WORKSHOP DE MANUTENÇÃO DE SOFWARE MODERNA
WMSWM

29 de maio de 2006

Vila Velha, Espírito Santo – Brasil

ANAIS

Promoção

SBC – Sociedade Brasileira de Computação
MCT – Ministério de Ciência e Tecnologia
PBQP Software – Programa Brasileiro da Qualidade e Produtividade em Software

Edição

Cláudia Maria Lima Werner (Universidade Federal do Rio de Janeiro)
Nabor das Chagas Mendonça (Universidade de Fortaleza)

Organização

UFRJ – Universidade Federal do Rio de Janeiro
UNIFOR – Universidade de Fortaleza
FAESA – Faculdades Integradas Espírito-Santenses

Realização

UFRJ – Universidade Federal do Rio de Janeiro
UNIFOR – Universidade de Fortaleza
FAESA – Faculdades Integradas Espírito-Santenses

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Workshop de Manutenção de Software Moderna (5.:2006 mai-29: Vila Velha)

Anais / Edição Cláudia Maria Lima Werner e Nabor das Chagas Mendonça – Vila Velha: Universidade Federal do Rio de Janeiro, UFRJ, Universidade de Fortaleza, UNIFOR, 2006.
v, 105 p.

ISBN 85-7669-062-4

Conhecido também como WMSWM 2006

1. Engenharia de Software. I. Werner, Cláudia Maria Lima. II. Mendonça, Nabor das Chagas. III. WMSWM (5.:2006: Vila Velha)

"Esta obra foi impressa a partir de originais entregues, já compostos pelos autores"

Editoração: Alexandre Dantas

Tiragem: 450 exemplares

Apresentação

É com grande satisfação que apresentamos a terceira edição do Workshop de Manutenção de Software Moderna – WMSWM´06 realizado em Vila Velha.

Um dos objetivos do WMSWM é de reunir a comunidade acadêmica interessada em problemas de manutenção de software, juntamente com profissionais que necessitam obter maior conhecimento sobre métodos e ferramentas modernos para lidar com esse novo cenário de manutenção de software.

Este ano foram submetidos 7 artigos técnicos e 5 relatos de experiência, dos quais foram selecionados 5 artigos técnicos e 4 relatos de experiência. Cada artigo foi avaliado por pelo menos 3 membros do comitê de programa, segundo os critérios estabelecidos na chamada.

Além da apresentação dos artigos técnicos e relatos de experiência, temos a realização de um painel sobre manutenção de software, envolvendo representantes da academia, governo e indústria.

Agradecemos o apoio do comitê organizador do SBQS, principalmente seu Coordenador Geral Prof. Ricardo de Almeida Falbo, da Universidade Federal do Espírito Santo, e Andrômeda Menezes, das Faculdades Integradas Espírito-Santenses, pela dedicação especial ao WMSWM. Aos doutorandos Leonardo Murta e Alexandre Dantas pelo apoio dado na elaboração da página do evento e da editoração desses anais. Aos autores dos artigos técnicos e relatos de experiência pela qualidade dos artigos. Finalmente, agradecemos aos membros do Comitê de Programa pelo excelente trabalho de avaliação. Sem a cooperação de todos este evento não seria possível!

Desejamos a todos um proveitoso workshop e uma ótima estadia em Vila Velha.

Vila Velha, maio de 2006.

Cláudia Maria Lima Werner
Nabor das Chagas Mendonça

Coordenadores do WMSWM 2006.

Workshop de Manutenção de Software Moderna

Coordenação

Cláudia Maria Lima Werner (COPPE/UFRJ)
Nabor das Chagas Mendonça (UNIFOR)

Comitê de Organização

Alexandre Dantas (COPPE/UFRJ)
Andromeda Menezes (FAESA)
Leonardo Murta (COPPE/UFRJ)

Membros do Comitê de Programa

Ana Regina Cavalcanti da Rocha (COPPE/UFRJ)
Antônio Francisco do Prado (UFSCar)
Cláudia Maria Lima Werner (COPPE/UFRJ)
Elisa H. Moriya Huzita (UEM)
Guilherme Horta Travassos (COPPE/UFRJ)
Junia Coutinho Anacleto Silva (UFSCar)
Karin Koogan Breitman (PUC-Rio)
Káthia Marçal de Oliveira (UCB)
Lucia Vilela Leite Filgueiras (Poli/USP)
Marcelo Augusto Santos Turine (UFMS)
Márcio Delamaro (UNIVEM)
Marcos Lordello Chaim (EACH, USP Leste)
Maria Istela Cagnin (UNIVEM)
Nabor das Chagas Mendonça (UNIFOR)
Nicolas Anquetil (UCB)
Paulo Cesar Masiero (ICMC/USP)
Paulo Henrique Monteiro Borba (UFPE)
Rodrigo Quites Reis (UFPA)
Rosana Teresinha Vaccare Braga (ICMC/USP)
Rosângela Penteado (UFSCar)
Rosely Sanches (ICMC/USP)

Sumário / Contents

Trabalhos Técnicos

Relatos de Experiência

Problemas em manutenção de software: caracterização e evolução

Mateus M. Paduelli, Rosely Sanches

Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP) – Avenida do Trabalhador São-carlense, 400.
Cep 13.560-970 – Caixa Postal 668 – São Carlos – SP – Brasil

{mateusmp, rsanches}@icmc.usp.br

Abstract. *The software engineering each time more is worried about the question of the software maintenance. This increasing concern can be verified by the events and works in the area, including the efforts in teaching software maintenance in an explicit way at the universities. This work looks for establish a parallel between the problems of maintenance identified more than two decades ago, with those observed currently, through a case study. The objective is to identify the characteristics of evolution of these problems, in way to raise indications of modifications that can have, or not, occurred among the mainly areas of difficulty in software maintenance.*

Resumo. *A engenharia de software cada vez mais se preocupa com a questão da manutenção de software. Essa preocupação crescente pode ser verificada pelos eventos e trabalhos na área, incluindo os esforços em ensinar manutenção de software de maneira explícita nas universidades. Este trabalho busca estabelecer um paralelo entre os problemas de manutenção identificados há mais de duas décadas, e aqueles observados atualmente com base em um estudo de caso. O objetivo é identificar as características de evolução desses problemas, de maneira a levantar indícios das modificações que possam ou não ter ocorrido entre as principais áreas de dificuldade em manutenção de software.*

1. Introdução

A atividade de manutenção de software está assumindo um papel cada vez mais importante dentro do chamado ciclo de vida de software. Ela é caracterizada pela modificação de um produto de software já entregue ao cliente, para a correção de eventuais erros, melhora em seu desempenho, ou qualquer outro atributo, ou ainda para adaptação desse produto a um ambiente modificado (IEEE, 1998).

Embora não exista um consenso sobre o valor exato do custo atrelado à atividade de manutenção, as pesquisas na área apontam, na totalidade dos casos, sempre mais de 50% dos investimentos realizados no software. De fato, para Bhatt *et al.* (2004), esse percentual corresponde a algo entre 67% a 75% do investimento total, enquanto para Polo *et al.* (1999) corresponde a um valor entre 67% e 90%. Ainda de acordo com esse mesmo autor, a razão desse custo elevado deve-se, em parte, à própria natureza da atividade de manutenção, caracterizada principalmente pela imprevisibilidade. Além dos altos custos financeiros, essa é também a atividade que exige maior esforço dentre as atividades de engenharia de software (Sneed, 2003).

A importância financeira atrelada à manutenção de software é ainda agravada quando se leva em consideração o risco para as oportunidades de negócio, que podem ser causadas pela falta de gerenciamento e compreensão total da dinâmica dessa atividade. De acordo com a pesquisa realizada por Dart *et al.* (1993), esse gerenciamento deve considerar três fatores: (i) ferramentas, (ii) pessoas, (iii) processos, revelando-se, pois, uma atividade gerencial complexa.

O objetivo deste trabalho é contribuir para o aumento do conhecimento em torno da atividade de manutenção de software, comparando os principais problemas de manutenção identificados há mais de duas décadas, com os obtidos atualmente por meio de um estudo de caso em uma empresa que desenvolve e mantém software comercial, de forma a identificar como esses problemas podem estar evoluindo.

Na s*eção 2* são tratados os conceitos referentes à manutenção de software, incluindo suas principais características e classificações. Os resultados de um dos primeiros estudos no sentido de constatar quais os problemas que mais iterferem na atividade de manutenção de software são apresentados na s*eção 3.*

Na *seção 4*, é apresentada a descrição e os resultados do estudo de caso realizado. Um levantamento de trabalhos correlatos é feito na *seção 5*. Finalmente, na s*eção 6,* são relacionadas as conclusões e propostas para trabalhos futuros.

2. Manutenção de Software

O processo de desenvolvimento de software, bem como os fatores de qualidade ligados a essa atividade e ao ciclo de vida de um software, é largamente discutido na literatura, por ser de fundamental importância para a garantia da competitividade e da qualidade do produto de software oferecido ao usuário final. No entanto, dentre as atividades previstas no ciclo de vida de um software, a manutenção é a que possui a menor atenção e conseqüente incipiência em seu ensino (Dias, 2004).

2.1 Principais Características

A manutenção de software é uma operação importante pois consome a maior parte dos custos envolvidos no ciclo de vida de um software (SWEBOK, 2004), e a falta de habilidade em mudar um software rapidamente, e de maneira confiável, pode causar a perda de oportunidades de negócio (Bennett e Rajlich, 2000).

A grande maioria dos estudos existentes nessa área utiliza a metodologia de "pesquisar-e-tranferir", no sentido de pesquisa fechada dentro do meio acadêmico, enquanto existe uma necessidade de utilizar a abordagem "indústria-como-laboratório", uma vez que é nas indústrias que a atividade de manutenção existe como atividade real e intensa no dia-a-dia (Niessink, 1999).

A atividade de manutenção não pode ser considerada como uma operação nova, pois desde que os primeiros softwares surgiram, a manutenção também surgiu (Zvegintzov e Parikh, 2005). No entanto, ela nunca foi de fácil previsão e controle, sendo muitas vezes realizada por equipes terceirizadas, que não possuem nenhum contato com o projeto inicial do software (Bhatt *et al.*, 2004). Ainda segundo esse autor, o aumento na complexidade dos softwares produzidos (tanto em termos de funcionalidades, como de técnicas) torna a previsão de esforços de manutenção muito vaga.

Essa dificuldade em estimar esforços torna-se mais acentuada quando se trata de sistemas legados. Assim como os softwares desenvolvidos mais recentemente, os

sistemas legados apresentam dificuldade de manutenção em função de sua complexidade e tamanho, porém essa dificuldade é agravada pela rotatividade de pessoal, documentação insuficiente e extensão das manutenções, muitas vezes realizadas sobre relações desconhecidas ou não-triviais sobre os componentes do software (Shirabad *et al.*, 2003). Não é de se esperar que uma empresa de grande porte troque todos seus sistemas somente pelo fato de que a tecnologia neles empregada está ultrapassada. Esses sistemas representam ativos importantes da organização e ela estará disposta a investir de maneira a manter o valor desses ativos (Sommerville, 2003).

Se o software atual funciona adequadamente, supõe-se que a organização terá uma preferência muito maior em aplicar a eles apenas os ajustes necessários em função de mudanças nos negócios ou outras necessidades de correções, do que substituí-los. Abandonar sistemas legados para iniciar projetos a partir do zero representa uma perda considerável (Silva e Santander, 2004), o que torna esses sistemas mais um desafio para os esforços de criar métodos, técnicas e processos adequados para tratar a manutenção de software. Como resultado desses esforços, pode-se citar a metodologia de pontos por função para manutenção de software, empenhada em ajudar a estimar custos para a atividade (Niessink e Vliet, 1997).

2.2 Tipos de Manutenção

As ações ligadas à manutenção de software podem ser agrupadas como pertencentes a quatro categorias (Lientz e Swanson, 1980), (Pressman, 2005): (i) adaptativas: referem-se a adequar o software ao seu ambiente externo (sistema operacional, regras de negócios), (ii) perfectivas: consistem em tornar o software mais completo por meio da inclusão de novas funcionalidades, (iii) corretivas: corrigem defeitos de funcionalidade, (iv) preventivas: evitam problemas futuros, normalmente por meio de uma reengenharia de software. Pressman ainda completa que o grande esforço necessário na manutenção se justifica pela abrangência do significado desse termo no contexto de software.

Um estudo com empresas de software (Souza *et al.*, 2004), constatou que dentre as empresas pesquisadas, as manutenções do tipo corretivas prevaleceram com 50% dos resultados, seguido pelas perfectivas (30%) e adaptativas (20%). Nessa pesquisa não se constatou nenhuma empresa realizando manutenção do tipo preventiva.

Durante a manutenção de um software, dificilmente as operações realizadas serão exclusivamente classificadas como pertencentes a apenas um dos tipos apresentados. De fato, o que se observa na prática é uma mistura desses tipos.

Embora normalmente aceita, essa classificação de atividades de manutenção não possui um consenso, podendo variar de acordo com o autor, conforme explica Sommerville (2003). As manutenções do tipo adaptativas, além de se referirem a adequar o software ao ambiente, podem também se referir a adequá-lo a novos requisitos. Já as manutenções perfectivas podem significar ora completar o software em termos de funcionalidades, ora manter as funcionalidades por meio da melhoria de sua estrutura e desempenho.

3. Problemas de Manutenção de Software

Em 1980, uma publicação (Lientz e Swanson, 1980) descrevia os resultados obtidos no mais longo estudo, até então realizado, no intuito de averiguar a forma como as organizações tratavam a questão de manutenção de software. Trata-se de um estudo realizado com base em 487 empresas, abrangendo organizações governamentais,

bancos, transportes, mineração, educação e indústrias de manufatura em geral, localizadas nos Estados Unidos e Canadá.

A pesquisa contou com duas fases. Na primeira, realizada com 69 organizações, foram levantadas as características de manutenção de software por elas realizadas. Essa fase inicial obteve as seguintes conclusões:

- A manutenção de software existente consome uma média de 48% das horas anuais do pessoal de sistemas e programação;
- Aproximadamente 60% dos esforços de manutenção são dedicados a manutenções perfectivas. Dentre esse percentual, dois terços se referem a esforços de melhoria de software;
- Problemas de natureza gerencial em manutenção foram vistos como mais importantes do que aqueles de natureza técnica;
- A manutenção de software existente tende a ser vista pela gerência como algo mais importante do que o desenvolvimento de novos sistemas.

Com base nessas conclusões, uma segunda fase foi conduzida na pesquisa, agora envolvendo 487 organizações, com o intuito de validar os resultados obtidos na primeira fase, buscando uma compreensão mais profunda e abrangente.

Para se obter esses resultados, a segunda fase buscou saber de diretores, gerentes e supervisores da área de informática dessas organizações, as características dos esforços de manutenção empregados em sistemas considerados de grande importância para a organização, que constituíam resultados de investimentos significativos.

Os principais resultados obtidos nessa segunda análise estão resumidos a seguir:

- Mais de um terço dos sistemas descritos tinham sua manutenção realizada por apenas uma pessoa;
- A maioria dos profissionais alocados para manutenção tinha outras tarefas ao mesmo tempo;
- Do total de manutenções, cerca de 20% representavam manutenções corretivas, 25% manutenções adaptativas e mais de 50% manutenções perfectivas (isso está de acordo com aos resultados obtidos na primeira fase da pesquisa);
- Quanto mais velho era o sistema, maior era o esforço que deveria ser empregado em sua manutenção, e mais sujeita a organização estava em perder o conhecimento em cima desse sistema, devido à rotatividade dos profissionais.

Finalmente, os autores chegaram aos principais problemas de manutenção enfrentados pelas organizações na época. Tais problemas estão descritos a seguir:

(i) Baixa qualidade da documentação dos sistemas;
(ii) Necessidade constante dos usuários por melhorias e novas funcionalidades;
(iii) Falta de uma equipe de manutenção;
(iv) Falta de comprometimentos com cronogramas;
(v) Treinamento inadequado do pessoal de manutenção;
(vi) Rotatividade dos profissionais.

Devido à sua representatividade em relação à definição dos problemas de manutenção de software de décadas atrás, esse trabalho será considerado como ponto de partida para a comparação dos problemas de manutenção do passado com os identificados em um estudo de caso atual.

4. Estudo de Caso

Um estudo de caso foi conduzido com a intenção de verificar os problemas de manutenção enfrentados por uma organização empenhada no desenvolvimento de software comercial. Essa organização mantém uma base de dados contendo registros históricos de manutenções sobre um sistema de informação por ela desenvolvido e mantido. As subseções a seguir descrevem o estudo e os resultados obtidos.

4.1 Metodologia

Para obtenção dos problemas de manutenção, a pesquisa contou com duas etapas. Na primeira, a preocupação foi em analisar estatisticamente os registros de solicitações de manutenção armazenados em um banco de dados construído especificamente para esse fim, visando identificar problemas de caráter técnico.

Na segunda etapa, o enfoque foi em relacionar os problemas de manutenção de caráter gerencial, por meio de entrevistas com os funcionários empenhados nas tarefas de desenvolvimento, planejamento e manutenção. Essas entrevistas buscaram entender como a organização tratava diferentes aspectos ligados à atividade de manutenção, e foi estruturada de forma a obter respostas para as seguintes questões:

(i) Como a organização gerencia a atividade de manutenção no sentido de:
   a. Tratar as solicitações registradas;
   b. Como avaliar as solicitações;
   c. Qual o critério de elaboração de cronograma;
   d. Como as atividades são distribuídas internamente;
   e. Como validar a manutenção efetuada;
   f. Como a versão alterada é fornecida ao cliente

(ii) Qual o processo para tratar documentação das manutenções?

(iii) Como é feito o controle de versões de arquivos modificados?

(iv) Quais critérios guiam a mudança de prioridades das manutenções?

(v) Qual o domínio que os profissionais possuem da tecnologia empregada?

(vi) Como são tratados os problemas de má especificação das manutenções?

(vii) Quais as principais dificuldades para entrega das manutenções no prazo?

Com base nos resultados dessas duas etapas, foi possível obter um quadro dos principais problemas ligados à atividade de manutenção de software na empresa, problemas esses relacionados tanto ao controle interno, no sentido de gerenciar pessoas e processos, como também aqueles ligados ao uso de tecnologias.

4.2 Registros de Manutenções

Os dados utilizados como base para a pesquisa estão organizados em uma base de dados construída e mantida por meio do *SQL Server 2000* com o objetivo específico de administrar as solicitações de manutenção referentes ao principal sistema de informação desenvolvido e mantido pela empresa de software analisada. Esse software foi construído em ambiente *Delphi* e apresenta recursos avançados como troca de dados com bancos, controle de permissões de usuários e integração com sistemas *Web*. Os registros de manutenção são inseridos na base de dados de três formas distintas: (i) pelo próprio cliente, através do sistema de *help-desk* disponível no site da empresa; (ii) pelos consultores da empresa, após visita ao cliente e levantamento de necessidades de manutenção; (iii) pelos próprios desenvolvedores, que podem identificar necessidades

de manutenção à medida que "evoluem" o software. Uma interface específica em linguagem *ASP* foi criada para a inserção dos registros de manutenção nessa base por parte dos desenvolvedores e consultores da organização, podendo ser acessado via internet.

Cada registro de manutenção contém informações, como por exemplo, qual o cliente que está solicitando, grau de prioridade, quem solicita e quem realizará a manutenção, tempo gasto previsto para a atividade, tipo de manutenção, *status* da implementação, datas de inserção do chamado e data prevista para entrega, observações, descrição da solução adotada, entre outros.

Após o registro de um chamado, sua execução dependerá da análise de viabilidade, efetuada pelo responsável do setor de desenvolvimento. O procedimento adotado está resumido na Figura 1.

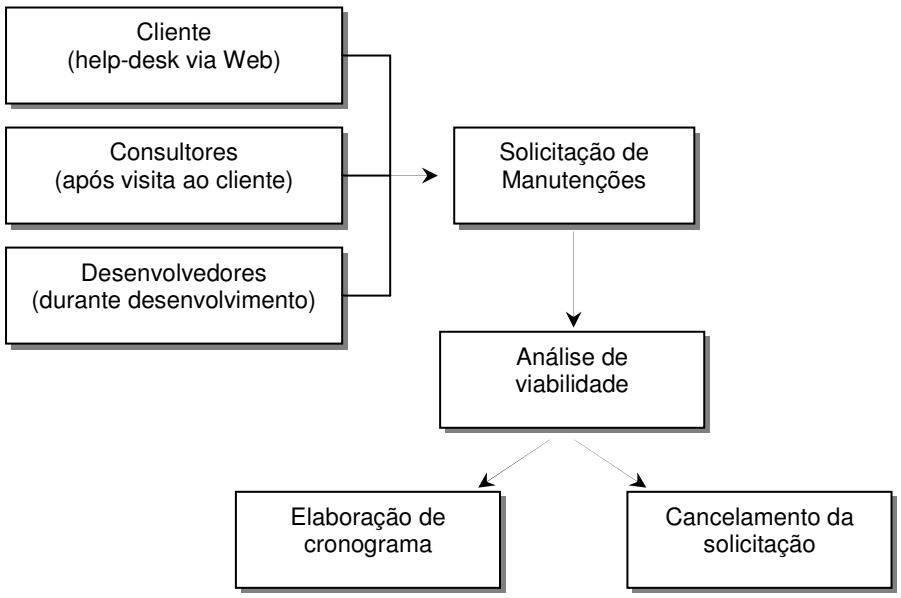


Figura 1. Etapas envolvidas na solicitação de manutenções

Eventualmente, solicitações de manutenção são consideradas inviáveis, sendo canceladas pelo responsável do setor de desenvolvimento. Uma vez cancelada, esse fato é informado ao cliente que solicitou, podendo ser revisada a necessidade para que uma proposta de manutenção mais adequada seja registrada.

4.3 Análise dos Dados

A base de dados, no momento da pesquisa, contava com mais de 3700 solicitações de manutenção, realizadas por 41 clientes. Esses clientes envolvem desde organizações de pequeno porte, utilizando o software em menos de 5 computadores, até empresas maiores, com até 50 máquinas utilizando o software de maneira simultânea. Normalmente essas empresas de maior porte são estruturadas por departamentos, e cada departamento utiliza um módulo específico do software.

Na Figura 2 é apresentado o número de solicitações registradas mensalmente desde a implantação do sistema de registro de manutenções, em abril de 2004, até a data de realização desse trabalho.

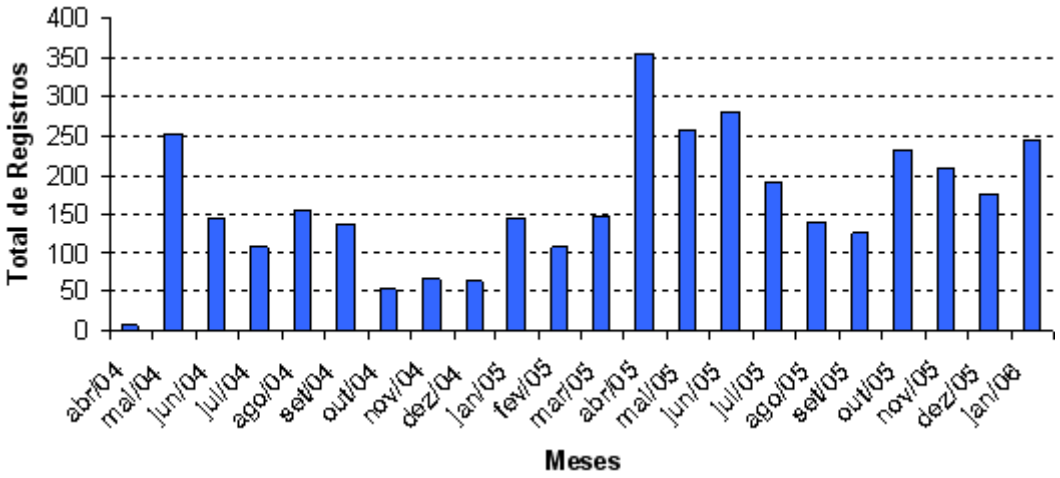


Figura 2. Total mensal de solicitações de manutenção

Nota-se, pela figura, que não existe uma constância no número de manutenções por mês, o que sugere uma imprevisibilidade nas necessidades de manutenção por parte dos clientes. Em abril de 2005 foi registrado o maior número de chamados de manutenção (355). Os dados representam uma média de 163,45 solicitações de manutenção por mês (desvio padrão de 83,7). Constatou-se ainda que a média de solicitações diária é de 8,59 novas requisições.

Os registros de solicitações de manutenção incluem desde operações simples, que resultam em poucos dias para a entrega, até solicitações mais complexas, que podem levar semanas, e até meses, para serem concluídas. Solicitações desse porte normalmente exigem que o projeto de um ou mais módulos seja reestruturado, o que justifica o tempo gasto para entrega ao cliente.

A distribuição, em termos de dias gastos para entrega de manutenções, é mostrada na Figura 3.

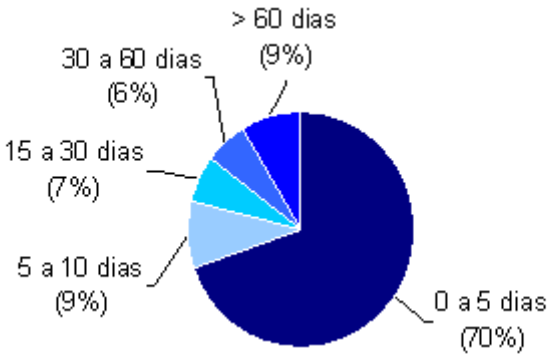


Figura 3. Dias necessários para entrega de manutenções

Uma análise do gráfico revela que as manutenções de menor grau de complexidade ocupam a maior parte do tempo dos mantenedores. No entanto, existem manutenções com complexidade suficiente para exigir mais de dois meses de trabalho.

Observou-se que a grande quantidade de manutenções de menor complexidade acaba por interferir no tempo de entrega das manutenções mais complexas, gerando atrasos. Isso ocorre em função da grande expectativa do cliente no sentido de obter respostas rápidas, o que força os mantenedores a liberar primeiro as manutenções mais

simples. Na Figura 4 é mostrado qual o grau de prioridade indicado pelo cliente ao fazer as solicitações de manutenção.

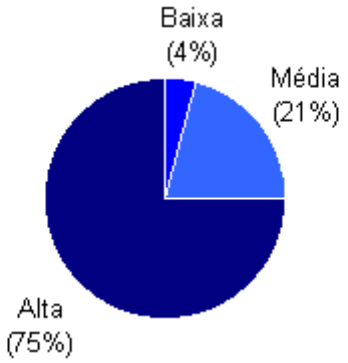


Figura 4. Distribuição de prioridades das manutenções

Percebe-se, pela figura anterior, que o cliente busca respostas rápidas, considerando com prioridade elevada a maior parte dos chamados de manutenção efetuados.

Do ponto de vista dos quatro tipos de manutenção software (*adaptativa, perfectiva, corretiva, preventiva*), realizou-se a quantificação das manutenções efetuadas dentro desses tipos. Para esse levantamento, foi considerada a classificação das manutenções informada em cada registro feito na base de dados em estudo. O resultado obtido é apresentado na Figura 5.

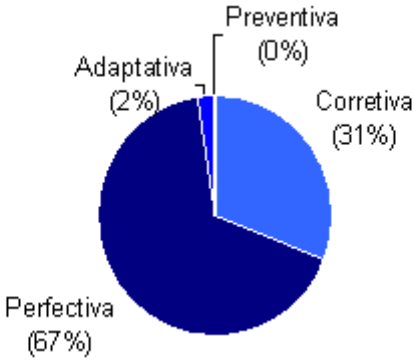


Figura 5. Distribuição dos tipos de manutenção

Pelos valores apresentados anteriormente, percebe-se que o maior esforço da organização está em atender requisições de novas funcionalidades, o que comprova o enfoque em manutenção perfectiva, demonstrado por outros estudos. Uma constatação preocupante foi a de que não existe uma atenção em avaliar e prevenir problemas futuros no software, o que poderia ser feito por um processo de reengenharia. Fica evidente que o enfoque está em manter o software em funcionamento, adequando-o na medida em que as necessidades surgem, ou seja, seria uma postura de "aguardar acontecer" e não de "buscar prevenir".

É esperado, pela literatura, que o tempo empenhado em tarefas de manutenção exceda o tempo de desenvolvimento, e esse fato buscou ser comprovado pela comparação entre tempo gasto com o desenvolvimento da primeira versão entregue ao cliente e o tempo gasto com a manutenção. Para isso, elegeram-se três módulos representativos do software analisado (aqui referenciados por *módulo 1*, *módulo 2* e *módulo 3*), com o intuito de contabilizar o tempo gasto em desenvolvimento e o gasto

em manutenção. O tempo de desenvolvimento inclui tempo gasto em projeto, codificação e testes.

Esse resultado é apresentado na Figura 6, que considera os registros de manutenção efetuados até o momento de realização deste trabalho.

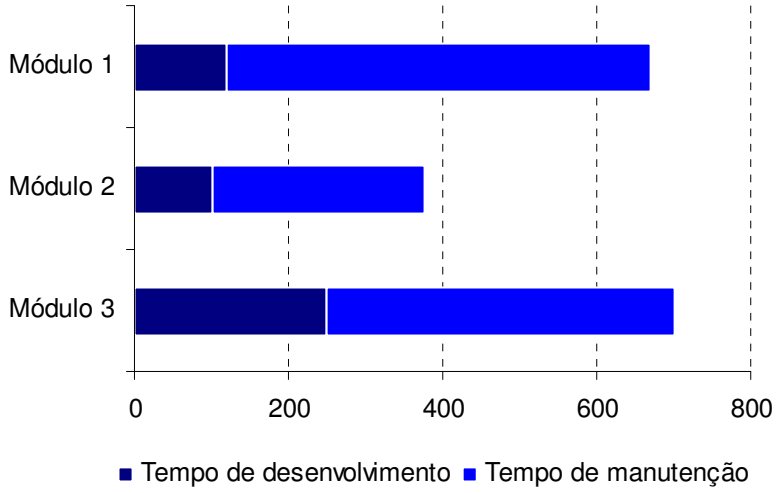


Figura 6. Tempo consumido em desenvolvimento e em manutenção (horas)

De acordo com a figura, percebe-se que a organização consumiu, nos três exemplos, mais de 60% do esforço empregado em atividades de manutenção, e esse percentual tende a subir com o passar do tempo.

4.4 Problemas Identificados

Durante entrevista com os responsáveis pelas manutenções, constatou-se que não existe um procedimento padrão a ser seguido para a execução dessas manutenções. O que existe é uma preocupação de agendar datas de entrega para as solicitações cujos clientes responsáveis insistem em respostas mais rápidas. Casos de manutenções de urgência, quando ocorrem, são priorizados em detrimento das demais.

Observou-se também que o código-fonte modificado não é documentado de maneira adequada, sendo muitas vezes atribuídas somente pequenas notas, ou o número do chamado que resultou na modificação de um trecho de código. Esse número de chamado está relacionado com o registro de necessidade de manutenção efetuado pelo cliente.

Os profissionais encarregados de manutenção relataram ainda que nem sempre o cliente compreende totalmente o que está solicitando, muitas vezes gerando discussões entre empresa/cliente, o que está relacionado com os problemas de comunicação entre usuário e desenvolvedor. Muitas vezes o cliente acredita que uma determinada manutenção será suficiente para adequar o software à sua nova necessidade de negócio, quando na verdade não é suficiente. Essa dificuldade de compreensão do software pelo próprio cliente acaba por gerar situações de desgaste na relação entre empresa-cliente.

Prazo de entrega é sempre um problema quando não existe um processo para conduzir a manutenção. Esse foi outro fato verificado, uma vez que nem sempre a manutenção, com data agendada e informada ao cliente, pode ser entregue no prazo

combinado, geralmente em função de re-trabalho em manutenções já realizadas, ou mudanças de prioridades de entrega de acordo com os clientes.

Em parte esse atraso ocorre pelo fato de o mantenedor ser também a pessoa que desenvolve, de forma que a tarefa de desenvolvimento precisa mesclar-se com a de manutenção, contribuindo para a diminuição de performance do mantenedor. Isso se agrava quando testes em uma nova funcionalidade estão sendo feitos em paralelo a alguma correção solicitada pelo cliente.

Um outro ponto importante verificado refere-se ao baixo interesse dos profissionais envolvidos em tarefas de manutenção, fato que pode ser comprovado pelo interesse generalizado desses profissionais em deixar a atividade de manutenção para dedicar-se ao desenvolvimento. Esse interesse foi manifestado por todos os profissionais de manutenção entrevistados.

Uma política pouco confiável se verificou nos procedimentos de teste das manutenções efetuadas. Em razão da complexidade do software, e da ausência de um processo de manutenção que guie as etapas compreendidas entre entender a solicitação do cliente, até testar a conformidade e funcionalidade da manutenção efetuada, a organização em questão optou por eleger alguns clientes e disponibilizar para eles versões de teste do software após manutenções, a fim de que o próprio cliente efetue testes mais aprimorados e aponte eventuais problemas. Essa abordagem não se mostrou eficiente, uma vez que nem sempre o cliente tinha tempo e boa vontade para avaliar uma versão de testes em ambiente separado daquele de produção.

Em função dessas constatações, e das análises efetuadas, foi possível montar o quadro a seguir, que envolve os principais problemas de manutenção identificados na organização.

Quadro 1. Problemas de manutenção de software

Problemas Gerenciais	Ausência de um processo de manutenção de software
	Grande expectativa dos usuários
	Elevada rotatividade de membros e funções dentro da equipe
	Sobrecarga de tarefas
	Prazos não condizentes com a complexidade do software
	Baixa motivação entre profissionais de manutenção
	Ausência de manutenção preventiva
	Falhas de comunicação com o usuário
	Atrasos na entrega
Problemas Técnicos	Registro inexistente ou superficial de manutenções anteriores
	Ausência de um ambiente computacional específico para manutenção
	Validação insuficiente de manutenções efetuadas
	Documentação insuficiente ou superficial
	Falta de compreensão do software e suas estruturas

Com esses resultados, o próximo passo seria responder à questão: *Os problemas de manutenção de software de hoje são distintos daqueles verificados no passado?*

Para tornar mais clara essa análise, o quadro a seguir foi construído com o intuito de fazer um paralelo entre os problemas apontados na *seção 3*, obtidos por Lientz e Swanson (1980), e os verificados aqui. O quadro procura associar os problemas passados com os atuais, relacionando-os de acordo com a sua natureza.

Quadro 2. Comparação entre problemas de manutenção

Problemas apontados por Lientz e Swanson (1980)	**Problemas verificados**
Baixa qualidade da documentação dos sistemas	*- Registro inexistente ou superficial de manutenções anteriores;* *- Documentação insuficiente ou superficial.*
Necessidade constante dos usuários por melhorias e novas funcionalidades	*- Grande expectativa dos usuários;* *- Falhas de comunicação com o usuário.*
Falta de uma equipe de manutenção	*- Ausência de um processo de manutenção de software;* *- Sobrecarga de tarefas;* *- Ausência de manutenção preventiva;* *- Ausência de um ambiente computacional específico para manutenção.*
Falta de comprometimentos com cronogramas	*- Prazos não condizentes com a complexidade do software;* *- Atrasos na entrega.*
Treinamento inadequado do pessoal de manutenção	*- Baixa motivação entre profissionais de manutenção;* *- Validação insuficiente de manutenções efetuadas;* *- Falta de compreensão do software e suas estruturas.*
Rotatividade dos profissionais	*- Elevada rotatividade de membros e funções dentro da equipe.*

Embora não se estabeleça uma equivalência exata, esse quadro mostra uma semelhança muito grande entre a natureza dos problemas obtidos em cada levantamento.

5. Trabalhos correlatos

Em uma pesquisa realizada por Dekleva (1992), uma lista de problemas de manutenção muito parecida com os apresentados nesse trabalho foi obtida, porém incluindo o problema da dificuldade da organização em medir a performance da equipe de manutenção.

Outros trabalhos relacionados a essa linha de estudo, obtiveram resultados muito semelhantes. Pode-se citar a pesquisa de Dart *et al.* (1993), conduzida pelo SEI (*Software Engineering Institute*) junto a uma agência do governo dos Estados Unidos. Nesse trabalho, alguns outros problemas foram levantados, que, no entanto, podem ser agrupados em alguma das categorias apresentadas no quadro 2. Por exemplo, os profissionais encarregados de tarefas de manutenção alegaram que acabam não tendo contato com ferramentas que representariam o estado-da-arte da tecnologia. Esse problema pode ser incluído no item *Baixa motivação entre profissionais de manutenção*. Ainda um outro exemplo com base na pesquisa do SEI, foi o de que se constatou que o conhecimento e as lições adquiridas com a experiência de manutenção

em um determinado sistema não estavam sendo disseminados para as outras equipes dentro da mesma empresa, o que pode ser comparado com o item *Ausência de um processo de manutenção de software*, do quadro 2.

6. Conclusões e Trabalhos Futuros

São escassos na literatura os trabalhos que visam documentar as principais áreas de problemas durante a atividade de manutenção de software.

Os resultados do estudo realizado nesse trabalho indicam que não houve grandes mudanças nas áreas de problemas identificadas no passado, com as levantadas com base no estudo de caso apresentado. Os mesmos problemas apontados antes, encontram equivalências com os identificados atualmente. Esse fato pode ser considerado como positivo, já que indica uma constância nas principais áreas de dificuldade em manutenção de software, independente da tecnologia ou técnica que esteja sendo empregada para o desenvolvimento. Isso permite que uma sistematização do estudo de problemas de manutenção possa ser estabelecida.

Se os problemas atuais fossem muito diferentes dos verificados no passado, então seria difícil decidir sobre quais problemas buscar técnicas e procedimentos para redução dos esforços empregados, porém não foi o que se verificou.

Pelos resultados apontados, percebe-se que os problemas ligados a questões gerenciais estão mais presentes do que aqueles de caráter mais técnico. De fato, contatou-se que conciliar manutenções e os anseios dos usuários não é uma tarefa simples, e infelizmente não vem obtendo o sucesso desejado.

Como proposta para trabalhos futuros, sugere-se a validação dos resultados apontados, por meio de trabalhos de investigação de problemas de manutenção mais abrangentes, envolvendo mais organizações, com características variadas. Uma pesquisa por meio de questionários, na qual os problemas identificados aqui fossem apresentados para profissionais à frente de equipes de desenvolvimento e manutenção de software, com o intuito de verificar se tais problemas ocorrem ou não em suas organizações, também constitui uma proposta de aprofundamento das conclusões apresentadas.

Esse trabalho vem agregar esforços em compreender melhor a atividade de manutenção de software, mostrando que a engenharia de software pode valer-se de uma sistematização dos principais problemas de manutenção, de maneira a estabelecer grupos de problemas bem definidos, que possam ser abordados de maneira independente pelas pesquisas, que buscariam obter técnicas e procedimentos para tratá-los.

Referências Bibliográficas

Bennett, K.H.; Rajlich, V.T. (2000) “Software maintenance and evolution: a roadmap”, In: Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, June.

Bhatt, P.; Shroff, G.; Misra, A.K. (2004) “Dynamics of software maintenance”, ACM SIGSOFT Software Engineering Notes, v. 29, n. 5 (September), p. 1-5.

Dart, S.; Christie, A.M.; Brown, A.W. (1993) “A Case Study in Software Maintenance”, Relatório Técnico CMU/SEI-93-TR-8, Carnegie Mellon University, June.

Dekleva, S. (1992) “Delphi study of software maintenance problems”, In: Proceedings of the Conference on Software Maintenance, November.

Dias, M.G.B. (2004) "Uma experiência no ensino de manutenção de software", In: Anais do Workshop de Manutenção de Software Moderna (WMSWM'04), Brasília, DF, Brasil, Outubro.

IEEE (1998) "Std 1219 – IEEE Standard for Software Maintenance", Institute of Electrical and Eletronic Engineers. New York, 52p.

Lientz, B.P.; Swanson, E.B. (1980) "Software Maintenance Management", Reading, MA: Addison-Wesley.

Niessink, F. (1999) "Software Maintenance Research in the Mire?", In: Proceedings of the Annual Workshop on Empirical Studies of Software Maintenance (WESS'99), Oxford, United Kingdom, September.

Niessink, F.; Vliet, H. (1997) "Predicting Maintenance Effort with Function Points", In: Proceedings of the International Conference on Software Maintenance (ICSM'97), Bari, Italy, October.

Polo, M.; Piattini, M.; Ruiz, F.; Calero, C. (1999) "Roles in the maintenance process", ACM SIGSOFT Software Engineering Notes, v. 24, n. 4 (July), p. 84-86.

Pressman, R.S. (2005) "Software Engineering: a practitioner's approach", 6.ed.,McGrawHill Higher Education.

Shirabad, J.S.; Lethbridge, T.C.; Matwin, S. (2003) "Applying data mining to software maintenance records", In: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative Research. Toronto, Ontario, Canadá, October.

Silva, L. de P.; Santander, V.F.A. (2004) "Uma Análise Crítica dos Desafios para Engenharia de Requisitos em Manutenção de Software", In: Anais do Workshop em Engenharia de Requisitos, Tandil, Argentina, Dezembro.

Sneed, H.M. (2003) "Critical Success Factors in Software Maintenance", In: Proceedings of the International Conference on Software Maintenance, Amsterdam, The Netherlands, September.

Sommerville, I. (2003) "Engenharia de Software", 6.ed., São Paulo: Addison Wesley.

Souza, S.C.B. de; Neves, W.C.G. das; Anquetil, N.; Oliveira, K.M. de. (2004) "Documentação Essencial para Manutenção de Software II", In: Anais do I Workshop de Manutenção de Software Moderna, Brasília, Brasil, Outubro.

SWEBOK (2004). "The Institute of Electrical and Electronics Engineers, Inc – IEEE", Guide to the Software Engineering Body of Knowledge, Version 2004.

Zvegintzov, N.; Parikh, G. (2005) "60 years of Software Maintenance: Lessons Learned", In: Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM 2005), Budapest, Hungary, September.

Estimativas de Manutenção de Software a partir de Casos de Uso

Yara Maria Almeida Freire, Arnaldo Dias Belchior

Universidade de Fortaleza – Mestrado de Informática Aplicada
Av. Washington Soares 1321, 60811-341 – Fortaleza – CE – Brasil
yara@bnb.gov.br, belchior@unifor.br

Abstract. *Activities to estimate size, effort, schedule and cost have been gaining more importance, once more reliable projects estimates are helpful on monitoring and plan. This work proposes a software maintenance estimates, developed based on approach of use cases, extending the technique TUCP (Technical Use Case Points), proposed for systems development projects estimates.*

Resumo. *As atividades de estimativas de tamanho, esforço, prazo e custo têm alcançado cada vez mais importância, uma vez que estimativas confiáveis relacionadas ao projeto auxiliam bastante seu planejamento e monitoração. Este trabalho propõe estimativas de manutenção de software, desenvolvido a partir da abordagem de casos de uso, estendendo a técnica TUCP (Pontos de Caso de Uso Técnico), proposta para estimativas de projetos de desenvolvimento de sistemas.*

1. Introdução

Ao longo do ciclo de vida de um sistema, seus requisitos originais sofrem manutenção, para refletirem as necessidades de seus clientes e as mudanças de seus usuários. A manutenção pode envolver a evolução, a adaptação ou a correção de artefatos do sistema.

No início dos anos noventa, estatísticas revelaram que muitas organizações alocaram, no mínimo, 50% de seus recursos financeiros em manutenção de software [April, 1995; Bourque *et al*., 1996]. Na década atual, a manutenção de software tem chegado a 70% dos custos de um sistema de software, tornando-se um dos grandes desafios da engenharia de software [Webster *et al*., 2004]. Isto evidencia a importância das pesquisas sobre manutenção, através das quais técnicas e métodos são criados ou metodologias existentes são adaptadas, para contemplar as atividades inerentes a este processo, como é o caso das estimativas de tamanho de um projeto de manutenção.

Segundo McPhee (1999), a estimativa de tamanho de software é um processo pelo qual uma pessoa ou um grupo de pessoas estima o tamanho de um produto de software. Para Ross (2004), o tamanho geralmente tem impacto na solução técnica e na gestão do projeto, já que estimativas imprecisas podem levar ao fracasso do projeto. Neste contexto, a precisão das estimativas de tamanho torna-se fundamental para a elaboração de cronograma e orçamento realistas, pois essas estimativas constituem-se na base para a derivação das estimativas de esforço, prazo, e custo (SEI, 2002).

Existem várias técnicas de estimativas de tamanho de software, principalmente para o desenvolvimento de aplicativos. Apesar da existência de diversas técnicas, a grande maioria delas não se aplica adequadamente a projetos de manutenção de software. Segundo Ramil, e Lehman (2000), a técnica de Análise de Pontos por Função (*Function Point Analysis*) é uma das mais conhecidas e utilizadas para se estimar o tamanho de um projeto de software, sendo um importante insumo para as estimativas de custos. No entanto, a FPA [FPCPM, 1999] calcula o tamanho da manutenção de um *software* como sendo o mesmo tamanho para desenvolvê-lo. Isto eleva o cálculo do tamanho da manutenção, tornando irreal a estimativa de tamanho.

Este trabalho propõe estimativas de tamanho, esforço, prazo e custo de manutenção de *software*, desenvolvido a partir da abordagem de casos de uso. Esse trabalho estende a técnica TUCP (Pontos de Caso de Uso Técnico), proposta por Monteiro *et al*. (2005) para estimativas de projetos de desenvolvimento de sistemas.

Este trabalho está organizado da seguinte forma: na seção 2, discorre-se sobre as estimativas de *software*; na seção 3, descreve-se a técnica TUCP, na qual este trabalho está baseado; na seção 4, mostra-se a proposta de extensão da TUCP para projetos de manutenção de *software* e uma aplicação real de sua utilização; na seção 6, apresentam-se as conclusões.

2. A Importância das Estimativas de Manutenção

Um processo de manutenção diz respeito a um conjunto de etapas bem definidas, que direcionam a atividade de manutenção de *software*, com o objetivo primordial de satisfazer as necessidades dos usuários de maneira planejada e controlada [Pigosky, 1996].

Segundo a NBR ISO/IEC 12207 (1998), o objetivo do processo de manutenção de *software* é modificar um produto de *software* existente, preservando sua integridade. Esse processo é ativado, quando o produto de *software* é submetido a alterações, a partir do momento em que se necessitam efetuar modificações no código e na documentação do sistema devido a um problema, adaptação ou necessidade de melhoria [Sousa, Oliveira e Anquetil, 2005].

A grande maioria dos sistemas, após sua implantação, sofre alterações durante todo seu ciclo de vida, especialmente os sistemas que suportam a área fim de uma empresa. Estes são os que mais sofrem manutenção, em virtude de que sua substituição implica normalmente em um risco maior e um maior custo para a empresa. Por isso, as empresas optam por realizar manutenções de seus sistemas mais críticos em vez de simplesmente substituí-los.

Em geral, os sistemas que sofrem manutenção têm seus requisitos originais alterados para atender uma nova legislação, pelo surgimento de novas funcionalidades, pela mudança de *hardware* ou do ambiente operacional, pela correção de erros ou pela prevenção de possíveis falhas.

Segundo literatura especializada, as empresas despendem um tempo bem maior no esforço de manutenção de seus aplicativos do que no desenvolvimento de novos sistemas. Isto implica que também o custo de manutenção desses aplicativos é muito superior ao custo do desenvolvimento de novos sistemas. Assim sendo, existe uma grande necessidade por técnicas de estimativas para projetos de manutenção de

software, para que essas empresas possam gerenciar adequadamente esse tipo de projeto de *software*.

Embora as atividades de manutenção sejam críticas e duradouras ao longo do ciclo de vida de um sistema, ainda não há estudos conclusivos e largamente utilizados para se estimar o esforço despendido com essas atividades.

A realização de estimativas é primordial nas questões que dizem respeito aos negócios de uma empresa, sendo o suporte para o planejamento e o gerenciamento de um projeto. Na elaboração de uma proposta de negócio entre cliente e fornecedor, as estimativas de custo e prazo devem ser conhecidas, para que se faça a análise de uma proposta viável ou não para ambos. Freqüentemente, nos contratos firmados são incorporadas e fixadas estimativas de custo e prazo. Como resultado disto, o fornecedor precisa ter estimativas precisas, pois um projeto subestimado poderá prejudicá-lo, visto que o cliente pagará apenas o custo estipulado no contrato assinado, e não sobre valores que não tinham sido previstos depois do fechamento do contrato [Monteiro *et al*., 2005].

A questão das estimativas nos remete ao dia-a-dia das empresas. Elas constroem *software* para vender, outras consomem estes aplicativos ou desenvolvem suas próprias soluções. Essas empresas têm a necessidade de verificar qual o tamanho do *software* que irão comercializar, consumir ou desenvolver. Através do tamanho pode-se determinar o esforço, o tempo a ser despendido, e o custo do aplicativo. Assim, poderão planejar melhor o projeto, gerenciar possíveis contratos envolvidos no processo e até ter a possibilidade de avaliar a produtividade das equipes do projeto.

Determinar o tamanho de um projeto de *software* torna-se então uma das principais atividades relacionadas ao processo de desenvolvimento de *software*. O tamanho do *software* significa a quantidade de trabalho a ser executado no desenvolvimento de um projeto em uma unidade de medida especificada. Cada projeto de *software* pode ser estimado segundo seu tamanho físico (que podem ser medidos através de especificação de requisitos, análise, projeto e código), baseado em funções que o usuário obtém, na complexidade do problema que o *software* irá resolver, e no reuso de projetos (que estima o quanto o projeto será reusado ou modificado a partir de um outro existente) (Fenton *et al*., 1997; McPhee, 1999).

A existência de técnicas que possam estimar com simplicidade e precisão o tamanho, esforço, prazo e custo dos projetos relacionados à manutenção de *software*, surge como de vital importância no cenário das empresas que desenvolvem, consomem ou constroem seus próprios aplicativos. Entretanto, a maioria das técnicas existentes para estimativas de projetos é mais aplicável a projetos de desenvolvimento de *software*, havendo ainda lacunas a serem preenchidas no que se refere às estimativas de manutenção de *software*.

A seguir, será apresentada a técnica TUCP (*Technical Use Case Point*) para estimativas de *software*, que serviu de base para a proposta de estimativas de manutenção de *software*.

3. Pontos de Caso de Uso Técnicos (TUCP)

A TUCP (Pontos de Caso de Uso Técnicos) é uma extensão da UCP (Pontos de Caso de Uso), proposta por Monteiro e*t al*. (2005), objetivando um cálculo mais acurado para as estimativas a partir de casos de uso. A utilização dessa técnica proporciona também

uma visão mais detalhada de estimativas para as principais etapas do ciclo de vida do *software*, possibilitando um acompanhamento mais efetivo do projeto. Toda esta seção está baseada em [Monteiro *et al*., 2005].

A TUCP congrega as seguintes questões:

- Elabora um *Guia para a elaboração de casos de uso*, uma vez que casos de uso bem especificados podem influenciar fortemente no tamanho do projeto.
- Conceitua transação no contexto de casos de uso, por ser utilizada como unidade de medida para o cálculo do tamanho dos casos de uso.
- Amplia a contagem de pontos de casos de uso *complexos*, inserindo os chamados pontos de caso de uso *n-complexos*.
- Desatrela os Fatores técnicos de ambiente (EF) do cálculo do tamanho, uma vez que o tamanho é uma grandeza física e não deve ter seu valor alterado em função de fatores ambientais.
- Considera os Fatores ambientais EFs apenas no cálculo do esforço, juntamente com o fator de produtividade (de forma semelhante a UCP).
- Granulariza o cálculo do esforço em etapas do processo de desenvolvimento e por caso de uso.

O grau de precisão da TUCP depende diretamente de a organização possuir um modelo que seja utilizado para a especificação de casos de uso, como também é dependente do entendimento claro do que seja o conceito de transação. Estas questões influenciam diretamente no cálculo do tamanho do projeto.

Uma transação é cada passo dos fluxos de eventos (básico e ou alternativo) de um caso de uso, onde ocorra um evento entre um ator e o sistema, e que deva ser executado por completo ou ainda a realização de algum processamento complexo nesses fluxos de eventos.

A seguir, serão apresentadas as estimativas de tamanho e esforço, segundo a TUCP.

3.1 Estimativa de Tamanho

A estimava de tamanho da TUCP envolve as seguintes etapas:

i. Contagem dos atores (UAW);

ii. Contagem dos casos de uso (TUUCW);

iii. Cálculo dos pontos de casos de uso não ajustados (TUUCP);

iv. Cálculo dos fatores de complexidade técnica (TCF);

v. Cálculo dos pontos de caso de uso técnicos (TUCP).

Contagem dos Atores

O peso total dos atores do sistema é calculado pela soma dos produtos dos atores de cada tipo pelo respectivo peso, UAW (*Unadjusted Actor Weight*), de forma semelhante a UCP [Karner, 1993], segundo a Tabela 1

Tabela 1. Classificação dos atores

Tipo	Descrição	Peso
Simples	Aplicação com APIs definidas	1
Médio	Aplicação com interface baseada em protocolo ou interação de usuário baseado em linhas de comandos	2
Complexo	Interação de usuário através de interface gráfica ou página Web	3

Contagem dos Casos de Uso

O peso total dos casos de uso do sistema é calculado a partir das Tabela 2, isto é, o TUUCW (*Technical Unadjusted Use Case Points Weight*).

Tabela 2. Contagem dos casos de uso

Tipo	Descrição	Peso
simples	até 3 transações incluindo os passos alternativos	5
médio	de 4 a 7 transações incluindo os passos alternativos	10
complexo	de 8 a *t* transações incluindo os passos alternativos	15
n-complexo	acima de *t* transações	P_X

Os casos de uso com até *t* transações serão calculados de forma semelhante a UCP [Karner, 1993]. Acima de *t* transações, o tipo de caso de uso será chamado de *n*-complexo, com o cálculo de seu peso (P_X) apresentado nas equações abaixo.

$$\text{TUUCW} = 15n + p$$

$$n = T / t$$

Nas equações acima, T = número de transações do caso de uso, e p = o peso obtido, quando o resto (r) da divisão de T / t é aplicado ao peso original (*simples*, *médio*, e *complexo*) (Tabela 2). Assim sendo, se $r = 0$, $p = 0$; se $r \in [1, 3]$, $p = 5$; se $r \in [4, 7]$, $p = 10$; e se $r \in [8, (t - 1)]$, $p = 15$.

A definição do valor de *t* poderá depender das características da organização, ou até mesmo das características de um dado tipo de projeto. Neste trabalho, foi utilizado $t = 11$ pelas mesmas razões apresentadas em Monteiro e*t al*. (2005).

Cálculo dos Pontos de Casos de Uso não Ajustados

O cálculo dos pontos de caso de uso técnicos não ajustados, TUUCP (*Technical Unadjusted Use Case Points*), é efetuado pela equação abaixo.

$$\text{TUUCP} = \sum \text{UAW} + \sum \text{TUUCW}$$

Cálculo dos Fatores de Complexidade Técnica

Os 13 fatores de complexidade técnica (TCF) são calculados de forma similar aos apresentados pela técnica UCP [Karner, 1993]. O TCF é obtido através da equação abaixo, em que TFactor corresponde ao somatório dos produtos entre o peso e a nota atribuída de cada um dos 13 fatores de complexidade técnica.

$$\text{TCF} = 0{,}6 + (0{,}01 * \text{TFactor})$$

Cálculo dos Pontos de Caso de Uso Técnicos

O cálculo da TUCP (*Technical Use Case Points*) ajustada é dado pela equação abaixo.

$$TUCP = TUUCP * TCF$$

3.2 Estimativa de Esforço

A estimava do esforço da TUCP envolve as seguintes etapas:

i. Cálculo dos fatores de ambiente (EF);

ii. Cálculo da produtividade (PROD);

iii. Cálculo do esforço técnico (Esforço).

Cálculo dos Fatores Ambientais

Os 8 fatores ambientais (EF) são calculados de forma similar aos apresentados pela técnica UCP [Karner, 1993]. Da mesma maneira que a UCP, os fatores ambientais dizem respeito aos requisitos não-funcionais associados ao projeto, tais como experiência da equipe, estabilidade do projeto e motivação dos programadores.

O fator ambiental (EF) é calculado pela equação abaixo, onde o EFator é o somatório dos produtos entre o peso e a nota atribuída de cada um dos 8 fatores ambientais.

$$EF = 1{,}4 + (-0{,}03 * EFator)$$

Cálculo da Produtividade

O fator de produtividade pode ser calibrado de acordo com a produtividade da equipe por tipos de atividade de um projeto. Esse fator pode ser obtido através de uma base histórica organizacional, onde podem ser armazenados dados, como, por exemplo, o fator de produtividade para cada etapa do ciclo de vida e características de projeto.

Com base na estimativa de tamanho, Karner (1993) propôs uma produtividade de 20 homens/hora (h.h) por UCP para projetos onde a equipe seja considerada estável e experiente, e 28 h.h por UCP para projetos em que os requisitos não sejam estáveis e com uma equipe não experiente.

Anda *et al*. (2001) mostraram que este esforço pode variar entre 15 h.h por UCP para projetos onde a equipe seja considerada estável e experiente, e 30 h.h por UCP para projetos em que os requisitos não sejam estáveis e com uma equipe não experiente.

Cálculo de Esforço do Projeto

A estimativa de esforço total para o projeto pode ser obtida a partir da equação abaixo, de forma semelhante a UCP.

$$Esforço = TUCP * EF * PROD$$

3.3 Estimativa de Prazo e Custo

As estimativas de prazo e custos podem ser obtidas a partir do esforço estimado da TUCP, disponibilidade de recursos, restrições do projeto, entre outras questões.

O cálculo do custo apenas está relacionado com o custo de esforço referente aos casos de usos do projeto Os gastos com gestão de projeto, gestão de configuração,

atividades de qualidade e despesas indiretas do projeto, entre outros custos, não estão sendo considerados.

4. Estimativas de Manutenção de *Software* a partir de Casos de Uso

A proposta de estimativas de manutenção de *software* a partir de casos de uso está baseada na TUCP (*Technical Use Case Point*), que originalmente foi concebida para tratar estimativas de caso de uso para projetos de desenvolvimento de *software*. A esta proposta, chamaremos de TUCP para manutenção de *software* ou TUCP-M.

A TUCP-M propõe, em linhas gerais, observar os seguintes itens:

- A elaboração de um documento de Solicitação de Mudanças deverá ser a fonte inicial para o processo de manutenção. Nesse documento, deverão ser identificadas e detalhadas as mudanças solicitadas.
- Os requisitos afetados com a solicitação de mudanças devem ser a base para a manutenção.
- A partir dos requisitos afetados devem ser identificados os casos de uso que serão afetados. Isto poderá ser feito, por exemplo, por meio de uma matriz de rastreabilidade do projeto, envolvendo requisitos e casos de uso.
- Para cada caso de uso afetado, devem ser identificados os passos do fluxo de eventos que foram mantidos no documento de Especificação de Caso de Uso. Recomenda-se o *template* desse documento proposto por Monteiro *et al*. (2005). Esses passos podem ser do tipo: *incluído*, *alterado* ou *excluído*.
- Efetuar apenas a contagem das transações dos passos dos fluxos de eventos dos casos de uso que foram mantidos, isto é, dos passos que foram incluídos, alterados ou excluídos.
- A partir do total das transações contabilizadas para a manutenção dos casos de uso, efetua-se o cálculo da estimativa de tamanho da manutenção e, em seguida, do esforço, prazo e custo, segundo a TUCP.

Quando uma manutenção de *software* é solicitada, o primeiro entendimento que se deve ter é quais requisitos novos estão sendo solicitados, quais deverão sofrer mudanças e quais deverão ser excluídos.

4.1 Calculando o Tamanho da Manutenção

Para calcular o tamanho da manutenção deve-se levar em consideração o total das transações calculadas a partir dos passos dos fluxos de eventos dos casos de uso que sofreram manutenção:

- *Passos incluídos*: são os novos passos inseridos no fluxo de eventos do caso de uso. Isto, provavelmente, conduzirá a um incremento da quantidade de transações do caso de uso (há passos que não possuem transações). Nesta situação, o cálculo do tamanho é o mesmo que o utilizado em um projeto de desenvolvimento de *software* estabelecido pela TUCP (seção 3.1).
- *Passos alterados*: são os passos que sofreram alguma alteração em sua estrutura, podendo haver alguma mudança na quantidade de suas transações.

Entretanto, o que deve ser avaliado é o tamanho necessário para realizar essa alteração. Mesmo que um caso de uso, ao ser alterado, chegue a permanecer com o mesmo número de transações (portanto, com o mesmo tamanho), o que deve ser considerado é o "esforço" para realizar as mudanças. Assim sendo, esse "esforço" de manutenção poderia ser de *n* transações, que seriam consideradas para o tamanho da manutenção.

- *Passos excluídos*: são os passos que tiveram sua descrição retirada do fluxo de evento. Isto, provavelmente, acarretará em uma diminuição da quantidade de transações do caso de uso. Uma vez mais, o que deve ser apurado é a quantidade de transações (o tamanho) da adaptação e seu conseqüente "esforço".

Finalizada a apuração da quantidade de transações requeridas para a inclusão (t_i), alteração (t_a), ou exclusão (t_e) de passos do fluxo de eventos de cada caso de uso, pode-se obter o tamanho da manutenção do projeto (*TUCP-M*$_{projeto}$). Assim sendo, tem-se que a equação abaixo.

$$\textbf{\textit{TUCP-M}}_{projeto} = \Sigma\, t_i + \Sigma\, t_a + \Sigma\, t_e$$

Em um processo de manutenção, considerando as etapas do ciclo de vida, o tamanho da manutenção de um projeto (*TUCP-M*$_{projeto}$) corresponde ao total do tamanho da manutenção realizada em cada uma de suas etapas (*E*), como também ao total do tamanho da manutenção de cada um dos casos de uso (*UC*), conforme equações abaixo.

$$\textbf{\textit{TUCP-M}}_{projeto} = \textbf{\textit{TUCP-M}}_{E1} + \textbf{\textit{TUCP-M}}_{E2} + \textbf{\textit{TUCP-M}}_{E3} + \ldots + \textbf{\textit{TUCP-M}}_{En}$$

$$\textbf{\textit{TUCP-M}}_{projeto} = \Sigma\, \textbf{\textit{TUCP-M}}_{E}$$

$$\textbf{\textit{TUCP-M}}_{projeto} = \textbf{\textit{TUCP-M}}_{UC1} + \textbf{\textit{TUCP-M}}_{UC2} + \textbf{\textit{TUCP-M}}_{UC3} + \ldots + \textbf{\textit{TUCP-M}}_{UCn}$$

$$\textbf{\textit{TUCP-M}}_{projeto} = \Sigma\, \textbf{\textit{TUCP-M}}_{UC}$$

De forma semelhante, o esforço de manutenção do projeto (*Esforço-M*$_{projeto}$) corresponde ao esforço total realizado em suas *n* etapas do ciclo de vida do projeto, como também ao esforço total de todos os seus casos de uso. Portanto, podem-se ter as equações abaixo.

$$\textbf{\textit{Esforço-M}}_{projeto} = \textbf{\textit{Esforço-M}}_{E1} + \textbf{\textit{Esforço-M}}_{E2} + \textbf{\textit{Esforço-M}}_{E3} + \ldots + \textbf{\textit{Esforço-M}}_{En}$$

$$\textbf{\textit{Esforço-M}}_{projeto} = \Sigma\, \textbf{\textit{Esforço-M}}_{E}$$

$$\textbf{\textit{Esforço-M}}_{projeto} = \textbf{\textit{Esforço-M}}_{UC1} + \textbf{\textit{Esforço-M}}_{UC2} + \textbf{\textit{Esforço-M}}_{UC3} + \ldots + \textbf{\textit{Esforço-M}}_{UCn}$$

$$\textbf{\textit{Esforço-M}}_{projeto} = \Sigma\, \textbf{\textit{Esforço-M}}_{UC}$$

Segundo Monteiro (2005), podem-se levantar percentuais de esforço por etapa de ciclo de vida (μ_E) que, em geral, seguem uma média histórica própria da organização. Dependendo das características de cada projeto, alterações nesses percentuais podem ser realizadas durante as estimativas. Vale salientar que $\Sigma\mu_E = 1$. Assim sendo, tem-se a equação seguir (para manutenção evolutiva, por exemplo).

$$\textbf{\textit{Esforço-M}}_{E} = \mu_E\, .\textbf{\textit{Esforço-M}}_{projeto}$$

As equações a seguir apresentam o esforço de manutenção para um caso de uso, considerando as etapas de seu ciclo de vida.

$$\textbf{Esforço-}M_{UC} = \mu_{E1}.\textbf{Esforço-}M_{UC} + \mu_{E2}.\textbf{Esforço-}M_{UC} + \ldots + \mu_{En}.\textbf{Esforço-}M_{UC}$$

$$\textbf{Esforço-}M_{UC} = \textbf{Esforço-}M_{UC\text{-}E1} + \textbf{Esforço-}M_{UC\text{-}E2} + \ldots + \textbf{Esforço-}M_{UC\text{-}En}$$

$$\textbf{Esforço-}M_{UC} = \Sigma\ \textbf{Esforço-}M_{UC\text{-}E}$$

Em um processo de manutenção, etapas do ciclo de vida do projeto podem ser parcialmente executadas (manutenções corretivas, por exemplo). Desta forma, o esforço de manutenção de cada etapa do ciclo de vida do projeto, para cada caso de uso, pode ser acrescido de um fator de manutenção ($f_{E\text{-}UC}$), pertencente ao intervalo [0, 1]. Neste caso, quando $f_{E\text{-}UC} = 0$, o esforço de manutenção da etapa considerada para o caso de uso em questão é nulo e, quando $f_{E\text{-}UC} = 1$, o esforço de manutenção é de 100%. O novo esforço de manutenção do caso de uso (*Esforço-*M_{UC}´) é dado pela equação abaixo.

$$\textbf{Esforço-}M_{UC}' = f_{E1\text{-}UC}(\mu_{E1}.\textbf{Esforço-}M_{UC}) + f_{E2\text{-}UC}(\mu_{E2}.\textbf{Esforço-}M_{UC}) + \ldots + f_{En\text{-}UC}(\mu_{En}.\textbf{Esforço-}M_{UC})$$

$$\textbf{Esforço-}M_{UC}' = \Sigma\ (f_{E\text{-}UC}(\mu_{E}.\textbf{Esforço-}M_{UC}))$$

O novo esforço de manutenção por etapa do ciclo de vida (*Esforço-*M_{E}´) é dado pelas equações abaixo.

$$\textbf{Esforço-}M_{E}' = \mu_{E}\ (f_{E\text{-}UC1}(\textbf{Esforço-}M_{UC1}) + f_{E\text{-}UC2}(\textbf{Esforço-}M_{UC2}) + \ldots + f_{E\text{-}UCn}(\textbf{Esforço-}M_{UCn}))$$

$$\textbf{Esforço-}M_{E}' = \mu_{E}\ \Sigma\ (f_{E\text{-}UC}(\textbf{Esforço-}M_{UC}))$$

O novo esforço de manutenção de um projeto de software, que utiliza casos de uso, poderá ser expresso pelas equações abaixo.

$$\textbf{Esforço-}M_{projeto} = \Sigma\ \textbf{Esforço-}M_{UC}'$$

$$\textbf{Esforço-}M_{projeto} = \Sigma\ \textbf{Esforço-}M_{E}'$$

A seguir será apresentada uma aplicação das estimativas de manutenção de software a partir das equações proposta nesta seção.

4.3 Aplicação das Estimativas de Manutenção de *Software*

Apresenta-se nesta seção, uma aplicação de estimativas de manutenção de *software* em um projeto real. O projeto é de médio porte e foi desenvolvido para a área financeira de uma empresa em 2004. Participaram da equipe de desenvolvimento um analista de sistema, um arquiteto de *software*, três desenvolvedores em .net e um desenvolvedor Cobol. O tempo de duração desse projeto foi de sete meses.

O esforço calculado para este projeto em manutenção (quatro casos de usos afetados) foi de 192,52 h.h. A equipe de manutenção é composta por um analista de sistema, um arquiteto de *software*, um desenvolvedor .net, e um desenvolvedor Cobol.

Após o preenchimento do Documento de Solicitação de Mudanças pelo cliente, constatou-se que seriam incluídos alguns requisitos e alterados outros requisitos do sistema. Após isto, partiu-se para a identificação dos casos de uso afetados com esta manutenção.

Identificados os casos de usos afetados, o passo seguinte foi identificar os passos dos fluxos de eventos de cada caso de uso, que sofreram inclusão, alteração ou e/ou exclusão. A Tabela 3 apresenta um dos casos de uso afetados nessa manutenção (UC01). São apresentados apenas os passos do caso de uso que tinham uma ou mais transações (existentes). O "estado" corresponde às ações de manutenção de incluir, alterar ou excluir um passo de seu fluxo de evento (FB = Fluxo Básico; A = Fluxo Alternativo; E = Fluxo de Exceção).

O caso de uso UC01 possuía 17 transações. Ao sofrer manutenção, considerando-se apenas seus passos inseridos e alterados, chegou-se a uma contagem de apenas 3 transações. Isto implica que o tamanho a ser mensurado para este caso de uso levará em conta apenas 3 transações e não as 17 transações.

Aplicando-se a TUCP para o cálculo do esforço deste caso de uso (seção 3) tem-se um esforço de 48,13 h.h.

Tabela 3. Manutenção do caso de uso UC01

Passo Original	Passo Atual	Transações Existentes	Transações de Manutenção	Estado
FB.4	FB.4	0	1	Alterado
	FB.4c	0	0	Incluído
FB.5	FB.5	3	1	Alterado
	FB.6	0	0	Incluído
FB.6.a	FB.7.a	1	-	-
FB.11	FB.11	2	-	-
A.1.4	A.1.4	2	0	Alterado
A.1.4.a	A.1.4.a	1	-	-
	A.2.2	0	1	Incluído
A.2.2.a	A.2.3.a	1	-	-
A.3.5	A.3.5	1	-	-
A.9.4	A.9.4	4	0	Alterado
A.9.4.a	A.9.4.a	1	0	-
E.4	E.4	1	-	-
Total		**17**	**3**	

A complexidade da manutenção do quatro casos de uso deste projeto foi calculada como "simples", isto é, foram mantidas até três transações em cada um desses casos de uso.

Para este projeto de manutenção foram consideradas as quatros etapa a seguintes para seu ciclo de vida (*E*): Requisitos (*Req*), Análise & Projeto (*A&P*), Codificação (*Cod*), e Teste (*Tst*).

A partir das equações da seção 4.2, foi construída a Tabela 4. Nesta aplicação, a manutenção para o projeto de software em questão era evolutiva. Portanto, todos os

valores de $f_{E\text{-}UC}$ foram iguais a 100%. Portanto, o esforço de manutenção de um projeto poderá ser expresso pelas equações abaixo:

Tabela 4. Esforço de manutenção dos casos de uso do projeto

Caos de Uso	Fator de manutenção por Etapa do Ciclo de Vida				Esforço de Manutenção
	f_{Req}	$f_{A\&P}$	f_{Cod}	f_{Tst}	***Esforço-M***$_{UC}$
UC01	100%	100%	100%	100%	48,13 h.h
UC02	100%	100%	100%	100%	48,13 h.h
UC03	100%	100%	100%	100%	48,13 h.h
UC04	100%	100%	100%	100%	48,13 h.h
Esforço do Projeto de Manutenção					**192,52 h.h**

Para a organização, em que foi realizada esta aplicação, foram considerados os seguintes percentuais de esforço por etapa de ciclo de vida (μ_E), utilizados em Monteiro (2005):

- μ_{Req} = 0,20
- μ_{Req} = 0,25
- μ_{Req} = 0,45
- μ_{Req} = 0,10

A Tabela 5 apresenta o esforço de manutenção por etapa do ciclo de vida do projeto em questão, considerando os valores de μ_E acima.

Tabela 5. Esforço de manutenção por etapa do ciclo de vida

Casos de Uso	Esforço de manutenção por Etapa do Ciclo de Vida				Esforço de Manutenção
	Esforço$_{Req}$	*Esforço*$_{A\&P}$	*Esforço*$_{Cod}$	*Esforço*$_{Tst}$	*Esforço-M*$_{UC}$
UC01	9,63 h.h	12,03 h.h	21,66 h.h	4,81 h.h	48,13 h.h
UC02	9,63 h.h	12,03 h.h	21,66 h.h	4,81 h.h	48,13 h.h
UC03	9,63 h.h	12,03 h.h	21,66 h.h	4,81 h.h	48,13 h.h
UC04	9,63 h.h	12,03 h.h	21,66 h.h	4,81 h.h	48,13 h.h
Esforço-M$_E$	38,52 h.h	48,12 h.h	21,66 h.h	86,64 h.h	
Esforço do Projeto de Manutenção					**192,52 h.h**

Após finalizado o projeto, foi verificado que seu esforço real foi de 231,00 h.h. A diferença entre o esforço estimado calculado utilizando-se a TUCP-M e o esforço real foi de 19,9%. Neste caso, o esforço de manutenção foi subestimado em torno de 20% do valor real. Vale salientar que, para esse projeto de manutenção, foi alocado inicialmente um analista de sistema e um programador. Posteriormente, esse analista entrou de férias e foi substituído por um outro analista. Após este outro analista já ter iniciado suas atividades neste projeto de manutenção, ele precisou ser realocado a um segundo projeto. Assim, um terceiro analista assumiu esse projeto de manutenção. Isto pode vir a justificar que o esforço real gasto para a manutenção tenha sido maior que o esforço estimado.

A seguir serão apresentadas as conclusões deste trabalho.

5. Conclusão

As estimativas são parte essencial em um projeto de desenvolvimento de *software*, em especial em um plano de projeto. Quanto mais realistas forem essas estimativas, mais contribuirão para o sucesso do projeto.

A maior parte dos projetos de *software* das empresas referem-se à manutenção. Apesar disso, as técnicas conhecidas para estimar o tamanho e esforço de projetos são mais direcionadas a novos projetos de *software*. No caso de projetos de manutenção, ainda há carências dessas técnicas.

O trabalho apresentado propôs uma forma de mensurar projetos de manutenção de *software*, que utilizam casos de uso com uma maior precisão a partir da TUCP (Pontos de caso de uso técnicos). O trabalho apresentado leva em consideração apenas os casos de uso afetados com a manutenção e os passos do fluxo de eventos que foram incluídos, alterados ou excluídos em virtude dessa manutenção. Desta forma, a estimativa de esforço realizada no projeto deste estudo de caso foi bem mais acurada, proporcionando assim, decisões mais apropriadas e planejamento mais realista.

Apesar de o resultado da aplicação com a TUCP-M ter sido considerado na organização como satisfatório (um erro em torno de 20%), esta técnica estará sendo utilizadas em um grande conjunto de novos projetos de manutenção, que utilizam casos de uso, para validá-la e aprimorá-la.

Referências

April, A., Abran, A, Industrial Research in Softwre Maintenance: Development of Productiviy Models, Guide Summer ’95 Conference and Solutions Fair, Boston, 1995.

Bourque P., Maya M., Abran A., A Sizing Measure for Adaptative Maintenance Work Products, IFPUG Spring Conference, Atlanta, April, 1996.

Fenton, N., Neil, M.; Pfleeger, S., (1997) “Software metrics: a rigorous & practical approach”. Boston: PWS Publishing Company.

FPCPM (1999) Function Point Counting Practices Manual, Version 4.1

Karner, G. Metrics for Objectory. (1993) Diploma thesis, University of Linköping, Sweden. No. LiTH-IDA-Ex-9344:21.

McPhee, C. S. (1999) “Software process management: software size estimation”. University of Calgary. Disponível em http://sern.ucalgary.ca/~cmcphee/ SENG621/ Software_Size_Estimation.html.

Monteiro, T. C., Pires, C. G. S., Belchior, A. D., (2005) “TUCP: Uma Extensão da Técnica UCP”, IV Simpósio Brasileiro de Qualidade de Software, Porto Alegre.

NBR ISO/IEC 12207, (1998) Tecnologia da Informação: processos de ciclo de vida de software, ABNT, Rio de Janeiro.

Pigosky, Thomas, (1996) “Pratical Software Maintenance – Best Pratices for Managing Your Software Investiments”, WILEY Computer Publishing, Nova York.

Ramil, J. F., Lehman, M. M., (2000) “Cost Estimation and Evolvability Monitoring for Software Evolution Process”, Workshop on Empirical Studies of Software Maintenance, San Jose, CA, USA.

Ross, M. (2004) “Size does Matter: Continuous Size Estimating and Tracking”. Quantitative Software Management. Disponível em http://www.qsm.com.

SEI (Software Engineering Institute). 2002 “CMMI-SW for Systems Engineering/Software Engineering”.Version 1.1 - CMU/SEI-2002-TR-012. Disponível em: http://www.sei.cmu.edu/publications/documents/02.reports/02tr002.html.

Sousa, K. D., Oliveira, K. M., Anquetil, N., (2005) “Uso do GQM para avaliar implantação de processo de manutenção de software”, IV Simpósio Brasileiro de Qualidade de Software, Porto Alegre.

Webster K. P.B., Marçal de Oliveira K., Anquetil N., (2004) "Priorização de Riscos para Manutenção de Software" (short paper), *Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento, 2004*, Madrid, Espanha.

RedoX-UML: Redocumentação de Aplicações Legadas COBOL Usando XML e UML

Jesuíno José de Freitas Neto, Nabor das Chagas Mendonça

Mestrado em Informática Aplicada – Universidade de Fortaleza
Av. Washington Soares, 1321 – CEP 60811-905 Fortaleza – CE
jesuino@gmail.com, nabor@unifor.br

Abstract. *According to recent data, there are a great number of COBOL applications still in operation in many parts of the world, especially in financial institutions. Most of those applications present serious documentation problems, which complicates from their maintenance to their integration to modern development platforms. This paper presents an environment for the re-documentation of COBOL applications using UML 2.0. This environment, named RedoX-UML, uses XML as an intermediary representation for COBOL source code, and open XML-based data manipulation technologies to reverse engineer a subset of UML 2.0 elements from the syntactic structure of a COBOL program represented in XML. In this way, the process offers a flexible and low-cost alternative to bridge the huge gap between legacy technologies and modern development platforms, thus facilitating their co-existence and integration.*

Resumo. *Segundo dados recentes, há um número elevado de aplicações Cobol ainda em operação em muitas partes do mundo, especialmente em grandes instituições financeiras. Boa parte dessas aplicações apresenta graves deficiências quanto à documentação, o que dificulta desde a sua manutenção até a sua integração com outras plataformas mais modernas de desenvolvimento. Este artigo apresenta um ambiente para a redocumentação de aplicações legadas Cobol, utilizando a UML 2.0. Este ambiente, denominado RedoX-UML, utiliza XML como representação intermediária do código fonte em Cobol, e tecnologias abertas de manipulação de dados XML para realizar a engenharia reversa de um subconjunto dos elementos da UML 2.0 a partir da estrutura sintática de um programa Cobol representada em XML. Dessa forma, o processo oferece uma alternativa flexível e de baixo custo para reduzir a enorme lacuna existente entre as tecnologias legadas e as novas plataformas de desenvolvimento, facilitando a sua convivência e integração.*

1. Introdução

COBOL é uma linguagem de terceira geração que teve sua primeira versão lançada em 1959 e é também um acrônimo para ***CO****mmon* ***B****usiness* ***O****riented* ***L****anguage.* Além disso, representa um dos melhores exemplos acerca do conceito de sistema legado,

posto que um grande número de sistemas foi desenvolvido nesta linguagem a partir da década de 60 e continuam em operação até hoje em diversas partes do mundo, atendendo a uma gama enorme de domínios de aplicação. São várias as razões para que esses sistemas continuem em operação, apesar da sua inevitável defasagem tecnológica: o alto custo de desenvolver um novo sistema que atenda de forma satisfatória o mesmo conjunto de requisitos; o valor contido em seu código na forma de regras de negócio ainda válidas; e muitas vezes a impossibilidade de migração de plataforma em função do alto poder de processamento e da alta confiabilidade oferecidos pelos computadores de grande porte. Um típico exemplo são os sistemas mantidos por décadas em instituições financeiras que processam um grande volume de dados, como bancos e seguradoras.

De acordo com um relatório do Gartner Group[1] publicado em 1999, em 1997 havia 300 bilhões de linhas de código em uso no mundo. Deste total cerca de 80% (240 bilhões de linhas) estavam escritas em Cobol. A redução anual deste montante é da ordem de dezenas de milhares, o que indica que por muitos anos ainda existirá um grande acervo de aplicações que deverá ser mantido. Por outro lado, a oferta de mão de obra disponível para plataformas Cobol não tem apresentado crescimento, e os processos de migração e conversão não indicam que o acervo total de aplicações esteja sendo reduzido na mesma proporção em que a mão de obra disponível decai. Esse contraponto poderá levar a uma situação crítica onde teremos um grande número de sistemas escritos em uma linguagem em que um contingente cada vez menor de profissionais tem proficiência, mas que, por outro lado, é responsável por uma parte significativas dos negócios das grandes empresas. Essa é, com certeza, uma visão preocupante para dirigentes de TI de empresas que têm de empreender esforços no sentido de habilitar novas equipes a compreender e evoluir seus acervos legados.

Este trabalho apresenta um ambiente, que compreende um processo e uma ferramenta, denominado RedoX-UML, destinado a auxiliar na redocumentação de aplicações legadas Cobol, utilizando como notação a UML 2.0 [OMG 2006a]. Utiliza-se a linguagem de marcação XML [W3C 2004] como representação intermediária do código fonte em Cobol, e tecnologias abertas de manipulação de dados XML para realizar a engenharia reversa do código Cobol para elementos da UML 2.0. O uso de XML traz alguns benefícios para o processo de redocumentação, entre eles a estruturação explícita do código no documento XML, facilitando a sua manipulação, e a possibilidade de reutilizar a abundância de tecnologias de manipulação de dados XML publicamente disponíveis. O objetivo do presente trabalho é prover uma alternativa flexível e de baixo custo para reduzir a lacuna existente entre as tecnologias legadas e as novas plataformas de desenvolvimento, facilitando a sua convivência e integração.

O restante do artigo está organizado da seguinte forma: na Seção 2 discutimos sobre o uso de XML na manutenção de software; na Seção 3 descrevemos RefaX, um arcabouço para refatoração de código baseado em XML e que serviu de base para o RedoX-UML, descrito em detalhes na Seção 4; em seguida, na Seção 5, comparamos o ambiente ora proposto com alguns trabalhos relacionados; e, finalmente, na Seção 6, concluímos o artigo com um sumário dos principais resultados e indicações de trabalhos futuros.

[1] Brown, Gary De Ward – COBOL: The failure that wasn't – COBOLReport.com

2. Uso de XML na manutenção de software

Como forma de deixar o código fonte independente de estilo de programação e mais fácil de ser manipulado e analisado por diferentes ferramentas de manutenção, há várias propostas para adotar XML [W3C 2004] para a representação de artefatos de código. Por exemplo, Mamas e Kontogiannis (2000) descrevem três representações de código baseadas em XML: JavaML, CppML e OOML. As duas primeiras destinam-se a representar código Java e C++, respectivamente, enquanto a última generaliza as duas primeiras, preservando apenas os elementos de sintaxe comum entre elas. Badros (2000) propõe uma outra representação XML para código Java, também chamada de JavaML. Outros exemplos incluem srcML [Maletic *et al.* 2002] e GXL [Holt *et al.* 2000].

O principal fator negativo associado ao uso de XML como representação de código-fonte é a limitação de desempenho [Anderson 2005]. Por outro lado, há inúmeros fatores positivos a serem considerados: facilidade de navegação e manipulação, a partir da representação explícita da estrutura sintática do código na hierarquia de elementos do documento XML; capacidade de criação de consultas mais poderosas, utilizando linguagens e ferramentas de alto nível de abstração; possibilidade de incluir referências cruzadas diretamente entre os elementos do código; e amplo suporte ferramental, em função da abundância de tecnologias abertas atualmente disponíveis para a manipulação de dados XML.

Todas estas vantagens vêm colocando XML como uma opção bastante plausível para representação e manipulação de artefatos de código. Por outro lado, não temos conhecimento da existência de uma representação em XML específica para a linguagem Cobol, que certamente está entre as que possuem o maior acervo de programas em operação no mundo. Assim, um dos objetivos desse trabalho é definir uma representação baseada em XML para programas Cobol, e, dessa forma, estender os benefícios do uso de XML ao processo de redocumentação de aplicações escritas nessa linguagem.

3. O Arcabouço RefaX

RefaX [Mendonça *et al.* 2004] é um arcabouço para refatoração de código baseado em um processo centrado em XML, como mostra a Figura 1. O objetivo do arcabouço é facilitar o desenvolvimento de ferramentas de refatoração mais flexíveis. Para isso, RefaX segue um estilo de arquitetura em camadas, como mostra a Figura 2, para oferecer aos desenvolvedores de ferramentas de refatoração vários serviços:

- Conversão de artefatos de código para representações em XML;
- Armazenamento dos artefatos convertidos em um repositório XML;
- Verificação e execução das operações de refatorações sobre os dados do repositório;
- Conversão da representação atualizada de volta para seu formato textual original.

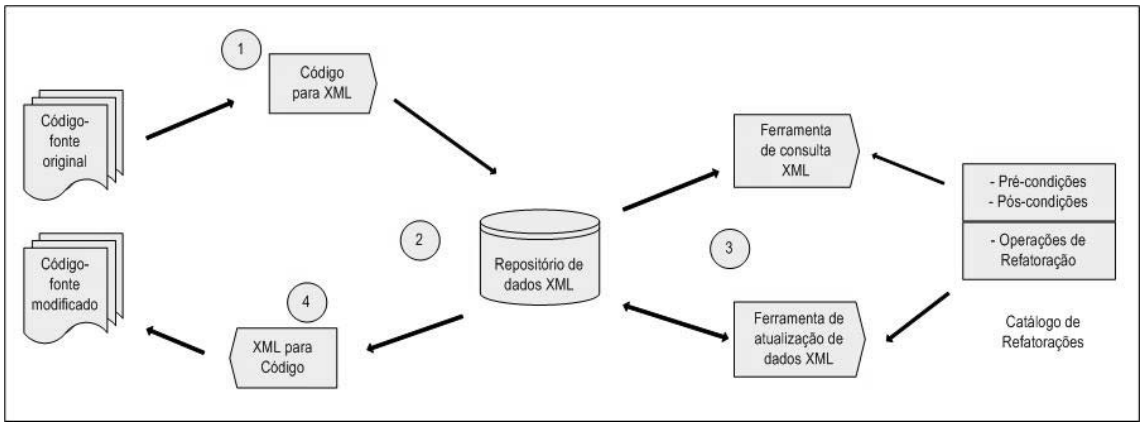


Figura 1 – Processo de refatoração com XML.

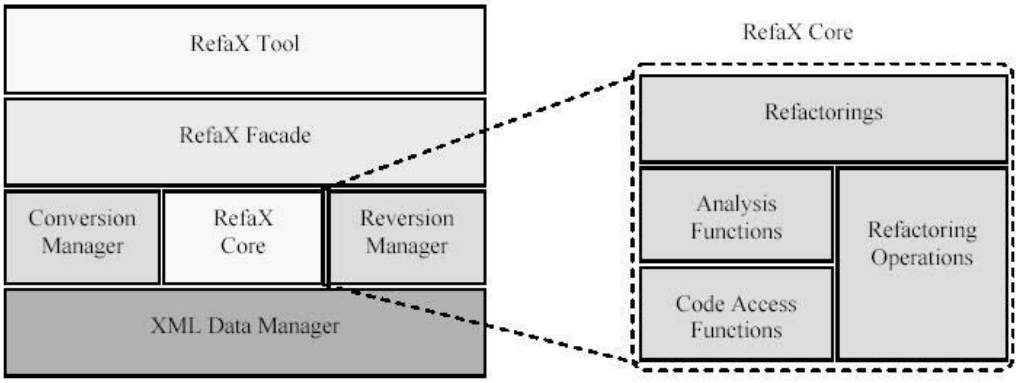


Figura 2 - Arquitetura do arcabouço RefaX.

O arcabouço RefaX continua em evolução através de duas grandes linhas de pesquisa. A primeira visa à incorporação de novas re-fatorações e a condução de experimentos de forma a avaliar a desempenho do produto e considerando as questões abordadas por Anderson (2005). A segunda, que é o foco deste trabalho, consiste na extensão do arcabouço para implementar um processo de redocumentação de aplicações legadas escritas na linguagem Cobol. Este ambiente, denominado RedoX-UML, é apresentado a seguir.

4. RedoX-UML: Um ambiente para redocumentação de aplicações legadas

Tendo como ponto de partida o arcabouço RefaX, esta seção apresenta o RedoX-UML, um ambiente para a redocumentação de aplicações legadas Cobol com uso da UML 2.0 e de representações de código em XML. A seguir apresentamos a motivação para a condução do projeto, bem como seus principais elementos, a arquitetura sendo desenvolvida para a sua implementação e as principais tecnologias utilizadas.

4.1. Motivação

A motivação principal deste projeto é reduzir os custos de manutenção de software do Banco ALFA[2], que será também objeto de estudo de caso para a validação do processo. A referida empresa compõe o segmento de bancos do mercado brasileiro e possui um grande número de aplicações legadas Cobol em constante evolução. Tais aplicações também necessitam conviver e integrar-se com diversas outras plataformas, dado o ambiente computacional heterogêneo existente. Abaixo, a Tabela 1, mostra um resumo do acervo de programas em linguagem Cobol do Banco ALFA.

Tabela 1 - Acervo de programas Cobol do Banco ALFA.

Versão do Cobol	Quantidade de programas
Cobol OS VS	7.354
VS Cobol II	4.160
Total	**11.514**

A metodologia de desenvolvimento de sistemas adotada na empresa é baseada no RUP, e já vêm sendo usada há alguns anos nos seus projetos de desenvolvimento e evolução de software. Um cenário bastante comum na realização desses projetos é a necessidade de atender os requisitos de integração entre plataformas, a qual tem criado situações de impasse para os profissionais responsáveis por levantamento de requisitos e especificação de casos de uso, que chamaremos de líderes de requisitos. Apesar de conseguirem progredir adequadamente para a especificação de componentes em plataformas OO, os líderes enfrentam dificuldades para produzir uma especificação que pudesse ser consumida pelos profissionais responsáveis pelos sistemas legados quando da produção de um componente de software na sua plataforma nativa.

A saída para esta situação foi manter a mesma metodologia para os sistemas legados apenas até o final da especificação de casos de uso, já que esses tipos de artefatos são independentes de linguagem ou plataforma e são feitos em texto plano. Ainda assim, havia a questão pendente sobre que artefatos deveriam ser produzidos para a plataforma legada, e que fossem equivalentes aos modelos de implementação nas plataformas OO. Ou seja, como realizar casos de uso para antigas plataformas de desenvolvimento?

Essas dificuldades levaram à implementação de um processo em que fosse possível unificar a linguagem dos líderes de requisitos e dos profissionais responsáveis pela implementação de componentes de software na plataforma legada. Esse requisito de comunicação no projeto implicou na capacitação dos profissionais da plataforma antiga em RUP e UML, e na adoção de alguns elementos da UML para orientar a implementação do projeto físico dos novos componentes de software das aplicações legadas. Os elementos comumente usados são:

- Diagrama de componentes, para representar a dependência entre programas com elementos externos tais como outros programas, objetos de bancos relacionais e arquivos do tipo *flat*, segundo uma visão física.

[2] Nome de fantasia usado propositadamente por questões de sigilo

- Diagramas de atividade para representar fluxos de programas ou parágrafos em aplicações *batch*;
- Diagramas de seqüência para representar fluxos de programas ou parágrafos em aplicações *on-line*;
- Diagramas de classe, para representar relacionamentos entre o programa e entidades externas como arquivos flat, objetos de banco de dados, segundo uma visão lógica.

Essa medida garantiu que novos projetos mantivessem um nível adequado de documentação, já que a mesma é feita de forma integrada ao processo de desenvolvimento e não após a conclusão dos artefatos de software.

Continuava-se, entretanto, com o grande acervo de programas não documentados e que em algum momento necessitavam de evolução. A estratégia de aplicar o processo de especificação manual para redocumentação foi descartada em função do grande volume de aplicações e do esforço e alto custo da operação.

4.2. Elementos do processo

A Figura 3 mostra os principais elementos do processo proposto para a redocumentação de aplicações Cobol utilizando XML. Note que o processo é bastante similar àquele mostrado na Figura 1. Em particular, parte dos componentes usados no processo de refatoração adotado em RefaX foi reutilizado e estendido para compor o processo de redocumentação. Os requisitos que foram considerados para implementação desse processo estão descritos na **Tabela 2**.

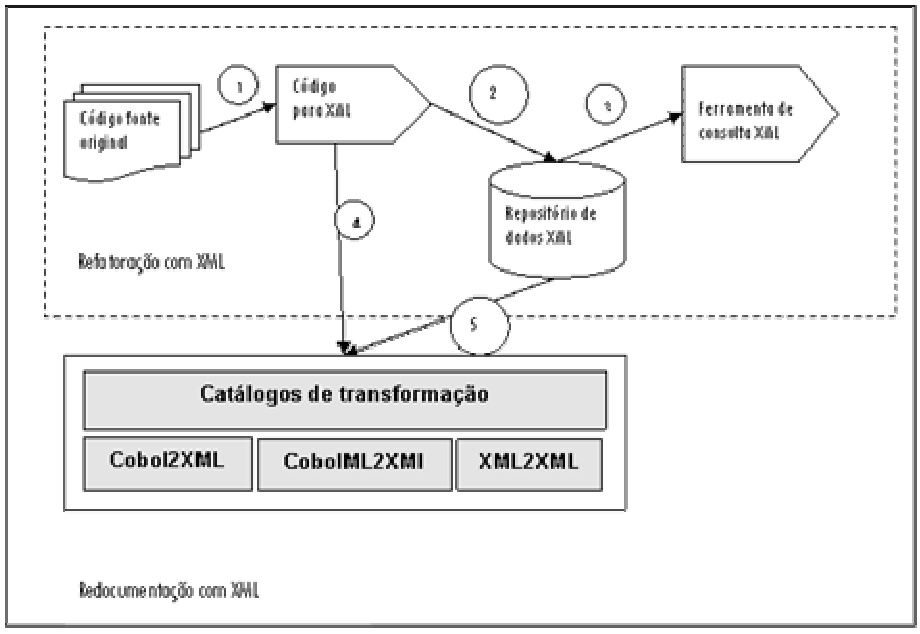


Figura 3 - Processo de redocumentação com XML

Tabela 2 - Necessidades e requisitos para implementação do RedoX-UML

Necessidade	Requisito a ser atendido
Funcionar de forma integrada ao ambiente de trabalho dos profissionais.	A empresa possui dois ambientes de trabalho: *Cobol IDE for Eclipse - Cobol*, que é um subprojeto do projeto *Eclipse,* e *IBM Websphere Developer for zSeries (WdZ)*, que está construído sobre a plataforma *Eclipse.* Além disso, as ferramentas de desenvolvimento disponíveis na empresa são IBM Rational.
Realizar a redocumentação de forma individualizada para um programa ou em lote, para um conjunto de programas.	A ferramenta a ser gerada deve ser empacotada de duas formas: como *plugin* do *Eclipse*, que garante o funcionamento nos dois ambientes de trabalho disponíveis, e na forma de uma aplicação independente para habilitar o processamento em lote.
Obedecer a padrões aceitos no mercado e aderentes ao ambiente de desenvolvimento da empresa.	Adotar a UML 2.0 como padrão de representação e XMI como mecanismo de exportação dos modelos gerados.
Implementar a redocumentação dos aspectos físicos dos programas e permitir a complementação das informações relacionadas a requisitos e regras de negócio pelos profissionais.	Os modelos gerados devem permitir acréscimos nas visões de caso de uso.

4.3. Arquitetura de implementação

A partir dos conceitos e componentes disponíveis no RefaX, e considerando as necessidades e requisitos já citados, é mostrada na Figura 4 uma visão geral da arquitetura da ferramenta RedoX-UML.

Esta arquitetura está organizada nas seguintes camadas ou componentes:

Figura 4 - Arquitetura de suporte ao RedoX-UML

- Gerenciador de dados XML: responsável pelo armazenamento dos arquivos em formato XML que forem gerados a partir da ferramenta que, para a implementação atual, é o próprio sistema operacional;
- Cobol2XML: esse componente é responsável pela transformação de programas fonte Cobol, para o formato XML. Essa representação Cobol em XML está sendo chamada de CobolML;
- CobolML2XMI: esse componente é responsável pela transformação do código-fonte Cobol representado em XML (CobolML) para o formato XMI [OMG 2006b], compatível com a UML;
- XML2XML: esse componente é responsável por outras transformações em formato XML que não o formato específico XMI. Um exemplo são transformações em XML compatíveis com Microsoft Excel;
- Camada de Gerenciamento das Transformações: esse componente é responsável pela coordenação das transformações e também como um elemento de acesso a todas as transformações necessárias à redocumentação;
- Camada de serviços: todos os serviços que devem ser externalizados pelo RedoX-UML estão expostos por esse componente;
- Camada de ferramentas: a partir da camada de serviços é possível construir ferramentas na forma de *plugins* para outros produtos e IDE's ou na forma de aplicações independentes para redocumentação de um lote de vários programas.

4.4. Mapeamento Cobol para UML

Para a consecução dos objetivos do projeto foi necessário adotar um mapeamento dos elementos da linguagem Cobol (ou *statements)* para os elementos da UML. A Tabela 3 mostra o mapeamento de alguns destes elementos.

Tabela 3 - Exemplos do mapeamento Cobol para UML.

Elemento da linguagem	O que ele representa	Mapeamento para UML
Programa Cobol	É o programa em si representado por um arquivo texto composto de diversos elementos	Um elemento na visão de distribuição com o estereótipo <<programa>>.
		Uma classe na visão lógica com o estereótipo <<programa>>.
		Um caso de uso com o mesmo nome do programa.
		Uma realização de caso de uso para cada programa.
Linkage Section	A *linkage section* é a área para troca de dados com outros programas.	Esta área é representada como uma operação (ou método) de visibilidade pública e seus elementos internos são mapeados como parâmetros do método.

Statement *Condicionais*	Representa uma ação de controle de fluxo.	O comando é mapeado como um estereótipo <<decision>>.
Statement *SQL Embedded*	Representa o acesso a um objeto (tabela ou visão) de um banco de dados relacional. São relevantes os objetos envolvidos, o tipo de acesso (select, insert, update, delete) e os campos do comando SQL.	Cada objeto é mapeado como uma classe com um destes estereótipos: <<table>> ou <<view>> .
		Os campos do comando SQL são mapeados como atributos da classe que representa o objeto.
		O tipo de acesso (select, insert, update, delete) cria uma dependência entre o programa e o objeto com os seguintes estereótipos respectivamente <<select>>, <<insert>>, <<update>>, <<delete>>.
Statements *Open, Close, Read, Write.*	Representam o acesso a um arquivo externo.	O arquivo externo é mapeado como uma classe com o estereótipo <<file>>.
		O tipo de acesso (read, write) cria uma dependência entre o programa e o arquivo externo os seguintes estereótipos respectivamente <<read>>, <<write>>.
Statement *Call abc*	Representa o acionamento de outro programa.	É mapeado como uma operação de visibilidade *private*.
		Implica em uma dependência do programa principal com o programa externo.

4.5. Tecnologias utilizadas

4.5.1. XSLT

XSLT [W3C 2005a] é um acrônimo para *Extensible Stylesheet Language Transformations* e é a uma linguagem que vêm sendo bastante aplicada na transformação de documentos XML. Uma transformação XSLT é expressa como um documento XML bem formado que inclui elementos definidos pela linguagem e elementos externos definidos pelo usuário. Uma de suas principais aplicações é a transformação de dados XML em HTML. Uma transformação expressa em XSLT descreve regras de transformação de uma árvore XML em outra através da associação de padrões a modelos. Um padrão é casado com elementos do documento, como nós ou atributos. Um processador XSLT percorre todos os nós do documento, compara-o a cada padrão do modelo, e caso eles casem, a regra é aplicada. A tecnologia XSLT está sendo usada para fazer a transformação da representação do código Cobol em XML para XMI.

4.5.2. XMI

XML Metadata Interchange (XMI) [OMG 2006b] é uma tecnologia padronizada pela OMG (Object Management Group) para a troca de informações entre ferramentas de

modelagem baseadas no padrão OMG-UML e repositórios de metadados baseados no padrão OMG-MOF. XMI está sendo usado para a representação em XML dos elementos da UML extraídos do código Cobol, e para a exportação desses elementos para ferramentas externas de modelagem.

4.5.3. Java

Como parte da arquitetura irá reusar e estender vários componentes do arcabouço RefaX, toda a implementação será feita na linguagem Java, obedecendo aos mesmos requisitos de projeto de arcabouços OO.

4.6. Aplicações do RedoX-UML

A implementação do produto RedoX-UML já está funcional, sendo possível sua utilização para extrair as seguintes informações de um programa Cobol:

- Dependências em relação a outros programas;
- Dependências em relação a fontes de dados (consideramos arquivos tradicionais e objetos relacionais de um banco de dados) incluindo o tipo de acesso (leitura, escrita) e elementos (tipos de registros, colunas) usados;
- Pesquisa de expressões regulares complexas.

Desnecessário dizer que as operações acima citadas são aplicáveis a um programa ou a um conjunto de programas. A partir destas funcionalidades e tirando proveito dos mecanismos de configuração disponíveis no produto já foi possível utilizá-lo da seguinte maneira:

4.6.1 Análise de Impacto

Um cenário de uso com sucesso do framework foi a realização de analises de impacto na migração de código Cobol do dialeto OS VS COBOL I para Cobol Enterprise. Essa migração implica em mudanças em elementos da linguagem Cobol (também chamados de *statements*) que estavam disponíveis para uso no dialeto Cobol I e não são mais suportados no dialeto Cobol Enterprise. O Banco ALFA possuía um acervo de 4.000 programas nessa situação e precisava ter uma estimativa do esforço de migração em horas. A estimativa de esforço é um processo repetitivo que compreende o trabalho de um especialista na linguagem que irá inspecionar o código fonte de um programa, identificar as alterações necessárias e estimar quanto tempo (em horas) é necessário para implementar as alterações. Essa ação será repetida para os demais programas. Com base nessas informações foi desenvolvida uma variação do RedoX-UML que tornou possível:

- A partir de uma lista de *statements* que necessitariam serem alterados, efetuou-se uma varredura no acervo de programas, identificando-se as ocorrências em cada um deles;
- A cada tipo de *statement* foi determinado um peso de complexidade para a migração sendo possível obter um valor que expresse a complexidade de migração de cada programa;

Complementarmente, especialistas na linguagem COBOL determinaram o esforço de alteração por faixa de complexidade. Desta forma e, com base neste conjunto de informações, foi possível determinar o esforço total de migração e a provável necessidade de alocação de pessoas.

Todas as informações de saída foram geradas em formato XML compatível com Microsoft Excel. Essa ação permitiu uma maior facilidade no manuseio das informações e representou uma evolução no processo anterior de análise de impacto baseado somente na inspeção manual de código e manuseio de relatórios tradicionais de referência cruzada gerados pelo próprio compilador.

4.6.2 Documentação de programas

A documentação de programas é o objetivo principal do RedoX-UML e em seu estágio atual já é possível gerar um arquivo XML que contém:

- A identificação do programa;
- As fontes de dados acessadas pelo programa (nesse caso, estamos considerando arquivos tradicionais e objetos de banco de dados relacionais);
- O modo de acesso (leitura, escrita, leitura e escrita) a cada uma destas fontes de dados e onde eles ocorrem na estrutura do programa;
- Que programas são acionados por um determinado programa e quais os parâmetros de entrada e saída do acionamento.

Esse conjunto de informações já permite, para um programa ou um conjunto de programas, gerar uma saída em formato XML que descreve o programa usando os elementos citados acima e, em seguida, aplicar uma transformação XSLT que gere um arquivo XMI compatível com UML tendo como entrada o arquivo XML gerado na etapa inicial. Os arquivos XMI gerados são em seguida importados para a ferramenta case *RSM – Rational Software Modeler*, que é compatível com UML 2.0.

5. Trabalhos relacionados

A redocumentação de aplicações legadas já foi abordada em vários trabalhos anteriores. Esta seção descreve alguns desses trabalhos e os compara como processo proposto neste artigo.

Sneed e Nyary (1995) descrevem um processo de reengenharia para a identificação de objetos em programas Cobol. Muito embora sua abordagem apontasse na direção de que os programas legados deveriam ser migrados para linguagens OO, usando como argumento a predominância da tecnologia de objetos, a prática mostrou que este tipo de processo de migração não foi adotado em larga escala, em função até dos riscos envolvidos [Sneed 1996], ocorrendo com freqüência à convivência entre os dois mundos.

A UML surgiu como uma notação para modelagem de aplicações OO, mas algumas iniciativas recentes abordam o uso da UML para descrever programas procedimentais. Tournier (2002) descreve uma abordagem para uso de modelagem OO para aplicações legadas Cobol, e propõe algumas diretrizes para representá-las usando a UML. Ng (2002) apresenta uma ferramenta para automatizar o processo de documentação de

sistemas legados com a UML. A ferramenta proposta, denominada de CobolRoseAddin, funciona de forma integrada ao produto Rational Rose e gera diagramas que representam a estrutura interna e o comportamento a partir de programas Cobol. Além da ferramenta, Ng estabelece algumas premissas de formato para os programas Cobol para que a redocumentação funcione a contento, propõe etapas para o processo de redocumentação e um modelo de organização de pacotes e subsistemas, aderente ao Rational Rose, para um programa Cobol, e faz um mapeamento inicial de alguns elementos da linguagem Cobol para a UML.

No RedoX-UML, incorporamos, atualizamos e ampliamos o mapeamento inicialmente proposto por Ng (2002) (por exemplo, elementos como *Linkage-Section*, *Data-Division* e acesso a arquivos tipo *flat* não foram considerados por Ng em seu mapeamento). Além disso, acrescentamos o suporte a um processo de redocumentação em lote de programas, entre outras funcionalidades.

Milham (2002) também abordou a questão do uso de elementos gráficos para entendimento de programas e representação de código legado, conduzindo uma investigação sobre a reengenharia de aplicações Cobol com uso de diagramas da UML. Ele conclui que sua expectativa quanto ao uso da notação é que as equipes mantenedoras dos sistemas obtenham um melhor entendimento dos mesmos e possam conduzir as evoluções atendendo aos requisitos dos usuários. Sua abordagem está baseada no uso de WSL [Ward 1992], de tal forma que o código-fonte Cobol é convertido para esta representação e desta ocorre nova transformação para UML. Ele não tece maiores detalhes sobre o processo de conversão de WSL para UML. Posteriormente, Millham *et al.* (2003) conduziram uma nova investigação, desta vez para determinar qual o nível de granularidade adequada para a representação de Cobol em UML, onde concluem que o nível ideal é o de linhas individuais de código.

Seguindo a linha de uso de XML para representação de código, Maruyama e Yamamoto (2004) (2005) propõem uma ferramenta CASE para representação de código Java em XML, denominada Sapid/XML. A partir do projeto SAPID/XML, e usando conceitos semelhantes aos utilizado em RefaX [Mendonça *et al.* 2004], os pesquisadores propuseram uma ferramenta de re-fatoração denominada Jrbx (Java Refactoring Browser with XML), que tira proveito da representação de XML do SAPID e suporta manipulação sobre o conteúdo XML. Além disso, a ferramenta utiliza gráficos de controle de fluxo (CFGs) e gráficos de dependência de programas (PDGs), sendo a conjunção das três tecnologias (XML, CFG e PDG) ressaltada como um diferencial que torna a ferramenta mais amigável.

Anquetil *et al.* (2005) descrevem um processo para redocumentação de aplicações Visual Basic. Além de apresentar as fases do processo e discutir possíveis estratégias de redocumentação, os autores apresentam uma ferramenta, denominada Redoc, sendo desenvolvida para automatizar o processo. Redoc também tem pontos comuns com o este trabalho, posto que trata de redocumentação, sendo seu foco a linguagem Visual Basic. Como vantagens da abordagem ora apresentada, têm-se a representação do sistema em XMI/UML e a possibilidade de tirar proveito das ferramentas disponíveis tanto para visualização de modelos UML como para consultas em arquivos XML como forma de obter informações de referências cruzadas.

Por fim, Milham *et al.* (2004) apresentam a ferramenta TAGDUR (acrônimo para *Transformation and Automatic Generation of Documentation in UML through Reegineering*) que, a partir do código-fonte em Cobol, e baseada em transformações

WSL, gera os diagramas UML de seqüência, componentes e distribuição. Os diagramas UML são representados em formato UXF (*UML Exchange Format*), que é um modelo baseado em XML e proposto por Suzuki e Yamamoto (1999). Os diagramas UXF podem ser importados por ferramentas de visualização compatíveis com esse padrão. Uma vantagem do RedoX-UML em relação a essa proposta é a adoção de padrões abertos para todo o processo de transformação (XML, XMI, UML) e adotar uma abordagem de implementação que funciona de forma autônoma e pode ser evoluída para funcionar também de forma integrada aos ambientes integrados de desenvolvimento (IDE) para Cobol.

6. Conclusão

O projeto RedoX-UML vem atingindo seus objetivos e com as funcionalidades que hoje estão disponíveis já agregou valor ao processo de manutenção de sistemas Cobol do Banco ALFA. Dentre as contribuições importantes do projeto, podemos citar:

- Proposta de um processo para documentar e re-documentar sistemas legados em Cobol, com uma abordagem alinhada às novas práticas de engenharia de software;
- Extensão da família RefaX, com o acréscimo de uma ferramenta para redocumentação de sistemas legados;
- Tirar proveito dos ambientes integrados de desenvolvimento e ferramentas de modelagem utilizadas na indústria de software, ao invés da adoção de uma estratégia de implementação de ambientes proprietários;
- Fornecer um mapeamento de códigos-fonte Cobol para uma representação XML, obedecendo a uma gramática e um processo de transformação Cobol baseados em um esquema XML; a esse processo incorporar uma transformação de modelo para modelo em que se gera um arquivo XMI a partir de um arquivo XML que representa o código COBOL.

Não obstante os benefícios já auferidos, novas funcionalidades serão incorporadas à ferramenta. O plano de evoluções previsto contempla:

- Incorporação de novos elementos da linguagem Cobol no mapeamento para UML;
- Geração de elementos dinâmicos da UML a partir do fluxo de execução de cada programa, permitindo que se façam inferências sobre o comportamento dos aplicativos;
- Incorporação dos elementos do tipo *jcl (Job Control Language)* no processo de redocumentação.

Além disso, em função da representação XML adotada e dos recursos disponíveis no RefaX, será possível promover re-estruturações de código, procurar similaridades e otimizar trechos de código. Adicionalmente, também se pretende, a partir do repositório XMI, implementar a extração de referências cruzadas complexas para mapear dependências entre componentes e permitir avaliar impactos de mudanças de forma mais precisa.

Referências

Anderson, P. (2005), "The Performance Penalty of XML for Program Intermediate Representations", *In Proceedings of the Fifth International Workshop on Source Code Analysis and Manipulation (SCAM 2005)*, Budapeste, Hungria. IEEE Computer Society Press.

Anquetil, N, Oliveira, K. M., Santos, A. G. M., Silva Jr., P. C. S., Araújo Jr., L. C. e Vieira, S. D. C. F. (2005), "Software Re-Documentation Process and Tool", *In Forum of the Conference on Advanced Information Systems Engineering (CAISE 2005)*, Porto, Portugal.

Badros, G. J. (2000), "JavaML: A Markup Language for Java Source Code", *In Proceedings of the 9th International World Wide Web Conference (WWW'00)*, Amsterdam, Holanda, Maio.

Holt, R. C., Winter, A. e Schürr, A. (2000), "GXL: Toward a Standard Exchange Format", *In Proceedings of the 7th Working Conference on Reverse Engineering (WCRE 00)*, IEEE Computer Society Press, Washington, DC, EUA.

Hou, D., Wong, K. e Hoover, H. J. (2005), "What Can Programmer Questions Tell Us About Frameworks?", *In Proceedings of the 13th International Workshop on Program Comprehension (IWPC 05)*, IEEE Computer Society Press, Washington, DC, EUA, pp.87-96.

IBM (2005), "IBM Websphere Developer for zSeries (WdZ)", Disponível on-line em http://www-306.ibm.com/software/awdtools/devzseries/. Último acesso em 02/05/2006.

Maletic, J. I., Collard, M. L. e Marcus, A. (2002), "Source Code Files as Structured Documents", *In Proceedings of the 10th International Workshop on Program Comprehension (IWPC 02)*, IEEE Computer Society Press, Washington, DC, EUA.

Mamas, E. e Kontogiannis, K. (2000), "Towards Portable Source Code Representations Using XML", *In Proceedings of the 7th Working Conference on Reverse Engineering (WCRE 00)*, IEEE Computer Society Press, Washington, DC, EUA.

Maruyama, K. e Yamamoto, S. (2004), "A CASE Tool Platform Using an XML Representation of Java Source Code", *In Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, IEEE Computer Society Press, Washington, DC, EUA, pp.158-167.

Maruyama, K. e Yamamoto, S. (2005), "Design and Implementation of an Extensible and Modifiable Refactoring Tool", *In Proceedings of the 13th International Workshop on Program Comprehension (IWPC 05)*, IEEE Computer Society Press, Washington, DC, EUA, pp.195-204.

Mendonca, N. C., Maia, P. H. M., Fonseca, L. A. e Andrade, R. M. C. (2004), "RefaX: A Refactoring Framework Based on XML", *In Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 04)*, IEEE Computer Society Press, Washington, DC, EUA, pp.147-156.

Millham, R. (2002), "An Investigation: Reengineering Sequential Procedure-Driven Software into Object-Oriented Event-Driven Software through UML Diagrams", *In Proceedings of the 26th International Computer Software and Applications*

Conference (COMPSAC 02), IEEE Computer Society Press, Washington, DC, EUA, p.731-733.

Millham, R., Pu, J. e Yang, H. (2004), "TAGDUR: A Tool for Producing UML Sequence, Deployment, and Component Diagrams Through Reengineering of Legacy Systems", *In Proceedings of the 8th IASTED International Conference on Software Engineering and Applications (SEA)*, Innsbruck, Austria.

Millham, R., Ward. M. e Yang, H. (2003), "Determining Granularity of Independent Tasks for Reengineering a Legacy System into an OO System", *In Proceedings of the International Computer Software and Applications Conference (COMPSAC 03)*, Dallas, Texas, EUA.

Moise, D. L. e Wong, K. (2005), "Extracting and Representing Cross-Language Dependencies in Diverse Software Systems", *In Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 05)*, IEEE Computer Society Press, Pittsburgh, PA, EUA.

Moise, D. L, Wong, K. e Sun, D. (2004), "Integrating a Reverse Engineering Tool with Microsoft Visual Studio .NET", *In Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering (CSMR 04)*, IEEE Computer Society Press, Washington, DC, EUA.

Ng, P. (2002), "Automated Modeling of Legacy Systems Using the UML", *The Rational Edge,* Setembro 2002.

OMG (2006a), "Unified Modeling Language (UML), version 2.0", Disponível on-line em http://www.omg.org/technology/documents/formal/uml.htm. Último acesso em 02/05/2006.

OMG (2006b), "XML Metadata Interchange", Disponível on-line em http://www.omg.org/technology/documents/formal/xmi.htm. Último acesso em 02/05/2006.

Projeto Eclipse (2005), "COBOL IDE for Eclipse", Disponível on-line em http://www.eclipse.org/cobol/. Último acesso em 02/05/2006.

Sneed, H. M. e Nyary, E. (1995), "Extracting object-oriented specification from procedurally oriented programs", *In Proceedings of the Second Working Conference on Reverse Engineering (WCRE 95)*, IEEE Computer Society Press, Washington, DC, EUA.

Sneed, H. M. (1996), "Object-Oriented COBOL Recycling". *In Proceedings of Third Working Conference on Reverse Engineering (WCRE 96)*, IEEE Computer Society Press, Monterey-CA, EUA, pp.169-178.

Sun, D. & Wong, K. (2005), "On Evaluating the Layout of UML Class Diagrams for Program Comprehension", *In Proceedings of the 13th International Workshop on Program Comprehension (IWPC 05)*, IEEE Computer Society Presxs, Washington, DC, EUA, pp.317-326.

Suzuki, J. e Yamamoto, Y. (1999), "Making UML Models Interoperables with UXF", *In Selected papers from the First International Workshop on The Unified Modeling Language UML 98*, Springer-Verlag, Londres, Inglaterra, pp.78-91.

Tichelaar, S. *et al.* (2000), "A Meta-model for Language-Independent Refactoring", *In Proceedings of the International Symposium on Principles of Software Evolution (ISPSE 2000),* Kanazawa, Japão, Novembro.

Tournier, C. (2002), "Modeling Guidelines for Legacy Applications", *The Rational Edge,* Setembro 2002.

W3C (1999), "XSL Transformations (XSLT) Version 1.0". W3C Recommendation November 1999. Disponível on-line em http://www.w3.org/TR/xslt . Último acesso em 02/05/2006.

W3C (2004), "Extensible Markup Language (XML)". W3C Recommendation February 2004. Disponível on-line em http://www.w3.org/TR/xml. Último acesso em 02/05/2006.

Ward, M. (1992), "The Syntax and Semantics of the Wide Spectrum Language", Relatório Técnico, Universidade de Durham, Inglaterra.

SOCRATES – Sistema Orientado a objetos para CaRActerização de refaToraçõES

André Toledo Piza de Moura[1], **Marcos Lordello Chaim**[2]

[1]Fundação Atech
Rua do Rocio, 313 – 04552-000 – São Paulo – SP – Brasil

[2]Escola de Artes, Ciências e Humanidades
Universidade de São Paulo
Av. Arlindo Bettio, 1000 – 03828-000 – São Paulo – SP – Brasil

apiza@atech.br,chaim@usp.br

Abstract. *Refactoring is the activity of modifying a computer program's source code without changing its external behavior. These changes aim at making the code more understandable. Nevertheless, this activity is complex and error-prone since it is usually carried out manually. This work introduces SOCRATES — Sistema Orientado a objetos para CaRacterização de refaToraçõES (Object Oriented System for Characterization of Refactorings) which purpose is to automatically detect points to be refactored, i.e., refactoring opportunities. SOCRATES atractiveness resides on how it achieves such a goal. It is built upon open source tools which demand little additional coding. The extra code makes the tools work together and implements the algorithms for refactoring opportunities identification. These algorithms are encapsulated and can be implemented efficiently. The present version of SOCRATES identifies the "obsolete parameter"refactoring opportunity and shows that the concepts used are valid.*

Resumo. *Refatoração é o ato de modificar o código fonte de um programa de computador sem, contudo, modificar seu comportamento observável. As modificações são feitas visando deixar o código mais fácil de ser entendido. Entretanto, esta atividade é complexa e sujeita a erros, uma vez que normalmente é realizada de forma manual. Este trabalho apresenta SOCRATES (Sistema Orientado a objetos para CaRActerização de refaToraçõES) cujo objetivo é fornecer auxílio automático para a identificação de pontos do software candidatos a serem refatorados, isto é, oportunidades de refatoração. A atratividade de SOCRATES está na forma como este objetivo é atingido: utilizando ferramentas de código aberto que requerem pouca codificação adicional. O código extra faz com que as ferramentas trabalhem em conjunto e implementa os algoritmos de identificação das oportunidades de refatoração. Estes algoritmos são encapsulados e podem ser implementados de forma eficiente. A presente versão do SOCRATES identifica de maneira automática a oportunidade de refatoração "parâmetro obsoleto"e mostra que os conceitos utilizados são válidos.*

1. Introdução

A atividade de manutenção tem importância fundamental no desenvolvimento de programas de computador. Estima-se que cerca de 60% do ciclo vida de um programa esteja

relacionado às atividades de manutenção [Mayrhauser and Vans 1995]. A manutenção é tradicionalmente entendida como a fase do ciclo de vida que acontece após o software ter sido entregue ao cliente. Durante a manutenção, diferentes tarefas são realizadas como: correção de defeitos que foram descobertos durante a utilização em produção do sistema; migração do software para novas plataformas tecnológicas (e.g., uma nova interface, uma nova arquitetura etc.); inclusão de novas funcionalidades identificadas *a posteriori* pelos usuários; e reorganização da estrutura ou do código do software visando facilitar modificações futuras.

Com o advento dos métodos ágeis de desenvolvimento de software [Beck 1999], atividades realizadas apenas depois da entrega do software ocorrem dentro do processo de desenvolvimento. Por exemplo, a realização de mudanças que visam melhorar a estrutura do software é uma das práticas dos métodos ágeis, em especial da chamada Programação Extrema [Beck 1999]. Esta é uma atividade típica de manutenção que em Programação Extrema é realizada no desenvolvimento.

Essas mudanças, conhecidas como manutenções preventivas, podem ser realizadas com o auxílio de técnicas e ferramentas. Grande parte destas técnicas estão reunidas sob o conceito de reestruturação/refatoração [Mens and Tourwé 2004]. *Reestruturação* são mudanças feitas na estrutura do software no sentido de deixá-lo mais fácil de ser entendido e menos custoso de ser modificado, sem, contudo, modificar seu comportamento observável. O termo *refatoração*, por sua vez, foi cunhado por Opydike [Opdyke 1992] (apud [Mens and Tourwé 2004]) como "o processo de modificação de um sistema (orientado a objeto) de software de forma que o sistema não tenha o seu comportamento externo alterado, embora a sua estrutura interna seja melhorada".

Segundo Mens e Touwé [Mens and Tourwé 2004], refatoração envolve o mesmo conceito de reestruturação, porém, aplicado ao contexto dos sistemas orientados a objetos, sendo a idéia central redistribuir classes, variáveis e métodos na hierarquia de classes a fim de facilitar futuras adaptações extensões. Fowler e Beck [Fowler 1999] referem-se às "estruturas no código que sugerem a (às vezes gritam pela) possibilidade de refatoração" como *maus cheiros*. Este termo inusitado (sua origem está relacionada com o cuidado de bebês pequenos — se está com mau cheiro então deve-se trocar a sua fralda) visa indicar pontos do software que são candidatos naturais à aplicação de refatorações. Neste sentido, os maus cheiros constituem oportunidades para aplicação de refatorações.

No entanto, a tarefa de identificar o ponto do programa candidato a ser refatorado não é trivial. Considere-se a situação de mau cheiro caracterizada por um *parâmetro obsoleto* em um método, isto é, um parâmetro que não é mais utilizado. Este mau cheiro é eliminado pela aplicação da refatoração *remove parameter* [Fowler 1999]. Mesmo a identificação de um mau cheiro trivial como um parâmetro obsoleto pode demandar uma investigação detalhada do código.

Isto porque a implementação de um método pode estar espalhada em várias subclasses que o redefinem. Assim, supondo que a classe implemente m métodos e que n subclasses desta classe redefinam este método, o desenvolvedor precisa inspecionar $m \times n$ métodos para detectar um parâmetro obsoleto. Além disso, essa tarefa deve ser realizada para todo parâmetro formal que o método define. Portanto, detectar a ocorrência de um parâmetro obsoleto é um processo custoso e demorado [Tourwé and Mens 2003].

Este trabalho apresenta SOCRATES (Sistema Orientado a objetos para CaRActerização de refaToraçõES) cujo objetivo é fornecer auxílio automático para a detecção de maus cheiros, isto é, para detectar oportunidades de refatoração. Dessa forma, está-se automatizando uma parte custosa da atividade de manutenção — e também de desenvolvimento, no caso dos métodos ágeis — que é a compreensão do software; especificamente, a compreensão do código para identificação de maus cheiros e posterior aplicação de refatorações. A atratividade de SOCRATES está no fato de utilizar ferramentas de código aberto e de requerer pouca codificação adicional. Esta codificação é direcionada para a orquestração das ferramentas e para a implementação de algoritmos de identificação de oportunidades de refatoração. Os algoritmos são encapsulados e podem ser implementados de forma eficiente.

O restante deste trabalho é organizado da seguinte maneira. Na próxima seção é apresentado o processo de refatoração e a tarefa que SOCRATES apóia deste processo; além disso, são discutidas as abordagens para realização desta tarefa. Na Seção 3, a estrutura do SOCRATES é apresentada utilizando um exemplo simplificado. Os trabalhos relacionados são discutidos na Seção 4. As conclusões são apresentadas na Seção 5.

2. Identificação de oportunidades de refatoração

De acordo com Mens e Touwé [Mens and Tourwé 2004], o processo de refatoração consiste das seguintes atividades principais:

1. identificar onde o software deve ser refatorado;
2. determinar quais refatorações devem ser aplicadas nos pontos identificados;
3. garantir que a refatoração aplicada preserva o comportamento original;
4. aplicar a refatoração;
5. verificar o efeito da refatoração nas características de qualidade do software (e.g., complexidade, manuntenibilidade) ou no seu processo (e.g., produtividade, custo, esforço);
6. Manter a consistência entre o código refatorado e outros artefatos (e.g., documentação, documentos de projeto, especificação de requisitos, testes).

SOCRATES tem como objetivo apoiar o passo 1 do processo de refatoração. Diferentes abordagens foram definidas para realizar este passo, a saber, análise cognitiva [Fowler 1999], análise do código [Balazinska et al. 2000, Tourwé and Mens 2003], análise dinâmica do programa [Kataoka et al. 2001] e análise baseada em métricas [Carneiro and Mendonça Neto 2003]. A análise cognitiva é a abordagem proposta por Fowler e colegas; ela basicamente preconiza que o programador, a partir dos maus cheiros catalogados [Fowler 1999] e na sua experiência, procure identificar os pontos onde existem oportunidades de refatoração. Esta é a abordagem mais utilizada devido a sua simplicidade, porém, é a mais sujeita a erros visto que é realizada manualmente e depende essencialmente da experiência do programador e do seu conhecimento dos padrões de código catalogados como mau cheiros.

A análise de código consiste na análise sintática e semântica do código para encontrar oportunidades de refatoração. Por exemplo, Balazinska et al. [Balazinska et al. 2000] classifica os diferentes tipos de código duplicado (chamados de *clones*) e aplica um algoritmo de identificação de clones para caracterizar oportunidades de refatoração. Tourwé e Mens [Tourwé and Mens 2003], por sua vez, desenvolveram a ferramenta SOUL que

implementa uma linguagem no estilo da linguagem PROLOG para realizar consultas ao código fonte Smalltalk. A vantagem de SOUL é que os algoritmos para identificação de oportunidades de refatoração podem ser escritos de forma elegante e concisa. No entanto, para a identificação de maus cheiros, muitas vezes, necessita-se percorrer a árvore sintática do programa, o que faz com que SOUL inclua predicados para estas tarefas.

A análise dinâmica procura identificar expressões invariantes no programa que constituem oportunidades de refatoração. As expressões invariantes são obtidas a partir de execuções do programa. Kataoka et al. [Kataoka et al. 2001] utilizam a ferramenta Daikon [Ernst et al. 2001] para identificar invariantes (e.g., uma variável cujo valor não se altera ou que é definida em função de outras). Estas invariantes podem ser úteis para identificar a presença de oportunidades de refatoração. O problema dessa abordagem é que ela requer diversas execuções do programa para a caracterização de uma invariante; porém, este processo pode ser custoso e não definitivo visto que em geral não é possível executar todas as entradas de um programa.

Outra abordagem para identificar oportunidades de refatoração é por meio de métricas. Um exemplo dessa abordagem são as métricas identificadas e definidas por Carneiro e Mendonça Neto [Carneiro and Mendonça Neto 2003]. Estes autores utilizaram a técnica GQM (GQM — *Goal Question Metric*) [Pigoski 1997] para identificar e definir métricas a partir do catálogo de Fowler. Em seguida realizaram um estudo de caso para verificar qual a relação entre métricas, maus cheiros e refatorações. O uso de métricas é interessante devido a sua simplicidade e baixo custo de aplicação; porém, sua utilização confiável requer a realização de estudos abrangentes e repetidos.

SOCRATES utiliza a abordagem de análise de código. Semelhantemente à ferramenta SOUL, utiliza recursos de inteligência artifical para detectar maus cheiros. Estes recursos são fornecidos por um mecanismo de regras implementado no contexto da linguagem Java [Drools 2005]. Este mecanismo ativa regras que implementam os algoritmos de identificação de oportunidades de refatoração. Esse algoritmos são escritos em Java e estão encapsulados. A idéia é que o mecanismo de regras possa combinar o resultado de diferentes algoritmos ou da aplicação de métricas para decidir pela ocorrência ou não de uma oportunidade de refatoração. Na Seção 4 é feita a comparação entre SOCRATES e outras abordagens semelhantes.

3. Descrição do SOCRATES

3.1. Estrutura

SOCRATES é um protótipo para identificação de possíveis oportunidades de refatoração em programas escritos em linguagem Java. Para atingir seu objetivo, ele recebe como entrada um programa Java que é analisado e mapeado para um arquivo em formato XML. Este arquivo contém informações sobre o código do programa tais como classes e métodos definidos, troca de mensagens entre objetos e definições e usos de variáveis.

A partir dessas informações e da utilização de regras pré-definidas, SOCRATES analisa o programa para identificar pontos candidatos a uma possível refatoração. Estas regras são disparadas por um motor de inferências capaz de guardar estados e informações auxiliares entre um disparo e outro, além de mudar o fluxo de aplicação destas regras dependendo dos resultados obtidos.

A estrutura do SOCRATES é mapeada em três componentes de negócios e quatro de infra-estrutura conforme descrito na Figura 1. Os componentes possuem relações de dependências sem ciclos e definem o fluxo de informações do SOCRATES. Esse fluxo de informação, apresentado na Figura 2, estende-se desde o momento em que o código fonte é lido até o momento em que se encontram os pontos candidatos a refatoração. Estes componentes foram construídos de forma que os efeitos dessa relação de dependência sejam minimizados e os componentes possam ser substituídos ou customizados sem que os impactos desta operação sejam desconhecidos. A seguir são descritos os componentes do SOCRATES.

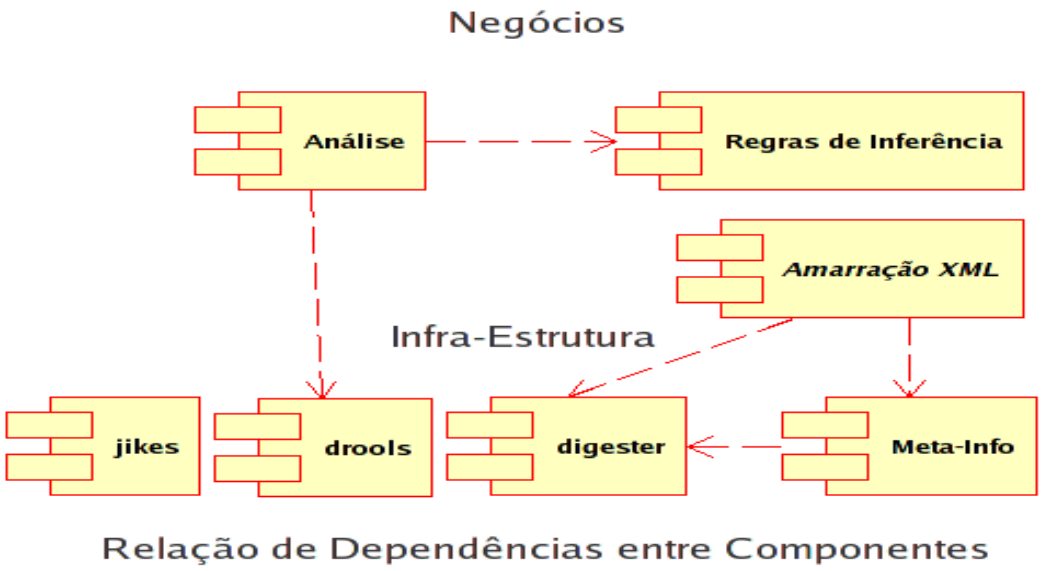


Figura 1. Estrutura de componentes do SOCRATES.

3.2. Extração da meta-informação do código fonte – Jikes

A ferramenta de infra-estrutura Jikes [Jikes 2005] é responsável por extrair do programa Java (que está sendo analisado quanto à possível aplicação de refatorações) informações a respeito da sua estrutura. As informações que são coletadas a respeito do programa são:

- Informações Estruturais:
  - Nome de pacotes, classes, atributos, métodos, parâmetros e variáveis locais.
- Informações Comportamentais:
  - Nome de classes importadas, declaração de objetos e envio de mensagens.
- Localização de Arquivos:
  - Localização do arquivo original Java em disco de acordo com as convenções do sistema operacional utilizado.

Jikes é um compilador Java desenvolvido inicialmente pela IBM (International Business Machines). Hoje, porém, é um projeto de software livre, podendo ser obtido no sítio SourceForge [Jikes 2005]. Badros [Badros 2000] modificou a versão original

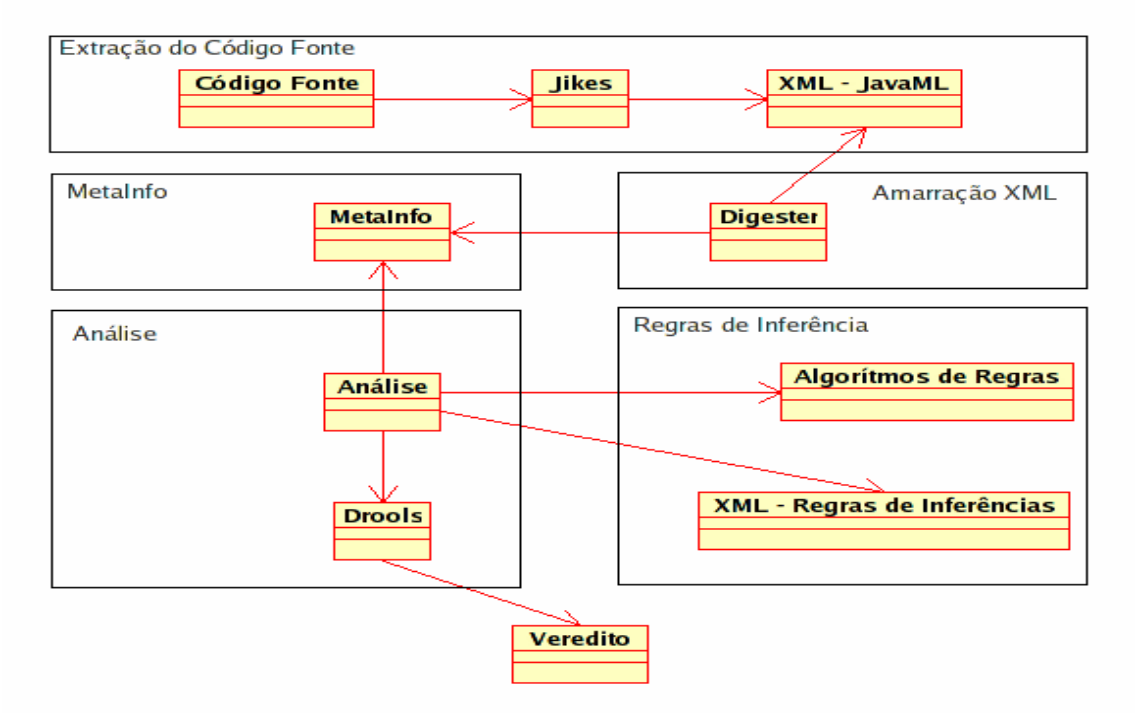


Figura 2. Fluxo de informações dos componentes do SOCRATES.

do compilador para criar arquivos XML que representam as informações sobre o código compilado.

Estas informações estarão disponíveis em formato XML e descrevem o mesmo programa Java em uma representação alternativa, porém, mais facilmente manipulável por outras ferramentas. Os elementos do arquivo XML gerado pela ferramenta Jikes segue o formato JavaML criado por Badros [Badros 2000]. A Figura 4 apresenta o resultado do processamento do trecho de programa contido na Figura 3 pelo Jikes.

Observando a Figura 4, nota-se que as informações resultantes da análise sintática e semântica do código estão contidas no arquivo XML gerado por Jikes. Por exemplo, na linha 4, o rótulo <`java-class-file ...`> indica a localização em disco do arquivo original Java. Os atributos do rótulo <`class ...`> (linha 5) indicam o nome da classe definida, no caso `''Exemplo''`, sua visibilidade (`public`), bem como a sua localização no código fonte (atributos `line`, `col`, `end-line`, `end-col`). Outras informações relativas ao método declarado (linhas 8 a 16) e ao uso e definição de variáveis (linha 14) estão igualmente definidas utilizando os elementos do padrão JavaML.

3.3. Amarração XML — XML *Binding*

O componente Amarração XML (XML *Binding*) é o responsável por tornar disponível para os outros componentes do SOCRATES as informações estruturais disponíveis no arquivo JavaML gerado por Jikes. Isto é realizado no SOCRATES amarrando-se o conteúdo do arquivo JavaML resultado da compilação do programa pelo Jikes a objetos Java acessíveis pelos demais componentes do SOCRATES.

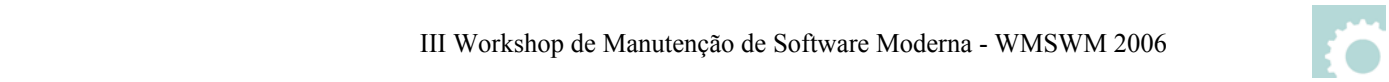
Para a realização dessa amarração, SOCRATES utiliza o componente de infra-estrutura Digester [Digester 2005]. O componente Digester lê o arquivo JavaML e cria

```
public class Exemplo{

   String variavel = "";

   public void metodo(String  parametro){
       variavel = "Parametro eh obsoleto !";
  }
}
```

Figura 3. Programa exemplo [Badros 2000].

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <!DOCTYPE java-source-program SYSTEM "java-ml.dtd">
3: <java-source-program>
4: <java-class-file name="C:/jikes/socrates/Exemplo.java">
5: <class name="Exemplo" visibility="public" line="1" col="0"
               end-line="9" end-col="0">
6: <superclass name="Object"/>
7: <field name="variavel" line="3" col="8" end-line="3" end-col="28">
      <type name="String"/><literal-string value=""/>
   </field>
8: <method name="metodo" visibility="public" id="Exemplo:mth-13"
                line="5" col="8" end-line="7" end-col="8">
9:    <type name="void" primitive="true"/>
10:   <formal-arguments>
11:      <formal-argument name="parametro"
               id="Exemplo:frm-11"><type name="String"/>
         </formal-argument>
12:   </formal-arguments>
13:   <block line="5" col="44" end-line="7" end-col="8">
14:      <assignment-expr op="=">
         <lvalue><var-set name="variavel"/></lvalue>
         <literal-string value="Parametro eh obsoleto !"/>
         </assignment-expr>
15:   </block>
16: </method>
17:</class>
18:</java-class-file>
19:</java-source-program>
```

Figura 4. Arquivo JavaML gerado pelo componente Jikes a partir da Figura 3.



uma classe Java que contém as informações estruturais necessárias para a análise do código. A classe gerada pelo Digester é instanciada em um objeto capaz de ser acessado pelos demais componentes do SOCRATES.

Digester [Digester 2005] é um *framework* criado pelo projeto *Jakarta Commons* da *Apache Software Foundation* (ASF) com o objetivo de configurar arquivos XML de forma que possam ser mapeados em objetos Java. A Figura 5 contém a classe Java (esta figura e as que se seguem foram obtidas a partir da interface do ambiente de programação Eclipse [Eclipse 2005]) mapeada pelo componente Digester a partir do arquivo JavaML contido na Figura 4.

A classe Java mapeada é a classe `Metainfo.java`[1] descrita na Figura 5. Esta classe, indicada na figura pela letra C maiúscula e seu nome (`Metainfo`), é composta por uma lista de objetos da classe `File` (arquivo) que representam os arquivos físicos em disco.

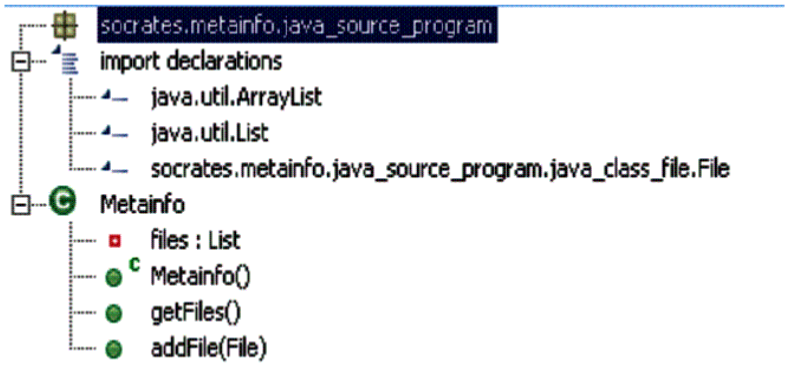


Figura 5. Classe Java mapeada pelo componente Digester a partir do arquivo JavaML.

A classe `File`, por sua vez, representada na Figura 6 a partir da letra C, possui um nome que representa sua localização física em disco, uma lista de objetos `Import` contendo informações sobre as classes que devam ser utilizadas internamente, um objeto do tipo `Package` que possui informações sobre a localização lógica e um objeto `Class` que é a classe que o arquivo em disco contém.

Por fim, a classe `Class`, representada na Figura 7, armazena as informações sobre todos os elementos utilizados internamente na classe de forma lógica, ou seja, informações sobre métodos, parâmetros, variáveis, chamadas de outros objetos e informações de hierarquia de heranças.

3.4. Meta-info

O componente *Meta-info* é a instanciação em objetos da classe Java mapeada pelo componente *Amarração XML*. Nos objetos, as informações estruturais relativas ao programa

[1] `Metainfo` refere-se à classe ou ao objeto Java. *Meta-Info* refere-se ao componente do SOCRATES.

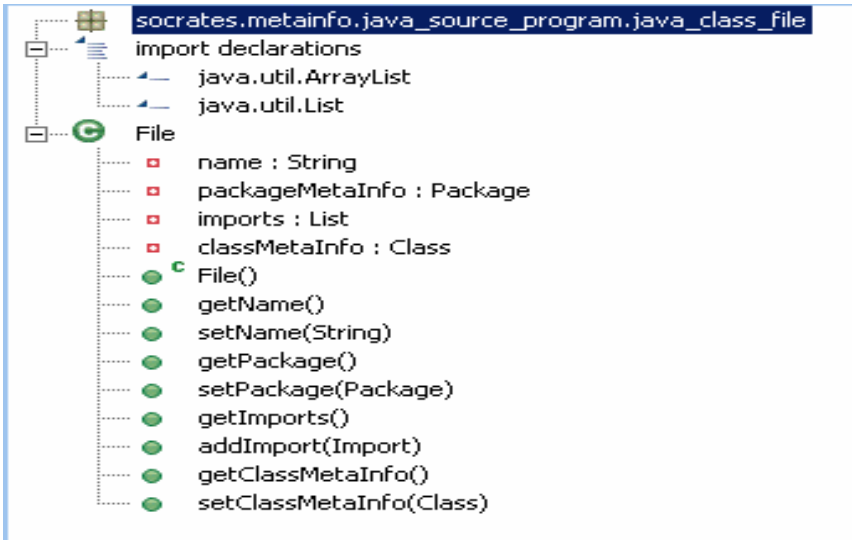


Figura 6. Classe que representa o arquivo Java em disco a ser analisado.

Figura 7. Classe que representa as meta-informações da classe Java a ser analisada.

Java analisado pelo Jikes e contidas no arquivo JavaML estão disponíveis na forma de atributos. A Figura 8 apresenta a instanciação dos objetos que compõe o componente *Meta-info* para o programa da Figura 3.

Assim, o componente *Meta-info* contém os itens de análise do programa e suas representações em objetos internos ao SOCRATES. Estes objetos internos serão utilizados pelos outros componentes para identificar as oportunidades de refatoração. Por exemplo, no processo de análise para identificar um parâmetro obsoleto precisamos investigar como um parâmetro e um método se relacionam para decidir se o parâmetro é ou não obsoleto. Assim, método e parâmetro são para o SOCRATES objetos Java que contêm informações de uso em sua forma original no código fonte.

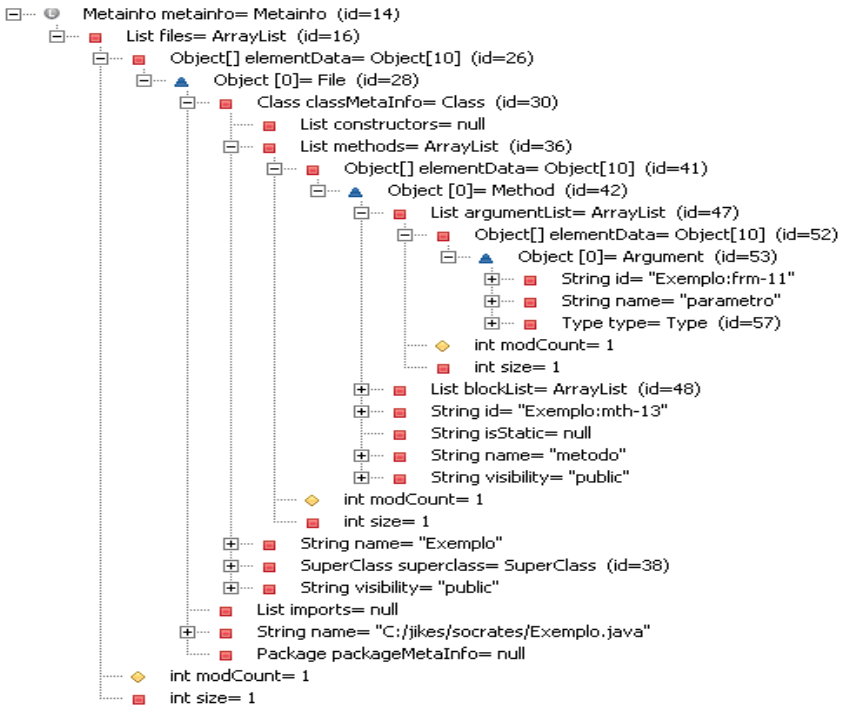


Figura 8. Instanciação da classe Java criada pelo componente *Digester* para utilização pelo SOCRATES.

3.5. Análise

O componente de análise é o responsável por comandar a execução dos algoritmos que identificam oportunidades de refatoração. Os algoritmos são aplicados aos objetos de *Metainfo* recém-extraídos. A ferramenta Drools [Drools 2005] é a infra-estrutura que permite a definição e a operação de regras que invocam os algoritmos de identificação de refatorações.

Drools é uma máquina de regras baseada em linguagens declarativas. Internamente, esta máquina processa as regras através de um grafo em que cada vértice representa uma regra e cada aresta um relacionamento entre elas. Nesta representação, uma regra é um conjunto que contém uma condição e uma conseqüência. Quando a condição for satisfeita, a conseqüência, que contém uma ação a ser realizada, será disparada. Após sua execução, de acordo com o grafo formado, a próxima regra será chamada.

Para construir o grafo, Drools interpreta arquivos XML que representam as regras. A Figura 9 contém as regras definidas para identificar o mau cheiro parâmetro obsoleto.

A regra da Figura 9 é dividida em três seções. A primeira seção no elemento *parameter* (linhas 2 a 4), define os parâmetros que podem ser passados para a regra. Estes parâmetros funcionam da mesma forma que parâmetros passados na chamada de métodos comuns à linguagem de programação Java; são cópias dos valores de objetos externos que podem ser utilizados nas outras seções da regra. A segunda parte, definida pelo elemento *condition* (linha 5), informa a condição de ativação da regra, é o momento em que os predicados cruzam com os fatos lógicos, que podem ser objetos do próprio programa a ser analisado. Por fim, o elemento *consequence* (linhas 6 a 18), informa o que deve

```
1: <rule name="parametro obsoleto">
2:    <parameter identifier="metainfo">
3:    <java:class>socrates.metainfo.java_source_program.Metainfo
      </java:class>
4:    </parameter>
5:    <java:condition>metainfo != null</java:condition>
6:    <java:consequence>
7:    socrates.rules.obsoleteparameter.ObsoleteParameter ob =
      new socrates.rules.obsoleteparameter.ObsoleteParameter(metainfo);
8:    java.util.List obsoleteParamenters = ob.getObsoleteParameters();
9:    if(!obsoleteParamenters.isEmpty()){
10:     System.out.println("Os seguintes parametros sao obsoletos: ");
11:       for (java.util.Iterator iObsolete = obsoleteParamenters.\-
          iterator(); iObsolete.hasNext();) {
12:          socrates.metainfo.java_source_program.java_class_file.\-
             class_decl.method.Argument argument = (socrates.metain\-
             fo.java_source_program.java_class_file.class_decl.meth\-
             od.Argument) iObsolete.next();
13:         System.out.println("* "+argument.getName());
14:       }
15:   } else{
16:   System.out.println("Não existem parametros obsoletos");
17:   }
18:   </java:consequence>
19:</rule>
```

Figura 9. Arquivo XML de Regras para identificar o mau cheiro parâmetro obsoleto.

acontecer se o teste da condição resultar em verdadeiro.

Na Figura 10 pode-se ver a classe `Brain` que realiza a análise. No canto esquerdo, encontra-se a sua estrutura; no canto direito inferior está localizada a divisão em pacotes; e no canto direito superior, estão seus dois métodos mais importantes que fazem o mapeamento do SOCRATES com o componente de infra-estrutura Drools. O método `setup` é responsável por passar o caminho correto dos arquivos XML de regras a serem executadas e o método `activate` dispara a execução destas regras.

Os arquivos XML de Regras possuem uma estrutura definida pela própria ferramenta de infra-estrutura para orquestrar o fluxo de execução dos algoritmos programados em Java que definem a implementação das regras de análise. Estes algoritmos constituem as Regras de Inferência discutidas a seguir.

3.6. Regras de Inferência

As regras de inferência são programas Java que implementam algoritmos para encontrar os mau cheiros. O cruzamento positivo entre as regras de inferência e um objeto da classe `Metainfo.class` indica que um candidato a refatoração foi identificado.

Na Figura 9, no elemento `consequence`, o objeto `metainfo` é passado como parâmetro pelo SOCRATES, que define o conjunto de informações extraídos do código

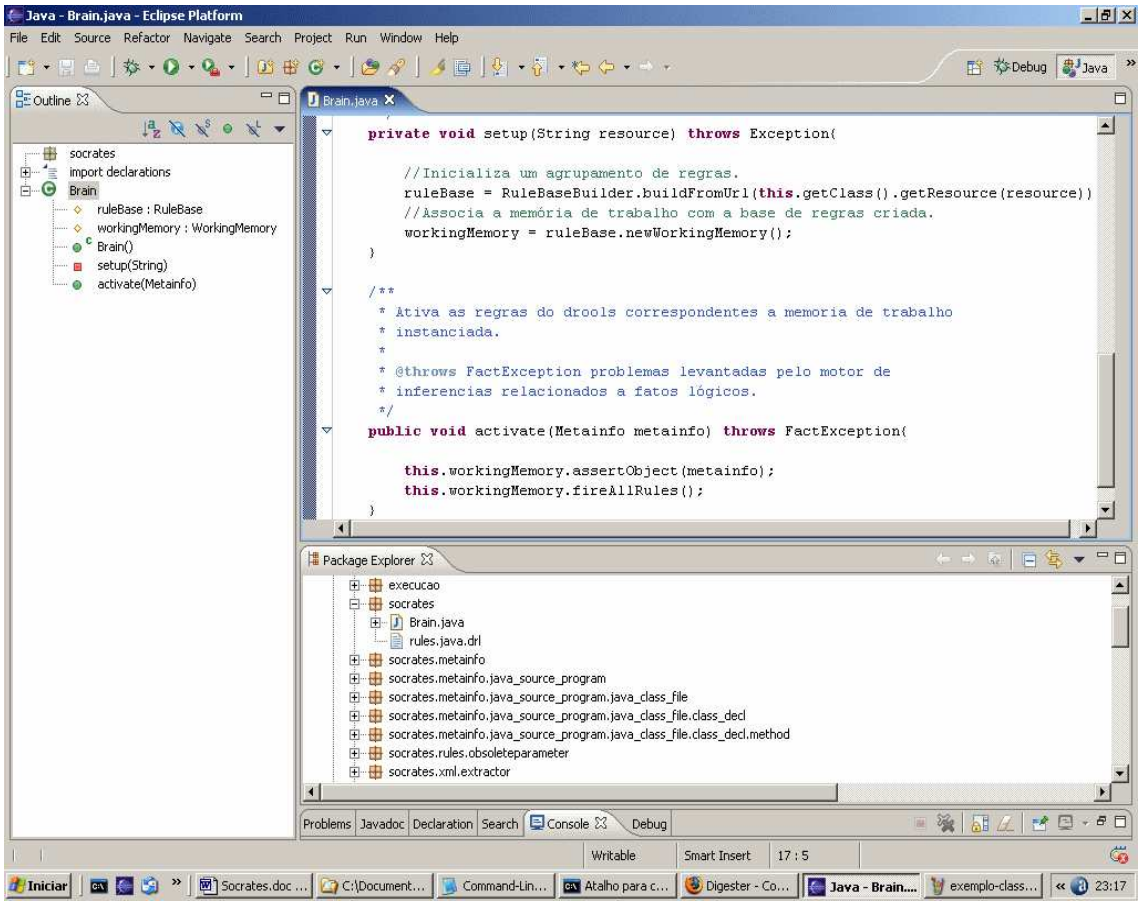


Figura 10. Representação estrutural do componente de análise e sua posição na estrutura de pacotes com todos os componentes do SOCRATES.

fonte a ser analisado. A condição testa se este objeto é nulo, caso não o seja, a conseqüência será ativada. Neste momento o algoritmo que verifica a existência de parâmetro obsoleto (linha 7) será instanciado e executado. O objeto `metainfo` contém todas as informações relevantes do programa a ser analisado para se chegar a um veredito. Caso algum parâmetro obsoleto seja encontrado o usuário do sistema será informado (linhas 8 a 17).

A regra de inferência é dada pela classe Java `socrates.rules.obsoleteparameter.ObsoleteParameter` que implementa o algoritmo que utiliza os dados contidos em objetos da classe `Metainfo.class` para identificar oportunidades de refatoração. Esta classe é ativada dentro do elemento <`java:consequence`> quando é criado um objeto (linha 7) que recebe como parâmetro um objeto da classe `Metainfo.class`. Na linha 8, o método `getObsoleteParameters()` aplica o algoritmo de identificação de parâmetros obsoletos no objeto da classe `Metainfo.class` e retorna uma lista de parâmetros. As linhas 9 a 17 contêm código Java para imprimir o resultado da análise do código.

3.7. Seqüência de atividades do SOCRATES

A seqüência de atividades do SOCRATES é apresentada na Figura 11 e cada passo é descrito a seguir:

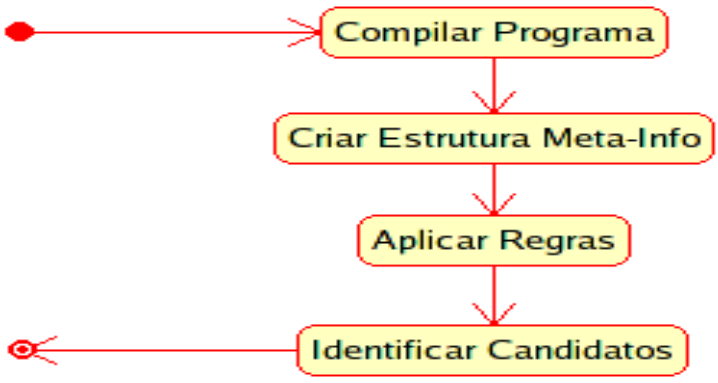


Figura 11. Seqüência de atividades do SOCRATES.

1. O programa fonte Java é analisado pela ferramenta Jikes gerando um arquivo XML com as meta-informações do programa a ser analisado. Este passo é representado pela atividade *Compilar Programa*.
2. O Componente *Amarração XML* recebe como entrada o arquivo JavaML gerado e o amarra a objetos Java capazes de serem manipulados pelo SOCRATES. O componente *Amarração XML* utiliza o Digester para fazer a amarração com objetos do componente *Meta-Info*. Este passo é representado pela atividade *Criar Estrutura Meta-Info*.
3. As classes de *Meta-Info* são, então, utilizadas pelos algoritmos de busca de candidatos a refatoração implementados no componente *Regras de Inferência*. Os arquivos XML de Regras gerenciam as aplicações dos algoritmos de busca que resultam na identificação positiva ou negativa de oportunidades de refatoração. Este passo utiliza o Drools como infra-estrutura e é representado pelas atividades *Aplicar Regras* e *Identificar Candidatos*.

Desta forma, SOCRATES consegue atingir seus objetivos de automatizar a identificação de oportunidades de refatoração. O resultado da análise para a identificação de parâmetros obsoletos para o exemplo simplificado contido na Figura 3 pelo SOCRATES é apresentado a seguir.

```
Os seguintes parametros sao obsoletos: *args
```

Como exemplo SOCRATES foi utilizado para identificar parâmetros obsoletos em classe do próprio SOCRATES [Moura 2006].

4. Trabalhos relacionados

SOCRATES possui características semelhantes à da ferramenta SOUL; porém, o projeto e a arquitetura são bastante distintos. SOUL utiliza 84 classes e aproximadamente 1100 métodos escritos em Smalltalk, o que faz dela uma aplicação de tamanho médio [Tourwé and Mens 2003]. Isto deve-se em grande parte à interface de programação lógica que SOUL provê e à implementação dos predicados que consultam o código fonte. A interface é genérica e permite que sejam escritos os algoritmos para identificação de maus cheiros. Os autores dão como exemplo a implementação da identificação de duas oportunidades de refatoração — *parâmetro obsoleto* e *interface obsoleta*.

A codificação da presente versão do SOCRATES, que identifica a ocorrência do mau cheiro *parâmetro obsoleto*, envolveu basicamente: o mapeamento dos elementos do formato JavaML para classes, métodos e atributos do componente *Meta-Info* (nesta versão apenas parte dos elementos de JavaML foram mapeados); a orquestração das regras tratadas pela ferramenta Drools e das regras de inferência realizada no módulo de *Análise*; e a implementação da regra de inferência propriamente dita. O maior trabalho de codificação foi concentrado na implementação da regra de inferência para identificação de parâmetros obsoletos. A classe Java que implementa esta regra possui 8 métodos [Moura 2006].

SOCRATES necessitou menos codificação porque utiliza componentes livres que realizam parte das tarefas que são codificadas em SOUL. Por exemplo, o mecanismo de regras de Drools permite a combinação de diferentes regras de inferência para decidir sobre a ocorrência de um mau cheiro. Este mecanismo pode ser utilizado para combinar o resultado da aplicação de métricas e de um algoritmo para decidir sobre a existência de um mau cheiro. Outro aspecto é que essas regras de inferência podem ser implementadas eficientemente pois são programas Java ativados pela máquina de regras.

Na ferramenta SOUL, os algoritmos e a consulta ao código Smalltalk necessitam ser codificados na linguagem de programação lógica ou incluídos como predicados da linguagem. A facilidade de inclusão da identificação de parâmetros obsoletos dá indícios de que a estrutura de SOCRATES é flexível e requer pouca codificação. Porém, a confirmação dessa hipótese requer a implementação da identificação de maus cheiros diferentes e mais complicados.

Uma alternativa à utilização da ferramenta Drools seria a utilização da linguagem Prolog em programas Java. Por exemplo, InterProlog [Declarativa 2006] é um *front-end* que permite programas Java invocarem objetivos Prolog e predicados Prolog invocarem métodos Java. É necessário, porém, investigar as soluções utilizando Prolog e Drools em termos de facilidade de criação de regras e de desempenho.

A utilização de ferramentas livres para reduzir a codificação para a implementação de SOCRATES assemelha-se a abordagens que procuram utilizar representações unificadas para auxiliar o processo de refatoração. O objetivo dessas abordagens é reutilizar ferramentas (e dessa maneira reduzir a codificação) que atuam sobre as representações únicas. Tichelaar et al. [Tichelaar et al. 2000] utilizam um meta-modelo para representar várias linguagens orientadas a objetos e Mendonça et al. [Mendonça et al. 2004] desenvolveram um *framework* centrado em XML para a aplicação de refatorações. SOCRATES é inspirado nestas abordagens, porém não tem o objetivo de ser genérico como elas. SOCRATES é voltado para uma linguagem alvo (no caso, Java).

Nesse sentido, a adaptação de SOCRATES para outras linguagens como COBOL ou C envolve o mesmo processo realizado para Java. Os pontos críticos dessa adaptação são: o mapeamento do código de COBOL ou C para XML e a codificação de regras de inferência para a identificação de maus cheiros próprios dessas linguagens.

5. Conclusões

Neste trabalho foi apresentado SOCRATES — Sistema Orientado para CaRActerização de RefaToraçõEs. O objetivo principal do SOCRATES é identificar de maneira automática oportunidades de refatoração e, dessa forma, reduzir o esforço associado à

aplicação de refatorações. Além desse objetivo, o projeto e a arquitetura de SOCRATES visam utilizar ferramentas de código aberto para reduzir a codificação necessária para identificar maus cheiros no código. A versão atual identifica o mau cheiro *parâmetro obsoleto* utilizando estes conceitos de projeto e de arquitetura.

Para atingir estes objetivos, SOCRATES utiliza componentes de infra-estrutura disponíveis em ferramentas livres (*Jikes*, *Digester* e *Drools*) e componentes (*Amarração XML*, *Meta-Info*, *Análise* e *Regras de Inferência*) especialmente desenvolvidos. Os componentes de SOCRATES requereram pouca codificação e basicamente orquestram o funcionamento dos componentes de infra-estrutura. A codificação necessária ficou concentrada nas *Regras de Inferência*. Estas regras constituem algoritmos que podem ser escritos em algumas linguagens de programação, entre elas Java. É importante observar que os algoritmos podem ser implementados de maneira eficiente para a detecção de oportunidades de refatoração.

SOCRATES foi construído tendo-se em mente que sua inteligência está nos algoritmos programados em Java e no fluxo de execução, ou ordem de disparo, destes algoritmos. Como decorrência desse *rationale*, a inclusão de regras para a identificação de novas oportunidades de refatoração é facilitada. É importante notar que utilizando esta solução é possível combinar diferentes algoritmos e técnicas para a tomada de decisão sobre a existência ou não de oportunidades de refatoração.

A primeira versão do SOCRATES abre perspectivas de vários trabalhos adicionais descritos a seguir.

- Estudo de caso com programas de maior porte. A presente versão do SOCRATES foi avaliada apenas para provar o conceito. No entanto, é importante verificar a eficácia do algoritmo implementado para detecção de parâmetros obsoletos em sistemas como muitas classes e métodos com diferentes assinaturas.
- Inclusão de novas oportunidades de refatoração e aplicação de refatorações. É importante adicionar ao SOCRATES a identificação de novas oportunidades de refatoração mais complexas (por exemplo, que envolvam a utilização de métricas) para verificar se sua arquitetura é flexível para incluí-las. Outra funcionalidade a incluir é a aplicação automática de refatorações. Regras podem ativar a aplicação de refatorações que solucionam os maus cheiros identificados.
- Integração das ferramentas. Os componentes do SOCRATES não estão integrados. Um trabalho importante é integrar estes componentes como um *plugin* de um ambiente integrado de desenvolvimento de programas Java (e.g., Eclipse).

Referências

Badros, G. (2000). JavaML: a markup language for Java source code. *Computer Networks*, 33:159–177.

Balazinska, M., Merlo, E., Dagenais, M., Lague, B., and Kontogiannis, K. (2000). Advanced clone-analysis to support object-oriented system refactoring. In *Working Conference on Reverse Engineering*, pages 98–107.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional.

Carneiro, G. F. and Mendonça Neto, M. G. (2003). Relacionando refactorings e métricas de código fonte — Um primeiro passo para a detecção automática de oportunidades de refactoring. In *Anais do 17o. Simpósio Brasileiro de Engenharia de Software*, pages 51–66, Manaus, AM.

Declarativa (2006). http://www.declarativa.com/interprolog/. Visitado em 02 de maio de 2006.

Digester (2005). http://jakarta.apache.org/commons/digester/.Visitado em 11 de outubro de 2005.

Drools (2005). http://drools.org/. Visitado em 11 de outubro de 2005.

Eclipse (2005). http://www.eclipse.org/. Visitado em 11 de outubro de 2005.

Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. (2001). Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

Jikes (2005). http://jikes.sourceforge.net/. Visitado em 05 de setembro de 2005.

Kataoka, Y., Ernst, M. D., Griswold, W. G., and Notkin, D. (2001). Automated support for program refactoring using invariants. In *Proceedings of the International Conference on Softaware Maintenance (ICSM)*, pages 736–743.

Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance an evolution. *IEEE Computer*, 28(8):44–55.

Mendonça, N. C., Maia, P. H. M., Fonseca, L. A., and Andrade, R. M. C. (2004). Building flexible refactoring tools with xml. In *Anais do 18o. Simpósio Brasileiro de Engenharia de Software*, pages 178–192, Brasília, DF.

Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

Moura, A. T. P. (2006). *SOCRATES — Sistema Orientado a objetos para CaRActerização de RefaToraçõEs*. Trabalho final, Mestrado Profissional, Instituto de Computação, Universidade Estadual de Campinas, Campinas.

Opdyke, W. F. (1992). *Refactoring Object-oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign.

Pigoski, T. M. (1997). *Practical software maintenance: best practices for managing your software investment*. John Wiley.

Tichelaar, S., Ducasse, S., Demeyer, S., and Nierstrasz, O. (2000). A meta-model for language-independent refactoring. In *Proceedings of the International Symposium on Principles of Software Evolution (ISPSE)*, pages 157–167.

Tourwé, T. and Mens, T. (2003). Identifying refactoring opportunities using logic meta programming. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR '03)*, pages 91–100.

Estudos Experimentais em Manutenção de Software: Observando Decaimento de Software Através de Modelos Dinâmicos

Marco Antônio Pereira Araújo, Guilherme Horta Travassos

UFRJ - Universidade Federal do Rio de Janeiro
COPPE – Programa de Engenharia de Sistemas e Computação

{maraujo, ght}@cos.ufrj.br

Abstract. *Due to the ever growing necessities of evolution in Software Systems, it is necessary to advance towards the understanding on how these systems suffer changes, so as to be more easily modified to accommodate these needs, throughout successive cycles of evolutive maintenance. In this fashion, an environment for experimental studies on software evolution is underway, utilizing simulation techniques based on Dynamic Systems models, in order to provide a pathway to evaluate the applicability of the Laws of Software Evolution in the context of object-oriented software. These Laws describe how a system behaves through its versions, regarding its decaying. The application of System Dynamics techniques is justified by the fact that evolving systems present a dynamic behavior, non-static, which needs to be considered. In this context, the semi-quantitative analysis offers the possibility of data analysis through trends of pre-determined software characteristics, providing for the observation of the behavior of evolving systems, as proposed in the environment presented in this work.*

Resumo. *Em função das necessidades crescentes de evolução em Sistemas de Software, torna-se necessário avançar no entendimento de como estes sistemas sofrem mudanças, de forma que possa ser modificado mais facilmente para acomodar estas necessidades, em seus sucessivos ciclos de manutenção evolutiva. Neste sentido, um ambiente para estudos experimentais em evolução de software está em construção, utilizando-se de técnicas de simulação baseadas em modelos de Dinâmica de Sistemas, no sentido de prover um caminho para avaliar a aplicabilidade das Leis de Evolução de Software no contexto de software orientado a objetos. Estas Leis descrevem como um sistema se comporta através de suas versões, considerando seu decaimento. A aplicação de técnicas de Dinâmica de Sistemas justifica-se pelo fato que sistemas em evolução possuem um comportamento dinâmico, não estático, que precisa ser considerado. Neste contexto, a análise semiquantitativa oferece a possibilidade de análise de dados através de tendências de características de software pré-determinadas, possibilitando a observação do comportamento de sistemas em evolução, conforme proposto no ambiente apresentado neste trabalho.*

1. Introdução

Uma questão recorrente no desenvolvimento de software é que sistemas mudam, seja por necessidade de atendimento a novos requisitos, seja por correções de defeitos identificados após a entrega dos produtos de software. Assim, a manutenção torna-se um desafio à qualidade do produto desenvolvido, podendo levar ao seu decaimento e dificultando a realização de futuras manutenções que, numa situação limite, pode levar à descontinuidade do produto (Dias et al., 2003).

Desta forma, um maior conhecimento sobre este processo de mudança torna-se necessário, no sentido de facilitar a acomodação de novas funcionalidades, diminuindo os riscos associados à perda de qualidade dos sistemas neste processo. Nesta perspectiva, a construção de um modelo que permita observar as principais características que podem levar ao decaimento de sistemas em evolução, simulando seu comportamento, intenciona contribuir para um maior entendimento deste processo evolutivo (Araújo e Travassos, 2005a) (Araújo e Travassos, 2006).

A construção deste modelo, em virtude da característica dinâmica de sistemas em evolução, envolve a estruturação de um ambiente apoiado por estudos experimentais em evolução de sistemas, no sentido de observar e simular tendências das características de software (i.e. tamanho, complexidade ou periodicidade) que possam afetar sua qualidade e determinar, desta forma, seu decaimento. A análise de dados absolutos coletados de sistemas reais através de métricas pré-estabelecidas e relacionadas a estas características de software não representa necessariamente o comportamento evolutivo futuro destes sistemas. Na verdade, permite apenas observar o comportamento passado, já ocorrido. Entretanto, o que se precisa, do ponto de vista gerencial, é tentar prever os riscos futuros das modificações que deverão ser aplicadas ao software baseado nestas tendências de comportamento observadas. Assim, tendências de crescimento ou diminuição das características de software podem oferecer informações relevantes sobre o comportamento do sistema em observação.

Desta forma, este trabalho apresenta a proposta de um ambiente que vem sendo construído na COPPE/UFRJ para observar o comportamento relacionado ao decaimento de software através da execução de estudos experimentais. Como base para observar o comportamento de sistemas em evolução, modelos lógicos que permitem a simulação de sucessivos ciclos de manutenção evolutiva, baseado em técnicas de Dinâmica de Sistemas, foram definidos. Para fortalecer a representação do comportamento dinâmico e a análise de informações considerando suas tendências de crescimento ou decaimento ao invés de seus valores absolutos, o ambiente explora a análise semiquantitativa de dados.

Este artigo é organizado em mais três seções além desta introdução. Na seção 2, é apresentado o ambiente para estudos experimentais em evolução de software, bem como sua arquitetura. A seção 3 descreve a utilização de técnicas de Dinâmica de Sistemas no contexto deste ambiente e, por fim, a seção 4 apresenta as considerações finais e perspectivas de trabalhos futuros.

2. Ambiente para Execução de Estudos Experimentais em Evolução de Software

Nos trabalhos de Araújo e Travassos (2005a), Araújo e Travassos (2005b), Araújo et al., (2005) e Araújo e Travassos (2006) apresenta-se um ambiente baseado no referencial teórico das Leis de Evolução de Software (LSE – *Laws of Software Evolution*) (Lehman, 1980) (Lehman e Ramil, 2002) (Lehman e Ramil, 2003) cujo objetivo é possibilitar a observação de como sistemas orientados a objetos se comportam ao longo de sucessivos ciclos de manutenção evolutiva, de forma a proporcionar uma melhor compreensão de seu processo de decaimento. Cada Lei de Evolução de Software apresenta uma formulação lógica que representa sua hipótese de observação baseada na sua definição original (Araújo et al., 2005).

Estas formulações lógicas representam a base para a observação de sistemas em decaimento, através de características de software previamente determinadas (i.e. tamanho, periodicidade, complexidade, esforço, modularidade, confiabilidade, eficiência, manutenibilidade), sendo consideradas como hipóteses para observação do comportamento das Leis de Evolução de Software, necessitando de investigação experimental (Araújo et al., 2005). Entretanto, uma formulação lógica verdadeira associada à hipótese de observação de uma LSE específica não necessariamente é resultado de decaimento de software. Assim, as LSE foram discutidas no sentido de identificar sua característica neutra (que revela uma situação inerente à própria LSE), sua implicação positiva (que revela uma situação oposta ao decaimento de software), e sua implicação negativa (que resulta em decaimento de software e descreve os interesses deste trabalho). Desta forma, as hipóteses de observação consideradas neste trabalho são baseadas nas implicações negativas para cada Lei de Evolução de Software. A Tabela 1 apresenta, como exemplo, estas questões relativas à Lei de Crescimento Contínuo. Estas discussões, relativas a cada uma das LSE, encontram-se descritas nos trabalhos de Araújo e Travassos (2005a), Araújo et al. (2005) e Araújo e Travassos (2006).

Tabela 1 - Características da Lei de Crescimento Contínuo

LSE	Definição / Formulação Lógica / Implicações Neutra, Positiva e Negativa
Crescimento Contínuo	**Definição:** A capacidade funcional de um sistema *E-type*[1] deve ser continuamente incrementada para manter a satisfação do usuário ao longo do tempo de vida do sistema
	Formulação Lógica: Tamanho aumenta **E** Periodicidade não aumenta
	Neutro: Sistemas em uso estão sujeitos ao crescimento contínuo **Implicação Positiva**: O incremento de novas funcionalidades pode manter a utilidade do produto, permanecendo útil em relação às mudanças no ambiente e às expectativas dos usuários **Implicação Negativa**: A falta de incremento de novas funcionalidades podem tornar o sistema progressivamente menos útil em relação às necessidades dos usuários

O conjunto das hipóteses de observação, descritas através de formulações lógicas estáticas, descrevem então o comportamento das LSE. Entretanto, torna-se claro que o comportamento de sistemas de software não é estático ao longo de seu ciclo de vida. Assim, através da identificação das ligações e influências entre as LSE, torna-se possível a construção de modelos de Dinâmica de Sistemas (Barros, 2001) no sentido de observar e simular o comportamento de sistemas de software através de sucessivos ciclos de manutenção evolutiva. Partindo desta perspectiva, torna-se possível a

[1] Um sistema *E-type* é um software que resolve um problema ou dirige-se a uma aplicação no mundo real.

realização de estudos experimentais *in virtuo*[2] e *in silico*[3] (Travassos e Barros, 2003). Estudos *in virtuo* são realizados a partir da coleta de métricas e análise das características de software apresentadas a partir da observação de sistemas reais. Estudos *in silico* são realizados a partir da simulação do comportamento das métricas e características de software consideradas, nas diferentes fases do processo de desenvolvimento. Desta forma, o ambiente proposto permitirá a realização destes tipos de estudos experimentais através de técnicas de Dinâmica de Sistemas, ao longo das etapas de seu processo de desenvolvimento. A Figura 1 ilustra a arquitetura proposta para o ambiente, no sentido de realizar estes estudos experimentais.

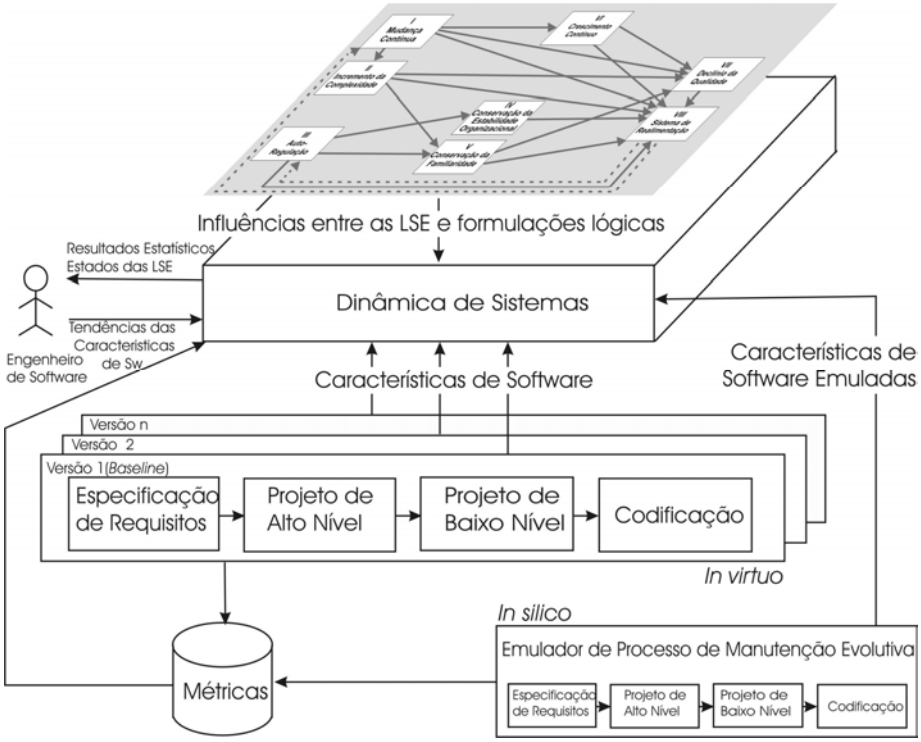


Figura 1 – A arquitetura do Ambiente

Além disso, as Leis de Evolução de Software não são independentes (Kitchenham, 1982) (Lehman e Ramil, 2001). Assim, as hipóteses apresentadas através de formulações lógicas, definidas pelas tendências (aumento, diminuição, constância) das características de software, foram discutidas no sentido de identificar as influências entre as LSE. Esta discussão é particularmente relevante uma vez que este trabalho tem por objetivo identificar o processo de decaimento de software e não estudar o comportamento individual de cada Lei de Evolução de Software.

Para descrever estas ligações entre as LSE, um diagrama SADT (*Structured Analysis and Design Technique*) (Ross e Schoman, 1977) foi construído, descrevendo como as Leis de Evolução de Software influenciam uma nas outras (Araújo et al., 2005). A estrutura deste diagrama representa *Entradas* que são os dados recebidos por uma atividade, *Saídas* que são os dados gerados ou transformados por uma atividade,

[2] *In virtuo*: estudos experimentais que envolvem interação entre participantes e um modelo computacional da realidade.
[3] *In silico*: estudos experimentais caracterizados por modelos computacionais descrevendo os participantes e mundo real.

Controle que são dados cuja utilização influenciam no processo de transformação de entrada e saída, e *Mecanismo* que representa o processador que realiza ou desempenha a atividade.

Como exemplo, a Figura 2 apresenta um fragmento do diagrama SADT, considerando a Lei de Crescimento Contínuo, sendo:

- **Entrada:** representada pela versão atual do sistema em observação;
- **Saída:** representado pelo estado da Lei de Crescimento Contínuo, uma vez que, em sendo verdadeiro, pode influenciar outras LSE, como a Lei de Declínio da Qualidade[4], considerando que novas funcionalidades podem impactar negativamente na qualidade do sistema, bem como na Lei de Sistema de Realimentação[5], que tem por objetivo analisar os estados das LSE de forma a propiciar realimentação entre elas e o sistema;
- **Controle:** representado pelo estado da Lei de Mudança Contínua[6], considerando-se que a Lei de Crescimento Contínuo pode ser influenciada pela primeira, uma vez que sucessivas mudanças podem acarretar crescimento do sistema, desde que novas funcionalidades sejam adicionadas; pela *baseline* do sistema; e pelas características de software Tamanho (quantidade de artefatos produzidos em cada etapa do ciclo de vida do software proposto) e Periodicidade (intervalo de tempo decorrido entre cada versão produzida de um artefato). Um artefato é um insumo para uma atividade do processo ou um produto gerado por uma atividade;
- **Mecanismo:** representado pela formulação lógica da LSE em observação que, neste caso, considera-se que para sistemas em crescimento contínuo, o Tamanho deve aumentar e, simultaneamente, a Periodicidade não deve aumentar.

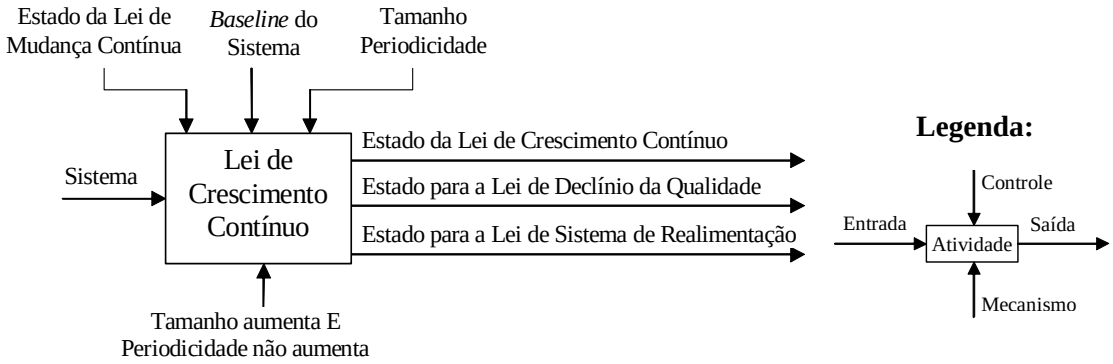


Figura 2 – Diagrama SADT para a Lei de Crescimento Contínuo

[4] **Lei de Declínio da Qualidade:** A menos que seja rigorosamente adaptada às mudanças do ambiente operacional, a qualidade de um sistema *E-type* parecerá declinar enquanto este evoluir (Lehman, 1980).
[5] **Lei de Sistema de Realimentação:** Processos de evolução *E-type* constituem em realimentação em multi-nível, multi-interação e multi-agente do sistema (Lehman, 1980).
[6] **Lei de Mudança Contínua:** Um sistema *E-type* deve ser continuamente adaptado senão torna-se-á progressivamente menos satisfatório em seu uso (Lehman, 1980).

Considerando um processo de desenvolvimento de software OO contendo as etapas de Levantamento de Requisitos, Projeto de Alto e Baixo Nível e Codificação, definiu-se um conjunto de métricas, associadas às características de software, que seriam aplicáveis a cada uma destas etapas (Araújo et al., 2005).

Este conjunto de métricas define um ambiente parametrizável, permitindo flexibilidade para se coletar seus valores e estudar o decaimento de software dentro das etapas do processo de desenvolvimento. Esta abordagem torna mais ameno o estudo da evolução de software, uma vez que pode ser adaptada em função do processo de desenvolvimento utilizado, coletando-se apenas aquelas métricas que puderem ser efetivamente extraídas do processo em uso. As métricas identificadas para a Lei de Crescimento Contínuo, considerando as diferentes etapas de um processo de desenvolvimento, são apresentadas na Tabela 2.

Tabela 2 - Métricas associadas por Característica em cada etapa do Processo de Desenvolvimento de Software considerando a Lei de Crescimento Contínuo

LSE	Etapa	Tamanho	Periodicidade
Crescimento Contínuo	Especificação de Requisitos	• Número de Pontos de Função • Número de Pontos de Casos de Uso	• Intervalo entre Versões
	Projeto de Alto Nível	• Número de Classes • Número de Métodos por Classe	• Intervalo entre Versões
	Projeto de Baixo Nível	• Número de Classes de Domínio • Número de Classes de Suporte • Número de Métodos por Classe • Número de Subsistemas	• Intervalo entre Versões
	Codificação	• Número de Linhas de Código Fonte • Número de Métodos por Classe	• Intervalo entre Versões

As métricas que dão apoio a cada uma das características nas diferentes etapas do processo de desenvolvimento foram baseadas, genericamente, nos trabalhos de Pressman (2002) e Pfleeger (2004). Métricas mais específicas relativas ao contexto do paradigma da orientação a objetos foram baseadas nos trabalhos de Chidamber e Kemerer (1994), Lorenz e Kidd (1994) e Travassos et al. (2001). As métricas apresentadas englobam métricas de processo e produto e devem ser coletadas a cada versão produzida do artefato, compondo uma base histórica de dados de forma a se poder estudar seus efeitos no decaimento de software.

Apesar do foco deste trabalho estar concentrado em metodologias orientadas a objetos, é possível que o ambiente aqui proposto possa ser útil quando considerados outros paradigmas de desenvolvimento, através da utilização de métricas específicas a estes paradigmas. Esta hipótese necessita de investigação experimental e, inicialmente, não faz parte dos objetivos deste trabalho.

3. Simulação do Comportamento das LSE através de Dinâmica de Sistemas

As técnicas da Dinâmica de Sistemas podem ser aplicadas para entender e influenciar a forma com que os elementos de um sistema variam ao longo do tempo (Barros, 2001). Estas técnicas utilizam estratégias de controle com retroalimentação (ciclos de *feedback*) para organizar as informações disponíveis a respeito de um sistema, criando modelos que podem ser simulados em um computador (Forrester, 1991).

Os modelos produzidos através das técnicas da Dinâmica de Sistemas permitem a descrição de características que não são facilmente expressas em outras técnicas de

modelagem. Estas características e sua aplicação na modelagem de processos de desenvolvimento de software são discutidas a seguir (Barros, 2001):

- **Comportamento Endógeno**: a Dinâmica de Sistemas assume que o comportamento de um sistema é provocado pela estrutura formada pela conexão entre seus componentes (Albin, 1997). Assim, os modelos são construídos para representar a estrutura do sistema, enquanto a simulação demonstra seu comportamento. A Dinâmica de Sistemas assume que o comportamento de fatores internos do modelo determina seu comportamento visível;
- **Integração**: pela filosofia de procurar as causas de um desvio de comportamento na estrutura do sistema, os modelos da Dinâmica de Sistemas integram seus diversos "microcomponentes". A integração destes elementos permite a inferência de seu papel no comportamento do sistema como um todo;
- **Sistemas Fechados**: a Dinâmica de Sistemas modela com clareza sistemas fechados, ou seja, sistemas caracterizados por ciclos de realimentação. Sistemas fechados assumem que uma decisão provoca uma cadeia de reações, que pode vir a influenciar as condições que exigiram a decisão original. Ciclos de realimentação são a estrutura básica de todos os sistemas, sendo a causa de grande parte do seu comportamento dinâmico (Whelan, 1996);
- **Causas e Conseqüências Distantes no Tempo**: em diversos sistemas, particularmente em sistemas complexos, as conseqüências de uma decisão podem se ramificar e transparecer apenas muito tempo depois da tomada da decisão. Esta distância no tempo dificulta a percepção da verdadeira origem de um problema, visto que podem existir diversas explicações conflitantes (Weick, 1979). A simulação de modelos de Dinâmica de Sistemas, por considerar fortemente os ciclos de realimentação e permitir o isolamento de variáveis do modelo, tem a capacidade de acompanhar as conseqüências de cada decisão no sistema;
- **Mapeamento de Modelos Mentais**: os modelos da Dinâmica de Sistemas não se baseiam apenas em informações numéricas. Embora a principal ferramenta de diagramação e as técnicas de simulação exijam que os modelos sejam descritos através de equações, que podem ser geradas a partir do conhecimento contido nos modelos mentais dos indivíduos. Esta propriedade, conforme observado por Merril (1995), permite que um modelo de Dinâmica de Sistemas represente comportamentos genéricos, visto que pode gerar modos de referência que ocorrem em condições para as quais não existem dados para a construção de modelos estatísticos.

Os modelos desenvolvidos com as técnicas da Dinâmica de Sistemas podem ser representados através de (Albin, 1997):

- **Diagramas de Causas e Efeitos:** representam o mecanismo mais simples para representação de modelos de Dinâmica de Sistemas. Posteriormente, estes diagramas podem ser refinados para Diagramas de Repositórios e Fluxos, que são mais precisos na representação do modelo;

- **Diagramas de Repositórios e Fluxos:** apresentam um nível de detalhe maior que os Diagramas de Causa e Efeito, forçando o responsável pela modelagem a refinar sua definição da estrutura do sistema.

O ambiente descrito neste trabalho utiliza-se do modelo de Dinâmica de Sistemas, representado através do Diagrama de Causas e Efeitos, com o objetivo de observar o comportamento de sistemas em evolução. Este ambiente é descrito em termos de características de software, com o intuito de generalizar o comportamento para diferentes níveis de abstração no desenvolvimento de software, considerando as etapas de Especificação de Requisitos, Projeto de Alto e Baixo Nível e Codificação. A idéia é que o modelo possa ser instanciado para estes diferentes níveis de abstração através da substituição das características de software por métricas específicas a cada uma destas etapas.

Assim, a Dinâmica de Sistemas mostra-se apropriada para os objetivos de simulação do comportamento das Leis de Evolução de Software, propostos deste trabalho. A Figura 3 apresenta este modelo de Dinâmica de Sistemas, considerando o Diagrama de Causas e Efeitos.

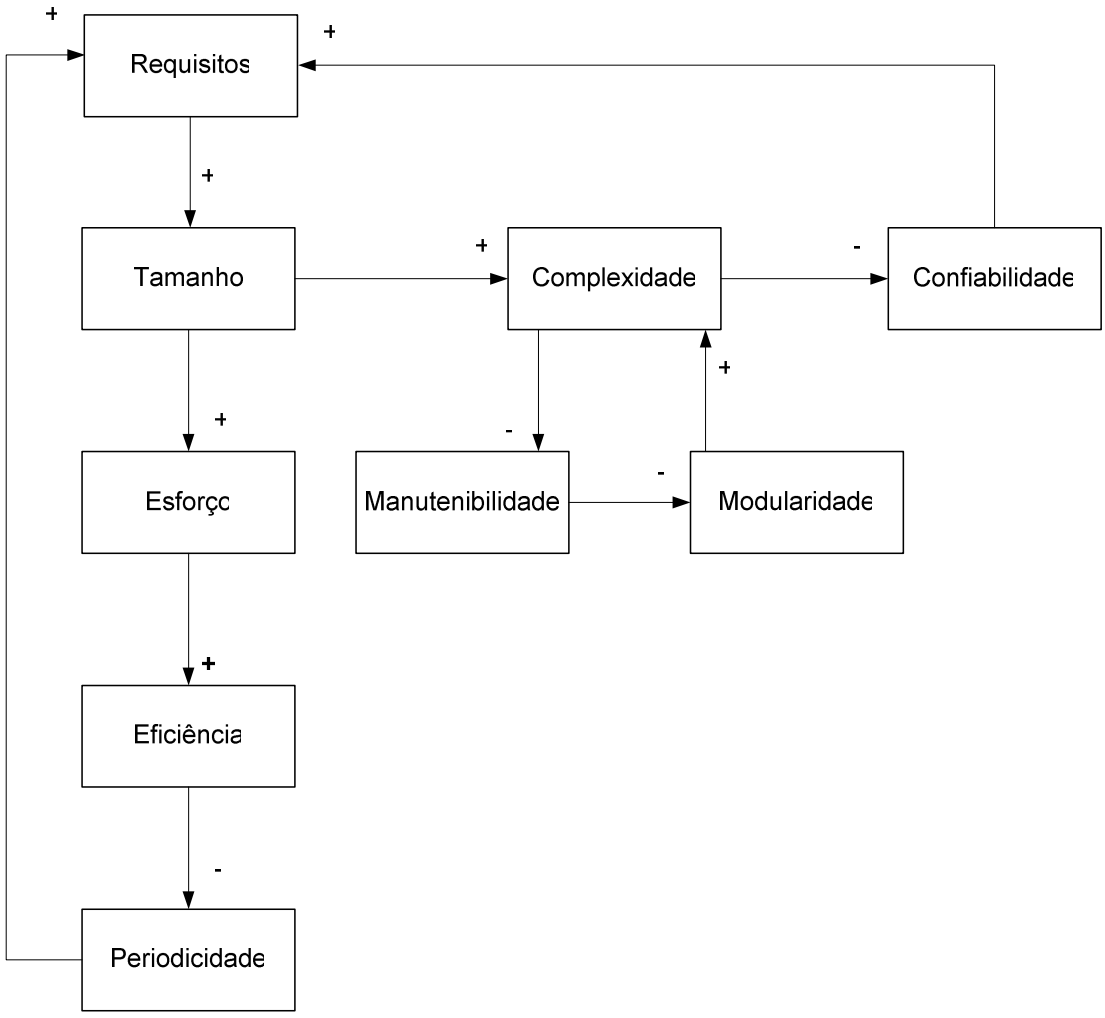


Figura 3 – Diagrama de Causas e Efeitos do Ambiente (hipóteses iniciais)

O modelo apresentado possui três ciclos de *feedback*, sendo:

1. **Requisitos – Tamanho[7] – Esforço[8] – Eficiência[9] – Periodicidade[10]:** à medida que novos requisitos precisam ser atendidos em função de manutenção evolutiva, aumenta o tamanho do sistema, que resulta em um maior esforço de desenvolvimento, tornando necessário o aumento da eficiência da equipe que, por sua vez, contribui para a diminuição da periodicidade entre as versões do sistema em evolução, fazendo com que mais requisitos possam ser atendidos;
2. **Requisitos – Tamanho – Complexidade[11] – Confiabilidade[12]:** à medida que novos requisitos precisam ser atendidos em função de manutenção evolutiva, aumenta o tamanho do sistema, que resulta em uma maior complexidade estrutural do mesmo que, por sua vez, contribui para a diminuição da confiabilidade do sistema e, consequentemente, faz com que novas manutenções sejam necessárias no sentido de atender requisitos de qualidade do sistema;
3. **Complexidade – Manutenibilidade[13] – Modularidade[14]:** um aumento de complexidade estrutural no sistema acarreta a diminuição de sua manutenibilidade que, por sua vez, faz com que a modularidade do sistema diminua, ocasionando aumento na complexidade do sistema.

Os três ciclos apresentados representam ciclos de *feedback* positivo, que se caracterizam por um aumento aplicado sobre uma variável provocar um aumento em outra variável do modelo. Estes tipos de ciclos reforçam o crescimento das variáveis de um modelo. Segundo Santos et al. (2001), modelos de Dinâmica de Sistemas são pobres em sua estrutura quando não possuem ciclos de *feedback*.

Os relacionamentos de Causa e Efeito entre as características de software necessitam de evidências ou resultados de estudos experimentais que os subsidiem. Neste sentido, está sendo aplicada uma abordagem de construção baseada em experimentação (Mafra e Travassos, 2006), com ênfase no planejamento e execução de estudos secundários (revisões sistemáticas) de forma a buscar estes indícios na literatura técnica especializada (Biolchini et al., 2005).

Este Diagrama de Causas e Efeitos representa a base para os estudos experimentais a serem realizados no contexto do ambiente descrito neste trabalho. Apesar deste tipo de diagrama ser considerado um precursor para a construção de um Diagrama de Repositórios e Fluxos, este último exige o conhecimento e manipulação de valores absolutos das variáveis envolvidas. Entretanto, as formulações lógicas

[7] **Tamanho:** caracterizado pela quantidade de artefatos (um insumo para uma atividade do processo ou um produto gerado por uma atividade) produzidos em cada etapa do ciclo de vida do software proposto.
[8] **Esforço:** considerado como o montante de artefatos manipulados (número de inclusões, modificações e exclusões em cada artefato).
[9] **Eficiência:** identificada pela quantidade de pessoas e recursos alocados, tempo consumido e produtividade média da equipe, por cada versão do artefato.
[10] **Periodicidade:** representa o intervalo de tempo decorrido entre cada versão produzida de um artefato.
[11] **Complexidade:** identificada através de elementos que podem medir a complexidade estrutural de um artefato.
[12] **Confiabilidade:** representada pela quantidade de defeitos identificados por artefato em cada versão do software.
[13] **Manutenibilidade:** caracterizada pelo tempo gasto na identificação de defeitos e também no tempo gasto em sua remoção, por versão do artefato.
[14] **Modularidade:** caracterizada através de características de acoplamento e coesão entre artefatos.

construídas para as LSE não estabelecem uma relação entre estas Leis e os valores absolutos de suas características, mas sim a descrição de seus comportamentos através das tendências destas características (aumento, diminuição, constância). Desta forma, o Diagrama de Causas e Efeitos é, num primeiro momento, suficiente para a construção inicial dos modelos de simulação através da Dinâmica de Sistemas.

Esta argumentação leva à definição do tipo de análise estatística de dados que será utilizada nos estudos experimentais. Como o modelo considera a tendência de crescimento ou diminuição das variáveis, ao invés de seus valores absolutos, uma análise quantitativa não se demonstra adequada a este contexto. Segundo Camiletti e Ferracioli (2002) podem-se classificar os processos de análise de dados em três dimensões:

- **Análise Quantitativa:** envolve uma variedade dos aspectos do reconhecimento de simples relacionamentos numéricos tais como comparar conjunto de números até a manipulação de relacionamentos algébricos. Esta dimensão envolve a compreensão de como uma mudança de uma variável afetará outra em um sistema específico;
- **Análise Qualitativa:** envolve fazer distinções entre categorias e tomadas de decisões. Isto pode consistir em examinar um conjunto de escolhas e tomar uma decisão baseada na consideração de suas conseqüências. Conseqüentemente, esta perspectiva exige a observação e a consideração de alternativas e da análise cuidadosa das evidências;
- **Análise Semiquantitativa:** envolve a descrição de situações onde o sentido de uma mudança em uma parte de um sistema é conhecido, mas não o tamanho do efeito desta mudança em outras partes. A análise destes efeitos pede a compreensão do sentido do relacionamento causal - aumento ou diminuição - mas não do conhecimento dos valores numéricos. A análise semiquantitativa explora o fato que tanto a análise quantitativa quanto a qualitativa não capturam todos os aspectos importantes de um sistema. Conseqüentemente, a construção de modelos de uma maneira semiquantitativa pode ser baseada em uma visão de pensamento sistêmico, que demanda que a compreensão de que o comportamento de um sistema é baseado nos relacionamentos causais entre as variáveis que o descrevem. Neste sentido, a causalidade desempenha um papel fundamental na modelagem semiquantitativa, sendo um papel importante no estabelecimento dos relacionamentos entre as variáveis do modelo.

Segundo Santos et al. (2001), o desenvolvimento de um modelo conceitual torna possível pensar sobre os principais eventos, objetos, processos e conceitos a serem traduzidos em variáveis e estabelecer então as interações possíveis da situação em estudo. A tradução do modelo conceitual em um semiquantitativo é possível, através de simulação, ao intuir sobre as tendências de comportamentos dinâmicos previstos e de possíveis relações matemáticas entre variáveis. A construção de um modelo quantitativo pressupõe um estudo rigoroso baseado em um processo iterativo envolvendo as representações conceituais e semiquantitativas para a compreensão da passagem da abordagem semiquantitativa para a quantitativa.

Desta forma, os relacionamentos entre variáveis que descrevem um sistema podem ser entendidos e representados através de uma representação Causa-Efeito,

quando considerando uma análise semiquantitativa, e simulados através de modelos de Dinâmica de Sistemas.

Assim, o modelo apresentado na Figura 3 é a base para a construção do modelo de Dinâmica de Sistemas objeto de pesquisa deste trabalho, utilizando-se de uma abordagem semiquantitativa. A Figura 4 apresenta este modelo utilizando a ferramenta WLinkIt (Camiletti e Ferracioli, 2002), um ambiente de modelagem computacional semiquantitativo.

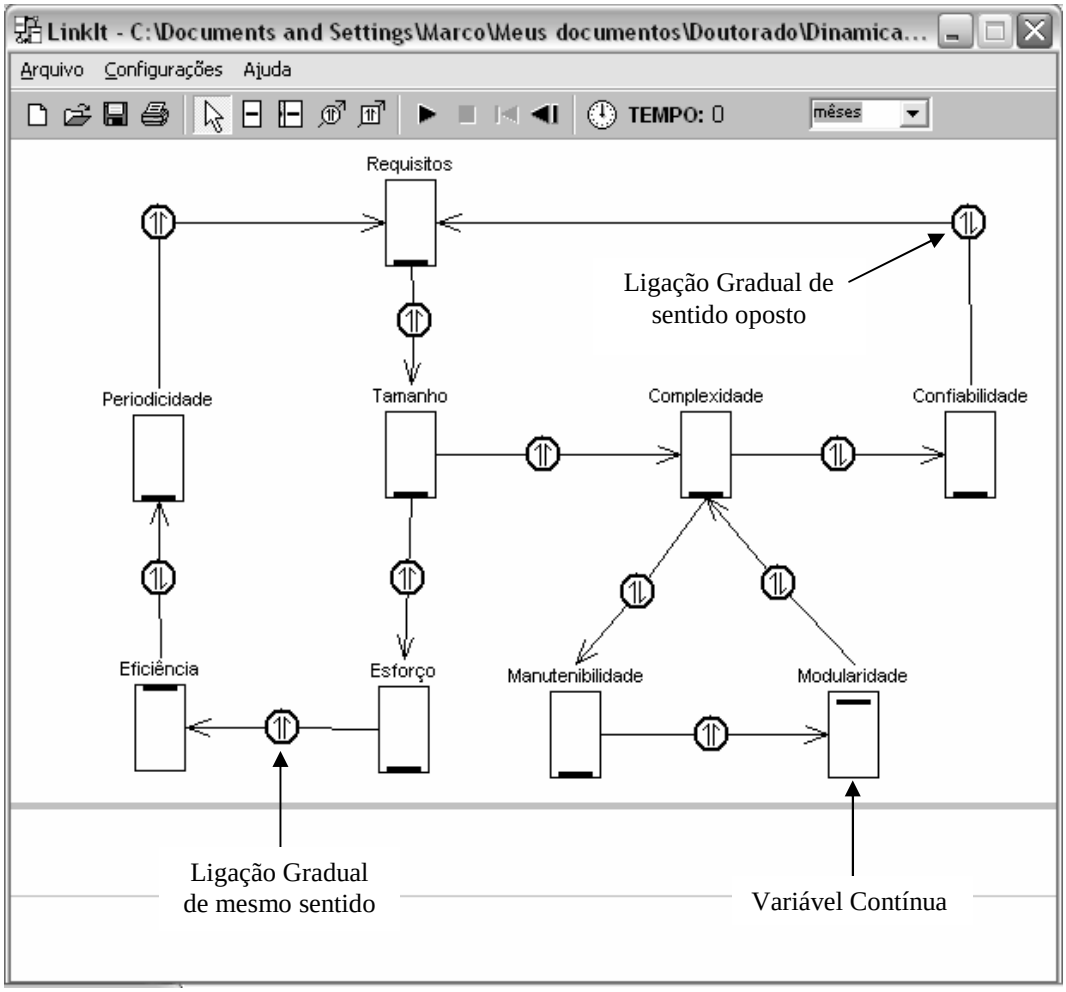


Figura 4 – Modelo semiquantitativo construído na ferramenta WLinkIt

Através deste modelo pode-se observar a relação de causa e efeito entre as variáveis consideradas, objetivando verificar a relação entre elas. No caso de observação de sistemas reais, estas variáveis podem ser substituídas por valores coletados de métricas disponíveis para cada etapa do processo de desenvolvimento de software considerado, caracterizando um experimento *in-virtuo*. No caso de simulação de futuras versões de um sistema, pode-se simular tanto as características de software como as próprias métricas, caracterizando um experimento *in-silico*.

Este modelo ainda tem o objetivo de servir de base para outro, mais genérico, representando o comportamento das Leis de Evolução de Software. Como cada LSE foi descrita através de uma formulação lógica envolvendo suas características de interesse, e também das influências entre as Leis (Araújo et al., 2005), o modelo da Figura 3 serve

ainda de subsídio para a observação do comportamento das LSE em função da simulação de suas características de software.

4. Considerações Finais e Trabalhos Futuros

Existem duas maneiras possíveis de trabalhar com construção de modelos baseados em Dinâmica de Sistemas (Santos et al., 2001). A primeira é através de atividades exploratórias onde o usuário explora um modelo ou uma representação previamente modelada. Neste caso, o usuário interage com as simulações e está limitado à manipulação dos parâmetros, o que promove habilidades básicas de pesquisa tais como o entendimento e variações de causalidade, e pode diretamente influenciar a aquisição de conhecimento, exigindo um nível mais baixo de conhecimento do usuário. A outra é através de atividades expressivas, onde o usuário apresenta ou expõe sua visão ou o modelo mental da situação que está sendo modelada. A construção de modelos promove habilidades de resolução de problemas mais gerais e a transferência destas habilidades para outras áreas de pesquisa.

Assim, espera-se experimentalmente avaliar a aplicabilidade das Leis de Evolução de Software no contexto de processos de desenvolvimento de software orientado a objetos através de um ambiente que permita estudar as causas de decaimento e suas conseqüências no processo de desenvolvimento de software. A realização de estudos experimentais pode oferecer algum conhecimento a respeito da viabilidade de construir modelos da decaimento de software usando as técnicas da simulação baseadas nos modelos de Dinâmica de Sistemas, o que permitirá simular o comportamento das hipóteses estabelecidas.

Em Kahen et al. (2001) encontra-se descrito um modelo de Dinâmica de Sistemas para investigação do processo de evolução de software. Entretanto, este modelo não referencia diretamente as Leis de Evolução de Software, apesar de basear-se neste conhecimento para sua formulação. O modelo apresentado tem a intenção de analisar o crescimento de sistemas e explorar as conseqüências de diferentes níveis de trabalho anti-regressivo. O objetivo é prover um modelo que, uma vez refinado e calibrado para representar um processo do mundo real e seu ambiente, possa ser usado como uma ferramenta para explorar o impacto de diferentes políticas e suporte a tomadas de decisão. A limitação deste modelo, quando comparado à proposta deste trabalho, baseia-se que está focado apenas na etapa de codificação, ignorando o processo de evolução de software nas demais etapas de desenvolvimento. Uma contribuição importante é a utilização de técnicas anti-regressivas como um processo dentro do modelo de dinâmica de sistemas. A utilização destas técnicas será objeto de futuros estudos no contexto do ambiente apresentado neste trabalho.

As próximas atividades incluem a execução de alguns estudos experimentais em sistemas de informação com dados disponíveis para avaliar as hipóteses formuladas, de acordo com o processo de experimentação definido por Amaral e Travassos (2003). Para estes estudos, um ambiente que oferece suporte ferramental para estudos experimentais está sendo definido e construído (Mian et al., 2004). Além disso, necessita-se considerar características de rejuvenescimento de software, como redocumentação, reestruturação, engenharia reversa e reengenharia (Pfleeger, 2004).

Agradecimentos

Reconhecemos a colaboração e a contribuição dos Profs. Bárbara Kitchenham e Márcio de Oliveira Barros, representados através dos valiosos comentários e material técnico provido.

Esta pesquisa recebe apoio do CNPq (Grant 472135/2004-0) no contexto do projeto eSEE.

Referências Bibliográficas

Albin, S., 1997. Building an System Dynamics Model Part 1: Conceptualization, IN: Relatório Técnico D-4597, MIT System Dynamics Group, Cambridge, MA

Amaral, E.A.G.G., Travassos, G.H., 2003, "A Package Model for Software Engineering Experiments", IEEE International Symposium on Empirical Software Engineering. Proceedings of the ISESE 2003. IEEE Computer, 2003. v.II. p.21 – 22.

Araújo, M.A.P., Travassos, G.H., 2005a, "Aplicação das Leis de Evolução de Software em Manutenção Evolutiva", Revista RTInfo – Edição Especial sobre Novas Abordagens para Manutenção de Software.

Araújo, M.A.P., Travassos, G.H., 2005b, "A Framework for Experimental Studies Planning in Object-Oriented Software Decay", 8º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes de Software - IDEAS'05, Valparaíso, Chile.

Araújo, M.A.P., Travassos, G.H., 2006, "An Environment to Observe Object-Oriented Software Evolution", Jornadas Iberoamericana de Ingeniería de Software e Ingeniería del Conocimiento - JIISIC'06, Puebla, México.

Araújo, M.A.P., Travassos, G.H., Kitchenham, B., 2005, "Evolutive Maintenance: Observing Object-Oriented Software Decay", Technical Report ES-672/05 - COPPE/UFRJ.

Barros, M.O., 2001. Gerenciamento de Projetos Baseados em Cenários: Uma Abordagem de Modelagem Dinâmica e Simulação. Tese de Doutorado, COPPE/UFRJ.

Biolchini, J., Mian, P.G., Natali, A.C., Travassos, G.H., 2005. Systematic Review in Software Engineering: Relevance and Utility. Technical Report. PESC - COPPE/UFRJ. Brazil. Disponível em: http://cronos.cos.ufrj.br/publicacoes/reltec/es67905.pdf.

Camiletti, G.G., Ferracioli, L., 2002, "The Use of Semiquantitative Computational Modelling in the Study of Predator-Pray System". X International Organization for Science and Technology Education, 2002, Foz do Iguaçu.

Chidamber, S.R., Kemerer, C.F., 1994. "A Metrics Suite for Object Oriented Design". IEEE Transactions on Software Engineering, vol. 20, No. 6, p. 476 – 493, ISSN: 0098-5589.

Dias, M.G.B, Anquetil, N., Oliveira, K.M., 2003. "Organizing the Knowledge Used in Software Maintenance". Journal of Universal Computer Science, vol. 9, no. 7, 641-658.

Forrester, J.W., 1991, System Dynamics and the Lessons of 35 Years, Relatório Técnico D-4224-4, MIT System Dynamics Group, Cambridge, MA.

Kahen, G., Lehman, M. M., Ramil, J.F., Wernick, P., 2001. "System dynamics modelling of software evolution processes for policy investigation: Approach and example". Journal of Systems and Software, 2001; 59 (3):271-281.

Kitchenham, B., 1982. "System evolution dynamics of VME/B". ICL Tech. J., 42-57.

Lehman, M.M., 1980, "Programs, Life Cycle and the Laws of Software Evolution", Proc IEEE Spec. Iss. on S.E., vol. 68, no. 9, pp. 1060 -1076. Sept l980. CCD Res. Rep. 80/7.

Lehman, M.M., Ramil, J.F., 2001. "Rules and Tools for Software Evolution Planning and Management". Annals of Software Engineering, November, vol. 11, no. 1, pp. 15-44 (30).

Lehman, M.M., Ramil, J.F., 2002, "An Overview of Some Lessons Learnt in FEAST", WESS'02 Eighth IEEE Workshop on Empirical Studies of Software Maintenance, Montreal.

Lehman, M.M., Ramil, J.F., 2003, "Software Evolution – Background, Theory, Practice", Information Processing Letters, vol. 88, pp. 33 – 44.

Lorenz, M., Kidd, J., 1994. Object-Oriented Software Metrics. Prentice-Hall.

Mafra, S.N., Travassos, G.H., 2006. "Estudos Primários e Secundários Apoiando a Busca por Evidência em Engenharia de Software". Relatório Técnico ES-687/06, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

Merril, D., 1995. Training Software Development Project Managers with a Software Project Simulator, Proposta de Tese de Mestrado, Arizona State University, Tempe, AZ, disponível na URL http://www.eas.asu.edu/~sdm.

Mian, P. G., Travassos, G. H., Rocha, A. R. C., 2004. "Towards a Computerized Infrastructure for Experimental Software Engineering". *Workshop de Teses e Dissertações em Engenharia de Software - WTES'04 – Simpósio Brasileiro de Engenharia de Software-SBES'04*.

Pfleeger, S.L., 2004. Engenharia de Software – Teoria e Prática, 2a. Ed., Prentice Hall.

Pressman, R.S., 2002. Engenharia de Software, 5a. Ed., Mc-Graw Hill.

Ross, D.T., Schoman, K.E., 1977. "Structured Analysis for Requirements Definition". IEEE Transactions on Software Engineering, v. 3, n. 1, p. 6-15, jan.

Santos, A.C.K., Sampaio, F.F., Ferracioli, L., 2001, "An Experiment Using The Hexagon Technique With Semiquantitative Computer Modeling". Proceedings of the 19th International Conference of the System Dynamics Society. Albany, New York, USA.

Travassos, G.H., Barros, M.O., 2003, "Contributions of In Virtuo and In Silico Experiments for the Future of Empirical Studies in Software Engineering", Proc. of the WSESE03, Fraunhofer IRB Verlag, Rome.

Travassos, G.H., Shull, F., Carver, J., 2001. “Working with UML: A Software Design Process Based on Inspections for the Unified Modeling Language”. Advances in Computers. San Diego: , v.54, n.1, p. 35 - 97.

Weick, K.E., 1979. The Social Psychology of Organization, 2 ed., Reading, MA: Addison-Wesley Publishing Co.

Whelan, J.G., 1996. Beginner Modeling Exercises Section 2: Mental Simulation of Simple Positive Feedback, Relatório Técnico D-4487, MIT System Dynamics Group, Cambridge, MA.

Manutenção Corretiva Baseada em Padrões e Antipadrões de Casos de Uso

Ascânio Zago Júnior[1], Maria Alice Grigas Varella Ferreira[1]

[1]Departamento de Engenharia de Computação e Sistemas Digitais – Escola Politécnica – Universidade de São Paulo – São Paulo, SP – Brasil.

ascaniozago@yahoo.com, maria.alice.ferreira@poli.usp.br

Abstract. *The deployment of use cases has been growing rapidly as the main technique that offers an intuitive and systematic mean to describe business processes and the capture of functional requirements. Concerning the development of object-oriented systems, use cases drive the whole software development process. Therefore, in a process for object-oriented systems driven by use cases, the use cases inspection is essential to the capture of errors. The objective of this paper is to present a strategy for the review of the use cases, employing use case patterns and anti-patterns, that emerged as the result of a work that had as goal the review of models related to the development of a Workflow product during corrective maintenance.*

Resumo. *A utilização de casos de uso tem crescido rapidamente como a principal técnica que oferece um meio sistemático e intuitivo para descrever processos de negócios e de captura de requisitos funcionais. No desenvolvimento de software orientado a objetos, casos de uso direcionam todo o processo de desenvolvimento de software. Assim, num processo de software orientado por casos de uso, a inspeção de casos de uso é essencial para a captura erros. O objetivo desse trabalho é apresentar uma estratégia de revisão de casos de uso baseada em padrões e anti-padrões de casos de uso, que resultou de um trabalho de revisão dos modelos de desenvolvimento de um produto de Workflow, durante a manutenção corretiva.*

1. Introdução

Segundo Sommerville (2003), o termo manutenção corretiva "é universalmente utilizado para se referir à manutenção para o reparo de defeitos". Apesar de ser considerada como um dos casos menos dispendiosos de manutenção, muitas vezes, quando a análise de requisitos ou o projeto da arquitetura não foram realizados com o devido cuidado, os custos podem crescer assustadoramente, implicando muitas vezes em reprojeto do sistema.

Para Jacobson et al. (1997), a engenharia de software é um processo de construção de vários modelos relacionados entre si. Cada modelo define um aspecto do sistema e é descrito usando vários diagramas e documentos. Cada modelo é examinado ou manipulado por pessoas diferentes, com diferentes interesses específicos, papéis e tarefas a quem se denomina *Stakeholder*. Na Orientação a Objetos (OO), um dos primeiros modelos a ser desenvolvido é o modelo de casos de uso. Um caso de uso especifica o comportamento de um sistema computacional – ou de uma de suas partes -

e descreve uma seqüência de ações realizadas para produzir um resultado de valor para um usuário [Booch Rumbaugh Jacobson 2000].

No desenvolvimento de software orientado a objetos, casos de uso direcionam todo o processo de desenvolvimento, auxiliando as atividades do projeto, que incluem a criação, verificação e validação (V&V) da arquitetura do sistema. Recentes estudos mostram que, dentre os processos de V&V, a inspeção de software – que analisa e verifica as representações do sistema, dentre elas os diagramas de projeto – é uma das técnicas mais efetivas para a eliminação de defeitos do sistema [Sommerville 2003]. As técnicas de inspeção podem ser aplicadas a todos os modelos gerados no processo de desenvolvimento de software [Mafra Travassos 2005] e, nesse trabalho, elas foram aplicadas na fase de manutenção, em consequência da necessidade de se realizar manutenção corretiva árdua em um dos sistemas desenvolvidos – o SGP – um sistema para controle de Workflow.

Independente dos aspectos gerenciais da inspeção, este trabalho descreve uma estratégia de revisão baseada em padrões e antipadrões de casos de uso, que foi aplicada para a detecção de defeitos específicos no diagrama e nas descrições dos casos de uso do sistema, que mostraram não atender às características exigidas pelos *Stakeholders*. Durante a manutenção, detectou-se que os principais problemas tinham a ver com casos de uso mal projetados e com a baixa qualidade da especificação de requisitos. A estratégia desenvolvida tem uma abordagem prática e focaliza a correção de problemas recorrentes a partir de soluções prontas e reutilizáveis.

A idéia de padrão, inicialmente proposta para a etapa de projeto do ciclo de vida de um sistema por Gamma et al. (1995) e denominada de padrão de projeto (*design pattern*), é aqui aplicada à construção dos casos de uso. A identificação precoce dos padrões que devem ser utilizados permite que se reutilize mais facilmente, nas etapas que se seguem, estruturas de análise, projeto e codificação. Padrões – segundo a conceituação de Gamma e colaboradores - são estruturas recorrentes, comuns a vários sistemas de software diferentes, e que se mostraram eficientes, podendo ser reusadas em situações similares, em novos projetos. Nesse trabalho, entretanto, essas estruturas foram aplicadas para auxiliar na correção dos defeitos identificados na fase de manutenção.

Já antipadrões são soluções ruins, que ao serem adotadas acarretam uma série de problemas [Brown et al. 1998]. Normalmente, considera-se que antipadrões constituem uma extensão aos padrões. Os antipadrões são também soluções encontradas em modelagens reais de sistemas, porém elas trazem prejuízos ao modelo, ao invés de trazerem benefícios. A identificação de antipadrões pode ser muito valiosa durante uma revisão de casos de uso. A identificação imediata de um antipadrão permite que se corrija facilmente a modelagem, empregando-se a solução recomendada para o problema detectado.

Este trabalho está organizado como se segue: na seção 2 descrevem-se rapidamente os conceitos de padrão e antipadrão; na seção 3 classificam-se os padrões em categorias, por afinidade conceitual entre eles; na seção 4 apresenta-se a estratégia de revisão de casos de uso e na seção 5 apresentam-se as conclusões do trabalho.

2. Padrões e antipadrões

"Casos de uso modelam um uso completo do sistema." [Jacobson 1992]. Isso significa que uma instância de um caso de uso contém a execução completa de um uso do sistema, ou seja, a execução das ações dentro do sistema, assim como suas interações com pessoas e sistemas externos (atores).

Dentro da conceituação de padrão proposta por Gamma e colaboradores (1995), um padrão de caso de uso seria uma estrutura de caso de uso típica que poderia ser utilizada em outro sistema, numa situação similar. Assim, padrões para casos de uso estabelecem soluções para problemas de modelagens [Overgaard and Palmkvist 2004].

Para a descrição dos padrões optou-se por um formato com as seguintes seções e que se baseia na proposta de Overgaard e Palmkvist (2004). Esta descrição mostrou-se adequada aos fins de revisão de casos de uso aplicada ao sistema considerado (SGP) [Zago 2005]:

- *Discussão*: descreve a análise envolvida para definição do padrão. Essa seção pode ocorrer para um grupo de padrões ou especificamente para um padrão. No caso de um grupo de padrões, a discussão é válida para todos os padrões. Os padrões utilizados no trabalho de revisão do SGP foram classificados em grupos, conforme características de contexto em que são empregados.
- *Parte do caso de uso afetada ao se aplicar o padrão:* a aplicação do padrão pode afetar duas partes em um caso de uso:
  - Modelo: quando a estrutura do Diagrama de Casos de Uso é afetada.
  - Descrição: quando a estrutura descritiva de um ou mais casos de uso é afetada.
- *Modelo:* diagrama UML, representando como o padrão se apresenta no Modelo de Casos de Uso.
- *Aplicabilidade*: descreve a situação em que o padrão pode ser aplicado.
- *Exemplo*: mostra um exemplo para melhorar o entendimento do padrão discutido. O exemplo pode ser aplicado para o grupo de padrões ou, especificamente, para um único padrão.

Quanto aos antipadrões, caracterizam situações que devem ser evitadas em casos de uso. Geralmente, retratam práticas ruins ou situações mal entendidas pelos desenvolvedores. Uma situação desse tipo é mostrada na Figura 1. Pantoquilho e colaboradores (2003) sugerem que uma seção relativa aos antipadrões associados a um padrão, deveria também ser incluída na descrição dos padrões de casos de uso.

Ao se detectar um antipadrão em um sistema, pode-se aplicar a correção recomendada em cada caso. No caso do antipadrão da Figura 1, a correção é simples. Os dois casos de uso devem ser concatenados em apenas um. Os casos de uso originais e seus relacionamentos devem ser apagados do Diagrama de Casos de Uso. Se existe a necessidade de manter os dois casos de uso em separado no modelo, pode-se definir relacionamentos do tipo inclusão ou extensão entre eles, mantendo o objetivo primário que é existir apenas uma instância de caso de uso.

Antipadrão: *Comunicação entre casos de uso*

Modelo:

Discussão:

Uma instância de um caso de uso nunca irá trocar mensagens com outras instâncias de casos de uso dentro de um mesmo sistema. O modelo apresenta dois casos de uso trocando mensagens. Se isso acontece, significa que nenhuma das instâncias executa um uso completo do sistema, pois ações adicionais são realizadas pela outra instância. Pode-se entender que, se ações de uma outra instância precisam ser executadas para considerar um uso completo do sistema, o caso de uso associado a essa instância deve ser incluído no primeiro caso de uso (relacionamento *include*). De outra forma, o primeiro caso de uso estará incompleto.

É comum observar comunicação entre casos de uso em modelos que foram gerados por desenvolvedores que tentam expressar a estrutura interna do sistema: modela-se o comportamento de uma parte do sistema em um caso de uso e outra parte do comportamento em outro caso de uso. Essa estruturação é errada, pois modelos de casos de uso modelam, por definição, o comportamento do sistema como um todo, e não suas partes. Além disso, modelos de casos de uso não devem expressar qualquer característica da estrutura interna do sistema.

Figura 1. Exemplo de antipadrão – comunicação entre casos de uso

3. Classificando os Padrões em Categorias

Nessa seção, introduzem-se as categorias de padrões que serão utilizadas na estratégia de revisão. Os padrões pesquisados e utilizados nesse trabalho foram divididos em duas categorias, conforme o domínio em que se aplicam:

- *Templates:* padrões que podem ser aplicados para domínios genéricos dentro de um determinado contexto;
- Domínio Específico: padrões que podem ser aplicados para domínios específicos dentro de um determinado contexto.

Dentro de cada uma das categorias, estão listados grupos de padrões. Cada grupo de padrões engloba um ou mais padrões relacionados a um contexto. Por exemplo, o grupo `Múltiplos Atores` engloba os padrões `Papéis Diferentes` e `Papel Comum`. Ambos os padrões referem-se ao contexto relacionado à utilização de vários atores para um mesmo caso de uso. Os padrões e as respectivas categorias e grupos estão divididos conforme a Figura 2.

A literatura apresenta uma vasta coleção de artigos sobre padrões. Geralmente, os padrões encontram-se descritos em catálogos. Overgaard e Palmkvist (2004), adotado nesse trabalho, apresentam um catálogo de padrões que sintetiza boa parte dos padrões pesquisados. Os padrões catalogados são normalmente empregados nas primeiras fases do desenvolvimento. Padrões também estão sendo continuamente apresentados em congressos específicos desta linha de pesquisa como o SugarLoafPlop, apresentado no Brasil (http://sugarloafplop2005.icmc.usp.br/index_pt.html). Outros congressos

similares ocorrem por todo mundo e informações sobre eles podem ser encontradas em http://www.hillside.net.

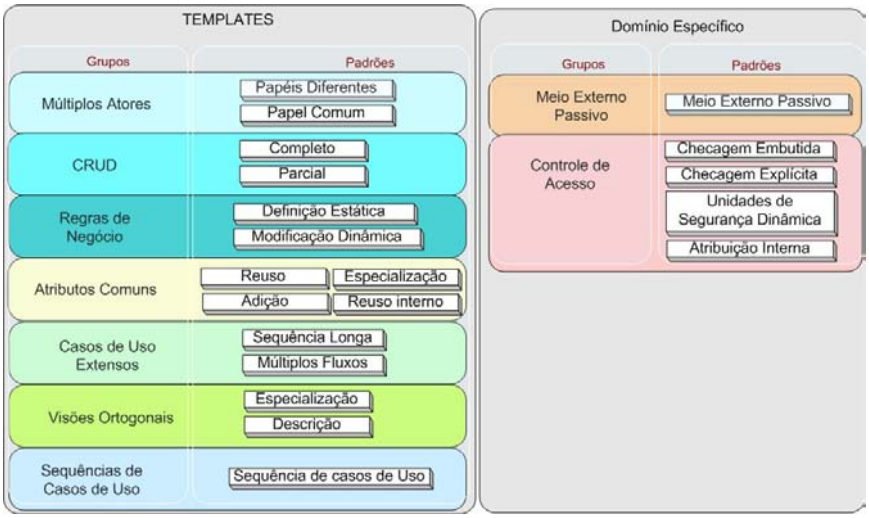


Figura 2. Padrões de casos de uso

4. Estrutura da Estratégia

A estratégia de revisão de casos de uso inicia-se ao se detectar um conjunto de Situações. Uma Situação descreve um acontecimento ou circunstância identificável no diagrama ou nas descrições dos casos de uso. Por exemplo, um caso de uso relacionado a mais de um ator indica uma Situação da estratégia. Para cada Situação deve ser analisado um conjunto de Tópicos, que podem ser aplicados no contexto da Situação. A Tabela 1 mostra um conjunto de Situações e os Tópicos relacionados. A Figura 3 encaminha o uso da estratégia.

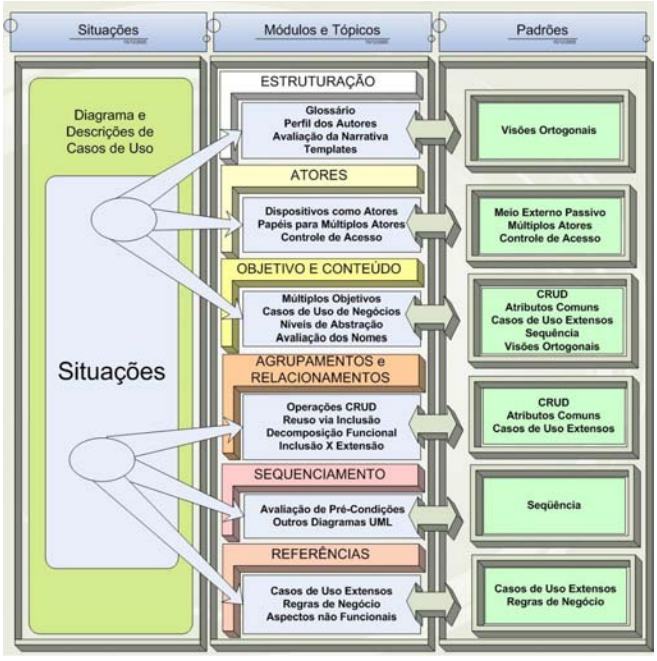


Figura 3. Estrutura de Módulos, Tópicos e Padrões da estratégia

Tabela 1. Descrição das Situações relacionadas aos Tópicos de revisão

Id	Situações	Tópicos a serem analisados
1	Um caso de uso troca informações com um meio externo passivo, como, por exemplo, líquido e gás.	Dispositivos como Atores
2	Um caso de uso relaciona-se a múltiplos atores.	Papéis para Múltiplos Atores
3	Um ator inicia o fluxo de ações e recebe informações de um mesmo caso de uso.	Papéis para Múltiplos Atores
4	Um caso de uso relaciona-se a algum tipo de controle de acesso no sistema.	Controle de Acesso
5	O modelo de casos de uso ou a descrição do fluxo de ações é suficiente na visão de alguns *Stakeholders*, mas incompleto na visão de outros.	Níveis de Abstração Casos de uso de Negócios
6	Um caso de uso tem um número excessivo de *Stakeholders* ou é confuso quanto ao entendimento do objetivo.	Múltiplos Objetivos Casos de uso de Negócios Avaliação dos Nomes
7	Uma funcionalidade do sistema possui operações de criação, leitura, atualização e remoção de informações com fluxos de ações similares.	Operações CRUD
8	Um subfluxo de ações se repete entre vários casos de uso.	Reuso via Inclusão
9	Um fluxo alternativo para um caso de uso é modelado à parte, como outro caso de uso.	Inclusão × Extensão Reuso via Inclusão
10	Um fluxo opcional de um caso de uso é incluído dentro de outro caso de uso.	Inclusão × Extensão Reuso via Inclusão
11	O modelo de casos de uso apresenta um número excessivo de casos de uso que se relacionam entre si.	Decomposição Funcional
12	Um caso de uso apresenta um subfluxo de ações muito pequeno, com uma ou duas linhas de texto.	Decomposição Funcional
13	Um modelo de casos de uso apresenta um relacionamento que indica ordem de execução entre casos de uso	Avaliação de Pré-Condições
14	A ordem de execução entre os casos de uso é complexa e difícil de ser entendida.	Outros Diagramas UML
15	O fluxo de ações de um caso de uso é muito extenso.	Casos de Uso Extensos Regras de Negócio Casos de uso de Negócios
16	O caso de uso tem detalhes da interface do usuário ou qualquer outro requisito que pode ser considerado não funcional	Aspectos não funcionais
17	A descrição de um caso de uso apresenta jargões, conceitos desconhecidos ou termos redundantes.	Glossário
18	A descrição de um caso de uso simples é confusa, pouco objetiva e apresenta erros de sintaxe.	Perfil dos Autores Múltiplos Objetivos
19	O fluxo de ações é descrito no futuro	Avaliação da Narrativa
20	Referência a pontos de Extensão	Avaliação da Narrativa
21	Referência a uma seqüência dentro do mesmo fluxo de ações	Avaliação da Narrativa
22	A estrutura da descrição dos casos de uso não segue um padrão.	*Templates*

À esquerda, na figura 3, encontram-se as Situações que são o ponto de partida para identificação de problemas na elaboração dos casos de uso. No painel central da figura, encontram-se os Tópicos em que a Situação pode se enquadrar, agrupados em grandes unidades denominadas Módulos. Os Tópicos que compõem cada Módulo apresentam conceitos inter-relacionados. No painel da direita encontram-se Padrões que orientam a solução do problema. Cada Módulo é mapeado em um conjunto específico de Padrões, que podem auxiliar na correção dos casos de uso. O Tópico, de maneira geral, direciona

a discussão e análise para um cenário real, que pode abordar o uso de antipadrões, e possíveis soluções, na maioria das vezes, baseadas no uso de padrões de casos de uso.

No estado atual da pesquisa e com base no sistema SGP, detectaram-se 22 Situações que puderam ser identificadas durante o processo de revisão. Cada Situação tem pelo menos um Tópico de revisão a ser considerado.

De maneira geral, a estratégia é muito simples e engloba os seguintes passos:

1) Selecionar na Tabela 1 a Situação apropriada para identificação de problemas no caso de uso;
2) Examinar qual Tópico se aplica a essa situação, dentre os Tópicos apresentados na Tabela 1;
3) Quando aplicável, utilizar um padrão de caso de uso apropriado, selecionado no conjunto de Padrões referentes ao Tópico adotado, conforme mostrado na Figura 3.

Vale ressaltar que a estratégia foi montada a partir de um projeto para um sistema de BPM (*Business Process Management)* e aborda as falhas de modelagem mais comuns em sistemas desse tipo, mas que certamente ocorrem em qualquer projeto. Entretanto, seria impossível montar uma estratégia para revisão que cobrisse todos os aspectos possíveis de todos os projetos e domínios de negócio. Os autores entendem que esse é um trabalho inicial que pode ser complementado com trabalhos futuros, conforme sugerido no item final.

5. Considerações finais

O trabalho de revisão dos casos de uso do SGP mostrou a importância em conhecer profundamente os conceitos que envolvem essa técnica de modelagem, considerando a relevância dos casos de uso para o processo de desenvolvimento.

O formato simples e a escrita em linguagem natural é um dos grandes benefícios na utilização de casos de uso, apesar de que a linguagem natura pode gerar uma séire de problemas como ambigüidades e omissões. Entretanto, como apontado pelo estudo de caso do SGP, foi a aplicação inadequada dos conceitos, com o uso de padrões incorretos, que resultou em um produto com uma série de problemas, principalmente defeitos oriundos da baixa qualidade da especificação, e não a linguagem.

O SGP foi desenvolvido novamente, tendo como base os modelos revisados. O produto final apresentou melhorias excepcionais de qualidade, com grande aceitação do mercado de gerenciamento de processos. Cabe destacar, entretanto que não foram aplicadas métricas, o que impediu a quantificação dos defeitos encontrados.

Considerando a estratégia para revisão como um trabalho inicial, fica como sugestão para trabalhos futuros, a adição de outros Padrões, Tópicos e novas Situações, que podem ser identificadas durante o processo de revisão de casos de uso. Também, pode-se considerar o desenvolvimento de uma ferramenta que apoie o trabalho de revisão e permita a introdução de medições. A complementação de novos itens pode transformar a estratégia para revisão num verdadeiro *framework* para modelagem de casos de uso.

References

Booch G., Rumbaugh, J. and Jacobson, I. (1999) “The Unified Modeling Language User Guide”, Addison-Wesley, Boston.

Booch, G., Rumbaugh, J. And Jacobson, I. (2000) “UML – Guia do Usuário”, Campus, Rio de Janeiro.

Brown, W. J., Malveau, R. C., McCormick III, H. W., Mowbray, T. J. (1998) “AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis”, John Wiley & Sons, Inc., New York.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley, Boston.

Jacobson, I., Griss, M. and Jonsson, P. (1997) “Software Reuse – Architecture, Process and Organization for Business Success”, ACM Press, New York.

Jacobson, I. (1999) “The Unified Software Development Process”, Object Technology Series, Addison-Wesley, Boston.

Jacobson, I. (1992) “Object-Oriented Software Engineering: A Use Case Driven Approach”, Addison-Wesley, Boston.

Mafra, S. N., Travassos, G. H. (2005) “Técnicas de Leitura de Software: Uma Revisão Sistemática”. 19º. SBES - Simpósio Brasileiro de Engenharia de Software, Uberlândia. Anais. v. 1. p. 72-87.

Overgaard, G. and Palmkvist, K. (2004) “Use Cases: Patterns and Blueprints”, Software Patterns Series, Addison-Wesley, Boston.

Pantoquilho, M., Raminhos, R. and Araújo, J. (2003) "Analysis Patterns Specifications: Filling the Gaps",VikingPlop 2003, Bergen, Norway, 18-21 September 2003.

Sommerville, I. (2003), “Engenharia de Software”, Addison-Wesley, São Paulo.

Zago Jr., A. (2005), “Estratégia para Revisão de Casos de Uso Baseada em Padrões”, Dissertação (Mestrado), Escola Politécnica, USP, São Paulo.

Personalização do Processo Gestão de Configuração de Software para Projetos de Manutenção

Pollyana Lima Barreto[1], André Villas Boas[2]

[1]Serviço Federal de Processamento de Dados (SERPRO)
SQAN QD. 601 Módulo G, Regional – 70.836-900 – Brasília – DF – Brasil

[2]CPqD – Telecom & IT Solutions
Rod. SP – 340, KM 118, Campinas – SP – Brasil

pollyana.barreto@serpro.gov.br, villas@cpqd.com.br

Abstract. *This article shows an experience in customizing the Software Configuration Management macroactivity of the SERPRO´s software development process. It highlights a study on software maintenance peculiarities against development activities, to show that the use of a development process to maintenance projects without a previous customization is unefficient, costly and bureaucratic.*

Resumo. *Este artigo relata uma experiência de personalização da macroatividade Gestão de Configuração de Software (GCS) do PSDS – Processo SERPRO de Desenvolvimento de Soluções. Ela aborda um estudo sobre as peculiaridades da manutenção de software, em relação às atividades de desenvolvimento, com a finalidade de mostrar que a utilização de um processo de desenvolvimento para projetos de manutenção, sem uma prévia personalização, é ineficiente, oneroso e torna-se burocrático, tanto para a organização quanto para o cliente.*

1. Introdução

A vida útil de um sistema não acaba no momento de sua entrega ao cliente ou usuário. O sistema final, de uma forma geral, está sujeito a contínuas modificações, mesmo depois de pronto, seja para pequenas correções, seja para introdução de novas necessidades dos usuários. Atualmente a manutenção é a fase do ciclo de vida do software que, normalmente, consome a maior parte dos recursos, tanto financeiros quanto humanos, das organizações de desenvolvimento de software. Segundo [PFLEEGER, 2004], cerca de 80% do orçamento total do ciclo de vida de um software são custos gastos com a manutenção desse software.

Entretanto, a manutenção de software tem recebido menos atenção durante o planejamento do desenvolvimento de um software. Além disso, conforme estudo feito por [DIAS, 2004], observa-se que há um problema cultural em engenharia de software na forma de uma visão um tanto quanto tendenciosa, pois vários segmentos do meio acadêmico e da indústria priorizam o ensino e aprendizado do chamado 'Desenvolvimento de Software', ou seja, a construção de um novo produto. Os processos propostos hoje em dia são, na sua maioria, processos orientados ao desenvolvimento que mesclam a manutenção ao longo do ciclo de vida (estratégia não muito eficaz quando aplicada em sistemas legados).

Este artigo aborda um estudo sobre as peculiaridades da manutenção de software, em relação ao desenvolvimento, com a finalidade de mostrar que a utilização de um processo de desenvolvimento para projetos de manutenção, sem uma prévia personalização, é ineficiente, oneroso e torna-se burocrático, tanto para a organização quanto para o cliente. O artigo está organizado da seguinte forma: na seção 2 discute-se alguns conceitos sobre manutenção de software e suas diferenças em relação ao desenvolvimento; na seção 3 apresenta-se, de forma sucinta, a gestão de configuração de software; na seção 4 é feito um relato de experiência sobre a personalização do processo de gestão de configuração de software dentro do SERPRO (Serviço Federal de Processamento de Dados); e na seção 5 apresentam-se algumas conclusões.

2. Manutenção de *Software*

Qualquer trabalho efetuado para modificar o sistema, depois que ele estiver em operação, é considerado como manutenção [PFLEEGER, 2004]. A manutenção é definida como a totalidade de atividades requeridas para prover suporte de custo efetivo para um sistema de software. As atividades são realizadas tanto durante o estágio de pré-entrega quanto no estágio de pós-entrega [PIGOSKI, 1996]. A manutenção é uma atividade inevitável e pode vir a ser a mais longa fase no ciclo de vida do sistema. Para alguns autores, tais como [CANNIG, 1972], a manutenção deve ser vista como um *iceberg*, onde se espera que o imediatamente visível seja tudo que existe, embora se saiba que uma enorme massa de possíveis problemas e custos fica sob a superfície.

Segundo [PRESSMAN, 2002], a fase de manutenção focaliza as modificações associadas com a correção de erros, com as adaptações necessárias à medida que o ambiente do software evolui, e com as alterações de requisitos do cliente. A fase de manutenção aplica novamente os passos das fases de definição e desenvolvimento, mas o faz no contexto de software existente. Dar manutenção pode ser muito mais complexo do que desenvolver um software novo, pois manter um software cujo domínio é desconhecido, que possui documentação incompleta, um registro superficial de mudanças, dentre outros, é muito mais trabalhoso do que se ele fosse totalmente refeito, seguindo uma metodologia de projeto e convertido, se for o caso, para uma plataforma mais atual, bastante conhecida no mercado.

Conforme [PFLEEGER, 2004], a manutenção é diferente do que quando se desenvolve um sistema, pois além de considerar os produtos do desenvolvimento, deve-se preocupar com o presente para estabelecer uma relação de trabalho com os usuários e operadores, a fim de descobrir se eles estão satisfeitos com o sistema. Também deve ser considerado o futuro, com o objetivo de prever o que pode acontecer de errado, de ponderar sobre as mudanças funcionais requeridas por uma alteração nas necessidades do negócio, e para considerar as modificações no sistema devido a alterações no hardware, software ou nas interfaces.

De acordo com o apresentado, modificação é inevitável, conseqüentemente, faz-se necessário o desenvolvimento de mecanismos para avaliar, controlar e realizar alterações. Uma forma de minimizar os vários problemas que podem ocorrer devido a modificações não controladas é a utilização da gestão de configuração de software.

3. Gestão de Configuração de Software

Como observa [BABICH, 1986], a gestão de configuração é uma disciplina útil para a gerência do projeto, pois lhe permite executar tarefas complexas de uma maneira mais

organizada, o que proporciona maximização de produtividade e minimização de erros, e isso é fundamental na fase de manutenção.

A Gestão de Configuração fornece subsídios para a gestão de todas as outras atividades de engenharia de software [CAPRETZ, 1992]. Cabe a ela identificar a configuração de um sistema em relação ao tempo com a finalidade de, sistematicamente, controlar as mudanças desta configuração, manter sua integridade e, desta forma, possibilitar seu rastreamento através do ciclo de vida do sistema [BERSOFF et all, 1980]. Ela é um dos processos de suporte previstos pelo modelo de ciclo de vida da [ISO 12207] e do ponto de vista da qualidade é uma necessidade para adequação às novas normas internacionais. Segundo o modelo de maturidade CMMI (*Capability Maturity Model Integration*) do SEI (*Software Engineering Institute*), para que uma empresa consiga ascender ao nível 2 do modelo, um dos passos é implementar o processo de gestão de configuração de software.

As funções de gestão de configuração são: identificação de configuração (quais itens constituem uma configuração), controle de configuração (que passos no processo de alteração afetam uma configuração), auditoria de configuração (quais as diferenças entre as versões) e contabilização da situação de configuração (que modificações foram feitas por determinado programador) [ISO 10007] [BERSOFF, 1984].

4. Gestão de Configuração em Projetos de Manutenção no SERPRO

O Serviço Federal de Processamento de Dados (SERPRO) é a maior empresa pública de prestação de serviços em tecnologia da informação do Brasil. Hoje, o SERPRO atua de forma a oferecer aos seus clientes soluções tecnológicas que lhes permitam a concentração nas suas atividades principais, com dados e informações adequadas às suas operações e decisões [SERPRO01]. Para tanto, este definiu internamente o PSDS – Processo SERPRO de Desenvolvimento de Soluções, baseado no RUP (*Rational Unified Process*,) e no modelo CMMI referente aos níveis 2 e 3 de maturidade.

O PSDS define quem faz o que, quando e como, para produzir software de qualidade que atenda às necessidades dos usuários, com custos e prazos aceitáveis e estimáveis e que possua arquitetura com possibilidade de evolução [PSDS]. Em cada macroatividade do PSDS (Requisitos, Gestão de Projetos, Gestão de Configuração, etc.) há uma visão geral, a definição da macroatividade, seus objetivos, as atividades e subatividades relacionadas, os artefatos a serem produzidos e um fluxo das atividades.

A macroatividade Gestão de Configuração de Software (GCS) possui as seguintes atividades [PSDS]: Planejar GCS e Mudanças, Criar Ambientes de Configuração, Fazer e Liberar Mudanças em Itens de Configuração, Administrar *Baselines*, Monitorar e Relatar a Situação da Configuração, Administrar Solicitações de Serviços e Administrar Solicitações de Mudança.

A forma como o PSDS deveria ser utilizado em cada projeto era definido de acordo com o seu escopo: sumário (projetos com esforço de até 5 HD – Homem Dia), resumido (até 30 HD), simplificado (até 150 HD) e completo (acima de 150 HD). Não havia uma preocupação em diferenciar projetos de manutenção de projetos de desenvolvimento, sendo que quase a totalidade dos projetos de escopo sumário e resumido, dentro do SERPRO, se encaixa em projetos de manutenção de software.

Inicialmente, para este trabalho, foi realizado um estudo sobre a macroatividade Gestão de Configuração de Software (GCS) do PSDS, verificando a sua compatibilidade com o modelo CMMI, além da aderência de conceitos específicos

(gestão de configuração, *baseline*, etc) com outros membros da literatura, dentre eles [BABICH, 1986], [PFLEEGER, 2004] e [PRESSMAN, 2002].

O objetivo deste trabalho foi verificar a real aplicabilidade da macroatividade GCS em todos os projetos de uma das Superintendências do SERPRO, a qual possui vários sistemas sobre sua responsabilidade, sendo que esses sistemas possuem plataformas tecnológicas diversificadas, o que viabilizava que o estudo aqui realizado fosse válido para toda a empresa. Este trabalho foi realizado pela coordenadora do grupo de GCS dessa Superintendência e também membro do grupo especialista de GCS do SERPRO, e contou com a colaboração indireta dos membros do grupo de GCS, que reportavam as dificuldades encontradas na utilização da macroatividade em projetos de manutenção, bem como dos membros do grupo de GQS (Garantia da Qualidade de Software), que mostravam as ocorrências e/ou desvios detectados nos projetos.

4.1. Versão Original da Macroatividade GCS no PSDS

Na atividade Planejar GCS e Mudanças era feito o planejamento de como as *baselines* seriam geradas, auditadas e promovidas, independentemente do escopo do projeto. Era obrigatória a geração de *baseline* ao final de cada fase para projetos que utilizavam ciclo de vida cascata e ao final de cada iteração/fase para projetos que utilizavam o ciclo de vida iterativo. No PSDS as fases são: Concepção, Elaboração, Construção e Transição, tendo essas os conceitos similares às fases do RUP – *Rational Unified Process.*

Ainda nesta atividade**,** era definido que as auditorias deveriam ser geradas até 5 dias úteis após a geração da *baseline*, independente do estado em que a mesma se encontrava, devendo ser auditadas todas as *baselines* geradas, com exceção apenas das *baselines* cujo estado era rejeitada. Segundo o PSDS uma *baseline* pode ter os seguintes estados, devendo ser promovidas nesta seqüência: compilada -> testada -> homologada -> implantada. Uma *baseline* só poderá ser promovida para compilada após a conclusão de todos os artefatos previstos para a mesma, e para testada após esses terem passado pela equipe de testes e estarem totalmente corretos, ou seja, não poderão ocorrer ajustes nos mesmos, para homologada após homologação e aceite do cliente, e implantada após aprovação por todos os membros do Comitê de Mudança em Configuração de Software (CMCS), responsável principalmente por garantir que a transição dos sistemas (novos ou em manutenção) para a produção se faça adequadamente e sem provocar instabilidade no ambiente produtivo como um todo e nos serviços em produção. Outro estado existente é o de rejeitada, que poderia ser aplicado a qualquer momento.

Era obrigatória também nesta atividade a geração de um Plano de Gestão de Configuração de Software (PGCS) para cada projeto de software que surgisse. O conteúdo desse plano tinha a definição dos responsáveis pelas atividades de GCS, os itens de configuração, o planejamento da geração das baselines, os padrões de nomenclatura a serem adotados, as ferramentas de GCS a serem utilizadas e os treinamentos necessários.

Na atividade Monitorar e Relatar Situação de Configuração era obrigatória a criação de um artefato chamado Relatório de Configuração do Projeto (RCP), gerado para cada projeto, o qual continha o relato da situação da configuração de software do projeto, tais como: quantidade de *baselines*, quantidade de solicitações de mudanças, a situação dos itens de melhoria levantados em auditoria, variação da cardinalidade tanto das *baselines* quanto das solicitações de mudanças, etc.

As demais atividades não foram descritas neste artigo, visto que não precisaram ter uma melhoria implementada, sendo, portanto, descritos somente os pontos conflitantes para a utilização dessas atividades para os projetos de manutenção.

4.1.1. Problemas Identificados para Projetos de Manutenção

Durante a verificação da aplicabilidade da macroatividade GCS, foi observado que não havia problema algum em aplicá-la nos projetos de escopo simplificado e/ou completo da forma que estava definido no PSDS, muito pelo contrário, só aumentava a qualidade e a integridade dos artefatos produzidos. Entretanto**,** para projetos de escopo sumário e/ou resumido a realidade era outra**:** o processo se tornava altamente burocrático, sem muita utilidade prática, exigindo a geração de muita documentação desnecessária, a qual acarretava em um custo bem maior para o projeto, conforme relatado abaixo:

Gerar um PGCS para cada projeto só tendia a onerar o custo do projeto, devido ao maior tempo alocado a um gestor de configuração. Esses projetos possuem, quase sempre, a mesma equipe de desenvolvimento, itens de configuração similares, utilizam as mesmas ferramentas e padrões de nomenclatura, e o tópico mais importante é que não é possível prever quando irá ocorrer um projeto desse escopo, já que esse tipo de demanda simplesmente aparece devido à solicitação do cliente.

Outro problema era a quantidade de *baselines* a serem geradas devido à grande quantidade de projetos com escopo pequeno. E, conforme descrito no PSDS, uma vez gerada a *baseline*, uma auditoria tinha que ser realizada, o que fazia com que as entregas destes projetos demorassem mais que o necessário.

Em relação à promoção de uma *baseline*, ou seja, o estado que indica o seu nível de estabilidade e qualidade em um determinado momento, o processo era bastante burocrático para esses dois tipos de projetos (sumário e resumido). Como eles eram projetos de resolução rápida, visto que não ultrapassavam 30 HD (Homem Dia), era complicado formalizar estas promoções, principalmente no que se refere aos projetos sumários, os quais não ultrapassavam 5 HD. Segundo o PSDS, para cada estado a ser promovida uma *baseline* deve existir um responsável para autorizar esta promoção, conforme descrito na Tabela 1.

Tabela 1. Responsáveis por autorizar a promoção de uma *baseline*

Estado da *baseline*	**Responsável por autorizar a promoção**
Compilada	Líder do Projeto
Testada	Líder do Projeto
Homologada	Líder do Projeto e Área de Negócio
Implantada	CMCS (Líder do Projeto, Área de Negócio e Área de Infra-estrutura)
Rejeitada	Líder do Projeto e Área de Negócio

A autorização para se promover uma baseline não é algo simples de se obter, devido a grande quantidade de pessoas responsáveis, principalmente no estado implantada, onde todos os membros do CMCS (Comitê de Mudança em Configuração de Software) devem fornecê-la. Este procedimento acaba sendo demorado, visto que envolve membros que não estão ligados diretamente ao projeto em questão, como a área de infra-estrutura (rede, servidores, etc.), da coordenação de negócios, dentre outros.

Um problema identificado em relação às auditorias de *baselines*, *é* o fato de uma *baseline* poder ser gerada, auditada e posteriormente rejeitada. Com isso será gerada uma nova *baseline*, a qual deverá ser novamente auditada e corre-se o risco dela ser rejeitada outra vez, ou seja, há um tempo gasto em auditorias de *baseline* com possibilidade de rejeição.

Para exemplificar os problemas citados acima, um dos sistemas da Superintendência teve, no mês de março/2006, cinqüenta e três (53) projetos de escopo sumário/resumido concluídos. Neste caso, já seriam 53 PGCSs criados, 53 *baselines* geradas e 53 auditorias realizadas, além dos 53 RCPs que também deveriam ser gerados. É importante ressaltar que essa Superintendência possui vários outros sistemas, sendo que esses possuem também uma grande quantidade de projetos de escopo sumário/resumido, e no contexto do SERPRO, hoje com dezenove (19) Superintendências, o problema se tornaria ainda maior.

4.2. PSDS - Versão Personalizada

Com a finalidade de resolver os problemas citados, foram realizadas as seguintes modificações no PSDS [BARRETO, 2005], específicas apenas para projetos de escopo sumário ou resumido.

Na atividade Planejar GCS e Mudanças deixou de ser obrigatória, a geração de *baselines* ao final de fases/iterações, sendo obrigatório apenas a geração de uma *baseline* ao final do desenvolvimento (fase de Construção) e essa poderá ser promovida diretamente para o estado implantada, não necessitando das demais promoções.

A grande quantidade de *baselines* geradas inviabilizava que todas elas fossem auditadas, e como um determinado sistema possui uma grande quantidade de projetos com escopo sumário e/ou resumido, quase sempre atendidos pela mesma equipe, foi definido que estas auditorias passariam a ser feitas por amostragem, sendo auditadas 20% das *baselines* geradas em um determinado mês, devendo ser no mínimo 4.

A melhoria mais significativa foi a criação de dois modelos do Plano de Gestão de Configuração de Software (PGCS): um para projeto e outro para sistema. Portanto, é criado um PGCS de projeto para cada projeto de escopo simplificado ou completo que surgir, visto que esses geralmente possuem características próprias. Já o PGCS de sistema é criado uma única vez para cada sistema dentro da organização, e todos os projetos de escopo sumário ou resumido que surgirem irão utilizar esse PGCS como referência. A diferença entre eles é apenas no item que trata do planejamento das *baselines*. Nesse item do PGCS de projeto é possível fazer um planejamento, com definição de datas de quando uma *baseline* será gerada e o momento exato, pois o projeto a que este plano se refere tem longa duração. Já no PGCS de sistema, como este tipo de solicitação pode ocorrer a qualquer momento, tendo meses em que é muito freqüente e em outros não, o que se faz é planejar como deverão ser tratadas as *baselines* que forem geradas devido aos projetos sumários e/ou resumidos.

Por este motivo na nova versão do PSDS, lançada no final de março/2006, foi acrescentada a definição de que *baselines* criadas devido a projetos sumários e/ou resumidos deverão ser geradas diariamente, contemplando todos os projetos com este tipo de escopo que foi implantado no dia anterior, ou seja, não será mais gerada uma *baseline* a cada modificação, e sim, realizada uma junção das modificações que ocorreram em um determinado sistema em um certo dia sendo gerada no dia seguinte uma única *baseline* referente a estas alterações.

Na atividade Monitorar e Relatar Situação de Configuração o Relatório de Configuração do Projeto – RCP foi substituído pelo Relatório de Configuração do Sistema – RCS, que possui a mesma finalidade do antigo RCP, já que contém as informações de todos os projetos ligados a um determinado sistema, independente do seu escopo, o que permite que seja criado apenas um por mês, para cada sistema, ao invés de vários para cada projeto, que era o que ocorria. Isto facilitou bastante a visão do Gerente Sênior, pois este passou a obter estas informações centralizadas em um único documento, além também da melhoria que se obteve com a redução da quantidade de relatórios a serem gerados.

Para exemplificar de uma forma geral o ganho que se teve com esta personalização, basta verificar o exemplo anterior apresentado sobre um sistema, onde foram concluídos cinqüenta e três (53) projetos somente em um mês. Da forma que estava definido na versão anterior do PSDS, era obrigatória a geração de 53 PGCSs, um para cada projeto, e com essa atualização passa-se a gerar apenas um PGCS, o PGCS do sistema. Ganho similar será obtido em relação aos demais artefatos citados (RCS e RAC) e quanto à quantidade de *baselines* a serem geradas.

5. Conclusão

Com a personalização da macroatividade GCS do PSDS para projetos de manutenção (escopo sumário ou resumido) foram obtidas as seguintes melhorias:

- Redução da quantidade de artefatos a serem produzidos, devido à criação dos artefatos PGCS de sistema e RCS;
- Redução, bastante significativa, da quantidade de *baselines* a serem geradas, devido ao conceito de geração diária de *baseline*;
- Redução da apropriação de membros da equipe em atividades específicas de GCS e conseqüentemente dos custos dos projetos;
- Aumento da satisfação do cliente, visto que o projeto será executado em um tempo menor e mantendo a qualidade no controle de configuração, conforme definido no modelo CMMI.

Com isso, esta proposta, realizada inicialmente em uma das Superintendências do SERPRO, hoje encontra-se publicada no site do PSDS para toda a organização, após ter sido validada pelos membros do grupo especialista de GCS, pelo supervisor dessa macroatividade dentro da empresa e por empresa externa consultora.

Além disso, todas as melhorias descritas não fogem das práticas e metas preconizadas pelo nível 2 de maturidade do modelo CMMI, pois estas continuam a ser cumpridas. A única diferença é a mudança na forma como elas serão realizadas e alcançadas, estando voltadas mais para a realidade da organização, considerando a diferença entre atender uma demanda de desenvolvimento e atender uma demanda de manutenção.

Conforme [CURY, 2005], os processos são a estrutura pela qual uma organização faz o necessário para agregar valor aos produtos de seus clientes. Logo, a qualidade dos produtos possui uma íntima relação com a qualidade dos processos utilizados para desenvolvê-los e mantê-los. Portanto, a utilização de um processo de software eficiente e eficaz, que leve em consideração as peculiaridades das atividades de manutenção, é essencial para garantir uma boa manutenção de software.

6. Referências Bibliográficas

[BABICH, 1986] BABICH, W. A. ***Software Configuration Management***, *Addison-Wesley*, 1986.

[BARRETO, 2005] BARRETO, P. L., PESSOA, M., VILLAS-BOAS, A., ***Customização do Processo Gestão de Configuração de Software para Projetos de Escopo Pequeno.*** Universidade Federal de Lavras, Brasília - DF, Brasil, Novembro de 2005.

[BERSOFF et all, 1980] BERSOFF, E. H., et all, ***Software Configuration Management-An Investment in Product Integrity***, Pretice-Hall, 1980.

[BERSOFF, 1984] BERSOFF, E. H. ***Elements of Software Configuration Management***, IEEE Trans SE, V10, N1 - Janeiro de 1984.

[CANNIG, 1972] CANNIG, R. ***The Maintenance 'Iceberg'***, *EDP Analyzer, vol.10, no.10, October* 1972.

[CAPRETZ, 1992] CAPRETZ, M. A. M., ***A Software Maintenance Method Based on the Software Configuration Management Discipline***, Tese de Doutorado, University of Durham, Inglaterra, Outubro de 1992.

[CURY, 2005] CURY, A., ***Organização & Métodos, uma visão holística.*** Ed. Atlas, 2005.

[DIAS, 2004] DIAS, M. G. B. ***Uma Experiência no Ensino de Manutenção de Software.*** 1º Workshop de Manutenção de Software Moderna (WMSWM). Brasília: Outubro, 2004.

[ISO 10007] ISO 10007 - ***Quality management - Guidelines for Configuration Management***, Abril de 1995.

[ISO 12207] ISO 12207 – ***Information Technology – Software Life Cycle Processes***, Agosto de 1995.

[PFLEEGER, 2004] PFLEEGER, S. L. ***Engenharia de Software: Teoria e Prática.*** 2nd edition. *São Paulo: Prentice Hall*, 2004.

[PIGOSKI, 1996] PIGOSKI, T. M. ***Practical Software Maintenance.*** *John Wiley & Sons, Inc*., 1996.

[PRESSMAN, 2002] PRESSMAN, R. S. ***Engenharia de Software.*** 5ed. Rio de Janeiro: *McGraw-Hill*, 2002.

[PSDS] ***Processo SERPRO de Desenvolvimento de Soluções***. Site do PSDS - Brasil. Disponível em: <http://psds.portalcorporativo.serpro>. Acesso em 17 de março de 2006.

[SERPRO01] ***A Instituição: Quem somos***. Site do SERPRO - Brasil. Disponível em: <http://www.serpro.gov.br>. Acesso em 05 de março de 2006.

Evolução da complexidade de três sistemas legados

Nicolas Anquetil[1], **Claudson Melo**[1], **Pedro Nogueira de Sá**[1], **Marcelo A. L. da Rocha**[1], **Nelson F. de Almeida Júnior**[1]

[1]Universidade Católica de Brasília
SGAN 916, Módulo B, Brasília, DF, 70790-160
anquetil@ucb.br

Resumo. *A complexidade de software é um assunto que tem grande relevância para a manutenção. O consenso é que quanto mais complexo um software, mais complicado fica sua manutenção, isso até um ponto onde considera-se que não vale mais a pena manter um sistema e se torna preferível redesenvolver ele. Apesar de já ter sido bastante estudada, a complexidade de software é um assunto que traz consigo ainda bastante idéias confusas ou falsas. Nesse artigo, são apresentados os resultados de medição de algumas métricas de complexidade de software sobre três sistemas reais a diferentes épocas das suas vidas. Esses resultados são analisados no contexto de uma afirmação já antiga que pretende que a complexidade de um software aumenta a medida que ele vem sendo mantido (2a lei de evolução de software de Lehman, formulada em 1980).*

1 Introdução

Devido aos altos custos que a manutenção de software envolve (90% do custo total de um sistema [Pigoski 1996]) a longo prazo (décadas), existe um forte incentivo para a definição de métricas que permitem estimar a dificuldade de se realizar manutenção em sistemas. Uma das métricas mais procuradas nesse sentido visaria medir a complexidade do sistema que, por sua vez, tem impacto sobre a facilidade de se fazer manutenção neste sistema.

Para Lehman, "a medida que um [sistema] evolui, sua complexidade aumenta, a menos que algum trabalho seja feito para mantê-la ou reduzi-la" (segunda lei de evolução de software [Lehman 1980]). Mas nem todos concordam com essa "lei"[1]. Num esforço para tentar obter dados concretos a respeito da evolução da complexidade de um software, foram realizados estudos de caso sobre três sistemas reais para analisar a evolução de algumas métricas de complexidade clássicas da literatura. Algumas constatações são propostas a partir desses resultados.

Na seção 2 são revistos alguns conceitos básicos sobre complexidade de software e métricas. Na seção 3, são descritas as condições experimentais, com os resultados dos experimentos apresentados e comentados na seção 4. Finalmente, a seção 5 apresenta nossas conclusões.

2 Complexidade de Software

Complexidade de sistema é um assunto que já foi muito estudado. Se todos parecem concordar na avaliação do que, a dificuldade de realizar modificações em um sistema

[1]Harry Sneed, comunicação pessoal

é diretamente impactada pela complexidade deste; como medir essa complexidade não é um consenso. Segundo Sneed [Sneed 1995], Zuse [Zuse 1990] lista 139 métricas de complexidade diferentes. Fenton e Pfleeger advertem da dificuldade de se avaliar complexidade com uma única métrica já que ela engloba muitas noções diferentes, algumas delas conflitantes [Fenton and Pfleeger 1997, p.322]. Um exemplo simples disto pode ser proposto a partir de dois aspectos da complexidade de software: o tamanho e a legibilidade. O tamanho de um software impacta diretamente a complexidade, quanto maior um software, maior é sua complexidade, pelo simples fato que ele contem mais detalhes a serem entendidos, re-lembrados e combinados juntos para formar um entendimento geral do sistema. De outro lado, todo programador C sabe que esta linguagem permite condensar em poucas instruções altamente ricas várias linhas de código mais convencionais (ver um exemplo simples na Figura 1). Nesse caso, o pequeno tamanho do programa C resultante pode impactar negativamente a legibilidade (aumentando a complexidade).

```
s="Ordem e Progresso";
for(;*s;putchar(*s++));
```

```
s="Ordem e Progresso";
for (i=0; i < strlen(s); i++)
{
   putchar(s[i]);
}
```

Figura 1. Um exemplo em C de dois aspectos conflitantes da complexidade: o tamanho e a legibilidade

São detalhadas aqui algumas dessas múltiplas métricas, relacionadas à complexidade, dentro das mais conhecidas:

Linhas de código: Métrica de tamanho do software, quanto mais linhas de código, mais complexo é o sistema. Essa métrica pode apresentar problemas relacionados ao que se mede exatamente (linhas físicas em arquivo, linhas de código sem comentários ou sem linhas em branco, instruções).

Linhas de comentário: Métrica de compreensibilidade do software. Considera-se que quanto mais linhas de comentário, mais fácil de entender será o software e portanto menos complexo a modificar. Um problema da métrica é relacionado à qualidade dos comentários (realmente explicam o sistema ou são irrelevantes). Um caso particular é o uso do comentário como ferramenta de controle de versão, ou seja, código que foi comentado para não ser executado. Esse tipo de comentário será chamado de "lixo de código" neste artigo. Na medida do possível o lixo de código deve ser descontado das linhas de comentário.

Linhas de "lixo de código": (ver ítem precedente) Métrica de compreensibilidade do software. Considera-se que quanto mais lixo de código, mais difícil a entender será o software e portanto mais complexo a modificar.

Número de rotinas: Métrica de tamanho do software, quanto mais rotinas, mais complexo é o sistema.

Ciclomática: (também chamada de métrica de McCabe [McCabe 1976]) Métrica de complexidade de fluxo de controle, quanto maior a complexidade ciclomática, mais complexo o sistema é. A complexidade ciclomática mede o número de caminhos diferentes possíveis na execução do código do programa. É primeiramente uma medida do número de testes mínimo a realizar para testar todo o código (todos os

caminhos de execução possíveis). É a única métrica para qual existem valores de interpretação especificados (ver Tabela 1).

Complexidade estrutural de Card e Glass: [Card and Aggresti 1988] Métrica de complexidade de estrutura, quanto maior a complexidade estrutural, mais complexo o sistema é. A complexidade estrutural S_r de uma rotina é o quadrado de seu fan-out (número de rotinas que ela chama) $S_r = f\_out(r)^2$.

Complexidade de dados de Card e Glass: [Card and Aggresti 1988] Métrica de complexidade de dados, quanto maior a complexidade de dados, mais complexo o sistema é. A complexidade de dados D_r de uma rotina é a soma de todas os parâmetros de entradas, dividido pelo fan-out $D_r = \sum p\_in(r)/f\_out(r)$.

Complexidade de sistema de Card e Glass: [Card and Aggresti 1988] Métrica de complexidade, quanto maior a complexidade de sistema de Card e Glass, mais complexo o sistema é. É a soma das duas precedentes $C_r = S_r + D_r$.

Valores	Interpretação
de 1 a 10	Rotina simples, sem risco
de 11 a 20	Rotina mais complexa, risco moderado
de 21 a 50	Rotina complexa, alto risco
mais que 50	Rotina impossível de testar, risco muito alto

Tabela 1. Valores de referência de interpretação da complexidade ciclomática

3 Três Casos Concretos

As métricas descritas na seção precedente foram aplicadas a várias versões de três sistemas reais. Antes de apresentar os sistemas usados nesses experimentos, será discutido um pouco mais em detalhe como algumas métricas foram coletadas.

3.1 Coleta das métricas

Devido ao grande tamanho dos sistemas e a necessidade de repetir as medições para várias versões destes, as métricas propostas na seção 2 foram coletadas automaticamente. Se faz necessário dar algumas precisões sobre como elas foram coletadas:

Linhas de código: Foram contadas as linhas de código ignorando as linhas vazias (brancas) ou com apenas comentários. As linhas com código e comentário final são contadas. Várias ferramentas de programação (IDE) oferecem a possibilidade de gera código automaticamente, principalmente para o código de interface gráfica. Nos estudos de casos realizados, esse código pode representar até metade de todas as linhas de código do sistema. Esse código não foi contado nessa métrica. Essa métrica foi aplicada ao sistema inteiro.

Linhas de comentário: Foram contadas todas as linhas de comentário, inclusive linhas com código e comentário final (portanto essas linhas são contadas em duas métricas, ver ítem precedente). Na medida do possível não foram contadas as linhas de "lixo de código" (ver próximo ítem). Essa métrica foi aplicada ao sistema inteiro.

Linhas de "lixo de código": As linhas de código comentadas como forma de excluir elas do sistema executado foram contadas separadamente das linhas de comentário. O algoritmo usado para identificar essas linhas foi de contar a proporção de *tokens* reservados da linguagem na linha. Os *tokens* reservados da linguagem são

os operadores (ex: +, -, (,), &, ...) e as palavras reservadas (ex: *if*, *then*, *while*, ...). Quando a proporção de *tokens* reservados for superior a 25% de todos os tokens da linha, esta foi considerada "lixo de código". Esse algoritmo foi experimentado sobre uma amostra de 5000 linhas de código para um dos sistemas e deu completa satisfação (todos os casos de lixo de código identificados manualemente foram identificados pelo algoritmo). Essa métrica foi aplicada ao sistema inteiro.

Número de rotinas: Sem dificuldade particular para linguagem orientada a objeto (VB, Java) foram contados os métodos. A métrica foi aplicada ao sistema inteiro.

Ciclomática: A métrica de complexidade de McCabe pode ser calculada como o número de instruções de desvio num programa mais 1. Instruções de desvio são testes (ex.: *if*) e laços (ex.: *for*, *while*). Essa métrica foi aplicada a cada rotina (média de todas as rotinas).

Complexidade estrutural de Card e Glass: Sem dificuldade particular, a métrica foi aplicada a cada rotina (média de todas as rotinas).

Complexidade de dados de Card e Glass: Sem dificuldade particular, a métrica foi aplicada a cada rotina (média de todas as rotinas).

Complexidade de sistema de Card e Glass: Sem dificuldade particular, a métrica foi aplicada a cada rotina (média de todas as rotinas).

3.2 Os sistemas estudados

No momento do seu estudo, o sistema 1 tinha 6 anos de funcionamento, e, como é de se esperar, tinha sofrido inúmeras manutenções. Ele é responsável pelo controle de todos os documentos e processos que tramitam na organização. Ele tem grande importância, pelo número de usuários (aproximadamente 800) e pelo volume de acessos diários realizados por seus usuários. Esse sistema é escrito em Visual Basic (aprox. 50.000 linhas de código) e utiliza um banco de dados SQL Server (aprox. 40 tabelas e 13 milhões de registros).

Foram analisadas 3 versões: a versão (a) de implantação do sistema (março 1998), a versão (b) intermediária entre a data de implantação e a data de realização do estudo (julho 2000) e a versão (c) em operação no momento do estudo (setembro 2003).

O sistema 2 é um sistema de folha de pagamento. Ele tem mais de 20 anos de idade mas sofreu uma grande modificação em 1999, passando a utilizar o banco de dados e ferramentas de desenvolvimento da Oracle. Ele é composto de 110 programas com aproximadamente 60.000 linhas de código e um banco de dados Oracle (aprox. 150 tabelas e 20 milhões de registros).

Foram analisadas 3 versões, escolhidas a partir do mesmo critério que o sistema 1: a versão (a) de re-implementação do sistema (setembro 1999), a versão (b) intermediária entre a data de re-implementação e a data de realização do estudo (dezembro 2001) e a versão (c) em operação no momento do estudo (outubro 2003).

O sistema 3 é um sistema em uso em vários estados do pais que faz a gestão das carteiras de motorista. Ele está escrito em Java e tem mais de 12.000 linhas de código e 5.000 métodos. Foram analisadas 3 versões correspondentes ao inicio dos anos 2003, 2004 e 2005.

4 Resultados

Para os três sistemas os resultados são apresentados e comentados por grupo de métricas: (i) Métricas de tamanho e de compreensibilidade, (ii) métricas de complexidade.

4.1 Métricas de tamanho

Foram experimentadas duas métricas de tamanho ($\sum$ LOC: total de linhas de código e #rotina: número de rotinas) e duas de compreensibilidade ($\sum$ LDoc: total de linhas de comentário e $\sum$ LLixo: total de linhas de lixo de código). Para ajudar na análise, foram acrescentadas duas métricas compostas: proporção de linhas de código por linha de comentário e proporção de linhas de código por linha de lixo de código. Os resultados são apresentados na Tabela 2.

Tabela 2. Resultados das métricas de tamanho

versão	$\sum$ LOC	#rotina	$\sum$ LDoc	LOC / LDoc	$\sum$ LLixo	LOC / LLixo
Sistema 1						
a	10233	64	514	19,9	456	22,4
b	21134	134	1350	15,7	1127	18,8
c	28211	175	1873	15,1	1876	15,0
Sistema 2						
a	39804	541	3540	11,2	2114	18,8
b	44335	556	4126	10,7	3074	14,4
c	61074	742	6352	9,6	5755	10,6
Sistema 3						
a	89783	3906	9634	9,3	2813	31,9
b	109985	4937	14340	7,7	3130	35,1
c	120892	5387	16441	7,4	3171	38,1

Algumas observações podem ser feitas:

- Nos três sistemas o tamanho cresce com as versões sucessivas. Isso, por se mesmo, é um fator de crescimento da complexidade;
- Nos três sistemas o número de rotinas (ou métodos para os sistema 1 e 3) cresce com as versões sucessivas. Além de ser um fator de crescimento da complexidade, isso também corresponde a uma outra lei de evolução de software de Lehman [Lehman 1980] (lei de crescimento contínuo) que diz que "a capacidade funcional de um [sistema] deve ser continuamente incrementada para manter a satisfação do usuário ao longo do tempo";
- Nos três sistemas, a documentação (comentário) cresce, o que é positivo. Ainda melhor, a proporção de linhas de código por linha de comentário diminui o que indica que a documentação aumenta mais rápidamente que o código. Esse fato foi uma surpresa para os autores. Isso aponta para uma melhoria da compreensibilidade do sistema.
- Nos três sistemas, o lixo de código cresce. Esse resultado era esperado, a cada nova manutenção cresce a possibilidade de criar mais lixo de código enquanto é menos frequente a exclusão deste. Para os sistemas 1 e 2, a proporção de linhas de código por linha de lixo de código diminui o que indica uma tendência negativa, a compreensibilidade do sistema diminui. Para o sistema 3, a tendência é inversa, o que é positivo. Além disso, deve ser notado que, neste ultimo sistema, a proporção de linhas de código por linha de lixo de código é muito maior que para os outros sistema ou para a documentação (ou szeja, tem menos lixo de código).

Os resultados dessas medições são coerentes para os três sistemas. Eles indicam um crescimento da complexidade dos sistemas ao longo do tempo devido ao aumento do tamanho destes. De outro lado, a métrica de compreensibilidade cresce também o que corresponde a uma diminuição da complexidade devida a uma maior facilidade de compreensão.

A lei de aumento da complexidade de Lehman se encontra parcialmente confirmada e parcialmente refutada. Vários aspectos conflitantes da complexidade de software já podem ser detectados nesse experimento.

4.2 Métricas de complexidade

Foram experimentadas quatro métricas de complexidade (CC: complexidade ciclomática, S_r: Complexidade estrutural de Card e Glass, D_r: Complexidade de dados de Card e Glass, C_r: Complexidade de sistema de Card e Glass). Para facilitar a análise dos resultados, para cada sistema foram realizadas medições para o sistema inteiro (nas três versões) e para o conjunto das rotinas comuns às três versões (note que o código dessas rotinas pode mudar de uma versão para outra). Os resultados são apresentados na Tabela 2.

Tabela 3. Resultados das métricas de complexidade

	Sistema inteiro				Rotinas comuns			
versão	CC	S_r	D_r	C_r	CC	S_r	D_r	C_r
Sistema 1								
a	89,8	89,8	1,18	91,0	97,7	97,7	1,28	99,0
b	182,4	182,4	1,93	184,3	168,6	168,6	0,83	169,4
c	179,5	179,5	1,01	180,5	235,9	235,9	0,83	236,7
Sistema 2								
a	11,97	414,6	1,41	345,4	12,17	460,8	1,51	462,3
b	12,41	393,5	1,42	327,2	15,79	460,9	1,50	462,4
c	13,08	306,2	1,37	256,8	19,43	461,0	1,52	462,5
Sistema 3								
a	3,66	787,6	0,713	449,4	3,80	823,2	0,725	823,9
b	3,64	849,0	0,717	849,7	3,85	889,0	0,732	889,8
c	3,62	875,9	0,708	876,6	3,91	957,3	0,734	958,0

Algumas observações podem ser feitas:

- Não é possível tirar uma conclusão comum aos três sistemas em relação à evolução da complexidade ciclomática (CC) para o sistema todo. Para um sistema, aumenta antes de diminuir, para outro, diminui a cada nova versão e para o terceiro, aumenta a cada nova versão.
- Em contrapartida, a mesma métrica medida apenas para as rotinas comuns às três versões aumenta a cada nova versão para os três sistemas. Isso parece indicar que a complexidade das rotinas presentes nas três versões vai aumentando. Comparando com a mesma métrica para o sistema inteiro, pode-se pensar, ou que as novas rotinas que aparecem durante a manutenção tem complexidade ciclomática menor (e portanto baixam a média do sistema inteiro), ou que as rotinas com maior complexidade ciclomática foram removidas (não seriam simplesmente simplificadas

porque não aparecem como rotinas comuns às três versões). Os dados não concordam com a segunda interpretação (por exemplo, para o sistema 3, dos 14 métodos mais complexos, apenas dois são excluidos nas outras versões). Por tanto, podemos deduzir que nesses três sistemas, as rotinas aumentam de complexidade ciclomática com as manutenções, as novas rotinas aparecem com complexidade ciclomática menor.

- As mesmas conclusões que para a complexidade ciclomática podem ser tiradas da complexidade estrutural de Card e Glass (S_r) em relação à falta de dados conclusivos para o sistema inteiro. É interessante notar que a evolução da complexidade estrutural de Card e Glass nem sempre acompanha à evolução da complexidade ciclomática. Isso parece indicar que são duas métricas independentes que medem realmente diferentes aspectos da complexidade.
- As mesmas conclusões que para a complexidade ciclomática podem ser tiradas em relação à evolução da complexidade estrutural de Card e Glass (S_r) das rotinas comuns às três versões. Essas conclusões são reforçadas pela aparente independência entre as duas métricas (ver ítem precedente).
- De novo, para o sistema inteiro, não podemos tirar nenhuma conclusão das medições da complexidade de dados de Card e Glass (D_r).
- Para as rotinas presentes nas três versões, os resultados da complexidade de dados de Card e Glass são muito menos claros. Isso apenas indica que a complexidade de dados evolui de maneira independente (e as vezes oposta) à complexidade de estrutura (ou de fluxo)
- Finalmente, a complexidade de sistema de Card e Glass (C_r) acompanha as evoluções da complexidade estrutural de Card e Glass devido à disproporção de valores entre seus dois componentes (complexidade estrutural + complexidade de dados).

Em resumo, os resultados mostram, de novo, evoluções diferentes, e as vezes opostas, de vários aspectos da complexidade de software. Os aspectos que parecem ser indicados pelos resultados seriam a complexidade de dados, a complexidade de estrutura e a complexidade de fluxo.

Alguns dos resultados estão de acordo com a lei de aumento da complexidade de Lehman quando consideradas apenas as rotinas comuns às várias versões de cada sistema. Quando considerado o sistema inteiro (com adição de novas rotinas e, em proporção menor, remoção de outras) o aumento da complexidade não pude ser observado (houve até casos de diminuição de complexidade com o tempo). Uma explicação provável seria que as novas rotinas do sistema são criadas com complexidade menor e vão se complicando com a realização de manutenções. A lei de aumento da complexidade de Lehman parece válida apenas para o código existente de um sistema que vem sendo modificado pelas manutenções.

5 Conclusão

Porque ela parece ter um grande impacto sobre o custo de manter sistemas de software, a complexidade de software é uma característica que foi amplamente estudada. Foi destacado neste artigo a segunda lei de evolução de software de Lehman [Lehman 1980] que postula que “a medida que um [sistema] evolui, sua complexidade aumenta, a menos que algum trabalho seja feito para mantê-la ou reduzi-la”. Num esforço para tentar verificar

esta lei, foram realizados experimentos com três sistemas diferentes, medindo diferentes aspectos da sua complexidade ao longo da vida deles (três versões para cada sistemas).

Os resultados sugerem algumas observações:

- Vários aspectos de complexidade com comportamento distinto ou até oposto puderam ser observados (ex.: tamanho, compreensibilidade, complexidade de estrutura, complexidade de fluxo, complexidade de dados).
- Foi observado um aumento do tamanho dos três sistemas, em termos de linhas de código e número de rotinas (ou métodos). Isso contribui a um aumento da complexidade desses sistemas.
- O aumento do número de rotinas parece apoiar uma outra lei de Lehman (crescimento da capacidade funcional).
- Foi observado um aumento da documentação (comentários) dos sistemas, tanto em termos absolutos (número de linhas de comentários) quanto em termos relativos (proporção de linhas de comentário por linhas de código). Isso contribui para um aumento da compreensibilidade dos sistemas ou uma diminuição da complexidade de compreensão.
- Foi observado um aumento da complexidade estrutural de Card e Glass e da complexidade ciclomática para as rotinas presentes nas três versões de cada sistema. As mesmas métricas não apresentam evolução clara quando considerado o sistema inteiro. Isso aponta para um aumento dessas complexidades para o código existente (2a lei de evolução de software), ao mesmo tempo que rotinas novas apareceriam com uma complexidade menor, contribuindo a diminuir a complexidade média do sistema inteiro.

Referências

Card, D. and Aggresti, W. (1988). Measuring software design complexity. *Journal of Systems and Software*, 8(3):185–97.

Fenton, N. E. and Pfleeger, S. L. (1997). *Software metrics, a rigorous and practical approach*. PWS Publishing Company, 2nd edition.

Lehman, M. M. (1980). Programs, life cycles and the laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–76.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–20.

Pigoski, T. M. (1996). *Practical Software Maintenance: Best Practices for Software Investment*. John Wiley & Sons, Inc.

Sneed, H. M. (1995). Estimating the costs of software maintenance tasks. In *International Conference on Software Maintenance, ICSM'95*, pages 168–81. IEEE, IEEE Comp. Soc. Press.

Zuse, H. (1990). *Software Complexity — Measure and Methods*. De Gruyter Verlag, Berling.

Estudo quantitativo da manutenção evolutiva em dois sistemas de código aberto

André Marques Poersch[1], Natália Sales Santos[1], Maria Augusta V. Nelson[1]
[1]Sistemas de Informação, Instituto de Informática
Pontifícia Universidade Católica de Minas Gerais
Av. Afonso Vaz de Melo, 1200 - 30640-070, Belo Horizonte, Minas Gerais
andrempo@gmail.com, santos_nate@yahoo.com.br, guta@pucminas.br

Abstract. *This paper presents a quantitative study of the evolution of requirements in two open source systems: Apache and Moodle. After mining the requirements and classifying them by category, domain and evolution, the results are analyzed. Some theories are made to attempt to justify the evolutionary patterns found in the systems.*

Resumo. *Apresenta-se um estudo quantitativo sobre a evolução de requisitos em dois sistemas de código aberto: Apache e Moodle. Após o levantamento dos requisitos e sua classificação por categoria, domínio e evolução, é feita uma análise dos resultados. Também são levantadas hipóteses para justificar os padrões de evolução de requisitos apresentados pelos sistemas.*

1. Introdução

1.1. Evolução de sistemas

Estudos feitos por Lientz e Swanson mostram que mais da metade do esforço total empregado no ciclo de vida de um sistema de software acontece na fase de manutenção [Lientz e Swanson, 1980]. Os autores classificam as atividades de manutenção em três categorias: a manutenção corretiva, que consiste em correção de defeitos; a manutenção adaptativa, que consiste na adaptação no sistema para que continue funcionando frente a mudanças em seu ambiente; e a manutenção evolutiva, que consiste na melhoria do sistema de algum modo. Segundo os autores, a manutenção evolutiva é responsável por mais de 60% das atividades de manutenção. O objetivo deste trabalho é analisar a manutenção evolutiva dos requisitos de dois sistemas de código aberto.

1.2 Sistemas de código aberto

O desenvolvimento de software de código aberto (*open source software*) vem crescendo mundialmente em número de projetos, tamanho dos projetos, áreas de aplicação, número de desenvolvedores, usuários e pesquisadores envolvidos. Software de código aberto "é um tipo de software cujo código-fonte é público" [OSI 2006].

Existem poucos estudos acadêmicos sobre o modelo de desenvolvimento empregado nas comunidades de desenvolvedores de código aberto. Os primeiros estudos acadêmicos [German 2005, Scacchi 2005] são recentes e mostram que, em termos de desenvolvimento de código aberto, a abordagem adotada diverge dos padrões mundiais da engenharia de software. O enfoque de um projeto de código aberto é na

implementação. Os programadores de um projeto, que na maioria das vezes estão localizados em regiões geográficas diferentes, discutem e tomam decisões em fóruns via Internet [German 2005]. Pouquíssima documentação a respeito do projeto é encontrada em documentos formais. Ao invés disso, toda a informação desses projetos de código aberto se encontra dispersa nos fóruns de discussões, nas requisições de mudanças, nos relatos de erros e no código. O código é a documentação do projeto [Robles *et al* 2005].

1.3 Evolução de sistemas de código aberto

Este trabalho explora a evolução de requisitos em software de código aberto. Dois sistemas de código aberto foram analisados, o servidor Apache e o sistema de gerenciamento de cursos Moodle. Comparado ao trabalho de German, este estudo se diferencia no objeto de estudo da evolução. German explora a evolução do código sem se deter aos requisitos dos sistemas de código aberto [German 2005]. Comparado ao trabalho de Scacchi, este estudo se diferencia em enfoque. Scacchi estudou o processo de levantamento e análise de requisitos em softwares de código aberto de forma geral, sem se deter à sua evolução [Scacchi 2005]. Espera-se que os resultados obtidos no estudo sejam representativos de sistemas de código aberto e que possam ser usados para auxiliar no planejamento de novos sistemas desse tipo.

Como os requisitos de um sistema de código aberto nunca são levantados e documentados antes do projeto iniciar seu desenvolvimento, o manual do usuário de uma determinada versão do sistema é tomado como a especificação de requisitos daquela versão [Berry *et al* 2004]. Outras fontes de informações como as *release notes* e requisições de mudanças também são usadas para análise da evolução dos requisitos.

2. Categorias de requisitos analisadas

Os tipos de requisitos analisados neste estudo pertencem a quatro grandes categorias apresentadas em [Pfleeger e Atlee 2005]: requisitos funcionais, restrições de projeto, restrições de processo e requisitos de qualidade ou não funcionais.

A categoria de requisitos funcionais está dividida em funcionalidade e dados. A categoria de restrições de projeto, como o próprio nome diz, restringe o sistema quanto ao seu ambiente físico, seus usuários e interfaces com outros sistemas. A categoria de restrições de processos abrange restrições de recursos, documentação e padrões de desenvolvimento de sistemas. A categoria de exigências de qualidade define o desempenho, usabilidade, segurança, confiabilidade e manutenibilidade do sistema.

3. Metodologia da pesquisa

O primeiro passo da pesquisa foi a seleção dos sistemas. Nesta etapa procurou-se por sistemas que: existiam há pelo menos três anos, para se caracterizar uma evolução de versões; possuíam documentação suficiente para possibilitar a pesquisa e o levantamento de requisitos; e foram desenvolvidos por vários colaboradores espalhados geograficamente. Foram escolhidos dois sistemas: o Apache, que é um servidor de páginas web [ASP 2006], e o Moodle, um sistema de gerenciamento de cursos na web [Moodle 2006].

O segundo passo foi o levantamento de requisitos de cada versão. A partir do manual do usuário da versão mais atual, foram levantados os requisitos da última

versão. Os requisitos das versões mais antigas foram retirados das *release notes* das respectivas versões, começando da mais antiga.

Em seguida, os requisitos foram classificados de acordo com sua categoria, versão, versão em que o requisito primeiro apareceu, fonte do requisito, domínio de aplicação dentro do sistema (por exemplo, no Moodle, requisitos de administração do sistema, de cursos, de exames, de exercícios) e evolução. Em se tratando da evolução, os requisitos foram classificados como novos, antigos/estáveis, ou melhorias (requisito antigo que foi modificado).

Após a classificação foram criados gráficos que mostram diversos aspectos da evolução dos requisitos e feita uma análise quantitativa dos mesmos

4. Estudo de caso

4.1. Apache

Para este estudo, o servidor web Apache foi selecionado dado a sua grande utilização (cerca de 70% do mercado) e por conter uma história de mais de 10 anos [ASP 2006]. Suas funcionalidades são mantidas através de uma estrutura de módulos. Neste estudo foram analisados os seguintes módulos: *access control, binding, configuration files, content negotiation, core, HTTP response, logging, virtual host.* Os outros módulos do Apache não fizeram parte deste estudo pois não sofreram modificações ao longo das versões analisadas.

Sua primeira versão oficial foi a 0.6.2 lançada em 1995, mas a sua base principal é a versão 1.3. Nas versões 2.x a escalabilidade do servidor foi ampliada tornando-a pouco compatível com as versões anteriores, uma vez que os seus módulos sofreram grandes alterações. Neste estudo foram analisadas as versões 1.3, 2.0 e 2.2. Para o levantamento de requisitos do Apache foram utilizados os manuais de usuários e a documentação dos módulos e das novas funcionalidades anunciadas em cada versão [ASP 2006].

Os gráficos a seguir apresentam a evolução dos requisitos. Para cada versão mostra-se a variação no número de requisitos em relação à versão anterior. A Figura 1 apresenta a evolução dos requisitos sob o ponto de vista do número de requisitos *acrescentados* em cada versão, classificados por categoria de requisitos de acordo com as categorias apresentadas na Seção 2. Pode-se observar que os requisitos de funcionalidade são os que aparecem em maior número nas fontes consultadas.

A Figura 2 apresenta a evolução dos requisitos sob o ponto de vista do número de requisitos *acrescentados* em cada versão, classificados por domínio da aplicação. No caso do Apache os módulos do sistema foram usados para classificar as diferentes áreas do domínio. Pode-se observar que os requisitos que tratam dos arquivos de configuração e do módulo principal aparecem em maior número. Observa-se também que muitos requisitos foram acrescentados na versão 2, refletindo a reestruturação dos módulos ocorrida nesta versão de acordo com o histórico fornecido.

A Figura 3 mostra a evolução dos requisitos sob o ponto de vista de melhorias ou requisitos completamente novos. Observa-se que a maioria dos requisitos acrescentados são requisitos novos ao sistema.

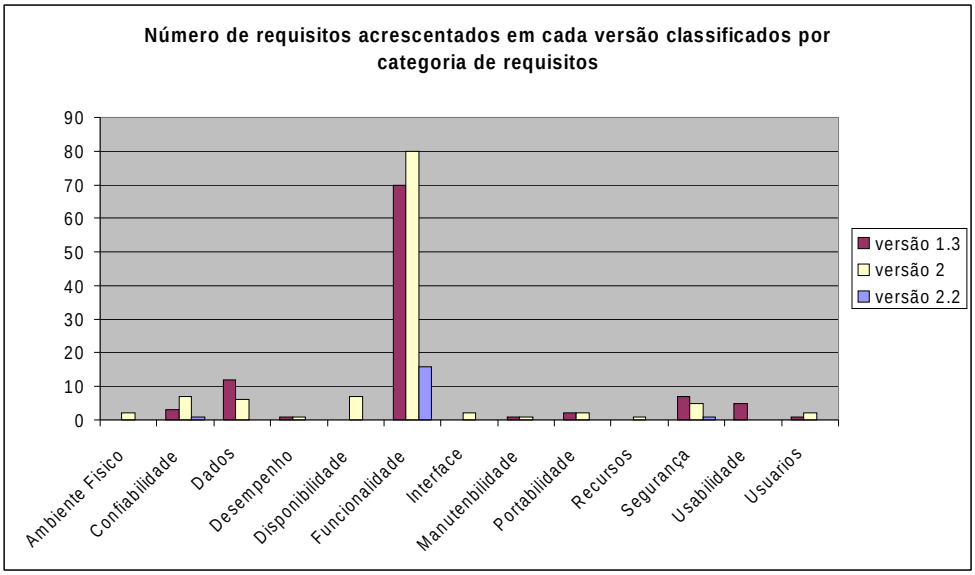


Figura 1. Apache — número de requisitos acrescentados em cada versão, classificados por categoria.

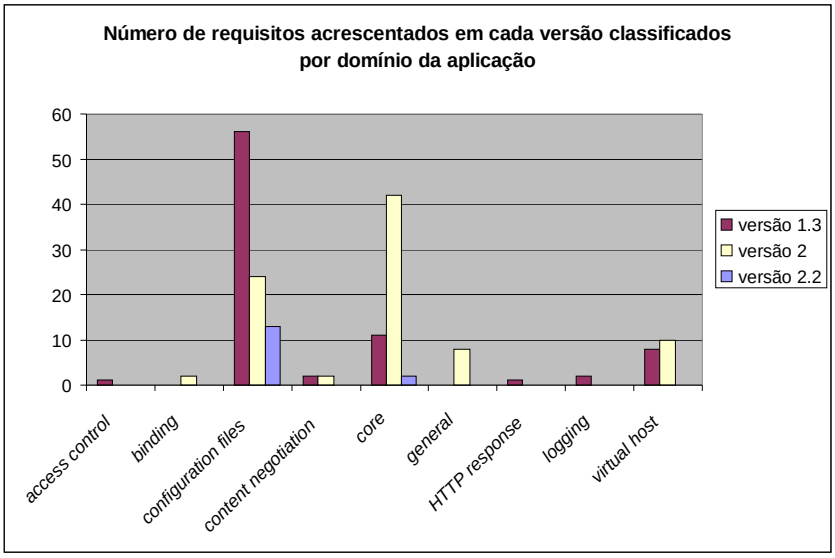


Figura 2. Apache — Número de requisitos acrescentados em cada versão classificados por domínio da aplicação.

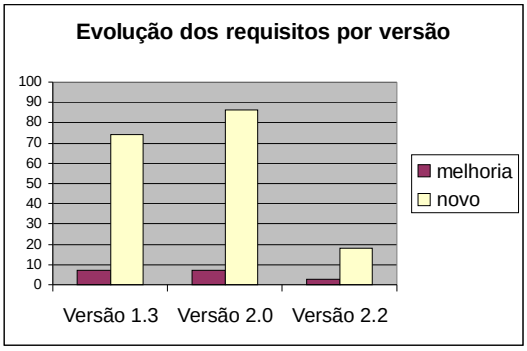


Figura 3. Apache — Evolução dos requisitos novos e melhorias por versão

4.2. Moodle

O Moodle é uma ferramenta para produzir cursos baseados na internet [Moodle 2006]. Foi criado por Martin Dougimas para ser um CMS (*Course Management System* - sistema de gerência de cursos) que suporta a "pedagogia social construcionista". Desde seu lançamento o Moodle vem sendo usado por instituições de ensino em todo o mundo.

O Moodle é organizado em forma de módulos de atividades: módulos de fórum, *quiz*, administração e outros, que podem ser adicionados ou removidos conforme a necessidade. As fontes usadas para o levantamento de requisitos foram as documentações do Moodle (manuais do usuário), as *release notes* e a lista de novas funcionalidades [Moodle 2006].

Os gráficos a seguir mostram a evolução dos requisitos no Moodle. Observe que neste estudo de caso os dados são cumulativos, ou seja, para cada versão mostra-se o número *total* de requisitos naquela versão. A Figura 4 apresenta dois gráficos que mostram a evolução do número de requisitos em cada versão nas categorias "funcionalidade" e "segurança". Pode-se observar que os requisitos de funcionalidade crescem de forma quase linear. Já os requisitos de segurança apresentam um pico de crescimento na versão mais recente do sistema.

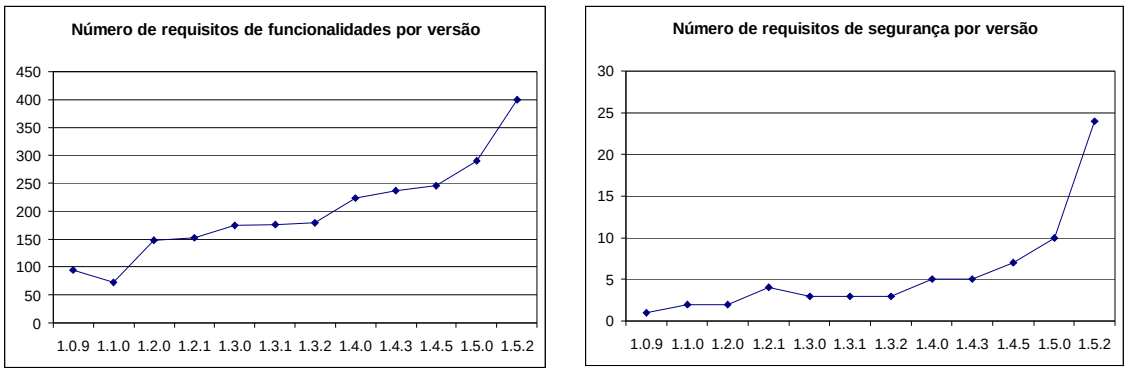


Figura 4. Moodle — Número de requisitos de funcionalidade e segurança em cada versão.

A Figura 5 mostra a evolução sob o ponto de vista do número de requisitos em cada versão em 5 dos 24 módulos do Moodle. Pode-se observar que o crescimento dos requisitos não segue um padrão regular. Alguns módulos, como o de linguagem, evoluem de versão em versão com um crescimento de requisitos quase linear. Outros, como o de administração, permanecem quase constantes mas crescem de modo acentuado em algumas versões.

A Figura 6 mostra a evolução sob o ponto de vista de requisitos antigos, melhorias ou requisitos completamente novos. Observa-se que existem poucas melhorias. A evolução acontece através da inserção de novos requisitos ao invés de melhorias em requisitos existentes. Os requisitos antigos representam os requisitos que não sofreram modificações de melhoria de uma versão para outra.

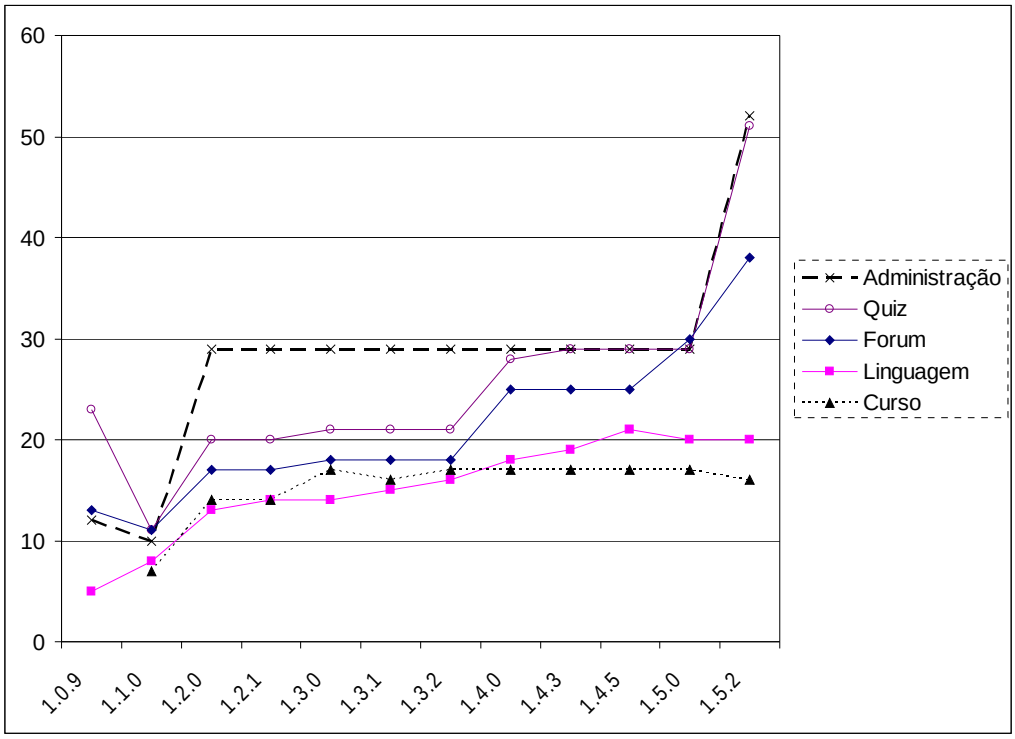


Figura 5. Moodle — Número de requisitos por versão relacionados a cinco módulos do Moodle

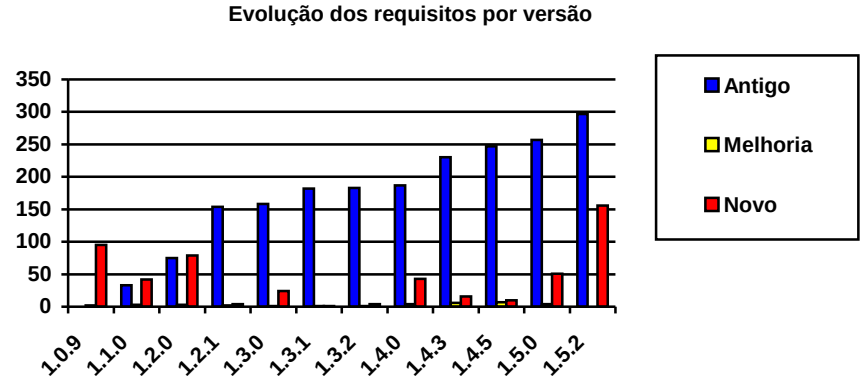


Figura 6. Moodle — Evolução dos requisitos antigos, novos e melhorias por versão

5. Análise dos resultados

A primeira análise a ser feita diz respeito às categorias de requisitos. Observou-se inicialmente que as categorias de restrições de processo e projeto são insignificantes frente ao total de requisitos dos sistemas analisados.

Os requisitos funcionais totalizam o maior número de requisitos na evolução dos sistemas. Isso pode ser explicado por duas hipóteses: sistemas de código aberto se preocupam mais com a funcionalidade do que com os requisitos de qualidade; ou ainda, não existem mecanismos apropriados para se expressar os requisitos de qualidade. Tomando a segurança (uma das subcategorias de requisitos de qualidade) para análise, observa-se que o tipo do sistema faz diferença na evolução do número de requisitos em

cada versão. O Apache apresenta mais requisitos de segurança nas versões iniciais e o Moodle nas versões finais. Isso mostra o quão crítico este tipo de requisito é para cada um dos sistemas.

Em ambos os sistemas a evolução consiste em grande parte de funcionalidades completamente novas e não melhorias dos requisitos das versões anteriores. Uma hipótese para esta observação é fato de que em sistemas de código aberto as primeiras versões podem ser consideradas versões preliminares quando são liberadas para o público. Desta forma, as primeiras versões ainda não contêm toda a funcionalidade prevista. Esta funcionalidade vai sendo acrescentada nas versões posteriores. Em contraste, no desenvolvimento do software proprietário as primeiras versões não são liberadas para o público. Apenas quando o software adquire uma certa estabilidade na funcionalidade é que passa a estar disponível no mercado.

Outra análise trata da evolução dos domínios de cada aplicação. No Apache percebe-se, por exemplo, que um grande número de requisitos acrescentados está relacionado aos arquivos de configuração. Observa-se também que o módulo principal (*core*) do Apache sofreu muitas modificações, melhorias ou foi acrescido de novas funcionalidades da versão 1.3 para a versão 2.0. Esta observação é outro indício da falta de compatibilidade entre a versão 2.0 e as anteriores. Por outro lado, outros domínios permaneceram praticamente inalterados. A distribuição da evolução entre os domínios não obedece a nenhum padrão.

No Moodle, percebe-se que há manutenção evolutiva na maioria das versões. O crescimento da funcionalidade é quase linear, porém há exceções, como a versão 1.3.2 em que praticamente não houve evolução, apenas correção de defeitos. Por outro lado, há também versões predominantemente evolutivas, como a versão 1.5.2.

6. Conclusões

Este trabalho apresentou um estudo quantitativo sobre a evolução de requisitos em dois sistemas de código aberto: o Apache e o Moodle. Algumas das observações feitas com base nos dados levantados são:

- A evolução consiste predominantemente de novas funcionalidades, mas há evolução também de requisitos de qualidade.
- No Moodle, onde há grande número de versões, a funcionalidade cresce de forma quase linear entre as funções, mas não tem distribuição uniforme nos módulos do sistema. Alguns crescem muito de uma versão para outra, enquanto outros permanecem inalterados.
- No Apache, alguns módulos evoluem muito mais do que outros.
- O crescimento de alguns tipos de requisito mostra a ênfase colocada naquele tipo de requisito em cada sistema. Por exemplo, segurança foi prioridade no Apache desde o princípio, enquanto que no Moodle requisitos desse tipo cresceram apenas nas últimas versões.
- Algumas versões são predominantemente evolutivas (por exemplo, a versão 1.5.2 do Moodle), enquanto outras são puramente corretivas (por exemplo, a versão 1.3.1 do Moodle).

- A maioria das atividades de evolução consiste de novas funcionalidades e não de melhorias em funcionalidades existentes.

Como trabalhos futuros, espera-se aprofundar as análises realizadas. Por exemplo, Anderson e Felici estudaram a evolução dos requisitos de um sistema de aviação, produzindo uma taxonomia do tipo de mudanças nos requisitos [Anderson e Felici, 2002]. A taxonomia inclui categorias como refraseamento, detalhamento e restrição. A evolução de requisitos do Apache e do Moodle deve ser estudada sob esta taxonomia para possibilitar novas análises. Outro trabalho futuro é uma comparação com a evolução de sistemas proprietários para avaliar se os padrões de evolução dos requisitos são semelhantes aos de código aberto ou se diferem, uma vez que os processos de desenvolvimento são, em geral, diferentes.

Agradecimentos

Os autores agradecem ao Fundo de Incentivo à Pesquisa (FIP) da Pontifícia Universidade Católica de Minas Gerais pelo apoio material e financeiro recebido.

Referências bibliográficas

Anderson, S. e Felici, M. (2002) Quantitative aspects of requirements evolution. In Proceedings of the 26th Annual International Computer Software and Application Conference, COMPSAC

ASP (2006) Projeto do servidor HTTP Apache. Disponível em http://httpd.apache.org

Berry, D. M., Daudjee, K., Dong, J., Fainchtein, I., Nelson, M.A.V., Nelson, T, e Ou, L. (2004) User's Manual as a Requirements Specification: case studies. *Requirements Engineering Journal* 9(1), pp. 67-81. Springer-Verlag.

German, D.M. (2005) Using Software Trails to Reconstruct the Evolution of Software. *Journal of Software Maintenance and Evolution: Research and Practice*. Aceito para publicação.

Lientz, B. P. e Swanson, E. B. (1980) Software Maintenance Management. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Moodle (2006), Projeto Moodle. Disponível em http://moodle.org

OSI (2006) Open Source Initiative. Disponível em http://www.opensource.org/

Pfleeger, S. e Atlee, J. (2005) Software Engineering: Theory and Practice, Pearson Prentice Hall, 3rd edition.

Robles, G. ; Amor, J. ; González-Barahona, J. e Herraiz, I. (2005) Evolution and Growth in Large Libre Software Projects. Proceedings of the 8th International Workshop on Principles of Software Evolution, Lisbon, Portugal.

Scacchi, W., Jensen, C., Noll, J., e Elliott, M. (2005) Multi-Modal Modeling, Analysis and Validation of Open Source Software Requirements Processes, *Proceedings of the First International Conference on Open Source Software*, Genova, Itália.