UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
COORDENAÇÃO DE INFORMÁTICA
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

SANDRO GOMES DE SOUZA

PROMOVENDO MELHORIAS EM CÓDIGO EXISTENTE: UM ESTUDO DE CASO COM REFATORAÇÕES E PADRÕES

TRABALHO DE DIPLOMAÇÃO

PONTA GROSSA
2011

SANDRO GOMES DE SOUZA

PROMOVENDO MELHORIAS EM CÓDIGO EXISTENTE: UM ESTUDO DE CASO COM REFATORAÇÕES E PADRÕES

Trabalho de Conclusão de Curso de Graduação, apresentado à disciplina de Trabalho de Diplomação, do Curso Superior de Tecnologia Análise e Desenvolvimento de Sistemas – COADS – da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Msc. Rogério Ranthum

PONTA GROSSA

2011

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Ponta Grossa

Diretoria de Graduação e Educação Profissional
Coordenação do Curso Superior de Tecnologia em
Análise e Desenvolvimento de Sistemas

TERMO DE APROVAÇÃO

PROMOVENDO MELHORIAS EM CÓDIGO EXISTENTE: UM ESTUDO DE CASO COM REFATORAÇÕES E PADRÕES

por

SANDRO GOMES DE SOUZA

Este Trabalho de Conclusão de Curso foi apresentado em 17 de novembro de 2011 como requisito parcial para a obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Prof. Msc. Rogério Ranthum
Prof. Orientador

_____
Profª. Drª. Simone Nasser Matos
Membro titular

_____
Prof. Cristian C. R. de Abreu
Membro titular

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

“Este trabalho é devotado a todos aqueles que em algum momento de sua vida pensaram em desistir, se sentiram perdidos e desamparados, beirando o total fracasso.

Nunca deixem de seguir em frente, pois nada é mais revigorante do que descobrir que estávamos errados.” (Autoria Desconhecida)

RESUMO

SOUZA, Sandro G. **Promovendo melhorias em código existente:** um estudo de caso com refatorações e padrões. 2011. 69 f. Trabalho de Conclusão de Curso (Graduação em Tecnologia em Análise e Desenvolvimento de Sistemas) – Programa de Graduação em Tecnologia, Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2011.

Sistemas de *software* desenvolvem-se sob os pilares da engenharia de *software*, entretanto são pressionados pelos curtos prazos das organizações e outros fatores construindo um cenário em que os sistemas produzidos satisfazem os requisitos, porém o código-fonte se torna confuso, inflexível, de difícil manutenção e evolução. Este trabalho apresenta e conceitua algumas das consequências e áreas envolvidas neste cenário. Apresenta, de forma compassada e através de um estudo de caso, o uso de técnicas de refatoração aliadas a padrões de projetos criacionais em uma aplicação simples. Afere algumas métricas para as mudanças ocorridas e aponta o valor desta solução como resposta ao cenário mencionado.

Palavras-chave: Refatoração, Padrões de projeto, Engenharia de Software, Métricas de Software.

ABSTRACT

SOUZA, Sandro G. **Promoting improvements in existing code:** a case study with refactorings and patterns. 2011. 69 p. Trabalho de Conclusão de Curso (Graduation in Technology Analysis and Systems Development) - Graduate Program in Technology, Federal Technological University of Paraná. Ponta Grossa, 2011.

Software systems develop under them pillars of the software engineering, however are pressured by the short term of the organizations and other factors creating a scenario where the produced systems meet the requirements, but the source-code becomes confused, inflexible, of difficult maintenance and evolution. This work presents and conceptualizes some of consequences and involved areas in this scene. It presents, of trimmed form and through a case study, the use of techniques of refactoring allied with creational *design* patterns in a simple application. Applies some metric for the occurred changes and points the value of this solution as reply to the mentioned scenario.

Keywords: Refactoring, Software Engineering, Software Patterns, Software Metrics.

SUMÁRIO

1 INTRODUÇÃO

Com o surgimento da engenharia de *software* e sua área de conhecimento chamada qualidade de *software*, ficou explícita a afirmação de que o desenvolvimento de um *software* não se limita apenas a codificação do mesmo.

Diversas outras etapas e processos foram inseridos antes e depois da codificação, de forma a garantir um desenvolvimento sistemático dos *softwares*. Como por exemplo, as etapas de planejamento, análise de risco, engenharia e avaliação do cliente em um processo seguindo o modelo espiral.

Mesmo assim, segundo Rapeli (2006), necessidades, curtos prazos das organizações e falta de uso da tecnologia adequada podem comprometer o ciclo de vida do *software*, tornando-os não bem projetados, e quando isso ocorre, maiores são os custos de manutenção do *software*.

Manutenção de *software* é o processo onde são descobertos novos problemas e requisitos. De acordo com Lientz e Swanson (1980, apud Rapeli, 2006, p.13) ela consiste em quatro tipos: corretiva, adaptativa, perfectiva e preventiva. Pressman (2006) afirma que 80% da fase de manutenção são de adaptação do sistema ao meio externo. Higo et al. (2003) colabora com a afirmação de que a manutenção de *software* é a atividade mais cara durante o processo de desenvolvimento de *software*.

Como forma de contornar estes ônus surgem saídas como as refatorações e o uso de padrões de *software*. Refatoração "é o processo de alteração de um sistema de *software* de modo que o comportamento do código não mude, mas que sua estrutura interna seja melhorada." (FOWLER, 2002, p. 9).

Como o processo de refatoração não prevê adição de novas funcionalidades, aplicá-lo no projeto inicial pode consumir um tempo não disponibilizado pelas organizações. De acordo com Fowler (2002), o projeto inicial é uma predição onde os requisitos estarão incompletos. Em contrapartida, à medida que o *software* se aproxima do produto final se torna mais difícil modificá-lo ou corrigi-lo, o que dificulta a sua manutenção e evolução.

Uma das possíveis soluções para minimizar estes problemas que afetam desenvolvedores e engenheiros de *software* são o uso de refatorações e padrões de projeto. Freire e Cheque (2007) avaliam que a refatoração torna o código mais fácil de ser entendido e menos custoso de ser alterado, garantindo simplicidade, flexibilidade e clareza. Frequentemente o *software* reutilizável tem que ser reorganizado, os padrões de projeto auxiliam a determinar como reorganizar um projeto.

Fundamentado nas premissas abordadas, torna-se necessário propor melhorias em um sistema existente de forma a melhorar a sua manutenabilidade e garantir o reuso da solução de forma a não comprometer o seu comportamento atual auxiliando os engenheiros e desenvolvedores de *softwares* na imprescindível tarefa de manutenção, explorando a reusabilidade de códigos orientados a objetos e implementados na linguagem Java, garantindo a evolução do *software*.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Diagnosticar através de um estudo de caso possíveis deficiências em uma aplicação e inserir refatorações simples que culminem em padrões de *software* ou em melhor abstração do sistema, facilitando a sua evolução, reusabilidade e manutenabilidade.

1.1.2 Objetivos Específicos

- Definir uma aplicação para o estudo de caso;
- Elencar possíveis deficiências da aplicação escolhida;
- Observar necessidade de padrões de criação ou da camada de integração;
- Aplicar refatorações para contornar o problema;
- Aferir métricas sobre o estudo, colhendo os benefícios observados.

1.2 JUSTIFICATIVA

Segundo Fowler (2002), refatoração reduz a quantidade de correções a serem executadas, facilitando a etapa de testes.

O uso de refatoração garante a redução de tempo no desenvolvimento do software, o que garante menores custos para a organização.

A manutenção, reusabilidade e evolução do software por meio de novas funções tendem a sofrer poucos erros de codificação.

Refatoração provê código mais limpo e fácil de ser entendido por outros programadores, otimizando a revisão de código.

O constante uso de refatoração e padrão de projetos favorecem o hábito de programação de acordo com as boas práticas de uma linguagem de programação.

Além de se alcançar um melhor *design*, refatorações podem obter um sistema estruturado de acordo com um padrão de projeto.

Refatoração é uma maneira de reestruturar o software para fazer descobertas de *design* mais explicitas e extrair componentes reutilizáveis, esclarecendo a arquitetura do sistema e preparando-o para ser feitas adições de modo mais fácil (FOWLER, 2000).

Refatoração é capaz de dinamizar o programa e reduzir duplicação de código no programa, podendo alavancar um investimento passado.

Prevenir o envelhecimento do *design* e garantir a flexibilidade adequada para permitir a integração tranquila de futuras extensões/alterações (MENS; TOURWÉ, 2004).

1.3 ORGANIZAÇÃO DO TRABALHO

O trabalho foi organizado em seis capítulos descritos a seguir.

O capítulo 2 fornece uma conceituação acerca de refatorações, engenharia de software, padrões de projetos e métrica de software necessária para o desenvolvimento do estudo.

O capítulo 3 aborda todo o processo de desenvolvimento do estudo de caso. Enumera as características da aplicação a ser melhorado, o diagnóstico e a metodologia abordada para o mesmo. Aplicam de forma pormenorizada as refatorações e os padrões de projeto elencados: *Factory Method*, *Abstract Factory*, *Singleton e Data Access Object*.

O capítulo 4 analisa o aspecto final da aplicação e exibe dados obtidos de sua avaliação traçando um comparativo entre as mudanças ocorridas. Realça também, as dificuldades encontradas em meio ao projeto como um todo.

O capitulo 5 elucida por meio de considerações finais todo o agregado de pontos de vista e asserções delimitadas ou encontradas durante o trabalho acadêmico.

O capítulo 6 apresenta as oportunidades de abordagem do tema em trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo relaciona alguns conceitos relacionados à:

- Refatoração*:* definições, seus principais elementos, uma visão geral dessa técnica.
- Engenharia de *software*: uma breve conceituação sobre a disciplina, a necessidade e suas áreas de aplicação.
- Métricas de *software*: conceitos e principais métricas utilizadas neste trabalho.
- Padrões de Projeto: história, conceitos, ramificações, principais características encontradas na literatura, detalhamento dos padrões *Factory Method*, *Abstract Factory*, *Singleton e Data Access Object*.

2.1 REFATORAÇÃO

Muitos são os instrumentos disponíveis para a construção de programas nos dias de hoje: diversas metodologias, *frameworks*, linguagens, ferramentas e ambientes de desenvolvimento, dentre outros que facilitam, e muito, sua elaboração. Todavia, um programa já desenvolvido e que esteja funcionando sem erros, mesmo atendendo aos requisitos dos clientes, não significa que o mesmo foi bem codificado, podendo originar uma estrutura interna de leitura difícil e lógica confusa.

Para promover uma melhora e diversos outros benefícios nessa estrutura interna são utilizadas as refatorações.

2.1.1 Definindo Refatoração

A refatoração, segundo Rapeli (2006), se constitui em uma das técnicas de manutenção para o software desenvolvido com o paradigma orientado a objetos.

A refatoração começou em começou nas áreas de desenvolvimento, onde programadores de orientação a objeto, usando *Smalltalk*[1], encontraram situações em que técnicas foram necessárias para oferecer melhor suporte no processo de desenvolvimento de *framework*[2] ou, mais genericamente, para apoiar o processo de mudança, porque um *framework* dificilmente vai estar certo na primeira vez, ele precisa evoluir para ganhar

[1] Linguagem de programação orientada a objetos fracamente tipada, uma das primeiras criadas.

[2] Conjunto de conceitos usado para resolver um problema de um domínio específico.

experiência, ou seja, seu código deverá ser lido e modificado com mais frequência do que codificado. A chave para manter o código legível e modificável é a refatoração, no particular aos *frameworks*, mas a aplicabilidade desta técnica amadureceu ao ponto em que um conjunto mais amplo de profissionais de software pode experimentar os benefícios da refatoração (FOWLER, 2001).

A refatoração é utilizada com *extreme programming*[3] para permitir a quebra do sistema em muitos objetos e métodos pequenos, reduzindo a chance do par de programadores mudarem a mesma classe ou método ao mesmo tempo. Refatorar ao longo de todas as iterações de um projeto torna a arquitetura simples e fácil de ser alterada, viabilizando o uso de uma arquitetura evolutiva (TELES, 2005).

A palavra refatoração tem duas definições dependendo do contexto. Refatoração, como substantivo, é a mudança feita na estrutura interna do software para fazê-la fácil de entender e mais barato de modificar sem mudanças observáveis no seu comportamento. Refatorar, como verbo, é a reestruturação do software por meio da aplicação de uma série de refatorações sem mudança observável em seu comportamento (FOWLER, 2001).

A refatoração intitula-se no processo de modificar um sistema de software de uma maneira que não altere o comportamento externo do código e ainda melhore a estrutura interna. Geralmente são pequenas modificações de *software*, feitas passo a passo e de maneira sistemática. Entretanto, uma refatoração pode envolver outras. Torna-se importante refatorar o software nesses pequenos passos, pois se é cometido algum erro na refatoração, é facilmente detectável.

Ainda dentro dos aspectos conceituais a cerca da refatoração, quando aplicada, o *design* do projeto tende a serem melhorados nos requisitos não funcionais de qualidade de software como simplicidade, flexibilidade, clareza e desempenho (KON; GOLDMAN, 2004).

Para Fowler (2001), em contraste com a otimização de desempenho, que também altera a estrutura interna do software, a única transformação feita para tornar o software mais fácil de ser entendido é a Refatoração. Embora a otimização de desempenho não mude o comportamento de um componente (a não ser a velocidade), os propósitos são diferentes. Visto que a otimização de desempenho quase sempre torna o código difícil de entender, em detrimento da otimização pretendida. Desta forma, um trecho de código refatorado sempre

[3] Metodologia ágil para desenvolvimento de *softwares* com requisitos vagos e em constante mudanças.

terá um melhor entendimento e produzirá os mesmos resultados finais encontrados antes da refatoração.

Cornélio (2004) aponta como pioneira na formalização da refatoração de programas orientados a objeto, a obra de Willian Opdyke em 1992. No trabalho citado identificam-se vinte e três refatorações primitivas e três exemplos de refatoração composta. Sendo que para cada refatoração existe um conjunto de pré-condições requeridas para o seu uso, garantindo a preservação de comportamento da transformação do código.

2.1.2 Adicionar funções e refatorar

O processo de desenvolvimento de *software* é permeado com a adição de novas funções e testes para se adequarem a uma determinada especificação para um dado requisito. Fowler (2001) aponta que o uso da refatoração no desenvolvimento de *software* divide este tempo em duas atividades distintas: adição de funções e refatoração.

Quando se adiciona função não se deve realizar mudança no código existente, apenas a adição da nova função e se preciso, a adição de teste para mensurar o progresso. Para Fowler (2001) quando se refatora, não há adição de função, somente reestruturação do código e não se adiciona testes no projeto (a menos que seja observada a falta de alguma na atividade de adição de função).

Fowler (2001) afirma que somente se muda um teste na atividade de refatoração quando é absolutamente necessário, a fim de lidar com uma mudança de interface. No decorrer do desenvolvimento do software estas duas atividades se intercalam, atendendo às necessidades de cada processo.

2.1.3 Reorganizando o projeto e evoluindo o *Software*

Quando um *software* começa a ser utilizado, sua evolução é direcionada para satisfazer mais requisitos e ser mais reutilizável. De forma geral, novos requisitos aumentam a quantidade de classes e operações. Este processo de crescimento torna o *software* muito inflexível quanto às mudanças, pois as classes definirão muitas operações e variáveis de instância não-relacionadas contendo uma hierarquia de classes que não mais corresponde a um domínio genérico de um determinado problema, mas sim a um conjunto de domínios e de maneira desordenada.

Frequentemente o software reutilizável tem que ser reorganizado ou refatorado. Os padrões de projeto auxiliam a determinar como reorganizar um projeto, reduzindo o volume de futuras refatorações (FOWLER, 2001).

Os padrões de projeto apresentados por Gamma (2000) representam muitas das estruturas resultantes da refatoração. O uso destes, no início da vida de um projeto, evita refatorações mais tarde. Mas mesmo que não se tenha aplicado um padrão a não ser após ter construído o sistema, ainda assim ele pode mostrar como mudá-lo.

Análogo aos padrões de projeto orientado a objetos proposto por Gama et al. (2000), as técnicas de refatoração podem ser agrupadas em catálogos.

A diferença entre padrões de projeto e as técnicas de refatoração, de acordo com Cornélio (2004) está na observação que as refatorações são encaradas como regras, enquanto um padrão de projeto é uma estratégia de desenvolvimento, pois captura o conhecimento do especialista de *software*. Uma refatoração precisa satisfazer um conjunto de pré-condições para que se preserve o comportamento do programa, padrões de projeto, entretanto, são os possíveis objetivos de uma refatoração.

Fowler (2001) apresenta um catálogo onde para cada refatoração é dado um nome e um resumo curto que o descreva, uma motivação que descreve porque a refatoração deve ser feita, uma mecânica, uma descrição passo a passo de como realizar refatorações, e, finalmente, um exemplo.

Antes de começar a refatorar, deve existir uma sólida suíte de testes[4] que devem ser *self-checking* onde cada mudança deve ser seguida pela compilação e pelo teste do programa. Não há nenhuma condição a ser satisfeita a fim garantir a preservação do comportamento, na realidade, a aproximação de Fowler à refatoração é baseada em compilação e ciclos de teste.

2.1.4 Aspectos positivos da refatoração

Refatorar pode e deve ser utilizada por diversos motivos, descritos a seguir:

2.1.4.1 *Refatorar melhora o projeto de software*

Sem a refatoração, o projeto do programa se deteriora, mudanças no código sejam para atingir objetivos em curto prazo ou aquelas feitas sem a completa compreensão do projeto retira a estrutura do código. A leitura se torna mais difícil e quanto mais

[4] Coleção de testes de *software* que indicam um conjunto específico de comportamento.

incompreensível o *design* do código, mais difícil de preservar o comportamento e mais rápido o programa enfraquece. O uso regular da técnica de refatoração ajuda o código do projeto a manter a sua estrutura.

A codificação mal projetada geralmente precisa de mais códigos para fazer as mesmas coisas em diversos lugares. Assim, um aspecto importante na melhoria do projeto é a eliminação do código duplicado. A importância desta se encontra nas modificações futuras ao código. Reduzir a quantidade de código não fará o sistema funcionar mais rapidamente, porque o efeito no *footprint*[5] dos programas é raramente significativo. Reduzir a quantidade de código, entretanto, faz uma diferença grande na manutenção do código: Quanto mais código há, mais difícil de modificá-lo corretamente, pois há mais código a compreender, eliminando as duplicatas é garantida a execução do trecho de código uma única vez. Essa é a essência de um projeto bom (FOWLER, 2001).

2.1.4.2 *Refatorar torna o software fácil de ser entendido.*

Desenvolvedores quase sempre estão focados na adição das funcionalidades e, deveras pressionados pelos prazos, negligenciam a legibilidade do código-fonte e o fato de um futuro desenvolvedor precisar fazer mudanças adicionando tempo desnecessário à manutenção.

Mesmo antes da versão final ainda existem muitas funcionalidades a serem testadas e aplicadas, tornando necessário o entendimento do sistema.

A refatoração auxilia a transformar o código-fonte mais legível e não altera seu comportamento interno, permitindo que o futuro programador, mesmo sem ter escrito aquele código, possa mudá-lo internamente conforme o seu próprio discernimento e visão do projeto (FOWLER, 2001).

2.1.4.3 *Refatorar auxilia na procura por erros.*

Quando se aplica a refatoração, existe um esforço para compreender o código-fonte para torná-lo mais simples e legível. Esta nova compreensão que se estabelece, conforme Fowler (2001), esclarece a estrutura do programa tanto em nível de *design* quanto de implementação, ajudando a localizar *bugs* do sistema, o que ajuda a tornar o código mais robusto.

[5] Quantidade de memória que um programa usa ou referencia quando em execução.

2.1.4.4 *Refatorar ajuda a programar rápido.*

Em um patamar superficial, a refatoração aparenta reduzir a velocidade de desenvolvimento no tempo desprendido em melhorias de *design*, legibilidade do código e redução de erros, todas estas melhorias são de qualidade do *software*. Fowler (2001) explica que pode ocorrer redução na velocidade de criação em um *software* caso possua um péssimo *design* mesmo com o uso de metodologias ágeis, pois se perde muito tempo com:

- A depuração e conserto de *bugs* em vez de adicionar novas funções;
- Demasiada demora no entendimento do sistema e procura de código duplicado;
- Novas funcionalidades demandam mais codificação, traduzidas em aplicação de remendos (*patches*[6]) sobre remendos em cima do código original.

2.1.5 Passos para aplicação de uma refatoração

Para Fowler (2001) o processo de refatoração pode ser realizado de acordo um número de diferentes atividades. Em primeiro, identifica-se em que parte o *software* precisa ser reformulado para definir quais das refatorações deve ser aplicada nestes locais e saber se estas conseguem garantir a preservação do comportamento para o dado problema.

Após, aplica-se a refatoração propriamente dita, para que se possa avaliar o efeito desta em características de qualidade do *software* ou o processo.

Por último, manter a consistência entre o programa reformulado com outros artefatos como os documentos do projeto, especificação de requisitos, testes, dentre outros (MENS; TOURWÉ, 2004).

2.1.6 Catálogo de Refatoração.

O *website refactoring.com*, mantido por Fowler (2001), contém as principais refatorações simples propostas em seu livro e em outras fontes. As que possuem maior relevância no contexto atual estão organizadas, nomeadas e descritas de acordo com o quadro a seguir.

[6] Pequenas atualizações ou correções em um programa de computador.

Nome	Situação	Atividade
Extract Class	Existe uma classe fazendo um trabalho que deve ser feito por duas.	Criar uma nova classe e mover os campos relevantes e métodos da classe antiga para a nova classe.
Move Method	Um método é, ou será utilizado por mais características de outra classe do que a classe na qual ele está definido.	Criar um novo método com um corpo semelhante na classe mais utilizada por ele, transformar o velho método em uma simples delegação, ou removê-lo completamente.
Move Field	Um campo é, ou será utilizado por outra classe mais do que a classe na qual ele está definido.	Criar um novo campo na classe alvo, e mudar todos os seus usuários.
Extract Method	Existe um fragmento de código que pode ser agrupado.	Torne o fragmento em um método, cujo nome explica seu propósito.
Replace Parameter With Method	Um objeto invoca um método, em seguida passa o resultado como parâmetro para outro método, sendo que o receptor também pode invocá-lo.	Remover o parâmetro e deixar que o receptor invoque diretamente o método em questão.
Extract interface	Diversos clientes usam o mesmo subconjunto de *interface* de uma classe. Ou duas classes têm parte de suas *interfaces* em comum.	Extrair o subconjunto em uma *interface*.
Extract Superclass	Existem duas classes com características semelhantes.	Criar uma superclasse e mover as características em comum para essa superclasse.
Substitute Algorithm	Deseja-se substituir um algoritmo por outro mais inteligível.	Substitua o corpo do método pelo novo algoritmo.
Move Class	Uma classe está em um pacote que contém outras classes cujas funções não estão relacionadas com a referida.	Move-se a classe para um pacote mais relevante ou cria-se um novo pacote se necessário, para uso futuro.

Quadro 1: Relação entre contexto e solução de algumas refatorações.

Fonte: Extraído do site: HTTP://*refactoring.com/catalog/* (s.d.).

2.1.7 *Bad Smells*

É um termo informal utilizado por Fowler (2001) para indicar uma estrutura dentro de um código que sugere a possibilidade de se aplicar refatoração com, por exemplo, um método muito longo, código duplicado, classe muito grande, diversos comentários para se explicar um bloco de código, entre outros. Com isso, relacionam-se estes *bad smells* (indicadores) com as refatorações do catálogo proposto por Fowler (MENS; TOURWÉ, 2004).

O quadro 2 resume o contexto desta seção com alguns exemplos:

bad smells	Refatorações
Código duplicado	*Extract Method* *Pull Up Field* *Extract Class* *Substitute Algorithm*
Método muito longo	*Extract Method* *Replace Temp With Query* *Introduce Parameter Object*
Classe muito grande	*Extract Class* *Extract Subclass* *Extract Interface* *Duplicate Observed Data*
Funcionalidade Invejada	*Move Method* *Extract Method*
Comentários	*Extract Method* *Rename Method* *Introduce* *Assertion*
Muitos parâmetros	*Replace Parameter with Method* *Preserve Whole Object* *Introduce Parameter Object.*

Quadro 2: Relação entre alguns *bad smells* e refatorações.

Fonte: Fowler (2001).

2.2 ENGENHARIA DE *SOFTWARE*

Esta seção destaca aspectos e definições importantes sobre engenharia de *software*, identificando seus objetivos e sua aplicabilidade.

2.2.1 Conceitos e definições

A Engenharia de Software caminha em paralelo com sistemas de informações para auxiliar as mesmas a tomarem decisões sob o foco de seu negócio empresarial ou de sua atividade pública. Em conformidade com Rezende (2005) a engenharia de *software* é metodologia de desenvolvimento e manutenção de sistemas modulares, com as seguintes características: processo (roteiro) dinâmico, integrado e inteligente de soluções tecnológicas; adequação aos requisitos funcionais do negócio do cliente e seus respectivos procedimentos

pertinentes; efetivação de padrões de qualidade, produtividade e efetividades em suas atividades e produtos; fundamentação da tecnologia da informação disponível, viável, oportuna e personalizada; planejamento e gestão de atividades, recursos, custos e datas.

A engenharia de *software* é a aplicação sistemática, disciplinada e com abordagem quantitativa para o desenvolvimento, operação e manutenção do *software* (IEEE, 1990, p. 67).

A engenharia de software envolve questões técnicas e não-técnicas, tais como a especificação do conhecimento, técnicas de projeto e implementação, conhecimento dos fatores humanos pelo engenheiro do software e ainda, gestão de projetos (SOMMERVILLE, 1992).

A engenharia de software é conexa, porém distinta, e envolve múltiplas variáveis, tais como arte, atendimento das necessidades humanas, conhecimentos científicos e empíricos, habilidades específicas, recursos naturais, formas adequadas, dispositivos, estruturas e processos. Não deve ser confundida com ciência da computação como um todo, pois ela usa resultados da ciência e fornece problemas para seus estudos (FILHO, 2001).

Schach (2010) define a engenharia de software como uma disciplina de escopo amplo com aspectos que podem ser categorizados na matemática, ciência da computação, economia, administração e psicologia. Tendo como objetivo a produção de um software sem de erros, entregue no tempo e orçamento certo de forma que satisfaça as necessidades do cliente ademais fácil de modificar quando as necessidades do usuário mudar.

Outras definições de Engenharia de software costumam omitir a vertente gerencial. Concentram-se apenas no aspecto tecnológico do problema. Os aspectos gerenciais do desenvolvimento de software devem receber uma atenção cada vez maior nessa disciplina (PRESSMAN, 1995).

Nesse sentido, a engenharia de software pode caracterizar o desenvolvimento de *software* de uma maneira prática, ordenada e medida para produzir sistemas satisfatórios aos usuários e que respeitem prazos e orçamentos (PETERS; PEDRYCZ, 2001).

2.2.2 Elementos fundamentais

Dentro da engenharia de software existem três elementos fundamentais que possibilitam ao gerente o controle do processo de desenvolvimento de software e oferece ao profissional uma base para construção de software de alta qualidade (REZENDE, 2005). Estes elementos serão descritos a seguir.

2.2.2.1 *Métodos de engenharia de software*

Proporcionam os detalhes de construção do software. Envolvem um amplo conjunto de tarefas que incluem planejamento e estimativa de projeto, análise de requisitos de software e de sistemas, projeto de estrutura de dados, arquitetura de programa e algoritmo de processamento, codificação, teste e manutenção. São os roteiros para desenvolvimento de software (REZENDE, 2005).

2.2.2.2 *Ferramentas de engenharia de software*

Proporcionam apoio automatizado ou semi-automatizado aos métodos. Existem diversas técnicas para sustentar os métodos, por exemplo, CASE (*Computer-Aided Software Engineering*), CAD (*computer-aided design*), Análise Estruturada, Orientação a Objetos, e respectivas ferramentas, tais como: de banco de dados, linguagens de programação, etc. São os instrumentos que proporcionam os detalhes de construção do software (REZENDE, 2005).

2.2.2.3 *Procedimentos da engenharia de software*

Possibilitam o desenvolvimento racional e oportuno do software, correlacionando os métodos com as ferramentas. Definem a sequência em que os métodos serão aplicados, os produtos para serem disponibilizados, controles de qualidade e avaliação. Os procedimentos antecedem e sucedem o software (REZENDE, 2005).

2.2.3 Objetivos da Engenharia de *Software*

É o aprimoramento de qualidade dos produtos de software e o aumento da produtividade dos engenheiros de software, além do atendimento aos requisitos de eficácia e eficiência. Ademais, o desenvolvimento de software deve submeter às leis similares as que governam a manufatura de produtos industriais em engenharias tradicionais, pois ambos são metodológicos (MAFFEO, 1992).

A engenharia de software visa sistematizar a produção, a manutenção, a evolução e a recuperação de produtos intensivos de *software*, de modo que ocorra dentro de prazos e custos estimados, com progresso controlado e utilizando princípios, métodos, tecnologia e processos em contínuo aprimoramento. Os produtos desenvolvidos e mantidos, seguindo um processo efetivo e segundo os preceitos da engenharia de software, asseguram, por construção,

qualidade satisfatória, apoiando adequadamente os seus usuários na realização de suas tarefas, operam satisfatória e economicamente em ambientes reais e podem evoluir continuamente, adaptando-se a um mundo em constante evolução (FIORINI et al. 1998).

2.2.4 Considerações finais

A Engenharia de *software* permeia os fundamentos de diversas disciplinas da computação para se estabelecer um olhar prático e perfeito no desenvolvimento de *softwares* economicamente viáveis, de qualidade, que respeitem os prazos e garanta sua posterior evolução. Ela se baseia em processos os quais se subsidiam de métodos, que, auxiliados por ferramentas, se comprometem na qualidade do produto final.

Entende-se processo de desenvolvimento de *software* como sendo um agregado de atividades ou passos para se construir um *software* com qualidade e que satisfaçam os prazos de entrega. Como atividades podem ser citadas a análise econômica, extração de requisitos, especificação, arquitetura de *software*, codificação, testes, documentação, suporte e a manutenção. Dentro deste processo são aplicados uma ou várias metodologias (paradigmas de *softwar*es), a fim de dar suporte à completude e gerencia das referidas atividades, em outras palavras, as metodologias são os tipos de abordagens (modelos) a serem aplicadas no processo de desenvolvimento para se concretizar as atividades propostas na construção do *software*.

Desagregado ao modelo ou paradigma escolhido, existe atividades acolitas como garantia de qualidade de *software*, gerenciamento da configuração do *software* e métricas de *software*.

2.3 MÉTRICAS DE *SOFTWARE*

Medição de *software* é a área da engenharia de *software* responsável por obter um significado quantitativo relacionados a alguns aspectos de um produto ou processo de *software*. Estes aspectos recebem valores e os mesmos são comparados entre si e com todo gerenciamento de qualidade de uma organização. Uma empresa pode comparar por meio de uma medição se é viável ou não a utilização de determinada tecnologia de teste de software por exemplo (SOMMERVILLE, 2004).

Para Schach (2010) sem a utilização da medição torna-se impossível localizar erros no início do processo de *software* antes que seja tarde demais. A medida é um valor medido para dimensionar um padrão.

Para Sommerville (2004) o uso de métricas é incomum em muitas empresas porque os benefícios não estão bem definidos. Isto se deve ao fato da carência da maioria das empresas nas etapas de processo de *software*, pois não há padrões para as métricas nem as empresas possuem todo conhecimento sobre como fazê-la. Sem uma ferramenta bem difundida ou um padrão fica difícil a aceitação das medições visto que retornam um valor aproximado que muitas vezes podem se converter em um valor não significativo ou que não represente de maneira adequada a realidade da etapa medida.

O que se aproveita são os indícios de que uma métrica pode levar a destinar seus esforços para contornar ou continuar com determinada atividade. Estes indicadores podem ser aproveitados durante todo o gerenciamento de qualidade. Uma métrica precisa ser aplicada de maneira empírica e intuitiva de forma a tender para uma determinada posição ao se tomar.

2.4 PADRÕES PARA PROJETO E DESENVOLVIMENTO DE *SOFTWARE*

Esta seção aborda conceitos, história e definições de padrões de projeto bem como suas características estruturais e relação dos principais trabalhos da literatura. De maneira específica tem-se também um apanhado geral acerca dos padrões de criação: *Abstract*, *Factory Method* e *Singleton* e o padrão de integração DAO (*Data Access Object*),

2.4.1 Surgimento e definição

Para o dicionário, o termo padrão é “Objeto que serve de modelo à feitura de outro.” (FERREIRA, 2008).

Schach (2010) destaca Christopher Alexander, arquiteto mundialmente famoso, como um dos indivíduos mais influentes no campo da engenharia de *software* orientado a objetos. Embora Alexander tenha admitido livremente saber pouco ou nada sobre objetos ou engenharia de *software*, a publicação de seu livro em 1977 sobre padrões de linguagens para arquitetura, nos quais descrevia padrões de arquitetura para cidades, construções, salas, jardins entre outros, teve suas idéias adotadas e adaptadas por engenheiros de *software*.

Em especial por um grupo de quatro engenheiros: Erich Gamma, Richard Helm, Ralph Johnson, e John Vlissides. Conhecidos como a gangue dos quatro, cuja publicação de

um livro sobre padrões de projeto resulta na vasta aceitação das idéias de Alexander entre a comunidade de programação orientada a objetos.

A idéia de padrão de acordo com Alexander (1977 apud GAMMA et al., 2000, p. 19) consta da afirmação que cada padrão descreve um problema que ocorrem repetidas vezes em determinados ambiente, e então descreve o núcleo da solução para esse problema, de tal forma que você pode usar esta solução um milhão de vezes, sem nunca fazê-lo da mesma maneira duas vezes.

O termo *Design Pattern* ou também conhecido como Padrões de Projeto é definido por Schach (2010) como uma solução para um problema de projeto em geral na forma de um conjunto de classes interagindo que têm que ser personalizado para criar um projeto específico.

Padrões são maneiras para descrever melhores práticas, bons projetos e capturar experiência de um jeito que seja possível para outros reutilizarem esta experiência (SOMMERVILLE, 2006).

De maneira mais objetiva, Gamma et al (2000) caracteriza padrões de projetos como referências descritivas entre objetos e classes comunicantes (papéis, colaborações e a distribuição de responsabilidade entre classes e instâncias participantes) que precisam ser personalizadas no intuito de resolver um problema de projeto em um contexto específico.

De acordo com Sommerville (2004), padrões de projeto associam-se ao projeto orientado a objetos em um projeto de *software*, pois dependem muitas vezes das características de objetos para garantir a generalidade das soluções. Todavia, essa abordagem deve ser perfeitamente aplicável durante todo o projeto de *software*, desde a criação de modelos de sistemas às etapas de projeto.

Em conformidade com Gamma et al. (2000) os padrões de projeto tornam mais fácil reutilizar projetos e arquiteturas bem-sucedidas, tornando acessíveis técnicas testadas e aprovadas, o que evita rotas de projetos que comprometam o reuso, a documentação e a manutenção de sistemas.

2.4.2 Estrutura dos Padrões de Projeto

De modo geral, um padrão de projeto possui quatro elementos principais: nome do padrão, problema, solução e consequências. Estes elementos essenciais constituem a descrição documentada da solução proposta.

Conforme Gamma et al. (2000) é definido os elementos base:

1. Nome do padrão: descreve em uma ou duas palavras sobre os outros elementos do padrão. Converge em adição do termo ao vocabulário de projeto.
2. Problema: descreve quando aplicar o padrão. Expõe o problema e o contexto. Pode também resultar em listas com as condições necessárias para aplicação do padrão.
3. Descreve de maneira abstrata a solução em si expondo como os relacionamentos, responsabilidades e colaborações entre os elementos que formam o padrão resolvem o problema em questão.
4. Consequências: Balanço positivo e negativo descrevendo o impacto que a aplicação do padrão acarretará com seu uso.

Estes elementos podem ser subdivididos conforme a necessidade de se melhor fazer entender com relação ao contexto e escopo do problema.

2.4.3 Padrões de Projeto

Design patterns, também chamados de padrões de projeto, intitula um dos catálogos mais famosos na área de *software* orientado a objetos. O livro *Design Patterns*: *Elements of Reusable Object-Oriented Software*, de 1995, apresenta um catálogo de vinte e três padrões de projetos divididos em três categorias: Estrutural, Comportamental e de Criação. A figura a seguir ilustra todos os padrões desse catálogo.

		Propósito		
		1. Criação	2. Estrutura	3. Comportamento
Escopo	Classe	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Figura 1 – Padrões do livro *Design* Pattern.
Fonte: Gamma et al. (2000).

Cada categoria trata de um tipo de solução específica, onde:

- Padrões Criacionais: Abstraem o processo de instanciação de objetos. Eles ajudam a tornar um sistema independente de como seus objetos são criados, ofertando flexibilidade para o que é criado, quem cria, como e quando cria;
- Padrões Estruturais: Preocupam-se com a forma como as classes e objetos são compostos para formar estruturas maiores;
- Padrões Comportamentais: Preocupam-se com algoritmos e a atribuição de responsabilidades entre os objetos, dando um grande foco na comunicação entre eles.

Os padrões se organizam pelo seu escopo, que é o contexto ao qual se aplica, e por seu propósito, que define o que faz. Ao contexto, pode-se ter herança em seus relacionamentos caso estejam ligados com classes (estático) e para os objetos pode-se alterar sua instancia em tempo de execução. Ao seu propósito, têm-se a criação de objetos (criacional), composição de classes ou objetos (estrutural) ou definindo a interação e distribuição de responsabilidades (comportamental).

Os padrões importantes para este trabalho permeiam o propósito de criação e podem tanto se relacionar com classes quanto a objetos. A seguir são detalhados os seguintes padrões de projeto: *Factory Method, Abstract Factory* e *Singleton.*

2.4.3.1 *Factory Method*

O *Factory Method* (Fábrica de Métodos), também conhecido como construtor virtual, é um padrão criacional com escopo definido em classes.

É utilizado quando se deseja criar um objeto, mas não se sabe qual classe instanciar até a hora de execução. A aplicação deste padrão define uma interface ou classe abstrata para a criação dos objetos, porém deixa que as suas subclasses decidam quem instanciar adiando a criação do objeto.

De acordo com Gamma et al (2000) pode ser aplicado este padrão quando uma classe não pode antecipar ou conhecer a classe dos objetos que deve criar, quando a decisão de qual objeto criar seja implementado pelas subclasses desta ou para que classes deleguem responsabilidade a alguma das várias subclasses ajudantes, e deseja-se localizar qual é a subclasse ajudante acessada.

A estrutura do padrão *Factory Method* é mostrada na figura a seguir.

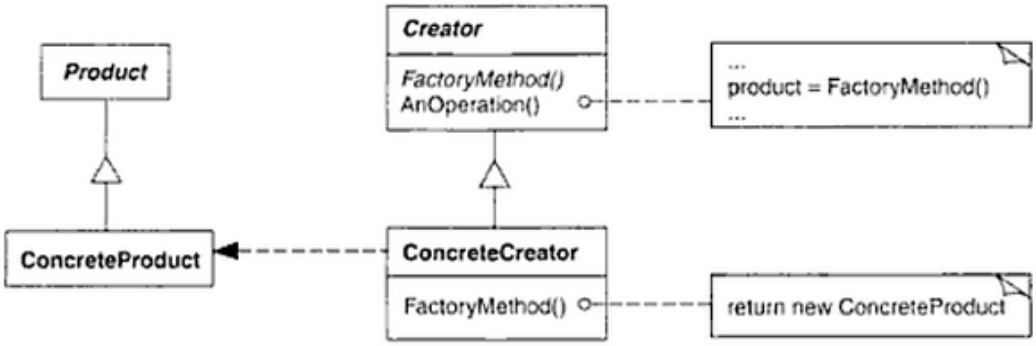


Figura 2 - estrutura do padrão *Factory Method.*
Fonte: Gamma et al. (2000).

A interface de objetos que o método de fabricação cria é definida em *Product* que possui a subclasse *ConcreteProduct*, responsável por implementar a interface da classe pai *Product*. *Creator* é a aplicação do sistema que declara o método fábrica, este retorna um objeto do tipo da interface de objetos, no caso, o tipo *Product*. Dentro da aplicação *Creator* pode existir também uma implementação por omissão do método fábrica, resultando no retorno também por omissão, de um objeto da classe *Concrete Product*. O criador concreto (*ConcreteCriator)* aplica *overriding* no método fábrica para retornar uma instância de um produto concreto.

As consequências do uso deste padrão de acordo com Gamma et al. (2000) reduz o acoplamento entre a classe que precisa de um objeto criado para com a classe que o instancia. Como a classe que deseja utilizar o objeto só enxerga a *interface*, qualquer objeto pode ser instanciado pela fábrica retirando essa relação de anexar a implementação concreta de uma classe para dentro de uma aplicação. Como desvantagem, dentro de um contexto em que somente é necessário criar um tipo apenas de um produto em particular, será necessário ao código fornecer as subclasses do método fábrica, o que não é interessante à utilização de apenas um produto. Ademais, o uso deste padrão fornece variações de um determinando objeto, muito mais flexível que criar um objeto diretamente.

A sua implementação pode ser observada de acordo com a figura exemplo a seguir:

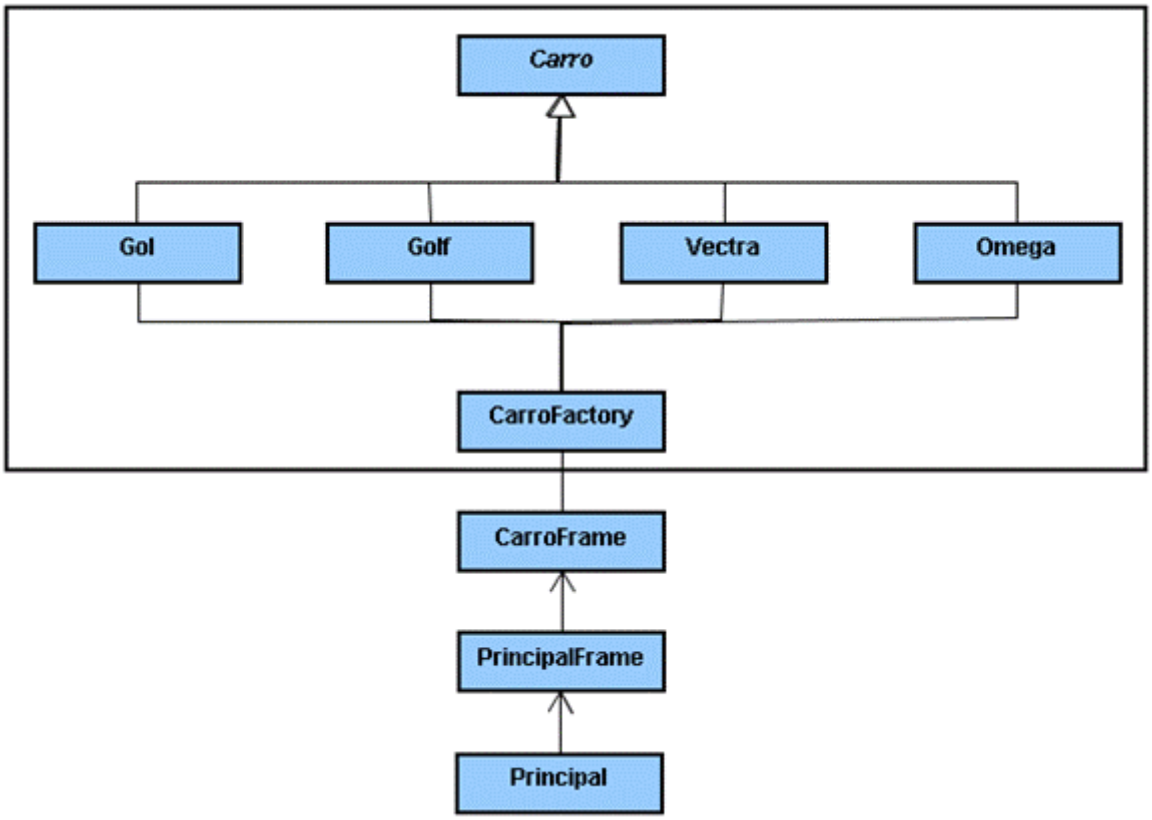


Figura 3 - Exemplo da Aplicação do Padrão Factory Method
Fonte: Laudelino et al.(s.d.).

Observa-se o *main* da aplicação (Principal) contendo uma interface de entrada (PrincipalFrame) e as opções de consulta exibidas em *CarroFrame*, este último invoca, com parâmetro da opção de consulta escolhida, o método fábrica de *CarroFactory*. A passagem do parâmetro é que identifica qual dos objetos de *Carro* deve ser instanciado.

2.4.3.2 *Abstract Factory*

O *Abstract Factory* (Fábrica abstrata), também conhecido como *kit*, é um padrão criacional com escopo definido em objetos. Em consoante à Gamma et al.(2000) o propósito de sua aplicação é a criação de um conjunto de objetos relacionados ou dependentes que sejam especificados nas suas classes concretas. Motiva-se o uso para criar uma espécie de *kit* de ferramentas as quais suportam múltiplas interações, como se existisse, por exemplo, variações de uma aplicação de cadastro de pedidos, contendo os objetos *ClienteA* e *PedidoA* para a aplicação de cadastro normal e objetos *ClienteB* e *PedidoB* para uma aplicação de cadastro especial. Para o tipo certo de cadastro a ser chamado o programa principal necessitaria codificar os dois tipos de cadastro e de maneira fixa. Já com o padrão *Abstract*

Factory, o programa principal não especifica a classe concreta ou implementação de cadastro normal e especial, pois o acesso a criação é via *interface* ou classe abstrata.

Como solução para se aplicar este padrão deve ser definida uma interface ou classe abstrata para as fábricas e uma classe filha (classe concreta) para cada conjunto de objetos relacionados. A figura a seguir ilustra um exemplo.

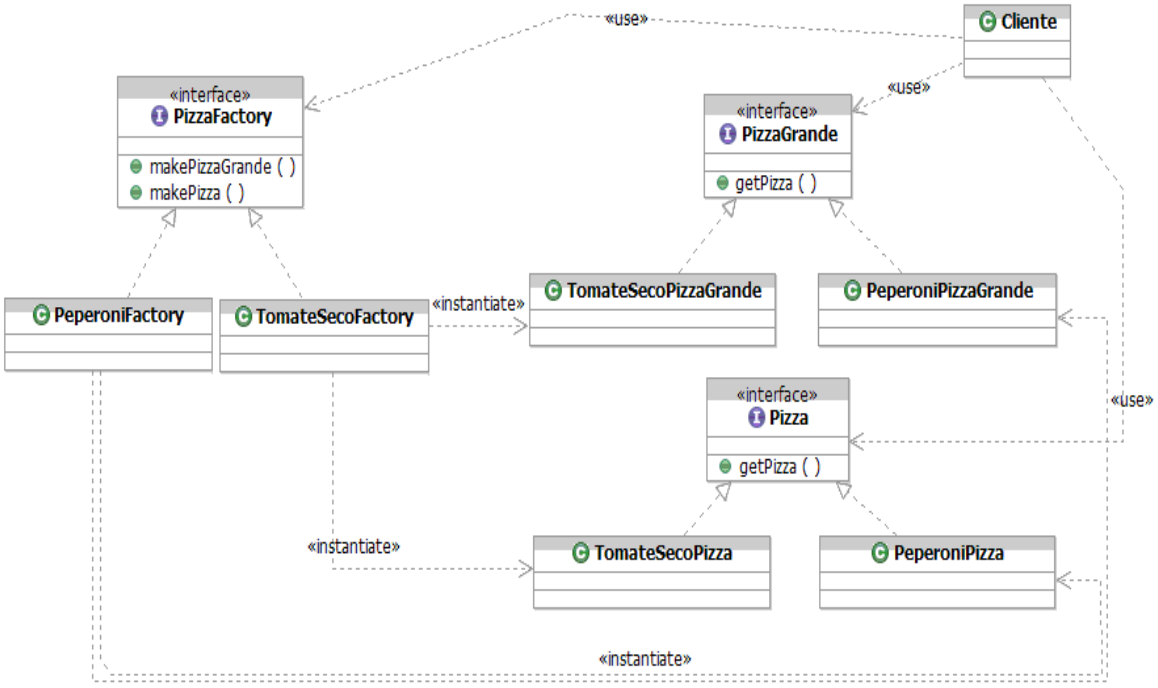


Figura 4 - Exemplo de Fábricas Abstratas.

Fonte: Cursino (2006).

Este exemplo é composto da fábrica abstrata *PizzaFactory* (*Abstract Factory*), superclasse das fábricas de métodos (família de objetos) *PeperoniFactory* e *TomateSecoFactory* (*ConcreteFactory1* e *ConcreteFactory2* respectivamente) que implementam as operações de criação das classes concretas TomateSecoPizzaGrande (*ProductA2*) e *TomateSecoPizza* (*ProductB2*) para o conjunto da família de objetos *TomateSecoFactory* (ConcreteFactory2) e, de forma análoga, *PeperoniPizzaGrande* (*ProductA1*) e *PeperoniPizza* (*ProductB1*) para o conjunto de objetos da família *PeperoniFactory* (ConcreteFactory1).

Para finalizar, a classe *Cliente* (*Client*) acessa somente as *interfaces PizzaGrande* (*AbstractProductA*) e Pizza (*AbstractProductB*) e a Fábrica abstrata *PizzaFactory* decide qual das famílias de objetos criar, *PeperoniFactory* ou *TomateSecoFactory*, ambas fábrica de método.

A decisão de somente fabricar pizza de peperoni ou de tomate seco, independente do tamanho, está implementada de alguma forma em *PizzaFactory* mas pode receber o parâmetro de alguma classe *Cliente*, de um arquivo externo ou até como parâmetro na chamada *main* em linha de comando.

O uso do padrão é viável quando se procura um sistema que seja independente de como seus produtos são criados, compostos ou armazenados. De acordo com Gamma et al. (2000), quando um sistema deve ser configurado como um produto de um conjunto de múltiplos produtos ou quando se precisa garantir a restrição a uma família de objetos-produto desenvolvida para ser usada em conjunto também é uma indicação para se aplicar o padrão. Em última etapa, também se aplica o *Abstract Factory* para esconder as implementações de um conjunto de objetos e revelar somente a assinatura dos métodos através de uma interface.

A aplicação do padrão traz como consequência retorna um maior controle das classes de objetos criadas por uma aplicação isolando as classes concretas. Uma fábrica concreta frequentemente é acompanhada do padrão *Singleton* e só aparece uma vez no sistema, quando é instanciada, e isto provê a flexibilidade de se trabalhar com várias famílias de objetos de uma só vez. O padrão *Abstract Factory* assegura também que classes de objetos projetadas para trabalharem juntas estejam agrupadas. Uma desvantagem deste padrão é a sua descaracterização quando se deseja suportar um novo produto que não faça parte de uma família. Esta modificação para atender o novo produto único, inutiliza a estrutura de famílias e modifica a fábrica abstrata (GAMMA et al., 2000).

2.4.3.3 *Singleton*

Este padrão é um dos mais simples de ser utilizado, fornece segundo seus autores, Gamma et al.(2000) uma forma centralizada do acesso a um objeto e garante que apenas uma única instância de uma classe seja criada. Dentro de alguns sistemas tornam-se importante que certas classes não sejam instanciadas várias vezes, como por exemplo, um arquivo de configuração que precisa manter seus valores padrão, um spooler de impressão, uma conexão fixa a um Banco de Dados, etc.

Sua estrutura e aplicabilidade podem ser analisadas conforme a figura a seguir.

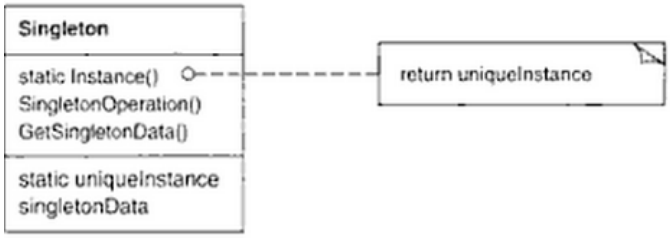


Figura 5 - Estrutura do padrão *Singleton*
Fonte: Gamma et al. (2000)

O padrão define uma operação estática (que dispensa a instanciação da classe a qual pertence) que deixa os clientes acessarem uma única instancia. É esta operação que centraliza o ponto de acesso a instância de uma classe. É aplicada para garantir que clientes usem uma instância estendida sem alterar seu código ou quando se quer dar acesso a uma instância através de um ponto bem conhecido (GAMMA et al., 2000, p. 130).

As vantagens são diversas: O *Singleton* tem controle sobre como e quando clientes acessam a instância, é melhor que variáveis globais, já que as "globais" podem ser encapsuladas na instância única, deixando um único nome externo visível. Várias classes *Singleton* (relacionadas ou não via herança) podem obedecer à mesma interface, permitindo que um *Singleton* particular seja escolhido para trabalhar com uma determinada aplicação em tempo de execução. É possível também modificar seu código para fazer com que o *Singleton* crie um número fixo, ou um número máximo de instâncias em vez de apenas uma única instância bastando apenas que a implementação interna do *Singleton* precise mudar. Sem contar que é mais flexível que métodos estáticos, pois permite o polimorfismo.

2.4.4 Padrões de projeto J2EE

São padrões de projeto baseados em *blueprints* para plataforma J2EE. De acordo com o *website* da empresa Sun Microsystems, Java *Enterprise Edition* (JEE) é uma plataforma para o desenvolvimento de aplicações empresariais distribuídas e multicamada como pode ser observada na figura a seguir:

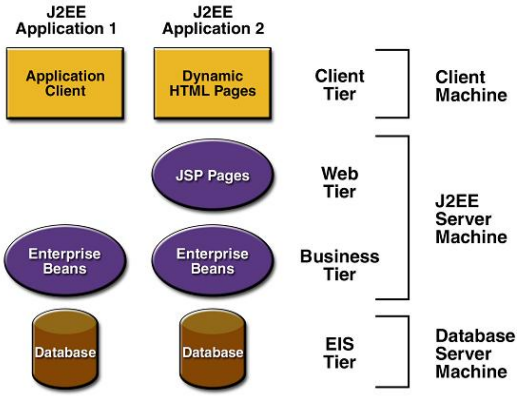


Figura 6 - Estrutura da J2EE

Fonte: Sun Java Center (s.d.).

Blueprints são procedimentos recomendados para desenvolver aplicações J2EE. Divide as aplicações em camadas de acordo com o quadro a seguir:

Camada	Descrição
Cliente	Interface do usuário ou de serviços. Tipicamente representa uma aplicação independente ou browser rodando applets ou páginas HTML.
Web	Consiste de servlets e páginas JSPcom o objetivo de capturar requisições e processar respostas para a camada cliente.
Negócio	Contém toda a lógica da aplicação e representa o modelo de negócio implementado em EJBs[7].
Integração	Contém lógica de acesso à camada de dados.
Camada de Dados	Consiste de sistemas de bancos de dados, transações e outros recursos legados.

Quadro 3 - camadas do Blueprint
Fonte: Adaptado de Alur et al. (2001).

Os padrões JEE apresentados por Alur, Crupi e Malks (2001) representam soluções consideradas melhores práticas para implementar vários componentes essenciais em cada uma das camadas identificadas pelo J2EE *Blueprints*.

O único padrão de projeto J2EE utilizado neste trabalho é o *Data Access Object* (DAO) que se encontra na camada de integração.

[7] Enterprise Java Beans

2.4.4.1 *Data Access Object*

Segundo Alur, Crupi e Malks (2001) o acesso aos dados varia dependendo da fonte dos dados. Um Banco de dados, varia muito, dependendo do tipo de armazenamento (bancos de dados relacionais, bancos de dados orientados a objetos, arquivos simples, e assim por diante) e da implementação do fornecedor. Colocar código de persistência diretamente no código do objeto que o utiliza ou do cliente amarra o código desnecessariamente à forma de implementação ficando difícil, por exemplo, passar a persistir objetos em XML, LDAP, etc. A figura a seguir ilustra o problema.

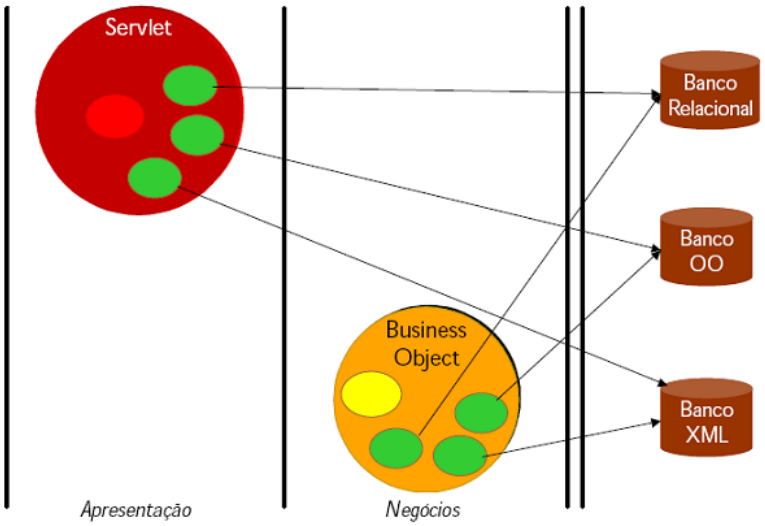


Figura 7 - Código de persistência inserido diretamente nos objetos que o utilizam.
Fonte: Extraído de Alur et al. (2001).

O padrão DAO (*Data Access Object*) é definido por Alur, Crupi e Malks (2001) como sendo responsável por abstrair fontes de dados e oferecer acesso transparente aos dados. É ele que gerencia a conexão com a fonte de dados para resgatar ou inserir dados, conforme a figura a seguir.

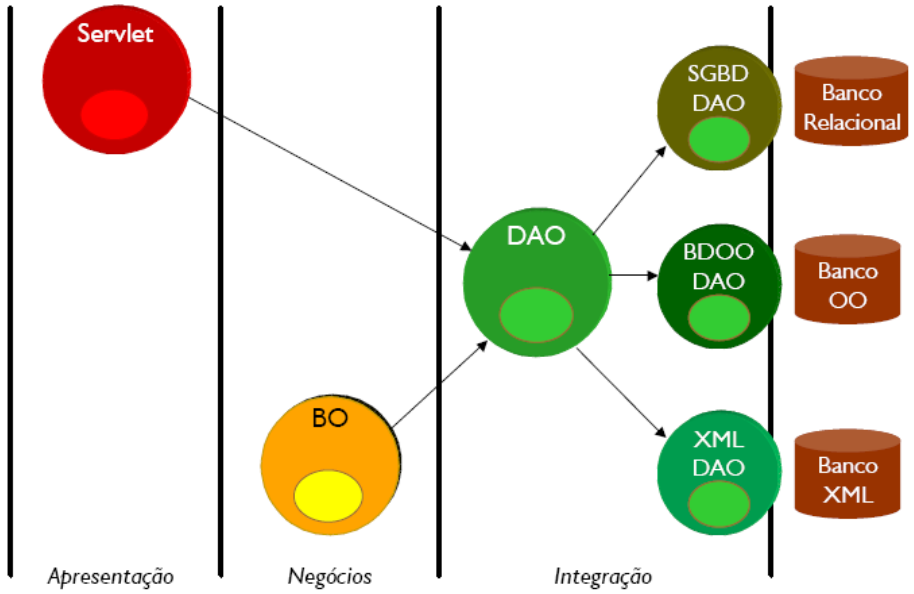


Figura 8 - padrão DAO aplicado.

Fonte: Extraído de Alur et al. (2001).

A camada de integração passa a conter a lógica de acesso para a camada de dados (EIS), que por sua vez, contém sistemas de banco de dados, transações e outros recursos de legado da empresa. A camada de integração está ligada à camada de negócios e é acionada sempre quando ela precisar de dados ou serviços que residem na camada de dados (EIS).

O padrão DAO define uma *interface* que pode ser implementada para cada nova fonte de dados usada, viabilizando a substituição de uma implementação por outra. Mas não mantêm o estado nem *cache* de dados.

A estrutura do padrão é mostrada na figura a seguir:

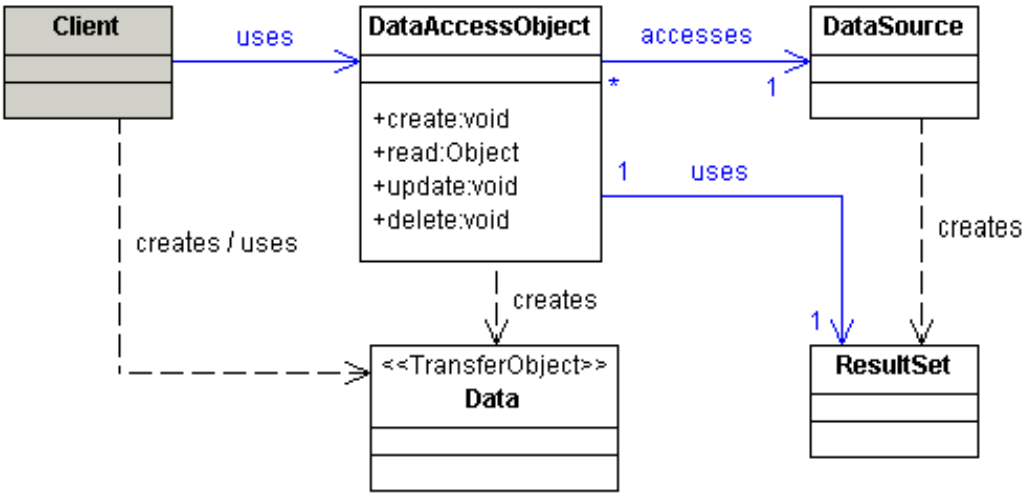


Figura 9 - Estrutura da DAO.

Fonte: Extraído de Alur et al. (2001).

O objeto que requer acesso a dados está representado em *client* (Business *Object*, servlet, etc.). *DataAccessObject* esconde detalhes da fonte de dados e pode conter métodos genéricos para ler e gravar, concentrando operações de banco mais comuns, viabilizando a substituição de uma implementação por outra. *DataSource* é implementação da fonte de dados enquanto *Data* é o objeto de transferência usado para retornar dados ao cliente. O *ResultSet* são os resultados de uma pesquisa feita ao Banco de dados.O diagrama de sequência da figura a seguir exemplifica todo este processo.

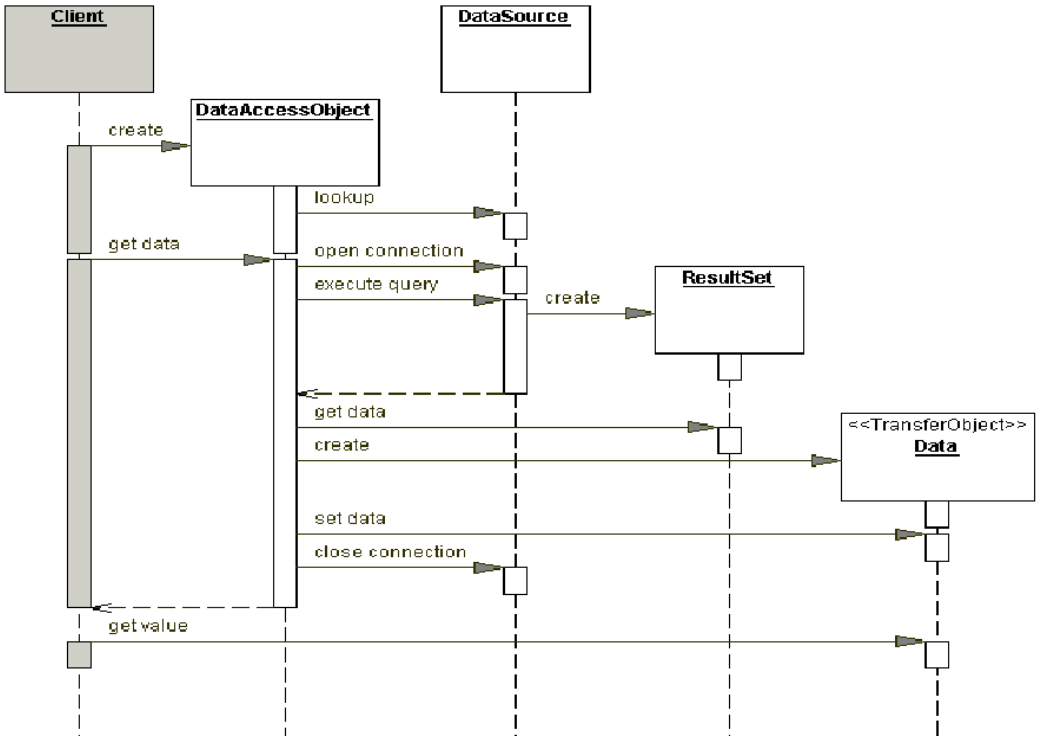


Figura 10 - Diagrama de sequência de uma pesquisa ao banco com o padrão DAO.
Fonte: Extraído de Alur et al. (2001).

Existem diversas estratégias de implementação para este padrão. De acordo com Alur, Crupi e Malks (2001) a estratégia básica é oferecer métodos padrões para criar, apagar, atualizar e pesquisar dados em um banco. Outra estratégia é utilizando os padrões já discutidos neste trabalho: *Factory Method* e *Factory Abstract*, criando uma fábrica para recuperar todos os objetos DAO da aplicação ou criando diversas implementações de fábricas que criam famílias de objetos DAO para diferentes fontes de dados. Esta última estratégia é ilustrada na figura a seguir, mostrando a fábrica abstrata (*DAOFactory*), as diversas fábricas concretas (*RdbFactory*, *XMLFactory* e *OdbFactory*) e os objetos DAO criados.

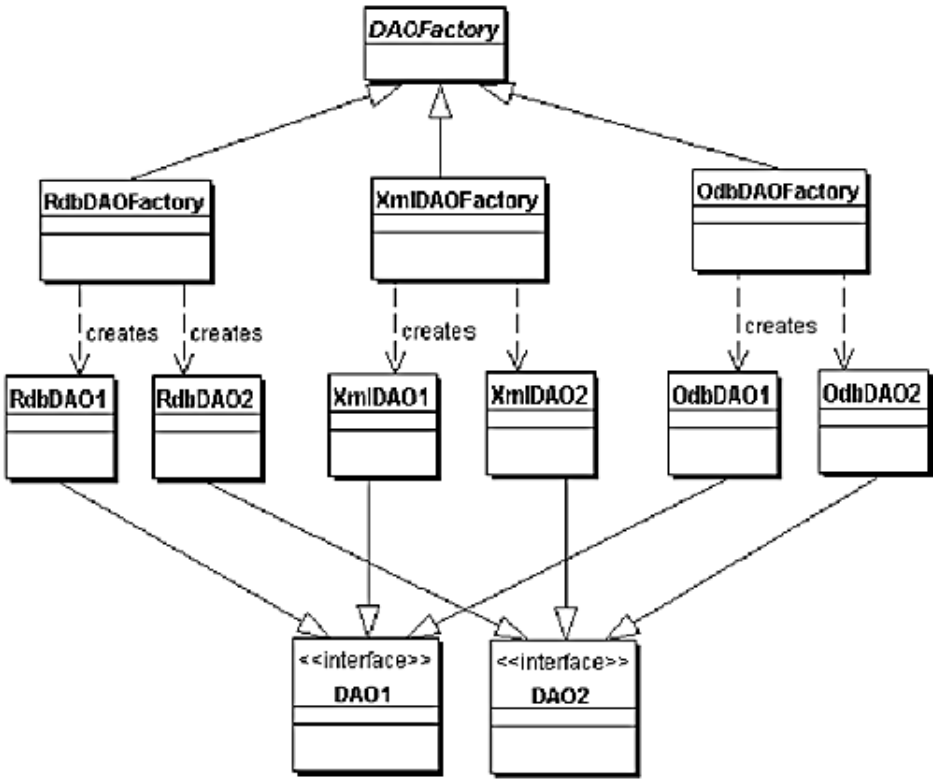


Figura 11 - Aplicação do padrão DAO com Abstract Factory
Fonte: Extraído de Alur et al. (2001).

Como consequencia do uso do padrão se obtém uma maior transparência quanto à fonte de dados, facilidade de migração para outras implementações. Bastando apenas implementar um DAO com mesma interface, há redução da complexidade do código nos objetos da camada de negócio e centralização de todo acesso aos dados de camadas diferentes. Em contrapartida pode ter um pequeno impacto na desempenho do sistema.

3 ESTUDO DE CASO

Este capítulo aborda o referido estudo de caso proposto por este trabalho. A seção 3.1 aborda o ambiente trabalhado. A seção 3.2 orienta-se com uma breve introdução sobre o sistema objeto de estudo e seus conceitos . Em seguida tem-se a seção 3.3 que aborda a visão técnica sobre o objeto de estudo (aplicação em Java). É nela que são observadas as deficiências da aplicação e possíveis melhorias no sistema com as técnicas de refatoração e inserção de padrões de projeto. Nas seções subsequentes intituladas "fase" são descritos as fases necessárias para obtenção do resultado proposto descrito anteriormente neste trabalho.

3.1 AMBIENTE DE TRABALHO

Para o desenvolvimento deste estudo foram usados o IDE *Ecplise* versão *Indigo Service Release* 1 e o Java *Database Connectivity* (JDBC).

Optou-se pelo Eclipse por este ser um ambiente de desenvolvimento para a linguagem Java, visto que é exatamente a mesma linguagem de desenvolvimento do objeto de estudo. Além desta, utiliza-se o IDE por possuir vários recursos de automatização para a aplicação das refatorações propostas pelo presente trabalho.

JDBC é uma API (*Application Programming Interface*) que permite o acesso a diferentes fontes de dados através de programação Java. Ela consiste em um conjunto de classes e interfaces escritas em Java, fornecendo conectividade para uma grande variedade de bancos de dados e arquivos. Um dos motivos que levaram a escolha do JDBC foi sua facilidade para enviar um comando SQL (*Structured Query Language*) para diferentes bancos de dados relacionais.

3.2 OBJETO DE ESTUDO: APLICAÇÃO AGENDA

O sistema Agenda constitui-se em uma aplicação em linguagem Java com acesso a banco de dados *Mysql*. Ela cadastra contatos e categoria destes. O sistema possui duas classes: *Agenda.java* e *Contato.java*, de acordo com a figura a seguir.

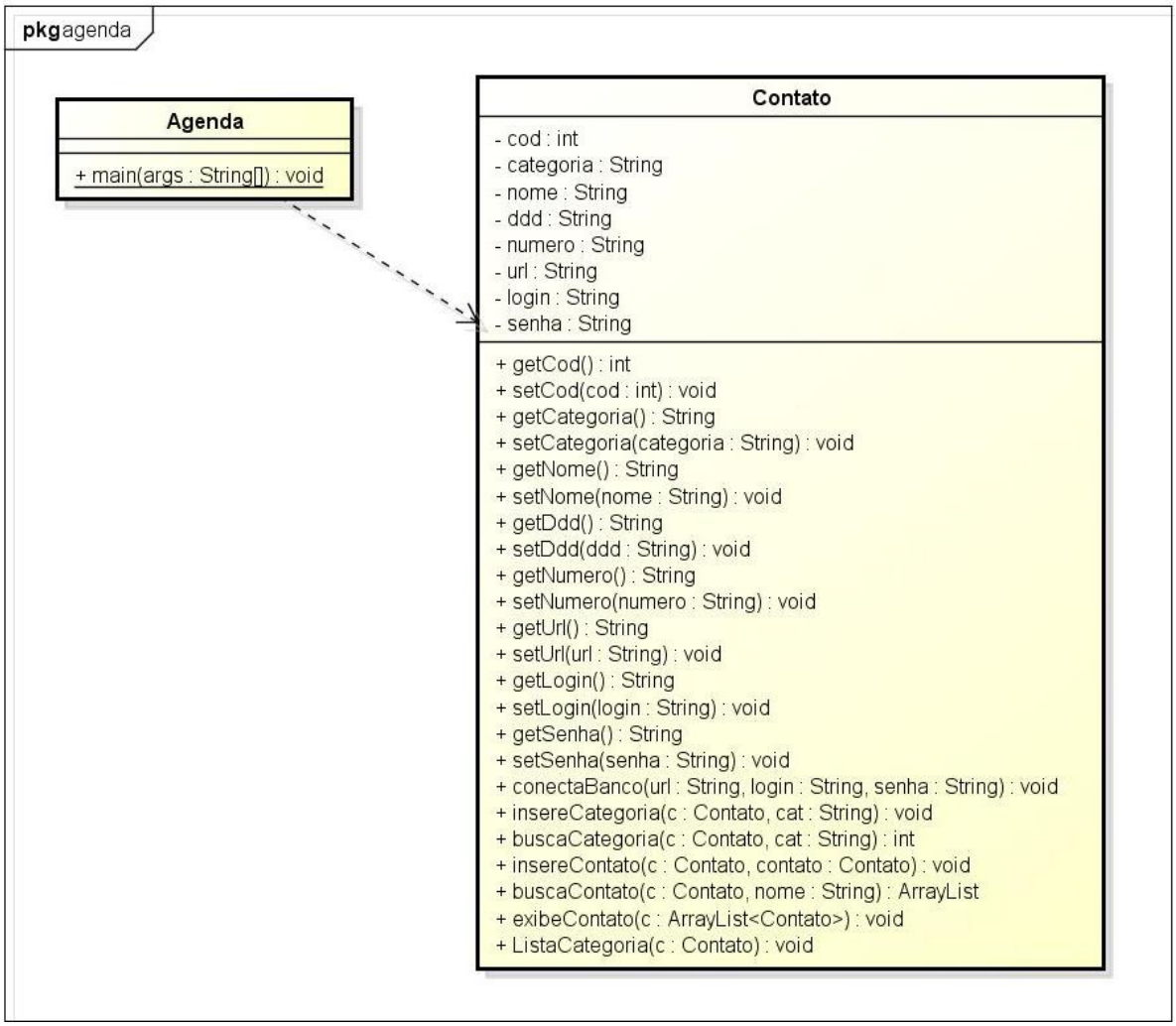


Figura 12 – Diagrama de classes da aplicação objeto de estudo.
Fonte: Autoria própria.

É utilizado um Sistema Gerenciador de Banco de Dados, o *Mysql*, que possui conexão realizada via JBDC utilizando o *driver* do *Mysql* versão 5.0.8.

A classe *Contato* contém os atributos do contato (*cod*, *categoria*, *nome*, *ddd* e *numero*) e dos dados para se estabelecer a conexão (*url*, *login* e *senha*) bem como todo o encapsulamento dos mesmos. Os métodos implementados nesta classe e suas funções são mostrados no quadro a seguir.

Assinatura do método	Funcionalidade
public void conectaBanco(String url, String login, String senha)	Preencher os campos com os dados da conexão.
public void insereCategoria(Contato c, String cat)	Insere uma categoria no Banco de Dados.
public int buscaCategoria(Contato c, String cat)	Busca uma categoria no Banco

	de Dados.
public void insereContato(Contato c, Contato contato)	Insere um contato no Banco de Dados.
public ArrayList buscaContato(Contato c, String nome)	Busca um contato no Banco de Dados.
public void exibeContato(ArrayList <Contato> c)	Exibe uma lista dos contatos encontrados com o nome informado.
public void ListaCategoria(Contato c)	Busca no Banco de Dados e informa a categoria do contato informado.

Quadro 4 – Métodos e funções da aplicação Agenda

Fonte: Internet.

A classe *Agenda* instancia a classe *Contato* para receber os valores dos campos que serão inseridos no Banco de dados e para executar o método que se conecta a ele. A classe *Scanner* (padrão do Java em *import java.util*) é instanciada para ler estes valores. "Ela possui o *Main* da aplicação e contém quatro opções de interação com o usuário: "1 - Cadastrar Contato", " 2 - Buscar Contato", " 3 - Listar Categoria" e " 4 - Sair". Uma nova categoria só se cadastra quando se adiciona um contato. Não existe interface gráfica, apenas comandos enviados pelo teclado.

O Banco de dados deste sistema possui duas tabelas (contato e categoria) cujos campos possuem os mesmos nomes dos atributos das classes para se trabalhar de igual para igual ao que está no Banco de dados.

3.3 ANALISANDO A APLICAÇÃO AGENDA

A classe *Contato* possui métodos de persistência, de lógica de negócio e de conexão com o banco. Existe código duplicado para os métodos de persistência.

A classe *Agenda* armazena valores da conexão, serve de *view* para o usuário e aponta para a classe concreta de contato.

As classes envolvidas na aplicação Agenda possuem muitas responsabilidades. Nenhum padrão de projeto é encontrado no sistema e sua codificação demonstra uma enorme dificuldade de uma possível manutenção ou expansão.

Para contornar estas dificuldades foram introduzidas duas vertentes de refatorações: refatorações normais que melhoram a abstração do projeto e refatorações que culminam em padrões. Os padrões utilizados foram: *Data Access Object*, *Factory Method*, *Abstract Factory* e *Singleton*.

Assim como qualquer refatoração simples é feita passo-a-passo, todo o processo de aplicação de refatorações para se obter o sistema final manutenível e passível de evolução foi realizado em fases e estas avaliadas, compiladas e testadas como forma de garantia de não modificar o resultado final da aplicação Agenda, que é armazenar contatos e categorias.

3.4 O USO DOS PADRÕES

O padrão DAO é aplicado para que a aplicação possa persistir em diversas outras fontes de dados de forma a tornar a etapa de implementação uma tarefa mais fácil de ser executada. Misturar o código de persistência com código de outras classes vincula desnecessariamente os objetos. Visto que a aplicação Agenda trata-se basicamente de cadastros, vê-se claramente uma boa oportunidade de se aplicar este padrão como forma de facilitar a evolução do mesmo.

O uso do *Abstract Factory* e do *Factory Method* neste trabalho torna-se necessário como forma de subsidiar o uso do padrão DAO. Uma vez que a aplicação possa possuir várias "famílias" de fonte de dados diferentes, a sua manutenção é facilitada com a utilização destes padrões. Ademais, os padrões citados provêm desacoplamento entre classes, o que torna o software mais manutenível como pode ser observado na classe principal que, após a aplicação do padrão, passa a depender apenas de interfaces e classes abstratas.

Também se aplica estes conceitos para a adição de novas funções na aplicação Agenda, como adição de diversos tipos de contatos, pois como a classe principal não sabe qual classe será instanciada, podem-se existir vários tipos de contatos, por exemplo.

O padrão *singleton* foi utilizado na medida em que foi detectada a real necessidade para centralizar o acesso de certos objetos no projeto e garantir que apenas uma única instância destes seja criada. Como, por exemplo, verificado no objeto de conexão com o banco de dados e nos objetos que persistem nele, pois apenas uma instância consegue executar todas as funções envolvidas nesta atividade.

As métricas foram utilizadas para avaliar todas as mudanças feitas pelas refatorações e adoção dos padrões de projetos criacionais. Com estas informações foi possível quantificar e comparar a aplicação Agenda da maneira como foi apresentada inicialmente com a versão final, após aplicação das fases deste trabalho. Assim é possível perceber se houve ou não uma melhora na aplicação Agenda. Ademais, todas as métricas utilizadas pertencem ao contexto deste trabalho, aferindo aspectos como acoplamento, coesão, dependência, herança, reusabilidade, manutenabilidade, dentre outros.

Para realizar a tarefa de medição dos aspectos da aplicação, utilizam-se três analisadores desenvolvidos em forma de *plugins* para o IDE *Eclispe*: Jdepend, Metrics e o PMD. Todos estes foram integrados a IDE através de uma interface para adição de *plugins* contida na própria IDE *Eclispe*.

O analisador Jdepend analisa classes *Java* e gera métricas sobre a qualidade do projeto para cada pacote *Java*, em termos de suas extensibilidades, reusabilidade e manutenabilidade. Em geral, avalia aspectos de dependência entre classes.

O analisador Metrics reúne diversas métricas a calcula durante o processo de construção do projeto no *Eclipse*. Possui valores padrões considerados satisfatórios para cada métrica, quando alguma destas ultrapassa os valores considerados normais, o *plugin* informa sobre. Aborda a maioria dos aspectos deste trabalho e através dele foi medido as seguintes métricas: McCabe Cyclomatic Complexity (CC), Weighted methods per Class (WMC), Depth of the inheritance tree (DIT), Number of Children (NOC), Lack of Cohesion of Methods (LCOM), Number of operations overriden by a subclass (NOO) e Specialization Index (SI).

O analisador PDM está relacionado principalmente com a qualidade do código-fonte. Verifica e informa a existência de problemas em potencial como *bugs*, códigos duplicados e expressões complexas.

3.5 FASE I

Problema: Classe C*ontato* possui muitas responsabilidades.

Refatorações: *Extract Class, Rename, Move Method e Move Field.*

A classe *Contato* possui métodos referentes à conexão, como o método *conectaBanco*, e possui certos atributos e métodos referentes a categoria, como *string* categoria e *insereCategoria*, respectivamente. Identificam-se duas classes internas à classe *Contato*: *Categoria* e *Conexao*.

Para evitar que a classe *Contato* realize o trabalho que duas outras deveriam fazer, é aplicado a refatoração *Extract Class*, que objetiva separar um conjunto de responsabilidades de uma classe para outra classe. É criado então uma nova classe (*Conexao.java*) e retira os campos referentes à conexão ao Banco de dados (*url*, *login* e *senha*) da classe *Contato* e adiciona nesta nova criada. Após esta etapa, o método *conectaBanco* da classe *Contato* precisa ser movido (*move method*) e renomeado (*rename*) para se encaixar na classe que agora o refere, a classe *Conexao*.

A segunda classe a ser extraída da classe *Contato* é a *Categoria*, que passa a conter o campo categoria e os métodos: *insereCategoria*, *buscaCategoria* e listaCategoria. De forma análoga são aplicadas as refatorações que funcionaram para a criação da classe *Conexao*.

A compilação do sistema revela erros derivados das refatorações que foram feitas, pois agora existem mais duas classes que não estão mais dentro da classe *Contato*.

São dois tipos de erros: as referências ao estabelecimento da conexão que agora está na classe *Conexao* e aos dois métodos (*insereCategoria* e *buscaCategoria*) que agora fazem parte da classe *Categoria*.

O primeiro tipo de erro aparece nas classes *Contato* e *Categoria*. A resolução dele se deu instanciando a classe *Conexao* nas duas outras classes citadas e inserindo o objeto onde necessário. Como mostrado a seguir:

```
private Conexao conexao = new Conexao();
```

```
…
try {
Connection conn = DriverManager.getConnection(c.getUrl(), c.getLogin(), c.getSenha());
....

Tornou-se:
...
try {
Connection conn = DriverManager.getConnection(conexao.getUrl(), conexao.getLogin(),
conexao.getSenha());
…
```

Para resolver o erro com relação aos métodos retirados da classe *Contato* antiga, aplica-se a mesma regra descrita anteriormente: foi criado um atributo para o objeto *categoria* na classe *Contato* e o mesmo denominado *nomeCategoria* tendo qualquer referencia ao antigo atributo dentro da classe *Contato* substituído por este novo.

Estas refatorações mostram um sistema com quatro classes: *Agenda*, *Contato*, *Categoria* e *Conexao*, cada um com os seus respectivos métodos e atributos. Com estas técnicas é alcançado um sistema com responsabilidades melhor distribuídas entre as classes.

Isto favoreceu a análise do antigo sistema e proporciona melhor abstração do todo, necessário as próximas fases do estudo.

3.6 FASE II

Problema: Trechos de código repetidos pelos métodos de persistência.

Refatorações: *Extract Method, Move Method, Replace Parameter With Method*.

Aplicada a fase anterior, é identificada uma similaridade entre os métodos que estabelecem a persistência no sistema: toda vez que algum deles é chamado, o próprio método é quem está encarregado de registrar o *driver Mysql* e estabelecer a conexão, como no quadro a seguir que mostra o método *insereContato*:

```
        public void insereContato(Contato c, Contato contato) {
            try {
                Class.forName("com.mysql.jdbc.Driver").newInstance();

                try {
                    Connection conn = DriverManager.getConnection(c.getUrl(), c.getLogin(),
c.getSenha());
                    try {
                        String sql = "INSERT INTO agenda.contato "
                                + "VALUES(NULL"
                                + ", " + contato.buscaCategoria(c, contato.getCategoria())
                                + ", '" + contato.getNome() + "'"
                                + ", '" + contato.getDdd() + "'"
                                + ", '" + contato.getNumero() + "');";

                        Statement stm = conn.createStatement();
                        try {
                            stm.executeUpdate(sql);
                            System.out.println("Contato cadastrado com sucesso.");
                        } catch (Exception ex) {
                            System.out.println("\nErro ao cadastrar: " + ex + "\n");
                        }
                    } catch (Exception ex) {
                        System.out.println("\nErro no banco!\n");
                    }
                } catch (Exception ex) {
                    System.out.println("\nErro na conexão!\n");
                }
            } catch (Exception ex) {
                System.out.println("\nDriver não pôde ser carregado!\n");
            }
        }
```

Quadro 5 – método insere contato.

Fonte: Autoria Própria.

Quando um fragmento de código dentro de um método pode ser agrupado em outro método, utiliza-se a refatoração *Extract Method*. Geralmente a maneira mais simples de

observar esta afirmação é quando se depara com códigos longos ou que de alguma maneira é difícil de entender o real propósito dele.

No método de *insereContato* é visto que o seu real propósito é executar o conteúdo da *String* sql, ou seja, o comando *Insert into* seguido dos atributos do objeto contato a cadastrar. Estabelecer a conexão e registrar o *driver* também são execuções necessárias para inserção do objeto, mas não necessariamente precisam estar dentro de cada método que for acessar o Banco de Dados.

Foi aplicado a refatoração *Extract Method* no trecho relativo ao estabelecimento da conexão e registro do *driver Mysql*. Isto deu origem ao método *estabeleceConexao*, responsável por executar as duas tarefas já citadas e retornar a variável *conn* do tipo *Connection* para que o método *insereContato* possa aplicar no seu *Statement* e concluir a operação de inserção.

O novo método pode ser observado no quadro a seguir:

```
        public Connection estabeleceConexao(){
            try {
                Class.forName("com.mysql.jdbc.Driver").newInstance();

                try {
                    Connection conn = DriverManager.getConnection(c.getUrl(), c.getLogin(),
c.getSenha());
        Return conn;

                } catch (Exception ex) {
                    System.out.println("\nErro na conexão!\n");
                }
            } catch (Exception ex) {
                System.out.println("\nDriver não pôde ser carregado!\n");
            }
                Return null;
        }
```

Quadro 6 – método estabeleceConexao()

Fonte: autoria própria.

Para finalizar, basta fazer a chamada do novo método de maneira correta em *insereContato*, no lugar da variável *conn* ficou o método *estabeleceConexao*.

Antes de repetir o processo e retirar todo o código duplicado dos métodos da classe *Contato* e aplicar também, de forma análoga, na classe *Categoria* (que também pode extrair *estabeleceConexao* de seus métodos) é adquirido um novo ponto de vista: Se o fragmento de um método por não condizer exatamente com o propósito dele, pode ser retirado e movido para um novo método que condiz estritamente ao que foi nomeado, em um nível acima, se um

método está em uma classe e dentro da mesma percebe-se que o seu propósito não está exatamente ali, pode-se então movê-lo também entre classes.

De certa forma, o novo método criado (*estabeleceConexao*) não está precisamente vinculado nem na classe *Contato* nem em *Categoria*. O verdadeiro propósito do método é a classe criada na primeira fase deste estudo de caso: *Conexao*. É nela em que deveria estar todos os comportamentos de conexão com o Banco de Dados.

Para isto é aplicado a refatoração *Move Method*, para mover este comportamento da classe *Contato* para a classe *Conexao*, o que é mais indicado.

O que resta a fazer, após esta refatoração, é aplicar a chamada correta para o novo método (*estabeleceConexao*) nas classes que persistirem no Banco de Dados bem como retirar o trecho de código relativo à conexão de seus métodos persistentes. O quadro a seguir exemplifica o método *insereContato* e suas mudanças.

```
        public void insereContato(Contato c) {
        ...
                Statement stm = conexao.estabeleceConexao().createStatement();
                try {
                        stm.executeUpdate(sql);
                        System.out.println("Contato cadastrado com
sucesso.");
                } catch (Exception ex) {
                        System.out.println("\nErro ao cadastrar: " + ex + "\n");
        ...
        }
```

Quadro 7 –método insereContato

Fonte: autoria própria.

Em comparação com o primeiro quadro exibido nesta fase é verificado que a assinatura do método também mudou. É analisado que em alguns métodos do programa antigo utilizavam-se de dois objetos contato de forma separada, onde um era criado apenas para inserir os dados de conexão e o outro para transferir os dados de persistência do objeto em si (acumulo de responsabilidades). A refatoração *Replace parameter with Method* substitui esse parâmetro por uma chamada de método para estabelecer a conexão e culminando assim na retirada de um dos objetos na assinatura dos métodos citados.

Com estas refatorações, não somente o código duplicado é retirado entre métodos de uma classe como também é retirado métodos entre classes do sistema.

Agora toda a etapa de inserção de dados referentes à conexão, carregamento e escolha do *driver* e estabelecimento da conexão está centralizada na classe *Conexao*. Já estão garantido os primeiros passos para um possível reuso desta.

3.7 FASE III

Problema: Métodos de Acesso a dados e a lógica de negócio na mesma classe.

Refatorações: *Extract Class, Move Method*.

Padrão Alvo: *Data Access Object*.

As classes *Categoria* e *Contato* possuem tanto seus métodos de acesso a dados quanto os de lógica de negócio juntos. Isto gera um acoplamento destas partes do sistema, diminui um pouco de sua coesão e reusabilidade. Este problema foi resolvido aplicando-se o padrão *Data Access Object* (DAO). Com isto, para cada classe citada extraem-se uma nova classe separando os métodos de acesso ao banco de dados daqueles da lógica de negócio.

Para a classe *Contato*, extrai-se a classe *ContatoDAOImpl*. Todos os métodos referentes à persistência (*insereContato* e *buscaContato*) que antes estavam na classe *Contato* passaram para a nova classe criada através da refatoração *Move Method.*

De forma análoga a classe *Contato*, aplicação desta fase para a classe *Categoria* originou a criação da classe *CategoriaDAOImpl*.

O uso do termo "DAO" para nomenclatura de classes e interfaces sinaliza o envolvimento destas com o padrão *Data Access Object*. O sufixo "Impl" indica que a classe possui os métodos concretos do objeto.

Ademais é adicionado um comando para instanciar a conexão em cada classe DAO que implementa os métodos de acesso a dados, o <***private*** *Conexao conexao* = ***new*** *Conexao();*>.

O próximo passo realizado é alterar a classe *Agenda* (que contém o *Main* da aplicação) para utilizar os métodos de persistência. Em resulto, conforme descrito a seguir:

```
public class Agenda{
        ...
Contato contato = new ContatoDAOImpl();
… }
```

A classe principal sofre a modificação para usar a DAO, desta maneira e de forma geral, a classe *Agenda* instancia um objeto da classe *Contato* para receber os valores a serem

digitados pelo usuário e acessa o Banco de Dados instanciando um objeto do tipo *ContatoDAOImpl* que executa concretamente os métodos de persistência.

Todos os métodos de acesso a dados que estavam nas classes de lógica de negócio são removidos.

3.8 FASE IV

Problema: Dependência entre o programa principal (Classe *Agenda*) e a implementação concreta da classe que acessa os dados.

Refatorações: *Extract Method, Move Method, Extract interface, Extract Superclass e Rename.*

Padrão Alvo: *Factory Method*.

Quando a aplicação principal (Agenda) instância um objeto de acesso a dados, *CategoriaDAOImpl* por exemplo, através do operador *new* cria-se uma relação de dependência entre estas duas classes.

Esta relação faz com que a classe *Agenda* tenha que saber exatamente qual classe concreta de acesso a dados utilizar, tanto em *CategoriaDAOImpl* quanto em *ContatoDAOImpl*. Esta inflexibilidade ocasiona a perda de dinamismo à aplicação.

Permitir que as classes de acesso ao Banco de Dados, *CategoriaDAOImpl* por exemplo, sejam instanciadas em tempo de execução e de maneira centralizada permite que novas variações (ou subclasses) destas possam ser criadas sem ter que modificar as classes que as instanciam e realizam a chamada de seus métodos. O método aplicado nesta fase para resolver este problema é o *Factory Method*.

A classe *Agenda*, após a análise, mostra-se a única classe que instancia os métodos de acesso ao Banco de Dados. As classes instanciadas são: *CategoriaDAOImpl* e *ContatoDAOImpl*. Para elucidar os processos desta fase, é descrito apenas uma delas, a classe *ContatoDAOImpl*.

Parte-se para a criação da *interface* que contém as assinaturas de métodos de acesso ao Banco de Dados, através da refatoração *Extract Interface* aplicada à classe concreta *ContatoDAOImpl*. A *interface* é mostrada a seguir:

```
public interface ContatoDAO {
   public void insereContato(Contato c);
   public java.util.ArrayList<Contato> buscaContato(String nome);
}
```

Através desta *interface*, novos métodos podem ser sobrescritos em suas subclasses.

Cria-se a classe *DAOFactoryImpl*, que receberá o método construtor de *ContatoDAOImpl*. A refatoração *Extract method* aplicada na linha em que a classe *Agenda* está instanciando este objeto produz um método público de criação que possui como retorno uma instância de *ContatoDAO*.

Através de outra refatoração, a *Move Method*, o método construtor criado com a refatoração *Extract Method* é movido para a classe *DAOFactoryImpl*. Após esta etapa, a refatoração *Rename* é aplicada a este construtor para dar uma referencia mais significativa ao nome do método, resultando na assinatura "public ContatoDAO createContatoDAO()".

Este processo é repetido de forma análoga com outra classe de acesso ao Banco de Dados, o *CategoriaDAOImpl*. Onde surge a *interface CategoriaDAO* e o método construtor de nome "createCategoriaDAO()", este último movido para a classe *DAOFactoryImpl*.

Existem variações do padrão F*actory Method* que permite a classe *Agenda* requisitar uma instância de um objeto diretamente à classe *DAOFactoryImpl*. Mas, este tipo não é levado em consideração neste contexto e, portanto, é aplicado a refatoração *Extract Superclass* em *DAOFactoryImpl* para se criar uma classe abstrata contendo os métodos abstratos de criação da mesma.

As mudanças feitas na classe principal (Agenda) da aplicação são mostradas no quadro a seguir:

```
public class Agenda {
…
        DAOFactory daoFactory = DAOFactoryImpl();
        ContatoDAO contatoDAO = daoFactory.createContatoDAO();
        CategoriaDAO categoriaDAO = daoFactory.createCategoriaDAO();

...
```

Quadro 8 – invocando métodos sem o operador *new*

Fonte: autoria própria.

Com estas mudanças a classe *Agenda* possui uma abstração da *Factory Method* e os objetos de persistência, porém, só saberá qual instancia irá receber em tempo de execução.

Tornou-se necessário, a partir desta fase, separar todas as classes em pacotes permitindo que se agrupem por funcionalidade e propósito garantindo também a melhor compreensão da aplicação como um todo. Foi aplicado a refatoração *Move Class* para esta tarefa e o resultado é exibido através da estrutura retratada na figura a seguir.

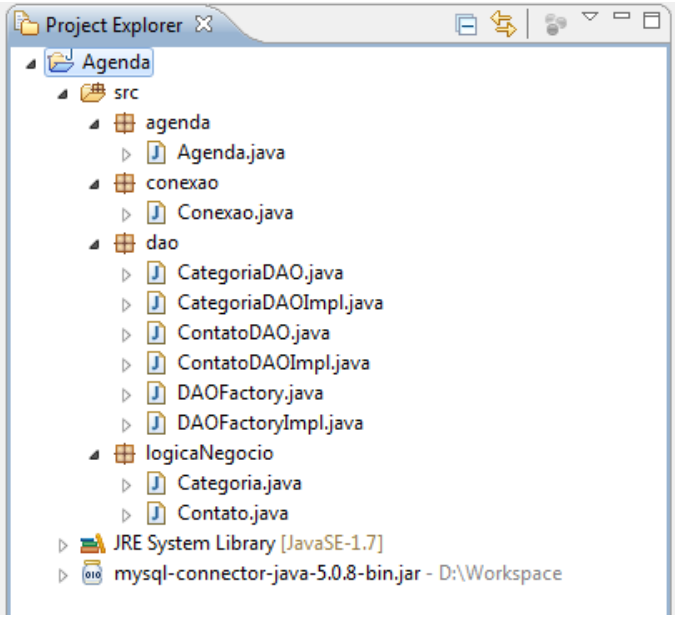


Figura 13 – operação *MoveClass*

Fonte: IDE *Eclipse* (2011).

Esta divisão separa de maneira lógica todas as partes da aplicação Agenda não estando ligada a um padrão de nomenclatura. O seu uso favorece o entendimento do estudo proposto.

3.9 FASE V

Problema: Algumas classes não precisam ter várias instâncias ao mesmo tempo.

Refatorações: *Rename*, *Substitute Algorithm.*

Padrão Alvo: *Singleton*.

Ao se instanciar um objeto, um pedaço de memória física está sendo ocupado para armazenar seus estados (atributos) e comportamentos. Chamar o construtor de uma classe deliberadamente pode comprometer o desempenho do sistema.

Quando uma classe não possui estados ou estes são compartilháveis, existe uma boa probabilidade de que não se precise de mais de uma instância daquele objeto sendo executada pelo sistema.

Na aplicação Agenda são observados estes casos nas classes: as implementações concretas de todos os *Data Access Object*, a fábrica que os cria e também a classe que conecta no Banco de Dados.

A classe *ContatoDAOImpl*, por exemplo, pode ser instanciada para persistir todos os objetos *Contato* que o sistema tenha instanciado em sua execução. Isso se deve a identificação

da relação em que apenas os objetos da classe *Contato* é que precisam, se necessário, ter múltiplas instâncias (contato João, contato Pedro, etc.) na medida em que só se precisa de uma instância de *ContatoDAOImpl* para efetuar a persistências destas pessoas.

Mesmo que, em fases anteriores, existia a variável de conexão com o banco nas classes *ContatoDAOImpl* e *CategoriaDAOImpl* não havia necessidade de se instanciar mais de uma de cada, pois o estado conexão era uma variável fixa (continha os dados da conexão ao Banco *Mysql*). O que culmina na unicidade dos métodos de criação das mesmas.

A maneira encontrada para contornar este problema foi a utilização do padrão *singleton*.

Aplicando o *code snippet*[8] conforme explicado no Anexo A deste trabalho consegue-se fazer com que todas estas classes aderem ao padrão *Singleton* e de uma maneira bastante prática. Bastou-se introduzir a palavra "aplicasingleton" em qualquer parte do código e logo em seguida o atalho *Ctrl+Space* para escolher o *snippet*, conforme mostra o resultado o quadro a seguir.

```
public class ContatoDAOImpl implements ContatoDAO {
…
/****************** Padrão SINGLETON ******************/
        private static ContatoDAOImpl instance;

        private ContatoDAOImpl() {
        }

        public static ContatoDAOImpl getInstance() {
                if (null == instance) {
                        instance = new ContatoDAOImpl();
                }
                return instance;
        }

        /****************** Padrão SINGLETON ******************/
…
```

Quadro 9 - padrão *Singleton* aplicado com um *snippet*.

Fonte: Adaptado do site http://snippets.dzone.com/tag/singleton.

Este quadro mostra como ficou o padrão *Singleton* na classe *ContatoDAOImpl*. Isto garante o controle de como e quando os clientes acessam o objeto de persistência para a classe *Contato* e possibilita, caso haja necessidade no futuro, o polimorfismo. Quando se utiliza campos estáticos não é possível esta característica de Orientação a Objetos.

[8] Fragmentos de código pré-fabricados e parametrizáveis que podem ser inseridos em uma aplicação.

Identificado na classe *Conexao* a existência de um método construtor não vazio, pois o mesmo continha a chamada de métodos "*setDados*" o que, ao aplicar o *snippet*, reporta uma falha. Bastou-se então substituir o algoritmo do método construtor privado criado pelo *snippet* com o conteúdo do método construtor não vazio e já existente em *Conexao* antes da aplicação do padrão. Para esta tarefa aplicou-se a refatoração *Substitute Algorithm*, pois é a utilizada quando se quer substituir o corpo de um método por um novo algoritmo, mais claro que o anterior. Em consequência, elimina-se o antigo método construtor da classe, restando apenas o método privado criado com o *snippet*.

A partir deste momento, qualquer chamada feita a um objeto *singleton* passou a referenciar-se ao seu método público "*getInstance()*". Este método tornou-se responsável por garantir a única existência de uma instância de uma classe dentro da aplicação Agenda.

3.10 FASE VI

Problema: Aplicação não suporta vários tipos de persistência.
Refatorações: *Rename,Move Class, Substitute Algorithm.*
Padrão Alvo: *Abstract Factory*.

Mesmo com as fases anteriores, a aplicação Agenda ainda se mostra como uma aplicação que não suporta diversos tipos de persistência.

O sistema atual utiliza o JDBC e o Banco de Dados *Mysql*. A utilização de outra forma de persistência pode atualmente ser feita de dois modos: reescrevendo as classes concretas que implementam os métodos de acesso a dados (*CategoriaDAOImpl* e *ContatoDAOImpl*) ou criando uma fábrica dessa nova persistência em conjunto com uma implementação concreta de cada entidade do sistema.

Ainda assim o sistema estaria acoplado a um único tipo de persistência. Um meio de criar conjuntos de objetos relacionados ou dependentes de forma a suportar qualquer fonte de dados pode ser obtido através do padrão *Abstract Factory*.

Para esta fase é definido o termo "família" como referência a todas as classes responsáveis por um determinado meio de persistência. Como por exemplo, a "família" *Mysql* está se referindo a fábrica concreta *DAOFactoryImpl* e às classes concretas *CategoriaDAOImpl* e *ContatoDAOImpl*, pois estas todas são responsáveis por persistirem através da *API* JDBC com o Banco de dados *Mysql*.

A idéia almejada por esta fase é fornecer uma família base para que a inserção de futuras novas "famílias" torne o sistema flexível.

Devido à natureza abstrata deste padrão faz-se necessária a inserção de uma "família" de exemplo no sistema, neste caso, adotado a "família" *XML*.

A forma mais rápida e prática deste feito é a utilização da *IDE* proposta para o trabalho e seu sistema automático de refatoração.

Cria-se um pacote no projeto *Agenda* com o nome de *temp*, identifica-se uma família existente no pacote *DAO* como, por exemplo, a família *Mysql* e após selecioná-las pelo eclipse executa-se a ação de copiar as mesmas para o pacote *temp* criado.

Confirmado os passos descritos no parágrafo acima é aplicado então para cada classe dentro do pacote *temp* a refatoração *Rename* para mudar o nome das classes e adequá-las a nova família desejada, a "família" *XML*. A *IDE* automaticamente verificam e substituem o nome da classe bem como as referências feitas a ela internamente, conforme a figura a seguir.

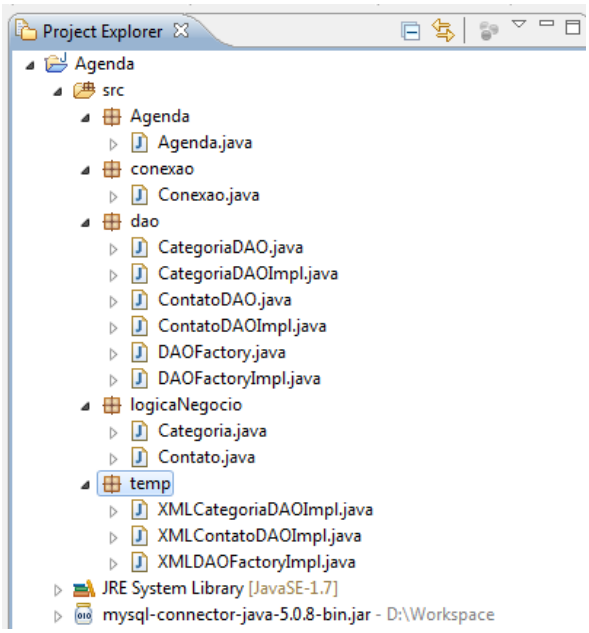


Figura 14 – nova organização do objeto de estudo

Fonte: IDE *Eclipse* (2011).

Em seguida a refatoração *Move Method* serve para deslocar a "família" *XML* para o pacote *dao*, que agora está mais relacionado com as suas funcionalidades, e apagar o pacote *temp* criado.

Após estes processos é observado que a fábrica concreta da nova família não precisa ser mais refatorada (a classe *XMLDAOFactoryImpl*). Isto é identificado pela existência das

interfaces que implementam o mesmo métodos para qualquer persistência entre as entidades e pela aplicação do padrão F*actory Method* nas fases anteriores.

Já para as implementações concretas (*XMLCategoriaDAOFactory* e *XMLContatoDAOImpl*) é aplicado a refatoração *Substitute Algorithm* para substituir o corpo dos métodos copiados da "família" *Mysql* pelo algoritmo que melhor satisfaça esta nova coleção de objetos.

Em seguida pode-se identificar a nítida necessidade da aplicação do padrão *Abstract Factory* na aplicação Agenda. Devido a fato de este estar em um nível de abstração maior que o *Factory Method*, aproveita-se da refatoração *Extract Superclass*, utilizadas em fases anteriores, para aplicar o padrão proposto na classe *DAOFactory* existente, conforme mostra o quadro a seguir.

```
public abstract class DAOFactory {
        private static final int PADRAO = 1;
        private static final int XML = 2;

        public static DAOFactory getDAOFactory(int qualFabrica) {
                switch (qualFabrica) {
                case PADRAO:
                        return DAOFactoryImpl.getInstance();
                case XML:
                        return XMLDAOFactoryImpl.getInstance();
                default:
                        throw new IllegalArgumentException("Fabrica não
existe!");
                }
        }

        public abstract ContatoDAO createContatoDAO();
        public abstract CategoriaDAO createCategoriaDAO();
}
```

Quadro 10 – Fábrica de DAOs.

Fonte: Autoria Própria

É definido uma constante privada estática do tipo inteiro para cada "família" implementada no sistema, em seguida é criado um método (getDAOFactory (int qualFabrica)) responsável por estabelecer qual das "famílias de métodos" executar. Todo o resto da classe *DAOFactory* continua inalterado.

Sempre que uma classe necessita instanciar *DAOFactory*, utiliza-se a linha de código "DAOFactory daoFactory = DAOFactory.*getDAOFactory*(1);".

O valor passado por parâmetro indica qual fábrica executar. Diversas outras abordagens foram testadas, como a passagem de argumentos da classe principal no console e

o uso de um arquivo externo para indicar qual das “famílias” a executar, entretanto esta se mostrou mais acessível para o entendimento e aplicação desta fase.

De fato, a inserção do código na classe abstrata *DAOFactory* se dá por um número simples de refatoração, todavia as refatorações anteriores se mostraram necessárias para a aplicação deste padrão ou no mínimo para a redução de sua complexidade quanto ao programa Agenda.

Cabe ressalvar que não se implementou verdadeiramente nenhuma das classes da “família” XML nem se pode ponderar sobre questões de conexão entre as “famílias” pois não são necessárias a formatação desta fase.

4 RESULTADOS OBTIDOS

A versão atual da aplicação está estruturada internamente de maneira totalmente diferente da versão antiga. A figura a seguir ilustra os pacotes desta nova versão e a sua relação.

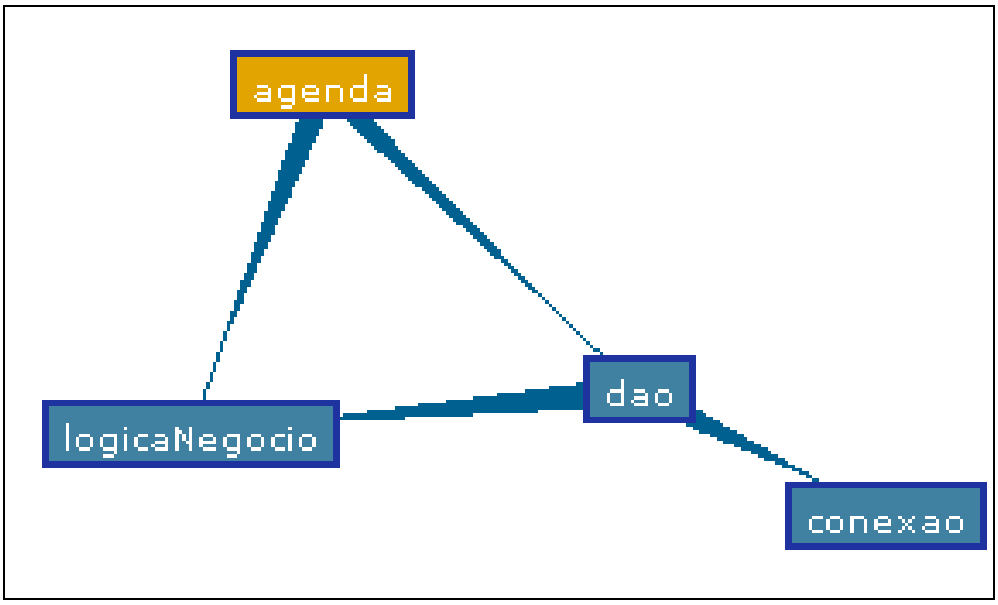


Figura 15 – Gráfico de dependência entre pacotes da versão atual.
Fonte: IDE *Eclipse* (2011).

Cada pacote ficou responsável por desempenhar uma determinada tarefa. O pacote *agenda* continuou sendo o *main* da aplicação, responsável por iniciar a agenda.

O pacote *logicaNegocio* atua como a camada de negócios, deveras não existirem ainda métodos e validações. É neste pacote que novas entidades devem ser abstraídas. O diagrama de classe deste pacote pode ser visualizado na figura a seguir:

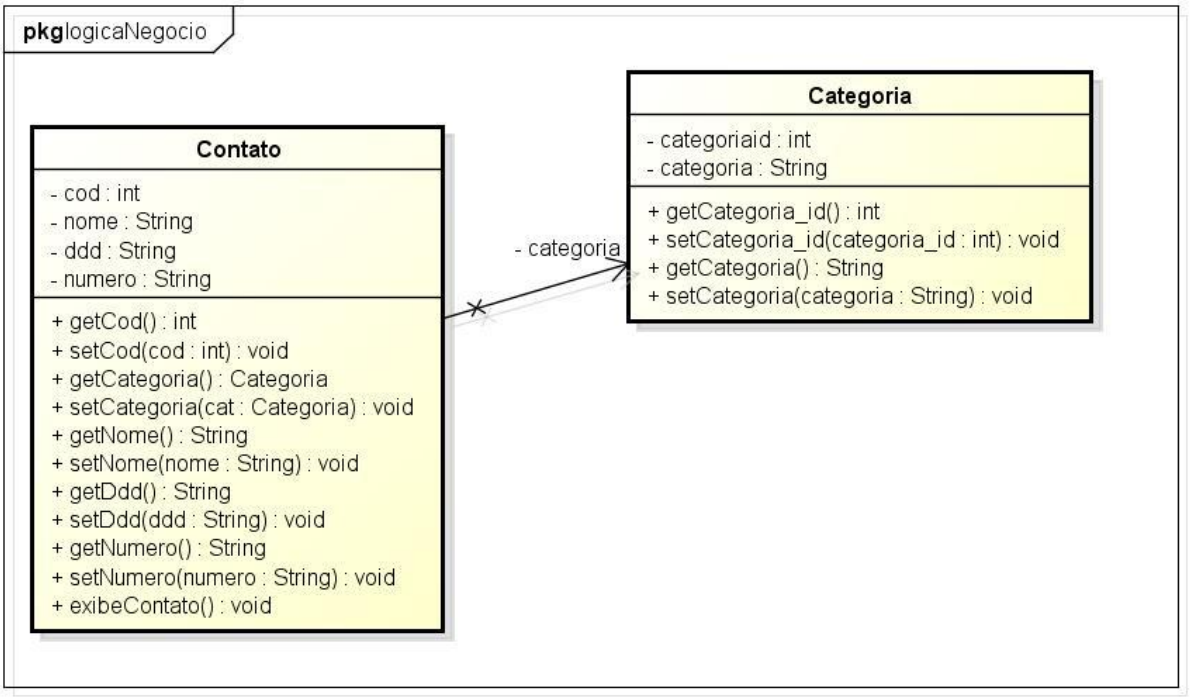


Figura 58 - diagrama de classes do pacote *logicaNegocio*.

Fonte: autoria própria.

O pacote *conexão* foi extraído do pacote *DAO* em etapas anteriores para dar maior mobilidade, ter uma visão clara de seu papel e poder ser aplicado outros padrões. O seu diagrama de classes é mostrado na figura a seguir.

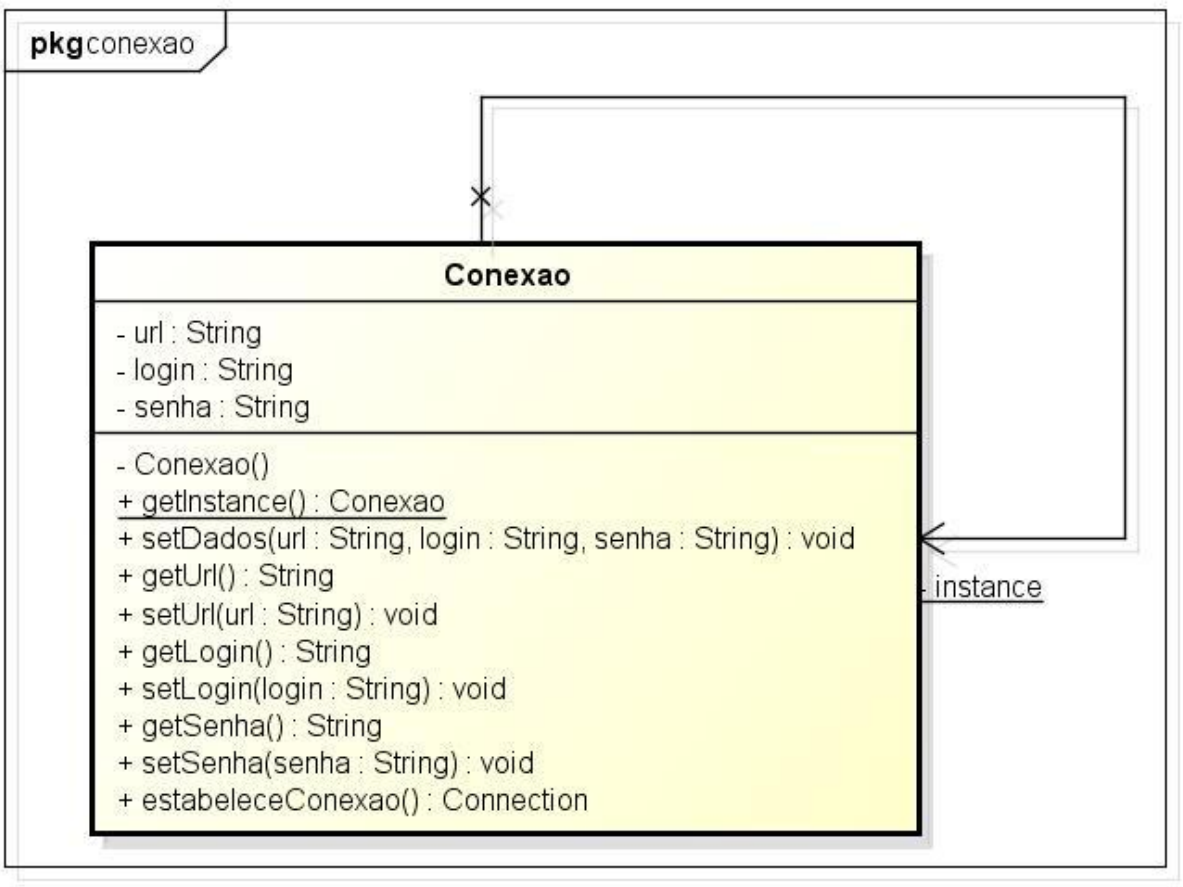


Figura 17 - diagrama de classes do pacote *conexão*.

Fonte: autoria própria.

O pacote *DAO* contém toda a lógica de persistência ao banco de dados. Ele abstrai e encapsula todo o acesso a uma fonte de dados para obter e persistir os objetos encontrados no sistema. Neste pacote aparecem os padrões *Abstract Factory, Singleton e Factory Method*. O uso destes padrões culmina em uma estrutura capaz de aumentar a coesão, reduzir o acoplamento, melhorar a reusabilidade.

O *Factory Method* eleva em um nível de abstração para criar uma *interface* de um objeto, desacoplando a ligação entre a classe que realiza a chamada da classe que se deseja instanciar, fazendo com que a escolha de qual classe instanciar seja das subclasses e não da classe que realiza a chamada. Isto pode ocorrer em tempo de execução.

Podem ser desenvolvidas novas entidades para esta aplicação de maneira mais simples e organizada, facilitando o reuso de componentes e de manutenção. O *Abstract Method* irá permitir persistir objetos em diversas fontes de dados, criando famílias de objetos persistentes para cada mecanismo de persistência.

O pacote *DAO* fornece os meios e formas de se expandir a aplicação Agenda. Conforme nascerem novos requisitos, novos objetos para persistir e novas famílias, toda estrutura já estará montada, não precisando ter dificuldades em implementá-las. A única ressalva é para os que não compreendem os padrões de projetos criacionais, item que pode causar um pequeno grau de complexidade, sem contar com a escrita de uma nova implementação de fábrica, objetos e afins. Mas o pacote *DAO* está muito genérico, podendo refletir vários domínios, o que ajuda no reuso e manutenção.

4.1 ANALISANDO COM JDEPEND

A aplicação do analisador de pacotes *Jdepend* como forma de métrica de qualidade do *design* em termos como extensibilidade, reusabilidade e manutenabilidade obteve-se os resultados de acordo com a tabela:

Tabela 1 - resultado do Jdepend.

Pacotes	Total Classes	CC	AB	Ca	Ce	A	I	D
Versão Antiga								
agenda	2	2	0	0	0	0.0	0	1.0
Versão Atual								
agenda	1	1	0	0	2	0	1	0
conexao	1	1	0	1	0	0	0	1
dao	9	6	3	1	2	0.33	0.66	0.00
logicaNegocio	2	2	0	2	0	0	0	1

Fonte: Analisador Jdepend para IDE *Eclipse* (2011).

Onde:

- CC - Contador de Classes Concretas
- AB - Contador de Classes Abstratas e Interfaces
- Ca - Acoplamento Aferente
- Ce - Acoplamento Eferente
- A - Abstração (0-1)
- I - Instabilidade (0-1)
- D - Distância da Sequência Principal (0-1)

A avaliação dos dados da tabela permite afirmar que a versão antiga do objeto de estudo não possui qualquer outra dependência entre pacotes (só existe o pacote *agenda*), porém é completamente concreta e estável, características que impedem que o pacote seja passível de extensão. Isto se comprova ainda mais com valor do indicador de Robert Martin (coluna D) que é igual a um, diagnosticando um péssimo *design* para o pacote.

A nova versão possui nível de abstração trinta e três por cento a mais que na versão antiga. O pacote que provê maior abstração é o mais utilizado neste trabalho para inserção de padrões de projetos criacionais e refatorações: o pacote *DAO*. Além disso, possui um índice de instabilidade bom aliado a uma distância da sequência principal baixo, fazendo-o possuir poucas dependências e ser usado por outros pacotes, tornando possível o uso de muita abstração no pacote. O pacote *agenda* também é considerado ideal (concreto e instável) nesta versão.

4.2 ANALISANDO COM METRICS

O analisador *metrics* obteve um conjunto de valores capazes de quantificar o estudo de caso de acordo com a seguinte tabela:

Tabela 2 – métricas de qualidade

		Versão Antiga				X	Versão atual			
	METRICS	Total	média	desvio padrão	máximo		Total	média	desvio padrão	máximo
	McCabe Cyclomatic Complexity (CC)		**2,292**	2,169	7			**1,582**	1,246	8
Chidamber & Kemerer	Weighted methods per Class (WMC)	55	**27,5**	20,5	48		87	**7,909**	3,604	15
	Depth of the inheritance tree (DIT)		**1**	0	1			**1,182**	0,386	2
	Number of children (NOC)	0	**0**	0	0		2	**0,182**	0,575	2
	Lack of Cohesion of Methods (LCOM)		**0,456**	0,456	0,912			**0,202**	0,334	0,889
Lorenz & Kidd	Number of operations overriden by a subclass (NOO)	0	0	0	0		0	0	0	0
	Specialization Index (SI)		0	0	0			0	0	0

Fonte: Analisador Metrics para IDE *Eclipse* (2011).

O termo versão antiga está relacionado com a aplicação que se objetivou realizar todas as mudanças enquanto versão atual corresponde aos campos em que as métricas foram colidas no projeto remanescente da última fase aplicada.

Com relação à métrica de complexidade ciclomática (CC) o número máximo alcançado na versão final do programa acresceu um ponto. Este campo na tabela apresenta a maior pontuação que um método dentro do referido projeto recebeu. Porém, a média entre todas as medidas e o desvio padrão exibidos na tabela mostram que a versão atual, de forma geral, possui menores índices de CC.

Para se estabelecer uma melhor visualização dos valores desta métrica torna-se necessário expandir a tabela. A versão antiga possuía somente duas classes onde a maior pontuação é sete para classe *Agenda*. A métrica de complexidade entre todos os pacotes da última versão pode ser avaliada a seguir.

Tabela 3: valores máximos da métrica CC por pacote (última versão).

Pacotes	Máximo
agenda	8
dao	4
conexao	3
logicaNegocio	1

Fonte: Analisador Metrics para IDE *Eclipse* (2011).

Partindo do princípio que a classe *Agenda* quase não sofreu alterações durante o estudo de caso, subir de sete para oito não é ruim visto que para uma aplicação simples, esta métrica possui uma faixa que vai até dez. Em compensação, todos os outros pacotes da nova versão possuem índices baixos de complexidade, tornando as funções mais fáceis de serem entendidas.

A métrica WMC reduziu em trinta e três pontos, baixando sua média em dezenove e meio por cento, diminuindo a especificidade da aplicação e favorecendo sua reutilização.

De acordo com os números DIT encontrados, a nova versão atinge o dobro de pontos (indo a dois) e indica que as superclasses criadas estão melhores preparadas para reutilização do que ao se utilizar a versão antiga. Esta métrica prevê uma melhora no nível de abstração do sistema, aumentando o reuso bem como a complexidade do sistema.

A versão antiga não possui nenhuma herança de acordo com a métrica NOC. Já a versão atual possui classes com no máximo dois filhos. Valores altos indicam que este conceito de orientação a objetos não ocorre de maneira satisfatória, o que não é o caso para a versão atual. O desvio padrão desta nova aplicação Agenda ligeiramente indica que a maioria das classes não reflete a quantia máxima da métrica NOC observada, o que impede a redução das chances de reutilização. As classes envolvidas nesta métrica com certeza correspondem a um padrão criacional adotado, o *Abstract Factory*.

A métrica LCOM reflete positivamente com uma queda maior que a metade da média da versão antiga para a versão atual do *software*. Esta redução permite afirmar que o sistema atual é mais coeso e possui menor complexidade que a versão antiga. Valores mais baixos garantem maiores acesso a reutilização das classes da aplicação.

Com relação à métrica NOO, as etapas que se sucederam perante este trabalho não demonstraram indícios de problemas estruturais na medida em que se manteve o valor zero mesmo após todas as alterações feitas e subclasses criadas.

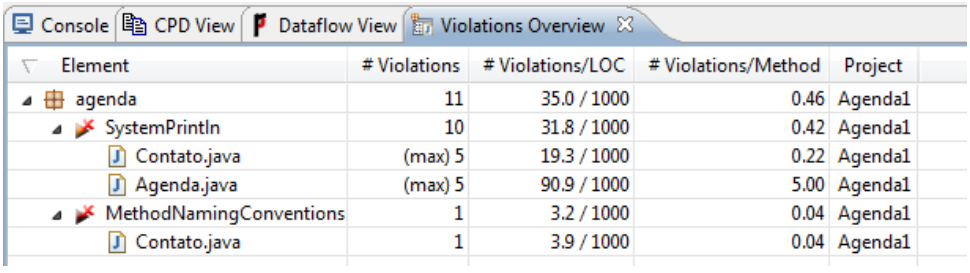
Assim como a métrica anterior, o índice de especialização (SI) não demonstrou problemas estruturais e garante que a reestruturação da aplicação não comprometeu princípios básicos da orientação a objetos como a reutilização. Este índice também relata que existem poucas ou nenhumas classes onde se adicionaram ou eliminaram métodos.

Outras métricas foram aplicadas aos dois projetos, como a Profundidade de blocos aninhados (NBD), cuja versão mais antiga alcançou máxima de seis, ultrapassando a faixa de segurança proposta pelo programa analisador em questão . Enquanto a nova versão conseguiu reduzir este valor para cinco, fora da área de risco. Esta métrica indica a profundidade de blocos aninhados no código-fonte.

Em questões de linhas de código, a versão antiga possuía trezentos e quatro linhas de código (LOC) enquanto a nova versão chegou à marca dos quatrocentos e quarenta e cinco. Ambas não se encontram fora do padrão, mas cabe salientar que a aplicação antiga é composta de duas classes apenas enquanto a versão atual conta com mais de dez classes e três novos padrões aplicados.

4.3 ANALISANDO COM PMD

Para verificar aspectos relativos ao código-fonte foram aplicados checagens com o analisador PMD. As violações relatadas nesta seção são apenas com relação às prioridades um e dois do analisador visto que as outras três, de níveis mais baixos, são caracterizadas como avisos e não possuem o mesmo peso que violações caracterizadas como erros propriamente ditos. A figura a seguir mostra a visão geral sobre erros considerados pelo programa analisador na primeira versão da aplicação, antes de serem executadas quaisquer fases do presente trabalho.



Element	# Violations	# Violations/LOC	# Violations/Method	Project
agenda	11	35.0 / 1000	0.46	Agenda1
SystemPrintln	10	31.8 / 1000	0.42	Agenda1
Contato.java	(max) 5	19.3 / 1000	0.22	Agenda1
Agenda.java	(max) 5	90.9 / 1000	5.00	Agenda1
MethodNamingConventions	1	3.2 / 1000	0.04	Agenda1
Contato.java	1	3.9 / 1000	0.04	Agenda1

Figura 18 - violações apontadas pelo PMD à versão antiga do programa Agenda.

Fonte: Analisador PDM para IDE *Eclipse* (2011).

A violação de nome *SystemPrintln*, encontrada nas duas únicas classes do projeto antigo, refere-se ao uso do comando de mesmo nome dentro da aplicação. Observa-se a falta de uma *interface* de comunicação o que resulta no uso deste meio para a impressão dos dados em tela.

MethodNamingConventions é uma violação apontada pelo mal uso de boas práticas de programação e *design*o de nomes para as operações. Constata-se que a primeira letra do método *ListaCategoria*() está em maiúscula, dando origem a esta violação.

Após a aplicação de todas as fases observadas em capítulos anteriores é realizada mais uma checagem com o analisador PMD cujo resultando é mostrado na figura a seguir.

Console | CPD View | Dataflow View | Violations Overview

Element	# Violations	# Violations/LOC	# Violations/Method	Project
logicaNegocio	8	106.7 / 1000	0.42	Agenda
SystemPrintln	4	53.3 / 1000	0.21	Agenda
Contato.java	4	88.9 / 1000	0.36	Agenda
MethodNamingConventions	4	53.3 / 1000	0.21	Agenda
Categoria.java	2	133.3 / 1000	0.50	Agenda
Categoria.java	2	133.3 / 1000	0.50	Agenda
dao	14	54.7 / 1000	0.36	Agenda
SystemPrintln	10	39.1 / 1000	0.26	Agenda
ContatoDAOImpl.java	(max) 5	70.4 / 1000	1.25	Agenda
CategoriaDAOImpl.java	(max) 5	71.4 / 1000	1.00	Agenda
MethodNamingConventions	4	15.6 / 1000	0.10	Agenda
CategoriaDAOImpl.java	1	14.3 / 1000	0.20	Agenda
XMLCategoriaDAOImpl.java	1	50.0 / 1000	0.20	Agenda
CategoriaDAO.java	1	250.0 / 1000	0.33	Agenda
XMLCategoriaDAOImpl.java	1	50.0 / 1000	0.20	Agenda
conexao	4	38.5 / 1000	0.20	Agenda
SystemPrintln	4	38.5 / 1000	0.20	Agenda
Conexao.java	2	38.5 / 1000	0.20	Agenda
Conexao.java	2	38.5 / 1000	0.20	Agenda
agenda	10	84.7 / 1000	5.00	Agenda
SystemPrintln	10	84.7 / 1000	5.00	Agenda
Agenda.java	(max) 5	84.7 / 1000	5.00	Agenda
Agenda.java	(max) 5	84.7 / 1000	5.00	Agenda

Figura 19 - violações apontadas pelo PMD à versão mais recente do programa Agenda.

Fonte: Analisador PDM para IDE *Eclipse* (2011).

Nota-se a presença das mesmas violações apresentadas na primeira versão. O que acontece agora é uma multiplicação destes mesmos dois erros por entre todas as novas classes derivadas da redistribuição de responsabilidades e implantação dos padrões criacionais nas fases que se sucederam.

Mais especificamente, a criação da nova classe *Categoria* (nova detentora do método que se aponta violação) originou a propagação deste mesmo erro em toda ação de troca de

mensagem para se listar as categorias dentro do sistema, função *design*ada ao método em questão. Em nenhum momento a aplicação de refatorações, em qualquer fase do estudo de caso, acrescentou uma violação de contexto diferente. Com um pouco de otimismo, o uso desta, além de prover um momento de reflexão ao código e apontar o erro que antes passava despercebido, manteve uma das premissas no que se diz respeito à preservação do comportamento.

4.4 DIFICULDADES ENCONTRADAS

As principais dificuldades encontradas foram com relação ao tipo e gênero da aplicação a ser refatorada e a aplicação de padrões de uma maneira simplista e de certa maneira adaptada de forma a contribuir positivamente para o trabalho.

A migração de projetos entre os computadores utilizados para realização deste trabalho através da IDE *Eclipse* se mostrou bastante confusa. Por muitas vezes tornou-se necessário recriar um projeto novo e adicionar as classes manualmente. A *interface* para a ação de importar projetos é muito resumida e sem muita explicação.

5 CONCLUSÕES

Durante todo este trabalho ficou evidente a necessidade de se construir *softwares* além do que espera pelo prazo que se possui.

Técnicas de refatoração foram explicadas e extensamente mostradas como instrumento essencial para se modificar a estrutura interna de um código-fonte sem que se perca o comportamento anterior. Conceitos e objetivos sobre a engenharia de *software* norteiam o referencial teórico como forma a auxiliar o entendimento e uso de métricas e padrões de projeto bem como as próprias refatorações. Os padrões de projetos e métricas em conjunto com outros conceitos explanados direcionaram o trabalho para compor a solução na qual se objetivou: promover melhorias em *software* existente de forma aplicar refatorações e padrões de projetos criacionais para poder garantir a sua evolução de maneira fácil, flexível e com poucos esforços de manutenção.

Em detrimento desta visão, fora apresentada uma aplicação simples de agenda com sérios problemas estruturais e algumas análises dos principais problemas se tornaram aspectos chaves do estudo de caso inferindo respostas a estes problemas através das técnicas já citadas.

Desde a aplicação da primeira fase do estudo já se observava melhorias empíricas sobre crescimento da aplicação. Após sete fases, a aplicação se transformou em uma versão muito mais robusta e flexível.

Esta se comprovou dentro das métricas que aferiram valores quantitativos demonstrando que os padrões e técnicas utilizadas favoreceram o desenvolvimento deste arcabouço que se transformou a aplicação. Disponível para assumir requisitos e soluções em diversos domínios e permeando várias fontes de dados vistos a sua generalidade adquirida com os conceitos agregados às técnicas abordadas, a aplicação da agenda de certa maneira ainda carece de aplicações de novas teorias e conceitos a cerca dos pacotes poucos analisados por este trabalho.

De posse dos resultados obtidos assim concluí-se de forma verdadeira, que grande parte das asserções feitas no início do trabalho culminou em uma forma nova e inspiradora de atender aos requisitos evolutivos do *software* de maneira flexível e manutenível: inserindo padrões de projetos por meio de refatorações.

6 TRABALHOS FUTUROS

De forma a dar continuidade a esta pequena contribuição ao meio acadêmico, podem-se:

- Explorar classes como: C*onexão*, de *logicaNegocios* assim como a classe principal de forma a aplicar tanto uma refatoração com padrões bem como para simples melhoria da estrutura;
- Inserir novos requisitos e utilizar novos padrões para analisar outras métricas;
- Procurar aplicar métricas antes do início de um projeto semelhante e avaliá-las como forma de gerencia de processos;
- Estabelecer um novo projeto de agenda, mas desta vez aplicando as refatorações em conjunto com os responsáveis por implementar novos requisitos com forma a provar a imutabilidade do comportamento proporcionado por esta técnica.

REFERÊNCIAS

ALUR, Deepak; CRUPI, John.; MALKS, Dan. **Core J2EE patterns**: best practices and *design* strategies. 1 ed. [S.l.]: Prentice Hall / Sun Microsystems, 2001.

CORNÉLIO, Márcio L. **Refactorings as Formal Refinements**. 2004. 327 f. Tese (Doutorado em Ciência da Computação) – Pós-Graduação em Ciência da Computação do Centro de Informática, Universidade Federal de Pernambuco, 2004. Disponível em: < http://dsc.upe.br/~mlc/PhDThesis/RefactoringsAsFormalRefinements.pdf> Acesso em: 26 ago. 2011.

CURSINO, Wilson S. **SCEA - Sun Certify Enterprise Architect : Guia de estudo**. Disponível em: < http://j2eecertify.bravehost.com/index.html.pdf> Acesso em: 26 out. 2011.

FERREIRA, Aurélio B. de H. **Mini Aurélio**. 7ª edição. Curitiba: Positivo, 2008.

FILHO, Wilson de P. P. **Engenharia de Software**: Fundamentos, Métodos e Padrões. Rio de Janeiro: LTC, 2001

FIORINI, Soeli. T.; STAA, Arndt, V.; BAPTISTA, Renan M. **Engenharia de Software com CMM**. Rio de Janeiro: Brasport, 1998.

FOWLER, Martin. **Refactoring**: improving the *design* of existing code. 1. Ed. 9ª Imp, Boston: Addison Wesley, 2002.

FOWLER, Martin: Refactoring in Alphabetical Order. Disponível em: < http://refactoring.com/catalog/>. Acesso em: 14 set. 2011.

FREIRE, A.; CHEQUE, P. Refatoração: Melhorando a Qualidade de Código Pré-Existente. In: CURSO DE VERÃO. 2007, São Paulo. **Anais**... IME/USP, 2007. Disponível em< http://ccsl.ime.usp.br/ agilcoop/files/2-Refatoracao.pdf >. Acesso em: 26 ago. 2011.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Padrões de projeto**: soluções reutilizáveis de software orientado a objetos. 1. ed. [S.l.], Bookman, 2000.

HIGO, Yoshiki; KUSUMOTO, Shinji; INOUE, Katsuro. On Refactoring for Open Source Java Program. Escola de Ciência da Informação e Tecnologia, Universidade de Osaka, 2003. Disponível em: <

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.8158&rep=rep1&type=pdf > Acesso 26 ago. de 2011.

INSTITUTE OF ELECTRIC AND ELECTRONIC ENGINEERS. **IEEE Std 610 12-1990**: Standard Glossary of Software Engineering Terminology. New York, 1990.

KON, Fábio; GOLDMAN, Alfredo. Métodos Ágeis e Programação eXtrema: Desenvolvendo Software com Qualidade e Agilidade. In: Simpósio Brasileiro de Engenharia de Software, 2003, Manaus. **Anais...** IME/USP, 2000. Disponível em< www.ime.usp.br/~gold/xp/SBES03-extended.ppt>. Acesso em: 13 out. 2011

LAUDELINO, Alvinei S.; KRIK, Daniel O.; MAIA, Fernando E. P. **Padrões de Projeto**. Disponível em: < http://www.pg.cefetpr.br/coinf/simone/patterns/factory.php> Acesso em: 26 out. 2011.

MAFFEO, Bruno. **Engenharia de Software e especificação de sistemas**. Rio de Janeiro: Campus, 1992.

MENS, Tom; TOURWÉ, Tom. A Survey of Software Refactoring. 2004. IEEE Transactions on software engineering, vol. 30, n. 2, p. 126-139, Jan. 2004. Disponível em: < http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.9021&rep=rep1&type=pdf > Acesso em: 26 set. 2011

PRESSMAN, Roger S. **Engenharia de Software**. 6. ed. São Paulo: Mcgraw-Hill Brasil, 2006.

PRESSMAN, Roger S. **Engenharia de Software**. 3. ed. São Paulo: Makron Books, 1995.

PETERS, James F; PEDRYCZ, Witold. **Engenharia de Software**: teoria e prática. Rio de Janeiro: Campus, 2001.

RAPELI, Leide R. C. **Refatoração de sistemas Java utilizando padrões de projeto: um estudo de caso**. 2006. 130f. Dissertação (Ciência da Computação) — Programa de Pós-Graduação em Ciência da Computação, Universidade Federal de São Carlos, 2005. Disponível em: < http://www.bdtd.ufscar.br/htdocs/tedeSimplificado/tde_arquivos/3/TDE-2006-03-17T12:41:40Z-909/Publico/DissLRCR.pdf > Acesso 26 ago. de 2011.

REZENDE, Denis A. **Engenharia de software e sistemas de informação**. 3. ed. Rio de Janeiro: Brasport, 2005.

SCHACH, Stephen R. **Object-oriented and classical software engineering**. 8. ed. New york: McGraw-Hill, 2010.

SOMMERVILLE, Ian. **Engenharia de Software**. 4. ed. [S.l.]: Addison-Wesley, 1992.

SOMMERVILLE, Ian. **Engenharia de Software**. 6. ed. [S.l.]: Addison-Wesley, 2004.

SOMMERVILLE, Ian. **Engenharia de Software**. 8. ed. [S.l.]: Addison-Wesley, 2006.

SUN JAVA CENTER. ***Blueprints***. Disponível em: < http://www.oracle.com/technetwork/java/javaee/blueprints/index.html> Acesso em: 20 out. 2011.

TELES, Vinícius M. **Um estudo de caso da adoção das práticas e valores do Extreme Programming**. 2005. 152 f. Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro - UFRJ, IM / DCC, 2005. Disponível em: <http://www.ime.usp.br/~ale/Dissertacao.pdf> Acesso em: 15 set. 2011.

ANEXO A – Java Singleton Template for Eclipse

Este é um *template* para criar facilmente uma implementação do padrão *Singleton* na IDE *Eclipse*. Abra Window->Preferences->Java->Editor->Templates em seguida clique em *new* e insira o código a seguir na área de texto do *pattern*. Adicione um nome, eu sugiro "singleton". Aonde quer que você digite este nome e pressione *Ctrl+Space* o código será inserido na sua classe, e você está bom para ir.

```
private static ${enclosing_type} instance;
private ${enclosing_type}(){}

public static ${enclosing_type} getInstance(){
	if(null == instance){
		instance = new ${enclosing_type}();
	}
	return instance;
}
```

Pode ser encontrado em: http://snippets.dzone.com/tag/singleton