Pós-graduação em Ciência da Computação

"Métricas como Ferramenta de Auxílio para o Gerenciamento de Dívida Técnica em Produtos de Software"

Por

Antonio Luiz de Oliveira Cavalcanti Júnior

Dissertação de Mestrado Profissional

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, OUTUBRO/2012

Universidade Federal de Pernambuco

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Antonio Luiz de Oliveira Cavalcanti Júnior

Métricas como Ferramenta de Auxílio para o Gerenciamento de Dívida Técnica em Produtos de Software

Este trabalho foi submetido à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre Profissional em Ciência da Computação.

ORIENTADOR: ***Prof. Fabio Queda Bueno Da Silva***

RECIFE, OUTUBRO/2012

Catalogação na fonte
Bibliotecária Jane Souto Maior, CRB4-571

Cavalcanti Júnior, Antonio Luiz de Oliveira
Métricas como ferramenta de auxílio para o gerenciamento de dívida técnica em produtos de software/ Antonio Luiz de Oliveira Cavalcanti Júnior. - Recife: O Autor, 2012.
66 f., fig., tab., quadro

Orientador: Fábio Queda Bueno da Silva.
Dissertação (mestrado profissional) - Universidade Federal de Pernambuco. CIn, Ciência da Computação, 2012.

Inclui referências e apêndice.

1. Engenharia de Software. 2. Engenharia de Software Baseada em Evidências. I. Silva, Fábio Queda Bueno da (orientador). I. Título.

005.1 CDD (23. ed.) MEI2014 – 112

Dissertação de Mestrado Profissional apresentada por **Antonio Luiz de Oliveira Cavalcanti Júnior** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título**, "Métricas como Ferramenta de Auxílio para o Gerenciamento de Débito Técnico em Produtos de Software",** orientada pelo **Professor Fabio Queda Bueno da Silva** e aprovada pela Banca Examinadora formada pelos professores:

_____

Prof. André Luís de Medeiros Santos
Centro de Informática / UFPE

_____

Prof[a]. Carolyn Seaman
University of Maryland Baltimore Conty

_____

Prof. Fabio Queda Bueno da Silva
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 01 de novembro de 2012.

_____

Prof. RICARDO MASSA FERREIRA LIMA
Vice-Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Ao meu filho Gabriel.

Minha incomensurável fonte de inspiração.
Minha inexorável fortaleza.

AGRADECIMENTOS

Existem três pessoinhas especiais que transformam em alegria qualquer momento vivido. Obrigado filho por você irradiar minha vida com sua luz e por me mostrar o que é ser feliz em completude. Obrigado Matheus, meu sobrinho querido, por ser tão generoso comigo em dividir seu sorriso. Obrigado Marina, minha sobrinha querida, por rir quando te jogo para cima. Obrigado por manterem vivo dentro de mim meu “menino maluquinho”.

Aos meus pais, Antonio Luiz e Márcia, que fizeram de mim o que sou hoje. Nunca terei como agradecer completamente por todos seus ensinamentos. Nunca conseguiria fazer isso com palavras.

Aos meus irmãos, Charles, o mais velho, que sempre carregou a responsabilidade do exemplo a ser seguido e o fez de forma exemplar; Michele, a emoção, que sempre nos faz refletir o quanto de razão e emoção devem participar de nossas decisões e a Alexandre, o caçula, a quem sempre posso recorrer para conversar sobre video−games. Obrigado por tudo.

Aos meus amigos, não listarei nomes para não correr o risco de cometer uma injustiça. Cada conversa que tive, cada comentário e opinião forjaram esse trabalho.

Ao meu orientador Prof. Fábio Queda que sempre esteve disponível dedicando seu precioso tempo a colocar as coisas nos trilhos quando nada parecia fazer sentido. Sou realmente grato pela oportunidade de trabalhar com você Fábio.

Aos professores e funcionários do Centro de Informática que trabalham duro para tornar nosso dia a dia mais fácil.

Em fim, a todos que direta ou indiretamente me ajudaram a concretizar esse trabalho.

Eppur, si muove!
(Contudo, se move!)

— GALILEU GALILEI

Ao ser forçado pela igreja católica a negar a Teoria Heliocêntrica, 1616.

RESUMO

É amplamente aceito que o custo de manutenção de um software representa mais de 90% do custo total do produto durante todo seu ciclo de vida. O custo de desenvolvimento do produto tende a se tornar insignificante principalmente em softwares com longos ciclos de vida. Softwares perdem sua importância e tendem a cair em desuso caso não sejam evoluídos. Naturalmente, durante esse processo de evolução diversas forças externas e internas corroboram para o decaimento da qualidade do código fonte do software, como por exemplo, pressões do mercado, diminuição dos orçamentos para manutenção, mudanças no time de desenvolvedores, dentre outros. Apesar de existirem técnicas para acompanhar a qualidade do código fonte de um software utilizando-se métricas, nenhuma leva em consideração o contexto e decisões tomadas que levaram à diminuição da qualidade se tornando então técnicas reativas. Para explicar esse decaimento de qualidade, inserir informações contextuais sobre as decisões tomadas, permitir planejamento preventivo e acompanhar o aumento dos custos de manutenção em softwares com código fonte já comprometido, é utilizada a metáfora de Dívida Técnica.

A metáfora de Dívida Técnica é uma analogia ao conceito de empréstimo financeiro. Ao comprometer a qualidade do código fonte do sistema em detrimento a algum benefício imediato, contrai-se uma dívida onde o valor do empréstimo é o esforço economizado na execução da tarefa. Como toda dívida, são aplicados juros sobre o empréstimo contraído e se a dívida não for adequadamente controlado, no futuro a maior parte do esforço de manutenção e evolução será gasto amortizando os custos da dívida. Apesar de intuitiva, muito popular na comunidade de profissionais de software e útil para diminuir a distância entre o vocabulário técnico e o gerencial, a metáfora tem sua definição em evidências anedóticas, por isso se faz necessário estudos que possam descrever o fenômeno com rigor científico.

Nesse contexto, o objetivo desse estudo é caracterizar a metáfora de Dívida Técnica sob a ótica de métricas de software, descrevendo relações existentes entre as tomadas de decisões e os impactos causados na qualidade, utilizando-se de suítes clássicas de métricas, fundamentado em 7 anos de dados históricos de projetos reais e acompanhamento por 8 meses de um projeto vivo.

PALAVRAS-CHAVE: Dívida Técnica. Evolução do software. Métricas de Software. Engenharia de Software Baseada em Evidências. Engenharia de Software Empírica.

ABSTRACT

It is widely accepted that the cost of maintaining a software represents over 90% of the total cost of the product throughout its life cycle. The cost of product development tends to become insignificant on software with long lifecycles. Software lose their importance and tend to fall into disuse if not evolved. Of course, during this process of evolution various external and internal forces to corroborate the decay of the quality of software source code, such as market pressures, reduced budgets for maintenance, changes in the development team, among others. Although there are techniques to monitor the quality of the source code of a software using metrics, takes no account of the context and decisions that led to decreased quality becoming so reactive techniques. To explain this decay of quality, insert contextual information on decisions, allowing preventive planning and monitoring the rising costs of maintaining software in source code already committed, is used the metaphor of Technical Debt.

The Technical Debt metaphor is an analogy to the concept of financial loan. By compromising the quality of the source code of the system at the expense of some immediate benefit, it contracts a debt where the loan amount is saved the effort in completing the task. Like any debt, apply interest on borrowing and debt is not adequately controlled in the future most of the maintenance effort and progress will be spent paying down debt costs. Although intuitive, very popular in the community of software professionals and helpful to decrease the distance between the managerial and technical vocabulary, the metaphor has its definition on anecdotal evidence, so it is necessary studies to describe the phenomenon with scientific rigor.

In this context, the aim of this study is to characterize the metaphor of Technical Debt from the viewpoint of software metrics, describing the relationship between decision making and impacts on quality, using classical suites of metrics, based on 7 years historical data from real projects and monitoring for 8 months of a project alive.

KEY-WORDS: Technical Debt. Software evolution. Software Metrics. Evidence Based Software Engineering. Empirical Software Engineering.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS/ACRÔNIMOS

IEEE	*Institute of Electrical and Electronics Engineers*
ISO/IEC	*International Organization for Standardization / International Electrotechnical Commission*
J2ME	*Java Platform, Micro Edition*
MS Exchange	*Microsoft Exchange*
OOPSLA	*Conference on Object-Oriented Programming Systems, Languages, and Applications*
RMS	*Record management store*
TD	*Technical Debt*

1. INTRODUÇÃO

1.1 MOTIVAÇÃO

Atualmente o termo manutenção de software vem dando espaço ao termo *evolução* de software entendido como melhorias realizadas, de caráter corretivo ou evolutivo, com o objetivo de manter o software útil. De acordo com a primeira lei de de Lehman (1996) um programa gradativamente perde sua importância e consequentemente cairá em desuso caso não seja devidamente evoluído. Esse processo se dá por que o gap funcional entre as regras implementadas no software e as necessidades reais dos usuários ficará cada vez maior tornando o software obsoleto com o passar do tempo. Ainda Lehman afirma que a própria evolução do software faz com que sua qualidade interna decaia e consequentemente fique cada vez mais difícil modificar o software para reagir a mudanças exigidas pelo mercado.

O decaimento da qualidade interna do sistema durante seu processo de evolução é um fenômeno conhecido como Dívida Técnica ou ainda Débito Técnico (do inglês *Technical Debt*, ou TD). O conceito de Dívida Técnica foi inicialmente concebido como uma metáfora por Cunningham (1992) seu relato de experiência do OOPSLA 1992:

> *"A primeira vez que a qualidade do código é comprometida é como se estivesse incorrendo em dívida. Uma pequena dívida acelera o desenvolvimento até que seja pago através da reescrita do código. O perigo ocorre quando a dívida não é paga. Cada minuto em que o código é mantido em inconformidade, juros são acrescidos na forma de reescrita."* (CUNNINGHAM, 1992, tradução nossa)

Cunningham criou a metáfora como forma de traduzir a termos financeiros o decaimento da qualidade do código fonte de um software e as implicações de maior custo de evolução no futuro. O objetivo de Cunningham era promover uma melhoria na comunicação entre a equipe técnica e a equipe de negócios para que as consequências futuras sobre as decisões de comprometimento da qualidade em detrimento a ganhos de curto prazo fossem devidamente avaliadas. Sendo uma analogia à dívida financeira temos então que a decisão de comprometimento da qualidade é uma dívida que precisará ser paga no futuro. Ainda, juros serão acrescidos adicionando um esforço extra para realizar futuras intervenções no sistema. A qualquer momento a dívida pode ser amortizada ou paga através de *refactorig* do código.

A metáfora de Dívida Técnica tem ganhado muitos adeptos pela sua simplicidade para explicar um fenômeno conhecido por todo desenvolvedor, porém, ainda carente de estudos formais. Ainda, por ela conseguir exprimir ao time de negócios as reais consequências que as pressões de prazo podem acarretar.

Como a compreensão da metáfora ainda é intuitiva e as técnicas e práticas utilizadas atualmente não possuem fundamentação teórica e estudos formais para definir com rigor o fenômeno (BROWN, et al., 2010) há uma necessidade de estudos teóricos como o de Oliveira (2011), que caracteriza a dívida com dados empíricos de um projeto de software real coletados e analisados a partir de 5 anos de seu ciclo de vida.

Ainda, como medir e controlar o conjunto de "empréstimos" de um software gerenciando sua dívida? Qual o custo para se realizar esse gerenciamento? Algumas propostas de gerenciamento de itens de dívida bem como de acompanhamento desses itens já foram feitas (Guo, et al., 2011; GUO, 2009; GUO & SEAMAN, 2011), porém todas as propostas visam um controle qualitativo e subjetivo das dívidas e a coleta de dados é realizada manualmente pelo time de projetos sem auxílio de ferramental específico. Ainda, durante o estudo de (Oliveira, 2011) e experimentos posteriores percebeu-se que as técnicas de coleta manual eram custosas e que necessitavam de dados objetivos que embasasse tecnicamente os itens encontrados.

Nesse cenário o estudo apresentado aqui tenta trazer à metáfora de Dívida Técnica um conjunto de ferramentas e técnicas para subsidiar com dados objetivos os itens de TD. O ferramental explorado já é utilizado para avaliação objetiva da qualidade de código fonte de software, porém, ainda não relacionado com TD. Contudo, a proposta objetiva e quantitativa não excluem alguns dados subjetivos essenciais que caracterizam a dívida como por exemplo a decisão tomada e suas razões. Apenas automatiza a coleta de dados sobre localização da dívida e impacto da dívida bem como subsidia as estimativas melhorando sua precisão, sem com isso onerar o processo de gerenciamento da dívida, pelo contrário, automatizando boa parte desse processo.

1.2 OBJETIVOS

O objetivo geral desse trabalho é correlacionar itens de Dívida Técnica com métricas de software já consolidadas pela comunidade científica e utilizadas pelo mercado para diminuir o esforço de gerenciamento da dívida bem como aumentar a precisão sobre as estimativas do valor de dívida e dos juros a serem pagos no decorrer do tempo.

Podemos decompor nos seguintes nos seguintes objetivos específicos:

» Definir um conjunto de métricas de software que possam ser correlacionadas aos itens de dívida.

» Definir que métricas relacionadas a TD evidenciam a inocorrência ou variação da Dívida Técnica.

» Desenvolver um método para medição e controle das métricas de forma que possa ser utilizado por processo de gerenciamento de TD.

» Aplicar o método desenvolvido em um projeto de software real testando sua validade.

1.3 ESTRUTURA DA DISSERTAÇÃO

O restante desta dissertação procede da seguinte forma. O Capítulo 2 sumariza os trabalhos que são relevantes para a teoria. O Capítulo 3 discute os métodos usados no trabalho empírico e apresenta o estudo de caso. Capítulo 4 apresenta os achados do trabalho de pesquisa. Capítulo 5 discute as implicações práticas e teóricas dos achados. E finalmente, o Capítulo 6 conclui a dissertação com uma lista de questões em aberto.

2. EMBASAMENTO TEÓRICO E TRABALHOS RELACIONADOS

2.1 MANUTENÇÃO E EVOLUÇÃO DE SOFTWARE

Em 1976 os pesquisadores Lehman and Parr publicam um estudo quantitativo que traz as primeiras evidências sobre a degradação da qualidade e manutenibilidade de software durante seu ciclo de vida (Lehman, 1976). Segundo Lehman and Parr, essa degradação se intensifica após a fase de implantação onde os usuários do sistema costumam achar erros no software, descobrir novos usos e solicitar novas funcionalidades. Neste mesmo trabalho os autores alertam sobre a necessidade de se manter a estrutura inicialmente planejada para o software e um código limpo e organizado, o que normalmente não é praticado por questões de economia e falta de planejamento. “Ataca-se apenas o que dá retorno Financeiro imediato, retorno de curto prazo.”. Esse trabalho marca o início da área de estudo sobre manutenção e evolução de software. Em um trabalho posterior Lehman define 5 leis para evolução de sistemas de software (Lehman, 1980) que mais tarde tornam-se 8 leis (Lehman, 1996), comumente aceitas e referenciadas pela comunidade de engenharia de software, como pode ser visto em (Sommerville, 2007). Na **Tabela 1** são listadas as 8 leis de Lehman.

I	Mudança contínua	Um software deve ser continuamente adaptado, caso contrário se torna progressivamente menos útil.
II	Complexidade crescente	À medida que um software é alterado, sua complexidade cresce, a menos que um trabalho seja feito para mantê-la ou diminuí-la.
III	Autorregularão	O processo de evolução de software é autorregulados próximo à distribuição normal com relação às medidas dos atributos de produtos e processos.
IV	Conservação da estabilidade organizacional	A não ser que mecanismos de retroalimentação tenham sido ajustados de maneira apropriada, a taxa media de atividade global efetiva num software em evolução tende a ser manter constante durante o tempo de vida do produto.
V	Conservação da Familiaridade	De maneira geral, a taxa de crescimento incremental e taxa crescimento a longo prazo tende a declinar.
VI	Crescimento contínuo	O conteúdo funcional de um software deve ser continuamente aumentado durante seu tempo de vida para manter a satisfação do usuário.
VII	Qualidade decrescente	A qualidade do software será entendida como declinante a menos que o software seja rigorosamente adaptado às mudanças no ambiente operacional.
VIII	Sistema de Retroalimentação	Processos de evolução de software são sistemas de retroalimentação em múltiplos níveis, em múltiplos laços (loops) e envolvendo múltiplos agentes.

TABELA 1 - LEIS DE LEHMAN

A concepção de Lehman é que todo software precisa ser considerado como um projeto inacabado, um sistema dinâmico e em contínua mudança e evolução. Esse conceito deve ser

concebido desde o inicio do ciclo de vida do software, ou seja, o software precisa ser desenvolvido de forma tal que beneficie a sua manutenção e evolução (Lehman, 1980).

2.1.1 DEFINIÇÃO

Durante o desenvolvimento da área de manutenção e evolução de software muitas propostas de definição surgiram, porém todas endereçam 2 aspectos principais: que a manutenção trata de mudanças corretivas e mudanças evolutivas no produto de software. Lehman propõe a primeira definição do que é manutenção de software como qualquer alteração feita após a primeira instalação do programa, seja para correção de erros introduzidos em qualquer fase de seu ciclo de vida, ou alterações feitas por mudanças ambientais que exigem a adaptação do software (Lehman, 1980).

Atualmente o padrão ISO/IEC 14764:2006 (ISO/IEC, 2006) estabelece a definição base para a comunidade de engenharia de software. Ele utiliza-se de definições de padrões anteriores como IEEE/EIA 12207.0:1996 (IEEE/EIA, 1996) e IEEE 1219:1998 (IEEE, 1998) complementando-as. Segundo o padrão IEEE 1219:1998, Standard for Software Maintenance, a manutenção de software é definida por:

> *"... modificações do produto de software após sua implantação para correção de falhas, melhoria de desempenho e outros atributos ou adaptação do produto para modificações ambientais, por exemplo mudanças nos requisitos."* (IEEE, 1998)

Encontramos definição correlata no padrão IEEE/EIA 12207.0:1996, Standard for Software Life Cycle Processes, que considera:

> *"A manutenção de software são modificações no código e documentação relacionada com o intuito de resolver problemas ou realizar melhorias no software com o objetivo de modificar produto de software existente preservando sua integridade."* (IEEE/EIA, 1996)

A norma mais atual ISO/IEC 14764:2006 (ISO/IEC, 2006), Standard for Software Maintenance, se utiliza dos mesmos termos que a norma IEEE/EIA 12207.0 e IEEE 1219:1998, apenas enfatizando os aspectos de manutenção pré-release como, por exemplo, o planejamento da fase de manutenção.

2.1.2 NATUREZA E CLASSIFICAÇÃO

As intervenções em manutenção de software são divididas em dois grandes grupos: Ações de Correção e Ações de Evolução do software (ISO/IEC, 2006; Lehman, 1980; Pigoski, 1996; Sommerville, 2007; Mens & Demeyer, 2008) se relacionando com algumas características e subdivisões como pode ser visto na Figura 1.

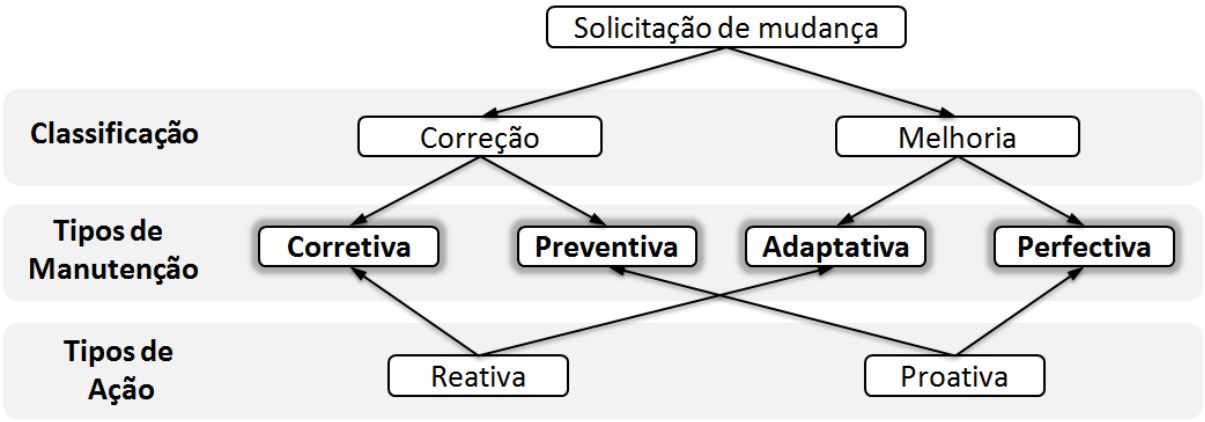


FIGURA 1 - CATEGORIAS DE MANUTENÇÃO DE SOFTWARE. ADAPTADO DA NORMA ISO/IEC 14764:2006.

As ações de correção podem ser corretivas e preventivas. As ações corretivas são aquelas que agem sobre uma falha encontrada no software durante seu uso no ambiente de produção. As ações corretivas são consideradas reativas, pois só trata o problema após seu acontecimento. As ações preventivas são aquelas que agem sobre uma falha que ainda não ocorreu no ambiente de produção, porém já foi identificada e considerada passível de acontecer. As ações preventivas são consideradas proativas, pois tratam os problemas antes que eles ocorram (ISO/IEC, 2006).

As ações de evolução podem ser adaptativas e perfectivas. As ações adaptativas são adequações do produto de software a mudanças ambientais como por exemplo, porting do software para uma nova plataforma, esse tipo de ação é considerada reativa. As ações Perfectivas são melhorias funcionais, de artefatos, organização ou desempenho do software, etc... Esse tipo de ação tenta prevenir possíveis problemas que podem ocorrer caso as melhorias não sejam implementadas. Como é de natureza preventiva é considerada proativa (ISO/IEC, 2006).

Tão importante quanto classificar os tipos de manutenção é entender como esses tipos se relacionam. Quando são realizadas ações adaptativas, perfectivas ou preventivas elas podem introduzir novos problemas e acarretar na necessidade de uma ação corretiva, que

por sua vez pode gerar a necessidade de uma nova ação preventiva (Grubb, 2003). Essa relação pode ser vista na Figura 2.

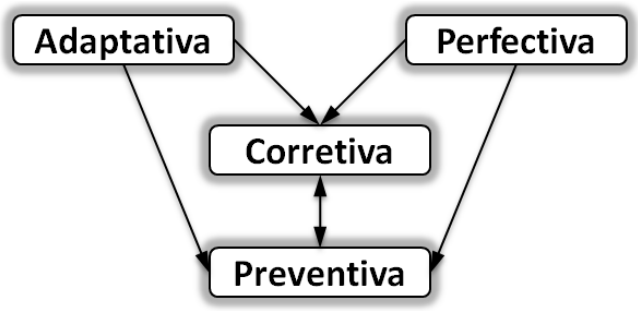


FIGURA 2- RELAÇÃO ENTRE OS TIPOS DE MANUTENÇÕES. RETIRADO DE (GRUBB AND TAKANG, 2003).

2.1.3 CUSTOS

Vários estudos realizados nas últimas décadas apontam para um custo crescente da manutenção e evolução de software durante seu ciclo de vida. A Figura 3 resume os resultados desses estudos (Pigoski, 1996). Provavelmente esse custo já ultrapassa o valor de 90% nos dias atuais. Ou seja, o custo de desenvolvimento de um novo sistema passou a ser pequeno perante o custo total que este sistema irá requerer durante sua vida. As explicações propostas para esse fato são que sistemas que foram desenvolvidos nos anos 1970 ou 1980 continuam ativos, sendo mantidos. Portanto, para esses sistemas, o custo do primeiro desenvolvimento, fixo, representa uma parte cada vez menor do custo total que não para de crescer a medida que o sistema continua ativo, sendo usado e modificado (Pigoski, 1996). Além do desenvolvimento de cada novo sistema contribuir para o aumento desse percentual ao longo do tempo, se desenvolvem sistemas cada vez maiores e mais complexos. Quanto maior e mais complexo o sistema, maior será o custo para evoluí-lo e modificá-lo (Lehman, 1980).

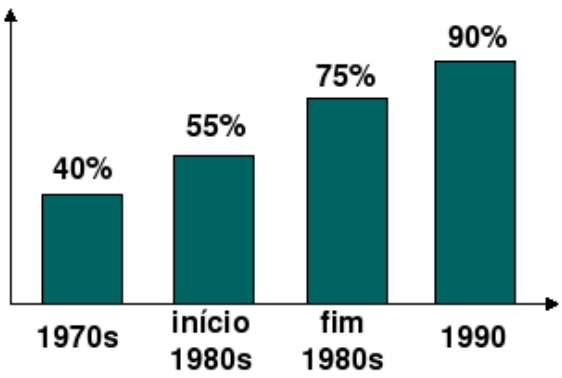


FIGURA 3 - EVOLUÇÃO DO CUSTO DE MANUTENÇÃO DE SOFTWARE NAS DÉCADAS DE 1970, 1980 E 1990.

Existem estudos que demonstram que o custo de correção de erros durante o ciclo de vida do software aumenta exponencialmente em relação ao tempo e atinge seus maiores valores na fase de manutenção do sistema (Grubb, 2003). A **Figura 4** ilustra essa tendência.

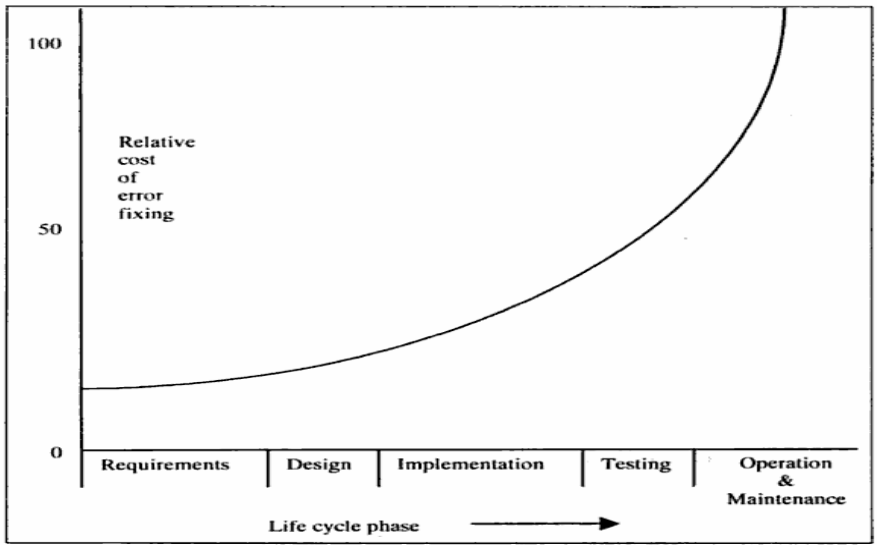
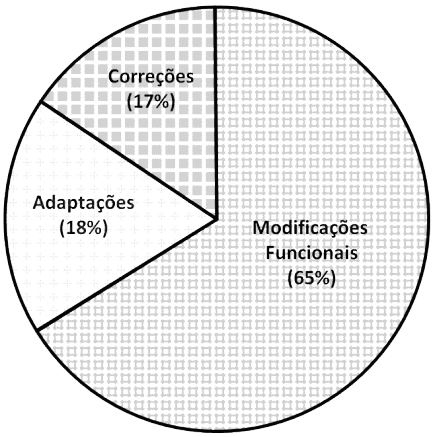
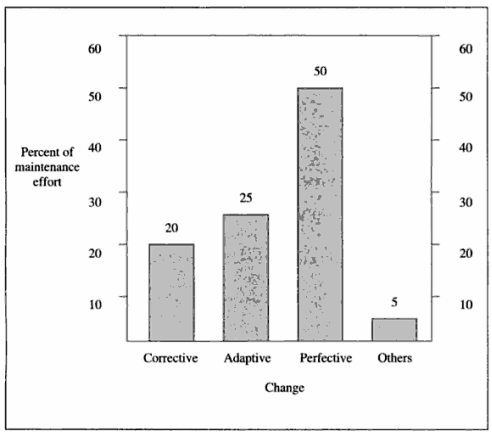


FIGURA 4 - CUSTO DE CORREÇÕES DURANTE O CICLO DE VIDA DO SOFTWARE. RETIRADO DE (GRUBB AND TAKANG, 2003)

Em contrapartida, outros estudos indicam que as correções de erros em sistemas representam apenas 17% do total de intervenções durante a fase de manutenção (Pigoski, 1996), conforme pode ser observado na Figura 5 (a) e que representam apenas 20% do custo total das intervenções (Grubb, 2003), como ilustra a Figura 5 (b). Isso não significa que o tratamento de erros nas fases iniciais do desenvolvimento de um sistema não é importante, apenas indica que a evolução das funcionalidades de um software deve ter tanta importância, ou maior, que a importância dada a prevenção e tratamento de erros atualmente.



(a) Percentual de tipos de intervenções em sistemas. Retirado de (Pigoski, 1996).



(b) Custos com os diferentes tipos de manutenção. Retirado de (Grubb, 2003).

FIGURA 5 - TIPOS DE INTERVENÇÃO (A) E CUSTOS DAS MANUTENÇÕES (B)

Sendo a manutenção de software uma atividade imprescindível na prática das organizações e correspondendo a mais de 90% do custo de um software durante todo seu ciclo de vida, torna-se clara a importância do desenvolvimento de métodos e técnicas eficientes para gerenciar as mudanças que esse software irá sofrer desde estágios iniciais do seu ciclo de vida. Apesar desta importância, na área de engenharia de software, a maior parte dos esforços se concentram em aprimorar métodos e técnicas de desenvolvimento com o objetivo de diminuir a necessidade de manutenção do software, presumidamente de natureza corretiva. O primeiro equívoco dessa abordagem é que a correção de erros representa uma pequena parte de todas as intervenções feitas em sistemas de software durante sua vida (Pigoski, 1996). Além disso, a evolução dos sistemas de software, do ponto de vista funcional, é um fator intrínseco e desejável devido às mudanças que acontecem no mundo real e não a características internas dos sistemas (Lehman, 1996).

Novos métodos e técnicas para prever impactos sobre decisões tomadas em relação as atividades de evolução e manutenção são imprescindíveis para seu efetivo gerenciamento. Com esse objetivo, a Dívida Técnica (Technical Debt) fundamentará e formalizará um conhecimento que é considerado verdadeiro pela área de engenharia de software, porém que ainda não tem mecanismos eficientes para medir e comprovar seus impactos. Espera se que a Dívida Técnica se torne uma poderosa ferramenta para gestão de impactos e apoio a tomada de decisões em manutenção e evolução de software.

2.2 THECNICAL DEBT

Cunhado por Ward Cunningham (Cunningham, 1992) o termo Dívida Técnica, ou Techinical Debt (TD) no original em inglês, descreve o encargo que uma organização incorre quando escolhe uma abordagem de projeto ou construção de software que é conveniente a curto prazo, porém que aumenta a complexidade e custo da manutenção do software a longo prazo, as vezes até mesmo a curto prazo ou médio prazo. Cunningham justifica a criação desse termo, na verdade metáfora como ele a considera, como uma forma de melhorar a comunicação entre pessoas da área de negócio das organizações com a área técnica. Essas pessoas entenderiam pouco de problemas técnicos de software, porém entendem bem o que é uma dívida e como os sistemas de juros sobre o montante da dívida funcionam. Dessa forma, as pessoas da área técnica conseguiriam passar completamente a severidade do adiamento de uma decisão técnica aos termos financeiros que essa acarretaria no futuro.

Assim como um empréstimo financeiros, a dívida técnica não é ruim por si só. Sendo bem usada e, principalmente mensurada e controlada, pode ser uma grande aliada ao gerenciamento de produtos de softwares. Porém não é possível controlar algo que ainda não possui métodos, técnicas ou indicadores para isso. Hoje a compreensão do TD é empírica e baseada no conhecimento tácito adquirido por profissionais ao longo dos seus anos de experiência. Existe pouca literatura sobre o tema e um conhecimento pulverizado sobre o conceito, mas não em seus termos, em livros de engenharia de software e gerenciamento de projetos de softwares além de muitas discussões em forums e blogs na internet.

2.2.1 CLASSIFICAÇÕES PROPOSTAS

Existem várias propostas para classificação de TD. A maioria delas não foram publicadas em trabalhos acadêmicos, nem tão pouco estudadas com métodos científicos. Essas propostas de classificação são fruto da experiência e observação de profissionais da área de desenvolvimento de software e serão listadas aqui para servir como primeira coletânea de classificações.

2.2.1.1 INTENCIONAL VS NÃO INTENCIONAL

A primeira classificação abordada será a de intencionalidade (McConnell, 2007).

- Não Intencional – A DT não intencional é gerada quando, por falta de conhecimento ou experiência, um profissional toma decisões erradas sobre o projeto ou construção de um software, ou ainda, quando a organização adquire um componente/pacote de software de terceiro sem conhecer a dívida técnica que esse já agrega.

- Intencional – A DT intencional, ou estratégica, ocorre quando a organização toma uma decisão de projeto ou construção de software para otimizar o esforço no presente consciente que no futuro essa decisão acarretará em um esforço maior que o economizado para manutenção desse software.

Primeiramente deve-se garantir que TD em projetos devam sempre ser do tipo intencional, ou seja, que esteja claro que as decisões tomadas terão consequências futuras. Mesmo que não possamos medir ainda, ou saber con exatidão, quais e quão caras serão essas consequências. O tipo Não Intencional é indesejado pelo simples fato de não existir nenhuma maneira de se gerenciar o que não se conhece.

2.2.1.2 CURTO VS LONGO

A classificação de Curto ou Longo Prazo, Short-term ou Long-term no original em inglês, indica quando a dívida será "cobrada", ou seja, quando será necessário dar atenção as decisões tomadas e despender o esforço economizado, com "juros" (Cunningham, 1992).

- Curto Prazo – O esforço economizado de imediato deve ser pago logo em seguida. Normalmente no lançamento de uma nova release do software.
    - Curto Prazo Isolado – Quando é introduzido um TD de curto prazo isolado e facilmente identificável.
    - Curto Prazo Espalhado – Quando existem vários TD pequenos espalhados de forma que dificulte sua localização e identificação.
- Longo Prazo – O esforço economizado poderá ser pago em períodos maiores.

O TD de curto prazo não é desejável, pois em pouco tempo tornam-se um ciclo vicioso que transforma o pagamento em ação reativa, ou seja, você não consegue pagar a dívida, apenas amortiza os "juros" gerados por ela incorrendo em novos TD de curto prazo. Já a dívida de longo prazo é a desejável, caso se tenha que incorrer em TD, pois permite a criação de uma estratégia e plano para pagamento da mesma.

2.2.1.3 FASE DO CICLO DE VIDA DO SOFTWARE

A terceira classificação tenta definir tipos de TD de acordo com a fase onde ele foi introduzido, ou seja, em que etapa do ciclo de vida do desenvolvimento do software ele foi criado (GUO, 2009).

- Dívida de Documentação – Acontece sempre que se adiciona, ou altera, o código fonte do software e a documentação em relação a ele não é atualizada.

- Dívida de Design – Acontece sempre que são geradas deficiências no software provenientes de práticas pobres de programação ou violações da arquitetura do sistema.
- Dívida de Teste – Acontece sempre que é tomada a decisão de não se testar alguma área da aplicação ou revisar código adicionado ao sistema.
- Dívida de Defeito – Ocorre quando defeitos conhecidos não são corrigidos para o lançamento da próxima release.

2.2.2 MOTIVAÇÕES PARA INCORRER EM TECHNICAL DEBT

Normalmente, incorrer em TD tem motivações de negócios, sejam táticas ou estratégicas. Não se incorre em TD se não há nenhuma compensação, a não ser que seja de forma não intencional. Quando se trata de TD intencional são propostas 3 motivações principais (McConnell, 2007):

- Time to Market – Quando atingir o mercado com um novo produto no menor prazo possível, lançar o produto antes do concorrente ou existir deadlines inadiáveis então pode-se optar por incorrer em TD intencional para diminuir o esforço de desenvolvimento do produto a curto prazo. Normalmente espera-se que o retorno gerado pelo adiantamento será maior que o custo adicional para manutenção do produto no futuro.
- Preservação do Capital Inicial – Quando se tem um capital inicial fixo para se desenvolver um novo produto, pode-se incorrer em TD intencional para preservar esse capital. Espera-se que o retorno que esse produto terá no futuro permitirá pagar pela sua manutenção mais cara e seu aprimoramento.
- Evitar Gastos Desnecessários – Quando um sistema é descontinuado, todo TD desse sistema é descontinuado com ele. Logo, quão mais próximo de ser descontinuado um sistema está mais difícil é de justificar o pagamento do TD ou de não incluir novos. Deixa-se de haver diferença entre uma modificação custosa porém “limpa” e uma modificação barata porém “suja”.

2.2.3 CARCTERIZAÇÃO DE DÍVIDA TÉCNICA

Em trabalhos recentes (Oliveira, 2011; Siebra, et al., 2012; GUO, 2009; Guo, et al., 2011; Gomes, et al., 2011; GUO & SEAMAN, 2011), propostas de coleta e caracterização de

itens de dívida técnica foram feitas a partir do estudo de toda história de um produto de software desde sua concepção até sua saída de mercado e arquivamento. No estudo os pesquisadores cruzaram os dados de 5 anos de trocas de e-mails, análise dos cronogramas de todas as evoluções do projeto, relatórios semanais gerenciais de acompanhamento de projeto bem como entrevista com o time envolvido no produto.

Em Gomes (2011) vemos a primeira proposta de caracterização de dívida técnica relacionada com suas componentes baseada em dados empíricos Figura **6**.

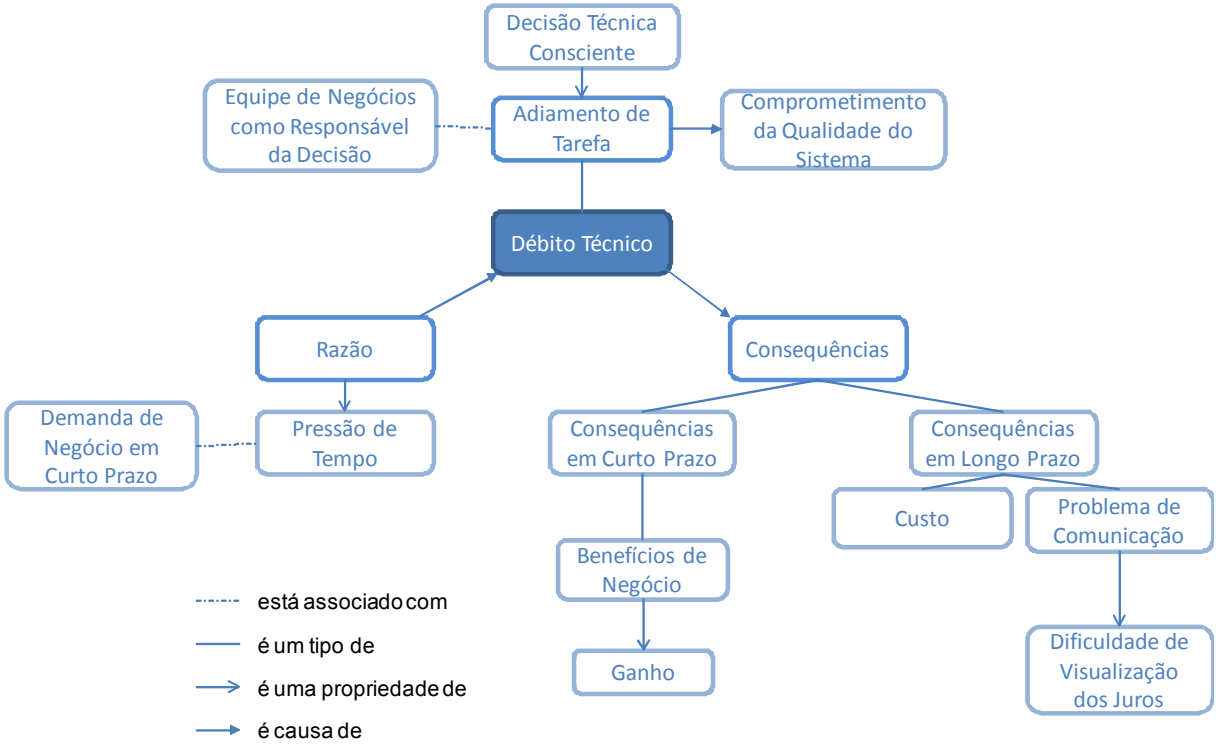


FIGURA 6 - CARACTERIZAÇÃO DE DÍVIDA TÉCNICA (OLIVEIRA, 2011)

Segundo essa classificação a dívida técnica é um tipo de adiamento de esforço, o que caracteriza a dívida. O adiamento de esforço é decidido pela equipe de negócio sendo consequência de uma decisão técnica consciente que compromete a qualidade do sistema. A dívida técnica tem suas razões e consequências, normalmente relacionadas com o esforço de manutenção presente e futura. Uma questão importante não avaliada com profundidade no trabalho de Oliveira foi a questão da intencionalidade da dívida. Para a Oliveira uma dívida técnica sempre é intencional, utilizando-se dos argumentos de (Bob, 2009), porém como já foi citado em (McConnell, 2007) admitir a dívida técnica como não intencional pode ser uma grande ferramenta para gerenciamento de bibliotecas com divida legada.

Em seu trabalho Oliveira (2011) consolida as principais referências sobre os conceitos relacionados à Dívida Técnica que podem ser vistos no Quadro 1.

QUADRO 1 - CONCEITOS RELACIONADOS À METÁFORA DE DÍVIDA TÉCNICA

Conceitos relacionados	Definição	Autores
Razão	Motivação que levou à decisão de comprometer a qualidade do sistema.	(BOB, 2009) (FOWLER, 2009) (MCCONNELL, 2007) (GUO, 2009)
Benefício	Consequências em curto prazo da contração da dívida.	(BOB, 2009); (CUNNINGHAM, 1992); (FOWLER, 2009); (GUO, 2009); (MCCONNELL, 2007)
Resultado	Consequências em longo prazo da contração da dívida.	(GUO e SEAMAN, 2011)
Principal	Valor que indica quanto custaria para pagar a dívida no momento.	(BOB, 2009); (CUNNINGHAM, 1992); (FOWLER, 2009); (GUO, 2009); (MCCONNELL, 2007)
Juros	Possível penalidade em decorrência da dívida (aumento de esforço ou impacto na produtividade de manutenção).	(BOB, 2009); (CUNNINGHAM, 1992); (FOWLER, 2009); (GUO, 2009); (MCCONNELL, 2007)
Ganho ou valor rendido	Valor gerado em resultado dos benefícios alcançados.	(BOB, 2009); (FOWLER, 2009)
Rentabilidade	Relação de custo/benefício. Considerando o custo para pagar o principal e os juros, e o valor ganho.	(GUO e SEAMAN, 2011)

Fonte: (Oliveira, 2011)

Além do discutido até então, temos um conjunto de propriedades já documentado por (BROWN, et al., 2010) que pode ser vistas no Quadro 2.

QUADRO 2 - CONJUNTO INICIAL DE PROPRIEDADES DE DÍVIDA TÉCNICA JÁ IDENTIFICADAS

Propriedade	Descrição
Visibilidade	Geralmente os problemas surgem quando a dívida não é visível o suficiente para todos os responsáveis pelas tomadas de decisão.
Valor	A dívida financeira é uma forma de criar valor. Em software, os atributos para definir tal propriedade são difíceis de serem elicitados.
Valor real	Além do valor do sistema criado pela Dívida Técnica, existem também os custos associados a contração da dívida, incluindo o impacto futuro.
Sinergia da dívida	A existência de muitas dívidas em um projeto pode levar o sistema a um estado irreparável, com relação a complexidade do código.
Ambiente	A dívida é relativa a um determinado ambiente.
Tipos de dívida	Quanto ao caráter deliberativo, a dívida pode ser estratégico, cometido intencionalmente, ou pode ser não intencional, cometido pelo uso de práticas ruins.
Impacto da dívida	A identificação dos componentes a serem modificados para o pagamento da dívida também é importante.

Fonte: Brown et al. (2010)

2.2.4 PROPOSTA DE GERENCIAMENTO DA DÍVIDA TÉCNICA

A caracterização da dívida técnica e seu rastreamento durante o ciclo de vida de um produto real permitiu testar o método de coleta e catalogação proposto por (GUO, 2009) e propor melhorias como em (Gomes, et al., 2011). Os formulários utilizados para catalogação das dívidas podem ser vistos no Quadro 3 e no Quadro 4. O formulário do Quadro 3 basicamente usa a proposta de (GUO, 2009) com a definição de campos:

- **Responsável** – Pessoa responsável por acompanhar a evolução da dívida no decorrer do tempo.
- **Tipo** – Tipo da dívida de acordo com a sessão 2.2.1.3.
- **Produto** – Nome do Software onde a dívida foi feito.
- **Data de Identificação** – Data quando a dívida foi feito.
- **Localização** – Descrição sobre onde a dívida pode ser encontrado no software.
- **Descrição** – Descrição sucinta da dívida.

- **Escopo Afetado** – Quais módulos/classes exatamente a dívida está.
- **Escopo Acoplado** – Quais módulos/classes podem ser afetados pela dívida.
- **Riscos** – Riscos relacionados à dívida.
- **Valor Base da Dívida** – Quantidade de Trabalho necessário para a mudança que não foi realizado
- **Valor Estimado do Juros** – Quanto de juros será pago
- **Probabilidade do Juros** – Qual a probabilidade de cobrança do juros.
- Tempo de Empréstimo – Identifica se o empréstimo deve ser pago em curto, médio ou longo prazo de acordo com a sessão 2.2.1.2.
- **Referêcias** – Qualquer documento, anotação ou comunicação que sejam relevantes para complementar as informações sobre a dívida.
- **Observações** – Qualquer outra informação não coberta pelos outros campos do formulário.

Item de Dívida Técnica			**ID 01**
Responsável		**Produto**	
Tipo		**Data de Identificação**	
Localização			
Descrição			
Escopo Afetado			
Escopo Acoplado			
Riscos			
Valor Base da Dívida		**Valor Estimado do Juros**	
Tempo de Empréstimo		**Probabilidade Estimada do Juros**	
Referencia			
Observações			

QUADRO 3 - FORMULÁRIO INICIAL PARA COLETA E CATALOGAÇÃO DE ITENS DE DÍVIDA TÉCNICA

Após a utilização do questionário em um ambiente de produção real notou-se a dificuldade que o time de desenvolvimento tinha para preenchê-lo e com isso a motivação e precisão para coleta de itens de TD decresceram em poucos ciclos de coleta e em alguns projetos completamente abandonada. Essa foi a principal motivação para simplificar o formulário na tentativa de viabilizar a coleta de itens de TD. O formulário simplificado, Quadro 4, retira alguns campos de detalhamento de informações e adiciona um campo relativo a intencionalidade da dívida com o objetivo de avaliar a classificação discutida na sessão 2.2.1.1 e não analisadas em (Oliveira, 2011).

Item de Dívida Técnica			ID 01
Responsável		**Produto**	
Tipo		**Data de Identificação**	
Localização			
Descrição			
Valor Base da Dívida		**Valor Estimado do Juros**	
Intencional		**Probabilidade Estimada do Juros**	

QUADRO 4 - FORMULÁRIO SIMPLIFICADO PARA COLETA E CATALOGAÇÃO DE ITENS DE DÍVIDA TÉCNICA

O formulário do Quadro 4 acima ainda é objeto de estudo e está sendo testado em experimentos com projetos reais para validar sua viabilidade bem como eficácia.

Discutiremos com mais detalhes a operacionalização da coleta de dados utilizando os formulários, as dificuldades encontradas na sessão de metodologia.

Apesar da metáfora de Dívida Técnica ser intuitiva e convincente, e já existirem estudos que trazem luz para algumas de suas características e conceitualização. Muitos pontos desta metáfora ainda não estão bem definidos. O Quadro 5 traz as questões de pesquisa em aberto sobre Dívida Técnica identificadas durante o 1º Internacional Workshop of Technical Debt (BROWN, et al., 2010).

QUADRO 5 - SUMÁRIO DAS QUESTÕES EM ABERTO SOBRE DEBITO TÉCNICO

	Questões de pesquisa
1	A dívida Técnica é uma metáfora apropriada para o gerenciamento de investimentos para medidas corretivas, urgentes e inesperadas, em projetos de desenvolvimento de software? Se não, existe alguma metáfora mais apropriada?
2	A metáfora de Dívida Técnica, ou alguma outra mais apropriada, pode levar a teorias úteis e testáveis sobre como usar, medir e pagar as dívidas?
3	Como identificar Dívida Técnica em um projeto de desenvolvimento de software?
4	Quais são os tipos de Dívida Técnica?
5	Quais as propriedades de Dívida Técnica? Quais parâmetros podem ajudar a identificar, comunicar, analisar e gerenciar Dívida Técnica?
6	Como o Dívida Técnica está relacionado a evolução e manutenção de software?
7	Como informações sobre Dívida Técnica podem ser coletadas empiricamente para desenvolver modelos conceituais?

8	Como o Dívida Técnica pode ser visualizado e analisado?
9	Para um determinado contexto, quais as maiores fontes de Dívida Técnica? Quais as mais úteis? Quais as mais custosas?
10	Como medir os diferentes tipos de Dívida Técnica?
11	Quais *thresholds* utilizar para monitorar TD?
12	Como analisar o *tradeoff* entre os requisitos arquiteturas do sistema? Como analisar sistematicamente como e quando redesenhar a arquitetura da aplicação para pagar a dívida?
13	Como identificar oportunidades de *refactoring*? Como avaliar as alternativas sugeridas?

Fonte: Brown et al. (2010)

2.3 MÉTRICAS DE SOFTWARE

Dada a importância do gerenciamento de Dívida Técnica, e como foi explanado na sessão 2.2.4 que a coleta de dados para gerenciamento de Dívida Técnica requer o envolvimento de pessoas chave e um esforço considerável para coleta de dados, além de estimativas que precisam de um grande conhecimento técnico do software e engajamento do time gerencial e técnico, fica evidente a necessidade de melhoria do processo de gerenciamento de TD. Nesse contexto as métricas de software podem ajudar tanto na fase de coleta de dados para TD bem como no acompanhamento do valor da dívida durante o ciclo de vida do produto.

A história de métricas de software remota a década de 70 quando Wolverton em seu trabalho relacionava a produtividade de desenvolvimento com o número de linhas de código escritas por programador propondo o critério "*person/month*" como uma métrica para esforço de desenvolvimento (Wolverton, 1974). Frederic Brooks, por sua vez, publicou em 75 o livro The Mithical Man/month (Brooks, 1975), que dentre outros assuntos, trata da fragilidade da medição de produtividade pela medição de linhas escritas por um programador. Brooks traz a tona a primeira referencia sobre a influencia de fatores humanos na produção de software e a primeira crítica a utilização de métricas de software desassociada de fatores ambientais. Apesar de Wolverton ter sido o primeiro a medir software com métricas, Halstead é que é considerado o pai da avaliação da qualidade do código fonte a partir de métricas (Halstead, 1972). Nesse trabalho ele define um conjunto de métricas baseadas na teoria da informação.

A discussão da validade da utilização de métricas de software para medição de produtividade e qualidade do produto de software gerado perdura até os dias de hoje, porém atualmente, vemos cada vez mais estudos empíricos com uma grande amplitude de coleta de dados em projetos reais que evidenciam a correlação de métricas de software com a qualidade do código fonte de um produto de software. Podemos ver isso em (Basili & Melo, 1996) que foram os primeiros a executar um estudo empírico para validação da mais aceita suíte de métricas da atualidade, (Olague, 2007) que faz uma validação empírica de duas suítes clássicas e uma terceira suíte derivada utilizando-se de software open source, (Lincke, 2010) que também estuda duas suítes clássicas e softwares open source, porém usando abordagem GQM , (Vasilescu, 2011) que propõe uma técnica para agregar as duas suítes clássicas e aplica sua técnica a softwares reais e, por fim, no estudo de (Rüdiger, 2007) que mapeia as principais métricas de software propostas com modelos de qualidade propostos pelas normas ISO 9004:2000, 9001:2000, 9126-1 e 9126-3 aplicando os experimentos em 4 projetos em 4 empresas diferentes, cada projeto utilizando uma linguagem de programação diferente e paradigmas diferentes. Esse estudo é o mais completo encontrado na literatura sobre a comprovação empírica da relação de métricas de

software com a qualidade do código fonte gerado e a posterior manutenibilidade do produto.

2.3.1 MÉTRICAS CLÁSSICAS

2.3.1.1 NÚMERO DE LINHAS DE CÓDIGO

A primeira métrica utilizada no desenvolvimento de software (Wolverton, 1974) é utilizada até hoje como unidade de esforço de desenvolvimento e evolução, bem como produtividade, como por exemplo na popular metodologia para estimativa de esforço COCOMO (Barry Boehm, 1995) e nas suítes clássicas como a C&K (Chidamber, 1994) e na MOOD.

Chamada de SLOC (Source Lines of Code) ou simplesmente LOC (Lines of Code) mede o número de linhas físicas de código excluindo-se linhas em branco e comentadas. Pode ainda ser classificado como SLOC lógico, que mede o número de comandos de um programa, sendo essa unidade dependente da linguagem de programação em questão. Por exemplo em Java SLOC Lógico é definido pelas ocorrências de ponto e vírgula e contagem de abertura de blocos de comandos não vazios. Dadas as grandes dimensões dos programas atuais, comumente é utilizada a unidade de milhares de linhas de código para medição ou KLOC.

Como já falado, a medição de produtividade e esforço utilizando-se apenas o SLOC desassociado de qualquer outra métrica é bastante questionado, já que a funcionalidade e complexidade não são necessariamente proporcionais ao SLOC. É comum termos trechos de código pequenos, de extrema complexidade e difíceis de manter o que leva a um grande esforço de desenvolvimento e manutenção, bem como trechos com centenas e até milhares de linhas de código de fácil manutenção e baixa complexidade o que diminuiria o esforço total para desenvolvimento e manutenção (Hyatt, 1995).

2.3.1.2 COMPLEXIDADE DO CÓDIGO

Métricas de complexidade de código são úteis para avaliação do quão difícil um software é de ser entendido, mantido ou desenvolvido. Normalmente é associada com o SLOC para estimativa de esforço de desenvolvimento e evolução. Ainda, a complexidade é utilizada para identificação dos módulos de um software que são mais propensos a defeitos, ou seja, os que devem ser melhor inspecionados e testados dado que quanto mais complexo o código mais difícil é deixá-lo livre de erros (Olague, 2007) e testá-lo (Laing V., 2001).

A métrica mais popular de complexidade foi proposta na década de 1970 pelo pesquisador McCabe que a chamou de Complexidade Ciclomática (McCabe, 1976) e continua útil até os dias de hoje para a maioria das linguagens de programação atuais.

A idéia fundamental da métrica de McCabe é mapear todos os fluxos possíveis que um programa pode tomar e montar um grafo com esses fluxos. A complexidade é definida por: Número de Arestas – Número de Nós + 2.

Números altos de complexidade devem ser evitados (acima de 10), pois tornam o código difícil de entender e manter, como pode ser visto na **Tabela 2**.

Complexidade	Classificação
Entre 1 e 10	Baixa complexidade. Fácil entender e testar.
Entre 11 e 20	Média complexidade. Relativamente difícil entender e testar.
Entre 21 e 50	Alta complexidade. Difícil entender e testar.
Maior que 51	Altíssima complexidade. Impossível entender e testar.

TABELA 2 - CLASSIFICAÇÃO DA COMPLEXIDADE CICLOMÁTICA

No código exemplo da **Listagem 1** definimos um algoritmo com uma estrutura de repetição e duas estruturas condicionais aninhadas. No caso de manutenção ou testes, o desenvolvedor ou testador deve compreender todas as possibilidades de fluxo que o algoritmo pode tomar, o que pode ser observado na **Figura 7**. Se utilizarmos o cálculo de McCabe já comentado temos: 11 arestas – 9 nós + 2 = 4, ou seja o valor da complexidade ciclomática desse código é igual a 4 o que o define como fácil de entender e testar.

```
01: public void doSomething(){
02:
03:     int x =100;
04:     int y =0;
05:
06:     while(x>y){
07:
08:         if (y % 2 == 0){
09:            x --;
10:         }else{
11:
12:           if (x % 2 == 0){
13:              System.out.println("x:"+x);
14:           }else{
15:              System.out.println("x:"+x);
16:           }
17:
18:         }
19:
20:         y++;
21:     }
22:
23: }
```

LISTAGEM 1 - EXEMPLO DE COMPLEXIDADE CICLOMÁTICA

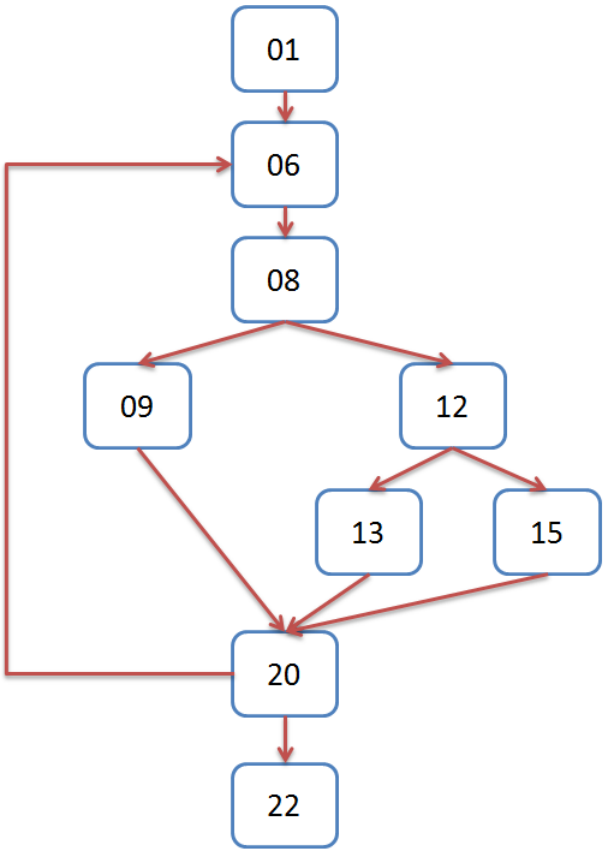


FIGURA 7 - FLUXOS DE EXECUÇÃO COMPLEXIDADE CICLOMÁTICA

2.3.1.3 PERCENTUAL DE LINHAS DE CÓDIGO COMENTADO

O percentual de linhas de código comentado é uma métrica que permite avaliar quão fácil é de entender um código e por consequência evolui-lo. Esse percentual pode ser calculoado pela razão entre o número de linhas comentadas e o número de linhas de código excluindo-se as linhas em branco. Segundo (Hyatt, 1995) um percentual efetivo de comentários deve girar em torno de 30%.

2.3.2 SUITE DE MÉTRICAS DE CHIDAMBER E KEMERER (CK)

Em 1994 Chidamber e Kemerer publicaram o que viria a se tornar a mais utilizada, popular e estudada suíte de métricas até então já definida (Chidamber, 1994). O objetivo dessa suíte era permitir que os desenvolvedores pudessem acompanhar a evolução do se seu código fonte subsidiando melhores escolhas de projeto no que diz respeito ao paradigma de Orientação a Objetos (Li, 1993). A suíte Chidamber e Kemerer (CK) propõem seis métricas apresentadas a seguir.

2.3.2.1 WEIGHTED METHOD PER CLASS (WMC)

WMC mede a complexidade total de uma classe a partir do cálculo da complexidade de seus métodos individuais. Os autores não prescrevem como medir a complexidade de cada método argumentando que para tornar o uso da WMC mais abrangente, a definição de que medida de complexidade usar é dada ao desenvolvedor. De forma geral utiliza-se a medida de Complexidade Ciclomática (McCabe, 1976) para cada método e o WMC é o somatório da Complexidade Ciclomática de cada método da classe, ou seja, a complexidade total da classe. Como já discutido, os valores de Complexidade Ciclomática devem ser baixos, são desejáveis então baixos valores de WMC. Também como no caso da Complexidade Ciclomática, o WMC indica o quão difícil é de entender uma classe e consequentemente mantê-la e evolui-la.

2.3.2.2 DEPTH OF INHERITANCE TREE (DIT)

DIT define a profundidade de uma classe em sua genealogia, ou seja, quão profunda na árvore de herança ela se encontra. Podemos entender também o DIT como quantas classes ancestrais podem afetar o comportamento da classe atual. A herança é o mais forte tipo de acoplamento entre classes, existem até propostas de padrões para se evitar a herança e seu demasiado acoplamento como pode ser visto em (Gamma, 1994). Quanto mais profunda na árvore de herança uma classe estiver, mais métodos ela deve herdar e com isso mais complexo é seu comportamento e mais suscetível a impactos por modificações em suas ancestrais. Pequenos valores de DIT são desejados, o valor 1 é o ideal (a classe não possui ancestrais).

2.3.2.3 NUMBER OF CHILDREN (NOC)

NOC mede a quantidade de filhos que uma classe possui. Essa medida permite determinar o nível de reutilização de um sistema. Quanto maior o número de NOC maior tende a ser a reutilização. Isso implica em um menor esforço de testes, dado que garantindo o comportamento do método da classe pai, os filhos o herdarão já devidamente testado. Diferente do DIT, que podemos ver como herança em profundidade, o NOC, herança em largura, é desejável.

2.3.2.4 COUPLING BETWEEN OBJECTS (CBO)

CBO mede o quão acoplada uma classe está a outra. O acoplamento considerado no CBO é a utilização de métodos e atributos. Um alto valor de CBO indica uma baixa capacidade de reutilização de classes. Uma visão mais granular do CBO define o acoplamento aferente, quantas classes referenciam a classe atual e o acoplamento eferente, quantas classes a classe atual referencia.

2.3.2.5 RESPONSE FOR A CLASS (RFC)

RFC mede a complexidade da classe em termos de chamadas de método. O RFC pode ser obtido somando-se o número de métodos da classe, não são considerados os herdados, com o número de chamadas distintas feitas a cada método. Cada chamada é contada apenas uma vez mesmo que o método seja chamado mais de uma vez e em métodos diferentes. Altos números de RFC indicam um relacionamento de chamadas internas complexo, ou seja, uma classe difícil de manter.

2.3.2.6 LACK OF COHESION IN METHODS (LCOM)

LCOM mede a falta de coesão de uma classe avaliando pares de métodos que partilham atributos da classe. O cálculo é feito obtendo-se todo o conjunto de pares de métodos que utilizam pelo menos um atributo em comum da classe. Calculam-se então todos os pares de métodos que não usam nenhum atributo em comum. O LCOM é obtido pelo número de pares do primeiro conjunto menos o número de pares do segundo conjunto. Para manter a coesão de um sistema valores baixos de LCOM são desejáveis.

2.3.3 Suite de Métricas para Design Orientada a Objetos (MOOD)

A suíte de métricas MOOD (Métricas para Design Orientada a Objetos) foi definida no trabalho de (Abreu & Carapuça, 1994) com o mesmo objetivo das métricas já definidas, avaliar a qualidade do código fonte de um software de forma quantitativa e automatizada. Abreu e Carapuça trazem uma perspectiva diferente à sua suíte, a de avaliar o mecanismo estrutural básico do paradigma de orientação a objetos, ou seja, métricas que avaliam o encapsulamento (MHF, AHF), a herança (MIF, AIF), o polimorfismo (POF), e a troca de mensagens (COF). As unidades básicas utilizadas para todas as métricas da suíte MOOD são os atributos da classes, os métodos da classe e seus relacionamentos. Nas sessões seguintes apresentam uma explicação sucinta sobre cada uma das métricas.

2.3.3.1 Method Hiding Factor (MHF)

O MHF é uma medida de funcionalidade da classe. A idéia fundamental é avaliar a proporção dos métodos públicos em relação aos privados, medir quanto o sistema está se beneficiando do encapsulamento. O número de funcionalidades de uma classe (métodos públicos) deve ser significativamente inferior ao número de métodos totais favorecendo um alto MHF.

A MHF baixo indica uma implementação que se utiliza pouco de abstração. Uma grande parte dos métodos são públicos e a probabilidade de erros é alta.

A MHF muito alta indica poucas funcionalidades em uma classe, ou seja, pode também indicar que o projeto da classe inclui um número elevado de métodos especializados que não passíveis de reutilização.

2.3.3.2 Method Inheritance Factor (MIF) e Attribute Inheritance Factor (AIF)

Uma classe que herda vários métodos (atributos) de suas classes ancestrais contribui para uma alta MIF e AIF. A classe filha que redefine seus métodos ancestrais (atributos) e acrescenta novos contribui para um menor MIF e AIF. Uma classe independente que não herda e não tem filhos contribui para uma menor MIF e AIF.

MIF e AIF devem estar em uma faixa razoável, não muito baixa e não muito alto também. Um valor muito alto indica uso abusivo de herança. Um valor muito baixo indica ausência de de herança ou uso indiscriminado de sobreposições.

2.3.3.3 POLYMORPHISM FACTOR (POF)

Fator de polimorfismo mede o grau de método de substituição na árvore de herança de classe. É igual ao número de substituições de método real dividido pelo número máximo de substituições de métodos possíveis. Ela indica o nível de reuso entre o pai e sua descendência e a abstração entre classes. Extremos de valores, altos e baixos, não são desejáveis para POF.

2.3.3.4 COUPLING FACTOR (COF)

Fator de acoplamento mede os acoplamentos reais entre as classes em relação ao número máximo de acoplamentos possíveis não levando em consideração o acoplamento criado pela herança. Como o acoplamento aumenta a complexidade. Reduz o encapsulamento e a reutilização valores elevados de COF devem ser evitados.

2.3.4 OUTROS MODELOS E MÉTRICAS

Além das suítes CK e MOOD diversos outros pesquisadores já propuseram métricas com o mesmo objetivo clássico: avaliação da qualidade do código fonte de um software a partir de indicadores quantitativos que pudessem ser calculados automaticamente a partir do próprio código fonte ou ainda dos documentos escritos em UML. A seguir são descritos algumas dessas outras propostas.

Em Lorenz e Kidd (Lorenz, 1994) são propostas métricas que consideram o tamanho a partir da contagem de atributos e operações para uma classe individual e valores médios para o sistema, herança com o modo pelo qual as operações são reusadas ao longo da

hierarquia de classes, estrutura interna avaliando coesão e operação e estrutura externa avaliando o acoplamento.

Em Bellin, Tyagi e Tyler (Bellin, et al., 1994) discutem métricas de acordo com a qualidade de abstração de um sistema OO. Basicamente temos três conjuntos: métricas para o número de métodos de uma classe, métricas para o número de atributos de uma classe, métricas para o tamanho da hierarquia de herança. Em seguida métricas de avaliação de reuso e após isso métricas para avaliação de acoplamento e coesão.

Encontramos ainda outros autores como (Li, 1993) que evidenciam uma grande preocupação com a mudança de características das métricas dependendo da linguagem de programação e plataformas utilizadas.

3. METODOLOGIA

Tendo sido estabelecido o primeiro conjunto de características de Dívida Técnica (Oliveira, 2011) possuímos um constructo mínimo para exploração de suas correlações com diversas outras variáveis do desenvolvimento de software. Com isso é possível desenvolver ferramental técnico e metodológico para gerenciamento de itens de TD a partir de métodos quantitativos e automáticos.

Esse trabalho aborda a identificação e acompanhamento automatizado dos itens de Dívida Técnica em um produto de software. Os métodos e técnicas propostos objetivam auxiliar o time de desenvolvimento e a gerência no controle da dívida tomada para o produto.

As perguntas que esse trabalho buscou responder foram:

1. É possível correlacionar Itens de TD com Métricas de Software?
2. Quais as métricas relevantes para descoberta e acompanhamento de itens de TD?
3. Como as métricas devem ser utilizadas com o intuito de dar suporte ao time de desenvolvimento e gerência na descoberta e acompanhamento de itens de TD?

Esse capítulo descreve as escolhas metodológicas feitas, bem como, uma definição breve dos métodos adotados para seleção da amostra, coleta e análise de dados e aplicação experimental.

3.1 DESENHO DO ESTUDO

Essa pesquisa está dividida em duas grandes etapas: a primeira realiza um estudo de caso quantitativo explanatório que busca responder a questão 1. A primeira discussão sobre a escolha de um estudo de caso explanatório é sobre a natureza dos estudos de casos serem de caráter exploratórios e não explanatórios. Em (Flyvbjerg, 2006) esse é o terceiro dos 5 mal-entendidos comuns sobre pesquisas utilizando estudos de caso.

> *"Mal-entendido 3: O estudo de caso é mais útil para gerar hipóteses; esse é o primeiro estágio do processo de pesquisa total. Após isso outros métodos de pesquisa são mais adequados para teste de hipóteses e construção teórica."* (Flyvbjerg, 2006) Tradução nossa.

O autor explica que apesar de comumente os estudos de caso serem de caráter exploratório eles não precisam estar limitados a esse propósito. Essa é uma visão compartilhada por (Runeson, 2009) incluindo os estudos confirmatórios. O principal motivo

de escolha dessa abordagem foi permitir que um estudo quantitativo fosse feito sobre um método com design flexível. O objetivo principal dessa etapa é explicar a correlação de métricas de software com itens de TD. A segunda etapa da pesquisa busca reafirmar os achados da primeira etapa em um novo caso e responder as perguntas 2 e 3 com o método de Pesquisa-Ação (Easterbrook, 2008) onde um experimento é realizado e o pesquisador controla e modifica suas variáveis no decorrer da experimentação. Como a segunda etapa tinha caráter tanto experimental confirmatório quanto explanatório o método Pesquisa-Ação se mostrou mais adequado a esse objetivo. A segunda etapa visa explicar quais métricas são relevantes a descoberta de itens de TD bem como o acompanhamento da dívida.

3.2 SELEÇÃO DA AMOSTRA

Na primeira etapa do estudo a seleção da amostra do caso a ser estudado bem como suas unidades já haviam sido definidas no estudo de Oliveira (2011) (Oliveira, 2011). O objeto de estudo dessa etapa é uma aplicação para plataformas móveis, implementado em J2ME, usada para acesso a serviços de e-mail, contatos e calendário no servidor Microsoft Exchange Server traduzido para 18 idiomas introduzido no mercado Europeu, Asiático e Sul-americano. Esse software possui 63.218 linhas de código e foi desenvolvido utilizando uma metodologia híbrida com ciclo geral RUP e algumas práticas XP. O software teve seu início de desenvolvimento em 2005 fechando seu ciclo de vida e sendo retirado oficialmente do mercado em 2011. A primeira release foi lançada em Abril de 2016 sendo considerada prova de conceito. A primeira release liberada para o mercado foi em Julho de 2007. As releases estudadas são mapeadas diretamente entre as eras definidas pelo estudo de 2011 de Oliveira ilustradas na Figura 8, como com algumas seleções intermediárias que permitem a avaliação da evolução do código via métricas. A seleção das releases com suas descrições pode ser observada no Figura 8 - Eras e decisões do produto avalidao na primeira etapa.

Quadro 6.

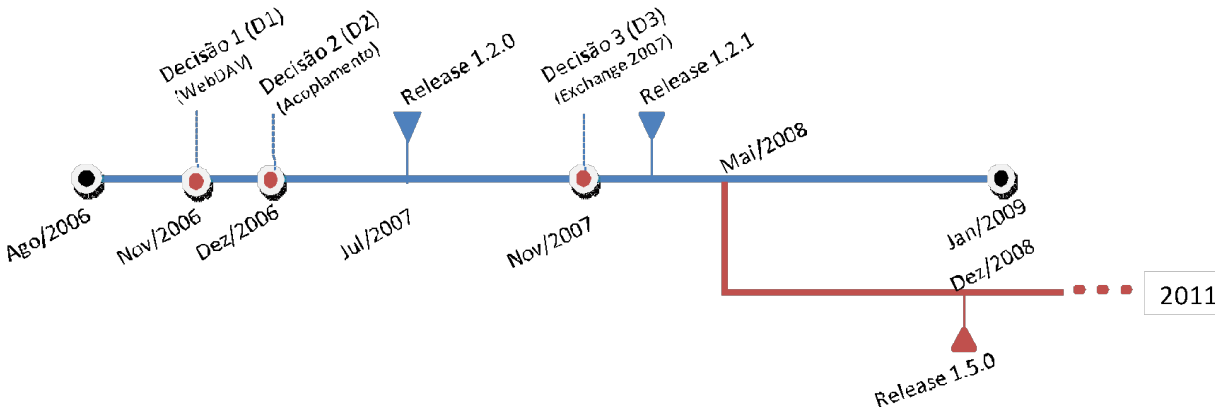


FIGURA 8 - ERAS E DECISÕES DO PRODUTO AVALIDAO NA PRIMEIRA ETAPA. (OLIVEIRA, 2011)

QUADRO 6 - RELEASES DO PRODUTO AVALIADO NA ETAPA 1

Release	Lançamento	Descrição
1.0.0	Abr/2006	Primeira release considerada prova de conceito.
1.1.2	Abr/2007	Primeira release implementando o protocolo WebDav
1.1.4	Ago/2007	Primeira release liberada para o mercado.
1.2.9	Jun/2008	Primeiro porting para outro dispositivo móvel.
1.2.22	Jan/2009	Release final utilizando o protocolo WebDav
1.5.1.5	Set/2008	Primeira release utilizando o protocolo ActiveSync
1.5.2.0	Fev/2009	Primeiro porting para outro dispositivo da versão 1.5.
1.5.3.0	Mar/2009	Segundo porting versão 1.5
1.5.4.0	Mai/2009	Grande mudança de código de persistência de dados
1.5.7.4	Jan/2010	Outra significativa mudança no sistema de armazenamento
1.5.7.17	Abr/2010	Pequena adição de língua estrangeira
1.5.8.1	Jan/2010	Novas mudanças na persistência
1.5.8.3	Fev/2010	Ultima release com nova funcionalidade antes do descontinuamento do produto.

Fonte: Elaboração Própria

Na segunda etapa do estudo foi selecionado um software com algumas características funcionais semelhantes, porém em plataforma distinta do objeto da primeira etapa. O software selecionado é um cliente para um ambiente de ensino a distância implementado em na plataforma Android com 41931 linhas de código. O projeto foi acompanhado por 76 releases lançadas entre dezembro de 2011 e outubro de 2012, tomamos como base o lançamento de releases semanais, porém em algumas situação aconteceu o lançamento de mais de uma release por semana. A aplicação foi desenvolvida fora do centro atual sendo todo código na release 2.0.0 considerado código legado. Ainda o código nessa release inicial foi analisado por dois líderes técnicos do projeto sendo considerado como “sujo” e difícil de manter, logo com uma clara dívida legada, o que nos remete a discussão de TD não intencional. Ainda algumas dessas releases sofreram modificações profundas de código com o objetivo de pagar a dívida técnica existente, porém o time durante essas releases ainda não tinham conhecimento sobre o que é dívida técnica nem que a qualidade do código estava sendo acompanhada.

3.3 FERRAMENTAL UTILIZADO

Inicialmente foram definidos alguns critérios para seleção do software a ser utilizado para coletar as métricas:

C1. Análise de código Java – Tanto para J2ME como para Android, dado que os projetos a serem analisados seriam nessas plataformas.

C2. Software Open-source – Dada a definição abstrata de algumas métricas, a forma de implementação das mesmas varia de ferramenta para ferramenta, com isso algumas ferramentas tendem a ter valores ligeiramente diferentes para mesmas métricas. Sendo o software open-source, dado que exista alguma divergência de valores, o cálculo da métrica pode ser inspecionado e aprovado para uso no estudo. Isso garante a transparência em relação aos resultados obtidos. Um restrição a esse critério é que o projeto open-source estivesse ativo e sendo constantemente evoluído.

C3. Popular entre acadêmicos e mercado – Foram priorizadas as ferramentas que fosse reconhecidas tanto na academia quanto no mercado. O critério utilizado para avaliação dessa popularidade foi a ocorrência de estudos acadêmicos e números de resultados em mecanismos de busca sobre a ferramenta.

C4. Cobertura de métricas – Dada a natureza exploratória e explanatória do estudo, de início naturalmente um maior número de métricas colhidas possibilitaria uma análise mais ampla de correlações com Dívida Técnica.

C5. Facilidade de uso em projetos reais – Uma ferramenta de fácil utilização, instalação, que realize coletas de métricas de forma autômata é preferível principalmente na segunda etapa do estudo e em projetos reais onde haja o gerenciamento de Dívida Técnica. Isso inclui a facilidade de acompanhar a evolução da qualidade de código no decorrer do projeto.

Após a definição dos critérios foi executada uma busca em dois indexadores científicos (ACM e IEEE Explore) com a string de busca “Software Metrics Tool” e foram avaliados os 200 primeiros resultados de cada indexador, ordenados por data, para formar uma lista inicial de ferramentas discutidas academicamente. Após isso, foi usada a mesma string de busca em inglês e português e analisados os 300 primeiros resultados da busca, tanto para o inglês como para o português, totalizando 600 links acessados. Foram listadas todas as ferramentas de métricas citadas em cada link. Não houve um critério de exclusão de links, todos os links desde a Wikipedia, Forums, Blogs, e sites comerciais foram acessados. Após a criação de uma lista inicial com quase 100 ferramentas de métricas (algumas ferramentas comentadas em nos links como de métricas na verdade tinham outros fins mas entraram na lista inicial de ferramentas. Após isso, foram eliminadas as entradas que não possuíam Web Site, ou que os links para os mesmos já não funcionavam. A lista final contava com as 39 entradas listadas no Quadro 7. Admite-se que por esse método de listagem inicial as ferramentas mais relevantes seriam catalogadas dado que elas deveriam ter ranking elevado em no mecanismos de busca Google, ainda, as já estudadas e comentadas pela academia também entrariam na lista. Após a montagem da lista o trabalho constava em acessar o site de cada ferramenta buscando classificá-la dentro dos critérios estabelecidos montando-se o Quadro 7.

QUADRO 7 – LISTA INICIAL DE FERRAMENTAS E ADESÃO AOS CRITÉRIOS

Ferramenta	Instalado e Testado	C1	C2	C3	C4	C5	Motivo Eliminação
PMD	SIM	X	X	X	G	X	
FindBugs	SIM	X	X	X	M	X	
CheckStyle	SIM	X	X	X	CS	X	
Sonar	SIM	X	X		G	X	
Analyst4j	NÃO	X	N	N	M	X	C2 – Não open-source
CCCC	NÃO	X	N	X	M	X	C2 – Não atualizado desde 2006
C & K Java Metrics	SIM	X	X	X	P	X	C4 – Apenas coleta a suíte C&K
Dependency Finder	NÃO	X	N	X	M	X	C2 – Não atualizado desde 2007
Eclipse Metrics 1.3.6	NÃO	X	N	N	M	X	C2 – Suporte só para Eclipse 3.1
Eclipse Metrics 3.4	NÃO	X	N	N	M	X	C2 – Não atualizado desde 2009
OOMeter	NÃO	X	N	X	M	X	C2 – Não atualizado desde 2004
Semmle	NÃO	X	N	N	M	X	C2 – Não open-source
Understand for Java	NÃO	X	N	N	M	X	C2 – Não open-source
VizzAnalyzer	NÃO	X	X	X	G	N	C5 – Sem sucesso na instalação e testes
CMTJava	NÃO	X	N	N	M	X	C2 – Não open-source
Resource Standard Metrics	NÃO	X	N	N	M	X	C2 – Não open-source
Code Pro AnalytiX	SIM	X	X	X	G	X	
JDepend	NÃO	X	N	X	P	N	C2 – Não atualizado desde 2009
JHawk	NÃO	X	N	N	P	X	C2 – Não open-source
JMetric	NÃO	X	N	N	P	X	C2 – Não atualizado desde 2001
Krakatau Metrics	NÃO	X	N	N	P	X	C2 – Não open-source
RefactorIT	NÃO	X	N	N	P	X	C2 – Não atualizado desde 2001
McCabe IQ	NÃO	X	N	N	P	X	C2 – Não open-source
Cobertura	SIM	X	X	X	P	N	C5 – Muito difícil interpretar o report
Clover	NÃO	X	N	N	P	X	C2 – Não open-source
CTC++ for Java and Android	NÃO	X	N	N	P	X	C2 – Não open-source
DevPartner	NÃO	X	N	N	P	X	C2 – Não open-source
EMMA	NÃO	X	N	N	M	N	C2 – Não atualizado desde 2005
Jtest	NÃO	X	N	N	P	X	C2 – Não open-source
Kalistick	NÃO	X	N	N	G	X	C2 – Não open-source
LDRA Testbed	NÃO	X	N	N	P	X	C2 – Não open-source
JaCoCo	SIM	X	X	X	P	X	
AgileJ StructureViews	NÃO	X	N	N	M	X	C2 – Não open-source
Hammurapi	NÃO	X	N	N	P	X	C2 – Não atualizado desde 2006
Soot	SIM	X	X	X	P	N	C5 – Difícil de usar
Squale	SIM	X	X	X	G	X	
SonarJ	NÃO	X	N	N	M	X	C2 – Não open-source
Klocwork	NÃO	X	N	N	M	X	C2 – Não open-source
JavaNCSS	NÃO	X	N	X	M	X	C2 – Não atualizado desde 2009

Fonte: Elaboração Própria

Após a eliminação das ferramentas que não atendiam aos critérios estabelecidos restaram 7 softwares a serem analisados. Após a instalação de todos e análise do conjunto de métricas coletadas, facilidade de uso, facilidade de entendimento dos relatórios gerados, o software escolhido para coletar as métricas do estudo e posteriormente integrar o acompanhamento de métricas em projetos reais foi o Sonar. O principal motivo da escolha do sonar foi que ele integra as duas ferramentas mais comentadas e conhecidas na área de métricas, o PMD e o FindBugs, além disso ele possui uma interface web que permite

acompanhar a evolução das métricas de um produto realease por release chegando ao nível de acompanhamento da evolução do código de um método específico. Isso o torna uma excelente ferramenta de groupware para acompanhamento da qualidade de código.

QUADRO 8 – LISTA DE FERRAMENTAS QUE ETENDEM AOS CRITÉRIOS

Ferramenta	Instalado e Testado	C1	C2	C3	C4	C5	Observações
PMD	SIM	X	X	X	G	X	Muito conhecido e usado – Desktop
FindBugs	SIM	X	X	X	M	X	Muito conhecido e usado – Desktop
CheckStyle	SIM	X	X	X	CS	X	Muito conhecido e usado – Desktop
Sonar	**SIM**	**X**	**X**	**X**	**G**	**X**	**Intergra PMD e Find bugs. Groupware**
Code Pro AnalytiX	SIM	X	X	X	G	X	Suportado pelo Google – Desktop
JaCoCo	SIM	X	X	N	P	X	Pequeno conjunto de análises. Sem nenhuma ocorrência acadêmica
Squale	SIM	X	X	N	G	X	Apesar de ser Gruopware. Sem nenhuma ocorrência acadêmica.

Fonte: Elaboração Própria

Após a escolha da ferramenta foi feita uma seleção das métricas implementadas pela ferramenta que tivessem definição acadêmica, ou pelo menos já estabelecida pela área de engenharia de software. Foram selecionadas as métricas listadas no Quadro 9. Todas as métricas selecionadas são implementações ou derivações diretas das métricas discutidas na sessão 2.3.

QUADRO 9 – LISTA DE MÉTRICAS SONAR

Relativas ao tamanho do software	
Linhas Físicas	Número de quebras de linhas nos arquivos de código fonte.
Java Doc	Número de entradas do tipo Java Doc
Linhas comentadas	Número de linhas comentadas excluindo Java Doc
Linhas de Código Comentadas	Número de linhas de código comentadas.
Linhas de Código	Número de linhas de código, incluindo código comentado.
Densidade dos Comentários	Número de Linhas comentadas / (Linhas de Código + Numero de Linhas Comentadas) *100
Pacotes	Número total de pacotes do sistema.
Classes	Número total de Classes do Sistema. Inclui classes internas, enum, interfaces e annotations.
Diretórios	Número total de diretórios analizados.
Acessores	Número de Gets e Sets utilizados para recuperar atributos de classes
Métodos	Número total de métodos do sistema excluindo os acessores. Inclui construtores.
API Pública	Número de classes públicas, métodos públicos e atributos públicos.
API Pública não documentada	Número de classes públicas, métodos públicos e atributos públicos sem Java Doc.
Densidade da Documentação de API Pública	(API Pública - API Pública não documentada) / API Pública * 100
Comandos	Número de Comandos Java.

Relativas a Duplicação

Linhas Duplicadas	Número de linhas de código duplicadas
Blocos Duplicados	Número de blocos de código duplicado
Arquivos com Duplicação	Número de arquivos que possuem duplicação de código (linha ou bloco)
Densidade de Duplicação	Número de Linhas Duplicadas /Número de Linhas Físicas* 100

Relativas ao Design OO

Profundidade da árvore de herança	A profundidade da árvore de herança (PID) fornece para cada classe de uma medida dos níveis de herança da parte superior hierarquia de objetos.
Número de Filhos	Um número da classe Filho (NOC) mede o número de descendentes diretos e indiretos da classe.
Resposta para a classe	A resposta definida de uma classe é um conjunto de métodos que podem potencialmente ser executados em resposta a uma mensagem recebida por um objeto dessa classe.
Acoplamento aferente	Acoplamento aferente Uma classe é uma medida de quantas outras classes usam a classe específica.
Acoplamento eferente	Acoplamentos de eferente Uma classe é uma medida de quantas classes diferentes são usado pela classe específico.
Falta de coesão dos métodos de	LCOM4 mede o número de "componentes conectados" em uma classe. Um componente conectado é um conjunto de métodos e campos relacionados. Deve haver apenas um componente desse tipo em cada classe. Se houver 2 ou mais componentes, a classe deve ser dividida em classes menores.
Ciclos de pacotes	Número mínimo de ciclos de pacotes detectado para ser capaz de identificar todas as dependências não desejadas.
As dependências de pacote	Número de dependências do pacote, a fim de remover todos os ciclos entre os pacotes.
Dependências do arquivo	Número de dependências de arquivo, a fim de remover todos os ciclos entre os pacotes.
Peso de borda do pacote	Número total de dependências de arquivos entre pacotes.
Índice emaranhado pacote	Dá o nível de emaranhamento dos pacotes, o que significa melhor valor 0% que não há ciclos e pior valor significa 100% .
Ciclos de arquivos	Número mínimo de ciclos de arquivo dentro de um pacote para ser capaz de identificar todas as dependências não-desejadas.
Dependências Suspeitas de Arquivo	Dependências do arquivo a serem eliminadas, a fim de remover ciclos entre arquivos dentro de um pacote.
Emaranhado Arquivo	Dá o nível de emaranhamento de um arquivo específico.
Peso de borda do Arquivo	Número total de dependências de arquivo dentro de um pacote.

Relativas a Complexidade

Complexidade Ciclomática	O Numero de Complexidade Ciclomática é também conhecido como Métrica de McCabe Métrica. Seu cálculo é contar o número de fluxos deferentes que um fragmento de código pode tomar em sua execução (comandos if, for, while, etc).
Complexidade média por método	Complexidade ciclomática média por método.
Distribuição de complexidade por métodos	Número de métodos para complexidades dadas.
Complexidade média por classe	Complexidade ciclomática média por classe.
Distribuição de complexidade por classe	Número de classes para complexidades dadas.
Complexidade média por arquivo.	Complexidade ciclomática média por arquivo.

3.4 Overview da Metodologia

As etapas e fases da pesquisa estão resumidas na Figura 9. A seguir descrevemos cada uma.

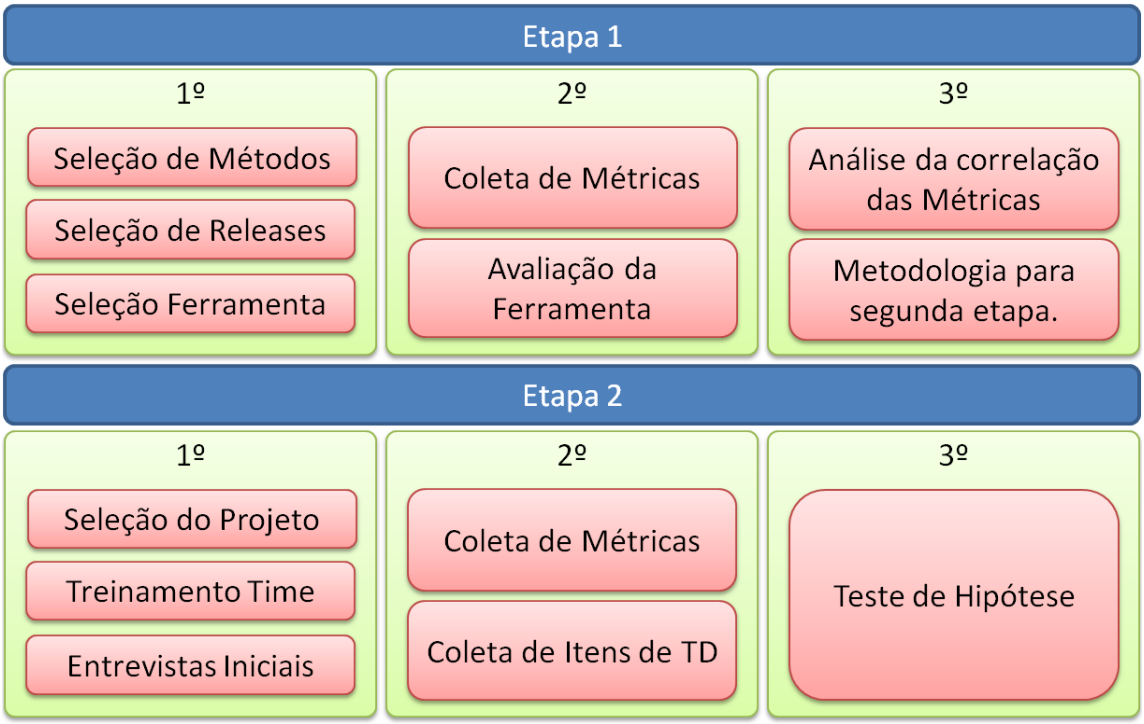


FIGURA 9 – OVERVIEW DA METODOLOGIA

3.4.1 PRIMEIRA ETAPA

Foram coletados os dados de métricas listados no Quadro 9 das releases listadas no quadro Quadro 6 e feito um comparativo com a lista de dívidas técnicas já levantadas pelo estudo de (Oliveira, 2011). O objetivo foi observar padrões na variação de valores das métricas entre as releases que foram introduzidos os itens de TD para se traçar uma correlação entre a variação e a ocorrência dos itens de TD possibilitando assim testar essa correlação em um outro projeto vivo. Após a análise dos dados coletados foi criado um plano para coleta de dados e controle de TD em um projeto vivo. O plano foi executado na segunda etapa do estudo.

3.4.2 SEGUNDA ETAPA

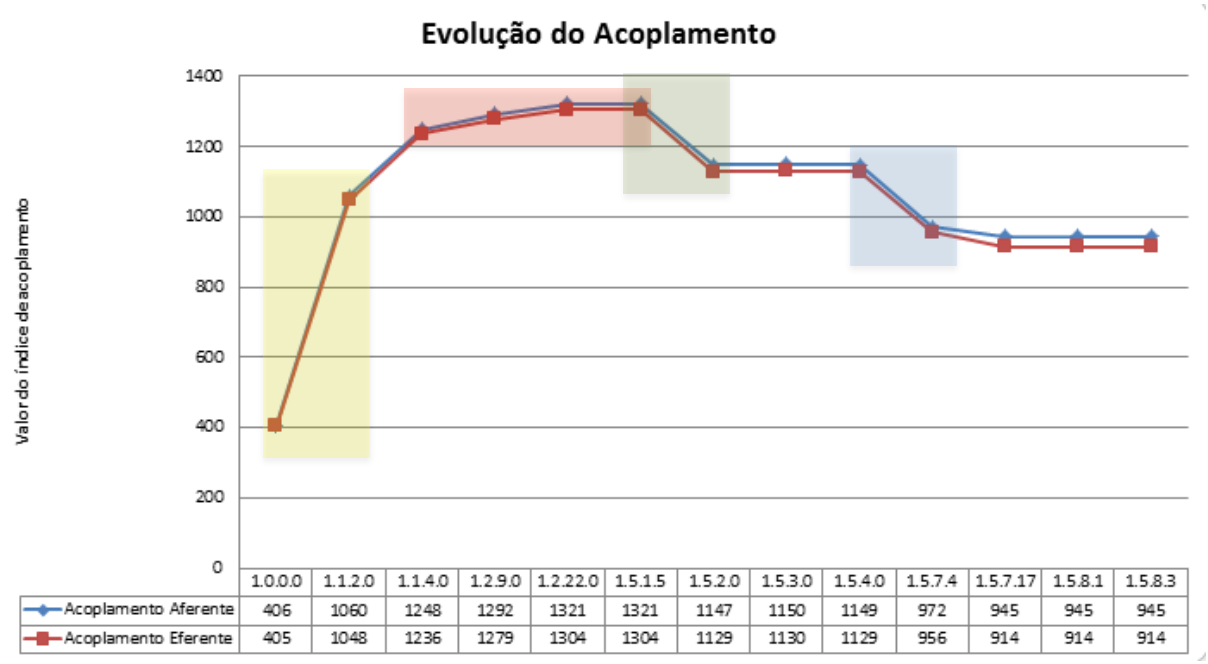
Durante os 2 primeiros meses de acompanhamento os líderes de equipe do projeto eram entrevistados semanalmente para coleta de informações que pudessem subsidiar a descoberta de itens de TD e possibilitar o cruzamento dos dados sobre métricas de software com esses itens. Durante esse período os líderes só tinham conhecimento que estava participando de uma pesquisa sobre Dívida Técnica e possuíam pouca informação do que seria exatamente Dívida Técnica. Após esse período foi executado um workshop com os líderes de equipe treinando-os em todos os aspectos conhecidos até então de TD, bem como nos métodos e no formulário utilizado para coleta e catalogação dos itens de TD apresentado no Quadro 3. Após esse período o time de desenvolvimento passou a ter visibilidade da pesquisa e foi treinado para levantar, cataloga e acompanhar a Dívida Técnica, porém não tinha visibilidade que existia uma avaliação do código fonte e coleta de métricas. O objetivo desse procedimento era cruzar os dados levantados pelo time sobre os itens de TD e os dados de métricas coletadas diretamente do código fonte da aplicação. Na última fase do estudo, últimos dois meses, o time foi notificado sobre a coleta de métricas e foi testado em campo a hipótese de descoberta e acompanhamento de itens de TD através dos dados já coletados.

4. RESULTADOS

Após compilação, arquivamento e análise dos dados coletados foram obtidos resultados esclarecedores sobre a correlação entre Dívida Técnica e métricas de software. A apresentação dos resultados será feita por etapa do estudo.

4.1 PRIMEIRA ETAPA

Na primeira etapa buscávamos o relacionamento entre o acoplamento do componente de comunicação com a camada de persistência da aplicação em questão. Logo foram selecionados as métricas Acoplamento Aferente e Acoplamento Eferente para avaliação de comportamento.



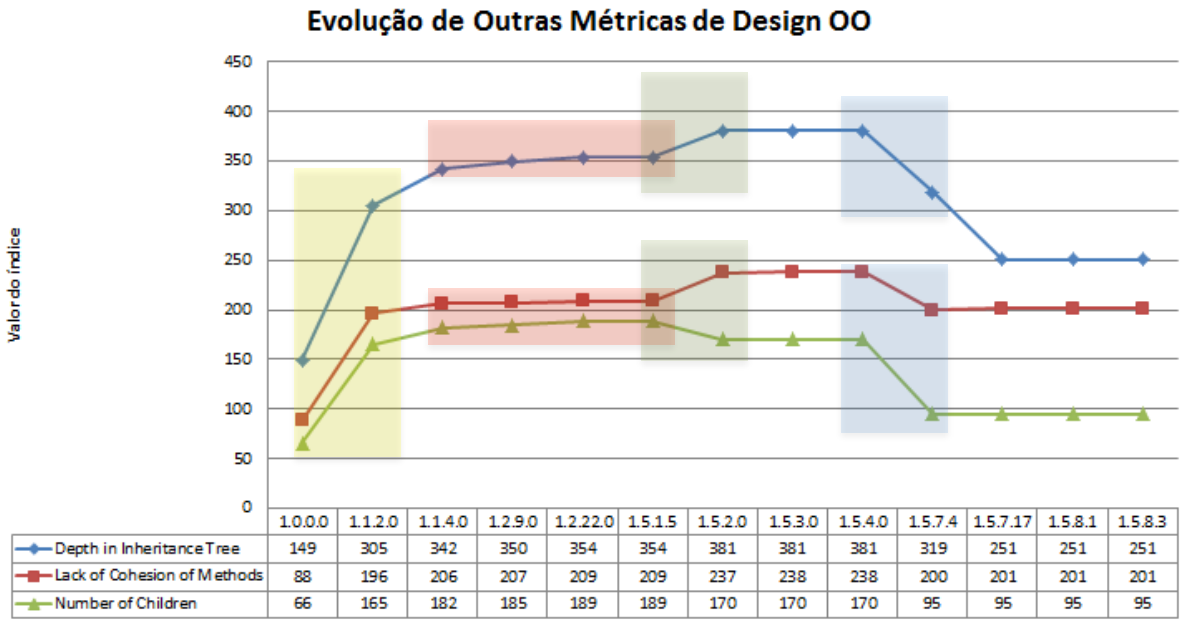
	1.0.0.0	1.1.2.0	1.1.4.0	1.2.9.0	1.2.22.0	1.5.1.5	1.5.2.0	1.5.3.0	1.5.4.0	1.5.7.4	1.5.7.17	1.5.8.1	1.5.8.3
Acoplamento Aferente	406	1060	1248	1292	1321	1321	1147	1150	1149	972	945	945	945
Acoplamento Eferente	405	1048	1236	1279	1304	1304	1129	1130	1129	956	914	914	914

FIGURA 10 - EVOLUÇÃO DO ACOPLAMENTO

Observando-se o gráfico da Figura 10 podemos notar um aumento de aproximadamente 260% do índice de acoplamento entre as releases 1.0.0.0 e 1.1.2.0 (caixa amarela). Esse aumento está diretamente relacionado com a nova implementação do protocolo WebDav e a decisão do acoplamento entre o componente de comunicação e o componente de persistência, ou seja, o TD previamente catalogado. Ainda, existe entre as releases 1.1.2.0 e 1.5.1.5 um aumento crescente no índice de acoplamento chegando a 10% como juros acumulados ao valor principal da dívida por causa das manutenções e evoluções nesses dois componentes (caixa vermelha). Entre as releases 1.5.1.5 e 1.5.2.0 houve um refatoramento no componente de persistência motivado pelo baixo desempenho do mesmo e problemas do sistema de armazenamento utilizado. Durante esse refatoramento houve um esforço significativo de tentar diminuir o acoplamento entre os dois componentes a partir de padrões de projetos aplicados. Com isso observa-se uma diminuição dos índices de

acoplamento em 14% (caixa verde). Entre as releases 1.5.4.0 e 1.5.7.4 o componente de persistência foi completamente reimplementado, porém ainda foi mantido o acoplamento entre a comunicação e a persistência por questões de desempenho. Observando-se o resultado dessa completa reimplementação podemos perceber que ela unicamente amortizou o juros acumulado pelo tempo, já que os índices de acoplamento voltaram a valores próximos do patamar de quando a dívida foi adiquirida (caixa azul). Isolando-se o esforço necessário para se realizar todas as implementações de refactor e reimplementação do componente temos que a amortização da dívida custou no total 813 horas de trabalho o que representa 71% do esforço de modificação total (Oliveira, 2011). Isso claramente evidencia que se o índice de acoplamento estivesse sendo acompanhado durante os lançamentos das releases, na primeira acumulação do juros, acontecido entre as releases 1.1.2.0 e 1.1.4.0, a gerência do projeto poderia ter iniciado um ciclo de refactor para saldar a dívida. Ainda, o valor principal da dívida poderia ter sido evitado, ou pelo menos menor, caso o valor do índice de acoplamento estivesse sendo gerenciado antes do lançamento da release 1.1.2.0.

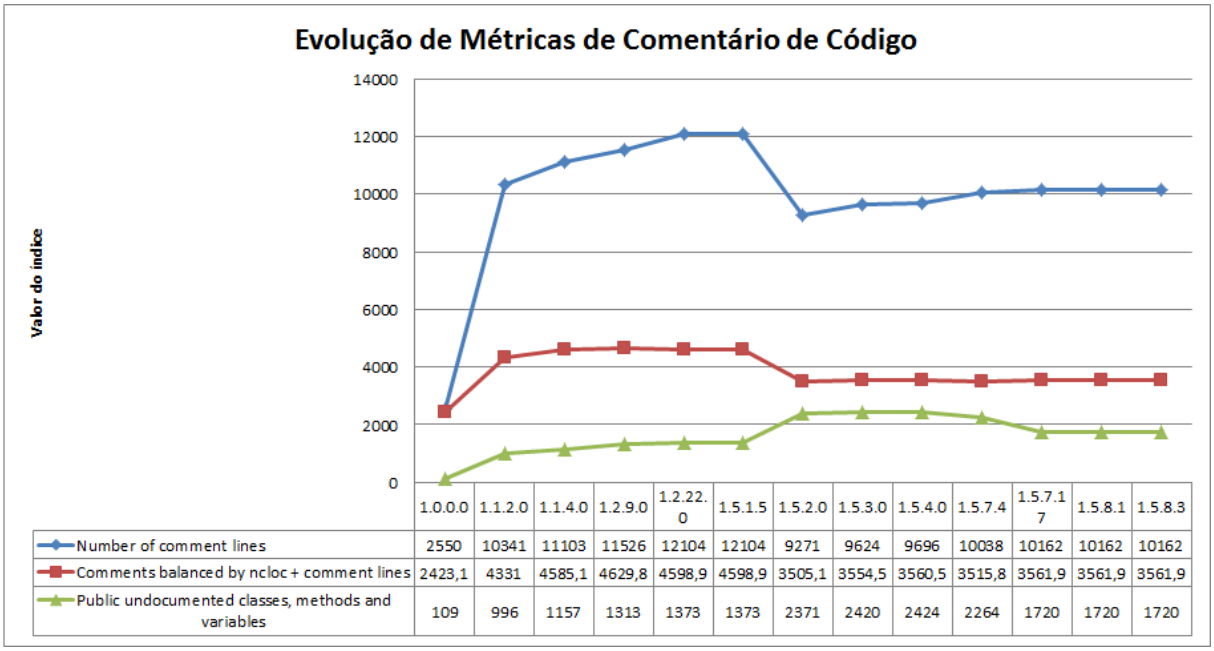
Além dos índices de acoplamento, diretamente relacionados ao TD catalogado por Oliveira (2011), foram observados que as métricas relacionadas o design OO acompanham a mesma tendência da curva dos índices de acoplamento, Figura 11, mostrando claramente a correlação dos achados para acoplamento com os índices de design OO.



	1.0.0.0	1.1.2.0	1.1.4.0	1.2.9.0	1.2.22.0	1.5.1.5	1.5.2.0	1.5.3.0	1.5.4.0	1.5.7.4	1.5.7.17	1.5.8.1	1.5.8.3
Depth in Inheritance Tree	149	305	342	350	354	354	381	381	381	319	251	251	251
Lack of Cohesion of Methods	88	196	206	207	209	209	237	238	238	200	201	201	201
Number of Children	66	165	182	185	189	189	170	170	170	95	95	95	95

FIGURA 11 - EVOLUÇÃO DAS MÉTRICAS DE DESIGN OO

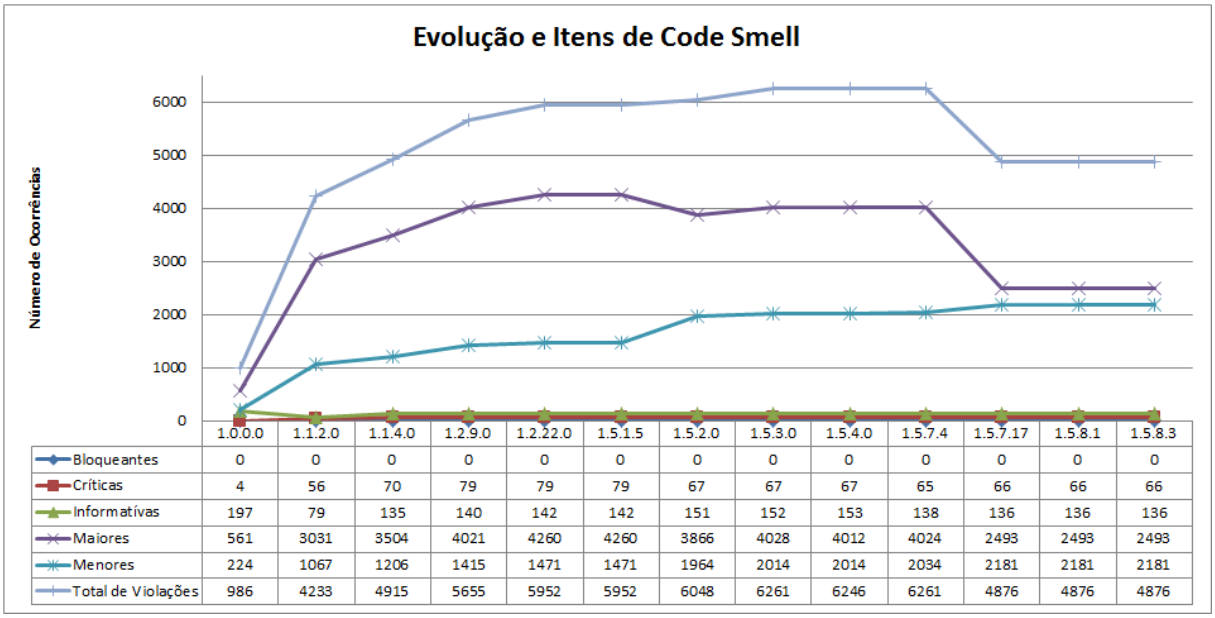
Observando-se as métricas relacionadas com a documentação de código evidencia-se que no processo de refatoramento, muitos dos comentários de código fonte foram perdidos Figura 12.



	1.0.0.0	1.1.2.0	1.1.4.0	1.2.9.0	1.2.22.0	1.5.1.5	1.5.2.0	1.5.3.0	1.5.4.0	1.5.7.4	1.5.7.17	1.5.8.1	1.5.8.3
Number of comment lines	2550	10341	11103	11526	12104	12104	9271	9624	9696	10038	10162	10162	10162
Comments balanced by ncloc + comment lines	2423,1	4331	4585,1	4629,8	4598,9	4598,9	3505,1	3554,5	3560,5	3515,8	3561,9	3561,9	3561,9
Public undocumented classes, methods and variables	109	996	1157	1313	1373	1373	2371	2420	2424	2264	1720	1720	1720

FIGURA 12 - EVOLUÇÃO DE MÉTRICAS DE COMENTÁRIOS DE CÓDIGO

Ainda, as ocorrências de Code Smell seguem a mesma tendência das curvas de acoplamento. Esse é um fato de extrema importância dado que existem dezenas de categorias de Code Smell dentro de cada classificação de criticidade, ou seja, corrobora com a hipótese de que as violações de Code Smell esteja diretamente relacionadas aos itens de TD.



	1.0.0.0	1.1.2.0	1.1.4.0	1.2.9.0	1.2.22.0	1.5.1.5	1.5.2.0	1.5.3.0	1.5.4.0	1.5.7.4	1.5.7.17	1.5.8.1	1.5.8.3
Bloqueantes	0	0	0	0	0	0	0	0	0	0	0	0	0
Críticas	4	56	70	79	79	79	67	67	67	65	66	66	66
Informativas	197	79	135	140	142	142	151	152	153	138	136	136	136
Maiores	561	3031	3504	4021	4260	4260	3866	4028	4012	4024	2493	2493	2493
Menores	224	1067	1206	1415	1471	1471	1964	2014	2014	2034	2181	2181	2181
Total de Violações	986	4233	4915	5655	5952	5952	6048	6261	6246	6261	4876	4876	4876

FIGURA 13 - EVOLUÇÃO DE ITENS DE CODE SMELL

4.2 SEGUNDA ETAPA

Na primeira etapa buscávamos o relacionamento entre o acoplamento do componente de comunicação com a camada de persistência da aplicação em questão. Logo foram selecionados as métricas de Acoplamento Aferente e Acoplamento Eferente para avaliação de comportamento. A lista inicial de TD foi levantada durante 2 meses com o time de desenvolvimento gerando um conjunto de itens iniciais. No decorrer do projeto a lista foi atualizada com dados decorrentes da evolução do software. A lista final de itens de TD pode ser vista no quadro Quadro 10. Os itens mais claros da lista não foram avaliados sob a ótica de métricas. Apenas os casos diretamente relacionados com o código fonte foram acompanhados.

QUADRO 10 - LISTA DE ITENS DE TD (PROJETO ETAPA 2)

ID	Tipo	Localização	Descrição
1	Teste	N/A	Revisão de Casos recebidos de fontes externas e verificação de adequada / atualização para as condições atuais de sistema e aplicação nas estratégias de cada um.
2	Doc.	N/A	Analisar cenários de teste por projeto, criar casos de testes e adiciona-los no Testlink.
3	Doc.	N/A	Para rastrear os riscos do sistema e melhor planejar os testes, é necessário que sejam criadas tags para maior rastreio dos problemas. Entretanto, não existe tal folksonomia e é preciso inseri-las em crs antigas para traçar requisitos para próximas sprints.
4	Doc.	N/A	Falta de documentação consolidada sobre sprints passadas, onde se alegava que não se documentava corretamente por se usar SCRUM no projeto deixa falhas em diversas pontas do projeto por não dar a devida visibilidade as funcionalidades
5	Projeto	Adapters e Fragments do projeto	Padronização de uso e implementação. Como se deixou cada um fazer de uma forma, então, tem se um custo mais alto atrelado a legibilidade do código.
6	Projeto	Ui.messages	Refatorar a parte de mensagens pensando em melhorar a legibilidade do código.
7	Projeto	Controller de submódulo	Tentar melhorar o uso da API para facilitar o desenvolvimento
8	Projeto	***ThemedActivity, ***.ui.MainSettingsThemedFragment	O código que faz a alteração de fontes e cores do <Aplicação> poderia ser refatorado para facilitar manutenção. A solução atual aparenta estar mais complexa do que poderia ser.
9	Projeto	***.util.FileLoader	Analisar outra solução para download de Arquivos na arquitetura atual do <Aplicação>.
10	Projeto	Todos os Adapters dos Listviews utilizados no projeto.	Poderíamos utilizar uma classe pai genérica para os adapters. No projeto atual existem várias classes com comportamentos semelhantes.

11	Projeto	DownloadController.java, *Fragment*.java, <Application>Activity.java	É necessário melhorar tanto a legibilidade do código quanto a eficácia do sistema de notificações. Apresenta falhas para alguns casos.
12	Doc.	N/A	Necessidade de diagramas de fluxos para todas as funcionalidades principais do sistema para auxílio em desenvolvimento e testes.
13	Doc.	Documento de requisitos e wireframes	A documentação não foi bem especificada desde o início do projeto. Alguns pontos só foram definidos ou foram modificados durante a fase de testes.
14	Projeto	Código que se comunica com o servidor.	Não houve uma definição da arquitetura final da aplicação. Atualmente, a arquitetura foi toda modificada e estamos modificando todo o código para atendê-la.
15	Doc.	Documento de arquitetura	Melhor documentação da arquitetura interna do <Aplicação>.
16	Teste	Projeto <Aplication>Test	Criação de testes unitários automáticos mais abrangentes.
17	Projeto	Classes: LMSController, AMSController, AuthController	Revisar e simplificar o código dos controllers que interagem com os conectores. Eles estão com uma lógica muito confusa e não está fácil de manter o código deles.
18	Projeto	PendingMessagesController	Revisar a lógica que está sendo usada no controlador de notificações.
19	Projeto	Widget, WidgetsController	Refatorar a criação e controle dos widgets.
20	<< ?>>	Todas as telas do projeto	A interface do projeto é feia e mal projetada. Sérios problemas de usabilidade.
21	Defeito	Notificações	Funcionalidade implementada sem uma definição de fluxo na Sprint 3. Fluxo criado apenas na Sprint 8. “Remendo de código”
22	<< ?>>	Troca de cor e tipo de fonte	Funcionalidade desnecessária para a aplicação, deixa a aplicação diferente do layout criado. Nunca vi outra aplicação ter tal funcionalidade.
23	<< ?>>	Definição do produto	Ainda não se sabe realmente pra quem o projeto está sendo feito. Com isso, todos os outros problemas aconteceram.
24	<< ?>>	Mudança de imagem de background	Funcionalidade desnecessária para a aplicação.
25	Doc.		Falta de especificação para o editor de Texto
26	Doc.		Falta de definição de workflows
27	Projeto	Na parte de Conectores	Falta de visibilidade nos próximos conectores
28	Projeto	<ApplicationMain>Controller.java	Foi necessário fazer um refactoring das classes de controller do projeto.
29	Projeto	<ApplicationMain>Controller.java	Remoção do padrão Singleton no controlador.

30	Projeto	FileLoader.java	Refatoramento do gerenciador de downloads

Durante o acompanhamento e evolução da lista foram coletados dados, release por release, de métricas para análise de correlação com os itens da lista. Logo nas primeiras coletas ficou claro quais métricas estava relacionadas com cada item de TD e a tendência de correlação se mostrou consistentes no decorrer do estudo. No Quadro 11 tem-se a correlação de métricas e itens de TD do Quadro 10.

QUADRO 11 - RELAÇÃO ENTRE TD E MÉTRICAS (PROJETO ETAPA 2)

Classes	Observações	ID TD
Controladores	Temos uma grande concentração de violações nos controladores da aplicação. Alguns deles são God Classes.	7, 11, 17, 18 e 19
Classes para gerenciamento de download de arquivos	Problemas com Falta de coesão. Comentários de código praticamente inexistentes. Alta complexidade ciclomática.	9 e 14
Adaptadores Dados/Interface	Abuso de duplicação de código	5, 10
Activity principal da aplicação	Acoplamentos desnecessários. Complexidade ciclomática.	8 e 11
Classes de Fragments	Complexidade ciclomática. Pouco Comentário. Muitas linhas de código.	5, 8, 11

Na Figura 14 evidencia 4 pontos importantes de refatoramento que incluíram com as melhorias um aumento significativo de acoplamento. As modificações são entre as releases 2.1.7.1 e 2.1.9.1, 2.2.19.2 e 2.3.2.1, 2.4.1.1 e 2.4.3.1 e finalmente 2.4.5.1 e 2.5.1.1.

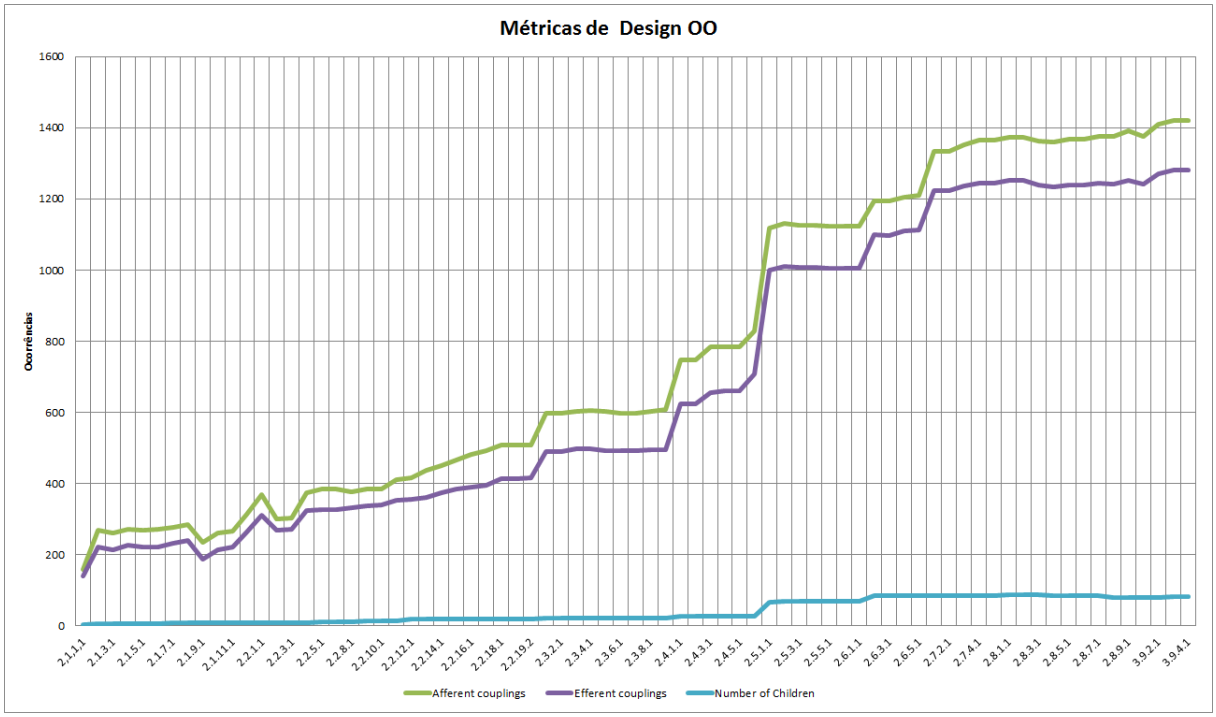


FIGURA 14 - DESIGN OO ETAPA 2

A tendência da curva de acoplamento evidência o TD agregado aos refatoramentos e o pagamento de itens de TD agregando novas dívidas. Na Figura 15 observa-se uma tendência semelhante nas curvas das métricas de tamanho OO.

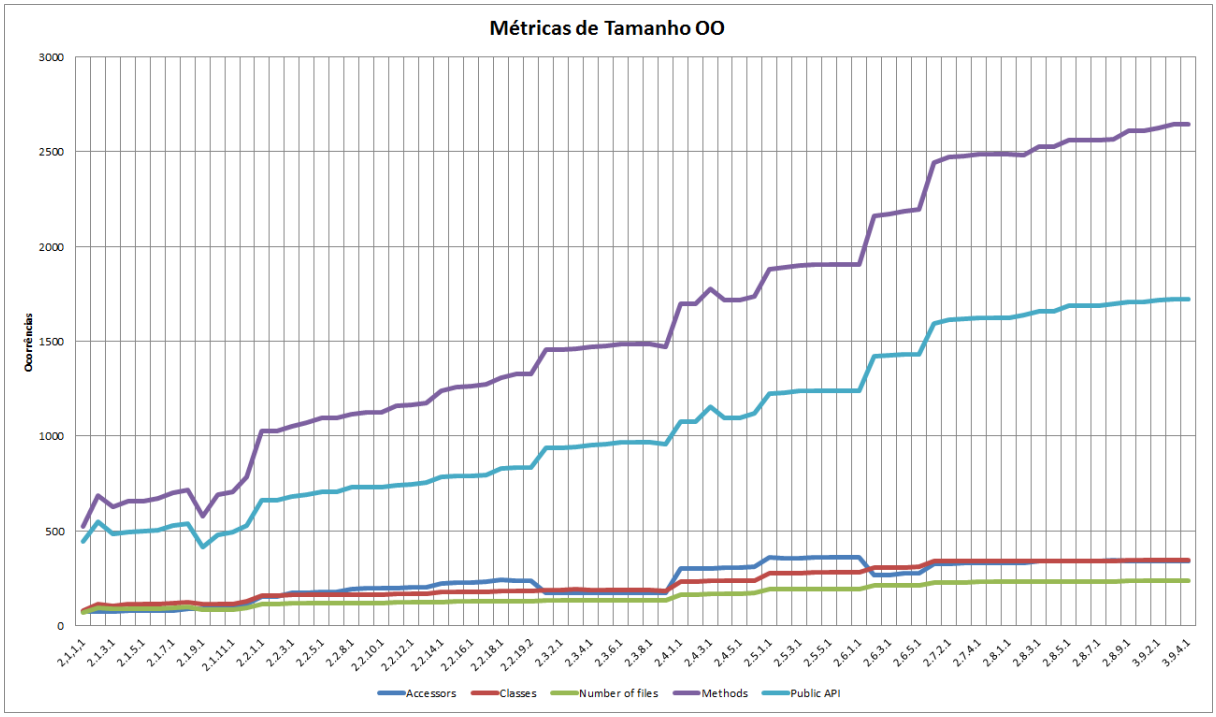


FIGURA 15 - MÉTRICAS DE TAMANHO ETAPA 2

Outro comportamento que foi legado e permaneceu consistente no decorrer do projeto foi a falta de documentação em forma de comentários de código. Apesar de não ser

um item de TD levantado pelo time de projeto, esse é obviamente relacionado aos itens de documentação 2, 3, 4, 12, 13, 15, 21, 25 e 26 da lista de TD no Quadro 10. Neste ponto cabe discutir o achado sobre a dificuldade de compreensão do código. O time justificou a maioria dos refatoramentos ocorridos como necessários para melhorar a legibilidade do código, porém sem nenhuma métrica de qualidade objetiva para subsidiar essa interpretação e decisão. Percebe-se que o mesmo ponto da aplicação sobre refatoramentos recorrentes e mudanças de código com a mesma localização evidenciando um trabalho de refatoramento não efetivo. Vemos na Figura 16 que os refatoramentos sucessivos não melhoraram a documentação do código, o que pode corroborar para o agravamento dos itens de TD de documentação do quadro Quadro 10. Ainda outro achado relevante foi que no primeiro processo de refactor feito, entre as releases 2.1.7.1 e 2.1.9.1 foi removida uma quantidade significativa de comentários. O time justificou como remoção código morto (comentado), porém analisando os logs de mudanças dessas releases observa-se que muito dos comentários retirados foram em decorrência das mudanças feitas pelo refatoramento e o código substituído não foi novamente documentado. Observa-se ainda que a API pública do projeto acompanha a curva de crescimento do código, porém o número de linhas documentadas fica relativamente o mesmo até a última release evidenciando a não documentação das novas APIs criadas.

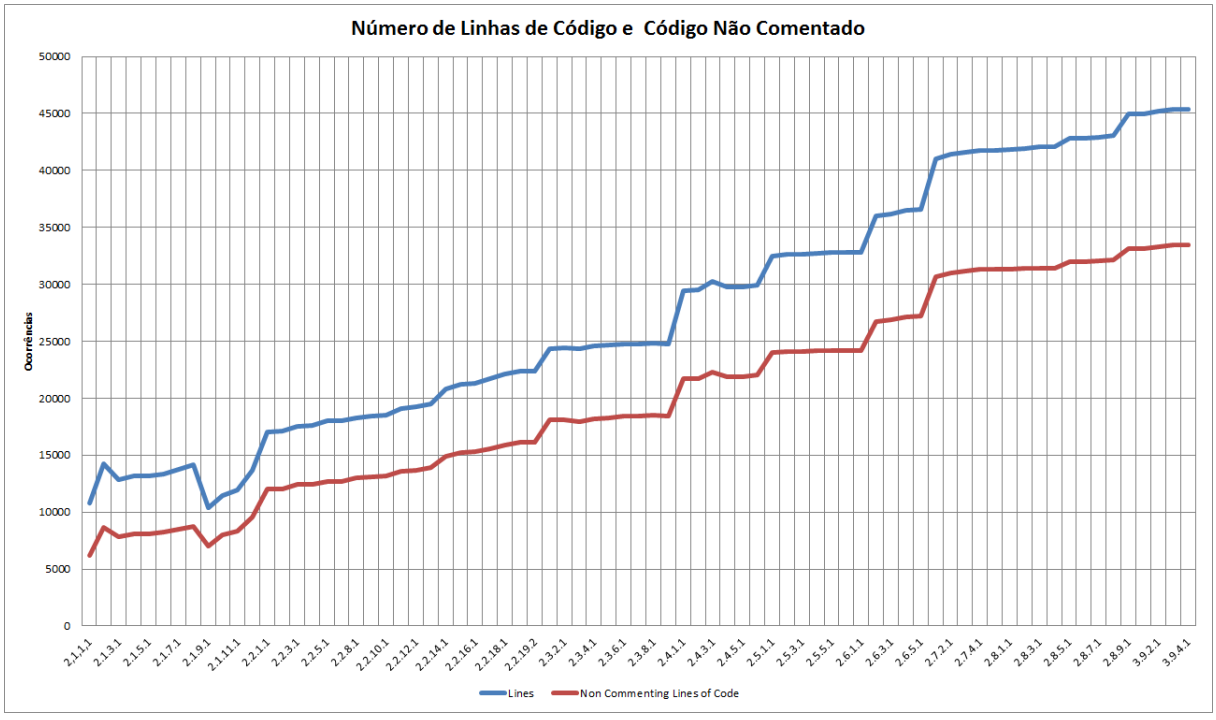


FIGURA 16 - NÚMERO DE LINHAS DE CÓDIGO POR CÓDIGO NÃO COMENTADO

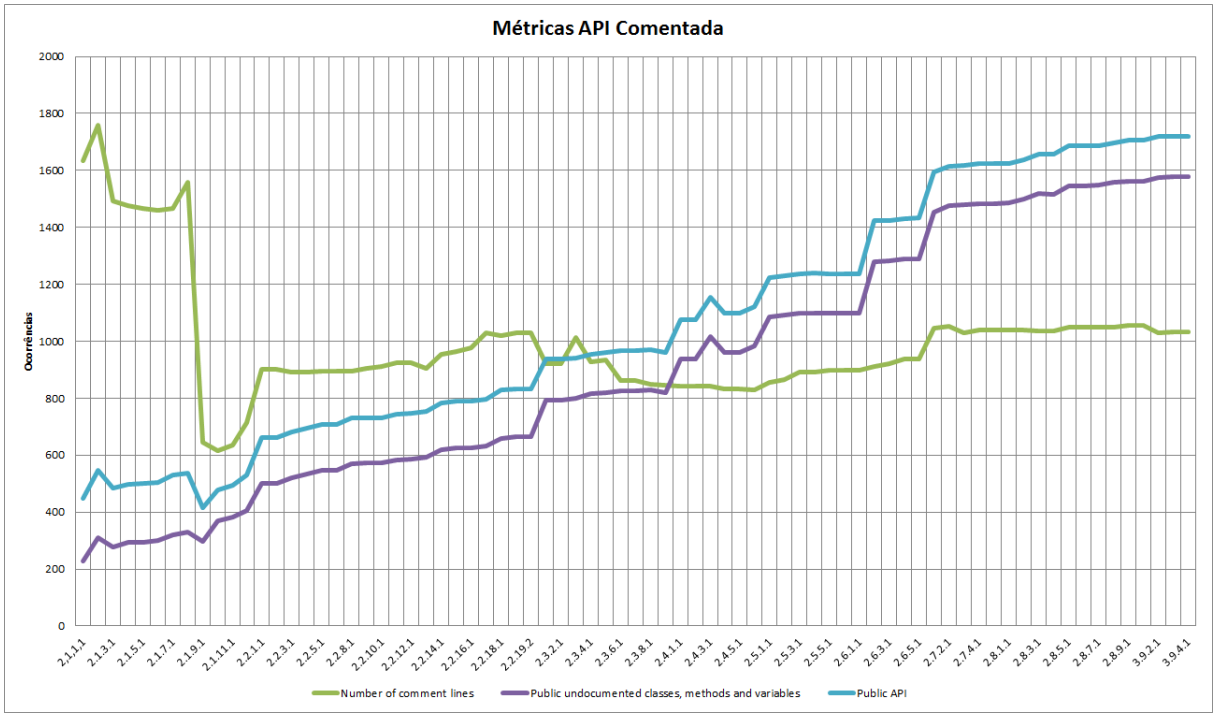


FIGURA 17 - API COMENTADA

Quando tratamos do acompanhamento das curvas de Code Smell , percebemos facilmente os pontos de refactor com a eliminação de ocorrência de itens.

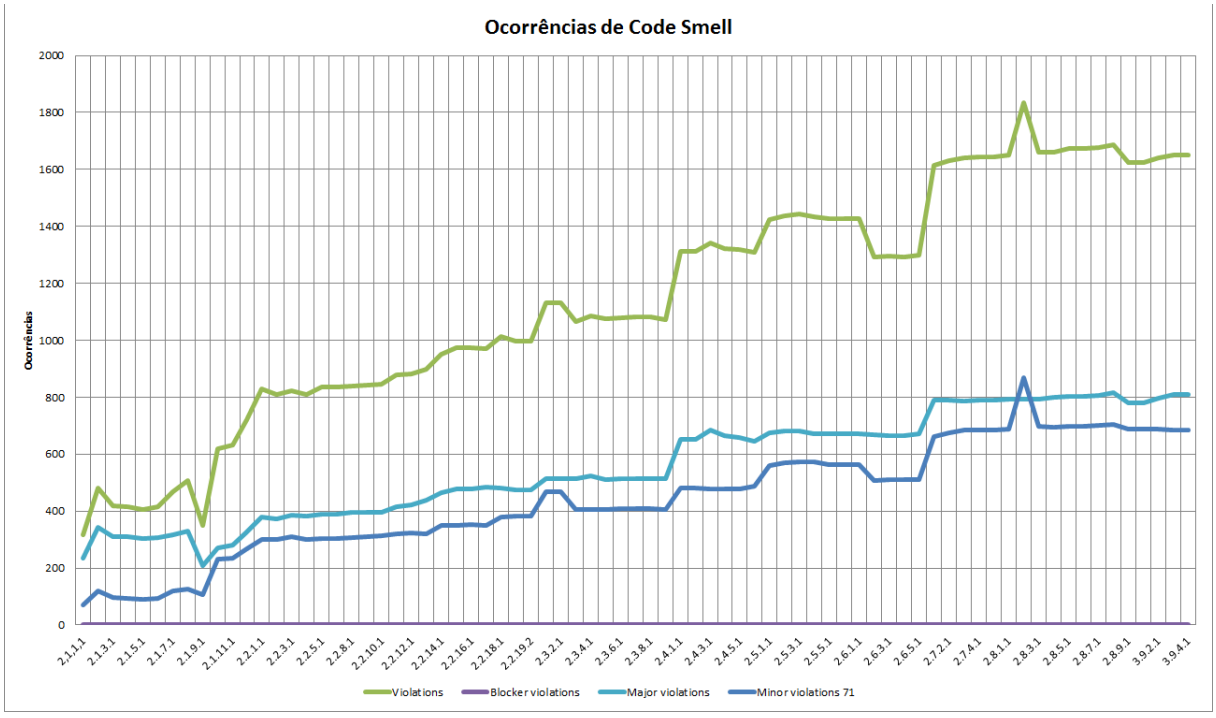


FIGURA 18 - OCORRÊNCIA DE CODE SMELL

Apesar da diminuição de itens de Code Smell no processo de refatoramento, não houve nem mesmo estabilização do número de violações. Isso é claramente em decorrência da não utilização de métricas objetivas para avaliação do que precisa ser refatorado, ainda, acompanhar a evolução do software a partir dessas métricas.

4.3 RESULTADOS INESPERADOS

Com a variação das métricas estudadas principalmente no que diz respeito as de design OO era plausível de se esperar que a complexidade ciclomática variasse de forma diretamente proporcional em consonância com a variação das violações de design OO. Porém o resultado obtido foi que a complexidade ciclomática ficou relativamente estável durante todo ciclo de vida do software como pode ser visto na **Figura 19**. Para caracterizar completamente esse comportamento é necessário um estudo mais granular dos dados sobre design OO e complexidade ciclomática, ainda, da complexidade ciclomática isolada por componente e por classe para entender quais os fatores de estabilização que foge do escopo desse trabalho.

Uma hipótese plausível é que a maior parte da complexidade ciclomática está concentrada em classes que sofreram pouca ou nenhuma intervenção evolutiva suavizando assim a curva de variação do índice, porém, como já foi dito, é preciso um estudo mais aprofundado para dar suporte a essa hipótese.

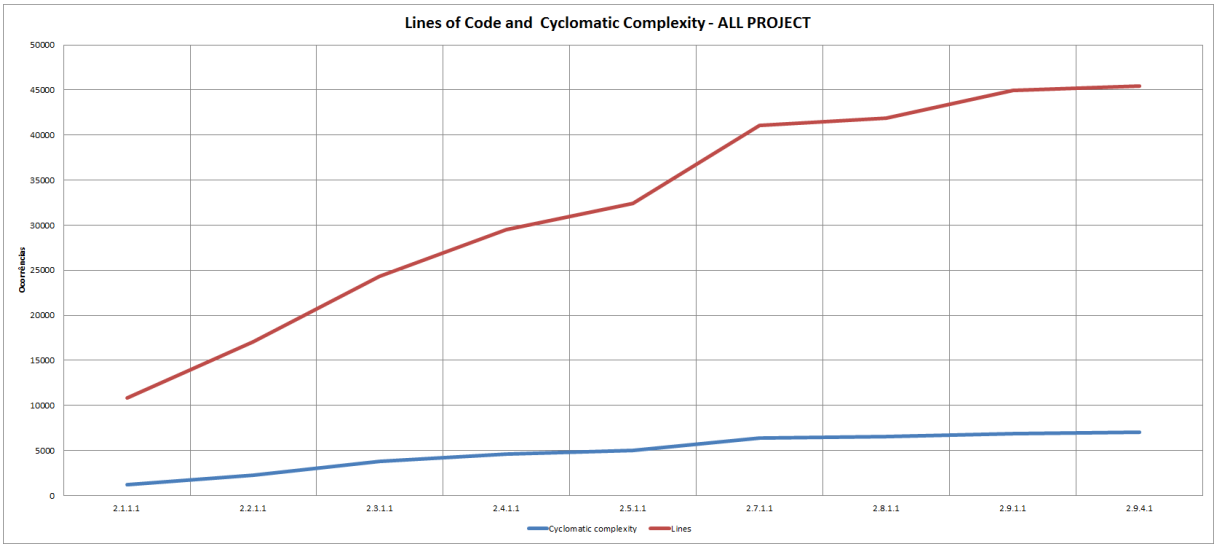


FIGURA 19 – COMPLEXIDADE CICLOMÁTICA

5. CONCLUSÃO

Esse trabalho explorou diversas facetas do conceito concebido para Dívida Técnica, tanto acadêmicas quanto tácitas no universo do desenvolvimento de software. A Dívida Técnica já não é apenas uma hipótese fundamentada em evidências anedóticas, porém ainda é preciso um trabalho intenso de coleta e documentação de dados empíricos para melhor conhecermos o fenômeno e para podermos definir práticas para controle do mesmo. Em Guo (2009), Guo e Seaman (2011) e Guo (2011), foram estudadas e documentadas várias características da Dívida Técnica com coleta de dados empíricos para fundamentação das mesmas. O trabalho de Oliveira (2011) usa-se dos dados e conceitos desses trabalhos e os estende com o estudo de outros projetos reais para uma definição formal e detalhada das características da Dívida Técnica bem como suas ocorrências fenomenológicas. Ainda Oliveria (2011) mostra que a gestão do projeto e a tomada de decisão em relação as intervenções para evolução do software tem relação direta com a Dívida Técnica do produto e que é exatamente nessa tomada de decisão que a gestão da Dívida Técnica deve acontecer. Com isso temos a fundamentação necessária do constructor da Dívida Técnica para desenvolvimento de trabalhos que estudam o fenômeno e suas apresentações nos diversos tipos de projeto de software e suas diversas formas de gerenciamento.

Nesse trabalho utilizamos do constructor embasado por Oliveira (2011) para tratar uma das dificuldades levantadas no seu estudo, a identificação da ocorrência de uma Dívida Técnica. Esses mesmas dificuldades são relatadas em Guo (2009), Guo e Seaman (2011) e Guo (2011) que propõem um formulário de coleta para guiar essa identificação. A identificação do incorrência em Dívida Técnica durante o gerenciamento das ações evolutivas é extremamente custoso (Oliveira, 2011). Necessita de avaliação especialista do impacto da evolução em cada composição do software. Dessa forma, assim como já fundamentado na área de qualidade de software, é interessante que possamos identificar e mensurar essa incorrência de forma automática pela análise dos artefatos de software. O que esse trabalho apresentou foi a utilização de métricas de qualidade de Software para gestão da Dívida Técnica. Foram selecionadas via estudo bibliográfico as as métricas clássicas e consolidadas para qualidade de código fonte de software. Avaliou-se o comportamento dessas métricas em um projeto de Software real durante seu completo ciclo de vida, desde sua primeira release em 2005, até sua descontinuidade em 2011. Durante seu ciclo de vida, esse software apresenta todas as intervenções evolutivas documentadas (ISO/IEC, 2006; Lehman M. M., Programs, life cycles, and laws of software evolution., 1980; Pigoski, 1996; Sommerville, 2007; Mens & Demeyer, 2008) se mostrando assim um exemplo completo para estudo do comportamento das métricas de software e seu relacionamento com o fenômeno da Dívida Técnica.

5.1 RESUMO DOS RESULTADOS

O estudo da primeira etapa deixa claro e documentado como a Dívida Técnica se relaciona com as métricas de acoplamento, design OO, comentários de código e Code Smell. Os dados apresentados mostram claramente que as métricas podem ser utilizadas para identificação e documentação dos pontos de incorrência em Dívida técnica. Isso corrobora com a hipótese desse trabalho que é a possibilidade de utilização de métricas de software para diminuição do esforço de catalogação e gerenciamento da Dívida Técnica em produtos de software.

Na segunda etapa do trabalho os itens de Dívida Técnica levantados da forma proposta por Oliveira (2011), que exige um esforço de especialista considerável, foi comparada com a curva de evolução das métricas para cada uma das 47 releases lançadas. Ficou evidente nessa comparação que os itens de Dívida Técnica poderiam não só terem sido identificados, mas também mensurados, a partir das métricas analisadas.

Com isso posto, fica comprovado que o uso de métricas de software para gestão da dívida técnica em produtos de software é factível e pode ser usado de duas formas principais: Identificação de itens de dívida técnica pelo acompanhamento das variações das métricas de software e mensuração dessa dívida através da análise dos índices de cada métrica isolada.

5.2 TRABALHOS FUTUROS

Para uma generalização do resultado para projetos de software de qualquer natureza é necessário a condução de estudos é a análises sobre a mesma ótica de mais projetos vivos.

Ainda, ficou evidente que existe uma necessidade de estudarmos o fenômeno de TD sobre uma ótica de violações de código de forma mais profunda. As violações de código são baseadas em construções subjetivas de qualidade o que pode agregar valor a análise de métricas dado o nível de abstração de TD.

Por fim, um estudo detalhado sobre a variação da complexidade ciclomática nesse mesmo estudo de caso, bem como em projetos controle, para determinar o motivo da estabilidade da complexidade ciclomática durante todo o ciclo de vida do caso em questão.

5.3 CONSIDERAÇÕES FINAIS

O trabalho realizado atingiu todos os seus objetivos, e deixa como principais contribuições o relacionamento entre itens de dívida técnica e métricas de software que podem ser utilizados para gerir a dívida técnica durante todo o ciclo de vida de produtos de software. Ainda quais métricas são mais relevantes para cada tipo de dívida técnica, como interpretar os valores das métricas e suas influências nos itens de da dívida e como o refatoramento funciona como uma amortização da dívida quando ele utiliza-se de dados objetivos (diminuição de um tipo específico de violação) e como é menos eficiente, sob a ótica da dívida técnica, quando é feito de forma subjetiva.

REFERÊNCIAS

Abreu, F. & Carapuça, R., 1994. Object-Oriented Software Engineering: Measuring and Controlling the Development Process. *Proceedings of the 4th International Conference on Software Quality.*

Barry Boehm, B. C. E. H. R. M. R. S. C. W., 1995. Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. *Annals of Software Engineering, Vol. 1*, pp. 57-94.

Basili, V. B. L. & Melo, W., 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, Vol. 22(Issue 10), pp. 751-761.

Bellin, D., Tyagi, M. & Tyler, M., 1994. Object-oriented metrics: an overview. *In Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research (CASCON '94)*.

Bob, U., 2009. *A Mess is not a Technical Debt.* [Online] Available at: http://blog.objectmentor.com/articles/2009/09/22/a-mess-is-not-a-technical-debt [Acesso em 17 Dezembro 2010].

BOB, U., 2009. *A Mess is not a Technical Debt.* [Online] Available at: http://blog.objectmentor.com/articles/2009/09/22/a-mess-is-not-a-technical-debt [Acesso em 17 Dezembro 2010].

Brooks, J. F. P., 1975. *The Mythical Man-Month: Essays on Software Engineering..* Boston, MA, USA: Addison-Wesley .

BROWN, N. et al., 2010. *Managing technical debt in software-reliant systems.* New Mexico, s.n., pp. 47-52.

BROWN, N. et al., 2010. *Managing technical debt in software-reliant systems.* New Mexico, s.n., pp. 47-52.

Chidamber, S. a. K. C., 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, Jun, Vol. 20(Issue 6), pp. 476-493.

Cunningham, W., 1992. *The WyCash Portfolio Management System.* Vancouver, s.n.

CUNNINGHAM, W., 1992. *The WyCash Portfolio Management System.* Vancouver, s.n.

Easterbrook, S. a. S. J. a. S. M.-A. a. D. D., 2008. Selecting Empirical Methods for Software Engineering Research. In:: F. a. S. J. a. S. D. Shull, ed. *Guide to Advanced Empirical Software Engineering.* London: Springer London, pp. 285-311.

FLYVBJERG, B., 2006. Five Misunderstandings. *Qualitative Inquiry*, 12(2), pp. 219-245.

Flyvbjerg, B., 2006. Five Misunderstandings About Case-Study Research. *Qualitative Inquiry*, April, vol. 12(no. 2), pp. 219-245.

FOWLER, M., 2009. *TechnicalDebtQuadrant.* [Online] Available at: http://martinfowler.com/bliki/TechnicalDebtQuadrant.html [Acesso em 18 Agosto 2010].

Gamma, E. H. R. J. R. V. J., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l.: Addison Wesley.

Gomes, R. et al., 2011. An Extraction Method to Collect Data on Defects and Effort Evolution in a Constantly Modified System. *In: International Workshop on Managing Technical Debt (MTD).*

Grubb, P. a. T. A., 2003. *Software Maintenance: Concepts and Practice.* 2nd Edition ed. s.l.:World Scientific.

GUO, Y., 2009. *Measuring and Monitoring Technical Debt.* s.l., s.n., pp. 25-46.

GUO, Y. & SEAMAN, C., 2011. *A portfolio approach to technical debt management.* s.l., s.n.

Guo, Y. et al., 2011. Tracking technical debt An exploratory case study. *In: IEEE International Conference on Software Maintenance - Early Research Achievements Track, IEEE International Conference on Software Maintenance.*

HAINAUT, J., CLEVE, A., HENRARD, J. & HICK, J., 2008. Systems, Migration of Legacy Information. In:: *Software evolution.* s.l.:s.n., pp. 105-138.

Halstead, M. H., 1972. Natural laws controlling algoritmic structure?. *ACM SIGPLAN Notices*, Volume v. 7, p. 19.

HIRA, A., 2010. *CodeCount.* [Online] Available at: http://sunset.usc.edu/research/CODECOUNT/ [Acesso em 10 06 2011].

Hyatt, L. H. R. a. L. E., 1995. Software Quality Metrics for object-Oriented Environments. *A NASA Technical Report*, Abril.

IEEE/EIA, 1996. *Standard for software life cycle processes. - IEEE/EIA 12207.0:1996.* s.l.:IEEE Computer Society.

IEEE, 1998. *IEEE Standard for Software Maintenance (IEEE Std 1219–1998).* s.l.:s.n.

ISO/IEC, 2006. *Software Engineering—Software Maintenance, ISO/IEC FDIS 14764:2006.* Geneva, Switzerland,: International Standards.

ISO-9001, 2001. *ISO 9001:2000 Quality management systems - Requirements.* s.l.:ISO/IEC.

ISO-9004, 2000. *Quality management systems - Guide lines for performance improvement.* s.l.:ISO/IEC.

ISO-9126-1, 2001. *Software engineering - Product Quality Part 1: Quality model.* s.l.:ISO/IEC.

ISO-9126-3, 2003. *Software engineering - Product Quality Part 3: Internal metrics.* s.l.:ISO/IEC.

KLINGER, T., TARR, P., WAGSTROM, P. & WILLIAMS, C., 2011. *An enterprise perspective on technical debt.* s.l., s.n.

KOSCHKE, R., 2008. Identifying and Removing Software Clones. In:: *Software evolution.* Berlin: Springer-Verlag, pp. 15-38.

Laing V., C. C., 2001. *Principal Components of Orthogonal OO Metrics,* s.l.: Software Assurance Technology Center (SATC).

LEHMAN, M. M., 1978. *Laws of Program Evolution - Rules and Tools for Programming Management.* s.l., s.n.

Lehman, M. M., 1980. Programs, life cycles, and laws of software evolution.. *Proceedings of the IEEE*, Setembro, 68(9), pp. 1060-1076.

LEHMAN, M. M., 1989. Uncertainty in Computer Application and Its Control Through the Engineering of Software. *Journal of Software Maintenance: Research and Practice*, 1(1), pp. 3-27.

LEHMAN, M. M., 1991. Software Engineering, the software process and their support. *Software Engineering Journal - Special issue on software process and its support*, 6(5), p. 243–258.

Lehman, M. M., 1996. *Laws of software evolution revisited.* London, Springer-Verlag, pp. 108-124.

LEHMAN, M. M., 1996. *Laws of software evolution revisited.* London, Springer-Verlag, pp. 108-124.

LEHMAN, M. M., 2001. *An Approach to a Theory of Software Evolution.* s.l., s.n., pp. 10-11.

LEHMAN, M. M., 2001. *Experiences with behavioural process modelling in FEAST, and some of its practical implications.* Haus Bommerholz, Springer, pp. 47-62.

Lehman, M. M. a. P. F. N., 1976. *Program evolution and its impact on software engineering..* Los Alamitos, CA, USA, IEEE Computer Society Press, p. 350–357.

LEHMAN, M. M. & BELADY, L. A., 1985. *Program Evolution – Process of Software Change.* London: Academic Press Professional, Inc..

LEHMAN, M. M. & RAMIL, J. F., 2001. Rules and Tools for Software Evolution Planning and Management. *Annals of Software Engineering*, November, 11(1), pp. 15-44.

LEHMAN, M. M. & RAMIL, J. F., 2002. Software Evolution and Software Evolution Processes. *Annals of Software Engineering*, Volume 14, p. 275 – 309.

LIENTZ, B. P. & SWANSON, E. B., 1980. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* s.l.:Addison-Wesley Longman Publishing Co., Inc..

Lincke, R. G. T. L. W., 2010. *Software Quality Prediction Models Compared.* Zhangjiajie, China, s.n., pp. 82-91.

Li, W. ,. H. S., 1993. *Maintenance Metrics for the Object Oriented Paradigm.* Baltimore, Maryland, IEEE.

Lorenz, M. &. K. J., 1994. *Object-Oriented Software Metrics.* s.l.:Prentice-Hall.

McCabe, T., 1976. A Complexity Measure. *IEEE Transactions on Software Engineering*, Dez, Volume vol.SE-2, pp. 308- 320.

McConnell, S., 2007. *Technical Debt.* [Online] Available at: http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx [Acesso em 18 Agosto 2010].

MCCONNELL, S., 2007. *Technical Debt.* [Online] Available at: http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx [Acesso em 18 Agosto 2010].

Mens, T. & Demeyer, S., 2008. *Software Evolution.* Berlin: Springer-Verlag.

MENS, T. & DEMEYER, S., 2008. *Software Evolution.* Berlin: Springer-Verlag.

Olague, H., 2007. Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. *IEEE Transactions on Software Engineering*, Junho, 33(6), pp. 402-419.

Oliveira, R. G. d., 2011. *Caracterização e Conceituação Teórica da Metáfora de Débito Técnico Através de um Estudo Exploratório.* Recife, PE: UFPE.

Pigoski, T. M., 1996. *Practical Software Maintenance: Best Practices for Managing Your Software Investment.* s.l.:Wiley.

Rüdiger, L., 2007. *Validation of a Standard- and Metric-Based Software Quality Model,* Växjö, Sweden: Växjö University.

Runeson, P. &. H. M., 2009. Guidelines for conducting and reporting case study research in software engineering. *Journal of Empirical Software Engineering*, apr, p. 131–164.

Siebra, C. et al., 2012. Managing Technical Debt in Practice: An Industrial Report. *In: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.*

Sommerville, I., 2007. *Engenharia de Software.* São Paulo: Pearson.

STRAUSS, A. & CORBIN, J., 2008. *Pesquisa qualitativa: Técnicas e procedimentos para o desenvolvimento de teoria fundamentada nos dados.* 2 ed. Porto Alegre: Artmed.

Vasilescu, B. S. A. v. d. B. M., 2011. *You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics.* Williamsburg, VA, USA, s.n., pp. 313-322.

Wolverton, R. W., 1974. The Cost of Developing Large-Scale Software. *IEEE Transactions on Computers*, June, c-23(6), pp. 615-636.

APÊNDICE A – WEB SITES DE TODAS AS FERRAMENTAS DE MÉTRICAS ANALISADAS

Ferramenta	Web Site
PMD	http://pmd.sourceforge.net/
FindBugs	http://findbugs.sourceforge.net/
CheckStyle	http://checkstyle.sourceforge.net/
Sonar	http://www.sonarsource.org/
Analyst4j	http://www.codeswat.com/cswat/index.php
CCCC	http://sourceforge.net/projects/cccc/
C & K Java Metrics	http://www.spinellis.gr/sw/ckjm/
Dependency Finder	http://depfind.sourceforge.net/
Eclipse Metrics 1.3.6	http://metrics.sourceforge.net/
Eclipse Metrics 3.4	http://marketplace.eclipse.org/content/eclipse-metrics-plugin-continued
OOMeter	http://www.ccse.kfupm.edu.sa/~oometer/
Semmle	http://semmle.com/
Understand for Java	http://www.scitools.com/
VizzAnalyzer	http://www.arisa.se/vizz_analyzer.php
CMTJava	http://www.testwell.fi/cmtjdesc.html
Resource Standard Metrics	http://msquaredtechnologies.com/
Code Pro AnalytiX	https://developers.google.com/java-dev-tools/codepro/doc/
JDepend	http://clarkware.com/software/JDepend.html
JHawk	http://www.virtualmachinery.com/jhawkprod.htm
JMetric	http://sourceforge.net/projects/jmetric/
Krakatau Metrics	http://www.powersoftware.com/products/
RefactorIT	http://sourceforge.net/projects/refactorit/
McCabe IQ	http://www.mccabe.com/
Cobertura	http://cobertura.sourceforge.net
Clover	http://www.atlassian.com/software/clover
CTC++ for Java and Android	http://www.verifysoft.com/fr_code_coverage_for_java_and_android.html
DevPartner	http://www.borland.com/products/devpartner/
EMMA	http://emma.sourceforge.net/
Jtest	http://www.parasoft.com/jsp/products/jtest.jsp
Kalistick	http://www.kalistick.com/
LDRA Testbed	http://www.ldra.com/index.php/en/products-a-services/ldra-tool-suite/ldra-testbed
JaCoCo	http://www.eclemma.org/jacoco/
AgileJ StructureViews	http://www.agilej.com/
Hammurapi	http://www.hammurapi.biz/hammurapi-biz/ef/xmenu/hammurapi-group/index.html
Soot	http://www.sable.mcgill.ca/soot/
Squale	http://www.squale.org/
SonarJ	http://www.hello2morrow.com/products/sonarj
Klocwork	http://www.klocwork.com/
JavaNCSS	http://javancss.codehaus.org/