UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Refatoração de sistemas Java utilizando padrões de projeto: um estudo de caso

Leide Rachel Chiusi Rapeli

Orientadora: Profª. Drª. Rosângela Ap. Dellosso Penteado

São Carlos – SP
2006

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

R216rs

Rapeli, Leide Rachel Chiusi.
Refatoração de sistemas Java utilizando padrões de projeto: um estudo de caso / Leide Rachel Chiusi Rapeli. -- São Carlos : UFSCar, 2006.
127 p.
Acompanha CD

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2005.

1. Manutenção de programas. 2. Padrões de projeto. 3. Engenharia reversa. 4. Reengenharia orientada a objetos. 5. Refatoração. I. Título.

CDD: 005.16 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Refatoração de sistemas Java utilizando padrões de projeto: um estudo de caso"

LEIDE RACHEL CHIUSI RAPELI

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Membros da Banca:

Profa. Dra. Rosângela Aparecida D. Penteado
(Orientadora – DC/UFSCar)

Profa. Dra. Júnia Coutinho Anacleto
(DC/UFSCar)

Prof. Dr. Marcos Lordello Chaim
(EACH/USP/SP)

São Carlos
Dezembro/2005

Para Renan e Bruna,
minhas maiores riquezas.

Agradecimentos

Agradeço a Deus pela vida e por me dar condições para a realização deste trabalho.

À minha orientadora Rosângela, pelo estímulo e dedicação para que este projeto fosse desenvolvido.

Aos meus pais Roberto e Isabel pelo apoio, compreensão e carinho em mais esta etapa de minha vida.

À minha irmã Mileide, com quem troquei experiências e que me acompanhou neste trabalho.

Ao meu querido esposo Clayton, por estar ao meu lado com palavras de incentivo e força, nos momentos mais críticos.

Aos meus filhos amados Renan e Bruna, aos quais atenção foi negada em alguns momentos devido à realização deste trabalho.

À minha amiga Cinthia pelo companheirismo e amizade durante todos estes anos.

Aos colaboradores da S&V, por acreditarem em meu potencial.

Às funcionárias do DC, Cristina e Mirian, pelo apoio administrativo.

Resumo

Esta dissertação apresenta um estudo de caso de refatoração de sistemas orientados a objetos. Sistemas desenvolvidos de acordo com o paradigma de orientação a objetos podem conter código reusável, apesar de nem sempre terem sido projetados para isso. Manutenção de sistemas não é uma tarefa fácil, nem mesmo para sistemas orientados a objetos. Padrões de projeto de software favorecem a implementação de soluções eficientes para problemas recorrentes, facilitando a reusabilidade e manutenibilidade. Nos casos dos sistemas não projetados com o seu uso, é possível refatorar o sistema usando-os, sem alterar a sua funcionalidade. Este estudo prospectivo refere-se à busca de diretrizes para auxiliar o engenheiro de software a conduzir esse tipo de refatoração. Para isso, sete sistemas implementados em Java que estavam disponíveis na Web foram usados. O estudo tem três etapas: a primeira refere-se à recuperação da funcionalidade e do modelo de classes do sistema existente de modo que, na segunda etapa, padrões de projeto de software aplicáveis possam ser implementados; e o modelo de classes previamente obtido possa ser atualizado com os padrões de projeto de software aplicados. Na terceira etapa, a funcionalidade do novo sistema é verificada por testes, para confirmar que a refatoração conduzida não a alterou. O sistema refatorado, usualmente, apresenta um aumento no número de LOC, mas torna-se mais manutenível, devido a melhor estruturação e coesão. O reuso é também facilitado. Uma restrição desta pesquisa é que o estudo de caso foi conduzido apenas para sistemas de informação de pequeno porte.

Abstract

This dissertation presents a study case on object-oriented systems refactoring. Systems developed according to the object-oriented paradigm may contain reusable code, even though not always have been designed for it. Systems maintenance is not an easy task, even for object-oriented systems. Software design patterns favour the implementation of efficient solutions for recurrent problems, easing reusability and maintainability. In cases of systems not designed using design patterns, it is possible to refactor the system using them, without altering the system functionality. This prospective study refers to the search for guidelines to help the software engineer to conduct this type of refactoring. For that purpose, seven systems implemented in Java, that were available in the Web, have been used. The study has three phases: the first refers to existing system functionality and class model recovery; so that, in the second phase, applicable software design patterns can be implemented; and the class model previously obtained can be updated with the software design patterns applied. In the third phase, the new system functionality is verified by tests, to confirm that the conducted refactoring has not altered it. The refactored system usually presents an increase in the LOC number, but becomes more maintainable due to better structuring and cohesion. Reuse is also eased. One restriction of this research is that the case study has been conducted only for small-scale information systems.

Sumário

Capítulo 1 Introdução **1**
1.1 Contextualização 1
1.2 Objetivo e Motivação 2
1.3 Relevância 2
1.4 Organização da Dissertação 3

Capítulo 2 Assuntos Abordados **4**
2.1 Considerações Iniciais 4
2.2 Padrões de Software 5
2.3 Padrões de Projeto de Software 7
2.3.1 Catálogo de Padrões de Projeto de Software 8
2.4 Manutenção de Software 13
2.5 Métricas de Software Orientado a Objetos 18
2.6 Padrões de Projeto x Refatoração x Métricas 20
2.7 Considerações Finais 22

Capítulo 3 Processo de Refatoração com Padrões de Projeto **23**
3.1 Considerações Iniciais 23
3.2 Abrangência e Concepção do Processo de Refatoração 24
3.3 Etapa 1: Entender o sistema 26
3.4 Etapa 2: Refatorar o código-fonte utilizando padrões de projeto 33
3.4.1 Refatoração com o padrão *Singleton* (Criação) 34
3.4.2 Refatoração com o padrão *Decorator* (Estrutural) 35
3.4.3 Refatoração com o padrão *Visitor* (Comportamental) 37
3.5 Etapa 3: Verificar sistema após refatoração 40
3.6 Considerações Finais 40

Capítulo 4 Estudo de Caso **42**
4.1 Considerações Iniciais 42
4.2 Detalhamento da Aplicação do Processo 43
4.3 Primeira Refatoração: Padrão *Singleton* 45
4.3.1 Etapa 1: Entender o sistema 45
4.3.2 Etapa 2: Refatorar o sistema utilizando padrões de projeto 49
4.3.3 Etapa 3: Verificar sistema após refatoração 50
4.4 Segunda Refatoração: Padrão *Decorator* 50
4.4.1 Etapa 1: Entender o sistema 51
4.4.2 Etapa 2: Refatorar o sistema utilizando padrões de projeto 51
4.4.3 Etapa 3: Verificar sistema após refatoração 53
4.5 Terceira Refatoração: Padrão *Visitor* 53
4.5.1 Etapa 1: Entender o sistema 54
4.5.2 Etapa 2: Refatorar o sistema utilizando padrões de projeto 54

4.5.3 Etapa 3: Verificar sistema após refatoração 56
4.6 Aplicação de Métricas 58
4.7 Considerações Finais 59
4.7.1 Outros sistemas estudados 61

Capítulo 5 Considerações Finais 64
5.1 Considerações Iniciais 64
5.2 Contribuições deste Processo 65
5.3 Trabalhos Futuros 65

Apêndice A 67
Categoria: Criação 67
Abstract Factory 67
Builder 70
Factory Method 73
Prototype 75
Categoria: Estrutural 76
Adapter 76
Bridge 78
Composite 80
Façade 82
Flyweight 84
Proxy 87
Categoria: Comportamental 89
Chain of Responsibility 89
Command 91
Interpreter 92
Iterator 96
Mediator 97
Memento 101
Observer 103
State 105
Strategy 108
Template Method 110

Anexos 112
Anexo 1: CD-ROM 112

Referências Bibliográficas 113

Lista de Figuras

Figura 1 – Processo de refatoração com padrões de projeto _____ 26
Figura 2 – Tela de *login* do sistema de videolocadora _____ 45
Figura 3 – Menu de opções do sistema de videolocadora _____ 46
Figura 4 – Modelo de classes do sistema de videolocadora antes da refatoração _____ 48
Figura 5 – Modelo de classes do sistema de videolocadora após as dez refatorações _____ 57

Lista de Tabelas

Tabela 1 – Classificação dos padrões de projeto (Gamma et al., 1995) 9
Tabela 2 – Padrões e reformulação de projeto 12
Tabela 3 – Sistemas de estudos de caso e padrões implementados 25
Tabela 4 – Leiaute dos dados a serem obtidos ao executar o sistema 27
Tabela 5 – Indícios da aplicabilidade dos padrões no sistema existente 28
Tabela 6 – Resumo do processo de refatoração com padrões de projeto 43
Tabela 7 – Interações do usuário com o sistema de videolocadora 46
Tabela 8 – Métricas utilizadas no sistema de videolocadora 58
Tabela 9 – Resultado de aplicação das métricas 59

Lista de Quadros

Quadro 1 – Classe Singleton_cp 35
Quadro 2 – Classe X_ap, padrão *Singleton* 35
Quadro 3 – Classe X, padrão *Decorator* 36
Quadro 4 – Interface IntX 36
Quadro 5 – Classe abstrata Decorator_cp 36
Quadro 6 – Classe ConcreteDecorator_cp 37
Quadro 7 – Classe Y_ap, padrão *Decorator* 37
Quadro 8 – Classe X_ap, padrão *Visitor* 38
Quadro 9 – Classe Z_ap, padrão *Visitor* 38
Quadro 10 – Classe Visitor_cp 38
Quadro 11 – Classe Y_ap, padrão *Visitor* 39
Quadro 12 – Classe XZVisitor_cp, padrão *Visitor* 39
Quadro 13 – Classe Y_ap, padrão *Visitor* 39
Quadro 14 – Código identificado com o padrão *Singleton* 49
Quadro 15 – Classe alterada com o padrão *Singleton* 50
Quadro 16 – Classe EmpFilme, identificada com o padrão *Decorator* 51
Quadro 17 – Classe Cliente_cp 52
Quadro 18 – Classe Decorator_cp 52
Quadro 19 – Classe ConcreteDecorator_cp 52
Quadro 20 – Classe RelatorioLocacoes, identificada com o padrão *Visitor* 54
Quadro 21 – Classe Cliente_cp 55
Quadro 22 – Classe Visitor_cp 55
Quadro 23 – Classe Relatorio1Visitor_cp 55
Quadro 24 – Classe RelatorioLocacoes_ap, com o padrão *Visitor* 56
Quadro 25 – Classe AbstractFactory_cp 68
Quadro 26 – Classe Factory_cp, padrão *Abstract Factory* 68
Quadro 27 – Classe ClassA, padrão *Abstract Factory* 68
Quadro 28 – Classe ClassB, padrão *Abstract Factory* 69
Quadro 29 – Classe X_ap, padrão *Abstract Factory* 69
Quadro 30 – Código a ser retirado da classe X_ap, padrão *Abstract Factory* 69
Quadro 31 – Código a ser inserido na classe X_ap, padrão *Abstract Factory* 70
Quadro 32 – Classe Builder_cp 70
Quadro 33 – Classe Director_cp 70
Quadro 34 – Classe herdeira de Builder_cp 71
Quadro 35 – Classe BuilderFactory_cp 72
Quadro 36 – Classe X_ap, padrão *Builder* 72
Quadro 37 – Interface da fábrica 73
Quadro 38 – Classe que implementa a interface da fábrica 74
Quadro 39 – Classe Factory_cp, padrão *Factory Method* 74

Quadro 40 – Classe X_ap, padrão *Factory Method* 75
Quadro 41 – Classe Y_ap, padrão *Prototype* 75
Quadro 42 – Código a ser retirado da classe X_ap, padrão *Prototype* 76
Quadro 43 – Código a ser inserido na classe X_ap, padrão *Prototype* 76
Quadro 44 – Classe X, padrão *Adapter* 76
Quadro 45 – Classe Adapter_cp, padrão *Adapter* 77
Quadro 46 – Código a ser retirado da classe Y_ap, padrão *Adapter* 78
Quadro 47 – Código a ser inserido na classe Y_ap, padrão *Adapter* 78
Quadro 48 – Classe X, padrão *Bridge* 78
Quadro 49 – Classe A, padrão *Bridge* 79
Quadro 50 – Classe Abstraction_cp, padrão *Bridge* 79
Quadro 51 – Classe W_ap , padrão *Bridge* 79
Quadro 52 – Classe X_cp , padrão *Composite* 80
Quadro 53 – Classe Y_ap, padrão *Composite* 81
Quadro 54 – Classe Y_ap, padrão *Composite* 81
Quadro 55 – Classe Y_ap, padrão Composite 82
Quadro 56 – Classes da interface *Façade* 83
Quadro 57 – Classe cliente da interface *Façade* 83
Quadro 58 – Classe Facade_cp 84
Quadro 59 – Reestruturação da classe C1 84
Quadro 60 – Classe X_ap, padrão *Flyweight* 85
Quadro 61 – Classe Flyweight_cp 85
Quadro 62 – Classe ConcreteFlyweight_cp 85
Quadro 63 – Classe FlyweightFactory_cp 86
Quadro 64 – Classe X_ap, padrão *Flyweight* 87
Quadro 65 – Classe X_ap, padrão *Proxy* 87
Quadro 66 – Classe Subject_cp, padrão *Proxy* 88
Quadro 67 – Classe Proxy_cp 88
Quadro 68 – Classe RealSubject_cp 88
Quadro 69 – Classe Handler_cp 89
Quadro 70 – Classe X_ap, padrão *Chain of Responsability* 89
Quadro 71 – Classe ConcreteHandler1_cp 90
Quadro 72 – Classe ConcreteHandler2_cp 90
Quadro 73 – Modificação na classe X_ap, padrão *Chain of Responsability* 91
Quadro 74 – Classe X_ap, padrão *Command* 91
Quadro 75 – Modificações na classe X_ap, padrão *Command* 92
Quadro 76 – Classe AbstractExpression_cp 93
Quadro 77 – Classe TerminalExpression_cp 93
Quadro 78 – Classe NonTerminalExpression_cp 93
Quadro 79 – Classe Context_cp, padrão *Interpreter* 94
Quadro 80 – Classe X_ap, padrão *Interpreter* 95
Quadro 81 – Classe X_ap, padrão *Iterator* 96
Quadro 82 – Modificações na classe X_ap, padrão *Iterator* 97
Quadro 83 – Classe X_ap, padrão *Mediator* 98
Quadro 84 – Classe Mediator_cp 98
Quadro 85 – Classe ConcreteMediator_cp 99
Quadro 86 – Classe ConcreteColleagueObj1_cp 100
Quadro 87 – Classe X_ap, padrão *Mediator* 101
Quadro 88 – Classe Memento_cp 102
Quadro 89 – Classe Originator_cp 102

Quadro 90 – Classe X_ap, padrão *Memento* ____ 103
Quadro 91 – Classe ConcreteSubject_cp ____ 104
Quadro 92 – Classe ConcreteObserver_cp ____ 104
Quadro 93 – Classe X_ap, padrão *Observer* ____ 105
Quadro 94 – Interface State_cp ____ 105
Quadro 95 – Classe Context_cp, padrão *State* ____ 106
Quadro 96 – Classe X_ap, padrão *State* ____ 106
Quadro 97 – Classe State1_cp ____ 107
Quadro 98 – Classe State2_cp ____ 107
Quadro 99 – Classe X_ap, padrão *State* ____ 107
Quadro 100 – Classe Strategy_cp ____ 108
Quadro 101 – Classe Context_cp, padrão *Strategy* ____ 109
Quadro 102 – Classe ConcreteStrategy1_cp ____ 109
Quadro 103 – Modificações na classe X_ap, padrão *Strategy* ____ 110
Quadro 104 – Classe X_ap, padrão *Template Method* ____ 110
Quadro 105 – Classe ConcreteMetodo1X_cp, padrão *Template Method* ____ 111
Quadro 106 – Classe X_ap, padrão *Template Method* ____ 111

Capítulo 1
Introdução

1.1 Contextualização

Sistemas de software que são liberados para serem utilizados pelos usuários finais são freqüentemente avaliados quanto aos seus benefícios, correteza, facilidade de operação, dentre outras características. Quando o software é finalizado e, após ser testado pelo usuário, não atinge seus objetivos e satisfação, deve sofrer correções a fim de se tornar adequado às necessidades. Mesmo quando o software não deve ser alterado por esse motivo, deve ser flexível o suficiente para que suporte, futuramente, novos requisitos ou ainda para que atenda a novas tecnologias. Nessas situações, é sempre mais vantajoso reutilizar o software existente a construir um novo, o que aumenta os custos, o tempo de trabalho e o esforço dispendido; além da possibilidade do novo sistema também não ser adequado ao uso.

As correções que devem ser realizadas no sistema em funcionamento são denominadas manutenção do software e envolvem não somente correção, mas adaptação a outras tecnologias e adição de nova funcionalidade, facilitando manutenções futuras. Para que a manutenção seja realizada adequadamente, o software deve ser flexível e, quando construído, deve prever as alterações. Dessa forma, para facilitar o desenvolvimento de sistemas mais flexíveis diversas técnicas e linguagens de programação são utilizadas como, por exemplo, programação orientada a objetos, linguagem Java, estruturação por meio de padrões de projeto, etc. Entretanto, mesmo que o desenvolvimento tenha ocorrido com métodos e ferramentas adequados, em algum momento os sistemas devem sofrer manutenções. Uma das técnicas de manutenção para o software desenvolvido com o paradigma orientado a objetos é a refatoração, em que o código do sistema é reestruturado e sua funcionalidade é mantida.

Para medir a qualidade do software, refatorado ou não, engenheiros de software utilizam diversas métricas de avaliação, presentes em várias etapas do ciclo de vida de

desenvolvimento do software. Dentre essas métricas merecem atenção as de projeto, pois avaliam o software que ainda está sendo construído e não apenas o produto final, que é mais difícil de ser alterado e corrigido.

Dada a grande quantidade de sistemas desenvolvidos com o paradigma orientado a objetos e que necessitam de manutenção, uma abordagem promissora é a refatoração utilizando padrões de projeto, pois ao mesmo tempo em que provê melhoria na estrutura do sistema, aumenta sua flexibilidade com o uso dos padrões. Estima-se que o software refatorado dessa forma seja mais fácil de se manter e, conseqüentemente, de sofrer evoluções.

1.2 Objetivo e Motivação

Dada a importância da refatoração utilizando padrões de projeto de software, para melhorar a manutenção e prover maior reusabilidade, este trabalho propõe um processo de refatoração utilizando padrões de projeto que foi elaborado com base em estudos de caso utilizando sistemas de informação. Esse processo conduz, de modo definido e organizado, o engenheiro de software à identificação de padrões de projeto em sistemas orientados a objetos já existentes, implementados com a linguagem Java e sua posterior refatoração com o padrão. Durante esse processo a documentação sobre o sistema é obtida e atualizada.

A principal motivação para a realização deste trabalho é a existência de sistemas orientados a objetos e implementados na linguagem Java, que possuem baixa reusabilidade de código, manutenção difícil e a adição de novas funções e/ou adequação a novos requisitos exigem muito esforço do engenheiro de software e dispendem grande custo à organização. Refatorando esses sistemas utilizando padrões de projeto de software objetiva-se aumentar sua manutenibilidade, reusabilidade e facilitar sua evolução, por meio de novas funções.

1.3 Relevância

O processo auxilia engenheiros de software a refatorar sistemas orientados a objetos, no domínio de sistemas de informação, utilizando padrões de projeto de software. A reestruturação desses sistemas resulta em redução de custos e esforços em manutenção. O processo pode ser utilizado tanto para reestruturação do código existente quanto como ferramenta de engenharia reversa, em que sua funcionalidade é conhecida.

1.4 Organização da Dissertação

Esta dissertação está organizada em cinco capítulos, além deste. Do capítulo 2 constam os assuntos que serviram de base para o desenvolvimento deste projeto. Foram estudados padrões e, com mais detalhamento, os padrões de projeto de software definidos por Gamma et al. (1995); as formas de manutenção de software e, dentre elas, refatoração e reengenharia; métricas de projeto de software e os trabalhos que relacionam padrões, refatoração e métricas.

No capítulo 3 são apresentadas diretrizes que auxiliam a refatoração de sistemas implementados em linguagem Java utilizando padrões de projeto. Essas diretrizes foram elaboradas de forma prospectiva utilizando-se sete sistemas obtidos na Internet, implementados em linguagem Java, cujo código estava disponível. Uma lista de indícios de que um padrão pode ser encontrado é apresentada para os 23 padrões de projeto propostos por Gamma et al. (1995).

O capítulo 4 aborda o estudo de caso realizado, a fim de exemplificar, analisar e avaliar o processo de refatoração proposto, embora não tenham sido aplicados os 23 padrões neste estudo de caso.

No capítulo 5 encontram-se as considerações finais comentando-se os resultados obtidos com o desenvolvimento do processo. Os trabalhos futuros a este também são apresentados.

Capítulo 2
Assuntos Abordados

2.1 Considerações Iniciais

Durante o ciclo de vida do software, uma das fases que mais consome recursos é a de manutenção. Isso acontece porque os sistemas não são bem projetados devido às necessidades urgentes do cliente e/ou da empresa, por não utilizarem tecnologia adequada e, na maioria das vezes, por não preverem mudanças futuras.

Dessa forma, desenvolvedores de sistemas procuram soluções para amenizar os problemas que ocorrem durante a manutenção. A busca por técnicas de desenvolvimento que sejam ágeis e que atendam rapidamente aos requisitos dos clientes vem crescendo na área de engenharia de software.

Padrões são soluções de sucesso comprovado que já foram usadas várias vezes, em um domínio específico. Vários padrões têm surgido, sendo que os de projeto (Gamma et al., 1995) foram os que impulsionaram os engenheiros de software a utilizarem e comprovarem o seu valor.

Dados os inúmeros projetos bem sucedidos que utilizam padrões de projeto de software, sua aplicação torna-se interessante à medida que aumenta a legibilidade do código e facilita sua manutenção. Refatoração utilizando padrões de projeto é uma abordagem promissora para a melhoria do projeto durante as atividades de desenvolvimento (Muraki e Saeki, 2001).

A fase de manutenção também é realizada por meio de refatoração e reengenharia, tornando os sistemas melhor estruturados e, conseqüentemente, aumentando sua manutenibilidade.

Desta seção em diante, os termos interface e interface com o usuário serão utilizados. Interface diz respeito ao conjunto de todas as assinaturas[1] definidas pelas operações de um

[1] Assinatura do método compreende seu nome, os objetos que aceita como parâmetro e o valor que retorna.

objeto. Já o termo interface com o usuário é a GUI (*Graphical User Interface*) adotada em um software para que o usuário final possa interagir com o mesmo.

Este capítulo tem por objetivo a apresentação dos temas relacionados ao projeto desenvolvido e está organizado da seguinte forma: a seção 2.2 apresenta os trabalhos relacionados aos padrões de software; a seção 2.3 aborda os padrões de projeto de software; a seção 2.4 trata de manutenção de software; a seção 2.5 apresenta métricas de projeto orientado a objetos; a seção 2.6 apresenta a relação entre padrões de projeto, refatoração e métricas e na seção 2.7 estão as considerações finais.

2.2 Padrões de Software

O uso do termo "padrão" teve início após as publicações do arquiteto Christopher Alexander (Alexander et al., 1977), a respeito de planejamento urbano e de arquitetura de construções. O autor afirma que um padrão, em geral, descreve um problema recorrente no ambiente real e o núcleo de sua solução, de tal forma que essa solução possa ser instanciada diversas vezes, sem nunca ser feita da mesma maneira.

A partir disso, diversos trabalhos relacionados a padrões são desenvolvidos até os dias atuais utilizando os conceitos definidos por Alexander. Na área de software, a popularização dos padrões acontece devido ao lançamento do livro *Design Patterns – Elements of Reusable Object-Oriented Software* (Gamma et al., 1995). Esse livro apresenta um catálogo com 23 padrões de projeto de software, que se tornou muito difundido e utilizado por propor boas soluções a problemas de projeto já amplamente discutidos.

Segundo Appleton (2000), para que uma solução, um algoritmo ou uma heurística possa ser considerado um padrão na área de engenharia de software, deve-se verificar a sua relação de recorrência, também denominada "prova dos três". Dessa forma, a solução candidata a padrão (até então chamada de "protopadrão") deve estar presente em pelo menos três sistemas distintos e sua eficácia ser comprovada. Entretanto, a recorrência não é a característica de maior importância de um padrão. Cada padrão por si só deve conter uma solução adequada e eficiente para um determinado problema, evidenciando sua utilidade.

Coplien (1991) afirma que bons padrões são caracterizados por: resolverem um problema, provarem um conceito, não apresentarem soluções óbvias, descreverem relacionamentos, servirem ao conforto e à qualidade de vida do homem.

Devido a essas particularidades, para que os padrões obtenham sucesso quando aplicados em cada fase do desenvolvimento, surgem categorias de padrões, que são mais específicas e que abrangem as necessidades de cada etapa. Entretanto, pode haver padrões que se classificam em mais de uma categoria.

Os padrões de arquitetura expressam a estrutura organizacional do sistema (Buschman et al., 1996). Contêm uma coleção de subsistemas pré-definidos, especificam suas responsabilidades e incluem regras para organizar o relacionamento entre eles.

Padrões de processo destinam-se a resolver problemas que ocorrem em todo o processo de desenvolvimento de software, como gerenciamento de configuração, testes, reengenharia, etc.

Os padrões de análise capturam modelos conceituais em um domínio de aplicação, permitindo seu uso em outras aplicações (Fowler, 1997). Esses padrões tratam as particularidades organizacionais, sociais e econômicas do sistema, que são o centro para a análise de requisitos e a aceitação e usabilidade do sistema final. Há duas contribuições principais dos padrões de análise: a) capturam modelos abstratos de forma ágil e os transformam em requisitos do problema real; b) facilitam a transformação do modelo de análise em um modelo de projeto, sugerindo padrões de projeto e soluções reais e eficazes para problemas comuns.

Padrões de projeto provêm um esquema para refinar os subsistemas ou componentes que compõem o sistema, ou o relacionamento entre eles (Buschman et al., 1996). Definem soluções para problemas que ocorrem na fase de projeto, em contextos particulares.

Os padrões de implementação (ou idiomas) são de baixo nível e definem regras específicas para determinadas linguagens de programação ou regras de estilos de programação (Buschman et al., 1996). Um idioma descreve como implementar particularidades de componentes ou seus relacionamentos utilizando características de uma dada linguagem.

Os padrões de manutenção documentam a manutenção de software especificando uma coleção de entidades logicamente relacionadas do ponto de vista de algum interesse em comum. (Hammouda e Harsu, 2004).

Devido à variedade de padrões, diversas formas para descrevê-los são utilizadas. Três são os formatos mais conhecidos: o de Alexander, o de Appleton (1997) o definido por Gamma et al. (1995). O formato de Alexander consiste na definição de três elementos: nome do padrão, a contextualização (e descrição); e a solução do problema (colaborações e conseqüências). O formato de Appleton define elementos essenciais à definição de qualquer

padrão: nome, problema, contexto, influências, solução, exemplos, contexto resultante, justificativa, padrões relacionados e usos conhecidos. Gamma et al. (1995) apresentam um formato diferente dos dois anteriores, a ser mostrado na seção 2.3. Contudo, os padrões podem ainda ser descritos de acordo com vários outros elementos, de acordo suas particularidades.

Além do agrupamento dos padrões quanto à fase do ciclo de vida na qual atuam, é possível agrupá-los em coleções, catálogos, sistemas e linguagens. Uma linguagem de padrões é uma coleção estruturada de padrões que se apóiam uns nos outros para transformar necessidades e restrições em uma arquitetura (Coplien, 1998). Grande parte de uma linguagem de padrões é formada pelos padrões em si, outra parte é constituída de uma visão sobre o domínio relacionado à linguagem, sua aplicabilidade e suas interações com outros padrões (Meszaros e Doble, 1998).

2.3 Padrões de Projeto de Software

De acordo com Gamma et al. (1995), os padrões de software orientados a objetos são definidos com base em entidades primitivas como objetos e classes. Por isso, um padrão deve abstrair, nomear e identificar pontos importantes de uma estrutura de projeto para torná-la útil na criação de um projeto orientado a objetos reutilizável. O padrão de projeto identifica as classes e instâncias participantes, seus papéis, colaborações e a distribuição de responsabilidades. Cada padrão trata de um problema ou um tópico particular de projeto orientado a objetos, descrevendo quando deve ser aplicado, se pode ser aplicado em função de outras restrições de projeto e as conseqüências, custos e benefícios de sua utilização.

Como comentado anteriormente, os padrões de projeto são descritos de acordo com o seguinte formato (Gamma et al., 1995):

- Nome e classificação: o nome deve expressar a essência do padrão e deve ser sucinto. A classificação é definida de acordo com o escopo e a finalidade do padrão.
- Intenção e objetivo: é uma breve descrição do que o padrão faz e qual problema soluciona.
- Também conhecido como: relata outros nomes do padrão, se existirem.
- Motivação: relata estudos de caso em que o padrão solucionou o problema no projeto, e ainda os elementos que foram organizados para tal.

- Aplicabilidade: atenta para as situações em que o padrão deve ser aplicado e como reconhecer tais situações.
- Estrutura: representação gráfica das classes do padrão.
- Participantes: descreve as classes e os objetos que participam do padrão.
- Colaborações: descreve como os participantes colaboram para executar suas tarefas.
- Conseqüências: descreve quais as relações de custo e benefício e os resultados da aplicação do padrão.
- Implementação: descreve sugestões, técnicas que devem ser conhecidas para a implementação, ou ainda considerações acerca de uma linguagem específica.
- Exemplo de código: exibe fragmentos de código do padrão implementado.
- Usos conhecidos: descreve projetos em que o padrão foi utilizado.
- Padrões relacionados: descreve os padrões intimamente ligados ao padrão em questão.

Segundo Gamma et al. (1995), padrões afetam a maneira como o software é projetado. Estipulando um vocabulário comum para o projeto, a comunicação, a documentação e a exploração de alternativas, padrões elevam o nível do projeto e das discussões em torno dele. O aprendizado de padrões ajuda um projetista inexperiente a trabalhar de maneira semelhante a um desenvolvedor experiente. O vocabulário comum torna desnecessário descrever o padrão empregado, apenas nomeia-se o padrão e espera-se que outros projetistas o conheçam.

2.3.1 Catálogo de Padrões de Projeto de Software

Os padrões de projeto variam de acordo com sua granularidade e seu nível de abstração. Assim, a classificação desses padrões é feita de acordo com dois critérios:

1. Finalidade: reflete o quê o padrão faz, podendo ser de criação (específicos para o processo de criação de objetos), estrutural (determinam a composição de classes ou objetos) ou comportamental (definem maneiras como os objetos interagem e distribuem responsabilidades).
2. Escopo: reflete a que o padrão se aplica, podendo ser a classes ou a objetos. Os padrões que lidam com classes determinam seus relacionamentos por meio de herança e, por isso, são estáticos. Os padrões que lidam com objetos são dinâmicos, pois podem ser alterados em tempo de execução.

A Tabela 1 apresenta a classificação dos padrões de projeto de acordo com esses dois critérios. O padrão *Adapter* é tanto um padrão estrutural de classes quanto de objetos.

Tabela 1 – Classificação dos padrões de projeto (Gamma et al., 1995)

		Finalidade		
		De criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	*Factory Method*	*Adapter*	*Interpreter* *Template Method*
	Objeto	*Abstract Factory* *Builder* *Prototype* *Singleton*	*Adapter* *Bridge* *Composite* *Decorator* *Façade* *Flyweight* *Proxy*	*Chain of Responsability* *Command* *Iterator* *Mediator* *Memento* *Observer* *State* *Strategy* *Visitor*

Os padrões de criação abstraem o processo de instanciação; ajudam a tornar um sistema independente de como seus objetos são criados, compostos e representados. Tornam-se importantes na evolução dos sistemas, quando a composição de objetos é mais utilizada do que a herança de classes. Esses padrões encapsulam conhecimento sobre quais classes concretas são usadas pelo sistema e ocultam o modo como as instâncias destas classes são criadas e compostas.

Os padrões estruturais preocupam-se com a forma como classes e objetos são compostos para formar estruturas maiores. Os padrões estruturais de classes utilizam herança para compor interfaces ou implementações. Padrões estruturais de objetos descrevem como compor objetos para obter funcionalidade adicional e, como essa composição pode ser alterada em tempo de execução, garante mais flexibilidade ao sistema.

Os padrões comportamentais preocupam-se com algoritmos e a atribuição de responsabilidades entre objetos, descrevendo padrões de comunicação entre objetos e classes. Padrões comportamentais de classes utilizam herança para distribuir o comportamento entre classes, enquanto que padrões comportamentais de objetos utilizam composição de objetos e, por isso, complementam e reforçam uns aos outros.

No CD-ROM anexo a esta dissertação encontram-se os 23 padrões de projeto de software propostos por Gamma et al. (1995) e utilizados neste trabalho, de forma resumida, apresentando sua Intenção, Aplicabilidade, Estrutura de Classes e Principais Conseqüências.

Mais detalhes sobre esses padrões podem ser encontrados no livro *Design Patterns – Elements of Reusable Object-Oriented Software* (Gamma et al., 1995).

Os padrões de projeto colaboram na elaboração de software orientado a objetos quando resolvem questões que surgem durante o desenvolvimento (Gamma et al., 1995) e, dentre elas, pode-se citar:

a) Identificação dos objetos corretos

Padrões de projeto ajudam a identificar abstrações menos óbvias bem como os objetos que podem representá-las. Por exemplo, objetos que representam um processo são de extrema importância no projeto, e não ocorrem na natureza. Para auxiliar nessa etapa recomenda-se a utilização dos seguintes padrões: *Strategy*, *State*, *Composite*.

b) Determinação da granularidade dos objetos

Padrões de projeto descrevem como organizar grandes quantidades de objetos de granularidade muito fina, auxiliando a decomposição de um objeto em outros menores, definindo objetos com a responsabilidade de criar outros e objetos que representam subsistemas completos, implementando uma tarefa em outro objeto ou grupo de objetos. Para auxiliar nessa etapa recomenda-se a utilização dos seguintes padrões: *Abstract Factory*, *Builder*, *Visitor*, *Command*, *Façade*, *Flyweight*.

c) Especificação da interface de objetos

Padrões de projeto ajudam a definir interfaces identificando seus elementos-chave, os tipos de dados enviados por elas, especificando quais elementos não devem ser incorporados à interface, estipulando como encapsular/salvar o estado de um objeto e restaurando-o para este estado posteriormente, especificando o relacionamento entre interfaces, exigindo que objetos tenham interfaces similares ou com restrições. Para auxiliar nessa etapa recomenda-se a utilização dos seguintes padrões: *Memento*, *Decorator*, *Proxy* e *Visitor*.

d) Herança de classes x herança de interface

A herança de classe define a implementação de um objeto em termos da implementação de outro objeto, compartilhando código e representação enquanto que a herança de interface descreve quando um objeto pode ser usado em lugar de outro. Muitos padrões de projeto dependem desta diferença: não compartilhando da mesma implementação, definindo uma implementação em comum, implementando classes abstratas

que são interfaces. Para auxiliar nessa etapa recomenda-se a utilização dos seguintes padrões: *Chain of Responsability*, *Composite*, *Command*, *Observer*, *State* e *Strategy*.

e) Programação para uma interface

Padrões de projeto auxiliam no momento de instanciação das classes concretas, com os padrões de criação. Eles abstraem o processo de criação inserindo maneiras diferentes de associar uma interface com sua implementação de forma transparente no momento da instanciação. Para auxiliar nessa etapa recomenda-se a utilização dos seguintes padrões: *Abstract Factory*, *Builder*, *Factory Method*, P*rototype* e *Singleton*..

f) Delegação

Delegação é uma maneira de tornar a composição de objetos tão poderosa para fins de reutilização quanto a herança. A principal vantagem da delegação é a facilidade em compor comportamentos em tempo de execução e mudar a forma como são compostos. A desvantagem é que o software dinâmico, altamente parametrizado, é difícil de ser compreendido. Alguns padrões de projeto dependem da delegação, mudando o comportamento de um objeto pela mudança dos objetos para os quais ele delega solicitações. Outros padrões o utilizam de maneira menos intensa, passando um objeto para outros em uma cadeia, etc. Para auxiliar nessa etapa recomenda-se a utilização dos seguintes padrões: *State*, *Strategy*, *Visitor*, *Mediator*, *Chain of Responsability*, *Bridge*.

g) Tempo de execução x tempo de compilação

Devido à disparidade entre as estruturas de um programa em tempo de execução e tempo de compilação, a estrutura de um sistema em tempo de execução deve ser imposta mais pelo projetista do que pela linguagem. Muitos padrões de projeto capturam a distinção entre estruturas de tempo de compilação e execução. Colaboram na construção de estruturas complexas em tempo de execução, envolvem estruturas de tempo de execução difíceis de serem entendidas, estabelecem padrões de comunicação que a herança não revela. Para auxiliar nessa etapa recomenda-se a utilização dos seguintes padrões: *Composite*, *Decorator*, *Observer* e *Chain of Responsability*.

h) Projeto para mudanças

A chave para aumentar a reutilização é antecipar novos requisitos, mudar os requisitos existentes e projetar sistemas de modo que eles possam evoluir, levando em

consideração a necessidade de mudar ao longo de sua vida. As mudanças não antecipadas são custosas. Os padrões de projeto ajudam a mudar segundo maneiras específicas.

A Tabela 2, elaborada a partir das considerações feitas pelos autores (Gamma et al., 1995), apresenta resumidamente os subsídios para que desenvolvedores de software possam reformular projetos utilizando padrões.

Tabela 2 – Padrões e reformulação de projeto

Motivo da reformulação	Descrição	Padrões
Criação de um objeto pela especificação explícita de uma classe	Especificar um nome de uma classe, quando se cria um objeto, compromete com uma implementação particular. Este compromisso pode dificultar futuras mudanças, pois objetos devem ser criados indiretamente.	*Abstract Factory* *Factory Method* *Prototype*
Dependência de operações específicas	Evitar solicitações codificadas torna mais fácil alterar a maneira como a solicitação é atendida.	*Chain of Responsability* *Command*
Dependência da plataforma de software e hardware	Deve-se projetar o software para limitar as dependências de plataformas.	*Abstract Factory* *Bridge*
Dependência de representações ou implementações de objetos	Ocultar a maneira como o objeto é representado, armazenado, localizado ou implementado de seus clientes, evitando a propagação de mudanças em cadeia.	*Abstract Factory* *Bridge* *Memento* *Proxy*
Dependências algorítmicas	Isolar algoritmos que são alterados ao longo do desenvolvimento e da reutilização.	*Builder* *Iterator* *Strategy* *Template Method* *Visitor*
Forte acoplamento	Evitar o forte acoplamento, pois classes fortemente acopladas são difíceis de serem reutilizadas isoladamente.	*Abstract Factory* *Bridge* *Chain of Responsability* *Command* *Façade* *Mediator* *Observer*
Estendendo a funcionalidade pelo reuso de subclasses	Utilizar a composição de objetos e a delegação para fornecer alternativas flexíveis à herança e à combinação de objetos.	*Bridge* *Chain of Responsability* *Composite* *Decorator* *Observer* *Strategy*

Motivo da reformulação	Descrição	Padrões
Incapacidade de alterar classes de modo conveniente	Alteração de uma classe que não pode ser convenientemente modificada, necessidade de código-fonte não disponível, etc.	*Adapter* *Decorator* *Visitor*

2.4 Manutenção de Software

Pressman (2005) diz que todo software sofre, inevitavelmente, modificações após ser entregue ao cliente final. Além disso, segundo Burge e Brown (2000) a manutenção de software tem sido uma das mais difíceis e custosas fases do ciclo de vida do software, especialmente em grandes sistemas. Isso porque quanto mais código envolvido, maiores as chances de haver interações não conhecidas, causando problemas quando atualizações e correções são feitas durante a manutenção.

Bennet e Xu (2003) afirmam que a manutenibilidade se refere à facilidade com a qual é possível manter e desenvolver software.

Manutenção, de acordo com Lientz e Swanson (1980) consiste em quatro tipos: corretiva, adaptativa, perfectiva e preventiva. A manutenção corretiva lida com o reparo de defeitos encontrados, que podem ser resultados de erros de projeto, de lógica ou codificação. Erros de projeto ocorrem quando, por exemplo, mudanças feitas no software são incorretas, incompletas, dificilmente comunicáveis ou quando o requisito a ser alterado não foi entendido. Erros de lógica resultam de testes e conclusões inválidos ou incompletos, implementação errada de especificações de projeto ou fluxo de dados com defeito. Erros de codificação são causados por implementação incorreta de detalhes de projeto e uso incorreto do código. Defeitos são causados por erros de processamento de dados e erros de desempenho do sistema. Todos esses erros fazem com que o software não esteja de acordo com a especificação.

A manutenção adaptativa realiza a adaptação do software de acordo com as modificações do ambiente, como alteração de hardware ou sistema operacional, mas também pode ser uma modificação das regras de negócio, políticas governamentais, padrões de trabalho. A necessidade de uma manutenção adaptativa só é reconhecida pelo monitoramento do ambiente.

A manutenção perfectiva lida com a acomodação a requisitos novos ou alterados pelo usuário. Aumenta interesses funcionais do sistema e atividades que melhoram seu

desempenho ou a interface com o usuário. Um pedaço do software tende a ser motivo para várias mudanças, resultando no aumento no número de requisitos, partindo da premissa de que o software se torna mais útil, os usuários tendem a se tornarem experientes em novas situações além das propostas pelo escopo inicial.

A manutenção preventiva concentra atividades que aumentam a manutenibilidade do sistema, como atualização de documentação, adição de comentários e melhoria da estrutura do sistema.

Há restrições ao termo manutenção de software. Alguns autores, como por exemplo Chapin e Cimitile (2001), consideram manutenção de software a corretiva, e como evolução de software as demais.

Pressman (2005) afirma que apenas 20% do trabalho de manutenção é relativo à manutenção corretiva, os outros 80% são de adaptação de sistemas ao ambiente externo, de melhorias solicitadas pelos usuários e de processos de reengenharia.

Basili (1995) apresenta uma abordagem incremental para melhorar a manutenção com enfoque na construção de modelos para as versões de manutenção do software. A abordagem apresenta bons resultados quando utilizada para construir um modelo de esforço em manutenções de software. Neste trabalho, Basili (1995) utiliza a divisão do processo de manutenção (Valett et al., 1994) em tarefas agrupadas de acordo com cinco categorias: análise/isolamento, projeto, implementação, teste e documentação.

As tarefas de análise e isolamento consistem na análise de diferentes alternativas de implementação, comparando seus efeitos quanto ao cronograma, custos e operação. O isolamento se refere ao tempo gasto tentando entender o problema.

As tarefas de projeto consistem em reprojetar o sistema baseando-se no entendimento das mudanças necessárias. Pode haver documentação semi-formal, como revisão de documentos de versões.

Nas tarefas de implementação estão embutidos a codificação e os testes de unidade, registrando os tempos gastos nessas tarefas. Pode haver documentação, como plano de testes de modificação do software. Os testes de unidade são realizados pelo mantenedor que realizou as mudanças.

As tarefas da fase de testes é composta por testes de integração, aceitação e regressão. Testes de integração verificam a integração entre os componentes do sistema. Testes de aceitação verificam se o sistema atende os requisitos do usuário final, sendo realizados por ele. Os de regressão garantem que outras partes do sistema não têm funcionalidade alterada após a manutenção.

Reúne-se a documentação sobre o sistema, que é escrita ou revisada, e a documentação do usuário, que consiste no guia do usuário ou outra documentação formal que não seja do sistema.

Segundo Stark et al. (1999), um dos fatores que deve ser analisado na atividade de manutenção é a alteração nos requisitos durante essa atividade, impactando na alteração dos custos, cronograma e qualidade do produto resultante. Por isso, define duas técnicas para se lidar com as mudanças em requisitos no ambiente de manutenção. A primeira é a análise exploratória de dados e ajuda a entender fontes, freqüência e os tipos das mudanças feitas. A segunda técnica afirma que um modelo de regressão ajuda na comunicação sobre custos e cronogramas afetados pelas mudanças nos requisitos.

Prechelt et al. (2000) realizam um experimento comparando a manutenção em sistemas projetados com padrões de projeto a sistemas com soluções simplificadas. Dessa experiência relatam três lições. A primeira afirma que não é sempre vantajoso utilizar padrões de projeto, quando há soluções mais simples. A segunda lição diz que se deve utilizar o bom senso de engenharia de software para encontrar exceções onde uma solução simples é desejada, mesmo se um padrão de projeto pode ser facilmente aplicado. A terceira lição sugere embora em engenharia de software se afirme que a utilização do padrão pode não ser uma boa solução, mesmo assim poder ser correto utilizá-lo. Caso haja dúvidas quanto à aplicação do padrão, deve-se aplicá-lo, a não ser que haja um motivo coerente para sua não aplicação. E a quarta e última lição afirma que o conhecimento sobre padrões quando se mantém um sistema que os utiliza colabora na tarefa, mesmo que o sistema seja extenso e complexo.

Dentre as técnicas de manutenção pode-se citar engenharia reversa, reengenharia e refatoração.

Engenharia reversa geralmente envolve a extração de artefatos de projetos a partir do código-fonte e a construção ou a sintetização em nível mais alto de abstração. Não é um processo de alteração do domínio do sistema ou a criação de um novo sistema baseado no domínio do antigo. É um processo de examinação, não um processo de mudança ou replicação (Chikofsky e Cross, 1990). Para Vlissides (1995) a engenharia reversa é uma atividade conveniente, pois geralmente é mais barato ao engenheiro reverter algo que já foi desenvolvido do que começar novamente, aproveitando a habilidade de engenheiros experientes e seus esforços de sucesso. Dessa forma, engenharia reversa é definida como o processo de analisar o domínio do sistema para: a) identificar os componentes e seus

relacionamentos; b) criar representações em outra forma ou em um nível de abstração maior (Chikofsky e Cross, 1990).

Após as pesquisas de Chikofsky e Cross, surgem diversas maneiras de explorar, manipular, analisar, componentizar, visualizar, referenciar e sumarizar artefatos de software. Isso inclui documentação de várias formas e representações imediatas de código, dados e arquitetura.

Deng e Kothari (2002) explicam que o software legado é muito complexo, tanto para entender seus algoritmos quanto para o entendimento de estratégias ou processos do programa. Afirmam ainda que para um desenvolvedor experiente em um domínio pode ser menos complicado o entendimento e a manutenção de um programa legado devido à sua capacidade de extrair os algoritmos e as estratégias desse sistema. Mas, de qualquer forma, essa tarefa é propensa a erros, os quais podem ser diminuídos com a utilização de ferramentas de suporte ao processo de engenharia reversa, específicas para um domínio. Shi e Olsson (2005) também apóiam o uso de ferramentas para análise de código e sua compreensão, recuperando a estrutura do programa e seus relacionamentos. Entretanto, concluem que sem documentação do sistema legado, ainda há grande esforço para o entendimento do código.

Reengenharia é um processo composto por engenharia reversa + Δ + mudança de linguagem de programação. Δ significa que pequenas modificações podem ser realizadas na funcionalidade do sistema legado. A mudança de linguagem envolve tanto as linguagens que pertencem ao mesmo paradigma do legado ou a outros paradigmas. Observa-se que quando a linguagem de programação no sistema alvo é do mesmo paradigma que a do legado, um modelo de projeto pode ser recuperado pelo processo de engenharia reversa e a nova implementação é feita a partir desse modelo. Já quando há mudança de paradigma de linguagem de programação, o modelo de projeto não deve ser obtido pelo processo de engenharia reversa, e sim um de análise, nível mais alto de abstração. A partir desse modelo, o de projeto pode ser construído para então ocorrer a implementação do sistema alvo utilizando essa nova linguagem de programação.

Bianchi e Caivano (2003) afirmam que o processo de reengenharia é intrusivo, pois requer que dados e procedimentos sejam reestruturados ao mesmo tempo. Dessa forma, pode ser necessário bloquear o sistema durante a execução do processo, até a sua conclusão. Todas as atividades de reengenharia devem ser executadas tanto no sistema legado quanto no que está sofrendo reengenharia, e há alto risco do novo sistema não ser equivalente ao legado no final do processo, solicitando manutenção corretiva e gerando um ciclo entre o processo

de manutenção e o de reengenharia. Por isso, eles propõem um processo de reengenharia iterativo e gradual, em alguns procedimentos de cada vez e, desta forma, poucas partes do sistema são bloqueadas a cada iteração.

Refatoração é o processo de mudar a estrutura interna de um sistema de software de forma a não alterar o comportamento externo do código, melhorando sua estrutura interna e tornando o sistema fácil de ser entendido e de manutenção menos dispendiosa. É um caminho disciplinado para limpar o código e minimizar as chances de introdução de erros (Fowler, 1999).

Cada passo do processo de refatoração é simples. Há movimentação de um atributo de uma classe para outra, remoção de parte do código de um método para dar origem a um novo método, inserção de algum código acima ou abaixo da hierarquia, etc. O efeito cumulativo dessas pequenas mudanças é que pode melhorar o projeto (Fowler, 1999).

Em seu livro, Fowler (1999) apresenta um guia destinado a programadores profissionais para a refatoração na linguagem Java (2005). Entretanto, a refatoração pode ser realizada em sistemas escritos em outras linguagens de programação.

A refatoração deve ser executada quando uma nova funcionalidade é inserida, quando um erro deve ser corrigido ou durante revisão de código, escrevendo-o de forma mais clara (Fowler, 1999).

Os benefícios alcançados com refatoração são (Fowler, 1999): melhoria do projeto do software; software mais fácil de ser entendido; auxilia a encontrar erros, torna o desenvolvimento mais rápido.

Uma vez que agilidade e correteza são características desejáveis em qualquer projeto de software, diversas ferramentas, em várias linguagens de programação, surgem para auxiliar o desenvolvedor na tarefa de refatoração. Na linguagem Java há a plataforma Eclipse, JFactor, Together-J, etc. Em C/C++ há Ref++, SlickEdit, entre outras. Além dessas ferramentas, há outras disponíveis para as linguagens de programação Visual Basic, Phyton, Delphi, .NET, etc., sendo que uma relação delas pode ser encontrada no site oficial de refatoração (Refactoring, 2005).

2.5 Métricas de Software Orientado a Objetos

Sobre a qualidade do software, Pressman (2005) diz que é uma mistura complexa de fatores que variam de acordo com a aplicação e com os clientes, sendo que há vários desses fatores de qualidade definidos.

Para avaliar a qualidade do produto software, Pressman (2005) afirma que há métricas que fornecem uma maneira quantitativa de avaliar a qualidade de atributos internos do produto, habilitando o engenheiro de software a avaliar a qualidade antes do produto ser construído. As métricas fornecem a visão aprofundada necessária para criar modelos efetivos de análise e projeto, código sólido, e testes rigorosos. Para que a métrica seja útil, deve ser independente da linguagem de programação e deve fornecer realimentação para o engenheiro de software.

Segundo o mesmo autor, há diversos tipos de métricas, de acordo com as fases no ciclo de vida do software. Métricas para o modelo de análise focalizam a funcionalidade, os dados e o comportamento. As métricas para projeto consideram aspectos da arquitetura, projeto em nível de componentes e projeto de interface. Métricas de projeto arquitetural consideram os aspectos estruturais do modelo de projeto. As métricas de projeto em nível de componentes fornecem indicação da qualidade do módulo. Métricas de projeto de interface fornecem indicação da adequação do leioute de uma GUI.

Para garantir a qualidade do software diversas métricas são criadas, como as de Chidamber e Kemerer (1994), para sistemas orientados a objetos. Os autores afirmam que o foco na melhoria do processo aumenta a demanda por medidas de software, ou métricas para gerenciamento do processo. Os autores propõem seis métricas, que são analiticamente avaliadas:

Métrica 1: Peso dos métodos por classe (*Weighted Methods Per Class*). O número de métodos e sua complexidade fornecem uma previsão do tempo de esforço requerido para desenvolver e manter a classe. Quanto maior o número de métodos de uma classe, maior é o impacto nas subclasses, desde que essas herdem todos os métodos definidos pela superclasse. Classes com grande número de métodos tendem a ser mais específicas, limitando a possibilidade de reuso.

Métrica 2: Profundidade da árvore de herança (*Depth of Inheritance Tree*). Quanto maior a profundidade de uma classe na hierarquia, maior o número de métodos a herdar, tornando-a mais complexa e seu comportamento mais imprevisível. Árvores profundas possuem maior complexidade de projeto, pois mais métodos e classes são envolvidos.

Aprofundando uma classe em particular na hierarquia, é maior o potencial de reuso de métodos herdados.

Métrica 3: Número de filhos (*Number of Children*). Quanto maior o número de filhos, maior é o reuso, pois herança é uma forma de reutilização. Quanto maior o número de filhos, maior é a probabilidade de uma abstração imprópria da superclasse. Se uma classe tem um grande número de subclasses, essas podem estar sendo mal utilizadas. O número de subclasses dá idéia do potencial de influência que a classe tem no projeto. Se a classe tem um grande número de filhos, pode ser necessário mais testes nos métodos desta classe.

Métrica 4: Acoplamento entre classes de objetos (*Coupling Between Object classes*). Duas classes são acopladas quando métodos declarados em uma classe usam métodos ou instâncias de variáveis definidas por outra classe. Quanto mais independente uma classe é, mais fácil é o reuso em outra aplicação. Para aumentar a modularidade e promover o encapsulamento, deve haver pouco acoplamento de classes. Quanto maior o número de acoplamentos, maior a sensibilidade à mudanças em outras partes do projeto, e a manutenção é dificultada. Uma medida de acoplamento é necessária para determinar qual é a complexidade necessária nos testes de várias partes do sistema. Quanto maior o acoplamento, mais rigorosos devem ser os testes.

Métrica 5: Resposta para uma classe (*Response For a Class*). Se um grande número de métodos pode ser invocado em resposta a uma solicitação, o teste de uma classe se torna mais complicado, pois requer um alto nível de entendimento do testador. Quanto maior o número de métodos que pode ser invocado de uma classe, maior a sua complexidade. O valor do pior caso para as possíveis respostas reside na alocação apropriada do tempo de testes.

Métrica 6: Falta de coesão nos métodos (*Lack of Cohesion Of Methods*). A coesão de métodos de uma classe é desejável, desde que promova o encapsulamento. A falta de coesão de classes implica que classes devem ser divididas em duas ou mais subclasses. Baixa coesão implica em complexidade e aumento na probabilidade de erros durante o desenvolvimento.

Misra e Bhavsar (2003) apresentam um estudo que investiga a usabilidade de métricas no nível de projeto/código orientado a objetos como indicadores de dificuldade em manter o software. Desta forma, podem ser feitas considerações sobre projeto/código em fases iniciais do desenvolvimento, reduzindo dificuldade e aumentando a qualidade. Afirmam ainda que em um projeto de software em particular, para aumentar a qualidade do

produto final, o desenvolvedor pode diminuir algumas métricas de projeto e aumentar outras. Os autores chegam às seguintes conclusões:

- Quanto maior o número de classes, o número de métodos, o tamanho da classe ou do método, maior o nível de dificuldade.
- O aumento no número de linhas de código aumenta significativamente a dificuldade.
- A dificuldade do software é menor se há muitos métodos e atributos escondidos. Esconder métodos diminui mais a dificuldade do software do que esconder atributos.
- O aumento de polimorfismo aumenta a dificuldade do programa, pois permite ligações em tempo de execução.
- Aumento na herança de atributos, de métodos e no nível de profundidade das árvores aumenta a dificuldade e diminui a qualidade.
- Um aumento no controle de densidade, respostas para classes, e peso dos métodos aumenta a dificuldade. Desses, o aumento das respostas para classes é o que mais aumenta a dificuldade.
- Um aumento na média da profundidade dos caminhos aumenta a dificuldade, pois é mais difícil entender o software.
- O aumento no acoplamento ou falta de coesão aumentam a dificuldade, pois acoplamento reduz encapsulamento e aumenta a complexidade do software. A coesão dos métodos aumenta o encapsulamento e a qualidade.

2.6 Padrões de Projeto x Refatoração x Métricas

Considerando que a refatoração em sistemas orientados a objetos tem crescido ultimamente devido à melhoraria da qualidade do software, ela juntamente com padrões de projeto de software tende a produzir maior qualidade nos sistemas.

Em 1996, Shull, Melo e Basili definem BACKDOOR, um método indutivo para arquitetura reversa de padrões de projeto em sistemas orientados a objetos. Uma base de conhecimento que descreve padrões utilizados pela organização é o artefato produzido quando esse método é utilizado.

Cinnéide (2000) propôs uma metodologia de transformações que introduz padrões de projeto em programas existentes na linguagem Java. Permite ao programador adiar seguramente a aplicação de padrões até que a flexibilidade provida por eles seja necessária.

Amiot et al. (2001) apresentam uma coleção de ferramentas e técnicas para ajudar projetistas de software a utilizar padrões. Essa coleção facilita: a escolha do padrão correto, de acordo com o contexto; a adaptação do padrão de acordo com os requisitos; a aplicação do padrão em uma determinada linguagem e a transformação de versões.

JIAD (*Java Based Intent Aspects Detector*) é uma ferramenta que automatiza a identificação de aspectos de intenção (*intent-aspects*) em código Java. Aspectos de intenção são uma coleção de elementos do programa ou estruturas que implicam a aplicabilidade de um padrão de projeto adequado através de interações dos elementos do programa. A ferramenta busca minimizar erros de inferência de padrões de projeto em sistemas existentes, possibilitando a refatoração (Rajesh e Janakiram, 2004).

Shi e Olsson (2005) propõem a automatização do processo de reconhecimento de padrões (de projeto e outros como concorrência), utilizando análise estrutural e semântica de código.

Jeon, Lee e Bae (2002) propõem uma abordagem de refatoração utilizando padrões de projeto em Java em que definem regras de inferência e estratégias de refatoração. Utiliza predicados em Prolog para representar o comportamento e estrutura do sistema.

Muraki e Saeki (2001) propõem medidas de projeto de software que auxiliam a determinar a aplicação dos padrões de projeto em processos de refatoração, tendo como principal foco aspectos específicos da estrutura do software. Definem as seguintes medidas (contemplando 10 dos 23 padrões de projeto) para guiar a refatoração: 11 métricas para declarações condicionais e duas famílias relacionadas às estruturas de herança. Os autores propuseram métricas, pois as propostas por Chidamber e Kemerer (1994) não são adequadas para medirem sistemas que apresentam padrões de projeto, pois consideram o número de métodos por classe, quantos níveis de herança existem, etc., sendo que padrões encorajam o uso de classes abstratas e delegação por herança.

Dos processos relacionados que envolvem refatoração com padrões de projeto, não há nenhum baseado exclusivamente no domínio de sistemas de informação, sendo que nesta pesquisa os sistemas estudados pertencem a este domínio. Neste trabalho não são utilizadas as medidas propostas por Muraki e Saeki, pois não são todos os padrões de projeto que podem ser medidos de acordo com suas definições.

2.7 Considerações Finais

Este capítulo apresentou alguns dos trabalhos técnicos e assuntos relevantes que contribuem para a realização desta dissertação. Os padrões de projeto constituem o núcleo da pesquisa, sendo essencial seu entendimento e conhecimento para posterior aplicação no processo de refatoração a ser apresentado no capítulo 3.

A manutenção de software é analisada para justificar a definição do processo, pois esse é definido com o objetivo de aumentar a manutenibilidade do sistema no qual é aplicado.

No processo proposto nesta dissertação há etapas de refatoração (na aplicação dos padrões de projeto no sistema) e de engenharia reversa (para recuperação de informações sobre o sistema). As métricas de Chidamber e Kemerer (1994) e algumas das medidas de dificuldade propostas por Misra e Bhavsar (2003) são utilizadas para que o engenheiro de software possa avaliar o esforço necessário para manter o software.

A refatoração foi adotada para que não haja alterações na funcionalidade do sistema durante o processo. As etapas de engenharia reversa são necessárias para o entendimento do sistema a ser refatorado e para que o mesmo seja documentado, aumentando sua manutenibilidade.

O capítulo seguinte apresenta o processo de refatoração utilizando padrões de projeto de software.

Capítulo 3

Processo de Refatoração com Padrões de Projeto

3.1 Considerações Iniciais

A necessidade de resolver problemas de projeto de software orientado a objetos já existente ou, simplesmente, de melhorar a sua arquitetura motiva a utilização de padrões de projeto reutilizando soluções já comprovadas. Dessa forma, a manutenção preventiva pode ser realizada com a refatoração do software utilizando padrões de projeto. Para que isso ocorra de forma organizada e auxilie o engenheiro de software nessa tarefa, a partir de estudos de caso foram elaboradas algumas diretrizes que podem ser aplicadas a sistemas que apresentem características semelhantes as dos utilizados neste estudo.

As diretrizes são compostas por três etapas executadas seqüencialmente:

1. Entender o Sistema;
2. Refatorar o Código-Fonte Utilizando Padrões de Projeto; e
3. Verificar Sistema Após Refatoração.

No restante deste trabalho o termo processo de refatoração é utilizado para referenciar essas diretrizes. Os termos código-fonte e código, padrões de projeto e padrões são utilizados como sinônimos, respectivamente.

Este capítulo define na seção 3.2 a abrangência e concepção do processo de refatoração. A seção 3.3 descreve a primeira etapa do processo, Entender o Sistema, na qual o engenheiro de software tem o primeiro contato com o software existente e identifica os possíveis padrões que podem ser aplicados no processo de refatoração. Na seção 3.4 encontra-se a segunda etapa do processo, Refatorar o Código-Fonte Utilizando Padrões de Projeto, na qual o padrão identificado na etapa anterior é utilizado para a refatoração. A seção 3.5 apresenta a terceira etapa do processo, Verificar Sistema Após Refatoração, na qual é verificado se o software refatorado corresponde exatamente ao sistema original. Na seção 3.6 estão colocadas as considerações finais.

3.2 Abrangência e Concepção do Processo de Refatoração

Os padrões de projeto definidos por Gamma et al. (1995) podem ser aplicados em sistemas de diversas naturezas (tempo real, informação, etc.). A estruturação do padrão de projeto nesses sistemas também pode sofrer várias modificações, de acordo com seu escopo, linguagem de programação e ainda, habilidade do desenvolvedor.

Desta forma, prevê-se que estimar formas de identificação de padrões de projeto em sistemas que já existem sem seu emprego é uma tarefa árdua. Por isso, este processo de refatoração utilizando padrões de projeto, é definido a partir da análise de sistemas existentes que pertencem ao domínio de sistemas de informação. A partir desses sistemas procurou-se indícios de cada um dos padrões de projeto propostos por Gamma et al. (1995) para daí resultar as diretrizes que podem ser aplicadas em outros sistemas. Ressalta-se que somente aplicações pertencentes ao domínio de sistemas de informação foram utilizadas, porém acredita-se que os indícios podem ser encontrados em sistemas pertencentes a outros domínios.

A literatura técnica dispõe de processos de refatoração utilizando padrões de projeto de software (Cinnéide, 2000; Amiot et al., 2001; Jeon, Lee e Bae, 2002; Kerievsky, 2004; Shi e Olsson, 2005), entretanto, o ganho com o desenvolvimento deste novo processo é a definição de uma lista de indícios para a identificação dos padrões no sistema existente e o detalhamento, de forma genérica, de diretrizes de refatoração utilizando cada padrão de projeto. Após a refatoração, a manutenção do sistema é facilitada, uma vez que o mesmo é documentado e sua estruturação é melhorada pelo uso dos padrões.

As diretrizes de refatoração com padrões de projeto foram elaboradas de forma prospectiva, isto é, a partir de estudos de caso realizados em sete sistemas desenvolvidos com a linguagem Java (2005), obtidos por meio da Internet: sistema de videolocadora, sistema de exibição de formas geométricas, editor de imagens, biblioteca, visualizador de árvore hierárquica, sistema de ordenação de números e letras e uma calculadora. O código-fonte desses sistemas, antes e após a refatoração, encontra-se no CD-ROM em anexo.

A Tabela 3 apresenta, nas linhas, os sistemas utilizados nos estudos de caso e, nas colunas, os padrões aplicados nesses sistemas. Por meio dessa tabela, evidencia-se que poucos padrões foram aplicados em mais de um sistema, isso se deve à natureza dos sistemas e/ou à aplicabilidade dos padrões, sendo que todos os padrões foram analisados quanto a sua presença em todos os sistemas.

Após a identificação da possibilidade de aplicação do padrão em um sistema, iniciava-se sua implementação, sendo esses procedimentos registrados, dando origem às diretrizes de refatoração. Uma vez implementado o padrão no sistema, esse era isolado e o original era avaliado quanto à presença de um novo padrão. Não houve, inicialmente, aplicação de vários padrões em um mesmo sistema, pois se acreditava que a definição dos procedimentos de refatoração poderia ser prejudicada. A aplicação de vários padrões em um sistema é apresentada no capítulo 4, estudo de caso.

Tabela 3 – Sistemas de estudos de caso e padrões implementados

Padrões / **Sistemas**	*Abstract Factory*	*Builder*	*Factory Method*	*Prototype*	*Singleton*	*Adapter*	*Bridge*	*Composite*	*Decorator*	*Façade*	*Flyweight*	*Proxy*	*Chain of Responsability*	*Command*	*Interpreter*	*Iterator*	*Mediator*	*Memento*	*Observer*	*State*	*Strategy*	*Template Method*	*Visitor*
Videoloca-dora		x		x	x	x			x			x		x		x						x	x
Formas geométricas	x		x				x						x						x	x	x		
Editor de imagens											x												
Biblioteca										x													
Árvore livros								x															
Ordenação																x	x	x					
Calculadora															x								
	criação					**estrutural**							**comportamental**										

A Figura 1 ilustra o processo de refatoração proposto sendo detalhadas as suas etapas e seus passos nas seções 3.3 a 3.5. Nesse processo, as setas numeradas indicam a seqüência de execução de cada passo. A seta que contém o número 5 indica que é possível reiniciar o processo a partir do passo 3, quando já houve uma ou mais refatorações no sistema, ou seja, o mesmo já é conhecido pelo engenheiro de software.

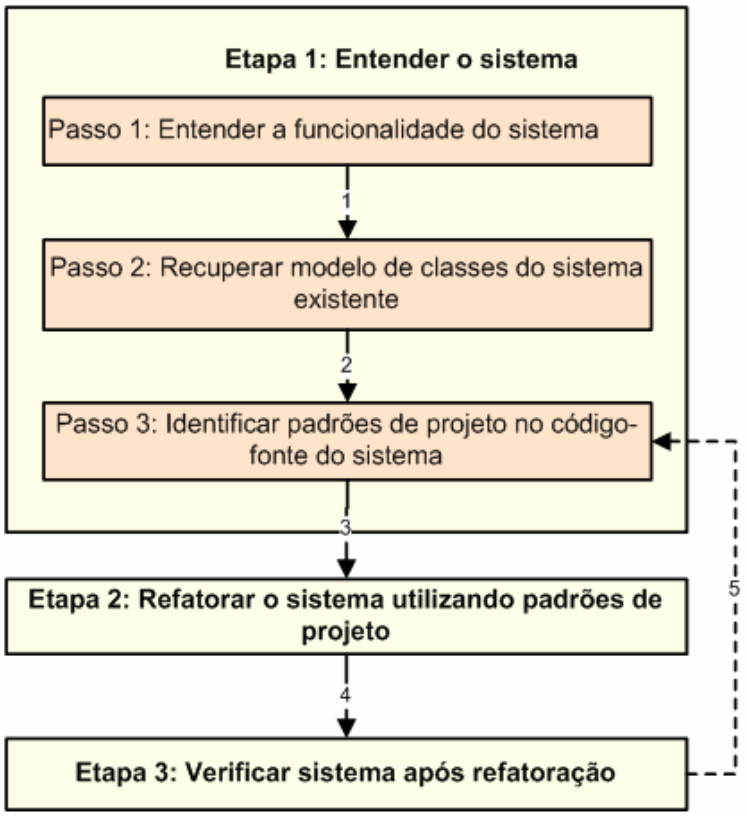

Figura 1 – Processo de refatoração com padrões de projeto

3.3 Etapa 1: Entender o sistema

Esta etapa possui três passos a serem executados seqüencialmente, com o objetivo principal de fornecer ao engenheiro de software mais informações a respeito do sistema, caso ainda não esteja familiarizado com o mesmo. O escopo do sistema e sua arquitetura geral são identificados nos passos 1 (entender a funcionalidade do sistema) e 2 (recuperar modelo de classes do sistema existente), possibilitando que o passo 3 (identificar padrões de projeto no código-fonte do sistema) seja executado com mais exatidão, pois define melhor os possíveis padrões a serem utilizados no processo de refatoração. Os passos desta etapa são apresentados no seguinte formato:

Identificação: define o passo da etapa.

Ação: representa a ação que deve ser executada no passo.

Objetivo: descreve o propósito e a justificativa da Ação.

Solução: descreve o que e como deve ser feito para que o Objetivo seja atingido.

A seguir são apresentados os três passos desta etapa.

Identificação: Passo 1.

Ação: Entender a funcionalidade do sistema.

Objetivo: Compreender a funcionalidade do sistema por meio de sua execução e armazenamento das entradas e saídas.

Solução:

a) Gerar uma tabela, como a mostrada na Tabela 4, contendo as seguintes informações:
   - Número da interação: números seqüenciais.
   - Interação do usuário com o sistema (Entrada): o sistema deve ser executado para que sejam identificadas todas as interações do usuário.
   - Resposta do sistema (Saída): listar as saídas fornecidas, geradas pela interação do usuário com o sistema, tanto as exibidas na tela do usuário quanto as apresentadas no console do programador. Listar os possíveis arquivos gerados e seu conteúdo. As informações que não forem aplicáveis ao sistema podem ser omitidas.

Tabela 4 – Leiaute dos dados a serem obtidos ao executar o sistema

Número da interação	Interação do usuário no sistema (Entrada)	Resposta do sistema (Saída)		
		Na tela do usuário	No console do programador	Arquivos gerados

Identificação: Passo 2.

Ação: Recuperar modelo de classes do sistema existente.

Objetivo: Recuperar/construir o modelo de classes do sistema atual a partir do código-fonte para reconhecimento de possíveis padrões existentes.

Solução:

a) Caso já exista o modelo de classes do sistema, deve-se avaliá-lo quanto à real representação das informações (classes, métodos, atributos e relacionamentos) nele contidas em relação as do código-fonte.

b) Caso não exista o modelo de classes, sua construção deve ser realizada. A partir do código-fonte existente, pode-se utilizar uma ferramenta computadorizada de software que recupere automaticamente o modelo, ou fazê-lo manualmente. Essa etapa é a de engenharia reversa e detalhes de como fazê-la podem ser encontrados em Cagnin (2005) e Pressman (2005), entre outros.

Identificação: Passo 3.

Ação: Identificar padrões de projeto no código-fonte do sistema.

Objetivo: A partir do entendimento da funcionalidade do sistema (passo 1) e do conhecimento de sua arquitetura (passo 2), identificar trechos de código compatíveis com os apresentados na Tabela 5, que representam indícios da existência do padrão.

Solução:

a) Se o desenvolvedor conhece a categoria do problema (estrutural, comportamental ou de criação), deve buscar a sua solução identificando os indícios da categoria apresentados na Tabela 5; ou

b) Se o desenvolvedor não conhece a categoria do problema, deve começar por qualquer uma delas ou percorrer seqüencialmente a Tabela 5, identificando os indícios para obter a solução desejada.

Tabela 5 – Indícios da aplicabilidade dos padrões no sistema existente

<table>
<tr><th>Padrão</th><th>Indícios</th></tr>
<tr><td colspan="2">Categoria: Criação</td></tr>
<tr><td>Abstract Factory</td><td rowspan="2">O código deve apresentar um ponto em que há a possibilidade de instanciação de uma ou mais classes. As possíveis classes a serem instanciadas devem ser subclasses da mesma superclasse, implementar a mesma interface ou serem dependentes umas das outras. Exemplo:
<pre>if (condicao1){
     ClasseA classeA = new ClasseA();
      ClasseB classeB = new ClasseB();
}else if (condicao2){
     ClasseC classeC = new ClasseC();
}
ClasseD classeD = new ClasseD();
ClasseE classeE = new ClasseE();</pre>Utilizar o padrão Abstract Factory quando for necessária a delegação para a instanciação de objeto(s) e o padrão Factory Method quando for necessária a herança para a instanciação de objeto(s).</td></tr>
<tr><td>Factory Method</td></tr>
<tr><td>Builder</td><td>O código deve criar vários objetos (partes) para a criação de um objeto mais complexo, sendo que o processo de criação de todos os objetos é o mesmo. Essas partes devem ser escolhidas de acordo com dados pré-existentes. Exemplo:
<pre>if (condicao1){
   button.setVisible(true);
   label.setText(“objeto 1”);
}else if (condicao2){
   jComboBox1.setVisible(true);
   jComboBox1.addItem(“obj1”);
   jComboBox1.addItem(“obj2”);
}</pre></td></tr>
</table>

<table>
<tr><th>Padrão</th><th>Indícios</th></tr>
<tr><td>Prototype</td><td>O código deve apresentar a criação de dois ou mais objetos do mesmo tipo e a cópia de valores de atributos de um objeto para os respectivos atributos de outro objeto.
Exemplo 1:
<pre>Objeto objetoA = new Objeto();
Objeto objetoB = new Objeto();
objetoB.setCodigo(objetoA.getCodigo());
objetoB.setNome(objetoA.getNome());
objetoB.setCampos(objetoA.getCampos());</pre>Exemplo 2:
<pre>JTextField jTextA = new JTextField();
JTextField jTextB = new JTextField();
jTextB.setText(jTextA.getText());</pre></td></tr>
<tr><td>Singleton</td><td>O código deve apresentar trechos de controle de um recurso, que deve ser utilizado por um cliente de cada vez. Exemplo:
<pre>ConexaoBancoDados conexao = new ConexaoBancoDados();
if(banco não está ativo){
   conexao.conectar();
   //declarações utilizando o banco de dados
   conexao.desconectar();
}else if(banco está ativo){
   Sysout.println(“Não foi possível conectar ao banco de
                                   dados: banco em atividade”);
}</pre></td></tr>
<tr><td colspan="2">Categoria: Estrutural</td></tr>
<tr><td>Adapter</td><td>O código deve apresentar classes clientes (por exemplo, classe <code>A</code>) que acessam outras (por exemplo, classe <code>B</code>), sendo que não é toda a funcionalidade necessária à classe <code>A</code> que é provida pelo método utilizado, da classe <code>B</code>. O código deve mostrar adaptações realizadas por <code>A</code> sobre os resultados fornecidos por <code>B</code>. Exemplo:
<pre>class A{//classe cliente
...
  private B b = new
B(params);
...
  forma = b.Desenhar(params);
  forma.colorir();
  forma.mostrarDesenho();
...
}</pre><pre>class B{
...
  public Desenho
Desenhar(params){
    //desenha
  }
...
}</pre></td></tr>
</table>

<table>
<tr><th>Padrão</th><th>Indícios</th></tr>
<tr><td>Bridge</td><td>O código deve apresentar o mesmo dado de várias formas. Entretanto, o mesmo modelo de dados deve ser utilizado para as distintas apresentações. Exemplo: a apresentação dos dados de uma tabela pode ser feita mostrando-se os dados na própria tabela, em uma lista, em um gráfico, etc.
<pre>//produtos é um vetor com os dados a serem apresentados
JList lista = new JList();
JTable tabela = new Tabela();
for(int i = 0; i < produtos.size(); i++){
   lista.add(produtos.get(i));//adiciona produtos na lista
   //adiciona produtos na tabela
}
//exibe lista
//exibe tabela</pre></td></tr>
<tr><td>Composite</td><td>O código deve apresentar hierarquização de algum objeto, de modo que possa ser mapeado (ou já esteja) em uma estrutura de árvore. O tratamento para objetos da mesma hierarquia deve ser igual. No código podem ser encontrados objetos das bibliotecas <code>DefaultMutableTreeNode</code>, <code>Vector</code> e/ou <code>Enumeration</code>.</td></tr>
<tr><td>Decorator</td><td>O código deve apresentar agregação de funcionalidade(s) a um ou mais objetos, em tempo de execução. Exemplo:
<pre>if(cliente é homem)
   cliente.AdicionaClienteSorteio();</pre></td></tr>
<tr><td>Façade</td><td>O código deve apresentar várias classes clientes que acessam os mesmos recursos, ou seja, métodos de outras classes. Esses recursos podem estar espalhados pelo sistema e são os candidatos a comporem a interface Façade, sendo que há a necessidade de simplificar as interfaces entre as classes.</td></tr>
<tr><td>Flyweight</td><td>O código deve apresentar a instanciação de muitos objetos da mesma classe, que sejam diferenciados apenas por poucos parâmetros. Exemplo:
<pre>private <<tipo>> obj1 = new <<tipo>><<params>>;
private <<tipo>> obj2 = new <<tipo>><<params>>;
private <<tipo>> obj3 = new <<tipo>><<params>>;
private <<tipo>> obj4 = new <<tipo>><<params>>;
...
private <<tipo>> objN = new <<tipo>><<params>>;

obj1.setAtt1(valor);   obj2.setAtt1(valor);
obj3.setAtt1(valor);   obj4.setAtt1(valor);
objN.setAtt1(valor);

obj1.setAtt3(valor1); obj2.setAtt3(valor1);
obj3.setAtt3(valor1); obj4.setAtt3(valor1);
objN.setAtt3(valor1);
...
obj1.setAtt2(valorA); obj2.setAtt2(valorB);
obj3.setAtt2(valorC); obj4.setAtt2(valorD);
objN.setAtt2(valorN);</pre></td></tr>
<tr><td>Proxy</td><td>O código deve apresentar alguma ação sendo executada enquanto o objeto que já existe (virtual proxy) é utilizado/apresentado.
Exemplo 1:
<pre>while(banco banco de dados está sendo inicializado)
   System.out.print(“Aguarde, iniciando banco de dados”);</pre></td></tr>
</table>

<table>
<tr><th>Padrão</th><th>Indícios</th></tr>
<tr><td></td><td>Exemplo 2:
<pre>if(ícone foi criado)
   //desenha ícone na tela
else {
   //apresenta uma mensagem contendo o status do
   //carregamento da imagem
}</pre></td></tr>
<tr><td colspan="2">Categoria: Comportamental</td></tr>
<tr><td>Chain of Responsability</td><td>O código deve permitir que várias classes tratem de uma requisição sem que uma tenha conhecimento da capacidade de tratamento da outra e sem que o cliente saiba qual classe deve tratá-la. A requisição é transferida entre as classes, até que uma possa atendê-la. Pode ou não haver prioridade entre as classes na transferência de uma requisição, de modo que, para que uma classe possa tratá-la, seja necessário que outra a trate antes. Exemplo:
<pre>if(condicao1){ //sem prioridade de tratamento
   classeA.Metodo1();
}else if(condicao2){ //com prioridade de tratamento
   classeB.Metodo2();
   classeC.Metodo3();
   classeA.Metodo1();
}</pre></td></tr>
<tr><td>Command</td><td>O código deve possuir ações a serem executadas quando houver interação do usuário com o sistema.
<pre>class A implements ActionListener{
...
 obj1.addActionListener(this);
 obj2.addActionListener(this);
 ...
 public void actionPerformed(ActionEvent event){
   if(event.getSource() == obj1)
      Metodo1();
   else if(event.getSource() == obj2)
      Metodo2();
 }
}</pre></td></tr>
<tr><td>Interpreter</td><td>O código deve permitir que suas operações sejam representadas como uma linguagem. Exemplo: analisador de expressões aritméticas.</td></tr>
<tr><td>Iterator</td><td>O código deve apresentar um objeto do tipo lista ou coleção que deve ser percorrido, de modo a identificar os elementos nele armazenados. Exemplo:
<pre>Vector v = new Vector();
...
for(int i = 0; i < v.size(); i++)
   Sysout.println(v.get(i));</pre></td></tr>
<tr><td>Mediator</td><td>O código deve apresentar interações complexas entre os componentes visuais do sistema, sendo que cada componente necessita da informação sobre um ou vários outros componentes, há forte acoplamento. Exemplo:
<pre>public <<tipo_ret>> Metodo1(<<params>>){
   componente1.setVisible(false);
   componente2.setEnabled(true); }
public <<tipo_ret>> Metodo2(<<params >>){
   componente1.setVisible(true);</pre></td></tr>
</table>

Padrão	Indícios

	componente1.setText("texto texto");
	componente2.setEnable(false);
	componente3.setBackground(cor);
	}

Memento	O código deve apresentar o armazenamento do estado interno de um objeto e recuperação deste estado posteriormente. Exemplo: implementação de funções de desfazer/refazer.
	//cliente é um objeto do tipo Cliente
	//vetor é um objeto do tipo Vector
	public void armazenarCliente(){
	vetor.add(cliente.getCodigo());
	vetor.add(cliente.getNome());
	vetor.add(cliente.getRG());
	...
	}
	public void recuperarCliente(){
	cliente.setCodigo(vetor.get(i++));
	cliente.setNome(vetor.get(i++))
	cliente.setRG(vetor.get(i++));
	...
	}

Observer	O código deve apresentar dados de várias maneiras ao mesmo tempo, sendo que quando os dados de uma representação mudam os demais devem ser notificados da mudança. Exemplo:
	...
	ObjetoA objA = new ObjetoA();
	ObjetoB objB = new ObjetoB();
	ObjetoC objC = new ObjetoC();
	...
	if(objA foi alterado OU objB foi alterado OU objC foi alterado){
	alterarApresentacaoObjA();
	alterarApresentacaoObjB();
	alterarApresentacaoObjC();
	}

State	O código deve comportar-se de maneira diferente de acordo com o estado do objeto. Deve haver declarações `if/else` ou `switch` extensas, que definem o comportamento a ser adotado. Exemplo:
	if(componente.getColor() igual COR_A)
	componente.setColor(corB);
	else if(componente.getColor() igual COR_B)
	componente.setColor(corC);
	else if(componente.getColor() igual COR_C)
	componente.setColor(corD);

Strategy	O código deve requisitar uma funcionalidade particular, que pode ser fornecida por várias classes, com o mesmo nome e implementações distintas. Deve haver declarações `if/else` ou `switch`, definindo qual funcionalidade deve ser invocada. Exemplo:
	if(condicao1) objA.Metodo();
	else if(condicao2) objB.Metodo();
	else objC.Metodo();

Template Method	O código deve apresentar uma classe com a possibilidade de um ou mais de seus métodos serem definidos em subclasses. Exemplo:

Padrão	Indícios

	`public void login(){`
	`  private JTextField usuario = new JTextField();`
	`  private JPasswordField senha = new JPasswordField();`
	`  private JButton btOk = new JButton("OK");`
	`  //inserção dos componentes na tela`
	
	`private void acaoBtOk(){`
	`  if(usuario.equals(“antonio”))`
	`  //inicializa sistema`
	`  else`
	`  System.out.print(“Acesso negado”);`
	`}`
	Outra implementação possível para o método `acaoBtOK`:
	`private void acaoBtOk(){`
	`  Conexao conexao = new Conexao();`
	`  if(conexao.VerificarUsuario(usuario.getText()))`
	`  //inicializa sistema`
	`  else`
	`  System.out.print(“Acesso negado”);`
	`}`
	Neste exemplo, o método `acaoBtOk` pode ser definido em uma subclasse.

Visitor	O código deve realizar uma ou mais operações em objetos (de interfaces distintas) ou ainda, em um grande número de objetos. Exemplo: geração do objeto relatório, que deve recolher determinados dados de todos os objetos relacionados a ele.
	`//objetos é o vetor que contém todos os objetos`
	`//a serem considerados na operação`
	`for(int i = 0; i < objetos.size(); i++)`
	`  soma = soma + objetos.getTotalGasto();`

3.4 Etapa 2: Refatorar o código-fonte utilizando padrões de projeto

Esta etapa tem duas atividades a serem executadas simultaneamente, cujos artefatos são o software refatorado com um padrão de projeto e o modelo de classes atualizado. A atualização do modelo de classes com o padrão é realizada ao mesmo tempo em que o software é implementado com esse padrão, dessa forma não há passos distintos, mas uma etapa que consiste de duas atividades relacionadas.

A apresentação do processo de refatoração é feita segundo as categorias dos padrões: de criação, estrutural e comportamental, propostas por Gamma et al. (1995). O formato utilizado é o que segue:

Identificação: número seqüencial.

Nome: nome do padrão.

Objetivo: descreve o problema particular de projeto que o padrão resolve.

Solução: apresenta os passos que devem ser seguidos para que o Objetivo seja atingido.

A nomenclatura utilizada para os nomes de classes e métodos, no processo de refatoração, é a seguinte:

- **_cp:** quando os métodos e/ou as classes são criados pelo processo de refatoração com o padrão. Exemplos: ClasseA_cp, MetodoA_cp.
- **_ap:** quando os métodos e/ou as classes são alterados pelo processo de refatoração com o padrão. Exemplo: ClasseA é renomeada para ClasseA_ap, MetodoA é renomeado para MetodoA_ap.
- Os métodos de classes criadas pelo processo de refatoração não têm nenhuma extensão acrescida ao seu nome.

Nas subseções seguintes é exibido o processo de refatoração para um padrão de cada uma das categorias do catálogo de padrões de Gamma et al. (1995). As refatorações para os demais padrões do catálogo de Gamma et al. (1995) encontram-se no Apêndice A.

Nestes processos de refatoração, alguns dos trechos de código representam o conteúdo da classe a ser inserida/alterada no sistema; outros trechos representam apenas um gabarito de como essa classe deve ser implementada e, nesse caso, deve-se alterar nomes de classes, métodos e seus tipos.

No processo de refatoração, quando um nome de padrão estiver grafado em itálico, refere-se ao padrão propriamente dito (não a uma classe ou a um método).

3.4.1 Refatoração com o padrão *Singleton* (Criação)

Problema: Garantir que uma classe tenha somente uma instância e fornecer um ponto global de acesso para a mesma.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

a) Criar a classe `Singleton_cp`, que representa o `Singleton`. Utilizar o código apresentado no Quadro 1.

Quadro 1 – Classe `Singleton_cp`

```
1 public class Singleton_cp {
2     static boolean instance_flag = false;
3     public Singleton(){
4           if (instance_flag)
5                 throw new RuntimeException();
6           else
7                 instance_flag = true;
      }
8     public void finalize(){
9           instance_flag = false;
      }
}
```

b) Retomar a classe que contém o trecho de código identificado com o padrão *Singleton* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo).

c) Na classe `X_ap`, retirar o código que faz o controle do recurso e substituí-lo pelo código apresentado no Quadro 2.

   i. Na linha 2 é declarado e instanciado o objeto do tipo *Singleton*.

   ii. Após a utilização do recurso, o mesmo é liberado para novo uso, linha 3.

Quadro 2 – Classe `X_ap`, padrão *Singleton*

```
1 public class X_ap{
...
2   private Singleton singleton = new Singleton();
...
   //utilização do recurso
3   singleton.finalize();//liberação do recurso
...
}
```

3.4.2 Refatoração com o padrão *Decorator* (Estrutural)

Objetivo: Dinamicamente, agregar responsabilidades adicionais a um objeto. Os *Decorators* fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidade.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

a) Identificar a classe de aplicação que tem funcionalidade agregada, denominada `X`. O Quadro 3 apresenta um gabarito essa classe `X`.

Quadro 3 – Classe `X`, padrão *Decorator*

```
1 public class X{
2   public <<tipo_ret>> Metodo1(<<params>>){...}
3   public <<tipo_ret>> Metodo2(<<params>>){...}
4   public <<tipo_ret>> Metodo3(<<params>>){...}
5   public <<tipo_ret>> Metodo4(<<params>>){...}
...
6   public <<tipo_ret>> MetodoN(<<params>>){...}
}
```

b) Verificar se existe uma interface (denominada `IntX`) referente a essa classe, caso não haja, deve-se criá-la. O Quadro 4 apresenta um gabarito para essa interface.
    i. Não é necessário definir todos os métodos presentes na classe `X`. Basta definir alguns métodos, os que forem essenciais à classe.

Quadro 4 – Interface `IntX`

```
1 public interface IntX{
2   public <<tipo_ret>> Metodo1(<<params>>);
3   public <<tipo_ret>> Metodo2(<<params>>);
4   public <<tipo_ret>> Metodo4(<<params>>);
}
```

c) Alterar a definição da classe `X`, de modo que ela implemente a interface criada no passo **b**. Renomear essa classe, inserindo a extensão _ap (`X_ap`).

d) Criar a classe abstrata `Decorator_cp`. Utilizar o gabarito apresentado no Quadro 5.
    i. Essa classe define a funcionalidade que pode ser adicionada nos objetos da classe `X_ap` (linha 5).
    ii. Deve implementar a interface `IntX` (linha 1): os métodos definidos pela interface `IntX` passam a ser abstratos (linhas 2, 3 e 4) e ainda não serão definidos, isso será feito nas subclasses de `Decorator_cp`.

Quadro 5 – Classe abstrata `Decorator_cp`

```
1 public abstract class Decorator_cp implements IntX{
   //métodos definidos pela interface
2   public abstract <<tipo_ret>> Metodo1(<<params>>);
3   public abstract <<tipo_ret>> Metodo2(<<params>>);
4   public abstract <<tipo_ret>> Metodo4(<<params>>);
    //método do padrão Decorator
5   public abstract <<tipo_ret>> MetodoAdicional(<<params>>);
}
```

e) Identificar a classe (`Y`, por exemplo) com o código que fornece a funcionalidade adicional a ser alocada para o padrão *Decorator* e renomeá-la, adicionando ao seu final a extensão _ap (classe `Y_ap`, por exemplo).

f) Criar a classe `ConcreteDecorator_cp`. Utilizar o gabarito apresentado no Quadro 6.

i. É uma subclasse de `Decorator_cp` (linha 1).
ii. Define todos os métodos abstratos herdados de `Decorator_cp` (linhas 2, 3 e 4).
iii. Inserir, no método `MetodoAdicional` (linha 5), o código identificado no passo **e**, e excluir esse código da classe `Y_ap`.

Quadro 6 – Classe `ConcreteDecorator_cp`

```
1 public class ConcreteDecorator_cp extends Decorator_cp{
2   public <<tipo_ret>> Metodo1(<<params>>){...}
3   public <<tipo_ret>> Metodo2(<<params>>){...}
4   public <<tipo_ret>> Metodo4(<<params>>){...}

5   public <<tipo_ret>> MetodoAdicional(<<params>>){
       //código contendo a funcionalidade adicional
    }
}
```

g) Criar, para cada funcionalidade adicional que houver, uma nova classe do tipo `ConcreteDecorator_cp`, e repetir os passos **e** e **f** para definir cada uma delas.
h) Reestruturar a classe `Y_ap`. Utilizar o gabarito apresentado no Quadro 7.
   i. Declarar e instanciar um objeto do tipo `ConcreteDecorator_cp` (linha 2).
   ii. Declarar e instanciar um objeto do tipo `X_ap` (linha 3).
   iii. Inserir, quando houver a necessidade de adicionar a funcionalidade no respectivo objeto, código semelhante ao apresentado na linha 4.

Quadro 7 – Classe `Y_ap`, padrão *Decorator*

```
1 public class Y_ap{
...
2   private ConcreteDecorator_cp c = new ConcreteDecorator_cp(<<params>>);
3   private X_ap x = new X_ap(<<params>>);
...
4   x.MetodoAdicional(<<params>>);
...
}
```

3.4.3 Refatoração com o padrão *Visitor* (Comportamental)

Objetivo: Representar uma operação a ser executada nos elementos de uma estrutura de objetos. *Visitor* permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.
   a) Identificar a classe do objeto que deve ser visitado. Pode haver mais de uma classe a ser visitada.
   b) Renomear a classe identificada no passo **a** (`X`, por exemplo), adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo).
   c) Inserir o método *accept* nas classes a serem visitadas. No Quadro 8 e no Quadro 9 há o código referente a esse método, inserido em duas classes a serem visitadas, `X_ap` e `Z_ap`.

Quadro 8 – Classe `X_ap`, padrão *Visitor*

```
1 public class X_ap{
...
2   public void accept(Visitor_cp v) {
3      v.visit(this);
   }
...
}
```

Quadro 9 – Classe `Z_ap`, padrão *Visitor*

```
1 public class Z_ap{
...
2   public void accept(Visitor_cp v) {
3      v.visit(this);
   }
...
}
```

   d) Criar a classe abstrata `Visitor_cp`. O Quadro 10 apresenta o gabarito para essa classe.
      i. Deve declarar métodos `visit` para todas as classes nas quais o método `accept` foi inserido, no passo **c**.

Quadro 10 – Classe `Visitor_cp`

```
1 public abstract class Visitor_cp {
2    public abstract void visit(X_ap visX);
3    public abstract void visit(Z_ap visZ);
}
```

   e) Identificar no código a função que o *Visitor* deve realizar sobre os objetos visitados. Renomear essa classe (`Y`, por exemplo), adicionando ao seu final a extensão `_ap` (classe `Y_ap`, por exemplo). O Quadro 11 apresenta o gabarito para essa classe.

Quadro 11 – Classe `Y_ap`, padrão *Visitor*

```
1 public class Y_ap{
2   private X_ap x = new X_ap(<<params>>);
3   private Z_ap z = new Z_ap(<<params>>);
...
4   while(há objetos a serem visitados){
5      //função envolvendo os objetos das classes X_ap e Z_ap
6      var += x.getVar();
7      var += z.getVar();
   }
8   System.out.println(var);
...
}
```

f) Criar a classe `XZVisitor_cp`. O Quadro 12 apresenta o gabarito para essa classe.

   i. Deve ser subclasse de `Visitor_cp` (linha 1) e deve implementar um método `visit` para cada um dos definidos na superclasse (linhas de 4 a 7).

   ii. Contém a funcionalidade do *Visitor*, a ser extraída da classe `Y_ap` (linhas 5, 7 e 8).

Quadro 12 – Classe `XZVisitor_cp`, padrão *Visitor*

```
1 public class XZVisitor_cp extends Visitor_cp{
2   private <<tipo>> var;
3   public XZVisitor_cp(){
      //inicializa variáveis
   }
4   public void visit(X_ap visitedX){
5      var += visitedX.getVar();
   }
6   public void visit(Z_ap visitedZ){
7      var += visitedZ.getVar();
   }
8   public <<tipo>> getVar(){
9      return var;   }
}
```

g) Reestruturar a classe `Y_ap`. O Quadro 13 apresenta o gabarito para essa classe.

   i. Declarar um objeto do tipo `XZVisitor_cp` (linha 4).

   ii. Retirar o código identificado no passo **e**, linhas de 5 a 8.

   iii. O acesso aos objetos `X_ap` é realizado como mostrado nas linhas de 9 a 11.

   iv. O acesso à funcionalidade do *Visitor* é realizado como mostrado na linha 12.

Quadro 13 – Classe `Y_ap`, padrão *Visitor*

```
1 public class Y_ap{
2   private X_ap x = new X_ap(<<params>>);
3   private Z_ap z = new Z_ap(<<params>>);
4   private XVisitor_cp xzvisitor = new XZVisitor_cp(<<params>>);
...
5   ~~while(há objetos a serem visitados)~~
6      ~~var += x.getVar();~~
7      ~~var += z.getVar();~~
```

```
8   System.out.println(var);
...
    //vetorX é um vetor que contém os objetos X_cp a serem visitados
    //vetorZ é um vetor que contém os objetos Z_cp a serem visitados
9   for (int i = 0; i < v.size(); i++){
10     vetorX.accept(xzvisitor);
11     vetorZ.accept(xzvisitor);      }
12  System.out.println(xzvisitor.getVar());
}
```

3.5 Etapa 3: Verificar sistema após refatoração

Esta etapa é composta de um passo cujo objetivo principal é verificar a funcionalidade do sistema após a realização do processo de refatoração e garantir que não há alteração em seu comportamento (a menos que a alteração seja planejada e desejada). O único passo desta etapa é apresentado no seguinte formato:

Identificação: define o passo da etapa.

Ação: representa a ação que deve ser executada no passo.

Objetivo: descreve o propósito e a justificativa da Ação.

Solução: descreve o que e como deve ser feito para que o Objetivo seja atingido.

A seguir é apresentado o único passo desta etapa.

Identificação: Passo 1.

Ação: Verificar a funcionalidade do sistema.

Objetivo: Reavaliar o sistema após a refatoração quanto ao seu comportamento, a fim de detectar que o processo não altera a sua funcionalidade.

Solução:

Utilizar as mesmas interações realizadas anteriormente no passo 1 da etapa 1 (entender a funcionalidade do sistema) para analisar se as saídas após a refatoração permanecem inalteradas. Caso haja alterações no comportamento do sistema, há indício de que o padrão não utilizado adequadamente. O engenheiro de software deve retornar ao início do processo.

3.6 Considerações Finais

Este capítulo apresentou o processo de refatoração com padrões de projeto, que fornece a organização e reestruturação de um sistema tornando-o mais manutenível.

O processo definido nesta dissertação tem três etapas distintas e seqüenciais, no entanto, durante a etapa 2 pode ser necessário iterações para a elaboração da documentação do sistema enquanto ocorre a refatoração do mesmo. Nem sempre o desenvolvedor de software constrói primeiro completamente os modelos de classes de projeto para depois realizar a implementação. Dessa forma, os princípios de desenvolvimento ágil (Ambler e Jeffries, 2002) podem ser utilizados.

Os indícios para aplicação de padrões obtidos a partir de estudos com sistemas do domínio de sistemas de informação podem não ser suficientes ao imediato reconhecimento desses padrões pela sua identificação. Nesse caso, é necessário que o engenheiro de software tenha alguma experiência em reconhecer a aplicação do padrão no sistema utilizando, posteriormente, o processo de refatoração com o padrão. Durante a elaboração dessa lista com indícios da existência de padrões de projeto, preocupou-se em enfatizar o que o sistema deve apresentar, com apenas exemplificações de trechos de código, o que não significa que o sistema deva, obrigatoriamente, ter exclusivamente os trechos de código apresentados. Assim, é mostrada apenas uma indicação de como o código deve se comportar e ser estruturado.

O processo de refatoração definido nesta dissertação de forma prospectiva aplica um padrão a cada iteração, ou seja, quando é iniciado o processo e identificado o padrão, a refatoração acontece para esse padrão isoladamente.

A restrição quanto à aplicação das diretrizes é que somente foram utilizados sistemas de informação como estudo de caso. Dessa forma, não há garantia de que seja aplicável em outros domínios, caso haja interesse, porém infere-se que o engenheiro de software, tendo conhecimento da aplicabilidade dos padrões e com as diretrizes apresentadas, possa realizar a manutenção preventiva em outros tipos de sistema.

Outro ponto que merece destaque é que há aumento do número de linhas de código quando os padrões de projeto são introduzidos no código-fonte. Em contrapartida, há melhor organização e estruturação do sistema, o que possibilita a rápida identificação de qual funcionalidade é tratada em cada parte deste sistema.

O capítulo seguinte apresenta um estudo de caso em que vários padrões são utilizados cumulativamente, a fim de mostrar a utilidade do processo de refatoração descrito neste capítulo.

Capítulo 4
Estudo de Caso

4.1 Considerações Iniciais

Após elaborar o processo de refatoração com padrões de projeto definido no capítulo 3 desta dissertação, um estudo de caso foi realizado para a aplicação do processo definido neste trabalho de maneira cumulativa em uma mesma aplicação. Para isso, o sistema de videolocadora foi o escolhido por apresentar diversos padrões, que anteriormente foram aplicados isoladamente.

Esse sistema de videolocadora foi obtido por meio da Internet e não havia documentação além do código-fonte. O sistema original tem 11 classes, sendo uma para banco de dados e as demais misturam tratamento de interface com o usuário e acesso ao banco. O total de linhas de código é de 1883. O banco de dados utilizado é o *MS-Access*, cuja conexão é feita via JDBC (*Java DataBase Connectivity*) utilizando ODBC (*Open Database Connectivity*), com o *driver* embutido no *MS-Access*.

Dez padrões de projeto são aplicados no sistema: *Template Method*, *Decorator*, *Prototype*, *Command*, *Proxy*, *Visitor*, *Iterator*, *Singleton*, *Builder* e *Adapter*. Pode-se notar que há padrões pertencentes às três categorias.

A forma utilizada para apresentação do estudo de caso é por meio de três padrões, um de cada categoria, e não dos dez para evitar que a dissertação ficasse muito extensa. No CD-ROM anexo à dissertação encontram-se os dez processos de refatoração realizados nesse sistema.

Os padrões *Singleton*, *Decorator* e *Visitor* foram os escolhidos para serem apresentados neste estudo de caso. Embora seja criado um modelo de classes a cada padrão utilizado, aqui será mostrado um referente ao sistema existente (antes das refatorações) e um do sistema após as dez refatorações, ou seja, apresentando as estruturas criadas com a

aplicação dos padrões, já que essa é a documentação que deve fazer parte da gestão de configuração de software.

Para facilitar o acompanhamento da execução do processo, as etapas do processo de refatoração são apresentadas resumidamente na Tabela 6.

Tabela 6 – Resumo do processo de refatoração com padrões de projeto

Etapa	Passos
Etapa 1: Entender o sistema	Passo 1: Entender a funcionalidade do sistema
	Passo 2: Gerar modelo de classes do sistema atual
	Passo 3: Identificar padrões de projeto no código-fonte do sistema
Etapa 2: Refatorar o código-fonte utilizando padrões de projeto	Refatorar o sistema com o padrão identificado e completar o modelo de classes elaborado no passo 2 da etapa 1.
Etapa 3: Verificar sistema após refatoração	Passo 1: Verificar a funcionalidade do sistema

Como são aplicados vários padrões de projeto no sistema de videolocadora, os passos 1 (entender a funcionalidade do sistema) e 2 (recuperar modelo de classes do sistema existente) da etapa 1 são executados uma só vez. Isso porque nas demais iterações para identificação de outros padrões, o sistema já é conhecido e a documentação já está atualizada.

Este capítulo está organizado da seguinte forma: na seção 4.2 o detalhamento dos padrões aplicados no sistema de videolocadora é apresentado; na seção 4.3 é realizada a primeira refatoração do sistema, com o padrão *Singleton* e a execução das etapas 1, 2 e 3. A seção 4.4 mostra a segunda refatoração, realizada com o padrão *Decorator*, por meio do passo 3 da etapa 1 e as etapas 2 e 3. A terceira refatoração, com o padrão *Visitor*, é apresentada na seção 4.5, inclui o passo 3 da etapa 1 e as etapas 2 e 3. A seção 4.6 analisa as métricas aplicadas no sistema utilizado para estudo de caso e na seção 4.7 são comentadas as considerações finais.

4.2 Detalhamento da Aplicação do Processo

Os dez padrões que foram aplicados ao sistema de videolocadora utilizando o processo de refatoração proposto são relacionados a seguir com breve comentário de sua aplicação no sistema.

1. *Template Method*: manipula as classes responsáveis pela funcionalidade de *login* no sistema.
2. *Decorator*: manipula as classes responsáveis pela funcionalidade de promoção, quando há o aluguel de um filme.
3. *Prototype*: manipula as classes responsáveis pela funcionalidade de cópia dos campos de endereço, no cadastro de um cliente.
4. *Command*: utilizado em todas as classes que possuem ações a serem executadas quando um componente visual é utilizado (ações de botões, por exemplo).
5. *Proxy*: manipula as classes responsáveis pela funcionalidade de exibição das imagens dos filmes, na mostra de filmes.
6. *Visitor*: manipula as classes responsáveis pela funcionalidade de exibição de relatórios do sistema.
7. *Iterator*: está presente nas classes que apresentam estruturas como vetores ou tabelas *hash*, a serem percorridos.
8. *Singleton*: está presente em todas as classes que tratam de interface como usuário, garantindo que quando uma tela é exibida, nenhuma seja mostrada ao mesmo tempo.
9. *Builder*: manipula as classes responsáveis pela funcionalidade de diferenciar entre a escolha de um cadastro de uma pessoa física e um de pessoa jurídica, no cadastro de cliente.
10. *Adapter*: manipula as classes responsáveis pela funcionalidade de exibição de um cadastro simples e um cadastro completo de cliente.

Além desses padrões aplicados infere-se que há possibilidade de aplicação de padrões se as seguintes situações estivessem inseridas na funcionalidade do sistema originalmente existente:

1. Caso os filmes pudessem ser visualizados em uma árvore hierárquica, de acordo com suas categorias, o padrão *Composite* poderia ser utilizado para compôr essa hierarquia.
2. Caso houvesse botões de atalho para acessar a mesma funcionalidade disponível no menu, o padrão *Flyweight* poderia ser utilizado inserindo esses ícones de atalho.

3. Caso houvesse opções de desfazer/refazer ações nos cadastros, o padrão *Memento* poderia ser aplicado armazenando o estado do objeto e resgatando esse estado posteriormente.

4.3 Primeira Refatoração: Padrão Singleton

Conforme comentado anteriormente, somente para a primeira refatoração, os passos 1 e 2 da etapa 1 são realizados, isso porque a refatoração com diversos padrões é realizada seqüencialmente e nenhuma alteração de funcionalidade é inserida no sistema de videolocadora entre as refatorações.

4.3.1 Etapa 1: Entender o sistema

Passo 1: Entender a funcionalidade do sistema

As funções existentes no sistema de videolocadora são reconhecidas por meio de sua execução, em que a tela representada pela Figura 2 é exibida para *login* e, em seguida, a tela representada pela Figura 3 é apresentada, contendo os menus de opções disponíveis ao usuário.

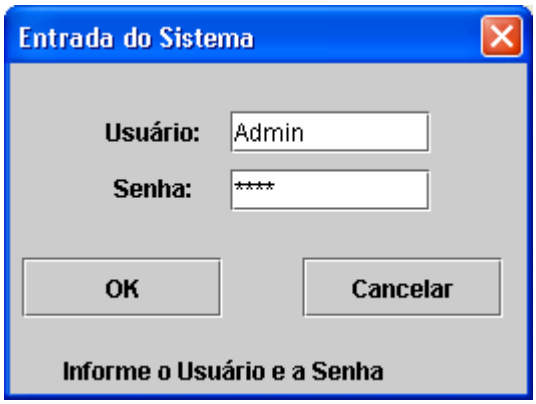

Figura 2 – Tela de *login* do sistema de videolocadora

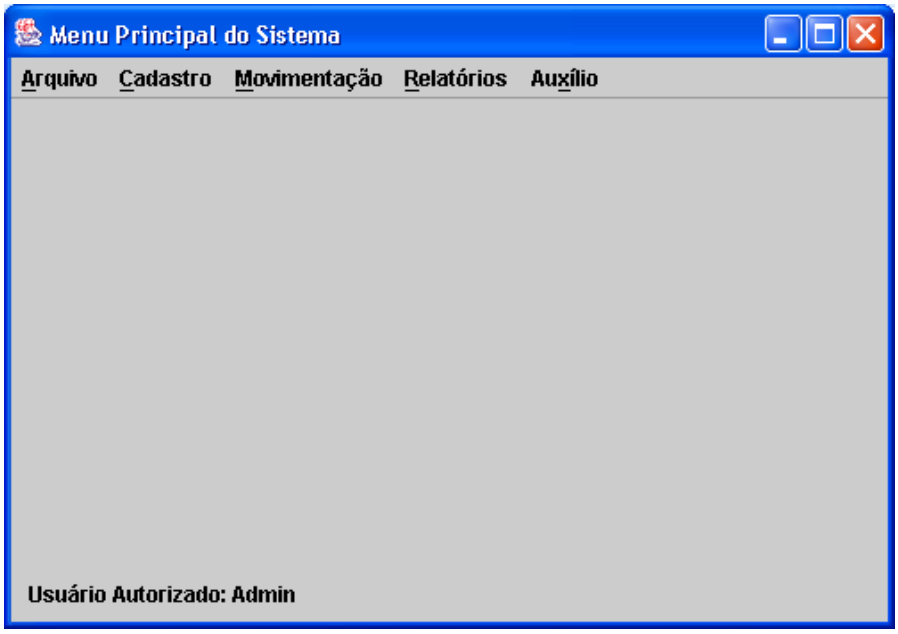

Figura 3 – Menu de opções do sistema de videolocadora

Como prevê o processo de refatoração definido nesta dissertação, as interações do usuário são registradas na Tabela 7 e serão novamente consideradas na execução da etapa 3. Na Tabela 7, a fonte `courier new em itálico` é usada para os nomes de campos e botões, que têm a primeira letra em maiúsculo, como no sistema. As informações inseridas durante as interações estão entre aspas. Essa tabela apresenta divisões, iniciadas pela palavra Operação, que são as opções que o usuário tem após seu *login* no sistema. A tabela com todas as interações existentes no sistema encontra-se no CD-ROM em anexo. Neste capítulo são mostradas algumas das interações existentes. A coluna "arquivos gerados" não é apresentada, pois nenhum arquivo é gerado por esse sistema.

Tabela 7 – Interações do usuário com o sistema de videolocadora

Nº Int.	Interação do usuário no sistema (Entrada)	Resposta do sistema (Saída)	
		Na tela do usuário	No console do programador
Operações de acesso ao sistema			
1	Executar o sistema.	A tela de *login* é apresentada contendo: campos `usuário`, `senha`; botões `Ok` e `Cancelar` e a mensagem "Informe o usuário e a senha".	Nenhuma mensagem é exibida.
2	Na tela de *login*, inserir o nome "Admin" e o usuário "root". Pressionar o botão `Ok`.	O menu principal do sistema é exibido com as opções: `Arquivo-Sair`; `Cadastro-Cliente`; `Cadastro-Filme`; `Movimentação-Aluguel`; `Movimentação-`	Nenhuma mensagem é exibida.

Nº Int.	Interação do usuário no sistema (Entrada)	Resposta do sistema (Saída)	
		Na tela do usuário	**No console do programador**
		Devolução; *Movimentação-Mostra de Filme*; *Relatórios-Locações*; *Auxílio-Sobre*.	
Operações do menu Arquivo			
3	Selecionar o menu *Arquivo* e, em seguida, *Sair*.	O sistema é finalizado.	Nenhuma mensagem é exibida.
Operações do menu Cadastro-Filme			
4	Selecionar o menu *Cadastro* e, em seguida, *Filme*.	É exibida uma tela de cadastro de filmes contendo campos para inserção de *Código*, *Nome* e *Loc. Imagem* (localização da imagem). Os botões *Gravar*, *Excluir* e *Limpar* são exibidos nessa tela.	Nenhuma mensagem é exibida.
...			
Operações do menu Cadastro-Cliente			
14	Inserir os valores "40", "João", "11111" e "01/01/2000" nos respectivos campos exibidos na ação 12. Pressionar o botão *Gravar*.	Quando o valor da matrícula é inserida, os botões *Gravar* e *Excluir* são habilitados. Quando o botão *Gravar* é pressionado, todos os campos são escondidos, retornando à tela exibida na ação 12.	Nenhuma mensagem é exibida.
...			

Passo 2: Recuperar modelo de classes do sistema existente

De acordo com as diretrizes de engenharia reversa apresentadas no passo 2 da etapa 1 do processo (seção 3.3), o modelo de classes foi gerado automaticamente pela ferramenta de software Omondo (2005), executado como um *plugin* na plataforma de desenvolvimento Eclipse (2005). A Figura 4 mostra o modelo de classes equivalente ao sistema de videolocadora, em que somente os métodos associados às classes são exibidos.

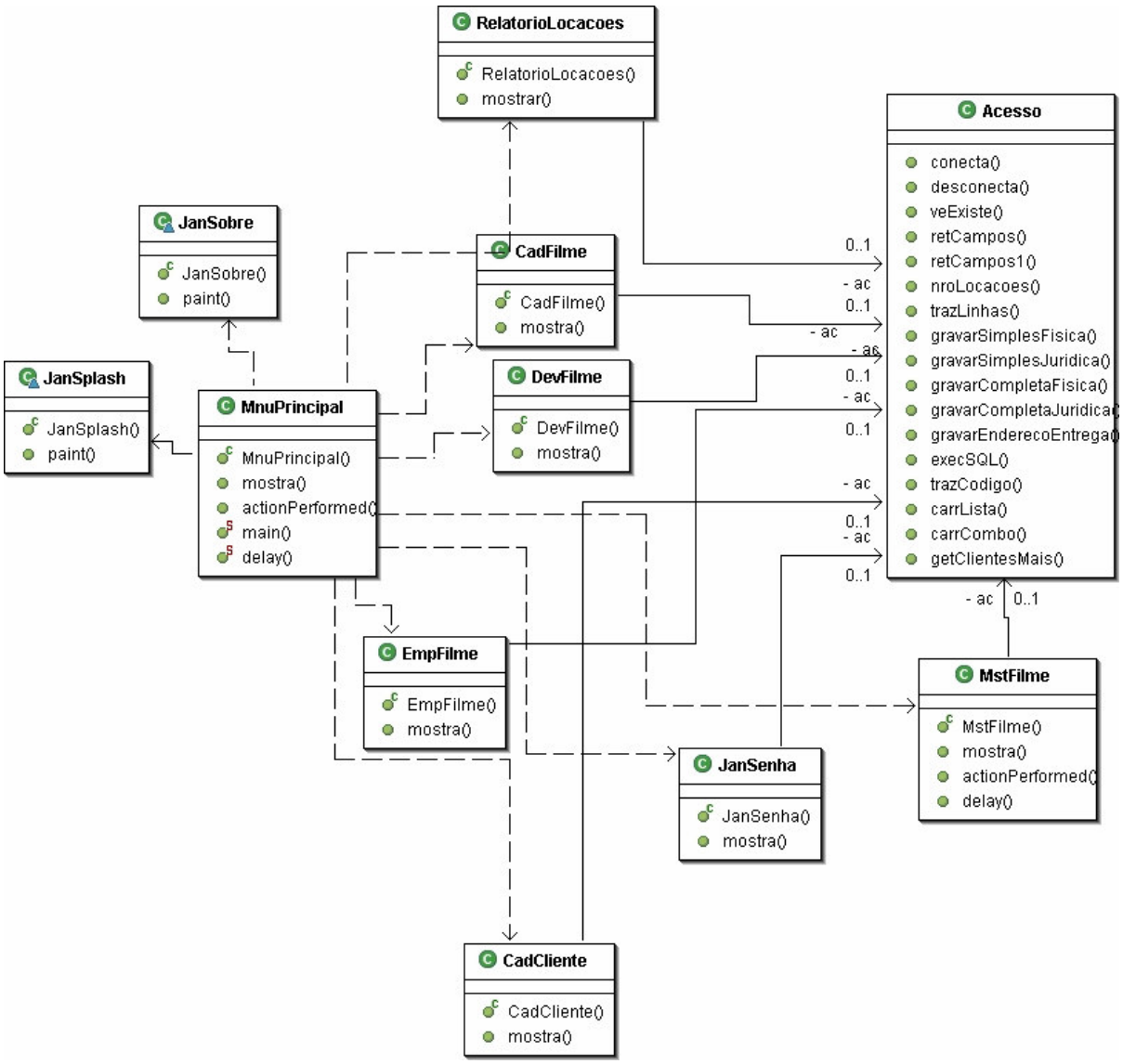

Figura 4 – Modelo de classes do sistema de videolocadora antes da refatoração

Passo 3: Identificar padrões de projeto no código-fonte do sistema

A solução proposta para esse passo da refatoração prevê duas alternativas, como apresentado na seção 3.3.

Neste estudo de caso, a segunda opção é executada, pois se deseja melhorar a estrutura do sistema para facilitar manutenções futuras e, por isso, há o interesse em aplicar padrões de projeto de software.

A partir do conhecimento da funcionalidade do sistema por meio de sua execução, percorre-se a lista de indícios (Tabela 5) e o código-fonte, verificando a possibilidade de aplicação de padrões de projeto no sistema de videolocadora.

O primeiro padrão de projeto identificado no sistema é o *Singleton*, aplicado para que somente uma tela seja exibida ao usuário a cada vez. O recurso a ser controlado pelo padrão são as opções disponíveis no menu do sistema. Para a identificação desse padrão não houve a necessidade de visualizar o código, bastando o entendimento do funcionamento do sistema, do padrão e dos dados apresentados na lista de indícios.

4.3.2 Etapa 2: Refatorar o sistema utilizando padrões de projeto

Uma vez identificado o padrão a ser aplicado, no passo 3 da etapa 1, deve-se refatorar o sistema utilizando esse padrão. Identifica-se no código-fonte as classes envolvidas no controle do recurso único, ou seja, no controle de exibição das telas. Esse código é mostrado no Quadro 14 e está presente em todas as classes que tratam da interface com o usuário sendo identificadas genericamente como `X`.

Quadro 14 – Código identificado com o padrão *Singleton*

```
class X extends JDialog{
...
   this.setModal(true);
...
```

Como definido anteriormente, a refatoração do modelo de classes será apresentada apenas ao final de todas as refatorações, não sendo exibida neste processo de refatoração com o padrão *Singleton*.

A seguir é apresentada a refatoração com o padrão *Singleton*, sendo que os itens **a**, **b** e **c**. referem-se aos passos de refatoração definidos para esse padrão, no capítulo 3.

a) A classe `Singleton_cp` é criada exatamente conforme definido no passo **a** da refatoração.
b) Todas as classes que tratam da interface com o usuário e contêm o código apresentado no Quadro 14 são alteradas, recebendo a extensão _ap.
c) Em todas as classes alteradas no passo **b** o código que faz o controle do recurso (apresentado no Quadro 14) é retirado e substituído pelo indicado pelo processo de refatoração. Uma das classes alteradas é apresentada no Quadro 15, em que as linhas que correspondem ao padrão estão em negrito.

Quadro 15 – Classe alterada com o padrão *Singleton*

```
public class MstFilme_ap extends JFrame implements ActionListener{
      //declaração de componentes visuais da tela
      private Graphics g;
      private Proxy_cp proxy;
      private Subject_cp subject;
      private DefaultListModel boxList = new DefaultListModel();
      private JList jboxList = new JList(boxList);
      private Acesso ac = new Acesso();
      private Singleton_cp singleton;
      private MstFilmeBD_cp mstFilme = new MstFilmeBD_cp();

      public MstFilme_ap() {
            singleton = new Singleton_cp();
            try {
                  mostra();
            } catch (Exception ex) {
                  ex.printStackTrace();
            }
      }
      public void mostra() throws Exception {
            this.getContentPane().setLayout(null);
            this.setTitle("Mostra de Filmes");
            janela = this.getContentPane();
            this.setSize(500, 380);
            //instanciação de componentes visuais da tela
            carregaLista();
            this.show();
      }
      public void actionPerformed(ActionEvent e){ ... }
      private void carregaLista() { ... }
      public void delay(int tempo) { ... }
      private void carregaImg_ap() { ... }
      ...
      private void aoFechar(WindowEvent e) {
        singleton.finalize();
        dispose();
      }
}
```

4.3.3 Etapa 3: Verificar sistema após refatoração

Uma vez realizada a refatoração com o padrão deve-se verificar que a funcionalidade do sistema não foi alterada, executando-o de acordo com as interações exibidas na Tabela 7.

4.4 Segunda Refatoração: Padrão Decorator

Esta segunda refatoração é realizada a partir do passo 3 (Identificar padrões de projeto no código-fonte do sistema) da etapa 1.

4.4.1 Etapa 1: Entender o sistema

Passo 3: Identificar padrões de projeto no código-fonte do sistema

A adição de funcionalidade prevista pelo padrão *Decorator* é evidenciada durante a ação de empréstimo de um filme. Caso o cliente tenha emprestado dez ou mais filmes, uma mensagem é exibida ao usuário, evidenciando que o cliente ganhou uma locação gratuita.

O trecho de código que corresponde a esse comportamento é apresentado no Quadro 16. O sistema de videolocadora não possui classe de aplicação. Dessa forma não existe, por exemplo, a classe `Filme` e sim uma classe para cadastro de filme que faz acesso ao banco de dados e tratamento da interface com o usuário. Assim, a funcionalidade é obtida das consultas realizadas diretamente no banco de dados.

Quadro 16 – Classe `EmpFilme`, identificada com o padrão *Decorator*

```
class EmpFilme extends JFrame {
...
  private void acaoBtEmprestar() {
      Acesso ac = new Acesso();
      if (ac.conecta()) {
        ac.execSQL("Insert into Locacao (cod_cliente, cod_filme) " +
            "values ('" + edCodCliente.getText() + "', '" +
                                        edCodFilme.getText() +  "')");

        ac.execSQL("Update Filme set sit_filme = 'A' " +
            " where cod_filme ='" + edCodFilme.getText() + "'");

        int nro_loca = ac.nroLocacoes(edCodCliente.getText());
        if(nro_loca >= 10){
              JOptionPane.showMessageDialog (this, lbNomCliente.getText()+
                                          "ganhou uma locação grátis!",
                                  "PROMOÇÃO!", JOptionPane.PLAIN_MESSAGE);

              ac.execSQL("Update Cliente set nro_locacoes = '" + 0 +
                          "' where cod_cliente ='" + edCodCliente.getText()
                                        + "'");
       }else{
         ac.execSQL("Update Cliente set nro_locacoes = '" + (nro_loca+1) +
                  "' where cod_cliente ='" + edCodCliente.getText() +
                                                                  "'");
       }
...
}
```

4.4.2 Etapa 2: Refatorar o sistema utilizando padrões de projeto

A seguir é detalhada a refatoração com o padrão *Decorator*, sendo que os itens de **a** até **f** referem-se aos passos de refatoração definidos para esse padrão, no capítulo 3.

a) Como o sistema de videolocadora não possui classes intermediárias entre as de interface e o banco de dados, foi criada a classe de aplicação equivalente ao objeto cliente, denominada `Cliente_cp`, Quadro 17. Essa classe possui os atributos inerentes ao objeto cliente, os métodos `get` e `set` equivalentes, além dos métodos `gravar`, `excluir` e `recuperarCampos`.

b) É definida a interface equivalente à classe `Cliente_cp`, criada no passo **a**, denominada `IntCliente_cp`.

c) A classe `Cliente_cp` torna-se implementadora da interface `IntCliente_cp`.

Quadro 17 – Classe `Cliente_cp`

```
public class Cliente_cp implements IntCliente_cp {
      private String nome;
      private String codigo;
      private String tel;
      private String dataCad;
      private int nro_locacoes;
      //métodos set e get
      public void gravar(String s){...}
      public void excluir(){...}
      public void recuperarCampos(){...}
}
```

d) A classe `Decorator_cp` criada é apresentada no Quadro 18.

Quadro 18 – Classe `Decorator_cp`

```
public abstract class Decorator_cp implements IntCliente_cp{
      public abstract void verificaPromocao(EmpFilme_ap e, String
cliente);
      public abstract void excluir();
      public abstract void recuperarCampos();
      public abstract void gravar(String tipo);
}
```

e) A classe `EmpFilme` é renomeada para `EmpFilme_ap`.

f) A classe `Decorator_cp` criada é apresentada no Quadro 19, sendo que o trecho de código referente ao método `verificaPromocao` foi extraído da classe `EmpFilme_ap` e está em negrito.

Quadro 19 – Classe `ConcreteDecorator_cp`

```
public class ConcreteDecorator_cp extends Decorator_cp{
      public ConcreteDecorator_cp() {}
      public  void excluir(){}
      public  void recuperarCampos(){}
      public  void gravar(String tipo){}
      public String resgataCliente(String codigo_cli){
            String nome_cli = null;
            Acesso ac = new Acesso();
            if (ac.conecta())
                  nome_cli = ac.nomeCliente(codigo_cli);
```

```
            ac.desconecta();
            return nome_cli;
      }
      public void verificaPromocao(EmpFilme_ap ef, String codigo_cli){
            Acesso ac = new Acesso();
            if (ac.conecta()) {
                  String nome_cli = resgataCliente(codigo_cli);
                  int nro_loca = ac.nroLocacoes(codigo_cli);
                  if (nro_loca >= 10) {
                        JOptionPane.showMessageDialog(
                              ef, nome_cli + " ganhou uma locação grátis!",
                              "PROMOÇÃO!", JOptionPane.PLAIN_MESSAGE);
                        ac.execSQL("Update Cliente set nro_locacoes = '"
                                    + 0 + "' where cod_cliente ='"
                                    + codigo_cli + "'");
                  } else {
                        ac.execSQL("Update Cliente set nro_locacoes = '"
                              + (nro_loca + 1)+ "' where cod_cliente ='"
                              + codigo_cli + "'");
                  }
                  ac.desconecta();
            }
      }
}
```

4.4.3 Etapa 3: Verificar sistema após refatoração

Uma vez realizada a refatoração com o padrão deve-se verificar se sua funcionalidade não foi alterada executando o sistema de acordo com as interações da Tabela 7. A aplicação do padrão *Decorator* envolve apenas uma função do sistema, entretanto, todas as interações identificadas na Tabela 7 são executadas para que se garanta que nenhuma outra parte do sistema tenha sido alterada.

4.5 Terceira Refatoração: Padrão Visitor

Esta terceira refatoração é realizada a partir do passo 3 (identificar padrões de projeto no código-fonte do sistema) da etapa 1.

4.5.1 Etapa 1: Entender o sistema

Passo 3: Identificar padrões de projeto no código-fonte do sistema

O padrão *Visitor* é identificado na função de geração de relatório, em que há a consulta em todos os registros de clientes para se identificar quais locaram dez ou mais filmes. O relatório exibe as informações dos clientes que estão nessa condição e o número total deles.

O trecho de código que corresponde a esse comportamento é apresentado no Quadro 20. A classe `RelatorioLocacoes` tem acesso ao método `getClientesMais`, da classe `Acesso` que, por sua vez, retorna os clientes com dez ou mais locações.

Quadro 20 – Classe `RelatorioLocacoes`, identificada com o padrão *Visitor*

```
class RelatorioLocacoes{
...
            if (ac.conecta()) {
                  cliente = ac.getClientesMais();
                  ac.desconecta();
             }
             while (i < cliente.size()) {
                  clienteNovo.add("Nome: " + cliente.get(i++));
                  clienteNovo.add("Código: " + cliente.get(i++));
                  clienteNovo.add("Data Cadastro: " + cliente.get(i++));
                  clienteNovo.add("Endereço: " + cliente.get(i++));
                  clienteNovo.add("Número Locações: " + cliente.get(i++));
                  clienteNovo.add("********************");
             }
             for (int j = 0; j < cliente.size(); j++)
                  jClientes.setListData(clienteNovo);
             JLabel3.setText(new Integer(cliente.size() / 5).toString());
             this.setVisible(true);
      }
...
}
```

4.5.2 Etapa 2: Refatorar o sistema utilizando padrões de projeto

A seguir é apresentada a refatoração com o padrão *Visitor*, sendo que os itens de **a** até **g** referem-se aos passos de refatoração definidos para esse padrão, no capítulo 3.

a) A classe que deve ser visitada, pela diretriz de refatoração apresentada, foi criada na refatoração com o padrão *Decorator*, e é `Cliente_cp`. O trecho inserido não existe no sistema original, pois não há uma classe de aplicação entre as classes `RelatorioLocacoes` e `Acesso`. Essa classe de aplicação, necessária, foi criada

quando houve a refatoração com o padrão *Decorator*; é a classe `Cliente_cp`. Para o padrão *Visitor*, é adicionado apenas o método `getClientesMais`.

b) A classe `Cliente_cp` não precisa ser renomeada, pois por meio de sua extensão, _cp, sabe-se que foi criada devido à refatoração com a utilização de um padrão de projeto.

c) O método `accept` é inserido na classe `Cliente_cp`, conforme mostrado no Quadro 21.

Quadro 21 – Classe `Cliente_cp`

```
public class Cliente_cp{
...
   public void accept(Visitor_cp v) { v.visit(this); }
...
}
```

d) A classe `Visitor_cp` é criada conforme apresentado no Quadro 22.

Quadro 22 – Classe `Visitor_cp`

```
public abstract class Visitor_cp {
    public abstract void visit(Cliente_cp cliente);
}
```

e) O trecho de código que manipula os objetos `Cliente_cp` é o apresentado no Quadro 20. A classe `RelatorioLocacoes` é renomeada para `RelatorioLocacoes_ap`.

f) A subclasse de `Visitor_cp` criada é a apresentada no Quadro 23.

Quadro 23 – Classe `Relatorio1Visitor_cp`

```
public class Relatorio1Visitor_cp extends Visitor_cp{
    private Vector clientesMais = new Vector();
    private int totalClientesMais = 0;
    public void visit(ClienteBD_cp cliente){
        if((int)cliente.getNro_locacoes() > 10){
            clientesMais.add("Nome: "+cliente.getNome());
            clientesMais.add("Código: "+cliente.getCodigo());
            clientesMais.add("Data Cadastro: "+cliente.getDataCad());
            clientesMais.add("Número Locações: " + new
                                Integer(cliente.getNro_locacoes()));
            clientesMais.add("********************");
            totalClientesMais++;
        }
    }
    public Vector getClientesMais(){ return clientesMais; }
    public int getTotalClientesMais(){ return totalClientesMais; }
}
```

g) A classe `RelatorioLocacoes` é reestruturada, conforme apresentado no Quadro 24. As linhas apresentadas em negrito representam as alterações solicitadas pela aplicação do padrão.

Quadro 24 – Classe `RelatorioLocacoes_ap`, com o padrão *Visitor*

```
public class RelatorioLocacoes_ap extends JFrame {
      //declaração dos componentes visuais da tela de relatório
      private Relatorio1Visitor_cp rl = new Relatorio1Visitor_cp();
      private ClienteBD_cp cli;
      private Enumeration e;
      private Singleton_cp singleton;
      public RelatorioLocacoes_ap() {
            singleton = new Singleton_cp();
            mostrar_ap();
      }
      public void mostrar_ap() {
            this.getContentPane().setLayout(null);
            this.setTitle("Relatórios");
            this.setSize(450, 500);
            //inserção dos componentes visuais da tela de relatório
            cli = new ClienteBDSimples_cp();
            e = cli.getClientesMais().elements();
            while(e.hasMoreElements()){
              cli.setNome((String)e.nextElement());
              cli.setCodigo((String) e.nextElement());
              cli.setDataCad((String) e.nextElement());
              cli.setNro_locacoes(((Integer)e.nextElement()).intValue());
              cli.accept(rl);
            }
            jClientes.setListData(rl.getClientesMais());
            JLabel3.setText(new Integer(rl.getTotalClientesMais()).toString());
            this.setVisible(true);
            this.show();
      }
      private void acaoBtFechar(ActionEvent e) {
            singleton.finalize();
            dispose();
      }
}
```

4.5.3 Etapa 3: Verificar sistema após refatoração

O sistema é executado de acordo com as interações apresentadas na Tabela 7 e verifica-se que a funcionalidade do sistema e o relatório exibido não têm alterações.

A Figura 3 apresenta o modelo de classes do sistema videolocadora após a refatoração com os dez padrões de projeto. Os métodos pertencentes às classes são omitidos, para melhor visualização do modelo.

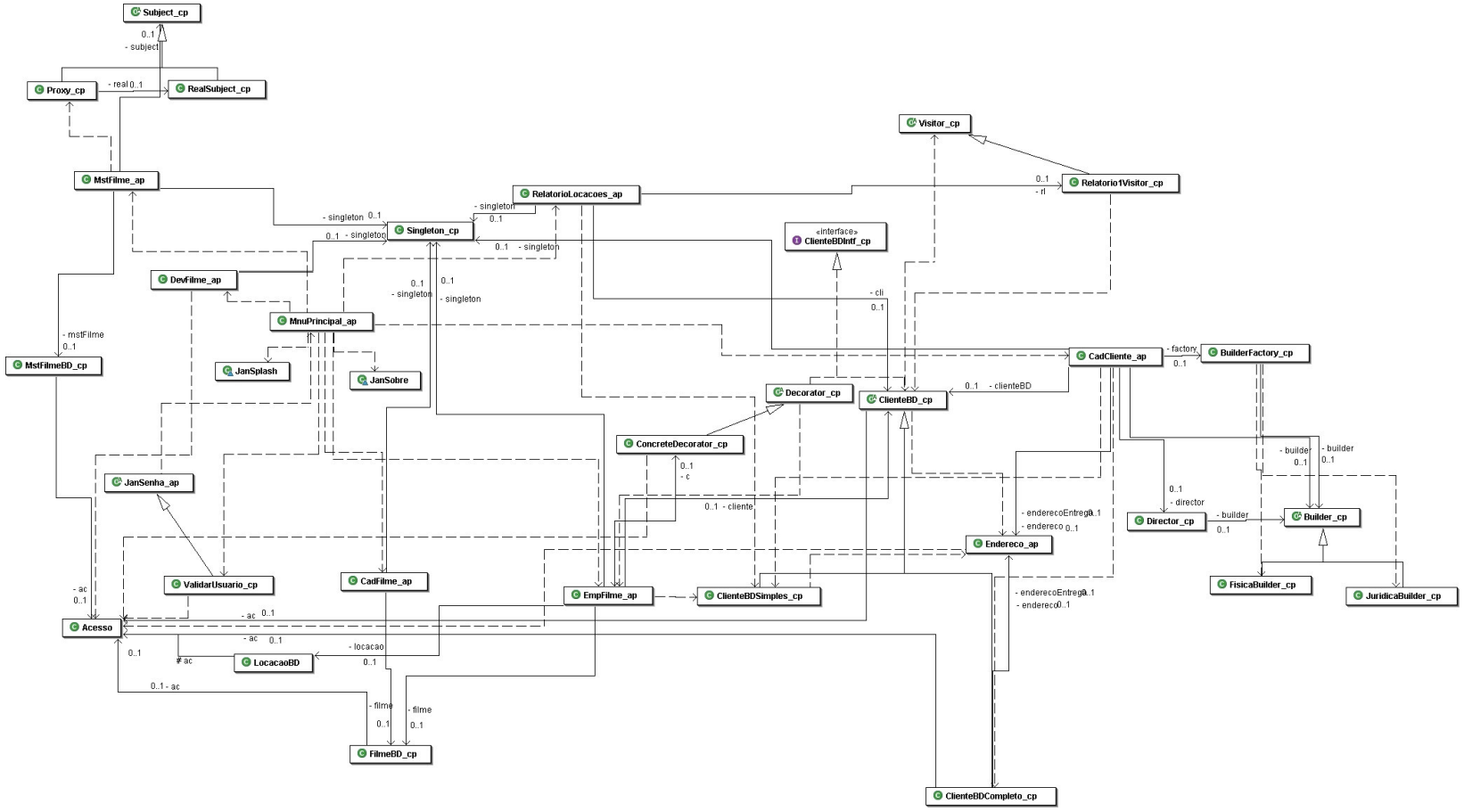

Figura 5 – Modelo de classes do sistema de videolocadora após as dez refatorações

4.6 Aplicação de Métricas

Uma forma de evidenciar os ganhos obtidos refatorando um sistema utilizando padrões de projeto é com a aplicação de métricas. A Tabela 8 mostra métricas definidas por Chidamber e Kemerer (1994) e outras propostas por Misra e Bhavsar (2003). Nessa tabela a primeira coluna contém um número seqüencial a ser utilizado na análise das métricas, seguido pelo nome da métrica, sua descrição e seus autores.

Tabela 8 – Métricas utilizadas no sistema de videolocadora

Nº	Métrica	Descrição	Autores
1	Métodos por classe	Quanto mais métodos há em uma classe, maior a dificuldade em mantê-la.	Chidamber e Kemerer
2	Número de filhos	Quanto maior o número de filhos, maior é o reuso e a probabilidade da superclasse ser utilizada de maneira inadequada.	
3	Falta de coesão de métodos	A falta de coesão indica que classes devem ser divididas em duas ou mais.	
4	Profundidade	Um aumento na média da profundidade dos caminhos aumenta a dificuldade, pois é mais difícil entender o software.	Misra e Bhavsar
5	Acoplamento/coesão	O aumento do acoplamento ou a falta de coesão aumentam a dificuldade, pois acoplamento reduz o encapsulamento e aumenta a complexidade; a coesão dos métodos aumenta o encapsulamento e a qualidade.	

As métricas são aplicadas no sistema original e após a realização das dez refatorações utilizando padrões. A Tabela 9 apresenta os resultados obtidos. A métrica 1 não teve alteração quanto à quantidade de métodos nas classes que já existiam no sistema. Portanto, não há aumento de dificuldade para a manutenção. Na métrica 2 observa-se um aumento no número de subclasses, isso porque há a geração de mais uma camada na arquitetura do sistema, ou seja, a classe de aplicação que antes não existia. Como prevê a métrica, há aumento do reuso e, com a definição de um processo de refatoramento definido e controlado, os riscos de que a superclasse seja utilizada inadequadamente são reduzidos. A métrica 3 indica que o software, antes da refatoração, era menos coeso do que após sua realização. Isso ocorre, pois métodos de diversas classes acessavam diretamente ao banco de dados e, ao mesmo tempo, controlavam/configuravam os componentes visuais da interface com o

usuário. Após a refatoração os métodos das classes de interface com o usuário não estão associados com o acesso ao banco, somente com as classes de aplicação. A métrica 4, se analisada superficialmente, determina que o software refatorado apresenta maior dificuldade em manutenção, visto que há mais níveis de herança do que no sistema original. Entretanto, o software refatorado é mais fácil de ser entendido justamente por ser mais coeso, com métodos para funções bem definidas e específicas. O reuso é outra vantagem da herança gerada, bem como a instanciação das classes somente quando elas são utilizadas. A análise da métrica 6 resulta que o software refatorado é mais fácil de ser mantido visto que há mais coesão nos métodos. As classes do sistema refatorado podem ser facilmente reutilizadas, dado o baixo acoplamento entre elas.

Tabela 9 – Resultado de aplicação das métricas

Nº	Antes das refatorações	Após as refatorações
1	Todas as classes somavam 69 métodos.	Todas as classes somam 69 métodos, sendo que duas tiveram um método adicionado e duas tiveram um método retirado.
2	Com exceção das classes herdadas da biblioteca Java (como `JDialog`, `JFrame`, por exemplo), não havia herança.	Há seis casos de herança.
3	Os mesmos métodos que acessavam o banco de dados configuravam componentes da interface com o usuário.	Várias classes de aplicação são criadas, evitando que métodos tenham mais de uma funcionalidade.
4	Como não havia herança, não havia profundidade de caminhos.	Os seis casos de herança obtidos pelos padrões geram, no máximo, dois níveis de profundidade na hierarquia de herança.
5	Não havia classes de aplicação.	Com as classes de aplicação, os métodos são mais coesos.

4.7 Considerações Finais

Este capítulo apresentou a aplicação do processo de refatoração com padrões de projeto em um sistema orientado a objetos, implementado na linguagem Java, utilizado para estudo de caso. As diretrizes de refatoração mostraram-se safisfatórias para tornar um sistema mais fácil de ser mantido e entendido.

No estudo de caso há classes que são utilizadas por classes de vários outros padrões. Um exemplo é a classe `ClienteBD_cp`, utilizada com o padrão *Decorator*, *Adapter* e *Visitor*. Da mesma forma, a classe `Singleton_cp` é utilizada por várias classes responsáveis por gerenciar a interface com o usuário. Outro ganho obtido com a aplicação do processo de refatoração foi a criação de mais uma camada no sistema, responsável exclusivamente por fornecer acesso à camada de banco de dados (classe `Acesso`), tornando o sistema mais robusto.

A nomenclatua adotada para nomear/renomear as classes facilita a identificação das que foram criadas devido à aplicação de padrões, já que os nomes são significativos. Isso pode ser verificado no modelo de classes do sistema refatorado, Figura 5. Além disso, visualiza-se facilmente as classes que foram alteradas devido à aplicação de um padrão, pois apresentam a extensão _ap. Esses dois fatores também contribuem para que manutenções futuras sejam facilitadas.

Dentre os passos relacionados às três etapas do processo, a que dispende mais tempo e esforços é a de refatoração com o padrão. Esse tempo refere-se à preparação do sistema para adequá-lo às novas classes e às características dos padrões de projeto. A identificação do padrão a ser utilizado, passo 3 da etapa 1, é apoiada pela lista de indícios apresentada na seção 3.3, porém também é necessário um conhecimento prévio do engenheiro de software sobre os padrões de projeto, aplicabilidade e estrutura de classes. Este processo de refatoração não prevê métodos de aprendizagem desses padrões. Por outro lado, caso o padrão seja implementado de maneira incorreta, resultando em arquitetura final ruim ou ainda, alteração em funções do sistema, o processo também não prevê um mecanismo definido para a correção do problema. Nesse caso, o processo de refatoração deve ser reiniciado, a fim de que o erro seja detectado e corrigido.

Embora não tenha sido possível a aplicação dos 23 padrões em um estudo de caso, com a experiência obtida pode-se inferir a aplicação de outros padrões caso algumas situações fossem encontradas no sistema, como mencionado na seção 4.2.

Embora Muraki e Saeki (2001) tenham afirmado que as métricas de Chidamber e Kemerer (1994) não são adequadas para medir a qualidade de sistemas que utilizam padrões de projeto, algumas delas podem ser exploradas em sistemas com essas características. Dentre elas destacam-se, principalmente, as que dizem respeito à possibilidade de reuso e alta coesão, artefatos presentes no software que é submetido ao processo de refatoração definido neste trabalho.

O processo definido nessa dissertação pode ser utilizado também para manutenção perfectiva em que apenas pequenas alterações de funcionalidade devem ser introduzidas. Essas alterações devem ser bem definidas, controláveis e específicas e podem ser realizadas por meio da execução dos passos de refatoração apresentados na seção 3.4. Nesse caso, quando o processo indica que determinado código do sistema existente deve ser modificado, substituído ou transferido, adapta-se o conceito, criando esse código, uma vez que não há a funcionalidade no sistema existente. Dessa forma, recomenda-se que o engenheiro de software tenha experiência na criação do código-fonte utilizado adequadamente os padrões de projeto.

A seguir são apresentados alguns comentários relativos à aplicação do processo de refatoração em outros sistemas.

4.7.1 Outros sistemas estudados

Além dos sete sistemas utilizados para a elaboração deste processo, vários outros foram consultados quanto à possibilidade de aplicação de padrões, porém não puderam ser utilizados, pois não foi reconhecida a existência de padrões de projeto no código-fonte existente. Os sistemas os quais um ou mais padrões foram aplicados (isoladamente) e que serviram como base para a elaboração dos processos de refatoração são: sistema de videolocadora, formas geométricas, editor de imagens, sistema de biblioteca, sistema de árvore hierárquica de livros, sistema de ordenação de números e letras e uma calculadora.

Dentre esses sistemas o de videolocadora foi o que possibilitou a aplicação de mais padrões, totalizando dez, como discutido no capítulo 4 (estudo de caso): *Builder*, *Prototype*, *Singleton*, *Adapter*, *Decorator*, *Proxy*, *Command*, *Iterator*, *Template Method* e *Visitor*.

O sistema de apresentação de formas geométricas, cuja finalidade é desenhar algumas formas geométricas básicas (arco, retângulo, oval e linha) de acordo com o solicitado pelo usuário e mudar a cor da tela, foi o segundo que possibilitou o uso de mais padrões. Nesse caso sete foram os aplicados:

1. *Abstract Factory*: utilizado para a construção das classes que representam as formas geométricas, assim delegação por herança foi utilizada.
2. *Factory Method*: utilizado para a construção das classes que representam as formas geométricas, assim delegação de objetos é a forma de implementação utilizada.

3. *Bridge*: desacopla o objeto forma do tipo que pode ser solicitado, relaciona-se às classes que manipulam as formas geométricas.
4. *Chain of Responsibility*: utilizado para tratar a maneira como uma solicitação de desenho é atendida.
5. *Observer*: utilizado para controlar o número de funções utilizadas pelo usuário no sistema e, quando o número máximo é atingido, nenhuma outra ação é permitida.
6. *State*: utilizado para que a decisão de qual forma deve ser desenhada não seja feita por meio de várias declarações `if/else`, como no sistema original.
7. *Strategy*: utilizado para adequar as diversas estratégias de desenho, ou seja, as várias formas geométricas que podem ser solicitadas, e que são contruídas da mesma forma, apresentando poucas diferenças no algoritmo.

No editor de imagens, que possibilita a edição de uma imagem existente ou a criação de uma nova, somente o padrão *Flyweight* foi aplicado, o qual insere na tela os ícones de atalho às funções disponíveis no menu do sistema.

No sistema de biblioteca, que permite o cadastro de usuários e livros, o empréstimo e a devolução e a visualização dos usuários e livros cadastrados, o padrão *Façade* foi aplicado dada a grande quantidade de classes e interfaces entre algumas classes mais relevantes para a realização das funções do sistema.

O sistema que apresenta a hierarquização de categorias de livros mostrados em uma árvore teve o padrão *Composite* aplicado, implementando não somente a visualização dos componentes hierarquizados, mas também um modelo contendo os mesmos dados estruturados.

No sistema de ordenação, que permite a criação de uma lista de números ou letras, sua posterior ordenação e possível retorno à lista original, foram aplicados dois padrões:

1. *Mediator*: utilizado para controlar a visibilidade dos botões do sistema, para que os demais componentes não necessitem conhecer essa informação, que é centralizada pelo padrão.
2. *Memento*: utilizado na funcionalidade de refazer/desfazer uma ordenação. Isso era feito utilizando-se duas listas, uma contendo os dados organizados da forma anterior e outra, os dados organizados da maneira atual.

Por fim, no sistema de calculadora, que permite a realização de cálculos aritméticos, foi aplicado apenas o padrão *Interpreter*, específico para o escopo do sistema, que é o de representar gramaticalmente uma linguagem.

No CD-ROM em anexo, estão disponíveis sobre esses sistemas o código original e o código-fonte com os padrões de projeto implementados conforme comentado anteriormente.

Capítulo 5
Considerações Finais

5.1 Considerações Iniciais

Após a realização dos estudos sobre padrões, manutenção, processos de engenharia reversa, reengenharia e refatoração, um conjunto de diretrizes, aqui chamado de processo de refatoração, foi apresenado. Seu objetivo é auxiliar os engenheiros de software na tarefa de manutenção preventiva de sistemas. Este capítulo apresenta alguns comentários dos outros sistemas utilizados como estudos de caso, seção 5.2; as contibuições que puderam ser observadas, seção 5.3 e, finalmente, trabalhos que podem dar continuidade a este, seção 5.4.

O processo de refatoração com padrões de projeto foi desenvolvido de forma prospectiva, isto é, com base na realização de alguns estudos de caso envolvendo a aplicação de padrões de projeto em sistemas originalmente desenvolvidos com o paradigma orientado a objetos e implementados com linguagem de programação Java. A partir da observação dos procedimentos realizados para que o sistema resultante apresentasse a mesma funcionalidade do original e o reconhecimento de padrões de projeto de software fosse possível para refatorar esse sistema, foram definidos passos para a realização da refatoração de acordo com as características de cada um dos padrões de projeto para cada padrão sugerido por Gamma et al. (1995). A lista de indícios da presença do padrão no sistema foi elaborada com base nos mesmos estudos de caso. Esses índicios procuram auxiliar o engenheiro de software com prováveis características que indicam a existência de um padrão.

A fim de que o engenheiro de software obtenha a documentação atualizada do sistema após esse processo de refatoração, manutenção preventiva, e também atenda as práticas de gestão de configuração de software, o modelo de classes do sistema resultante deve ser elaborado. Essa elaboração ocorre juntamente com a refatoração do sistema de forma iterativa, na etapa 2 do processo.

5.2 Contribuições deste Processo

O estudo de caso realizado a partir o sistema de videolocadora possibilitou a aplicação das diretrizes para a refatoração de sistemas utilizando padrões de forma cumulativa. Dessa forma, uma das contribuições deste projeto é a lista de indícios que fornece ao engenheiro de software características da existência de um padrão de projeto a partir do código existente ou da funcionalidade desse sistema. O engenheiro de software então, pode implementar esse padrão de projeto possibilitando maior organização e melhor documentação.

Sabe-se que a utilização de padrões de projeto não é trivial quando um sistema é desenvolvido e também apresenta dificuldades para o seu reconhecimento em um sistema existente. As diretrizes aqui elaboradas amenizam essa dificuldade, mas não a elimina, pois exige do engenheiro de software conhecimento dos padrões de projeto.

Embora somente tenham sido utilizados sistemas de informação simples, com funcionalidade bem conhecida, infere-se que o processo pode ser aplicado a sistemas em outros domínios. A dificuldade na obtenção de sistemas com código disponível, que pudessem ser utilizados para validação do processo foi o motivo pelo qual outros tipos de sistemas não foram utilizados.

A utilização de algumas métricas propostas por Chidamber e Kemerer (1994) fornecem subsídios para a avaliação do processo aplicado de forma positiva. No entanto, a observação feita por Muraki e Saeki (2001) de que essas métricas não são suficientes para avaliar a refatoração também pode ser observada.

A aplicação de métricas para avaliar a manutenibilidade do sistema de videolocadora antes e após a execução do processo de refatoração, resulta que há melhoria na estrutura com a aplicação desse processo, conforme citado na seção 4.6 sobre a análise das métricas aplicadas.

5.3 Trabalhos Futuros

A continuação deste trabalho possibilitará que pontos deixados em aberto aqui possam ser analisados de forma a consolidar as diretrizes apresentadas. Algumas sugestões são:

- Realização de mais estudos de caso, a fim de que a lista de indícios de padrões possa ser aprimorada e generalizada, especificando com mais eficiência como deve ser o comportamento do sistema e como o seu código deve estar para que determinado padrão seja aplicado.
- Definição de alterações no processo de refatoração, avaliando a possibilidade de aplicar mais de um padrão durante um só processo de refatoração.
- Definição de mecanismos de correção, caso o processo de refatoração não seja executado adequadamente.
- Avaliação de ferramentas automatizadas que possam apoiar o processo, tanto na identificação do padrão quanto em sua aplicação no sistema.
- Refinamento dos indícios apresentados para localização de padrões de forma mais facilitada no sistema existente.
- Aplicação do processo de refatoração em sistemas existentes em diversos domínios.

Apêndice A

Este Apêndice detalha os processos de refatoração para os padrões de projeto definidos por Gamma et al. (1995) que não foram definidos no capítulo 3. A apresentação dos padrões obedece as categorias definidas por Gamma et al. (1995). Os padrões *Singleton*, *Decorator* e *Visitor* são omitidos, pois são apresentados na seção 3.4.

Categoria: Criação

Abstract Factory

Objetivo: Fornecer uma interface para a criação de uma família de objetos relacionados ou dependentes sem especificar suas classes concretas.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.
   a) Criar a classe abstrata `AbstractFactory_cp`. Utilizar o gabarito apresentado no Quadro 25.
      i. As fábricas que podem ser instanciadas devem ser declaradas (linhas 2 e 3).
      ii. Há um método estático, denominado `getFactory`, que recebe um parâmetro utilizado para decidir qual fábrica deve ser instanciada (linha 5).
      iii. O número de comparações deve ser igual ao número de fábricas disponíveis (linhas 5 e 7).
      iv. `MetodoX` e `MetodoY` são os métodos a serem implementados pelas subclasses (linhas 9 e 10).

Quadro 25 – Classe `AbstractFactory_cp`

```
1 public abstract class AbstractFactory_cp {
2     private static final FactoryA factoryA = new FactoryA();
3     private static final FactoryB factoryB = new FactoryB();
      //instanciação de outras fábricas que podem ser utilizadas
4  static final AbstractFactory_cp getFactory(<<tipo_param>> type){
5   if(type == <<comparacaoA>>)
6     return factoryA;
7   else if(type == <<comparacaoB>>)
8     return factoryB;
   ...
   }
9  public abstract ClassA MetodoX(<<parametros>>);
10 public abstract ClassB MetodoY(<<parametros>>);
   ...
}
```

b) Criar a classe `Factory_cp`. Utilizar o gabarito apresentado no Quadro 26.

   i. Esta classe é herdeira de `AbstractFactory_cp` e deve implementar os métodos abstratos definidos pela superclasse (linha 1).

   ii. O conteúdo dos métodos implementados por esta classe deve representar a instanciação dos objetos da fábrica (linhas 3 e 5).

   iii. O método `MetodoX` é responsável pela instanciação de um objeto do tipo `ClassA` (linha 2). Enquanto que o método `MetodoY` (linha 4) é responsável pela instanciação de um objeto do tipo `ClassB`. Se houver a necessidade de parâmetros, deve-se inseri-los.

   iv. Definir métodos semelhantes a `MetodoX` e/ou `MetodoY`, de acordo com os objetos que serão instanciados por meio da fábrica.

Quadro 26 – Classe `Factory_cp`, padrão *Abstract Factory*

```
1 public class Factory_cp extends AbstractFactory_cp{
2     public ClassA MetodoX(<<parametros>>){
3           return new ClassA(nome_param);
      }
4     public ClassB MetodoY(<<parametros>>)
5           return new ClassB();
      }
}
```

c) Se ainda não existem, criar as classes a serem instanciadas pela fábrica. As classes `ClassA` e `ClassB` são exemplos dessas classes, apresentadas no Quadro 27 e Quadro 28, cujos objetos são instanciados pela fábrica (classe `Factory_cp`).

Quadro 27 – Classe `ClassA`, padrão *Abstract Factory*

```
1 public class ClassA {
      ...
2     public ClassA(<<tipo_param>> <<nome_param>>)
```

```
3     {     //código-fonte pertinente ao construtor da classe ClassA   }
      //métodos da classe ClassA
}
```

Quadro 28 – Classe `ClassB`, padrão *Abstract Factory*

```
1 public class ClassB {
      ...
2     public ClassB()
      {     //código-fonte pertinente ao construtor da classe ClassB   }
      //métodos da classe ClassB
}
```

d) Retomar a classe que contém o trecho de código identificado com o padrão *Abstract Factory* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo).

e) Reestruturar a classe `X_ap`. Utilizar o gabarito apresentado no Quadro 29.

   i. Declarar os objetos (linhas 2 e 3) que serão criados pela fábrica (classe `Factory_cp`).

   ii. Criar a classe que representa a fábrica abstrata, `AbstractFactory_cp` (linha 4).

   iii. Inserir o código responsável pela instanciação da classe fábrica correta (linha 5).

Quadro 29 – Classe `X_ap`, padrão *Abstract Factory*

```
1 public class X_ap{
...
2    private ClassA classA;
3    private ClassB classB;
4    private AbstractFactory_cp factory = new AbstractFactory_cp();
5    factory = AbstractFactory_cp.getFactory(<<nome_param>>);
...
}
```

f) Substituir na classe `X_ap` os trechos de código antes responsáveis por instanciar as classes as quais passam a serem instanciadas pela fábrica (classe `AbstractFactory_cp`).

   i. O Quadro 30 apresenta o gabarito do trecho de código a ser retirado da classe `X_ap`.

   ii. O Quadro 31 apresenta o gabarito do trecho de código a ser inserido na classe `X_ap`.

Quadro 30 – Código a ser retirado da classe `X_ap`, padrão *Abstract Factory*

```
1 public class X_ap{
...
2    classA = new ClassA(nome_param);
```

```
3     classB = new ClassB();
...
}
```

Quadro 31 – Código a ser inserido na classe `X_ap`, padrão *Abstract Factory*

```
1 public class X_ap{
...
2     classA = factory.MetodoX(nome_param);
3     classB = factory.MetodoY();
...
}
```

g) As chamadas aos demais métodos das classes `ClassA` e `ClassB` permanecem inalteradas, uma vez que a fábrica já instancia esses objetos.

Builder

Problema: Separar a construção de um objeto complexo da sua representação de modo que o mesmo processo de construção possa criar diferentes representações.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

a) Criar a classe abstrata `Builder_cp`. Utilizar o gabarito apresentado no Quadro 32.

i. Esta classe agrupa o conjunto de componentes específicos para cada representação em um componente do tipo `JPanel`. O método `getPanel` retorna o atributo `panel` (linha 4) e o método `createComponents`, abstrato, deve ser definido pelas classes herdeiras (linha 5).

Quadro 32 – Classe `Builder_cp`

```
1 public abstract class Builder_cp {
2     protected JPanel panel;
3     public JPanel getPanel() {
4       return this.panel;
      }
5     public abstract void createComponents();
}
```

b) Criar a classe `Director_cp`. Utilizar o código-fonte apresentado no Quadro 33.

i. Solicita ao *builder* (já instanciado) a criação dos componentes específicos através do método `build` (linha 6).

Quadro 33 – Classe `Director_cp`

```
1 public class Director_cp {
2     private Builder_cp builder;
3     public Director_cp(Builder_cp bldr)
4     {     builder = bldr;   }
```

```
5     public void build()
6     {     builder.createComponents();   }
}
```

c) Criar as classes herdeiras de `Builder_cp` (*builders*), que devem definir o método `createComponents`. O Quadro 34 apresenta o gabarito para essas classes.

   i. Substituir `<<Nome>>` (linha 1) por um nome significativo para a classe. Por exemplo, se o nome for `A`, a classe será denominada `ABuilder_cp`.

   ii. Criar os componentes necessários à composição do objeto mais complexo substituindo `<<tipo_comp1>>` (linha 2) pelo tipo do primeiro componente e `<<nome_comp1>>` (linha 2) pelo seu nome.

   iii. Instanciar o atributo `panel` (linha 7), no método `createComponents`, definido como protegido pela superclasse (`Builder_cp`, linha 2 do Quadro 32).

   iv. Definir a apresentação do objeto através da classe `GridLayout`: onde `arg1` é o número de linhas e `arg2` é o número de colunas (linha 8). Para mais informações a respeito da classe `GridLayout`, ver Java (2005).

   v. Adicionar todos os componentes (criados no passo **c.ii**) ao componente `panel` (linhas 9 e 10).

Quadro 34 – Classe herdeira de `Builder_cp`

```
1 public class <<Nome>>Builder_cp extends Builder_cp{
      //criação dos componentes necessários ao objeto
2     private <<tipo_comp1>> <<nome_comp1>> = new <<tipo_comp1>>
3                                                 (<<parametros>>);
4     private <<tipo_comp2>> <<nome_comp2>> = new <<tipo_comp2>>
5                                                 (<<parametros>>);
      ...
6     public void createComponents() {
7           panel = new JPanel();
8           panel.setLayout(new GridLayout(arg1, arg2));
9           panel.add(<<nome_comp1>>);
10          panel.add(<<nome_comp2>>);
            ...
      }
}
```

d) Criar a classe fábrica `BuilderFactory_cp`, que decide qual dos *builders* deve ser instanciado. Utilizar o gabarito apresentado no Quadro 35.

   i. O método `getBuilder` (linha 3) recebe como parâmetro um nome, que é comparado ao tipo do *builder*. Deve haver tantas comparações quantas forem as classes herdeiras de `Builder_cp`.

ii. Os termos `<<String_comparacao_A>>` e `<<String_comparacao_B>>` (linhas 4 e 6) devem ser substituídos pelos valores adequados, a fim de que a comparação possa ser realizada corretamente.

Quadro 35 – Classe `BuilderFactory_cp`

```
1 public class BuilderFactory_cp {
2     private Builder_cp builder = null;
3     public Builder_cp getBuilder(String type){
4           if(type.equals("<<String_comparacao_A>>"))
5                 builder = new ABuilder_cp();
6           else if(type.equals("<<String_comparacao_B>>"))
7                 builder = new BBuilder_cp();
8           //else...
9           return builder;
      }
}
```

e) Retomar a classe que contém o trecho de código identificado com o padrão *Builder* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo).

f) Reestruturar a classe `X_ap`. Utilizar o gabarito apresentado no Quadro 36.

i. Criar os objetos representantes das classes `Builder_cp`, `Director_cp` e `BuilderFactory_cp` (linhas 2, 3 e 4).

ii. Retomar o trecho de código que engatilha a exibição de um ou outro grupo de componentes (linhas 7 e 9) e substituí-lo.

iii. Substituir `<<tipo_retorno>>`, `<<nome_metodo>>` e `<<tipo_param>>` pelo tipo de retorno do método, seu nome e o tipo dos parâmetros (se houver), respectivamente (linha 5).

iv. Substituir `<<atributo_comparacao>>` pelo valor adequado, para a comparação com o atributo `tipo`.

v. Substituir `<<String_comparacao_A>>` e `<< String_comparacao_B>>` (linhas 6 e 8, respectivamente) pelos valores adequados, definidos no Quadro 35.

vi. As linhas de 11 a 17 do Quadro 36 devem ser mantidas.

Quadro 36 – Classe `X_ap`, padrão *Builder*

```
1 public class X_ap{
2   private Builder_cp builder;
3   private Director_cp director;
4   private BuilderFactory_cp factory = new BuilderFactory_cp();
...
5  public <<tipo_retorno>> <<nome_metodo>>(<<tipo_param>> tipo){
6     if(tipo == <<atributo_comparacao1>>){
7           builder = factory.getBuilder("<<String_comparacao_A>>");
```

```
8      }else if(tipo == <<atributo_comparacao2>>){
9            builder = factory.getBuilder("<<String_comparacao_B>>");
10     }
11     director = new Director_cp(builder);
12     director.build();
13     JPanel panel1 = builder.getPanel();
14     Container c = getContentPane();
15     c.add(generalPanel);
16     generalPanel.add(panel1);
17     show();
   }
...
}
```

Factory Method

Problema: Definir uma interface para criar um objeto, mas deixar as subclasses decidirem qual classe instanciar. Permite adiar a instanciação para subclasses.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

   a) Criar a interface que define todos os métodos a serem criados pela fábrica. Utilizar o gabarito apresentado no Quadro 37.

      i. `<<Intf>>` é o nome da interface (linha 1); `<<tipo_retorno>>` são os tipos de retorno dos métodos (linhas 2, 3 e 4); `<<nome_metodo1>>`, `<<nome_metodo2>>`, `<<nome_metodoN>>` são os nomes dos métodos a serem criados pela fábrica (linhas 2, 3 e 4); `<<parametros>>` (linhas 2, 3 e 4) são os parâmetros dos métodos, se houver.

Quadro 37 – Interface da fábrica

```
1 public interface <<Intf>> {
2      public <<tipo_retorno>> <<nome_metodo1>>(<<parametros>>){}
3      public <<tipo_retorno>> <<nome_metodo2>>(<<parametros>>){}
       ...
4      public <<tipo_retorno>> <<nome_metodoN>>(<<parametros>>){}
       //inserir quantos métodos forem necessários
}
```

   b) Criar as classes que implementam a interface definida no passo **a**. Utilizar o gabarito apresentado no Quadro 38.

      i. `<<Nome_classe>>` (linha 1) é o nome da classe e **<<parametros>>** são os parâmetros necessários aos métodos.

Quadro 38 – Classe que implementa a interface da fábrica

```
1 public class <<Nome_classe>> implements <<Intf>> {
      //construtor
2     public <<Nome_classe>>(<<parametros>>){
      //código-fonte adequado
      }
3     public <<tipo_retorno>> <<nome_metodo1>>(<<parametros>>){
      //código-fonte adequado
      }
      ...
4     public <<tipo_retorno>> <<nome_metodoN >>(<<paramametros>>){
      //código-fonte adequado
      }
}
```

c) Criar a classe fábrica `Factory_cp` que analisa os parâmetros fornecidos e instancia a classe correta. Utilizar o gabarito apresentado no Quadro 39.

   i. O objeto `intf` (linha 2) representa a classe `Intf`, criada no passo **a**.

   ii. O método `createClass` (linha 3) retorna um objeto do tipo `Intf`, de acordo com a subclasse a ser invocada.

   iii. `<<Nome_classe>>` e `<<Nome_classe_1>>` (linhas 5 e 7, respectivamente) implementam a interface `Intf`.

Quadro 39 – Classe `Factory_cp`, padrão *Factory Method*

```
1 public class Factory_cp {
2     private Intf intf;
3     public Intf createClass(<<tipo_param>> param) {
4           if (param == <<variavel_comparacao>>)
5                 intf = new <<Nome_classe>>(<<parametros>>);
6           else
7                 intf = new <<Nome_classe_1>>(<<parametros>>);
      }
}
```

d) Retomar a classe que contém o trecho de código identificado com o padrão *Factory Method* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo).

e) Reestruturar a classe `X_ap`. Utilizar o gabarito apresentado no Quadro 40.

   i. Criar e instanciar o objeto representante da classe `Factory_cp` (linha 2).

   ii. Criar o objeto da interface definida no passo **a** (linha 3).

   iii. Retomar o trecho de código que cria os objetos que passam a ser criados pelas classes que implementam a interface `Intf` e substituir esse trecho pela utilização do objeto fábrica, como mostram as linhas de 4 a 6.

Quadro 40 – Classe `X_ap`, padrão *Factory Method*

```
1 public class X_ap{
...
2    private Factory_cp factory = new Factory_cp();
3    private Intf intf;
4    intf = factory.createClass(<<parametros>>);
5    intf.<<nome_metodo1>>(<<parametros>>);
6    intf.<<nome_metodoN>>(<<parametros>>);
...
}
```

Prototype

Problema: Especificar os tipos de objetos a serem criados usando uma instância protótipo e criar novos objetos pela cópia deste protótipo.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.
   a) Retomar a classe que contém o trecho de código identificado com o padrão *Prototype* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo).
   b) Retomar a classe que contém o objeto a ser "clonado" (classe `Y`, por exemplo) e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `Y_ap`, por exemplo).
   c) Tornar a classe `Y_ap` implementadora da interface `Cloneable` (linha 1 do Quadro 41), que define o método `clone`. Definir esse método exatamente como apresentado nas linhas de 2 a 7 do Quadro 41.

Quadro 41 – Classe `Y_ap`, padrão *Prototype*

```
1 public class Y_ap implements Cloneable{
...
2     public Object clone(){
3           try {
4                 return super.clone();
5           } catch (Exception e) {
6                       System.out.println(e.getMessage());
7                 return null;
            }
      }
...
}
```

   d) Retomar o trecho de código que contém dois objetos idênticos (classe `X_ap`) e identificar o trecho de código em que um objeto (por exemplo, `objeto1`) torna-se idêntico ao outro (por exemplo, `objeto`). O Quadro 42 apresenta o gabarito do código a ser retirado; o Quadro 43 apresenta o gabarito do código a ser inserido.

i. No código a ser retirado (Quadro 42) `objeto` e `objeto1` (linhas 2 e 3) são os objetos original e "clone", respectivamente.
ii. No código a ser retirado (Quadro 42), nas linhas 4 e 5 ocorre a cópia de dados de um objeto para o outro.
iii. No código a ser inserido (Quadro 43), há a instanciação do objeto original e a declaração do objeto "clone" (linhas 2 e 3, respectivamente).
iv. Na linha 4 (Quadro 43) o objeto original é "clonado".

Quadro 42 – Código a ser retirado da classe `X_ap`, padrão *Prototype*

```
1 class X_ap{
...
2  private X_ap objeto = new X_ap();
3  private X_ap objeto1 = new X_ap();
...
4  objeto1.setCampo1(objeto.getCampo1());
5 objeto1.setCampo2(objeto.getCampo2());
...
}
```

Quadro 43 – Código a ser inserido na classe `X_ap`, padrão *Prototype*

```
1 classe X_ap{
...
2   private X_ap objeto = new X_ap();
3   private X_ap;
...
4   objeto1 = (X_ap)objeto.clone();
...
}
```

Categoria: Estrutural

Adapter

Objetivo: Converter a interface de uma classe em outra interface, esperada pelos clientes. O *Adapter* permite que classes com interfaces incompatíveis trabalhem em conjunto, o que, de outra forma, seria impossível.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.
   a) Identificar a classe que necessita ser adaptada, por exemplo, `X`. Utilizar o gabarito apresentado no Quadro 44.

Quadro 44 – Classe `X`, padrão *Adapter*

```
1 public class X{
   //atributos da classe
2   public <<tipo_ret>> Metodo1(<<params>>){...}
```

```
3   public <<tipo_ret>> Metodo2(<<params>>){...}
    ...
4   public <<tipo_ret>> MetodoN(<<params>>){...}
}
```

b) Identificar a classe cliente de `X`, ou seja, que acessa métodos de `X`, e o código que realiza as adaptações necessárias nas informações obtidas de `X`, a fim de que se tornem válidas. Essa classe é chamada de `Y`. Renomear a classe `Y`, adicionando ao seu final a extensão `_ap` (classe `Y_ap`, por exemplo). Pode haver mais de uma classe cliente de `X` e que adapte um ou mais métodos.

c) Criar a classe adaptadora `Adapter_cp`. Se houver a necessidade de criar mais de um adaptador, criar outras classes do tipo `Adapter_cp`. Utilizar o gabarito apresentado no Quadro 45.

   i. Inserir os métodos necessários à interface adaptadora, que devem invocar métodos da classe `X` (linhas 5 e 7). Os trechos de código identificados no passo **b** (classe `Y`) devem ser transferidos adequadamente para os métodos equivalentes da classe `Adapter_cp`.

   ii. `<<params>>` (linha 2) são os parâmetros do construtor da classe `X` (se houver); `<<tipo>>` (linha 3) é o tipo do atributo param3; `<<tipo_param>>` é o tipo dos parâmetros (linhas 4 e 6); `param1` e `param2` (linhas 4 e 6) são parâmetros (se houver).

Quadro 45 – Classe `Adapter_cp`, padrão *Adapter*

```
1 public class Adapter_cp{
   //atributos da classe
2   private X obj = new X(<<params>>);
3   private <<tipo>> param3;
   ...
4   public <<tipo_ret>> MetodoA(<<tipo_param>> param1,
                                <<tipo_param>> param2){
       //operações diversas, transferidas da classe Y, opcionais
5       obj.Metodo1(param1, param2, param3);
       //operações diversas, transferidas da classe Y, opcionais
  }
6   public <<tipo_ret>> MetodoB(<<tipo_param>> param1,
                                <<tipo_param>> param2){
       //operações diversas, transferidas da classe Y, opcionais
7       obj.Metodo2(param2, param3, param1);
       //operações diversas, transferidas da classe Y, opcionais
  }
...
}
```

d) Retirar da classe `Y_ap` o objeto representante da classe `X` (linha 2 do Quadro 46) e substituí-lo por um da classe `Adapter_cp` (linha 2 do Quadro 47). O Quadro 46

mostra o código a ser retirado da classe `Y_ap` e o Quadro 47, o código a ser inserido nessa classe.

Quadro 46 – Código a ser retirado da classe `Y_ap`, padrão *Adapter*

```
1 public class Y_ap{
2   private X obj = new X(<<params>>);
   ...
3   obj.Metodo1(<<params>>);
4   obj.Metodo2(<<params>>);
...
}
```

Quadro 47 – Código a ser inserido na classe `Y_ap`, padrão *Adapter*

```
1 public class Y_ap{
2   private Adapter_cp adapter = new Adapter_cp(<<params>>);
   ...
3   adapter.MetodoA(<<params>>);
4   adapter.MetodoB(<<params>>);
...
}
```

Bridge

Problema: Desacoplar uma abstração da sua implementação, de modo que as duas possam variar independentemente.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

   a) Identificar a classe que realiza a representação dos mesmos dados de formas distintas. O Quadro 48 apresenta um gabarito para essa classe, denominada `X`.

Quadro 48 – Classe `X`, padrão *Bridge*

```
public class X{
   //v é uma estrutura que possui os dados a serem apresentados
   //código pertinente à representação da maneira A
   //código pertinente à representação da maneira B
   //código pertinente à representação da maneira C
}
```

   b) Criar classes separadas para cada tipo de representação dos dados, identificados no passo **a**. De acordo com o Quadro 48, as classes a serem criadas são `A`, `B` e `C`, cujos conteúdos são os definidos pela classe `X`. Os dados devem ser passados como parâmetros, pois serão necessários para a representação. O Quadro 49 apresenta um gabarito para essas classes.

Quadro 49 – Classe `A`, padrão *Bridge*

```
public class A{
   public A(<<tipo>> v){//v é a estrutura a ser utilizada
      //código pertinente à representação da maneira A
   }
}
```

c) Criar a classe `Abstraction_cp`, o *Bridge*. Utilizar o gabarito apresentado no Quadro 50.

   i. Deve haver constantes pré-definidas para identificar as classes criadas no passo **b**, as quais o *Bridge* pode instanciar (linha 2).

   ii. O construtor deve instanciar uma classe (linhas 6 e 7) dentre as possíveis representações, identificadas no passo **b** e, para isso, são necessários parâmetros (linha 5).

Quadro 50 – Classe `Abstraction_cp`, padrão *Bridge*

```
1 public class Abstraction_cp{
2   static public final int CONSTA = 1, CONSTB = 2;
3   private A objA;
4   private B objB;
5   public Abstraction_cp(<<tipo>> param1){
6      if(param1 == CONSTA) objA = new A(<<params>>);
7      else if(param1 == CONSTB) objB = new B(<<params>>);  ...
    }
...
}
```

g) Identificar a classe (`W`, por exemplo) que faz referência à classe `X` e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `W_ap`, por exemplo).

h) Reestruturar a classe `W_ap`. Utilizar o gabarito apresentado no Quadro 51.

   i. Retirar os objetos representantes da classe `X`.

   ii. Inserir quantos objetos do tipo `Abstraction_cp` forem necessários (linhas2 e 3).

   iii. Inserir as respectivas referências às classes `A`, `B` e/ou `C`, de acordo com a representação dos dados necessária (linhas 4 e 5).

Quadro 51 – Classe `W_ap` , padrão *Bridge*

```
1 public class W_ap{
2   private Abstraction_cp abs1;
3   private Abstraction_cp abs2;
4   abs1 = new Abstraction_cp(abs.CONSTA);
5   abs2 = new Abstraction_cp(abs.CONSTB);   …
}
```

i) Excluir a classe `X` do sistema.

Composite

Objetivo: Compor objetos em estruturas de árvore para representarem hierarquias parte-todo. *Composite* permite aos clientes tratarem de maneira uniforme objetos individuais e composição de objetos.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.
   a) Criar a classe de aplicação para o elemento a ser hierarquizado, denominada `X_cp`. Utilizar o gabarito apresentado no Quadro 52.
      i. A classe deve conter os métodos `set` e `get` (linhas 6 a 11) pertinentes aos atributos do elemento.
   b) Inserir na classe `X_cp` os métodos para manipulação do elemento a ser hierarquizado. O código apresentado entre as linhas 12 e 30 do Quadro 52 são referentes a esses métodos.
      i. `addElement` (linha 12) insere um novo elemento na hierarquia; `getElements` (linha 14) obtém todos os elementos e `getChild` (linha 16) obtém o filho de um elemento da hierarquia.

Quadro 52 – Classe `X_cp` , padrão *Composite*

```
1 public class X_cp{
2   private <<tipo>> att1;
3   private <<tipo>> att2;
4   private <<tipo>> att3;
5   private Vector vectorX = new Vector();
6   public void setAtt1(<<tipo>> param1){ att1 = param1; }
7   public void setAtt2(<<tipo>> param2){ att2 = param2; }
8   public void setAtt3(<<tipo>> param3){ att3 = param3; }
8   public <<tipo>> getAtt1(){ return att1; }
10  public <<tipo>> getAtt2(){ return att2; }
11  public <<tipo>> getAtt3(){ return att3; }
12  public void addElement(X objX){
13     vectorX.addElement(objX);
    }
14  public Enumeration getElements() {
15     return vectorX.elements();
    }
16    public Book getChild(<<tipo>> param) {
17    X newX = null;
18    boolean found = false;
19     Enumeration e = elements();
20     while (e.hasMoreElements() && (!found)) {
21          newX = (X) e.nextElement();
22          if(newX.getAtt1() == param)
23             found = true;
24          if (!found) {
25            newX = newX.getChild(s);
26            found = (newX != null);
```

```
            }
        }
27      if (found)
28          return newX;
29      else
30          return null;
   }
}
```

c) Identificar a classe que contém o código que insere os elementos na hierarquia, denominada `Y` e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `Y_ap`, por exemplo).

d) Identificar o código que insere os elementos na hierarquia, que pode ser semelhante ao do gabarito apresentado no Quadro 53, o qual utiliza a biblioteca `DefaultMutableTreeNode`, de Java (2005).

Quadro 53 – Classe `Y_ap`, padrão *Composite*

```
1 public class Y_ap{
...
2   DefaultMutableTreeNode raiz = new DefaultMutableTreeNode("Raiz");
   ...
3   DefaultMutableTreeNode e1 = new DefaultMutableTreeNode("E1");
4   raiz.add(e1);
5   DefaultMutableTreeNode e2 = new DefaultMutableTreeNode("E2");
6   e1.add(e2);
   ...
}
```

e) Declarar, na classe `Y_ap`, os objetos referentes à classe `X_ap` e estruturar a hierarquia. Utilizar o gabarito apresentado no Quadro 54.

   i. O número de objetos da classe `Y_ap` deve ser igual ao número de níveis da hierarquia. Nas linhas 3, 4 e 5 são apresentados três níveis, a partir da raiz (linha 2).

   ii. O trecho de código identificado no passo **c** deve ser substituído por código de criação do modelo da hierarquia (linhas 6 e 8). Nesse gabarito, há dois elementos no segundo nível (abaixo da raiz) e um elemento no terceiro nível.

Quadro 54 – Classe `Y_ap`, padrão *Composite*

```
1 public class Y_ap{
2   private X_cp raizModelo = new objX(“Raiz”);
3   private X_cp objN1 = new X_cp(“Nivel 1”);
4   private X_cp objN2 = new X_cp(“Nivel 1”);
5   private X_cp objN3 = new X_cp(“Nivel 2”);
6   raizModelo.addElement(objN1);
7   raizModelo.addElement(objN2);
8   objN2.addElement(objN3);
...
}
```

f) Criar na classe `Y_ap` a estrutura hierárquica, utilizando a biblioteca `DefaultMutableTreeNode`. Utilizar o gabarito apresentado no Quadro 55.

i. Essa criação deve acontecer após a execução do passo **d**, ou seja, após a criação do modelo hierárquico.

ii. `raizModelo` (linha 6) deve ser substituído pelo nome do objeto da classe `X_cp`; `getAtt1` (linhas 5 e 11) é o nome do método que retorna o conteúdo a ser exibido para cada elemento na hierarquia.

iii. Importar as bibliotecas `javax.swing.JTree` (linha 3), `javax.swing.tree.DefaultMutableTreeNode` (linha 2) e `java.util.Enumeration` (linha 5).

Quadro 55 – Classe `Y_ap`, padrão Composite

```
1 import javax.swing.JTree;
2 import javax.swing.tree.DefaultMutableTreeNode;
3 import java.util.Enumeration;
4 public class Y_ap{
...
5     DefaultMutableTreeNode raizVisual =
                    new DefaultMutableTreeNode(raizModelo.getAtt1());
6     JTree tree = new JTree(raizModelo);
7     DefaultMutableTreeNode node;
8     Enumeration e = raizModelo.elements();
9     while (e.hasMoreElements()) {
10       X_cp newX = (X_cp) e.nextElement();
11       node = new DefaultMutableTreeNode(newX.getAtt1());
12       raiz.add(node);
13       createNodes(node, newX);
     }
...
}
```

Façade

Objetivo: Fornecer uma interface unificada para um conjunto de interfaces em um subsistema. *Façade* define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

a) Identificar o subsistema, ou seja, quais classes terão suas interfaces utilizadas no padrão *Façade*. Essas classes não são alteradas. O Quadro 56 apresenta o gabarito para essas classes.

Quadro 56 – Classes da interface *Façade*

```
1 public class A1{
   //atributos da classe
2   public <<tipo_ret>> MetodoAA1(<<params>>){...}
3   public <<tipo_ret>> MetodoBA1(<<params>>){...}
4   public <<tipo_ret>> MetodoCA1(<<params>>){...}
...
}
...
5 public class AN{
   //atributos da classe
6   public <<tipo_ret>> MetodoAN1(<<params>>){...}
7   public <<tipo_ret>> MetodoBN1(<<params>>){...}
8   public <<tipo_ret>> MetodoCN1(<<params>>){...}
...
}
```

b) Identificar as classes clientes (`X`, por exemplo) que fazem acesso aos métodos das classes identificadas no passo **a** e renomeá-las, adicionando aos seus finais a extensão `_ap` (classe `X_ap`, por exemplo). O Quadro 57 apresenta um gabarito para uma dessas classes.

   i. `attA1` é um atributo da classe `A1` e `attA2`, um atributo da classe `A2` (linhas 2 e 3).

   ii. O cliente `C1` solicita, por meio do atributo `attA1`, um método da classe `A1`, cujo parâmetro é do tipo `A2` (linha 4).

   iii. O cliente `C1` solicita, por meio do atributo `att1`, o método `MetodoBA1` e, por meio do atributo `attA2`, o método `MetodoAA2` (linhas 5 e 6, respectivamente).

Quadro 57 – Classe cliente da interface *Façade*

```
1 public class C1{
...
2   private A1 attA1 = new A1(<<params>>);
3   private A2 attA2 = new A2(<<params>>);
   ...
4   attA1.MetodoAA1(attA2.MetodoCA2(<<params>>));
5   attA1.MetodoBA1(<<params>>);
6   attA2.MetodoAA2(<<params>>);
...
}
```

c) Criar a classe `Facade_cp`. Utilizar o gabarito apresentado no Quadro 58.

   i. Declarar e instanciar todas as classes identificadas no passo **a** (linhas 2 e 3).

   ii. Compor a funcionalidade alcançada nas classes identificadas no passo **b** criando métodos com as mesmas chamadas na classe `Facade_cp` (linha 4).

   iii. Ajustar parâmetros e tipos de retorno para os métodos (linha 4).

Quadro 58 – Classe `Facade_cp`

```
1 public class Facade_cp{
...
2   private A1 attA1 = new A1(<<params>>);
3   private A2 attA2 = new A2(<<params>>);
4   public <<tipo_ret>> MetodoFacade1(<<params>>){
5      attA1.MetodoAA1(attA2.MetodoCA2(<<params>>));
6      attA1.MetodoBA1(<<params>>);
7      attA2.MetodoAA2(<<params>>);
   }
8  public <<tipo_ret>> MetodoFacade2(<<params>>){...}
...
9  public <<tipo_ret>> MetodoFacadeN(<<params>>){...}
}
```

d) Reestruturar o código de todas as classes clientes identificadas no passo **b**. O Quadro 59 apresenta um gabarito para essa classe.

   i. Excluir os objetos que acessam métodos das classes do subsistema (identificadas no passo **a**), linhas 2 a 6.

   ii. Instanciar o objeto do tipo `Facade_cp` (linha 7).

   iii. Invocar o respectivo método (linha 8) da classe `Facade_cp`, que substitui as declarações e chamadas de métodos excluídas em i.

Quadro 59 – Reestruturação da classe `C1`

```
1 public class C1{
...
2   private A1 attA1 = new A1(<<params>>);
3   private A2 attA2 = new A2(<<params>>);
   ...
4   attA1.MetodoAA1(attA2.MetodoCA2(<<params>>));
5   attA1.MetodoBA1(<<params>>);
6   attA2.MetodoAA2(<<params>>);
7   private Facade_cp facade = new Facade_cp(<<params>>);
...
8   facade.MetodoFacade1(<<params>>);
...
}
```

Flyweight

Objetivo: Usar compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

a) Retomar a classe que contém o trecho de código identificado com o padrão *Flyweight* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo). O apresenta um gabarito para essa classe.

Quadro 60 – Classe `X_ap`, padrão *Flyweight*

```
1 public class X_ap{
2   private <<tipo>> obj1 = new <<tipo>><<params>>;
3   private <<tipo>> obj2 = new <<tipo>><<params>>;
4   private <<tipo>> obj3 = new <<tipo>><<params>>;
5   private <<tipo>> obj4 = new <<tipo>><<params>>;
6   private <<tipo>> objN = new <<tipo>><<params>>;
7   obj1.setAtt1(valor);   obj2.setAtt1(valor);   obj3.setAtt1(valor);
8   obj4.setAtt1(valor);   objN.setAtt1(valor);
9   obj1.setAtt3(valor1);  obj2.setAtt3(valor1);  obj3.setAtt3(valor1);
10  obj4.setAtt3(valor1);  objN.setAtt3(valor1);
11  obj1.setAtt2(valorA);  obj2.setAtt2(valorB);
12  obj3.setAtt2(valorC);  obj4.setAtt2(valorD);  objN.setAtt2(valorN);
...
}
```

b) Criar a interface `Flyweight_cp`, que define métodos para a manipulação de estados extrínsecos dos objetos compartilhados. O Quadro 61 apresenta um gabarito para essa classe.

   i. O método `createObj` (linha 2) define a instanciação do objeto a ser compartilhado.

   ii. Os métodos `setAtt1` e `setAtt3` (linhas 3 e 4) são os compartilhados.

Quadro 61 – Classe `Flyweight_cp`

```
1 public interface Flyweight_cp{
2   public <<tipo_ret>> createObj(<<tipo>> <<params>>);
3   public void setAtt1(<<tipo>> valor);
4   public void setAtt3(<<tipo>> valor);
}
```

c) Criar a classe `ConcreteFlyweight_cp`, que implementa a interface `Flyweight_cp` e define, por meio de parâmetros, os dados que são intrínsecos (os quais diferenciam um objeto do outro). O Quadro 62 apresenta um gabarito para essa classe.

   i. Os parâmetros denominados `valor` são os dados intrínsecos (linhas 5 e 6).

Quadro 62 – Classe `ConcreteFlyweight_cp`

```
1 public class ConcreteFlyweight_cp implements Flyweight_cp{
2   private <<tipo>> att1;
3   private <<tipo>> att3;
4   public <<tipo_ret>> createObj(<<tipo>> <<params>>){...}
5   public void setAtt1(<<tipo>> valor){att1 = valor;}
6   public void setAtt3(<<tipo>> valor)(att3 = valor;)
7   public <<tipo_ret>> MetodoN(<<params>>){...}
}
```

d) Criar a classe que representa a fábrica de criação dos *flyweights*. O Quadro 63 apresenta um gabarito para essa classe.

i. Nenhuma classe cliente deve acessar a classe `ConcreteFlyweight_cp` diretamente, isso deve ser feito por meio da fábrica, garantindo o compartilhamento dos objetos.

ii. O método `getFlyweight` retorna uma instância de `ConcreteFlyweight_cp` (linha 4).

iii. O método `getInstance` retorna um objeto do tipo fábrica, `FlyweightFactory_cp` (linha 10).

Quadro 63 – Classe `FlyweightFactory_cp`

```
1 public class FlyweightFactory_cp {
2   private static ConcreteFlyweight_cp fw;
3   private static FlyweightFactory_cp factory =
                                          new FlyweightFactory_cp();
4   public static ConcreteFlyweight_cp getFlyweight() {
5     if(fw != null)
6        return fw;
7     else {
8        fw = new ConcreteFlyweight_cp();
9        return fw;
      }
   }
10   public static FlyweightFactory_cp getInstance(){
11     return factory;
   }
}
```

e) Reestruturar a classe `X_ap`, identificada no passo **a**. O Quadro 64 apresenta o gabarito de reestruturação.

i. Declarar objetos do tipo `Factory_cp` e `FlyweightFactory_cp` (linhas 6 e 7).

ii. Inserir a chamada ao método `getFactory`, da classe `FlyweightFactory_cp` (linha 8).

iii. Excluir os trechos de código em que vários objetos da mesma classe são instanciados (linhas 2 a 5).

iv. No local do código excluído, inserir código que solicita à fábrica a instanciação (linha 9) do objeto, que se torna compartilhado.

v. Utilizar o objeto `flyweight` para acessar os métodos de `ConcreteFlyweight` (linhas 10 e 11).

Quadro 64 – Classe `X_ap`, padrão *Flyweight*

```
1 public class X_ap{
2   private <<tipo>> obj1 = new <<tipo>><<params>>;
3   private <<tipo>> obj2 = new <<tipo>><<params>>;
4   private <<tipo>> obj3 = new <<tipo>><<params>>;
5   private <<tipo>> obj4 = new <<tipo>><<params>>;

6   private FlyweightFactory_cp factory;
7   private Flyweight_cp flyweight;
8   flyweight = factory.getFactory();
9   flyweight.createObj(<<params>>);
...
10  flyweight.setAtt1(<<params>>);
11  flyweight.setAtt3(<<params>>);
...
}
```

Proxy

Objetivo: Fornece um substituto (*surrogate*) ou marcador da localização de outro objeto para controlar o acesso ao mesmo.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

   a) Retomar a classe que contém o trecho de código identificado com o padrão *Proxy* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo). O Quadro 65 apresenta o gabarito para essa classe.

      i. `obj` é o objeto a ser criado sob demanda (linha 2).

      ii. O método `disponivel` (linha 3) avalia se o objeto já deve ser criado ou não. Caso ainda não deva ser criado, um outro objeto deve ser utilizado, substituindo-o (linha 4).

      iii. O método `criar` (linha 5) instancia o objeto no momento adequado.

Quadro 65 – Classe `X_ap`, padrão *Proxy*

```
1 public class X_ap{
2   private <<tipo>> obj = new <<tipo>>(<<params>>);
   ...
3   while(!obj.disponivel()){
4      //substitui objeto por outro
    }
5   obj.Metodo1();
6   obj.Metodo2();
7   obj.criar(<<params>>);
...
}
```

b) Criar a classe abstrata `Subject_cp`, que contém o método `request` a ser herdado e definido pelas classes `Proxy_cp` e `RealSubject_cp`. O Quadro 66 apresenta o gabarito para essa classe.
   i. O único dado a ser alterado nesse gabarito é o valor dos parâmetros (`<<params>>`, linha 2) do método `request`.

Quadro 66 – Classe `Subject_cp`, padrão *Proxy*

```
1 public abstract class Subject_cp {
2   public abstract void request(<<params>>);
}
```

c) Criar a classe `Proxy_cp`. O Quadro 67 apresenta o gabarito para essa classe.
   i. É uma subclasse de `Subject_cp`.
   ii. Instanciar um objeto da classe `RealSubject_cp`, `real` (linha 2).
   iii. Retirar da classe identificada no passo **a** o código que trata da substituição do objeto real (linhas de 3 e 4 do Quadro 65) e inserí-lo no método `request` (linhas 4 e 5 do Quadro 67) da classe `Proxy_cp`.
   iv. Solicitar ao objeto `real` a criação do objeto real, no momento adequado (linha 6).
   v. Realizar ajustes relacionados a parâmetros e bibliotecas.

Quadro 67 – Classe `Proxy_cp`

```
1 public class Proxy_cp extends Subject_cp {
2   private RealSubject_cp real = new RealSubject_cp(<<params>>);
3   public void request(<<params>>){
4      while(!obj.disponivel()){
5         //substitui objeto por outro
       }
6      real.request(<<params>>);
    }
}
```

d) Criar a classe `RealSubject_cp`. O Quadro 68 apresenta o gabarito para essa classe.
   i. É uma subclasse de `Subject_cp`.
   ii. Retirar da classe identificada no passo **a** o código responsável pela criação do objeto real (linhas de 5 a 7 do Quadro 65) e inserí-lo no método `request` (linhas de 4 a 6 do Quadro 68) da classe `Proxy_cp`.
   iii. Realizar ajustes relacionados a parâmetros e bibliotecas.

Quadro 68 – Classe `RealSubject_cp`

```
1 public class RealSubject_cp extends Subject_cp {
2   private <<tipo>> obj = new <<tipo>>(<<params>>);
3   public void request(<<params>>){
```

```
4       obj.Metodo1();
5       obj.Metodo2();
6       obj.criar(<<params>>);
    }
}
```

Categoria: Comportamental

Chain of Responsibility

Objetivo: Evitar o acoplamento do remetente de uma solicitação ao seu receptor, ao dar a mais de um objeto a oportunidade de tratar a solicitação. Encadear os objetos receptores, passando a solicitação ao longo da cadeia até que um objeto a trate.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

   a) Criar a interface `Handler_cp`, que define o método (linha 2) que trata as solicitações. O Quadro 69 apresenta o modelo dessa classe.

Quadro 69 – Classe `Handler_cp`

```
1 public class Handler_cp {
2    private void HandleRequest(<<tipo>> request);
}
```

   b) Retomar a classe que contém o trecho de código identificado com o padrão *Chain of Responsability* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo). O Quadro 70 apresenta um gabarito para essa classe.

Quadro 70 – Classe `X_ap`, padrão *Chain of Responsability*

```
1 public class X_ap {
...
2   if(comando == request1){
3      Metodo2();
4      objK = new Obj();
       ...
5   }else if(comando == request2){
6      objY = new ObjM(valor);
7      Metodo2(valor1);
8      Metodo3();
       ...
9   }else if(...){...}
...
}
```

c) Criar as classes `ConcreteHandler1_cp`, `ConcreteHandler2_cp`, …, `ConcreteHandlerN_cp`, que fazem parte da cadeia de classes que tratam as solicitações. O Quadro 71 e o Quadro 72 apresentam gabaritos para essas classes.

   i. Cada classe da cadeia deve implementar a interface `Handler_cp` (linha 1) e definir o método `HandleRequest` (linha 3).

   ii. Declarar, em cada classe da cadeia, o objeto sucessor, a quem será enviada a solicitação, caso não seja tratada (linha 2).

   iii. Se a classe `ConcreteHandlerN_cp` em questão pode tratar a solicitação, deve fazê-lo (linha 4) conforme descrito no passo **c.iii**; caso contrário, a solicitação é repassada à classe sucessora na cadeia (linha 8).

   iv. Reestruturar a classe `X_ap`: retirar o código responsável por tratar cada tipo de solicitação (linhas 3 e 4, 5 a 9 do Quadro 70) e transferí-lo para as respectivas classes `ConcreteHandler_cp` (linhas 5 e 6 do Quadro 71, linhas 5 a 7 do Quadro 72) criadas com essa finalidade. Esse código deve ser inserido, na classe `ConcreteHandler_cp`, no trecho que trata a solicitação (linha 4)

Quadro 71 – Classe `ConcreteHandler1_cp`

```
1 public class ConcreteHandler1_cp implements Handler_cp {
...
2   private ConcreteHandler2_cp sucessor;
3   public void HandleRequest(<<tipo>> request){
4      if(request == comandoX){
5          Metodo2();
6          objK = new Obj();
       ...
7      }else{
8         sucessor.HandleRequest(request);
       }
   }
}
```

Quadro 72 – Classe `ConcreteHandler2_cp`

```
1 public class ConcreteHandler2_cp implements Handler_cp {
...
2   private ConcreteHandler3_cp sucessor;

3   public void HandleRequest(<<tipo>> request){
4      if(request == comandoY){
5        objY = new ObjM(valor);
6        Metodo2(valor1);
7        Metodo3();
         ...
8      }else{
9        sucessor.HandleRequest(request);
       }
   }
}
```

d) Declarar na classe `X_ap` um objeto de uma das subclasses de `Handler_cp` que deve ser o primeiro a tratar a solicitação quando ela for disparada. O Quadro 73 apresenta o gabarito para essa modificação.

Quadro 73 – Modificação na classe `X_ap`, padrão *Chain of Responsability*

```
1 public class X_ap {
...
2  private ConcreteHandler1_cp handler1 = new ConcreteHandler1_cp();
   ...
3  handler1.HandleRequest(request);
   ...
}
```

Command

Objetivo: Encapsular uma solicitação como um objeto, desta forma permitindo parametrizar clientes com diferentes solicitações, enfileirar ou fazer o registro (*log*) de solicitações e suportar operações que podem ser feitas.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

   a) Retomar a classe que contém o trecho de código identificado com o padrão *Command* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo). O Quadro 74 apresenta um gabarito para essa classe.

Quadro 74 – Classe `X_ap`, padrão *Command*

```
1 public class X_ap implements ActionListener{
2   private <<tipo>> obj1 = new <<tipo>>(<<params>>);
3   private <<tipo>> obj2 = new <<tipo>>(<<params>>);
...
4   obj1.addActionListener(this);
5   obj2.addActionListener(this);
...
6   public void actionPerformed(ActionEvent event){
7      if(event.getSource() == obj1)
8         Metodo1();
9      else if(event.getSource() == obj2)
10        Metodo2();
11     else if(...){...}
       ...
   }
}
```

   b) Criar na classe `X_ap` as classes que realizam as ações (*commands*) quando os objetos são acionados.

i. Essas classes devem implementar a interface `ActionListener` e, conseqüentemente, o método `actionPerformed` (linhas 6, 7, 9 e 10).

ii. O conteúdo do método `actionPerformed` dessas classes deve ser extraído das declarações `if/else` presentes no método `actionPerformed` (linha 6 do Quadro 74) da classe `X_ap`.

iii. Instanciar as classes criadas no passo **b.ii** para cada ação que deve ser executada, no objeto responsável pela ação (linhas 4 e 5).

iv. Excluir o método `actionPerformed` da classe `X_ap`.

c) Excluir o código `implements ActionListener` da classe `X_ap` (linha 1).

Quadro 75 – Modificações na classe `X_ap`, padrão *Command*

```
1 public class X_ap{
2   private <<tipo>> obj1 = new <<tipo>>(<<params>>);
3   private <<tipo>> obj2 = new <<tipo>>(<<params>>);
...
4   obj1.addActionListener(new Classe1(<<params>>));
5   obj2.addActionListener(new Classe2(<<params>>);
...
6   class Classe1 implements ActionListener{
7       public void actionPerformed(ActionEvent e){
8          Metodo1();
        }
    }
9    class Classe1 implements ActionListener{
10      public void actionPerformed(ActionEvent e){
11          Metodo2();
       }
    }
      ...
   }
}
```

Interpreter

Objetivo: Dada uma linguagem, definir uma representação para a sua gramática juntamente com um interpretador que usa representação para interpretar sentenças da linguagem.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

a) Criar a interface `AbstractExpression_cp`, que define a operação de interpretação da expressão. O Quadro 76 apresenta o gabarito referente a essa classe.

i. `<<tipo>>` é o tipo de retorno do método (linha 2).

ii. O nome do método, `Interpret` (linha 2), não deve ser substituído, bem como o tipo e nome de seu parâmetro, `Context` e `c`, respectivamente.

Quadro 76 – Classe `AbstractExpression_cp`

```
1 public interface AbstractExpression_cp{
2   public <<tipo_ret>> Interpret(Context c);
}
```

b) Criar a classe abstrata `TerminalExpression_cp`. O Quadro 77 apresenta o gabarito referente a essa classe.

i. Essa classe implementa a interface `AbstractExpression_cp` (linha 1).

ii. O construtor dessa classe deve receber como parâmetro o valor do símbolo não-terminal (linha 3).

iii. Essa classe implementa uma operação de interpretação de acordo com os símbolos terminais definidos para a gramática (linha 5). Nesse caso, solicita ao objeto do tipo `Context` que obtenha o valor do símbolo não-terminal.

Quadro 77 – Classe `TerminalExpression_cp`

```
1 public class TerminalExpression_cp implements AbstractExpression_cp {
2  private String var;
3  public TerminalExpression(String v) {
4     var = v;
  }
5  public <<tipo_ret>> Interpret(Context c) {
6     return c.getValue(var);
  }
}
```

c) Criar as classes `NonTerminalExpression_cp`. O Quadro 78 apresenta o gabarito referente a essas classes.

i. Essas classes implementam a interface `AbstractExpression_cp` (linha 1).

ii. São declarados dois objetos privados do tipo `Expression_cp` (linhas 2 e 3), a serem tratados pelo método `Interpret`.

iii. Essas classes implementam operações de interpretação de acordo com os símbolos não-terminais definidos para a gramática (linha 7). Por isso, deve haver uma classe dessas para tratar cada símbolo não-terminal.

iv. Os conteúdos dos métodos `Interpret` (linha 7) dessas classes devem ser retirados da classe `X_ap`. Esse é o código responsável por realizar a operação/avaliação dos dados da expressão.

Quadro 78 – Classe `NonTerminalExpression_cp`

```
1 public class NonTerminalExpression1_cp implements
                                          AbstractExpression_cp {
2   private Expression_cp left;
3   private Expression_cp right;
4   public NonTerminalExpression1_cp(Expression_cp left,
```

```
                                                    Expression_cp right){
5     this.left = left;
6     this.rigth = right;
  }
7  public <<tipo_ret>> Interpret(Context c) {
      //operações adequadas envolvendo os objetos left e right
  }
}
```

d) Criar a classe `Context_cp`, que contém informações globais ao interpretador. O Quadro 79 apresenta o código referente a essa classe.

   i. No construtor, a expressão recebida como parâmetro (linha 3) é transferida para um vetor.

   ii. O método `getExpression` (linha 8) retorna um objeto do vetor definido no passo **d.i**.

Quadro 79 – Classe `Context_cp`, padrão *Interpreter*

```
1 public class Context_cp {
2   Vector expression = new Vector();

3   public Context(String expr) {
4     int i = 0;
5     while(expr.length() > i){
6        expression.add(i, expr.substring(i));
7        i++;
      }
   }
8  public <<tipo_ret>> getExpression() {
9     return (<<tipo_ret>>)expression.get(0);
   }
}
```

e) Retomar a classe que contém o trecho de código identificado com o padrão *Interpreter* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo).

f) Reestruturar a classe `X_ap`. Utilizar o gabarito apresentado no Quadro 80.

   i. Instanciar um objeto do tipo `Context_cp` (linha 2).

   ii. Criar o método `isTerminal`, que recebe um objeto do tipo `String` como parâmetro (parte da expressão a ser interpretada) e retorna `true` quando o objeto é um terminal e `false`, caso contrário. Nesse método, `<<terminalA>>` e `<<terminalB>>` (linha 3) são os valores dos terminais definidos pela gramática.

   iii. Definir o método `getNonTerminalExpression`, cujo gabarito é apresentado da linha 8 à 14. Ele recebe como parâmetros um objeto `String` e dois

`Expression_cp`. Esses últimos devem ser transmitidos como parâmetros aos construtores das classes `NonTerminalExpression_cp`, a serem instanciadas. Nesse método deve haver tantas comparações (linhas 9, 11 e 13) quantos forem os símbolos não-terminais da gramática.

iv. Definir o método (`MetodoVarredura`, linha 16) para realizar a varredura dos símbolos da expressão (linha 8), que deve receber a expressão como parâmetro. Sua lógica deve ser extraída da própria classe `X_ap`, do código que realiza a operação de tratamento de cada símbolo da expressão.

v. Definir o método `Interpret` (linha 17), que invoca os métodos: de varredura dos símbolos da expressão (linha 18) e de construção da árvore sintática abstrata (linha 19).

vi. Definir método `buildTree` (linhas 21 a 33), que constrói a árvore sintática abstrata. Não deve ser utilizado como mostrado (sem substituições). Caso a classe `X_ap` apresente código responsável por construir a árvore sintática, esse deve ser excluído.

Quadro 80 – Classe `X_ap`, padrão *Interpreter*

```
1 public class X_ap{
2   private Context_cp context = new Context_cp();
...
3   private boolean isTerminal(String str) {
4       if ((str.equals(<<terminalA>>))||(str.equals(<<terminalB>>))
5          return true;
6       else
7          return false;
    }
8    private NonTerminalExpression getNonTerminalExpression(String
                            operation, Expression l, Expression r) {
9      if (operation.trim().equals(<<operacaoA>>)) {
10        return new NonTerminalExpression1_cp(l, r);
       }
11     if (operation.trim().equals(<<operacaoA>>)) {
12        return new NonTerminalExpression2_cp (l, r);
       }
13     if (operation.trim().equals(<<operacaoC>>)) {
14        return new NonTerminalExpression3_cp (l, r);
       }
       ...
15     return null;
    }
16    public String MetodoVarredura(String str){...}
17    public <<tipo_ret>> Interpret() {
18       String pfExpr = <<MetodoVarredura>>(expression);
19       Expression_cp rootNode = buildTree(pfExpr);
20       return rootNode.Interpret(context);
    }
21    private Expression buildTree(String expr) {
22       Stack s = new Stack();
23       for (int i = 0; i < expr.length(); i++) {
```

```
24         String currChar = expr.substring(i, i + 1);
25            if (isOperator(currChar) == false) {
26               Expression_cp e = new TerminalExpression_cp (currChar);
27               s.push(e);
28             } else {
29               Expression_cp r = (Expression_cp) s.pop();
30               Expression_cp l = (Expression_cp) s.pop();
31               Expression_cp n = getNonTerminalExpression(currChar, l,
                                                                    r);
32               s.push(n);
               }
         }
33       return (Expression_cp) s.pop();
     }
24
...
}
```

Iterator

Objetivo: Fornecer um meio de acessar, seqüencialmente, os elementos de um objeto agregado sem expor a sua representação subjacente.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.
   a) Retomar a classe que contém o trecho de código identificado com o padrão *Iterator* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo). O Quadro 81 apresenta o gabarito para essa classe
      i. Deste passo em diante na refatoração com o padrão *Command*, quando o objeto do tipo `Vector` for mencionado, também pode ser utilizado um objeto do tipo `Hashtable`.
      ii. `v` é o vetor a ser acessado (linha 3).
      iii. O acesso ao vetor `v` pode ser feito de maneira semelhante à mostrada na linha 6.

Quadro 81 – Classe `X_ap`, padrão *Iterator*

```
1 import java.util.Vector;
...
2 public class X_ap{
...
3   private Vector v = new Vector();
4   for(int i = 0; i < v.size(); i++){
5      //operações de obtenção dos dados do vetor
6      v.get(i);
    }
}
```

b) Reestruturar a classe `X_ap`. Utilizar o gabarito apresentado no Quadro 82.

i. Utilizar a biblioteca `Enumeration` (linha 1) de Java (2005), que define a funcionalidade do *Iterator*. Tanto a classe `Vector` quanto `Hashtable` possuem métodos (`elements` e `keys`, respectivamente) que retornam uma coleção do tipo `Enumeration`.

ii. Declarar um objeto do tipo `Enumeration` (linha 3) e não instanciá-lo, pois `Enumeration` é uma interface.

iii. Adicionar os elementos do vetor `v` na coleção `enum` (linha 5).

iv. Substituir o código de acesso ao vetor (linhas 4 a 6 do Quadro 81) pelo acesso à coleção `enum` (linhas 6 a 8 do Quadro 82).

Quadro 82 – Modificações na classe `X_ap`, padrão *Iterator*

```
1 import java.util.Enumeration;
...
2 public class X_ap{
...
3   private Vector v = new Vector();
4   private Enumeration enum;
5   enum = v.elements();
...
6   while(enum.hasMoreElements()){
7      //operações de obtenção dos dados da coleção
8      enum.nextElement();
    }
}
```

Mediator

Objetivo: Definir um objeto que encapsula a forma como um conjunto de objetos interage. O *Mediator* promove o acoplamento fraco ao evitar que os objetos se refiram uns aos outros explicitamente e permite variar suas interações independentemente.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

a) Retomar a classe que contém o trecho de código identificado com o padrão *Mediator* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo). O Quadro 83 apresenta o gabarito para essa classe.

i. O grupo de objetos que interagem é composto pelos objetos `obj1`, `obj2` e `obj3`.

Quadro 83 – Classe `X_ap`, padrão *Mediator*

```
1 public class X_ap{
   //declaração de objetos da classe
2   private <<tipo>> obj1 = new <<tipo>>(<<params>>);
3   private <<tipo>> obj2 = new <<tipo>>(<<params>>);
4   private <<tipo>> obj3 = new <<tipo>>(<<params>>);
5   obj1.addActionListener(this);
6   obj2.addActionListener(this);
7   obj3.addActionListener(this);
...
8  public void actionPerformed(ActionEvent e){
9     if(e.getSource() == obj1){
10        obj1.setEnabled(true);
11        obj1.setText("texto texto");
12        obj2.setVisible(false);
          ...
13     }else if(e.getSource() == obj2){
          ...
14        obj3.setText("texto texto");
15        obj2.setVisible(true);
16        obj1.setText("texto texto");
17     }else if(e.getSource() == obj3){
          ...
       }
    }
...
}
```

b) Criar a interface `Mediator_cp`, que define os métodos para a comunicação entre os objetos a serem mediados (`obj1`, `obj2` e `obj3` do Quadro 83). O Quadro 84 apresenta o gabarito para essa classe.

   i. O método `initializeObjs` (linha 2) é responsável por solicitar a criação e instanciação dos objetos e pode receber parâmetros, sendo que um objeto da classe `ConcreteMediator_cp` é parâmetro obrigatório.

   ii. Os métodos `set`, das linhas 3, 4 e 5, devem ser criados para todos os objetos que são mediados.

   iii. Os métodos das linhas 6 a 14 foram definidos de acordo com o gabarito do Quadro 83 e são ilustrativos, devem ser criados de acordo com as operações a serem realizadas sobre os objetos controlados pelo mediador.

Quadro 84 – Classe `Mediator_cp`

```
1 public interface Mediator_cp{
2   public void initializeObjs(<<params>>, ConcreteMediator_cp
                                                        mediator);
3   public void setObj1(<<tipo>> obj1);
4   public void setObj2(<<tipo>> obj2);
5   public void setObj2(<<tipo>> obj2);
6   public void setVisibleObj1(String visible);
7   public void setVisibleObj2(String visible);
8   public void setVisibleObj3(String visible);
9   public void setEnabledObj1(String enabled);
```

```
10  public void setEnabledObj2(String enabled);
11  public void setEnabledObj3(String enabled);
12  public void setTextObj1(String text);
13  public void setTextObj2(String text);
14  public void setTextObj3(String text);
...
}
```

c) Criar a classe `ConcreteMediator_cp`. O Quadro 85 apresenta o gabarito para essa classe

   i. Essa classe implementa a interface `Mediator_cp` (linha 1).

   ii. Declarar e instanciar todos os objetos a serem mediados (linhas de 2 a 4).

   iii. O método `initializeObjs` (linha 5) seta o *mediator* recebido como parâmetro nos objetos declarados no passo **c.ii** (linhas de 6 a 8).

   iv. O método `initializeObjs` solicita às respectivas classes a criação visual do objeto (linhas de 9 a 11).

   v. O método `initializeObjs` seta todos os objetos criados no passo **c.iv** (linhas de 12 a 14) e, por isso, deve-se implementar todos os métodos `set` (linhas de 13 a 16) dos objetos a serem mediados.

   vi. Implementar os métodos definidos pela interface `Mediator_cp` (linhas 21 a 30).

Quadro 85 – Classe `ConcreteMediator_cp`

```
1 public class ConcreteMediator_cp extends implements Mediator_cp{
2   private <<tipo>> obj1 = new ConcreteColleagueObj1_cp(<<params>>);
3   private <<tipo>> obj2 = new ConcreteColleagueObj2_cp(<<params>>);
4   private <<tipo>> obj3 = new ConcreteColleagueObj3_cp(<<params>>);
...
5   public void initializeObjs(<<params>>, ConcreteMediator_cp
                                                          mediator){
6     obj1.setMediator(mediator);
7     obj2.setMediator(mediator);
8     obj3.setMediator(mediator);
...
9     obj1.createObj1(<<params>>);
10    obj2.createObj2(<<params>>);
11    obj3.createObj3(<<params>>);
...
12    setObj1(obj1);
13    setObj2(obj2);
14    setObj3(obj3);
   }
15  public void setObj1(<<tipo>> obj1){
16     this.obj1 = obj1;
    }
17  public void setObj2(<<tipo>> obj2){
18     this.obj2 = obj2;
    }
19  public void setObj3(<<tipo>> obj3){
```

```
20      this.obj3 = obj3;
    }
...
21  public void setVisibleObj1(String visible){
22      obj1.setVisibleObj(visible);
    }
23  public void setVisibleObj2(String visible){
24      obj2.setVisibleObj(visible);
    }
25  public void setVisibleObj3(String visible){
26      obj3.setVisibleObj(visible);
    }
27  public void setEnabledObj1(String enabled){
28      obj1.setEnabledObj(enabled);
    }
    ...
29  public void setTextObj1(<<tipo>> obj, String text){
30    obj1.setText(text);
   }
   ...
}
```

d) Criar as classes dos objetos a serem mediados. O Quadro 86 apresenta o gabarito para essas classes.

   i. São definidas de acordo com os objetos da classe `X_ap`.

   ii. Podem ser subclasses de outra.

   iii. Devem declarar o objeto o qual definem (linha 3).

   iv. Devem definir o método `setMediator` conforme apresentado na linha 5.

   v. Devem definir o método de criação, `createObj1` no gabarito, linha 7.

   vi. Devem definir os métodos de configuração que são aplicáveis ao objeto, definidos pela interface `Mediator_cp`, linhas de 10 a 12.

Quadro 86 – Classe `ConcreteColleagueObj1_cp`

```
1  public class ConcreteColleagueObj1_cp {
2   private ConcreteMediator1_cp mediator;
3   private <<tipo>> obj1;
   // declarações de outros objetos da classe
4   public ConcreteColleague1_cp(<<params>>) {...}
5   public void setMediator(ConcreteMediator_cp mediator){
6      this.mediator = mediator;
    }
7   public void createObj1() {
8      obj1 = new <<tipo>>(<<params>>);
       obj1.addActionListener(this);
9      //operações sobre a criação do objeto, como configuração
       //de cor, formato, eventos, etc.
    }
10   public void setVisibleObj1_cp(boolean value){
11      obj1.setVisible(value);
     }
12   public void setEnabledObj1_cp(boolean value){
11      obj1.setEnabled(value);
     }
```

```
13   public void actionPerformed(ActionEvent e){
14     obj1.setEnabled(true);
15     obj1.setText("texto texto");
16     mediator.setVisibleObj2(false);
     }
}
```

e) Reestruturar a classe `X_ap`. Utilizar o gabarito apresentado no Quadro 87.

i. Retirar todas as declarações e instanciações dos objetos que são controlados pelo *Mediator* (linhas de 2 a 5). Esses são os objetos para os quais foram criadas classes, como a definida no passo **d**.

ii. Declarar um objeto do tipo `ConcreteMediator_cp` (linha 5).

iii. Inserir a solicitação da execução do método `initialize`, para que os objetos retirados no passo **f** sejam instanciados e inseridos adequadamente (linha 6). Sugere-se que essa solicitação seja realizada no construtor da classe `X_ap`.

iv. Criar o método `setMediator` (linha 7).

Quadro 87 – Classe `X_ap`, padrão *Mediator*

```
1 public class X_ap{    ...
2   private <<tipo>> obj1 = new <<tipo>>(<<params>>);
3   private <<tipo>> obj2 = new <<tipo>>(<<params>>);
4   private <<tipo>> obj3 = new <<tipo>>(<<params>>);
5   private ConcreteMediator_cp mediator =
                                  new ConcreteMediator_cp(<<params>>);
    ...
6   mediator.initialize(<<params>>, mediator);
    ...
7   public void setMediator(ConcreteMediator_cp mediator){
8      this.mediator = mediator;
    }
}
```

Memento

Objetivo: Sem violar o encapsulamento, capturar e externalizar um estado interno de um objeto, de maneira que o objeto possa ser restaurado para este estado mais tarde.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

a) Criar a classe `Memento_cp`. Utilizar o gabarito apresentado no Quadro 88.

i. O objeto que tem seu estado interno armazenado é do tipo `ArrayList`. Desta forma, o tipo do objeto `state` deve ser alterado, utilizando-se um tipo igual ao do objeto que deve ter ser estado interno armazenado. Deste passo em

diante na refatoração com o padrão *Memento*, quando o objeto `ArrayList` for mencionado, subentende-se que outros tipos podem ser utilizados. Por isso, podem ser necessárias algumas modificações no construtor da classe `Memento_cp` e no método `setMemento`, da classe `Originator_cp` (Quadro 89).

Quadro 88 – Classe `Memento_cp`

```
1 import java.util.ArrayList;
2  public class Memento_cp {
3     private ArrayList state = new ArrayList();
4     public Memento_cp(ArrayList state) {
         this.state.addAll(state);
      }
5     public ArrayList getState(){
6        return state;
      }
}
```

b) Criar a classe `Originator_cp`. O Quadro 89 apresenta o código referente a essa classe.

Quadro 89 – Classe `Originator_cp`

```
1 import java.util.ArrayList;
2 public class Originator_cp {
3   private ArrayList list = new ArrayList();
4   public ArrayList setMemento(Object object){
5      if(object instanceof Memento_cp){
6         Memento_cp memento = (Memento_cp)object;
7         list.addAll(memento.getState());
       }
8      return list;
    }
9   public Memento_cp createMemento(ArrayList list){
10     return new Memento_cp(list);
    }
}
```

c) Retomar a classe que contém o trecho de código identificado com o padrão *Memento* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo).

d) Reestruturar a classe X_ap. Utilizar o gabarito apresentado no Quadro 90.

   i. Declarar e instanciarum objeto do tipo `Originator_cp` (linha 3).

   ii. Declarar um objeto do tipo `Memento_cp` (linha 4).

   iii. Declarar um objeto do tipo `ArrayList` (linha 6).

   iv. Localizar os trechos de código que armazenam os estados internos do objeto em questão e substituir cada um pelas linhas 4 e 5.

- O objeto `list` é o que tem seu estado interno armazenado, por isso é passado como parâmetro ao método `createMemento` (linha 7).

v. Localizar os trechos de código em que o objeto em questão é restaurado para algum estado anterior, armazenado e substituir cada um pela linha 9.

- Na linha 9, `i` é a posição da lista em que se encontra o estado que deve ser resgatado ao objeto `list`.

Quadro 90 – Classe `X_ap`, padrão *Memento*

```
1 import java.util.ArrayList;
2 public class X_ap{
3    private Originator_cp originator = new Originator_cp();
4    private Memento_cp m;
5    private ArrayList list = new ArrayList();
6    private ArrayList listMemento = new ArrayList();
...
7    m = originator.createMemento(list);
8    listMemento.add(m);
     //operações com o objeto list e novo armazenamento
9    list = originator.setMemento(listMemento.get(i));
...
}
```

Observer

Problema: Definir uma dependência um-para-muitos entre objetos, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

a) Criar a classe `ConcreteSubject_cp`, que representa o objeto que será "observado", ou seja, disponibilizado para observação. O Quadro 91 apresenta o gabarito para essa classe.

i. Essa classe deve ser herdeira da classe `Observable` (linha 2), da biblioteca Java (2005).

ii. A cada alteração do objeto que é observado, seu estado deve ser mudado e seus observadores devem ser notificados através da chamada dos métodos `setChanged` e `notifyObservers` (linhas 9 e 10, respectivamente), definidos em `Observable`.

iii. `<<tipo>>` é o tipo do objeto e `<<obj_observado>>` é o seu nome (linha 3). Essa declaração é utilizada em outros passos da refatoração com esse padrão.

Quadro 91 – Classe `ConcreteSubject_cp`

```
1 import java.util.Observable;
2 public class ConcreteSubject_cp extends Observable {
3    private <<tipo>> <<obj_observado>>;
4    public ConcreteSubject_cp(<<tipo>> <<obj_observado>>){
5       this.<<obj_observado>> = <<obj_observado>>;
     }
6    public <<tipo_objeto>> get<<obj_observado>> () {
7       return <<obj_observado>>;
    }
8    public void atualiza<<obj_observado>> () {
       //alteração do objeto observado
9      setChanged();
10     notifyObservers(new <<tipo_objeto>>(<<obj_observado>>));
     }
}
```

b) Criar a classe `ConcreteObserver_cp`, que atualiza os dados quando há mudanças do objeto controlado pelo padrão (objeto "observado"). O Quadro 92 exibe o gabarito correspondente a essa classe.

   i. Essa classe deve implementar a interface `Observer` (linha 3) e, conseqüentemente, definir o método `update` (linha 7), responsável por engatilhar o mecanismo de atualização dos observadores.

Quadro 92 – Classe `ConcreteObserver_cp`

```
1 import java.util.Observable;
2 import java.util.Observer;
3 public class ConcreteObserver_cp implements Observer {
4    private <<tipo_objeto>> <<obj_observado>>;
5    public ConcreteObserver_cp() {
6       //inicializar o objeto
     }
7    public void update(Observable obj, Object arg) {
8       if (arg instanceof <<tipo>>){
9          //código correspondente à atualização do objeto
        }
     }
}
```

c) Retomar a classe que contém o trecho de código identificado com o padrão *Observer* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo).

d) Reestruturar a classe `X_ap`. Utilizar o gabarito apresentado no Quadro 93.

   i. Criar os objetos observável e observador, denominados `objSubject` e `objObserver` (linhas 2 e 3).

   ii. Instanciar os objetos definidos no passo **d** e adicionar o objeto observador ao observado (linha 6).

Quadro 93 – Classe `X_ap`, padrão *Observer*

```
1 public class X_ap{
2   private ConcreteSubject_cp objSubject;
3   private FunctionsObserver_cp objObserver;
...
4   objSubject = new ConcreteSubject();
5   objObserver = new ConcreteObserver();
6   objSubject.addObserver(objObserver);
}
```

e) Substituir o código identificado como o da existência do padrão no passo 3 da etapa 1 pelos métodos da classe observada (métodos `get` e `set`, criados no passo **a**. As notificações de mudanças do objeto observado são feitas automaticamente pelo método `update`.

State

Objetivo: Permite a um objeto alterar seu comportamento quando o seu estado interno muda. O objeto parecerá ter mudado sua classe.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

a) Criar a classe `State_cp`, para definir os métodos que representam os possíveis estados dos objetos. O Quadro 94 apresenta o gabarito para essa interface.

i. Deixar que as subclasses definam o corpo dos métodos.

Quadro 94 – Interface `State_cp`

```
1 public class State_cp {
2    public <<tipo>> HandleMethod1(<<params>>){ }
3    public <<tipo>> HandleMethod2(<<params>>){ }
...
}
```

b) Criar a classe `Context_cp`, que define os interesses dos clientes. O Quadro 95 apresenta o gabarito para essa interface.

i. Declarar um objeto do tipo `State_cp`, que define o estado corrente (linha 2).

ii. No construtor, definir qual é o estado inicial (linha 4).

iii. Criar os métodos `setState` (linha 5) e `getState` (linha 8).

iv. Criar os métodos que solicitam ao estado atual a execução da ação (linhas de 9 a 12).

Quadro 95 – Classe `Context_cp`, padrão *State*

```
1 public class Context_cp {
2   private State_cp currentState;
3   public Context_cp() {
4       setState(new State1());
   }
5   public void setState(State_cp state){
6      currentState = state;
    }
7   public State_cp getState(){
8      return currentState;
    }
9   public void Method1() {
10     currentState.HandleMethod1(this);
    }
11  public void Method2() {
12     currentState.HandleMethod2(this);
    }
...
}
```

c) Retomar a classe que contém o trecho de código identificado com o padrão *State* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo). O Quadro 96 apresenta o gabarito para essa classe.

   i. Quando há um evento envolvendo determinado objeto, verifica-se o estado (linhas 9, 11, 15 e 17), em seguida, muda-se o estado vigente (linhas 10, 12, 16 e 18).

Quadro 96 – Classe `X_ap`, padrão *State*

```
1 public class X_ap{
2   private <<tipo>> obj = new <<tipo>>(<<params>>);
3   private <<tipo>> button1 = new <<tipo>>(<<params>>);
4   private <<tipo>> button2 = new <<tipo>>(<<params>>);
5   button1.addActionListener(this);
6   button2.addActionListener(this);
...
7   public void actionPerformed(ActionEvent e){
8      if(e.getSource() == button1){
9         if(obj.getState() == state1){
10           obj.setState(state2);
11        }else if(obj.getState == state2){
12           obj.setState(state1);
13        }else(...)
14    }else if(e.getSource() == button2){
15        if(obj.getState() == state1){
16           obj.setState(state1);
17        }else if(obj.getState == state2){
18           obj.setState(state1);
19        }else(...)
20    }
21    else(...){...}
...
}
```

d) Criar as subclasses de `State_cp`, que implementam os comportamentos definidos pela superclasse. O Quadro 97 e o Quadro 98 apresentam gabaritos para duas dessas classes.

   i. O conteúdo dos métodos deve ser extraído do comportamento da classe `X_ap`.

Quadro 97 – Classe `State1_cp`

```
1 public class State1_cp extends State_cp{
2   public State1(<<params>>){...}
3   public <<tipo>> HandleMethod1(Context_cp context){
4      context.setState(new State2(<<params>>));
    }
5   public <<tipo>> HandleMethod2(Context_cp context){
6      context.setState(new State1(<<params>>));
    }
}
```

Quadro 98 – Classe `State2_cp`

```
1 public class State2_cp extends State_cp{
2   public State2(<<params>>){...}
    public <<tipo>> HandleMethod1(Context_cp context){
4      context.setState(new State1(<<params>>));
    }
5   public <<tipo>> HandleMethod2(Context_cp context){
6      context.setState(new State1(<<params>>));
    }
}
```

e) Reestruturar na classe `X_ap` o trecho de código referente ao apresentado no passo **c**. O Quadro 99 apresenta o gabarito para essa classe.

   i. Criar um objeto do tipo `Context_cp` (linha 7).

   ii. Nos trechos de código em que houver obtenção do estado atual (linhas 10, 12, 16 e 18), substituir pelo objeto `context`. Após a decisão de qual ação tomar, utilizar o `context` novamente para que o estado correto seja usado (linhas 11, 13, 17 e 19).

Quadro 99 – Classe `X_ap`, padrão *State*

```
1 public class X_ap{
...
2   private <<tipo>> obj = new <<tipo>>(<<params>>);
3   private <<tipo>> button1 = new <<tipo>>(<<params>>);
4   private <<tipo>> button2 = new <<tipo>>(<<params>>);
5   button1.addActionListener(this);
6   button1.addActionListener(this);
7   private Context_cp context = new Context_cp(<<params>>);
...
8   public void actionPerformed(ActionEvent e){
9      if(e.getSource() == button1){
10        if(context.getState() == state1){
11           context.Method2();
12        }else if(obj.getState == state2){
13           context.Method1();
```

```
14          }else(...)
15      }else if(e.getSource() == button2){
16          if(obj.getState() == state1){
17             context.Method1();
18          }else if(obj.getState == state2){
19             context.Method1();
20           }else(...)
21      }
22      else(...){...}
...
}
```

Strategy

Objetivo: Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. *Strategy* permite que o algoritmo varie independentemente dos clientes que o utilizam.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.

a) Criar a interface `Strategy_cp`, que define os métodos de todos os algoritmos suportados. O Quadro 100 apresenta o gabarito para essa interface.

i. Caso o sistema já possua uma estrutura de herança, em que uma superclasse ou interface determina quais são os métodos das subclasses, essa deve ser a interface `Strategy`, apenas deve-se renomeá-la com a extensão `_ap`.

Quadro 100 – Classe `Strategy_cp`

```
public interface Strategy_cp {
      public <<tipo>> Metodo1(<<params>>);
...
}
```

b) Criar a classe `Context_cp`. O Quadro 101 apresenta um gabarito para essa classe.

i. Declara um objeto `Strategy_cp` (linha 2).

ii. Seu construtor garante que o objeto `strategy` não é nulo (linha 4).

iii. Define-se tantos métodos (linhas 5 e 7) quantas forem as estratégias (subclasses de `Strategy_cp`), que atribuem a estratégia escolhida ao objeto `strategy`.

iv. Implementar o método definido pela interface `Strategy_cp`, invocando a funcionalidade através do objeto `strategy`, que deve possuir a estratégia escolhida (linha 10).

Quadro 101 – Classe `Context_cp`, padrão *Strategy*

```
1 public class Context_cp {
2   private Strategy_cp strategy;
3   public Context_cp(<<params>>){
      //garante que o objeto strategy não é nulo
4     setMetodo1(<<params>>);
    }
5   public void setConcreteStrategy1(<<params>>){
6       strategy = new ConcreteStrategy1_cp(<<params>>);
    }
7   public void setConcreteStrategy2(<<params>>){
8       strategy = new ConcreteStrategy2_cp(<<params>>);
    }
9   public <<tipo>> Metodo1(<<params>>){
10      strategy.Metodo1(<<params>>);
    }
...
}
```

c) Criar as subclasses de `Strategy_cp`. Utilizar o gabarito apresentado no Quadro 102.
   i. Caso o passo **a.i** tenha sido executado, não é necessário criar as subclasses de Strategy_cp, pois elas já existem na estrutura de herança do sistema original.
   ii. Caso o código não apresente herança, ou seja, todas as estratégias estão na mesma classe, difundidas, deve-se separá-las em subclasses de `Strategy_cp`.

Quadro 102 – Classe `ConcreteStrategy1_cp`

```
public class ConcreteStrategy1_cp implements Strategy_cp {
   //outros métodos necessários à classe
   public <<tipo>> Metodo1(<<params>>){
      //algoritmo que define a estratégia 1
   }
...
}
```

d) Retomar a classe que contém o trecho de código identificado com o padrão *Strategy* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo).

e) Reestruturar a classe `X_ap`. O Quadro 103 apresenta o gabarito para essa classe.
   i. Excluir o código referente às estratégias, caso o passo **c.ii** tenha sido executado.
   ii. Declarar um objeto do tipo `Context_cp` (linha 2).
   iii. Localizar o código que decide qual das estratégias deve ser escolhida. Substituir o código pelo apresentado na linha 4, em que o objeto `strategy` solicita a configuração da estratégia escolhida.

Quadro 103 – Modificações na classe `X_ap`, padrão *Strategy*

```
1 public class X_ap{
...
2   private Context_cp context = new Context_cp(<<params>>);
...
3      if (<<teste estratégia 1>>)
4         context.setConcreteStrategy1(<<params>>);
5      else if (<<teste estratégia 2>>)
6         context.setConcreteStrategy2(<<params>>);
7      else(...)
...
}
```

Template Method

Objetivo: Definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para subclasses. *Template Method* permite que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo.

Solução:

1. Para cada passo apresentado a seguir, atualizar o modelo de classes obtido no passo 2 da etapa 1.
   a) Retomar a classe que contém o trecho de código identificado com o padrão *Template Method* (classe `X`, por exemplo) no passo 3 da etapa 1 e renomeá-la, adicionando ao seu final a extensão `_ap` (classe `X_ap`, por exemplo).
   b) Identificar o trecho de código candidato a ser implementado em uma subclasse de `X_ap`. O Quadro 104 apresenta o gabarito para essa classe.
      i. Deve haver partes invariantes no código, que representam o esqueleto do algoritmo, ou seja, o *Template Method* (linha 2).
      ii. As partes variantes do algoritmo são implementadas pelos métodos `Metodo1` e `Metodo2` (linhas 3 e 4). Esses dois métodos são candidatos a comporem subclasses de `X_ap`.
      iii. Não é obrigatório que as partes variantes do algoritmo estejam em métodos, é possível que o código variante esteja inserido diretamente e, nesse caso, todo ele deve ser transferido para subclasse de `X_ap`.

Quadro 104 – Classe `X_ap`, padrão *Template Method*

```
1 public class X_ap{
...
2   public <<tipo>> Metodo(<<params>>){
       //declarações invariantes
3      Metodo1(<<params>>);
       //declarações invariantes
4      Metodo2(<<params>>);
       //declarações invariantes
```

```
    }
}
```

c) Criar as subclasses de `X_ap`, `ConcreteX_cp`, para todas as partes variantes do algoritmo, identificadas no passo **b.ii**. O Quadro 105 apresenta o gabarito para essa classe.

Quadro 105 – Classe `ConcreteMetodo1X_cp`, padrão *Template Method*

```
1 public class ConcreteMetodo1X_ap extends X_ap{
2     public <<tipo>> Metodo1(<<params>>){...}
}
```

d) Reestruturar a classe `X_ap`. O Quadro 106 apresenta o gabarito para essa classe.

i. Tornar essa classe abstrata (linha 1).

ii. Todos os métodos a serem definidos em subclasses de `X_ap` devem ser declarados como abstratos (linhas 2 e 3).

iii. Caso o código de `X_ap` tenha sido identificado no passo **b.ii**, nada deve ser alterado, pois ao invocar os métodos (linhas 5 e 6), que agora são definidos como abstratos, serão procurados entre as subclasses.

iv. Caso o código de `X_ap` tenha sido identificado no passo **b.iii**, deve-se excluir o código variante (se ainda não o tiver sido) e inserir as chamadas aos respectivos métodos abstratos (linhas 5 e 6).

Quadro 106 – Classe `X_ap`, padrão *Template Method*

```
1 public abstract class X_ap{
     //atributos da classe
2    protected <<tipo>> Metodo1(<<params>>);
3    protected <<tipo>> Metodo2(<<params>>);
...
4   public <<tipo>> Metodo(<<params>>){
       //declarações invariantes
5      Metodo1(<<params>>);
       //declarações invariantes
6      Metodo2(<<params>>);
       //declarações invariantes
   }
}
```

Anexos

Anexo 1: CD-ROM

Esta dissertação é acompanhada de um CD-ROM que contém o material de apoio relativo ao projeto realizado, como os estudos de caso e um resumo de padrões de projeto. Além disso, é disponibilizada a ferramenta de desenvolvimento Eclipse (2005).

Referências Bibliográficas

ALEXANDER. C. **The Timeless Way of Building**. New York: Oxford University Press, 1979.

ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M.; IACOBSON, M.; FIKSDAHL-KING, I. **A Pattern Language: Towns, Buildings, Construction**. New York: Oxford University Press, 1977.

AMBLER, W.; JEFFRIES, R. **Agiles Modeling: Effective Practices for Extremming Programing and the Unified Process**. John Wiley & Sons, 2002.

AMIOT, H.; COINTE, P.; GUÉHÉNEUC, Y.; JUSIEN, N. *Instantiating and Detecting Desigh Patterns: Putting Bits and Pieces Together.* In: Annual International Conference on Automated Software Engineering (ASE), 16., San Diego. **Proceedings…**, 2001.

APPLETON, B. Patterns and software: Essential concepts and terminology. Disponível em: http://www.bradapp.net/docs/patterns-intro.html, acesso em Maio 2004.

BASILI, V.; BRIAND, L.; CONDON, S.; KIM, Y.; MELO, W.; VALETT, J. *Understanding and Predicting the Process of Software Maintance Releases.* Relatório Técnico, University of Maryland, 1995.

BENNET, K.; XU, J. *Software Services and Software Maintenance*. In: European Conference On Software Maintenance And Reengineering (CSMR), 7., Itália, **Proceedings...**, 2003.

BERGEY, J.; SMITH, D.; TILLEY, S.; WEIDERMAN, N.; WOODS, S. *Why reengineering Projects Fail*. Software Engineering Institute (CMU/SEI), Relatório Técnico, Pittsburgh, 1999.

BIANCHI, A.; CAIVANO, D. *Iterative Reengineering of Legacy Systems*. IEEE Transactions on Software Engineering, vol. 29, nº 3, p. 225-241, 2003.

BOURGE, J.; BROWN, D. *Rationale Support for Maintenance of Large Scale Systems*. Journal of Software Maintenance and Evolution: Research and Practice, 2000.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. **Pattern Oriented Software Architecture: A System of Patterns**. England: John Wiley & Sons Ltd. Sussex, 1996.

CAGNIN, M. **PARFAIT: uma contribuição para a reengenharia de software baseada em linguagens de padrões e frameworks**. Tese (Doutorado em Ciência da Computação) - Universidade de São Paulo, São Carlos, 2005.

CHAPIN, N.; CIMITILE, A. *Journal of Software Maintenance and Evolution: Research and Practice*, 13., vol. 1, 2001.

CHIDAMBER, S.; KEMERER, C. *A Metrics Suite for Object-Oriented Design*. IEEE Transactions on Software Engineering, vol. 20, nº 6, p. 476-493, 1994.

CHIKOFSKY, E.; CROSS, H. *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Transactions on Software Engineering, vol. 7, nº 1, p. 13-17, 1990.

CINNÉIDE, M. **Automated Application of Design Patterns: A Refactoring Approach**. Tese (Doutorado em Ciência da Computação) - Trinity College, Dublin, 2000.

COOPER, J. **The Design Patterns Java Companion**. Addison-Wesley, 1998.

COPLIEN, J. *Software Design Patterns: Common Questions and Answers*. In: L. Rising – The Patterns Handbook: Techniques, Strategies, and Applications, Cambridge University Press, p. 311-320, 1998.

COPLIEN, J. A Pattern Definition. Disponível em: http://hillside.net/patterns/definition.html, acesso em Maio de 2004.

CUNNINGHAM, W.; BECK, K. *Using Pattern Languages for Object-Oriented Programs*. In: Workshop on Specification and Design for Object-Oriented Programming (OOPSLA), Relatório Técnico, Florida, 1987.

DENG, Y.; KOTHARI, S. *Using conceptual roles of Data for Enhanced Program Comprehension*. In: Working Conference on Reverse Engineering, 9., Virginia. **Proceedings…**,2002.

ECLIPSE. Disponível em: www.eclipse.org, acesso em Nov 2005.

FOWLER, M. **Analysis Patterns, Reusable Object Models**. Massachusetts: Addison-Wesley, 1997.

FOWLER, M. **Refactoring: Improving the Design of Existing Code**. Boston: Addison-Wesley, 1999.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns – Elements of Reusable Object-Oriented Software**, New York: Addison-Wesley, 1995.

GRAND, M. **Patterns in Java: a Catalog of Reusable Design Patterns Illustrated with UML**. Wiley Computer Publishing, 1998. vol. 1.

HAMMOUDA, I; HARSU, MAARIT. *Documenting Maintenance Tasks Using Maintenance Patterns*. In: European Conference on Software Maintenance and Reengineering (CSMR), 8., Tampere. **Proceedings…**, p. 37-47, 2004.

JAVA, 2005. Disponível em: http://www.java.com, acesso em Ago 2005.

JEON, S.; LEE, J.; BAE, D. *An Automated Refactoring Approach to Design Pattern-Based Program Transformations in Java Programs*. In: Asia-Pacific Software Engineering Conference, 9., **Proceedings…**, 2002.

KERIEVSKY, J. **Refactoring to Patterns**. Addison-Wesley, 2004.

LIENTZ, B.; SWANSON, B. **Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations**. Massachusetts: Addison-Wesley, 1980.

MARTIN, J.; McCLURE, C. **Software Maintenance: The Problem and Its Solutions**. New Jersey: Prentice Hall, 1983.

MESZAROS, G.; DOBLE, J. **Pattern Languages of Program Design**, Massachusetts: Addison-Wesley, 1998.

MISRA, S.; BHAVSAR, V. *Measures of Software System Difficulty*. American Society for Quality (ASQ), Magazines and Journals, Software Quality Professional, vol. 5, nº 4, 2003.

MULLER, H.; HAHNKE, J.; SMITH, D.; STOREY, M.; TILLEY, S.; WONG, K. *Reverse Engineering: A Roadmap*. In: Future of Software Engineering. Ireland: Anthony Finkelstein, 2000.

MURAKI, T.; SAEKI, M. *Metrics for Applying GOF Design Patterns in Refactoring Processes*. In: International Conference on Software Engineering, 4., Viena. **Proceedings…**, p. 27-36, 2001

OMONDO. Disponível em: www.omondo.com, acesso em Nov 2005.

PRECHELT, L.; UNGER, B.; BRÖSSLER, P. *A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions*. IEEE Transactions on Software Engineering, vol. 27, nº 12, p. 1134-1144, 1998.

PRESSMAN, R. **Software Engineering**, McGraw-Hill, 2005.

RAJESH, J.; JANAKIRAM, D. *JIAD: A Tool to Infer Design Patterns in Refactoring*. In: International Conference on Principles and Practice of Declarative Programming, 6., Verona. **Proceedings…**, 2004.

REFACTORING. Disponível em: www.refactoring.com, acesso em Nov 2005.

SHI, N.; OLSSON, R. *Reverse Engineering of Design Patterns for High Performance Computing*. In: Workshop on Patterns in High Performance Computing. Illinois, 2005.

SHULL, F.; MELO, W.; BASILI, V. *An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems*. Relatório Técnico, University of Maryland, 1996.

STARK, G.; OMAN, P.; SKILLICORN, A.; AMEELE, R. *An Examination of the Effects of Requirements Changes on Software Maintenance Releases*. In: Journal of Software Maintenance Research and Practice, Relatório Técnico, Vol. 11, p. 293-309, 1999.

UML. Disponível em: www.uml.org, acesso em Nov 2005.

VALETT, J.; CONDON, J.; BRIAND, L.; KIM, Y.; BASILI, V. *Building an Experience Factory for Maintenance*, In: Annual Software Engineering Workshop, 19., NASA Goddard Space Flight Center, **Proceedings…**,1994.

VLISSIDES, J. **Reverse Architecture**. In: Software Architectures Seminar. Schloss Daghstuhl, Germany, 1995.