Heliomar Kann da Rocha Santos

RUMO AO REJUVENESCIMENTO AUTOMÁTICO DE SOFTWARE GUIADO POR ATRIBUTOS DE QUALIDADE

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

Orientador: Prof. Dr. Leonardo Gresta Paulino Murta

Co-orientador: Prof. Dr. Viviane Torres da Silva

Niterói

2011

Heliomar Kann da Rocha Santos

RUMO AO REJUVENESCIMENTO AUTOMÁTICO DE SOFTWARE GUIADO POR ATRIBUTOS DE QUALIDADE

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

Aprovada em Agosto de 2011.

BANCA EXAMINADORA

Prof. Dr. Leonardo Gresta Paulino Murta – Orientador
UFF

Prof. Dr. Viviane Torres da Silva – Co-Orientador
UFF

Prof. Dr. Anselmo Antunes Montenegro
UFF

Prof. Dr. Cláudia Maria Lima Werner
COPPE/UFRJ

Niterói

2011

Ao meu Orientador Leonardo Gresta Paulino Murta.

AGRADECIMENTOS

A Deus, pois sem ele nada seria possível.

Ao meu Orientador Leonardo Murta por acreditar, apoiar, motivar e viabilizar que eu concretizasse um bom mestrado.

À minha Co-orientadora, Viviane Torres da Silva, pela ajuda em momentos importantes, mesmo com licença maternidade.

Aos meus amigos Gleiph Ghioto e Daniel Castellani por ajudarem no desenvolvimento e evolução do Oceano.

Ao João Felipe, aluno de iniciação científica que ajudou a desenvolver trabalhos árduos neste tema.

Ao meu amigo e futuro sócio Roberto Menezes por apoio moral. Ele foi peça fundamental para eu ingressar no mestrado.

À minha amiga Paula Ceccon, pelo apoio motivacional, que consequentemente me ajudou a manter o foco no mestrado, diante de conturbações nesses últimos seis meses.

A todos os docentes que tive o prazer de conhecer no Instituto de Computação da UFF, por terem contribuído com minha formação tanto profissionalmente quanto pessoalmente.

À minha família por acreditar que eu seria capaz, em especial minha mãe, Sônia Lieselotte e ao meu pai, Geraldo Monteiro, embora distantes, sempre me apoiaram com orgulho.

Aos meus amigos e amigas pela felicidade e companheirismo em todos os momentos.

À CAPES pelo apoio financeiro.

EPÍGRAFE

“Seja você a mudança que quer ver no mundo” M. Ghandi

RESUMO

Normalmente, as equipes de desenvolvimento dedicam uma enorme quantidade de tempo e esforço na manutenção de funcionalidades de software pré-existentes. No entanto, como muitas dessas tarefas de manutenção não são planejadas, o software tende a degradar ao longo do tempo, causando débitos técnicos, principalmente em seus requisitos não-funcionais, tais como a reutilização, compreensibilidade e manutenibilidade. Assim, este trabalho propõe o monitoramento contínuo de alguns atributos de qualidade de software através de métricas e execução automática de manutenção perfectiva (i.e., refatorações) para promover o seu rejuvenescimento em resposta a anti-padrões ou "maus cheiros" (do inglês, *bad smells*) previamente detectados. A abordagem criada foi avaliada através de um estudo experimental, que encontrou evidências de que o rejuvenescimento automático de software é possível. Também foram encontrados indícios de padrões entre refatorações e atributos de qualidade, ou seja, quais das refatorações utilizadas melhoram, pioram ou não alteram os atributos de qualidade.

Palavras-chave: Rejuvenescimento de Código, Refatoração, Atributos de Qualidade, Métricas, Sistema Multiagentes.

ABSTRACT

Usually, development teams devote a huge amount of time and effort on maintaining preexisting software features. However, because many of these maintenance tasks are not planned, the software tends to degrade over time, causing technical debts, mainly on its non-functional requirements, such as reusability, understandability, and maintainability. Thus, this work proposes the continuous monitoring of some quality attributes of the software through metrics and the execution of automatic perfective maintenance (i.e., refactorings) to promote its rejuvenation in response to previously detected anti-patterns or bad smells. The created approach was evaluated through an experimental study, which found evidences that automatic software rejuvenation is possible. It also found evidence of patterns between refactorings and quality attributes, in other words, which of the refactorings used for improve, make them worse or do not change the quality attributes.

Keywords: Rejuvenation of Source Code, Refactoring, Quality Attributes, Metrics, Multi-Agent Systems.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API – *Application Programming Interface*

AST – *Abstract Syntax Tree*

BDI – *Belief Desire Intention*

DAO – *Data Access Object*

HTTP – *Hypertext Transfer Protocol*

IDE – *Integrated Development Environment*

GCS – Gerência de Configuração de Software

JPA – *Java Persistence API*

JVM – *Java Virtual Machine*

JSF – *Java Server Faces*

JSON – *JavaScript Object Notation*

PEC – Peixe-Espada Cliente

PES – Peixe-Espada Servidor

SCV – Sistema de Controle de Versões

SMA – Sistema multiagente

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO

1.1 – PROBLEMA E MOTIVAÇÃO

O desenvolvimento de software é diferente do desenvolvimento de outros produtos em diversos aspectos (Brooks 1987). Na manufatura e distribuição de software, por exemplo, é necessário apenas um gravador de CD/DVD ou de um processo de implantação em um servidor. Isso é certamente mais barato e mais simples do que a manufatura e distribuição de carros, casas e produtos agrícolas, por exemplo. No entanto, devido à facilidade do processo de manufatura de software, suas manutenções e evolução tendem a ser mais frequentes se comparadas a produtos físicos.

Este cenário conduz a um desafio: como conceber arquiteturas de software robustas o suficiente para serem capazes de apoiar às mudanças, suportarem evoluções consecutivas, acolherem mudanças nos requisitos, entre outras modificações? Embora existam várias orientações nesse sentido (Chrissis et al. 2006, Shaw e Garlan 1996, Softex 2009), ainda é difícil projetar arquiteturas de software que possam ser facilmente adaptadas em resposta à evolução dos requisitos. Vários fatores tornam a evolução de software difícil ou cara. Os mais notáveis são descritos a seguir:

Mudanças de tecnologia – O custo para treinamento de equipes que se encarregarão da manutenção e infraestrutura de tecnologias legadas pode se tornar muito alto, se comparado à aprendizagem e à adoção de novas tecnologias, devido ao tamanho, complexidade e tempo de vida do software. Esse fator ocorre devido à crescente criação de novas tecnologias, cada vez mais intuitivas e de alto nível, capazes de resolver os mesmos problemas com menos tempo de treinamento e implementação.

Evasão de mão de obra especializada – Durante a vida de um software bem-sucedido, as pessoas se aposentam, algumas são demitidas e outras contratadas. Em algumas situações, a equipe de desenvolvimento original é completamente substituída por uma equipe nova, levando à perda de memória organizacional.

Volume de trabalho – É comum o desenvolvimento de software ocorrer “sob pressão”, geralmente devido a prazos mal estimados. Esse fator faz com que o tempo

estabelecido pelo mercado muitas vezes suprima a qualidade, causando débitos técnicos[1] (Kerievsky 2004) no produto de software.

Esses fatores em conjunto, entre outros, contribuem ao longo do tempo para degenerar a arquitetura do software, mesmo quando melhores práticas são seguidas. O efeito resultante é que, após constantes entregas e manutenções de um produto de software por um longo período de tempo, a inclusão de novas funcionalidades torna-se cada vez mais difícil (Pressman 2004). Este é o resultado mais visível do envelhecimento de software (Parnas 1994).

Mesmo aplicando as orientações para se criar arquiteturas mais adaptáveis durante a construção do software, a estratégia mais comum para combater o envelhecimento do software é a realização de manutenções perfectivas periódicas. Conforme definido na norma ISO/IEC 9126 (2001), "manutenção perfectiva prevê melhorias para os usuários, melhorias da documentação do software e recodificação para melhorar o desempenho do software, manutenibilidade, ou outros atributos do software". Entretanto, a manutenção perfectiva normalmente exige um elevado conhecimento do software e um grande esforço para ser executada. Além disso, a manutenção perfectiva não deveria ser executada em paralelo com outras atividades de desenvolvimento, devido às dificuldades de junção (do inglês, *merge*) das mudanças estruturais (Mens 2002). Esses requisitos para a aplicação da manutenção perfectiva vão de encontro aos fatores acima mencionados (ou seja, mudanças de tecnologias, evasão de mão de obra e volume de trabalho), obrigando as empresas a adiarem indefinidamente a manutenção perfectiva e, consequentemente, aumentando o débito técnico.

1.2 – OBJETIVO

Dado o exposto, o objetivo deste trabalho é identificar se é possível rejuvenescer o código fonte a partir do contínuo monitoramento e aplicação de refatorações de forma totalmente automática. Com isso, é possível distinguir duas principais questões de pesquisa deste trabalho:

Questão 1 – Refatorações automáticas são capazes de rejuvenescer o código fonte?

[1] Débito Técnico é um termo usado para descrever o aumento do custo de manutenção (evolutiva ou corretiva) de um sistema devido às soluções rápidas e pouco planejadas, tomadas durante o seu desenvolvimento (Klinger et al. 2011). Um sistema está com débito técnico quando possui o acúmulo de decisões que infligem sua arquitetura.

Questão 2 – Caso a questão anterior seja respondida positivamente, determinadas refatorações automáticas são mais indicadas para rejuvenescer o código fonte sob a ótica de atributos de qualidade específicos?

Para alcançar este objetivo, uma abordagem denominada Peixe-Espada[2] poderia ser proposta. Essa abordagem utilizaria um Sistema Multiagente (SMA) para executar manutenção perfectiva autonomamente no produto de software, sem sobrecarregar a equipe de desenvolvimento. Um SMA pode ser visto como um conjunto de agentes de software que são entidades autônomas (i.e., capazes de executar sem a necessidade da intervenção humana ou de outros agentes), adaptativas (i.e., podem adaptar seu comportamento às mudanças no ambiente no qual estão inseridos), interativas e orientadas a objetivos (Wooldridge e Ciancarini 2001). O SMA proposto seria executado durante o tempo ocioso dos computadores do ambiente de desenvolvimento através do agendamento do período de ociosidade destes.

É importante notar que a abordagem Peixe-Espada se destinaria a empresas que adotam um ciclo de trabalho habitual de, por exemplo, oito horas por dia, deixando os recursos computacionais disponíveis durante o restante do tempo. O Peixe-Espada exploraria este fato como uma oportunidade de usar o parque computacional ocioso para realizar a manutenção perfectiva. Essa estratégia também contribuiria para o problema de merge de mudanças estruturais, discutido anteriormente, já que neste contexto, não haveria paralelismo com as tarefas de desenvolvimento.

A utilização dos recursos computacionais durante períodos de inatividade não é novidade. Por exemplo, iniciativas como o *World Community Grid* (http://www.worldcommunitygrid.org) visam encontrar cura para doenças de forma distribuída, executando em computadores ociosos de usuários comuns. No entanto, o aproveitamento desta oportunidade em Engenharia de Software é raro (entre poucas exceções, é possível citar o Hudson (Smart 2010), software de integração contínua). Normalmente, scripts são agendados para executar durante a noite nos servidores, mas os computadores do desenvolvimento estão desligados durante este período.

Em suma, os principais objetivos deste trabalho são:

[2] Peixe-Espada é uma espécie de peixe de hábitos noturnos, similar à nossa abordagem, que trabalha no período de ociosidade dos computadores de um ambiente de desenvolvimento, principalmente à noite.

- Definir e implementar uma infraestrutura robusta e extensível para o rejuvenescimento de software.
- Avaliar a infraestrutura implementada utilizando projetos reais.
- Identificar padrões iniciais entre refatorações e atributos de qualidade.

A abordagem Peixe-Espada seria avaliada com projetos de características distintas, como o tamanho (de pequeno a médio), equipes de desenvolvimento, e a natureza (comercial, open-source e acadêmico). Com o objetivo de apresentar resultados que indiciem que a aplicação de refatorações de maneira totalmente automática seria uma forma viável e promissora para lidar com o envelhecimento precoce do software. Além disso, é possível que esses resultados estabeleçam alguns padrões iniciais entre refatorações e melhorias dos atributos de qualidade.

1.3 – CONTEXTO

A Gerência de Configuração de Software (GCS) é uma subárea da Engenharia de Software. Seu principal objetivo é possibilitar a evolução controlada do software (Dart 1991). Para atingir esse objetivo, a GCS é dividida em três sistemas complementares (Murta 2006): (1) Sistema de Controle de Versões (SCV); (2) Sistema de Controle de Mudanças; e (3) Sistema de Gerenciamento de Construção. Estes sistemas podem ser apoiados por ferramentas ou executados manualmente.

O uso do GCS é indispensável para o desenvolvimento profissional de software. Ele pode ser observado em modelos de maturidade, como CMMI (Chrissis et al. 2006) e MPS.BR (Softex 2009), que introduzem essa exigência em seus níveis iniciais, respectivamente, o nível 2 e o nível F. Dessa forma, a abordagem Peixe-Espada pressupõe que, pelo menos, o SCV seja natural nas equipes de desenvolvimento. Como o Peixe-Espada também se baseia nos conceitos e ferramentas de SCV, alguns conceitos básicos são definidos nesta seção para o entendimento da abordagem.

O ciclo de trabalho habitual realizado por equipes de desenvolvimento que utilizam SCV (mais especificamente, um SCV centralizado) compreende as seguintes tarefas:

1. O desenvolvedor chega ao trabalho e precisa fazer alterações em um projeto específico, porém, a versão mais recente do código fonte está no repositório do SCV. Assim, o desenvolvedor tem que preencher um espaço de trabalho local com a versão do projeto mais atual contida no repositório, utilizando a operação de ***checkout***. Essa operação é capaz de recuperar qualquer versão do repositório, não sendo restrita à mais atual.
2. O desenvolvedor faz as alterações desejadas no espaço de trabalho local e decide compartilhar com o restante da equipe. Sendo assim, o desenvolvedor tem que enviar a versão atual do espaço de trabalho para o repositório do SCV através da operação de ***checkin***. Esta operação também é conhecida como ***commit*** em muitas ferramentas de SCV.
3. Finalmente, outros desenvolvedores, que estão trabalhando no mesmo projeto, podem querer atualizar seus espaços de trabalho com a versão do projeto contida no repositório do SCV. Para fazer isso, eles podem utilizar a operação de ***update***, que combina a versão mais atual do repositório com a versão do seu espaço de trabalho local. Essa operação altera apenas no espaço de trabalho do desenvolvedor.

Alguns conceitos superficiais sobre Sistema de Gerenciamento de Construção também são necessários para o entendimento desta dissertação.

A Gerência de Construção é necessária para diminuir a dificuldade comumente encontrada nas atividades de construção e liberação de sistemas complexos, principalmente quando há repetição dessas atividades. A Gerência de Construção controla a transformação de itens fonte em itens derivados (Murta 2006). Apoiada por ferramentas, ela é responsável por criar um processo automático para geração de um produto derivado a partir de um código fonte, fazendo a gestão de dependências dos módulos envolvidos, evitando compilações desnecessárias e calculando transitividade entre as dependências. O produto gerado pode ser um módulo ou um empacotamento qualquer (e.g., para Java, um .jar, .class, .war), com suas dependências ou não, de acordo com o processo previamente configurado. Os Sistemas de Gerenciamento de Construção evoluíram muito desde suas primeiras implementações, abrangendo tarefas como: gestão de processo de construção; gestão de dependências;

utilização do repositório organizacional[3]; e geração do site do projeto. O Maven (Company 2008) é uma ferramenta atual largamente utilizada para Gerenciamento de Construção, que possui todas essas funcionalidades. O Maven foi criado para dar suporte nativo à linguagem de programação Java. Sua estrutura permite acoplamento de diversos *plug-ins*, o que permite suportar outras linguagens de programação e oferecer funcionalidades diversas.

1.4 – ORGANIZAÇÃO

O restante desta dissertação está organizado da seguinte maneira:

O Capítulo 2 apresenta uma revisão da literatura sobre refatorações de software, descrevendo desde refatorações manuais até as que podem ser totalmente automatizadas. Esse capítulo também apresenta, brevemente, algumas medições de software, dado que as refatorações totalmente automatizadas precisam ser avaliadas para serem aceitas. Ao final deste capítulo, são apresentadas as abordagens relacionadas que tratam de refatorações totalmente automatizadas e melhoria de código fonte. Além disso, são discutidas as razões que justificam a criação de uma nova abordagem.

O Capítulo 3 apresenta a abordagem Peixe-Espada, mostrando em detalhes qual foi o processo utilizado para reduzir o envelhecimento precoce de software através da adoção de um SMA. Dentro do SMA é discutida a função de cada agente, bem como todos os detalhes para se alcançar um ciclo de rejuvenescimento. Nesse capítulo é visto como as refatorações são aplicadas, avaliadas, aceitas e como essas informações são disponibilizadas para a equipe de desenvolvimento em geral. Também são discutidas algumas estratégias de paralelismo, objetivando o melhor aproveitamento dos recursos computacionais.

O Capítulo 4 apresenta o protótipo implementado, juntamente com um exemplo de sua utilização. Nesse capítulo são detalhadas as ferramentas que foram desenvolvidas para atender a abordagem, assim como o passo a passo completo para executar o sistema sobre os projetos passíveis de serem submetidos à abordagem Peixe-Espada. Além disso, também é discutida a estrutura criada para o Peixe-Espada, quais são as restrições do protótipo e suas extensões naturais.

[3] O repositório organizacional é o local onde são armazenados e gerenciados os artefatos gerados no ambiente de desenvolvimento, como módulos internos, projetos empacotados em diferentes versões, etc.

No Capítulo 5 são mostrados os resultados da avaliação do Peixe-Espada e o processo utilizado para a obtensão de tais resultados. Esse capítulo descreve os projetos utilizados na avaliação e quais foram os reflexos da aplicação do Peixe-Espada nesses projetos. Nesse capítulo também são tecidas algumas considerações sobre estratégias de paralelismos (introduzidas no Capítulo 3) e possíveis ameaças ao estudo.

Finalmente, o Capítulo 6 conclui esta dissertação, sumarizando as contribuições da abordagem Peixe-Espada, discutindo algumas limitações e apresentando possíveis trabalhos futuros.

CAPÍTULO 2 – REFATORAÇÃO DE SOFTWARE

2.1 – INTRODUÇÃO

Refatorar é uma forma de mudar internamente o software sem alterar seu comportamento, com o objetivo de melhorar seus atributos de qualidade ou, como Martin Fowler *et al.* (1999) definem, é "uma alteração feita na estrutura interna do software para torná-lo mais fácil de ser entendido e menos custoso de ser modificado, sem alterar seu comportamento observável". As principais motivações para fazer uma refatoração são (Kerievsky 2004): tornar um código fonte mais legível, flexível e, consequentemente, mais manutenível. Para isso, o processo de refatoração consiste em remover duplicação de código, simplificar lógica condicional e tornar o código mais claro.

As refatorações podem ser tão pequenas quanto renomear um método ou tão grandes quanto unificar duas hierarquias, durando desde segundos a dias, semanas ou meses para serem concluídas. Padrões de projeto (Gamma et al. 1995) são utilizados com frequência para auxiliar as refatorações, o que não significa que o objetivo de tais refatorações seja necessariamente atender a um padrão de projeto. Nesse sentido, Kerievsky (2004) enfatiza que uma refatoração deveria se completar no momento que o código atendesse melhor às necessidades do projeto em questão e não necessariamente quando um padrão de projeto fosse atingido.

Metodologias como XP (Beck 1999) e desenvolvimento dirigido por testes (Beck 2002) pregam a necessidade constante de refatoração, sendo um procedimento comum: (1) escrever um caso de teste; (2) escrever um código que faça o teste passar; (3) refatorar o código que fez o teste passar; e (4) continuar o ciclo em função de novos casos de teste. Desta forma, refatoração está se tornando, cada vez mais, uma atividade recorrente no cotidiano de desenvolvimento de software.

Nas próximas seções são apresentados diversos tipos de refatorações, desde as refatorações manuais convencionais (Seção 2.2) até um subconjunto de refatorações ditas automatizáveis (Seção 2.3). Essas refatorações podem também ser avaliadas de forma automática, como é apresentado na Seção 2.4. Já na Seção 2.5, são apresentados trabalhos de pesquisa relacionados à temática de refatoração de software, principalmente abordagens

automáticas. Por fim, na Seção 2.6 são apresentadas as considerações finais e justificativas para a criação de uma nova abordagem para melhoria automática de software.

2.2 – REFATORAÇÕES MANUAIS

Como apresentado na Seção 2.1, refatorar é uma realidade das equipes de desenvolvimento. Algumas equipes refatoram seus códigos fonte instintivamente, ou seja, ao se depararem com o código, sentem a necessidade de fazer alguma melhoria e a fazem sem seguir metodicamente uma sequência de passos pré-definida. No entanto, outras equipes, com diferentes graus de maturidade, possuem especialistas que auxiliam na tarefa de refatoração. Esses especialistas ajudam a localizar e criar pequenas tarefas encadeadas para cada refatoração, minimizando as chances de criação de novos *bugs* nesse processo muitas vezes não trivial. Essa necessidade de refatorar normalmente é reforçada quando o código não está legível o suficiente ou quando há a necessidade de uma extensão e/ou generalização.

Martin Fowler *et al.* (1999) explicam detalhadamente cada uma de suas 72 refatorações. Cada refatoração catalogada é descrita com um exemplo, motivações e a maneira mais indicada de se concluir a refatoração em questão. Por exemplo, na refatoração *Extract Superclass*, uma nova superclasse é criada para classes que possuam interesses comuns entre si. A Figura 1 exemplifica essa refatoração criando a classe *Pessoa* com os métodos comuns às classes *Aluno* e *Funcionario*. Esses métodos em comum (*getNome*) são removidos dessas classes, que por sua vez são colocadas como subclasses de *Pessoa*. Outros métodos são declarados na classe criada e sobrescritos nas classes herdeiras (*getCustoAnual*) por eventuais diferenças de implementação. A motivação base para tal refatoração é a remoção de código duplicado, facilitando assim eventuais manutenções, por não mais haver a necessidade de se realizar alterações de um mesmo interesse em pontos distintos.

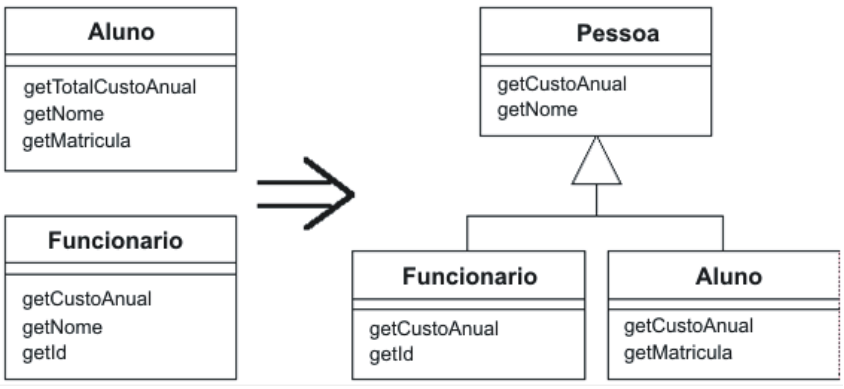


Figura 1: Refatoração *Extract Superclass*.

Uma das refatorações mais rotineiras é a *Extract Method* (Fowler et al. 1999). Um método pode ser extraído por vários motivos: para aumentar a legibilidade do código, para aumentar a coesão de cada método criado, etc. Fowler costuma dizer: "Se você sente a necessidade de criar um comentário, primeiro tente refatorar o código, então qualquer comentário se tornará supérfluo". A adoção de refatorações não justifica a ausência de documentação no geral, mas claramente facilita o entendimento do código por si só. Um exemplo de *Extract Method* é mostrado na Figura 2 e Figura 3. A variável *_nome* não é passada por parâmetro no método criado por se tratar de uma variável de instância.

```
void imprimeDivida() {
        imprimeFaixa();

        //imprime detalhes
        System.out.println ("nome:         " + _nome);
        System.out.println ("quantidade" + getDebito());
}
```

Figura 2: Antes da refatoração *Extract Method* (Fowler et al. 1999).

```
void imprimeDivida() {
        imprimeFaixa();
        imprimeDetalhes(getDebito());
}

void imprimeDetalhes(double debito) {
        System.out.println ("nome:         " + _nome);
        System.out.println ("quantidade" + debito);
}
```

Figura 3: Depois da refatoração *Extract Method* (Fowler et al. 1999).

Existe, no entanto, refatorações com outros propósitos, cujo produto final é um padrão de projeto, por exemplo. Há alguns questionamentos sobre tal fato, dado que aplicar um padrão de projeto não significa que necessariamente o projeto vá melhorar, tendo em vista que muitos padrões de projetos trazem um grau extra de complexidade. Por esses motivos que Kerievsky (2004) detalha as refatorações para padrões aplicando pequenos passos bem definidos, cujo critério de parada depende da necessidade do projeto, e não necessariamente quando o padrão de projeto é atingido.

2.3 – REFATORAÇÕES AUTOMATIZÁVEIS

As refatorações possuem diversas finalidades, porém todas elas têm o propósito geral de facilitar o desenvolvimento e evolução do software. Atualmente quase todas as refatorações são realizadas com o auxílio de ferramentas e IDEs em um processo interativo. Não há uma definição consensual na literatura sobre refatorações automatizáveis. Sendo

assim, este trabalho classifica como *refatorações automatizáveis* aquelas que não necessitam de decisões subjetivas, em outras palavras, aquelas refatorações que não dependem necessariamente da interação humana. Seguindo essa definição, as refatorações que necessitam de nomeações de métodos, classes, superclasses, interfaces, pacotes, etc., não são automatizáveis. Por exemplo, na Figura 1 a definição de um nome para a nova classe (*Pessoa*) foi necessária, assim como a definição do nome para o novo método (*imprimeDetalhes*) na Figura 3. Obviamente, uma abordagem poderia inferir esses nomes de alguma forma, o que certamente dificultaria o entendimento do código fonte, divergindo do principal propósito de uma refatoração.

Atendendo às exigências supracitadas, as refatorações automatizáveis consideradas neste trabalho são (O'Keeffe e Ó Cinnéide 2008):

- *Push Down Field* – Move um atributo de uma classe para alguma de suas subclasses. Esta refatoração tenta simplificar o projeto pela redução do número de classes que acessam o atributo.
- *Pull Up Field* – Move um atributo de uma classe para a sua superclasse imediata. Esta refatoração destina-se a eliminar a duplicação de declarações de atributo nas classes irmãs.
- *Push Down Method* – Move um método de uma classe para uma de suas subclasses. Esta refatoração destina-se a simplificar o projeto pela redução de tamanho de interfaces e classes de onde esses métodos deverão sair.
- *Pull Up Method* – Move um método de uma classe para sua superclasse imediata. Esta refatoração destina-se a ajudar a eliminar métodos duplicados entre classes irmãs e, consequentemente, reduzir duplicação de código no geral.
- *Collapse Hierarchy* – Remove uma classe não folha, ou seja, uma classe que possua subclasses, de uma hierarquia de herança. Esta refatoração destina-se a reduzir a complexidade do projeto pela remoção de classes supérfluas contidas nele.
- *Increase Field Security* – Aumenta a segurança de um atributo público para protegido ou de um atributo protegido para privado. Essa refatoração aumenta o encapsulamento de dados. Ela também é chamada de *Encapsulate Fields*

quando são criados métodos públicos *getters* e *setters* para realizar, respectivamente, leitura e escrita do atributo privado.

- *Decrease Field Security* – Diminui a segurança de um atributo privado para protegido ou de um atributo protegido para público. Esta refatoração reduz o encapsulamento dos dados, que por alguma razão esteja influenciando negativamente no projeto. Uma medida de ganho de desempenho pode ser considerada um exemplo, com a eliminação das chamadas aos métodos para o uso direto do atributo. Para que essa medida não prejudique a manutenção do código-fonte, esta refatoração pode ser aplicada apenas no momento de construção do projeto.
- *Replace Inheritance with Delegation* – Substitui uma relação de herança entre duas classes com uma relação de delegação, assim, a subclasse terá um atributo do tipo da superclasse. Esta refatoração é usada para corrigir uma situação em que uma subclasse não utiliza características suficientes de uma superclasse que justifique a relação de especialização.
- *Replace Delegation with Inheritance* – Substitui uma relação de delegação entre duas classes por uma relação de herança, onde a classe delegação torna-se uma subclasse da classe antiga delegada. Esta refatoração pode ser utilizada na situação onde uma classe está utilizando funcionalidades suficientes de outra classe que justifique a criação de uma relação de herança.
- *Increase Method Security* – Aumenta a segurança de um método, alterando seu acesso de protegido para privado, ou de público para protegido. Esta refatoração pode reduzir a quantidade de interfaces públicas de uma classe.
- *Decrease Method Security* – Diminui a segurança de um método, alterando seu acesso de protegido para público ou de privado para protegido. Esta refatoração pode aumentar a quantidade de interfaces públicas de uma classe. Quanto maior o número de métodos públicos, maior é o número de serviços que a classe disponibiliza, no entanto, muitos métodos podem não estar sendo utilizados.
- *Make Superclass Abstract* – Declara uma classe sem construtor e explicitamente abstrata. Essa refatoração aumenta algumas medidas de abstração e também pode ser vista como uma refatoração intermediária para facilitar outras refatoração.

O processo de aplicação de uma refatoração automática não é trivial, devido às tarefas complexas envolvidas, como a busca do que será refatorado (*sintomas de refatoração*) e a aplicação da refatoração. Para detectar candidatos a refatoração automática é necessário satisfazer pré-condições específicas de cada refatoração. A Tabela 1 mostra algumas refatorações e suas respectivas pré-condições.

Tabela 1: Refatorações automáticas e pré-condições.

Refatoração	Pré-Condições
Encapsulate Fields	Atributos públicos. Os alvos são classes que possuem esses atributos e as classes que os referenciam.
Pull Up Methods	Método cuja classe tem que ter uma superclasse pertencente ao projeto. Este método não deve estar sobrescrevendo nenhum método de sua hierarquia de superclasses. Os alvos são métodos, suas classes e superclasses.
Pull Up Fields	A superclasse do atributo deve estar no projeto, nenhum outro atributo da superclasse pode ter o mesmo nome do atributo alvo. Os alvos são atributos, suas classes e superclasses.
Clean Imports	Classes com *imports* não utilizados. Essas classes são os alvos.

A partir de observação da literatura (Ferber et al. 2009, O'Keeffe e Ó Cinnéide 2008, Seng et al. 2006, Taneja et al. 2007), é possível notar que as soluções mais utilizadas para realizar buscas de *sintomas* e aplicação das refatorações são:

Árvore sintática abstrata – Do inglês *Abstract Syntax Tree* (AST), trata-se de uma estrutura em forma de árvore que representa estruturas sintáticas, de acordo com a gramática formal da linguagem. Cada nó da árvore denota um elemento do código fonte. Nós não folhas representam operadores, enquanto nós folhas representam operandos. A sintaxe é abstrata por não representar todos os detalhes da sintaxe real. Por exemplo, delimitadores de escopo são omitidos da árvore como mostrado no pequeno exemplo na Figura 4. Essa estrutura permite realizar operações diretamente sobre o código fonte. Diversas APIs fornecem funções para ler, inserir, deletar e alterar um código fonte representado em uma AST, permitindo seu uso pleno no contexto de refatorações.

Metaprogramação – A metaprogramação destina-se a manipular programas, fragmentos de programas ou a si mesma. Operações realizadas com a metaprogramação geram novos programas e/ou fragmentos de programas, de forma simples e intuitiva (Wuyts 2001). As linguagens para metaprogramação são chamadas metalinguagens. Os compiladores

são bons exemplos do uso de metaprogramação. Ela também pode ser utilizada por geradores de documentos (Brand e Visser 1996), verificadores de plágios (Donaldson et al. 1981, Wise 1992) e sistemas de transformação de programas (Partsch e Steinbrüggen 1983), como busca de sintomas, sugestão e aplicação de refatorações.

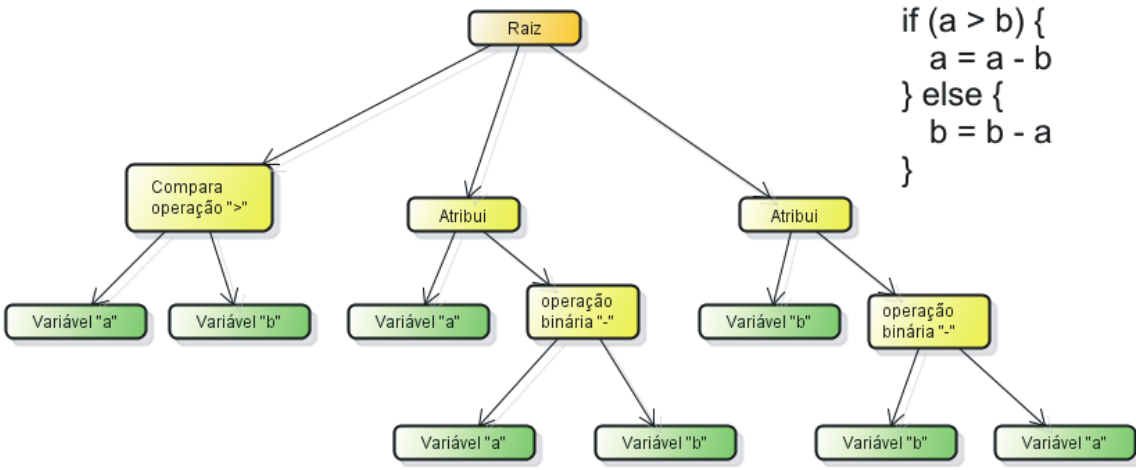


Figura 4: Exemplo de AST.

É possível identificar conhecidas IDEs que utilizam AST para manipulação de código fonte, como *NetBeans* (http://netbeans.org/), Eclipse (http://www.eclipse.org/), *IntelliJ* (http://www.jetbrains.com/idea/), etc. Existem também algumas ferramentas específicas para refatorações e extração de métricas que utilizam AST, como *RefactorIt* (http://refactorit.sourceforge.net/), *JRefactory* (http://jrefactory.sourceforge.net/), dentre outras. Já para metaprogramação, as ferramentas são mais restritas. Uma representante é a linguagem *MetaJ* (De Oliveira et al. 2004) em Java, que possibilita representação e manipulação da linguagem objeto (linguagem alvo da manipulação) através do uso de *templates* próprios.

2.4 – AVALIAÇÃO DA QUALIDADE DE REFATORAÇÕES AUTOMATIZÁVEIS

De uma forma geral a pessoa que efetuou uma refatoração manual é responsável por avaliar os benefícios das refatorações e aceitá-las ou não. O mesmo vale para refatorações automáticas *on-line*, onde uma pessoa está ordenando e acompanhando cada refatoração. No entanto, quando se deseja executar uma refatoração automática, de forma independente, é necessário que existam critérios para avaliar se houve ou não melhoria com a aplicação da refatoração. As próximas subseções fazem uma breve apresentação de qualidade do produto de software e avaliações automáticas por meio de métricas.

2.4.1 – QUALIDADE DE PRODUTO DE SOFTWARE

Uma das principais normas sobre qualidade de produto de software é a ISO/IEC 9126 (ISO 2001), que define qualidade de produto de software através da padronização de um modelo de qualidade. Esse modelo define seis características de qualidade de software e suas respectivas subcaracterísticas. Característica de qualidade de software também é conhecida por atributos de qualidade de software e assim será chamada ao longo desta dissertação.

A qualidade do produto de software pode ser vista com o fim de atender as necessidades reais do usuário[4], tão detalhadamente quanto possível (ABNT 2003). Não é possível determinar todos os requisitos de um produto dado que (ABNT 2003): frequentemente, o usuário não sabe de suas reais necessidades; as necessidades podem mudar depois de terem sido explicitadas; usuários diferentes podem ter ambientes operacionais diferentes; e, em se tratando de um software de prateleira, pode ser impossível consultar todos os usuários.

A Figura 5 evidencia as diferentes visões de qualidade do produto e suas métricas nos distintos ciclos de vida do software. Os principais conceitos a respeito das visões de qualidade são descritos a seguir, de acordo com a norma NBR ISO/IEC 9126-1 (ABNT 2003):

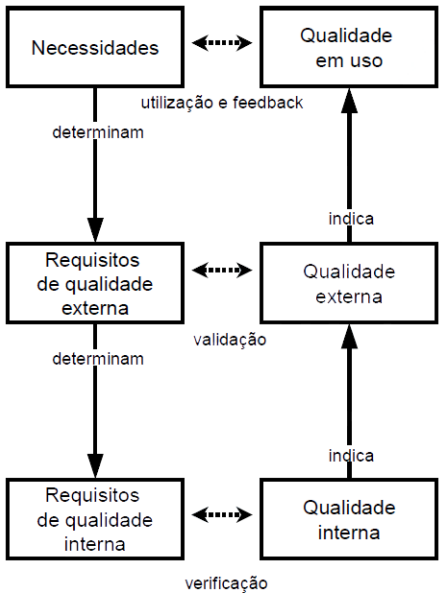


Figura 5: Visões de qualidade do produto (ABNT 2003).

[4] Refere-se a qualquer tipo de usuário em potencial, incluindo operadores e mantenedores, sendo que seus requisitos podem ser diferentes.

- Qualidade interna – É a qualidade esperada do ponto de vista interno do produto, ou seja, código fonte, documentação, estruturação, estratégias de avaliação e de verificação utilizadas durante o desenvolvimento, etc. Esta qualidade interessa ao desenvolvedor.
- Qualidade externa estimada (ou prevista) – É a qualidade estimada ou prevista para o produto final de software, em cada estágio de desenvolvimento e para cada característica de qualidade, baseada no conhecimento da qualidade interna.
- Qualidade externa – É a qualidade esperada do ponto de vista externo do produto, ou seja, quando o software é executado, quais e como são realizados os testes em ambientes simulados. Durante os testes, os defeitos devem ser eliminados, na medida do possível. Esta qualidade é comumente cobrada pelo usuário.
- Qualidade em uso estimada (ou prevista) – Trata-se da qualidade estimada para o produto final do software baseada na qualidade interna e externa. Em cada estágio de desenvolvimento e para cada característica de qualidade em uso.
- Qualidade em uso – É a visão da qualidade do produto de software do ponto de vista do usuário. Essa visão se dá no momento em que o produto é usado em um ambiente e contexto de uso especificado. Ela mede o quanto usuários podem atingir seus objetivos em um determinado ambiente e não as propriedades do software em si. Em outras palavras, ela é medida em termos do resultado do uso do software neste ambiente e não das propriedades do próprio software, sendo assim, eficácia, produtividade e satisfação são possíveis características de uso.

A norma ISO/IEC 9126 (ISO 2001) é dividida em quatro partes. A primeira é o modelo de qualidade, a segunda, terceira e quarta parte são referentes às métricas externas, internas e de qualidade. A Figura 6 apresenta o modelo de qualidade externa e interna. Para este trabalho, apenas a qualidade interna é relevante, visto que uma das formas de avaliação das refatorações automatizáveis é através da medição da qualidade interna do software.

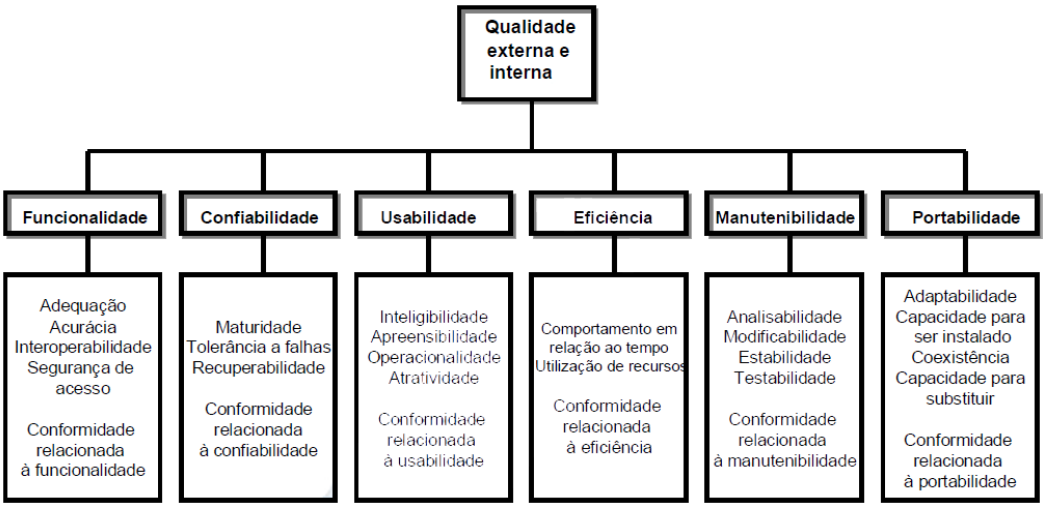


Figura 6: Modelo de qualidade interna e externa (ISO 2001, ABNT 2003).

As definições dos atributos de qualidade de software dadas pela ISO/IEC 9126 (ISO 2001) são detalhadas a seguir:

- **Funcionalidade** – Capacidade do produto de software de prover funções que atendam às necessidades explícitas e implícitas, quando o software estiver sendo utilizado sob as condições especificadas. Este atributo de qualidade está relacionado com que o software faz para atender às necessidades a ele atribuídas.
- **Confiabilidade** – Capacidade do produto de software de manter um nível de desempenho especificado, quando usado em condições pré-determinadas. Este atributo de qualidade está relacionado à: Maturidade – capacidade do produto de software de evitar falhas decorrentes de defeitos no software; Tolerância a falhas – capacidade do produto de software de manter um nível de desempenho especificado em casos de defeitos no software ou de violação de sua interface especificada; Recuperabilidade – capacidade do produto de software de restabelecer seu nível de desempenho especificado e recuperar os dados diretamente afetados no caso de uma falha; e Conformidade – capacidade do produto de software de estar de acordo com normas, convenções ou regulamentações.
- **Usabilidade** – Capacidade do produto de software de ser compreendido, aprendido, operado e atraente ao usuário, quando usado sob as condições especificadas.

- **Eficiência** – Capacidade do produto de software de apresentar desempenho apropriado, relativo à quantidade de recursos usados, sob as condições especificadas. Os recursos podem ser outros produtos, configuração de hardware e software do sistema e materiais (ex. papel para impressão, disquetes). Para um sistema que é operado por um usuário, a combinação de funcionalidade, confiabilidade, usabilidade e eficiência pode ser medida externamente pela qualidade em uso.
- **Manutenibilidade** – Capacidade do produto de software de ser modificado. As modificações podem incluir correções, melhorias ou adaptações do software, devido a mudanças no ambiente, nos seus requisitos ou especificações funcionais.
- **Portabilidade** – Capacidade do produto de software de ser transferido de um ambiente para outro. O ambiente pode ser organizacional, de hardware ou de software.

Existem, no entanto, outros atributos de qualidade de software que quantificam outras características, como os descritos a seguir (Bansiya e Davis 2002):

- **Reusabilidade** – Reflete na presença de características em projetos orientados a objetos, que permitem que fragmentos do projeto sejam reaplicados a novos problemas sem esforço significante.
- **Flexibilidade** – Característica que permite a incorporação de novas mudanças no projeto. É a capacidade do projeto de ser adaptado para prover capacidades relacionadas funcionalmente.
- **Compreensibilidade** – Propriedade do projeto de ser facilmente entendido. Este atributo de qualidade está diretamente relacionado à complexidade da estrutura do projeto.
- **Extensibilidade** – Refere-se à existência e uso de propriedades do projeto que permitem a incorporação de novos requisitos. Em outras palavras, é a capacidade do projeto ser estendido.
- **Efetividade** – Capacidade do projeto de alcançar o comportamento desejado, usando conceitos e técnicas de projetos orientados a objetos.

2.4.2 – MÉTRICAS DE SOFTWARE

Avaliar a qualidade do código fonte de um software manualmente não é uma tarefa simples, consome muito tempo de trabalho, desgasta a equipe e ainda assim, está propensa a erros humanos. Uma inspeção completa, realizada por profissionais da área, em todos os níveis de abstração do software (lógico, infraestrutura, interface homem computador, etc) é de extrema efetividade e pode ser feita manualmente. Porém, em muitos casos, essa prática torna-se inviável devido aos altos custos envolvidos. A fim de auxiliar no processo de avaliação e medição de software, são utilizadas algumas métricas. As métricas medem, de maneira quantitativa, diferentes atributos do sistema, componente ou processo (IEEE 1990). Por exemplo, o número de linhas de código de um software é uma métrica cujo limite inferior é zero, e o limite superior é infinito. A Tabela 2 mostra algumas propriedades que podem ser medidas em um projeto de software desenvolvido utilizando orientação a objetos.

Tabela 2: Definições de propriedades de projeto (Bansiya e Davis 2002).

Propriedade	Descrição
Tamanho do Projeto	Trata-se de uma medida do número de classes usadas no projeto.
Hierarquias	Hierarquias são usadas para representar diferentes generalizações e especializações concebidas em um projeto. É uma contagem do número de classes que possuem subclasses, mas não possuem superclasses no projeto.
Abstração	Uma medida do aspecto de generalização e especialização de um projeto. Classes que tem uma ou mais descendentes expõem essa propriedade de abstração.
Encapsulamento	Define o fechamento dos dados e comportamento como uma construção única. Nos projetos orientados a objetos, refere-se à criação de classes que impedem o acesso às declarações de atributos, tornando-os privados para proteger a representação interna desses atributos.
Acoplamento	Define a interdependência de um objeto com outros em um projeto. É a medida do número de outros objetos que são acessados por um dado objeto para que este funcione corretamente.
Coesão	Avalia o parentesco de métodos e atributos de uma classe. Fortes sobreposições nos parâmetros de métodos e tipos de atributos é uma indicação de forte coesão.
Composição	Medida de relacionamentos do tipo: “parte de”; “tem”; “consiste de”; ou “parte do todo”. Que são classificados como relacionamentos de agregação em um projeto orientado a objeto.
Herança	Uma medida de relacionamentos “é um” entre classes. Este relacionamento é relatado ao nível de aninhamento de classes em uma hierarquia de heranças.
Polimorfismo	A habilidade para substituir objetos cujas interfaces referenciam outra em tempo de execução.
Mensagens	A medida do número de métodos públicos que estão disponíveis como serviços para outras classes. Essa é uma medida dos serviços que a classe provê.
Complexidade	Uma medida do grau de dificuldade no entendimento e compreensão de estruturas internas e externas das classes e seus relacionamentos.

Para medir as propriedades supracitadas, Bansiya e Davis (2002) propõem as métricas mostradas na Tabela 3.

Tabela 3: Propriedades de projeto e suas respectivas métricas (Bansiya e Davis 2002).

Propriedade	Métrica	Descrição
Tamanho do Projeto	*Design Size In Classes(**DSC**)*	Indica o número de classes de um projeto.
Hierarquias	*Number of Hierarchies (**NHO**)*	Indica o número total de hierarquias de classes do projeto.
Abstração	*Average Number of Ancestors (**ANA**)*	Indica o número médio de classes que cada classe do projeto herda informações.
Encapsulamento	*Data Access Metric (**DAM**)*	Indica a razão entre os atributos privados (protegidos) e o número total de atributos. Sendo assim, seus valores variam de 0 a 1.
Acoplamento	*Direct Class Coupling (**DCC**)*	Indica o número de classes diferentes com que uma classe se relaciona. Essa métrica inclui classes utilizadas diretamente na declaração de atributos e as classes utilizadas em parâmetros dos métodos.
Coesão	*Cohesion Among Methods of Class (**CAM**)*	Computa o parentesco entre métodos de uma classe baseada na lista de parâmetros dos métodos. Essa métrica é computada usando a soma das interseções dos parâmetros de um método com o conjunto independente máximo de todos os tipos de parâmetros de uma classe. Um valor dessa métrica desejável deve ser próximo a 1 (varia de 0 a 1) (Bansiya et al. 1999).
Composição	*Measure of Aggregation (**MOA**)*	Essa métrica mede a relação parte-todo com base nos atributos declarados. A métrica é a contagem do número de declaração de dados cujos tipos são classes definidas pelo usuário.
Herança	*Measure of Functional Abstraction (**MFA**)*	Indica a razão entre os métodos herdados e todos os métodos acessíveis de uma classe.
Polimorfismo	*Number Of Polymorphic Methods (**NOP**)*	Indica o número de métodos que apresentam comportamento polimórfico.
Mensagens	*Class Interface Size (**CIS**)*	Indica o número de métodos públicos em uma classe.
Complexidade	*Number of Methods (**NOM**)*	Indica o número de métodos de uma classe.

O estudo de métricas de software é largamente explorado, existindo assim muitos outros trabalhos na literatura a respeito (Booch 1990, Coad e Yourdon 1990, Fenton e Pfleeger 1998, Finkbine 1996, Kan 2002).

2.4.2.1 – CÁLCULO DOS ATRIBUTOS DE QUALIDADE

Para medir os atributos de qualidade de um software, alguns trabalhos propõem modelos de métricas. A Tabela 4 mostra uma forma de calcular alguns atributos de qualidade utilizando as métricas já descritas. As métricas utilizadas para medir os atributos de qualidade também são chamadas de métricas de qualidade.

Tabela 4: Cálculo dos atributos de qualidade (Bansiya e Davis 2002).

Atributos de Qualidade	Fórmula
Reusabilidade	(-0.25*__DCC__ + 0.25*__CAM__ + 0.50*__CIS__ + 0.50*__DSC__)
Flexibilidade	(+0.25*__DAM__ - 0.25*__DCC__ + 0.50*__MOA__ + 0.50*__NOP__)
Compreensibilidade	(-0.33*__ANA__ + 0.33*__DAM__ - 0.33*__DCC__ + 0.33*__CAM__ - 0.33*__NOP__ - 0.33*__NOM__ - 0.33*__DSC__)
Funcionalidade	(+0.12*__CAM__ + 0.22*__NOP__ + 0.22*__CIS__ + 0.22*__DSC__ + 0.22*__NOH__)
Extensibilidade	(+0.50*__ANA__ - 0.50*__DCC__ + 0.50*__MFA__ + 0.50*__NOP__)
Efetividade	(+0.20*__ANA__ + 0.20*__DAM__ + 0.20*__MOA__ + 0.20* __MFA__ + 0.20*__NOP__)

Dessa maneira, a aplicação das refatorações de forma totalmente automática pode ser avaliada conforme a necessidade de melhoria de alguma métrica ou atributo de qualidade. O uso de métricas pode guiar uma refatoração basicamente de duas formas: (1) aplicando-se uma refatoração sabendo de antemão que melhorará a métrica pretendida; (2) ou simplesmente aplicando-se várias refatorações e a cada aplicação verificar se houve melhoria da métrica pretendida. Algumas dessas abordagens são apresentadas na próxima seção.

2.5 – ABORDAGEM DE REFATORAÇÕES AUTOMÁTICAS

O processo de revisão da literatura foi realizado em duas fases. A primeira fase foi feita através da busca de assuntos chave nas principais fontes da área (ACM, IEEE, ScienceDirect e SpringerLink). Os assuntos chave procurados foram: Refatorações automáticas; Melhoria automática da qualidade do software; e Processo de melhoria automática de atributos de qualidade. As buscas foram feitas com permutação e sinônimos desses assuntos (“automatic refactoring”, “automatic software improvement”, “improve quality attribute”, “process of improving quality attribute”, “process of automatic refactoring”, “evaluation of refactoring with metrics”, “evaluation of refactoring with quality attribute”). Na segunda fase, foi realizada uma busca cruzada de referências, ou seja, os

trabalhos encontrados na primeira fase que mais se aproximaram de abordagens automáticas foram utilizados para se obter trabalhos relevantes que referenciam e/ou são referenciados pelos trabalhos inicialmente escolhidos. Naturalmente, muitos trabalhos são descartados por não terem intersecção aos assuntos básicos e a busca se esgota.

Em ambas as fases, os resultados foram analisados para inicialmente descartar todos os trabalhos que não se enquadram em abordagens automáticas. Posteriormente, outro filtro manual foi aplicado para se obter os trabalhos com maior relevância sobre aplicação de refatorações e avaliações automáticas, de forma a manter o foco nos problemas de equipes de desenvolvimento, cujas deficiências são tempo para realizar refatorações, experiência da equipe, falta de processo de melhoria do software, etc. Sendo assim, restaram alguns grupos que se encaixam, tais como os trabalhos de refatorações automatizáveis (Moore 1996, O'Keeffe e Ó Cinnéide 2008, Sagonas e Avgerinos 2009, Abdeen et al. 2009), sugestão de refatorações (Simon et al. 2001, Tourwe e Mens 2003, Czibula e Czibula 2008, Tsantalis e Chatzigeorgiou 2010), detecção de maus cheiros (do inglês, *bad smells*) e/ou antipadrões (Simon et al. 2001, Meyer 2006, Khomh et al. 2009, Moura 2009), reengenharia (Mancoridis et al. 1998, Simon et al. 2001, Meyer 2006, Ferber et al. 2009) e que tratam de refatorações específicas (Vikram Jamwal e Sridhar Iyer 2005, Taneja et al. 2007, Ferber et al. 2009).

Muitos desses trabalhos utilizam o conceito de Engenharia de Software Baseada em Buscas (do inglês, *Search Based Software Engineering* − SBSE), onde abordagens baseadas em buscas para resolver problemas de otimização em engenharia de software são aplicadas (Harman e Clark 2004). Um trabalho importante que aponta diversos trabalhos acadêmicos em SBSE pode ser encontrado em Harman et al. (2009).

O'Keeffe e Ó Cinnéide (2004) apresentam uma abordagem de refatoração automática de software. Para avaliar a qualidade das refatorações aplicadas, os autores utilizam algumas heurísticas existentes na literatura. Essas heurísticas baseiam-se em maximizar ou minimizar métricas. A fim de evitar conflitos de interesses entre as heurísticas, os autores listam uma ordem de prioridades entre as mesmas. Em um trabalho posterior, O'Keeffe e Ó Cinnéide (2008) se basearam nas funções de avaliações de Bansiya e Davis (2002) que proveem um conjunto de métricas para calcular atributos de qualidade de software, como, por exemplo, flexibilidade, reusabilidade e legibilidade. Para aceitar as refatorações, os autores utilizam

quatro técnicas, sendo que três delas são variações do algoritmo *Hill Climbing*[5] (subida de encosta). A primeira variação examina os vizinhos até encontrar uma solução de boa qualidade (*First-Ascendant Hill Climbing*). A segunda encontra a melhor de todas as soluções em um grupo de soluções (*Steepest-Ascendant Hill Climbing*). A terceira é uma variação da primeira, em que consecutivos recomeços são aplicados até se encontrar a melhor solução (*Multiple-Restart Hill Climbing*). Por fim, eles também utilizam a heurística *Simulated Anneling* (Kirkpatrick et al. 1983), que aceita soluções piores para escapar dos ótimos locais.

Seng et al. (2006) também apresentam uma abordagem para refatoração automática. A refatoração "Mover Métodos" foi utilizada e aplicada em um projeto Open Source. Eles utilizam uma estrutura própria para simulação e aplicação de refatorações. Novas refatorações são geradas a partir do cruzamento sucessivo entre as possíveis refatorações, sendo que as mutações que não atendem às pré e pós-condições são descartadas. A fim de analisar os resultados, os autores adotaram uma função de avaliação que utiliza valores de coesão, acoplamento e complexidade pela extração de métricas. Por fim, tem-se uma lista de possíveis refatorações, o que faz o processo de escolha e refatoração não ser totalmente automático. Os autores também se preocuparam em poder voltar ao estado original do projeto, dado os passos para gerar as transformações. A principal contribuição deste trabalho é a nova abordagem de SBSE proposta por eles e avaliada em um projeto Open Source.

Abdeen et al. (2009) criaram uma abordagem automática para movimentação de classes entre pacotes, baseando-se no princípio de coesão e acoplamento. Uma das parametrizações são valores para considerar a distância da modularização final para a inicial, diferenciando de outras abordagens que buscam a melhor modularização diretamente. Para gerar a reestruturação, pequenas perturbações são realizadas e a avaliação da qualidade das movimentações é feita através de métricas de pacotes e classes como, por exemplo, coesão, acoplamento e dependências cíclicas. A abordagem foi avaliada em quatro projetos open-source escolhidos por suas características distintas como número de classes, pacotes, dependências entre classes, dependências entre pacotes, dependências cíclicas, etc.

[5] Esse algoritmo consiste em um procedimento de busca local que elege um candidato, dentro de um conjunto de soluções (Lacerda e Carvalho 1999).

Ferber et al. (2009) desenvolveram uma abordagem automática avaliada por métricas para reduzir o acoplamento das classes aplicando movimentação dos então chamados *grupos de membros*. Esses grupos são formados por métodos e atributos das classes que podem ser movimentados sem alteração do comportamento. A avaliação dessa abordagem foi realizada em dois conhecidos projetos open-source (JEdit - http://sourceforge.net/projects/jedit/ e JMeter - http://jakarta.apache.org/jmeter/).

Uma representante da área de reengenharia é a abordagem de Tahvildari e Kontogiannis (2004). Eles desenvolveram um framework para sugestão de melhoria automática de design pela detecção de falhas de projeto. A heurística *Key classes* (Bauer e Karlsruhe 1999) foi utilizada para detectar essas falhas. As refatorações aplicadas foram feitas via transformações de metapadrões de Tahvildari (2004) e a avaliação destas transformações foi realizada com métricas para manutenibilidade, dadas por Laguë e April (1996). A abordagem realizou experimentos com quatro projetos open-source.

Outra representante da área de reengenharia é a abordagem de Moura (2009), nomeada ROOCS (*Reengineering Object Oriented Software to Components*). Essa abordagem tem o objetivo de reagrupar sistemas orientados a objetos em componentes. Para isso, a autora oferece um ferramental integrado a um ambiente de reutilização que visa reorganizar candidatos a componentes (i.e., pacotes). A reestruturação é realizada conforme o desenvolvimento orientado a componentes. Métricas que extraem informação sobre manutenibilidade e reusabilidade são utilizadas para atestar essa conformidade. A avaliação da abordagem foi realizada utilizando processos sistemáticos de experimentação sobre a ferramenta ArchTrace (Murta et al. 2006), do grupo de reutilização da COPPE/UFRJ.

2.6 – CONSIDERAÇÕES FINAIS

Refatoração de software é uma operação constante no contexto atual de desenvolvimento de software. Dentre as diversas utilidades da refatoração, a capacidade de melhorar a manutenção do código fonte de um software se destaca, tendo em vista que na grande maioria dos sistemas de software, a manutenção ocupa a maior parte de sua vida útil (Pressman 2004). Atualmente, as IDEs são as maiores contribuidoras para o ferramental de apoio às refatorações. Mesmo assim, como discutido anteriormente, determinadas refatorações não são triviais por diferentes razões. No intuito de diminuir erros humanos e aumentar a confiança dessas operações, uma atenção especial foi dada às refatorações que não

necessitam de dados subjetivos para serem aplicadas: as refatorações automatizáveis. Essas refatorações, no entanto, necessitam ser avaliadas de alguma forma e uma solução para isso é a utilização de métricas.

Como mostrado na Seção 2.5, existem muitas abordagens que buscam a melhoria de código fonte de forma automática. Mais especificamente, essas abordagens atacam problemas e questões das áreas de: reengenharia de software; *search-based software engineering*; buscas de maus cheiros e/ou antipadrões; aplicação de refatorações automatizáveis; e extração de métricas. Dentre esses trabalhos, não foi encontrada uma solução que pudesse facilmente ser incluída nos diversos cenários de desenvolvimento que usufruem dos benefícios básicos da gerência de configuração como um Sistema de Controle de Versões e um Sistema de Gerência de Construção. Também não foi encontrada uma solução que pudesse "aprender", ao longo de sucessivas aplicações de refatorações automatizáveis, quais têm maiores chances de melhorar determinada métrica e/ou atributo de qualidade, evitando assim possíveis desperdícios de recursos computacionais. Por fim, não foram encontradas abordagens explicitamente projetadas para acolher com facilidade novas métricas ou refatorações automatizáveis, e que fizessem uso do parque computacional ocioso do desenvolvimento, disponibilizando um monitoramento contínuo através de uma interface centralizada.

CAPÍTULO 3 – ABORDAGEM PEIXE-ESPADA

3.1 – INTRODUÇÃO

Como observado no Capítulo 2, existem diversos trabalhos com reconhecidas contribuições que tratam de problemas específicos dentro do universo de melhoria automática de software.

Este trabalho propõe uma abordagem que se distingue das demais desde seu propósito original, que é amenizar o envelhecimento precoce de software, via uma solução integrada ao cenário típico de desenvolvimento com GCS, e ser pouco intrusiva. Deste modo, propõe-se um framework arquitetado para permitir extensão independente de linguagens em refatorações automatizáveis e funções de avaliação por métricas, oferecendo uma solução de paralelismo com a utilização do parque computacional ocioso das máquinas de desenvolvimento. Além disso, o sistema se apoia em ferramentas de GC largamente utilizadas para controle de versões e gerenciamento de construção. Essa medida permite o fácil rastreamento das alterações e armazenagem de cada passo de refatoração, evitando problemas com dependências dos projetos ao executar nas distintas máquinas. Sua arquitetura também suportaria a expansão para outras ferramentas de GC para controle de versões.

A fim de atender a essas necessidades, é proposta a criação da abordagem Peixe-Espada. O Peixe-Espada foi concebido na forma de um sistema multiagente (SMA) distribuído para gerenciar máquinas ociosas, usando este poder computacional para executar melhorias relativas ao código fonte existente. Vale notar que a concepção do Peixe-Espada como um SMA teve como principal motivação facilitar a organização dos seus vários módulos na forma de agentes autônomos e distribuídos em diferentes computadores, viabilizando uma melhor separação dos interesses envolvidos na abordagem. A abordagem é guiada por atributos de qualidade para permitir que gerentes de desenvolvimento possam escolhê-los na melhoria de seus sistemas de software. Dentro dessa arquitetura multiagentes, um *agente orquestrador* é criado para cada atributo de qualidade. O objetivo principal do *agente orquestrador* é melhorar seu respectivo atributo de qualidade em um determinado projeto. Para isso, *agentes orquestradores* delegam tarefas aos *agentes trabalhadores*, cujas responsabilidades são: executar as operações necessárias para obter o código fonte; detectar sintomas de refatoração; executar as refatorações no código fonte; e avaliar as refatorações aplicadas.

Os *agentes orquestradores* executam em um servidor remoto, enquanto os *agentes trabalhadores* executam nos computadores dos desenvolvedores durante o tempo ocioso, geralmente à noite. A Figura 7 mostra a visão geral desta abordagem com o fluxo básico de trabalho para se obter um ciclo de rejuvenescimento bem sucedido. As etapas para completar um *ciclo de rejuvenescimento* são: (1) Inicialmente, o *agente trabalhador* tem que fazer *checkout* do código fonte, criando assim a *configuração atual* em seu *espaço de trabalho* (do inglês, *workspace)*; (2) feito o *checkout*, o agente trabalhador pede ao *agente orquestrador* uma sequência de refatorações e as realiza na configuração atual, gerando *configurações derivadas*; (3) posteriormente, as *configurações derivadas* são avaliadas e a melhor delas (em relação ao atributo de qualidade alvo do *agente orquestrador*) é eleita; e (4) se a *configuração eleita* for melhor do que a *configuração atual (ciclo de sucesso)*, ela passa a ser a *configuração atual* e o ciclo recomeça. Considera-se que um *ciclo de sucesso* é atingido quando, no final de um ciclo, a *configuração eleita* torna-se melhor que a *configuração atual* em relação ao atributo de qualidade escolhido inicialmente. Caso contrário, é considerado um *ciclo de fracasso* e a *configuração eleita* é descartada.

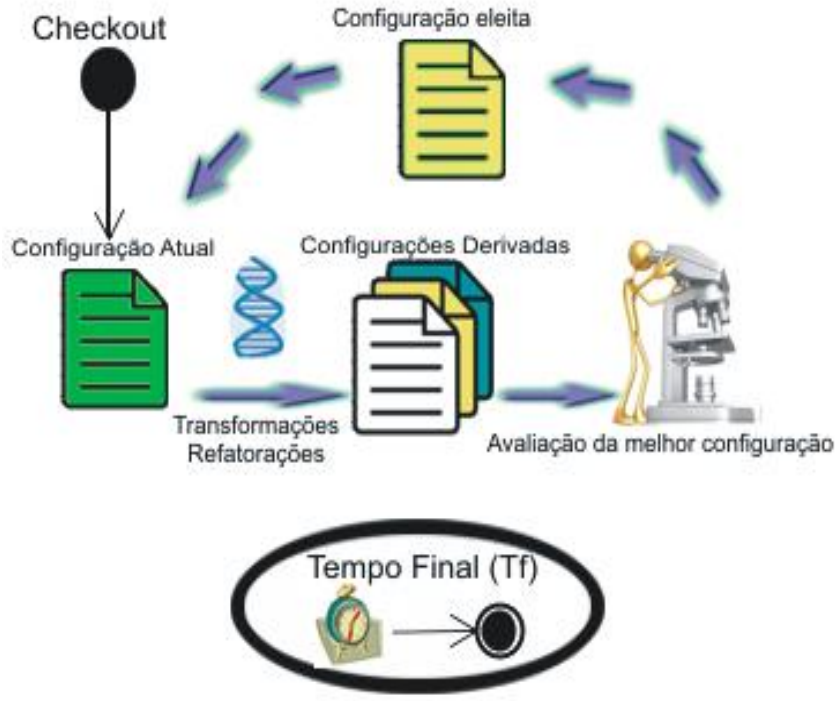


Figura 7: Ciclo de rejuvenescimento proposto pela abordagem Peixe-Espada.

No final de cada ciclo, os *agentes trabalhadores* informam ao *agente orquestrador* os resultados (ou seja, se a refatoração melhorou, piorou ou não mudou o atributo de qualidade). O *agente orquestrador*, por sua vez, armazena este conhecimento para apoiar o SMA na

escolha das refatorações futuras. É importante notar que este conhecimento representa a capacidade de aprendizagem do SMA. Ele permite priorizar as refatorações que têm maior probabilidade de melhorar um determinado atributo de qualidade, diminuindo a geração de configurações derivadas inúteis nas execuções subsequentes.

Se nenhuma das refatorações sugeridas melhorar a *configuração atual*, o ciclo de trabalho termina. Além disso, se terminar o período de ociosidade do computador com o sistema em execução, os *agentes trabalhadores* geram relatórios das alterações realizadas, as entregam ao *agente orquestrador* e encerram seus trabalhos. Todas as alterações bem sucedidas são armazenadas em um ramo do repositório do SCV. Assim, a equipe de desenvolvimento pode incorporar as mudanças que julgar pertinente. As próximas seções detalham a estrutura do sistema multiagente (Seção 3.2), as funções e objetivos de cada tipo de agente (Seção 3.3), como os agentes alcançam seus objetivos (Seção 3.4), a abordagem de refatoração (Seção 3.5), e algumas considerações finais (Seção 3.6).

3.2 – ESTRUTURA DO SISTEMA MULTIAGENTES

Como discutido anteriormente, o *agente orquestrador* é executado em um servidor remoto e os *agentes trabalhadores* nos computadores dos desenvolvedores. O *agente orquestrador* está vinculado a um atributo de qualidade em um projeto de software. Além disso, cada *agente trabalhador* é vinculado a um *agente orquestrador* ao qual deve obedecer. A estrutura do Peixe-Espada também permite que cada *agente trabalhador* seja criado em computadores diferentes, respeitando os respectivos tempos de ociosidade.

Um dos aspectos chave do Peixe-Espada é a forma como os agentes colaboram no sistema distribuído para alcançarem seus objetivos. Devido aos recursos operacionais (processamento, IO, memória, etc.) e à complexidade das etapas envolvidas no rejuvenescimento de software, a utilização de processamento paralelo é fundamental. O Peixe-Espada permite três diferentes estratégias de paralelismo: (1) melhorar diferentes projetos em paralelo; (2) melhorar distintos atributos de qualidade em paralelo em cada projeto; e (3) acelerar a melhoria de um atributo de qualidade de um projeto específico.

Caso 1: Normalmente, uma equipe de desenvolvimento de software desenvolve diferentes produtos. Com essa estratégia, é possível melhorar diferentes projetos ao mesmo tempo, utilizando parte do poder computacional ocioso para cada projeto. Por exemplo, em uma configuração com 5 computadores de desenvolvedores, até 5 projetos podem ser

melhorados em paralelo. Cada projeto terá um *agente orquestrador* executando individualmente no servidor, com pelo menos um grupo de *agentes trabalhadores* em execução em cada um desses 5 computadores.

Caso 2: Por outro lado, um projeto pode ser melhorado em termos de diferentes atributos de qualidade. Nessa estratégia, diferentes *agentes trabalhadores* podem trabalhar em paralelo para melhorar diferentes atributos de qualidade de um mesmo projeto. Em outras palavras, essa estratégia dedica parte do poder computacional ocioso para melhorar cada atributo de qualidade de um determinado projeto. Por exemplo, em um ambiente com 5 computadores de desenvolvedores, até 5 *agentes orquestradores* podem ser configurados para melhorar 5 atributos de qualidade diferentes de um mesmo projeto, com pelo menos um grupo de *agentes trabalhadores* em cada máquina operando por *agente orquestrador*.

Caso 3: Finalmente, diferentes *agentes trabalhadores* podem trabalhar para o mesmo *agente orquestrador*. Nesta estratégia, cada grupo de *agentes trabalhadores* recebe uma sequência distinta de refatoração, mas todas as sequências têm o objetivo de melhorar o mesmo atributo de qualidade do projeto para o qual o *agente orquestrador* foi cadastrado. No final, as melhorias de cada grupo são disponibilizadas em ramos separados. Por exemplo, em um ambiente com 5 computadores de desenvolvedores, um *agente orquestrador* pode ser configurado para melhorar um atributo de qualidade de um projeto específico e ter 5 grupos de *agentes trabalhadores* à sua disposição.

É importante notar que estas estratégias podem ser combinadas para atingir os diferentes propósitos de rejuvenescimento de software. Por exemplo, uma empresa com 32 desenvolvedores, responsáveis por 4 produtos principais, pode configurar o Peixe-Espada com 8 *agentes orquestradores*. Cada par de *agente orquestrador* pode ser responsável por melhorar dois atributos de qualidade diferentes em um mesmo projeto. Além disso, quatro grupos de *agentes trabalhadores* podem servir a cada *agente orquestrador*. Desta forma, 32 grupos de *agentes trabalhadores* estariam operando em paralelo (cada grupo alocado a um computador) para melhorar dois atributos de qualidade de cada um dos 4 projetos.

Supondo que o Peixe-Espada possua Efetividade e Reusabilidade cadastrados como atributos de qualidade, os *agentes orquestradores* podem gerar sugestões para sequências de refatoração, como exemplificado na Tabela 5. A partir dessas sequências de refatoração, é possível mostrar um exemplo com dados sintéticos para o cenário acima citado (Tabela 6).

Tabela 5: Refatorações e exemplo de sequência de refatorações.

Refatorações	Sequências Sugeridas
R1: PullUpMethods	S1: {R1, R2, R3, R4}
R2: EncapsulateFields	S2: {R2, R3, R4, R1}
R3: PullUpFields	S3: {R3, R4, R1, R2}
R4: PushDowMethods	S4: {R4, R1, R2, R3}

A sigla AO representa um *agente orquestrador*, então AO1 representa o *agente orquestrador* número 1 e assim por diante. Já a sigla AT representa um grupo de *agentes trabalhadores*, então AT2.1 seria o grupo número 2 registrado para servir o AO1. A última coluna desta tabela representa 4 grupos de *agentes trabalhadores* distintos a serviço de cada um dos 8 *agentes orquestradores*. Sendo assim, (M1, AT2.1, S1) mostra que na máquina 1 (M1) encontra-se executando o grupo número 2, que está servindo ao *agente orquestrador* número 1 (AT2.1), cuja sequência de refatorações sugerida é a S1.

Tabela 6: Exemplo de paralelismo misto.

Ag. O.	Proj.	Atr. Qualidade	{ (Máquina, Agente Trabalhador, Sequência de Refatoração) , … }
AO1	P1	Efetividade	{ (M1, AT1.1, S1), (M2, AT2.1, S2), (M3, AT3.1, S3), (M4, AT4.1, S4) }
AO2	P1	Reusabilidade	{ (M5, AT5.2, S1), (M6, AT6.2, S2), (M7, AT7.2, S3), (M8, AT8.2, S4) }
AO3	P2	Efetividade	{ (M9, AT9.3, S1), (M10, AT10.3, S2), (M11, AT11.3, S3), (M12, AT12.3, S4) }
AO4	P2	Reusabilidade	{ (M13, AT13.4, S1), (M14, AT14.4, S2), (M15, AT15.4, S3), (M16, AT16.4, S4) }
AO5	P3	Efetividade	{ (M17, AT17.5, S1), (M18, AT18.5, S2), (M19, AT19.5, S3), (M20, AT20.5, S4) }
AO6	P3	Reusabilidade	{ (M21, AT21.6, S1), (M22, AT22.6, S2), (M23, AT23.6, S3), (M24, AT24.6, S4) }
AO7	P4	Efetividade	{ (M25, AT25.7, S1), (M26, AT26.7, S2), (M27, AT27.7, S3), (M28, AT28.7, S4) }
AO8	P4	Reusabilidade	{ (M29, AT29.8, S1), (M30, AT30.8, S2), (M31, AT31.8, S3), (M32, AT32.8, S4) }

Como pode ser observado na Tabela 6, o Projeto P2 está sendo refatorado por dois *agentes orquestradores* AO3 e AO4, que atuam respectivamente sobre os atributos de qualidade Efetividade e Reusabilidade. Para isso, o *agente orquestrador* AO3 conta com os grupos de *agentes trabalhadores* AT9, AT10, AT11 e AT12 em cada uma das 4 máquinas M9, M10, M11 e M12, respectivamente. Para melhorar Efetividade, o agente AO3 sugeriu a

sequência de refatorações S1 para o AT9, S2 para o AT10, S3 para o AT11 e S4 para o AT12. O Agente AO4 atua paralelamente de forma similar nas máquinas entre M13 e M16. No final, AO3 e AO4 terão relatórios das refatorações juntamente com o conhecimento adquirido relacionando refatorações com seus atributos de qualidade e a referências para os ramos criados no repositório de versões contendo as alterações bem sucedidas, do mesmo projeto P2 para atributos de qualidade diferentes.

Todo agente do Peixe-Espada desempenha papéis e é composto por planos e objetivos. Para alcançar um objetivo específico, os planos são compostos por ações. As próximas seções mostram em detalhes o sistema multiagente proposto, explicando cada plano, como eles estão organizados em ações, e como ocorre a comunicação entre os agentes.

3.3 – PAPÉIS E OBJETIVOS DOS AGENTES

Como citado anteriormente, o Peixe-Espada é composto por *agentes orquestradores* e *agentes trabalhadores*. No entanto, os *agentes trabalhadores*, que compõem um grupo, são especializados em quatro papéis distintos: *gerente local*, *medidor*, *refatorador* e *gerente de configuração*. Assim, cada papel de *agente trabalhador* é responsável por um grupo pré-determinado de ações coesas, respectivamente, a coordenação local dos demais agentes nas máquinas dos desenvolvedores, a medição dos atributos de qualidade, a aplicação de refatorações, e a interação com o repositório do SCV. Deste ponto em diante, um *agente trabalhador* que desempenha um determinado papel será chamado apenas pelo seu papel, por exemplo, o *agente trabalhador* desempenhando papel de medidor, será chamado apenas de *agente medidor*.

Como mostrado na Figura 8, os agentes seguem uma hierarquia pré-definida. O *gerente local* é o único que pode se comunicar com o *agente orquestrador*. Os papéis e objetivos são descritos a seguir.

O ***agente orquestrador*** trabalha como gerente global de todo o SMA. Os *agentes orquestradores* estão localizados no servidor remoto, e seus objetivos são melhorar o atributo de qualidade previamente associado a eles em um projeto. Eles também gerenciam o conhecimento sobre a relação entre refatorações e atributos de qualidade. Isto permite que os *agentes orquestradores* sugiram sequências de refatoração mais adequadas para os *agentes trabalhadores*, pelo fato de priorizarem as refatorações mais bem sucedidas na melhoria de um determinado atributo qualidade.

Em contraste, o ***agente gerente local*** é a interface entre outros agentes e os *agentes orquestradores*. Seu objetivo é manter os outros *agentes trabalhadores* colaborando harmoniosamente em uma máquina do desenvolvimento.

O ***agente medidor*** é responsável por calcular o atributo de qualidade provido pelo agente orquestrador nas diversas configurações dos projetos. Os *agentes medidores* são empregados antes e depois de uma refatoração para permitir a decisão em manter ou descartar os resultados da refatoração.

O ***agente refatorador*** é especializado na busca de *sintomas* relacionados a cada refatoração automática e aplicação da refatoração. Seu objetivo é aplicar refatorações sem quebrar[6] o código fonte do projeto.

Finalmente, o ***agente gerente de configuração*** é responsável por interagir com o repositório do SCV e atividades afins. O objetivo deste agente é realizar com sucesso ações que incluem *checkin*, *checkout*, criação de ramo e reversão de modificações locais.

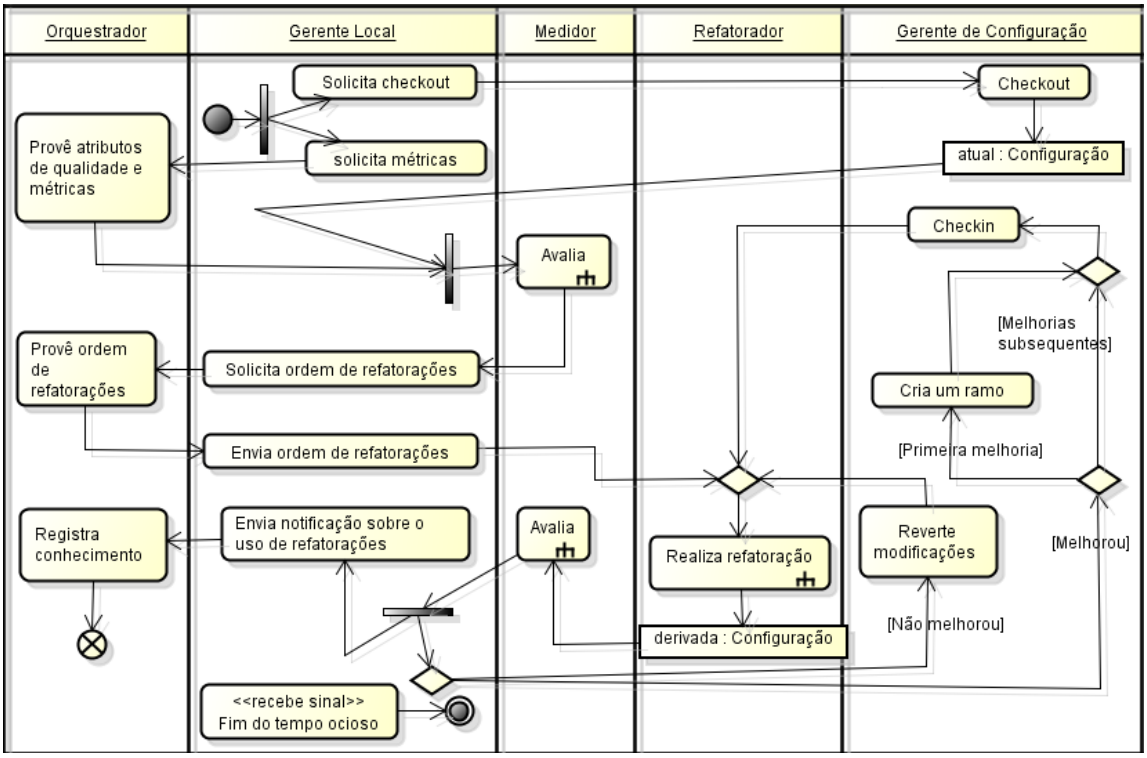


Figura 8: Diagrama de atividades UML mostrando a interação entre os agentes.

[6] Quebrar o código fonte nesta abordagem é não compilar ou não passar nos testes automatizados.

3.4 – PLANOS E AÇÕES DOS AGENTES

Segundo a arquitetura BDI (Rao e Georgeff 1995), os agentes têm crenças, desejos e intenções. Eles executam ações agrupadas como planos para atingir suas intenções. Além disso, cada agente do Peixe-Espada desempenha um papel que define tarefas que os agentes realizam no sistema (Zambonelli et al. 2001). A Figura 8 mostra uma visão geral de cada ação descrita nesta seção. Essa figura apresenta a interação entre os cinco papéis de agentes através de um diagrama de atividade UML, com o propósito de fornecer uma ideia geral do ciclo de trabalho e mostrar a hierarquia imposta aos agentes Peixe-Espada.

Em um ciclo normal de trabalho, o ***agente orquestrador*** executa as seguintes ações: (1) registrar os *agentes trabalhadores*; (2) fornecer a URL do repositório de código fonte para *checkout*; (3) fornecer as métricas para calcular um atributo de qualidade; (4) fornecer a ordem de execução das refatorações cadastradas; e (5) armazenar informações sobre melhora, piora ou não alteração do atributo de qualidade de cada refatoração realizada.

O ***agente gerente local*** é responsável pelas seguintes ações: (1) solicitar ao *agente gerente de configuração* o *checkout* da *configuração atual*; (2) solicitar ao *agente orquestrador* o fornecimento das métricas para calcular o atributo de qualidade que deve ser melhorado; (3) solicitar ao *agente medidor* o valor de um atributo de qualidade da *configuração atual*; (4) solicitar ao *agente orquestrador* uma ordem de refatoração, (5) solicitar ao *agente refatorador* para seguir uma ordem de refatorações; (6) solicitar ao *agente medidor* os valores do atributo de qualidade das *configurações derivadas*; (7) informar ao *agente orquestrador* sobre melhora, piora ou não alteração do atributo de qualidade para cada refatoração; (8) solicitar ao *agente orquestrador* a URL do ramo onde serão integradas as refatorações bem sucedidas no projeto; e (9) solicitar ao *agente gerente de configuração* o *checkin* da *configuração escolhida* em um ramo específico.

O ***agente medidor*** também segue um plano em resposta às solicitações do *agente gerente local*. Este plano é composto de cinco ações específicas (Figura 9): (1) extrair os valores das métricas de uma configuração específica; (2) normalizar os valores das métricas previamente extraídas; (3) calcular os atributos de qualidade dessas medidas; (4) comparar a melhora da configuração de acordo com o atributo de qualidade medido; e (5) gerar informações sobre a aplicação das refatorações. A primeira ação de avaliação, apresentada na Figura 8, representa a medição da *configuração atual* e a normalização dos valores das

métricas extraídas. A segunda ação de avaliação envolve a medição das configurações derivadas e comparação dos valores de atributos de qualidade normalizados de ambas as configurações, esses fluxos de avaliação encontram-se na Figura 9.

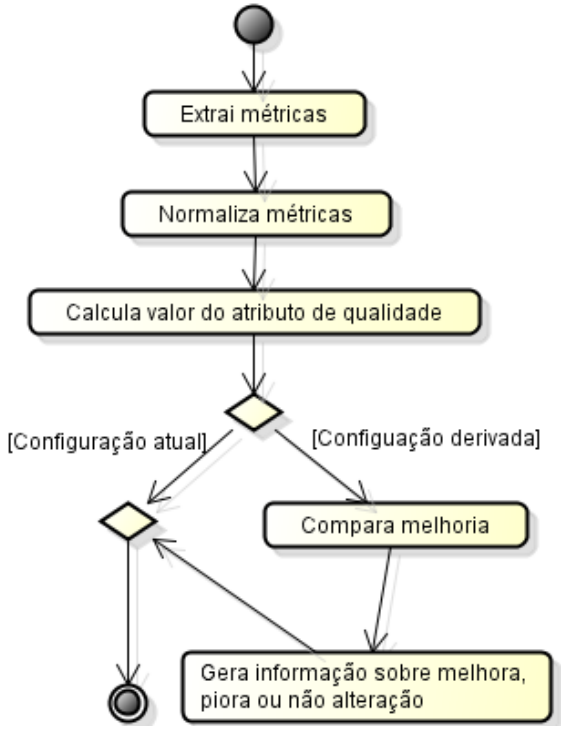


Figura 9: Ação "Avalia" expandida.

O processo de normalização consiste em dividir todos os valores de métricas pelos valores da primeira medição, dessa forma têm-se as mudanças proporcionais de cada métrica em cada medição relativas à medição inicial (configuração original). Por exemplo, dado que na primeira medição foram encontrados os valores {1; 2; 3} para as métricas A, B e C, respectivamente, e na segunda medição os valores {1; 4; 1,5} para a mesma sequência de métricas, ao realizar a normalização os valores da primeira medição seriam {1; 1; 1} e o da segunda {1; 2; 0,5} ({1/1; 4/2; 1,5/3}). A Tabela 7 mostra um exemplo de medições antes de normalização e a Tabela 8 depois do processo de normalização. Atualmente, o *agente medidor* está trabalhando com métricas e atributos de qualidade definidos por Bansiya e Davis (2002) (Seção 2.4.2), as normalizações aqui apresentadas também seguem esse trabalho. No entanto, a arquitetura do Peixe-Espada permite ampliações futuras para acomodar métricas e atributos de qualidade adicionais, se necessário.

A Tabela 7 mostra exemplos de medições não normalizadas para calcular o atributo de qualidade Extensibilidade, conforme a fórmula +0.50*ANA - 0.50*DCC + 0.50*MFA +

0.50*NOP (Tabela 4). Ao realizar a normalização (Tabela 8), apenas os valores das métricas são normalizados, já os valores do atributo de qualidade são recalculados.

Tabela 7: Valores de métricas não normalizadas.

#	ANA	DCC	MFA	NOP	Extensibilidade
1	1,801905	2,009365	0,739524	17	8,766032
2	1,801905	2,009365	0,739532	17	8,766036
3	1,801905	2,011446	0,739507	18	9,264982
4	1,801905	2,012487	0,739518	19	9,764468

Tabela 8: Valores de métricas normalizadas.

#	ANA	DCC	MFA	NOP	Extensibilidade
1	1	1	1	1	1
2	1	1	1,000011	1	1,000005
3	1	1,001036	0,999977	1,058824	1,028882
4	1	1,001554	0,999992	1,117647	1,058043

O ***agente refatorador*** emprega as seguintes ações durante o seu trabalho: (1) solicitar a lista ordenada de refatorações ao *agente gerente local*; (2) buscar *sintomas de refatoração* para cada refatoração sugerida sobre uma determinada configuração; e (3) aplicar cada refatoração. Como algumas refatorações não são automatizáveis, esse agente está trabalhando atualmente com as refatorações apresentadas na Tabela 1. Entretanto, a arquitetura do *agente refatorador* também permite futuras ampliações para acomodar refatorações adicionais, se necessário. Como discutido anteriormente, este agente divide a refatoração em duas fases principais: *busca* e *aplicação*. A fase de *busca* procura identificar elementos que possam ser refatorados. Os resultados desta fase são conjuntos de elementos de programação (ou seja, pacotes, classes, métodos, atributos, etc.) que são os insumos para uma refatoração em particular. A fase de *aplicação* consiste em realizar as transformações sobre os elementos do código fonte indicados na fase anterior. É importante notar que para ser um sintoma de refatoração os elementos necessitam respeitar pré-condições específicas de cada refatoração (Tabela 1).

Finalmente, o ***agente gerente de configuração*** realiza as seguintes ações para atingir seu objetivo: (1) solicitar ao *agente gerente local* a URL do projeto no repositório de código fonte; (2) fazer *checkout* da *configuração atual* e gerenciar os *espaços de trabalho*; (3) solicitar ao *agente gerente local* a URL do ramo para fazer *checkin* das alterações bem sucedidas no projeto; (4) criar um novo ramo; (5) fazer *checkin* da *configuração escolhida*

para o ramo; e (6) reverter modificações em razão de uma refatoração que não melhorou o atributo de qualidade em relação à *configuração atual*, quando há quebra de código ou quando solicitado pelo *agente gerente local*.

3.5 – ABORDAGEM DE REFATORAÇÃO

Como descrito anteriormente, o *agente refatorador* se baseia nas refatorações automatizáveis previamente implementadas no Peixe-Espada. Aprofundando no seu plano de ações, nota-se que sua tarefa principal é a refatoração (Figura 8). Essa tarefa complexa é descrita nesta seção através da busca de *sintomas*, aplicação e avaliação da refatoração, seguindo as particularidades do Peixe-Espada.

3.5.1 – APLICAÇÃO DA REFATORAÇÃO COMPLETA

Como descrito na Seção 2.3, os *sintomas de refatoração* são elementos de código fonte que respeitam pré-condições para cada refatoração. Além disso, a refatoração no Peixe-Espada é feita pela busca e aplicação de *sintomas de refatoração*. No entanto, a fase de aplicação foi dividida em mais duas etapas: a busca por *alternativas de refatoração* e *aplicação do conjunto de alternativas* já solucionadas. As *alternativas de refatoração* são encontradas quando existe mais de uma forma física de aplicar uma *refatoração mínima*. Uma *refatoração mínima*, por sua vez, é caracterizada pela refatoração dos elementos encontrados em um único *sintoma de refatoração*. O processo completo é detalhado a seguir:

(1) Busca por ***sintomas de refatoração***: são identificados elementos de programação que respeitam as pré-condições específicas de uma dada refatoração (Tabela 1). Como já discutido, a busca dos sintomas pode ser realizada via consultas na AST do projeto ou via metaprogramação (Seção 2.3).

(2) Busca por ***alternativas de refatoração***: as *alternativas de refatoração* são identificadas para que no momento de aplicação da *refatoração mínima* não ocorram conflitos que impeçam a refatoração, por existir mais de uma forma física de se realizar a *refatoração mínima*. Nesse momento também é verificado se alguma *alternativa de refatoração* não é aplicável ou se ela altera uma funcionalidade do sistema, nesses casos, essas alternativas são descartadas. Para as *alternativas* restantes, é gerado um conjunto de transformações, que indicam como e onde alterar o código fonte para aplicar cada *alternativa*. Uma *alternativa de refatoração* pode gerar mais alternativas de refatoração. Quando isto ocorre, mais de um conjunto de transformações é

gerado. Quando uma *refatoração mínima* não tem nenhuma *alternativa de refatoração***,** considera-se que o conjunto de transformações é a *refatoração mínima* em si.

(3) Aplica cada ***conjunto de transformações***: cada *conjunto de transformações* é aplicado separadamente no código da seguinte forma:
   a. Tenta-se aplicar um conjunto de transformações.
   b. Se novas *alternativas de refatorações* surgirem após a aplicação, volta-se ao passo 2, sem descartar o conjunto de transformações atual.
   c. Se o código fonte estiver quebrado, o conjunto atual é descartado e passa-se para o passo 3.e.
   d. Caso não haja nenhuma *alternativa de refatoração* e o código fonte não esteja quebrado, calculam-se os valores dos atributos de qualidade pré-determinados. Nesse momento é dito que os conjuntos de transformação estão completos, pois não há mais alternativas de refatoração. Os conjuntos completos com seus respectivos valores dos atributos de qualidade são armazenados para serem avaliados posteriormente.
   e. O código fonte é revertido para o estado anterior ao do passo 3.a. O código fonte é sempre revertido aqui, pois cada conjunto de transformações possui os elementos necessários para a aplicação da refatoração, com essa estrutura, um único espaço de trabalho é suficiente para gerar todas as refatorações.

(4) Se existirem conjuntos de transformações a serem aplicados e calculados, volta-se ao passo 3.a.

(5) Aceita o melhor ***conjunto completo de transformações***: após a aplicação e cálculo do atributo de qualidade de todos os conjuntos completos de transformações, escolhe-se o conjunto com maior valor de atributo de qualidade para que a refatoração seja reaplicada no código fonte original. Se o atributo de qualidade da *configuração eleita* não for melhor que o atributo de qualidade da *configuração atual*, a refatoração é descartada. As refatorações aceitas são disponibilizadas em um ramo (*branch*) a parte da linha de desenvolvimento principal do software.

A Figura 10 mostra sintomas de duas refatorações e as possíveis resoluções de *alternativas de refatorações*. Na busca de sintomas para *PullUpMethods*, foi encontrado o método “anda”, e para *EncapsulateFields*, o atributo “nome” foi encontrado.

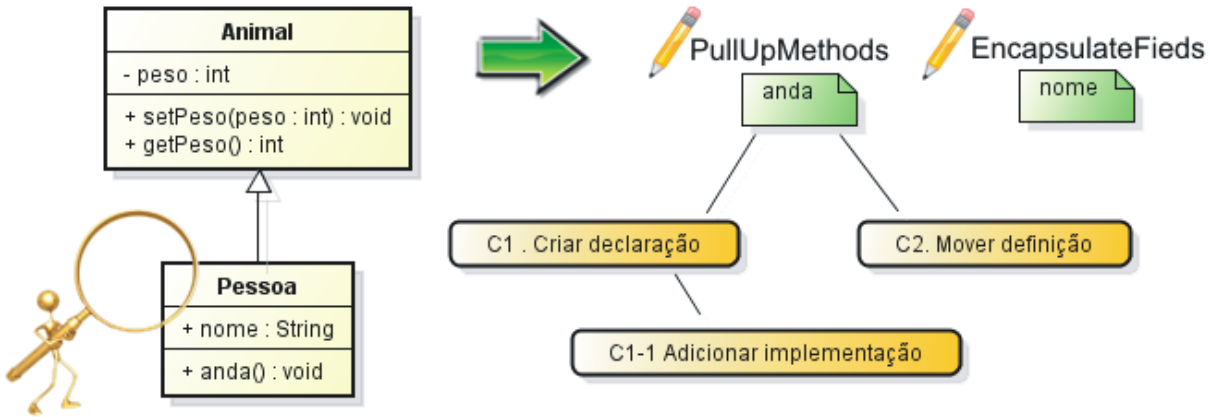


Figura 10: Exemplo de aplicação da abordagem de refatoração.

Ainda na Figura 10, o processo completo para a refatoração utilizando *PullUpMethods* para o método "andar" é feito da seguinte forma:

- Duas alternativas de refatorações foram encontradas. Logo, dois conjuntos de transformações foram gerados, C1 e C2.
- Ao tentar aplicar C1, verificou-se que uma nova *alternativa de refatoração* surgiu. Então, o conjunto C1 foi usado para formar um novo conjunto, o C1-1 com as duas transformações.
- Como nenhuma outra *alternativa de refatoração* foi encontrada após a aplicação das transformações do conjunto C1-1, é feita uma verificação para checar se as alterações realizadas pelo conjunto C1-1 não quebraram o código. Caso não tenham quebrado: (1) o atributo de qualidade do conjunto é calculado; (2) o conjunto completo é armazenado em uma estrutura para análise posterior; (3) o código é revertido para um momento anterior à aplicação das resoluções do conjunto C1.
- Após a aplicação da transformação do conjunto C2, nenhum conflito novo foi encontrado. Então, o conjunto é considerado como completo e deverá ocorrer a verificação para detectar se o código quebrou. Caso não esteja quebrado: (1) o atributo de qualidade é calculado; (2) o conjunto completo é armazenado; e (3) o código é revertido.
- Com todos os conjuntos completos, busca-se o conjunto com maior valor de atributo de qualidade gerando a *configuração eleita*. Se o atributo de qualidade da *configuração eleita* não for melhor que o atributo de qualidade da *configuração atual*, a refatoração é descartada. Caso *contrário, o conj*unto escolhido é reaplicado ao c*ódigo original, rea*lizando assim um *ciclo de sucesso*.

A aplicação *de EncapsulateFields* não possui *alternativas de refatoração*, mesmo assim cada *refatoração mínima* é aplicada e verificada isoladamente para conferir se o código fonte não está quebrando.

3.6 – CONSIDERAÇÕES FINAIS

A abordagem Peixe-Espada foi criada para amenizar o envelhecimento precoce de software, oferecendo uma solução abrangente que visa atender às seguintes necessidades:

- Monitoramento contínuo do código fonte.
- Processamento distribuído e paralelo.
- Aplicação de refatorações automatizáveis.
- Avaliação da aplicação de refatorações via métricas e atributos de qualidade.
- Utilização do parque computacional do desenvolvimento ocioso.
- Baixo nível de intrusão, evitando atrapalhar o dia a dia de desenvolvimento.

Além disso, a estrutura do Peixe-Espada é flexível, visando facilitar extensões, como a implementação de novas refatorações, métricas, atributos de qualidade, entre outros. Todos esses requisitos tornam o Peixe-Espada um framework robusto e passível de experimentações em termos de tecnologias[7] e estratégias[8], indiciando diferenciais positivos para a abordagem Peixe-Espada.

[7] Extensões para novas tecnologias de SCV, por exemplo, assim como criação de novas refatorações e medições.

[8] Estratégias de paralelismos, soluções distribuídas e diversas possibilidades de experimentação.

CAPÍTULO 4 – PROTÓTIPO

4.1 – INTRODUÇÃO

A abordagem Peixe-Espada foi implementada em dois módulos: o Peixe-Espada Cliente (PEC) e o Peixe-Espada Servidor (PES). O PES encontra-se inserido no ambiente Oceano. O Oceano é uma aplicação web desenvolvida para fornecer soluções de interesses relacionados à GC, como cadastros de projetos de software, cadastro de itens de configuração (IC) e conectores para sistemas de controle de versão. Esse ambiente também engloba outras funcionalidades para evolução controlada de software, como coleta de métrica, composição de atributos de qualidade e outras abordagens de propósitos acadêmicos. O PES é o centralizador das informações que o PEC necessita, ou seja, todos os acessos ao banco de dados, cadastros e *agentes orquestradores*, algoritmos de coleta de métricas, entre outros, encontram-se disponibilizados via interface do PES. Já o PEC contém a implementação dos *agentes trabalhadores* e os algoritmos de busca e aplicação de refatorações. O PEC é uma implementação desktop que utiliza a tecnologia *Java Web Start* (Marinilli 2001). Dessa maneira, o esforço de instalação é minimizado, visto que a partir de um link disponibilizado no PES, o PEC é atualizado e executado.

Em termos gerais, as atividades necessárias à execução do sistema como um todo são demonstradas na Figura 11, baseando-se nos seguintes passos: (1) a qualquer momento, em uma máquina do ambiente de desenvolvimento, um gerente da equipe pode acessar o PES e realizar o cadastro de um *agente orquestrador*[9] informando um projeto de software e o atributo de qualidade que ele pretende melhorar; (2) com o *agente orquestrador* cadastrado, o gerente pode acessar o link que contém a implementação dos agentes trabalhadores, para realizar a instalação da aplicação *Web Start*. Ao clicar no link, o processo de instalação verifica automaticamente atualizações do PEC. A instalação ocorre se o PEC não estiver instalado, contudo, se o PEC instalado estiver em uma versão desatualizada em relação ao servidor, a versão é atualizada. Em ambos os casos a aplicação é executada; (3) com o PEC instalado, o gerente pode realizar o agendamento do período de ociosidade da máquina em que o PEC estará executando. Para isso, o gerente deve informar, o *login/senha* do Oceano, o *agente orquestrador* para o qual os agentes trabalhadores deverão servir e a data de início e

[9] O cadastro do agente orquestrador só é necessário se o mesmo não existir.

término do período de ociosidade da máquina. Até o passo 3 o sistema necessita de interação. Os passos 4, 5 e 6 representam respectivamente os *checkouts*, diálogos entre os *agentes trabalhadores* e *agentes orquestradores*, e *checkins* de refatorações bem sucedidas juntamente com seus relatórios, como já discutido na Seção 3.2 e na Seção 3.3.

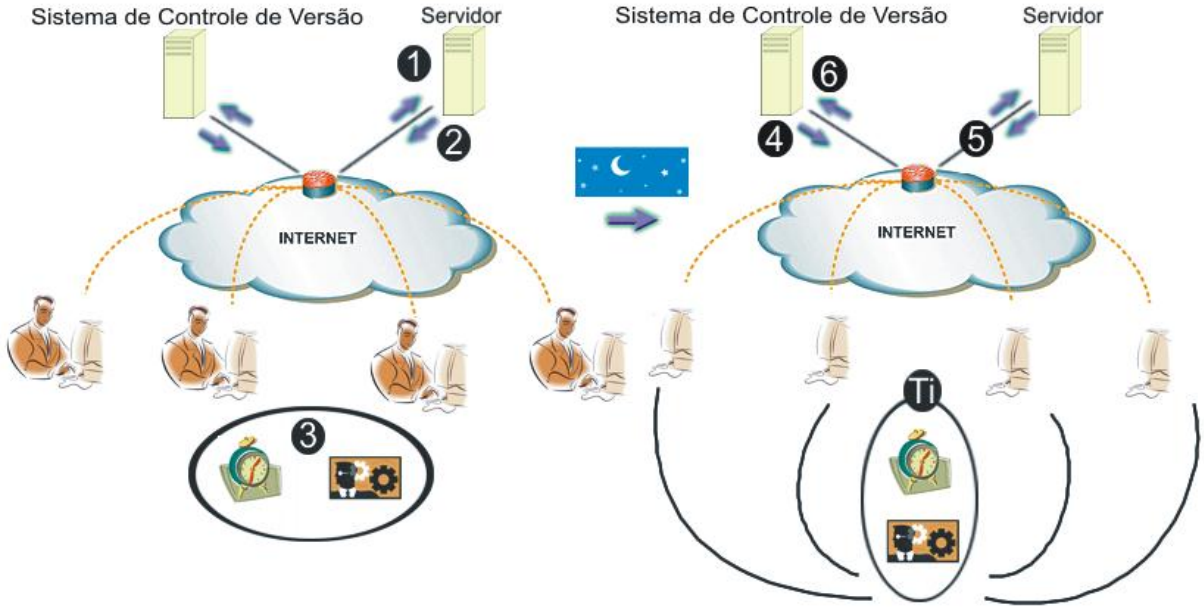


Figura 11: Fluxo geral de execução da abordagem.

Ainda na Figura 11, as linhas tracejadas simbolizam uma topologia de rede qualquer para acesso ao servidor e ao SCV. Já as setas contínuas do ambiente sem desenvolvedores (e.g., à noite), simbolizam os agentes (ícone de pessoa com engrenagens) alocados em cada uma das máquinas iniciando seus trabalhos no tempo inicial (Ti).

As principais características da arquitetura do Oceano são marcadas pela utilização das seguintes tecnologias nas diferentes camadas internas: especificação JPA 1.0 (Yang 2010) para modelos de persistência na camada de acesso a dados, utilizando o padrão de projetos DAO (Nock 2003); JSF 1.2 (Geary e Horstmann 2004) integrado com *Facelets* (Aranda e Wadia 2008) e *RichFaces* (Katz 2008) na camada de visualização; o padrão de projeto *Application Service* (Alur et al. 2003) para a implementação das regras de negócios em uma camada interna separada; e AspectJ (Laddad 2009) para modularização de alguns interesses ortogonais como transações com banco de dados, buscas genéricas, entre outros (Santos 2008).

Este capítulo apresenta o protótipo em detalhes, com exibição das telas e explicações de como a abordagem é automatizada. A arquitetura do protótipo é apresentada na Seção 4.2.

A Seção 4.3 mostra a realização do cadastro das dependências para um *agente orquestrador*. O cadastro do *agente orquestrador* propriamente dito é discutido na Seção 4.4. A Seção 4.5 apresenta a execução do Peixe-Espada nas máquinas de desenvolvimento. O monitoramento dos *agentes trabalhadores* e seus respectivos *orquestradores* é mostrado na Seção 4.6 e, por fim, a Seção 4.7 faz as considerações finais.

4.2 – ARQUITETURA DO PROTÓTIPO

Tendo em vista que o Peixe-Espada visa amenizar o envelhecimento precoce de software, é interessante que o cliente usufrua do protótipo sem maiores dificuldades, garantindo uma boa interação com o sistema e, consequentemente, facilitando o monitoramento contínuo. O desenvolvimento da interface com o usuário foi realizado atendendo às principais partes envolvidas no desenvolvimento: o desenvolvedor, que verá seu código sendo alterado, e o gerente, que necessita monitorar e melhorar alguns atributos de qualidade de seus sistemas de software. No desenvolvimento do protótipo, também foi considerado o ponto de vista científico, para viabilizar a repetição dos experimentos e extensibilidade da abordagem.

A Figura 12 mostra o diagrama resumido de classes conceituais do Oceano com foco nas entidades utilizadas pelo PES. Os dados do Oceano e PES são quase todos armazenados em um banco de dados relacional PostgreSQL (Inc et al. 2002). Informações não relevantes até então, como, por exemplo, quais foram os *agentes trabalhadores*, o tempo de trabalho de cada um, o aproveitamento de cada um, entre outras, não são armazenadas. Ainda nesse diagrama, é possível notar que informações básicas são persistidas para métricas e refatorações. Essas entidades são extensíveis via programação, ou seja, para a abordagem acolher uma nova métrica e/ou refatoração, é necessário implementar esses recursos e incluir algumas informações no banco de dados. O mesmo ocorre para atributos de qualidade (*QualityAttribute*). Classes como as que mapeiam projetos de software (*SoftwareProject*), itens de configuração (*ConfigurationItem*), usuários do oceano (*OceanoUser*), vínculos entre um usuário do Oceano com os projetos nele cadastrados (*ProjectUser*) e agentes (*Agent*), são extensíveis via componentes, ou seja, existem interfaces para o cadastro dessas informações, como é apresentado nas próximas seções.

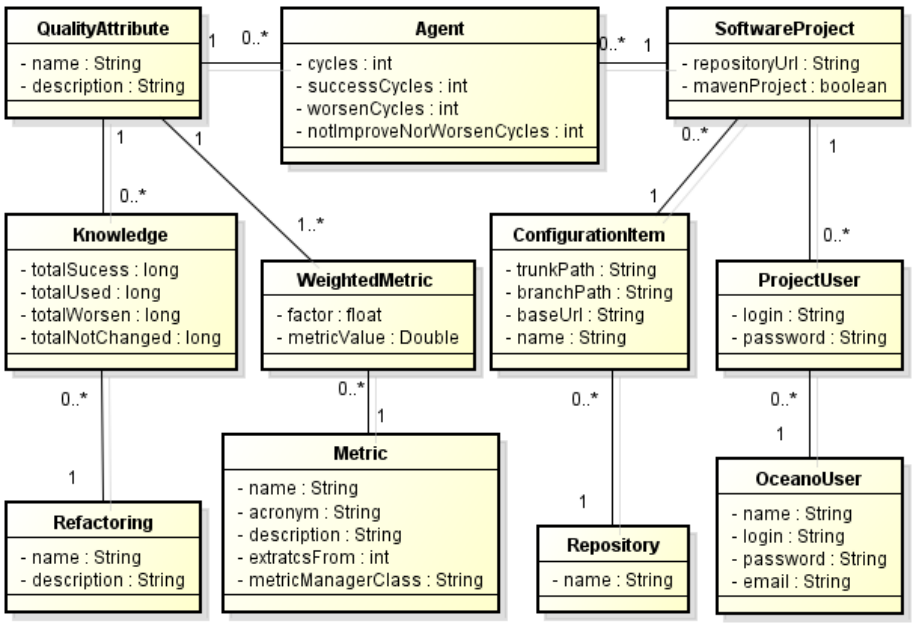


Figura 12: Diagrama de classes persistentes resumido Oceano e PES.

O diagrama mostrado na Figura 13 tem o enfoque nas classes que implementam interesses do Oceano, como SCVs (interface *VCS*), algumas métricas (*LineOfCode*, *NumberOfClasses*, etc.) e fábricas de objetos (*VCSFactory*, *MetricFactory*). Esse diagrama complementa o diagrama mostrado na Figura 12, mostrando, por exemplo, qual foi a arquitetura criada para a extração de uma métrica. As classes *NumberOfPolymorphicMethods*, *DesignSizeInClasses*, *NumberOfClass*, entre outras, implementam as regras para a extração de cada métrica e possuem seus dados básicos mapeados na classe Metric (nome, sigla, etc. – vide Figura 12) que são persistidos no banco de dados. Além disso, nesse diagrama também é possível visualizar que os SCVs também são pontos de extensão via programação. Neste protótipo, tem-se implementado apenas as funcionalidades relativas ao Subversion (Collins-Sussman et al. 2008) (classe *SVN*, que utiliza o framework SVNKit - http://svnkit.com/).

Um dos padrões de projetos utilizados no Oceano é o *Application Service* (Alur et al. 2003), cujo objetivo principal é criar uma camada específica para regras de negócio. Nessa camada encontram-se implementadas tanto regras para garantir a integridade dos dados persistentes, quanto regras específicas de alguma das ferramentas supracitadas, como, por exemplo, o cálculo de um determinado atributo de qualidade.

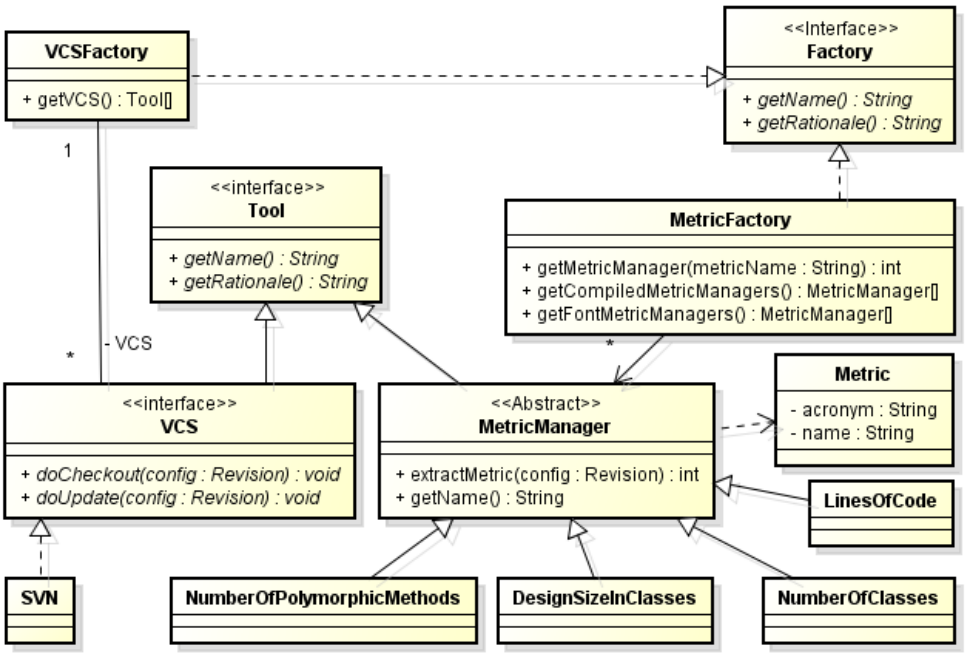


Figura 13: Diagrama de classes resumindo ferramentas/funcionalidades do Oceano.

Com relação ao PEC, a Figura 14 apresenta a estrutura simplificada de sua arquitetura. As refatorações e as buscas de sintomas de refatoração, atualmente implementadas, são baseadas no projeto *open-source* RefactorIt[10]. Ou seja, foram criadas classes específicas para cada refatoração, que além de realizarem a busca de sintomas, informam os dados necessários para que métodos internos ao RefactorIt executem cada refatoração de forma automática. Algumas refatorações são mais complexas do que outras, por existir mais de uma forma de executar a mesma refatoração (Seção 3.5). A arquitetura criada permitiu desde o desenvolvimento de refatorações simples, como *CleanImports* e *EncapsulateFields*, até as mais complexas, como *PullUpMethods*, *PushDownFields*, que por sua vez, utilizam aplicadores específicos na resolução de refatorações (*PullPushRefactoringTool* e *RefactoringApplier*). A interface *RefactoringTool* define métodos básicos às demais refatorações, como por exemplo, busca de sintomas (*findAllSympthons)*, preparação para aplicação de uma refatoração indicada para um sintoma (*preparedNextSymptoms*), entre outros. A arquitetura base é composta pela interface *RefactoringTool* e pela classe abstrata *AbstractRefactoringTool*. Sendo assim, para desenvolver uma refatoração, é necessário criar uma classe que estenda *AbstractRefactoringTool*. As classes *PullPush** foram criadas por dividirem interesses comuns com as refatorações indicadas por suas subclasses.

[10] http://sourceforge.net/projects/refactorit

Além disso, o PEC faz a comunicação com o PES através de classes de serviços e protocolos próprios, utilizando JSON[11] na transferência de dados via protocolo HTTP.

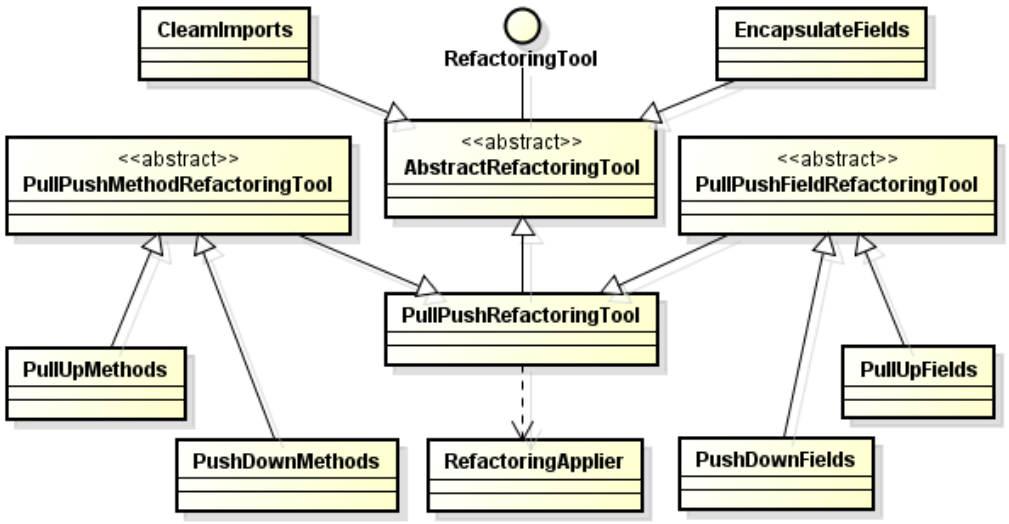


Figura 14: Diagrama de classes reduzido.

Atualmente, para que um projeto de software seja passível de rejuvenescimento, via utilização deste protótipo, ele deve atender às seguintes restrições: ser versionado pelo Subversion; ser um projeto Maven (Company 2008); ser desenvolvido em Java; e ser um projeto Desktop. O protótipo foi desenvolvido basicamente utilizando a linguagem Java. Contudo, dado que o Peixe-Espada oferece uma solução distribuída para processamento e centralizada para monitoramento, foram desenvolvidos pontos explícitos de extensão, de forma a aproveitar toda essa estrutura, como mostrado a seguir:

Extensão das refatorações – Criação de novas refatorações, tanto para Java quanto para outras linguagens. A extensão de refatorações deve ser realizada juntamente com a extensão das métricas de avaliação, como por exemplo, no caso de criação de refatorações para outras linguagens.

Extensão das Métricas – Criação de novos modelos de métricas e atributos de qualidade, para guiar e avaliar as refatorações. Esse tipo de extensão aumenta a robustez da abordagem, gerando maior confiança na aceitação das refatorações.

[11] http://www.json.org/

Extensão dos SCV – Atualmente o sistema dá suporte ao Subversion, mas possui arquitetura para dar suporte a outros SCVs, como Git (Loeliger 2009), Mercurial (O'Sullivan 2009), Bazaar[12], etc. A extensão nesse sentido implica na implementação das funcionalidades básicas de um SCV como *checkin*, *checkout*, criação de ramos e reversão de modificações locais.

4.3 – DEPENDÊNCIAS NO OCEANO

Oceano é uma aplicação web construída do zero, concebida inicialmente pelo autor desta dissertação para centralização de interesses relacionados à GC e disponibilização de uma arquitetura persistente que pudesse oferecer cadastros básicos a outros trabalhos de pesquisa nessa área. Atualmente, o Oceano encontra-se em constante evolução com a participação de quatro alunos de mestrado, um aluno de projeto final e dois de iniciação científica. Esse ambiente ainda caminha para soluções mais modulares, no entanto, muitos recursos nele implementados já estão sendo largamente utilizados por todos os envolvidos.

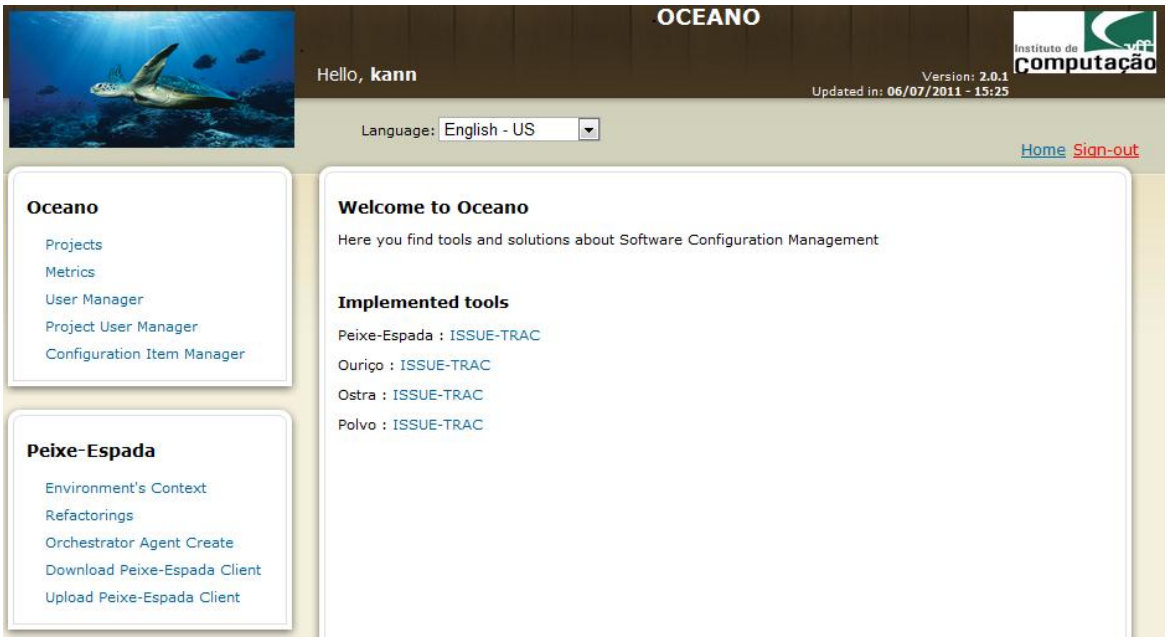


Figura 15: Ambiente Oceano.

Como já discutido, alguns cadastros de dados são disponibilizados no Oceano. A abordagem Peixe-Espada necessita que projetos de software, sob controle de versões, sejam cadastrados no Oceano. Para realizar este cadastro, é necessário que exista no sistema um

[12] http://bazaar.canonical.com/en/

Item de Configuração (IC), que por sua vez dependa de um SCV cadastrado. O ambiente Oceano é mostrado na Figura 15, assim como o menu do PES apresentado com título *Peixe-Espada*. O Oceano está disponível para utilização na url https://gems.ic.uff.br/oceano com usuário e senha *annonynouns*.

A Figura 16 mostra quais são os dados necessários para realizar o cadastro de um IC. O campo "*Base URL*" representa a URL base em que se encontra o versionamento do IC, o "*Trunk Path*" representa onde se encontra o projeto de software em desenvolvimento e o "*Branch Path*" o endereço de onde são criados os ramos para o IC. Convém ressaltar que o tipo de repositório é um dado interno ao sistema por depender da programação das funcionalidades da ferramenta que implementa o SCV específico (Figura 13).

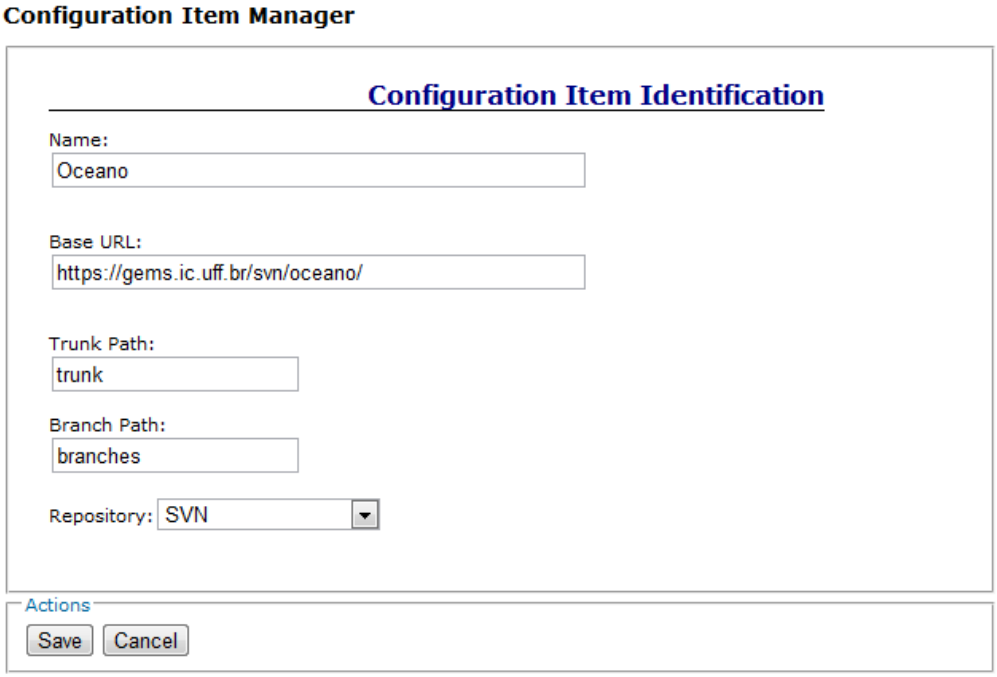


Figura 16: Cadastro do Item de Configuração.

Dado que o IC já se encontra cadastrado, o cadastro do Projeto de Software pode ser realizado, como indicado na Figura 17, selecionando um IC e informando a URL completa do projeto e se este é um projeto Maven. É importante notar que no campo *Repository URL* pode conter qualquer localização do projeto dentro do IC, como, por exemplo, uma URL dentro de um ramo.

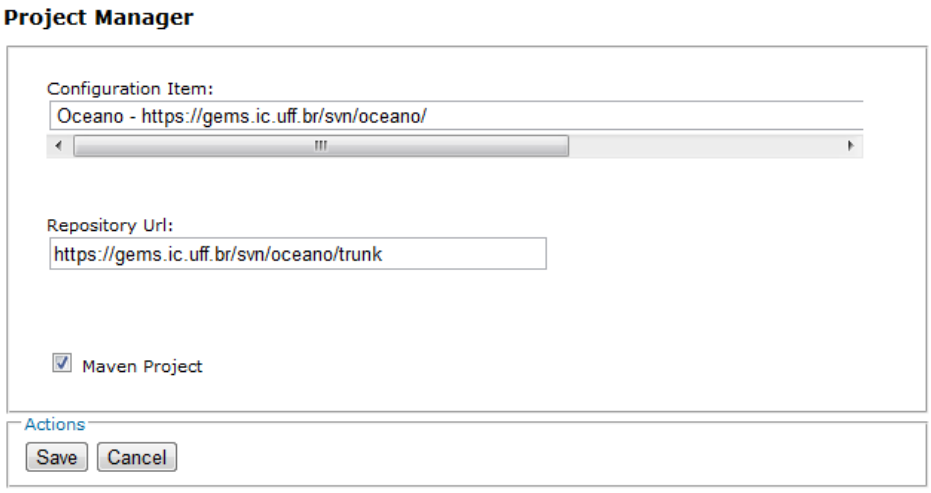


Figura 17: Cadastro de Projetos.

Por Fim, é necessário criar o vínculo de um usuário do Oceano com um projeto de software versionado devidamente cadastrado. Esse vínculo é feito acessando o link "*Project User Manager*" do menu (Figura 15), selecionado um usuário do Oceano e clicando em *Add*, como mostrado na Figura 18. As informações necessárias para a criação desse vínculo são, o usuário e a senha do SCV (Figura 19).

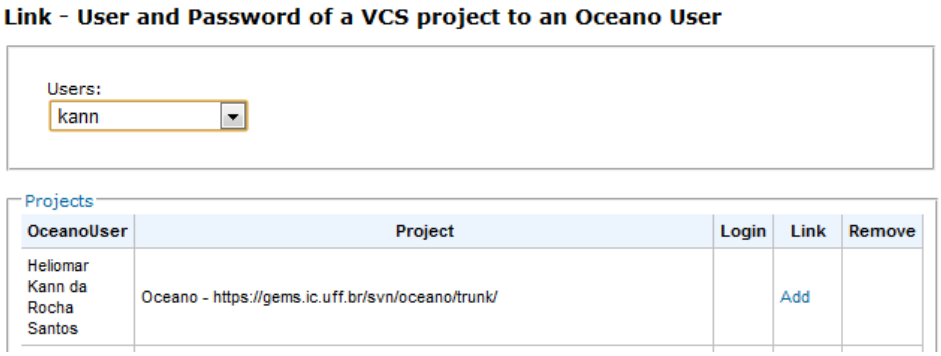


Figura 18: Criação do vínculo entre um usuário do Oceano e um Projeto versionado.

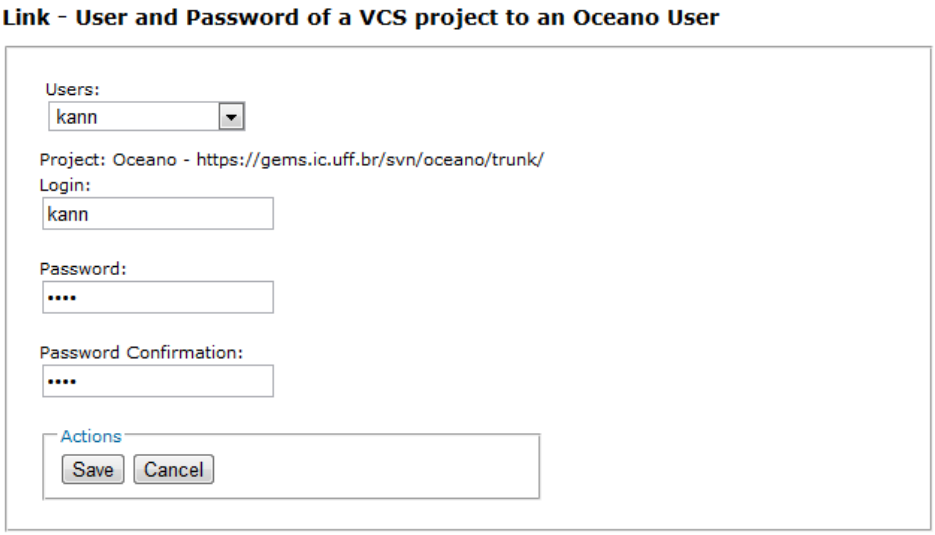


Figura 19: Informações para criar o vínculo indicado na Figura 18.

4.4 – CRIAÇÃO DOS AGENTES ORQUESTRADORES

Dado que todas as dependências estão cadastradas no ambiente Oceano, como mostrado na seção anterior, o usuário pode realizar o cadastro dos agentes orquestradores no PES. Vale lembrar que neste protótipo, a criação de um agente orquestrador é permanente, ou seja, ao se cadastrar um agente orquestrador para o projeto "Oceano" objetivando melhorar a Flexibilidade, esse agente poderá ser utilizado para realizar melhorias nesse atributo de qualidade a qualquer momento. As refatorações ocorrerão sempre na versão mais atual do projeto de software vinculado ao agente orquestrador.

Com o menu mostrado na Figura 15, o PES oferece: o acompanhamento dos agentes (*Environment's Context*); as refatorações cadastradas (*Refactorings*); a criação dos *agentes orquestradores* (*Orquestrator Agent Create*); Download do PEC (*Download* Peixe-Espada *Client*); e disponibilização de uma nova versão do PEC (*Upload* Peixe-Espada *Client*).

Ao acessar a opção de criação de *agentes orquestradores*, é necessário informar apenas um atributo de qualidade e um projeto vinculado para o usuário do Oceano que se encontra logado no sistema (Figura 20). Os atributos de qualidade são dados internos ao Oceano por dependerem da implementação dos algoritmos de métricas e cálculo do atributo de qualidade em si.

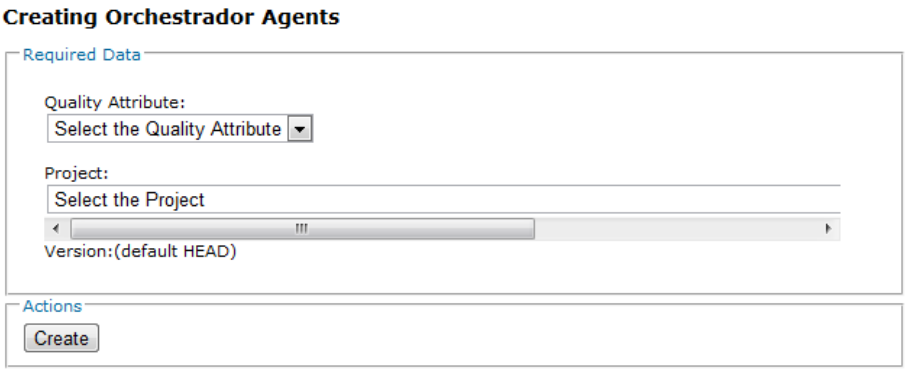


Figura 20: Cadastro de Agentes Orquestradores.

Como discutido anteriormente, os cadastros no PES são feitos apenas na primeira vez, evitando recadastramentos e possibilitando reaproveitamento dos dados. Como, por exemplo, os agentes orquestradores, que sempre estarão disponíveis para melhorar o projeto, em relação ao seu atributo de qualidade, utilizando a versão mais atualizada do projeto no SCV.

4.5 – EXECUÇÃO DO PEIXE-ESPADA CLIENTE

O PEC é o módulo que contém os *agentes trabalhadores*. Para acessar esse módulo, basta clicar no link "Download Peixe-Espada Client" (Figura 15), ou no atalho criado na área de trabalho, caso ele já tenha sido instalado. Com o aplicativo aberto, acessando o menu "File > New", é possível realizar o agendamento do período de ociosidade da máquina em que o PEC encontra-se executando (Figura 21).

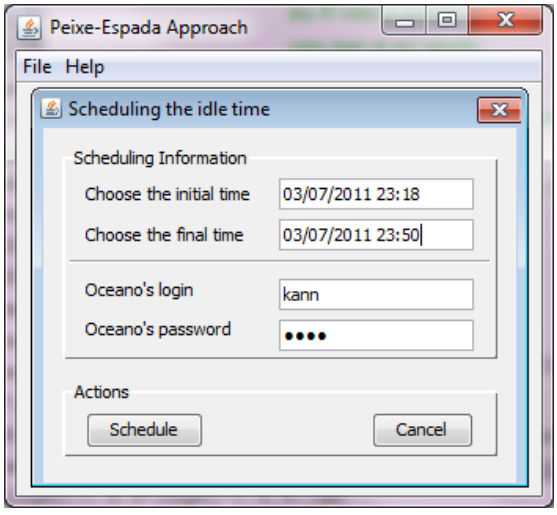


Figura 21: Agendamento do período de ociosidade.

Ainda na Figura 21, o usuário (*Oceano's login*) e senha (*Oceano's password*) do Oceano são exigidos para que, através do vínculo entre um usuário do Oceano e um projeto, os *agentes orquestradores* possam ser identificados. Feito o agendamento, deve-se informar

para qual *agente orquestrador* os *agentes trabalhadores* deverão servir, como indicado na Figura 22.

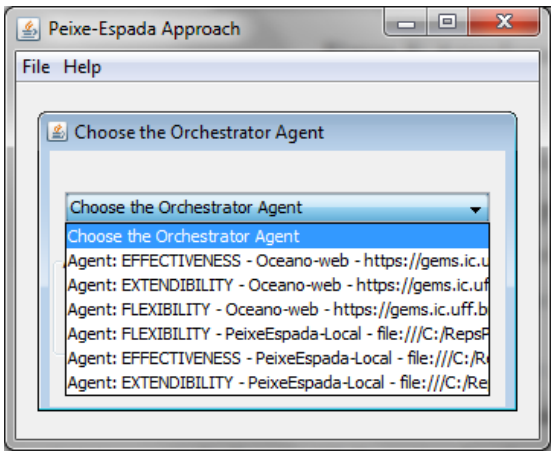


Figura 22: Selecionando um *agente orquestrador*.

A partir daí, é exibida uma tela de acompanhamento dos *agentes trabalhadores* (Figura 23), inicialmente descrevendo as informações básicas sobre o *agente gerente local*, como projeto, data de início, data de término e *status*. Porém, quando se inicia o período de ociosidade, as tarefas dos demais agentes trabalhadores são apresentadas também nessa tela, como indicado na Figura 23. Assim, as demais informações disponibilizadas são: o local do espaço de trabalho onde foi realizado o *checkout* e, consequentemente, ocorrerão as refatorações; versão do projeto no SCV; as métricas utilizadas para calcular os atributos de qualidade; a fórmula para o atributo de qualidade; informações gerais sobre o andamento de cada refatoração, como, por exemplo, a tentativa de aplicação de alguma refatoração, os valores de métricas extraídos, a criação de ramos no SCV, as melhoras, pioras e não alterações do atributo de qualidade, etc.

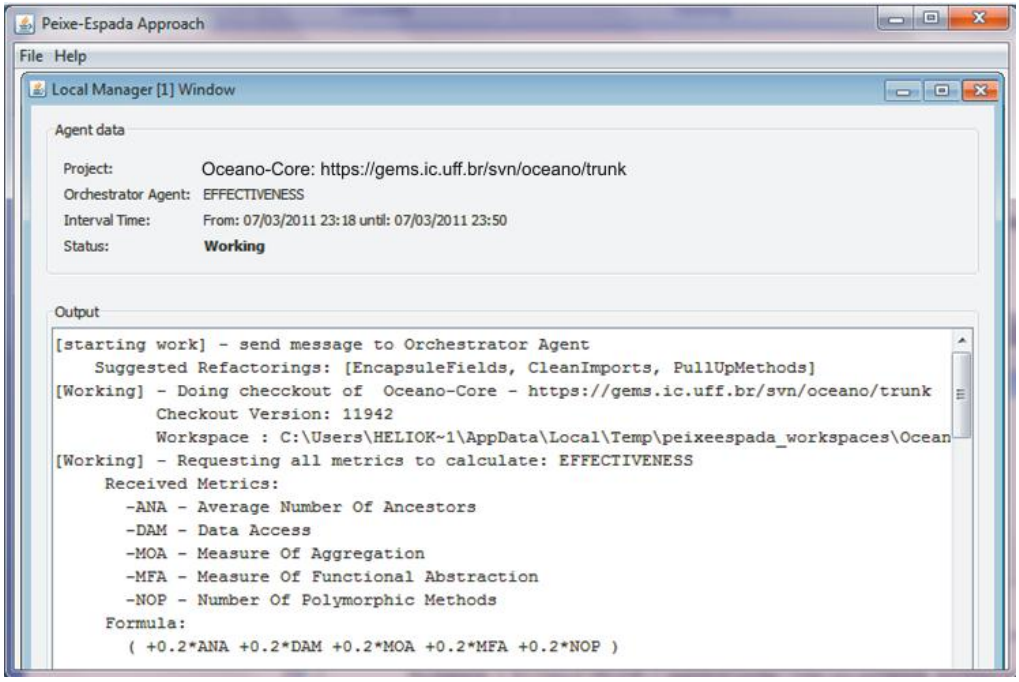


Figura 23: Tela de acompanhamento dos agentes trabalhadores.

Ao término do trabalho, é gerado um resumo que contém todas as refatorações realizadas com sucesso, assim como o ramo onde elas encontram-se disponíveis. Documentos da evolução histórica (*log*) com valores de métricas e valores dos atributos de qualidade também são gerados. Eles servem para o acompanhamento em geral, principalmente para apoiar a compreensão dos resultados durante avaliações experimentais. Exemplos desses documentos são mostrados na Figura 24 e na Figura 25, respectivamente.

```
       Encapsulating Field:
br.uff.iduff2.service.academico.DesvinculacaoAlunoService.boletimService

       Encapsulating Field:
br.uff.iduff2.service.academico.AlunoRematriculadoService.alunoDAO

       Encapsulating Field:
br.uff.iduff2.service.academico.AlunoRematriculadoService.alunoService

       Upping Method:
br.uff.iduff2.managedbeans.academico.QuadroHorariosMB.listaCursoNativeQueryPorAluno
         Para: br.uff.iduff2.managedbeans.BaseMB
             Selected alternative to be used in the refactoring:
                 ->listaCursoNativeQueryPorAluno()
                         must have access to the following members
                    =>Change access modifier of the following members

           Upping Method:
br.uff.iduff2.managedbeans.academico.QuadroHorariosMB.alterar
           Para: br.uff.iduff2.managedbeans.BaseMB
               Selected alternative to be used in the refactoring:
                  ->alterar(TurmaInscricaoDecorada)
                         must have access to the following members
                    =>Change access modifier of the following members
```

Figura 24: Exemplo de relatório final de refatorações bem sucedidas.

```
DAM;DCC;MOA;NOP;FLEXIBILITY
0,512632004908915;2,008324661810614;1060,0;16,0;537,6260768357746
0,512632004908915;2,010405827263267;1060,0;17,0;538,1255565444114
0,512632004908915;2,0093652445369408;1060,0;17,0;538,125816690093
0,5125352065316348;2,010405827263267;1060,0;18,0;538,6255323448171
0,5124868073119829;2,010405827263267;1060,0;19,0;539,1255202450121
0,5120270149733844;2,011446409989594;1060,0;20,0;539,625145151246
0,5120028153635585;2,011446409989594;1060,0;21,0;540,1251391013435
```

Figura 25: Exemplo de relatório final para métricas nas refatorações bem sucedidas.

É importante notar que cada refatoração automatizável gera um arquivo contendo os valores das métricas das *refatorações mínimas* realizadas com sucesso, facilitando a visualização dos valores de uma refatoração automatizável para um atributo de qualidade. Para isso, o arquivo indicado na Figura 25 teria um nome no formato "Projeto_Refatoracao_AtributoQualidade.csv".

4.6 – MONITORAMENTO DOS AGENTES

Além dos documentos finais e janelas de saídas de texto nas máquinas locais, o usuário da abordagem Peixe-Espada pode acompanhar o trabalho dos agentes de forma geral e simplificada via interface do PES. A tela de acompanhamento é apresentada pela Figura 26, Figura 27 e Figura 28. Essa tela também exibe o conhecimento do SMA como um todo. A partir dessa funcionalidade, é possível visualizar quais *agentes trabalhadores* estão ativos e quais estão inativos, aguardando o início do período de ociosidade do computador (Figura 27). Também é possível ver qual é o aproveitamento de cada *agente trabalhador* (grupo de *agentes trabalhadores* por máquina), assim como o aproveitamento acumulado do *agente orquestrador*. Este depende não só do aproveitamento dos *agentes trabalhadores* ativos, mas também de todos os agentes que já trabalharam para ele.

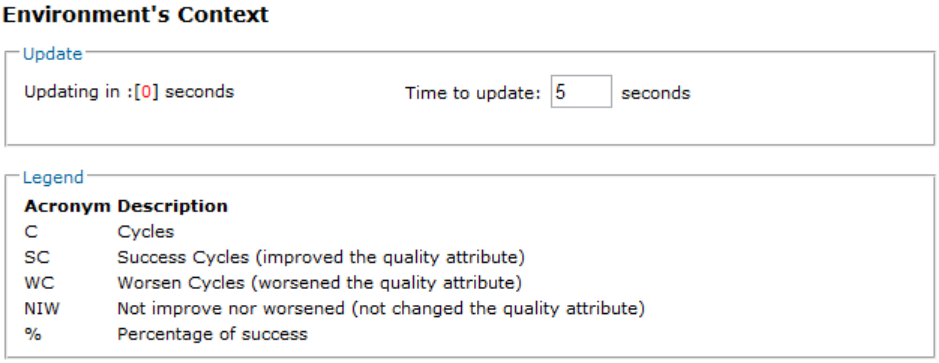
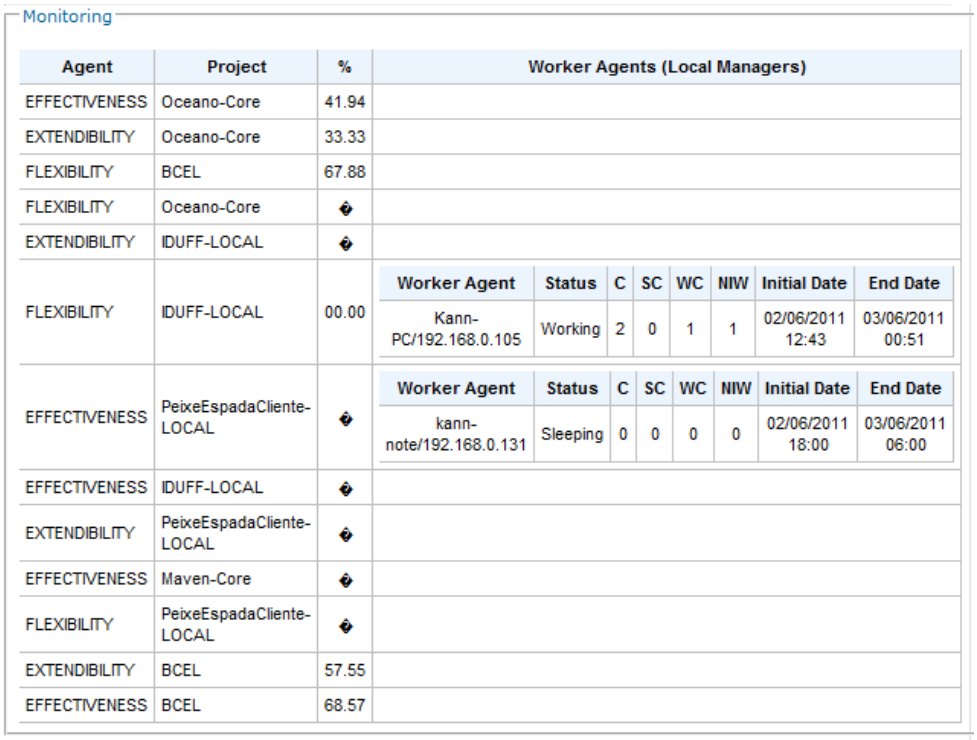


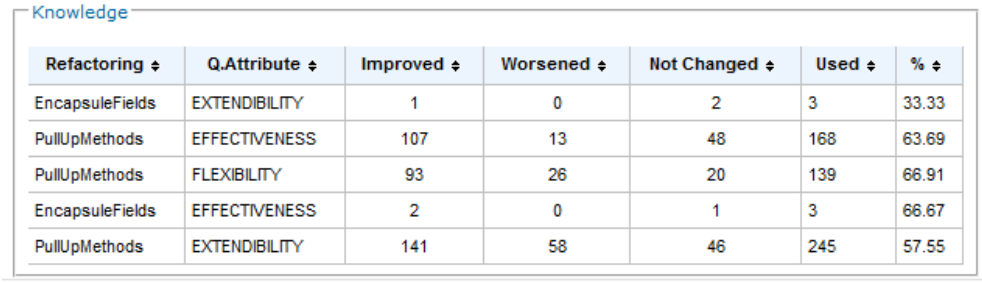
Figura 26: Legenda na tela de monitoramento.



Monitoring

Agent	Project	%	Worker Agents (Local Managers)
EFFECTIVENESS	Oceano-Core	41.94	
EXTENDIBILITY	Oceano-Core	33.33	
FLEXIBILITY	BCEL	67.88	
FLEXIBILITY	Oceano-Core	🔲	
EXTENDIBILITY	IDUFF-LOCAL	🔲	
FLEXIBILITY	IDUFF-LOCAL	00.00	Worker Agent: Kann-PC/192.168.0.105 | Status: Working | C: 2 | SC: 0 | WC: 1 | NIW: 1 | Initial Date: 02/06/2011 12:43 | End Date: 03/06/2011 00:51
EFFECTIVENESS	PeixeEspadaCliente-LOCAL	🔲	Worker Agent: kann-note/192.168.0.131 | Status: Sleeping | C: 0 | SC: 0 | WC: 0 | NIW: 0 | Initial Date: 02/06/2011 18:00 | End Date: 03/06/2011 06:00
EFFECTIVENESS	IDUFF-LOCAL	🔲	
EXTENDIBILITY	PeixeEspadaCliente-LOCAL	🔲	
EFFECTIVENESS	Maven-Core	🔲	
FLEXIBILITY	PeixeEspadaCliente-LOCAL	🔲	
EXTENDIBILITY	BCEL	57.55	
EFFECTIVENESS	BCEL	68.57	

Figura 27: Acompanhamento dos agentes na tela de monitoramento.



Knowledge

Refactoring	Q.Attribute	Improved	Worsened	Not Changed	Used	%
EncapsuleFields	EXTENDIBILITY	1	0	2	3	33.33
PullUpMethods	EFFECTIVENESS	107	13	48	168	63.69
PullUpMethods	FLEXIBILITY	93	26	20	139	66.91
EncapsuleFields	EFFECTIVENESS	2	0	1	3	66.67
PullUpMethods	EXTENDIBILITY	141	58	46	245	57.55

Figura 28: Conhecimento do SMA na tela de monitoramento.

O conhecimento adquirido pelo SMA (Figura 28) é composto por dados obtidos dos *agentes trabalhadores*. Na atual versão do Peixe-Espada, o conhecimento representa o resultado em termos de melhoras, pioras e não alterações da aplicação de cada *refatoração mínima* de todos os projetos juntos. É a partir desse conhecimento que os *agentes orquestradores* ordenam as sugestões de refatorações, priorizando as de maior aproveitamento global para o atributo de qualidade ao qual estão vinculados.

A tela de acompanhamento também oferece um campo para o usuário informar o tempo de atualização dos dados (Figura 26). O campo "*Time to Update*" marca o tempo em segundos para a próxima atualização automática dessa tela.

4.7 – CONSIDERAÇÕES FINAIS

Como pode ser observado, o protótipo foi criado com a preocupação de atender minuciosamente à abordagem Peixe-Espada, além de possuir algumas características, como a arquitetura com pontos de extensão. O ambiente Oceano permitiu que a implementação do PES contivesse apenas os interesses do SMA, pois suas dependências já foram previamente implementadas no Oceano pelo autor desta dissertação em conjunto com os demais alunos envolvidos em pesquisas relacionadas (Seção 4.3). A instalação do PEC é um ponto importante desta abordagem como um todo, facilitando o uso do cliente devido ao fato de não necessitar de um processo de instalação (é necessário apenas possuir uma Máquina Virtual Java instalada). Os relatórios finais contendo as refatorações bem sucedidas e as evoluções dos valores das métricas, divididas por refatorações automatizáveis e atributos de qualidade também são pontos importantes, por contribuírem com o entendimento das alterações realizadas e gerarem dados para estudos. Esses relatórios juntamente com a integração das alterações em ramos isolados facilitam o entendimento de cada alteração, possibilitando o rastreamento completo, via SCV, para ajudar a equipe de desenvolvimento na aceitação das refatorações realizadas.

CAPÍTULO 5 – AVALIAÇÃO

5.1 – INTRODUÇÃO

A principal questão de pesquisa deste trabalho consiste em saber se é possível fazer continuamente o rejuvenescimento automático de código fonte guiado por atributos de qualidade. Este capítulo é responsável por avaliar se a abordagem Peixe-Espada é capaz de amenizar o envelhecimento precoce de software. Para isso, é importante aplicar a abordagem sobre projetos que sejam não apenas distintos, mas que também possuam diferentes características, como arquitetura, equipes de desenvolvimento, entre outras. Durante o processo de avaliação também é importante analisar se existem padrões entre refatorações e atributos de qualidade para cada projeto em específico. Esses padrões podem indicar que algumas refatorações são úteis para melhorar determinados atributos de qualidade (*reforço positivo*) ou piorar outros atributos de qualidade (*reforço negativo*).

O processo de avaliação compreende atividades bem definidas, de forma a sistematizar a execução dos experimentos. As principais atividades são as seguintes:

Obter dados reais das equipes de desenvolvimento – O Peixe-Espada é fortemente apoiado por GC, principalmente no que tange ao controle de versões. Dessa forma, os dados utilizados para o experimento foram obtidos através de versões de projetos de software que fazem uso do Subversion.

Escolher períodos fixos de tempo para aplicação do Peixe-Espada – Como já discutido, o PEC permite o agendamento do período de ociosidade. Uma abordagem nesse sentido seria remover a restrição de tempo e deixar o PEC executar até que todas as refatorações se esgotassem. No entanto, notou-se que, dependendo do tamanho do projeto, a execução dos experimentos se estende por dias. Esse fato reforçou a justificativa de utilizar o agendamento do período de ociosidade. Sendo assim, sem perda de generalidade, para executar o experimento foi escolhido um tempo de 12 horas. É importante notar que um dos critérios de parada é o término das oportunidades de refatoração. Nesse caso, os projetos menores terminam antes.

Fazer uma análise quantitativa e qualitativa sobre os resultados – A partir de uma mesma versão de um projeto de software, aplicar isoladamente uma refatoração automatizável por atributo de qualidade e gerar arquivos para cada trio (refatoração automatizável, atributo

de qualidade e projeto). Em outras palavras, a medição das aplicações das *refatorações mínimas* de cada refatoração automatizável em um projeto gera arquivos de dados para criação de gráficos e análises posteriores. Nesse cenário, ao fazer o experimento com 3 atributos de qualidade, 2 refatorações automatizáveis e 4 projetos, são obtidos 6 arquivos de evoluções dos valores das métricas por projeto para serem analisados, considerando que todas as refatorações automatizáveis tiveram *reforço positivo* sobre cada um dos atributos de qualidade. No protótipo atual, as medidas de refatorações aplicadas que não geram reforços positivos não são gravadas em arquivo. As refatorações utilizadas neste experimento foram *PullUpMethods* e *EncapsulateFields*, e os atributos de qualidade medidos foram, Flexibilidade, Extensibilidade e Efetividade. Outras refatorações como *PullUpFields*, *PushDownFields* e *PushDownMethods* foram implementadas e estão em fase de testes. Assim como os atributos de qualidade Funcionalidade, Compreensibilidade e Reusabilidade.

Foram utilizados 3 computadores no experimento. Vale ressaltar que essa configuração foi utilizada apenas para agilizar a obtenção dos dados, pois como explicado na atividade acima, a aplicação das refatorações é feita de forma isolada. Sendo assim, a estratégia de paralelismo utilizada para o experimento foi melhorar um projeto de cada vez, almejando melhoria dos 3 atributos de qualidade ao mesmo tempo. Para isso, foram cadastrados agentes orquestradores para cada par projeto/atributo de qualidade, com um grupo de *agentes trabalhadores* a eles alocados, em cada computador. Essa configuração não gera conflitos físicos, dado que cada agente fará *checkout* da mesma versão do projeto e integrará suas refatorações de *reforço positivo* em ramos isolados. As configurações dos computadores utilizados no experimento são descritas na Tabela 9.

Tabela 9: Configurações dos computadores utilizados no experimento.

	Processador	Memória	HD
PC1	Intel® Core™ i3 2.4 GHz	4GB – DDR3	500 GB
PC2	Intel® Core™ 2 Quad 2.66 GHz	4GB – DDR2	500 GB
PC3	Intel® Core™ 2 Duo 2.0 GHz	2GB – DDR2	250 GB

O PC1, além de executar o PEC, também executou o Oceano contendo o PES. O repositório do SCV foi copiado para cada computador, diminuindo assim o tempo de acesso a ele, visto que esse experimento não visava medir desempenho concorrente e sim obter os

dados das refatorações para cada par projeto/atributo de qualidade, para que uma avaliação quantitativa e qualitativa fosse realizada.

O restante deste capítulo está dividido na descrição dos projetos de software selecionados para executar os experimentos (Seção 5.2), os resultados gerais e individuais obtidos (Seção 5.3), as ameaças ao estudo (Seção 5.4), e por fim, as considerações finais (Seção 5.5).

5.2 – PROJETOS

Além da necessidade de projetos de software com diferentes características para a execução dos experimentos, o protótipo do Peixe-Espada impõe algumas exigências tecnológicas, como ser um projeto Maven e ser versionado pelo Subversion. Além disso, foram adicionados alguns requisitos extras para prover uma análise mais ampla do experimento. Esses requisitos são: os projetos devem ter tamanhos distintos; devem ser desenvolvidos por equipes diferentes; e devem ser projetos reais. A seguir, os projetos escolhidos são descritos e suas propriedades são mostradas na Tabela 10. Essas propriedades foram obtidas através da ferramenta StatSvn (http://www.statsvn.org/).

PEC e Oceano-Core (**OC**) – Como explicado no Capítulo 4, o PEC é parte da implementação da abordagem Peixe-Espada destinada a executar na máquina dos desenvolvedores, contendo os agentes trabalhadores e algoritmos de refatoração. Já o OC é a principal dependência do Oceano (ambiente web de GC). Nele encontram-se implementações de coleta de métricas, coleta de atributos de qualidade, cadastro de projetos e funcionalidades de SCV. As tecnologias utilizadas nesses projetos encontram-se descritas na Seção 4.1.

Sistema Acadêmico Administrativo da UFF (**IDUFF** - http://id.uff.br) – O IDUFF é o atual sistema de gerência acadêmica da UFF. Nele estão disponibilizadas funcionalidades para alunos e funcionários. As funcionalidades para alunos vão desde atualização de dados cadastrais à inscrição em disciplinas online e geração de declarações. As funcionalidades para funcionários estão ligadas aos vínculos dos mesmos, ou seja, um professor pode lançar notas, o departamento pode criar turmas, a administração acadêmica pode incluir cursos e assim por diante. A arquitetura desse sistema se caracteriza pela divisão em camadas lógicas internas, ou seja, camadas de negócios (classes *services* do padrão *Application Service*), camadas de acesso a dados (do inglês, *Data Access Object* - classes DAO), camada de visualização (interface web) e camadas de controladores que fazem a ponte entre a camada de visualização

e a camada de negócios. Algumas das tecnologias envolvidas nesse projeto são: Hibernate (Bauer e King 2004), *AspectJ* (Laddad 2009), *Facelets* (Aranda e Wadia 2008), *Spring* (Mak et al. 2010), *JBoss Seam* (Farley 2007), JSF 1.2 (Geary e Horstmann 2004) e *RichFaces* (Katz 2008). Outra característica desse projeto é a sua constante evolução e manutenção por equipes de rotatividade alta. Apesar do Peixe-Espada não suportar projetos web, por não garantir o funcionamento das camadas de visualização, o IDUFF foi escolhido por ser uma aplicação real, relativamente grande, em ativa manutenção e com código fonte ao alcance do autor deste trabalho.

Byte Code Engineering Library (**BCEL** - http://jakarta.apache.org/bcel/) – O BCEL é um projeto *open-source* que tem a intenção de dar aos usuários uma maneira conveniente de analisar, criar e manipular arquivos binários de Java (bytecodes). BCEL já está sendo usado em vários projetos, tais como compiladores, otimizadores, ofuscadores, geradores de código e ferramentas de análise de código.

Tabela 10: Propriedade dos projetos selecionados para os experimentos.

	PEC	**OC**	**IDUFF**	**BCEL**
Data de medição	6/Abr/11	6/Abr/11	30/Mar/11	6/Abr/11
Linhas de código	8.119	27.554	241.818	142.832
Desenvolvedores	2	6	27	5
Arquivos	79	290	1.746	504
Classes	65	266	1.037	405
Linhas de código Java	6.636	26.375	163.913	58.010

5.3 – RESULTADOS

Como previamente discutido, as refatorações utilizadas na avaliação foram *EncapsulateFields* e *PullUpMethods*, medidas com os atributos de qualidade Efetividade, Flexibilidade e Extensibilidade. A Tabela 11 mostra a quantidade de *sintomas de refatoração* (SR) encontrados para cada um dos projetos.

Tabela 11: Sintomas de refatoração.

	PEC	OC	IDUFF	BCEL
SR *PullUpMethods*	110	123	3415	799
SR *EncapsulateFields*	6	5	97	0

Os resultados das aplicações das refatorações sobre os sintomas são mostrados na Tabela 12, em termos de melhora, piora ou não alteração do atributo de qualidade. Nessa tabela, R1 representa *EncapsulateFields* e R2 *PullUpMethods*.

Tabela 12: Resultados gerais das refatorações.

	PEC	OC	IDUFF	BCEL
EFETIVIDADE Aplicado (Melhorou | Piorou | Não alterou)				
R1	73 (63 | 3 | 7)	28 (11 | 1 | 16)	128* (110 | 14 | 4)	140* (96 | 12 | 32)
R2	1 (1 | 0 | 0)	3 (1 | 0 | 2)	31 (31 | 0 | 0)	0
FLEXIBILIDADE				
R1	73 (63 | 3 | 7)	28 (11 | 1 | 16)	57* (6 | 45 | 6)	137* (93 | 25 | 19)
R2	1 (1 | 0 | 0)	3 (1 | 0 | 2)	31 (0 | 31 | 0)	0
EXTENDIBILIDADE				
R1	73 (63 | 3 | 7)	28 (11 | 0 | 17)	81* (37 | 44 | 0)	245* (141 | 58 | 46)
2	1 (0 | 0 | 1)	3 (0 | 0 | 3)	31 (0 | 31 | 0)	0

Como pode ser observado, *aplicado = melhorou + piorou + não alterou*, porém, o número de refatorações aplicadas é distinto do número de sintomas. Isso ocorre porque a aplicação da refatoração indicada por um sintoma segue apenas alguns pré-requisitos (Tabela 1), o que não garante a conclusão nem a aceitação da refatoração em si. As refatorações que são descartadas e, consequentemente, não têm informações armazenadas sobre atributos de

qualidade ocorrem nos seguintes casos: quando a aplicação de um *sintoma de refatoração* faz o código fonte não compilar; quando se percebe alteração de funcionalidade (a ferramenta de refatoração é capaz de indicar essas situações); ou quando a refatoração deixa de fazer sentido após refatorações anteriores terem sido aplicadas, em vista da reestruturação do código (ex., subir um método, considerando que uma refatoração anterior subiu um método com a mesma assinatura de uma classe irmã).

Ainda na Tabela 12, a aplicação dos sintomas de refatoração de *PullUpMethods*, indicadas com "*", não foram concluídas para o IDUFF e BCEL. Como dito anteriormente, o período de 12 horas foi utilizado no experimento para cada refatoração.

Como discutido anteriormente, a fim de evitar o desperdício de processamento devido às caras tarefas de entrada e saída, o Peixe-Espada usa informação sobre o sucesso das refatorações (razão entre melhorado e aplicado, vide Tabela 12) para sugerir as sequências de refatorações. A Tabela 13 mostra o aproveitamento das refatorações aplicadas para os atributos de qualidade em cada projeto. Desconsiderando *PullUpMethods* para o IDUFF e BCEL, pois nem todas possibilidades de refatorações foram aplicadas, é possível perceber que uma mesma refatoração tem aproveitamentos distintos para diferentes projetos, independentemente do atributo de qualidade. No entanto, não foram analisados a fundo os motivos que geraram essas diferenças, suspeita-se que seja pela grande diferença entre o número de sintomas encontrados nos diferentes projetos. Já os motivos das semelhanças de *PullUpMethods* para os atributos de qualidade nos mesmos projetos (exceto IDUFF e BCEL) são vistos nas próximas subseções, com o detalhamento dos resultados para cada projeto.

Tabela 13: Aproveitamento das refatorações aplicadas.

	PEC	**OC**	**IDUFF**	**BCEL**
PullUpMethods				
Efetividade	86.30%	39.29%	85.94%	68.57%
Flexibilidade	86.30%	39.29%	10.53%	67.88%
Extensibilidade	86.30%	39.29%	45.68%	57.55%
EncapsulateFields				
Efetividade	100%	33.33%	100%	-
Flexibilidade	100%	33.33%	100%	-
Extensibilidade	0.0%	0.0%	0.0%	-

Ainda em relação à Tabela 13, é possível observar alguns padrões, como: *PullUpMethods* é uma boa refatoração para Extensibilidade, Efetividade e Flexibilidade. Isso ocorre devido ao aumento do valor do Número de Métodos Polimórficos (NOP), mas isso depende da forma que os métodos vão para a superclasse. Um método pode ser declarado em uma superclasse e cancelado nas subclasses ou pode ser totalmente movido para a superclasse. Dependendo da situação, a métrica NOP pode ser afetada ou não. Se a declaração do método for criada como abstrata na superclasse, essa métrica é afetada positivamente, já se o método for totalmente movido para a superclasse, NOP não é afetada.

Durante o experimento, também foi observado que as tarefas de medição e refatoração são caras em termos de uso de recursos computacionais. Como mostrado na Tabela 12, nem todas as refatorações foram aplicadas. Isso ocorreu devido ao tempo gasto para executar as medições. Por exemplo, no PEC foram obtidos os seguintes valores médios e aproximados de tempos para *PullUpMethods* no PC1: 12 segundos para descobrir todos os 110 sintomas; 8 segundos para aplicar uma refatoração mínima; 44 segundos para a medição de Flexibilidade (compilação: 8 segundos; DAM: 34 segundos; DCC: 2 segundos; MOA: 0,07 segundos; e NOP: 0,08 segundos). Já para o IDUFF no mesmo computador, os tempos foram: 24 segundos para descobrir todos os 3.415 sintomas; 41 segundos para aplicar uma refatoração mínima; 17 minutos para a medição de Flexibilidade (compilação: 14 segundos; DAM: 8 minutos; DCC 8 minutos; MOA 0,6 segundos; e NOP 0,6 segundos).

O fato acima citado apoia a necessidade de preocupação da priorização das refatorações via aprendizado do SMA, como já discutido, e sugere a seguinte heurística: dado que as refatorações não afetam todos os valores das métricas, o cálculo do atributo de qualidade poderia ser feito excluindo as métricas não alteradas, trazendo um ganho expressivo de desempenho para a abordagem.

As próximas subseções discutem as medições e resultados para cada projeto em particular. Os valores utilizados nos gráficos já estão normalizados (vide exemplo na Tabela 7 e na Tabela 8), lembrando que o Peixe-Espada faz medições para a aplicação de cada *refatoração mínima*. Sendo assim, qualquer alteração nos valores das métricas, por mínima que seja, deve ser considerada para a verificação de reforço positivo ou negativo. Nesse caso, é esperado que o tamanho do projeto seja inversamente proporcional à margem de alteração das medidas, ou seja, em um projeto grande com muitos sintomas e provavelmente muitas aplicações, as *refatorações mínimas* afetarão pouco os valores das métricas, porém qualquer

alteração é considerada. Os gráficos possuem o eixo das abscissas (X) representando as refatorações bem sucedidas (a primeira não é uma refatoração e sim a configuração original, por isso os valores normalizados são 1) e o eixo das ordenadas (Y) representando os valores normalizados das métricas e atributos de qualidade.

5.3.1 – PEIXE-ESPADA CLIENTE

Como visto na Tabela 12 e na Tabela 13, o PEC realizou apenas uma refatoração para *EncapsulateFields*, sendo DAM (*Data Access Metric*) a métrica responsável por melhorar os atributos de qualidade Efetividade e Flexibilidade. A Extensibilidade não obteve alteração. A métrica DAM aumenta porque seu cálculo é baseado na razão de atributos privados por atributos públicos e *EncapsulateFields* aumenta o número de atributos privados e diminui o número de atributos públicos

A refatoração *PullUpMethods* obteve *reforços positivos* significantes para os 3 atributos de qualidade medidos. A Figura 29 mostra a evolução das medidas para Extensibilidade nas refatorações bem sucedidas de *PullUpMethods*. Como esse atributo de qualidade é calculado através da fórmula 0,5*(ANA – DCC + MFA + NOP), seu crescimento ocorreu devido ao aumento da métrica NOP, como já explicado (aumento do número de métodos polimórficos). A métrica DCC influencia negativamente a Extensibilidade, porém seu crescimento foi pequeno, iniciando na décima oitava refatoração com o valor já normalizado de 1,016129 e terminando com 1,112903 na septuagésima terceira refatoração. O gráfico mostra apenas até a refatoração 18 (a primeira é a medição original), as medidas da Figura 29 aumentam sempre de um valor muito parecido e os demais medidas não se alteram, dessa forma, o comportamento dos valores das métricas se manteve.

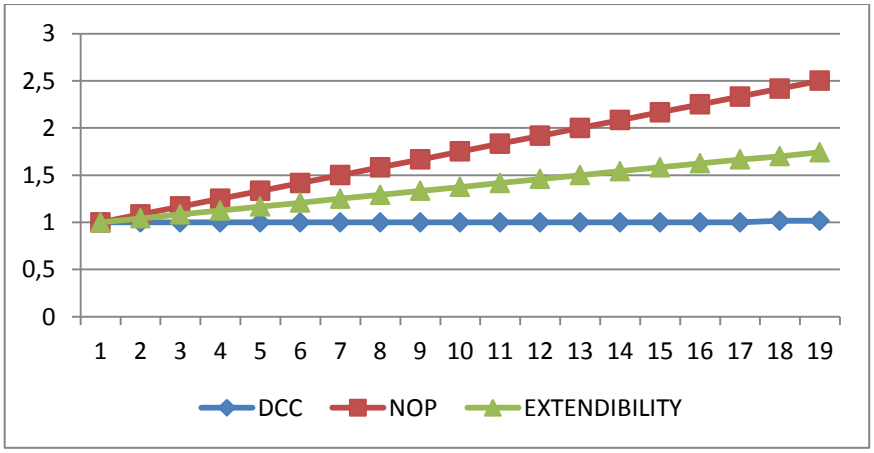


Figura 29: Extensibilidade para *PullUpMethods* no PEC.

Uma evolução muito parecida ocorreu para Flexibilidade. Devido a sua fórmula 0,25* (DAM – DCC) + 0,5*(MOA + NOP). A métrica DCC (que afeta negativamente a Flexibilidade) cresceu pouco e a métrica responsável pela melhora do atributo de qualidade foi novamente NOP.

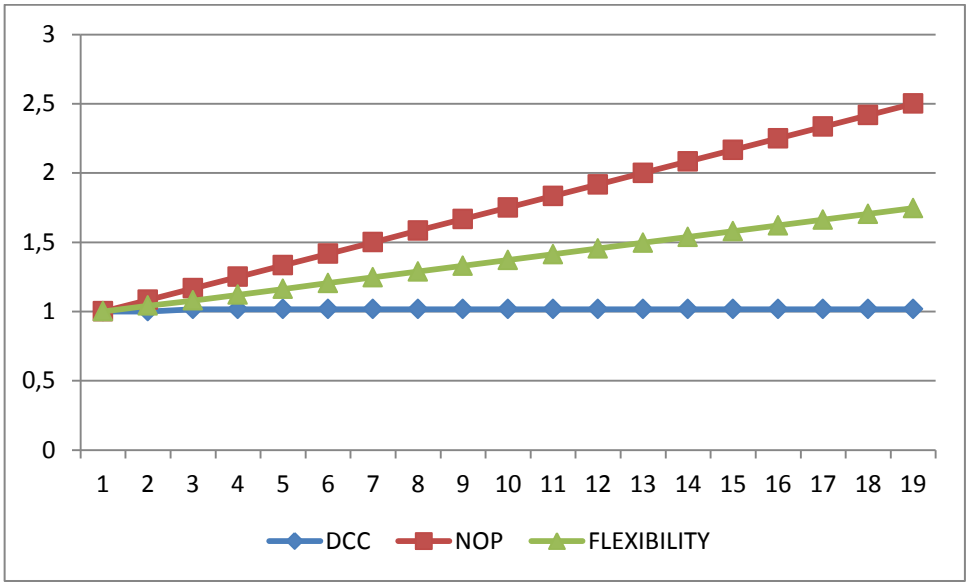


Figura 30: Flexibilidade para *PullUpMethods* no PEC.

Para o atributo de qualidade Efetividade, com a fórmula 0,2*(ANA + DAM + MOA + MFA + NOP), a única métrica afetada foi a NOP, ocasionando o crescimento mostrado na Figura 31.

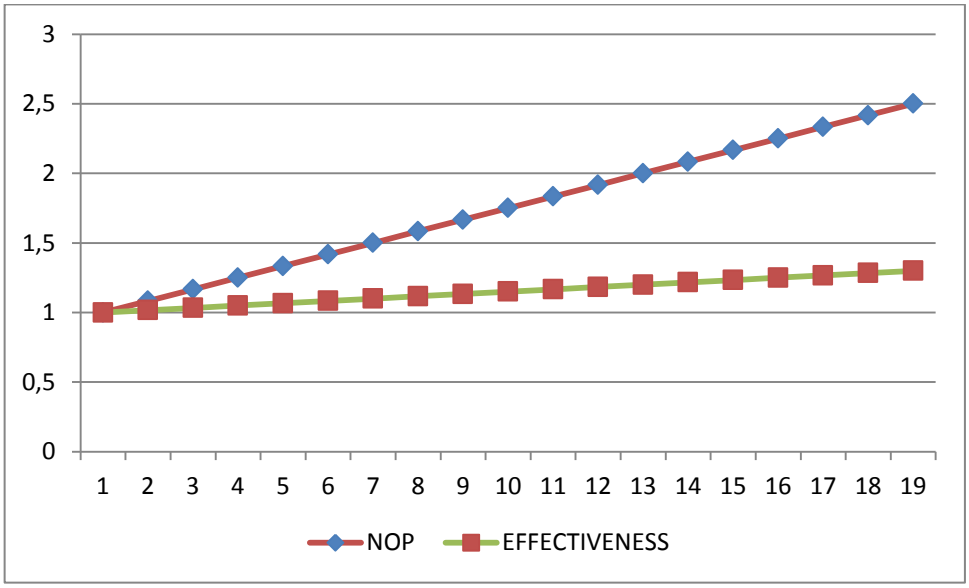


Figura 31: Efetividade para *PullUpMethods* no PEC.

O crescimento desses atributos de qualidade para o PEC manteve o comportamento até a septuagésima terceira refatoração bem sucedida.

5.3.2 – OCEANO-CORE

Apesar do OC ter apresentado poucas aplicações tanto para *EncapsulateFields* quanto para *PullUpMethods* (vide Tabela 12), os resultados não foram muito diferentes para as refatorações de *reforço positivo*. O único reforço positivo para *EncapsulateFields* melhorou a métrica DAM, melhorando indiretamente os atributos de qualidade Efetividade e Flexibilidade, tal qual discutido no PEC. O gráfico de evolução das 11 refatorações de PullUpMethods com reforços positivos para Extensibilidade é mostrado na Figura 32. A métrica ANA não foi alterada, DCC sofreu apenas uma alteração positiva, MFA sofreu 5 alterações negativas quase imperceptíveis e NOP obteve seu crescimento em todos os reforços positivos utilizando *PullUpMethods*, como já discutido.

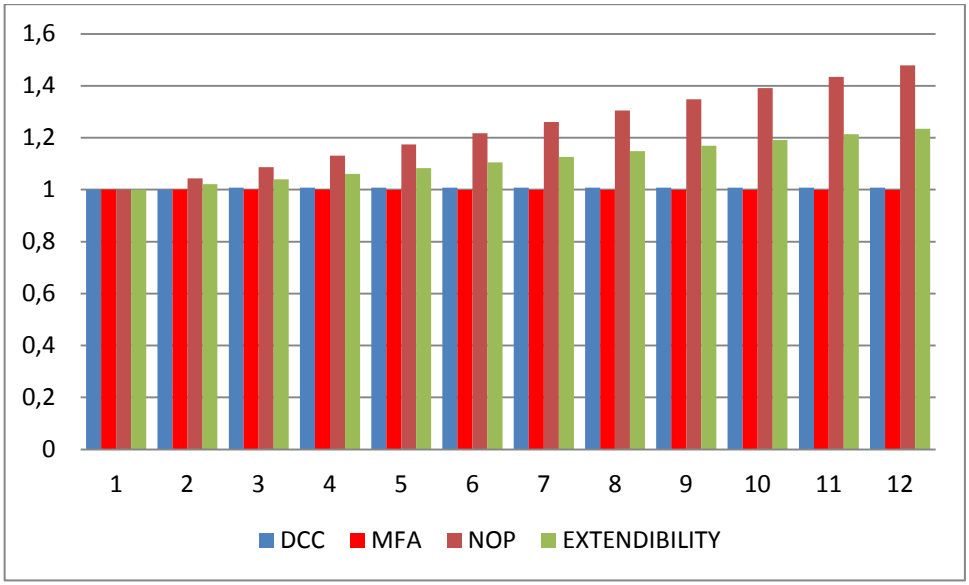


Figura 32: Extensibilidade para *PullUpMethods* no OC.

A métrica MFA sofreu um pequeno decréscimo. Ela é calculada pela razão entre os métodos herdados e os métodos acessíveis ou oferecidos de uma classe. A refatoração *PullUpMethods* aumenta o número de métodos herdados, quando o método sobe integralmente para a superclasse e não apenas a sua declaração. Porém, se a superclasse recebe apenas a declaração do método (caso a superclasse seja abstrata), cada uma das subclasses vai aumentar o número de métodos oferecidos, mas não herdados, uma vez que o método abstrato terá que ser sobre-escrito forçosamente. A Figura 33 mostra um caso de

PullUpMethods ocorrido no OC que explica a queda de MFA, alguns métodos são sintéticos para facilitar a compreensão.

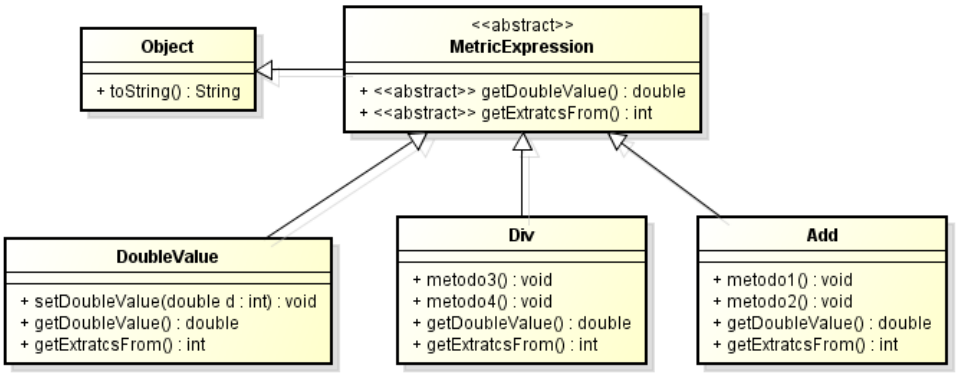


Figura 33: Exemplo para cálculo de MFA.

No exemplo dado na Figura 33, supondo que a classe *Object* tenha apenas 1 método público, *DoubleValue* possui MFA = 1/4 (métodos herdados/métodos oferecidos), ou seja, 1 referente ao método herdado de *Object*, pois *MetricExpression* não oferece nenhum método implementado para aumentar o valor de métodos herdados e o valor 4 é obtido pelos 4 métodos que *DoubleValue* oferece (3 próprios e 1 de *Object*). Sendo assim, a classe *Div* e *Add* possuem MFA = 1/5. Após realizar *PullUpMethods* com o método *setDoubleValue*(...) de *DoubleValue* para *MetricExpression*, a métrica NOP sofreu um aumento, dado que o método alvo ficou abstrato, aumentando o número de métodos polimórficos, mas MFA sofreu uma queda. Recalculando MFA, *DoubleValue* continua com 1/4, já *Div* e *Add* passam a ter 1/6, pois o método abstrato forçou sua sobre-escrita, aumentando o número de métodos oferecidos.

Já DCC sofreu apenas uma alteração positiva ao subir um método cuja declaração de um parâmetro era desconhecida pela superclasse. Esse caso pode ser visualizado com o mesmo exemplo supracitado, porém colocando um parâmetro de tipo não primitivo no método *setDoubleValue*.

Da mesma forma que no PEC, Flexibilidade manteve o padrão do comportamento de Extensibilidade para as variações dos valores das métricas na aplicação de *PullUpMethods* em reforços positivos, como mostrado na Figura 34.

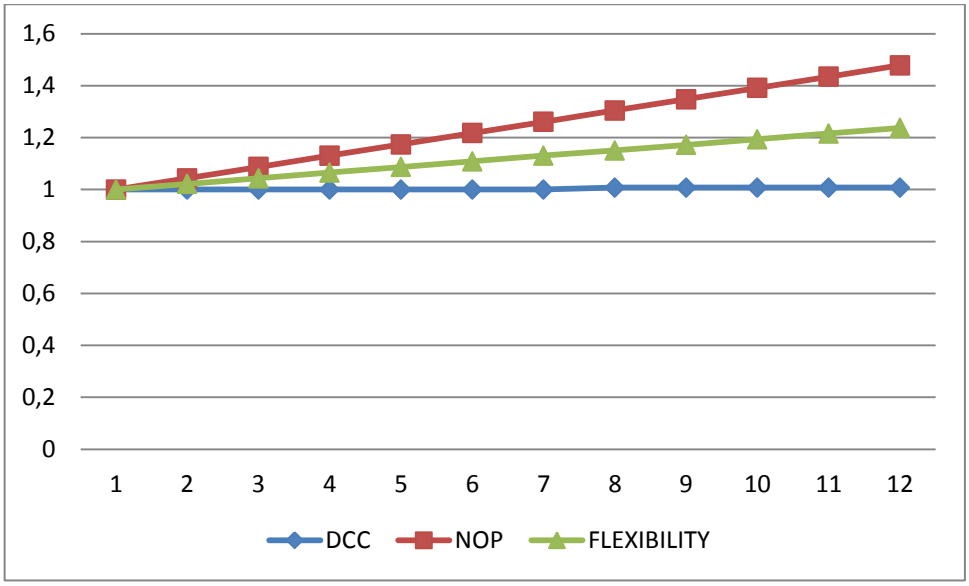


Figura 34: Flexibilidade para *PullUpMethods* no OC.

Já para Efetividade, apresentada na Figura 35, a refatoração *PullUpMethods* afetou a métrica MFA negativamente, como discutido anteriormente. A métrica DCC obteve apenas uma alteração positiva, lembrando que DCC é a única métrica que possui fator negativo na fórmula para Efetividade. Contudo, NOP manteve o crescimento, favorecendo a Efetividade.

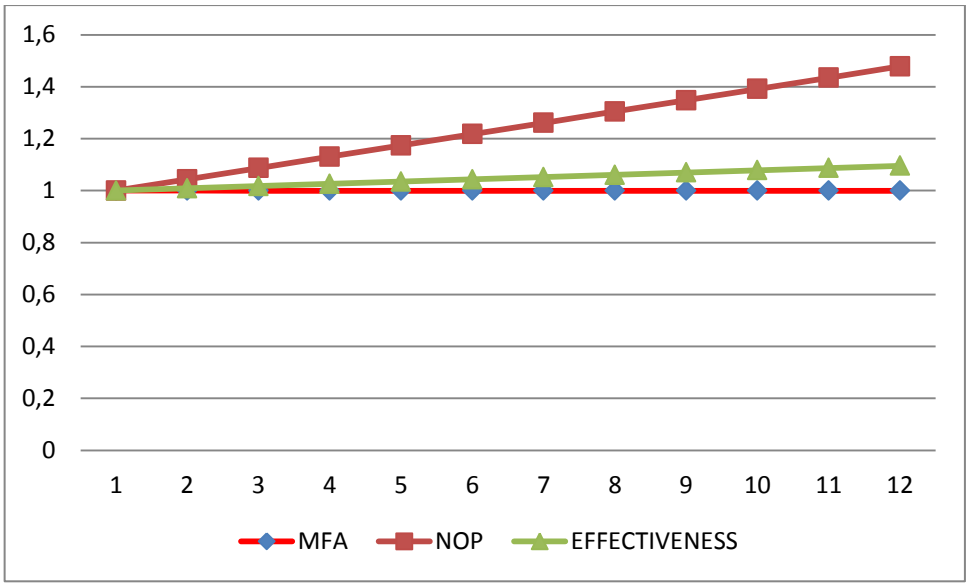


Figura 35: Efetividade para *PullUpMethods* no OC.

5.3.3 – SISTEMA ACADÊMICO ADMINISTRATIVO DA UFF

Como já discutido, o IDUFF é o maior sistema entre os projetos de software utilizados nos experimentos e o intervalo de tempo de 12 horas para a execução do Peixe-Espada não foi suficiente para a conclusão da aplicação das refatorações indicadas pelos *sintomas de*

refatoração. Porém, os resultados obtidos nos reforços positivos mantiveram o padrão observado nos demais projetos. Acredita-se que o tamanho do projeto tenha influenciado no crescimento das medidas, no entanto, todas as alterações são consideradas pela abordagem Peixe-Espada, por menor que sejam. Vale notar que se o crescimento mantiver, mesmo que pequeno, com o número crescente de aplicações de refatorações, ao final o crescimento acumulado pode ser significante para o projeto como um todo ou para os pontos específicos onde houve mais aplicações.

A Figura 36 mostra o comportamento nas 19 primeiras refatorações de reforço positivo com *EncapsulateFields* para Flexibilidade. Da mesma forma que no OC e no Peixe-Espada, o crescimento da Flexibilidade deu-se em função do aumento da métrica DAM, as demais métricas não se alteraram.

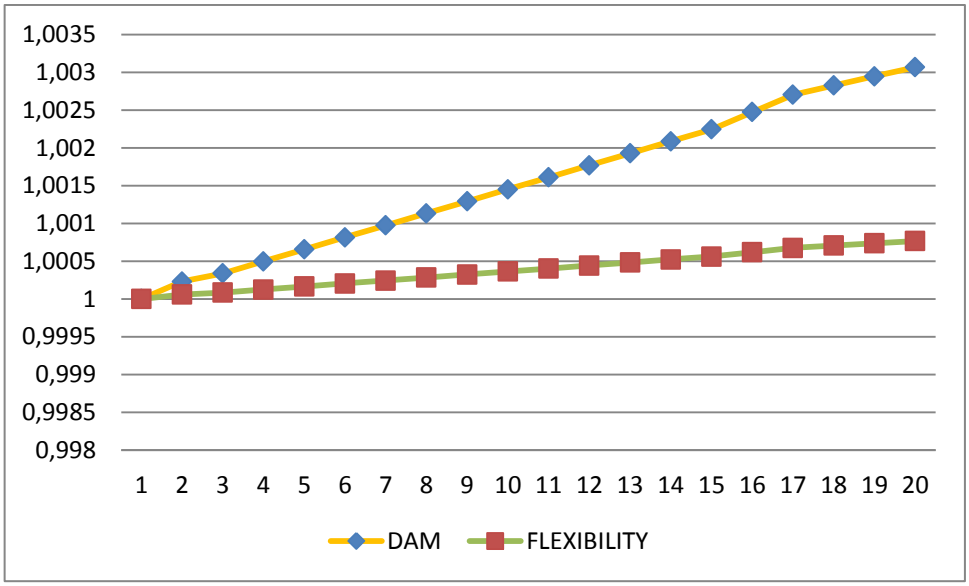


Figura 36: Flexibilidade para *EncapsulateFields* no IDUFF.

Além do crescimento da métrica DAM, a Figura 37 apresenta um decaimento na métrica MFA, ocasionando um crescimento menor para Efetividade. Nesse caso, MFA decresce quando um campo encapsulado, porque aumenta o número de métodos oferecidos (MFA = métodos herdados/métodos oferecidos).

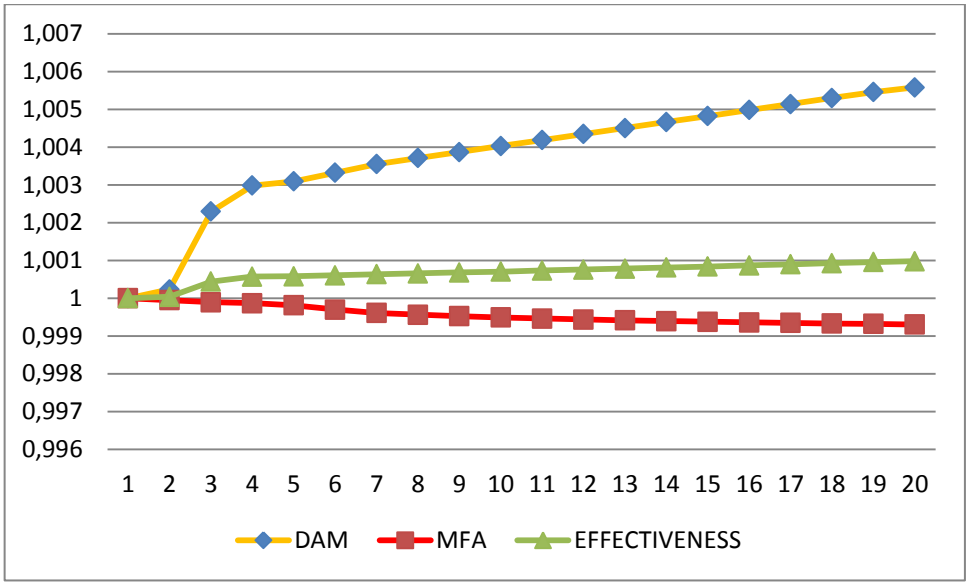


Figura 37: Efetividade para *EncapsulateFields* no IDUFF.

Como *PullUpMethods* não obteve todos os resultados medidos no período estipulado de tempo para rodar os experimentos, os gráficos obtiveram variações no número de refatorações realizadas. Contudo, a Figura 38 mostra que o crescimento da Extensibilidade continua se comportando de maneira muito parecida com o crescimento da métrica NOP. A Figura 39 representa a mesma medição para Extensibilidade, porém em um gráfico separado devido ao crescimento impercebível de DCC e MFA comparado à NOP e ao valor de Extensibilidade.

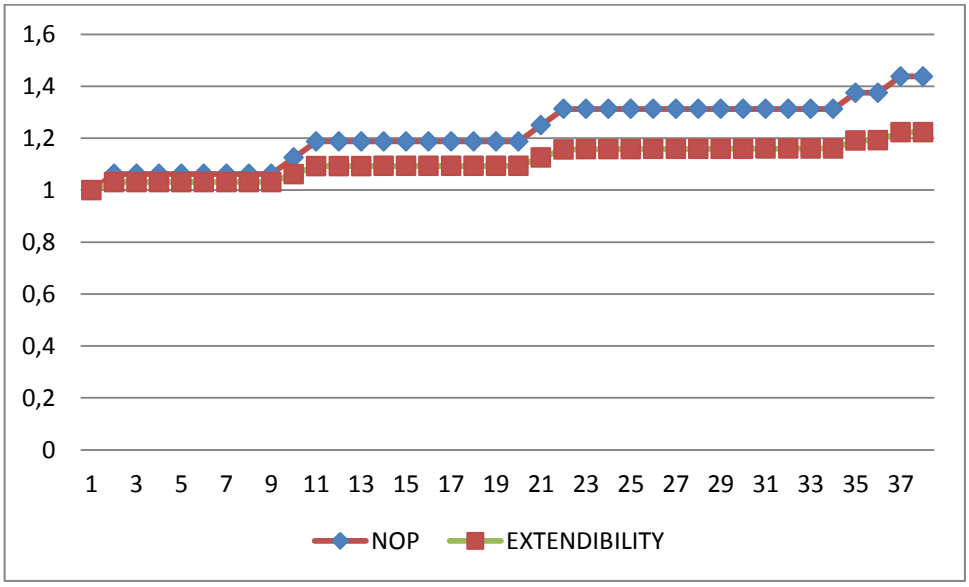


Figura 38: Extensibilidade com NOP para *PullUpMethods* IDUFF.

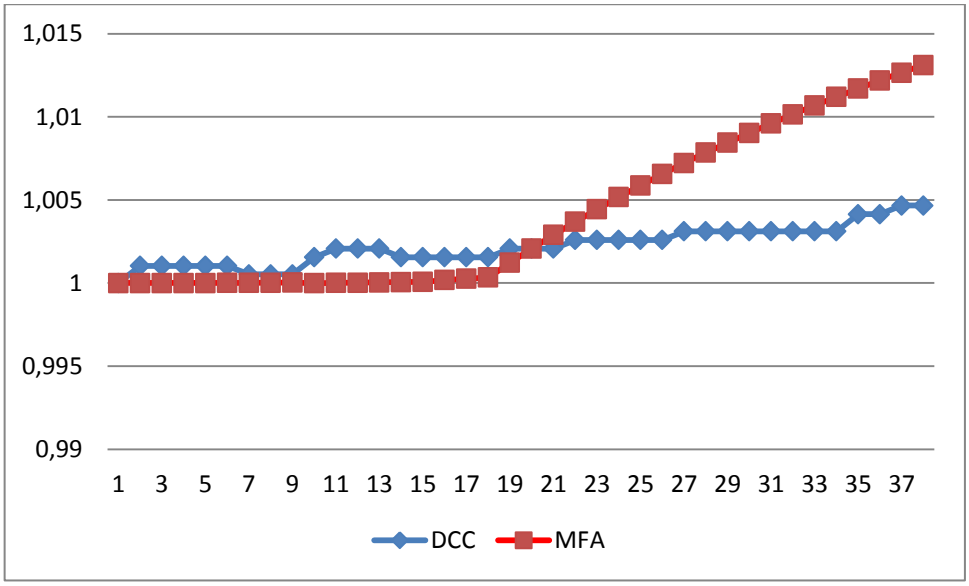


Figura 39: *PullUpMethods* para DCC e MFA no IDUFF.

A máquina PC1, que trabalhou para melhoria da Extensibilidade do IDUFF, conseguiu aplicar com sucesso um maior número de refatorações. Entretanto, o comportamento foi semelhante ao ocorrido com Extensibilidade. Ou seja, o atributo de qualidade Efetividade cresceu em maior escala quando houve crescimento da métrica NOP e crescimentos em escalas menores com pequenas variações ocorreram devido às demais métricas (DAM e MFA). A métrica NOP foi retirada da Figura 40 para que DAM e MFA fossem vistas em relação à Efetividade. DAM obteve um decréscimo minúsculo (de 1 para 0,998686) comparado ao crescimento de NOP (de 1 para 1,6875), que novamente ditou o padrão de crescimento do atributo de qualidade Efetividade.

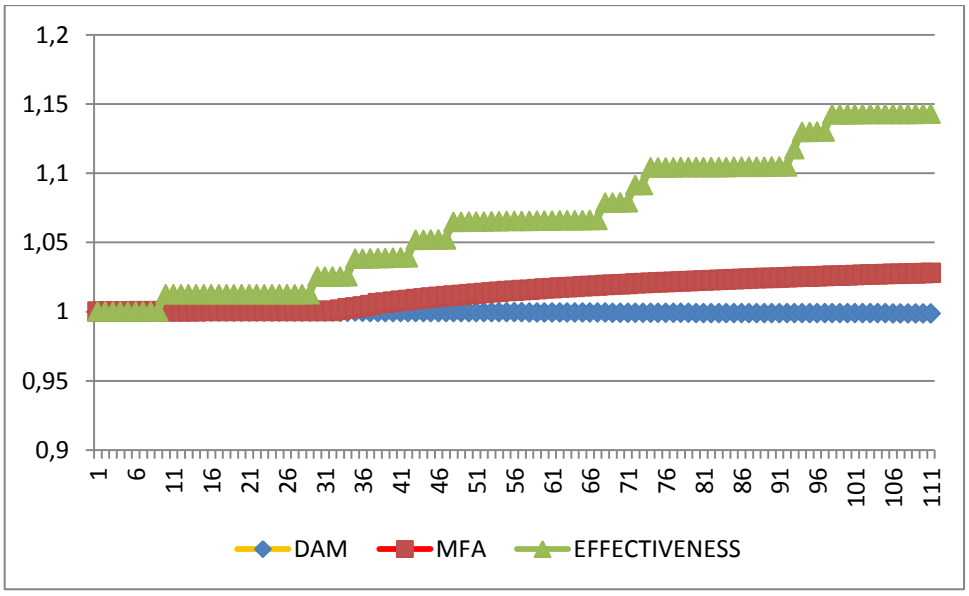


Figura 40: *PullUpMethods* para Efetividade no IDUFF.

Para Flexibilidade, o padrão se repetiu. A Figura 41 mostra um quase imperceptível decrescimento da métrica DAM, um pequeno crescimento de DCC e o atributo de qualidade sendo guiado praticamente pelo comportamento de NOP. Especialmente para essa medição, foi realizada uma investigação mais detalhada, tendo em vista o pequeno número de reforços positivos. O computador PC1 executou esse experimento obtendo os seguintes dados: 76 refatorações que foram revertidas por quebra de código fonte; 45 refatorações aplicadas que pioraram Flexibilidade; 4 resoluções de refatorações insolucionáveis (isso pode ocorrer dado que o sintoma aponta uma refatoração que não é mais possível de ser feita, considerando as refatorações realizadas anteriormente); 6 refatorações aplicadas que não surtiram efeito em Flexibilidade; e apenas 6 refatorações de reforço positivo. Como não são gerados arquivos para reforços negativos, esse experimento não pode detectar ao certo o que ocorreu de diferente para esse atributo de qualidade. Porém, suspeita-se que o pequeno número de refatorações de reforços positivos foi devido ao tempo de extração das métricas DAM e DCC, como mostrado na Seção 5.3. Essas métricas foram verificadas como as mais lentas para a medição do IDUFF. Para esse atributo de qualidade, o mesmo experimento foi realizado 3 vezes e o padrão de resultados se manteve.

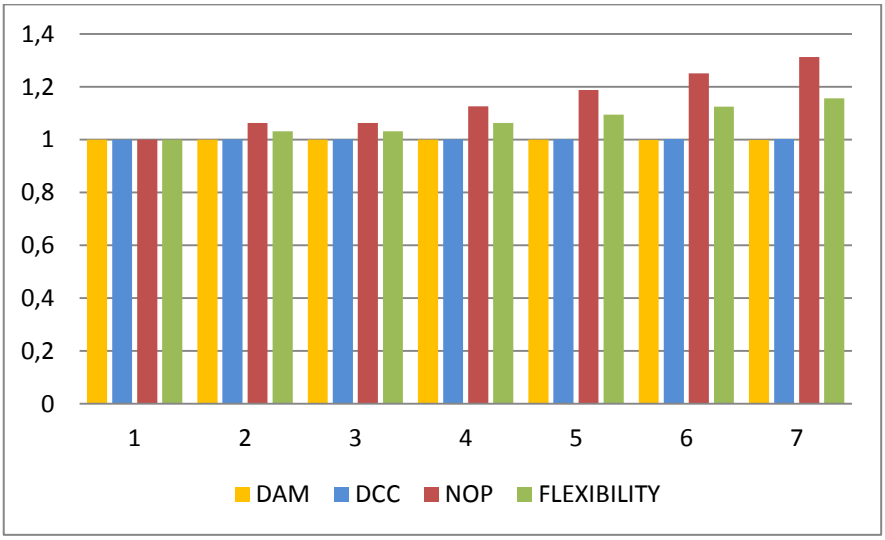


Figura 41: *PullUpMethods* para Flexibilidade no IDUFF.

5.3.4 – *BYTE CODE ENGINEERING LIBRARY*

Como mostrado na Tabela 12, o BCEL não obteve nenhum *sintoma de refatoração* para *EncalsulateFields*. Já para *PullUpMethods*, assim como no IDUFF, o período de 12h para execução não foi o suficiente para aplicar todas as refatorações possíveis, indicadas pelos *sintomas de refatoração*. No entanto, os resultados adquiridos indiciaram similaridade aos

demais projetos. A Figura 42, por exemplo, mostra um pequeno crescimento de DCC (influência negativa), enquanto NOP, por razão já discutida, dita o padrão de crescimento do atributo de qualidade Extensibilidade na refatoração com *PullUpMethods*.

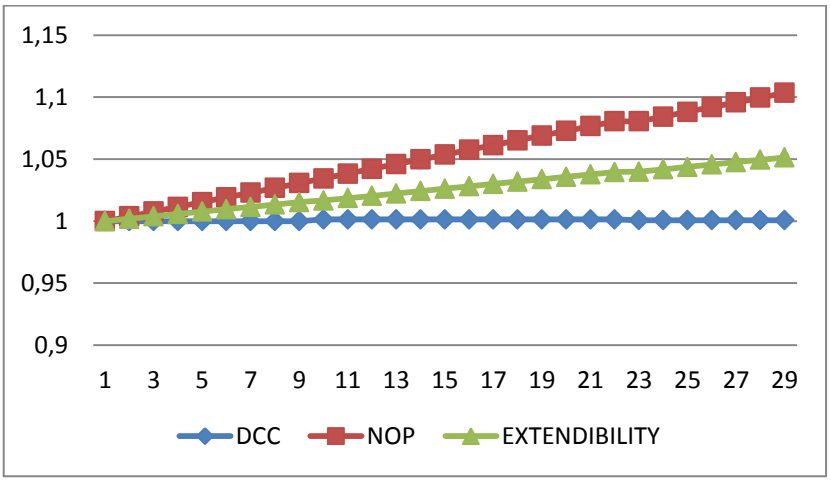


Figura 42: *PullUpMethods* para Extensibilidade no BCEL.

Como em todos os outros casos, Flexibilidade com *PullUpMethods* obteve um comportamento similar à Extensibilidade, como mostrado na Figura 43.

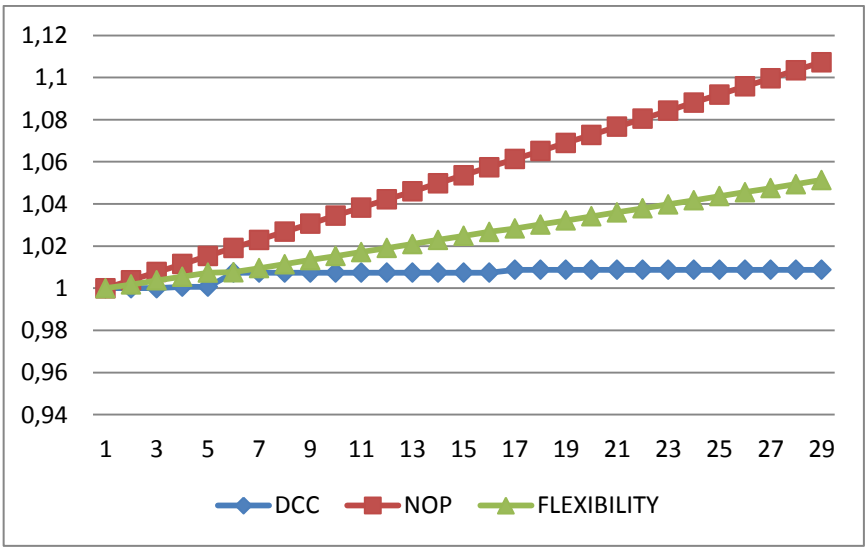


Figura 43: *PullUpMethods* para Flexibilidade no BCEL.

Para Efetividade, a refatoração *PullUpMethods* no BCEL obteve o comportamento similar às métricas do OC e PEC, ou seja: NOP determinou o crescimento principal da Efetividade; ocorreu um decrescimento muito pequeno de MFA (não foi adicionado ao gráfico devido ao quase imperceptível decréscimo de 1 a 0,982142); e não houve alteração de outras métricas.

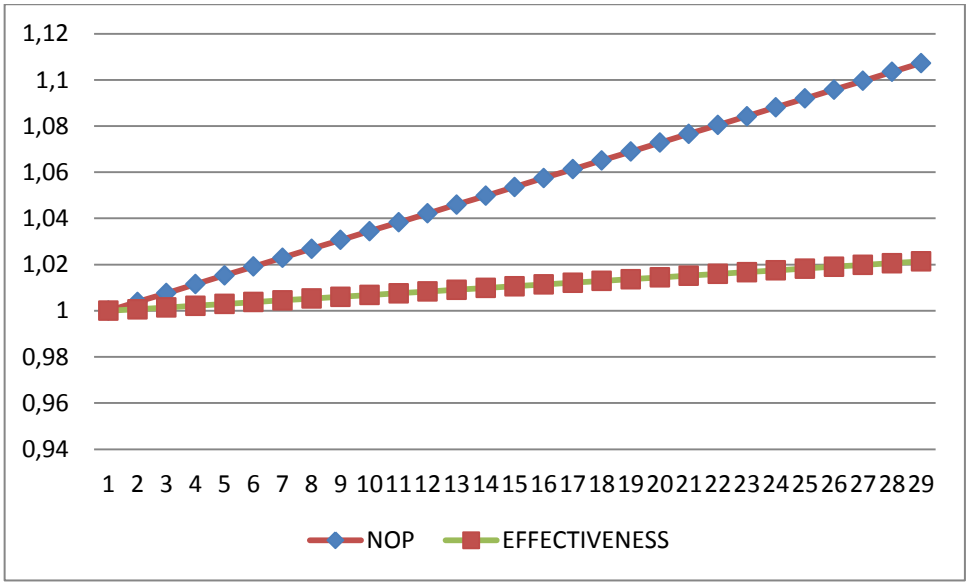


Figura 44: *PullUpMethods* para Efetividade no BCEL.

5.4 – AMEAÇAS AO ESTUDO

Como pode ser observado, *PullUpMethods* é uma boa refatoração para Extensibilidade Efetividade e Flexibilidade. Isso ocorre devido ao aumento do valor do Número de Métodos Polimórficos (NOP). *EncapsulateFields* se mostrou uma boa refatoração para Efetividade e Flexibilidade. No entanto, é possível identificar algumas ameaças à validade do experimento, como:

Número de projetos – O experimento foi realizado com apenas quatro projetos, sendo assim, não é possível generalizar as evidências apresentadas.

Avaliação – O experimento foi realizado utilizando base em dados históricos (repositórios de SCV) e avaliado apenas quantitativamente. Por isso, é necessário que esses resultados sejam validados por integrantes das equipes de desenvolvimento dos projetos, como gerentes e desenvolvedores.

Medição – A abordagem Peixe-Espada utilizou como modelo inicial para medição e aceitação das refatorações, métricas e atributos de qualidade descritos na literatura (Bansiya e Davis 2002). Por se tratar de apenas um modelo, aumentam-se os riscos de imprecisões e erros.

Efeitos Colaterais – Foram considerados apenas os benefícios técnicos de refatorações em termos de arquitetura de software, sem analisar os impactos dessas refatorações automáticas em certas técnicas utilizadas. Por exemplo, Java é a linguagem de

programação suportada pelo Peixe-Espada, no entanto, não foram avaliados os efeitos sobre técnicas de reflexão (Forman e Forman 2004) e códigos interceptados por programação orientada a aspectos (Laddad 2009), em projetos que por ventura teriam essas técnicas aplicadas.

5.5 – CONSIDERAÇÕES FINAIS

A utilização de projetos reais para a realização do experimento foi considerada um fator determinante para as resposta às questões de pesquisa. Naturalmente, a avaliação em um espaço amostral maior aumentaria o suporte estatístico, no entanto, como observado nos resultados, a aplicação de determinadas refatorações aumenta de forma homogênea alguns valores de métricas, surtindo *reforços positivos* nos atributos de qualidade. O caminho utilizado pela abordagem Peixe-Espada partiu da aplicação de refatorações para a verificação de quais características melhorariam. O caminho inverso também poderia ter sido explorado. Ou seja, a realização de um estudo sobre as métricas poderia indicar a utilização ou criação de refatorações específicas que as melhorassem, dessa forma o sistema saberia a priori quais seriam as refatorações mais indicadas. Ao invés disso, o Peixe-Espada, aprende que *PullUpMethods* melhora NOP e, consequentemente, gera *reforços positivos* em Flexibilidade, Efetividade e Extensibilidade, por exemplo. Contudo, a partir de uma refatoração nova, a abordagem Peixe-Espada permite inferir padrões diretamente pela observação experimental, o que é considerado mais um ponto positivo.

O protótipo criado permitiu que o SMA, sem conhecimento algum, pudesse aprender sobre refatorações e as priorizar em função da melhoria de algum atributo de qualidade. Esse conhecimento adquirido foi essencial para diminuir a utilização das “pesadas” tarefas de entrada e saída da abordagem como medições e refatorações. Como já discutido, as medições atualmente implementadas são tarefas de forte demanda computacional. Esses recursos podem ser melhor aproveitados se o SMA utilizar uma refatoração que tem maiores chances de melhorar o projeto, como é o caso após o aprendizado. A partir do protótipo, também é possível concluir que a relação mais forte ocorre entre refatorações e métricas. Nesse caso, com o conhecimento das métricas que uma refatoração afeta, é possível inferir se ela é boa ou não para melhorar um determinado atributo de qualidade. No entanto, como pode ser observado no experimento, as características do projeto podem fazer uma mesma refatoração afetar os valores das métricas de formas diferentes. Por exemplo, subir a declaração de um método afeta positivamente NOP, mas o comportamento de MFA depende da arquitetura do

projeto: se a superclasse for abstrata e tiver mais de uma subclasse, MFA decresce; se a superclasse não for abstrata e tiver mais de uma subclasse, MFA cresce; se a superclasse tiver apenas uma subclasse, MFA não se altera.

Por fim, era esperado que as refatorações melhorassem algum atributo de qualidade, visto que elas são criadas com o intuito de melhorar determinados aspectos do código fonte. Mesmo considerando que o número de refatorações descartadas foi maior que o número de refatorações aceitas, a abordagem de rejuvenescimento foi eficiente nos contextos utilizados, dado que em todos os casos houve melhoras de atributos de qualidade. Sendo assim, os pontos resumidos nesta seção forneceram respostas positivas para os projetos analisados tanto a questão de pesquisa 1 (“Refatorações automáticas são capazes de rejuvenescer o código fonte? ”), quanto à questão de pesquisa 2 (“em resposta positiva para a questão 1, é possível encontrar padrões entre refatorações e atributos de qualidades”).

CAPÍTULO 6 – CONCLUSÕES

6.1 – EPÍLOGO

O desenvolvimento de bens não tangíveis, como software, gera uma grande facilidade para a manufatura e entrega, divergindo consideravelmente da maioria dos bens tangíveis. Essa divergência também se reflete na manutenção do software, pois é desejável que um projeto de software possua uma flexibilidade muito maior para extensão e correção de defeitos, mesmo depois de sua entrega. Alguns fatores como, por exemplo, excesso de carga de trabalho, evasão ou renovação de mão de obra especializada e mudanças constantes das tecnologias do desenvolvimento, fazem com que equipes frequentemente gerem débito técnico em seus sistemas de software, mesmo adotando metodologias para o desenvolvimento de software de qualidade. Isso ocorre devido às falhas no levantamento de requisitos, falhas na estipulação de prazos, entre outros problemas comumente encontrados no desenvolvimento de software.

Refatorações de software são tarefas rotineiras na maioria dos ambientes de desenvolvimento de software, embora nem sempre sejam utilizadas para diminuir débitos técnicos antigos. Algumas refatorações poderiam ser totalmente automatizadas, evitando eventuais desperdícios de tempo da equipe de desenvolvimento e contribuindo com a diminuição dos erros dessa tarefa muitas vezes não trivial. A existência de refatorações automatizáveis, ou seja, aplicadas e avaliadas automaticamente, são fortes indicadores de que um processo completamente automático para rejuvenescimento de software pode ser concebido.

Tendo em vista que o rejuvenescimento de software é o foco deste trabalho, uma pesquisa foi realizada na literatura para levantar os trabalhos relacionados. Porém, quase a totalidade dos trabalhos encontrados aborda o assunto de maneiras específicas, como: identificação de refatorações automatizáveis; sugestão de reengenharias; sugestão de refatorações; detecção de maus cheiros (do inglês *bad smells*); entre outras as abordagens relacionadas (Seção 2.5). Com esses resultados, percebeu-se na literatura uma carência de abordagens que proporcionassem, ao mesmo tempo, facilidade de extensão e uso contínuo pelas equipes de desenvolvimento para amenizar o envelhecimento precoce de código fonte.

Nesse sentido, foi desenvolvida a abordagem Peixe-Espada, cujo objetivo é amenizar, de forma contínua, o envelhecimento precoce de código fonte aplicando refatorações automatizáveis guiadas por atributos de qualidade. Além disso, a abordagem Peixe-Espada foi desenvolvida para rodar nas máquinas de desenvolvimento em momentos de ociosidade, permitindo a utilização de algumas estratégias de paralelismo, necessárias devido ao uso intensivo dos recursos computacionais envolvidos nas operações de refatoração e medição.

A implementação do Peixe-Espada gerou um protótipo para ser utilizado no ambiente de desenvolvimento, de forma que os desenvolvedores pudessem, a qualquer momento, verificar quais foram as refatorações realizadas, quais valores de métricas foram alterados e em qual ramo do repositório as refatorações bem sucedidas estão incorporadas. A interface web do sistema permite o acompanhamento de todo o SMA, disponibilizando informações como: quais são os agentes trabalhadores ativos; quais são os agentes orquestradores e quais agentes trabalhadores os servem; qual é o aproveitamento de cada grupo de *agentes trabalhadores* na aplicação das refatorações; qual é o aproveitamento acumulado de um *agente orquestrador*; e qual é o conhecimento obtido para cada atributo de qualidade, ou seja, a porcentagem de melhora, piora ou não alteração do atributo de qualidade ao aplicar cada tipo de refatoração.

Por fim, a utilização de G C permitiu o fortalecimento do protótipo por dar suporte a um dos SCV mais utilizados e consolidados (Subversion) juntamente com o Maven, um Sistema de Construção também bastante aceito e consolidado. Além disso, as tarefas da abordagem foram facilitadas devido à utilização das operações bases de um SCV, como o *checkin*, *checkout*, reverter modificações locais e criação de ramos isolados.

6.2 – CONTRIBUIÇÕES

O trabalho mostrou que é possível automatizar a atenuação do envelhecimento precoce de software. A abordagem Peixe-Espada oferece isso por meio de acompanhamento contínuo e melhoria dos atributos de qualidade via refatorações automatizáveis.

Os experimentos permitiram indicar que alguns projetos de diferentes dimensões podem ser beneficiados com a estratégia do Peixe-Espada de refatoração automática. Os experimentos também favoreceram a descoberta de relações entre refatorações e atributos de qualidade. Por exemplo, a refatoração *EncapsulateFields* melhorou Efetividade e Flexibilidade, mas não melhorou Extensibilidade. Por outro lado, *PullUpMethods* é uma boa

refatoração para todos os três atributos de qualidades analisados (Flexibilidade, Extensibilidade e Efetividade).

Outro ponto importante da abordagem Peixe-Espada é a sua capacidade de execução agendada nas máquinas do ambiente de desenvolvimento. A execução distribuída, juntamente com as estratégias de paralelismo e o conhecimento adquirido pelo SMA, foram cruciais para a execução dos experimentos, pois permitiram um melhor aproveitamento dos "caros" recursos computacionais envolvidos nas tarefas de refatoração e medição.

Finalmente, concluí-se que esse trabalho possui as seguintes contribuições objetivas:

- Abordagem para rejuvenescimento automático e contínuo de código fonte avaliada em projetos de diferentes naturezas (acadêmica, open-source e comercial).
- Utilização do parque computacional ocioso do ambiente de desenvolvimento para melhoria dos próprios projetos de software.
- Plataforma para estudo de padrões entre refatorações automatizáveis e atributos de qualidade.
- Viabilização do projeto Oceano como plataforma para implementações acadêmicas relacionadas à GC.
- Protótipo implementado com o ciclo completo da abordagem Peixe-Espada e arquitetura extensível em alguns pontos.

6.3 – LIMITAÇÕES

A implementação da abordagem Peixe-Espada está atualmente limitada a projetos Java *desktop* versionados pelo Subversion. Além disso, o protótipo dá suporte apenas a projetos Maven pelo uso consolidado desta tecnologia no mercado, constantes evoluções e facilidades para realização de tarefas como compilação, gerência de dependências e execução de testes.

Os experimentos consideram apenas os benefícios técnicos de refatorações em termos de arquitetura de software, sem analisar os impactos dessas refatorações automáticas em certas técnicas utilizadas. Por exemplo, Java é a linguagem de programação suportada pelo Peixe-Espada, no entanto, não foram avaliados os efeitos sobre técnicas de reflexão (Forman e Forman 2004) e códigos interceptados por programação orientada a aspectos (Laddad 2009) em projetos que por ventura teriam essas técnicas aplicadas.

Em virtude do fechamento de escopo deste trabalho, o Peixe-Espada se baseou em apenas um modelo de medição para fazer a aceitação das refatorações. O modelo apresentado por Bansiya (2002) define métricas para o cálculo dos seguintes atributos de qualidade: Funcionalidade, Efetividade, Extensibilidade, Flexibilidade, Compreensibilidade e Reusabilidade. Dentre os quais só foram avaliados Extensibilidade, Flexibilidade e Efetividade sob as refatorações *PullUpMethods* e *EncapsulateFields*. Sendo assim, aumentam-se os riscos de imprecisões e as chances de gerar falsos positivos para o ciclo de rejuvenescimento. O número reduzido de refatorações experimentadas também gera consequências negativas, dado que os resultados positivos nessas refatorações demonstram apenas indícios de uma possível abordagem de rejuvenescimento, sem a possibilidade de generalizar para as demais refatorações automatizáveis.

Por fim, não foram realizados experimentos com projetos utilizando cobertura de testes dos mesmos, bem como não foram avaliados os resultados das refatorações bem sucedidas com as equipes de desenvolvimento responsáveis pelos projetos utilizados nos experimentos. Também não foram realizados experimentos com desenvolvedores para avaliar a usabilidade do protótipo como um todo e, consequentemente, a possibilidade do seu uso contínuo.

6.4 – TRABALHOS FUTUROS

O desenvolvimento da abordagem Peixe-Espada gerou uma gama de possibilidades para trabalhos futuros. Um trabalho futuro natural é a extensão da abordagem com a implementação de novas refatorações automatizáveis e outros modelos de atributos de qualidade, dado que a arquitetura do Peixe-Espada foi criada para facilitar essas extensões. Outro trabalho natural seria a realização de um estudo detalhado com as equipes responsáveis pelos projetos utilizados nos experimentos, a fim de avaliar tanto as refatorações automatizáveis quando as métricas e o grau de rejuvenescimento, caso as avaliações tenham sido positivas. Contudo, devido à existência da abordagem e sua implementação, é possível enumerar outros trabalhos futuros com contribuições acadêmicas de relevâncias mais acentuadas, como por exemplo:

Aplicação de *Search-based Software Engineering* (SBSE) – O Peixe-Espada não foi implementado com preocupações explícitas de SBSE. No entanto, seria interessante que maiores preocupações com otimizações e desempenho sejam consideradas utilizando

conceitos existentes de SBSE ou mesmo criando abordagens específicas para este trabalho. A maneira que o protótipo se comporta para realizar as refatorações não possui avaliação de máximos locais e globais, por exemplo. Em outras palavras, dada uma lista de *sintomas de refatoração*, o Peixe-Espada tenta aplicar cada uma das refatorações indicadas pelos sintomas, verificando se melhorou o atributo de qualidade em relação à *configuração corrente*. A refatoração é aceita apenas se o atributo de qualidade melhorar, passando a ser a *configuração corrente* e sendo integrada a um ramo do repositório do SCV. Nesse sentido, uma nova estratégia poderia guardar os projetos refatorados em uma estrutura de grafo, mesmo quando os valores de atributos de qualidade não fossem melhores. Assim, outras refatorações indicadas pelos *sintomas de refatoração* poderiam ser aplicadas sobre essas versões já refatoradas contidas no grafo segundo algum critério de busca, na esperança de obter atributos de qualidade melhores do que as refatorações que melhoram imediatamente a configuração atual. Outra forma de uso de SBSE seria com o Peixe-Espada tendo multiobjetivos, ou seja, ter como objetivo a melhoria de mais de um atributo de qualidade para uma mesma sequência de refatoração ao mesmo tempo. Nesse caso, algoritmos de busca poderiam ser utilizados para manter a melhoria evitando conflitos entre atributos de qualidade.

Auxílio na utilização de refatorações não automatizáveis – Uma vez que refatorações são constantes no contexto de desenvolvimento de software, utilizar a abordagem Peixe-Espada de forma interativa, para obter os dados subjetivos que tornam uma refatoração não automatizável, pode ser muito relevante. Dessa maneira, a refatoração aplicada poderia reutilizar toda a infraestrutura já disponível no Peixe-Espada, como a informação de qual atributo de qualidade melhorou, piorou ou não alterou, visualização de relatórios, criação de ramos isolados no repositório de versões, etc.

Maior exploração do paralelismo – A medição de um atributo de qualidade e a aplicação de uma refatoração em si não são paralelizáveis. A utilização de técnicas de programação paralela poderia melhorar o desempenho da abordagem. Contudo, não foi realizada nenhuma busca na literatura sobre essa questão.

Mapeamento de características dos projetos que influenciam nas refatorações e atributos de qualidade – Não foram realizados estudos para mapear quais são as características que levaram a aplicação de uma mesma refatoração gerar resultados diferentes para projetos distintos em relação a um mesmo atributo de qualidade. Esse mapeamento pode ser utilizado como informação que, agregada ao conhecimento adquirido pelo SMA, ensinaria

os agentes a realizar refatorações com maiores chances de melhora do projeto. Isso seria possível através de uma varredura automática feita no projeto antes dos agentes trabalhadores iniciarem as refatorações.

Estabelecimento de limites para aceitação/rejeição das refatorações – Atualmente, o Peixe-Espada faz medições para a aplicação de cada *refatoração mínima*. Sendo assim, qualquer alteração nos valores das métricas, por menor que seja, é considerada para a verificação de *reforço positivo* ou *negativo*. Nesse caso, é esperado que o tamanho do projeto seja inversamente proporcional à margem de alteração dos valores das métricas, ou seja, em um projeto grande, com muitos *sintomas* e provavelmente muitas aplicações, as *refatorações mínimas* afetarão pouco às medidas, porém atualmente qualquer alteração é considerada. Um trabalho nesse sentido poderia avaliar melhor a relevância das métricas alteradas, de forma a estabelecer limites tanto para a aceitação como para rejeição da aplicação de uma refatoração.

Novos experimentos – Por fim, o experimento realizado no Peixe-Espada foi restrito a responder às questões de pesquisas deste trabalho. No entanto, a realização de experimentos a partir de processos diversificados pode gerar resultados significantes para a abordagem. Nesse caso, é possível enumerar alguns experimentos: (1) Ativar/desativar o conhecimento do SMA para validar a relevância do seu uso; (2) Mapear os casos de fracasso (piora ou não alteração dos atributos de qualidade) para reforçar o estudo atualmente feito; (3) Executar o Peixe-Espada em ambientes de desenvolvimento por tempos determinados, a fim de analisar o paralelismo e a efetividade da abordagem; (4) Executar várias sequências de refatorações, a fim de analisar o comportamento dos valores dos atributos de qualidade.

REFERÊNCIAS

Abdeen, H. and Ducasse, S. and Sahraoui, H. and Alloui, I., (2009), "Automatic Package Coupling and Cycle Minimization". In: *Working Conference on Reverse Engineering (WCRE)*, p. 103–112, Washington, DC, USA.

ABNT, (2003), *Engenharia de software - Qualidade de produto Parte 1: Modelo de qualidade*, Associação Brasileira de Normas Técnicas. Disponível em: <http://www.abntcatalogo.com.br/norma.aspx?ID=2815>.

Alur, D. and Malks, D. and Crupi, J., (2003), *Core J2EE Patterns: Best Practices and Design Strategies*. 2 ed. Prentice Hall.

Aranda, B. and Wadia, Z., (2008), *Facelets Essentials: Guide to JavaServer(TM) Faces View Definition Framework*. 1 ed. Apress.

Bansiya, J. and Davis, C. G., (2002), "A Hierarchical Model for Object-Oriented Design Quality Assessment", *IEEE Transactions on Software Engineering*, v. 28, n. 1, p. 4-17.

Bansiya, J. and Etzkorn, L. H. and Davis, C. and Li, W., (1999), "A Class Cohesion Metric For Object-Oriented Designs", *Journal of Object-Oriented Programming (JOOP)*, p. 47-52.

Bauer, C. and King, G., (2004), *Hibernate in Action*. Manning Publications.

Bauer, M. and Karlsruhe, F., (1999), "Analyzing Software Systems by Using Combinations of Metrics". In: *European Conference on Object-Oriented Programming (ECOOP), Workshop on Experiences in Object-Oriented Re-Engineering*, p. 170 - 171, Springer-Verlag London, UK.

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

Beck, K., (2002), *Test Driven Development: By Example*. Addison-Wesley Professional.

Booch, G., (1990), *Object Oriented Analysis and Design with Applications*. Benjamin-Cummings Publishing Company, Subs of Addison Wesley Longman, Inc.

Brand, M. van den and Visser, E., (1996), "Generation of formatters for context-free languages", *ACM Trans. Softw. Eng. Methodol.*, v. 5, n. 1 (jan.), p. 1–41.

Brooks, F. P. J., (1987), "No Silver Bullet Essence and Accidents of Software Engineering", *IEEE Computer*, v. 20, n. 4, p. 10-19.

Chrissis, M. B. and Konrad, M. and Shrum, S., (2006), *CMMI(R): Guidelines for Process Integration and Product Improvement*. 2 ed. Addison-Wesley Professional.

Coad, P. and Yourdon, E., (1990), *Object Oriented Analysis (2nd Edition)*. 2 ed. Prentice Hall PTR.

Company, S., (2008), *Maven: The Definitive Guide*. 1 ed. O'Reilly Media.

Czibula, I. G. and Czibula, G., (2008), "Clustering Based Automatic Refactorings Identification". *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, p. 253-256, Washington, DC, USA.

Dart, S., (1991), "Concepts in Configuration Management Systems". In: *International Workshop on Software Configuration Management (SCM)*, p. 1-18, Trondheim, Norway.

Donaldson, J. L. and Lancaster, A.-M. and Sposato, P. H., (1981), "A plagiarism detection system", *SIGCSE Bull.*, v. 13, n. 1 (fev.), p. 21–25.

Farley, J., (2007), *Practical JBoss Seam Projects*. 1 ed. Apress.

Fenton, N. E. and Pfleeger, S. L., (1998), *Software Metrics: A Rigorous and Practical Approach, Revised*. 2 ed. Course Technology.

Ferber, M. and Hunold, S. and Krellner, B. and Rauber, T. and Reichel, T. and Rünger, G., (2009), "Reducing the Class Coupling of Legacy Code by a Metrics-Based Relocation of Class Members". *Central and East European Conference on Software Engineering Techniques (CEE-SET)*, Krakow, Poland.

Finkbine, P. D., (1996), "Metrics and Models in Software Quality Engineering", *ACM SIGSOFT Software Engineering Notes*, v. 21 (jan.), p. 89.

Forman, I. R. and Forman, N., (2004), *Java Reflection in Action*. Manning Publications.

Fowler, M. and Beck, K. and Brant, J. and Opdyke, W. and Roberts, D., (1999), *Refactoring: Improving the Design of Existing Code*. 1 ed. Addison-Wesley Professional.

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

Geary, D. and Horstmann, C. S., (2004), *Core JavaServer Faces*. Prentice Hall.

Harman, M. and Clark, J., (2004), "Metrics Are Fitness Functions Too". In: *International Symposium on Software Metrics (Metrics)*, p. 58–69, Washington, DC, USA.

Harman, M. and Mansouri, S. A. and Zhang, Y., (2009), "Search based software engineering: A comprehensive analysis and review of trends techniques and applications", *Department of Computer Science, King's College London, Tech. Rep. TR-09-03*

IEEE, (1990), *Std 610.12 - IEEE Standard Glossary of Software Engineering Terminology*, Institute of Electrical and Electronics Engineers.

Inc, C. P. and Worsley, J. C. and Drake, J. D., (2002), *Practical PostgreSQL*. O'Reilly Media.

ISO, (2001), *ISO/IEC 9126 - Software engineering - Product quality*, International Organization for Standardization.

Kan, S. H., (2002), *Metrics and Models in Software Quality Engineering*. 2 ed. Addison-Wesley Professional.

Katz, M., (2008), *Practical Richfaces [PRAC RICHFACES NEW/E]*. Apress.

Kerievsky, J., (2004), *Refactoring to Patterns*. Addison-Wesley Professional.

Khomh, F. and Vaucher, S. and Guéhéneuc, Y.-G. and Sahraoui, H., (2009), "A Bayesian Approach for the Detection of Code and Design Smells". In: *International Conference on Quality Software (QSIC)*, p. 305-314, Los Alamitos, CA, USA.

Kirkpatrick, S. and Gelatt, C. D. and Vecchi, M. P., (1983), "Optimization by Simulated Annealing", *Science*, v. 220, n. 4598 (maio.), p. 671 -680.

Klinger, T. and Tarr, P. and Wagstrom, P. and Williams, C., (2011), "An enterprise perspective on technical debt". In: *2nd Working on Managing Technical Debt (MTD)*, p. 35–38, Waikiki, Honolulu, HI, USA.

Lacerda, E. G. M. and Carvalho, A. C. P. L. F., (1999), "Introdução aos Algoritmos Genéticos", *Anais XIX Congresso Nacional da Sociedade Brasileira de Computação*

Laddad, R., (2009), *Aspectj in Action: Enterprise AOP with Spring Applications*. Second Edition ed. Manning Publications.

Laguë, B. and April, A., (1996), "Mapping of Datrix™ software metrics set to ISO 9126 Maintainability sub-characteristics", *Software Engineering Standards Issues (SES'96)*

Mak, G. and Rubio, D. and Long, J., (2010), *Spring Recipes: A Problem-Solution Approach*. 2 ed. Apress.

Mancoridis, S. and Mitchell, B. S. and Rorres, C. and Chen, Y. and Gansner, E. R., (1998), "Using Automatic Clustering to Produce High-Level System Organizations of Source Code". *International Workshop on Program Comprehension (IWPC)*, p. 45–52, Washington, DC, USA.

Marinilli, M., (2001), *Java Deployment with JNLP and WebStart*. Sams.

Mens, T., (2002), "A State-of-the-Art Survey on Software Merging", *IEEE Transactions on Software Engineering*, v. 28, n. 5, p. 449-462.

Meyer, M., (2006), "Pattern-based Reengineering of Software Systems". *Working Conference on Reverse Engineering (WCRE)*, p. 305–306, Washington, DC, USA.

Moore, I., (1996), "Automatic inheritance hierarchy restructuring and method refactoring". In: *ACM SIGPLAN Notices*, p. 235–250, New York, NY, USA.

Moura, A. M. M., (2009), *Uma Abordagem de Reengenharia de Sistemas Orientados a Objetos para Componentes Baseada em Métricas*. Mestrado, UFRJ, COPPE

Murta, L. G. ., (2006), *Gerência de Configuração no Desenvolvimento Baseado em Componentes*. Tese de Doutorado, UFRJ, COPPE, D.Sc.; Tese de D.Sc.

Murta, L. G. P. and van der Hoek, A. and Werner, C. M. L., (2006), "ArchTrace: A Tool for Keeping in Sync Architecture and its Implementation". In: *Brazilian Symposium on Software Engineering (SBES), Tools Session*, p. 127-132, Florianópolis, Brazil.

Nock, C., (2003), *Data Access Patterns: Database Interactions in Object-Oriented Applications*. 1 ed. Addison-Wesley Professional.

O'Keeffe, M. and Ó Cinnéide, M., (2008), "Search-based refactoring for software maintenance", *Journal of Systems and Software*, v. 81, n. 4 (abr.), p. 502-516.

O'Keeffe, M. and Ó Cinnéide, M. O., (2004), "Towards automated design improvement through combinatorial optimisation", *Workshop on Directions in Software Engineering Environments (WoDiSEE)*

De Oliveira, A. A. and Braga, T. H. and De Almeida Maia, M., (2004), "MetaJ: An Extensible Environment for Metaprogramming in Java", *Journal of Universal Computer Science (JUCS)*, p. 872-891.

Parnas, D. L., (1994), "Software aging". In: *International Conference on Software Engineering (ICSE)*, p. 279–287, Boston, MA, USA.

Partsch, H. and Steinbrüggen, R., (1983), "Program Transformation Systems", *ACM Comput. Surv.*, v. 15, n. 3 (set.), p. 199–236.

Pressman, R. S., (2004), *Software Engineering: A Practitioner's Approach*. 6 ed. McGraw-Hill Science/Engineering/Math.

Rao, A. S. and Georgeff, M. P., (1995), "BDI agents: From theory to practice". In: *International Conference on Multi-Agent Systems (ICMAS)*, p. 312–319, San Francisco, California.

Sagonas, K. and Avgerinos, T., (2009), "Automatic refactoring of Erlang programs". In: *International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, p. 13–24, New York, NY, USA.

Santos, H. K. R., (2008), *Modularização de Interesses Ortogonais com AspectJ*. Monografia, Universidade Federal Fluminense - UFF

Seng, O. and Stammel, J. and Burkhart, D., (2006), "Search-based determination of refactorings for improving the class structure of object-oriented systems". In: *Conference on Genetic and Evolutionary Computation (GECCO)*, p. 1909–1916, Washington,USA.

Shaw, M. and Garlan, D., (1996), *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.

Simon, F. and Steinbrückner, F. and Lewerentz, C., (2001), "Metrics Based Refactoring". In: *European Conference on Software Maintenance and Reengineering (CSMR)*, p. 30, Washington, DC, USA.

Smart, J. F., (2010), *Continuous Integration with Hudson*. O'Reilly Media, Inc.

Softex, (2009), *MPS.BR - Melhoria de Processo do Software Brasileiro - Guia Geral*, Associação para Promoção da Excelência do Software Brasileiro. Disponível em: <http://www.softex.br/mpsbr/_home/default.asp>.

Tahvildar, L. and Kontogiannis, K., (2004), "Improving design quality using meta-pattern transformations: a metric-based approach: Research Articles", *Journal of Software Maintenance and Evolution: Research and Practice (JSMERP)*, v. 16 (jul.), p. 331–361.

Tahvildari, L., (2004), "Quality-Driven Object-Oriented Re-Engineering Framework". In: *IEEE International Conference on Software Maintenance*, p. 479–483, Washington, DC, USA.

Taneja, K. and Dig, D. and Xie, T., (2007), "Automated detection of api refactorings in libraries". In: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, p. 377–380, New York, NY, USA.

Tourwe, T. and Mens, T., (2003), "Identifying refactoring opportunities using logic meta programming". In: *European Conference on Software Maintenance and Reengineering (CSMR)*, p. 91-100, Washington, DC, USA.

Tsantalis, N. and Chatzigeorgiou, A., (2010), "Identification of refactoring opportunities introducing polymorphism", *Journal of Systems and Software*, v. 83, n. 3 (mar.), p. 391-404.

Vikram Jamwal and Sridhar Iyer, (2005), "Automated refactoring of objects for application partitioning". *Asia-Pacific Software Engineering Conference (APSEC)*, p. 671 - 678.

Wise, M. J., (1992), "Detection of similarities in student programs: YAP'ing may be preferable to plague'ing". In: *SIGCSE Technical Symposium on Computer Science Education*, p. 268–271, Kansas City, Missouri, United States.

Wooldridge, M. and Ciancarini, P., (2001), "Agent-oriented software engineering: the state of the art". , p. 1-28, Limerick, Ireland.

Wuyts, R., (2001), *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. Ph.D Disponível em: <scg.unibe.ch/archive/phd/Wuyts-phd.pdf>.

Yang, D., (2010), *Java(TM) Persistence with JPA*. Outskirts Press.

Zambonelli, F. and Jennings, N. R. and Wooldridge, M., (2001), "Organizational abstractions for the analysis and design of multi-agent system". In: *International Workshop, on Agent-Oriented Software Engineering (AOSE)*, p. 235–251., Secaucus, NJ, USA.