Ministério da Educação
Centro Federal de Educação Tecnológica do Paraná
Programa de Pós-Graduação em Eng. Elétrica e Informática Industrial

TESE DE DOUTORADO

por

MARCOS ANTONIO QUINÁIA

CONTRIBUIÇÃO A UMA METODOLOGIA PARA IDENTIFICAÇÃO E ESPECIFICAÇÃO DE PADRÕES ARQUITETURAIS DE *SOFTWARE*

Banca Examinadora:

Prof. Dr. Paulo Cézar Stadzisz – CEFETPR (Orientador)

Profª. Drª. Elisa Hatsue Moriya Huzita – UEM

Prof. Dr. João Umberto Furquim de Souza – UEPG

Prof. Dr. Cesar Augusto Tacla – CEFETPR

Prof. Dr. Luis Ernesto Merkle – CEFETPR

Curitiba, agosto de 2005.

MARCOS ANTONIO QUINÁIA

CONTRIBUIÇÃO A UMA METODOLOGIA PARA IDENTIFICAÇÃO E ESPECIFICAÇÃO DE PADRÕES ARQUITETURAIS DE *SOFTWARE*

Orientador:
PROF. DR. PAULO CÉZAR STADZISZ

Tese apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI) do Centro Federal de Educação Tecnológica do Paraná (CEFET-PR), como parte dos requisitos para obtenção do título de Doutor em Ciências. Área de Concentração: Informática Industrial. Sub-área: Engenharia de *Software*.

Curitiba, agosto de 2005.

Ficha catalográfica elaborada pela Biblioteca do CEFET-PR – Unidade Curitiba

Q7c Quináia, Marcos Antonio
Contribuição a uma metodologia para identificação e especificação de padrões arquiteturais de software / Marcos Antonio Quináia. – Curitiba : CEFET-PR. 2005.
Xv, 141 f. : il. ; 30 cm

Orientador: Prof. Dr. Paulo Cézar Stadzisz
Tese (Doutorado) – CEFET-PR. Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial. Curitiba, 2005.

1. Engenharia de software. 2. Arquitetura de software. 3. Análise de domínio. 4. Reuso. 5. UML. 6. Padrão arquitetural. I. Stadzisz, Paulo Cézar. II. CEFET-PR. Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial. III. Título.

CDD : 005.1

À Sueli, Bruno e Vitor.

Agradeço a Deus por ser minha luz e força.
Agradeço a minha esposa e filhos que pacientemente souberam tolerar as longas horas do nosso convívio que foram subtraídas e dedicadas a esta tese.
Agradeço a meus pais e familiares que sempre me apoiaram.
Agradeço ao professor Paulo Cézar Stadzisz pela amizade, paciência e profissionalismo demonstrados na orientação deste trabalho de tese.
Agradeço aos professores avaliadores deste trabalho pela honra que me fazem em participar da banca.
Agradeço ao CEFET-PR por proporcionar este doutorado.
Agradeço à UNICENTRO pela dispensa para fazer este curso.
Agradeço aos colegas do LSIP pelo companheirismo, pelas trocas de idéias e pelos momentos de convivência e descontração que passamos a longo do período deste curso.

Sumário

Lista de Figuras VIII
Lista de Tabelas X
Lista de Siglas e Acrônimos XI
Glossário de Termos XIII
Resumo XIV
Abstract XV
1 Introdução 16
1.1 Contextualização e Objetivo do Trabalho 16
1.2 Fatores Motivadores 20
1.3 Trabalhos Relacionados 22
1.4 Organização Desta Tese 24
2 Fundamentação Teórica 25
2.1 Arquitetura de Software 25
2.2 Padrões 30
2.3 Padrões Arquiteturais / Estilos Arquiteturais 39
2.4 Componentização e Componentes de Software 40
2.5 UML – Unified Modeling Language 43
2.6 Modelo de Requisitos 44
2.7 Análise de Domínio 46
2.8 Modelo e Metamodelo 48
2.9 Considerações Finais 49
3 Tipologias Propostas 50
3.1 Extensão do Modelo de Casos de Uso para Padrões Arquiteturais 50
3.1.1 Tipologia para Atores e Casos de Uso 51
3.1.2 Combinação de Tipos 54
3.1.3 Relacionamentos entre Atores e Casos de Uso 56
3.1.4 Relacionamentos entre Casos de Usos 57
3.1.5 Aplicando a Extensão de Caso de Uso na Especificação de Padrões Arquiteturais 58
3.2 Tipologia para as Classes Genéricas 60
3.2.1 Tipos Aplicáveis às Classes 60
3.2.2 Relacionamentos entre as Casses Genéricas 61
(i) Associações 61
(ii) Agregações 62
(iii) Generalizações / Especializações 63
3.3 Tipologia para os Diagramas de Seqüência Genéricos 64
3.4 Tipologia para os Diagramas de Estados Genéricos 64
3.5 Tipologia para os Diagramas de Componentes Genéricos 66
3.5.1 Tipos Aplicáveis aos Componentes 67
3.5.2 Relacionamentos entre os Componentes Genéricos 67
3.6 Considerações Finais 68
4 Metamodelo para Descrição de Padrão Arquitetural 69
4.1 Objetivos do Metamodelo 69

4.2 Metamodelo Proposto 70
4.2.1 Modelo de Descrição do Padrão 71
4.2.2 Modelo de Definição do Padrão 71
4.2.3 Modelo Arquitetural do Padrão 72
(i) Modelo de Requisitos 73
(ii) Modelo Estrutural 73
(iii) Modelo da Dinâmica 74
(iv) Modelo de Implementação 74
4.3 Processo para a identificação e descrição de Padrões Arquiteturais 74
4.3.1 Análise de Domínio 75
(i) Escolha de Domínio 76
(ii) Seleção de Sistemas e Análise de Custo / Benefício 76
(iii) Extração das Especificações dos Sistemas 76
4.3.2 Análise dos Requisitos 77
(i) Analisar Equivalências Funcionais 77
(ii) Elicitar Requisitos Genéricos 79
4.3.3 Análise das Arquiteturas 80
(i) Analisar Classes 81
(ii) Analisar Interações 84
(iii) Analisar Estados 86
(iv) Analisar Componentes 87
4.4 Considerações Finais 89
5 Estudo de Caso 90
5.1 Identificação e Descrição do Padrão Arquitetural para os Sistemas de Biblioteca 90
5.1.1 Análise do Domínio 90
(i) Escolha do Domínio 90
(ii) Seleção de Sistemas e Análise de Custo / Benefício 91
(iii) Extração das Especificações dos Sistemas 91
5.1.2 Análise dos Requisitos 91
(i) Análise de Equivalências Funcionais 91
(ii) Elicitação de Requisitos Genéricos 93
5.1.3 Análise das Arquiteturas 96
(i) Análise de Classes 96
(ii) Análise de Interações 104
(iii) Análise de Relacionamentos entre as Classes 105
(iv) Análise de Estados 106
(v) Análise de Componentes 107
5.1.4 Catalogação do Padrão Arquitetural 108
5.2 Considerações Finais 111
6 Conclusões e Perspectivas Futuras 112
6.1 Conclusões 113
6.2 Publicações 114
6.3 Perspectivas Futuras 115
7 Bibliografia 117
7.1 Referências 117
7.2 Complementar 123
Anexo I – Análises de Equivalência Funcional dos Casos de Uso 125
Anexo II – Diagramas de Seqüência Genéricos 129
Anexo III – Diagramas de Classes 136

Lista de Figuras

Figura 2.1 – Exemplo de Componente 41
Figura 2.2 – Exemplo de Caso de Uso 45
Figura 2.3 – Exemplo de Ator 45
Figura 2.4 – Exemplo simples de Modelo de Casos de Uso 46
Figura 2.5 – Ciclo de vida da Engenharia de Domínio e da Engenharia de Sistemas [SEI-CMU 2003b]. 47
Figura 2.6 – Um exemplo de metamodelo [OMG 2003]. 48
Figura 3.1 – Notação para Atores e Casos de Uso segundo a UML 52
Figura 3.2 – Notação para Atores e Casos de Usos Opcionais 52
Figura 3.3 – Notação para Atores e Casos de Usos Variantes 53
Figura 3.4 – Notação para Atores e Casos de Usos Múltiplos 54
Figura 3.5 – Notação para Atores e Casos de Usos Múltiplos e Opcionais 54
Figura 3.6 – Notação para Atores e Casos de Usos Variantes e Múltiplos 55
Figura 3.7 – Notação para Atores e Casos de Usos Opcionais e Variantes 55
Figura 3.8 – Notação para Atores e Casos de Usos Múltiplos, Variantes e Opcionais 56
Figura 3.9 – Relacionamento inválido 56
Figura 3.10 – Outras relações inválidas 57
Figura 3.11 – Relacionamentos permitidos entre atores e casos de uso 57
Figura 3.12 – Relacionamentos entre Casos de Usos 58
Figura 3.13 – Exemplo de um Modelo de Casos de Uso Genérico 59
Figura 3.14 – Tipologia para as classes 61
Figura 3.15 – Relacionamento com uma classe opcional 62
Figura 3.16 – Relacionamento com uma classe variante 62
Figura 3.17 – Relacionamentos de agregação de classes opcionais 62
Figura 3.18 – Relacionamentos de agregação de classes variantes 63
Figura 3.19 – Relacionamento de herança de classes opcionais 63
Figura 3.20 – Relacionamento de herança de classes variantes 63
Figura 3.21 – Notação para os diagramas de seqüência 64
Figura 3.22 – Notação para os diagramas de estados com estereótipo <<op>> 65
Figura 3.23 – Notação para os diagramas de estados com estereótipo <<var>> 66
Figura 3.24 – Descrição de um componente 66
Figura 3.25 – Notação para componentes genéricos 67
Figura 3.26 – Notação para relacionamento com componentes genéricos opcionais 68
Figura 3.27 – Notação para relacionamento com componentes genéricos variantes 68
Figura 4.1 – Relacionamentos externos de um padrão arquitetural 71
Figura 4.2 – Elementos de descrição de um padrão arquitetural 71
Figura 4.3 – Elementos de definição de um padrão arquitetural 72
Figura 4.4 – Modelo da arquitetura de um padrão 73
Figura 4.5 – Notação básica dos diagramas SADT 74
Figura 4.6 – Visão Geral do Processo Proposto 75
Figura 4.7 – Decomposição da fase Análise de Domínio 75
Figura 4.8 – Decomposição da fase Análise dos Requisitos 77
Figura 4.9 – Exemplo de caso de uso Variante 79
Figura 4.10 – Exemplo de caso de uso Múltiplo 80
Figura 4.11 – Decomposição da etapa Análise das Arquiteturas 81
Figura 4.12 – Unificação de classes de interface dos sistemas do domínio para a classe de interface do padrão 82
Figura 4.13 – Unificação de classes de controle dos sistemas do domínio para a classe de controle do padrão 83
Figura 4.14 – Unificação de classes de entidade dos sistemas do domínio para a classe de entidade do padrão 84
Figura 4.15 – Diagrama de seqüência evidenciando parte opcional e variante 86
Figura 4.16 – Diagrama de Estados evidenciando parte opcional e variante 87
Figura 4.17 – Unificação de componentes dos sistemas para o componente do padrão 88
Figura 5.1 – Modelo de Casos de Uso Genérico para o subsistema de movimentação 95
Figura 5.2 – Correspondência entre os atributos das classes dos sistemas analisados e os atributos do padrão 98
Figura 5.3 – Correspondência entre os métodos das classes dos sistemas analisados e os métodos do padrão 102
Figura 5.4 – Classes de entidade do padrão 103
Figura 5.5 – Classes de interface 103
Figura 5.6 – Classes de controle 104

Figura 5.7 – Diagrama de seqüência para o caso de uso Emprestar Exemplar 105
Figura 5.8 – Diagrama de classes para o caso de uso Emprestar Exemplar 106
Figura 5.9 – Diagrama de estados para a classe Emprestimo 107
Figura 5.10 – Diagrama de componentes para o subsistema movimentação 108
Figura II.1 – Diagrama de seqüência genérico para o caso de uso Ler Código de Barras 129
Figura II.2 – Diagrama de seqü. genérico para o caso de uso Consultar Histórico De Empréstimos De Usuário 130
Figura II.3 – Diagrama de seqüência genérico para o caso de uso Emitir Aviso De Devolução Em Atraso 130
Figura II.4 – Diagrama de seqüência genérico para o caso de uso Reservar Obra 131
Figura II.5 – Diagrama de seqüência genérico para o caso de uso Consultar Reserva 131
Figura II.6 – Diagrama de seqüência genérico para o caso de uso Cancelar Reserva de Obra 132
Figura II.7 – Diagrama de seqüência genérico para o caso de uso Emitir Multa 132
Figura II.8 – Diagrama de seqüência genérico para o caso de uso Cancelar Multa 133
Figura II.9 – Diagrama de seqüência genérico para o caso de uso Renovar Empréstimo 134
Figura II.10 – Diagrama de seqüência genérico para o caso de uso Devolver Exemplar 135
Figura III.1 – Diagrama de classes genérico para o caso de uso Ler Código de Barras 136
Figura III.2 – Diagr. de classes genérico para o caso de uso Consultar Histórico De Empréstimos De Usuário 137
Figura III.3 – Diagrama de classes genérico para o caso de uso Cancelar Multa 137
Figura III.4 – Diagrama de classes genérico para o caso de uso Emitir Aviso De Devolução Em Atraso 138
Figura III.6 – Diagrama de classes genérico para o caso de uso Consultar Reserva 139
Figura III.7 – Diagrama de classes genérico para o caso de uso Cancelar Reserva de Obra 140
Figura III.8 – Diagrama de classes genérico para o caso de uso Emitir Multa 140
Figura III.9 – Diagrama de classes genérico para o caso de uso Renovar Empréstimo 141
Figura III.10 – Diagrama de classes genérico para o caso de uso Devolver Exemplar 141

Lista de Tabelas

Tabela 2.1 – Diferenças entre a arquitetura e programa 27
Tabela 4.1 – Subsistemas, Casos de uso e Atores 78
Tabela 5.1 – Casos de uso do padrão 92
Tabela 5.2 – Elicitação de requisitos do subsistema movimentação 94
Tabela 5.3 – Resumo da equivalência funcional do caso de uso Emprestar Exemplar 94
Tabela I.1 – Análise do caso de uso consultar histórico de empréstimos de usuário 125
Tabela I.2 – Análise do caso de uso devolver exemplar 126
Tabela I.3 – Análise do caso de uso emitir aviso de devolução em atraso 126
Tabela I.4 – Análise do caso de uso renovar empréstimo 126
Tabela I.5 – Análise do caso de uso ler código de barras 126
Tabela I.6 – Análise do caso de uso reservar obra 127
Tabela I.7 – Análise do caso de uso cancelar reserva de obra 127
Tabela I.8 – Análise do caso de uso consultar reserva 127
Tabela I.9 – Análise do caso de uso emitir multa 128
Tabela I.10 – Análise do caso de uso cancelar multa 128
Tabela I.11 – Campos definidos para o caso de uso cancelar multa 128

Lista de Siglas e Acrônimos

µRapid: Nome de uma Linguagem de Descrição de Arquitetura.

ACME: Nome de uma Linguagem de Descrição de Arquitetura desenvolvida na Universidade Carnegie Mellon.

ADL (*Architectural Description Language*): Linguagem de Descrição de Arquitetura.

Aesop: Nome de um Ambiente Gerador de Projetos de Arquitetura de *Software* desenvolvido na Universidade Carnegie Mellon.

C: Linguagem de Programação de Computadores.

C++: Linguagem de Programação de Computadores.

CASE (*Computer-Aided Software Engineering*): Engenharia de *software* Auxiliada por Computador.

Catalysis: Processo de desenvolvimento de *software* desenvolvido por Desmond F. D'Souza e Alan C. Wills.

CBS (*COTS-Based Systems*): Sistemas baseados em COTS.

ChiliPLoP (*Southwestern Conference on Pattern Languages of Programs*): Conferência do sudoeste Norte Americano sobre Linguagens de Padrões de Programas.

COTS (*Commercial off-the-shelf*): Produtos ou componentes prontos para uso que podem ser facilmente obtidos comercialmente.

EHDM (*Enhanced Hierarchical Development Methodology*): Metodologia de Desenvolvimento Hierárquica Melhorada.

ESSI (*European Systems & Software Initiative*): Aliança Européia que objetiva auxiliar suas organizações para o aumento de eficiência e qualidade na produção de *software*.

EuroPLoP (*European Conference on Pattern Languages of Programs*): Conferência Européia sobre Linguagens de Padrões de Programas.

GoF (*Gang of Four*) Gangue dos Quatro: Atribuição aos autores do livro *Design Patterns - Elements of Reusable Object-Oriented Software* (Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides).

JAIST (*Japan Advanced Institute of Science and Technology*): Instituto Avançado Japonês para Ciência e Tecnologia.

Koala PLoP (*Australian Conference on Pattern Languages of Programs*): Conferência Australiana sobre Linguagens de Padrões de Programas.

Mensore PLoP (*East Asia Pattern Language of Programming Conference*): Conferência da Ásia Oriental sobre Linguagens de Padrões de Programas.

NFR (*Non-Functional Requirements*): Requisitos não-funcionais.

OMG (*Object Management Group*): Consórcio que produz e mantém especificações para a indústria de sistemas computacionais objetivando a interoperabilidade de aplicações de empresas.

OOD (*Object Oriented Design*): Projeto orientado a objeto.

OOPSLA'87 Conferência sobre Linguagens de Programação de Sistemas e Aplicações Orientados a Objetos.

OP: Opcional.

PLoP (*Pattern Languages of Programs*): Conferência Norte Americana sobre Linguagens de Padrões de Programas.

PML (*Pattern Markup Language*): Linguagem de Marcação de Padrões.

POSA1 (*Pattern-Oriented Software Architecture Volume 1*): Arquitetura de *Software* Orientada por Padrão, volume 1.

POSA2 (*Pattern-Oriented Software Architecture Volume 2*): Arquitetura de *Software* Orientada por Padrão, volume 2.

RMF (*Requirements Modeling Framework*): Estrutura de Modelagem de Requisitos.

RSL (*Requirements Specification Language*): Linguagem de especificação de Requisitos.

RUP (*Rational Unified Process*): Processo de desenvolvimento de *software* desenvolvido pela empresa Rational.

SADT (*Structured Analysis and Design Technique*): Técnica para análise e projeto estruturados.

SEI-CMU (*Software Engineering Institute – Carnegie Mellon University*): Instituto de Engenharia de *Software* da Universidade Carnegie Mellon.

SEKAS (*Software Engineering für Kommunikations- und Automatisierungssysteme*): Nome de uma companhia alemã de desenvolvimento de *software* .

SEPIOR (*Software Engineering Process Improvement through systematic application of Object Oriented techniques and Reusability*): Melhoria no Processo de Engenharia de *Software* através de aplicação sistemática de técnicas Orientadas a Objeto e Reusabilidade. Acrônimo utilizado em vários experimentos de processos de engenharia de *software* realizados na companhia SEKAS.

SGBD: Sistema Gerenciador de Banco de Dados. Tradução para DBMS – *Data Base Management System*.

SugarLoafPLoP (*The Latin American Conference on Pattern Languages of Programming*): Conferência Latino-Americana sobre Linguagens de Padrões de Programas.

UML (*Unified Modeling Language*): Linguagem de Modelagem Unificada.

UP (*Using Patterns*): Conferência internacional sobre uso de padrões.

VAR: Variante.

XML (*Extensible Markup Language*): Linguagem de Marcação Extensível.

WEB (teia): Rede mundial de computadores.

Wright: Nome de uma Linguagem de Descrição de Arquitetura desenvolvida na Universidade Carnegie Mellon.

Glossário de Termos

Arquitetural: Arquitetônico (Dicionário Aurélio).

Componente (*component*): que ou o que entra na composição de alguma coisa (Dicionários Aurélio e Michaelis). Na engenharia de *software*, componente refere-se à parte que integra um determinado *software*. Uma melhor definição de componente encontra-se no capítulo de fundamentação teórica.

Componentização (*componentware*): embora o equivalente próximo em português seja "composição", neste trabalho, optou-se por utilizar o termo componentização, que não existe oficialmente na língua portuguesa. Componentização refere-se à construção (montagem) de *software* a partir de elementos (componentes de *software*) já definidos, testados e aprovados. Ver: http://www.sei.cmu.edu/cbs/icse98/papers/p6.html.

Internet: um sistema interconectado de redes de trabalho que conecta computadores em volta do mundo através do protocolo TCP/IP (Fonte: *The American Heritage® Dictionary of the English Language, Fourth Edition*).

Metamodelo (*metamodel***)**: termo não encontrado em dicionários eletrônicos da língua portuguesa (Michaelis, Houaiss). O termo é definido no *Webster's New Millennium™ Dictionary of English, © 2003 Lexico Publishing Group, LLC* como *metamodel (noun) a model that defines the components of a conceptual model, process, or system*.

Reuso: este termo não consta em dicionários da língua portuguesa (ver Dicionários Aurélio, Michaelis e Priberam). Neste trabalho "reuso" é utilizado como sinônimo para "reutilização"; assim como o verbo "reusar" é usado como sinônimo para "reutilizar".

Reutilização: ato ou efeito de reutilizar (Dicionários Aurélio e Michaelis).

Reutilizar: tornar a utilizar; dar nova utilização (Dicionários Aurélio e Michaelis).

Resumo

Embora o conceito de reuso não seja novo em engenharia de *software*, a conscientização sobre sua importância e seu impacto no desenvolvimento de sistemas computacionais ainda não é generalizada. Uma das principais dificuldades encontradas pelas empresas de desenvolvimento é a inexistência de métodos efetivos para reuso de *software*.

As pesquisas em engenharia de *software* que enfocam o reuso de *software* têm gerado contribuições crescentes na forma de paradigmas (como a orientação a objeto), de conceitos (como domínios e componentes) e de métodos e processos. Mais recentemente, o conceito de Padrão (*Pattern*) possibilitou novas formas de representações e a proposição de métodos ainda mais orientados ao reuso.

Os padrões de *software* denominados padrões de projeto e padrões de programação (ou idiomas) têm sido bem explorados na literatura científica. Modelos de representação e exemplos são encontrados em artigos e livros especializados. O conceito de Padrão Arquitetural, entretanto, não possui a mesma maturidade. Trata-se de um conceito mais abrangente envolvendo modelos e técnicas mais extensos e complexos. Essencialmente, um Padrão Arquitetural é um modelo genérico de uma solução de *software* para um determinado domínio. Seu desenvolvimento permite que se possa reusar toda a arquitetura de um sistema em novos projetos semelhantes.

O objetivo deste trabalho é oferecer uma contribuição para uma metodologia de identificação e descrição de Padrões Arquiteturais de *Software*.

Para atingir este objetivo foram criadas cinco tipologias para serem empregadas como extensão da UML na descrição de casos de uso, classes, seqüências, estados e componentes. Esta descrição diferenciada visa evidenciar, nos diagramas da UML, as partes do padrão que são variantes, opcionais e múltiplas. Estas tipologias são empregadas no metamodelo desenvolvido para a descrição dos padrões arquiteturais. A utilização do metamodelo e, conseqüentemente, das tipologias é guiada pelos passos de um processo que também foi criado para servir como um roteiro para se chegar à descrição do padrão arquitetural.

Palavras-chave: Engenharia de *software*; Arquitetura de *software*; Análise de domínio; Reuso; UML; Padrão arquitetural.

Abstract

Although the concept of reuse is not new in software engineering, the awareness of its importance and its impact in the development of computational systems is not generalized yet. One of the main difficulties found by development companies is the absence of effective methods for software reuse.

The researches on software engineering that focuses software reuse have generated increasing contributions in the form of paradigms (such as object orientation), concepts (such as domain and component), and methods, languages and processes. More recently, the concept of Pattern allowed new representation forms and the proposal of methods even more reuse-oriented.

Software patterns named as Design Patterns and Programming Patterns (or Idioms) have been well explored at scientific literature. Representation models and examples related to these patterns type are found in articles and specialized books. The concept of Architectural Pattern, however, does not have the same maturity. This is a more including concept involving more extensive and complex models and techniques. Essentially, an Architectural Pattern is a generic model of a software solution to a determined domain. Its development may allow the reuse of all the architecture of a system in new similar projects.

This work aims to offer a contribution for a methodology of Software Architectural Patterns identification and description.

In order to achieve this objective, five typologies have been created to be employed as UML extension in the description of use cases, classes, sequences, states and components. This differentiated description aims to evidence, in the UML diagrams, variant, optional, and multiple pattern parts. These typologies are used in the developed metamodel for the architectural patterns description. The use of the metamodel, and consequently the use of typologies, is guided by the steps of a process, which was also created to serve as a guideline to accomplish the architectural pattern description.

Keywords: Software Engineering; Software Architecture; Domain Analysis; Reuse; UML; Architectural Pattern.

1 Introdução

It is not the answer that enlightens, but the question.

[Eugène Ionesco]

O objetivo deste capítulo é oferecer ao leitor um panorama referente a este trabalho de tese. As seções deste capítulo apresentam o contexto inerente ao trabalho, bem como os objetivos pretendidos com a realização deste trabalho (seção 1.1). São também apresentados os fatores motivadores que justificam esta pesquisa (seção 1.2); trabalhos que são relacionados com o objetivo desta pesquisa (seção 1.3) e por fim, a organização dos capítulos que compõem esta tese (seção 1.4).

1.1 Contextualização e Objetivo do Trabalho

Atualmente, o computador está sendo empregado em todo o mundo, nas mais diversas áreas tais como serviços, lazer, medicina, lar, engenharia, comércio e indústria. Esta forte evolução de consumo vem acompanhada por um conseqüente aumento na complexidade dos sistemas computacionais. Para fazer frente a essas necessidades crescentes e para garantir qualidade e produtividade no processo de desenvolvimento de novos sistemas computacionais, torna-se fundamental a aplicação de técnicas rigorosas de engenharia de *software* [BROWN 1996], [JACOBSON et al 1997], [PRESSMAN 2004].

Historicamente, desde o surgimento da engenharia de *software*, a sua adoção e difusão tem sido lenta. A engenharia de *software* tem sido relegada à segundo plano, postergada em sua utilização ou simplesmente ignorada em muitas empresas e instituições, mantendo o empirismo, a prática do desenvolvimento artesanal e isolado e também possibilitando o aparecimento de mitos de informática entre desenvolvedores de *software* e usuários. Um desses mitos é o de que *software* (programa de computador) é flexível indicando, assim, que as mudanças requeridas podem ser facilmente acomodadas. Esse e outros mitos trouxeram grandes dificuldades à área de desenvolvimento de *software*, notadamente com relação ao descumprimento de cronogramas e previsões de custos, culminando com a chamada crise de *software*. Esta crise é caracterizada pela insuficiência em se produzir *software* em quantidade e qualidade necessária para atender a demanda existente [PRESSMAN 2004].

Na última década, a engenharia de *software* tem sido empregada como um fator diferencial necessário para enfrentar os grandes desafios relacionados à informatização e automação. A engenharia de *software* tem ganhado mais espaço no mundo da informática, dada a necessidade por melhorar a qualidade dos programas construídos, otimizar seu processo de desenvolvimento e fazer frente a sistemas mais complexos.

Vários avanços em métodos e técnicas de construção de *software* têm emergido ao longo dos anos. Dentre esses avanços, pode-se citar a orientação a objeto, interfaces gráficas, Web, UML, engenharia de domínio, engenharia de requisitos, engenharia reversa, reengenharia, ferramentas CASE e reuso de componentes. Esses avanços culminaram com o surgimento de um novo paradigma denominado "paradigma da industrialização de *software*". Este paradigma, aplicado ao desenvolvimento de *software*, busca inspiração em outros ramos industriais, como por exemplo, a indústria automobilística, onde o uso de engenharia é efetivo.

O reuso não é novidade em outras áreas, como por exemplo, na área de *hardware* onde um novo produto pode ser projetado reutilizando-se componentes que foram projetados para um produto anterior. Um outro exemplo clássico de reutilização de componentes ocorre na indústria automobilística, onde a criação de um novo automóvel encontra as facilidades de poder reutilizar, desde o nível de projeto até o de montagem, um grande conjunto de peças já existentes (como rodas, motor, câmbio, sistema de freios e injeção de combustível) deixando para os desenvolvedores a tarefa de criar apenas aquelas partes exclusivas do novo automóvel.

Na indústria de *software*, pode-se empregar este mesmo conceito de reutilização de componentes, fazendo com que a construção de um novo *software* corresponda à montagem ou integração de componentes que foram utilizados em projetos anteriores, com a adição de novas características e / ou funcionalidades que se deseja para o novo sistema.

É mais rápido elaborar um novo sistema a partir de componentes predefinidos do que escrever todo o código desde o início. Além disso, o esforço necessário para se garantir confiabilidade no produto pode ser consideravelmente menor quando o trabalho se fundamenta em elementos pré-fabricados, visto que são aproveitadas partes de *software* projetadas, construídas e testadas previamente.

No contexto da reutilização de *software*, o conceito de Padrões (*Patterns*) emerge como um dos mais importantes meios facilitadores para se chegar à efetiva reutilização de *software*. Um Padrão pode ser entendido como um modelo a ser seguido para se conseguir um determinado objetivo [APPLETON 2000a]. Por exemplo, para se construir um novo vestido, pode-se passar a um costureiro as medidas para os cortes e costuras necessárias ou, de outra forma, treiná-lo para utilizar um Padrão ou modelo que deve ser seguido. Ao se seguir apenas as especificações das medidas o costureiro não tem a exata imagem do que está construindo. Ao seguir um Padrão, por outro lado, o costureiro tem uma percepção antecipada dos resultados a serem alcançados, pois um modelo prefigura o produto. Assim, na construção de *software*, a adoção de Padrões previamente estabelecidos pode facilitar a construção, pois os elementos constituintes do novo *software* já foram testados e aprovados, e já se conhece o seu comportamento, possibilitando que se tenha uma imagem mais precisa dos resultados que serão alcançados.

Padrões têm sido utilizados em diferentes domínios, tais como: ambientes organizacionais e seus processos, ensino, arquitetura e missões críticas [APPLETON 2000b], [COPLIEN 2003]. Em engenharia de *software*, os Padrões ganharam popularidade e tornaram-se importantes a partir do trabalho de Erich Gamma e outros que apresentaram um conjunto de Padrões aplicáveis a problemas recorrentes na área de projeto de sistemas computacionais [GAMMA et al 1995]. Outros autores também apresentaram novas visões e modelos que também contribuíram para a popularização de Padrões:

- James Coplien e Douglas Schmidt apresentaram diversos Padrões em uma conferência (PloP) em 1995 [COPLIEN e SCHMIDT 1995].
- Martin Fowler escreveu o primeiro livro dedicado a Padrões de Análise. A obra é uma referência para modelagem utilizando o conceito de Padrões [FOWLER 1996].

- Frank Buschmann e outros escreveram um livro que é uma fonte de Padrões de projeto em arquitetura de *software* [BUSCHMANN et al 1996].

Cada Padrão está associado a um problema específico e recorrente no projeto ou implementação de *software*. A idéia central, em torno de Padrões, é capturar experiências comprovadamente corretas em desenvolvimento de *software* e tornar essas experiências, em forma de Padrão, disponíveis para futuras utilizações em outros projetos de *software* [APPLETON 2000b].

Ainda relacionado ao contexto da reutilização de *software*, existe a necessidade de se criar novos componentes de *software* em cada domínio, montando-se assim bibliotecas de componentes reutilizáveis. Esses componentes reusáveis (reutilizáveis) podem ser então empregados em desenvolvimentos de outros sistemas computacionais [COPLIEN 1996].

Existe uma preocupação crescente com a organização da estrutura computacional, das partes componentes e dos relacionamentos existentes em um sistema computacional. A maneira como o *software* é construído e organizado tem influência direta na sua funcionalidade, eficiência, manutenibilidade e outros atributos de qualidade de um *software*. A esta organização do *software* se dá o nome de arquitetura de *software*.

A arquitetura de *software* está relacionada a reuso de *software* na medida em que proporciona uma melhor visão das partes componentes do *software* e como elas são estruturadas. Dessa forma, um *software* que tenha uma arquitetura bem definida oferecerá melhores condições para que suas partes sejam reutilizadas na criação de outros sistemas de *software* e famílias de produtos.

A arquitetura de *software* tem ganhado importância em engenharia de *software* atualmente devido a vários fatores intrinsecamente relacionados a reuso de *software*. Dentre esses fatores, tem-se: (i) crescimento da utilização de métodos orientados a objetos que possibilitam o emprego mais efetivo de técnicas de reuso; (ii) aumento da complexidade e tamanho dos sistemas computacionais e (iii) surgimento de novos tipos de arquitetura de *software* que incluem padrões e componentes.

Uma área merecedora de atenção é a criação de métodos / técnicas voltados para a descoberta e especificação de Padrões visando a estruturação e formalização destes em diferentes áreas de aplicação [COPLIEN 1996].

Este trabalho de doutorado procura aliar o potencial oferecido por padrões e componentes de *software* aplicando esse potencial no reuso de arquiteturas de *software,*

fornecendo uma contribuição para identificação e especificação de padrões voltados para a arquitetura de *software*.

O objetivo deste trabalho é apresentar um metamodelo para ser utilizado na descrição de padrões arquiteturais[1] de *software*. O metamodelo proposto utiliza algumas tipologias para evidenciar as características genéricas dos modelos que compõem os padrões arquiteturais. Para realizar a identificação dos padrões foi desenvolvido um processo para guiar na utilização do metamodelo.

1.2 Fatores Motivadores

Um programador experiente dificilmente cria uma solução nova totalmente distinta de outras já implementadas. Normalmente este programador reutiliza os elementos recorrentes das soluções de problemas que já foram resolvidos anteriormente para construir um novo programa. Esse cenário é válido para o desenvolvimento de *software* (em termos de programação e projeto) e para muitas atividades humanas (em engenharia em particular) onde se observa a recorrência de determinados problemas e a possibilidade de reutilização total ou parcial de soluções já desenvolvidas. Neste contexto, os elementos recorrentes de uma solução já desenvolvida podem ser apresentados como um Padrão que pode se repetir em diversos problemas que nem sempre se encontram relacionados [BUSCHMANN et al 1996]. Assim, essa recorrência de problemas semelhantes torna a aplicação de várias técnicas, dentre elas as de mineração e especificação de Padrões para reuso, um grande campo a ser explorado.

Brad Appleton cita as motivações relacionadas a seguir para pesquisas voltadas à identificação e especificação de Padrões [APPLETON 2000a]:

- Possibilidade de captura de especialidades circunstanciadas em um domínio de aplicação: cada domínio de aplicação tem um conjunto de características específicas que podem ser capturadas e encapsuladas em Padrões;
- Documentação de soluções: cada Padrão estabelecido tem documentado seu nome, contexto e problema em que sua aplicação é indicada, bem como outros itens de documentação (p. e. solução, estrutura, usos do padrão, conseqüências do uso, motivação para o uso) que tornam mais fácil saber sobre a aplicabilidade daquele Padrão especificamente. Neste sentido, os Padrões poderiam compor uma biblioteca

[1] Neste trabalho, o termo padrão arquitetural é referente à arquitetura de *software*. Portanto um padrão arquitetural serve de modelo para se instanciar arquiteturas de *software*.

de soluções ou guias para a determinação de possíveis soluções para problemas de desenvolvimento de *software*;

- Reuso do conhecimento de desenvolvedores mais experientes: com a reutilização de um Padrão, não é necessário "reinventar a roda", pode-se aproveitar toda a experiência de quem já construiu o Padrão. Neste sentido, Padrões poderiam contribuir para a capitalização de conhecimento em engenharia de *software*;
- Padrões são uma forma de compartilhar vocabulário na resolução de problemas recorrentes: novamente, a reutilização possibilita o compartilhamento de experiências e, através disso, possibilita também o compartilhamento de vocabulário ou notação;
- Condução mais rápida aos resultados através da utilização de Padrões já estabelecidos no desenvolvimento de novos sistemas: o emprego de Padrões possibilita a reutilização de componentes já fabricados e testados reduzindo tempos e custos de desenvolvimento.

Essa busca de Padrões caracteriza-se como uma resposta de caráter prático da indústria de *software* ao crescimento acelerado da demanda por *software* em consonância com o aumento do nível de abstração das ferramentas disponíveis.

Um dos trabalhos pioneiros com Padrões [GAMMA et al 1995] apresenta experiências bem sucedidas com projetos de *software* orientados a objetos na forma de *Design Patterns*. Este trabalho tem grande importância por mostrar um campo novo, inexplorado e com ampla aplicabilidade no desenvolvimento de *software*.

Em 1995 Coplien alertou para a falta de especificação de Padrões em diversas áreas ainda inexploradas. Coplien abordou a questão de Padrões em organizações e processos (*process and organization Patterns*), comentando que o estudo de processos e organizações já está em estágio maduro, entretanto, a forma de Padrões para a descrição desses processos ainda é um tema de pesquisa [COPLIEN e SCHMIDT 1995].

Shull e outros comentam que poucas pesquisas têm focalizado o desenvolvimento de técnicas voltadas para o descobrimento, captura, formalização e avaliação de Padrões aplicáveis em desenvolvimento de *software* [SHULL et al 1996].

Atuando como fomentadoras dessa nova linha de pesquisas sobre Padrões e suas aplicações em *software*, várias conferências internacionais (PLoP, EuroPLoP, KoalaPLoP, MensorePLoP, SugarLoafPLoP, UP, e ChiliPLoP), abordando exclusivamente o tema Padrão, têm sido organizadas nos últimos anos pela comunidade de pesquisadores

envolvidos com desenvolvimento de *software*. Essas conferências internacionais têm o objetivo de auxiliar na divulgação e fomento de trabalhos relacionados a Padrões.

A importância de pesquisas em engenharia de *software* para a melhoria e resolução de problemas relacionados ao processo de construção de *software* - especialmente pesquisas sobre Padrões - pode ser sentida notando-se o grande interesse que tais pesquisas têm despertado em grande parte da indústria de *software* e de organismos governamentais [ATP 2003], [BROWN 1998].

A arquitetura de *software* tem se tornado o centro das atenções principalmente pelo aumento sempre crescente da demanda por produtos de *software* de grande porte e / ou produtos de *software* para domínios específicos [SHAW e GARLAN 2003], [MENDES 2002].

Esta preocupação ocorre principalmente porque sistemas menores não têm problemas de organização de sua arquitetura, já os sistemas maiores são passíveis de possuírem problemas advindos da má estruturação e relacionamento das partes componentes de um *software*. Nota-se uma sinergia entre a arquitetura de *software* e a componentização de *software*, que procura estabelecer meios para se produzir componentes de *software* bem como produzir sistemas computacionais a partir de componentes já estabelecidos.

Para os sistemas de grande porte, os benefícios advindos da adoção de uma arquitetura de *software* adequada, são vários e os mais evidentes são: prover as bases necessárias para análise e projeto e facilitar a manutenção posterior desses sistemas [SHAW e GARLAN 2003].

Embora já exista esclarecimento suficiente quanto aos benefícios tangíveis e intangíveis, nota-se ainda uma tendência dos projetistas de *software* em relegar pouca importância à arquitetura de *software* [MENDES 2002], [SEI-CMU 2003a]. Isso indica que pesquisas que envolvam arquiteturas de *software* podem e devem ser empreendidas para preencher esse grande campo ainda não completamente explorado.

1.3 Trabalhos Relacionados

Garg e outros [GARG et al 2003] apresentam uma abordagem na forma de um ambiente que dê suporte para gerenciar a evolução de uma linha de produção de *software*. Embora a abordagem trate de variação e opcionalidade, o trabalho não aborda a

multiplicidade e a descrição de um padrão arquitetural que represente um conjunto de *software*.

Keller e Wendt [KELLER e WENDT 2003] apresentam uma abordagem sistemática chamada FMC (Conceitos de Modelagem Fundamentais) para descrever a arquitetura conceitual de sistemas computacionais. FMC é uma ferramenta da tomada de decisão e de planejamento, que facilita a comunicação entre o arquiteto e as partes interessadas. A base conceitual da abordagem FMC fornece diversos mecanismos para reduzir a complexidade ao descrever a arquitetura de um sistema (decomposição hierárquica, separação da estrutura componente do comportamento e dos dados, distinguir componentes ativos e passivos e isolar os estados do controle dos estados dos dados). A abordagem dos autores consiste em descrever um sistema e não um conjunto de sistemas, como pretendido neste trabalho de tese.

Pinzger e Gall [PINZGER e GALL 2002] apresentam uma abordagem para a recuperação de arquitetura baseada em padrões arquiteturais e estilos. A abordagem dos autores usa estruturas do código de fonte como padrões e introduz uma aproximação iterativa e interativa da recuperação da arquitetura construída em cima de tais padrões de baixo nível extraídos do código de fonte. As associações entre instâncias de padrões extraídos e elementos arquiteturais tais como os módulos parecem resultar em visões novas e de alto-nível do sistema computacional. Estas visões do padrão fornecem a informação para um refinamento consecutivo do padrão para agregar e abstrair padrões de alto nível que permitem finalmente a descrição de uma arquitetura de sistema, baseando-se somente nas informações de um sistema e não de um conjunto de sistemas.

Bachmann e Bass [BACHMANN e BASS 2001] apresentam a experiência com o gerenciamento explícito de variabilidade dentro de uma arquitetura do *software*. Os autores descrevem como a gerência das variações em uma arquitetura pode ser feita de forma explícita e como o uso dos pontos de variação conectados às escolhas de um cliente pode ajudar a navegar nos lugares apropriados na arquitetura. Embora este trabalho aborde variabilidade, o mesmo não trata da descrição de um padrão de arquitetura de *software*.

Gurp e outros [GURP et al 2001] apresentam, discutem e fornecem uma estrutura da terminologia e dos conceitos a respeito da variabilidade, cujas técnicas se tornam cada vez mais importantes. Uma clara indicação desta tendência é a recente emergência das linhas de produção de *software*. As linhas de produção de *software* são sistemas computacionais grandes e industriais que pretendem especializar em produtos de *software* específicos. Adicionalmente os autores apresentam três padrões recorrentes de variabilidade e sugerem

um método para controlar a variabilidade em linhas de produção de *software*. A exemplo do parágrafo anterior, este trabalho também não trata da descrição de um padrão de arquitetura de *software*.

Guo e outros [GUO et al 1999] apresentaram um método semi-automático para recuperação (extração e análise) de arquitetura de *software*. O método guia o usuário na reconstrução de arquiteturas de *software* baseado no reconhecimento de padrões. Os padrões são reconhecidos através da codificação de heurísticas para aplicar em ferramentas de engenharia reversa. Este método aplica-se a um sistema computacional na tentativa de abstrair sua arquitetura. Este trabalho aplica-se na reconstrução da arquitetura baseando-se apenas em um sistema e não em um conjunto de sistemas.

1.4 Organização Desta Tese

O trabalho referente a esta tese está descrito em seis capítulos além desta introdução, das referências bibliográficas e dos anexos.

No capítulo 2 são apresentados os conceitos fundamentais que formam a base de conhecimento utilizada para estruturar este trabalho. São enfocados conceitos de arquitetura de *software*, de padrões de *software*, de componentes, da UML, de modelo de requisitos, de análise de domínio e de modelo e metamodelo.

O capítulo 3 apresenta as tipologias que foram construídas neste trabalho e que são usadas no metamodelo de descrição de padrões. São apresentadas as tipologias para modelos de casos de uso, para os modelos de classes, para os diagramas de seqüência, para os diagramas de estados e para os diagramas de componentes.

O capítulo 4 descreve o metamodelo criado para ser utilizado na descrição de padrões de arquitetura de *software*. São descritos o metamodelo proposto e o processo para identificação e descrição de padrões arquiteturais de *software*.

Encontra-se no capítulo 5 um estudo de caso executado para ilustrar a utilização do metamodelo desenvolvido, como também do processo de identificação e descrição de padrões.

As conclusões sobre o trabalho desenvolvido, assim como as perspectivas dos possíveis desdobramentos em trabalhos futuros podem ser encontradas no capítulo 6.

2 Fundamentação Teórica

… the key to reusable software is to reuse analysis and design: not code.

[NEIGHBORS 1980]

Este capítulo apresenta uma visão conceitual sobre os principais temas relacionados com este trabalho. São abordados assuntos relativos à arquitetura de *software* onde são apresentados os principais conceitos e tipos de arquiteturas atualmente existentes. Os padrões de *software* são descritos de forma a esclarecer os conceitos e tipos de padrões existentes. São também apresentados os padrões arquiteturais; foco central deste trabalho. A componentização de sistemas é também abordada, apresentando-se as ligações que a mesma possui com arquitetura e padrões de *software*.

São apresentadas algumas pesquisas desenvolvidas atualmente em universidades e centros de pesquisa, relacionados com arquiteturas de *software*, padrões de *software* (padrões em geral e padrões arquiteturais), componentização de *software* e UML.

2.1 Arquitetura de Software

Segundo Bass e outros [BASS et al 1998] a arquitetura de um sistema computacional é a estrutura ou estruturas do sistema que compreendem componentes de *software*, suas propriedades visíveis externamente e o relacionamento entre esses componentes.

Arquitetura de *Software* pode ser definida como a estrutura de composição dos sistemas computacionais, o relacionamento entre os componentes dos sistemas, bem como os princípios e diretrizes que norteiam o projeto e sua evolução ao longo do tempo [CLEMENTS et al 2003], [MENDES 2002], [GARLAN e SHAW 1994], [MAIER et al 2001].

Atualmente, os algoritmos e estruturas de dados computacionais já não constituem os principais problemas de projeto de *software*. Devido ao gradual aumento do tamanho dos sistemas computacionais, um foco tem sido dado aos aspectos arquiteturais.

A arquitetura de sistemas baseados em componentes apresenta um novo conjunto de problemas de projeto que têm sido tratados de várias formas incluindo diagramas informais e termos descritivos, linguagens de interconexão de módulos, moldes e *frameworks* para sistemas que servem às necessidades de domínios específicos e modelos formais dos mecanismos de integração de componentes [GARLAN e SHAW 1994].

Segundo a SEI-CMU, a arquitetura de *software*: (i) forma a espinha dorsal para a construção de bons sistemas computacionais, possibilitando uma especificação mais eficaz dos atributos funcionais e de qualidade; (ii) representa um investimento capitalizado e um modelo abstrato reutilizável que pode ser transferido de um *software* para o outro; (iii) representa um veículo comum para a comunicação entre todos os envolvidos com o desenvolvimento de um sistema, permitindo uma melhor resolução de problemas oriundos de objetivos conflitantes [SEI-CMU 2003a].

A arquitetura de *software* tem por objetivo estabelecer a estrutura de composição do *software* para a satisfação dos requisitos especificados. O arquiteto de *software*, de posse da especificação das necessidades do usuário e conhecedor dos possíveis meios (modelos de arquiteturas) existentes para a satisfação desses requisitos, estabelece uma arquitetura de *software* que atenda aos requisitos do cliente usando os recursos existentes e disponíveis no mercado.

Como um fator da evolução tecnológica, existe a possibilidade de um rompimento ou aprimoramento dos paradigmas estabelecidos procurando satisfazer alguma necessidade nova. Dessa forma, pode-se construir um novo modelo de arquitetura de *software*, baseado ou não nos modelos já existentes, procurando satisfazer alguma necessidade para a qual ainda não há modelo arquitetural satisfatório.

Segundo Garlan e Shaw [SHAW e GARLAN 2003], no processo de desenvolvimento de *software*, a arquitetura de *software* encontra-se em uma escala cujo escopo compreende:

- Modelo do usuário: o sistema é visto sob a ótica do usuário, ou seja, o usuário mostra a sua visão do *software* requerido.
- Requisitos: visão das necessidades dos usuários e exigências impostas ao *software*.
- **Arquitetura:** o sistema é modelado de maneira a se definir os componentes e seus inter-relacionamentos.
- Código: fornece a visão dos algoritmos e estruturas de dados do *software*.
- *Software* executável: compreende o *software* a ser distribuído ao cliente requerente.

Existem vários modelos de arquitetura de *software*, dentre os quais pode-se citar: arquitetura cliente servidor, arquitetura distribuída e orientada a objeto, "pipeline" e arquitetura de rede geral [SHAW e GARLAN 2003].

Fazendo-se uma analogia entre a arquitetura e programa de computador propriamente dito (Tabela 2.1), existem algumas diferenças que são importantes entender para melhor compreender o papel da arquitetura dentro do contexto de desenvolvimento de *software* [SHAW e GARLAN 2003].

Tabela 2.1 – Diferenças entre a arquitetura e programa

ARQUITETURA	PROGRAMA
Interações entre partes	Implementação de partes
Propriedades estruturais	Propriedades computacionais
Declarativa	Operacional
Geralmente estática	Geralmente dinâmico
Performance em nível de sistema	Performance algorítmica
Limite além dos módulos	Limite circunscrito aos módulos
Composição de subsistemas	Cópia de código ou chamada a funções de biblioteca

Em engenharia de *software*, linguagens são utilizadas para vários propósitos tais como linguagens de programação (*e. g.* C, C++), linguagens de especificação de requisitos (*e. g.* RSL, RMF e EHDM), linguagens de modelagem (*e. g.* UML) e linguagens de descrição de arquiteturas de *software* (*e.g.* Wright, μRapid). As Linguagens de Descrição de Arquitetura (Architectural Description Language - ADL) são linguagens formais que têm o objetivo de representar uma arquitetura conceitual de sistemas computacionais. Uma

ADL visa modelar a estrutura de componentes, os conectores e inter-relações entre eles [MEDVIDOVIC e TAYLOR 2000]. Garlan e Shaw [SHAW e GARLAN 1994] apresentam uma lista de propriedades que caracterizam uma ADL ideal:

- **Composição**: deve ser possível descrever um sistema como uma composição de componentes e de conexões independentes;
- **Abstração**: deve ser possível descrever os componentes e as interações em uma arquitetura de *software* de uma maneira que prescreva clara e explicitamente suas funções abstratas em um sistema;
- **Reusabilidade**: deve ser possível reusar componentes, conectores e padrões em diferentes descrições arquiteturais, mesmo se eles foram desenvolvidos fora do contexto da arquitetura do sistema;
- **Configuração**: descrições arquiteturais devem focalizar a descrição da estrutura do sistema, independente dos elementos sendo estruturados. Elas devem também suportar reconfiguração dinâmica;
- **Heterogeneidade**: deve ser possível combinar descrições arquiteturais múltiplas e heterogêneas;
- **Análise**: deve ser possível executar análises ricas e variadas de descrições arquiteturais. Por exemplo, checagem de tipos, análise de desempenho, uso de recursos, etc.

Existem vários centros de pesquisa que atualmente têm linhas de pesquisa voltadas para a construção de ADLs. Alguns exemplos de ADLs desenvolvidos na Universidade Carnegie Mellon são:

- ACME [CS-CMU 2003a] é uma linguagem simples e genérica para descrição de arquitetura que pode ser usada como um formato de intercâmbio para ferramentas de projeto de arquiteturas e / ou como base para o desenvolvimento de nova ferramenta de análise e projeto arquitetural. A linguagem ACME provê uma infra-estrutura genérica e extensível para descrição, representação, geração e análise de arquiteturas de *software*.
- O sistema Aesop [CS-CMU 2003b] é composto de ferramentas para produção rápida de projetos de arquitetura de *software* e ambientes de análise que são personalizados para suportar estilos arquiteturais de domínios específicos, possuindo um repositório para armazenamento, recuperação e reuso de elementos de projetos arquiteturais. Aesop possibilita também a cooperação com outras ferramentas.

Os próximos parágrafos dessa seção apresentam outras pesquisas sobre arquitetura de *software*.

David Garlan e outros [GARLAN et al 2004] apresentam um *framework* chamado *Rainbow*. Esta estrutura usa arquiteturas do *software* e uma infra-estrutura reusável para suportar a auto-adaptação de sistemas computacionais. O uso de mecanismos externos de adaptação permite a especificação explícita de estratégias de adaptação para sistemas múltiplos. A estrutura *Rainbow* adota uma abordagem baseada em arquitetura. Ela provê a infra-estrutura reusável junto com mecanismos para especializar essa infra-estrutura às necessidades de sistemas específicos. Estes mecanismos de especialização deixam o desenvolvedor de potencialidades de auto-adaptação escolher quais aspectos do sistema modelar e monitorar, que circunstâncias devem provocar a adaptação, e como adaptar o sistema. O objetivo chave e desafio preliminar desta estrutura é suportar o reuso de estratégias de adaptação e infra-estrutura para sistemas diferentes.

Sartipi e Kontogiannis [SARTIPI e KONTOGIANNIS 2003] apresentaram uma técnica para reconstrução de projetos de alto nível de sistemas computacionais a partir de código fonte. A técnica é baseada na combinação de padrões e em padrões arquiteturais definidos pelo usuário do sistema computacional. Os padrões arquiteturais são representados usando-se uma linguagem de descrição que é mapeada para um grafo relacional que permite especificar os componentes legados e seus dados, bem como as interações do fluxo de controle. Tal descrição de padrões é vista como perguntas que são aplicadas em um grafo entidade-relacionamento que representa a informação extraída do código fonte do sistema computacional. Um algoritmo multi-fase controla o processo de combinação dos dois grafos para o qual a pergunta é satisfeita e suas variáveis são instanciadas. Um mecanismo de escore baseado em associação é usado para pontuar os resultados gerados pelo processo de combinação.

O Departamento de Engenharia de *Software* e Ciência da Computação da Universidade de Karlskrona na Suécia [RISE 2003] possui um grupo que empreende pesquisas no sentido de definir uma técnica para descrever arquiteturas considerando-se requisitos de qualidade (e.g. flexibilidade, manutenibilidade e desempenho).

A Universidade do Sul da Califórnia [CSE-USC 2003] possui um grupo que estuda arquiteturas de *software* e seu impacto sobre todo o ciclo de vida dos sistemas computacionais. O objetivo maior do grupo é poder raciocinar sobre os atributos do *software* resultante baseado em sua arquitetura, onde por atributos entende-se custo, desempenho e confiabilidade.

Aldrich, Chambers e Notkin [ALDRICH et al 2002] desenvolvem ArchJava, uma extensão para Java que unifica a arquitetura do *software* com uma implementação orientada a objetos. Os autores declaram que o sistema de tipos ArchJava assegura que o código da implementação se adeque às restrições arquiteturais. É apresentado um estudo de caso que demonstra que ArchJava pode expressar dinamicamente alterações arquiteturais dentro do código de implementação e sugere que o programa resultante possa ser fácil de entender e desenvolver. Os autores declaram que a unificação proposta evita problemas tradicionais das abordagens existentes que separam implementação da arquitetura do *software*, permitindo inconsistências que causam confusão, violam propriedades arquiteturais e inibem a evolução do *software*.

MAISA (*Metrics for Analysis and Improvement of Software Architectures*) [VERKAMO 2003] é um projeto de pesquisa que desenvolve métodos e ferramentas para analisar automaticamente a qualidade da arquitetura de *software* e também para descobrir algumas propriedades centrais do *software* analisado. A análise é desenvolvida em nível de projeto e complementada com análises em nível de código do *software*.

2.2 Padrões

O conceito de Padrão (*Pattern*) surgiu a partir do trabalho do arquiteto Christopher Alexander que percebeu que todas as construções de edifícios, que eram funcionais e confortáveis, possuíam algumas características em comum. Alexander procurava o estabelecimento de Padrões para esses problemas recorrentes relacionados à arquitetura e engenharia civil [ALEXANDER 1977], [ALEXANDER 1979].

Na conferência OOPSLA'87, Kent Beck e Ward Cunningham apresentaram trabalho usando a idéia de Padrão de Alexander para projeto de GUI (*Graphical User Interface*) em Smalltalk [APPLETON 2000b].

O impulso na utilização desse conceito, aplicado à área de construção de *software*, tem como marco a publicação do livro intitulado “Design Patterns: Elements of Reusable Object-Oriented *Software*” de autoria de Erich Gamma et al (GoF - *Gang of Four*) [GAMMA et al 1995]. Este livro é um catálogo pioneiro que descreve Padrões de projeto para programas orientados a objeto. A partir deste trabalho, surgiram várias conferências (PLoP, EuroPLoP, Koala PLoP, Mensore PLoP, SugarLoafPLoP e ChiliPLoP) voltadas ao tema e também diversas publicações [APPLETON 2000a].

Existem várias definições para o termo “Padrão”. Algumas destas definições são:

- “Cada Padrão é pertinente a um problema que ocorre freqüentemente em nosso ambiente e descreve a estrutura de uma solução para este problema de tal maneira que se possa utilizar esta solução um milhão de vezes e nunca repeti-la sequer duas vezes da mesma maneira” [ALEXANDER 1979].
- “Um padrão é uma peça de literatura (texto) que descreve um problema de projeto e uma solução geral para o problema em um contexto particular” [COPLIEN 1996].

Para Brad Appleton, Padrão é um assunto recente em OOD (*Object Oriented Design*) e pode ser visto como [APPLETON 2000a]:

- Uma forma de identificação de boas estruturas que se repetem na prática;
- A descrição de soluções práticas para problemas do mundo real;
- Uma maneira de mostrar como e quando aplicar uma solução e gerar a estrutura desejada para um problema;
- Uma forma de documentar a resolução de problemas em engenharia de *software*.

A idéia central, em torno de Padrões, é capturar experiências que sejam comprovadamente corretas em desenvolvimento de *software*. Assim, cada Padrão está relacionado a um problema específico e recorrente no projeto ou implementação de *software*.

Segundo Buschmann e Coplien, as principais características dos Padrões são [BUSCHMANN et al 1996], [COPLIEN 1996]:

- Padrões são relacionados à solução para um problema recorrente em uma dada situação;
- Padrões descrevem um conjunto de componentes, classes ou objetos e suas responsabilidades e relacionamentos;
- Padrões provêem vocabulário e entendimento comum para o projeto, facilitando a aplicação e dispensando explicações detalhadas sobre um Padrão aplicado;
- Padrões auxiliam na construção de arquiteturas de *software* heterogêneas e complexas, possibilitada pela montagem de sistemas computacionais através de blocos;
- Padrões facilitam o gerenciamento da complexidade de *software*, por contribuírem para o particionamento do *software* em unidades (blocos).

Os Padrões podem ser classificados de acordo com sua aplicação em (i) Padrões de Análise (*Analysis Patterns*), (ii) Padrões de Projeto (*Design Patterns*), (iii) Padrões de Arquitetura (*Architectural Patterns*) e (iv) Idiomas (*Idioms*) [APPLETON 2000b].

(i) **Padrões de Análise** são Padrões específicos voltados à Análise do Sistema nos quais se enfoca a análise do domínio do problema. Costuma-se agrupar os Padrões de Análise por áreas de aplicação no mundo real, como: Sistema de Vendas, Contabilidade e Produção. É claro que há situações em que um Padrão pode ser reaproveitado em várias áreas. Os Padrões desenvolvidos por Fowler [FOWLER 1996] são utilizados para descrever soluções empregadas durante as fases de análise de requisitos e modelagem conceitual dos dados. Padrões de análise refletem estruturas conceituais do domínio da aplicação e não soluções computacionais. Em sistemas orientados a objeto, um Padrão de análise descreve um conjunto de classes, possivelmente pertencentes a diferentes hierarquias de classes, e as relações estruturais existentes entre elas.

Padrões de análise podem ser vistos, portanto, como uma forma de guiar a descrição de projetos durante o processo de modelagem de requisitos de muitas aplicações.

(ii) **Padrões de Projeto** de *software* descrevem uma família de soluções para um problema de projeto de *software*. Estes Padrões consistem de um ou vários elementos de projeto de *software*, tais como módulos, interfaces, classes, objetos, métodos, relações entre elementos e uma descrição do seu comportamento.

Alguns exemplos de Padrões de Projeto de *software* são Model/View/Controller, Blackboard, Client/Server e Process Control [BUSCHMANN et al 1996].

Um Padrão de Projeto deve ser "esperto"[2], genérico, bem elaborado, simples e reutilizável para sistemas orientados a objetos, conforme descrito a seguir [ERICKSSON 1997]:

- Esperto: é uma solução adequada na qual um projetista novato não pensaria de imediato;
- Genérico: não depende especificamente de uma linguagem de programação, um tipo de sistema ou domínio de aplicação;
- Bem Elaborado: já foi testado e aprovado em sistemas anteriores;
- Reutilizáveis: são documentados de tal maneira que tornam fáceis seu reuso.

[2] O termo “esperto” neste trabalho refere-se ao termo em inglês “smart”.

A maneira usada para descrever Padrões de projeto é chamada de "forma ou molde" (*template*). Alguns autores, como os citados a seguir, criaram "formas" para detalhar Padrões de projeto.

Existem algumas formas de especificação de padrões aceitas pela comunidade [GAMMA et al 1995], [BUSCHMANN et al 1996], [COPLIEN e SCHMIDT 1995], cada uma utilizando uma notação própria. As figuras representativas de um padrão também não possuem um consenso, sendo totalmente diversas as formas utilizadas, ficando a cargo do projetista do padrão a criação e utilização de figuras que melhor representem as partes componentes do padrão em desenvolvimento.

Segundo a "Gang of Four" [GAMMA et al 1995], uma forma de especificação aplicável aos Padrões de projeto tem a seguinte composição:

- Nome do Padrão: descreve a essência do Padrão. Um bom nome é vital, porque ele irá se tornar parte do vocabulário de projeto da equipe de desenvolvimento;
- Intenção: um parágrafo curto que responde as seguintes questões: O que o padrão de projeto faz? Qual sua razão e intenção? Para qual tipo de problema ele foi projetado?
- Apelido: qual outro nome ou apelido pode ser usado para este padrão de projeto?
- Motivação: um cenário que ilustra um problema de projeto e como a estrutura de classe e objeto no Padrão resolve o problema.
- Aplicabilidade: quais são as situações nas quais o Padrão pode ser aplicado com sucesso?
- Estrutura de composição: as classes e / ou objetos participantes do Padrão de projeto e suas responsabilidades.
- Colaborações: como os participantes colaboram para executar suas responsabilidades.
- Conseqüências: como os Padrões suportam seus objetivos? Quais são os custos e benefícios na utilização deste Padrão? Que aspectos da estrutura do sistema o Padrão irá afetar?
- Implementação: sobre que problemas, soluções ou técnicas deve-se estar consciente quando se implementa o Padrão? Existem questões específicas de linguagem?
- Código exemplo e utilização: fragmentos de código que ilustram como se deve implementar o Padrão.
- Usos conhecidos: exemplos (pelo menos dois) do Padrão encontrado no mundo real.

- Padrões relacionados: quais Padrões de projeto estão estreitamente relacionados com este? Quais as diferenças importantes? Com qual outro Padrão deveria este ser usado?

Buschmann e outros [BUSCHMANN et al 1996] apresentaram outra forma para a descrição de um Padrão de projeto que contém os seguintes itens:

- Nome do Padrão: ao se designar um bom nome a um Padrão automaticamente aumenta-se o vocabulário que passará a ser adotado por outros desenvolvedores. Assim, quando se quiser expressar uma solução, não há a necessidade de se explicar todo o processo de solução, mas sim apenas citar qual o Padrão a ser utilizado;
- Problema: descreve quando um Padrão deve ser aplicado explicando o problema em si e o contexto onde é encontrado. Em alguns casos o problema pode conter uma lista de condições que devem ser satisfeitas antes da utilização do Padrão;
- Solução: descreve os passos e a idéia da solução encontrada para o problema. Junto com esta solução, pode-se ter diagramas que auxiliam o seu melhor entendimento. Dentro destes diagramas, adota-se alguma notação conhecida para orientação a objeto tais como diagramas de classes, diagramas de estados e cenários;
- Conseqüências: são os resultados alcançados com a aplicação do Padrão. Através delas podem ser verificadas as possibilidades da utilização do Padrão para a solução do problema proposto no contexto especificado. Elas podem, inclusive, especificar a linguagem de programação utilizada, bem como peculiaridades ligadas diretamente à implementação. Uma vez que estas conseqüências podem afetar, também, o reuso, devem ser especificados os impactos ocasionados à flexibilidade, à extensibilidade e à portabilidade do sistema.

Coplien e Schmidt [COPLIEN e SCHMIDT 1995] apresentam um modelo que constitui-se de seis partes:

- Problema: descreve o problema a ser resolvido;
- Contexto: descreve o contexto no qual a solução descrita resolve o problema;
- Forças: apresenta o conjunto de fatores limitantes que atuam no problema;
- Solução: descreve a solução do problema descrito;
- Contexto Resultante: descreve o contexto resultante após a aplicação da solução;
- Racionalidade: descreve a razão e exemplos que justificam a solução.

(iii) **Padrões Arquiteturais** expressam esquemas de organização estrutural de fundamental importância para o desenvolvimento do *software*. Eles fornecem um conjunto de subsistemas pré-definidos, especificando suas responsabilidades e incluindo regras para os relacionamentos entre eles [BUSCHMANN et al 1996]. Padrões Arquiteturais representam os aspectos estruturais de um sistema computacional, por exemplo, a sua modularização e interface entre módulos. Neste trabalho, os padrões arquiteturais serão abordados em uma seção específica (seção 2.3).

(iv) **Idiomas** representam os Padrões de mais baixo nível na escala do desenvolvimento de *software*. Em contraste com os Padrões de projetos e arquiteturais, que se direcionam a princípios estruturais gerais, idiomas descrevem o modo de resolver problemas específicos de implementação de uma linguagem. Idiomas também podem mostrar diretamente a implementação concreta de um Padrão de projeto específico. Idiomas demonstram o uso competente das características da linguagem de programação, podendo ser aproveitados na tarefa de aprendizagem da linguagem.

Dentro do conceito de idioma, um estilo de programação é caracterizado pelo modo como as características da linguagem de programação são usadas na implementação de uma solução, assim como os tipos de comandos de controle utilizados, a nomeação de elementos de programa e todo o formato do código fonte.

Assim como os Padrões de projetos, os idiomas facilitam a comunicação entre desenvolvedores e agilizam o desenvolvimento e a manutenção do *software*. A coleção de idiomas de uma equipe de projetos forma um acervo intelectual da empresa.

Para um melhor esclarecimento sobre Padrões, alguns conceitos relacionados com Padrões são apresentados a seguir.

- Pacotes (*Packages*): um pacote é a unidade básica de produto no desenvolvimento de *software*. Qualquer coisa que se puder criar, manter, distribuir, vender, alterar e administrar genericamente como uma unidade separada em um sistema computacional, pode ser chamada de um pacote. Um pacote pode conter [D'SOUZA e WILLS 1998]:
  - ✓ Tipos e classes - especificações e implementações de objetos;
  - ✓ Código fonte e código compilado;
  - ✓ Colaborações - esquemas para interação de objetos;

- ✓ Padrões e *frameworks* - esquemas de tipos e colaborações que são abstraídos em um Padrão, opcionalmente incluindo código de implementação genérica, e que são usados em muitas partes de projetos;
- ✓ Requisitos - descrições do que é esperado do sistema ou componente;
- ✓ Diagramas - apresentação de elementos do modelo em várias formas visuais;
- ✓ Outros pacotes aninhados - como um agrupamento e construções macro-visíveis;
- ✓ Documentos de teste e verificação - especificação de comportamento correto esperado;
- ✓ Relatórios de erros - com sintomas, hipóteses das causas, justificativas, cenários e dados de teste para reproduzir o erro;
- ✓ Documentação narrativa acompanhando o modelo (preferencialmente em forma de hipertexto).

- *Model Framework*: são projetos, especificações, pedaços de código e outros artefatos de *software*, armazenados em bibliotecas e que podem ser subseqüentemente combinados em muitas configurações diferentes. Assim, a ferramenta básica para representar e combinar *frameworks* é uma forma genérica de pacote, chamada *Model framework* ou *template package* [D'SOUZA e WILLS 1998].
- *Framework*: é um pacote projetado para ser importado com substituições. Ele desdobra-se para prover uma versão de seu conteúdo que é especializado através das substituições feitas. Um *framework* pode instanciar a descrição de um Padrão, uma família de Padrões mutuamente dependentes, uma colaboração, um refinamento de Padrão, a própria construção do modelo e mesmo um conjunto de propriedades genéricas fundamentais. *Frameworks* podem ser reconstruídos em outros *frameworks*.

Em um nível mais básico, a estrutura dos *frameworks* representa a base para a organização de Padrões [D'SOUZA e WILLS 1998]. *Frameworks* são realizações físicas de uma ou mais soluções de Padrões. Padrões são as instruções de como implementar estas soluções. Os componentes são as unidades básicas de composição de padrões instanciados em estruturas *frameworks*. Os componentes envolvem apenas um pequeno número de classes enquanto um *framework* envolve vários componentes e classes que interagem. Um *framework* é geralmente projetado para prover uma solução completa enquanto um componente provê apenas um conjunto específico de serviços.

Um Padrão é um conjunto de idéias que pode ser aplicado em muitas situações. Assim, um *framework* está no "coração" de um ou mais Padrões; embora um Padrão possa conter outras informações tais como avisos de quando e onde ser usado. Quando se mantém um *framework* em uma biblioteca, ele deve ser empacotado com todas essas informações auxiliares [D'SOUZA e WILLS 1998].

Os próximos parágrafos dessa seção apresentam pesquisas atuais sobre padrões de *software*.

Blilie [BLILIE 2002] apresenta casos práticos onde os padrões podem ser utilizados beneficiando a produção de *software* científico. O autor declara que padrões são técnicas consagradas na arquitetura de *software* orientado a objetos, notadamente para os sistemas comerciais e que para os sistemas científicos pouco se utiliza padrões, visto que a tecnologia de objetos ainda não é tão utilizada.

Henninger [HENNINGER 2002] propôs a Web Semântica[3] como um meio para iniciar a representação dos relacionamentos entre padrões e rastrear quais são, freqüentemente, mais usados ou recomendados. Esta abordagem suporta o processo de encontrar padrões e também permite a construção de agentes que fazem os desenvolvedores saberem quando um dado padrão é aplicável. A justificativa indicada pelo autor é a de que padrões, particularmente de projeto e usabilidade, tornaram-se uma maneira popular de disseminar o estado corrente do conhecimento em certas áreas de desenvolvimento de *software*. Com isso, vários livros [ALEXANDER 1977], [GAMMA et al 1995], [BUSCHMANN et al 1996], [FOWLER 1996], [D'SOUZA e WILLS 1998], [LARMAN 1999], [CHEESMAN e DANIELS 2001] que abordam o tema padrões têm sido escritos e pessoas estão usando a abordagem de padrões para codificar "conhecimento", abrangendo desde práticas administrativas até padrões para avaliação de riscos. A explosão continuada de coleções de padrões tem causado dois problemas evidentes. O primeiro problema é relativo à qualidade e como saber se um padrão provê boas recomendações. O segundo problema é encontrar o padrão adequado para um problema em particular.

Pesquisas desenvolvidas por Suzuki e outros [SUZUKI 2003] criaram alguns formatos para intercâmbio de modelos UML. Suzuki relata uma linguagem chamada PML (*Pattern Markup Language*) que é um formato baseado em XML para descrever padrões de *software*. Este facilita a interoperabilidade entre ferramentas de desenvolvimento, intercomunicação entre desenvolvedores de *software* e extensão natural de um ambiente de

[3] A Web Semântica é a representação de dados na *World Wide Web* [BERNERS-LEE et al 2001].

desenvolvimento Web existente. A PML permite explicitamente codificar padrões de informação e auxilia a desenvolver facilmente ferramentas baseadas em padrões.

Noble e Biddle [NOBLE e BIDDLE 2002] proporcionam uma visão semiótica dos padrões de projeto, tratando um padrão como um símbolo (signo) constituído de intenções dos programadores e suas realizações no programa. Os autores declaram que considerar padrões como símbolos para referenciar muitas questões comuns concernentes a padrões de projeto pode auxiliar tanto programadores, no uso dos padrões, como autores, na escrita dos padrões. O artigo sugere que a semiótica auxilia a resolver problemas relacionados a padrões que incluem as diferenças entre padrões, formas variantes de padrões comuns, a identificação dos padrões, a organização de coleções de padrões e os relacionamentos entre padrões.

Ré et al [RÉ et al 2001] publicaram o artigo *A Pattern Language for Online Auctions Management* que contém uma linguagem de padrões voltados para administração de leilões em tempo real na Web. A linguagem é composta de dez padrões de análise que oferecem soluções para suportar todas as funcionalidades básicas do domínio de aplicação de leilões via Web.

Araujo e Weiss [ARAÚJO e WEISS 2002] publicaram o artigo *Linking Patterns and Non-Functional Requirements* (NFR) cujo objetivo é propor uma nova e complementar representação para padrões baseada na análise de requisitos não funcionais suportados por um *framework* NFR. A motivação desse novo formato é promover um melhor entendimento e estruturação dos padrões. Esta forma de representação de padrões provê suporte para desenvolver linguagens de padrões.

Corsaro et al [CORSARO et al 2002] em *Virtual Component A Design Pattern for Memory-Constrained Embedded Applications* apresentam um padrão que ajuda a reduzir o tamanho de memória para aplicativos *middleware*, particularmente aqueles baseados em CORBA e J2EE.

Welch et al [WELCH et al 2002] apresentam alguns padrões voltados para a administração de recursos de *software* tais como: monitoração de estado de recursos de computadores e redes.

Arsanjani [ARSANJANI 2002] publicou um conjunto de padrões que inclui os domínios: organizacional, metodologia, arquitetura e tecnologia de implementação voltados para arquitetura de serviços de Web.

Silva e Price [SILVA e PRICE 2002] apresentaram em *Component Interface Pattern*, um padrão que permite construir um componente com uma interface precisamente especificada por meio de pontos de vista estrutural e comportamental. O padrão é útil para produzir um componente que será conectado a outros componentes com estrutura de interface construída usando padrão de interface de componente.

Cross e Schmidt [CROSS e SCHMIDT 2002] publicaram uma coleção de padrões que permitem prover e especificar qualidade de serviços em sistemas embarcados e de tempo-real.

2.3 Padrões Arquiteturais / Estilos Arquiteturais

Um padrão arquitetural descreve uma solução genérica bem comprovada para um problema recorrente relacionado à estrutura e organização de *Software*. O esquema de solução é especificado pela descrição de suas partes constituintes, suas responsabilidades e relacionamentos, e as formas através das quais elas colaboram [BUSCHMANN et al 1996].

Um padrão arquitetural expressa um esquema de organização estrutural fundamental para sistemas computacionais, isto é, um padrão arquitetural está ligado a aspectos que abrangem o sistema todo e não apenas partes do sistema – como é o caso dos padrões de projeto. Um padrão arquitetural define a estrutura de *software*, define os seus componentes e conectores e define também as regras que regulam as relações entre os componentes através dos conectores.

A literatura apresenta alguns Padrões arquiteturais. O livro POSA1 [BUSCHMANN et al 1996] foi um pioneiro na descrição de padrões arquiteturais e traz alguns padrões como: *Layers-Analysis*, *Pipes and Filters*, *Blackboard*, *Broker*, *Model-View-Controller*, *Presentation-Abstraction-Control*, *Microkernel* e *Reflection*.

Outro livro POSA2 [SCHMIDT et al 2000] descreve vários padrões, embora estes padrões não sejam enquadrados exatamente como padrões arquiteturais, segundo a definição dada em POSA1. Alguns padrões descritos em POSA2 são: *Interceptor*, *Reactor*, *Proactor*, *Half-Sync/Half-Async* e *Leaders/Followers*.

Alguns dos padrões arquiteturais descritos na literatura podem ser vistos também no sítio Internet da empresa Rational [RATIONAL 2003a].

Padrões arquiteturais são denominados por Garlan e Shaw como “estilos arquiteturais”, querendo significar que os estilos são os modelos e os padrões arquiteturais são instâncias dos estilos [GARLAN e SHAW 1994]. Dessa forma, pode-se dizer que um estilo arquitetural é o meta-modelo (ou meta-padrão) de uma família de padrões arquiteturais visto que cada estilo arquitetural caracteriza uma família de padrões arquiteturais que têm as mesmas propriedades estruturais e semânticas [MONROE et al 1997].

A seguir são apresentadas algumas definições de estilo arquitetural feitas por autores e pesquisadores da área:

- Um estilo arquitetural define a estrutura organizacional de uma família de sistemas computacionais, detalhando a forma de comunicação dos componentes e conectores que pode ser usada em instâncias daquele estilo [GARLAN e SHAW 1994].
- Um estilo arquitetural é um conjunto coordenado de regras arquiteturais que restringem as funções / características dos elementos arquiteturais e dos relacionamentos permitidos entre esses elementos dentro de qualquer arquitetura que está em conformidade com este estilo [FIELDING 2000].
- Um estilo arquitetural é a descrição dos tipos de componentes e sua topologia. Um estilo inclui uma descrição do padrão de dados e controle de interação entre os componentes e uma descrição informal dos benefícios e desvantagens de se usar aquele estilo. Estilos arquiteturais são importantes artefatos de engenharia visto que eles definem classes de projetos com suas respectivas propriedades associadas. Os estilos oferecem evidência baseada em experiência de como cada classe tem sido usada historicamente, explicando porque cada classe tem suas propriedades específicas [KLEIN e KAZMAN 1999].

2.4 Componentização e Componentes de Software

O conceito de componentização de *software* está relacionado com a idéia de criação e reutilização de componentes de *software* na construção de novos sistemas computacionais.

Um componente é uma unidade de *software* independente que encapsula seu projeto e implementação e oferece interfaces bem definidas para comunicação com outras partes do *software* [D’SOUZA e WILLS 1998].

Embora os conceitos de componentes sejam parecidos com o de classes, existem algumas diferenças: (i) componentes são partes de implementação enquanto as classes representam abstrações lógicas; (ii) componentes possuem operações que são acessadas apenas através de suas interfaces, ao passo que as classes podem ter atributos e operações que podem ser acessadas diretamente (iii) um componente pode ter mais que uma classe.

A Figura 2.1 mostra um exemplo típico de componente constituído de duas interfaces. Ainda segundo D´SOUZA e WILLS [D'SOUZA e WILLS 1998], um componente deve ser genérico de forma a ser reutilizado em vários projetos de desenvolvimento de *software*. Os componentes, usados como meio de reutilização de partes de *software*, se relacionam com outros componentes através de conectores. Os conectores, também denominados interfaces, representam a ponte que liga os componentes, permitindo a interação entre os mesmos.

Figura 2.1 – Exemplo de Componente

Na componentização, um desenvolvedor (projetista) não precisa saber detalhes de como os componentes são construídos. Ao invés disso, o desenvolvedor precisa saber como utilizar os componentes, ou seja, precisa saber como são suas interfaces. Partindo dessa premissa, cada componente deve ter explícitos os seus propósitos e a maneira de reutilizá-lo.

De acordo com CARNEY apud GIMENES et al [GIMENES et al 2000], existem basicamente três tipos de componentes: componentes construídos por um usuário, componentes construídos para um domínio específico e componentes genéricos chamados COTS (*commercial off-the-shelf*).

De forma resumida, os componentes são importantes pelas seguintes razões [BASS et al 2001]:

- Componentes aumentam a produtividade de programadores que passarão a desenvolver *software* a partir de componentes previamente construídos.
- Componentes reduzem o tempo de inserção de um produto de *software* no mercado, pois muitos componentes já existem prontos, o que diminui o tempo necessário para se construir um *software*.

- Componentes provêm a base para comércio de partes de *software* reutilizáveis.

Os parágrafos seguintes, dessa seção apontam algumas pesquisas atuais sobre componentes e componentização.

O SEI (*Software Engineering Institute*) [SEI-CMU 2003c] empreende pesquisas envolvendo CBS (sistemas baseados em componentes comerciais de prateleira). Estas pesquisas focam os desafios de montar sistemas a partir de componentes preexistentes e os desafios para modificação de sistemas antigos visando aproveitar a estratégia CBS.

Ohori [OHORI 2003] relata que a indústria elétrica Oki e a JAIST têm se unido para empreender um projeto conjunto de cinco anos, com início em abril de 2003, objetivando estabelecer fundamentos teóricos e práticos para componentização segura de *software*

Bergner et al [BERGNER et al 2003] publicaram um artigo que apresenta algumas das experiências obtidas no ESSI Process Improvement Experiment SEPIOR. O objetivo do SEPIOR é a introdução sistemática de metodologias de desenvolvimento de *software* orientadas a objetos e orientadas a componentização. Estas metodologias são aplicadas em sistemas de manufatura assistida por computadores. O artigo focaliza a adoção dessas tecnologias na companhia de *software* SEKAS, onde são analisados os impactos comerciais, técnicos e humanos em um exemplo de desenvolvimento de um componente de administração de alarme.

Zenger [ZENGER 2002] apresentou um artigo que investiga abstrações para programação orientada a objetos em nível de linguagem de programação. O autor propõe um modelo simples para componentes no topo de uma linguagem orientada a objeto baseada em classes. O modelo é formalizado como uma extensão do *Java Featherweight*. O modelo inclui um conjunto mínimo de primitivas para construir dinamicamente, estender e criar componentes de *software*. O modelo também suporta características como dependências de contexto explícitas, composição, extensibilidade imprevista de componente e encapsulamento forte. O método apresenta um sistema de tipos seguro que garante definição, composição e evolução de componentes que deve ser usado de forma a garantir criação de componentes flexíveis e extensíveis.

2.5 UML – Unified Modeling Language

A Linguagem de Modelagem Unificada (UML) é uma linguagem modelo para a indústria para especificar, visualizar, construir e documentar sistemas computacionais. Usando a UML, pode-se modelar qualquer tipo de aplicação, para ser executada em várias combinações de *hardware*, sistema operacional, linguagem de programação e de rede. A flexibilidade da UML permite modelar aplicações distribuídas que usam algum *middleware* existente no mercado [RATIONAL 2003b], [OMG 1998]. Entretanto, a UML apresenta ainda algumas dificuldades tais como imprecisão da semântica, personalização limitada, inabilidade para intercambiar diagramas e suporte ineficiente para modelagem de componentes.

Há atualmente diversos pesquisadores elaborando e discutindo propostas de melhorias para a nova versão 2.0 da UML (em andamento). As melhorias que estão por vir deverão, entre outras: reestruturar e refinar a linguagem tornando-a fácil para aplicar, implementar e personalizar; criar um pacote central que representa o núcleo dos conceitos de modelagem da UML; melhorar o suporte para desenvolvimento baseado em componentes; refinar as capacidades de especificação arquitetural; aumentar a escalabilidade, precisão e integração dos diagramas de comportamentos [KOBRYN 2003], [U2 PARTNERS 2003].

Nesta linha de pesquisas que visam agregar melhorias à UML, Suzuki e Yamamoto [SUZUKI e YAMAMOTO 1999] descrevem uma extensão de meta-modelos UML para a especificação reflexiva de componentes de *software*. Reflexão é o princípio de projeto que permite a um sistema ter a representação de si mesmo em uma maneira que o faz facilmente se adaptar para um ambiente em mudança. Segundo os autores o trabalho permite reconhecer e entender componentes reflexivos nos níveis altos de abstração em estágios iniciais do processo de desenvolvimento. O trabalho apresentado permite alavancar a documentação, aprendizado, modelagem visual, reuso e desenvolvimento progressivo e regressivo dos projetos em meta níveis.

Sullivan [SULLIVAN 2003] aborda as melhorias que a nova versão da UML que será submetida à OMG (*Object Management Group*) deverá ter. A nova versão incluirá melhorias que facilitarão o desenvolvimento dirigido a modelos. Entretanto, as transformações serão transparentes para muitos desenvolvedores. Pretende-se trazer à UML uma linguagem de alto nível que transcenda muitas linguagens textuais permitindo tanto a geração de código quanto a engenharia reversa, talvez até o ponto de executabilidade direta em alguns modelos UML.

Hansen [HANSEN 2003] também aborda sobre as novas funcionalidades da UML 2.0. Hansen afirma que o *Object Management Group* (OMG) estendeu ainda mais a especificação da UML, permitindo aos desenvolvedores de *software* descreverem de forma mais clara as funcionalidades de suas aplicações através da modelagem das transações de código complexo. Diagramas de seqüência na UML 2.0 agora fornecem os meios para se modelar "fragmentos combinados" dentro dos próprios diagramas de seqüência. Esta extensão possibilitará expressar componentes lógicos – tais como alternativas, opções, exceções, ligações paralelas, laços, negações, regiões críticas e asserções – diretamente nos diagramas de seqüência. Além disso, as capacidades dos diagramas de seqüência estendidos permitem a usuários aninharem diagramas de seqüência e adornarem seus códigos fonte com capacidades de modelagem mais detalhadas.

2.6 Modelo de Requisitos

A especificação de requisitos de um sistema computacional é uma das atividades mais importantes para assegurar o sucesso do seu desenvolvimento, pois as fases seguintes no processo do desenvolvimento dependem da qualidade desta especificação [LARMAN 1999], [STEELTRACE 2002]. O levantamento, a verificação e a especificação mal sucedidos dos requisitos podem conduzir à entrega de um produto de *software* que não esteja de acordo com as necessidades do usuário final. Outra conseqüência é que o *software* provavelmente será produzido com atraso e extrapolando o orçamento previsto.

A especificação de *software* que objetiva o reuso deve levar em consideração a premissa de que a atividade de especificação adequada dos requisitos de *software* terá forte influência no sucesso de seu reuso.

Particularmente para os padrões arquiteturais, é importante a correta especificação dos requisitos que caracterizam as suas funcionalidades, uma vez que os padrões servem como modelos a serem seguidos. A conseqüência direta da existência de um padrão é o fato que os sistemas criados a partir do padrão arquitetural deverão implementar as funcionalidades que representam os requisitos prescritos no padrão.

Uma forma de modelar os requisitos de um sistema computacional é através de modelo de casos de uso [BITTNER e SPENCE 2003]. O modelo de casos de uso de um sistema apresenta uma visão que mostra os requisitos do sistema modelados através de ícones que representam casos de uso e atores.

O termo Caso de Uso foi apresentado pela primeira vez por Ivar Jacobson [JACOBSON 1987] na conferência OOPSLA'87. Para Jacobson um Caso de Uso é uma seqüência de transações executadas por um usuário e um sistema em um diálogo. Para Booch e outros autores, um Caso de Uso é a especificação de um conjunto de seqüências de ações que um sistema executa para produzir um resultado de valor para os atores. Um Caso de Uso é representado graficamente por uma elipse nomeada internamente ou logo abaixo da elipse. *E.g.* Figura 2.2 [BOOCH et al 2000], [JACOBSON et al 1999], [BITTNER e SPENCE 2003].

Figura 2.2 – Exemplo de Caso de Uso

Um ator representa um conjunto coerente de papéis que os usuários de casos de uso desempenham quando interagem com esses casos de uso. Tipicamente, um ator representa um usuário, um dispositivo de hardware ou até outro sistema que interage com o sistema. Os atores, embora sejam utilizados na modelagem, não fazem parte do sistema [BOOCH et al 2000], [JACOBSON et al 1999], [BITTNER e SPENCE 2003]. Um ator é representado graficamente por um boneco (em inglês: *stickman*) nomeado logo abaixo. *E.g.* Figura 2.3.

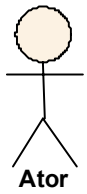


Figura 2.3 – Exemplo de Ator

Um Modelo de Casos de Uso é o modelo descritivo de um sistema composto por todos os atores e casos de uso e os relacionamentos entre eles. Os relacionamentos entre atores e casos de uso são representados por linhas que os ligam. O propósito de um Modelo de Casos de Uso é resumir o que o sistema deve fazer, permitindo a desenvolvedores de *software* e a usuários chegarem a um consenso com relação aos requisitos do sistema [JACOBSON et al 1999], [BITTNER e SPENCE 2003]. Também, segundo Booch e outros, os diagramas de casos de uso são importantes principalmente para a organização e modelagem dos comportamentos de um sistema [BOOCH et al 2000]. A Figura 2.4 mostra um exemplo simples de um Modelo de Casos de Uso para um sistema de compras pela Internet.

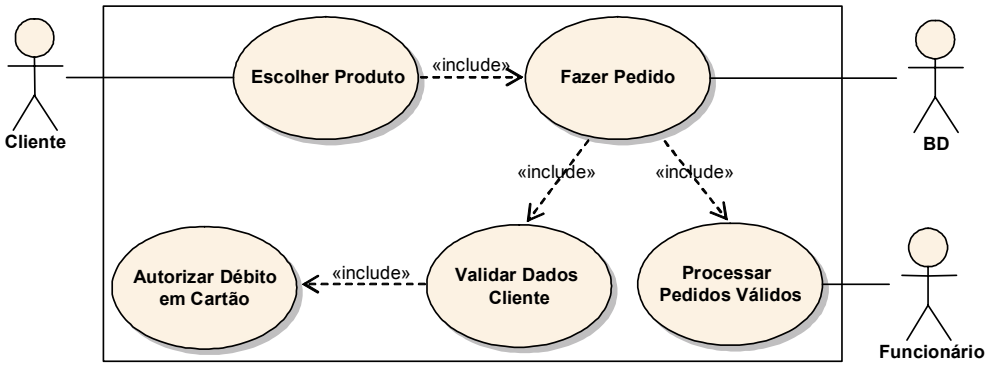


Figura 2.4 – Exemplo simples de Modelo de Casos de Uso

Um Modelo de Caso de Uso pode ser dividido em vários diagramas chamados Diagramas de Casos de Uso. Essa divisão é aconselhada caso um Modelo de Caso de Uso se torne muito extenso [BOOCH et al 2000], [BITTNER e SPENCE 2003], [JACOBSON et al 1999].

2.7 Análise de Domínio

O conceito de Análise de Domínio foi introduzido por Neighbors [NEIGHBORS 1980]: "A Análise de Domínio é uma tentativa de identificar objetos, operações e relações entre o que peritos em um determinado domínio percebem como importante".

A Análise de Domínio é um processo que está inserido no contexto da Engenharia de Domínio (Veja Figura 2.5). A Engenharia de Domínio compreende a definição, análise, especificação da estrutura e construção de componentes voltados para domínios [SEI-CMU 2003b].

Outro conceito relacionado à Análise de Domínio é o próprio conceito de Domínio. Genericamente um Domínio representa um campo de atividade ou interesse. No contexto da Engenharia de *software*, mais especificamente, um Domínio representa uma área de aplicação, um campo para o qual sistemas computacionais, que compartilham um conjunto de requisitos comuns, são desenvolvidos [PRIETO-DIAZ 1990], [BERARD 1992], [ROSETI e WERNER 1999].

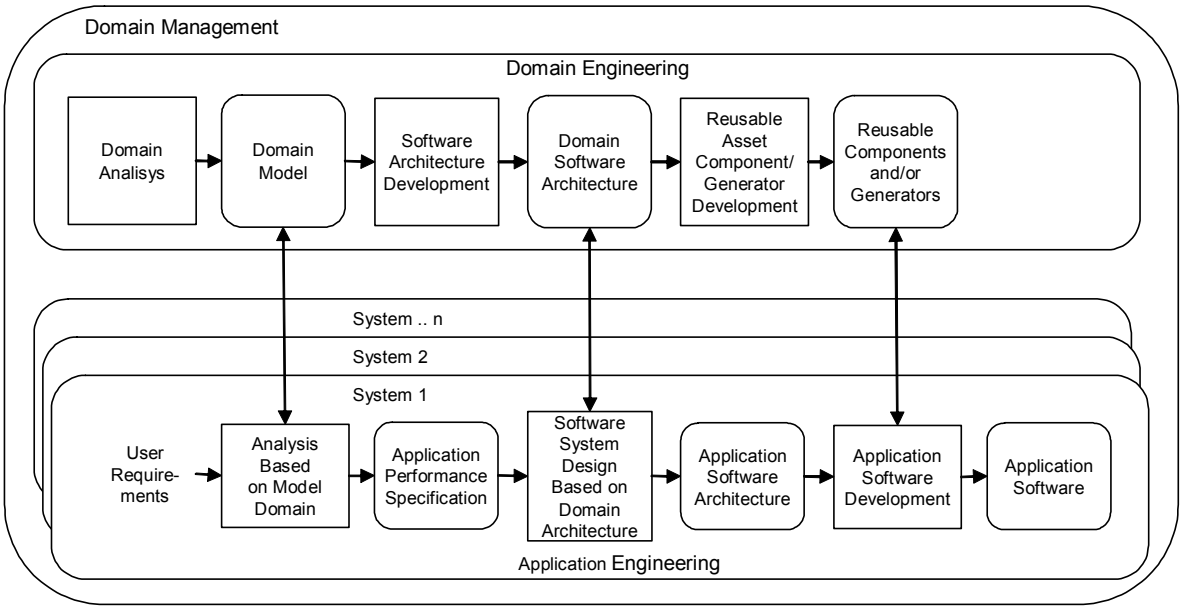


Figura 2.5 – Ciclo de vida da Engenharia de Domínio e da Engenharia de Sistemas [SEI-CMU 2003b].

O processo da Análise de Domínio é voltado para a descoberta de características que dizem respeito a uma classe de sistemas equivalentes e não a um sistema específico. O processo consiste de três passos básicos que são repetidos para a descoberta de diferentes tipos de componentes para uma classe de sistemas [PRIETO-DIAZ 1990]:

1. Identificação de entidades (partes) reutilizáveis;
2. Abstração ou generalização;
3. Classificação e catalogação para reuso posterior.

A Análise de Domínio possui atributos de escala, ou seja, o processo da Análise de Domínio é voltado para o reuso em larga escala através da captura de partes semelhantes dos sistemas circunscritos em um domínio e agrupando essas partes em modelos de domínios, com o objetivo de facilitar o reuso. Dessa forma a Análise de Domínio visa a construção de padrões de sistemas e não de sistemas isolados [PRIETO-DIAZ 1990], [JACOBSON et al 1997], [FOREMAN 1996].

A Análise de Domínio possui algumas características que a diferencia da Análise de Sistemas:

- A Análise de Domínio está para a Engenharia de Domínio assim como a Análise de Sistemas está para a Engenharia de Sistemas.
- A Análise de Domínio é concernente preferivelmente com as ações e objetos que ocorrem em todos os sistemas circunscritos em uma área de aplicação (domínio), enquanto que a Análise de Sistemas é concernente com as ações específicas para um sistema particular.

- A Análise do Domínio é útil quando sistemas similares devem ser construídos, de modo que o custo da Análise do Domínio possa ser amortizado entre todos os sistemas. Assim, para a construção de um único sistema é mais indicado usar a Análise de Sistema.

2.8 Modelo e Metamodelo

Modelo: Genericamente um modelo é uma forma típica para reproduzir ou imitar [PRIBERAM 2004]. No contexto da engenharia de *software* um modelo se refere a uma forma de descrição esquemática de um sistema computacional. Um sistema computacional pode ser representado através de modelos. Dessa forma, cada modelo de um sistema apresenta uma visão característica do sistema.

Metamodelo: Segundo o dicionário Webster um Metamodelo pode ser visto como um modelo que define os componentes de um modelo conceitual, processo ou sistema [WEBSTER'S 2004]. Outra definição considera um metamodelo como uma acepção precisa dos conceitos e regras necessárias para criar modelos semânticos [METAMODEL 2004a]. Outra autora [TANNENBAUM 2001] define um metamodelo como um conjunto de metadados inter-relacionados utilizados para definir modelos. Um metamodelo define formalmente os elementos (“classe”, “atributo”, “tabela”, etc) do modelo, sua sintaxe e semântica. A Figura 2.6 exemplifica um metamodelo para componentes. Este metamodelo define a estrutura de construção dos componentes básicos que podem compor um modelo de componentes.

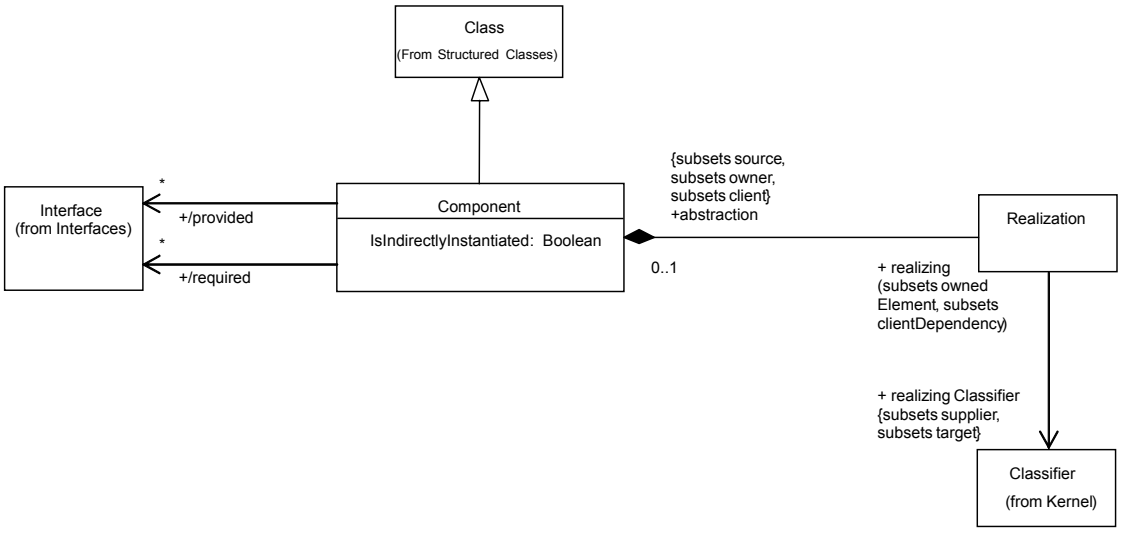


Figura 2.6 – Um exemplo de metamodelo [OMG 2003].

2.9 Considerações Finais

Este capítulo apresentou a fundamentação teórica do trabalho proposto. Foram descritos conceitos importantes como: arquitetura de *software* que forma a espinha dorsal de todo sistema computacional; padrões que têm se mostrado como um meio eficiente para reutilização de *software*; padrões arquiteturais que são tipos de padrões em nível mais alto de abstração, e estilos arquiteturais que são meta padrões arquiteturais que definem a estrutura organizacional de famílias de sistemas computacionais; componentes de *software* que são unidades básicas de comercialização de partes de *software* e componentização de *software* que pode ser vista como a aplicação da engenharia de *software* objetivando construir unidades componentes e também montar *software* a partir das unidades de componentização; a UML é aceita mundialmente como a linguagem de modelagem de *software*; os Casos de Uso da UML são mencionados separadamente por serem parte intrínseca deste trabalho; a Análise de Domínio que possibilita o reuso adequado de informações referentes a domínio e também Modelo e Metamodelo, visto que a representação de conceitos deste trabalho os usam.

3 Tipologias Propostas

The greatest difficulty is not in accepting new ideas, but getting rid of the old ones.

[John Maynard Keynes]

Este capítulo apresenta as tipologias definidas e utilizadas neste trabalho para representar elementos genéricos nos diversos modelos utilizados. São apresentadas: na seção 3.1 a tipologia que se aplica na especificação dos requisitos através dos casos de uso e atores genéricos, na seção 3.2 a tipologia utilizada na especificação das classes genéricas, na seção 3.3 a tipologia específica para os diagramas de seqüência genéricos, na seção 3.4 a tipologia para os diagramas de estados genéricos. E, por fim, a seção 3.5 apresenta a tipologia para os diagramas de componentes genéricos.

3.1 Extensão do Modelo de Casos de Uso para Padrões Arquiteturais

Apesar da UML ser a linguagem mais utilizada atualmente para a especificação de sistemas computacionais, contando com uma variedade de modelos que se prestam à especificação de diferentes visões de um projeto de *software*, o modelo de casos de uso da UML não permite representar elementos variantes de maneira adequada na especificação de soluções genéricas para problemas recorrentes. Estas soluções freqüentemente incluem

partes opcionais, múltiplas e partes que podem variar entre as instâncias ou especializações das soluções.

A fim de aplicar o modelo de casos de uso na especificação de requisitos de padrões arquiteturais, uma extensão para o diagrama clássico do modelo de casos de usos da UML foi proposta. Esta extensão permite expressar as abstrações necessárias para a descrição dos requisitos dos padrões arquiteturais.

Neste trabalho, denomina-se Diagrama de Casos de Uso Genérico aqueles diagramas que são usados para especificar requisitos de um padrão arquitetural a ser empregado em várias instâncias diferentes de aplicação e que, portanto, conterá especificações que nem sempre são as mesmas em todas as instâncias daquele padrão.

Durante a modelagem de um padrão, o projetista se depara com necessidades variadas em cada instância onde o padrão é utilizado. Dividindo-se um padrão arquitetural em partes, pode-se dizer que existem partes desse padrão que serão opcionais, pois dependendo da instância, esta parte será ou não utilizada. Existem outras partes desse padrão que serão variantes, ou seja, a forma como essa parte é implementada nas instâncias varia. Existem, ainda, outras partes que poderão existir em multiplicidade dependendo da necessidade da instância onde o padrão está sendo empregado. Dessa forma, este trabalho inclui os termos: Opcional, Variante e Múltiplo que serão utilizados na descrição dos requisitos dos padrões arquiteturais.

Esta seção descreve uma extensão para o modelo de casos de uso da UML [QUINÁIA e STADZISZ 2003]. A extensão compreende três estereótipos básicos (nomeados: opcional, variante e múltiplo) e algumas combinações deles, desenvolvidos para serem usados na modelagem dos requisitos de padrões arquiteturais. Estes estereótipos são propostos como um avanço sobre conceitos tradicionais usados para representar atores e casos de uso em UML.

3.1.1 Tipologia para Atores e Casos de Uso

Atores e Casos de Uso Clássicos

Atores e casos de uso clássicos podem ser usados na modelagem de padrões conforme são descritos na UML [RUMBAUGH et al 1998]. Os atores e casos de uso clássicos representam, respectivamente, atores específicos (entidades externas) e casos de uso específicos (serviços ou funcionalidades do sistema) que estarão presentes em todas as instâncias do padrão. A figura 3.1 ilustra a notação clássica da UML para atores e casos de uso.

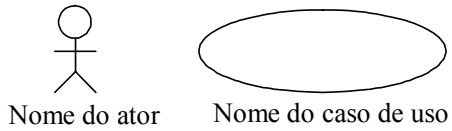


Figura 3.1 – Notação para Atores e Casos de Uso segundo a UML

Atores e Casos de Uso Opcionais

Um ator opcional é uma entidade externa ao sistema que pode ou não ser incluída em instâncias do padrão dependendo das características específicas do sistema que está sendo projetado. Um exemplo de ator opcional é um administrador de sistema que pode não existir em algumas instâncias de um dado padrão arquitetural.

Da mesma maneira, um caso de uso opcional indica uma funcionalidade que pode ou não existir em um sistema que é modelado usando o padrão. Um exemplo do caso de uso opcional é um diálogo de início de uma sessão (*login*). Algumas instâncias dos padrões arquiteturais podem não incluir este serviço enquanto outras instâncias podem requerer a identificação de usuário por meio deste caso de uso.

A notação adotada usa o rótulo ou estereótipo **op** e um retângulo em volta do ator para indicar que esse ator é opcional. Para o caso de uso opcional usa-se apenas o rótulo **op** dentro da elipse que representa o caso de uso. A figura 3.2 mostra a notação proposta para atores e casos de uso opcionais.

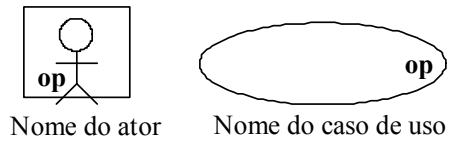


Figura 3.2 – Notação para Atores e Casos de Usos Opcionais

Atores e Casos de Uso Variantes

Um ator variante expressa uma entidade externa genérica do padrão arquitetural. Por Entidade Externa Genérica entende-se uma entidade que existe em toda instância do padrão, porém suas propriedades ou características variam entre as instâncias do padrão. Um exemplo de ator variante pode ser um *software* externo implementado em diferentes versões.

Um caso de uso variante representa um serviço ou uma funcionalidade variante. Trata-se de uma abstração ou uma unificação de um conjunto de casos de uso similares. Como um exemplo de caso de uso variante, pode-se ter um serviço para estabelecer conexão entre computadores por meio de uma rede que possa ser de tipos ou tecnologias diferentes. Um outro exemplo é um serviço de pagamentos que possa diferir na maneira

pela qual o pagamento de um cliente é processado. O serviço básico ou a funcionalidade do caso de uso variante estará sempre presente, mas muda seguindo aspectos específicos de cada instância do padrão.

A notação adotada para atores e casos de uso variantes aplica um rótulo ou estereótipo **var** ao lado do ícone correspondente (figura 3.3).

Figura 3.3 – Notação para Atores e Casos de Usos Variantes

Ator e Caso de Uso Múltiplo

Um ator ou caso de uso múltiplo representa um ator ou um caso de uso que pode aparecer uma ou mais vezes nos sistemas computacionais projetados a partir do padrão.

Um ator múltiplo expressa um conjunto de atores do mesmo tipo, isto é, esses atores têm papel similar com relação aos casos de uso aos quais estão associados. Para cada instância do padrão, o número exato de atores relacionados a um dado ator múltiplo depende das características específicas do sistema que está sendo projetado. Pelo menos um ator relacionado a cada ator múltiplo deve estar presente em cada instância do padrão. Exemplos de atores múltiplos são usuários de um sistema ou as suas impressoras. Pode haver vários usuários do sistema ou várias impressoras em um dado sistema, que estão interagindo com o sistema ao mesmo tempo. Pelo menos um usuário ou impressora são requeridos.

Um caso de uso múltiplo indica uma funcionalidade do padrão que pode ter múltiplas execuções independentes em várias *threads* ou em processadores diferentes, nas instâncias que utilizam o padrão. O número exato de execuções independentes depende das características específicas do sistema que está sendo projetado. Um exemplo de caso de uso múltiplo pode ser um serviço de controle de dispositivo usado para controlar um número de dispositivos externos. Para cada instância do padrão haverá um número específico de serviços de controle de dispositivo dependendo do número de dispositivos externos a serem controlados.

A notação adotada para atores e casos de uso múltiplos é um fundo acinzentado no ícone correspondente (figura 3.4).

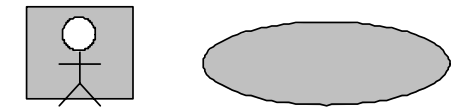


Figura 3.4 – Notação para Atores e Casos de Usos Múltiplos

3.1.2 Combinação de Tipos

Os três classificadores propostos (opcional, variante e múltiplo) podem ser reunidos para permitir combinações dos tipos como descrito a seguir.

Ator e Caso de Uso Múltiplo e Opcional

Um ator múltiplo e opcional é uma combinação de tipos que representa, ao mesmo tempo, um ator que possa ser opcional e múltiplo. Um exemplo dessa combinação é um dispositivo que possa ou não estar presente (característica de opcionalidade) para um sistema que esteja sendo projetado. Se presente, o número de dispositivos (atores) varia entre as instâncias do padrão (característica de multiplicidade).

Similarmente, os casos de uso opcionais e múltiplos são aplicáveis quando pode haver um número variante de execuções simultâneas de um mesmo caso de uso. Entretanto, a funcionalidade não é absolutamente requerida em cada instância do padrão. O caso de uso é de fato opcional. Por exemplo, um certo relatório pode não ser gerado em uma instância do padrão, mas pode estar disponível para múltiplas execuções simultâneas em outras instâncias daquele mesmo padrão.

A notação adotada usa o fundo acinzentado (para denotar multiplicidade) e o rótulo **op** (para denotar opcionalidade), como ilustrado na figura 3.5.

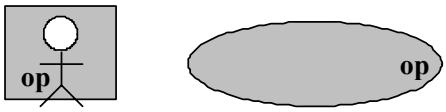


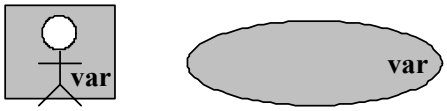
Figura 3.5 – Notação para Atores e Casos de Usos Múltiplos e Opcionais

Ator e Caso de Uso Variante e Múltiplo

Um ator que seja variante e múltiplo ao mesmo tempo significa que ele está sempre presente mas suas características e cardinalidade mudam entre as instâncias do padrão. Um exemplo de ator que seja variante e múltiplo ao mesmo tempo pode ser uma impressora que sempre é requerida para uma dada família de sistemas (representadas por um padrão arquitetural) mas que pode estar presente em diferentes modelos e quantidades para cada instância do padrão.

De forma similar, um caso de uso variante e múltiplo representa uma funcionalidade que está sempre presente nas instâncias do padrão, mas cujas características variam entre as instâncias do padrão, e existe a possibilidade de ocorrências múltiplas de execuções simultâneas para a mesma funcionalidade. Por exemplo, um relatório pode variar na forma como ele é apresentado, ou seja, cada instância do padrão pode apresentar esse relatório de forma diferente, visto que o mesmo é variante. Ao mesmo tempo, este relatório está disponível para múltiplas execuções simultâneas.

A notação adotada usa o fundo acinzentado (para denotar multiplicidade) e o rótulo **var** (para denotar variabilidade), conforme ilustra a figura 3.6.



Nome do ator Nome do caso de uso

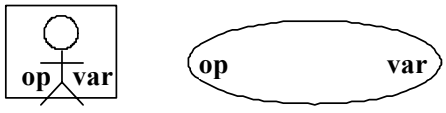
Figura 3.6 – Notação para Atores e Casos de Usos Variantes e Múltiplos

Ator e Caso de Uso Opcional e Variante

Um ator Opcional e Variante representa uma entidade externa ao sistema, que é opcional para as instâncias do padrão, ao mesmo tempo em que as suas características variam. Um exemplo de ator opcional e variante pode ser um SGBD, que pode ser usado (característica de opcionalidade) em um sistema computacional e que seja fornecido por diferentes fornecedores (característica de variabilidade).

Um Caso de Uso opcional e variante representa um serviço opcional que pode ser instanciado de diferentes formas. Se estiver presente em uma dada instância do padrão, suas características são específicas para aquela instância. Como exemplo de um caso de uso que seja opcional e variante ao mesmo tempo, pode-se ter um caso de uso referente a um módulo para pagamento com cartão. Esse caso de uso pode ou não ser implementado nas instâncias do padrão, sendo que as características desse caso de uso são variantes nas instâncias onde for implementado.

A notação adotada para representar atores e casos de uso que sejam variantes e opcionais ao mesmo tempo é ilustrada na Figura 3.7.



Nome do ator Nome do caso de uso

Figura 3.7 – Notação para Atores e Casos de Usos Opcionais e Variantes

Ator e Caso de Uso Múltiplo, Variante e Opcional

Um ator múltiplo, variante e opcional é uma entidade externa variante que pode aparecer ou não em cada instância do padrão, e que tem uma cardinalidade maior ou igual a um. Como um exemplo, considere uma impressora que possa estar ou não presente em um sistema. Se estiver presente, pode variar no tipo. Além disso, pode haver uma ou mais impressoras acopladas ao sistema.

Um exemplo de caso de uso múltiplo, variante e opcional pode ser uma tela de início de uma sessão que pode ser opcional em instâncias específicas do padrão. Esta funcionalidade, quando requerida, pode variar em algumas versões diferentes e pode ser realizada em várias *threads* ou processadores para usuários diferentes ao mesmo tempo.

A Figura 3.8 mostra a notação para esta combinação, tanto para ator quanto para caso de uso.

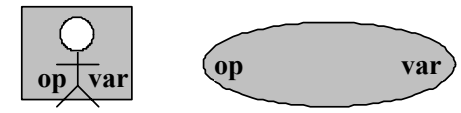


Figura 3.8 – Notação para Atores e Casos de Usos Múltiplos, Variantes e Opcionais

3.1.3 Relacionamentos entre Atores e Casos de Uso

A aplicação da extensão proposta impõe algumas limitações no relacionamento entre atores e casos de uso. Esta seção discute os relacionamentos e as limitações que podem ocorrer ao se usar a extensão para especificação de requisitos de padrões arquiteturais.

O relacionamento mostrado na Figura 3.9 não pode ocorrer visto que o ator é comum (sempre presente em qualquer instância do padrão) e o caso de uso é opcional. Como o caso de uso é opcional, ele pode ou não estar presente no sistema instanciado a partir do padrão. Se o caso de uso não estiver presente, então o ator estará isolado, sem nenhum relacionamento com serviços do sistema.

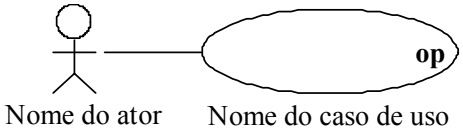


Figura 3.9 – Relacionamento inválido

Os relacionamentos ilustrados na Figura 3.10 sofrem o mesmo problema do caso precedente mostrado na Figura 3.9. Em outras palavras, os casos de uso são todos opcionais e, assim, não há nenhuma garantia de que eles aparecerão nos sistemas

instanciados a partir do padrão. Conseqüentemente, os atores poderiam estar isolados e esta situação não faz sentido.

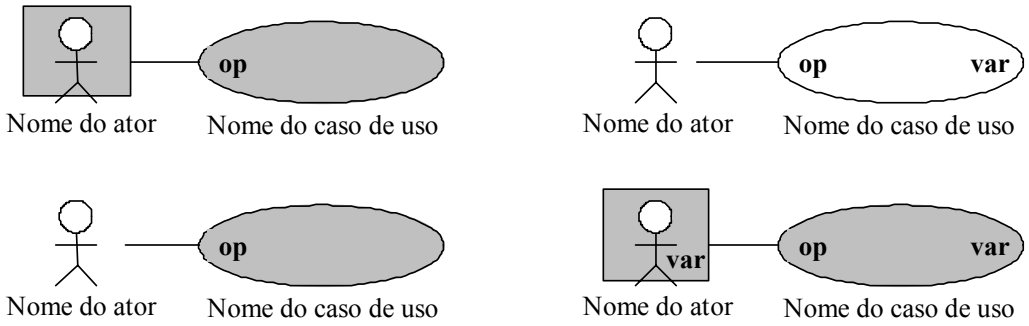


Figura 3.10 – Outras relações inválidas

Embora a Figura 3.11 exemplifique um relacionamento entre um ator comum (que deve aparecer em cada instância do padrão) e um caso de uso opcional (que aparece opcionalmente), este mesmo ator tem também outro relacionamento com um caso de uso comum. Isto permite aceitar esta configuração de associações. Quando o caso de uso opcional não for implementado em nenhum sistema projetado a partir do padrão, o ator não ficará isolado visto que esse ator tem também um outro relacionamento com um caso de uso comum.

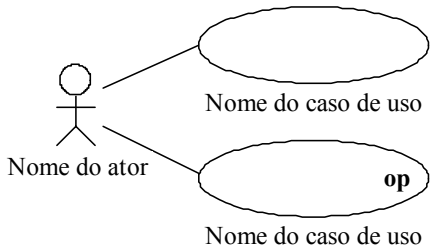


Figura 3.11 – Relacionamentos permitidos entre atores e casos de uso

Podem também ocorrer outros relacionamentos diferentes dos mostrados nas Figuras 3.9, 3.10 e 3.11. Esses outros relacionamentos não são aqui abordados, pois os exemplos mostrados já são suficientes para o escopo do trabalho proposto.

3.1.4 Relacionamentos entre Casos de Usos

A UML tem três tipos de relacionamentos (inclusão, extensão e herança) que podem ser aplicados aos casos de uso. Adicionalmente às associações entre atores e casos de uso, pode ocorrer a necessidade de descrever os relacionamentos entre casos de uso quando se está projetando padrões arquiteturais usando a extensão proposta. Existem algumas limitações nos relacionamentos entre os casos de uso devido a sua tipologia. O diagrama de casos de uso da Figura 3.12 ilustra uma situação possível (adaptado de [RUMBAUGH et al 1998]).

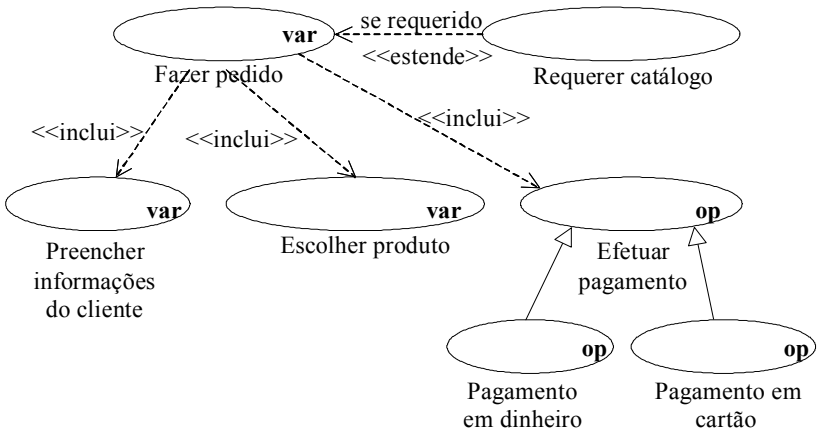


Figura 3.12 – Relacionamentos entre Casos de Usos

O caso de uso principal é **Fazer pedido**. Como existem subcasos de uso, ligados a este caso de uso, que são variantes e opcional, então este caso de uso principal é também variante. Isto pode ser verificado porque (i) os casos de uso **Preencher informações do cliente** e **Escolher produto** são variantes; (ii) o caso de uso **Efetuar pagamento** é opcional (**op**) e, conseqüentemente, as especializações **Pagamento em dinheiro** e **Pagamento em cartão** são opcionais também por causa do relacionamento de generalização.

Do parágrafo anterior pode-se inferir as seguintes regras (ou restrições):

REGRA 1: Para cada subcaso de uso que for variante ou opcional o caso de uso principal correspondente deverá ser variante.

REGRA 2: Nos casos de herança entre casos de uso, as especializações herdam o estereótipo da generalização.

3.1.5 Aplicando a Extensão de Caso de Uso na Especificação de Padrões Arquiteturais

Para ilustrar a aplicabilidade da extensão do modelo de caso de uso descrita, a Figura 3.13 mostra seu uso na especificação dos requisitos de um padrão arquitetural. Este padrão modela os sistemas cujo objetivo é controlar o processo de empréstimo de obras em uma biblioteca. O modelo de casos de uso não está vinculado a um sistema computacional específico mas a um modelo abstrato (modelo de referência) para uma família de sistemas com finalidades similares.

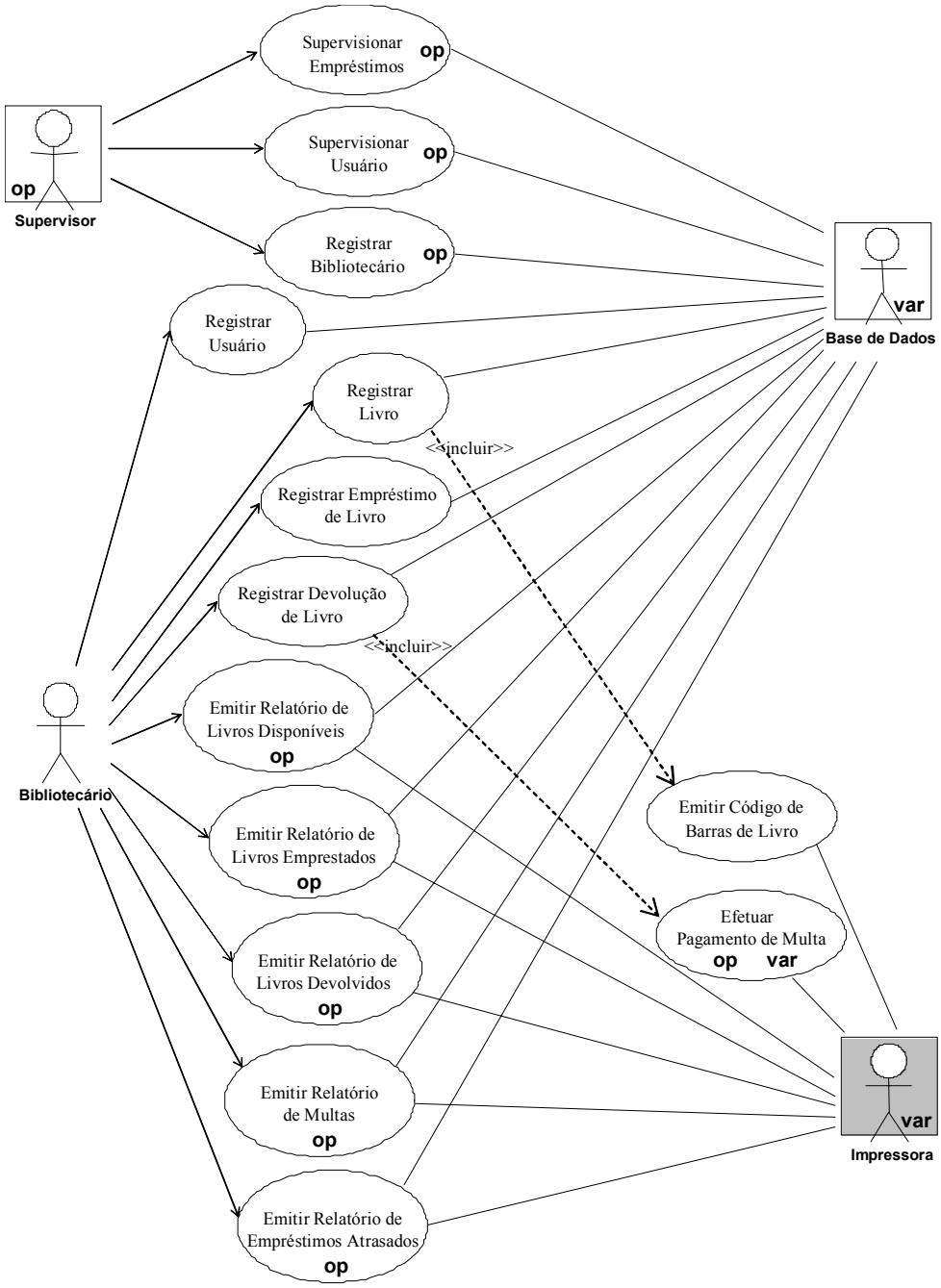


Figura 3.13 – Exemplo de um Modelo de Casos de Uso Genérico

Basicamente, este padrão fornece doze serviços modelados nos seguintes casos de uso: Supervisionar Empréstimos, Supervisionar Usuário, Registrar Bibliotecário, Registrar Usuário, Registrar Livro, Registrar Empréstimo de Livro, Registrar Devolução de Livro, Emitir Relatório de Livros Disponíveis, Emitir Relatório de Livros Emprestados, Emitir Relatório de Livros Devolvidos, Emitir Relatório de Multas e Emitir Relatório de Empréstimos Atrasados.

Os casos de uso **Supervisionar Empréstimos**, **Supervisionar Usuário**, **Registrar Bibliotecário** e o ator **Supervisor** são opcionais (**op**) porque eles podem ou não ser implementados em cada sistema instanciado a partir do padrão.

Cada relatório do padrão é rotulado com (**op**), o que significa que cada sistema computacional instanciado a partir do padrão considerado poderá ou não ter os relatórios implementados, ou seja, cada instância terá a sua configuração de relatórios.

O ator **Impressora** é variante (**var**) e múltiplo (fundo cinza), possibilitando que uma ou mais impressoras de tipos diferentes possam ser configuradas para cada instância do padrão, desde que os *drives* apropriados estejam disponíveis. A impressora não é opcional (**op**) visto que todas as instâncias de *software* construídas a partir do padrão necessitam imprimir o código de barra do livro (**Emitir Código de Barras de Livro)**.

O ator **Base de Dados** pode ser de vários tipos e também pode ser fornecido por diferentes fornecedores. Por esta razão ele é rotulado como **var**, de variante.

Pagamento de Multa é um subcaso de uso variante (**var**) e opcional (**op**) ao mesmo tempo. Isto significa que cada sistema instanciado a partir deste padrão pode ou não implementar esta funcionalidade e, em caso afirmativo, a funcionalidade pode ser implementada de diferentes formas, de acordo com a necessidade de cada instância.

Os casos de uso **Registrar Usuário**, **Registrar Livro** (incluindo o caso de uso **Emitir Código de Barras de Livro**), **Registrar Empréstimo de Livro** e **Registrar Devolução de Livro** são casos de uso constantes em todos os sistemas instanciados a partir do padrão.

3.2 Tipologia para as Classes Genéricas

Durante a criação do padrão arquitetural existe a etapa na qual se faz a criação do modelo estrutural (de classes). Este modelo é freqüentemente descrito através dos diagramas de classes que apresentam as classes, interfaces e relacionamentos que juntos constituem a estrutura do padrão [BOOCH et al 2000].

3.2.1 Tipos Aplicáveis às Classes

Esta seção apresenta uma tipologia para as classes genéricas do padrão arquitetural baseada nos tipos (opcional e variante) definidos na seção 3.1. Os estereótipos <<OP>> e

<<VAR>> são usados (separados ou de forma combinada) nas classes para denotar os significados de opcionalidade[4] e variabilidade das classes genéricas.

Uma classe com estereótipo <<OP>> significa uma classe que pode ou não fazer parte do modelo de classes do padrão, visto que essa classe é opcional no padrão.

Uma classe com estereótipo <<VAR>> significa uma classe que deve fazer parte do modelo de classes do padrão, embora possa haver variação entre as instâncias desta classe. Essas variações podem ser atributos e / ou métodos que sejam variantes ou mesmo opcionais para essa classe.

Estas características de opcionalidade e variabilidade conferem à classe alguns diferenciais no tratamento dos seus relacionamentos que serão abordados numa seção específica deste capítulo.

Uma classe variante significa que existe pelo menos um de seus atributos ou métodos que é variante ou opcional.

A Figura 3.14 apresenta três classes com estereótipos. Em a) tem-se uma classe opcional; em b) tem-se uma classe variante; e em c) a combinação dos dois estereótipos que podem ser aplicados às classes.

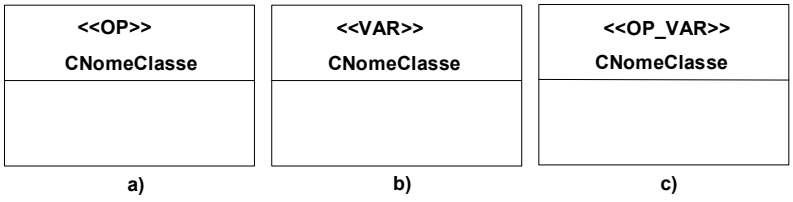


Figura 3.14 – Tipologia para as classes

3.2.2 Relacionamentos entre as Casses Genéricas

A tipologia proposta para classes genéricas implica algumas limitações no relacionamento entre as classes, como na tipologia para atores e casos de uso que foi apresentada na seção 3.1. Os relacionamentos entre as classes, previstos na UML [OMG 1998], são Associações, Agregações e Generalizações / Especializações.

(i) Associações

A associação é um tipo de relacionamento importante em sistemas orientados a objetos que permite que objetos de uma classe se comuniquem com objetos de outra classe.

[4] O termo opcionalidade não foi encontrado em dicionários da língua portuguesa. É usado neste trabalho como substantivo feminino que denota a qualidade do que é opcional.

A Figura 3.15 mostra uma associação entre a ClasseX e a ClasseY. Como a classe Y é opcional, faz-se necessário que o relacionamento também seja opcional e vinculado à existência da ClasseY, para evitar o caso dessa classe não ser utilizada quando o padrão for instanciado.

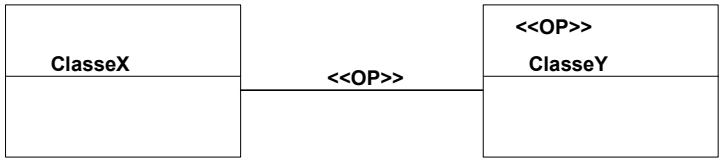


Figura 3.15 – Relacionamento com uma classe opcional

A Figura 3.16 mostra um relacionamento entre uma ClasseX e uma ClasseY variante. Neste caso, mesmo sendo a ClasseY variante o relacionamento poderá existir.

Figura 3.16 – Relacionamento com uma classe variante

(ii) Agregações

A agregação é um relacionamento de pertinência definido entre classes que permite estabelecer a inclusão de objetos de uma classe no interior de objetos de outra classe [STADZISZ 2002].

A Figura 3.17 apresenta dois relacionamentos de agregação, onde em a) a ClasseX agrega um objeto da ClasseY e em b) a ClasseZ agrega um objeto da ClasseW. Como nos dois casos pelo menos uma das classes é opcional, isso faz com que os relacionamentos sejam opcionais, tanto em a) como em b). Em b) pode-se ainda afirmar que a ClasseZ será variante porque agrega uma classe opcional (ClasseW).

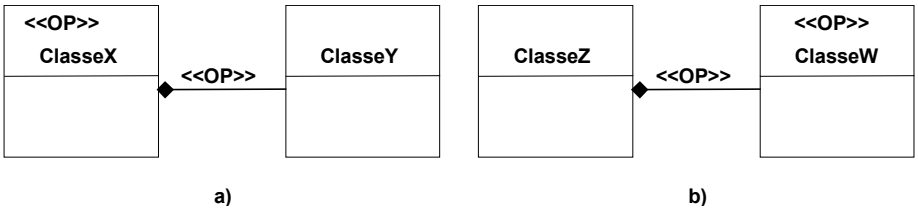


Figura 3.17 – Relacionamentos de agregação de classes opcionais

A Figura 3.18 mostra dois relacionamentos de agregação onde, em a) a ClasseX agrega objetos da ClasseY. Neste caso, nada se pode afirmar sobre o estereótipo da ClasseY. Em b) a ClasseZ agrega objetos da ClasseW. Neste caso, como a classe variante é a agregada (ClasseW), a classe agregadora (ClasseZ) também será variante.

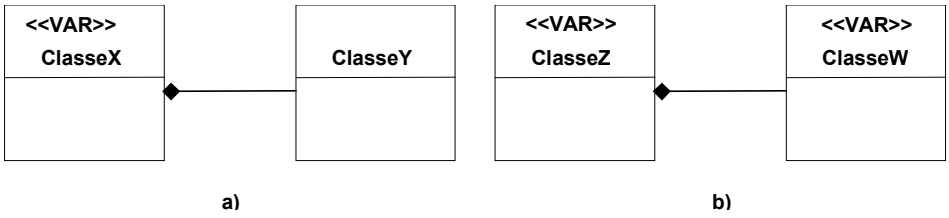


Figura 3.18 – Relacionamentos de agregação de classes variantes

(iii) Generalizações / Especializações

A generalização / especialização é um tipo de relacionamento entre duas classes que ocorre em termos estruturais, onde uma das classes será considerada base e a outra será considerada derivada [STADZISZ 2002].

A Figura 3.19 mostra dois relacionamentos de generalização / especialização onde a ClasseY herda os atributos e métodos da ClasseX, e a ClasseW herda os atributos e métodos da ClasseZ. Em a) como a ClasseX é opcional, também o será o relacionamento, entretanto, a ClasseY será variante por herdar atributos e métodos opcionais da ClasseX. Essa situação é pouco provável porque dificilmente uma classe derivada poderia existir sozinha, sem a classe base. Em b) como a ClasseW é opcional, o relacionamento com a ClasseZ também o será, entretanto, nada se pode afirmar sobre a ClasseZ.

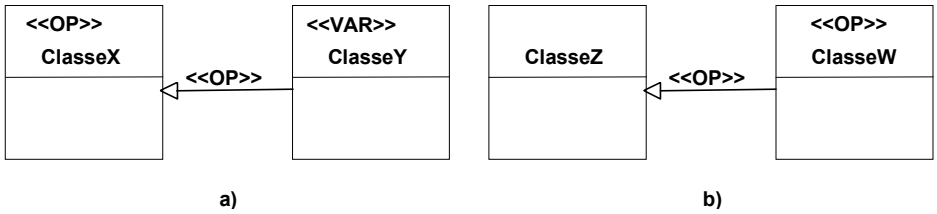


Figura 3.19 – Relacionamento de herança de classes opcionais

A Figura 3.20 mostra relacionamentos de generalização / especialização onde em a) como a ClasseX é variante, também será variante a ClasseY. Em b) embora a ClasseW seja variante, nada se pode afirmar sobre a ClasseZ. Os relacionamentos, propriamente ditos, não variam.

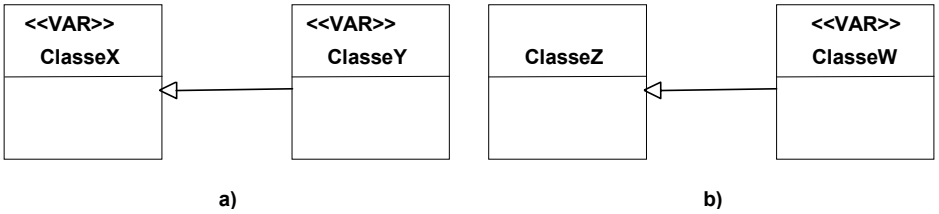


Figura 3.20 – Relacionamento de herança de classes variantes

3.3 Tipologia para os Diagramas de Seqüência Genéricos

Os diagramas de seqüência, como definido pela UML, mostram as comunicações que acontecem entre os objetos ao longo do tempo. Estas comunicações expressam as interações entre os objetos que possibilitam que os casos de uso sejam realizados. Estes diagramas são criados por caso de uso (um para cada cenário).

Neste trabalho procura-se evidenciar as partes não comuns dos diagramas de seqüência utilizando um retângulo que identifica a parte não comum do diagrama. Os atores mostrados nos diagramas de seqüência poderão utilizar a notação apresentada na seção 3.1. Para a representação dos objetos nos diagramas de seqüência, poderá ser utilizada a notação dos estereótipos das classes, conforme apresentado na seção 3.2.

A Figura 3.21 mostra um diagrama de seqüência hipotético com um retângulo menor que contém em seu canto superior direito o estereótipo <<**op>>**. Este estereótipo indica que essa parte do diagrama, contida no retângulo, é opcional, ou seja, em cada instanciação do padrão esta parte poderá ou não ser implementada. Esta mesma figura apresenta outro retângulo maior que contém em seu canto superior direito o estereótipo <<**var**>>, indicando que a parte do diagrama (contida no retângulo maior) é variante, ou seja, em cada instanciação do padrão esta parte poderá ser implementada de forma diferente (a critério do projetista).

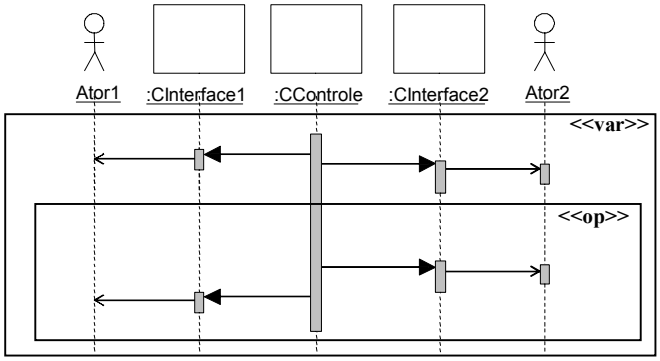


Figura 3.21 – Notação para os diagramas de seqüência

3.4 Tipologia para os Diagramas de Estados Genéricos

A UML utiliza os diagramas de estados para especificar a dinâmica dos sistemas. Estes diagramas mostram o comportamento dos objetos durante a execução do *software*. Para as classes não comuns, os diagramas de estados devem refletir qual parte do seu

comportamento não é comum. Para tanto é utilizado um diagrama de estados por classe genérica.

De forma semelhante aos diagramas de seqüência, os diagramas de estados, correspondentes às classes genéricas, têm uma notação que, inicialmente, utiliza um retângulo delimitador da parte genérica. Depois o conteúdo deste retângulo passa a ser considerado um superestado (estado composto). Para mostrar qual é o tipo do estereótipo da parte delimitada pelo superestado, usa-se no seu canto superior direito, o estereótipo correspondente da parte genérica, podendo assumir <<**op**>> ou <<**var**>> ou a combinação destes.

A Figura 3.22 mostra dois estágios (a e b) de um exemplo hipotético de um diagrama de estados para uma classe genérica. Neste diagrama, em a) os estados **E** e **F** são opcionais e foram colocados em um retângulo representando a parte opcional (estereótipo <<**op**>>). Em b) o retângulo foi transformado em um superestado **G** que contém os estados **E** e **F**. O estereótipo do superestado denota a opcionalidade do mesmo, onde cada instância do padrão poderá ou não implementar este estado dessa classe genérica. Caso o conteúdo do superestado **G,** que denota a parte opcional, não seja implementado, então **Mens4**, **Mens5** e **Mens6** deixam de existir, juntamente com os estados **E**, **F** e **G**.

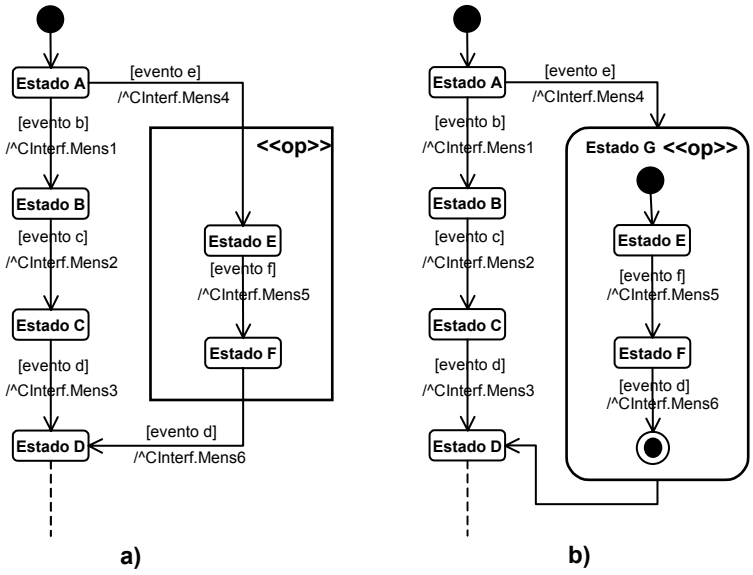


Figura 3.22 – Notação para os diagramas de estados com estereótipo <<op>>

A Figura 3.23 mostra um exemplo hipotético de um diagrama de estados para uma classe genérica. Neste diagrama, em a) os estados **B**, **C** e **D** estão contidos no retângulo com estereótipo <<**var**>>. Isto denota a variabilidade do conteúdo do retângulo, onde cada instância do padrão deverá implementar o conteúdo deste retângulo dessa classe genérica, porém poderão ocorrer variações nas formas de implementação desse conteúdo. Em b) o

retângulo foi transformado em um superestado **F** variante (estereótipo <<**var**>>). Dessa forma, as mensagens **Mens1** e **Mens3** agora chegam ao estado **F**. Este estado poderá sofrer variação na implementação interna, porém as suas interfaces são preservadas.

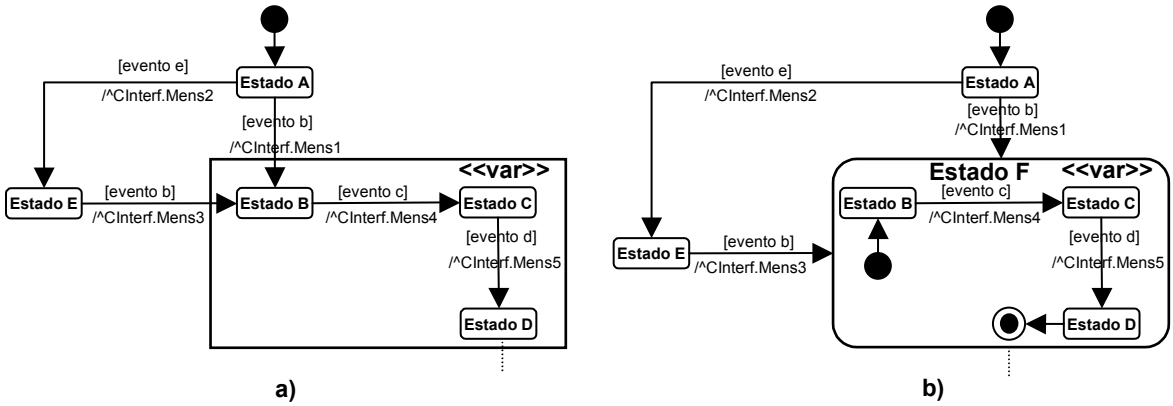


Figura 3.23 – Notação para os diagramas de estados com estereótipo <<var>>

3.5 Tipologia para os Diagramas de Componentes Genéricos

A UML define os Diagramas de Componentes como aqueles que mostram a organização e dependências entre os componentes. Ainda segundo a UML, um componente é uma parte modular e substituível de um sistema que encapsula seu conteúdo [OMG 1998]. O comportamento de um componente é externado através de interfaces que são fornecidas pelo componente, bem como de interfaces que são requeridas pelo componente.

A Figura 3.24 mostra a notação da UML 2.0 para componentes. O componente tem um estereótipo <<**component**>> e opcionalmente pode conter um ícone (retângulo com dois retângulos na borda esquerda) representativo de componentes nas versões anteriores da UML. Logo abaixo do estereótipo vem o nome do componente. O componente tem do lado externo a notação para as interfaces providas (reta com um círculo na ponta) e requeridas (reta com um semicírculo na ponta) pelo componente.

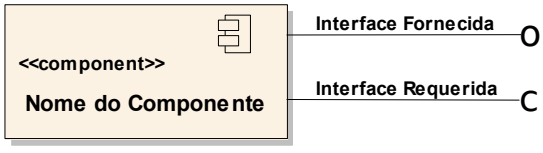


Figura 3.24 – Descrição de um componente

3.5.1 Tipos Aplicáveis aos Componentes

Esta seção apresenta uma tipologia para os componentes do padrão arquitetural baseada nos tipos (opcional e variante) definidos na seção 3.1. Os estereótipos <<OP>> e <<VAR>> são usados (separados ou de forma combinada) nos componentes para denotar os significados de opcionalidade e variabilidade dos componentes genéricos.

A Figura 3.25 apresenta a notação básica para os componentes genéricos do padrão arquitetural. O estereótipo da UML para componente <<**component**>> pode ser combinado com os estereótipos propostos neste trabalho (<<**op**>> e <<**var**>>). A combinação resulta em:

a) o estereótipo combinado <<**component_OP**>> que significa um componente genérico opcional, ou seja, um componente que poderá ou não ser implementado nas instâncias do padrão.

b) o estereótipo combinado <<**component_VAR**>> que significa um componente genérico variante, ou seja, um componente que deverá ser implementado nas instâncias do padrão, porém ocorrerão variações nas formas de implementação das instâncias.

c) o estereótipo combinado <<**component_OP_VAR**>> que significa um componente genérico opcional e variante, ou seja, um componente que poderá ou não ser implementado (estereótipo <<**op**>>) nas instâncias do padrão e, se for implementado, poderão ocorrer variações (estereótipo <<**var**>>) nas formas de implementação das instâncias.

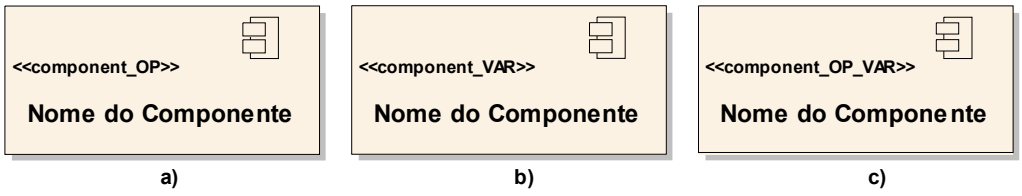


Figura 3.25 – Notação para componentes genéricos

3.5.2 Relacionamentos entre os Componentes Genéricos

A tipologia proposta para componentes genéricos impõe algumas limitações no relacionamento entre os componentes, à semelhança da tipologia para atores e casos de uso que foi apresentada na seção 3.1.

A Figura 3.26 apresenta dois componentes. O Componente2 é opcional, provê uma Interface Y e requer a Interface Z. O Componente1 será variante, pois se o Componente2 não existir então o Componente1 não poderá requerer a Interface Y.

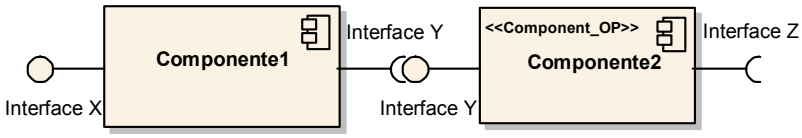


Figura 3.26 – Notação para relacionamento com componentes genéricos opcionais

A Figura 3.27 apresenta três componentes. O Componente2 é variante e disponibiliza uma Interface Y e requer a Interface Z. Neste caso, as Interfaces Y e Z deverão ser respeitadas e não devem ser variantes, para continuarem a ser compatíveis respectivamente com Componente1 e Componente3.

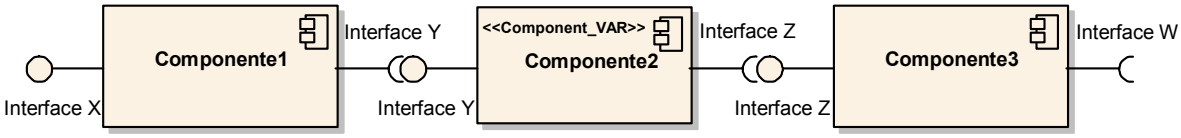


Figura 3.27 – Notação para relacionamento com componentes genéricos variantes

3.6 Considerações Finais

Neste capítulo foram apresentadas tipologias para: atores, casos de uso e classes. Foram apresentadas, ainda, notações para diagramas de: seqüência, estados e componentes. Estas tipologias / notações são usadas para evidenciar características de opcionalidade, variabilidade e multiplicidade dos elementos que compõem os diagramas da UML.

A UML, em sua atual versão 2.0, contempla multiplicidade e opcionalidade através do conceito de cardinalidade aplicado aos relacionamentos. A multiplicidade é tratada através do símbolo "*" e a opcionalidade, através do símbolo "0..1". Entretanto a UML 2.0 não contempla o conceito de variabilidade que é proposto nesta tese juntamente com os conceitos de opcionalidade e multiplicidade.

Decidimos ficar com a nossa notação para expressar os conceitos de variabilidade, opcionalidade e multiplicidade por dois motivos. Primeiro porque estes conceitos que estão nesta tese antecedem a UML 2.0. Fato este que pode ser comprovado por artigos que publicamos antes da liberação da UML 2.0 e que abordam estes conceitos. E segundo porque acreditamos que os conceitos (opcionalidade e opcionalidade) aplicados diretamente aos casos de uso e aos atores refletem melhor a idéia que o conceito quer expressar, em comparação ao uso de cardinalidade que é aplicado no relacionamento.

Todas as tipologias / notações que são propostas neste capítulo formam a base empregada no metamodelo de descrição de padrões arquiteturais, conforme abordado no próximo capítulo.

4 Metamodelo para Descrição de Padrão Arquitetural

It is only when no solution seems possible that we arrive at new solutions.

[Edgar Morin]

Este capítulo descreve o metamodelo proposto para a descrição de padrões arquiteturais. São apresentados, na primeira seção, os objetivos do metamodelo proposto. Na segunda seção apresenta-se o metamodelo propriamente dito, que é preenchido utilizando-se as tipologias apresentadas no capítulo 3. A última seção apresenta os passos que devem ser seguidos para a identificação e descrição de um padrão arquitetural utilizando o metamodelo.

4.1 Objetivos do Metamodelo

O metamodelo proposto é um modelo abstrato de representação utilizável para descrição de padrões arquiteturais de *software*, que por sua vez são modelos de representação de uma solução arquitetural para problemas computacionais recorrentes. O metamodelo proposto foi desenvolvido de forma a conter elementos adequados para expressar com clareza os padrões arquiteturais.

Um padrão descrito a partir do metamodelo serve como base para derivação / instanciação de modelos de sistemas baseados na arquitetura delineada através do padrão, resolvendo, dessa forma, problemas recorrentes em certo domínio de aplicação computacional.

Um metamodelo é uma forma de abstração composta por um conjunto de peças prontas e regras usadas para a construção de modelos. Ele fornece um modelo de referência abstrato para a construção de representações. Dessa forma, um metamodelo se presta como um meio de representação adequado que possibilita a abstração necessária para a descrição de padrões dentro de um domínio de interesse.

O emprego de um metamodelo para a construção de padrões arquiteturais permite facilitar o processo de criação de padrões e estabelecer uma organização ou notação uniforme para padrões arquiteturais.

A notação da linguagem UML serviu de base para a criação do metamodelo proposto, embora outra abordagem (*e.g.*: formalismo matemático, conjunto de regras, etc.) pudesse ter sido utilizada.

4.2 Metamodelo Proposto

Esta seção apresenta o metamodelo proposto para a descrição de um padrão arquitetural. O primeiro nível de abstração do metamodelo proposto define uma visão geral dos relacionamentos externos que um padrão arquitetural pode ter. A Figura 4.1 apresenta um diagrama de classes que descreve um Padrão Arquitetural como uma classe, mostra os eventuais relacionamentos desse padrão bem como sua descrição.

A figura apresenta alguns relacionamentos. O relacionamento **instancia** denota a possibilidade de instanciação do padrão para a criação de projetos de sistemas computacionais podendo haver inúmeras instanciações. O relacionamento **adota** mostra que um dado padrão adota um estilo arquitetural, ou seja, possui algumas características que o define como pertencendo a um estilo arquitetural, conforme visto na seção 2.3. A figura apresenta, ainda, dois relacionamentos **usa** que denotam a possibilidade de um padrão arquitetural empregar padrões de projeto e componentes reusáveis.

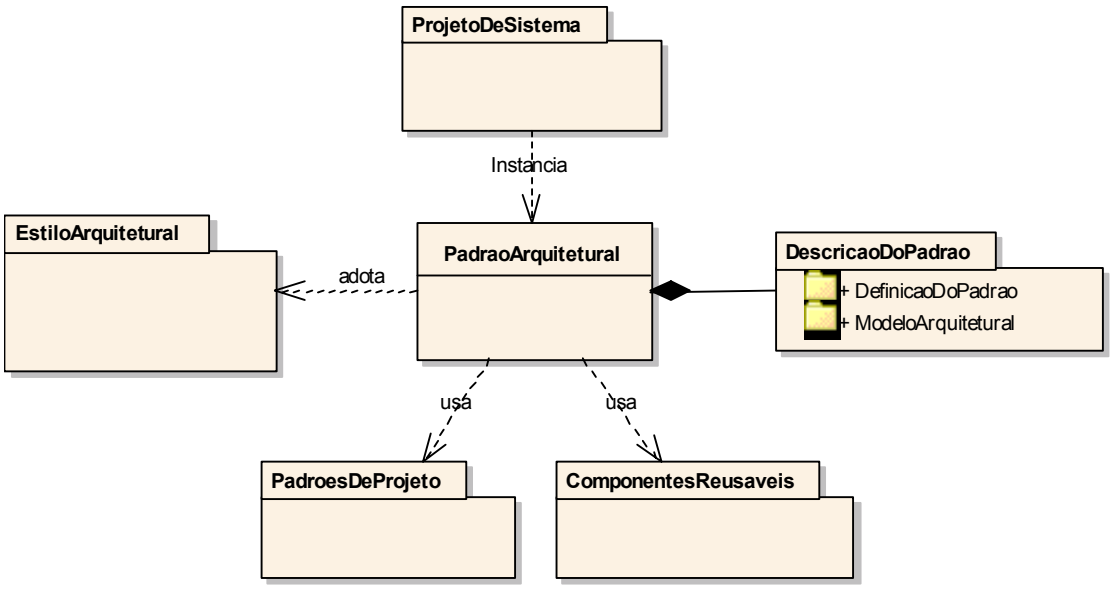


Figura 4.1 – Relacionamentos externos de um padrão arquitetural

4.2.1 Modelo de Descrição do Padrão

A Figura 4.2 apresenta o modelo geral de descrição de um padrão, composto por dois pacotes: **DefinicaoDoPadrao** e **ModeloArquiteturalDoPadrao**. O pacote **DefinicaoDoPadrao** contém informações gerais de identificação e definição do Padrão Arquitetural. Esta descrição abrange os elementos de definição estabelecidos pela GoF (seção 2.2). O pacote **ModeloArquiteturalDoPadrao** contém, de fato, os modelos de descrição estrutural, dinâmica, de requisitos e de implementação do Padrão Arquitetural. Os conteúdos destes pacotes são detalhados nas Figuras 4.3 e 4.4 e descritos a seguir.

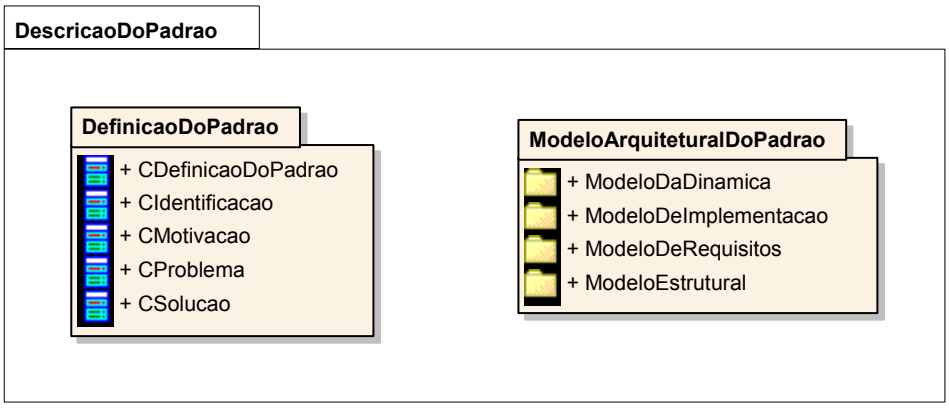


Figura 4.2 – Elementos de descrição de um padrão arquitetural

4.2.2 Modelo de Definição do Padrão

A Figura 4.3 apresenta o modelo de definição do padrão arquitetural na forma de um agregado de elementos que identificam, contextualizam e justificam o padrão. O diagrama apresenta a classe **CIdentificacao** que contém elementos que identificam o padrão. O

atributo **Identificador** registra o nome do padrão (*e.g.*: Padrão Arquitetural para Sistemas de Folha de Pagamento). O atributo **Autor** descreve o nome do criador do padrão (*e.g.*: EmpresaX/Equipe21). O atributo **Versão** registra o código da atualização do padrão (*e.g.*: Versão1.10-Data 13/02/2004)

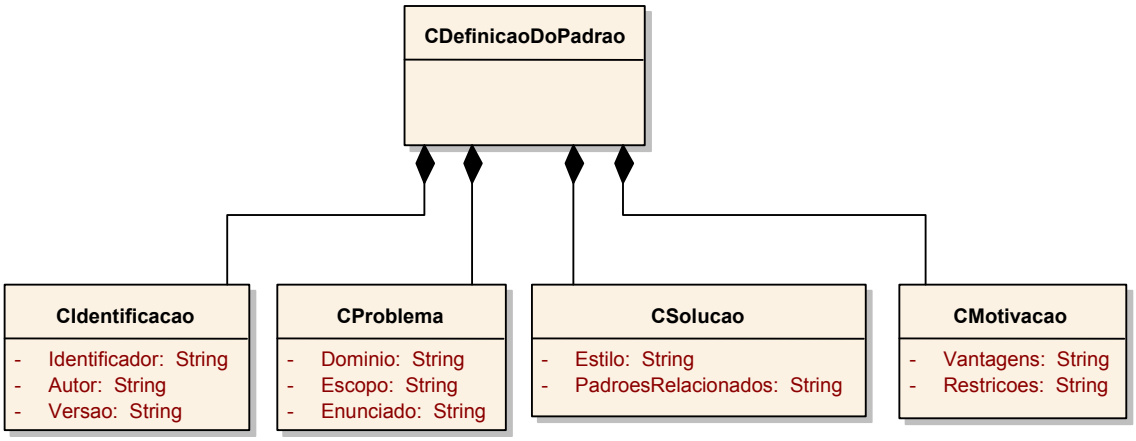


Figura 4.3 – Elementos de definição de um padrão arquitetural

A classe **CProblema** descreve o problema ao qual se aplica o padrão arquitetural. Ela é composta pelos atributos **Dominio** que registra o domínio de problema do padrão (*e.g.*: Domínio de Recursos Humanos, ou Gestão de Estoque), o atributo **Escopo** que delimita o padrão dentro do domínio (*e.g.*: Escopo de Folha de Pagamento) e, por fim, o atributo **Enunciado** que declara explicitamente o problema objeto do padrão (*e.g.*: Este padrão arquitetural tem o objetivo de representar uma classe de sistemas de gestão de folha de pagamento, servindo como base para o projeto e implementação de novos sistemas desta classe).

A classe **CSolucao** descreve o estilo da solução delineada pelo Padrão, bem como os padrões que se relacionam com esse problema / solução. Esta classe é composta pelos atributos **Estilo** que declara o estilo arquitetural adotado no padrão (*e.g.*: Cliente/Servidor) e o atributo **PadroesRelacionados** que declara os padrões que têm relacionamento direto com o padrão descrito (*e.g.*: Padrão Arquitetural para Cálculo de Imposto de Renda).

A classe **CMotivacao** é composta pelo atributo **Vantagens** que declara textualmente os motivos que justificam a utilização do padrão dentro do seu domínio e pelo atributo **Restricoes** que declara, também textualmente, as eventuais restrições da utilização do padrão.

4.2.3 Modelo Arquitetural do Padrão

O modelo arquitetural do padrão apresenta a descrição dos requisitos e de sua composição. Trata-se de um modelo genérico que se aplica a um domínio de aplicação, ou

seja, o modelo pode se conformar a diferentes instâncias de sistemas dentro de um mesmo domínio.

A Figura 4.4 apresenta o modelo arquitetural proposto. O modelo é composto pelo modelo de requisitos, pelo modelo estrutural, pelo modelo da dinâmica e pelo modelo de implementação, descritos a seguir.

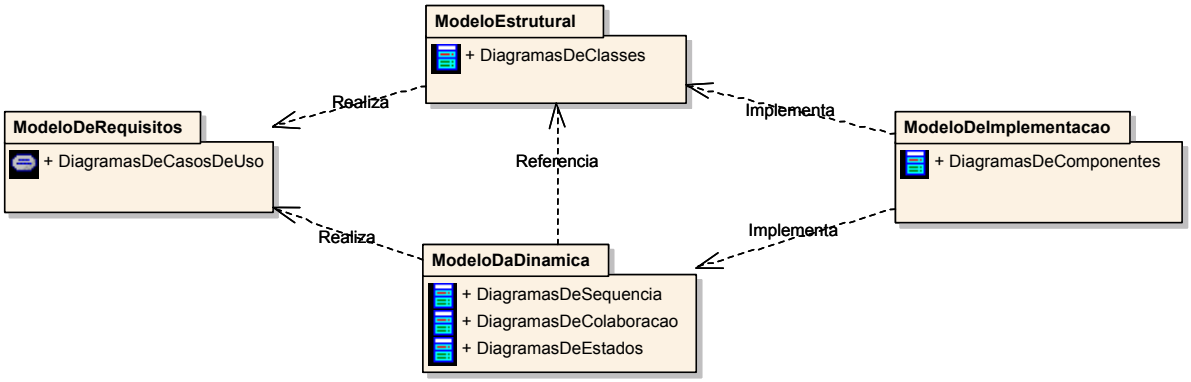


Figura 4.4 – Modelo da arquitetura de um padrão

(i) Modelo de Requisitos

O modelo de requisitos é uma peça fundamental do Padrão Arquitetural. Através deste modelo é possível a descrição dos requisitos do padrão, incluindo elementos (atores e casos de uso) variantes, opcionais e múltiplos, que caracterizam o padrão arquitetural. Desta forma, o modelo de requisitos será um modelo genérico que pode ser instanciado de diferentes maneiras nos sistemas baseados no padrão arquitetural.

Os requisitos do padrão arquitetural são extraídos de forma semelhante ao que se faz nos processos clássicos de análise usando a UML. Através da análise de diversos sistemas computacionais que fazem parte do domínio do problema, os requisitos genéricos são elicitados de forma que o modelo de requisitos possa evidenciar os casos de uso e atores genéricos do padrão arquitetural que está sendo construído. O processo de construção deste modelo de requisitos, utilizando a extensão do modelo de casos de uso da UML proposta na seção 3.1, será apresentado na seção 4.3 deste capítulo.

(ii) Modelo Estrutural

O modelo estrutural é outra peça fundamental do Padrão Arquitetural. Este modelo tem por objetivo descrever a estrutura de classes do Padrão. Os diagramas de classes da UML são usados neste modelo, com adição dos tipos para as classes genéricas, conforme

apresentado na seção 3.2. O processo de construção dos diagramas de classes genéricas será apresentado na seção 4.3 deste capítulo.

(iii) Modelo da Dinâmica

O Modelo da Dinâmica de um Padrão Arquitetural tem por objetivo apresentar as interações entre os objetos bem como o comportamento de cada objeto. Através deste modelo pode-se estudar a seqüência das operações executadas nos casos de uso, bem como estudar como os objetos de cada classe se comportam durante a execução das operações. Neste trabalho, o modelo da dinâmica é representado pelos diagramas de seqüência e diagramas de estados da UML. Os processos de construção dos diagramas de seqüência genéricos, utilizando a tipologia apresentada na seção 3.3, e de construção dos diagramas de estados genéricos utilizando a tipologia apresentada na seção 3.4, serão detalhados na seção 4.3 deste capítulo.

(iv) Modelo de Implementação

Segundo o RUP [KRUCHTEN 2000], o Modelo de Implementação estabelece as partes utilizadas na integração do sistema físico. O Modelo de Implementação tem o objetivo de fornecer uma visão com perspectiva de mais baixo nível do sistema, ou seja, é uma visão mais concreta e menos abstrata. Neste trabalho, o Modelo de Implementação é composto pelo Diagrama de Componentes. O processo de construção dos diagramas de componentes genéricos, utilizando a tipologia apresentada na seção 3.5, será detalhado na próxima seção deste capítulo.

4.3 Processo para a identificação e descrição de Padrões Arquiteturais

Esta seção tem a finalidade de apresentar o processo para identificação e descrição de padrões arquiteturais. A descrição do processo é feita através de diagramas SADT [ROSS e SCHOMAN 1977], cuja notação é resumida na Figura 4.5.

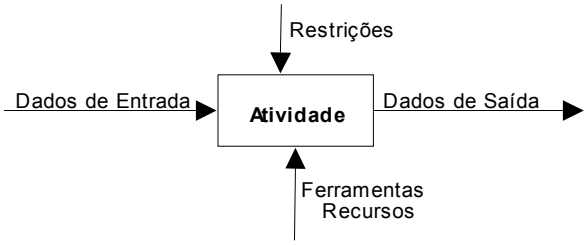


Figura 4.5 – Notação básica dos diagramas SADT

O processo proposto é constituído de três fases: Análise de Domínio, Análise de Requisitos e Análise das Arquiteturas. Estas fases são organizadas na forma de três macroatividades, conforme apresentado na Figura 4.6.

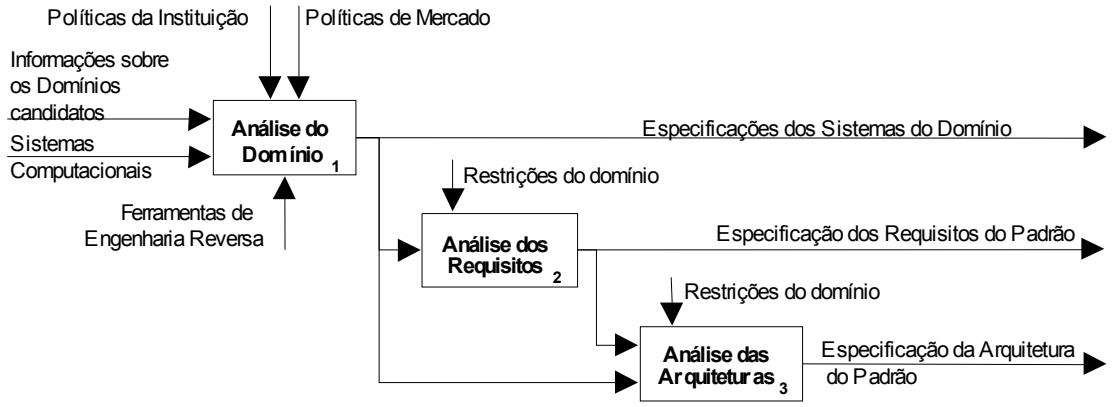


Figura 4.6 – Visão Geral do Processo Proposto

4.3.1 Análise de Domínio

Esta fase constitui o ponto inicial do processo. Nela é feita a escolha do domínio mediante identificação de prioridades e análise de riscos. São também selecionados os sistemas computacionais pertencentes ao domínio escolhido. Após a seleção dos sistemas, deve-se efetuar uma análise de custo / benefício para se verificar as possíveis vantagens da utilização dos sistemas escolhidos, como fonte para a criação do padrão. E, por fim, extraem-se as especificações dos sistemas selecionados. Ao final desta fase, obtêm-se as especificações dos sistemas objetos de estudo pertencentes ao domínio de aplicação para o qual se pretende criar o padrão arquitetural. A Figura 4.7 mostra a decomposição da fase Análise de Domínio.

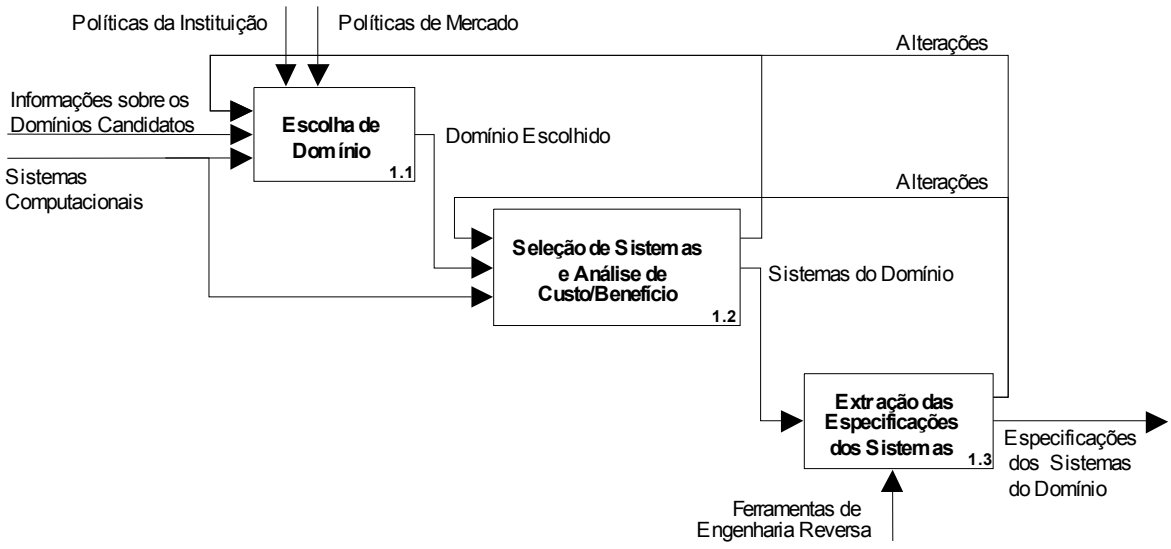


Figura 4.7 – Decomposição da fase Análise de Domínio

(i) Escolha de Domínio

Esta fase visa a escolha do domínio de aplicação para se criar o padrão arquitetural. A seleção é feita mediante a identificação das prioridades da empresa / instituição no tocante aos domínios de seu interesse. Esta escolha é totalmente dependente dos objetivos e metas da instituição que está empreendendo a definição e descrição do padrão arquitetural. Após a escolha, deve-se proceder a uma análise de riscos fazendo-se um balanço para justificar a definição de um padrão arquitetural para o domínio pretendido. Outro fator que tem influência na escolha do domínio são as políticas de mercado (externas à empresa / instituição) que regem o contexto no qual o domínio escolhido está inserido. A análise de riscos possibilita a validação ou não do prosseguimento da descrição do padrão arquitetural para o domínio escolhido.

(ii) Seleção de Sistemas e Análise de Custo / Benefício

O objetivo desta fase é obter os sistemas computacionais, com a respectiva documentação, já desenvolvidos e disponíveis, que implementem uma solução para a qual se deseja criar o padrão. Sugere-se que pelo menos três sistemas significativamente representativos do mesmo domínio sejam analisados para a composição do padrão arquitetural [ANTIPATTERNS 1998]. Busca-se ainda obter outras informações (*e.g.:* informações sobre o domínio) disponíveis que possam auxiliar o desenvolvimento do padrão arquitetural desejado. Dos sistemas selecionados, faz-se uma análise de custo / benefício para verificar as vantagens da utilização de cada sistema escolhido, como fonte para a criação do padrão.

(iii) Extração das Especificações dos Sistemas

Nesta fase busca-se extrair as especificações de cada um dos sistemas computacionais que foram escolhidos na fase anterior. Para esta extração pode-se utilizar ferramentas próprias para engenharia reversa, as quais podem, a partir do código fonte, recuperar especificações de sistemas computacionais em nível de projeto. Estas especificações são basicamente os diagramas da linguagem UML. Caso os sistemas computacionais selecionados na fase anterior já tenham a documentação com os diagramas UML, não haverá necessidade de extrair os diagramas, entretanto, deve-se certificar que a documentação fornecida dos sistemas escolhidos está atualizada.

4.3.2 Análise dos Requisitos

Nesta etapa é efetuada a análise dos requisitos dos sistemas computacionais pertencentes ao domínio escolhido na fase anterior. Através dessa análise se obtém os requisitos genéricos do padrão arquitetural. A Figura 4.8 mostra a decomposição da fase Análise de Requisitos, sendo que as entradas básicas (especificações dos requisitos dos sistemas) provêm da fase 1.3, conforme mostrado na figura 4.7.

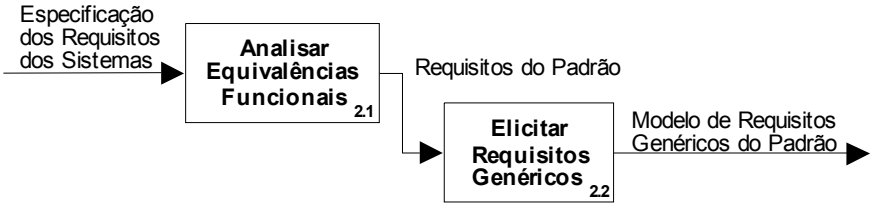


Figura 4.8 – Decomposição da fase Análise dos Requisitos

(i) Analisar Equivalências Funcionais

Esta atividade compreende a construção e preenchimento de uma tabela (Tabela 4.1) que conterá os casos de uso que representam as funcionalidades de cada sistema. Essa tabela deve ser exaustiva de modo que contenha todos os casos de uso de cada sistema do domínio de aplicação.

No processo de preenchimento da Tabela 4.1 deve-se unificar os casos de uso referentes a cada sistema. Para esta unificação usa-se a técnica da análise de equivalência [STADZISZ 1997] dos casos de uso. Essa técnica consiste basicamente em identificar relações de equivalência entre os casos de uso dos sistemas.

Os casos de uso que realizam a mesma função são funcionalmente equivalentes, ou seja, participam de uma mesma relação de equivalência. Por exemplo, dados dois casos de uso pertencentes a dois sistemas do domínio de aplicação do padrão. Se estes dois casos de uso forem funcionalmente equivalentes, então existe uma relação de equivalência entre os dois.

Uma relação de equivalência entre casos de uso dos sistemas de um domínio leva à criação de uma classe de equivalência de casos de uso. Esta classe de equivalência contém todos os casos de uso que participam da relação de equivalência. Uma Classe de Equivalência de Casos de Uso pode ser definida na forma CEUC = $\{uc_1, ..., uc_n\}$ / $uc_k \in$ CEUC se uc_k é um caso de uso que implemente a funcionalidade de CEUC, ou seja uc_k participa da relação de equivalência que estabelece CEUC.

Através do emprego desta técnica de análise de equivalência dos casos de uso estabelecem-se relações de equivalência, que por sua vez geram classes de equivalência. Estas classes de equivalência representam os casos de uso de todos os sistemas analisados que implementam o caso de uso. Esta tarefa de unificação é bastante subjetiva e demanda atenção e conhecimento do domínio para reunir corretamente os casos de uso nas respectivas classes de equivalências.

Tabela 4.1 – Subsistemas, Casos de uso e Atores

SUBSISTEMA	CASO DE USO	Sistemas que implementam				Atores			
						At1	At2	…	Atn
Cadastro	Cadastrar x	1	1	2	3	x	x		
	Cadastrar y			2	3				x
	...	...	...	...	...	...	...	...	...
Movimentação	Entrada no estoque	1	1	2	3	x			
	Saída no estoque	1	1	2	3		x		
	Estorno	1	1	2			x		
...	...	...	...	...	...	...	...	...	...

Analisando-se a Tabela 4.1 pode-se identificar os casos de uso que são comuns a todos os sistemas e, também, aqueles que são específicos a pelo menos um dos sistemas analisados. Cada caso de uso que for implementado por apenas um ou alguns dos sistemas selecionados são específicos e são denominados casos de uso opcionais.

Durante a descrição das tarefas busca-se o agrupamento dos casos de uso em possíveis subsistemas. Entende-se por subsistema uma parte de um sistema computacional que agrega um conjunto de casos de uso desse sistema. Busca-se ainda a identificação dos atores que participam dos casos de uso do sistema.

Os subsistemas que comporão o padrão arquitetural devem ser identificados nessa tabela. Cada subsistema deve conter os casos de uso que lhe forem concernentes. Para essa identificação dos subsistemas usa-se, novamente, a técnica da análise de equivalência [STADZISZ 1997] aplicada aos subsistemas. Para cada subsistema, através do estabelecimento de uma relação de equivalência, cria-se uma classe de equivalência CESUB = $\{sub_1, ..., sub_n\}$ / $sub_k \in$ CESUB se sub_k é um subsistema que implemente os casos de uso de CESUB.

Esse processo de análise, separação dos subsistemas e composição desta tabela única permite ainda a identificação dos atores do padrão arquitetural. Para essa identificação dos atores usa-se, novamente, a técnica da análise de equivalência aplicada aos atores. Para cada ator, cria-se uma classe de equivalência CEAT = $\{at_1, ..., at_n\}$ / $at_k \in$ CEAT se at_k é

um ator que desempenhe o mesmo papel e seja relacionado aos mesmos casos de uso dos atores de CEAT.

(ii) Elicitar Requisitos Genéricos

Esta fase visa identificar atores e casos de uso genéricos que são variantes, opcionais e múltiplos. Identificam-se, na Tabela 4.1, os casos de uso que são implementados por parte (não todos) dos sistemas. Estes casos de uso são fortes candidatos a serem casos de uso Opcionais. Os casos de uso que são implementados por todos os sistemas analisados são os não Opcionais e, conseqüentemente, fazem parte do núcleo do Padrão Arquitetural.

A elicitação dos casos de uso Variantes é feita procurando-se em cada caso de uso implementado pelos sistemas, as diferenças que possam caracterizar esse caso de uso como Variante. Para esta procura por diferenças entre os casos de uso dos sistemas, a classe de equivalência CEUC (criada, conforme seção 4.3.2 subseção i) é insuficiente uma vez que essa classe de equivalência foi criada por uma relação de equivalência funcional. Querendo significar que dois casos de uso podem ser funcionalmente equivalentes, porém cada um é implementado de uma maneira.

Uma técnica efetiva para a definição dos casos de uso variantes consiste em analisar as informações que são mostradas / editadas na execução dos casos de uso em cada sistema que o implemente. Essa técnica pode indicar as diferenças entre os casos de uso nos sistemas analisados. A Figura 4.9 mostra um exemplo de caso de uso variante usado para efetivar pagamento. Este caso de uso é variante, pois analisando-se a execução de vários sistemas que o implementa, verifica-se que há diferença na forma de concretização do pagamento. Em um sistema o pagamento pode ser feito apenas em dinheiro ou cheque, em outro sistema analisado o pagamento pode ser feito com dinheiro, cheque e cartão de débito, analisando-se ainda outro sistema verifica-se que o pagamento pode ser feito com as opções já citadas e acrescentando-se ainda cartão de crédito. Assim, este caso de uso é variante, pois representa um conjunto de casos de uso que são funcionalmente equivalentes, porém são implementados de formas diferentes.

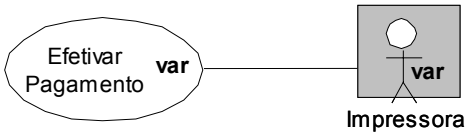


Figura 4.9 – Exemplo de caso de uso Variante

Os casos de uso Múltiplos são elicitados procurando-se evidenciá-los a partir da observação das funcionalidades de cada sistema que pertence ao domínio de aplicação. Os

casos de uso que permitem / requerem várias execuções independentes em várias *threads* ou em processadores diferentes são considerados casos de uso Múltiplos. A Figura 4.10 mostra um exemplo de caso de uso múltiplo usado para controlar um dispositivo atuador. Este caso de uso será multiplicado, dependendo da necessidade da instância do padrão. Em uma instância do padrão que utiliza este caso de uso existe um dispositivo, já em outra instância do padrão existem três dispositivos, cada qual controlado com exclusividade pelo seu controlador. Outra instância do padrão utiliza oito dispositivos atuadores, cada qual também é controlado com exclusividade pelo seu controlador.

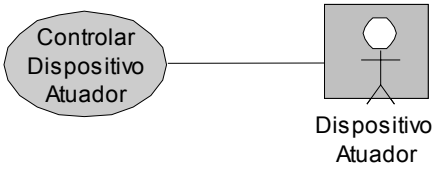


Figura 4.10 – Exemplo de caso de uso Múltiplo

Nesse processo de elicitação procura-se, ainda, identificar os atores genéricos concernentes aos vários casos de uso. Esses atores são identificados através da observação dos atores de cada sistema e o seu comportamento. O exemplo da Figura 4.9 mostra um ator (impressora) que é variante e múltiplo. A elicitação das características deste ator foi feita observando-se a variabilidade de modelos e a quantidade de impressoras utilizadas nos sistemas que implementam o caso de uso Efetivar Pagamento. Da mesma forma, a Figura 4.10 mostra um ator (dispositivo atuador) que é múltiplo. A elicitação da característica deste ator foi feita observando-se a quantidade de dispositivos atuadores que são encontrados nas instâncias do padrão que implementam o caso de uso Controlar Dispositivo Atuador.

4.3.3 Análise das Arquiteturas

Esta etapa tem por objetivo analisar a estrutura e comportamento dos sistemas que fazem parte do domínio de aplicação escolhido. Das estruturas dos sistemas, analisam-se as classes, relacionamentos e componentes, e dos comportamentos desses sistemas, analisam-se as interações e modelos de estados. A especificação dos requisitos do padrão, desenvolvida na etapa anterior, é também utilizada nesta etapa. A Figura 4.11 mostra a decomposição da etapa Análise das Arquiteturas.

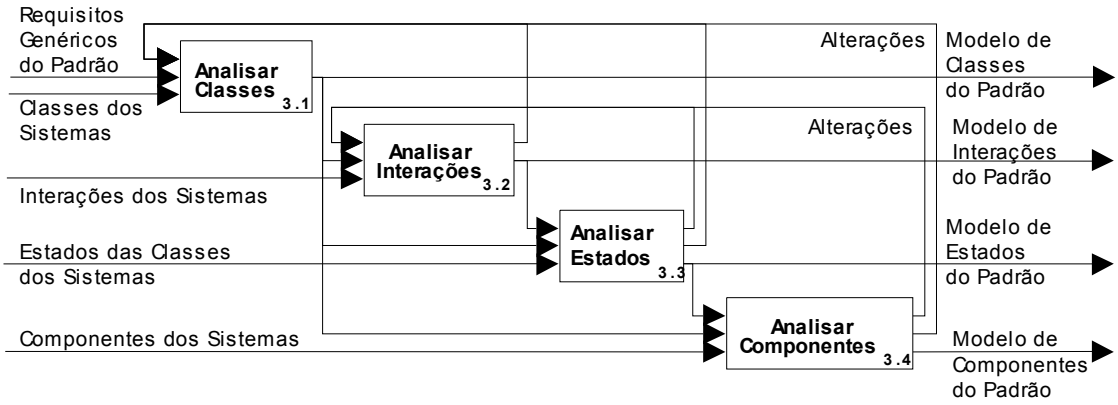


Figura 4.11 – Decomposição da etapa Análise das Arquiteturas

(i) Analisar Classes

Os modelos de classes dos sistemas que integram o domínio de aplicação devem ser usados para a identificação das classes que comporão o padrão arquitetural. Caso não existam esses modelos de classes dos sistemas, pode-se efetuar um processo de engenharia reversa para cada sistema para extrair os seus modelos de classes correspondentes.

Analisando-se os modelos de classes dos sistemas, o modelo de classes do padrão arquitetural é sintetizado. Para esta síntese usa-se a tipologia para as classes genéricas definida na seção 3.2. Esta síntese é baseada em uma análise de equivalência das classes que é composta, basicamente, pelos passos a seguir:

1) Para um ator genérico "X" do padrão, criar uma Classe de Equivalência de Classes de Interface CECI_X = $\{ci_1, ..., ci_n\}$ / $ci_k \in$ CECI_X se ci_k é uma classe de interface com um ator equivalente a X.

2) Criar uma classe genérica do padrão para a classe de equivalência CECI_X. Se o ator genérico X for variante, então a classe genérica correspondente será variante. Se o ator X for opcional, então a classe genérica correspondente será opcional.

Por exemplo: seja CECI_X = {CIBibliotecario, CIFuncionario, CIAtendente} uma classe de equivalência de classes de interface dos sistemas de um domínio de aplicação (Figura 4.12).

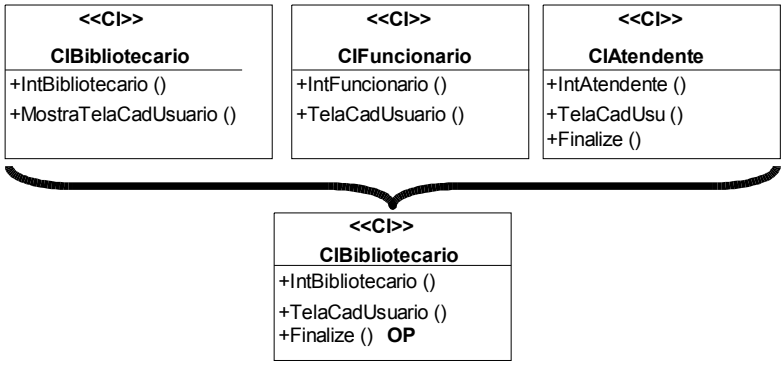


Figura 4.12 – Unificação de classes de interface dos sistemas do domínio para a classe de interface do padrão

Ainda do exemplo da Figura 4.12 deve-se definir os atributos e métodos da classe do padrão que são sintetizados a partir dos atributos e métodos das classes que pertencem à classe de equivalência CECI_X, da seguinte forma:

3) Cada atributo, de cada uma das classes, deverá ser analisado para verificar se o mesmo pertence a uma classe de equivalência de atributo CEAT = $\{cat_1, ..., cat_n\}$ / $cat_k \in$ CEAT se cat_k é um atributo com as mesmas características (tipo de dado) e finalidade dos atributos de CEAT.

4) Criar um atributo genérico para a classe de equivalência CEAT da classe do padrão. Se o atributo existir em apenas algumas classes, então ele será opcional. Se o tipo desse atributo variar de uma classe para outra, então esse atributo será variante.

5) Cada método, de cada uma das classes, deverá ser analisado para verificar se o mesmo pertence a uma classe de equivalência de método CEMT = $\{cmt_1, ..., cmt_n\}$ / $cmt_k \in$ CEMT se cmt_k é um método com as mesmas características (interface) e finalidade dos métodos de CEMT.

6) Criar um método genérico para a classe de equivalência CEMT da classe do padrão. Se o método existir em apenas algumas classes, então ele será opcional. Se o método variar de uma classe para outra, então esse método será variante.

Por exemplo: os métodos IntBibliotecario(), IntFuncionario() e IntAtendente() (Figura 4.12) possuem as mesmas características e, portanto, pertencem à mesma classe de equivalência. Dessa forma a classe do padrão passa a ter o método IntBibliotecario().

Os métodos MostraTelaCadUsuario(), TelaCadUsuario() e TelaCadUsu() possuem as mesmas características e, portanto, pertencem à mesma classe de equivalência. O método do padrão, correspondente a estes métodos dos sistemas é TelaCadUsuario().

O método Finalize() existe apenas em uma classe de um dos sistemas do domínio, então este método deve ser colocado também no padrão, porém este método é opcional, visto que apenas um dos sistemas do domínio o utiliza.

7) Para um caso de uso genérico "Y" do padrão, criar pelo menos uma Classe de Equivalência de Classes de Controle CECC_Y = $\{cc_1, ..., cc_n\}$ / $cc_k \in$ CECC_Y se cc_k é uma classe de controle para um caso de uso equivalente a Y.

8) Criar uma classe genérica do padrão para a classe de equivalência CECC_Y. Se o caso de uso genérico Y for variante, então a classe genérica correspondente será variante. Se o caso de uso Y for opcional, então a classe genérica será opcional.

Por exemplo: seja CECC_Y = {CCEmitirRecibo, CCImprimeRecibo, CCRecibo} uma classe de equivalência das classes de controle dos sistemas do domínio de aplicação (Figura 4.13).

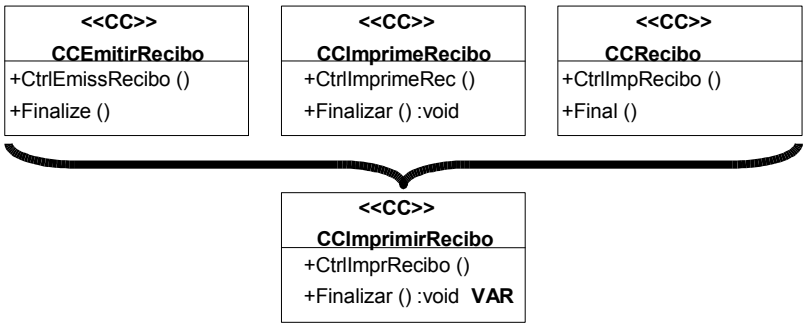


Figura 4.13 – Unificação de classes de controle dos sistemas do domínio para a classe de controle do padrão

Ainda do exemplo da Figura 4.13 deve-se definir os atributos e métodos do padrão que são sintetizados a partir dos atributos e métodos das classes que pertencem à classe de equivalência CECC_Y. Para tanto, os passos 3 e 4 devem ser repetidos para os atributos e os passos 5 e 6 devem ser repetidos para os métodos da classe de equivalência CECC_Y.

9) Para cada caso de uso do padrão e para cada entidade que congrega um grupo de dados que juntos possuem identidade (representam entidades do mundo real), criar uma Classe de Equivalência de Classes de Entidade CECE_Z = $\{ce_1, ..., ce_n\}$ / $ce_k \in$ CECE_Z se houver uma equivalência da entidade representada por ce_k com relação aos demais elementos de CECE_Z e entre, pelo menos, parte dos atributos e métodos de ce_k e atributos e métodos de CECE_Z.

10) Criar uma classe genérica do padrão para a classe de equivalência CECE_Z. Se a classe de entidade existir em parte dos sistemas do domínio de aplicação, então essa classe

será opcional. Se existir pelo menos um atributo ou método dessa classe que seja variante ou opcional, então a classe será variante.

Por exemplo: seja CECE_Z = {CEUsuario, CEAluno, CEDadosAluno} uma classe de equivalência de classes de entidade dos sistemas do domínio de aplicação (Figura 4.14).

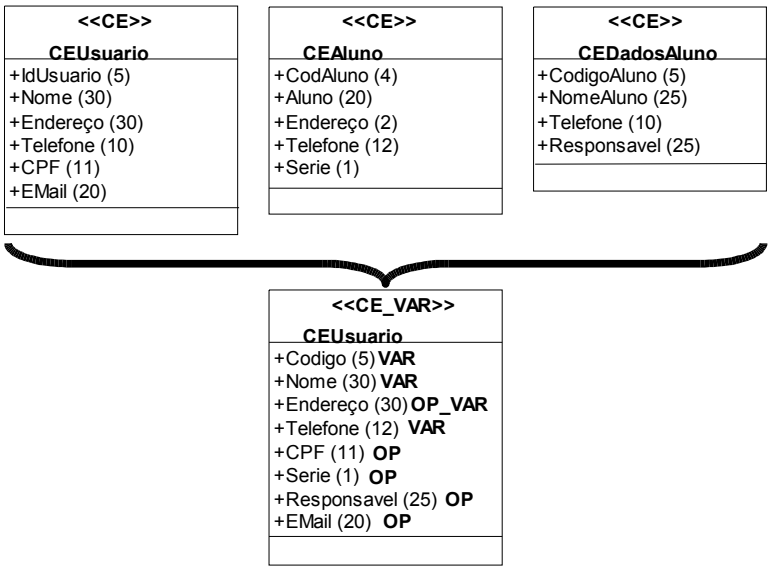


Figura 4.14 – Unificação de classes de entidade dos sistemas do domínio para a classe de entidade do padrão

Ainda do exemplo da Figura 4.14 deve-se definir os atributos e métodos da classe do padrão, que são sintetizados a partir dos atributos e métodos das classes que pertencem à classe de equivalência CECE_Z. Para tanto, os passos 3 e 4 (para atributos) e 5 e 6 (para métodos) devem ser repetidos para os atributos e métodos da classe de equivalência CECE_Z.

Os passos que foram descritos nesta seção são diretrizes básicas e é possível que o projetista do padrão insira novas classes, relacionamentos, atributos e métodos no padrão com vistas à otimização do projeto do mesmo, conforme previsto nos fluxos de Alterações, na Figura 4.11. Muitas classes podem ser subdivididas e outras podem ser fundidas para melhor se adequar aos requisitos que se espera do padrão.

(ii) Analisar Interações

Os diagramas de interações mostram de que maneira os objetos e grupos de objetos interagem para realizar os casos de uso. As interações são descritas, basicamente, pelos diagramas de Seqüência e de Colaborações. Neste trabalho, utilizam-se apenas os diagramas de seqüência.

Caso os diagramas de seqüência dos sistemas que fazem parte do domínio de aplicação estejam disponíveis, estes diagramas devem ser utilizados para a síntese dos

diagramas de seqüência dos casos de uso do padrão arquitetural, seguindo-se o PASSO 1 descrito a seguir. Caso os diagramas de seqüência dos sistemas que fazem parte do domínio de aplicação não estejam disponíveis, então o PASSO 1 deve ser pulado.

PASSO 1) Para cada classe de equivalência de casos de uso (CEUC) que foi construída conforme foi definido na seção 4.3.2, deve-se analisar os diagramas de seqüência dos sistemas referentes a esta classe de equivalência para se criar o diagrama de seqüência correspondente do caso de uso do padrão feito no PASSO 2. Esta análise é auxiliar na elaboração do diagrama de seqüência do padrão.

PASSO 2) Para cada caso de uso do padrão arquitetural, deve-se construir um diagrama de seqüência que descreve a seqüência normal de comunicações entre objetos para execução do caso de uso. Os diagramas de seqüência para os casos de uso do padrão que não são opcionais, nem variantes e nem múltiplos são desenvolvidos da maneira convencional, utilizando-se os recursos oferecidos pela UML [BOOCH et al 2000]. Já os diagramas de seqüência que são relativos a casos de uso opcionais, variantes e múltiplos devem explicitar nos diagramas de seqüência quais partes são diferenciadas e quais não são. A notação adotada para isso está descrita na seção 3.3.

PASSO 3) Após a criação de cada diagrama de seqüência do padrão, deve-se definir as partes do diagrama que são opcionais ou variantes. As regras para essa definição são:

Regra 1: Se existir algum ator opcional, então as interações desse ator serão opcionais. Por conseqüência, a interface do ator também será opcional.

Regra 2: Se existir algum objeto referente a alguma classe que seja opcional, então as interações desse objeto serão opcionais.

Regra 3: Se existir algum ator que seja variante, então as interações que sejam relativas a esse ator também serão variantes.

Regra 4: Se existir algum objeto referente a alguma classe que seja variante, então as mensagens relativas a esse objeto também poderão ser variantes.

Estes passos podem ser complementados com a execução dos casos de uso nos sistemas referentes ao caso de uso do padrão. Essa execução pode revelar detalhes da implementação deste caso de uso nos diferentes sistemas do domínio de aplicação. Estes detalhes permitem avaliar a generalidade do caso de uso e, conseqüentemente, do diagrama de seqüência correspondente.

A Figura 4.15 apresenta um exemplo de diagrama de seqüência para um caso de uso de pagamento de multa. A parte do diagrama, externa do retângulo, é igual para todos os

sistemas analisados que implementam este caso de uso de pagamento de multa. Já a parte interna do retângulo é opcional e variante, conforme indica os estereótipos no canto superior direito do retângulo no diagrama. Esta parte interna do retângulo é opcional porque nem todos os sistemas, do domínio de aplicação, analisados que implementam este caso de uso fornecem uma forma de pagamento de multa diferenciada. Eles fornecem apenas uma forma fixa de pagamento da multa. Esta mesma parte interna do retângulo é variante porque, analisando-se a forma como o caso de uso é implementado em cada um dos sistemas, verificou-se que há variação na forma de implementação desta parte interna do retângulo.

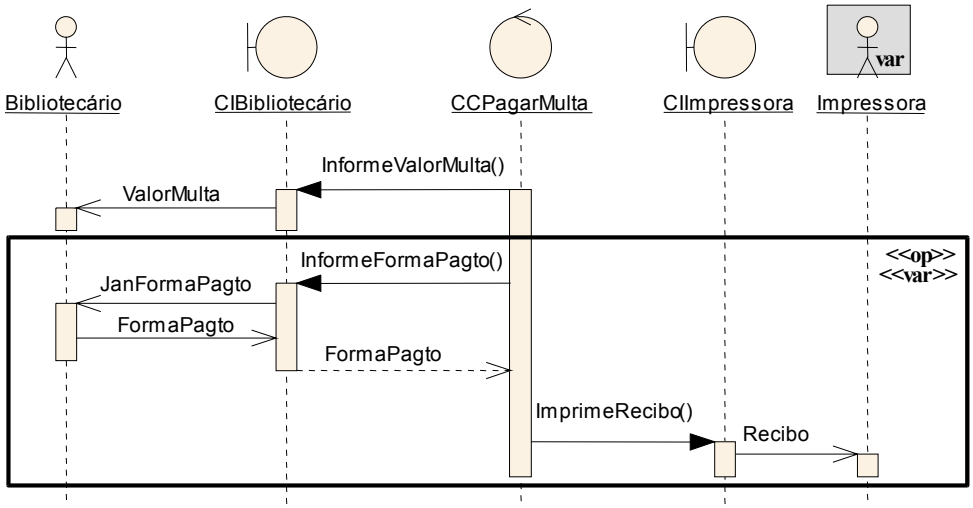


Figura 4.15 – Diagrama de seqüência evidenciando parte opcional e variante

Algumas ferramentas de modelagem para a UML possibilitam desenvolver automaticamente os diagramas de colaboração. Estes diagramas serão úteis para estabelecer as relações entre as classes.

(iii) Analisar Estados

De maneira análoga aos diagramas de seqüência, os diagramas de estados são criados usando-se a UML para a representação das classes comuns. Para se descrever os estados e mensagens de classes opcionais ou variantes usa-se um retângulo que evidencia essas partes, conforme foi descrito na seção 3.4.

A Figura 4.16 apresenta um diagrama de estados para a classe CCPagarMulta relativo a Figura 4.15. A parte do diagrama que fica externa ao superestado Pagando é igual para todos os sistemas analisados que implementam esta classe do caso de uso de pagamento de multa. Já a parte interna do superestado é opcional e variante, conforme indica os estereótipos no canto superior direito do retângulo no diagrama. O superestado é

opcional porque nem todos os sistemas do domínio que foram analisados e que implementam esta classe fornecem uma forma de pagamento de multa diferenciada, fornecem apenas uma forma fixa de pagamento da multa. O superestado é ainda variante porque, analisando-se a forma como a classe é implementada em cada um dos sistemas, verificou-se que há variação na forma de implementação desta área representada pelo superestado.

Outro aspecto do diagrama da Figura 4.16 é que no caso da parte opcional (superestado) não ser implementada, as mensagens InformeFormaPagto e ImprimeRecibo não existirão e, assim, após a mensagem InformarValorMulta o fluxo será redirecionado para o estado Pronto.

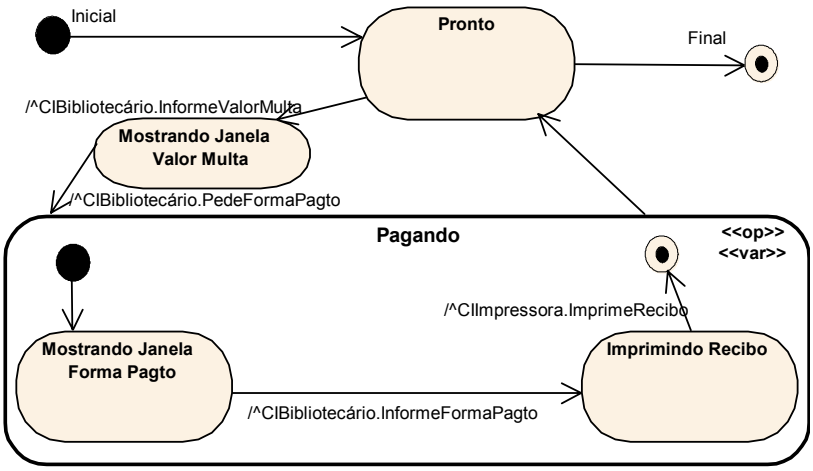


Figura 4.16 – Diagrama de Estados evidenciando parte opcional e variante

(iv) Analisar Componentes

Os componentes do padrão são definidos pelo projetista, de duas maneiras: através da análise dos componentes dos sistemas do domínio, caso existam, e através da análise das tabelas e diagramas que compõem a descrição do padrão.

Caso exista o modelo de componentes de cada sistema que compõe o domínio de aplicação, então se deve seguir os passos básicos, a seguir, para a análise desses modelos de componentes e o desenvolvimento do diagrama de componentes do padrão:

PASSO 1) Para cada componente dos sistemas do domínio de aplicação verificar se esse componente pertence a uma Classe de Equivalência de Componentes na forma CECOMP_W = $\{ccomp_1, ..., ccomp_n\}$ / $ccomp_k \in$ CECOMP_W se $ccomp_k$ é um componente que tiver as mesmas, ou parte, das características (mesma funcionalidade e mesmas interfaces) dos componentes de CECOMP_W.

PASSO 2) Criar um componente genérico do padrão para a classe de equivalência CECOMP_W. Se o componente não existir em todos os sistemas do domínio de aplicação, então este componente será opcional. Se houver variação no conteúdo e/ ou interface dos componentes que pertençam a CECOMP_W, então esse componente será variante.

Por exemplo (Figura 4.17): seja CECOMP_W = {Movimentação, Empréstimo} uma classe de equivalência dos componentes dos sistemas do domínio de aplicação. Analisar as interfaces de cada componente de CECOMP_W.

Criar o componente do padrão referente a esta classe de equivalência de tal forma que o componente genérico do padrão tenha as mesmas interfaces dos componentes de CECOMP_W.

PASSO 3) Cada interface de cada componente de CECOMP_W deverá ser analisada para verificar se a mesma pertence a uma Classe de Equivalência de Interface CEINT = $\{int_1, ..., int_n\}$ / $int_k \in$ CEINT se int_k é uma interface com as mesmas, ou parte, das características (mesmas entradas ou saídas) das interfaces de CEINT.

PASSO 4) Criar uma interface genérica para a classe de equivalência CEINT do componente do padrão. Se a interface não existir em todos os componentes de CECOMP_W, então essa interface será opcional. Se houver variação nas interfaces de CEINT, então essa interface será variante.

Por exemplo: as interfaces CEmprestar:Classe e CEmprestimo:Classe (Figura 4.17) possuem as mesmas características (participam de uma mesma relação de equivalência) e, portanto, pertencem à mesma classe de equivalência. Dessa forma, o componente do padrão passa a ter a interface CEmprestar:Classe.

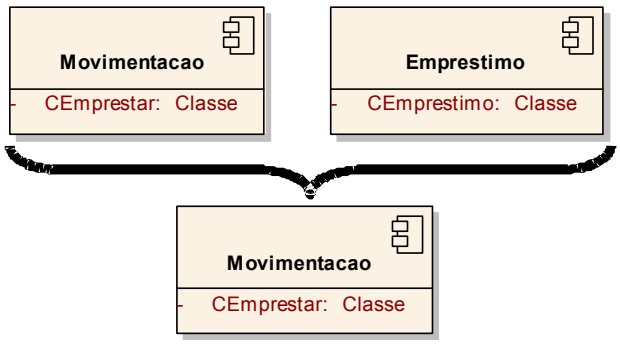


Figura 4.17 – Unificação de componentes dos sistemas para o componente do padrão

Caso não exista o modelo de componentes dos sistemas do domínio, então se deve construir esse modelo analisando-se os diagramas e tabelas que descrevem o padrão. Os

casos de uso genéricos elicitados na forma descrita na seção 4.3.2 podem ser alocados em pacotes que contêm os componentes genéricos do padrão.

Este trabalho da definição dos componentes exige uma minuciosa análise, por parte do projetista, para identificar as funcionalidades que possuem algumas características comuns (relação de interdependência) para colocá-las em um mesmo componente.

4.4 Considerações Finais

Neste capítulo foram apresentados o metamodelo de descrição de padrões arquiteturais e o processo que guia os passos na utilização do metamodelo. O metamodelo caracteriza-se como um conjunto de construtores que podem ser utilizados para a descrição de cada parte do padrão arquitetural. O processo apresenta a forma de instanciação das partes do metamodelo, na composição do padrão arquitetural. O próximo capítulo mostra uma aplicação do metamodelo e do processo propostos.

5 Estudo de Caso

A gram of action is worth more than a ton of theory.

[Friedrich Engels]

Este capítulo apresenta um estudo de caso desenvolvido com o objetivo de mostrar a aplicabilidade do metamodelo proposto. O estudo de caso foi desenvolvido sobre um conjunto de sistemas computacionais para informatização de bibliotecas. São apresentados os passos para se chegar à descrição do padrão bem como a descrição final do padrão.

5.1 Identificação e Descrição do Padrão Arquitetural para os Sistemas de Biblioteca

Esta seção apresenta as fases que foram executadas para a descrição do padrão arquitetural para o domínio de informatização de bibliotecas.

5.1.1 Análise do Domínio

(i) Escolha do Domínio

Nos dias atuais, com o aumento crescente da quantidade de publicações de todos os tipos, é imperativa a utilização de um sistema computacional que seja adequado ao controle do acervo de bibliotecas. Os sistemas de informatização de biblioteca constituem um domínio promissor onde existe uma variedade de tipos de bibliotecas (bibliotecas

pessoais, bibliotecas empresarias, bibliotecas públicas, etc.) aplicadas a diferentes tipos de necessidades.

Especificamente neste caso, por se tratar de um estudo de caso, não houve a identificação de prioridades para a escolha de domínio.

(ii) Seleção de Sistemas e Análise de Custo / Benefício

Esta fase foi executada basicamente através de buscas pela Internet. Foram pesquisados muitos sistemas de bibliotecas pagos e sistemas de código fonte aberto em diferentes linguagens tais como: Delphi, Visual Basic, Clipper, Java e PHP. Foram selecionados quatro sistemas (código livre) implementados em Java e quatro sistemas comerciais, dos quais se obteve apenas documentação sobre o funcionamento dos sistemas.

A atividade de Análise de Custo / Benefício não foi executada, pois se tratava de um estudo de caso acadêmico.

(iii) Extração das Especificações dos Sistemas

Na busca pela modelagem de cada sistema, foram escolhidos quatro sistemas devido as suas características. Por exemplo, os sistemas que apresentavam características de sistemas orientados a objetos foram selecionados visando facilitar o processo de extração dos modelos. Este trabalho demandou um grande esforço devido a grande quantidade de sistemas aplicados a este domínio, porém cada um de fabricante diferente e com características diferentes que dificultaram imensamente o processo de análise.

Houve momentos na extração, que novas seleções tiveram que ser efetuadas, pois verificou-se que a modelagem extraída de alguns sistemas não apresentava características de que o sistema tinha sido construído sob o paradigma da orientação a objetos. Este trabalho foi feito como tentativa de minimizar os problemas de análise de sistemas com características muito diferentes.

5.1.2 Análise dos Requisitos

(i) Análise de Equivalências Funcionais

Esta fase foi desenvolvida da forma descrita a seguir. Todas as documentações disponíveis referentes aos sistemas computacionais objetos de estudo foram reunidas e, então, começou-se a preencher a tabela (Tabela 5.1) contendo os requisitos do primeiro sistema. Terminada a tabela contendo os requisitos do primeiro sistema, criaram-se colunas, na mesma tabela, uma para cada sistema. Essas colunas indicam se o sistema

analisado implementa ou não um determinado caso de uso. Esse método abreviou a etapa de unificação dos casos de uso. Conforme a análise dos sistemas avançou, marcou-se na tabela os casos de uso implementados pelo sistema, acrescentando apenas aqueles casos que não existiam na tabela.

Ainda durante esta etapa, foram estabelecidos os agrupamentos de casos de uso que formaram os subsistemas, tomando-se como base a relação entre as funcionalidades dos casos de uso (*e.g.*: Todos os casos de uso que dizem respeito às tarefas de emprestar, devolver e reservar estão agrupados no subsistema Movimentação / Circulação. Todos os casos de uso concernentes aos cadastramentos estão reunidos no subsistema Cadastros). Outra tarefa executada juntamente nesta etapa foi a identificação dos possíveis usuários do sistema (funcionários da biblioteca) e suas relações com os casos de uso. Ao final da análise dos requisitos de todos os sistemas se obteve a tabela (Tabela 5.1) contendo os casos de uso de todos os sistemas analisados.

Tabela 5.1 – Casos de uso do padrão

Tipo de usuário do sistema	SUBSISTEMA	CASOS DE USO	Sistemas que implementam			
Atendente, Bibliotecário, Chefe	Consulta e geral	Login	1	2	3	
		Consultar o acervo	1	2	3	4
		Sugerir obras para aquisição			3	4
		Alterar Layout de Telas	1		3	4
		Sistema de mensagens	1		3	
Atendente	Movimentação / Circulação	Emprestar exemplar	1	2	3	4
		Consultar histórico de empréstimos de usuário	1	2	3	4
		Devolver exemplar	1	2	3	4
		Emitir aviso de devolução em atraso	1			4
		Renovar empréstimo		2	3	4
		Reservar obra	1	2	3	4
		Cancelar reserva de obra	1	2	3	4
		Consultar reserva	1	2	3	4
		Emitir multa	1	2	3	4
		Cancelar multa	1	2		4
		Ler código de barras	1	2	3	4
Chefe	Gerencial	Emitir relatórios gerenciais	1		3	4
		Emitir gráficos gerenciais			3	4
		Fazer cópia de segurança	1			
		Restaurar cópia de segurança	1			
		Fazer inventário dos exemplares			3	
		Fazer auditoria no sistema				
		Importar catálogo de obras			3	
		Controlar assinaturas de periódicos			3	4
		Atualizar arquivo histórico	1	2	3	

Tipo de usuário do sistema	SUBSISTEMA	CASOS DE USO	Sistemas que implementam			
Bibliotecário	Cadastros	Cadastrar tipo de usuário		2	3	
		Cadastrar usuário	1	2	3	4
		Imprimir carteira de identificação de usuário			3	
		Cadastrar valor de multa por dia	1	2	3	
		Cadastrar feriado		2	3	
		Cadastrar unidade		2	3	
		Cadastrar editora / fornecedor	1		3	
		Cadastrar disciplina	1			4
		Cadastrar curso	1		3	4
		Cadastrar pedido de obra			3	4
		Comunicar solicitante de obra pedida				4
		Cadastrar tipo de obra	1	2	3	4
		Cadastrar obra	1	2	3	4
		Classificar obra			3	4
		Catalogar obra			3	4
		Cadastrar exemplar		2	3	4
		Digitalizar obra	1			4
		Imprimir Etiqueta de lombada de exemplar	1		3	4
		Imprimir código de barras de exemplar	1		3	4
		Controlar legislação e jurisprudência				4

(ii) Elicitação de Requisitos Genéricos

Para limitar este estudo de caso restringiu-se o universo de estudo apenas ao subsistema de Movimentação / Circulação, conforme mostrado na Tabela 5.2. Foram acrescentados a esta tabela colunas para as análises dos casos de uso quanto aos requisitos genéricos de opcionalidade, variabilidade e multiplicidade. Foram acrescentadas também colunas (à direita da tabela) para a análise dos atores participantes de cada caso de uso. Esses atores foram identificados e marcados nas colunas correspondentes da Tabela 5.2.

Nesta fase foram identificados os casos de uso genéricos (variantes, opcionais e múltiplos). A identificação dos casos de uso Opcionais foi feita diretamente através da visualização da Tabela 5.2. Aqueles casos de uso que não são implementados por todos os sistemas analisados foram considerados Opcionais. Os casos de uso implementados por todos os sistemas analisados foram considerados como não opcionais.

Tabela 5.2 – Elicitação de requisitos do subsistema movimentação

Subsistema	Casos de uso	Sistemas que implementam				op	var	mult	Atores				
									Atendente	SGBD	LeitÓtico	SistEmail	Impressor
Movimentação / Circulação	Emprestar exemplar	1	2	3	4		x	x	A	S			I
	Consultar histórico de empréstimos de usuário	1	2	3	4		x	x	A	S			
	Devolver exemplar	1	2	3	4		x	x	A	S			I
	Emitir aviso de devolução em atraso	1			4	x	x	x	A	S		E	
	Renovar empréstimo		2	3	4	x	x	x	A	S			I
	Reservar obra	1	2	3	4		x	x	A	S			
	Cancelar reserva de obra	1	2	3	4		x	x	A	S			
	Consultar reserva	1	2	3	4		x	x	A	S			
	Emitir multa	1	2	3	4		x	x	A				I
	Cancelar multa	1	2		4	x	x	x	A	S			
	Ler código de barras	1	2	3	4		x	x	A		L		

A identificação dos casos de uso Variantes foi feita analisando-se individualmente cada caso de uso para cada sistema em que o caso de uso é implementado, na busca por semelhanças / diferenças que pudessem caracterizar o caso de uso como sendo variante ou não. A Tabela 5.3 mostra o resumo da análise de equivalência funcional do caso de uso Emprestar Exemplar.

Tabela 5.3 – Resumo da equivalência funcional do caso de uso Emprestar Exemplar

SISTEMA 1	SISTEMA 2	SISTEMA 3	SISTEMA 4
No. Aluno	Código da pessoa	Data da Operação	Situação do usuário
Nome aluno	Nome da pessoa	Hora da Operação	Código do usuário
Cód. do professor	No. tombo	Tipo de Operação	Categoria do usuário
No. Obra	Data empréstimo	No. Usuário	Nome do usuário
Título da obra	Data Devolução	No. tombo	Série/Setor do usuário
Série aluno	Operador do Empréstimo	Data prevista para devolução	Tipo de publicação
Nível aluno	Quantidade de Renovações	Data de devolução	Código da publicação
Turma		No. do título na base	Registro da publicação
Data empréstimo		Código do tipo de usuário e objeto emprestado	Classificação da publicação
Data Prevista Devolução		Nome do bibliotecário responsável pela operação	Título da publicação
Data Devolução			Data empréstimo
Data Aviso 1			Prazo
Data Aviso 2			Data devolução
Data Aviso 3			Notas justificativas
			Mensagens de rodapé de recibo

A Tabela 5.3 contém quatro colunas referentes aos quatro sistemas computacionais analisados. Os conteúdos das colunas são os campos que são editados / mostrados quando for executado o caso de uso de empréstimo de exemplar. As cores de fundo das células identificam os campos dos sistemas analisados que são equivalentes, de forma que as células que tiverem a mesma cor representam informações equivalentes entre os sistemas.

Analisando-se dessa forma, pode-se ver que embora esse caso de uso tenha quatro campos que constam em todos os sistemas, existem outros campos que são particulares de cada sistema, fazendo com que esse caso de uso varie de um sistema para outro. Isto é suficiente para rotular este caso de uso como variante. Outras tabelas das análises de equivalência funcional estão no Anexo I. A justificativa para que a totalidade dos casos de uso desse subsistema analisado fosse variante é o fato de todos os sistemas analisados serem de projetos e implementações diferentes. Este fato é que causa uma maior diferença no modo como os casos de uso são construídos.

Terminadas as análises dos casos de uso, construiu-se o modelo genérico dos casos de uso para o subsistema de Movimentação / Circulação do padrão arquitetural do sistema de biblioteca, apresentado na Figura 5.1.

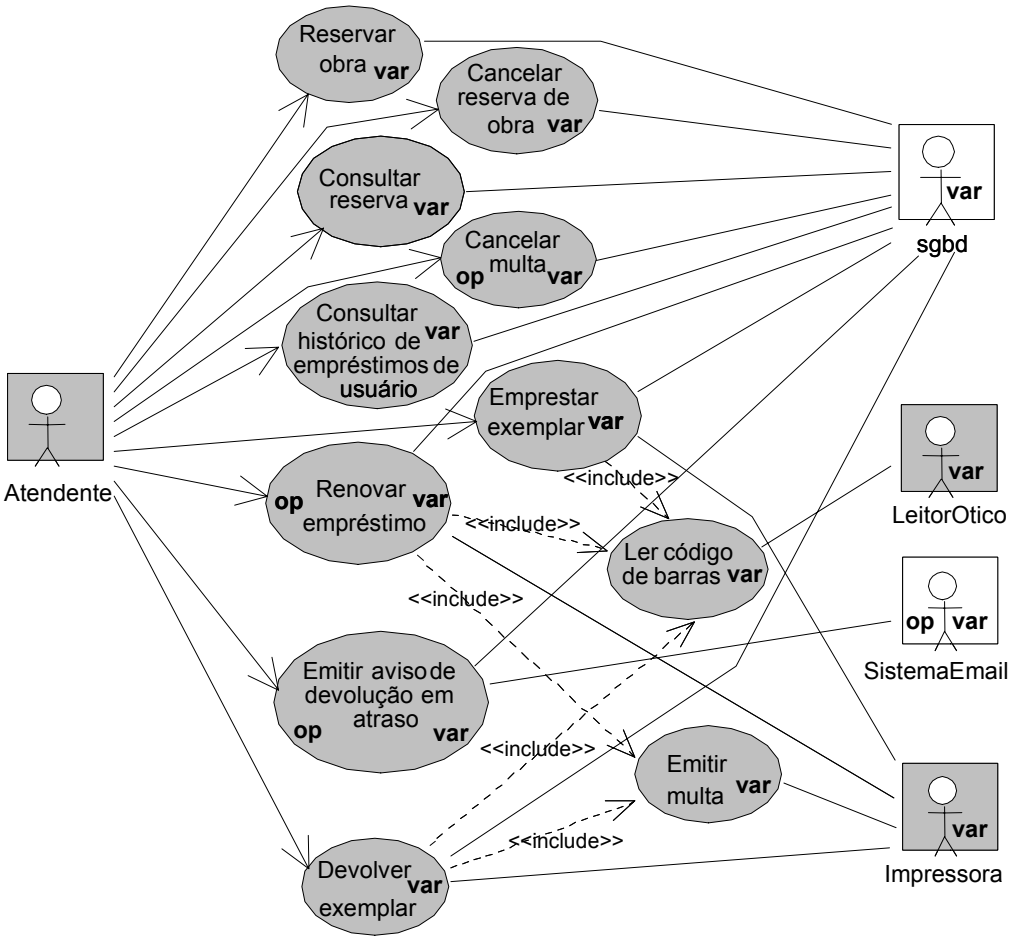


Figura 5.1 – Modelo de Casos de Uso Genérico para o subsistema de movimentação

5.1.3 Análise das Arquiteturas

(i) Análise de Classes

A análise de classes foi executada sobre os modelos dos quatro sistemas pertencentes ao domínio de aplicação do padrão arquitetural. As classes selecionadas para efeito de exemplificação neste estudo são referentes às entidades **Livro** e **Exemplar**.

Dois sistemas do domínio analisado têm duas classes, uma para os objetos da classe Livro e outra para os objetos da classe Exemplar. Os outros dois sistemas analisados apresentam apenas a classe Livro. Foi decisão de projeto que o padrão permanecesse com duas classes, uma para os objetos da classe Livro e outra para os objetos da classe Exemplar. Esta decisão foi tomada levando-se em consideração que um livro pode ter mais de um exemplar.

A Figura 5.2 apresenta, nas laterais, as classes dos quatro sistemas analisados e, na coluna central, as classes do padrão arquitetural para o domínio de aplicação. Estas classes formam uma classe de equivalência para as entidades Livro e Exemplar. Cada atributo das classes dos sistemas tem um correspondente nas classes do padrão. As classes dos sistemas tiveram seus atributos analisados para se descobrir as relações de equivalências. Assim, o atributo correspondente da classe no padrão foi criado. Na Figura 5.2, as ligações foram coloridas apenas para facilitar a visualização.

Os itens a seguir apresentam as classes de equivalência dos atributos para a classe Livro:

1. O atributo **ISBN** do padrão é um String, criado a partir dos atributos ISBN:String (Biblio11), BookID:Int (Biblio28), ISSN:Int (Biblio20) e ISBN:String (Biblio 27).
2. O atributo **titulo** do padrão é um String, criado a partir dos atributos titulo:String (Biblio11), BookName:Int (Biblio28), title:Int (Biblio20) e title:String (Biblio 27).
3. O atributo **autor** do padrão é um String, criado a partir dos atributos autor:String (Biblio11), BookAuthor:String (Biblio28), AUTHOR:Int (Biblio20) e author:String (Biblio 27).
4. O atributo **areaInteresse** do padrão é um String, criado a partir dos atributos areaInteresse:String (Biblio11) e KEYWORDS:Int (Biblio20).
5. O atributo **edicao** do padrão é um String, criado a partir do atributo edicao:String (Biblio11).

6. O atributo **publicador** do padrão é um String, criado a partir dos atributos editora:String (Biblio11) e PUBLISHER:Int (Biblio20).
7. O atributo **localDePublicacao** do padrão é um String, criado a partir do atributo PL_OF_PUB:Int (Biblio20).
8. O atributo **dataDePublicacao** do padrão é um String, criado a partir do atributo PUB_DATE:Int (Biblio20).
9. O atributo **regular** do padrão é um Int, criado a partir do atributo REGULAR:Int (Biblio27).
10. O atributo **reserva** do padrão é um Int, criado a partir do atributo RESERVE:Int (Biblio27).
11. O atributo **raro** do padrão é um Int, criado a partir do atributo RARE:Int (Biblio27).
12. O atributo **copias** do padrão é um Int, criado a partir do atributo COPIES:Int (Biblio20).

A análise para os atributos da Classe Exemplar foi feita à semelhança da análise feita para a classe Livro. Os itens a seguir apresentam as classes de equivalência dos atributos para a classe Exemplar:

1. O atributo **ISBN** do padrão é um String, criado a partir dos atributos livro:CLivro (Biblio11) e ISBN:Int (Biblio20).
2. O atributo **numeroExemplar** do padrão é um Int, criado a partir dos atributos numeroRegistro:Int (Biblio11) e ITEM_NO:Int (Biblio20).
3. O atributo **ultimoUsuario** do padrão é um CUsuarioComum, criado a partir do atributo ultimoUsuario: CUsuarioComum (Biblio11).
4. O atributo **dataEmprestimo** do padrão é um Date, criado a partir do atributo dataEmprestimo: Date (Biblio11).
5. O atributo **disponivel** do padrão é um boolean, criado a partir dos atributos disponivel:boolean (Biblio11), BookStatus:String (Biblio28) e reservedBy:Cardholder (Biblio27).
6. O atributo **maxDiasEmprestimo** do padrão é um Int, criado a partir do atributo maxDiasEmprestimo:Int (Biblio11).
7. O atributo **multaPorDia** do padrão é um float, criado a partir do atributo multaPorDia:float (Biblio11).

8. O atributo **novoId** do padrão é um Int, criado a partir do atributo novoId:int (Biblio11).
9. O atributo **saidaChecadaPor** do padrão é um Cardholder, criado a partir do atributo checkedOutBy:Cardholder (Biblio27).
10. O atributo **salaDeLeitura** do padrão é um Int, criado a partir do atributo READINGROOM:Int (Biblio27).
11. O atributo **numeroClasse** do padrão é um Int, criado a partir do atributo CLASS_NO:Int (Biblio20).
12. O atributo **informacoesDeCopia** do padrão é um Int, criado a partir do atributo COPY_INFO:Int (Biblio20).
13. O atributo **localizacao** do padrão é um Int, criado a partir do atributo location:Int (Biblio27).

É conveniente notar que existem atributos que não foram colocados nas classes do padrão visto que estes atributos são específicos de implementação e dependem da linguagem adotada.

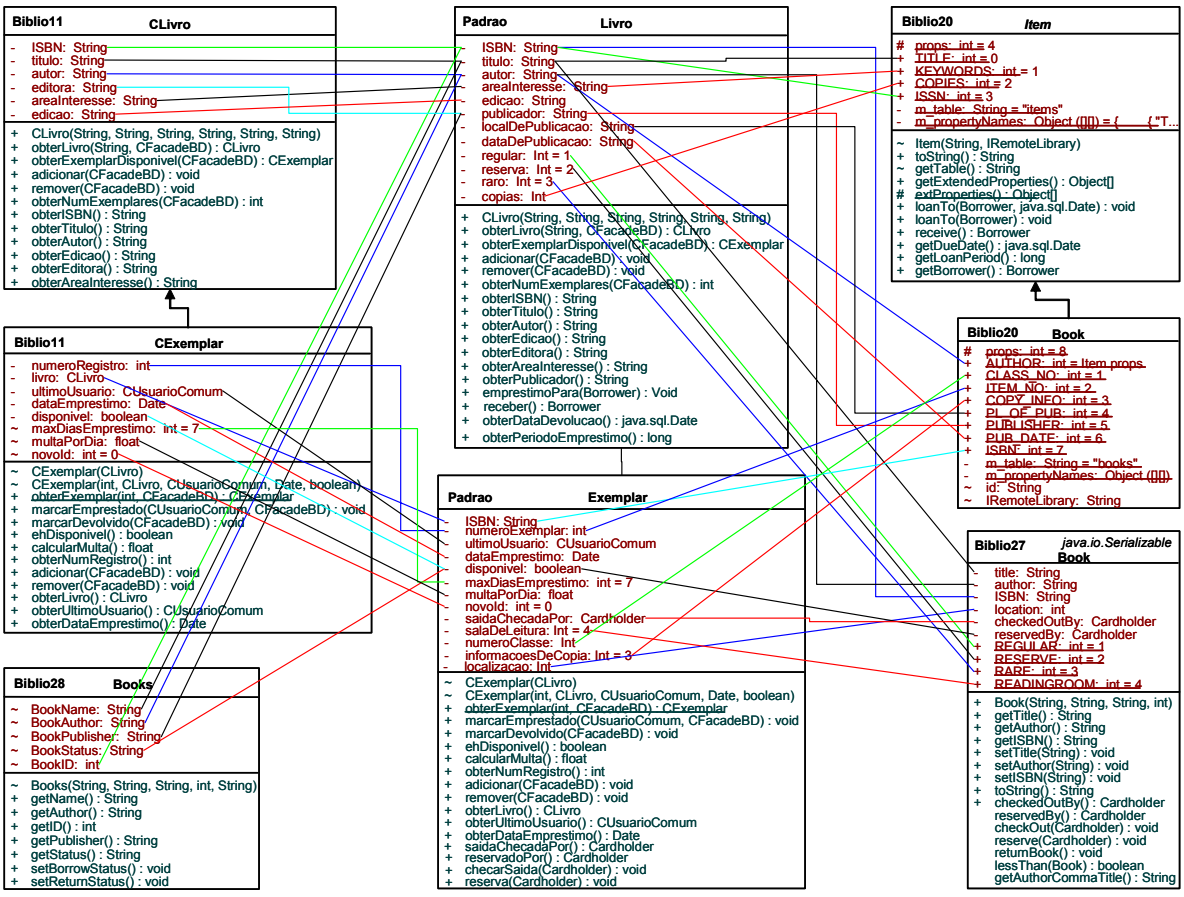


Figura 5.2 – Correspondência entre os atributos das classes dos sistemas analisados e os atributos do padrão

As mesmas classes (da Figura 5.2) foram analisadas quanto aos métodos. Os métodos das classes dos sistemas (colunas laterais) foram então agrupados nas classes de equivalência. As classes de equivalência formaram os métodos correspondentes do padrão (coluna central), tanto para a classe Livro, quanto para a classe Exemplar. A Figura 5.3 apresenta as ligações entre os métodos dos sistemas e os métodos correspondentes do padrão. As ligações na Figura 5.3 foram coloridas apenas para facilitar a visualização.

Os itens a seguir apresentam as classes de equivalência dos métodos para a classe Livro:

1. O método **CLivro**(String, String, String, String, String, String) do padrão é criado a partir dos métodos CLivro(String, String, String, String, String, String) (Biblio11), Books(String, String, String, int, String) (Biblio28) e Book(String, String, String, int) (Biblio27).
2. O método **obterLivro**(String, CFacadeBD): CLivro do padrão é criado a partir dos métodos obterLivro(String, CFacadeBD): CLivro (Biblio11) e getID(): int (Biblio28).
3. O método **obterExemplarDisponivel**(CFacadeBD): Cexemplar do padrão é criado a partir do método obterExemplarDisponivel(CFacadeBD): CExemplar (Biblio11).
4. O método **adicionar**(CFacadeBD): void do padrão é criado a partir do método adicionar(CFacadeBD): void (Biblio11).
5. O método **remover**(CFacadeBD): void do padrão é criado a partir do método remover(CFacadeBD): void (Biblio11).
6. O método **obterNumExemplares**(CFacadeBD): int do padrão é criado a partir do método obterNumExemplares(CFacadeBD): int (Biblio11).
7. O método **obterISBN**(): String do padrão é criado a partir dos métodos obterISBN(): String (Biblio11), **getISBN**(): String (Biblio27) e setISBN(String): void (Biblio27).
8. O método **obterTitulo**(): String do padrão é criado a partir dos métodos obterTitulo(): String (Biblio11), getName(): String (Biblio28), getTitle(): String (Biblio27) e setTitle(String): void (Biblio27).
9. O método **obterAutor**(): String do padrão é criado a partir dos métodos obterAutor(): String (Biblio11), getAuthor(): String (Biblio28), getAuthor(): String (Biblio27) e setAuthor(String) : void (Biblio27).
10. O método **obterEdicao**(): String do padrão é criado a partir do método obterEdicao(): String (Biblio11).

11. O método **obterEditora**(): String do padrão é criado a partir do método obterEditora(): String (Biblio11).
12. O método **obterAreaInteresse**(): String do padrão é criado a partir do método obterAreaInteresse(): String (Biblio11).
13. O método **obterPublicador**(): String do padrão é criado a partir do método getPublisher(): String (Biblio28).
14. O método **emprestimoPara**(Borrower): Void do padrão é criado a partir dos métodos loanTo(Borrower, java.sql.Date): void (Biblio20) e loanTo(Borrower): void (Biblio20).
15. O método **receber**(): Borrower do padrão é criado a partir do método receive(): Borrower (Biblio20).
16. O método **obterDataDevolucao**(): java.sql.Date do padrão é criado a partir do método getDueDate(): java.sql.Date (Biblio20).
17. O método **obterPeriodoEmprestimo**(): long do padrão é criado a partir do método getLoanPeriod(): long (Biblio20).

A definição dos métodos da classe Exemplar seguiu a mesma seqüência da análise feita para os métodos da classe Livro. Os itens a seguir apresentam as classes de equivalência dos métodos para a classe Exemplar:

1. O método **CExemplar**(CLivro) do padrão é criado a partir do método CExemplar(CLivro) (Biblio11).
2. O método **CExemplar**(int, CLivro, CUsuarioComum, Date, boolean) do padrão é criado a partir do método CExemplar(int, CLivro, CUsuarioComum, Date, boolean) (Biblio11).
3. O método **obterExemplar**(int, CFacadeBD): Cexemplar do padrão é criado a partir do método obterExemplar(int, CFacadeBD): Cexemplar (Biblio11).
4. O método **marcarEmprestado**(CUsuarioComum, CFacadeBD): void do padrão é criado a partir dos métodos marcarEmprestado(CUsuarioComum, CFacadeBD): void (Biblio11) e setBorrowStatus(): void (Biblio28).
5. O método **marcarDevolvido**(CFacadeBD): void do padrão é criado a partir dos métodos marcarDevolvido(CFacadeBD): void (Biblio11) e setReturnStatus(): void (Biblio28).

6. O método **ehDisponivel**(): boolean do padrão é criado a partir dos métodos ehDisponivel(): boolean (Biblio11) e getStatus(): String (Biblio28).
7. O método **calcularMulta**(): float do padrão é criado a partir do método calcularMulta(): float (Biblio11).
8. O método **obterNumRegistro**(): int do padrão é criado a partir do método obterNumRegistro(): int (Biblio11).
9. O método **adicionar**(CFacadeBD): void do padrão é criado a partir do método adicionar(CFacadeBD): void (Biblio11).
10. O método **remover**(CFacadeBD): void do padrão é criado a partir do método remover(CFacadeBD): void (Biblio11).
11. O método **obterLivro**(): Clivro do padrão é criado a partir dos métodos obterLivro(): Clivro (Biblio11) e returnBook(): void (Biblio27).
12. O método **obterUltimoUsuario**(): CusuarioComum do padrão é criado a partir dos métodos obterUltimoUsuario(): CusuarioComum (Biblio11) e getBorrower(): Borrower (Biblio20).
13. O método **obterDataEmprestimo**(): Date do padrão é criado a partir do método obterDataEmprestimo(): Date (Biblio11).
14. O método **saidaChecadaPor**(): Cardholder do padrão é criado a partir do método checkedOutBy(): Cardholder (Biblio27).
15. O método **reservadoPor**(): Cardholder do padrão é criado a partir do método reservedBy(): Cardholder (Biblio27).
16. O método **checarSaida**(Cardholder): void do padrão é criado a partir do método checkOut(Cardholder): void (Biblio27).
17. O método **reserva**(Cardholder): void do padrão é criado a partir do método reserve(Cardholder): void (Biblio27).

Convém notar que existem métodos, das classes analisadas, que estão relacionados a aspectos de implementação e não de projeto. Então, por decisão de projeto do padrão, optou-se por não colocar estes métodos nas classes do padrão.

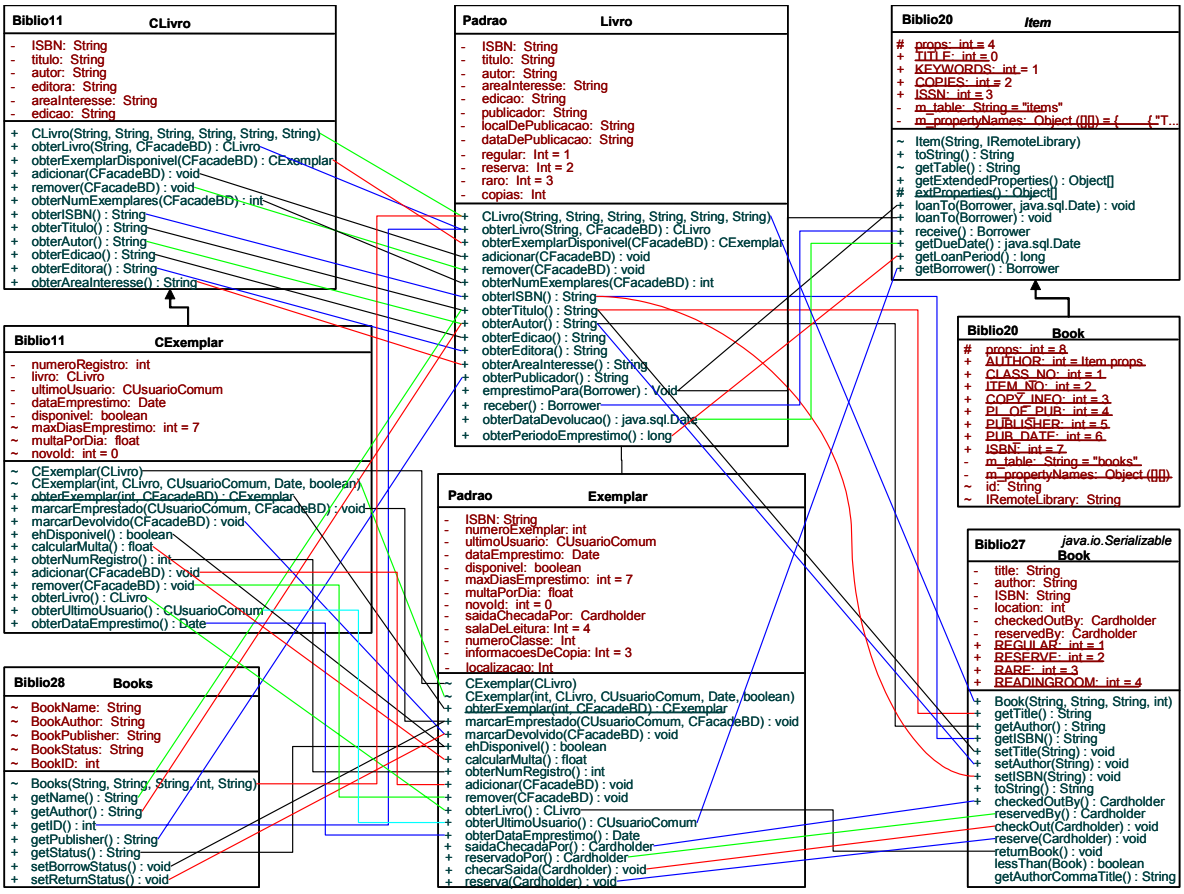


Figura 5.3 – Correspondência entre os métodos das classes dos sistemas analisados e os métodos do padrão

Por meio das análises das classes dos sistemas obtiveram-se as classes do padrão. A Figura 5.4 apresenta as classes de entidade do padrão, considerando-se o caso de uso Emprestar Exemplar.

Figura 5.4 – Classes de entidade do padrão

Após a obtenção das classes do padrão, analisou-se a Figura 5.1 para determinar os estereótipos das classes. A Figura 5.5 apresenta as classes de interface. A classe de interface do SGBD foi definida com o estereótipo <<VAR>>; a classe de interface do Leitor Ótico e a classe de interface da Impressora foram definidas com o estereótipo <<VAR>>; a classe de interface do Sistema de E-mail foi definida com <<OP_VAR>>.

As outras classes (interface e controle) foram criadas e não extraídas, pois a diferença entre as implementações dos sistemas era grande.

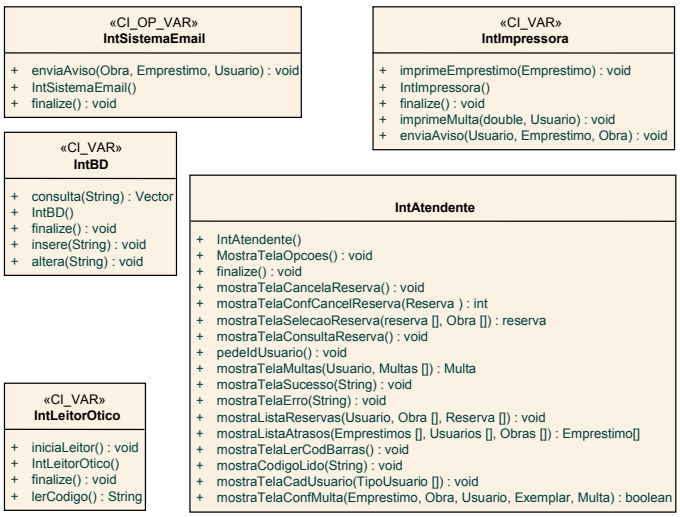


Figura 5.5 – Classes de interface

A Figura 5.6 apresenta as classes de controle e o Anexo III oferece uma visão geral de todas as classes do padrão arquitetural.

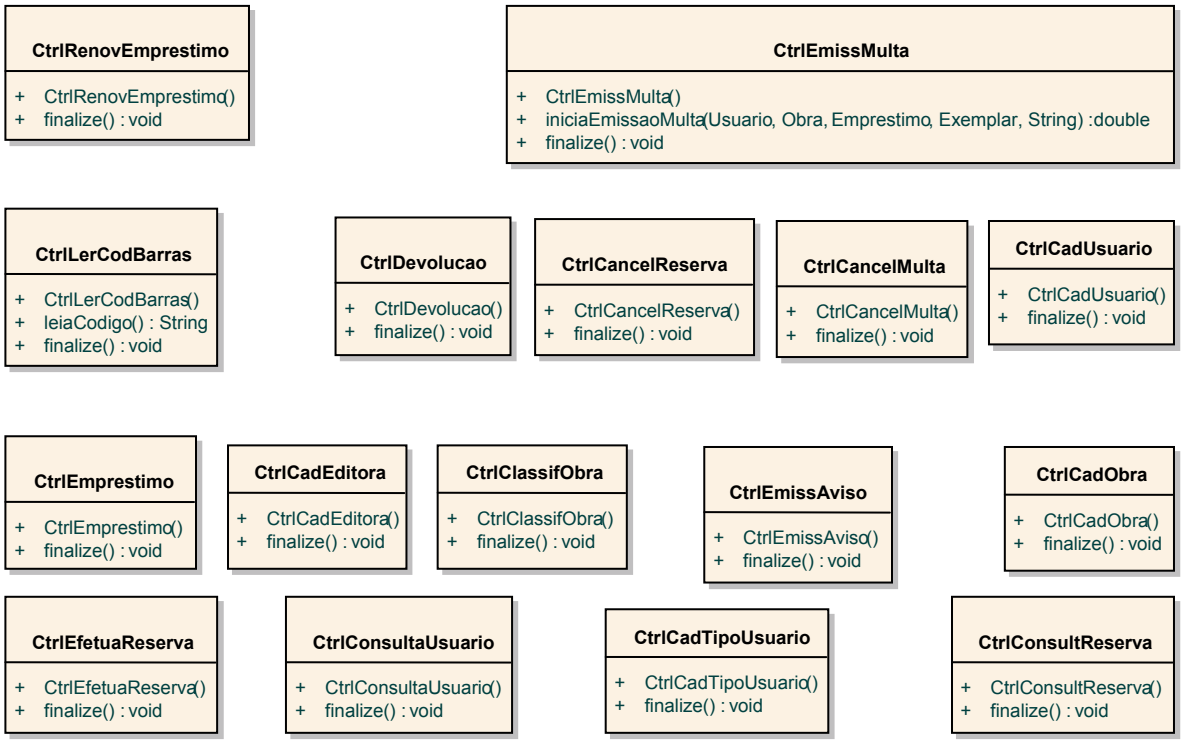


Figura 5.6 – Classes de controle

(ii) Análise de Interações

A análise de interações é feita para o levantamento das comunicações entre os objetos de um sistema computacional. Os diagramas de seqüência são importantes nessa fase por apresentarem ao longo de uma linha de tempo, a seqüência de comunicações entre os objetos do padrão.

Neste estudo de caso, devido à ausência dos diagramas de seqüência dos sistemas objeto de estudo recorreu-se a uma técnica diferenciada para se separar as partes variantes das invariantes no diagrama de seqüência. Primeiro desenvolveu-se os diagramas de seqüência com base nas informações já levantadas e nos diagramas do padrão que já estavam prontos (diagrama de casos de uso e diagramas de classes). Segundo, através da análise dos estereótipos das classes, pode-se constatar quais partes do diagrama de seqüência correspondente seriam potencialmente variantes entre os sistemas analisados.

Procurou-se, no diagrama de seqüência as classes variantes. Assim, pode-se traçar o retângulo diferenciador no diagrama de seqüência distinguindo a parte variante da não variante.

A Figura 5.7 apresenta o diagrama de seqüência para o caso de uso Emprestar Exemplar. Neste diagrama pode-se notar que a parte variante corresponde a chamada dos métodos das classes variantes. Outra parte diferente são os ícones dos atores que representam o SGBD e a impressora. Estes, são aqui também representados como variantes e a impressora é ainda um ator múltiplo, conforme a notação da seção 3.1.

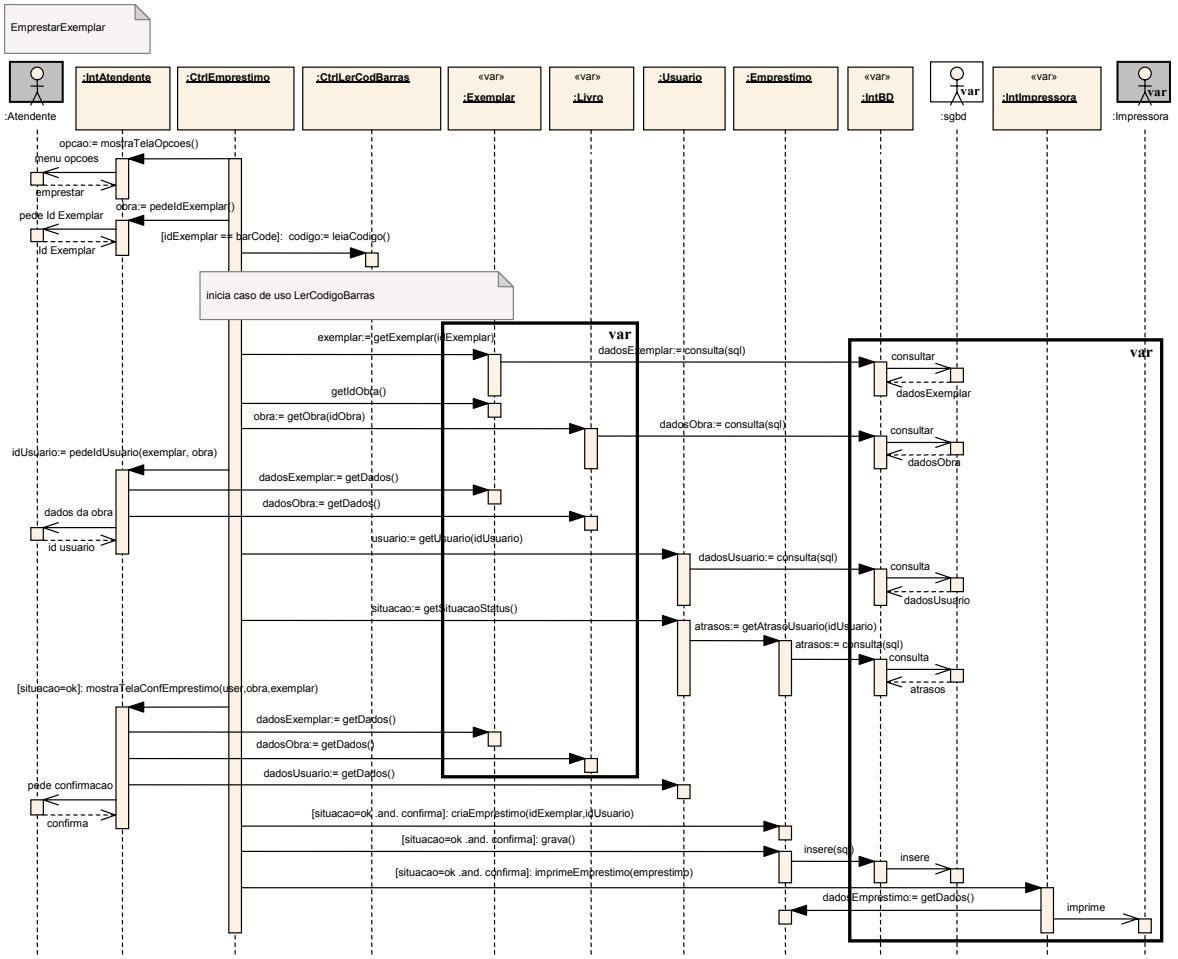


Figura 5.7 – Diagrama de seqüência para o caso de uso Emprestar Exemplar

(iii) Análise de Relacionamentos entre as Classes

Os relacionamentos entre classes podem ser obtidos a partir da análise dos diagramas de seqüência. Deste modo, para o padrão arquitetural em estudo buscou-se, nos diagramas de seqüência do Anexo II, os relacionamentos existentes.

A análise do diagrama de seqüência da Figura 5.7 mostra os relacionamentos entre os objetos das classes do padrão para o caso de uso Emprestar Exemplar. Estes relacionamentos são apresentados na Figura 5.8. Os relacionamentos com linha e rótulo na cor azul denotam relacionamentos que são ativados (no diagrama de seqüência da Figura 5.7) dentro do retângulo que denota a parte variante (**var)**. Essa forma de colorir os

relacionamentos dos diagramas de classes do padrão facilita a visualização dos relacionamentos que são variantes.

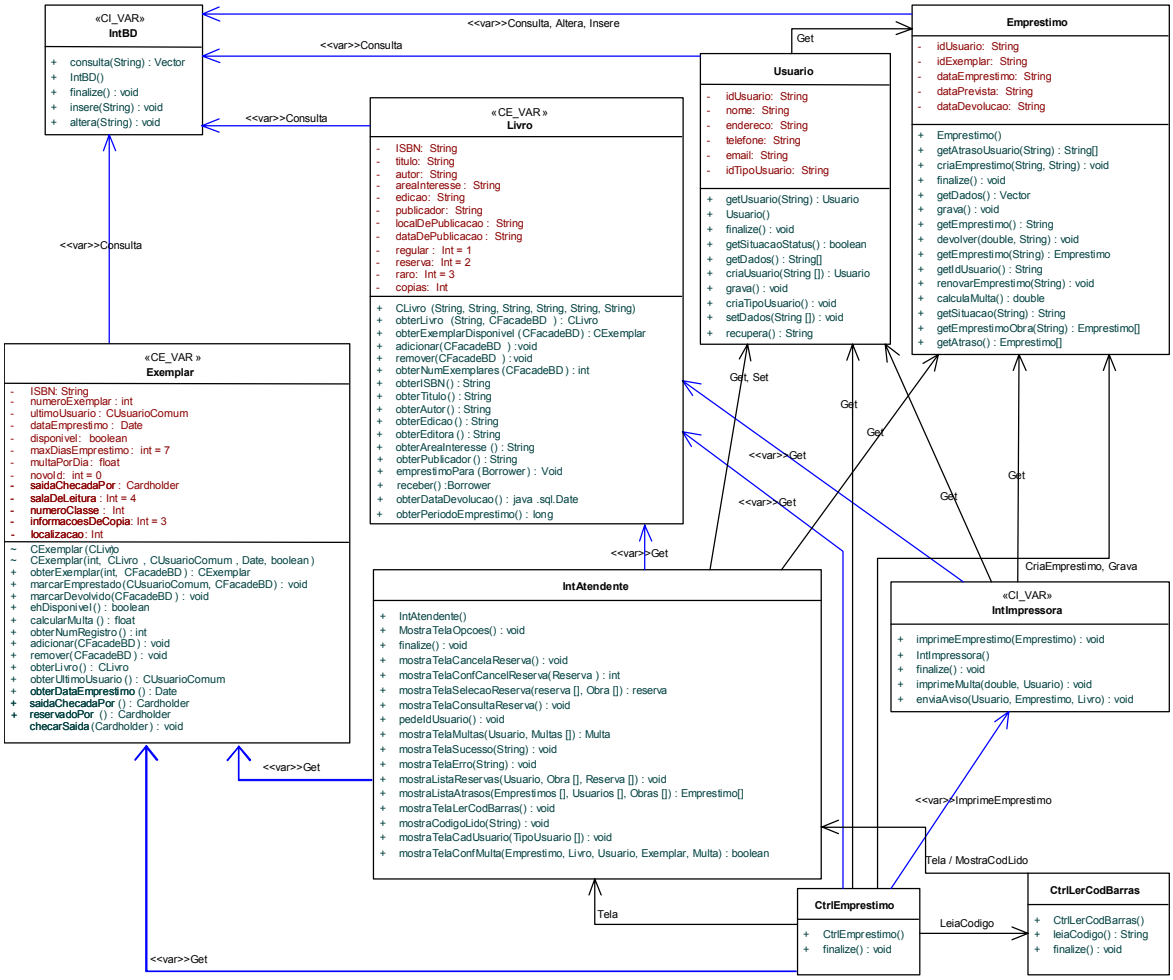


Figura 5.8 – Diagrama de classes para o caso de uso Emprestar Exemplar

As análises dos demais diagramas de seqüência do Anexo II foram executadas e os relacionamentos foram todos colocados nas figuras do Anexo III.

É pertinente lembrar que estes relacionamentos foram levantados através da análise dos diagramas, não impedindo que o projetista possa fazer outras análises e levantar outros relacionamentos tais com os relacionamentos de agregação e os de generalização / especialização.

(iv) Análise de Estados

A documentação dos sistemas que compõem o domínio de aplicação não possui os diagramas de estados. Portanto, os diagramas de estados para as classes do padrão foram construídos segundo as convenções da UML, utilizando-se as informações dos diagramas do padrão que já estavam prontos, em especial os diagramas de seqüência.

Após a construção dos diagramas, o próximo passo foi a representação da parte variante e opcional. Para isso utilizaram-se os passos descritos a seguir:

1. Verificou-se, em cada diagrama de seqüência do padrão, as partes variantes e opcionais para o diagrama de estados representativo de cada classe.
2. Assinalou-se os estados cujas mensagens estão em área variante ou opcional nos diagramas de seqüência.
3. O diagrama de estado foi demarcado com um superestado denominado **ativo** representativo da parte variante e opcional.

A Figura 5.9 apresenta o diagrama de estados para a classe Emprestimo. O conteúdo do superestado **ativo** representa a parte que foi assinalada como variante nos diagramas de seqüência do padrão.

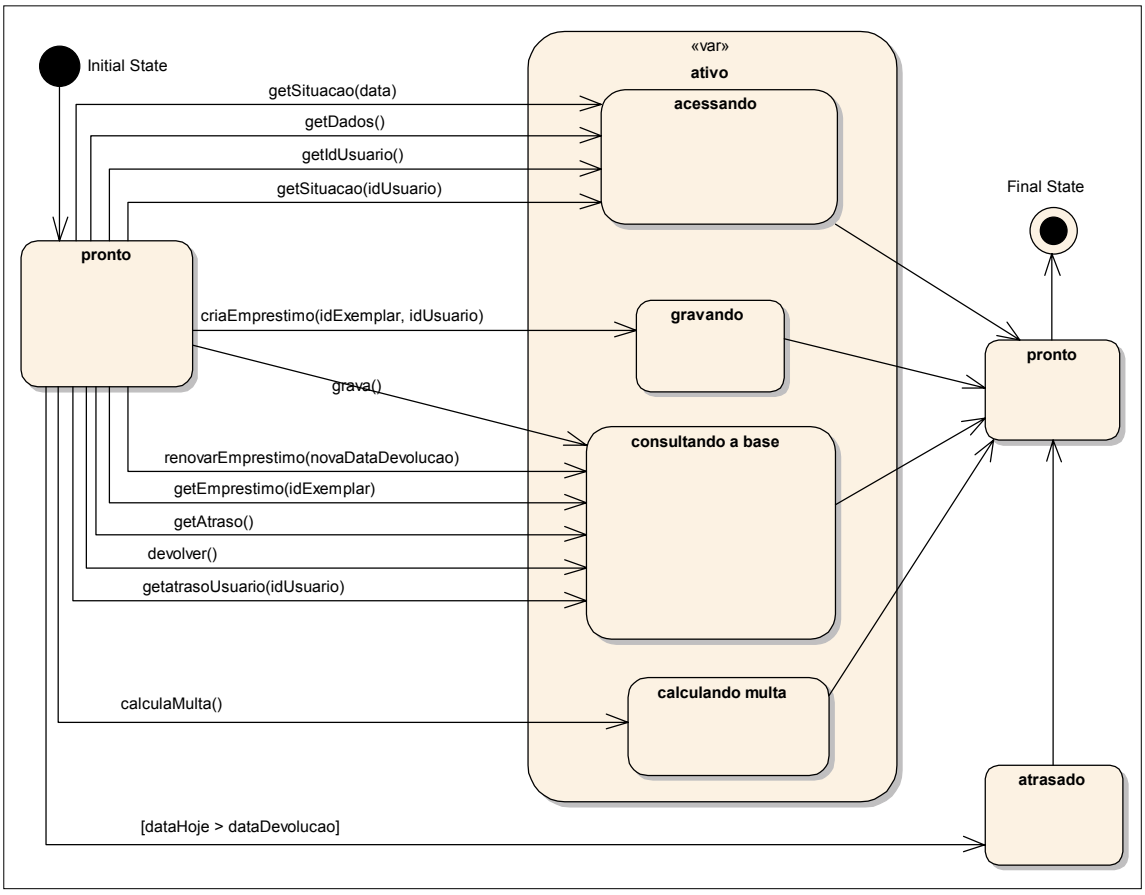


Figura 5.9 – Diagrama de estados para a classe Emprestimo

(v) Análise de Componentes

A documentação dos sistemas pertencentes ao domínio de aplicação do padrão arquitetural não continha os diagramas de componentes de cada sistema. Portanto, para a formação do diagrama de componentes do padrão arquitetural, baseou-se nas informações

coletadas nos requisitos dos sistemas e nos diagramas que já foram elaborados para o padrão.

Foram formados componentes e pacotes. Cada caso de uso que continha um elemento variante, opcional ou múltiplo foi colocado em um pacote ou componente separado. A Figura 5.10 apresenta o diagrama de componentes e pacotes do subsistema movimentação.

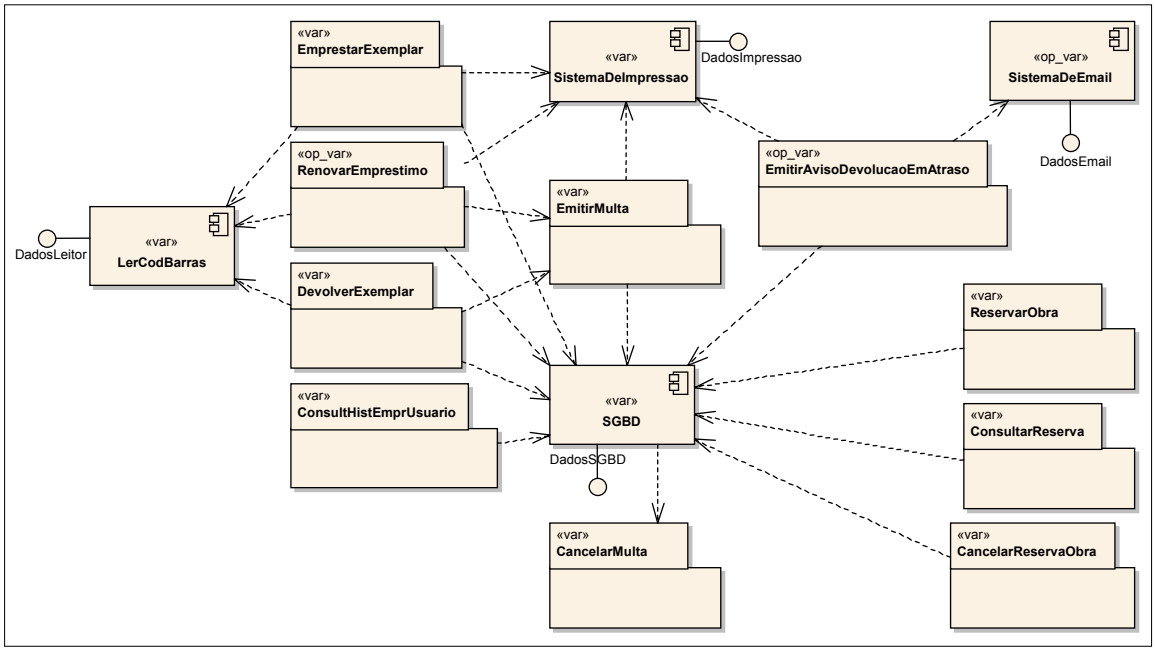


Figura 5.10 – Diagrama de componentes para o subsistema movimentação

É importante salientar que a definição da arquitetura do padrão demanda do projetista o conhecimento do domínio de aplicação, para se identificar e agrupar componentes e pacotes que tenham relacionamentos.

5.1.4 Catalogação do Padrão Arquitetural

O padrão arquitetural aqui apresentado é um padrão voltado para o domínio de informatização de bibliotecas. Tendo em vista que este padrão foi discutido e apresentado em seções deste capítulo e anexos, as informações que já foram mostradas ou apontadas em anexos terão mencionado apenas a sua localização.

Seguindo o metamodelo para catalogação que foi proposto no capítulo 4, notadamente conforme as Figuras 4.2 a 4.4 tem-se:

DEFINIÇÃO DO PADRÃO:

CIDENTIFICACAO:

1. **Identificador**: InformatizaçãoDeBibliotecas
2. **Autor**: Marcos Antonio Quináia
3. **Versão**: 1.0

CPROBLEMA:

1. **Domínio**: este padrão arquitetural é aplicável ao domínio de informatização de bibliotecas.
2. **Escopo**: este padrão abrange os sistemas de informatização de bibliotecas de diversos tamanhos, mas que seja uma biblioteca com características de biblioteca pública, onde usuários da biblioteca podem fazer empréstimos, devoluções e reservas de obras.
3. **Enunciado**: o objetivo deste padrão é possibilitar a sua utilização no projeto e posterior construção de sistemas de informatização de bibliotecas.

CSOLUCAO:

1. **Estilo**: Este padrão arquitetural pertence ao estilo MVC [GAMMA et al 1995; CLEMENTS et al 2003], por ser um padrão para sistemas de interação com usuários.
2. **Padrões Relacionados**: BibliotecasParticulares.

CMOTIVACAO:

1. **Vantagens**: a principal vantagem da utilização deste padrão é que o mesmo já reúne informações referentes ao domínio de aplicação compatível com a atuação de sistemas de informatização de bibliotecas. Este padrão já possui um estudo e detalhamento de projeto que pode ser reutilizado quando se for projetar e construir algum sistema de biblioteca.

2. **Restrições**: Um dos aspectos negativos da componentização é que nem todos os componentes servem para os requisitos de todos os clientes que necessitem deste componente. Entretanto, como os componentes do padrão arquitetural são descritos de forma a evidenciar as partes opcionais, variantes e múltiplas, isto ameniza / neutraliza esta restrição pois os componentes que não forem requeridos não precisam ser implementados visto que são opcionais e os componentes que forem variantes poderão ser implementados de diferentes formas nos sistemas computacionais oriundos do padrão arquitetural.

MODELO ARQUITETURAL DO PADRÃO:

MODELO DE REQUISITOS

O modelo de requisitos, composto pelo diagrama de casos de uso, foi apresentado na seção 5.1.2 e complementado com o Anexo I.

MODELO ESTRUTURAL

O modelo estrutural, composto pelo diagrama de classes, foi apresentado na seção 5.1.3 e complementado com o Anexo III.

MODELO DA DINÂMICA

O modelo da dinâmica, composto pelos diagramas de seqüência e de estados, foi apresentado na seção 5.1.3 e complementado com o Anexo II, com exceção dos diagramas de colaboração que não foram implementados.

MODELO DE IMPLEMENTAÇÃO

O modelo de implementação, composto pelo diagrama de componentes, foi apresentado na seção 5.1.3.

5.2 Considerações Finais

Neste capítulo apresentou-se um estudo de caso sobre a criação de um padrão arquitetural aplicado ao domínio de sistemas de controle de biblioteca. Neste estudo procurou-se avaliar a aplicabilidade das tipologias propostas (capítulo 3), na forma do metamodelo e seguindo o processo, que foram propostos no capítulo 4. O estudo de caso abrangeu desde a seleção de sistemas do domínio, passando pela elicitação dos requisitos, e terminando com a análise e composição da arquitetura do padrão arquitetural, onde foram elicitadas as partes opcionais, variantes e múltiplas do padrão.

6 Conclusões e Perspectivas Futuras

No sensible decision can be taken without considering that the world is not only what it is, but what it is going to be.

[Isaac Asimov]

Existem vários fatores que motivaram essa pesquisa no contexto de reutilização de *software* empregando padrões arquiteturais em domínios de aplicação específicos. Alguns desses fatores são:

1. Aumento na complexidade do processo de desenvolvimento de *software*;
2. Na medida em que cresce a industrialização de *software*, o reuso desponta como fator preponderante no processo de desenvolvimento de *software*.
3. A constante busca por melhorias em métodos, modelagem e especificação de sistemas computacionais que têm o objetivo de facilitar a representação dos componentes, das relações e da estrutura dos sistemas computacionais;
4. A grande aceitação e disseminação dos padrões de *software* que procuram facilitar a reutilização de partes de outros sistemas em novos desenvolvimentos;
5. A carência de métodos e modelos que descrevam soluções arquiteturais para domínios específicos;

Tendo em mente esses fatores motivadores, foi possível visualizar o desenvolvimento de uma contribuição que focasse a reutilização de arquiteturas de *software* na forma de padrões.

6.1 Conclusões

Este trabalho teve como foco o desenvolvimento de um metamodelo para descrição de padrões de arquitetura de *software* para aplicação em um domínio específico. O objetivo básico motivador desta tese é oferecer uma contribuição para a reutilização de arquiteturas de *software*. Esta reutilização se faz na forma dos padrões de arquitetura de *software* que forem criados através do emprego do metamodelo proposto. Ela se materializa nas instanciações efetuadas a partir dos padrões criados, gerando sistemas computacionais.

A contribuição deste trabalho está diretamente relacionada com a facilidade de identificação e descrição de padrões arquiteturais. Todo modelo de processo é prescritivo e, de certa forma, almeja aumentar a velocidade em alcançar os resultados propostos. Assim, a existência de um método como o proposto contribuirá para uma maior rapidez e precisão na identificação de padrões arquiteturais.

A possibilidade de caracterização de padrões que são aplicáveis a determinados domínios de aplicação representa uma contribuição potencial. Esta caracterização é feita através da elicitação de diferentes características (partes variantes, invariantes, múltiplas e opcionais) presentes no padrão arquitetural descrito. Outra forma de caracterização seria possível através da descoberta de diferentes tipos de padrões que podem ser aplicados a um determinado domínio de aplicação.

Com relação ao metamodelo proposto, sua contribuição pode ser expressa como uma melhor forma de representação dos padrões. A expectativa é que o metamodelo criado neste trabalho venha a contribuir para facilitar a identificação, especificação e caracterização de padrões arquiteturais e suas partes constantes, opcionais, variantes e múltiplas.

Neste trabalho foi apresentada uma extensão desenvolvida para a especificação de requisitos genéricos que podem ser variantes, múltiplos e opcionais de uma arquitetura de *software*. Com isso tem-se uma contribuição que favorece a especificação de detalhes dos requisitos de padrões arquiteturais.

A vantagem da utilização desta extensão é possibilitar uma adequada visualização dos atores e casos de uso que, além de genéricos, são também variantes, opcionais e múltiplos. Esta visualização possibilita a identificação de atores e casos de uso que possam ser usados em diferentes situações que gerem instanciações diferenciadas para o mesmo domínio.

Foram desenvolvidas também outras tipologias (diagramas de classes genéricos, diagramas de seqüência genéricos, diagramas de estados genéricos e diagramas de componentes genéricos) aplicáveis à especificação de modelos, evidenciando os aspectos de opcionalidade e variabilidade, que são diferenciais de cada modelo. Uma contribuição advinda do uso dos ícones e notações propostos nessas tipologias é a descrição diferenciada dos elementos opcionais, variantes e invariantes do padrão. Esta identificação possibilita a verificação do grau de flexibilidade ou adaptabilidade do padrão a outras situações de implementação para o domínio de aplicação do padrão.

A vantagem da utilização do diagrama de classes genérico é mostrar em uma estrutura genérica, as classes que possuem características de opcionalidade e variabilidade. A vantagem da utilização do diagrama de seqüência genérico é visualisar com mais precisão as partes das seqüências de chamadas dos métodos entre os objetos de classes que são opcionais e variantes. A vantagem da utilização do diagrama de estados genérico é poder verificar quais mudanças no comportamento da classe genérica afetarão os seus estados. A vantagem da utilização do diagrama de componentes genéricos é a exata tradução (mapeamento) dos requisitos para os componentes que conterão as partes genéricas do padrão. Esta tradução (mapeamento) permite auxiliar a verificar aspectos da coesão dos componentes, bem como identificar os componentes do padrão que poderão sofrer variações nas instâncias.

O metamodelo proposto permite a criação de padrões de arquiteturas de *software* que sejam possivelmente mais flexíveis e adaptáveis a várias situações diferentes dentro do mesmo domínio, aumentando as chances de reutilização do padrão.

Um fator restritivo à utilização deste metamodelo proposto é a sua dependência da linguagem UML, no sentido de que o metamodelo foi feito para ser utilizado com a UML.

6.2 Publicações

Ao longo do desenvolvimento deste trabalho, algumas publicações (descritas a seguir) foram feitas reforçando as motivações para a continuidade da pesquisa. Nos

eventos onde os trabalhos foram apresentados surgiram trocas de experiências e realimentações positivas para melhorias nas implementações dos objetivos.

- *Identificação de Padrões Arquiteturais Usando Engenharia Reversa*. Marcos Antonio Quináia e Paulo Cézar Stadzisz. In Proceedings do WMSWM/SBES'04, Brasília, 2004.
- *A Use Case Extension for Architectural Patterns*. Marcos Antonio Quináia e Paulo Cézar Stadzisz. In Proceedings do CSITeA'03, Rio de Janeiro, 2003.
- *A Pattern System to Supervisory Control of Automated Manufacturing System*. Paulo Cezar Stadzisz; Jean Marcelo Simão e Marcos Antonio Quináia. In Proceedings do SugarLoafPLoP. Eds. Rossana Andrade e Robert Hanmer, Porto de Galinhas- PE, 2003.
- *Padrão Arquitetural para Sistemas Computacionais de Controle Supervisório*. Jean Marcelo Simão; Marcos Antonio Quináia e Paulo Cezar Stadzisz. In Proceedings do SugarLoafPLoP. Eds. Rosana T. Vaccare Braga e Joseph W. Yoder, Itaipava - RJ, 2002.

Os autores foram convidados a resubmeter a primeira, das publicações acima citadas, em periódico nacional nível Qualis B.

6.3 Perspectivas Futuras

Os três parágrafos seguintes dizem respeito a panoramas gerais (padrões, arquitetura de *software* e fábrica de *software*) que, de certa forma, são concernentes a este trabalho. O último parágrafo desta seção, juntamente com seus tópicos, estão diretamente relacionados às perspectivas futuras deste trabalho.

PADRÕES

A tecnologia de padrão é um meio viável para a reutilização efetiva de *software* [APPLETON 2000a]. Essa constatação pode ser confirmada visto a aceitação dos padrões nos meios de desenvolvimento de *software* [COPLIEN 2003], [APPLETON 2000b]. Com isso, pode-se vislumbrar uma perspectiva promissora de trabalhos que enfoquem a utilização de padrões de *software*.

ARQUITETURA DE *SOFTWARE*

Os modelos podem ser usados para caracterizar e guiar o projeto de *software*. [SHAW e GARLAN 1996]. Assim, uma contribuição que possibilite a criação de modelos, na forma de padrões, é importante e permite antever um futuro onde a arquitetura dos sistemas computacionais será dirigida por modelos.

FÁBRICA DE *SOFTWARE*

As chamadas fábricas de *software* estão emergindo rapidamente nos meios de produção de *software,* possibilitando a construção de sistemas computacionais de forma padronizada [SEI-CMU 2005]. Pode-se prever a utilização de modelos para construção de *software* em larga escala.

Especificamente a este trabalho de doutorado, futuros trabalhos e pesquisas são admissíveis. A seguir são elencadas algumas possibilidades que podem ser empreendidas para dar continuidade ao trabalho aqui apresentado.

- Partes do metamodelo serão aplicadas nos próximos dois anos em um projeto de pesquisa submetido e aprovado pelo CNPq.
- Aplicar o metamodelo na descrição de padrões referentes a outros estudos de casos para verificar sua adequabilidade em outros domínios.
- Aprofundar o estudo sobre a caracterização de padrões arquiteturais, possibilitando um auxílio na elicitação de categorias de arquiteturas de *software*.
- Estudar notações que tornem os conceitos de opcionalidade, variabilidade e multiplicidade aplicáveis ao processo de desenvolvimento de *software*, porém sem depender de uma linguagem de descrição.
- Pesquisar sobre a aplicabilidade dos conceitos de opcionalidade, variabilidade e multiplicidade na especificação de famílias de produtos de *software*.
- Aplicar o metamodelo na descrição de um conjunto de sistemas de tempo real para verificar a extensão do mesmo.
- Desenvolver tipologias para outros diagramas da UML, visando maior aprofundamento neste assunto.
- Submeter o metamodelo para a apreciação do OMG (Object Management Group).
- Verificar a possibilidade de aplicar os conceitos de variabilidade, opcionalidade e multiplicidade ao nível de domínio, em modelos como o de negócios.

7 Bibliografia

7.1 Referências

[ALDRICH et al 2002] Jonathan Aldrich, Craig Chambers, and David Notkin. *Architectural Reasoning in ArchJava*. In: LNCS 2374 - Springer-Verlag. pp. 334-367. Mai. 2002.

[ALEXANDER 1977] Christopher Alexander, Sara Ishikawa, Murray Silverstein. *A Pattern Language: Towns Buildings, Constructions*. Oxford University Press, New York, 1977.

[ALEXANDER 1979] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[ANTIPATTERNS 1998] ANTIPATTERNS. *The Software Patterns Criteria*. In: http://www.antipatterns.com/whatisapattern/, Data de acesso: 08/04/2003.

[APPLETON 2000a] Brad Appleton. *Patterns in a Nutshell*. In: http://www.cmcrossroads.com/bradapp/docs/patterns-nutshell.html, Data de acesso: 07/06/2000.

[APPLETON 2000b] Brad Appleton. *Patterns and Software: Essential Concepts and Terminology*. In: http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html, Data de acesso: 07/06/2000.

[ARAÚJO e WEISS 2002] Ivan Araújo e Michael Weiss. *Linking Patterns and Non-Functional Requirements*. In: Proceedings of 9th Conference on Pattern Language of Programs. Monticello. pp. 52. 2002.

[ARSANJANI 2002] Ali Arsanjani. *Towards a Pattern Language for Web Services Architecture*. In: Proceedings of 9th Conference on Pattern Language of Programs. Monticello. pp. 21. 2002.

[ATP 2003] ATP (Advanced Technology Program). *ATP Focused Program: Component-Based Software*. In: http://www.atp.nist.gov/atp/focus/cbs.htm, Data de acesso: 08/06/2003.

[BACHMANN e BASS 2001] Felix Bachmann e Len Bass. *Managing variability in software architectures*. In: ACM SIGSOFT *Software* Engineering Notes, Proceedings of the 2001 symposium on *Software* reusability: putting *software* reuse in context, Vol: 26 Issue: 3. pp. 126-132. mai. 2001.

[BASS et al 1998] Len Bass, Paul Clements e Rick Kazman. *Software Architecture in Practice.* Addison-Wesley, Boston, 1998.

[BASS et al 2001] Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord e Kurt Wallnau. *Volume I: Market Assessment of Component-Based Software Engineering*. Nota Técnica CMU/SEI-2001-TN-007. 2001.

[BERARD 1992] Edward V. Berard. *Essays in Object-Oriented Software Engineering*. Volume 1. Prentice Hall, 1992.

[BERGNER et al 2003] K. Bergner, P. Bininda, A. Blessing, W. Daxwanger, T. Krenzke, A. Rausch, O. Schmid e M. Sihling. *Developing Reusable Software Components in CAM Environments*. In: http://www4.in.tum.de/~rausch/publications/2000/WKBA00.pdf, Data de acesso: 07/03/2003.

[BERNERS-LEE et al 2001] Tim Berners-Lee, James Hendler e Ora Lassila. *The Semantic Web*. In: Scientific American, maio. 2001.

[BITTNER e SPENCE 2003] Kurt Bittner e Ian Spence. *Use Case Modeling*. Addison-Wesley, 2003.

[BLILIE 2002] C. Blilie. *Patterns in Scientific Software: An Introduction*. In: IEEE Computing In Science & Engineering. Vol.4 No.3. pp. 48-53. 2002.

[BOOCH et al 2000] Grady Booch, Ivar Jacobson e James Rumbaugh. *UML Guia do Usuário*. Editora Campus. 2000.

[BROWN 1996] Alan W. Brown. *Component-Based Software Engineering - Selected Papers from the Software Engineering Institute*. Wiley-IEEE Computer Society Press, 1996.

[BROWN 1998] Willian J. Brown, Raphael C. Malveau, Hays W. McCormick e Thomas J. Mowbray. *Antipatterns: Refactoring Software, Architectures and Projects in Crisis*. John Wiley & Sons, 1998.

[BUSCHMANN et al 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad e Michael Stal. *Pattern - Oriented Software Architecture: A System of Patterns*. New York, John Wiley & Sons, 1996.

[CHEESMAN e DANIELS 2001] John Cheesman e John Daniels. *UML Components - A Simple Process for Specifying Component-Based Software*. Addison Wesley, 2001.

[CLEMENTS et al 2003] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord e Judith Stafford. *Documenting Software Architecture – Views and Beyond*, Addison Wesley, 2003.

[COPLIEN 1996] James O. Coplien. *Software Patterns*. SIGS Books & Multimedia, 1996.

[COPLIEN 2003] James O. Coplien. *A Pattern Definition*, In: http://hillside.net/patterns/definition.html. Data de acesso: 07/03/2003.

[COPLIEN e SCHMIDT 1995] James O. Coplien e Douglas C. Schmidt. *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[CORSARO et al 2002] A. Corsaro, D. C. Schmidt, R. Klefstad e C. O'Ryan. *Virtual Component A Design Pattern for Memory-Constrained Embedded Applications*. In: Proceedings of 9th Conference on Pattern Language of Programs. Monticello. pp. 13. 2002.

[CROSS e SCHMIDT 2002] J. K. Cross e D. C. Schmidt. *Quality Connector A Pattern Language for Provisioning and Managing Quality-Constrained Services in Distributed Real-time and Embedded Systems*. In: Proceedings of 9th Conference on Pattern Language of Programs. Monticello. pp. 20. 2002.

[CS-CMU 2003a] CS-CMU. *The Acme Architectural Description Language*. In: http://www-2.cs.cmu.edu/~acme/. Data de acesso: 24/02/2003.

[CS-CMU 2003b] CS-CMU. *The Aesop Project*. Inhttp://www-2.cs.cmu.edu/afs/cs/project/able/www/aesop/aesop_home.html. Data de acesso: 24/02/2003.

[CSE-USC 2003] CSE-Center for *Software* Engineering / USC-University of Southern California. *Software Architecture*. In: *http://sunset.usc.edu/research/software*_architecture/index.html. Data de acesso: 10/03/2003.

[D'SOUZA e WILLS 1998] Desmond F. D'Souza e Alan C. Wills. *Objects, Components and Frameworks With UML - The Catalysis Approach*, Addison Wesley, 1998.

[ERICKSSON 1997] Hans-Erik Ericksson e Magnus Penker. *UML Toolkit*, John Wiley & Sons, 1997.

[FIELDING 2000] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures* (Tese de doutorado). University of California, Irvine. 2000.

[FOREMAN 1996] J. Foreman. *Product Line Based Software Development- Significant Results, Future Challenges*. *Software* Technology Conference, Salt Lake City, UT, Abril, 1996.

[FOWLER 1996] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Massachussets: Addison-Wesley Longman, 1996.

[GARG et al 2003] Akash Garg, Matt Critchlow, Ping Chen, Christopher Van der Westhuizen e André van der Hock. *An Environment for Managing Evolving Product Line Architectures*. In: Proceedings of the International Conference on *Software* Maintenance (ICSM). IEEE Computer Society. pp. 358-367. 2003.

[GAMMA et al 1995] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, MA, 1995.

[GARLAN e SHAW 1994] D. Garlan e M. Shaw. *An Introduction to Software Architecture*. CMU-CS-94-166. 1994.

[GARLAN et al 2004] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl e Peter Steenkiste. *Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure*. In: IEEE Computer Society, Vol: 37, Issue: 10. pp. 46-54. Out. 2004.

[GIMENES et al 2000] Itana M. S. Gimenes, Leonor Barroca, Elisa H. M. Huzita e Adriana Carniello. *O Processo de Desenvolvimento Baseado em Componentes Através de Exemplos*. Escola Regional de Informática (ERI 2000), Foz do Iguaçu. 2000.

[GUO et al 1999] George Yanbing Guo, Joanne M. Atlee e Rick Kazman. *A Software Architecture Reconstruction Method*. In: Proceedings of the TC2 First Working IFIP Conference on *Software* Architecture. pp. 15-34. 1999.

[GURP et al 2001] Jilles Van Gurp, Jan Bosch e Mikael Svahnberg. *On the Notion of Variability in Software Product Lines*. In: Proceedings of the Working IEEE/IFIP Conference on *Software* Architecture. pp. 45-55. 2001.

[HANSEN 2003] B. Hansen. *UML 2.0 Sequence Diagramming*. Sun ONE Studio Developer Resources. In: http://forte.sun.com/ffj/articles/describe3.html. Data de acesso: 24/02/2003.

[HENNINGER 2002] S. Henninger. *Using the Semantic Web to Construct an Ontology-Based Repository for Software Patterns*, In: Workshop on the State of the Art in Automated *Software* Engineering, Irvine. pp. 18-22. 2002.

[JACOBSON 1987] Ivar Jacobson. *Object-oriented development in an industrial environment*. In: Proceedings of Conference on Object-oriented programming systems, languages and applications. Orlando, pp. 183-191. 1987.

[JACOBSON et al 1997] I. Jacobson, M. Griss e P. Jonsson. *Software Reuse: Architecture, process and Organization for Business Success*. Addison Wesley Longman. 1997.

[JACOBSON et al 1999] I. Jacobson, Grady Booch e James Rumbaugh. *The Unified Software Development Process*. Addison Wesley. 1999.

[KELLER e WENDT 2003] Frank Keller e Siegfried Wendt. 2003. *FMC: An Approach Towards Architecture-Centric System Development*. In: Proceedings of 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, abr. pp. 173-182. 2003.

[KLEIN e KAZMAN 1999] M. Klein e R. Kazman. *Attribute-Based Architectural Styles*. CMU/SEI-99-TR-022. 1999.

[KOBRYN 2003] C. Kobryn. *What to Expect From UML 2.0*. http://www.sdtimes.com/opinions/guestview_048.htm. Data de acesso: 25/02/2003.

[KRUCHTEN 2000] P. Kruchten. *The Rational Unified Process - An Introduction*, 2ª ed. Addison-Wesley-Longman (2000).

[LARMAN 1999] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice-Hall. 1999.

[MAIER et al 2001]Mark W. Maier, David Emery e Rich Hilliard. *Software* Architecture: Introducing IEEE Standard 1471. In: IEEE Computer, Vol.: 34, Issue: 4. pp. 107-109. abr. 2001.

[MEDVIDOVIC e TAYLOR 2000] Nenad Medvidovic e Richard N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages*. In: IEEE Transactions on *Software* Engineering. Vol. 26, No. 1. pp. 70-93. jan. 2000.

[MENDES 2002] Antonio Mendes. *Arquitetura de Software: Desenvolvimento Baseado na Arquitetura*. Editora Campus. 2002.

[METAMODEL 2004a] Metamodel. *Model vs. metamodel vs. meta-metamodel*. In: http://www.metamodel.com/article.php?story=2002101021353113. Data de acesso: 04/08/2004.

[MONROE et al 1997] R.T. Monroe, A. Kompanek, R. Melton e D. Garlan. *Architectural Styles, Design Patterns, and Objects.* In: IEEE *Software*, Vol.14, No.1, pp. 43-52. jan, 1997.

[NEIGHBORS 1980] James M. Neighbors. *Software Construction Using Components*. Department of Information and Computer Science - University of California, Irvine (tese doutorado), 1980.

[NOBLE e BIDDLE 2002] J. Noble e R. Biddle. *Patterns as Signs*. In: LNCS (*Lecture Notes in Computer Science*). Springer-Verlag, Vol.2374, pp. 368-391. mai, 2002.

[OHORI 2003] A. Ohori. *Foundations for Relaiable Componetware*. In: http://www.jaist.ac.jp/~ohori/research/oki-comp.html. Data de acesso: 07/03/2003.

[OMG 1998] OMG. *Unified Modeling Language Specification V1.2*. In: http://www.jeckle.de/uml_spec.htm#umlv1.2. Data de acesso: 19/11/2004.

[OMG 2003a] OMG. *UML 2.0 Superstructure Specification*. In: http://www.omg.org/docs/ptc/03-08-02.pdf. Data de acesso: 19/08/2003.

[PINZGER e GALL 2002] Martin Pinzger e Harald Gall. *Pattern-supported architecture recovery.* In: Proceedings of 10th International Workshop on Program Comprehension, pp. 53-61. 2002.

[PRESSMAN 2004] R. S. Pressman. *Software Engineering: A Practitioner's Approach, 6ª ed.* McGraw-Hill. 2004.

[PRIBERAM 2004] Priberam. Dicionário da Língua Portuguesa On-Line. Editora Universal. In: http://www.priberam.pt/dlpo/dlpo.aspx. Data de acesso: 04/08/2004.

[PRIETO-DIAZ 1990] Rubén Prieto-Díaz. *Domain Analysis: An Introduction*. In: ACM SIGSOFT Software Engineering Notes Vol.15, Issue 2. pp. 47-54. abr, 1990.

[QUINÁIA e STADZISZ 2003] M. A. Quináia e P. C. Stadzisz. *A Use Case Extension for Architectural Patterns*. In: CSITeA'03, Rio de Janeiro, pp. 43-48. 2003.

[RATIONAL 2003a] Rational. *Architectural Patterns*. In: http://www.rationalrose.com/models/architectural_patterns.htm. Data de acesso: 17/03/2003.

[RATIONAL 2003b] Rational. *UML Resource Center*. In: http://www.rational.com/uml/index.jsp?SMSESSION=NO. Data de acesso: 18/06/2003.

[RÉ et al 2001] R. Ré, R. T. V. Braga e P. C. Masiero. *A Pattern Language for Online Auctions Management*. In Proceedings of 8th Conference on Pattern Language of Programs. Monticello. pp. 1-18. 2001.

[RISE 2003] RISE - *Research in Software Engineering* - Dept. of *Software* Engineering and Computer Science, University of Karlskrona/Ronneby, Sweden. In: http://www.ipd.bth.se/rise/research_areas.htm. Data de acesso: 24/02/2003.

[ROSETI e WERNER 1999] M. Z. Roseti e C. M. L. Werner. *Aquisição de Conhecimento no Contexto de Análise de Domínio*. In: SBES'99, Florianópolis, pp. 16. 1999.

[ROSS e SCHOMAN 1977] D.Ross e K. Schoman. *Structured Analysis for Requirements Definition*. IEEE Transactions on *Software* Engineering 3(1), pp. 6-15. jan. 1977.

[RUMBAUGH et al 1998] J. Rumbaugh, I. Jacobson e G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley. 1998.

[SARTIPI e KONTOGIANNIS 2003] K. Sartipi e K. Kontogiannis. *Pattern-based Software Architecture Recovery*. In: Proceedings of the Second ASERC Workshop on *Software* Architecture. Banff Center, Alberta, Canada. pp. 7. fev. 2003.

[SCHMIDT et al 2000] D. Schmidt, M. Stal, H. Rohnert e F. Buschmann. *Pattern-Oriented Software Architecture*, Volume 2, Patterns for Concurrent and Networked Objects, John Wiley & Sons, 2000.

[SEI-CMU 2003a] SEI-CMU. *Software Architecture*. In: http://www.sei.cmu.edu/ata/ata_init.html. Data de acesso: 13/03/2003.

[SEI-CMU 2003b] SEI-CMU. *Domain Engineering and Domain Analysis*. In: http://www.sei.cmu.edu/str/descriptions/deda.html. Data de acesso: 25/03/2003.

[SEI-CMU 2003c] SEI-CMU. *COTS-Based Systems (CBS) Initiative*. In: http://www.sei.cmu.edu/cbs/index.html. Data de acesso: 25/07/2003.

[SEI-CMU 2005] SEI-CMU. *Software Product Lines*. In: http://www.sei.cmu.edu/productlines/. Data de acesso: 05/04/2005.

[SHAW e GARLAN 1994] M. Shaw e D. Garlan. *Characteristics of Higher-level Languages for Software Architecture*. CMU-CS-94-210. 1994.

[SHAW e GARLAN 1996] Mary Shaw e David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall Publishing, 1996.

[SHAW e GARLAN 2003] M. Shaw e D. Garlan. *Tutorial Slides on Software Architecture* In: http://www-2.cs.cmu.edu/afs/cs/project/tinker-arch/www/html/TutorialSlides/SoftArch/quickindex.html. Data de acesso: 13/03/2003.

[SHULL et al 1996] F. Shull, W. Melo e V. Basili. *An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems*. University of Maryland Technical Report CS-TR-3597. 1996.

[SILVA e PRICE 2002] R. P. Silva, e R. T. Price. *Component Interface Pattern*. In: Proceedings of 9th Conference on Pattern Language of Programs. Monticello. pp. 17. 2002.

[STADZISZ 1997] Paulo. C. Stadzisz. *Contribuition à une Méthodologie de Conception Intégrée des Familles de Produits pour l'Assemblage*. L'Universite de Franche-Comte. (Tese doutorado). 1997.

[STADZISZ 2002] Paulo. C. Stadzisz. *Projeto de Software usando a UML*. (apostila) Centro Federal de Educação Tecnológica do Paraná. Curitiba. 2002.

[STEELTRACE 2002] Steeltrace. *How Requirements enable Reuse*. In: http://www.steeltrace.com/download/Whitepaper%20How%20Requirements%20enable%20Reuse.pdf. Data de acesso: 25/03/2003.

[SULLIVAN 2003] Tom Sullivan. *UML 2.0 spec to come this fall*. In: http://www.infoworld.com/article/02/08/21/020821hnuml2_1.html. Data de acessso: 24/02/2003.

[SUZUKI 2003] J. Suzuki. *Uml Exchange Format & Pattern Markup Language*. In: http://www.yy.cs.keio.ac.jp/~suzuki/project/uxf/. Data de acessso: 24/02/2003.

[SUZUKI e YAMAMOTO 1999] J. Suzuki e Y. Yamamoto. *Extending UML for Modeling Reflective Software Components*. In: Proceedings of Second International Conference on Unified Modeling Language. Springer Vol. 1723, pp. 220-235. 1999.

[TANNENBAUM 2001] Adrienne Tannenbaum. *Metadata Solutions: Using Metamodels, Repositories, XML and Enterprise Portals to Generate Information on Demand*, New York, Addison Wesley, 2001.

[U2 PARTNERS 2003] U2 Partners. *2nd revised submission to OMG RFP ad/00-09-02: Unified Modeling Language: Superstructure version 2.0*. 2003.

[VERKAMO 2003] A. Verkamo. *MAISA: Metrics for Analysis and Improvement of Software Architectures*. In: http://www.cs.helsinki.fi/group/maisa/. Data de acesso: 16/03/2003.

[WEBSTER'S 2004] Webster's New Millennium™ Dictionary of English, © 2003 Lexico Publishing Group, LLC. In: http://dictionary.reference.com/search?q=metamodel. Data de acessso: 08/04/2004.

[WELCH et al 2002] L. R. Welch, T. Marinucci, M. W. Masters e P. V. Werme. *Dynamic Resource Management Architecture Patterns*. In: Proceedings of 9th Conference on Pattern Language of Programs. Monticello. pp. 13. 2002.

[ZENGER 2002] M. Zenger. *Type-Safe Prototype-Based Component Evolution*. In: LNCS 2374 - Springer-Verlag. pp. 470-497. mai. 2002.

7.2 Complementar

[ADOLPH e BRAMBLE 2003] Steve Adolph and Paul Bramble. *Patterns for Effective Use Cases*. Addison-Wesley. 2003.

[ARANGO E PRIETO-DÍAZ 1991] G. Arango e R. Prieto-Díaz. *Domain Analysis Concepts and Research Directions*. In: Domain Analysis and Software Systems Modeling, eds., IEEE Computer Society Press, 1991.

[BIGGERSTAFF e RICHTER 1989] T. Biggerstaff e C. Richter. *Reusability framework, assessment, and directions*. In: Software reusability: vol. 1, concepts and models, ACM Press, pp. 1-17.1989.

[BOOCH 1998] Grady Booch. *Quality Software and the Unified Modeling Language*. In: http://www.rational.com/support/techpapers/softuml.html. 1998. Data de acesso 06/05/2005.

[JACOBSON et al 1992] I. Jacobson, M. Christerson, P. Jonsson e G. Övergaard. *Object-Oriented Software Engineering: A Use-Case Driven Approach*. Addison Wesley. 1992.

[METAMODEL 2004b] Metamodel. *What is metamodeling, and what is it good for?* . In: http://www.metamodel.com/staticpages/index.php?page=20021010231056977. Data de acesso: 04/08/2004.

[OMG 2003b] OMG. *Introduction to OMG's Unified Modeling Language (UML)*. In: http://www.omg.org/gettingstarted/what_is_uml.htm. Data de acesso: 18/06/2003.

[RATIONAL 2003c] Rational. *Rational Unified Process*. In: http://www.rational.com/products/rup/index.jsp. Data de acesso: 27/06/2003.

[SARTIPI 2003] Kamran Sartipi. *Software Architecture Recovery based on Pattern Matching*. In: Proceedings of the International Conference on *Software* Maintenance. Amsterdam. pp. 293-296. set. 2003.

Anexo I – Análises de Equivalência Funcional dos Casos de Uso

Este anexo apresenta as tabelas utilizadas nas análises de equivalência para os casos de uso do subsistema Movimentação do Estudo de Caso. Cada tabela é referente a um caso de uso. Cada coluna, de cada tabela, contém informações de campos que são editadas / mostradas quando o caso de uso for executado. Foram utilizadas cores para visualizar os campos que são inter-relacionados. Dada uma tabela, os campos de mesma cor são campos equivalentes entre os sistemas analisados.

Tabela I.1 – Análise do caso de uso consultar histórico de empréstimos de usuário

SISTEMA 1	SISTEMA 2	SISTEMA 3	SISTEMA 4
Nome do Usuário	Código Usuário	Número Usuário	Código Usuário
Número do Usuário	Nome Usuário	Data operação	Data empréstimo
Série	Código Obra	Horário operação	Data prevista devolução
Grau	Título	Tipo de operação	Data devolução
Turma	Empréstimo	Número tombo	Tipo
Obra	Devolução	Data prevista devolução	Registro obra
Data Empréstimo		Multa devida	Título
Data prevista Devolução		Número título para catálogo	
Devolução		Código do tipo de usuário e objeto emprestado	
Aviso 1		Nome bibliotecário responsável pela operação	
Aviso 2			
Aviso 3			

Tabela I.2 – Análise do caso de uso devolver exemplar

SISTEMA 1	SISTEMA 2	SISTEMA 3	SISTEMA 4
Obra	Número tombo	Número Usuário	Número usuário
Nome Aluno	Status	Data operação	Nome usuário
Série	Resultado	Horário operação	Código obra
Número aluno	Título	Tipo de operação	Título obra
Turma	Valor multa	Número tombo	Data empréstimo
Número empregado	Código usuário	Data prevista devolução	Data prevista devolução
Nome empregado	Nome usuário	Multa devida	Data devolução
Data empréstimo		Número título para catálogo	Multa paga
Data prevista devolução		Código do tipo de usuário e objeto emprestado	Notas de operação
Data devolução		Nome bibliotecário responsável pela operação	Mensagem de rodapé de recibo

Tabela I.3 – Análise do caso de uso emitir aviso de devolução em atraso

SISTEMA 1	SISTEMA 4
Número do empréstimo	Tipo Obra
Código Obra	Código Obra
Número aluno	Registro
Série	Classificação
Categoria	Cutter
Turma	Unidade
Nome aluno	Título
Data empréstimo	Situação do Usuário
Data prevista devolução	Código Usuário
Data devolução	Nome Usuário
Aviso 1	Série
Aviso 2	Apelido
Aviso 3	Curso

Tabela I.4 – Análise do caso de uso renovar empréstimo

SISTEMA 2	SISTEMA 3	SISTEMA 4
Número usuário	Data da Operação	Código do usuário
Nome usuário	Hora da Operação	Nome do usuário
Ação	Tipo de Operação	Tipo obra
Número tombo	No. Usuário	Código obra
Título	No. tombo	Registro obra
Autor	Data prevista para devolução	Título obra
Data prevista devolução	Multa devida	Data empréstimo
	No. do título na base	Prazo
	Código do tipo de usuário e objeto emprestado	Data prevista devolução
	Nome do bibliotecário responsável pela operação	Data renovação
		Notas justificativas
		Mensagens de rodapé de recibo

Tabela I.5 – Análise do caso de uso ler código de barras

SISTEMA 1	SISTEMA 2	SISTEMA 3	SISTEMA 4
Número usuário	Cód Pessoa	No. Usuário	Cód usuário
Obra	Cód obra	No. Tombo	Cód obra

Tabela I.6 – Análise do caso de uso reservar obra

SISTEMA 1	SISTEMA 2	SISTEMA 3	SISTEMA 4
No. Aluno 1	Código da pessoa	Data da reserva	Situação do usuário
Data reverva 1	Nome da pessoa	Hora da reserva	Código do usuário
No. Aluno 2	Código reserva	Tipo de Operação	Categoria do usuário
Data reverva 2	Código obra	No. Usuário	Nome do usuário
No. Aluno 3	Título obra	No. tombo	Curso usuário
Data reverva 3	Autor obra	Nome do bibliotecário responsável pela operação	Série usuário
Código obra	Código exemplar		Apelido usuário
Título	Estado exemplar		Tipo de publicação
Código aluno	Volume		Código da publicação
Data Empréstimo	Data reserva		Reg. da publicação
Data prevista devolução	Data entrada		Classificação da publicação
Data devolução	Data limite		Cutter
Data Aviso 1	Posição		Comp
Data Aviso 2			Título da publicação
Data Aviso 3			

Tabela I.7 – Análise do caso de uso cancelar reserva de obra

SISTEMA 1	SISTEMA 2	SISTEMA 3	SISTEMA 4
No. Aluno 1	Código da pessoa	Data da reserva	Situação do usuário
Data reverva 1	Nome da pessoa	Ação	Código do usuário
No. Aluno 2	Código reserva		Categoria do usuário
Data reverva 2	Código obra		Nome do usuário
No. Aluno 3	Título obra		Curso usuário
Data reverva 3	Autor obra		Série usuário
Código obra	Data entrada		Apelido usuário
Título	Data limite		Tipo de publicação
Código aluno	Posição		Código da publicação
Data Empréstimo	Data reserva		Registro da publicação
Data prevista devolução			Classificação da publicação
Data devolução			Cutter
Data Aviso 1			Comp
Data Aviso 2			Título da publicação
Data Aviso 3			

Tabela I.8 – Análise do caso de uso consultar reserva

SISTEMA 1	SISTEMA 2	SISTEMA 3	SISTEMA 4
No. Aluno 1	Código da pessoa	Data da reserva	Situação do usuário
Data reverva 1	Nome da pessoa	Hora da reserva	Código do usuário
No. Aluno 2	Código reserva	Tipo de Operação	Categoria do usuário
Data reverva 2	Código obra	No. Usuário	Nome do usuário
No. Aluno 3	Título obra	No. tombo	Curso usuário
Data reverva 3	Autor obra	Nome do bibliotecário responsável pela operação	Série usuário
Código obra	Data entrada		Apelido usuário
Título	Data limite		Tipo de publicação
Código aluno	Posição		Código da publicação
Data Empréstimo	Data reserva		Registro da publicação
Data prevista devolução			Classificação da publicação
Data devolução			Cutter
Data Aviso 1			Comp
Data Aviso 2			Título da publicação
Data Aviso 3			

Tabela I.9 – Análise do caso de uso emitir multa

SISTEMA 1	SISTEMA 2	SISTEMA 3	SISTEMA 4
Recibo	Código da pessoa	No. tombo	Código do usuário
Código aluno	Nome da pessoa	Data Operação	Reg. Fascículo
Código obra	Código multa	Horário operação	Data Empréstimo
Empréstimo	Código obra	Tipo operação	Data Prevista Devolução
Dias atraso	Título obra	No. usuário	Data Devolução
Valor total	Data da multa	Data prevista devolução	Data Renovação
Data pagto	Valor da multa	Multa devida	Código da publicação
Observação		Código do tipo de usuário e objeto emprestado	Valor multa

Tabela I.10 – Análise do caso de uso cancelar multa

SISTEMA 1	SISTEMA 2	SISTEMA 4
Código obra	Código da pessoa	Código do usuário
Título	Nome da pessoa	Reg. Fascículo
Código aluno	Código multa	Data Empréstimo
Série	Código obra	Data Prevista Devolução
Turma	Título obra	Data Devolução
No. Empregado	Data multa	Data Renovação
Data Empréstimo	Valor multa	Código da publicação
Data prevista devolução		Valor multa
Data devolução		
Valor multa		
Nome Aluno		
No. Recibo		
Dias de atraso		
Data de pagto		
Observações		

A Tabela I.10 contém informações de campos que são mostrados e contém também informações de campos que são editados. Os campos da Tabela I.11 foram elicitados a partir da Tabela I.10 pelo projetista do padrão arquitetural. Os campos são bastante parecidos, embora o sistema 1 mostre mais informações que os demais sistemas. Porém, essas informações são apenas para visualização e não para edição. Portanto, para fins de padronização, decidiu-se por manter um conjunto de informações comuns aos sistemas (informações que são mostradas / editadas), fazendo-se com que esse caso de uso não fosse opcional.

Tabela I.11 – Campos definidos para o caso de uso cancelar multa

Padrão	
Campo	**Ação**
Código usuário	editado
Nome usuário	mostrado
Código obra	mostrado
Título	mostrado
Código multa	mostrado/selecionado
Valor multa	mostrado
Data da multa	mostrado
Data do cancelamento	sistema

Anexo II – Diagramas de Seqüência Genéricos

Este anexo apresenta os diagramas de seqüência genéricos para os casos de uso do subsistema Movimentação do Estudo de Caso

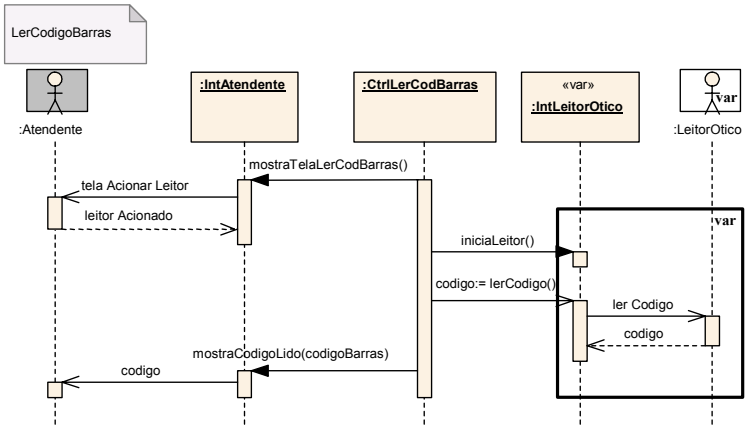


Figura II.1 – Diagrama de seqüência genérico para o caso de uso Ler Código de Barras

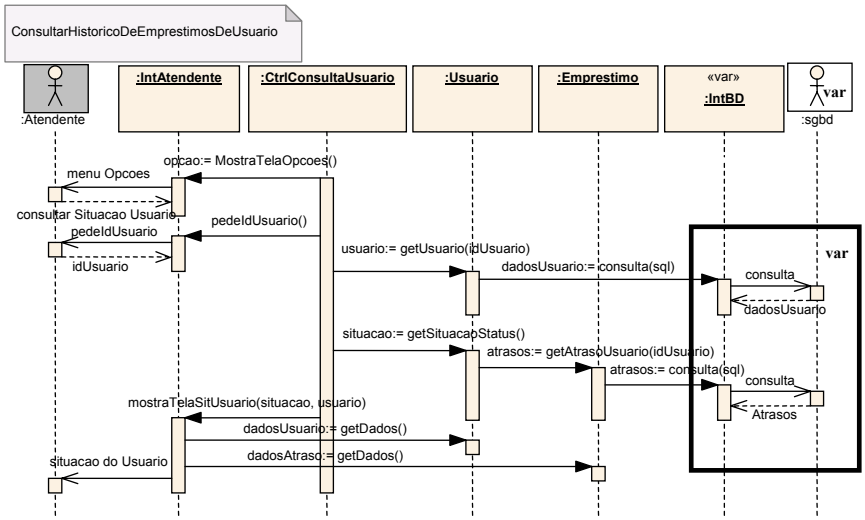


Figura II.2 – Diagrama de seqü. genérico para o caso de uso Consultar Histórico De Empréstimos De Usuário

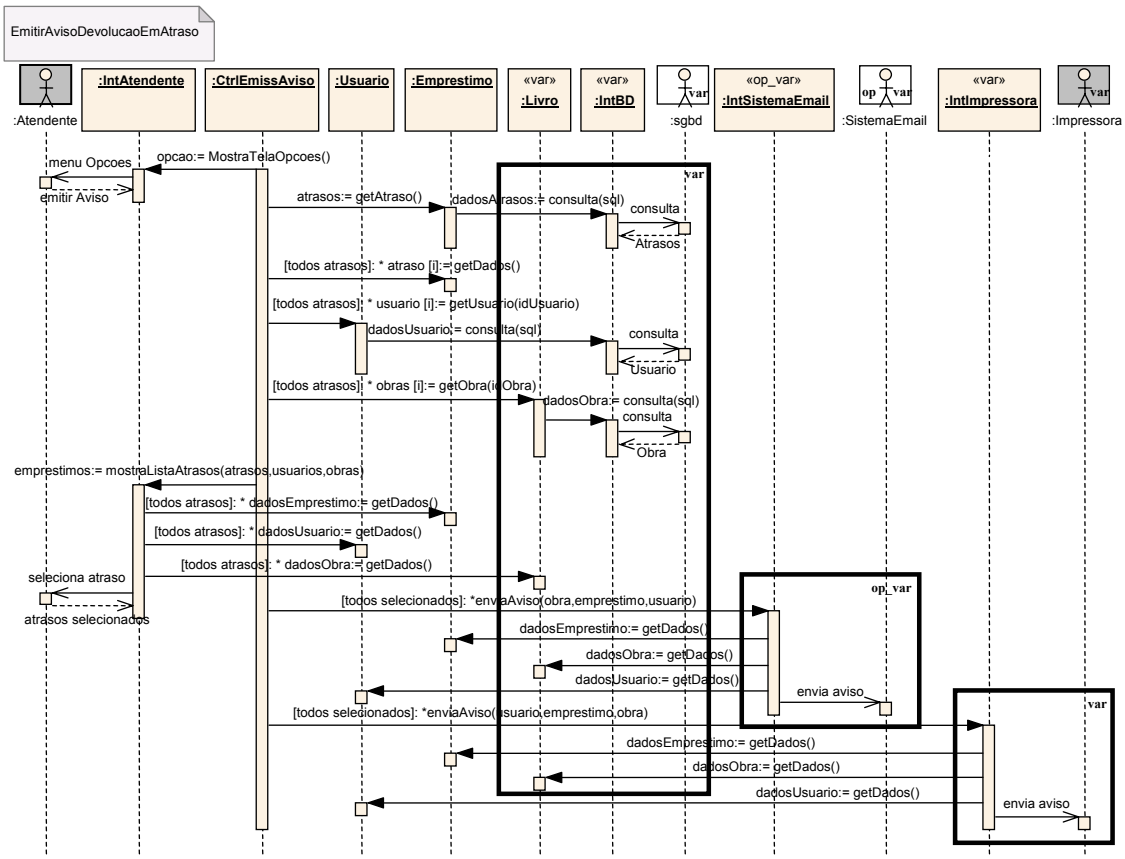


Figura II.3 – Diagrama de seqüência genérico para o caso de uso Emitir Aviso De Devolução Em Atraso

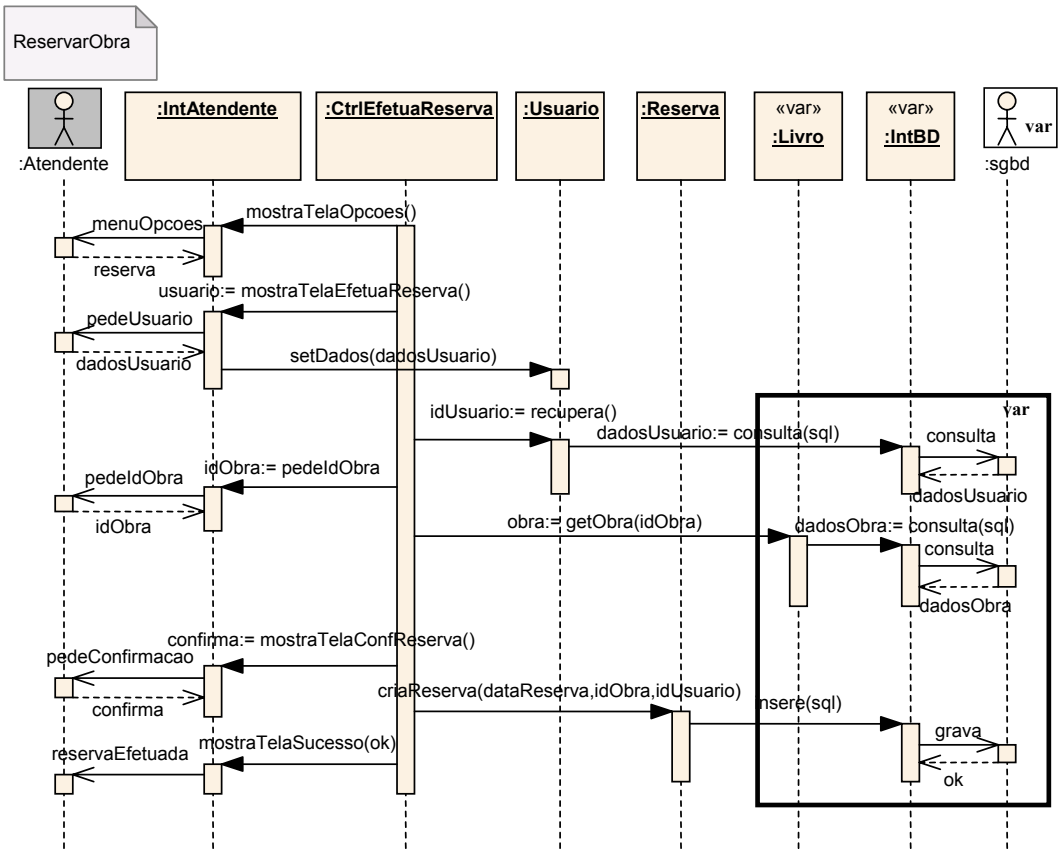


Figura II.4 – Diagrama de seqüência genérico para o caso de uso Reservar Obra

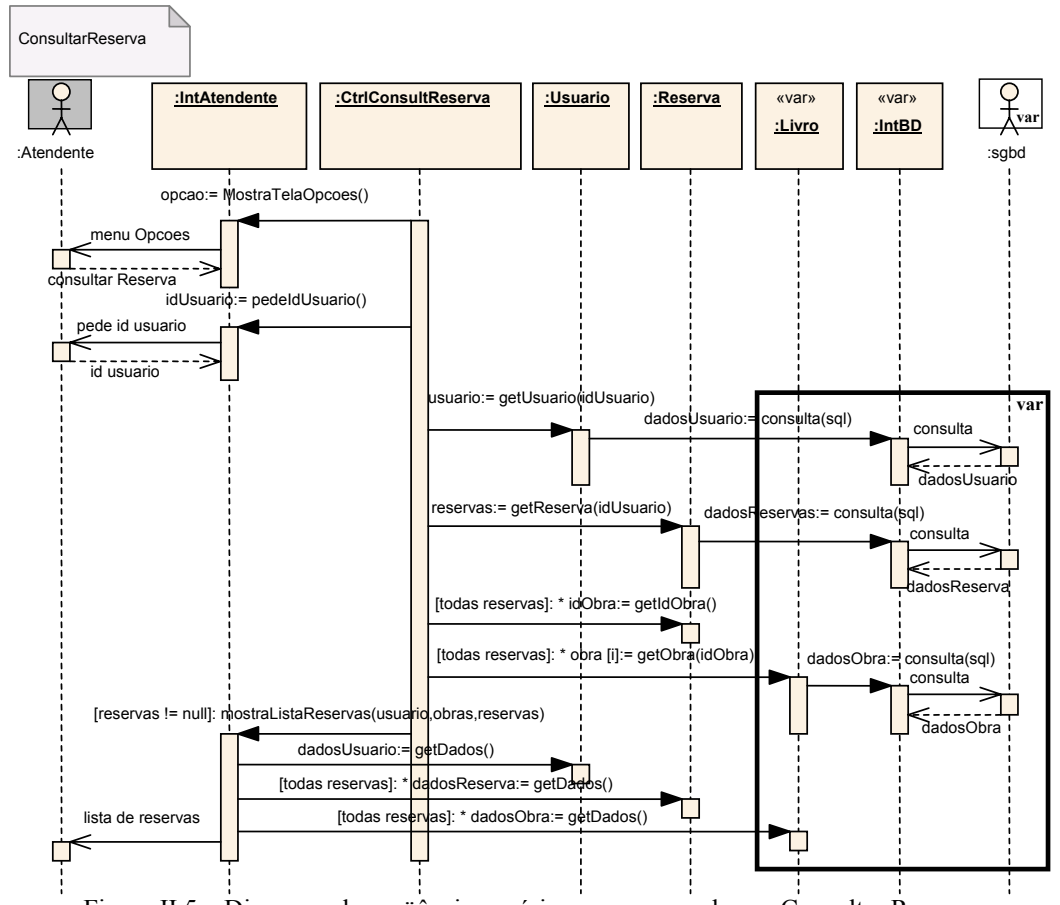


Figura II.5 – Diagrama de seqüência genérico para o caso de uso Consultar Reserva

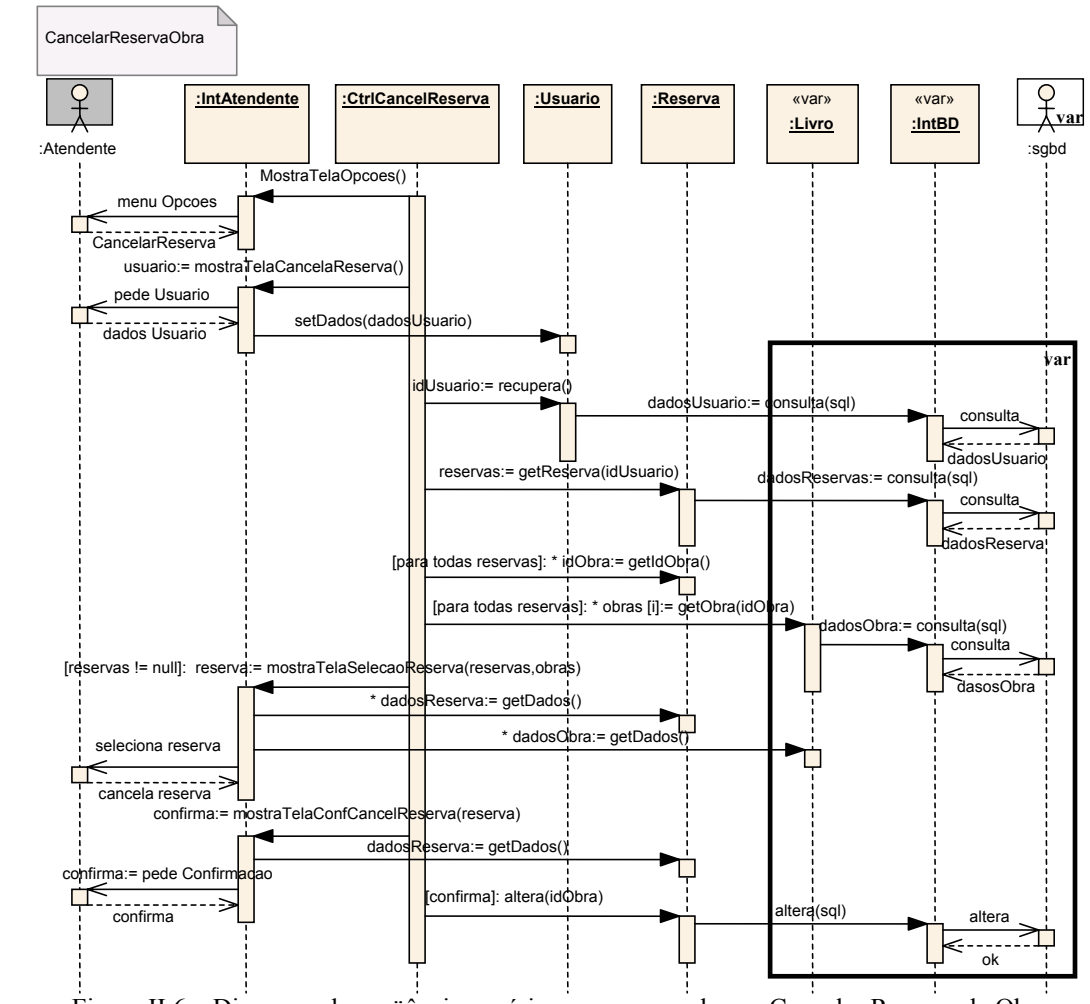


Figura II.6 – Diagrama de seqüência genérico para o caso de uso Cancelar Reserva de Obra

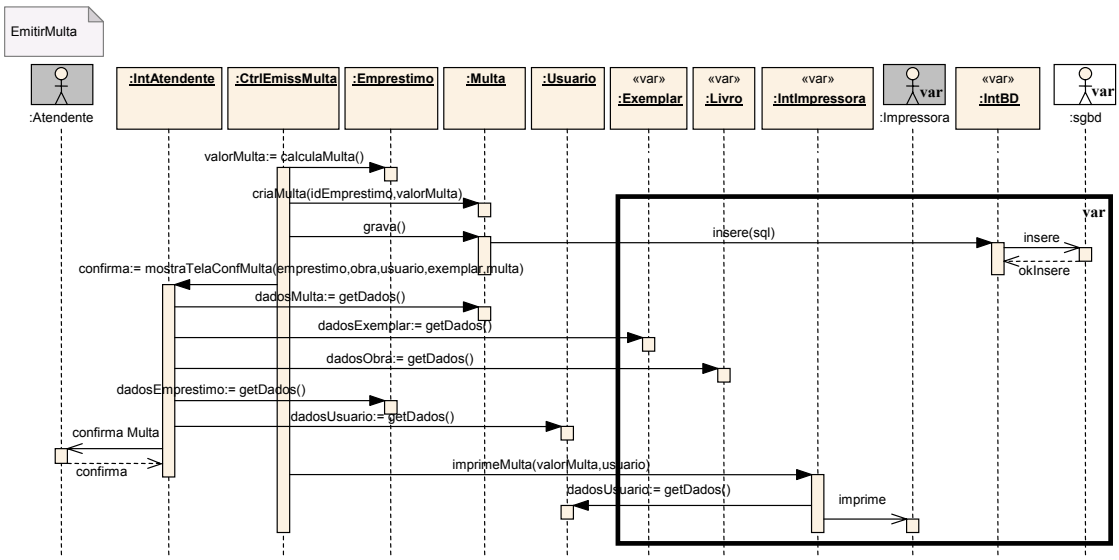


Figura II.7 – Diagrama de seqüência genérico para o caso de uso Emitir Multa

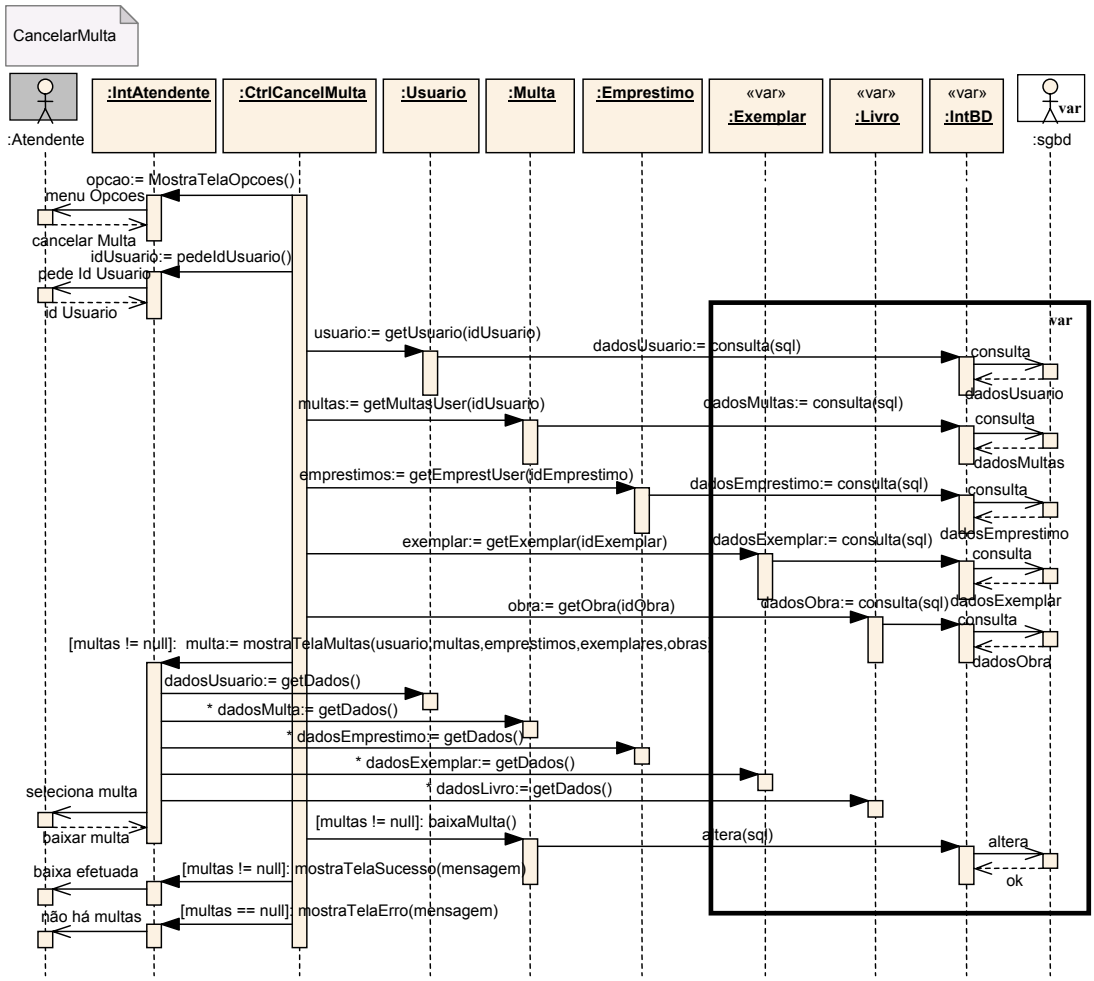


Figura II.8 – Diagrama de seqüência genérico para o caso de uso Cancelar Multa

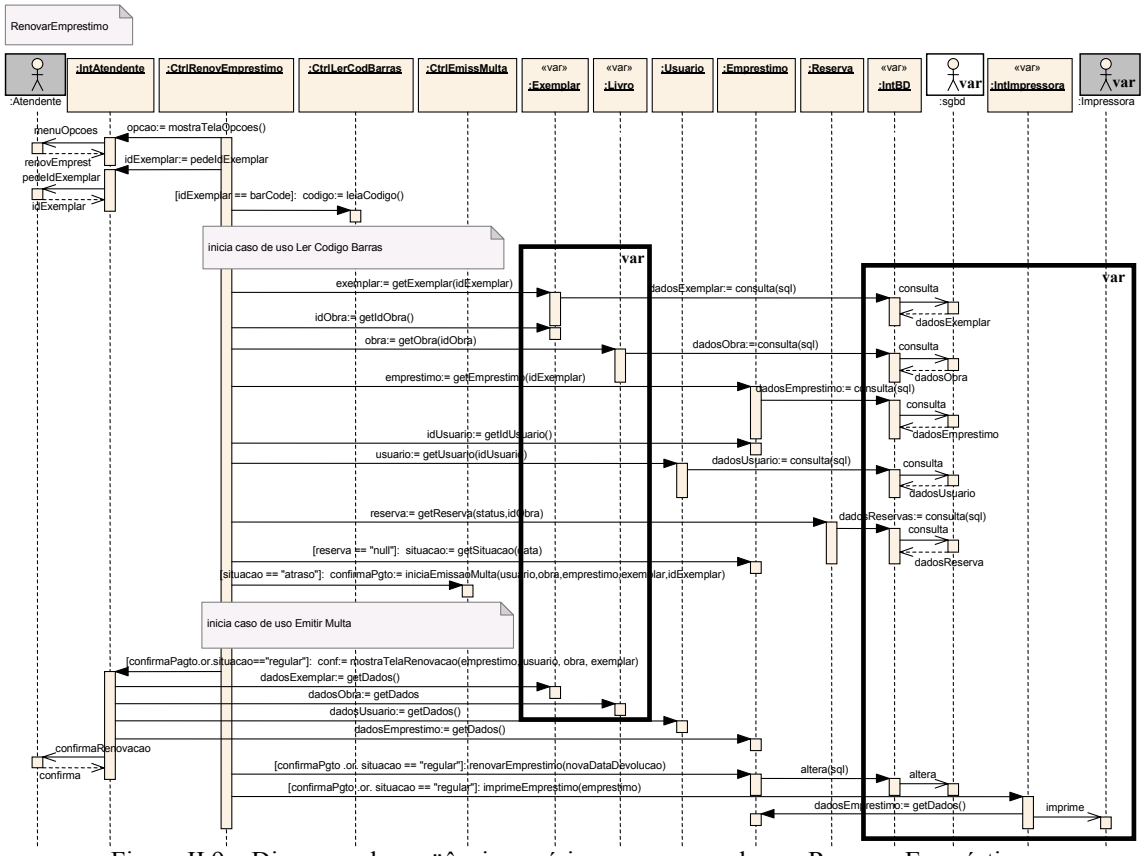


Figura II.9 – Diagrama de seqüência genérico para o caso de uso Renovar Empréstimo

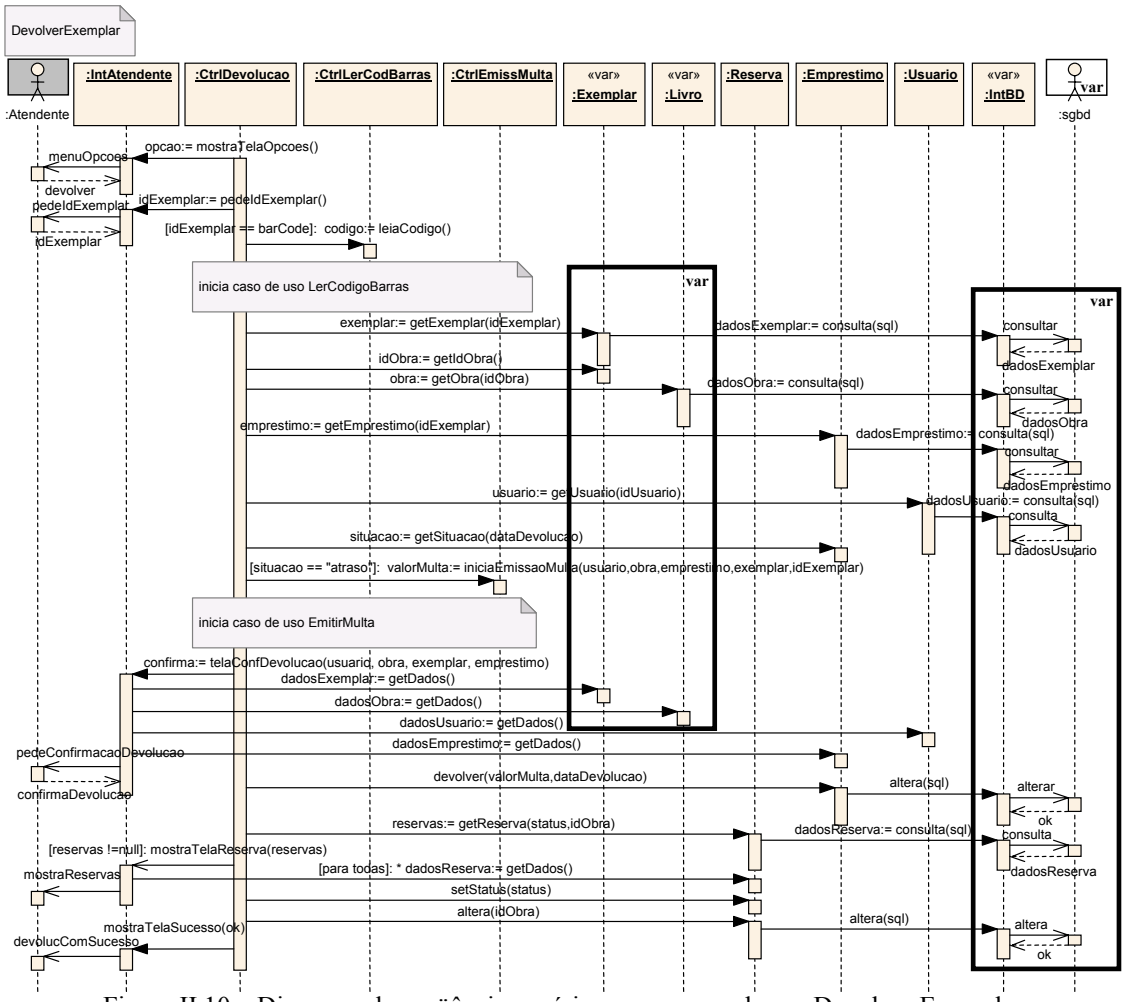


Figura II.10 – Diagrama de seqüência genérico para o caso de uso Devolver Exemplar

Anexo III – Diagramas de Classes

Este anexo apresenta os diagramas de classes para os casos de uso do subsistema Movimentação do Estudo de Caso

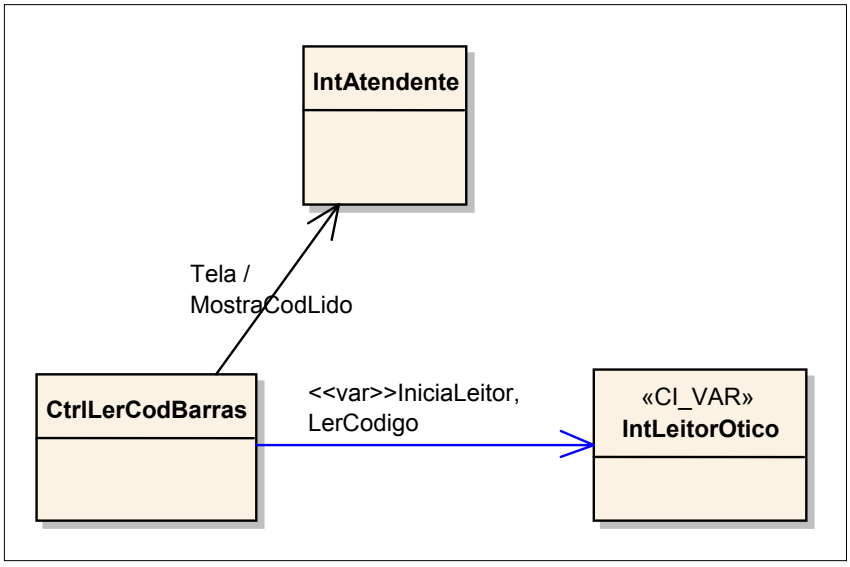


Figura III.1 – Diagrama de classes genérico para o caso de uso Ler Código de Barras

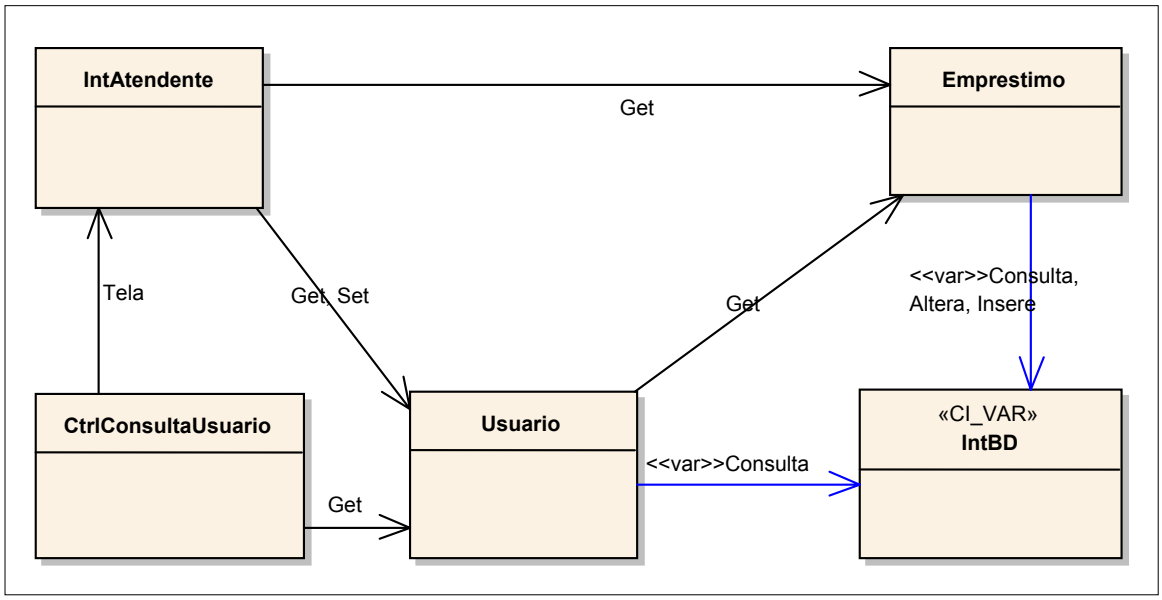


Figura III.2 – Diagr. de classes genérico para o caso de uso Consultar Histórico De Empréstimos De Usuário

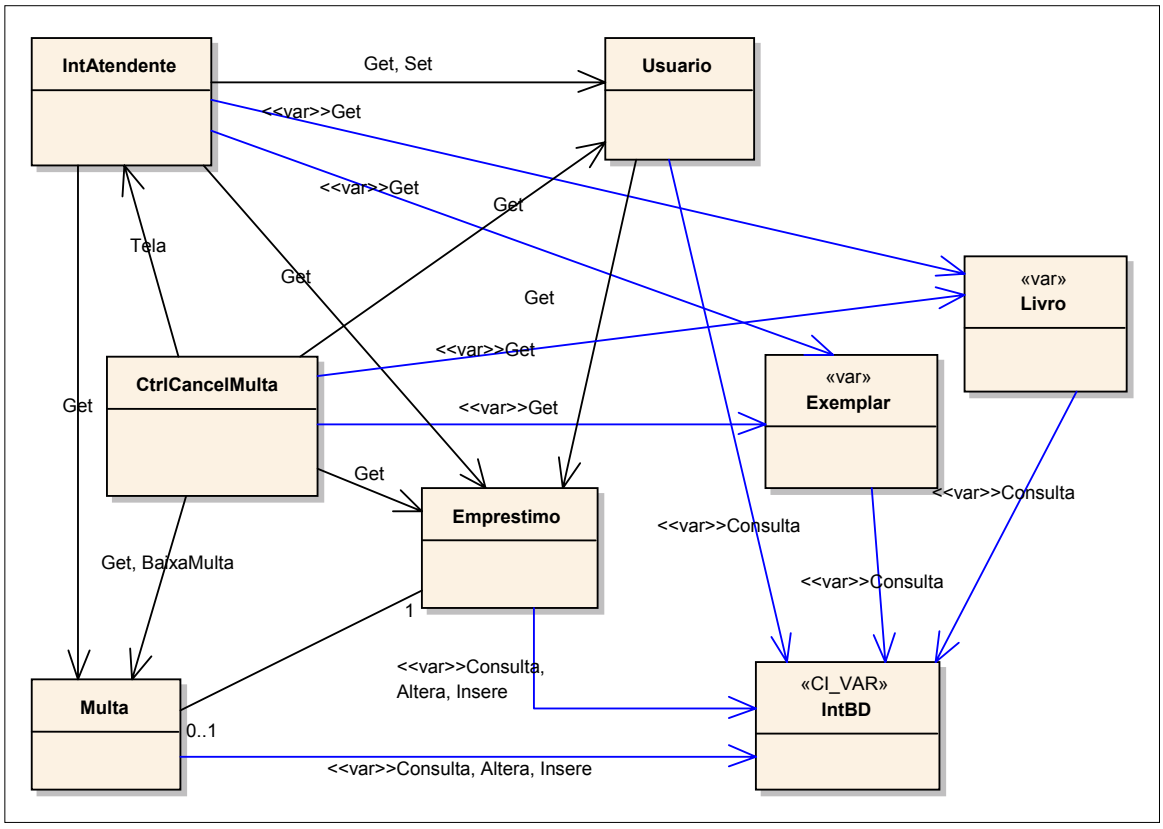


Figura III.3 – Diagrama de classes genérico para o caso de uso Cancelar Multa

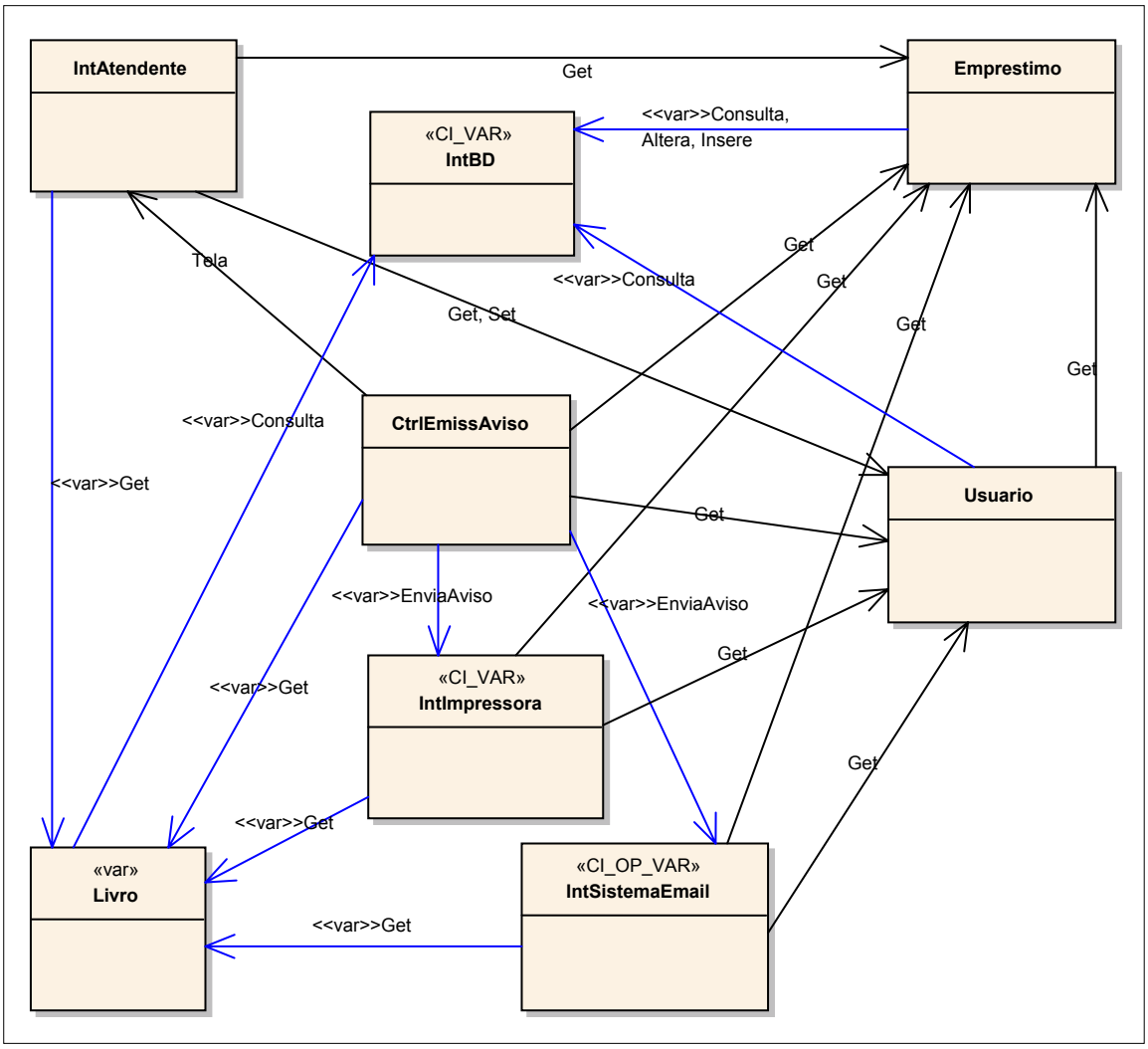


Figura III.4 – Diagrama de classes genérico para o caso de uso Emitir Aviso De Devolução Em Atraso

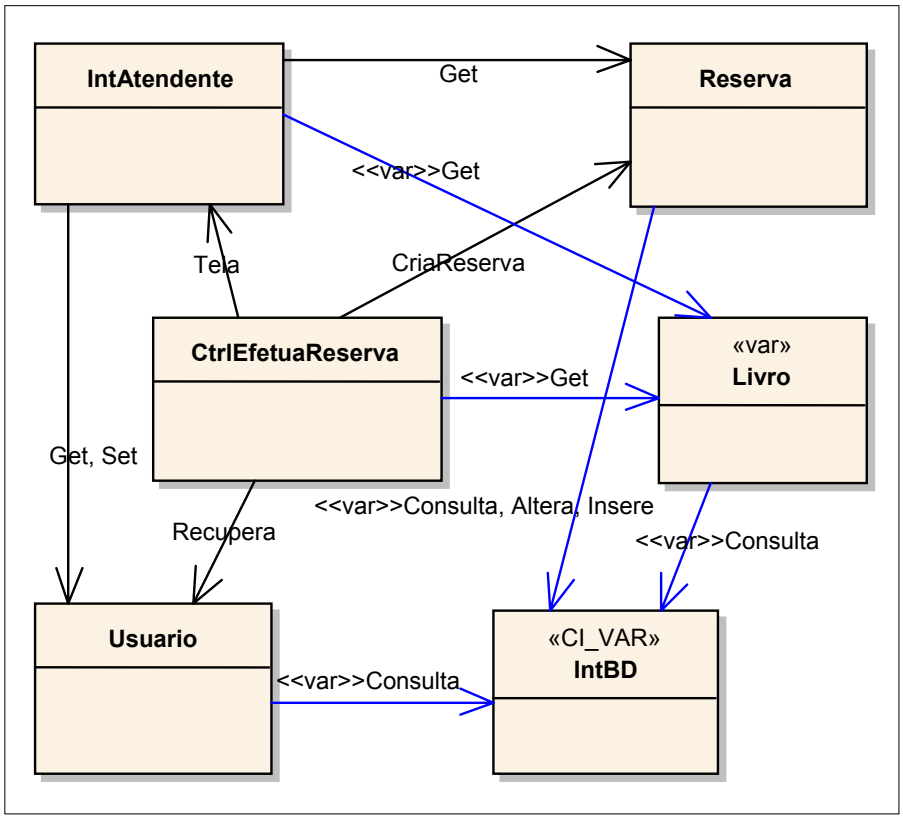


Figura III.5 – Diagrama de classes genérico para o caso de uso Reservar Obra

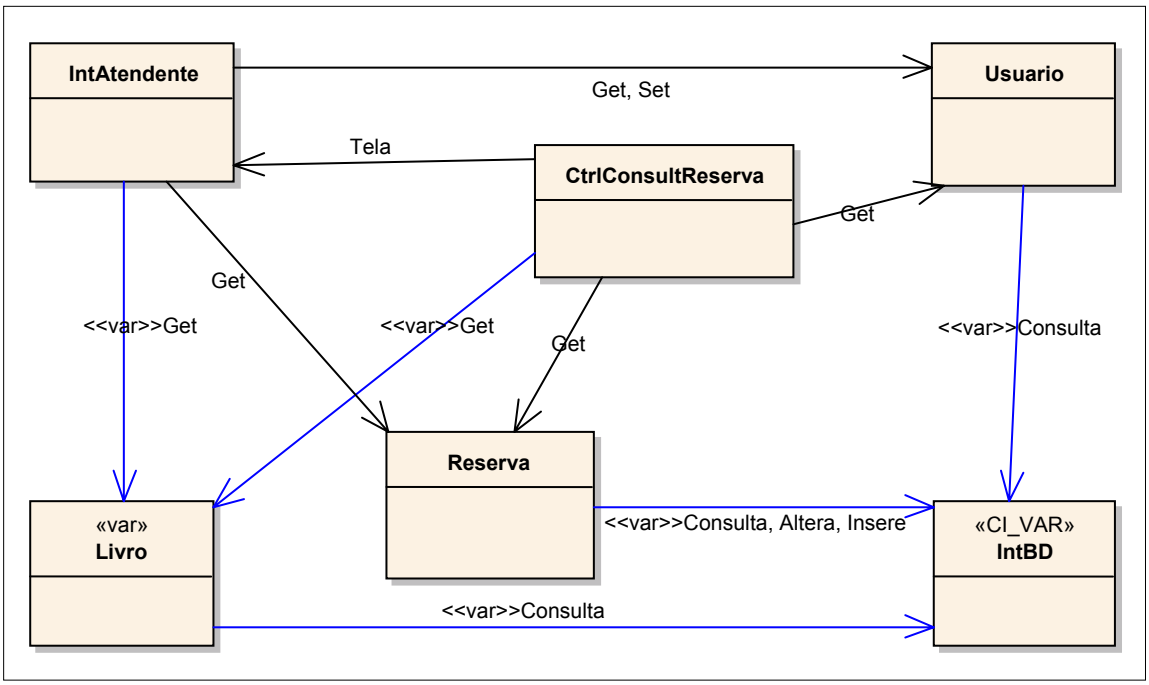


Figura III.6 – Diagrama de classes genérico para o caso de uso Consultar Reserva

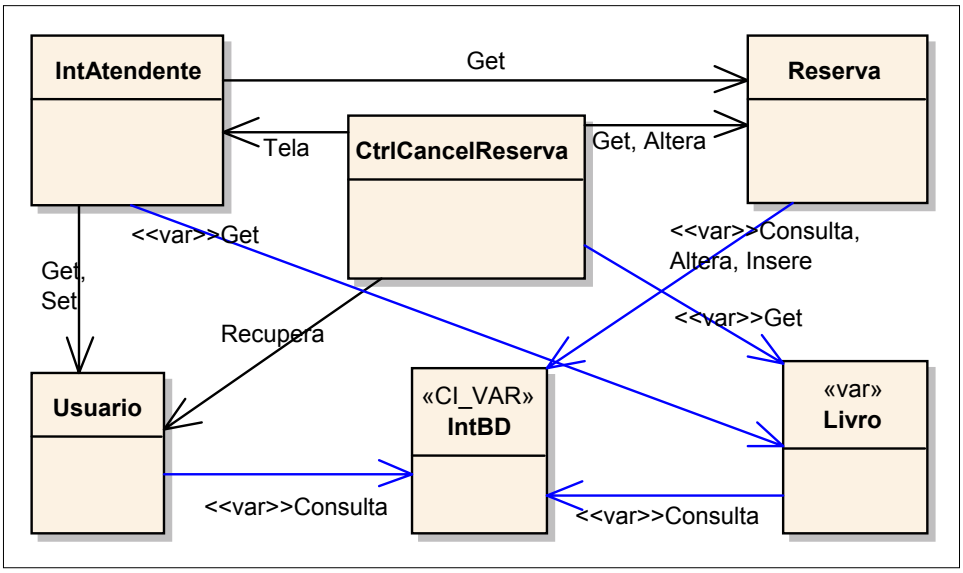


Figura III.7 – Diagrama de classes genérico para o caso de uso Cancelar Reserva de Obra

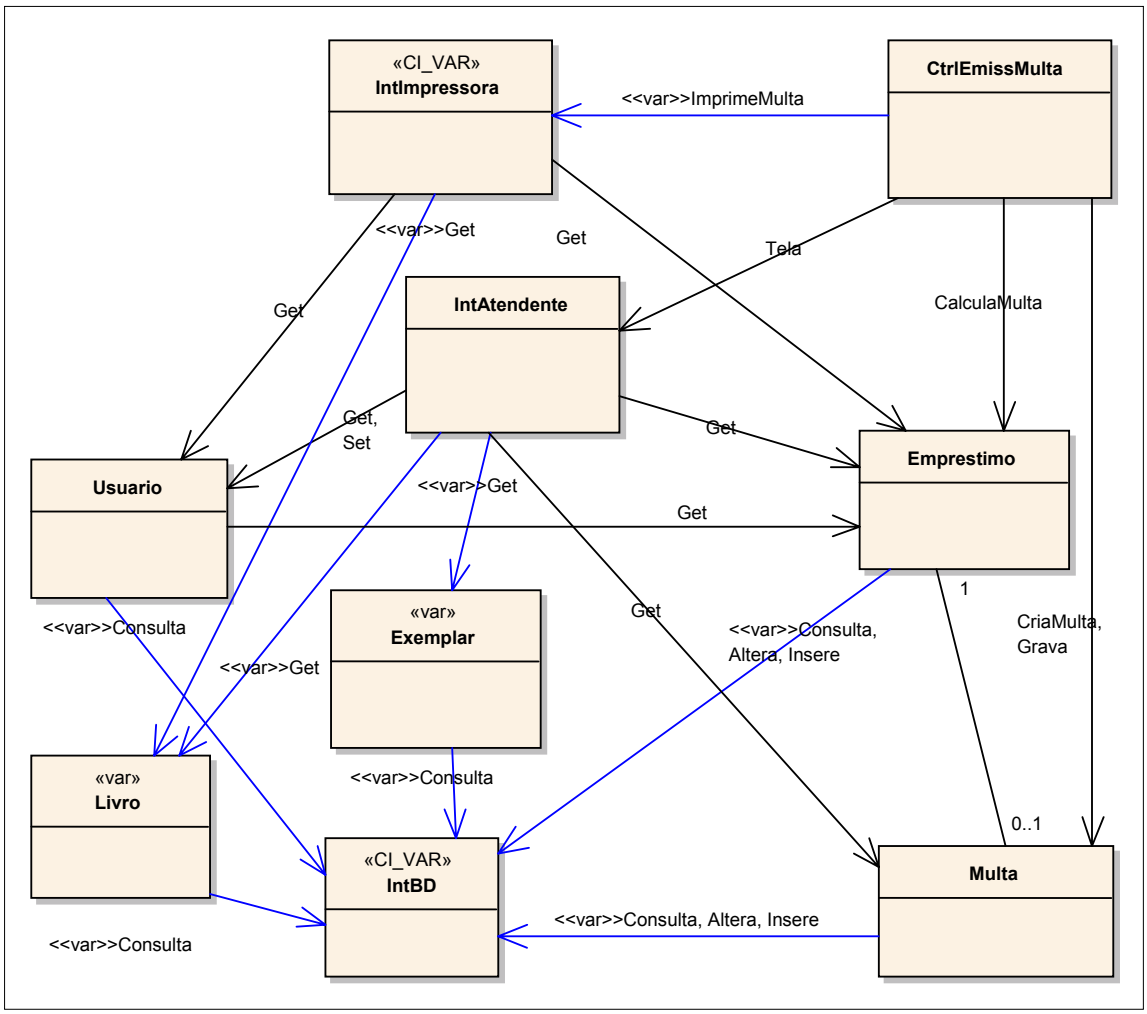


Figura III.8 – Diagrama de classes genérico para o caso de uso Emitir Multa

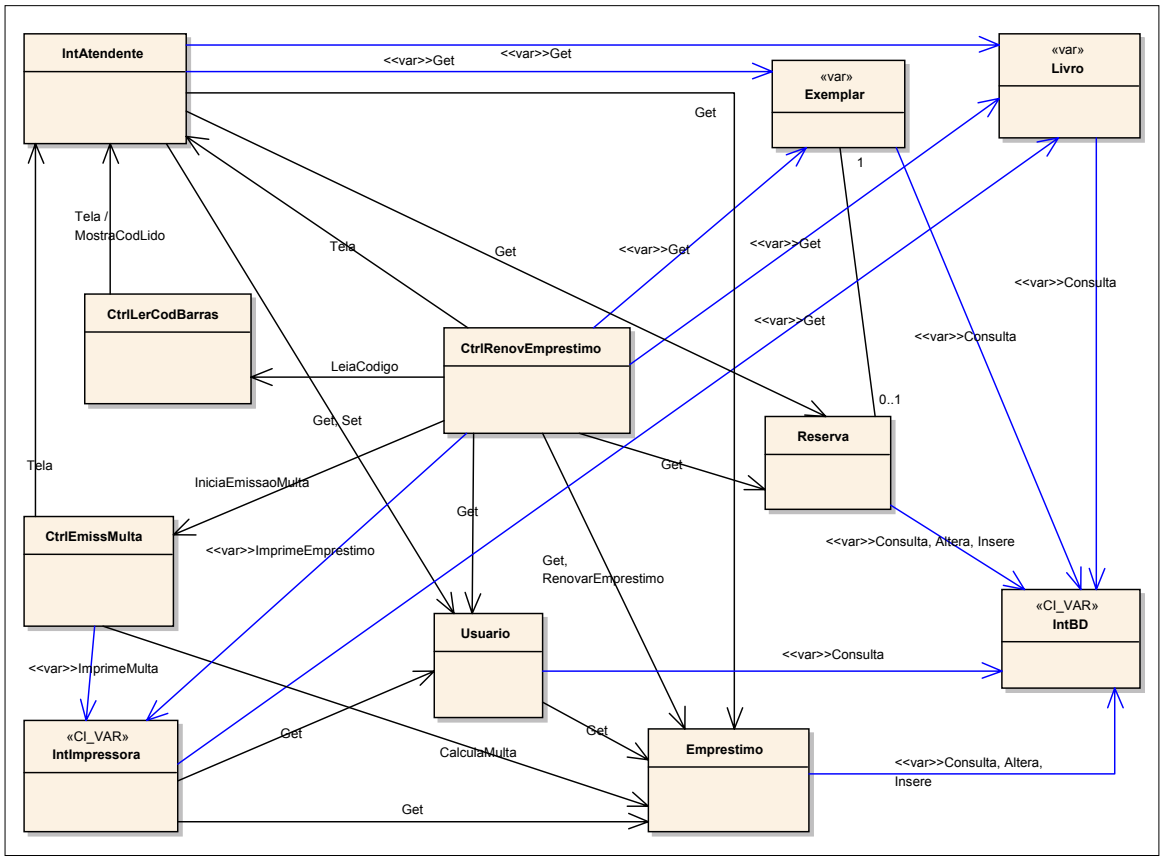


Figura III.9 – Diagrama de classes genérico para o caso de uso Renovar Empréstimo

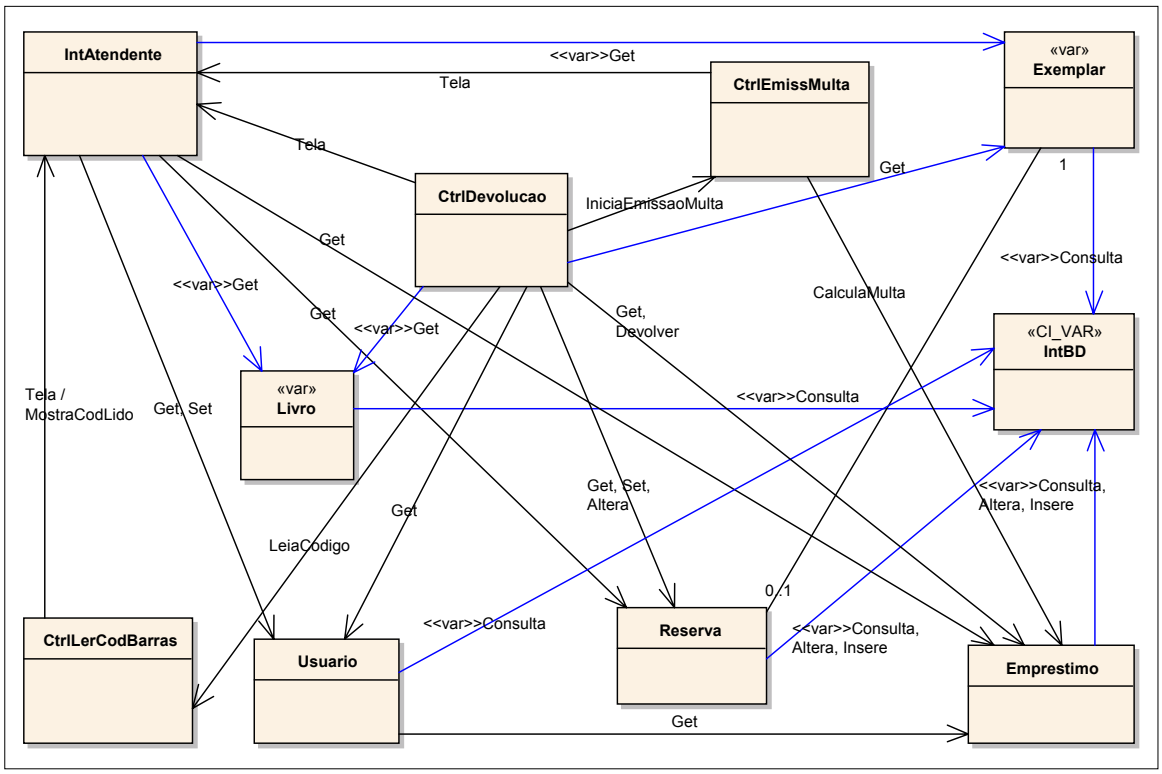


Figura III.10 – Diagrama de classes genérico para o caso de uso Devolver Exemplar