CENTRO PAULA SOUZA
GOVERNO DO ESTADO DE SÃO PAULO

Faculdade de Tecnologia de Americana
Curso Superior de Bacharelado em
Análise de Sistemas e Tecnologia da Informação

MANUTENIBILIDADE DE *SOFTWARE* E ESTUDO DE CASO APLICADO AO *SOFTWARE* OPENBIBLIO

JOSIANE ROSA DE OLIVEIRA GAIA

Americana, SP
2013

CENTRO PAULA SOUZA
GOVERNO DO ESTADO DE SÃO PAULO

Faculdade de Tecnologia de Americana
Curso Superior de Bacharelado em
Análise de Sistemas e Tecnologia da Informação

MANUTENIBILIDADE DE *SOFTWARE* E ESTUDO DE CASO APLICADO AO *SOFTWARE* OPENBIBLIO

JOSIANE ROSA DE OLIVEIRA GAIA

josiane.gaia@fatec.sp.gov.br

Trabalho Monográfico, desenvolvido em cumprimento à exigência curricular do Curso Superior de Bacharelado em Análise de Sistemas e Tecnologia da Informação da Fatec-Americana, sob orientação do Prof. Me. Anderson Luiz Barbosa.

Área: Engenharia de *Software*/ Manutenção de *Software*

Americana, SP
2013

BANCA EXAMINADORA

Prof. Me. Anderson Luiz Barbosa
(Orientador)

Prof. Esp. César Augusto Crócomo

Prof. Me. Kleber de Oliveira Andrade

AGRADECIMENTOS

Agradeço primeiramente a DEUS, que é a razão do meu viver, o idealizador e o realizador de todos os meus sonhos.

Agradeço a minha família, em especial a meus pais, José Antônio Gaia e Anivair Maria de Oliveira Gaia, que tudo fazem para me ajudar e apoiar nas minhas lutas. São o meu consolo e minha alegria em todos os momentos de minha vida. Ao meu namorado Lucas, que me apoia e incentiva a não desistir da caminhada. Ao meu cãozinho Max, fiel companheiro, amigo leal que mesmo não sabendo falar, sentava ao meu lado enquanto eu desenvolvia este projeto e me animava com seu jeitinho de ser. A minha afilhada Laura, luz que ilumina minha vida e me dá forças para prosseguir.

Agradeço aos meus amigos e colegas de classe, em especial a Tamara, Tiago, Natália, Marshall, Amaury, Guilherme, Diego e Rodrigo que permaneceram me dando forças e me incentivando a perseverar nestes quatro anos.

Agradeço aos meus amigos da ETEC de Hortolândia, que estiveram presentes e torcendo para esta conquista. Em especial ao Hemerson Laranjeira, Wagner Silva, Juliana Sá, Juliana Savitsky, Juliana Luvizotti, Priscila Martins, Célia Barufaldi, Karina Crispim, Patrícia Negro, Rogério Passos, Gustavo Piva, Daniel Funari, Rubiane Peixoto, Aloísio Luzo, Renan Silva, Giovane Guassu, Vera Cajarana e Marli Dionísio.

Agradeço a todos os meus mestres, a todos os professores e professoras que tive em toda a minha vida e que deixaram marcas especiais em meu coração. Estes que me ensinaram não somente o conhecimento para o mundo técnico, mas para toda a vida. Que acreditaram em mim e no meu potencial e me ensinaram com amor e dedicação.

Agradeço ao meu orientar Anderson Luiz Barbosa, sempre paciente em me ouvir, dedicado em me orientar e sábio ao mostrar o melhor caminho a seguir.

Agradeço a todos aqueles que amo e que me amam de todo o coração. A todos que fizeram orações por mim e me dedicaram palavras de amor e carinho que me deram forças para seguir em frente.

"Apostai sobre os grandes ideais, sobre as coisas grandes. Nós, cristãos, não fomos escolhidos pelo Senhor para coisinhas pequenas, ide sempre mais além, rumo às coisas grandes. Jovens, jogai a vida por grandes ideais!"

Papa Francisco

Dedico este trabalho a meus pais e a todos aqueles que eu amo. Dedico também a todos os que, de alguma forma, contribuíram para a concretização deste trabalho monográfico e a conclusão de minha faculdade. Dedico, sobretudo aqueles que acreditam num futuro melhor e que sabem que cada um de nós é responsável por fazer a diferença com suas atitudes.

RESUMO

O presente texto conceitua o tema manutenibilidade de *software*, que é a capacidade de um *software* receber manutenção. O desenvolvimento se deu baseado no objetivo de fazer um levantamento bibliográfico sobre o assunto e aplicar as informações obtidas em estudo de caso no *software* OpenBiblio. Como metodologia utilizou-se a pesquisa bibliográfica, documental e o estudo de caso. O desenvolvimento organizou-se trazendo uma fundamentação teórica sobre engenharia de *software* e os processos de desenvolvimento, seguido por uma descrição da manutenção de *software* e suas características importantes, para depois adentrar no tema manutenibilidade, explicando seu conceito, surgimento e explorando algumas métricas importantes. O estudo de caso é apresentado em seguida, elaborado com base em uma métrica escolhida e discutindo os resultados obtidos. As considerações finais fazem um apanhado geral do trabalho monográfico, destacando os objetivos atingidos e não atingidos, as dificuldades encontradas no estudo de caso, as conclusões sobre o tema e as indagações geradas no decorrer do desenvolvimento.

Palavras Chave: manutenção de *software*, manutenibilidade de *software,* OpenBiblio

ABSTRACT

The present text conceptualizes the maintainability theme, which is the software capability of receiving maintenance. This monograph development was based in a bibliography research about the subject, followed by a case study at OpenBiblio software. The methodology is bibliography research, document research and case study. The development was organized with a theoretical foundation about software engineer and development processes. After that, there is a software maintenance description focused on important key issues. Lastly, there is an explanation about maintainability, its concepts, history and some important metrics. The case study is presented in sequence, based in a preview metric choose. It discusses about the study and the results obtained with it. The final remarks make a review of important topics at the paper, reviewing the research objectives, the difficulty and the questions originated by this monograph.

Keywords: software maintenance, software maintainability, OpenBiblio.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O uso de *software* tem se tornado uma rotina dentro das organizações. As inúmeras facilidades trazidas com sua utilização criaram não somente a dependência, mas também a utilização em larga escala de aplicações, sejam elas ferramentas produzidas e distribuídas de forma genérica, tais como antivírus, pacotes de escritório e navegadores de internet, ou sejam aplicações personalizadas de acordo com as necessidades do cliente, como é o caso de muitos sistemas de informações, que integram todas as áreas de uma organização.

A produção de *software* passou por inúmeras evoluções ao longo dos tempos. Novos modelos de desenvolvimento, metodologias e técnicas foram difundidas e tem sido aplicadas para melhoria na qualidade dos *softwares* desenvolvidos. Todavia, ainda existem uma vasta gama de problemas que impedem os *softwares* se tornem satisfatórios aos clientes que os utilizam. Dentre estes problemas, surge a manutenção de *software*.

Manutenção de *software* é as alterações no produto após a sua entrega ao cliente e consome mais da metade dos recursos despendidos no desenvolvimento de um *software*. Imagina-se que por ser de tal importância, a atividade de manutenção receba um alto valor nas organizações, mas isso não é o que acontece na realidade.

No mundo real, *softwares* são produzidos em prazos apertados, em meio as constantes alterações de funcionalidades pelos clientes e na pressão da entrega de algo funcional. Com isso, muitos *softwares* são produzidos sem nenhuma preocupação com seu futuro, mesmo sabendo que dias após a entrega serão recebidas solicitações de mudanças.

Dentro desta problemática, surge um conceito que permite verificar a capacidade de um *software* em receber manutenção, denominado manutenibilidade de *software*. Seu diagnóstico permite criar análises daquilo que se necessita ser melhorado para aumentar os níveis de manutenibilidade, consequentemente reduzindo custo com manutenções e ampliando a qualidade do sistema.

Com base nessas proposições, o trabalho se **justificou** pela importância do conhecimento sobre manutenção de *software* e a característica manutenibilidade,

proporcionando conhecimento sobre sua importância para os profissionais de TI que trabalham com desenvolvimento de *softwares*.

O **Problema** estudado figurou em meio ao questionamento: Como determinar a manutenibilidade de um *software* e por que os *softwares* devem ser manuteníveis?

Essa pergunta levantou duas **hipóteses**, sendo a primeira esclarecida por Pressman (2011, p.664), o qual explica que um *software* para ser manutenível deve ter “modularidade eficaz”, “utilizar padrões de projeto”, “usar padrões e convenções de codificação bem definidos”, passar por uma “variedade de técnicas de garantia de qualidade” e ser criado por engenheiros de *software* que saibam que poderão não estar ligados ao projeto quando for detectada a necessidade de manutenção. A segunda hipótese é dada por Sommerville (2007, p. 326) que explica sobre os benefícios na aplicação de investimentos em desenvolvimento para reduzir os custos com manutenção.

O **objetivo geral** deste trabalho constituiu em estudar a conceituação e os benefícios da manutenibilidade de *software*, visando aplicar os conhecimentos obtidos no estudo de caso a realizar no *software* OpenBiblio.

Já os **objetivos específicos** foram a realização de um levantamento bibliográfico que conceituasse o tema, seguido por um estudo do conteúdo e aplicação do conhecimento obtido em estudo de caso no *software* OpenBiblio.

Para o desenvolvimento da monografia, a **metodologia** utilizada para desenvolvimento foi a pesquisa aplicada, com abordagem qualitativa do problema, visando atingir os objetivos através da pesquisa bibliográfica, documental e estudo de caso.

O trabalho foi estruturado em seis capítulos, sendo que o primeiro traz uma introdução sobre o tema. O segundo faz uma conceituação geral que visa situar o leitor sobre termos e conceitos iniciais relativos ao tema do trabalho. O terceiro visa explicar a conceituação do termo manutenção de *software*, fazendo uma abordagem geral sobre sua funcionalidade. Já o quarto capítulo adentra no tema da monografia, a manutenibilidade de *software*, trazendo um estudo sobre suas características. O quinto capítulo traz as informações sobre o estudo de caso realizado e o sexto capítulo apresenta as Considerações Finais, fazendo uma revisão geral do trabalho e seus objetivos, e demais itens pertinentes ao desenvolvimento da monografia.

2 CONCEITUAÇÃO SOBRE ENGENHARIA DE *SOFTWARE*

Este trabalho tem como objetivo tratar do assunto manutenibilidade de *software*. Para abordar este assunto, este capítulo faz uma conceituação sobre *software*, Engenharia de *Software*, IEEE, SWEBOK e Processos de *Software*, situando o leitor para uma melhor compreensão do assunto que será abordado.

O termo *software* é definido por Pressman (2011, p.32) como:

> "[...] instruções (programas de computador) que, quando executadas, fornecem características, funções e desempenho desejados; (2) estruturas de dados que possibilitam aos programas manipular informações adequadamente; e (3) informação descritiva, tanto na forma impressa como na virtual, descrevendo a operação e o uso de programas."

Já Sommerville (2007, p.4) acrescenta que "*Software* não é apenas um programa, mas também todos os dados de documentação e configuração associados, necessários para que o programa opere corretamente".

A Engenharia de Software é definida por Sommerville (2007, p.5) como:

> "[...] uma disciplina de engenharia relacionada com todos os aspectos de produção de *software*, desde os estágios iniciais de especificação do sistema até sua manutenção, depois que este entrar em operação."

O autor Pressman (2011, p.39) acrescenta que a Engenharia de *Software* utiliza metodologias, procedimentos e técnicas para desenvolver e manter *softwares* com prazos e custos coerentes, além de dotá-los de qualidade.

Uma importante e reconhecida instituição para a área de Engenharia de *Software* é o IEEE[1]. Suas publicações de documentos proveem informações que guiam a área de desenvolvimento de *software*, sendo uma importante referência a utilizar. A sigla IEEE (pronuncia-se "I3E") significa *Institute of Electrical and Electronics Engineers* (Instituto de Engenheiros Elétricos e Eletrônicos).

Esta monografia tem um referencial teórico apoiado em dois documentos de propriedade do IEEE: o *Standard for Software Maintenance* (Padrão para Manutenção de *Software)* publicado e aprovado em 1998; e o SWEBOK (*Software*

[1] Mais informações disponíveis em < http://www.ieee.org/about/index.html >

Engineering Body of Knowledge – Conjunto de Conhecimentos sobre Engenharia de *Software*), em sua versão Alpha produzida em 2012.

Dentro da área de produção de *software*, faz-se necessário compreender a parte de Processos de *Software*, que será abordada na subseção um deste capítulo.

2.1 Processos de Software

Segundo Pressman (2011, p.32), “Software é desenvolvido ou passa por um processo de engenharia; ele não é fabricado no sentido clássico”. Os processos para o desenvolvimento de sistemas são denominados Processos de Software, que são definidos por Sommerville (2007, p.6) como “[...] conjunto de atividades e resultados associados que produz um produto de *software*”.

Modelos de processos de desenvolvimento de *software* são definidos por Sommerville (2007, p.43) como “[...] representação abstrata de um processo de um processo de *software”*. Braude (2005, p.22) define como “procedimento seguido pela equipe de desenvolvimento para produzir uma aplicação”.

Para Bellin (1993, p.104) “todas as metodologias estruturadas possuem um ciclo de vida idealizado para todo o processo de desenvolvimento e manutenção de sistemas”. De acordo com o autor, algumas empresas fazem adaptações nestes processos, tornando-os mais próximos de suas realidades. Para Bellin (1993, p.106), o desenvolvimento de um *software*, independente do modelo escolhido possuirá três fases principais: **desenvolvimento** (concepção do que será feito), **implementação** (codificar, testar e colocar o *software* em operação) e **manutenção** (corrigir, ajustar ou adaptar o *software*).

A justificativa para a adoção de um processo de *software* é dada por Silva (2007, p.50):

> “A pressa exagerada em dar solução a um determinado problema pode acarretar outros bem mais complexos para serem resolvidos. Por isso, é imprescindível seguir uma certa metodologia de trabalho com ênfase num planejamento bem elaborado”.

Neste trabalho foram escolhidos três processos de *software*: modelo cascata, modelo espiral e modelo XP. Os dois primeiros modelos possuem fundamentação tradicional e o modelo XP fundamenta-se nas metodologias ágeis.

As metodologias ágeis surgiram do Manifesto para o Desenvolvimento Ágil de *Software*, criado no ano de 2001 e amplamente adotado pelas empresas de desenvolvimento de *software*, por trazer uma abordagem mais prática e realista do processo de desenvolvimento, onde se preza indivíduos e interações, *software* operando, colaboração de clientes e equipe, além de rápidas respostas à mudanças (PRESSMAN, 2011, p.81). Já as metodologias tradicionais possuem toda uma fundamentação teórica e documental que antecedem o processo de desenvolvimento, retardando o produto final de *software* e muitas vezes gerando desconforto e falta de confiança por parte dos clientes para com os desenvolvedores.

2.1.1 Modelo Cascata

O modelo cascata foi o primeiro modelo a ser publicado. Seu nome é proveniente das fases encadeadas entre si (SOMMERVILLE, 2007, p.44). De acordo com Pressman (2011, p.59):

> "O modelo cascata, algumas vezes chamado de ciclo de vida clássico, sugere uma abordagem sequencial e sistemática para o desenvolvimento de *software*, começando com o levantamento de necessidades por parte do cliente, avançando pelas fases de planejamento, modelagem, construção, emprego e culminando no suporte contínuo do *software* concluído".

A Figura 1 mostra o modelo cascata de desenvolvimento. Percebe-se que as três fases iniciais (comunicação, planejamento e modelagem) do desenvolvimento do *software* são totalmente teóricas, produzindo uma gama de textos que servirão como base na fase de construção. Após a construção, inicia-se a fase de emprego onde o *software* é entregue, recebe suporte e manutenção. Deve-se ressaltar que uma fase somente será iniciada após o término da anterior, o que aumenta a espera para o início do desenvolvimento.

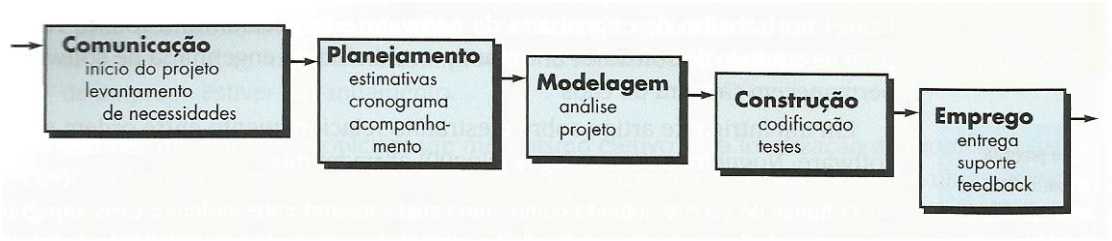

Figura 1: O modelo cascata.
Fonte: Pressman (2011, p.60).

Como forma de melhorar o modelo cascata, foi desenvolvido o modelo em V. Segundo Pressman (2011, p.60), este modelo “descreve a relação entre as ações de garantia da qualidade de software e as ações associadas à comunicação, modelagem e atividades de construção iniciais”.

A Figura 2 apresenta o modelo em V. Neste modelo, à medida que o desenvolvimento prossegue pela flecha do lado esquerdo, a validação da qualidade é garantida pelos processos localizados do lado direito. No modelo em V, o processo torna-se mais rápido se comparado ao modelo cascata, entretanto ainda existe um retardo no desenvolvimento devido a produção excessiva de documentação textual.

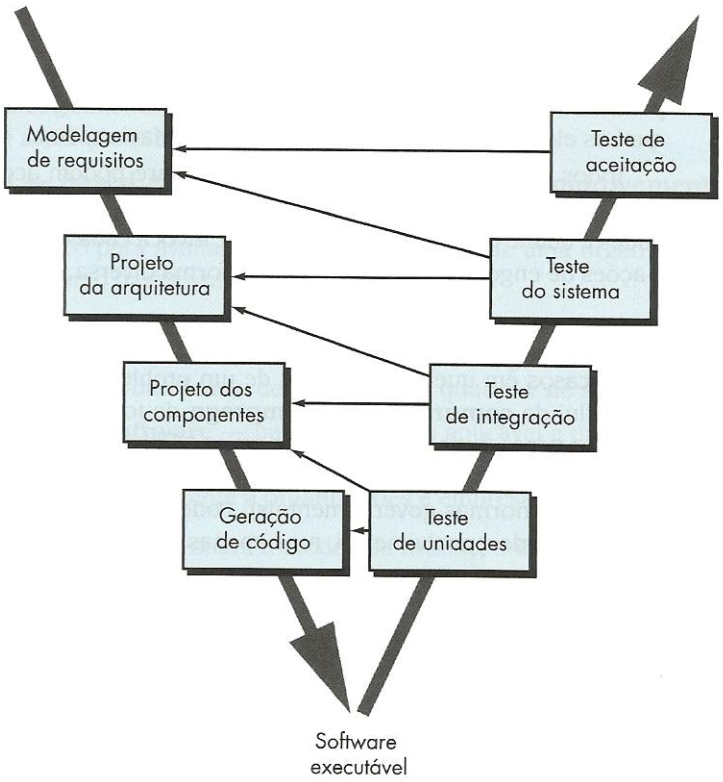

Figura 2: Modelo V.
Fonte: Pressman (2011, P.60).

2.1.2 Modelo Espiral

O modelo espiral se baseia no desenvolvimento de versões de forma evolutiva, em que cada versão incrementa funcionalidades ao produto (PRESSMAN, 2011, p.64).

De acordo com Pressman (2011, p.65), este modelo transmite uma ideia mais realista do desenvolvimento, visto que um software passa por diversas evoluções durante sua vida útil.

Este modelo pode ser considerado um paralelo entre as metodologias tradicionais e as metodologias ágeis, pois apesar de contemplar planejamentos e modelagens (tradicional), faz uma fragmentação no desenvolvimento e torna-o contínuo, fazendo com que haja a entrega de um produto funcional constantemente (ágil).

A Figura 3 mostra a representação do modelo espiral. O modelo possui cinco fases, sendo elas comunicação, planejamento, modelagem, construção e emprego. Na fase de comunicação são esclarecidos detalhes do que será desenvolvido, depois nas fases de planejamento e modelagem são feitas as partes documentais do modelo. Feito isso, segue-se para a fase de construção, onde o *software* é desenvolvido. Por fim, a fase de emprego faz a entrega do produto desenvolvido naquela rodada do ciclo. É possível verificar que o ciclo é contínuo e acontecerá diversas vezes, antes de gerar um produto final.

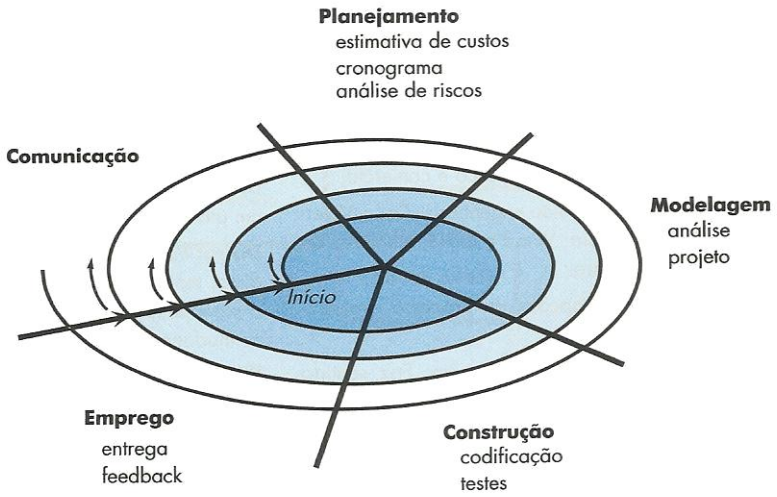

Figura 3: Modelo espiral típico.
Fonte: Pressman (2011, p.65).

2.1.3 Modelo XP: eXtreme Programming

Segundo Sommerville (2007, p.263-264), a *extreme programming* (XP) “é talvez o mais conhecido e mais amplamente usado dos métodos ágeis”. O autor acrescenta que:

> “[...] todos os requisitos são expressos como cenários (chamados histórias do usuário), que são implementados diretamente como uma série de tarefas. Os programadores trabalham em pares e desenvolvem testes para cada tarefa antes da escrita do código. Todos os testes devem ser executados com sucesso quando um novo código é integrado ao sistema. Há um pequeno espaço de tempo entre os *releases* do sistema.”

A Figura 4 ilustra o processo de desenvolvimento XP. Observa-se que o processo é contínuo, ou seja, inicia-se na seleção de histórias e finaliza na avaliação do sistema, voltando em seguida à seleção de uma nova história. Um ponto importante nesta metodologia é a redução da produção textual e o foco na entrega de versões do *software* com maior qualidade.

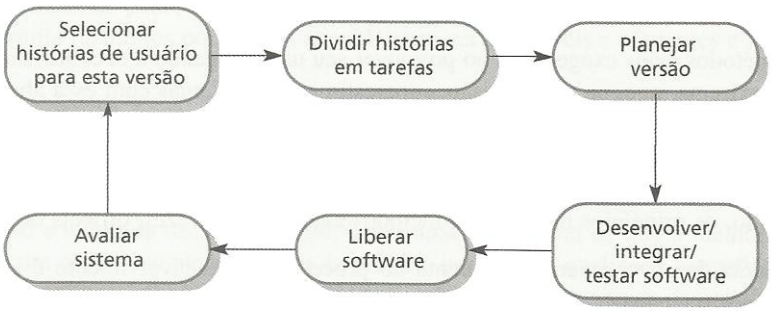

Figura 4: Ciclo de um release em extreme programming.
Fonte: Sommerville (2007, p.264).

Para Pressman (2011, p.87-88), há cinco valores que direcionam as atividades desenvolvidas na XP, sendo eles: comunicação, simplicidade, *feedback*, coragem e respeito. A comunicação deve ser efetiva entre a equipe de desenvolvimento e os clientes, a simplicidade é garantida pelo projeto das funcionalidades a serem desenvolvidas de imediato, o *feedback* é a implementação do *software*, as respostas do cliente e da equipe de desenvolvimento, a coragem aborda a disciplina pois o projeto visa atender as necessidades de hoje, e o respeito entre os envolvidos no projeto.

Na Figura 5 destacam-se quatro fases importantes: planejamento, projeto, codificação e testes. No planejamento são definidas as histórias e seus critérios de aceitação que serão checados ao final da história. Na parte de projeto é feita uma descrição sucinta e criado um protótipo. A codificação é feita em pares, ou seja, dois programadores “atacam” o mesmo problema e desenvolvem uma solução. Nessa fase também são feitos testes para validar o trabalho executado. Por fim, há a fase de testes, onde são feitos os testes de unidade e aceitação. Cada vez que o ciclo XP é executado, uma versão do *software* é produzida. Este processo também é um ciclo contínuo, assim como o modelo espiral, sendo realizado quantas vezes for necessário.

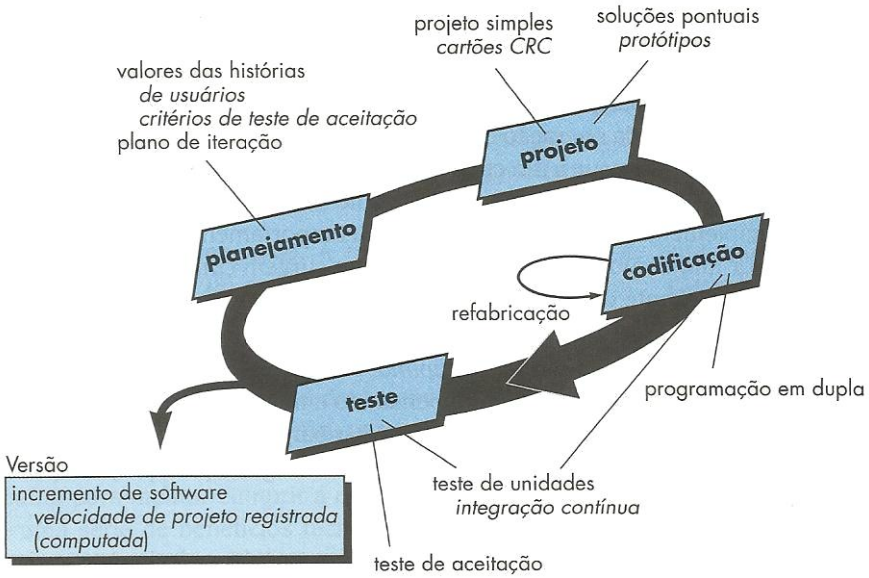

Figura 5: O processo da Extreme Programming (XP).
Fonte: Pressman (2011, p.88).

3 MANUTENÇÃO DE *SOFTWARE*

No capítulo anterior foi exposto que a manutenção de *software* faz parte do ciclo básico de processos de desenvolvimento, sendo um elemento necessário para garantir a estabilidade do produto e aumentar sua durabilidade no mercado.

Sabe-se que a tecnologia da informação trouxe uma série de mudanças nos ambientes organizacionais, proporcionando melhorias na eficácia dos processos e apoio na tomada de decisão. Pressman (2011, p.38) explica que:

> "[...] indivíduos, negócios e governos dependem, de forma crescente, de software para decisões estratégicas e táticas, assim como para o controle e para operações cotidianas. Se o software falhar, as pessoas e as principais empresas poderão vivenciar desde pequenos inconvenientes a falhas catastróficas. Depreende-se, portanto, que um software deve apresentar qualidade elevada".

Tendo como premissa a citação anterior de Pressman, o desenvolvimento de *software* deve resultar em produtos que satisfaçam as necessidades dos clientes, atendendo a todos os requisitos que o usuário necessita, sendo assim, o *software* deverá mudar ou evoluir.

A necessidade de alterações é percebida logo quando o *software* entra em operação, onde são descobertos os defeitos no sistema, mudanças no ambiente de operação e novas necessidades do usuário, tornando necessária a modificação no produto que foi entregue. Sendo assim, a manutenção de *software* torna-se uma parte integral de seu ciclo de vida (IEEE, 2012, p.1).

Para Pressman (2011, p.38), conforme o valor da aplicação cresce, existe a probabilidade de que o número de usuários e a longevidade desta aumentem, fazendo com que haja uma "demanda por adaptação e aperfeiçoamento", o que leva a concluir que "um *software* deve ser passível de manutenção".

Manutenção de *software* é definida pelo IEEE (1998, p.4) como modificação de um produto de *software* pós-entrega para correção de falhas, melhoria de desempenho e outros atributos, ou adaptação de um produto para mudanças do ambiente. Bellin (1993, p.106) acrescenta que a manutenção de *software* é um "pequeno ciclo de vida do sistema", que inclui as fases de análise, projeto e implementação.

Segundo Sommerville (2007, p.326):

> “a manutenção de software é um processo geral de mudanças de um sistema depois que ele é entregue. (...) as mudanças feitas no software podem ser mudanças simples para corrigir erros de codificação, podem ser mudanças mais extensas para corrigir erros de projetos ou melhorias significativas para corrigir erros de especificação ou para acomodar novos requisitos.”

Embora a atividade de manutenção possua processos e características específicas, há procedimentos que se assemelham a atividade de desenvolvimento de *software*, tais como a gestão da configuração (IEEE, 2012, p.7).

O IEEE (2012, p.1) divide a atividade de manutenção em dois estágios: Pré-entrega (*Pre delivery*) e Pós-entrega (*Post delivery*), no qual a pré-entrega se refere as atividades de planejamento de operações pós-entrega, manutenibilidade e determinação da logística de transição de atividades. A pós entrega envolve as modificações de *software* e o treinamento e operação/interação com o *help desk*.

3.1 Tipos de Manutenção

A manutenção de *software* é caracterizada pelo IEEE (1998, p.3-4) em quatro tipos distintos: adaptativa, corretiva, emergencial e perfectiva, descritas a seguir.

- **Manutenção Adaptativa** (*Adaptative Maintenance*) constitui em alterações realizadas para adaptar o *software* às mudanças no ambiente em que ele opera;
- **Manutenção Corretiva** (*Corrective Maintenance*) corresponde às mudanças a serem realizadas para correção de falhas descobertas;
- **Manutenção Emergencial** (*Emergency Maintenance*) envolve as manutenções corretivas não programadas, mas que devem ser realizadas para que o sistema continue operante;
- **Manutenção Perfectiva** (*Perfective Maintenance*) engloba as alterações que possam melhorar o desempenho ou a manutenibilidade do sistema.

O IEEE (2012, p.5) caracteriza as manutenções como corretiva, adaptativa, perfectiva e preventiva, onde a manutenção preventiva é a detecção e correção de defeitos do sistema entregue antes que elas se tornem falhas operacionais.

Segundo Sommerville (2007, p.326), a distinção entre os tipos de manutenção nem sempre será clara quando colocada em prática, pois ao adicionar novas funcionalidades no sistema, defeitos certamente irão surgir. Além disso, o autor explica que “os defeitos de *software* frequentemente são expostos porque os usuários usam o sistema de maneiras imprevisíveis”, sendo que um dos melhores meios de corrigir estes defeitos é modificar o *software* de modo a acomodá-lo com o modo de trabalho dos usuários.

As mudanças no ambiente que requerem manutenções adaptativas que são exemplificadas por Sommerville (2007, p.326) como mudanças no hardware, plataformas, sistema operacional ou mesmo a troca dos *softwares* de apoio que operam integrados a aplicação.

De acordo com Sommerville (2007, p.326), as execuções de mudanças para adaptação a novos ambientes e requisitos consomem a maior parte dos esforços despendidos com a atividade de manutenção. A Figura 6 mostra o gráfico da distribuição dos esforços com manutenção de *software*. Observa-se que o reparo de defeitos ocupa somente 17% dos esforços de manutenção, a adaptação de *software* 18% e a adição ou modificação de funcionalidades 65%.

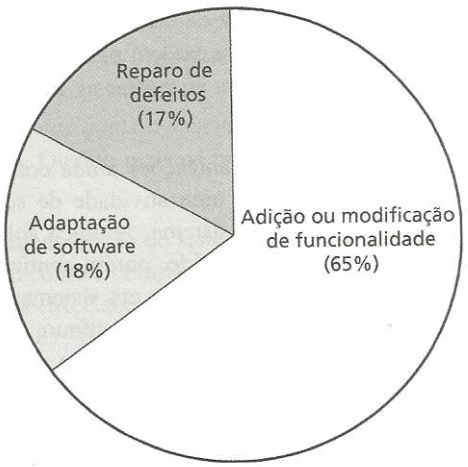

Figura 6: Distribuição dos Esforços de Manutenção.
Fonte: Sommerville (2007, p.327).

3.2 Áreas do conhecimento de manutenção de *software*

O SWEBOK (IEEE, 2012, p.3) divide as áreas do conhecimento de manutenção de *software* segundo a Figura 7. As cinco áreas são: fundamentos da manutenção de software, pontos chaves da manutenção, técnicas de manutenção e ferramentas da manutenção, explicadas no decorrer desta subsessão.

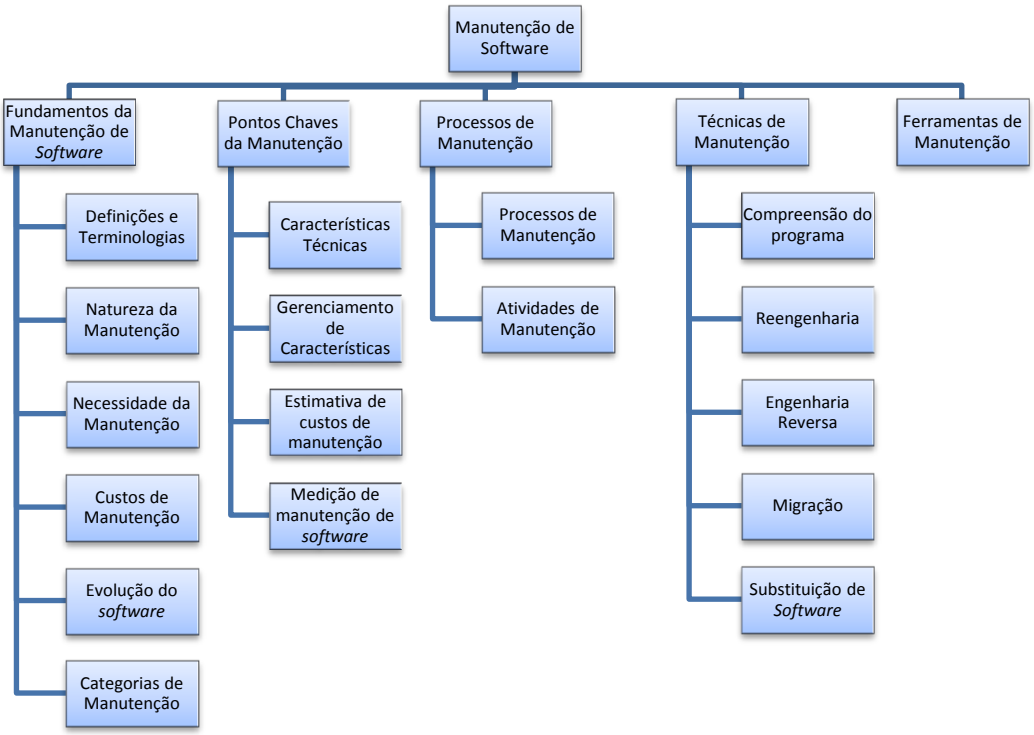

Figura 7: Estrutura de Tópicos de Manutenção de Software.
Adaptado de: IEEE (2012, p.3).

3.2.1 Fundamentos da Manutenção de *Software*

Os fundamentos da manutenção de *software* trazem uma visão geral dos conceitos e terminologias necessárias para entender os processos envolvidos na manutenção de *software*. Eles são divididos em: definições e terminologias, natureza da manutenção, necessidade de manutenção, custos de manutenção, evolução do *software* e categorias da manutenção.

Do ponto de vista das **Definições e terminologias**, manutenção de *software* é essencialmente um dos muitos processos técnicos da engenharia de *software*, cujo objetivo é modificar *softwares* existentes preservando sua integridade (IEEE, 2012, p. 1-2).

A **Natureza da manutenção** organiza os processos a serem executados, através da solicitação da mudança, determinação do impacto, modificações, testes e entrega da nova versão de *software* (IEEE, 2012, p. 2-3).

O mantenedor é o responsável por realizar as atividades de manutenção, devendo aprender sobre o sistema através do conhecimento que lhe deverá ser passado pelo desenvolvedor. Manter um *software* também significa dar suporte ao desenvolvimento, ao código e a documentação de forma progressiva com o ciclo de vida do *software* (IEEE, 2012, p. 2-3).

A **Necessidade de Manutenção** visa continuar satisfazendo os requisitos do usuário através da correção de falhas, melhoria do *design* do *software*, implementação de melhorias, comunicação entre sistemas, adaptações com diferentes plataformas, hardwares, *softwares*, sistemas ou qualquer outro meio que venha a interagir com o produto (IEEE, 2012, p 3-4).

No âmbito dos **Custos de Manutenção**, cerca de 80% de todas as manutenções realizadas não são de caráter corretivo. A atividade de manutenção consome grande parte dos recursos financeiros destinados ao sistema. Dessa forma, compreender suas categorias e os fatores que influenciam na manutenibilidade do *software* pode ajudar a reduzir custos e aumentar a estabilidade da aplicação no mercado (IEEE, 2012, p.4).

A **Evolução do *software*** é auxiliada pela atividade de manutenção, caracterizada como um processo evolucionário do *software*, auxiliado pelo entendimento sobre o que acontece com o sistema no passar do tempo. Um *software* jamais será completo e sempre necessitará evoluir. Quando essa evolução acontece, devem ser tomadas atitudes que reduzam a complexidade da aplicação (IEEE, 2012, p.4).

As **Categorias de Manutenção** são corretiva, adaptativa, perfectiva e preventiva, explanadas na subseção 2. As manutenções que visam corrigir falhas e defeitos do sistema são as corretivas e preventivas. Já as manutenções que visam aplicar melhorias são a adaptativa e perfectiva (IEEE, 2012, p.5).

3.2.2 Pontos Chave da Manutenção de Software

Manter um *software* requer a superação de inúmeros desafios inerentes da atividade de manutenção. Por exemplo, planejar novas versões do sistema em paralelo com as correções emergenciais que garantam seu correto funcionamento é um dos desafios comuns enfrentados pelas equipes de desenvolvimento. Os pontos chaves abordam quatro tópicos da manutenção, sendo eles: características chave, gerenciamento de características, estimativa de custos de manutenção e medição da manutenção de *software* (IEEE, 2012, p.5).

As **Características Técnicas** englobam os itens necessários para a realização da manutenção do sistema, sendo subdividas em em quatro partes: entendimento do sistema, testes, análise de impacto e a manutenibilidade do *software* (IEEE, 2012, p.5).

O mantenedor deverá compreender o código-fonte, as funcionalidades, o funcionamento e a documentação referente ao *software* para que ele saiba onde serão feitas as modificações (IEEE, 2012, p.5).

Testes de *software* devem ser realizados nas partes alteradas do sistema, ajudando a garantir a validade das modificações e impedindo que problemas na aplicação sejam replicados. Constitui-se um desafio encontrar tempo para realizar os testes enquanto os membros da equipe estão envolvidos em outras atividades (IEEE, 2012, p.6).

Analisar o impacto que a manutenção de *software* poderá causar no sistema possibilita identificar tudo que será afetado pela requisição da mudança, estimando os recursos necessários, a gravidade do problema e elaborando as possíveis soluções para tal (IEEE, 2012, p.6).

A manutenibilidade é a capacidade de um produto de *software* para receber alterações, sejam elas de caráter corretivo, adaptativo ou com finalidades de melhorias. Essa característica particular do *software* influencia diretamente em sua qualidade, tornando viável a realização de um controle da mesma durante o desenvolvimento do *software*, visando reduzir os custos com manutenção (IEEE, 2012, p.7).

O **Gerenciamento de Características** deve alinhar os objetivos organizacionais da empresa desenvolvedora com o produto de *software* que ela

possui. Dessa forma, é possível demonstrar positivamente como as atividades de manutenção de *software* irão impactar no retorno (lucro) para a empresa, tendo em vista que há uma dificuldade das organizações em compreender a necessidade do consumo de recursos com manutenção (IEEE, 2012, p.7).

Destacam-se neste subitem três aspectos importantes: a valorização do mantenedor de *software*, as decisões de aspecto organizacional e a escolha por *outsourcing*[2].

A valorização da figura do mantenedor é um aspecto que deve ser trabalho, principalmente de forma a conscientizar a equipe de desenvolvimento. Em geral, funcionários que realizam manutenção de *software* não obtém um reconhecimento da função exercida, recebendo estereótipos ruins e deixando de sentir vontade em realizar manutenção em aplicações (IEEE, 2012, p.7).

Decidir os aspectos organizacionais da manutenção organiza os processos a serem realizados. As decisões envolvem a determinação da equipe mantenedora, que nem sempre será a equipe de desenvolvimento e a avaliação da solicitação em particular, visando aderir a melhor solução possível (IEEE, 2012, p.8).

Segundo o IEEE (2012, p.8), *Outsourcing* é uma escolha a ser feita quando há uma missão crítica a realizar no *software*. As empresas tem receio desta opção, tendo em vista a perda do controle de parte do produto de *software*. A função dos *outsourcers* é determinar o escopo dos serviços de manutenção requeridos, os termos de acordo do serviço e os detalhes contratuais, reduzindo e agilizando o trabalho da empresa contratante.

As **Estimativas de Custos de Manutenção** visam compreender as categorias de manutenção de *software* e fazer previsões dos custos de formas mais precisas (IEEE, 2012, p.8).

Essa estimativa requer uma avaliação dos fatores técnicos e não técnicos que podem influenciar esta atividade. Essa determinação pode ser apoiada com modelos matemáticos combinados com a experiência da pessoa que faz a estimativa. Em ambos é importante possuir um histórico de dados de manutenções passadas, que irão calibram as equações e prover os resultados desejados.

Aliar a tais formas a experiências passadas com manutenções permite traçar uma estimativa de custos mais eficaz, que leva em conta a realidade do

[2] *Outsourcing*: terceirização dos serviços.

produto, do cliente e a natureza da solicitação. Os atributos que compõe a previsão devem ser determinados pelo mantenedor, que deverá se valer daquilo que é importante para sua empresa, com base no contexto organizacional em que a mesma atua (IEEE, 2012, p.9).

O último ponto chave é a **Medição da Manutenção de *Software* que** engloba os vários atributos a serem analisados na atividade de manutenção, onde há um inter-relacionamento entre os processos, os recursos e os produtos (IEEE, 2012, p.9).

3.2.3 Processos de Manutenção

O ramo dos Processos de Manutenção subdivide-se em dois, sendo os Processos de Manutenção e as Atividades de Manutenção, ambos descritos nesta subseção.

Os **Processos de Manutenção** envolvem os processos de implementação da mudança, análise do problema e da modificação, revisão e aceitação da mudança realizada, migração para o produto alterado e a substituição do *software* (IEEE, 2012, p.10).

As **Atividades de Manutenção** englobam além das atividades comuns ao desenvolvimento do *software*, as atividades únicas referentes ao processo manutenitivo, tais como as atividades de suporte, o planejamento das atividades de manutenção, a gestão da configuração para manutenção e a qualidade de *software* pós-modificação do produto (IEEE, 2012, p.11).

As Atividades Únicas subdividem-se em entendimento do programa, transição, aceitação/rejeição de requisição de mudanças, *help desk* de manutenção, análise de impacto e acordo de nível de serviços de manutenção (IEEE, 2012, p.11).

O entendimento do programa traz ao mantenedor um conhecimento geral do funcionamento do *software* e da integração dos módulos que este possui, proporcionando a correta alteração com o mínimo de impacto (IEEE, 2012, p.11).

A transição é a sequência de atividades controladas e coordenadas, a serem realizadas durante a transferência do desenvolvedor para o mantenedor do sistema (IEEE, 2012, p.11).

A aceitação/rejeição de requisições de mudanças é feita através de uma análise realizada pela organização detentora do *software*, onde são avaliadas as solicitações de modificações propostas pelo usuário, verificando o tamanho, o esforço e a complexidade requeridos para atender a solicitação. Além disso, a manutenção pode ser rejeitada ou transferida para o setor de desenvolvimento, (IEEE, 2012, p.11).

O *help desk* de manutenção é a união entre as experiências do usuário final e as funções coordenadas de suporte com o objetivo de priorizar as solicitações de manutenção e estimar os custos de requisitos de mudança (IEEE, 2012, p.11).

O acordo de nível de serviços de manutenção, licenças e contratos de manutenção são os acordos contratuais que descrevem os serviços a serem prestados e seus objetivos. Os documentos contratuais são também uma forma de organizar, controlar e gerenciar as solicitações dos clientes, aumentando a qualidade do histórico de manutenção do *software* que será utilizado na medição da manutenção, explicada no tópico 3.2.2 (IEEE, 2012, p.11).

Por sua vez, as Atividades de Suporte são aquelas que auxiliam na manutenção de *software*, provendo a documentação das mudanças realizadas, a gestão da configuração, verificação e validação de requisitos, resolução dos problemas apresentados, segurança da qualidade de *software*, revisões e auditorias. É também responsabilidade do suporte treinar os mantenedores e os usuários do sistema (IEEE, 2012, p.11).

Já o Planejamento das Atividades de Manutenção contempla o planejamento de negócios (nível organizacional), o planejamento de manutenção de *software* (nível de transição), o planejamento de liberação de versões (nível de *software*) e o planejamento de requisições de mudanças individuais do *software* (nível de requisição). Esses documentos devem englobar o escopo das manutenções, os processos de adaptações do *software*, identificação da organização da manutenção e a estimativa de custos, sendo o planejamento também parte do histórico de manutenção, apontado na subseção 3.2.2 (IEEE, 2012, p.11-12).

Outro documento importante é o Plano de Manutenção de *Software,* que deve ser elaborado durante o desenvolvimento da aplicação, especificando como será o reporte de problemas e as solicitações de mudanças no sistema (IEEE, 2012, p.12).

Além disso, as atividades de manutenção devem contemplar a gestão da configuração de software. Segundo Pressman (2011, p.515), a configuração de *software* é o conjunto de programas de computador, produtos que descrevem estes programas e os dados ou o próprio conteúdo que os preenchem. A gestão da configuração é "[...] um conjunto de atividades que foram desenvolvidas para gerenciar alterações através de todo o ciclo de vida de um *software"*. Para o IEEE (2012, p.12) ela é um dos elementos críticos da manutenção de sistemas, pois cria um controle organizado das mudanças realizadas no *software*, devendo fornecer os procedimentos necessários para verificar, validar e auditar os passos que serão utilizados para identificar, autorizar, implementar e liberar o produto de *software*.

O IEEE (2012, p.13) ressalta que a Qualidade de *Software* não deve ser tratada como um resultado proveniente das atividades de manutenção do *software*. Para que o produto seja satisfatório ao cliente, é necessário que haja um controle eficiente de qualidade desde o desenvolvimento até a manutenção. Segundo Sommerville (2007, p.424):

> "bons gerentes de qualidade têm por objetivo desenvolver uma 'cultura de qualidade' na qual todos os responsáveis pelo desenvolvimento de produto estão comprometidos em atingir um alto nível de qualidade do produto. Eles encorajam a equipe a assumirem a responsabilidade pela qualidade de seus trabalhos e a desenvolverem novas abordagens para o aprimoramento da qualidade."

3.2.4 Técnicas de Manutenção

Dentre todas as técnicas utilizadas pelos mantenedores para manutenções de *software*, destacam-se a compreensão do programa, a reengenharia, a engenharia reversa, a migração e a substituição.

A **Compreensão do Programa** é a leitura atenta do código fonte e da documentação, preferencialmente clara e concisa, para o entendimento do *software* que será modificado (IEEE, 2012, p.13).

A **Reengenharia** visa examinar o *software* e alterá-lo para reconstituir de uma forma nova um produto já existente, sendo uma ferramenta de apoio para substituição de *software* legado. Uma das técnicas de reengenharia amplamente

utilizada é a refatoração, que visa reorganizar o programa sem modificar seu comportamento (IEEE, 2012, p.13).

A **Engenharia Reversa** é o processo de analisar e identificar os componentes do *software* e seus relacionamentos, criando representações e abstrações de alto nível. Este processo é passivo, ou seja, não resulta em um novo *software*, apenas provê uma visão oriunda do código fonte existente (IEEE, 2012, p.13). Para Pressman (2011, p.669, 670) é:

> "[...] o processo para analisar um programa na tentativa de criar uma representação do programa em um nível mais alto de abstração do que o código-fonte (...) é um processo de recuperação do projeto".

A **Migração** requer a criação de um plano que determine as ações necessárias, os requisitos, ferramentas, conversão do produto e dados, execução, verificação e suporte (IEEE, 2012, p.14).

A **Substituição** deve ser executada quando a vida útil do sistema chega ao fim. A decisão de substituir um *software* deve ser executada após uma análise cautelosa das causas e consequências de tal ato (IEEE, 2012, p.14).

3.2.5 Ferramentas de Manutenção

Além do referencial teórico e documental que auxilia as atividades de manutenção, podem ser utilizadas ferramentas de apoio para a compreensão e modificação do programa, tais como analisadores estáticos, analisadores dinâmicos, analisadores de fluxo de dados, entre outros (IEEE, 2012, p.14).

3.3 Problemas Comuns da Manutenção de *Software*

O processo manutenitivo de *software* é rodeado de problemas que se iniciam no desenvolvimento e acarretam, dentre várias consequências, o aumento considerável nos custos para manter o *software* em funcionamento. Pressman (2011, p.663) explica que a manutenção consome de 60% a 70% dos recursos destinados ao *software*.

Para o mesmo autor, a mobilidade dos profissionais é outro fator problema para manter o *software*. Em geral, a equipe que desenvolveu o projeto original pode não estar mais na organização, ou mesmo os profissionais que fizeram manutenções anteriores podem ter saído da empresa, podendo ainda não haver alguém com conhecimento sobre o *software* que precisa ser mantido (PRESSMAN, 2011, p.663).

Sommerville (2007, p.327) aponta cinco fatores relevantes que contribuem para os altos custos com a manutenção de *software*: estabilidade da equipe, responsabilidade contratual, idade e estrutura do programa e o menosprezo pela atividade de manutenção.

Assim como Pressman (2011, p.663), Sommerville (2007, p.327) aponta o problema da rotatividade da equipe desenvolvedora, gerando a necessidade de contratação de funcionários que não conhecem o sistema, demandando grande esforço para compreensão do *software* antes de aplicar as alterações.

A responsabilidade contratual é outro fator agravante, pois geralmente os contratos de desenvolvimento e manutenção são separados, podendo este último ser realizado por outra empresa. Isso faz com que os desenvolvedores não se preocupem com a qualidade dos códigos desenvolvidos e tampouco com a facilidade que estes devem possui para receber manutenção (SOMMERVILLE, 2007, p.327).

Embora a atividade de manutenção requeira profissionais capacitados, é comum que empresas aloquem funcionários recém-contratados para manter *softwares*. Em geral, esses funcionários são inexperientes e não possuem familiaridade com a aplicação. A causa desses remanejamentos incorretos é a visão errônea de que a manutenção de *software* é uma atividade que exige menos responsabilidade (SOMMERVILLE, 2007, p.327).

À medida que a idade do programa aumenta, sua estrutura se degrada por não possuir as mais novas técnicas/tecnologias da engenharia de *software*, tornando difícil compreender e modificar a aplicação. Sistemas legados, por exemplo, podem ter sido desenvolvidos sem nenhuma técnica de engenharia de *software*, além de ter uma documentação perdida, inconsistente ou inexistente (SOMMERVILLE, 2007, p. 327-328).

O menosprezo das atividades de manutenção é oriundo da visão preconceituosa onde se valoriza o desenvolvimento e trata-se a manutenção como

atividade de “segunda classe”, fazendo com que o incentivo para essa seja quase inexistente (SOMMERVILLE, 2007, p.328).

Sommerville (2007, p.328) propõe algumas soluções para tais problemas que envolvem principalmente a conscientização dos desenvolvedores para com a atividade de manutenção de *software*. Para o autor, a organização deve pensar em sistemas que irão evoluir com o passar do tempo, portanto o *software* deve ser desenvolvido com técnicas de engenharia de *software* que melhorem a estrutura do programa.

Deve-se prever as partes do sistema que possuem maior complexidade para receber manutenção e avaliar a facilidade de manutenção dos componentes afetados do sistema (SOMMERVILLE, 2007, p.328).

As mudanças tem uma tendência natural para degradar a estrutura do *software*, reduzindo sua facilidade de manutenção e aumentando os custos para manter o sistema, visto que os custos de manutenção de *software* estão diretamente ligados à facilidade de modificação do programa (SOMMERVILLE, 2007, p.328).

4 MANUTENIBILIDADE

Após compreender os processos relacionados à manutenção de *software* apresentado no capítulo anterior, observa-se que há uma profunda ligação da atividade de manutenção com a qualidade do produto, tendo em vista que, uma vez que o produto de *software* não possa ser modificado, ele deixa de atender às necessidades do cliente, diminuindo sua qualidade. Para saber se um *software* pode receber manutenção, é importante realizar um estudo sobre a manutenibilidade de *software*. Manutenibilidade é a abordagem das características necessárias para que o *software* consiga receber manutenção. Seu conceito é explorado na norma SQuaRE(IS0/IEC 25000).

4.1 SQuaRE : ISO/IEC 25000

A norma SQuaRE significa *Software product Quality Requirements and Evaluation*, ou Requisitos de Qualidade e Avaliação do Produto de Software, sendo também conhecida como ISO/IEC 25000 (KOSCIANSKI e SOARES, 2007, p.204).

O surgimento dessa norma deu-se através da união entre as normas ISO/IEC 9126, que apresentava os instrumentos necessários para avaliar e aferir de forma qualitativa e quantitativa a presença da qualidade no *software,* e a norma ISO/IEC 14598 que engloba aspectos gerenciais e indica metodologias e documentações relevantes (KOSCIANSKI e SOARES, 2007, p. 204-205).

De acordo com Koscianski e Soares (2007, p.204), a norma SQuaRE apresenta medições e padrões para aferir a qualidade de *software*, abordando seis características chave, sendo elas: funcionalidade, manutenibilidade, usabilidade, confiabilidade, eficiência e portabilidade.

4.2 Manutenibilidade

Koscianski e Soares (2007, p.212) definem manutenibilidade como a “[...] facilidade de modificação de um produto de *software*”. Tal característica envolve

uma análise de métricas que possibilitem diagnosticar a complexidade de manter um determinado produto de *software*.

Para Pressman (2011, p.361), a facilidade de manutenção acontece quando um *software* pode ser adaptado ou corrigido num curto espaço de tempo, provendo também todas as informações necessárias para fazer as modificações.

De acordo com Pressman (2011, p.664), um software manutenível apresenta as seguintes características:

> "[...] apresenta uma modularidade eficaz (...), utiliza padrões de projeto (...) que permitem entendê-lo facilmente. Foi construído usando padrões e convenções de codificação bem definidos, levando a um código-fonte autodocumentado e inteligível. Passou por uma variedade de técnicas de garantia de qualidade (...) que descobriu potenciais problemas de manutenção antes que o software fosse lançado. Foi criado por engenheiros que reconhecem que não estarão por perto quando as alterações tiverem de serem feitas. Portanto, o projeto e a implementação do software deve 'ajudar' a pessoa que for fazer a alteração."

A manutenibilidade é dividida em quatro características principais, denominadas modificabilidade, analisabilidade, estabilidade e testabilidade (KOSCIANSKI e SOARES, 2007, p.212).

Segundo KOSCIANSKI e SOARES (2007, p.213), a **modificabilidade** faz referência ao processo de implementação de *software* e pode ser traduzida como as alterações realizadas no código-fonte. Essa característica pode ser melhorada com a documentação interna do produto, arquitetura adequada de *software* e a clareza no código-fonte.

Para Koscianski e Soares (2007, p.213), a **analisabilidade** caracteriza o quão fácil é identificar as deficiências, falhas e defeitos, bem como a causa que gerou os comportamentos anômalos.

A **estabilidade** do *software* é a "[...] capacidade de o produto evitar que modificações levem a efeitos inesperados" (KOSCIANSKI e SOARES, 2007, p.213).

KOSCIANSKI e SOARES (2007, p.213) ainda explicam que a **testabilidade** é a "[...] capacidade de o *software* permitir que, uma vez modificado, seja validado".

4.3 Métricas para Manutenibilidade de Software

Segundo Pressman (2011, p.538), a engenharia de *software* difere das outras disciplinas relacionadas às engenharias por não se fundamentar em leis quantitativas, sendo muitas vezes avaliada de uma forma qualitativa. Embora possua um grau de dificuldade ampliado devido à subjetividade com que muitas vezes são implantadas, as métricas constituem um auxílio para a criação de *softwares* com maior qualidade.

O conceito de métricas envolve o entendimento de três termos (com o foco voltado para a engenharia de software): medida, medição e métricas. As **medidas** são aquelas que fazem indicações quantitativas, de capacidade ou tamanho de atributos de produto ou processo. A **medição** é a determinação da medida. A **métrica** é “uma medida quantitativa do grau com o qual um sistema, componente ou processo possui determinado atributo” (PRESSMAN apud IEEE, 2011, p.539).

Para Koscianski e Soares (2007, p.226) tanto o custo quanto a complexidade da aplicação das métricas devem ser compatíveis com os benefícios da avaliação a ser realizada. Além disso, as métricas precisam ser repetíveis, reproduzíveis, objetivas e imparciais para que seu resultado possa ser benéfico no aumento da qualidade do *software*.

Na manutenibilidade de *software*, o uso de métricas permite prever o esforço necessário para modificar o *software*, e a criação de um histórico de dados que acompanhe todo o processo de desenvolvimento da aplicação (KOSCIANSKI; SOARES, 2007, p.228).

4.3.1 Pontos de Função

Ponto de função é uma métrica voltada para a medição de tamanho de software, onde se leva em consideração as funcionalidades e a complexidade das linhas de código (KOSCIANSKI e SOARES, 2007, p.230).

Para Pressman (2011, p.543), essa métrica é uma forma efetiva de medir as funcionalidades oferecidas pelo sistema e, quando aliada a dados históricos, pode ser utilizada para “[...] estimar o custo necessário para projetar, codificar e

testar o software (...) prever o número de componentes e/ou o número de linhas projetadas de código-fonte no sistema implantado".

Koscianski e Soares (2007, p. 230) explicam que essa métrica pode ser aplicada antes mesmo do início da codificação, através da descrição da arquitetura do projeto, sendo útil para estimar o esforço da implementação e o cronograma do projeto. Ressaltam, também, que ela pode ser aplicada em qualquer plataforma e em sistemas já em funcionamento.

A determinação do ponto de função considera as entradas, saídas e consultas fazendo a contagem de dados e transações realizadas. A contagem de dados considera "[...] arquivos lógicos internos ou arquivos de interface externos". Já as transações observam as "[...] entradas externas, saídas externas ou consultas externas". Os arquivos são os "[...] grupos de dados logicamente relacionados" (KOSCIANSKI e SOARES, 2007, p.230-231).

De acordo com Pressman (2011, p.544), a métrica de pontos de função deverá considerar o número de entradas externas, o número de saídas externas, o número de consultas externas, o número de arquivos lógicos internos e o número de arquivos de interface externos. A seguir serão explicados cada um destes termos.

O **número de entradas externas** são os dados fornecidos pelo usuário ou outras aplicações. O **número de saídas externas** é composto pelos dados que se derivam da própria aplicação e gera informações para o usuário, ou seja, telas, relatórios e mensagens. O **número de consultas externas** são as entradas que resultam em respostas do sistema geradas na forma de saídas. O **número de arquivos lógicos internos** corresponde ao "[...] agrupamento lógico de dados que reside dentro das fronteiras do aplicativo e é mantido através de entradas externas". O **número de arquivos de interface externos** agrupa os dados residentes fora da aplicação, mas fornece conteúdo útil que poderá ser utilizado pela própria aplicação (PRESSMAN, 2011, p.544).

O multiplicador denominado Fator de Ajuste baseia-se em quatorze características do sistema, sendo elas: comunicação de dados (1), funções distribuídas (2), desempenho (3), configuração do equipamento (4), volume de transações (5), entrada de dados on-line (6), interface com o usuário (7), atualizações on-line (8), processamento complexo (9), reusabilidade (10), facilidade de implantação (11), facilidade operacional (12), múltiplos locais (13) e flexibilidade a mudanças (14), onde, para cada categoria (n_i), atribui-se um valor entre 0 (nenhuma

influência) e 5 (influência total), para que possam ser realizados os cálculos com base na Equação 1 (KOSCIANSKI e SOARES, 2007, p.323):

$$FA = 0{,}65 + 0{,}01 \times (n1 + n2 + \cdots + n14)$$

Equação 1: Fator de Ajuste

Após realizar o cálculo do Fator de Ajuste (*FA*), calcula-se o número de pontos de função (PFA) a partir da Equação 2 (KOSCIANSKI; SOARES, 2007, p.232):

$$PFA = FA \times PF$$

Equação 2: Número de Pontos de Função

4.3.2 Complexidade Ciclomática

Segundo Pressman (2011, p. 434), “a complexidade ciclomática tem um fundamento na teoria dos grafos e fornece uma métrica de *software* extremamente útil”. Koscianski e Soares (2007, p.234) explicam que a complexidade ciclomática faz uma caracterização numérica do código-fonte, mediante uma avaliação da quantidade diferente de caminhos de execução.

De acordo com Koscianski e Soares (2007, p.234):

> “A premissa básica é que o nível de alinhamento de laços e comandos de decisão no código tem relação com a complexidade de execução e com a complexidade psicológica, ou seja, o esforço necessário para compreendê-lo. Exemplificando, um comando *if* abre duas possibilidades de execução diferentes, o que pode significar o dobro de esforço de teste e verificação.”

O cálculo da complexidade ciclomática é realizado com base num grafo que represente os caminhos possíveis de execução do programa. Os nós são representados graficamente por círculos, sendo os comandos. As arestas ou ligações são representadas por setas, indicando o fluxo de execução. Segundo Koscianski e Soares (2007, p.234), “cada nó do grafo corresponde a um trecho de código ou um comando”. O grafo de fluxo é representado conforme a Figura 8. Nota-se que no grafo os comandos de decisão também são representados (cada um em

sua forma específica) nos nós, mostrando claramente o fluxo de execução do *software*.

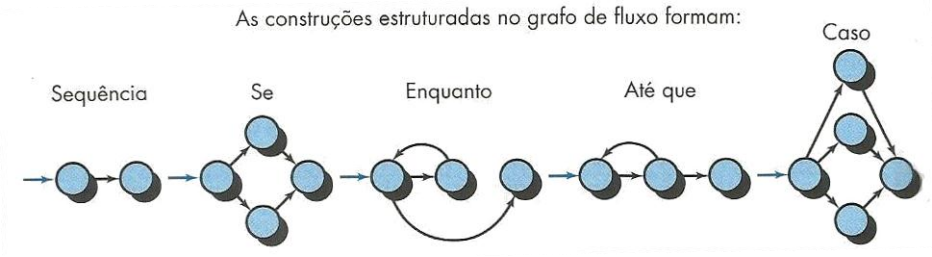

Figura 8: Notação de grafo de fluxo.
Fonte: Pressman (2011, p.432).

A Figura 9 foi apresentada por Koscianski e Soares (2007, p.235), exemplificando os nós de um código-fonte e seu respectivo grafo de fluxo. Observa-se que o nó número 1 representa três linhas de código, sendo assim agrupado por possuir instruções similares que não afetam no caminho de execução. Já os nós 2 e 3, embora possuam instruções similares (comando de decisão *if*), acarretam em caminhos de execução diferentes, sendo representadas em nós diferentes. Os nós 4 e 5 distribuem a execução do comando *while*.

```
int fib (int n) {
    int a = 1;
1   int b = 1;
    int c = 2;

2   if (0 == n) return 0;
3   if (3 >  n) return 1;

4   while (n-- > 2) {
       c = a+b;
5      a = b;
       b = c;
    }
6   return c;
```

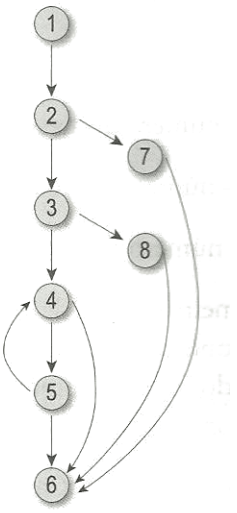

Figura 9: Exemplo de grafo de execução de um programa.
Fonte: Koscianski; Soares (2007, p. 235).

Cada bloco numerado representa um nó no grafo de fluxo. Com base nos nós, é possível definir os caminhos de execução ou caminhos independentes. Segundo Pressman (2011, p.433), “um *caminho independente* é qualquer caminho

através do programa que introduz pelo menos um novo conjunto de comandos de processamento ou uma nova condição". No exemplo da Figura 9, os caminhos possíveis seriam:

1 – 2 – 7

1 – 2 – 3 – 8

1 – 2 – 3 – 4 – 6

1 – 2 – 3 – 5 – 4 – 5 - ... – 4 – 6

Para este exemplo, existem quatro caminhos possíveis de execução, ou seja, a complexidade ciclomática é quatro. A equação para calcular a complexidade ciclomática é dada pela Equação 3:

$$v(G) = P + 1$$

Equação 3: Complexidade Ciclomática verificada por nós predicados

onde $v(G)$ representa a complexidade ciclomática e P o número de nós predicados. Os nós predicados são aqueles que apresentam condições que alteram o fluxo de execução (PRESSMAN, 2011, p.434).

Aplicando os valores numéricos a Equação 3, a representação matemática seria:

$$v(G) = 3 + 1$$
$$v(G) = 4$$

Outra maneira de obter a complexidade ciclomática é dada pela Equação 4, onde *E* corresponde ao número de arestas contidas no grafo de fluxo e *N* aos nós do grafo (PRESSMAN, 2011, p.434).

$$v(G) = E - N + 2$$

Equação 4: Complexidade Ciclomática verificada por arestas e nós

4.3.3 Acoplamento e Coesão

O acoplamento e a coesão têm por finalidade analisar a "[...] dependência entre componentes de um programa, sejam sub-rotinas, objetos ou módulos inteiros" (KOSCIANSKI e SOARES, 2007, p.238).

Koscianski e Soares (2007, p.238) definem coesão como “[...] associação entre dois componentes de um *software* em relação à realização de uma dada tarefa”. Para Pressman (2011, p.266) no paradigma da orientação a objetos, a “[...] coesão implica um componente ou classe encapsular apenas atributos e operações que estejam intimamente relacionados entre si e com a classe ou o componente em si”.

Acoplamento é definido por Koscianski e Soares (2007, p.239) como “[...] o grau de associação entre dois componentes”. Pressman (2011, p.267) acrescenta que é uma medida de caráter qualitativo que caracteriza o grau do relacionamento/ligação entre classes. Quanto maior a interdependência das classes, maior será o acoplamento.

Para verificar o acoplamento de um sistema, devem-se classificar os acoplamentos em doze categorias, representadas por *i*. Depois, deve ser contabilizado o número de ocorrências para cada categoria (n_i). Os valores serão armazenados na Equação 5. Ressalta-se que *i* é o maior valor para a qual existe $n_i > 0$ (KOSCIANSKI; SOARES, 2007, p.239).

$$m = i + \frac{n_i - 1}{2\,(n_i + 1)}$$

Equação 5: Acoplamento do Sistema

Segundo Koscianski e Soares (2007, p.239) é desejável que haja um alto grau de coesão, o que significa que o código está bem estruturado, e um baixo grau de acoplamento, o que facilita a compreensão do *software* e reduz as dificuldades na manutenção.

4.3.4 Índice de Maturidade do *Software*

Pressman (2011, p.562) apresenta uma métrica específica para atividades de manutenção de *software*, denominada índice de maturidade de *software*. Para efetuar o cálculo, considera-se o número de módulos na versão atual(M_t), número de módulos na versão atual que foram alterados(F_c), número de módulos na versão atual que foram acrescentados (F_a) e o número de módulos da versão anterior que foram excluídos da versão atual (F_d). Os dados são aplicados na Equação 6:

$$SMI = \frac{M_t - (F_a + F_c + F_d)}{M_t}$$

Equação 6: Índice de Maturidade do *Software*

Quanto mais próximo de 1.0, mais estável o produto. Segundo Pressman (2011, p.562), essa métrica também pode ser utilizada para planejar a manutenção de *software*.

5 ESTUDO DE CASO

Conforme proposto no objetivo geral, apresentado no capítulo 1, este trabalho monográfico visa aplicar os conhecimentos adquiridos com a revisão bibliográfico presente nos capítulos 2, 3 e 4 através de um estudo de caso que visa determinar a manutenibilidade do *software* OpenBiblio.

O *software* OpenBiblio foi escolhido por possuir uma licença de distribuição livre. Um *software* de caráter livre (também conhecido como *open source*) difere dos softwares proprietários por liberar seu uso gratuitamente e disponibilizar o código fonte, permitindo sua cópia, alteração e redistribuição, desde que sejam seguidos os termos de licença propostos pelos desenvolvedores. O oposto de um *software open source* é o *software* proprietário, limitado por licenças de uso, sem código fonte aberto e, em sua maioria, requerendo o pagamento ao proprietário para aquisição da licença (TAURION, 2004, p.15,17).

Como o estudo da manutenibilidade de *software* prevê uma avaliação detalhada do *software* como um todo (código fonte e documentação), se optou por escolher um *software* livre que pudesse fornecer as informações necessárias para realização do estudo. Para aproximar o estudo de caso à realidade dos leitores deste trabalho, foi escolhido um *software* para gestão de bibliotecas, o OpenBiblio.

5.1 OpenBiblio

O *software* OpenBiblio é um gestor de bibliotecas *open source*, podendo ser utilizado, alterado e distribuído livremente. Informações adicionais pode ser encontradas no *site* do projeto de desenvolvimento[3].

Como este *software* é livre, seu desenvolvimento é dado por uma equipe principal e pelos colaboradores. A equipe principal é fixa e constituída por pessoas que têm a responsabilidade de continuar o projeto, desenvolvendo novas funcionalidades, avaliando os trechos de códigos dos colaboradores e liberando novas versões oficiais. Por sua vez, a equipe de colaboradores são os usuários avançados do sistema que possuem necessidades específicas e fazem implementações para supri-las. Os colaboradores disponibilizam os trechos de

[3] Disponível em: < http://obiblio.sourceforge.net/index.php/Main/OpenBiblio>

códigos fonte por eles alterados no repositório BitBucket[4]. Um repositório de códigos tem por finalidade armazenar os códigos fontes dos projetos de *software* e gerenciar as alterações feitas pelos desenvolvedores, minimizando os riscos de perda de código por erros nas alterações enviadas.

O projeto foi desenvolvido na linguagem de programação PHP e com o banco de dados MySQL. Como o PHP permite desenvolver aplicações para a *internet*, é possível utilizá-lo tanto em ambiente *web* quanto em ambientes locais. Na utilização via *web*, o OpenBiblio é instalado em um provedor *web*, que armazenará todo o seu banco de dados e permitirá o acesso ao *site* em qualquer lugar do planeta. Na utilização local, o armazenamento e o acesso serão feitos somente dentro do espaço de utilização, não visível via internet.

As funcionalidades do *software* são divididas em quatro menus principais: Circulação, Catalogando, Administração e Relatórios. A Figura 10 mostra a tela inicial com a descrição dos quatro menus principais.

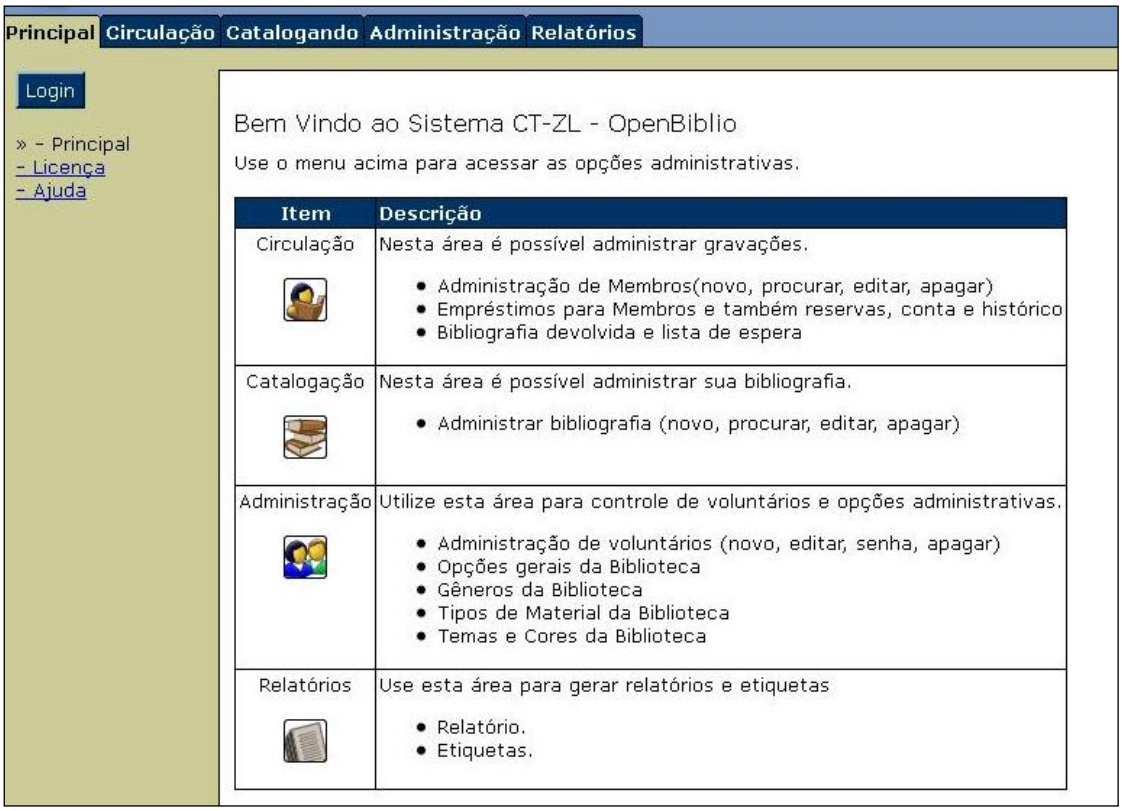

Figura 10: Tela inicial do OpenBiblio.

[4] Disponível em <https://bitbucket.org/mstetson/obiblio-10-wip/>

No menu Circulação, há três funcionalidades principais: Procurar Membros, Novo Membro e Devolução. A Figura 11 mostra a tela para procura de membros cadastrados na biblioteca. A Figura 12 mostra a tela para cadastro de novo membro. É interessante destacar que o *software* OpenBiblio permite a personalização de algumas funcionalidades, possibilitando uma maior adequação do sistema às necessidades da Biblioteca. Neste caso, observa-se que a Classe de membros é “aluno”, exemplificando a utilização em um ambiente escolar.

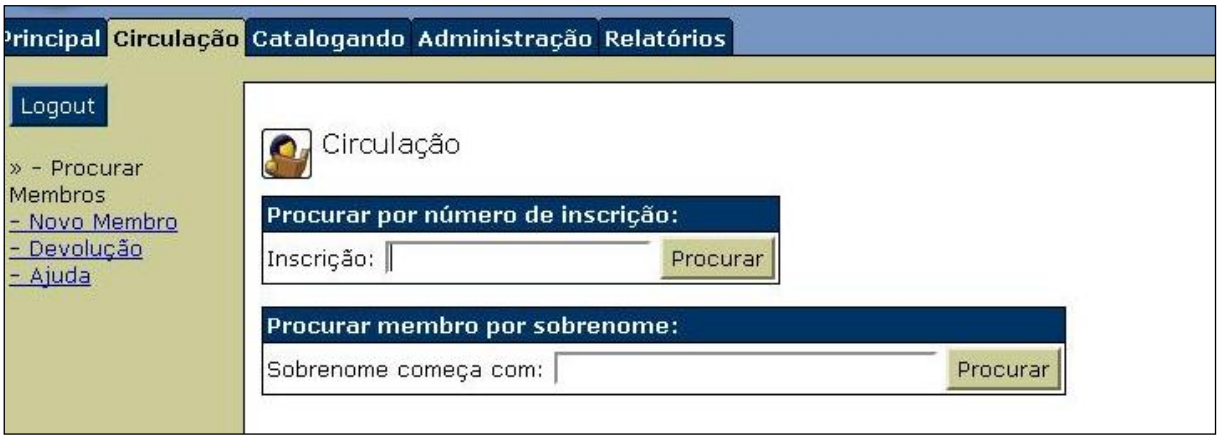

Figura 11: Menu Circulação - Procura de Membros.

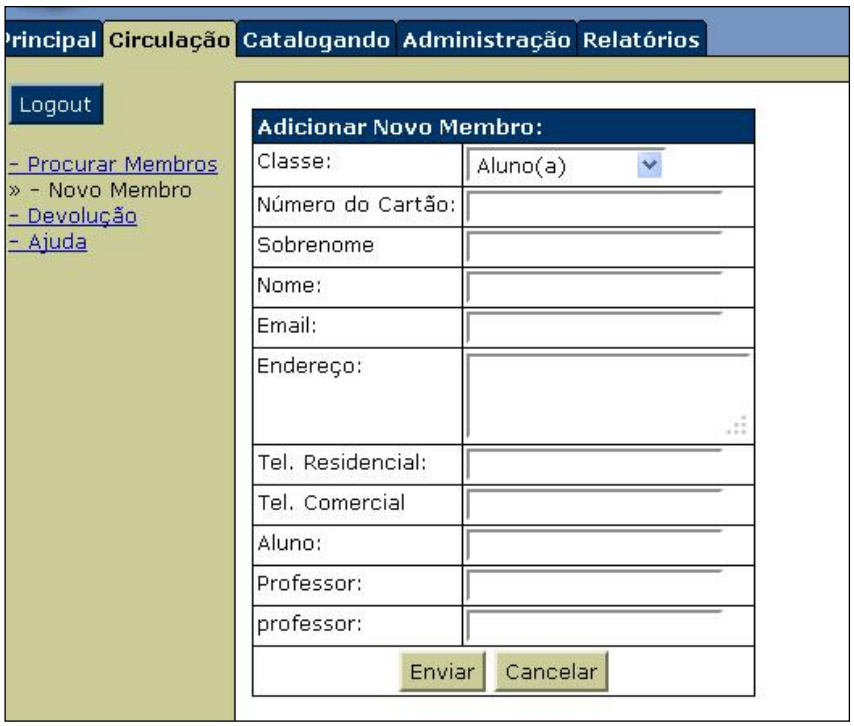

Figura 12: Menu Circulação - Cadastro de Membro.

No menu Catalogando são disponibilizadas as funcionalidades Procurar Bibliografias, Nova Bibliografia e Importar Dados do MARC. A Figura 13 mostra o

Cadastro de Bibliografias. O OpenBiblio permite o cadastro de “Tipo de Material”, tendo em vista que uma biblioteca também pode catalogar revistas, CDs, apostilas e outros. Outros dois itens que merecem destaque são o OPAC e o MARC.

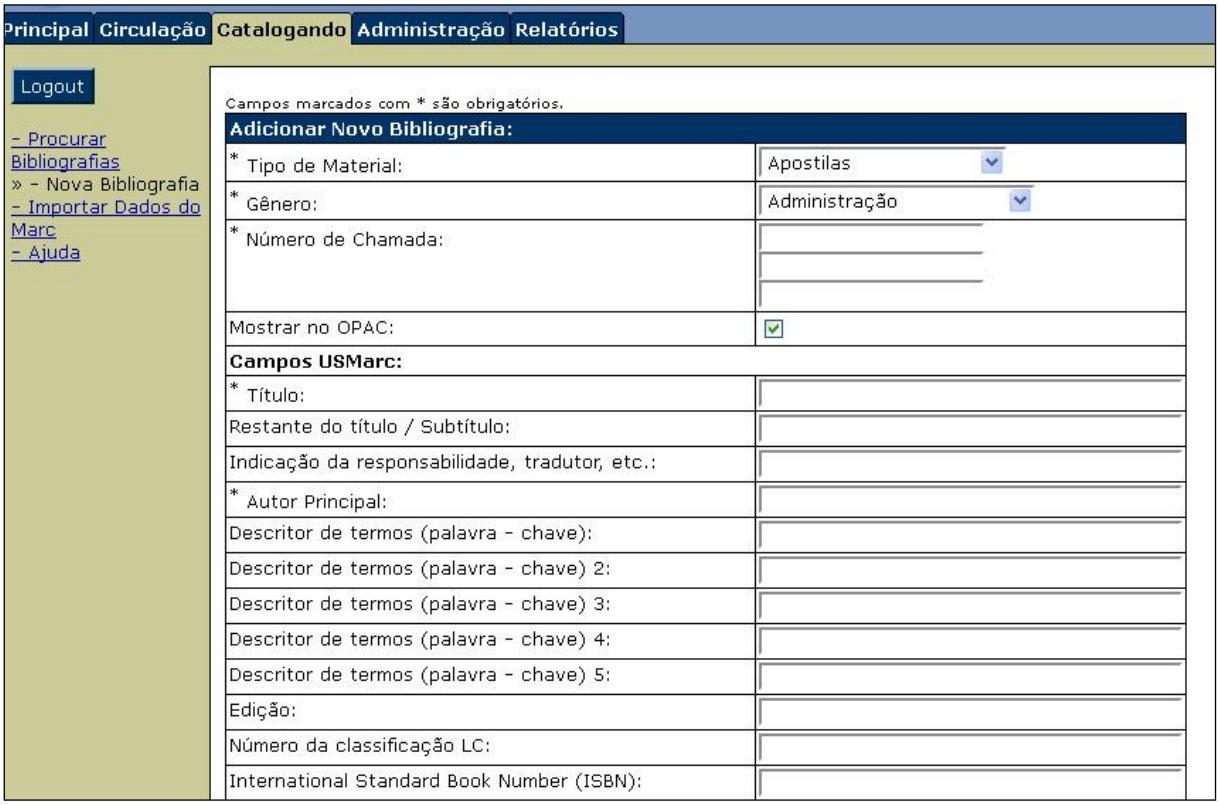

Figura 13: Menu Catalogando - Cadastro de Bibliografia.

Segundo Aracri, et al (nd), OPAC é uma sigla inglesa para *Online Public Access Catalog* (Catálogo Público de Acesso Online), que visa fazer a gerência, recuperação e manipulação de informações disponibilizadas em meio eletrônico, de forma mais rápida e acessível independente da distância. Dentro do OPAC existem alguns padrões e formatos, sendo um deles utilizado pelo OpenBiblio: o MARC. Este padrão é o *Machine Readable Cataloging*, responsável por intercambiar informações catalográficas entre diferentes sistemas. Observa-se, na Figura 13, que há uma opção específica no OpenBiblio para importação de dados via MARC.

Cabe ressaltar que a pesquisa por OPAC ocorre de forma separada da pesquisa ao sistema local. Isso é observado na Figura 14, onde foi feito um recorte que enfatiza os links exibidos na parte inferior central de todas as telas do sistema. A Figura 15 mostra a opção específica de pesquisa via OPAC.

Figura 14: Menu OPAC.

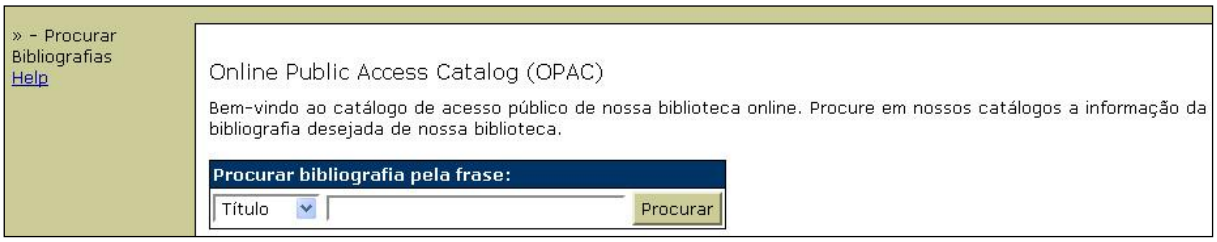

Figura 15: Pesquisa por OPAC.

No menu Administração, estão concentradas todas as funções administrativas da biblioteca, tais como gerenciamento de gêneros, membros e voluntários, personalização de campos, temas e dados da biblioteca. A Figura 16 mostra a funcionalidade “Tipos de Materiais” do menu Administração.

Figura 16: Menu Administração - Tipos de Materiais.

Por fim, o menu Relatórios, exibido na Figura 17, mostra a Lista de Relatórios disponibilizada pelo *software*. São gerados alguns relatórios estatísticos, tais como o Balanço de Devolução dos Membros e as Bibliografias Mais Populares. É possível também Buscar Exemplares, Bibliografias Devolvidas, Membros com Livros Atrasados e Reservas.

Figura 17: Menu Relatórios - Lista de Relatórios.

5.2 Determinação da Manutenibilidade

Para avaliar a manutenibilidade do *software* OpenBiblio, escolheu-se a métrica Complexidade Ciclomática, descrita no subitem 4.3.2. A escolha foi motivada levando em consideração os dois critérios apontados por Koscianski e Soares (2007, p.226) onde devem ser avaliados o custo e a complexidade da aplicação das métricas. Como a métrica complexidade ciclomática mostra-se a mais simples para análise, foi possível aplica-la neste trabalho, levando em conta o tempo para o desenvolvimento do mesmo.

A métrica Pontos de Função, descrita no subitem 4.3.1, é inviável por analisar minuciosamente todo o código fonte. Já a métrica de Acoplamento, apontada no subitem 4.3.3, requer um estudo aprofundado do *software* e suas dependências, exigindo um tempo para desenvolvimento não coberto para realização deste trabalho. Por fim, a métrica para cálculo da maturidade do *software,* apresentada no subitem 4.3.4, requer informações sobre as versões para que seja feito um comparativo, as quais não são disponibilizadas no *site* do OpenBiblio.

O estudo de caso será baseado na última versão estável, a 0.7.1. Após avaliação da estrutura do OpenBiblio, com a métrica Complexidade Ciclomática sendo aplicada em duas classes, localizadas dentro do pacote de classes do *software*: BiblioSearchQuery.php e StaffQuery.php.

Essas classes foram escolhidas de acordo com a quantidade de comandos que influenciariam o fluxo de execução do programa. O limite de duas

classes foi estipulado de acordo com o tempo disponível para realização da monografia.

5.2.1 Classe BiblioSearchQuery.php

A classe BiblioSerachQuery.php tem por finalidade prover acesso aos dados das bibliografias. Em sua estrutura geral, é descendente da classe Query.php e possui onze funções implementadas. O código fonte encontra-se no Anexo A.

Das onze funções, as seis primeiras apresentam implementações sem nenhum comando de decisão, sendo, portanto, desconsideradas para o cálculo da complexidade ciclomática. Para cada uma das outras sete funções, será feito o grafo de fluxo para ilustração dos caminhos de execução e ser calculada a sua complexidade.

A função **search**() possui vinte e três nós predicados, constituindo a função mais complexa desta classe. Na estrutura dessa função, observa-se uma enorme quantidade de comandos de decisão e repetição, além de comando encadeados. A complexidade ciclomática é de 24.

O grafo de fluxo da função **getCriteria()** é representado na Figura 18. Observa-se que há três nós predicados que afetam o fluxo de execução, sendo representados pelos números 2, 3 e 5. A complexidade ciclomática para esta função é quatro.

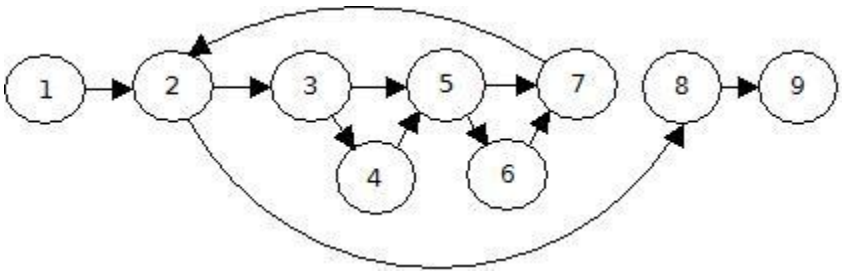

Figura 18: Grafo de Fluxo - Função getCriteria().

Na Figura 19 é apresentado o grafo de fluxo para a função **getLike**(), observa-se que há dois nós predicados, representados pelos números 2 e 5. Nesta função a complexidade ciclomática é três.

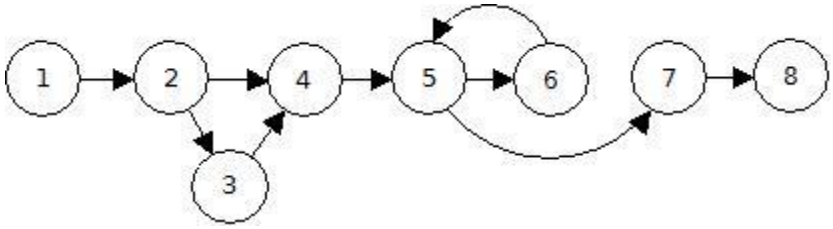

Figura 19: Grafo de Fluxo - Função getLike().

O grafo de fluxo para a função **doQuery**() está representado na Figura 20. Observa-se que os nós predicados correspondem aos números 3 e 5, sendo que a função apresenta complexidade ciclomática 3.

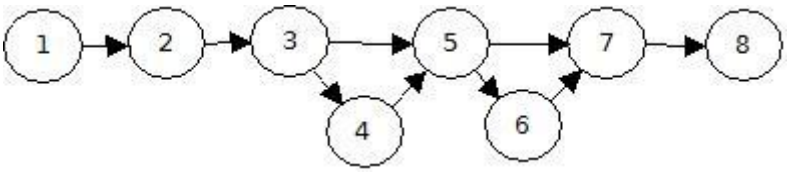

Figura 20: Grafo de Fluxo - Função doQuery().

A Figura 21 apresenta o grafo de fluxo para a função **fetchRow**(). Esta função é a segunda mais complexa da classe, tendo oito nós predicados o que gera uma complexidade ciclomática igual a nove. Os nós predicados são representados pelos números 2, 6, 8, 10, 12, 14, 16 e 18.

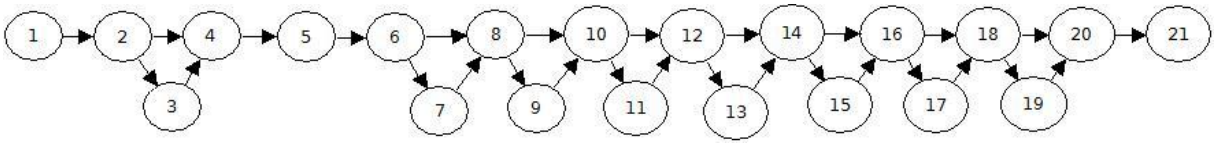

Figura 21: Grafo de Fluxo - Função fetchRow().

5.2.2 Classe StaffQuery.php

A classe StaffQuery.php tem como finalidade prover acesso aos dados por funcionários da biblioteca. Assim como a classe discutida no subitem 5.2.1, também é descendente da classe Query.php, localizada no mesmo pacote. O código fonte encontra-se no Anexo B.

Dentre suas nove funções, foram elencadas cinco para o cálculo da complexidade ciclomática. Para cada uma das funções, será elaborado o grafo de fluxo e apresentado o resultado do cálculo.

A função **execSelect**(), é representada pelo grafo de fluxo mostrado na Figura 22. Observa-se que há somente um nó predicado, representado pelo número 2, sendo a complexidade ciclomática desta função equivalente a dois.

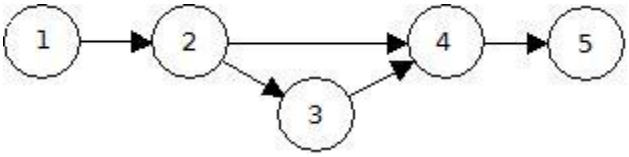

Figura 22: Grafo de Fluxo - Função execSelect().

O grafo apresentado na Figura 23 apresenta os sete nós predicados da função **fetchStaff**(). Os nós correspondem aos índices 2, 5, 8, 11, 14, 17 e 20. A complexidade ciclomática desta função equivale a oito.

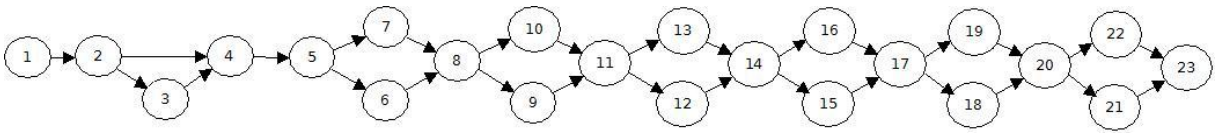

Figura 23: Grafo de Fluxo - Função fetchStaff().

Para a função **dupUserName**(), o grafo de fluxo é representado na Figura 24. Há dois nós predicados, correspondentes aos índices 2 e 5. Para esta função a complexidade ciclomática é igual a três.

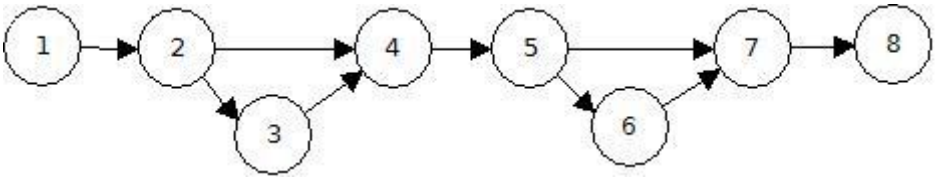

Figura 24: Grafo de Fluxo - Função dupUserName().

Na Figura 25 é apresentado o grafo de fluxo para a função **insert**(). É possível observar a presença de três nós predicados, representados nos nós 2,4 e 7. Para esta função, a complexidade ciclomática equivale a quatro.

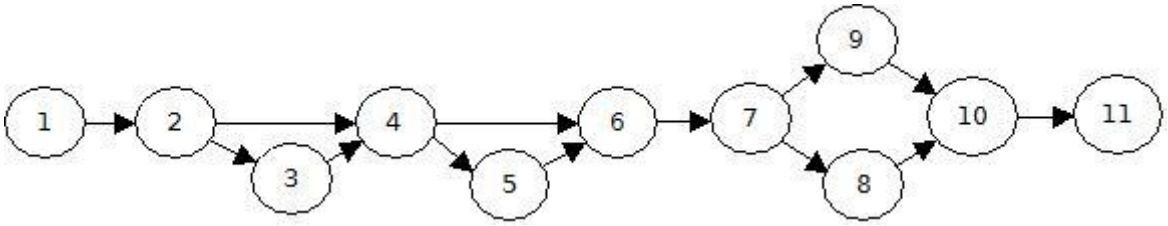

Figura 25: Grafo de Fluxo - Função insert().

A última função discutida é a função **update**(). O grafo de fluxo desta função está representado na Figura 26. Como a função *update* desempenha quase

em sua totalidade a mesma função que a *insert()*, o grafo de fluxo, os nós predicados e a complexidade ciclomática mantém-se a mesma, equivalendo ao valor quatro.

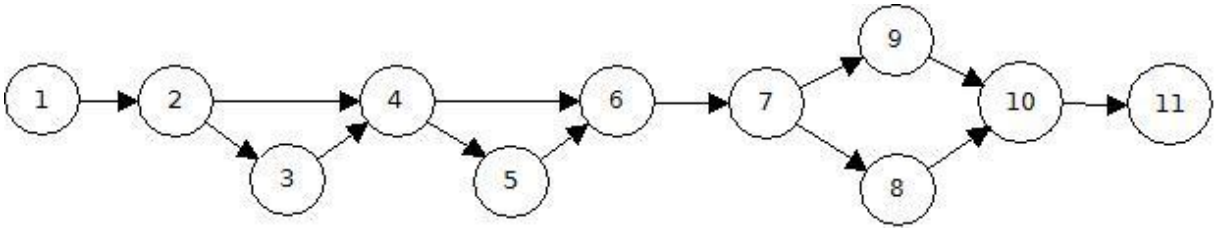

Figura 26: Grafo de Fluxo - Função update().

5.3 Discussão dos Resultados

A complexidade ciclomática permite avaliar o quão complexo é um determinado trecho do código. Nos trechos analisados, foi possível observar dois extremos como resultado: complexidade baixa e alta. Os resultados são apresentados na Tabela 1. Conforme exposto por Pressman (2011, p.538), a aplicação de métricas para a engenharia de *software* é qualitativa, portanto os valores definidos para comparação entre baixa ou alta foram estipulados pelo autor deste trabalho monográfico, tendo como base os valores obtidos no estudo de caso.

Tabela 1: **Complexidade Ciclomática das funções analisadas nas classes BiblioSearchQuery.php e StaffQuery.php**

Classe BiblioSearchQuery.php		**Classe StaffQuery.php**	
Função	**v(G)**	**Função**	**v(G)**
search()	24	fetchStaff()	8
fetchRow()	5	insert()	4
getCriteria()	4	update()	4
getLike()	3	dupUserName()	3
doQuery()	3	execSelect()	2

Fonte: Resultados obtidos através do estudo de caso apresentado no Capítulo 5, subseção 5.2.

Considerou-se como baixa a complexidade de funções que não ultrapassaram o valor cinco. As funções foram getCriteria() = 4, getLike() = 3 e doQuery() = 3, todas da classe BiblioSearchQuery.php. Para a classe

StaffQuery.php, as funções com complexidade baixa foram getSelect() = 2, dupUserName() = 3, insert() = 4 e update() = 4. A baixa complexidade ciclomática destas funções aumenta a facilidade de modificação, portanto, aumenta sua manutenibilidade.

Estipulou-se para alta complexidade o valor acima de cinco. Dessa forma, a classe BiblioSearchQuery.php possui as seguintes funções de complexidade ciclomática com valor alto: search() = 24 e fetchRow() = 9. A classe StaffQuery.php tem a função fetchStaff() = 8. Nestes casos a dificuldade na compreensão do código aumenta, assim como o impacto ao aplicar mudanças, reduzindo sua manutenibilidade.

Ao avaliar funções com complexidade ciclomática alta, percebe-se que as dificuldades para aplicar modificações aumentam, bem como o risco para os resultados pós-mudança. Uma função que chama atenção é a função search(), presente na classe BiblioSearchQuery.php. Sua complexidade ciclomática é maior que o dobro da complexidade para funções como a fetchRow(). Ao analisar seu código, observa-se uma grande quantidade de comandos de seleção encadeados (um comando dentro do outro) que criam uma árvore de dependência complexa. Essa função reduz a manutenibilidade do sistema e uma possível solução seria aplicar a técnica de refatoração, fazendo com que uma função menos complexa fosse desenvolvida e, caso necessário, novas funções fossem geradas.

A aplicação da métrica complexidade ciclomática provê informações importantes que geram uma ideia do grau de dificuldade em modificar um *software*, todavia, observa-se que existe uma necessidade de aplicação de outras métricas para conseguir obter um resultado confiável sobre a manutenibilidade do *software*. Ao estudar duas ou mais métricas seria possível fazer comparações entre os resultados e estabelecer um ponto de equilíbrio entre eles, aumentando a confiabilidade dos resultados obtidos.

Outro ponto a considerar é a questão qualitativa, que dificulta a obtenção de um resultado concreto quando o estudo é realizado por uma única pessoa. O ideal seria que duas ou mais pessoas fizessem o estudo de forma conjunta para determinação da manutenibilidade. Dessa forma, as pessoas poderiam discutir suas opiniões e chegar num consenso que trouxesse o resultado o mais próximo da realidade.

O fator tempo é outro limitador da determinação da manutenibilidade. A determinação exige um estudo aprofundado do *software* e sua documentação, o que requer um tempo considerável para execução. Neste estudo de caso, por exemplo, foi necessário limitar o estudo para apenas duas classes, pois seria impossível expandi-lo com o tempo disponível para desenvolver a monografia.

Dentre estas considerações, os objetivos relativos ao estudo de caso foram parcialmente obtidos, tendo em vista que não possível realizar a completa determinação da manutenibilidade do *software* OpenBiblio. Todavia, foi possível ter uma visão do processo como um todo, onde observou-se que há funções de alta manutenibilidade e outras com baixa ou podendo ser considerada nula, como a função search().

6 CONSIDERAÇÕES FINAIS

Este trabalho monográfico permitiu explorar uma pequena parte do campo da manutenção de *software*, que é tão presente nas organizações e tão pouco ressaltada. A compreensão das dificuldades enfrentadas para esta atividade proporcionou um entendimento sobre o tema manutenibilidade e a sua importância na qualidade de *software.*

A manutenibilidade é um estudo minucioso de vários aspectos do *software*, não somente envolvendo a codificação, mas também a documentação do mesmo. Embora o estudo seja trabalhoso, percebeu-se que é necessário fazê-lo para melhorar a qualidade do *software*, reduzindo os custos e esforços necessários para modificá-lo.

Percebeu-se, também, que a manutenibilidade pode ser trabalhada desde o desenvolvimento do sistema, e deve ser continuada nas fases de manutenção. Durante as fases iniciais do ciclo de vida, independentemente do processo escolhido, é preciso incorporar técnicas que aumentem a manutenibilidade do sistema, fazendo com que este seja entregue já com altos índices de manutenibilidade. Todavia, o controle deve ser continuado de modo que a aplicação, mesmo com todas as modificações que irá receber, continue sendo dotada de altos índices de manutenibilidade.

Como foi apresentado na discussão do estudo de caso, determinar a manutenibilidade do *software* requer uma equipe (duas ou mais pessoas), tempo e a escolha de, no mínimo duas métricas, para proporcionar um resultado condizente com a realidade do produto. É importante destacar que, embora as métricas envolvam fatores qualitativos, a determinação deve ser isenta de sentimentalismo pelo produto desenvolvido, visando trazer confiabilidade aos resultados da manutenibilidade.

Com relação à questão problema proposta pelo trabalho (como determinar a manutenibilidade de um *software* e por que os *softwares* devem ser manuteníveis?), foi encontrada uma resposta parcial. A primeira parte foi respondida de forma parcial, pois foi verificado que a determinação da manutenibilidade requer um grupo de pessoas e um conjunto de métricas específicas, selecionados de acordo com as características e necessidades do projeto, fazendo um estudo

completo em todas as áreas do *software*. Já a segunda parte do questionamento foi respondida, pois se observou que quanto mais fácil é modificação de um código, ou seja, quanto maior a sua manutenibilidade, maior a qualidade do *software* e a satisfação do cliente.

Quanto aos objetivos, estes também foram parcialmente atingidos, pois se observou que a determinação da manutenibilidade exige uma concentração de pessoas e esforços em torno do mesmo problema, reunindo todas as informações possíveis sobre o projeto e utilizando-as para verificar a manutenibilidade. Constatou-se, também, que essa atividade tem seu grau de complexidade elevado de acordo com o tamanho do *software* a avaliar.

No que concerne às hipóteses levantadas durante a fase de projeto da monografia, estas foram confirmadas. As duas hipóteses encontradas faziam referência aos benefícios da qualidade durante a fase de desenvolvimento, visando a redução de custos com manutenção e foram confirmadas com o trabalho acadêmico. Além da confirmação, o trabalho monográfico permitiu a complementação do conteúdo, com base nos pressupostos levantados.

Em relação ao estudo de caso, o objetivo inicial era fazer uma análise completa do *software*, utilizando-se todas as métricas descritas na subseção 4.3. Entretanto, ao contrastar a complexidade da aplicação das métricas com o tamanho do *software*, tornou-se inviável aplica-las, levando em consideração o tempo disponível para realização da monografia. Então foi decidido utilizar somente uma métrica aplicada a duas classes de um mesmo pacote, fazendo que os resultados permitissem um comparativo entre elas e provesse uma ideia de como funciona a determinação da manutenibilidade.

Outro contratempo encontrado foi relativo ao *software* escolhido. Embora ele fosse livre, há uma insuficiência de documentação e ausência de alguns elementos necessários para diagnosticar a manutenibilidade do sistema. A falta de alguns dados exigidos para a aplicação das métricas reduziu de quatro métricas possíveis de aplicação, apresentadas na subseção 4.3, para duas possíveis métricas para serem aplicadas: pontos de função e complexidade ciclomática. Sendo assim, foi escolhida a complexidade ciclomática, por ser possível de aplicação do tempo disponível para realização do trabalho.

Ao realizar este trabalho monográfico, foi possível compreender a necessidade de utilização de boas práticas de programação e documentação, de

modo a organizar os processos e criar produtos de *software* de alta qualidade e manutenibilidade, aumentando seu tempo de vida no mercado.

Por fim, novos questionamentos foram levantados, sendo eles: Como inserir a cultura da manutenibilidade de *software* no ambiente organizacional? Existem ferramentas que podem auxiliar na determinação de manutenibilidade? Até que ponto é benéfico investir esforços (custo e tempo) para determinar a manutenibilidade de um *software*? Estes questionamentos poderiam gerar novos trabalhos e aprofundamentos de pesquisas.

7 REFERÊNCIAS

ARACRI, Carolina. RODRIGUES, Marília. SILVA, Renata. **OPAC**. Marília: Universidade Estadual de São Paulo, nd. Disponível em: < http://www.slideshare.net/Re _Biblio/slides-opac>. Acesso em: 20 Mai. 2013.

BELLIN, David. **Manutenção de *software***: guia para administração de pequenos sistemas. Traduzido por José Renato Adorni Martins. São Paulo: Makron Books, 1993.

BITBUCKET. **Obiblio-1.0-wip**. Disponível em: <https://bitbucket.org/mstetson/obiblio-10-wip/>. Acesso em: 20. Abr 2013.

BRAUDE, Eric. **Projeto de *software*: da programação à arquitetura**: uma abordagem baseada em java. Traduzido por Edson Furmankiewicz. Porto Alegre: Bookman, 2005.

IEEE: INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. ***About* IEEE**. Disponível em <http://www.ieee.org/about/index.html>. Acesso em 23. Abr 2013.

_____. **IEEE Standard for *software* maintenance**.1998. Disponível em <http://homes.ieu. edu.tr/~kkurtel/Documents/IEEE%20Std%201219-1998%20*Software*%20Maintenan ce.pdf>. Acesso em 04. Abr 2012.

_____. ***Software* engineering body of knowledge**. Alpha Version. 2012. Disponível em: <https://computer.centraldesktop.com/home/viewfile?guid=217167737FFE0762C69847C48FDFD48EDE6FF4003&id=16981975>. Acesso em: 04 Abr. 2012.

KOSCIANSKI, André. SOARES, Michel dos S. **Qualidade de *software***: aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software. 2ªed. São Paulo: Novatec, 2007.

OPENBIBLIO. **OpenBiblio**. Disponível em <http://obiblio.sourceforge.net/index.php/Main/OpenBiblio>. Acesso em: 25. Abr 2013.

PRESSMAN, Roger S. **Engenharia de *software***: uma abordagem profissional. Traduzido por Ariovaldo Griesi, Mario Mouro Fecchio.7ªed. Porto Alegre-RS: AMGH, 2011.

SILVA, Nelson Peres. **Análise e estruturas de sistemas de informação.** São Paulo: Érica, 2007.

SOMMERVILLE, Ian. **Engenharia de *software***. Traduzido por Selma Shin Shimizu Melnikoff, Reginaldo Arakaki, Edilson de Andrade Barbosa. 8ªed. São Paulo: Pearson Addison-Wesley, 2007.

TAURION, Cesar. ***Software* livre**: potencialidades e modelos de negócio. Rio de Janeiro: Brasport, 2004.

ANEXO A – Classe BiblioSearchQuery.php

```
<?php
/* This file is part of a copyrighted work; it is distributed with NO WARRANTY.
 * See the file COPYRIGHT.html for more details.
 */

require_once("../shared/global_constants.php");
require_once("../classes/Query.php");
require_once("../classes/BiblioSearch.php");
require_once("../classes/BiblioField.php");
require_once("../classes/Localize.php");

/*****************************************************************************
 * BiblioQuery data access component for library bibliographies
 *
 * @author David Stevens <dave@stevens.name>;
 * @version 1.0
 * @access public
 ***************************************************************************** */
class BiblioSearchQuery extends Query {
  var $_itemsPerPage = 1;
  var $_rowNmbr = 0;
  var $_currentRowNmbr = 0;
  var $_currentPageNmbr = 0;
  var $_rowCount = 0;
  var $_pageCount = 0;
  var $_loc;

  function BiblioSearchQuery() {
    $this->Query();
    $this->_loc = new Localize(OBIB_LOCALE,"classes");
  }
  function setItemsPerPage($value) {
    $this->_itemsPerPage = $value;
  }
  function getLineNmbr() {
    return $this->_rowNmbr;
  }
  function getCurrentRowNmbr() {
    return $this->_currentRowNmbr;
  }
  function getRowCount() {
    return $this->_rowCount;
  }
  function getPageCount() {
    return $this->_pageCount;
  }

  /*****************************************************************************
   * Executes a query
   * @param string $type one of the global constants
   *              OBIB_SEARCH_BARCODE,
   *              OBIB_SEARCH_TITLE,
   *              OBIB_SEARCH_AUTHOR,
   *              or OBIB_SEARCH_SUBJECT
   * @param string @$words pointer to an array containing words to search for
   * @param integer $page What page should be returned if results are more than one page
   * @param string $sortBy column name to sort by.  Can be title or author
   * @return boolean returns false, if error occurs
   * @access public
   *****************************************************************************   */
  function search($type, &$words, $page, $sortBy, $opacFlg=true) {
    # reset stats
    $this->_rowNmbr = 0;
    $this->_currentRowNmbr = 0;
    $this->_currentPageNmbr = $page;
    $this->_rowCount = 0;
    $this->_pageCount = 0;

    # setting sql join clause
    $join = "from biblio left join biblio_copy on biblio.bibid=biblio_copy.bibid ";
```

```
    # setting sql where clause
    $criteria = "";
    $joins = "";
    $short = "";
    $words = array_unique(unserialize(strtolower(serialize($words))));
    if ((sizeof($words) == 0) || ($words[0] == "")) {
      if ($opacFlg) $criteria = "where opac_flg = 'Y' ";
    } else {
      if ($type == OBIB_SEARCH_BARCODE) {
        $criteria = $this->_getCriteria(array("biblio_copy.barcode_nmbr"),$words);
      } elseif ($type == OBIB_SEARCH_AUTHOR) {
        $drop=1;
        for ($i = 0; $i < count($words); $i++) {
          if (strlen($words[$i]) <= $drop) continue;
          $joins = $joins + 1;
          $join .= "left join biblio_field as bf".$i." on bf".$i.".bibid=biblio.bibid ";
          $join .= "and bf".$i.".tag in ('110', '700', '710') ";
          $join .= "and bf".$i.".field_data ";
          $join .= $this->mkSQL("like %Q ", "%".$words[$i]."%");
          # word boundaries for short words: prevent excessive wildcard matching in WHERE
          if (strlen($words[$i]) < $drop + 3) {
            $join .= "and bf".$i.".field_data ";
            $join .= $this->mkSQL("rlike %Q ", "[[:<:]]".$words[$i]);
          }
          $join .= "and not bf".$i.".subfield_cd regexp('[0-9]') ";
        }
        $criteria = $this->_getCriteria(array("biblio.author","biblio.responsibility_stmt"),$words,$bField=true,$drop);
      } elseif ($type == OBIB_SEARCH_SUBJECT) {
        $drop=1;
        for ($i = 0; $i < count($words); $i++) {
          if (strlen($words[$i]) <= $drop) continue;
          if (strlen($words[$i]) <= $drop + 1) $short = $short + 1;
          $joins = $joins + 1;
          $join .= "left join biblio_field as bf".$i." on bf".$i.".bibid=biblio.bibid ";
          # Tags equal to Locum connector class for III - http://thesocialopac.net/
          $join .= "and bf".$i.".tag in (
            '600', '610', '611', '630', '650', '651',
            '653', '654', '655', '656', '657', '658',
            '690', '691', '692', '693', '694', '695',
            '696', '697', '698', '699'
          ) ";
          $join .= "and bf".$i.".field_data ";
          $join .= $this->mkSQL("like %Q ", "%".$words[$i]."%");
          if (strlen($words[$i]) < $drop + 3) {
            $join .= "and bf".$i.".field_data ";
            $join .= $this->mkSQL("rlike %Q ", "[[:<:]]".$words[$i]);
          }
          $join .= "and not bf".$i.".subfield_cd regexp('[0-9]') ";
        }
        $criteria = $this-
>_getCriteria(array("biblio.topic1","biblio.topic2","biblio.topic3","biblio.topic4","biblio.topic5"),$words,$bField=true,$drop);
      } elseif ($type == OBIB_SEARCH_CALLNO) {
        $criteria = $this->_getCriteria(array("biblio.call_nmbr1","biblio.call_nmbr2","biblio.call_nmbr3"),$words);
      } elseif ($type == OBIB_SEARCH_KEYWORD) {
        $drop=1;
        for ($i = 0; $i < count($words); $i++) {
          if (strlen($words[$i]) <= $drop) continue;
          if (strlen($words[$i]) <= $drop + 1) $short = $short + 1;
          $joins = $joins + 1;
          $join .= "left join biblio_field as bf".$i." on bf".$i.".bibid=biblio.bibid ";
          $join .= "and bf".$i.".tag in (";
          if (strlen($words[$i]) > 8) $join.= " '010', '020', '022', '024',";
          $join .= "
            '110', '130', '245', '250', '260',
            '300', '336', '337', '338', '340',
            '380', '381', '382', '384', '383', '384',
            '400', '410', '440', '490',
            '500', '501', '502', '505', '511', '520',
            '521', '526',
            '600', '610', '611', '630', '650', '651',
            '653', '654', '655', '656', '657', '658',
            '690', '691', '692', '693', '694', '695',
            '696', '697', '698', '699',
            '700', '710', '730',
            '800', '810', '830', '856'
          ) ";
```

```
        $join .= "and bf".$i.".field_data ";
        $join .= $this->mkSQL("like %Q ", "%".$words[$i]."%");
        if (strlen($words[$i]) < $drop + 3) {
          $join .= "and bf".$i.".field_data ";
          $join .= $this->mkSQL("rlike %Q ", "[[:<:]]".$words[$i]);
        }
        $join .= "and not (bf".$i.".tag = '260' and bf".$i.".subfield_cd in ('a', 'b', 'e', 'f', 'g')) ";
        if ($opacFlg) $join .= "and not (bf".$i.".tag in ('526', '856') and bf".$i.".subfield_cd = 'x') ";
        $join .= "and not bf".$i.".subfield_cd regexp('[0-9]') ";
      }
      $criteria = $this-
>_getCriteria(array("biblio.author","biblio.responsibility_stmt","biblio.title","biblio.title_remainder","biblio.topic1","biblio.topic2","bib
lio.topic3","biblio.topic4","biblio.topic5"),$words,$bField=true,$drop);
    } else {
      $criteria = $this->_getCriteria(array("biblio.title","biblio.title_remainder"),$words);
    }
    if ($opacFlg) $criteria = $criteria."and opac_flg = 'Y' ";
  }

  # limit number of joins and short words
  if ($joins > 29 or $short > 3) {
    $msg = "Enclose adjacent \"words to be found\" with quotation marks.";
    if ($opacFlg) header("Location: ../opac/index.php?msg=".U($msg));
    else header("Location: ../catalog/index.php?msg=".U($msg));
    exit();
  }

  # setting query that will return all the data
  # sql_calc_found_rows is efficient for counting rows on unefficient queries...
  $sql = "select sql_calc_found_rows ";
  if ($bField) $sql .= "distinct ";
  $sql .= "biblio.* ";
  $sql .= ",biblio_copy.copyid ";
  $sql .= ",biblio_copy.barcode_nmbr ";
  $sql .= ",biblio_copy.status_cd ";
  $sql .= ",biblio_copy.due_back_dt ";
  $sql .= ",biblio_copy.mbrid ";
  $sql .= $join;
  $sql .= $criteria;
  $sql .= $this->mkSQL(" order by %C ", $sortBy);

  # setting limit so we can page through the results
  $offset = ($page - 1) * $this->_itemsPerPage;
  $limit = $this->_itemsPerPage;
  $sql .= $this->mkSQL(" limit %N, %N", $offset, $limit);

  //exit("sql=[".$sql."]<br>\n");

  # Running search sql statement
  if (!$this->_query($sql, $this->_loc->getText("biblioSearchQueryErr2"))) {
    return false;
  }

  # Calculate stats based on row count
  $this->_rowCount = implode(mysql_fetch_row(mysql_query('select found_rows();')));
  $this->_pageCount = ceil($this->_rowCount / $this->_itemsPerPage);
  return true;

}


/*************************************************************************
 * Utility function to get the selection criteria for a given column and set of values
 * @param string $col bibid of bibliography to select
 * @param array reference &$words array of words to search for
 * @return string returns SQL criteria syntax for the given column and set of values
 * @access private
 *************************************************************************   */
function _getCriteria($cols,&$words,$bField=false,$drop="") {
  # setting selection criteria sql
  $prefix = "where ";
  $criteria = "";
  for ($i = 0; $i < count($words); $i++) {
    # Drop very short words when querying biblio_field
    if ($bField and strlen($words[$i]) > $drop) array_push($cols, "bf".$i.".field_data");
    $criteria .= $prefix.$this->_getLike($cols,$words[$i]);
```

```
    $prefix = " and ";
    if ($bField and strlen($words[$i]) > $drop) array_pop($cols);
  }
  return $criteria;
}

function _getLike(&$cols,$word) {
  $prefix = "";
  $suffix = "";
  if (count($cols) > 1) {
    $prefix = "(";
    $suffix = ")";
  }
  $like = "";
  for ($i = 0; $i < count($cols); $i++) {
    $like .= $prefix;
    $like .= $this->mkSQL("%C like %Q ", $cols[$i], "%".$word."%");
    $prefix = " or ";
  }
  $like .= $suffix;
  return $like;
}

/*********************************************************************************
 * Executes a query to select ONLY ONE SUBFIELD
 * @param string $bibid bibid of bibliography copy to select
 * @param string $fieldid copyid of bibliography copy to select
 * @return BiblioField returns subfield or false, if error occurs
 * @access public
 ********************************************************************************   */
function doQuery($statusCd,$mbrid="") {

  $sql = "select biblio.* ";
  $sql .= ",biblio_copy.copyid ";
  $sql .= ",biblio_copy.barcode_nmbr ";
  $sql .= ",biblio_copy.status_cd ";
  $sql .= ",biblio_copy.status_begin_dt ";
  $sql .= ",biblio_copy.due_back_dt ";
  $sql .= ",biblio_copy.mbrid ";
  $sql .= ",biblio_copy.renewal_count ";
  $sql .= ",greatest(0,to_days(sysdate()) - to_days(biblio_copy.due_back_dt)) days_late ";
  $sql .= "from biblio, biblio_copy ";
  $sql .= "where biblio.bibid = biblio_copy.bibid ";
  if ($mbrid != "") {
     $sql .= $this->mkSQL("and biblio_copy.mbrid = %N ", $mbrid);
  }
  $sql .= $this->mkSQL(" and biblio_copy.status_cd=%Q ", $statusCd);
  $sql .= " order by biblio_copy.status_begin_dt desc";

  if (!$this->_query($sql, $this->_loc->getText("biblioSearchQueryErr3"))) {
    return false;
  }
  $this->_rowCount = $this->_conn->numRows();
  return true;
}

/*********************************************************************************
 * Fetches a row from the query result and populates the BiblioSearch object.
 * @return BiblioSearch returns bibliography search record or false if no more bibliographies to fetch
 * @access public
 ********************************************************************************   */
function fetchRow() {
  $array = $this->_conn->fetchRow();
  if ($array == false) {
    return false;
  }

  # increment rowNmbr
  $this->_rowNmbr = $this->_rowNmbr + 1;
  $this->_currentRowNmbr = $this->_rowNmbr + (($this->_currentPageNmbr - 1) * $this->_itemsPerPage);

  $bib = new BiblioSearch();
  $bib->setBibid($array["bibid"]);
  $bib->setCopyid($array["copyid"]);
  $bib->setCreateDt($array["create_dt"]);
  $bib->setLastChangeDt($array["last_change_dt"]);
```

```
    $bib->setLastChangeUserid($array["last_change_userid"]);
    $bib->setMaterialCd($array["material_cd"]);
    $bib->setCollectionCd($array["collection_cd"]);
    $bib->setCallNmbr1($array["call_nmbr1"]);
    $bib->setCallNmbr2($array["call_nmbr2"]);
    $bib->setCallNmbr3($array["call_nmbr3"]);
    $bib->setTitle($array["title"]);
    $bib->setTitleRemainder($array["title_remainder"]);
    $bib->setResponsibilityStmt($array["responsibility_stmt"]);
    $bib->setAuthor($array["author"]);
    $bib->setTopic1($array["topic1"]);
    $bib->setTopic2($array["topic2"]);
    $bib->setTopic3($array["topic3"]);
    $bib->setTopic4($array["topic4"]);
    $bib->setTopic5($array["topic5"]);
    if (isset($array["barcode_nmbr"])) {
      $bib->setBarcodeNmbr($array["barcode_nmbr"]);
    }
    if (isset($array["status_cd"])) {
      $bib->setStatusCd($array["status_cd"]);
    }
    if (isset($array["status_begin_dt"])) {
      $bib->setStatusBeginDt($array["status_begin_dt"]);
    }
    if (isset($array["status_mbrid"])) {
      $bib->setStatusMbrid($array["status_mbrid"]);
    }
    if (isset($array["due_back_dt"])) {
      $bib->setDueBackDt($array["due_back_dt"]);
    }
    if (isset($array["days_late"])) {
      $bib->setDaysLate($array["days_late"]);
    }
    if (isset($array["renewal_count"])) {
      $bib->setRenewalCount($array["renewal_count"]);
    }

    return $bib;
  }


}

?>
```

ANEXO B – Classe StaffQuery.php

```
<?php
/* This file is part of a copyrighted work; it is distributed with NO WARRANTY.
 * See the file COPYRIGHT.html for more details. */

require_once("../shared/global_constants.php");
require_once("../classes/Query.php");

/**************************************************************************
 * StaffQuery data access component for library staff members
 *
 * @author David Stevens <dave@stevens.name>;
 * @version 1.0
 * @access public
 ************************************************************************** */
class StaffQuery extends Query {
  /**************************************************************************
   * Executes a query
   * @param string $userid (optional) userid of staff member to select
   * @return boolean returns false, if error occurs
   * @access public
   ************************************************************************   */
  function execSelect($userid="") {
    $sql = "select * from staff";
    if ($userid != "") {
      $sql .= $this->mkSQL(" where userid=%N ", $userid);
    }
    $sql .= " order by last_name, first_name";
    return $this->_query($sql, "Error accessing staff member information.");
  }
  /**************************************************************************
   * Executes a query to verify a signon username and password
   * @param string $username username of staff member to select
   * @param string $pwd password of staff member to select
   * @return boolean returns false, if error occurs
   * @access public
   ************************************************************************   */
  function verifySignon($username, $pwd) {
    $sql = $this->mkSQL("select * from staff "
                        . "where username = lower(%Q) "
                        . " and pwd = md5(lower(%Q)) ",
                        $username, $pwd);
    return $this->_query($sql, "Error verifying username and password.");
  }

  /**************************************************************************
   * Updates a staff member and sets the suspended flag to yes.
   * @param string $username username of staff member to suspend
   * @return boolean returns false, if error occurs
   * @access public
   ************************************************************************   */
  function suspendStaff($username)
  {
    $sql = $this->mkSQL("update staff set suspended_flg='Y' "
                        . "where username = lower(%Q)", $username);
    return $this->_query($sql, "Error suspending staff member.");
  }

  /**************************************************************************
   * Fetches a row from the query result and populates the Staff object.
   * @return Staff returns staff member or false if no more staff members to fetch
   * @access public
   ************************************************************************   */
  function fetchStaff() {
    $array = $this->_conn->fetchRow();
    if ($array == false) {
      return false;
    }
    $staff = new Staff();
    $staff->setUserid($array["userid"]);
    $staff->setLastName($array["last_name"]);
    $staff->setFirstName($array["first_name"]);
```

```
  $staff->setUsername($array["username"]);
  if ($array["circ_flg"] == "Y") {
    $staff->setCircAuth(true);
  } else {
    $staff->setCircAuth(false);
  }
  if ($array["circ_mbr_flg"] == "Y") {
    $staff->setCircMbrAuth(TRUE);
  } else {
    $staff->setCircMbrAuth(FALSE);
  }
  if ($array["catalog_flg"] == "Y") {
    $staff->setCatalogAuth(true);
  } else {
    $staff->setCatalogAuth(false);
  }
  if ($array["admin_flg"] == "Y") {
    $staff->setAdminAuth(true);
  } else {
    $staff->setAdminAuth(false);
  }
  if ($array["reports_flg"] == "Y") {
    $staff->setReportsAuth(TRUE);
  } else {
    $staff->setReportsAuth(FALSE);
  }
  if ($array["suspended_flg"] == "Y") {
    $staff->setSuspended(true);
  } else {
    $staff->setSuspended(false);
  }
  return $staff;
}

/*****************************************************************************
 * Returns true if username already exists
 * @param string $username staff member username
 * @param string $userid staff member userid
 * @return boolean returns true if username already exists
 * @access private
 *****************************************************************************
 */
function _dupUserName($username, $userid=0) {
  $sql = $this->mkSQL("select count(*) from staff where username = %Q "
                      . " and userid <> %N", $username, $userid);
  if (!$this->_query($sql, "Error checking for dup username.")) {
    return false;
  }
  $array = $this->_conn->fetchRow(OBIB_NUM);
  if ($array[0] > 0) {
    return true;
  }
  return false;
}
/*****************************************************************************
 * Inserts a new staff member into the staff table.
 * @param Staff $staff staff member to insert
 * @return boolean returns false, if error occurs
 * @access public
 *****************************************************************************   */
function insert($staff) {
  $dupUsername = $this->_dupUserName($staff->getUsername());
  if ($this->errorOccurred()) return false;
  if ($dupUsername) {
    $this->_errorOccurred = true;
    $this->_error = "Username is already in use.";
    return false;
  }
  $sql = $this->mkSQL("insert into staff values (null, sysdate(), sysdate(), "
                      . "%N, %Q, md5(lower(%Q)), %Q, ",
                      $staff->getLastChangeUserid(), $staff->getUsername(),
                      $staff->getPwd(), $staff->getLastName());
  if ($staff->getFirstName() == "") {
    $sql .= "null, ";
  } else {
    $sql .= $this->mkSQL("%Q, ", $staff->getFirstName());
```

```
    }
    $sql .= $this->mkSQL("'N', %Q, %Q, %Q, %Q, %Q) ",
                         $staff->hasAdminAuth() ? "Y" : "N",
                         $staff->hasCircAuth() ? "Y" : "N",
                         $staff->hasCircMbrAuth() ? "Y" : "N",
                         $staff->hasCatalogAuth() ? "Y" : "N",
                         $staff->hasReportsAuth() ? "Y" : "N");
    return $this->_query($sql, "Error inserting new staff member information.");
  }
  /**************************************************************************
   * Update a staff member in the staff table.
   * @param Staff $staff staff member to update
   * @return boolean returns false, if error occurs
   * @access public
   **************************************************************************   */
  function update($staff) {
    /*************************************************************
     * If changing username check to see if it already exists.
     *************************************************************/
    $dupUsername = $this->_dupUserName($staff->getUsername(), $staff->getUserid());
    if ($this->errorOccurred()) return false;
    if ($dupUsername) {
      $this->_errorOccurred = true;
      $this->_error = "Username is already in use.";
      return false;
    }

    $sql = $this->mkSQL("update staff set last_change_dt = sysdate(), "
                        . "last_change_userid=%N, username=%Q, last_name=%Q, ",
                        $staff->getLastChangeUserid(), $staff->getUsername(),
                        $staff->getLastName());
    if ($staff->getFirstName() == "") {
      $sql .= "first_name=null, ";
    } else {
      $sql .= $this->mkSQL("first_name=%Q, ", $staff->getFirstName());
    }
    $sql .= $this->mkSQL("suspended_flg=%Q, admin_flg=%Q, circ_flg=%Q, "
                         . "circ_mbr_flg=%Q, catalog_flg=%Q, reports_flg=%Q "
                         . "where userid=%N ",
                         $staff->isSuspended() ? "Y" : "N",
                         $staff->hasAdminAuth() ? "Y" : "N",
                         $staff->hasCircAuth() ? "Y" : "N",
                         $staff->hasCircMbrAuth() ? "Y" : "N",
                         $staff->hasCatalogAuth() ? "Y" : "N",
                         $staff->hasReportsAuth() ? "Y" : "N",
                         $staff->getUserid());
    return $this->_query($sql, "Error updating staff member information.");
  }

  /**************************************************************************
   * Resets a staff member password in the staff table.
   * @param Staff $staff staff member to update
   * @return boolean returns false, if error occurs
   * @access public
   **************************************************************************   */
  function resetPwd($staff) {
    $sql = $this->mkSQL("update staff set pwd=md5(lower(%Q)) "
                        . "where userid=%N ",
                        $staff->getPwd(), $staff->getUserid());
    return $this->_query($sql, "Error resetting password.");
  }
  /**************************************************************************
   * Deletes a staff member from the staff table.
   * @param string $userid userid of staff member to delete
   * @return boolean returns false, if error occurs
   * @access public
   **************************************************************************   */
  function delete($userid) {
    $sql = $this->mkSQL("delete from staff where userid = %N ", $userid);
    return $this->_query($sql, "Error deleting staff information.");
  }

}

?>
```