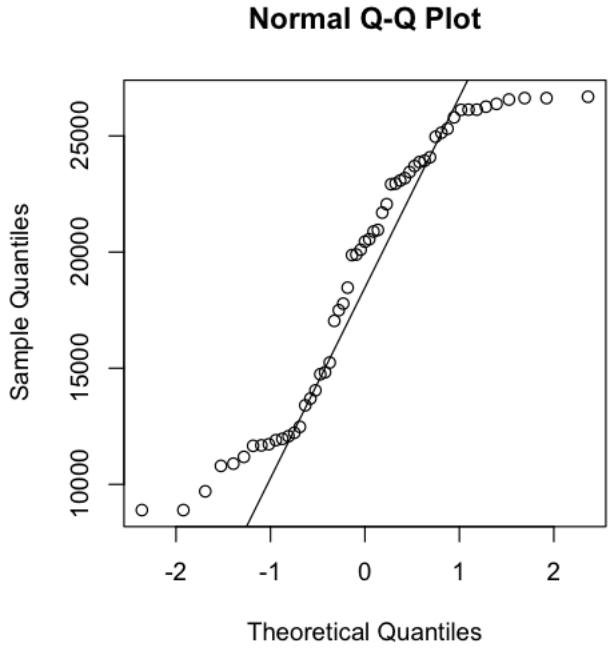

Figura 4.11: Gráfico indicador da não normalização dos dados de WMC e Bad Smells extraídos do Hibernate.

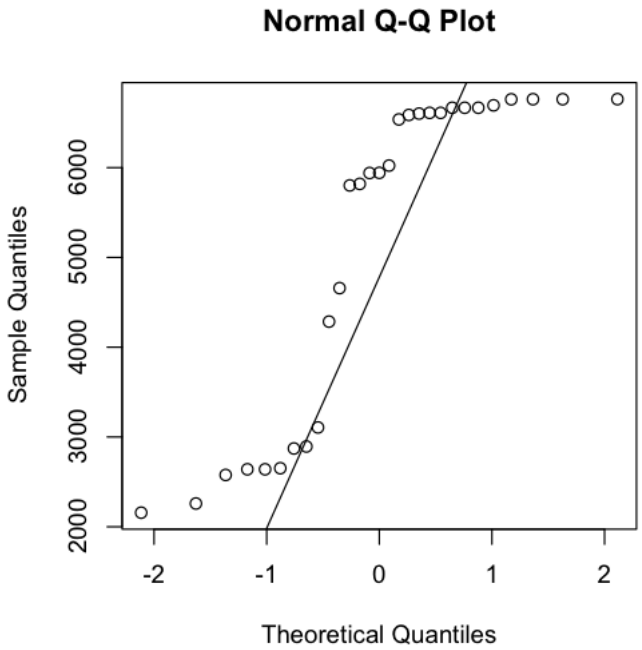

Figura 4.12: Gráfico indicador da não normalização dos dados de WMC e Bad Smells extraídos de um dos cinco softwares proprietários analisados.

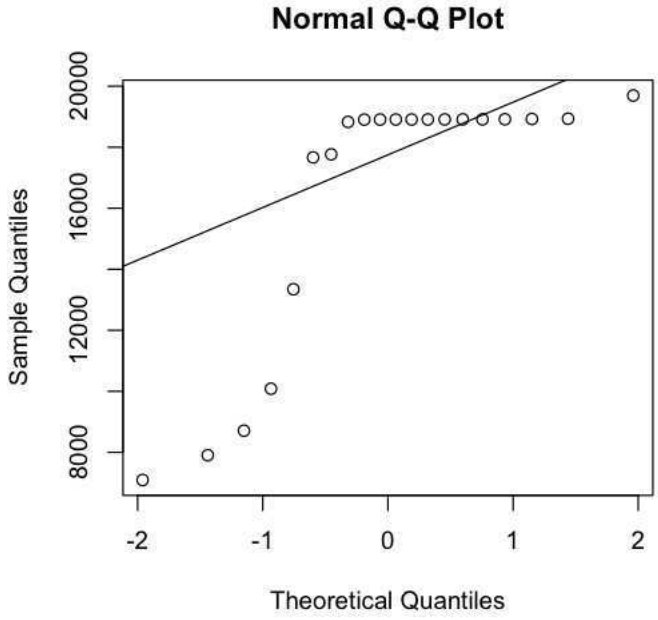

Figura 4.13: Gráfico indicador da não normalização dos dados de WMC e Bad Smells extraídos de outro dos cinco softwares proprietário analisado.

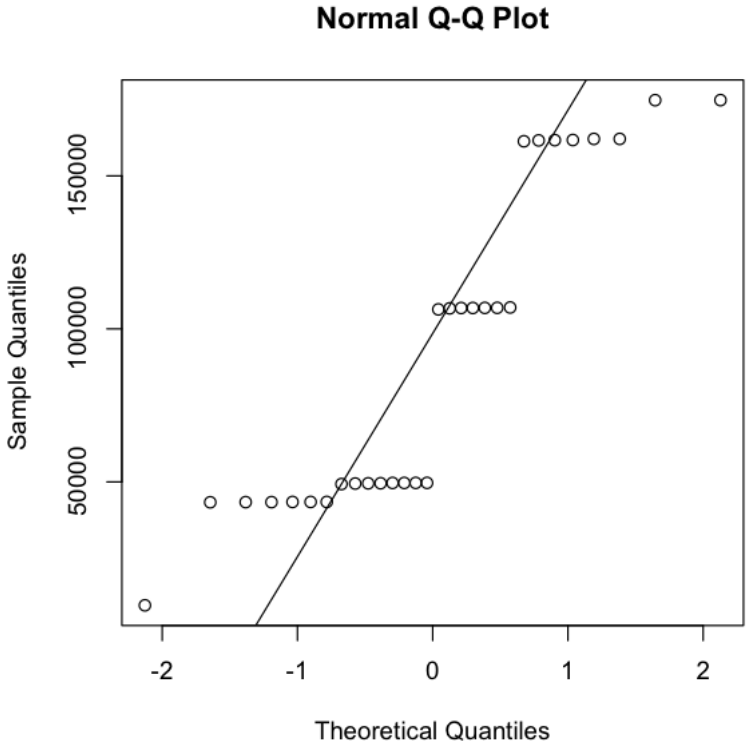

Figura 4.14: Gráfico indicador da não normalização dos dados de LOC e WMC extraídos do JDK.

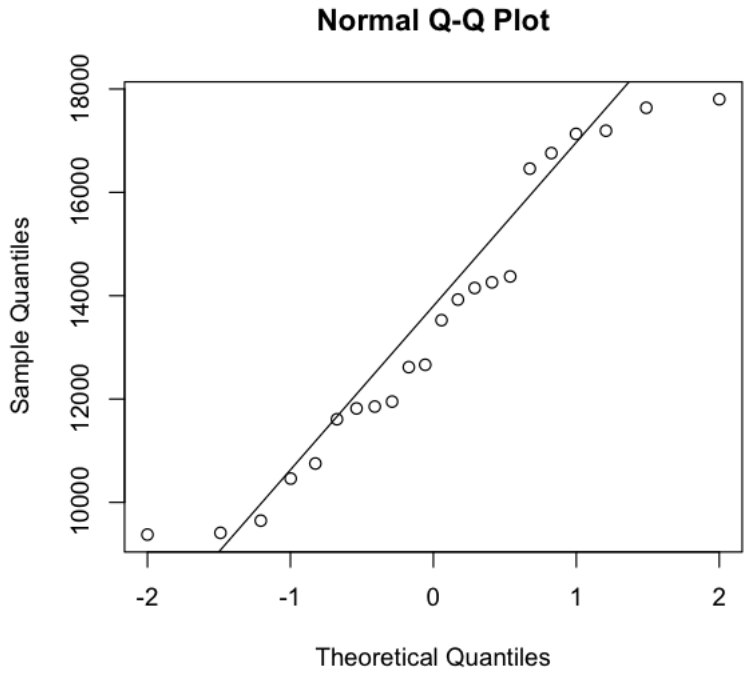

Figura 4.15: Gráfico indicador da não normalização dos dados de LOC e WMC extraídos do ZK.

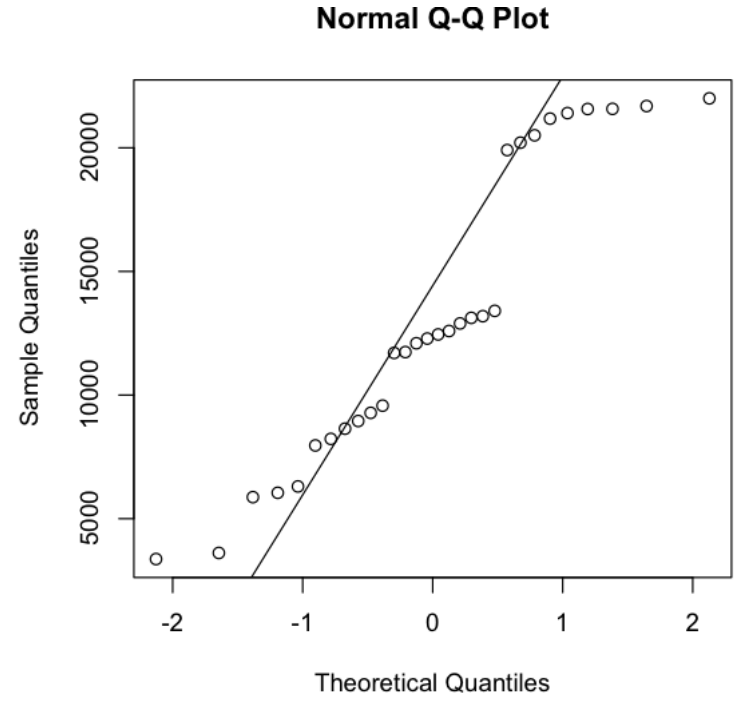

Figura 4.16: Gráfico indicador da não normalização dos dados de LOC e WMC extraídos do Spring.

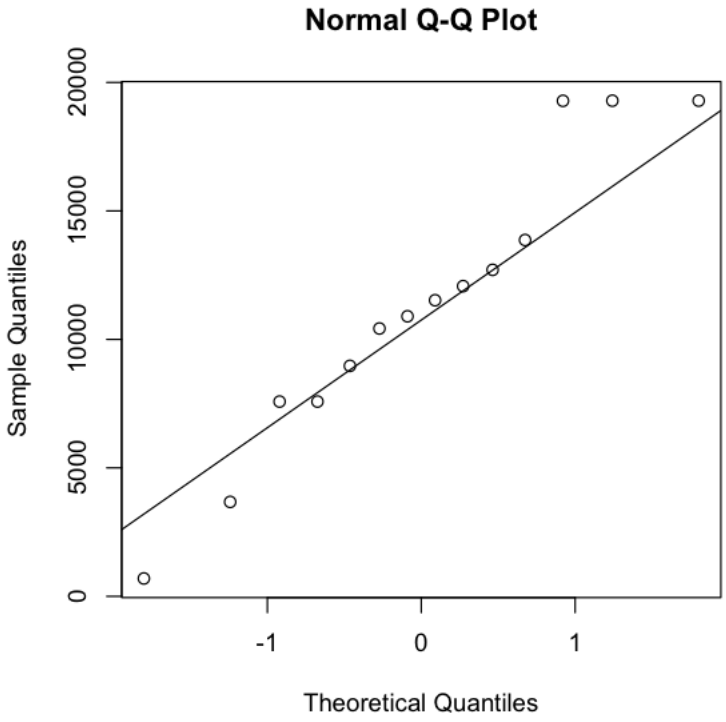

Figura 4.17: Gráfico indicador da não normalização dos dados de LOC e WMC extraídos de um dos cinco softwares proprietário analisado.

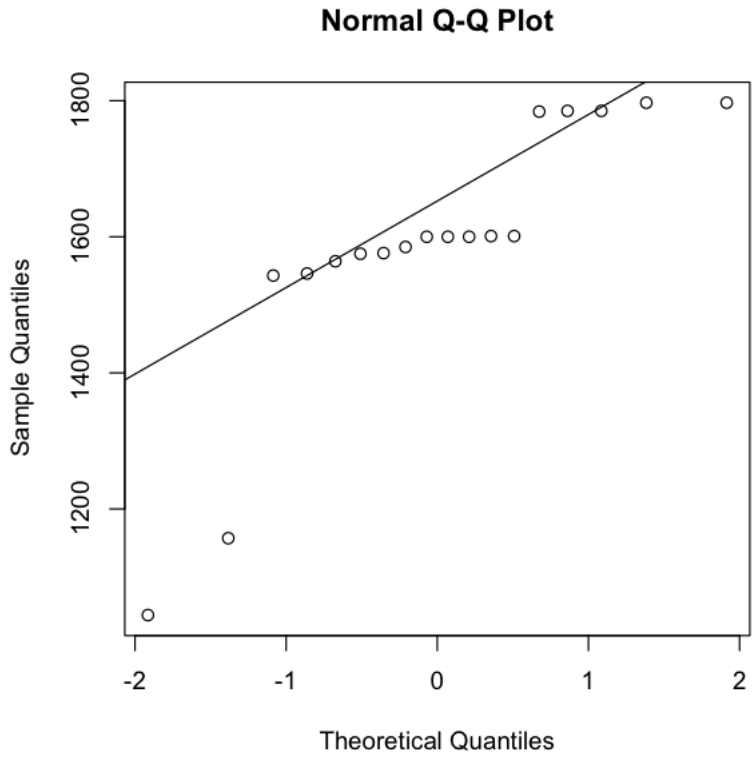

Figura 4.18: Gráfico indicador da não normalização dos dados de LOC e WMC extraídos de outro dos cinco softwares proprietário analisado.

Tabela 4.3: Dados estatíticos dos softwares analisados.

Software	**Índice de correlação WMC x *Bad Smells***	**p-valor**	**r-quadrado**
Azureus	0.9354839	2.2e-16	87.51%
FindBugs	-0.5839455	4.518e-05	-34.09%
Hibernate	0.9418182	1.290e-10	88.70%
JDK	0.9862069	1.110e-15	97.26%
Saxon	0.9555556	3.817e-13	91.30%
Spring	0.9989898	2.2e-16	99.79%
Tomcat	0.9015873	2.2e-16	81.28%
ZK	0.9988707	2.459e-06	99.77%
Software proprietário 1	0.9863946	2.946e-08	97.29%
Software proprietário 2	0.9598166	5.512e-09	92.12%
Software proprietário 3	0.915637	2.127e-13	83.83%
Software proprietário 4	0.994429	9.817e-07	98.88%
Software proprietário 5	0.9937586	7.372e-14	98.75%

Tabela 4.4: Dados estatíticos dos softwares manipulados para eliminação dos *bad smells*.

Software	**Índice de correlação WMC x *Bad Smells***	**p-valor**	**r-quadrado**
Hibernate	-0.618419	2.2e-16	-38.24%
Spring	-0.517984	4.387e-05	-26.83%

Capítulo 5

Conclusão e Trabalhos Futuros

Complexidade e qualidade são características de software que sempre remetem uma a outra. Dessa maneira, controlar a complexidade de software é uma tarefa que vem sendo estudada há muitos anos como forma de melhorar a qualidade do produto de software.

Apesar da crença de que existe uma relação de causa-efeito entre métricas de complexidade e qualidade em termos de *bad smells*, nós demonstramos através de experimentos realizados em oito softwares *opens source* e cinco proprietários que, apesar da forte correlação estatística entre a contagem de *bad smells* e da complexidade em termos de WMC, não existe relação de causa-efeito entre eles.

Eliminar *bad smells* simplesmente, não contribui na redução da complexidade do software. Nosso trabalho serve como contribuição no processo de determinação de qualidade de software, uma vez que correlacionamos dois aspectos que influenciam a qualidade. No entanto, ainda falta muito para termos posicionamentos mais contudentes no que diz respeito ao fator mais relevante que é qualidade de software.

A respeito dos experimento com os sistemas proprietários, os experimento com *bugs* sofreram limitações na realização, uma vez que, por questões de sigilo da empresa que contruiu os sistemas, os dados dos *bug reports* não puderam ser disponibilizados. Dessa maneira, entendemos que realizar mais experimentos com sistemas proprietários que possam ser melhor analisados, é uma atividade a ser desenvolvida em trabalhos futuros. Isso permitirá uma maior generalização dos resultados.

No que diz respeito aos softwares *open source*, é interessante aumentar o número da amostra. Analisar mais softwares também é uma tarefa a ser realizada em trabalhos futuros,

uma vez que isso fundamentaria ainda mais nossos resultados.

Outra possibilidade para trabalhos futuros é minerar outras classes de dados que reflitam a qualidade do software, como, por exemplo, a variação de número de usuário de um sistema: Se ao evoluir um software perde usuários, pode ser que, no processo evolutivo, alguma funcionalidade fundamental teve algum aspecto degradado o que implicou em queda da qualidade do software.

Levantar dados de outras fontes de *bugs* como listas de discussão e fóruns e, em seguida, estudar uma maneira de quantificar estes dados e a partir daí combinar estes dados com outros parâmetros de qualidade a fim de contribuir com o que existe em termos de qualidade, também pode ser realizado futuramente.

As correlações que encontramos com *bad smells* e WMC sempre tendiam a +1 ou -1, sem nunca permear a faixa central. Investigar a razão de isto acontecer é uma outra possibilidade de trabalhos futuros.

Uma outra opção para trabalhos futuros é analisar aspectos de qualidade e complexidade ao logo da evolução de software com métricas de processo. Seria interessante partir das equações 2.7 e 2.6 de Handerson e Sellers [HS96] e realizar análises de regressão e verificar o grau de influência da complexidade em determinado ponto de um processo de desenvolvimento. Isso possibiitaria verificar quais variáveis deveriam ser tratadas prioritariamente a fim de melhorar a qualidade do software como um todo.

Capítulo 6

Trabalhos Relacionados

A relação entre qualidade e complexidade de *design* de software vem a tempos sendo exaustivamente investigada. Lehman estudou os sistemas da IBM durante trinta anos e postulou as chamadas leis da evolução de software. Uma dessas leis afirma explicitamente que a complexidade do software irá crescer ao longo da vida dele, ao menos que alguma estratégia de controle de complexidade seja seguida de modo a mantê-la o mais constante possível ou reduzi-la [LR06]. Este trabalho mostrou que a remoção de *bad smells*, não é uma tarefa indicada a ser realizada, se o objetivo for controlar a complexidade do software.

Capiluppi [CFRH+07] estudou a evolução de apenas um software proprietário, desenvolvido com métodos ágeis e identificou que a quantidade relativa de trabalhos de controle da complexidade, como remoção de *bad smells* é inversamente proporcional a complexidade ciclomática. Contrariando este resultado, nosso trabalho, mediante a análise de um número muito maior de sistemas analisados, mostra que que eliminar *bad smells* não implica em redução da complexidade do *design*.

Shatnawi, li e Zang [RS06] cletaram métricas a partir do respositório do projeto Eclipse e a contagem de *bugs* do *bug report* a fim de identificar relações entre estas duas classes de dados e partir daí verificar se as métricas coletadas poderiam ser usadas na predição de probabilidade erros em classes de software orientado a objeto. A contagem de erros por classe era feita pela presença da classe em questão em cada *bug* reportado. Este estudo empíric mostrou a existência de correlação entre *bugs* catalogados em *bugs reports* e WMC. Ao contrário deste resultado, nós identificamos praticamente a inexistência de correlação estatística entre estas duas variáveis, com uma amostra bem maior.

No que diz respeito a trabalhos relacionados a ferramentas que auxiliassem no processo de mineração de repositórios. O Kenyon [BEJWKG05] e o FishEye [Ins] São as que mais se assemelham ao CodeMiner.

O FishEye se mostrou ser uma ferramente bastante versátil e robusta. Ela oferece a possibilidade de minerar algumas classes de dados em repositórios de software, e visualizar estas informações graficamente, assim como a possibilidade de visualização em forma de relatórios. O FishEye permite coletar os dados dos repositórios basicamente de duas maneiras: A primeira e mais simples é levantando informações gerais dos projetos e a segunda e mais complexa é especificando através de um sistema de *queries* customizadas, que informações devem ser retornadas ao usuário. O fishEye é uma aplicação *web* muito bem estruturada e robusta. No entanto, nosso interesse era ter uma estrutura que nos fornecesse o máximo de versatilidade, podendo ser utilizada como uma API; ou seja, utilizar os recursos fornecidos por uma linguagem de programação, no nosso caso Java, para lidar com os dados de entrada e saída da ferramenta.

O Kenyon por sua vez se tratava de uma ferramenta que se mostrava útil para desenvolver as nossas atividades. Trata-se de uma API que, segundo a descrição feita no artigo em que foi publicada, seria mais do que útil para nossa pesquisa. O grande problema foi a impossibilidade de utilizar o Kenyon, pelo fato de esta não estar disponível onde está indicado no artigo, nem mesmo nas páginas *web* dos autores ou do laboratório em que foi desenvolvido.

Bibliografia

[BD02] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17, 2002.

[BEJWKG05] Jennifer Bevan, Jr. E. James Whitehead, Sunghun Kim, and Michael Godfrey. Facilitating software evolution research with kenyon. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 177–186, New York, NY, USA, 2005. ACM.

[BMHWMM98] William J. Brown, Raphael C. Malveau, III Hays W. McCormick, and Thomas J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[BZ07] Victor R. Basili and Marvin V. Zelkowitz. Empirical studies to build a science of computer science. *Commun. ACM*, 50(11):33–37, 2007.

[CFR05] Andrea Capiluppi, Alvaro E. Faria, and Juan F. Ramil. Exploring the relationship between cumulative change and complexity in an open source system. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 21–29, Washington, DC, USA, 2005. IEEE Computer Society.

[CFRH+07] A. Capiluppi, J. Fernandez-Ramil, J. Higman, H. C. Sharp, and N. Smith. An empirical study of the evolution of an agile-developed software system. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 511–518, Washington, DC, USA, 2007. IEEE Computer Society.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[Cur99] B. Curtis. In search of software complexity. *IEEE Trans. Softw. Eng.*, pages 95–106, 1999.

[CY91] Peter Coad and Edward Yourdon. *Object Oriented Design*. Prentice-Hall, 1991.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[eAM96] Fernando Brito e. Abreu and Walcelio Melo. Evaluating the impact of object-oriented design on software quality. In *METRICS '96: Proceedings of the 3rd International Symposium on Software Metrics*, page 90, Washington, DC, USA, 1996. IEEE Computer Society.

[EGK+01] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, 2001.

[Eva04] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[FP08] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous Approach*. Revisited Printing, 2008.

[FPB87] Jr. Frederick P. Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[Gla01] Robert L. Glass. Frequently forgotten fundamental facts about software engineering. *IEEE Software*, 18(3):110–111, 2001.

[Gol06] Eliyahu M. Goldratt. *The Haystack Syndrome: Sifting Information Out of the Data Ocean*. North River Press Publishing Corporation, 2006.

[GVG04] Jean-Francois Girard, Martin Verlage, and Dharmalingam Ganesan. Monitoring the evolution of an oo system with metrics: An experience from the stock market software domain. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 360–367, Washington, DC, USA, 2004. IEEE Computer Society.

[Hal77] M. H. Halstead. *Elements of Software Science*. Elsevier/North-Holland, 1977.

[HS91] Brian Henderson-Sellers. *Software Engineering. The Production of Quality Software*. macmillan, 1991.

[HS92] Brian Henderson-Sellers. An empirical study of software metrics. *IEEE Trans. Softw. Eng.*, 13(6):17–19, 1992.

[HS96] Brian Henderson-Sellers. *Objetc-Oriented Metrics - Measures of Complexity*. Prentice Hall, 1996.

[Ins] FishEye: Source Code Repository Insight. Atlassian. *http://www.atlassian.com/software/fisheye/*.

[ISO91] International standard iso/iec 9621. information technology: Software product evaluation: Quality characteristics and guidelines for their use. Technical report, International Standard Organisation (ISO), 1991.

[Jav] JavaCVS. Netbeans. *http://javacvs.netbeans.org/*.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, pages 22–35, 1988.

[Kaf85] Dennis Kafura. A survey of software metrics. In *ACM '85: Proceedings of the 1985 ACM annual conference on The range of computing : mid-80's perspective*, pages 502–506, New York, NY, USA, 1985. ACM.

[KKNM07] David G. Kleinbaum, Lawrence L. Kupper, Azhar Nizam, and Keith E. Muller. *Applied Regression Analysis and Multivariable Methods*. Duxbury Press, 4th edition, 2007.

[Lak96] John Lakos. *Large Scale C++ Software Design*. Addisson Wesley, 1996.

[LB85] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, 1985.

[LC87] H. F. Li and W. K. Cheung. An empirical study of software metrics. *IEEE Trans. Softw. Eng.*, 13(6):697–708, 1987.

[Leh69] M. M. Lehman. The programming process. Technical report, IBM Research Division, 1969.

[Leh96] M. M. Lehman. Laws of software evolution revisited. In *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, London, UK, 1996. Springer-Verlag.

[Lin04] Ruediger Lincke. Weighted method count. *http://www.arisa.se/compendium/node97.html*, 2004.

[Lis87] Barbara Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.

[LR89] K. J. Lieberherr and A. J. Riel. Contributions to teaching object-oriented design and programming. In *OOPSLA '89: Conference proceedings on*

Object-oriented programming systems, languages and applications, pages 11–22, New York, NY, USA, 1989. ACM.

[LR01] M. M. Lehman and J. F. Ramil. An approach to a theory of software evolution. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 70–74, New York, NY, USA, 2001. ACM.

[LR06] M. M. Lehman and J. F. Ramil. *Software Evolution, Chapter 1 in Software Evolution and Feedback: Theory and Practice*. John Wiley and Sons, 2006.

[Man04] Mika V. Mantyla. Developing new approaches for software design quality improvement based on subjective evaluations. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 48–50, Washington, DC, USA, 2004. IEEE Computer Society.

[Mar02] Robert Cecil Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[McC76] Thomas J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[MD08] Tom Mens and Serge Demeyer. *Software Evolution*. Springer, 2008.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[M.S92] M.Shepperd. Products, processes and metrics. In *Inf. Software Technol.*, pages 674–680, 1992.

[NR69] P. Naur and B. Randell. Software engineering. Technical report, NATO Science Committee, 1969.

[oF] State of Flow. Eclipse-metrics. *http://eclipse-metrics.sourceforge.net/*.

[Par94] David Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[PMD] PMD. sourceforge. *http://pmd.sourceforge.net/index.html*.

[Rie96] Arthur Riel. *Object Oriented Design Heuristcs*. Addisson Wesley, 1996.

[RMW01] A. R. Rocha, C. M. Maldonado, and J. C. Weber. *Qualidade de Software*. Prentice Hall, 2001.

[RS06] Wei Li e Huaming Zhang Raed Shatnawi. Predicting error probability in the eclipse project. In *Innternational Conference on Software Engineering Research*, 2006.

[Sca03] Walt Scacchi. Understanding open source software evolution. In *Applying, Breaking, and Rethinking the Laws of Software Evolution*. John Wiley and Sons Inc, 2003.

[Sli05] Stefan Slinger. *Code Smell Detection in Eclipse*. PhD thesis, Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Department of Software Technology, Software Engineering Group, Software Evolution Research Lab, 2005.

[Was04] Larry Wasserman. *All Statistcs - A Concise Course in Statistical Inference*. Springer, 2004.

[Zus94] H. Zuse. Foundation of the validation of object-oriented software measures. *Theorie und Praxin der Softwaremussung*, pages 136–214, 1994.