Alexandre José Tiberti

DESENVOLVIMENTO DE SOFTWARE DE
APOIO AO PROJETO DE ARRANJO FÍSICO DE FÁBRICA
BASEADO EM UM *FRAMEWORK* ORIENTADO A OBJETO

Tese apresentada à Escola de Engenharia de
São Carlos, Universidade de São Paulo,
como parte dos requisitos para obtenção do
Título de Doutor em Engenharia Mecânica.

Orientador: Prof. Dr. Eduardo Vila Gonçalves Filho

São Carlos
2003

*Dedico este trabalho às duas pessoas mais importantes em minha vida:
minha esposa Sônia e meu filho Arthur.*

AGRADECIMENTOS

Ao Professor Eduardo Vila Gonçalves Filho pelo trabalho de orientação e paciência ao longo desses anos, sem os quais este trabalho não teria sido realizado.

À minha esposa Sônia e a meu filho Arthur pelo amor e compreensão nas horas difíceis, e pela dedicação, esforço e paciência nas horas que lhes faltei em função da concretização deste trabalho.

Ao Professor Arthur José Vieira Porto pela confiança e credibilidade depositados no início deste trabalho.

Aos colegas do Laboratório de Simulação, à Beth, e a todos que, de uma forma ou de outra, contribuíram para o término deste trabalho.

À Professora Renata Pontin de Mattos Fortes pela ajuda e incentivo, ao final deste trabalho, que muito contribuíram para sua conclusão.

Por fim, aos professores e funcionários do Departamento de Engenharia Mecânica que contribuíram para realização deste trabalho.

RESUMO

TIBERTI, A.J. (2003). *Desenvolvimento de Software de Apoio ao Projeto de Arranjo Físico de Fábrica Baseado em um Framework Orientado a Objeto*. Tese (Doutorado) – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2003.

Nas últimas quatro décadas, métodos e algoritmos para análise e formação de arranjos físicos de máquinas, equipamentos, departamentos e serviços auxiliares vêm sendo desenvolvidos para reduzir o fluxo de materiais e o deslocamento de pessoal entre eles, objetivando diminuir custos e aumentar a eficiência do sistema produtivo. Diversos autores têm proposto algoritmos e ferramentas computacionais para aumentar a produtividade do projetista de arranjo físico, mas todos focam em um determinado conjunto de algoritmos, método, ou tipo de arranjo físico. Neste contexto, este trabalho de pesquisa buscou desenvolver um software que atue em um escopo mais amplo que o dessas ferramentas computacionais. A partir de uma estrutura baseada em um *framework* de aplicação orientado a objeto, o software proposto é capaz de suportar a introdução de novos métodos, algoritmos e ferramentas gráficas para análise e formação dos diversos tipos de arranjos físicos existentes, à medida que estes forem necessários, sem sofrer modificações ou gerar alterações significativas em seu código fonte. Isso permite ao desenvolvedor adaptar facilmente o software proposto às necessidades de um projeto de arranjo físico.

Palavras-Chave: projeto de arranjo físico; desenvolvimento de software; *frameworks* de aplicação; orientação a objeto.

ABSTRACT

TIBERTI, A.J. (2003). *Facilities Layout Support System Development Based on an Object-Oriented Application Framework*. Ph.D. Thesis – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2003.

In the last four decades, methods and algorithms have been developed to analyse and arrange the layout of machines, equipments, departments, and auxiliary services in order to minimize the movement of people and material flow. It is normally made to reduce costs and improve the production system efficiency. Several authors have proposed computational algorithms and tools to increase productivity of the layout designer, but all of them focus on a limited set of algorithms or methods, or on a particular layout type. In this research work, a computer program with a broader scope than current computational tools has been developed. By adopting a structure based on an object-oriented application framework, the proposed software supports the addition of new methods, algorithms, and graphical tools, which can be applied, as required, on several different layout design types without significant structural or source code changes. It enables the developer to adapt the proposed software for new layout design requirements easily.

Keywords: facilities layout design; software development; application framework; object-oriented.

LISTA DE FIGURAS

Figura 2.1 – Classificação de arranjo físico por volume-variedade	9
Figura 2.2 – Diagrama P-Q ou diagrama de Pareto	9
Figura 2.3 – Arranjo físico posicional (TOMPKINS et al., 1996, p.289).....	10
Figura 2.4 – Arranjo físico por processo ou funcional.....	11
Figura 2.5 – Arranjo físico por produto (TOMPKINS et al., 1996, p.289).....	13
Figura 2.6 – Linha de produção de papel (SLACK et al., 1997, p.217).....	13
Figura 2.7 – Arranjo físico celular	15
Figura 2.8 – Exemplos de células fractais (MONTREUIL, VENKATADRI e RARDIN, 1999).....	17
Figura 2.9 – Tipos de Módulos (IRANI e HUANG, 1998).....	18
Figura 2.10 – Fases do sistema SLP (MUTHER, 1978, p.5)	19
Figura 2.11 – Procedimentos do sistema SLP (MUTHER, 1978, p.7)	20
Figura 2.12 – Exemplo de Diagrama de Relacionamento.....	21
Figura 2.13 – Seqüência de decisões de arranjo físico (SLACK, 1997, p.212)	25
Figura 2.14 – Gráfico de tipos de processos de manufatura (a) e tipos de processos de serviços (b) (SLACK, 1997, p.135 e 137).....	26
Figura 2.15 – Processo geral de projeto de planta (TOMPKINS, 1996, p.14).....	27
Figura 2.16 – Modelo detalhado do processo de arranjo físico de fábrica (GONÇALVES FILHO, 2001, p.54)	28
Figura 3.1 – Modelo Espiral (PRESSMAN, 2002, p.35).....	41
Figura 3.2 – Modelo de Desenvolvimento Baseado em Componentes (PRESSMAN, 2002, p.40).....	42
Figura 3.3 – Modelo de Processo Orientado a Objeto (PRESSMAN, 2002, p.533).....	43
Figura 3.4 – Modelo de processo para a CBSE (PRESSMAN, 2002, p.707).....	46
Figura 4.1 – Síntese da abordagem tecnológica adotada.....	71
Figura 4.2 – Visão geral da arquitetura do software proposto	73
Figura 4.3 – Exemplo de uma Estrutura de Projeto	76
Figura 4.4 – Exemplo de uma árvore de projeto	76
Figura 4.5 – Exemplo de uma Estrutura da Aplicação.....	77
Figura 4.6 – O Metamodelo	79
Figura 4.7 – Exemplo de contexto de projeto	82
Figura 4.8 – Exemplo de uso do contexto geral.....	84
Figura 4.9 – Síntese da solução computacional adotada.	87

Figura 5.1 – Estrutura do <i>framework</i> de operação.	90
Figura 5.2 – Estrutura de um <i>framework</i> da Aplicação.....	91
Figura 5.3 – Diagrama de Classes dos módulos do <i>framework</i> de operação.	92
Figura 5.4 – Superclasses do <i>framework</i>	94
Figura 5.5 – Diagrama de sequência para a execução de um procedimento operacional.	95
Figura 5.6 – Exemplo de um diagrama classe-objeto para o algoritmo ROC.....	97
Figura 5.7 – Tabelas da BCA com as informações sobre o diagrama classe-objeto do algoritmo ROC da Figura 5.6.....	98
Figura 5.8 – Diagrama de classes dos pontos variáveis e classes auxiliares do <i>framework</i>	100
Figura 5.9 – Diagrama de classes do CIU com a Interface com o Usuário.....	102
Figura 5.10 – Diagrama de classes do MCO com o CIU, o GOD e o GPI.	103
Figura 5.11 – Diagrama de classes do MCO com suas sessões seguras e o Procedimento Operacional.....	104
Figura 5.12 – Organização hierárquica das sessões seguras.	105
Figura 5.13 – Diagrama de classes do GOD com o ROD, GPI e BCA.....	109
Figura 5.14 – Diagrama de classes do GPI com a BCA.....	111
Figura 5.15 – Esquema de mapeamento das propriedades dos objetos de dados.	115
Figura 5.16 – Diagrama de classes do ROD com a BCA.....	116
Figura 5.17 – Exemplo de mapeamento de um Objeto de Dados.	117
Figura 5.18 – Tabela <i>Application</i> da BCA.	118
Figura 5.19 – Tipos de elementos do Metamodelo.	119
Figura 5.20 – Diagrama de classes para construção das Estruturas de Roteiro.	120
Figura 5.21 – Rota seguida pelos mecanismos do método <i>GetScript()</i>	121
Figura 5.22 – Diagramas de classes dos tipos de vínculos entre Objetos de Dados.	122
Figura 5.23 – Métodos do Objeto de Dados usados para controle dos vínculos.....	123
Figura 5.24 – Entidade de vínculos do ROD.	124
Figura 5.25 – Modelo de processo para uso do <i>framework</i> da Aplicação.....	126
Figura 5.26 – Modelo Entidade-Relacionamento da BCR.....	129
Figura 5.27 – Tela principal do SGCA.	129
Figura 6.1 – Interface Padrão do SAPAF.....	133
Figura 6.2 – Pontos variáveis do <i>framework</i> do SAPAF: Procedimentos Operacionais.	135
Figura 6.3 – Pontos variáveis do <i>framework</i> do SAPAF: Interfaces com o Usuário.	136
Figura 7.1 – Estrutura do Projeto Piloto e dos Objetos de Dados.....	140

Figura 7.2 – Visualização de dados e comentários.	142
Figura 7.3 – Comparação visual de matrizes binárias.....	142
Figura 7.4 – Interface de seleção, controle e visualização de resultados do algoritmo genético.	143
Figura 7.5 – Diagrama de-para e o resultado produzido pelo algoritmo MST.	143
Figura A.1 – Exemplo de diagrama classe-objeto.....	154
Figura C.1 – Superclasses das classes reservadas.	178
Figura C.2 – Superclasses <i>TCriteria</i> e suas derivadas.	180

LISTA DE TABELAS

Tabela 2.1 – Relação entre tipos de processo e tipos básicos de arranjo físico (SLACK et al., 1997, p.213).	8
Tabela 2.2 – Exemplo de um Diagrama De-Para.	21
Tabela 4.1 – Necessidades do usuário × requisitos do usuário.	63
Tabela 4.2 – Necessidades do sistema X requisitos do sistema.	65
Tabela 4.3 – Características do software proposto.....	67

LISTA DE ABREVIATURAS E SIGLAS

BCA	Biblioteca de Componentes das Aplicações
BCR	Biblioteca de Componentes Reusáveis
BDOO	Banco de Dados Orientado a Objeto
BDO-R	Banco de Dados Objeto-Relacional
BDR	Banco de Dados Relacional
CAD	<i>Computer Aided Design</i>
CBSE	<i>Component-Based Software Engineering</i> (Engenharia de Software Baseada em Componente)
CIU	Controlador de Interfaces com o Usuário
GOD	Gerenciador de Objetos de Dados
GPI	Gerenciador de Procedimentos e Interfaces
MCO	Mecanismo de Controle Operacional
ODL	<i>Object Definition Language</i> (Linguagem de Definição de Objetos)
ODMG	<i>Object Data Management Group</i>
OML	<i>Object Management Language</i> (Linguagem para Gerenciamento de Objetos)
OO	Orientação a Objeto
OOA	<i>Object-Oriented Analysis</i> (Análise Orientada a Objeto)
OOAF	<i>Object-Oriented Application Frameworks</i> (Frameworks de Aplicação Orientados a Objeto)
OOD	<i>Object-Oriented Design</i> (Projeto Orientado a Objeto)
OOP	<i>Object-Oriented Programming</i> (Programação Orientada a Objeto)
OQL	<i>Object Query Language</i> (Linguagem de Consulta a Objetos)
ROD	Repositório de Objetos de Dados
SAPAF	Software de Apoio ao Projeto de Arranjo Físico
SGBD	Sistema Gerenciador de Banco de Dados
SGBDOO	Sistema Gerenciador de Banco de Dados Orientado a Objeto
SGBDR	Sistema Gerenciador de Banco de Dados Relacional
SGCA	Software Gerenciador de Componentes das Aplicações
SLP	<i>Systematic Layout Planning</i> (Planejamento de Arranjo Físico Sistemático)
SQL	<i>Structured Query Language</i> (Linguagem de Consulta Estruturada)
UML	<i>Unified Modeling Language</i> (Linguagem de Modelagem Unificada)

SUMÁRIO

RESUMO	iii
ABSTRACT	iv
LISTA DE FIGURAS	v
LISTA DE TABELAS	viii
LISTA DE ABREVIATURAS E SIGLAS	ix
CAPÍTULO 1. INTRODUÇÃO.....	1
1.1. Generalidades do Trabalho.....	1
1.2. Objetivos do Trabalho	3
1.3. Justificativa do Trabalho	3
1.4. Estrutura do Trabalho	5
CAPÍTULO 2. PROJETO DE ARRANJO FÍSICO.....	7
2.1. Considerações Iniciais	7
2.2. Tipos de Arranjo Físico	7
2.2.1. Escolha do Tipo de Arranjo Físico	8
2.2.2. Arranjo Físico Posicional	10
2.2.3. Arranjo Físico por Processo ou Funcional	10
2.2.4. Arranjo Físico por Produto	12
2.2.5. Arranjo Físico Celular	14
2.2.6. Arranjo Físico Fractal	15
2.2.7. Arranjo Físico Modular	17
2.3. Metodologias de Projeto de Arranjo Físico.....	19
2.3.1. O Modelo de Muther	19
2.3.2. O Modelo de Apple	22
2.3.3. O Modelo de Slack	25
2.3.4. O Modelo de Tompkins.....	27
2.3.5. O Modelo de Gonçalves Filho.....	28
2.4. Sistemas Computacionais de Apoio ao Projeto de Arranjo Físico.....	30
2.5. Considerações Finais	32
CAPÍTULO 3. DESENVOLVIMENTO DE SOFTWARE ORIENTADO A OBJETO	35
3.1. Considerações Iniciais	35
3.2. Princípios da Orientação a Objeto.....	36
3.3. Processo de Software.....	38
3.3.1. Modelos de Processo de Software	39
3.3.2. Modelo de Processo Orientado a Objeto	42
3.4. Métodos Orientados a Objeto.....	43
3.5. Processo de Software Utilizando a UML	44
3.6. Engenharia de Software Baseado em Componentes	46
3.7. Desenvolvimento de Software Baseado em <i>Frameworks</i> de Aplicação	50
3.7.1. <i>Frameworks</i> de Aplicação Orientados a Objeto (OOAF)	50
3.7.2. Processo de Desenvolvimento de OOAF's	53
3.8. Banco de Dados.....	54
3.9. Considerações Finais	57

CAPÍTULO 4. ANÁLISE DO PROBLEMA E SOLUÇÃO ADOTADA.....	60
4.1. Considerações Iniciais	60
4.2. Processo de Software Adotado	61
4.3. Levantamento e Análise do Problema	63
4.4. Abordagem Tecnológica Adotada	69
4.5. Elementos da Solução Adotada	73
4.5.1. Arquitetura do Software	73
4.5.2. Estruturas de Projeto	75
4.5.3. Estruturas das Aplicações	77
4.5.4. O Metamodelo	78
4.6. Considerações Finais	84
CAPÍTULO 5. O FRAMEWORK ORIENTADO A OBJETO	88
5.1. Considerações Iniciais	88
5.2. Estrutura do <i>Framework</i>	89
5.3. Projeto do <i>Framework</i> de Operação	91
5.3.1. Visão Geral	91
5.3.2. Aplicação do Metamodelo no Projeto do <i>Framework</i>	96
5.3.3. Controlador de Interfaces com o Usuário (CIU)	101
5.3.4. O Mecanismo de Controle Operacional (MCO)	102
5.3.5. O Gerenciador de Objetos de Dados (GOD)	108
5.3.6. O Gerenciador de Procedimentos e Interfaces (GPI)	111
5.3.7. O Repositório de Objetos de Dados (ROD)	113
5.3.8. A Biblioteca de Componentes das Aplicações (BCA)	117
5.3.9. Relacionamentos de Vínculo entre Objetos de Dados	121
5.4. Uso do <i>Framework</i> de Operação	124
5.5. Uso do <i>Framework</i> da Aplicação	126
5.6. O Software Gerenciador de Componentes das Aplicações (SGCA)	128
5.7. Aspectos da Implementação do <i>Framework</i> de Operação	130
CAPÍTULO 6. O FRAMEWORK DA APLICAÇÃO (SAPAF)	132
6.1. Considerações Iniciais	132
6.2. Tipo da Aplicação e sua Interface Padrão	132
6.3. Modelagem do <i>Framework</i> da Aplicação	134
6.4. Uso do <i>Framework</i> da Aplicação	137
CAPÍTULO 7. UMA INSTANCIÇÃO DO FRAMEWORK (SAPAF)	139
7.1. Considerações Iniciais	139
7.2. Modelagem da Estrutura do Projeto Piloto	139
7.3. Resultados Obtidos	141
CAPÍTULO 8. CONCLUSÕES	144
8.1. Resumo do Trabalho Efetuado	144
8.2. Conclusões Principais	145
8.3. Limitações do Trabalho	146
8.4. Trabalhos Futuros	147
REFERÊNCIAS	148
GLOSSÁRIO	152
APÊNDICE A – Diagramas e gabaritos para utilização do <i>Framework</i>	153

APÊNDICE B – Descrição das interfaces dos módulos e pontos variáveis do <i>Framework</i>	159
APÊNDICE C – Classes Reservadas do <i>Framework</i>	177

CAPÍTULO 1. INTRODUÇÃO

1.1. Generalidades do Trabalho

Nas quatro últimas décadas as indústrias de manufatura vêm sofrendo grandes mudanças para atender as novas exigências do mercado consumidor nacional e internacional. Essas mudanças iniciaram nos anos 60 com a tentativa das empresas de reduzir custos para competir em preço e conquistar o mercado consumidor, ampliaram com a busca da qualidade nos anos 70, e acirraram com a quebra das barreiras geopolíticas e econômicas no final dos anos 80 e início dos anos 90.

Nesse período, novas tecnologias e novas filosofias de produção foram surgindo para auxiliar as empresas a melhorar sua produtividade. *Just-in-time*, Tecnologia da Produção Otimizada (OPT) e Produção Enxuta (*Lean Production*) são exemplos de algumas dessas filosofias de produção. Comando Numérico (CN), Robótica, Projeto Auxiliado por Computador (CAD), Engenharia Auxiliada por Computador (CAE) e Planejamento de Processos Auxiliado por Computador (CAPP) são exemplos de algumas dessas tecnologias aplicadas com sucesso nas indústrias.

No entanto, sucessos isolados em alguns departamentos de uma empresa não garantem o sucesso dessa empresa junto ao mercado. A empresa deve ser vista como um todo (visão holística) e as mudanças devem ser conduzidas para fazer a empresa conquistar maior fatia de mercado ou aumentar sua lucratividade. Nesse contexto, a manufatura tem um papel determinante. Um sistema de manufatura enxuto e eficiente pode ser considerado um importante passo para garantir o sucesso de uma empresa perante o mercado. Segundo Gonçalves Filho (2001), um sistema de manufatura eficiente pode ser obtido combinando-se quatro variáveis: (1) tecnologia de fabricação atualizada, (2) um arranjo físico otimizado, (3) mão de obra treinada e motivada, e (4) um gerenciamento adequado da produção.

O arranjo físico determina a disposição física de máquinas, equipamentos, e serviços auxiliares como banheiros, refeitório, e outros, assim como áreas reservadas

para tráfego, armazenagem de material, peças e ferramental. Segundo Heragu (1997), 30% a 75% do custo de um produto pode ser atribuído a despesas com movimentação de materiais. Sendo assim, um arranjo físico adequado pode minimizar a movimentação de materiais e pessoal no chão de fábrica e, dessa forma, diminuir custos e aumentar a eficiência do sistema produtivo.

Projetos de arranjo físico são processos complexos e demorados, cujo desenvolvimento é feito por uma equipe de profissionais ou por um especialista conhecido como projetista de arranjo físico. O trabalho desses profissionais é definir um novo arranjo físico ou efetuar melhorias em um arranjo físico existente para atender aos novos objetivos da empresa perante o mercado. Para auxiliar nesse processo, vários autores vêm propondo ao longo dos últimos 50 anos diversos modelos de processo de projeto de arranjo físico, juntamente com um número grande e crescente de métodos, ferramentas gráficas e algoritmos para auxiliar em cada passo ou atividade desses modelos. No entanto, a dificuldade em se aplicar esses modelos, métodos, ferramentas gráficas e algoritmos, devido a sua complexidade ou ao volume expressivo de dados a ser obtido e analisado, torna o processo demorado, e algumas vezes as mudanças no maquinário ou processo produtivo ocorrem antes mesmo que o projeto do arranjo físico seja concluído (TOMPKINS et al., 1996). Com o rápido avanço dos computadores e suas aplicações, diversos autores têm proposto ferramentas computacionais para auxiliar na aplicação desses métodos e algoritmos, gerando arranjos em um curto intervalo de tempo. Isso possibilitou ao projetista obter rapidamente um número razoável de resultados para comparação e testes de desempenho.

No entanto, apesar de todos os esforços de pesquisa nessa área, a maioria das ferramentas computacionais desenvolvidas focam em um determinado conjunto de algoritmos ou métodos, fornecendo suporte a um número restrito de atividades do projeto de arranjo físico. Isso limita o escopo de trabalho dos projetistas, forçando-os a utilizar várias ferramentas computacionais ou manuais para desenvolver todas as atividades necessárias. Nesse processo, surgem diversas dificuldades com relação à preparação dos dados, escolha do algoritmo com a função-objetivo mais apropriada, interpretação dos resultados, dentre outras dificuldades apresentadas por Hassan (2000).

Dentro deste contexto, este trabalho de pesquisa buscou desenvolver um software cuja flexibilidade em sua estrutura (baseada em um *framework* de aplicação orientado a objeto – OOAF) permite introduzir, de maneira simples, novos métodos, algoritmos e

ferramentas gráficas para análise e formação dos diversos tipos de arranjos físicos existentes, à medida que estes forem necessários, sem necessitar modificações nessa estrutura ou no código fonte do software. Isso permite que ele atue em escopo mais amplo que o da maioria das ferramentas computacionais de apoio ao projeto de arranjo físico existentes.

1.2. Objetivos do Trabalho

O objetivo principal deste trabalho é o desenvolvimento de um software de apoio ao projeto de arranjo físico, inicialmente com finalidades acadêmicas e de pesquisa, com:

- uma estrutura que suporte incorporar um número razoável de diferentes tipos e estruturas de dados, algoritmos e ferramentas gráficas empregados no desenvolvimento dos projetos de arranjo físico, e que facilite a introdução futura de novos desses elementos sem sofrer modificações ou gerar alterações significativas em seu código fonte; e
- um ambiente de trabalho que disponibilize ao projetista de arranjo físico, de maneira orientada, os dados, algoritmos e ferramentas gráficas necessários ao desenvolvimento de várias atividades de análise e formação (detalhamento) de arranjos físicos para que o mesmo possa desenvolver essas atividades com rapidez e eficiência.

Além disso, a abordagem tecnológica adotada para que o software proposto obtivesse essas características utiliza um *framework* de aplicação orientado a objeto (OOAF) cuja forma de desenvolvimento e uso podem ser considerados como um segundo objetivo conseqüente deste trabalho.

1.3. Justificativa do Trabalho

A idéia de desenvolver o software de apoio proposto neste trabalho surgiu, inicialmente, da dificuldade em se adquirir um software com um ambiente de trabalho adequado e código fonte aberto que pudesse ser aplicado no desenvolvimento de diversos trabalhos de pesquisa na área de arranjos físico, tais como:

- avaliar o grau de eficiência de novos métodos e algoritmos na formação, construção ou melhoria de arranjos físicos, assim como dos já existentes;

- estudar formas de integração de um software para arranjo físico com sistemas de simulação e CAD para agilizar as atividades de avaliação e detalhamento dos arranjos físicos formados;
- facilitar a preparação dos dados necessários à aplicação de cada método ou algoritmo; e
- preservar e gerenciar de maneira eficientes os dados, obtidos e gerados durante os estudos, para comparação e análise futura de resultados.

Apesar dos estudos sobre arranjos físicos serem antigos e numerosos, a maioria dos softwares desenvolvidos implementa um método ou conjunto de algoritmos específicos, e dificilmente permitem a inclusão de novos algoritmos e métodos por duas razões principais:

- Softwares comerciais não costumam permitir alterações, uma vez que seu código fonte é de propriedade intelectual do fabricante e este raramente o coloca nas mãos de terceiros.
- Softwares acadêmicos, em geral, são implementados com o objetivo de testar e validar um método ou algoritmo específico. Como softwares dedicados a um pequeno domínio do problema acabam aceitando apenas elementos deste domínio, a introdução de novos elementos fora deste domínio nos mesmos implicaria em modificações significativas em seus códigos fontes e, até mesmo, em suas estruturas, inviabilizando a reutilização dos mesmos.

Além disso, a utilização desses softwares nem sempre é muito amigável. Vários deles contêm diversos algoritmos cujos dados de entrada e saída costumam ser diferentes e necessitam ser preparados e interpretados. Essas tarefas muitas vezes são feitas manualmente e consomem muito tempo do projetista, além de facilitarem a ocorrência de erros. Daí a necessidade de se ter um software capaz de disponibilizar um ambiente de trabalho com interfaces amigáveis e ferramentas para automatizar o trabalho de preparação e interpretação dos dados.

A utilização de um OOAF para o desenvolvimento do software proposto veio da necessidade de se definir uma estrutura que suporte a inclusão dos diferentes tipos de algoritmos existentes, juntamente com seus diversos tipos de dados e rotinas de preparação e conversão dos mesmos, de maneira simples, sem sofrer modificações ou gerar alterações significativas no código fonte.

1.4. Estrutura do Trabalho

Neste capítulo, são apresentados os elementos motivadores deste trabalho de pesquisa e seus objetivos. Nos capítulos seguintes, são discutidos esses elementos, apresentado o problema, e proposto uma solução computacional cuja abordagem tecnológica adotada resultou na principal contribuição deste trabalho de pesquisa: um software baseado em um *framework* de aplicação orientado a objeto cuja forma de desenvolvimento e uso permitem a inclusão de novos componentes sem necessitar modificações estruturais ou gerar alterações em seu código fonte.

No Capítulo 2, é feita a primeira parte da revisão bibliográfica, na qual são apresentados os principais elementos que compõem um processo de projeto de arranjo físico. Ao final, são discutidas as principais dificuldades verificadas nesse processo e como um software de apoio poderia minimizá-las.

No Capítulo 3, é apresentada a segunda e última parte da revisão bibliográfica. São apresentados os elementos tecnológicos envolvidos em um desenvolvimento de software orientado a objeto, focando nas tecnologias empregadas na solução adotada para o desenvolvimento do software proposto. Ao final, é discutida a utilização dessas tecnologias em um software de apoio ao projeto de arranjo físico.

No Capítulo 4, é levantado o problema e proposta a solução. Os elementos e dificuldades apresentados no Capítulo 2 são analisados e transformados em um conjunto de requisitos e características funcionais e estruturais para o software proposto que caracterizam o domínio do problema. Em seguida, é apresentada a abordagem tecnológica adotada em resposta a esse problema, que combina as tecnologias apresentadas no Capítulo 3. Fechando o capítulo, são apresentados os elementos que compõem essa solução e sua utilização no processo de desenvolvimento do software proposto, cujos detalhes são apresentados nos capítulos seguintes.

No Capítulo 5, é apresentado o *framework* de aplicação utilizado no desenvolvimento do software proposto. São fornecidos detalhes de seu projeto conceitual, de seu desenvolvimento e de sua implementação, e mostrado, em linhas gerais, como ele pode ser utilizado no desenvolvimento de um software dentro de seu domínio de aplicação.

No Capítulo 6, é apresentado um segundo *framework*, denominado *framework* da Aplicação, que é utilizado como base no desenvolvimento das diversas aplicações do software proposto.

No Capítulo 7, é apresentada uma Aplicação completa, desenvolvida para demonstrar a solução, que utiliza alguns dos algoritmos apresentados no Capítulo 2 para três tipos de arranjos físicos tradicionais: por produto, celular e funcional.

O Capítulo 8 encerra o texto descritivo desta tese. São apresentados um resumo da mesma, as principais conclusões obtidas, os pontos fracos e fortes da solução adotada e trabalhos que podem ser desenvolvidos no futuro para a sua melhoria.

CAPÍTULO 2. PROJETO DE ARRANJO FÍSICO

2.1. Considerações Iniciais

Projetos de arranjos físicos são processos complexos, normalmente designados a engenheiros, também chamados de projetistas de arranjo físico, cujo trabalho é definir um novo arranjo físico ou efetuar melhorias em um arranjo físico existente para atender aos novos objetivos da empresa perante o mercado. Um bom projeto de arranjo físico pode garantir um sistema de manufatura mais enxuto e mais eficiente, dois importantes fatores para garantir o sucesso de uma empresa no mercado competitivo.

Por essas e outras razões, inúmeras pesquisas vêm sendo desenvolvidas ao longo dos últimos 50 anos para definir os tipos de arranjo físico mais adequados a um ou outro processo produtivo, métodos e algoritmos para auxiliar na análise e formação desses arranjos físicos, e modelos de processos para orientar o projetista em seu trabalho. Com a evolução da informática nos últimos 40 anos, surgiram muitos sistemas computacionais de apoio ao projeto de arranjo físico que implementam diversos desses algoritmos e métodos.

Neste capítulo, serão discutidos esses tópicos e apresentados os principais elementos e dificuldades no desenvolvimento de projetos de arranjos físicos que caracterizaram o domínio do problema analisado para a concepção do software proposto.

2.2. Tipos de Arranjo Físico

Na literatura, podem ser encontrados diversos tipos de arranjo físico, cada qual com suas vantagens, aplicações e formas. No entanto, a grande maioria é derivada de quatro tipos básicos ou é uma combinação dos mesmos. Esses quatro tipos de arranjo físico são denominados:

- Arranjo físico posicional;
- Arranjo físico por processo ou funcional;
- Arranjo físico por produto; e
- Arranjo físico celular.

Esses quatro tipos são apresentados nessa seção, assim como dois outros tipos não convencionais, conhecidos como arranjo físico fractal e arranjo físico modular. Também são citados alguns métodos e algoritmos para análise e formação dos mesmos, bem como algumas técnicas para escolha do tipo de arranjo físico mais apropriado a um tipo de processo de produção.

2.2.1. Escolha do Tipo de Arranjo Físico

Na literatura, podem ser encontradas várias formas de se selecionar um tipo de arranjo físico. As mais comuns são:

- Baseado no processo de manufatura; e
- Baseado no volume-variedade de produtos.

A escolha do arranjo físico é uma técnica que busca relacionar os diversos tipos de processo com os tipos de arranjo físico mais apropriados a sua implementação. Segundo Slack et al. (1997), ela não é uma técnica totalmente determinística, apenas sugere uma forma de seleção. A Tabela 2.1 mostra uma sugestão de relacionamento entre os diversos tipos de processos existentes e os quatro tipos básicos de arranjo físico.

Tabela 2.1 – Relação entre tipos de processo e tipos básicos de arranjo físico (SLACK et al., 1997, p.213).

Tipos de Processo de Manufatura	Tipos Básicos de Arranjo Físico	Tipos de Processo de Serviço
Processo por Projeto	Arranjo Físico Posicional	Serviços Profissionais
Processo Tipo <i>Jobbing</i>	Arranjo Físico por Processo	
Processo Tipo <i>Batch</i>	Arranjo Físico Celular	Loja de Serviços
Processo em Massa	Arranjo Físico por Produto	Serviços de Massa
Processo Contínuo		

A escolha do arranjo físico baseada no volume-variedade de produtos é uma técnica que busca relacionar um tipo de arranjo físico à quantidade e à variedade de produtos fabricados. A Figura 2.1 mostra uma relação entre tipos de arranjo físico e o volume-variedade de peças. Essa relação não é precisa mas pode auxiliar na escolha do

tipo de arranjo físico mais adequado para um grupo de peças ou produtos. Uma outra forma é mostrada na Figura 2.2 e trata-se de um diagrama P-Q (Produto-Quantidade) ou diagrama de Pareto. A interpretação gráfica desse diagrama é feita de duas maneiras: (1) se a curva é muito acentuada significa que mais que um tipo de arranjo físico deverá ser adotado (Figura 2.2a); (2) se a curva é suave significa que apenas um tipo de arranjo físico necessita ser adotado (Figura 2.2b).

Figura 2.1 – Classificação de arranjo físico por volume-variedade

Figura 2.2 – Diagrama P-Q ou diagrama de Pareto

2.2.2. Arranjo Físico Posicional

O arranjo físico posicional possui uma característica que o difere dos demais: enquanto nos outros tipos de arranjo físico o material a ser transformado vai até a estação de trabalho, nele as estações de trabalho vão até o material para transformá-lo (Figura 2.3).

Arranjos físicos posicionais podem ser encontrados na fabricação de navios, na montagem de aviões e na construção de edifícios, cujas dimensões e peso inviabilizam ou impossibilitam sua movimentação. Outros exemplos são citados por Slack et al. (1997):

- Cirurgia de coração: o paciente não pode ser movido do local e os recursos de transformação devem ir até ele.
- Manutenção de computadores de grande porte: produto muito grande e delicado para ser movido até a fábrica.
- Construção de rodovias: o produto não pode ser movido.

Figura 2.3 – Arranjo físico posicional (TOMPKINS et al., 1996, p.289)

Na literatura referenciada, apenas Slack et al. (1997) apresentam um procedimento para formação desse tipo de arranjo físico, dividido em seis passos, que buscam definir a forma de instalação e como os possíveis locais em que os recursos transformadores serão centralizados, e fazer uma análise baseada em um conjunto de critérios de avaliação para determinar a localização de cada um desses locais em torno do produto a ser transformado.

2.2.3. Arranjo Físico por Processo ou Funcional

O arranjo físico por processo, também conhecido como arranjo físico funcional, é caracterizado pelo agrupamento de equipamento e máquinas que desempenham o mesmo processo em um mesmo espaço físico denominado seção ou departamento. Sendo assim, o material em transformação percorrerá um roteiro de um departamento a

outro para que as operações necessárias a sua transformação sejam realizadas (Figura 2.4). Como produtos diferentes costumam possuir diferentes necessidades, eles percorrem diferentes roteiros, criando padrões de fluxo bastante complexos (SLACK et al., 1997). Segundo Gonçalves Filho (2001), este é um tipo de arranjo físico que surgiu com a primeira revolução industrial, e é a forma mais antiga de organização de recursos de produção.

Figura 2.4 – Arranjo físico por processo ou funcional

Nesse tipo de arranjo físico, as seções ou departamentos são dispostos de acordo com o fluxo de materiais entre eles. Isso gera um alto fluxo interdepartamental e um baixo fluxo intradepartamental, como pode ser observado na Figura 2.4 em que as rotas dos três produtos são traçadas no desenho esquemático do arranjo físico.

Arranjos físicos por processo ou funcionais podem ser encontrados no chão de fábrica na forma de seções de pintura, soldagem, torneamento, montagem, etc., e na parte administrativa das indústrias ou empresas prestadoras de serviço na forma de departamentos, como departamento de compras, vendas, atendimento ao cliente, suporte técnico, etc.. Outros exemplos são citados por Slack et al. (1997):

- Hospital: alguns processos, como raios-X e laboratório, são necessários a um grande número de tipos diferentes de pacientes, enquanto outros processos pedem altos níveis de utilização de recursos, como leitos e equipes de atendimento.
- Supermercado: alguns processos, como locais que dispõem de vegetais enlatados, oferecem maior facilidade na escolha e reposição do produto se mantidos agrupados. Alguns setores, como o de comida congelada, são mantidos assim por necessitarem de sistemas de refrigeração. Já vegetais frescos costumam ser mantidos juntos para atrair a atenção do cliente.

Os métodos e algoritmos empregados no projeto desse tipo de arranjo físico são conhecidos como métodos e algoritmos para construção ou melhoria de *layouts* de bloco

ou *layout* de planta. Construção e melhoria de *layouts* são as duas formas de classificações para esses métodos e algoritmos encontradas na literatura:

- Métodos e algoritmos de construção utilizam apenas os dados para formação de arranjos físicos para definir novos *layouts* de bloco, ignorando qualquer *layout* existente, caso haja algum.
- Métodos e algoritmos de melhoria utilizam os dados para formação de arranjos físicos para otimizar um *layout* de bloco existente.

Por ser o mais antigo tipo de arranjo físico, muitos autores propuseram algoritmos computacionais para sua construção ou melhoria. Os algoritmos computacionais mais antigos conhecidos são: *Computerized Relative Allocation of Facilities Technique* (CRAFT), BLOCPLAN, *Layout Optimization with Guillotine Induced Cuts* (LOGIC) (TOMPKINS et al., 1996), e *Maximum Spanning Tree* (MST) (BAASE, 1978). Esses algoritmos, em geral, objetivam minimizar fluxo ou maximizar adjacência entre os diversos departamentos. Em trabalhos mais recentes, novas tecnologias vêm sendo empregadas no desenvolvimento desse tipo de algoritmo, como o uso de um algoritmo genético híbrido (HGA) proposto por Lee e Lee (2002) para tentar resolver o problema de arranjo físico de departamentos com áreas de tamanhos diferentes.

2.2.4. Arranjo Físico por Produto

O arranjo físico por produto, ou arranjo físico para linha de produção ou linha de montagem como também é conhecido, é caracterizado pela forma de transporte do material em transformação, que é conduzido através das estações de trabalho, onde sofre operações de fabricação ou montagem, na forma de um fluxo unidimensional. Nesse tipo de arranjo físico, os recursos produtivos são organizados sequencialmente de acordo com as operações requisitadas pelo material a ser transformado. Sendo assim, o material em transformação segue um roteiro pré-definido, passando por todas as estações de trabalho da linha.

Arranjos físicos por produto podem ser encontrados na fabricação de produtos em larga escala, como eletrodomésticos, sapatos, bebidas, etc.. Outros exemplos são citados por Slack et al. (1997):

- Montagem de automóveis: uma sequência de processos é utilizada para completar cada modelo.

- Programas de vacinação em massa: todos os clientes requerem a mesma seqüência de atividades, tais como identificação, vacinação e aconselhamento.
- Restaurante self-service: os clientes passam por várias seções, como frios, saladas, prato principal, sobremesas e café, seguindo sempre da mesma maneira.

A Figura 2.5 mostra um exemplo na área metal-mecânica com quatro linhas de fabricação nas quais o produto segue unidirecionalmente de uma estação a outra, terminando em uma única estação de montagem. A Figura 2.6 mostra um outro exemplo na área de papel e celulose com um linha de produção de bobinas de papel. Note que o processo segue sempre a mesma seqüência iniciando pelo cozimento da matéria prima, seguido pela sua limpeza, refinamento, mistura, alinhamento, prensagem, secagem, e finalizando com o embobinamento do produto acabado.

Figura 2.5 – Arranjo físico por produto (TOMPKINS et al., 1996, p.289)

Figura 2.6 – Linha de produção de papel (SLACK et al., 1997, p.217)

A maior preocupação na formação desse tipo de arranjo físico é com o balanceamento da linha, cujo objetivo é organizar as estações de trabalho em uma seqüência tal que otimize a medida de eficiência adotada, como diminuir a ociosidade nas estações, reduzir o número de estações, etc.. Na literatura, podem ser encontrados vários métodos e algoritmos para isso. Ghosh e Gagnon (1989) fazem uma revisão sobre

o problema do balanceamento da linha e discutem vários métodos baseados em algoritmos desenvolvidos entre os anos 60 e 80, como o CONSOAL, CLAB, MALB, NULISP e MUST. Askin e Standridge (1993) apresentam, além do CONSOAL, o *Ranked Positional Weight* (RPW). De Lit et al. (2001) apresentam um algoritmo genérico (OGA) para seqüenciamento e ordenação da linha baseado em uma função com múltiplos critérios de custo que considera o fluxo e a dependência de cada estação de trabalho.

2.2.5. Arranjo Físico Celular

Arranjos físicos celulares surgiram com as mudanças de foco produtivo a partir da década de 60 e são formados para produzir famílias de peças (IRANI, 1999; TOMPKINS et al., 1996). Uma família de peças é composta por dois ou mais tipos de peças similares. Esta similaridade é baseada em várias características, sendo operações de fabricação e geometria das peças as características mais comumente empregadas. Sendo assim, um arranjo físico celular é caracterizado pelo agrupamento de máquinas e equipamentos utilizados no processo de transformação de uma família de peças, que são dispostos em um espaço físico denominado célula.

Ao contrário do arranjo físico por processo, observa-se que no arranjo físico celular existe um alto fluxo intradepartamental (dentro da célula) e um baixo fluxo interdepartamental (fora da célula). Em geral, o fluxo de material de uma peça se dá dentro da célula, mas algumas peças podem necessitar de processamento em uma ou mais máquinas pertencentes a outras células. Apesar disso, pode-se dizer que o padrão de fluxo se aproxima do padrão de fluxo observado em arranjos físicos por produto (GONÇALVES FILHO, 2001).

A Figura 2.7 mostra um exemplo na área metal-mecânica com duas células de manufatura com o fluxo seguido por duas peças. Exemplos de outras áreas são citados por Slack et al. (1997):

- Áreas para produtos específicos em supermercados: são áreas reservadas para clientes que utilizam o supermercado para tomar lanche em seu horário de almoço. Nesses locais costumam ser colocados juntos vários produtos diferentes, como pães, frios, iogurtes, sucos, etc..

- Maternidade em um hospital: clientes que necessitam de atendimento em maternidade formam um grupo bem definido que necessita de diversos recursos específicos que podem ser agrupados em um único local.

Figura 2.7 – Arranjo físico celular

Devido a grande utilização desse tipo de arranjo físico, existe uma vasta literatura sobre métodos e algoritmos empregados em seus projetos. As abordagens mais conhecidas são codificação e classificação, análise de fluxo da produção, técnicas de agrupamento, procedimentos heurísticos e modelos matemáticos (TOMPKINS et al., 1996). Muitos algoritmos foram criados para auxiliar em cada uma dessas abordagens utilizando as mais variadas técnicas que vão desde formas de agrupamento baseadas em lógica matemática e heurística, como o DCA (*Direct Clustering Algorithm*), o ROC (*Rank Order Clustering*), coeficiente de similaridade (KUSIAK e CHO, 1992), e K-Means (IRANI et al., 2000), até algoritmos genéticos, como os propostos por Billo, Bidanda e Tate (1994), De Lit, Falkenauer e Delchambre (2000), Brown e Sumichrast (2001), e Gonçalves Filho (2001). Em geral, os algoritmos matemáticos e heurísticos tomam como entrada uma matriz peça x máquina 0-1 ou binária, em que cada elemento representa a necessidade (1) ou não (0) de operação de uma peça em uma máquina, e objetivam agrupar peças e máquinas. Já, os algoritmos genéticos costumam trabalhar com funções com múltiplos objetivos e tentam otimizar o fluxo intercelular e intracelular e/ou custos. Por esta razão, costumam tomar como entrada matrizes de tempo de transporte de uma peça entre uma máquina e outra, demanda de peças, tamanho de lotes, dentre outros, e produzem como resultado grupos de peças x máquinas.

2.2.6. Arranjo Físico Fractal

O arranjo físico fractal, proposto por Benoit Montreuil, Uday Venkatadri, e Ronald L. Rardin (VENKATADRI, RARDIN e MONTREUIL, 1997), é um dos tipos

de arranjo físico mais recentes e sua aplicação é muito pouco conhecida. Ele é caracterizado pelo agrupamento de máquinas e equipamentos em unidades denominadas células fractais. Cada célula fractal é vista como uma mini-fábrica dentro da fábrica e deve ser capaz de produzir todos, senão a maioria, dos produtos que entram no sistema de fabricação.

Segundo Gonçalves Filho (2001), o fato de uma peça poder ser processada em qualquer célula fractal, dá a esse tipo de arranjo físico uma flexibilidade não encontrada em arranjos físicos celulares. Está é a sua grande vantagem. No entanto, por conter tipos diversos de processos e por processar peças de diversos tipos pouco similares, uma célula fractal apresenta maior dificuldade de gerenciamento e seu fluxo interno é menos otimizado que em uma célula de manufatura.

Segundo Montreuil, Venkatadri e Rardin (1999), as células fractais não necessitam obrigatoriamente ser totalmente independentes, ou as estações de trabalho estarem distribuídas da mesma maneira e ordem dentro de cada célula fractal. A Figura 2.8 mostra três exemplos distintos sobre este assunto. A Figura 2.8 (a) mostra um arranjo físico fractal em que as quatro células são idênticas e todas as estações de trabalho aparecem proporcionalmente dentro delas; nesse caso, todas as células podem fabricar qualquer produto que entre no sistema de fabricação. A Figura 2.8 (b) mostra o mesmo arranjo físico fractal com quatro células, mas a disposição das estações é diferente em cada uma; nesse caso, uma célula fractal possui melhor desempenho em termos de fluxo para produzir um determinado grupo de peças que outras. A Figura 2.8 (c) mostra um arranjo físico fractal similar aos anteriores, no entanto, a estação de trabalho do tipo **D** na célula 4 é compartilhada pelas células 3 e 4; nesse caso se não houver o compartilhamento, a célula 3 não poderá processar qualquer produto que solicite essa estação de trabalho.

Gonçalves Filho (2001) comenta que esse tipo de arranjo físico pode ser aplicado à manufatura de produtos com alta variedade e em lotes pequenos, ou em situações nas quais a fabricação do produto requer mão-de-obra intensiva e equipamentos simples e baratos, como a produção de cestas de madeira.

Por existir muito pouca literatura sobre esse tipo de arranjo físico, apenas os métodos desenvolvidos pelos autores que o propuseram, descritos em Venkatadri, Rardin e Montreuil (1997) e em Montreuil, Venkatadri e Rardin (1999), podem ser aplicados em seu projeto.

Figura 2.8 – Exemplos de células fractais (MONTREUIL, VENKATADRI e RARDIN, 1999)

2.2.7. Arranjo Físico Modular

O arranjo físico modular é um dos tipos de arranjo físico mais recentes e foi apresentado por Shahrukh A. Irani e Heng Huang em 1998 (IRANI e HUANG, 1998). Segundo os autores, os quatro tipos de arranjos físicos convencionais, Posicional, Funcional, por Produto e Celular (vistos anteriormente) não são eficientes o bastante para reduzir o fluxo de material em um chão de fábrica que produz uma grande variedade de produtos diferentes, e apresentam o arranjo físico modular como uma alternativa para resolver esse problema. A idéia básica desse tipo de arranjo físico é usar os arranjos físicos funcional, por produto e celular como blocos de construção, montados na forma de uma rede orientada pelo fluxo de material dos produtos. Sendo assim, o arranjo físico modular é caracterizado por um conjunto de módulos organizados de acordo com a rede de fluxo de material geral, e cada módulo é composto por um grupo de máquinas conectadas por uma rede de fluxo de material local que exhibe um padrão de fluxo característico de um tipo de *layout* específico, categorizados da seguinte maneira:

- Módulo *flowline*: arranjo linear de máquinas no qual um movimento entre quaisquer duas operações ocorre sempre para frente. A peça pode utilizar todas as máquinas ao longo da linha ou pular algumas, se necessário (Figura 2.9a).
- Módulo *flowline* ramificado (convergente/divergente): semelhante ao módulo *flowline*, mas com peças que utilizam um subconjunto de operações comuns e outro específico. As peças se movem ao longo da linha e, em um determinado

ponto, desviam para um ramo com operações específicas, retornando à linha após o término dessas operações específicas (Figura 2.9b).

- Módulo celular: formado por um conjunto de máquinas diferentes capazes de processarem uma família de peças sem que qualquer peça necessite visitar qualquer departamento ou máquina fora da célula. Nesse caso, tanto as seqüências de processamento quanto as máquinas visitadas por todas as peças possuem um alto grau de similaridade (Figura 2.9c).
- Módulo centro de usinagem: formado por uma única máquina com alto grau de automação capaz de realizar diversas operações diferentes (Figura 2.9d).
- Módulo funcional: semelhante ao arranjo físico funcional tradicional, no qual o fluxo de material é aleatório (Figura 2.9e).
- Módulo fluxo padronizado: arranjo no qual existe um fluxo de material dominante e uma precedência entre as operações, definindo uma hierarquia entre elas. Este módulo pode ser decomposto em módulos *flowline* e *flowline* ramificado (Figura 2.9f).

Apesar de ser um tipo de arranjo físico recente, seus autores publicaram diversos artigos detalhando métodos e algoritmos para auxiliar em seu projeto, como os detalhados em Irani e Huang (1998) e Huang e Irani (1999).

Figura 2.9 – Tipos de Módulos (IRANI e HUANG, 1998)

2.3. Metodologias de Projeto de Arranjo Físico

Projetos de arranjo físico, em geral, são processos complexos que envolvem uma seqüência de atividades, muitas vezes organizadas em passos ou etapas, cujo objetivo final é a produção de um arranjo físico da fábrica. Por essa razão, diversos autores propuseram, ao longo das últimas 5 décadas, modelos para orientar o trabalho do projetista no processo de desenvolvimento desses projetos.

Nesta seção, são apresentados alguns desses modelos e discutida a utilização de softwares de apoio em algumas de suas diversas atividades.

2.3.1. O Modelo de Muther

O modelo de processo criado por Muther (1978), denominado *Systematic Layout Planning* (SLP), é um dos mais antigos e bem conhecidos modelos aplicados em projetos de arranjo físico. Segundo Muther (1978, p.7): “O sistema SLP é uma sistematização de projetos de arranjo físico. Consiste de (1) uma estrutura de fases, (2) de um modelo de procedimentos, e (3) de uma série de convenções para identificação, avaliação e visualização dos elementos e das áreas envolvidas no planejamento.”. A estrutura de fases é formada por quatro fases distintas, mostradas na Figura 2.10.

Figura 2.10 – Fases do sistema SLP (MUTHER, 1978, p.5)

Na fase I – **localização** – determina-se a localização da área onde será feito o planejamento das instalações.

Na fase II – **arranjo físico geral**, também conhecido com *layout* de blocos – estabelece-se a posição relativa entre as diversas áreas com base nos padrões de fluxo entre as atividades desempenhadas em cada uma dessas áreas.

Na fase III – **arranjo físico detalhado** – determina-se a localização de cada máquina e de cada equipamento na área definida.

Na fase IV – **implantação** – procura-se definir o planejamento de cada passo necessário à implantação do novo arranjo físico, obter sua aprovação, e fazer a movimentação física dos elementos que compõem o novo arranjo físico para seus devidos lugares.

As fases II e III constituem o projeto de arranjo físico. Em ambas, o modelo de procedimentos mostrado na Figura 2.11 é aplicado da mesma maneira. Na fase II, esse modelo de procedimentos é aplicado para obter o arranjo físico geral, enquanto que na fase III, ele é aplicado para detalhar cada área ou departamento definido no arranjo físico geral.

Figura 2.11 – Procedimentos do sistema SLP (MUTHER, 1978, p.7)

Os dados de entrada P, Q, R, S, T, e atividades significam: (P) produtos fabricados (material), (Q) quantidades (volume de fabricação), (R) roteiro de fabricação (seqüência do processo), (S) serviços de suporte, (T) tempos de fabricação e preparação, e atividades (ou áreas) a serem incluídas no arranjo físico.

Os dados P, Q e R combinados formam o **fluxo de material** (quadro 1 na Figura 2.11) que resulta numa tabela chamada de diagrama De-Para, como exemplificado pela Tabela 2.2. As linhas e colunas dessa tabela representam cada elemento do arranjo físico

(máquinas, equipamentos, departamentos, ou atividades) e os valores em seus cruzamentos são medidas quantitativas de fluxo de material entre esses elementos, como distância, custo de transporte, e número de viagens.

Tabela 2.2 – Exemplo de um Diagrama De-Para.

	Pintura	Almoxarifado	Soldagem	Fundição	Montagem
Pintura	*	30	70	0	20
Almoxarifado		*	0	15	20
Soldagem			*	0	30
Fundição				*	0
Montagem					*

Os dados P, Q e S combinados dão origem a tabela de **inter-relações entre atividades e serviços** (quadro 2 na Figura 2.11) que resulta numa tabela chamada de diagrama de relacionamento, como exemplificado na Figura 2.12. Cada linha representa uma entrada dessa tabela e contém um elemento do arranjo físico. As letras no cruzamento dessas entradas são medidas qualitativas que determinam a necessidade de proximidade ou não entre esses elementos.

Figura 2.12 – Exemplo de Diagrama de Relacionamento

Essas duas tabelas combinadas formam o **diagrama de inter-relações** (quadro 3 na Figura 2.11), que é o próprio diagrama de relacionamentos modificado de acordo com a intensidade do fluxo entre os departamentos. Este diagrama é combinado com o **espaço necessário** para cada unidade (quadro 4 na Figura 2.11) que é equilibrado de acordo com o **espaço disponível** (quadro 5 na Figura 2.11) na área selecionada. O resultado é o **diagrama de inter-relações de espaço** (quadro 6 na Figura 2.11) que é um diagrama 2D contendo a área de cada uma das unidades relacionadas entre si por meio de linhas, cuja quantidade indica o grau de proximidade que cada unidade deve ter. Esse diagrama já é um arranjo físico, mas necessita ser ajustado de acordo com todas as

condições de mudanças (quadro 7 na Figura 2.11), que consistem de métodos de movimentação, recursos de estocagem, necessidades de pessoal, serviços de suporte, etc.. Além disso, cada nova idéia deve levar em consideração várias **limitações práticas** (quadro 8 na Figura 2.11), como custos, segurança, normas, energia disponível, etc.. O resultado final desses procedimentos são dois ou mais planos (arranjos físicos) alternativos (planos X, Y e Z na Figura 2.11). Por fim, um desses planos é selecionado por meio de uma **avaliação** (quadro 9 na Figura 2.11) baseada em custos e valores intangíveis.

Por ser um método sistemático, o sistema SLP é fácil de ser automatizado por um software de apoio, principalmente as fases II e III. Além disso, ele utiliza poucos tipos de dados diferentes que são transformados em um único tipo de dado (diagrama de relacionamento) que é tomado como parâmetro de entrada pela maioria dos algoritmos de construção ou melhoria de *layouts* de blocos encontrados na literatura, como os apresentados na seção 2.2.3.

2.3.2. O Modelo de Apple

O modelo de processo proposto por Apple (1977) consiste de uma seqüência de vinte passos que podem ser seguidos independentemente do tipo de arranjo físico. A seqüência sugerida não necessita ser seguida sistematicamente, e, em muitos casos, pode ser modificada para melhor atender às necessidades do projeto. Apple coloca ainda que, em algumas situações, pode-se retornar a passos anteriormente concluídos para verificação ou modificação de algumas de suas partes não previstas durante o desenvolvimento. Os vinte passos do modelo são apresentados e discutidos a seguir:

1. **Obtenha os dados básicos.** Neste passo, é feito o levantamento dos dados necessário ao projeto do arranjo físico junto aos funcionários da empresa, como previsão de vendas, produtos e suas quantidades, política de estoque, desenhos das peças e listas de materiais, roteiros de produção, lista de operações, padrões de tempo, porcentagem de refugo, arranjos físicos existentes, desenhos dos prédios, dentre outros.
2. **Análise dos dados básicos.** Neste passo, é feita uma análise cuidadosa dos dados anteriormente obtidos para determinar seus inter-relacionamentos e prepará-los para os passos seguintes.

3. **Desenvolva o processo de produção.** Neste passo, é determinada a seqüência de processos e operações necessários para transformação do produto.
4. **Planeje o padrão de fluxo de materiais.** Neste passo, é feito o planejamento de um padrão de fluxo de material geral para assegurar movimentação mínima dos diversos componentes dos produtos.
5. **Estude um plano geral de movimentação de materiais.** Neste passo, são determinados os métodos genéricos de movimentação de material, desde o recebimento, passando pelas operações, até a expedição, baseada no padrão de fluxo anteriormente definido.
6. **Calcule as necessidades dos equipamentos.** Neste passo, são determinadas quantas unidades de cada tipo de equipamento serão necessárias.
7. **Planeje áreas individuais de trabalho.** Neste passo, são feitos o planejamento detalhado de cada operação, estação de trabalho, área, processo, etc., e são definidas as inter-relações entre máquinas, operações e equipamentos auxiliares (recursos).
8. **Selecione equipamentos específicos de movimentação de material.** Neste passo, são definidos os equipamentos específicos para movimentação de cada material.
9. **Coordene grupos de operações relacionadas.** Neste passo, é feita a integração de áreas individuais de trabalho, como centros de produção, estações de trabalho, departamentos ou processos, que ainda não foram coordenadas anteriormente (passos 4 e 7).
10. **Defina o relacionamento das atividades.** Os passos anteriores trabalharam atividades de produção. No entanto, existem outras atividades auxiliares e de serviços que não foram trabalhadas até este passo. Sendo assim, neste passo são relacionadas essas atividades auxiliares e de serviço com as atividades de produção.
11. **Determine os requisitos de armazenamento.** Nesse passo, é determinado o espaço necessário para o armazenamento de matéria prima, material em processo, produtos acabados, e ferramental.
12. **Planeje as atividades auxiliares e de serviços.** Os passos anteriores deram ênfase a determinação dos espaços para as atividades de produção sem haver qualquer preocupação com as atividades auxiliares e de serviços, como

restaurantes, banheiros, etc.. Sendo assim, este passo consiste no planejamento dessas atividades que serão integradas ao plano mestre.

- 13. Determine os requisitos de espaço.** Neste passo, é estimado o espaço necessário para cada atividade de produção, auxiliar, e de serviço, e o espaço total necessário para todo arranjo físico.
- 14. Aloque as áreas das atividades no espaço total disponível.** Neste passo, são mapeadas as áreas das atividades dentro do espaço definido para o arranjo físico, gerando um arranjo físico preliminar.
- 15. Considere os tipos de prédios.** Neste passo, são avaliados a forma, o número de andares, as limitações estruturais, dentre outros do(s) prédio(s) para evitar problemas com a distribuição do arranjo físico, como falta de espaço ou modificações no arranjo físico para ajustá-lo ao prédio.
- 16. Construa o arranjo físico mestre.** Neste passo, é feito um projeto detalhado do arranjo físico mostrando todas as áreas produtivas, auxiliares e de serviço. Esse trabalho pode ser feito manualmente (desenhos ou modelos em três dimensões) ou auxiliado por computador (CAD).
- 17. Avalie, ajuste e verifique o arranjo físico com o pessoal apropriado.** Neste passo, são feitas a avaliação final pelo projetista do arranjo físico e a avaliação da opinião das pessoas ligadas ao projeto.
- 18. Obtenha a aprovação.** Neste passo, é obtida a aprovação da gerência da fábrica ou das pessoas com autorização para isso.
- 19. Instale o arranjo físico.** Neste passo, é feita a movimentação dos equipamentos e máquinas para os locais determinados pelo projeto. A supervisão do projetista é importante, pois ele entende todos os detalhes do projeto. Os erros detectados nesse passo devem ser avaliados antes de uma decisão final ser tomada.
- 20. Acompanhe o desempenho do arranjo físico.** Mesmo que a instalação tenha sido feita como planejado, não há garantias de que tudo irá funcionar como previsto. Equipamentos não trabalham como deveriam, materiais não são movimentados com planejado, pessoas não seguem o método definido, dentre outros. Sendo assim, um acompanhamento e possíveis correções podem ser necessários.

O modelo de Apple é complexo e envolve muitas ferramentas gráficas, tabelas e uma grande variedade de tipos de dados diferentes que vão sendo transformados ao longo do processo. Assim como o sistema SLP, várias de suas atividades podem ser automatizadas por um software de apoio, principalmente as que envolvem análise de fluxos e padrões de movimentação, e construção do arranjo físico mestre e seu detalhamento. No entanto, como passos podem ser pulados, adaptados, ou refeitos, o software de apoio não poderá conduzir sistematicamente o trabalho do projetista. Nesse caso, o projetista deverá conhecer mais profundamente o modelo e o software de apoio deverá apenas fornecer as ferramentas (algoritmos e rotinas de preparação e conversão de dados) necessárias para o desenvolvimento das atividades, podendo, no máximo, indicar ou sugerir a sequência apropriada.

2.3.3. O Modelo de Slack

O modelo de processo proposto por Slack et al. (1997) é bem simplificado e consiste de três etapas principais: (1) seleção do tipo de processo, (2) seleção do tipo de arranjo físico, e (3) seleção do projeto detalhado do arranjo físico. A Figura 2.13 ilustra a sequência dessas etapas, chamadas de decisões.

Figura 2.13 – Sequência de decisões de arranjo físico (SLACK, 1997, p.212)

A primeira etapa – **seleção do tipo de processo** – baseia-se no volume-variedade dos produtos e nos objetivos estratégicos de desempenho para selecionar qual tipo de processo é o mais adequado para a fabricação desses produtos. Gráficos e tabelas comparativas, como os mostrados na Figura 2.14, podem ser utilizados nesta etapa para auxiliar na decisão.

A segunda etapa – **seleção do tipo de arranjo físico** – baseia-se no tipo de processo escolhido e, novamente, nos objetivos estratégicos de desempenho para selecionar qual tipo de arranjo físico é o mais adequado para a fabricação dos produtos. Novamente gráficos e tabelas comparativos, como os apresentados na seção 2.2.1, podem ser usados nesta etapa para auxiliar na decisão.

A terceira e última etapa – **seleção do projeto detalhado do arranjo físico** – baseia-se no tipo de arranjo físico escolhido para projetar o *layout* adequado. Diagramas de relacionamento, como os utilizados no modelo de Muther (Figura 2.12, p.21) podem ser utilizados nesta etapa para auxiliar na organização dos recursos.

Figura 2.14 – Gráfico de tipos de processos de manufatura (a) e tipos de processos de serviços (b) (SLACK, 1997, p.135 e 137)

O modelo de Slack é bastante simples e um software de apoio pode ser utilizado basicamente na etapa três para projetar o arranjo físico a ser escolhido, tomando como entrada diagramas de relacionamento. Atualmente, existem muitos softwares de apoio ao projeto de *layout* de blocos que partem de diagramas de relacionamento, alguns são discutidos na seção 2.4.

2.3.4. O Modelo de Tompkins

Tompkins et al. (1996) apresenta um modelo genérico composto por nove atividades agrupadas em três fases que pode ser adaptado para projetar uma planta industrial, como indústrias automotivas, siderúrgicas, e usinas, ou para projetar uma planta do setor de serviços, como hospitais, magazines, e restaurantes. A Figura 2.15 mostra esse modelo na forma de um processo genérico. As setas circundando o processo no sentido anti-horário significam que o processo é repetido diversas vezes para efetuar melhorias durante o tempo de vida do arranjo físico (ciclo de vida). Esse processo cíclico só termina quando o arranjo físico não for mais utilizado.

Figura 2.15 – Processo geral de projeto de planta (TOMPKINS, 1996, p.14)

O modelo de Tompkins é muito similar ao modelo de Slack, apresentado na seção 2.3.3, mas é complementado com outras atividades necessárias ao arranjo físico. Essas atividades, por sua vez, são similares às descritas nos modelos anteriores. Sendo assim, sua automatização por um software de apoio transcorre de maneira similar. No entanto, por se tratar de um modelo genérico que pode ser adaptado para um projeto de planta específico, o software de apoio deve ser capaz de oferecer ao projetista um ou mais desses modelos específicos, que podem ou não compartilhar as mesmas atividades e ferramentas, que serão disponibilizadas ao projetista à medida que este desenvolve um processo baseado no modelo específico.

2.3.5. O Modelo de Gonçalves Filho

O modelo de processo proposto por Gonçalves Filho (2001) é composto por cinco fases distintas divididas em atividades e sub-atividades que objetivam analisar necessidades de mudanças no arranjo físico, e planejar, projetar e efetuar essas mudanças considerando a possibilidade de combinação entre tipos de arranjo físico diferentes. A Figura 2.16 ilustra esse processo, cujas fases e atividades são descritos nos parágrafos seguintes.

Figura 2.16 – Modelo detalhado do processo de arranjo físico de fábrica (GONÇALVES FILHO, 2001, p.54)

Fase de preparação. Nesta fase, é avaliada a necessidade de mudança no arranjo físico e são definidos os objetivos a serem alcançados com essas mudanças, bem como definido o time de projeto e montado o plano de atividades para essas mudanças. Várias ferramentas são sugeridas para auxiliar nessas atividades, como QFD (*Quality Function Deployment*), e softwares de planejamento e gerenciamento de atividades ou gráficos de Gantt.

Fase de pré-projeto. Nesta fase, o time de projeto avalia o arranjo físico atual, colhe dados quantitativos para métrica do sistema atual que serão comparados com os dados do novo arranjo físico para verificar a eficiência das mudanças, e faz um estudo aprofundado sobre conceitos modernos de arranjo físico que podem ser adotados como solução. São sugeridas várias técnicas para avaliação do arranjo físico atual, dentre elas o uso do QFD e diagramas de causa e efeito, e um diagrama para escolha dos tipos de arranjo físico baseado no tipo de produto e no tipo de processo produtivo.

Fase de projeto. Nesta fase, o time de projeto deverá definir um arranjo físico que atenda aos requisitos definidos nas duas fases anteriores. Esta fase é dividida em várias

atividades e sub-atividades que iniciam com a obtenção, verificação e partição dos dados sobre a fábrica e os produtos, seguido pela definição dos componentes do arranjo físico a partir da análise P-Q-L e definição do arranjo físico interno de cada componente, e finalizadas com a definição do arranjo físico global dos componentes. Um componente do arranjo físico é uma possível configuração de máquinas, equipamentos e outros recursos para processar um ou mais tipos de produto, ou seja, um componente de arranjo físico pode ser definido com um tipo de arranjo físico, como os apresentados na seção 2.2. Os dados obtidos, como mencionado, estão relacionados à fábrica e aos produtos. A partição é feita sobre os produtos, caso haja grande variedade, para formar famílias de peças, utilizando tecnologia de grupo ou *features*. A análise P-Q-L (Produto e Quantidade e/ou Lucro) é similar a análise volume-variedade mostrada nas seções 2.2.1, e é feita sobre cada família de peça para determinar o componente de arranjo físico adequado. Para a definição do arranjo físico interno de cada componente são apresentadas várias técnicas, uma para cada tipo de componente. Por fim, a definição do arranjo físico global dos componentes é feita utilizando uma sequência de procedimentos similar à utilizada pelo sistema SLP para definição do arranjo físico geral.

Fase de validação. Nesta fase, o arranjo físico definido na etapa anterior é construído para efeito de visualização, avaliação e simulação. Várias técnicas podem ser empregadas para isso, como desenhos manuais do prédio, dos equipamentos, e das máquinas em escala, desenho do prédio, maquetes, sistemas CAD (2D e 3D), e realidade virtual (RV).

Fase de implementação. Está é a última fase do modelo e corresponde à movimentação e disposição das máquinas, equipamentos e serviços auxiliares de acordo com o projeto escolhido. As pessoas afetadas pelas mudanças são comunicadas para que tomem ciência da importância dessas mudanças e é feito um acompanhamento periódico para verificação do desempenho do novo arranjo físico.

O modelo de Gonçalves Filho, assim como o de Apple, é bastante complexo e várias de suas atividades podem ser automatizadas por um software de apoio, principalmente na fase de projeto. A idéia de definir componentes de arranjo físico e depois organizá-los em um arranjo físico geral, define uma subdivisão para o projeto em pequenos projetos (subprojetos) para definição do arranjo físico interno de cada componente, que depois são tomados como entrada para o projeto do arranjo físico

geral. Sendo assim, o software de apoio deve permitir a criação e execução controlada de cada subprojeto e do projeto maior, fornecendo as ferramentas necessárias à execução de cada atividade ou sub-atividade à medida que cada uma for sendo desenvolvida pelo projetista.

2.4. Sistemas Computacionais de Apoio ao Projeto de Arranjo Físico

Desde a década de 60, vários autores vêm propondo algoritmos e programas computacionais para auxiliar na análise e formação de arranjos físicos. Alguns deles são citados a seguir e representam apenas uma amostra dos inúmeros softwares comercializados e encontrados na literatura, mas exemplificam a diversidade em suas aplicações. Os softwares PFAST e SADCel, apresentados no final da seção, merecem destaque por se assemelharem ao software proposto.

TOMPKINS et al. (1996) cita alguns dos programas mais antigos implementados para otimizar *layouts* de bloco: CRAFT, MCRAFT, CORELAP e BLOCPLAN. Todos tomam como entrada de dados principal um diagrama de relacionamentos ou um diagrama de-para e adotam uma função objetivo baseada em custo ou distância para otimizar o arranjo físico. Eles podem ser utilizados tanto para a construção quanto para a melhoria de *layouts* de bloco. No entanto, como implementam apenas um algoritmo, podem ser utilizados apenas como ferramentas isoladas que dependem de outras ferramentas de análise para gerar seus dados iniciais.

Foulds (1997) apresenta o LayoutManager, um sistema de suporte à decisão que também pode ser aplicado para a construção ou melhoria de *layouts* de bloco. Como ferramenta de trabalho, ele se assemelha aos programas anteriores, toma como entrada principal um diagrama de relacionamento, e utiliza três métodos de otimização baseados em três critérios distintos: maximização da adjacência entre os departamentos, eficiência da adjacência dos departamentos, e tempo de transporte entre pares de departamentos. Apesar de seu diferencial, ele também é usado como uma ferramenta isolada que depende de outras ferramentas de análise para gerar seus dados iniciais.

Vosniakos, Ziaaie-Moayyed e Mamalis (1997) apresentam o projeto de um software que integra uma ferramenta CAD, usada para criar um modelo estático, a uma ferramenta de simulação inteligente que auxilia na construção e melhoria de arranjos físicos. Apesar de seu diferencial, ele considera o arranjo físico em sua fase de

detalhamento e necessita do apoio de outros softwares para efetuar as análises e gerar o arranjo físico.

Oh (1997) apresenta um sistema especialista para balanceamento de linha (ELBS) que utiliza um método heurístico baseado em um conjunto de regras para compor e simular arranjos físicos por produto. Este é um dos muitos casos de software desenvolvido para implementar um método específico e, portanto, possui um escopo de atuação limitado a projetos de arranjos físicos por produto.

Driscoll e Thilakawardana (2001) apresentam o pacote de software (A~Line) que inclui diversos algoritmos para efetuar análises baseadas no método de balanceamento de linha proposto por eles. Aparentemente, o software foi projetado para aceitar futuros algoritmos. No entanto, assim como o ELBS, está limitado à análise e formação de arranjos físicos por produto.

Luong et al. (2002) apresentam o projeto de um software para tomada de decisão na formação de células de manufatura que utiliza uma base de conhecimento para auxiliar o projetista na escolha do tipo de célula e em sua composição. Apesar de seu diferencial, ele implementa um método de análise e formação específico para células de manufatura e necessitaria diversas adaptações para funcionar com outro tipo de arranjo físico.

Balakrishnan, Cheng e Wong (2003) apresentam o FACOPT, utilizado para auxiliar na formação de *layouts* de bloco, que utiliza dois métodos heurísticos: *simulated annealing* e algoritmo genético. Este é outro caso de software desenvolvido com finalidades específicas.

Irani et al. (2000) apresentam o PFAST, um software que disponibiliza ao projetista um conjunto de ferramentas para a análise e otimização do fluxo de produção. Os autores apresentam uma proposta mais ampla que permite, a partir de dados obtidos das folhas de processo: (1) efetuar análise de agrupamento para formação de células de manufatura; (2) otimizar as rotas das peças dentro e fora das células; (3) e compor os *layouts* de blocos das mesmas, considerando inclusive corredores para transporte de material. Para cada uma dessas três atividades, o PFAST possui dois ou mais algoritmos que podem ser aplicados livremente. O PFAST é um software similar ao que está sendo proposto neste trabalho. No entanto, ele possui limitações quanto a sua aplicação, pois, trabalha com um tipo de arranjo físico específico – células de manufatura – e possui um conjunto limitado de algoritmos, os quais são disponibilizados livremente ao projetista

que deve conhecê-los para que sejam aplicados corretamente. Além disso, seus dados são mantidos em arquivos cuja organização é determinada pelo projetista, o que pode dificultar sua recuperação futura ou compartilhamento.

Tahara (2001) apresenta o SADCel, um software desenvolvido como uma aplicação de um *framework* criado para a integração de vários métodos para formação de células de manufatura. Sendo um sistema de apoio à decisão, ele define critérios associados aos diversos algoritmos que podem ser selecionados a partir do método escolhido e de um conjunto de restrição estabelecidas pelo projetista. Apesar de ser dedicado a um único tipo de arranjo físico, como o PFAST, ele apresenta algumas características que o aproximam da solução proposta neste trabalho. Primeira, ele pode possuir um grande número de algoritmos que serão disponibilizados ao projetista à medida que este necessita. Segunda, seus dados são armazenados em um banco de dados, o que facilita sua localização e recuperação. Terceira e última, seu *framework* foi construído para permitir fácil introdução de novos algoritmos e tipos de dados. No entanto, seu ambiente de trabalho baseado em cenários e seleção de algoritmos por restrição define uma forma linear de operação que limita o projetista a realizar uma análise de cada vez, e não foi prevista nenhuma forma de permitir comparar visualmente diversos resultados obtidos, isto é, poder colocar lado a lado os resultados obtidos em cada análise para verificar qual apresenta melhor resultado.

2.5. Considerações Finais

Os tópicos apresentados neste capítulo fornecem uma amostra da complexidade existente no desenvolvimento de projetos de arranjos físicos e caracterizam o domínio do problema a ser analisado. As principais conclusões obtidas durante esse processo de análise são descritas nos parágrafos a seguir juntamente com as dificuldades observadas no desenvolvimento de projetos de arranjo físico.

A primeira dificuldade observada reside na escolha e aplicação do modelo mais apropriado aos objetivos do projeto, assim como na escolha do arranjo físico que irá satisfazer esses objetivos. Um software de apoio capaz de efetuar análises do tipo P-Q ou P-Q-L poderia auxiliar na tomada dessas decisões.

A segunda dificuldade observada reside na escolha do método e dos algoritmos mais adequados para a análise e formação do tipo de arranjo físico escolhido, devido à extensa quantidade dos mesmos e as diversas diferenças em sua nomenclatura, função

objetivo, dados de entrada, etc.. Hassan (2000) apresenta várias dessas dificuldades e sugere um conjunto de normas para minimizá-las. No entanto, não há sinais evidentes de que isso irá ocorrer logo. Como alternativa, um software de apoio com um número razoável de métodos e algoritmos diferentes, organizados de maneira orientada ao processo ou tipo de arranjo físico, poderia auxiliar na tomada dessas decisões.

A terceira dificuldade observada reside na preparação dos dados. O volume de dados necessários para a análise varia de acordo com a quantidade de peças e produtos produzidos, de máquinas utilizadas nos processos de fabricação, dos roteiros de fabricação, dentre outros. No entanto, o tempo gasto para interpretá-los e gerar as tabelas e matrizes utilizadas, em geral, pelos métodos e algoritmos pode variar de alguns dias até meses. Com relação a esse problema, um software de apoio capaz de gerenciar os dados obtidos e oferecer rotinas de preparação e conversão de dados poderia auxiliar na aplicação mais rápida dos métodos e algoritmos, assim como na interpretação dos resultados.

A quarta dificuldade observada reside na escolha do resultado mais adequado (capaz de melhor satisfazer os objetivos do projeto). Cada método ou algoritmo possui uma função objetivo diferente e na maioria das vezes considera apenas parte do problema. Por exemplo, algoritmos de agrupamento, como o ROC e Coeficiente de Similaridade, consideram similaridades no uso de máquinas pelas peças, e auxiliam a determinar famílias de peças, mas não consideram o fluxo intra e intercelular. Isso não garante uma redução significativa no fluxo ou tempo de transporte, conduzindo a uma segunda análise para verificação do mesmo. Já, o algoritmo genético proposto por Gonçalves Filho (2001) trabalha com múltiplas funções objetivo e tenta efetuar o agrupamento considerando não só similaridades no uso de máquinas pelas peças, mas também o fluxo intra e intercelular. No entanto, isso não garante que a solução obtida pelo algoritmo genético seja mais adequada que a obtida pelo ROC ou Coeficiente de Similaridade. Uma alternativa para verificar isso, é o uso de softwares de simulação. Daí a importância de um software de apoio capaz de: (1) realizar comparações baseadas em critérios predeterminados; (2) permitir comparação visual dos dados com uma interface amigável; e (3) possuir meios de integração com softwares de simulação para avaliar os resultados obtidos.

A quinta e última dificuldade observada reside na escolha do próprio software de apoio. A escolha pode ser feita com base no tipo de arranjo físico e no objetivo a ser

atingido. Como a maioria dos softwares de apoio foca em um tipo de arranjo físico, método ou conjunto de algoritmos, a adoção, de um em particular pode limitar o espaço de solução desejado e não garantir que o arranjo físico formado irá atingir o objetivo definido. Por outro lado, as dificuldades em se adotar vários softwares residem na preparação dos dados, como descrito anteriormente, o que acaba desestimulando o desenvolvimento de vários planos alternativos. Liggett (2000) faz um levantamento sobre o problema de alocação de espaço para composição de *layouts* de bloco e aponta diversas falhas em softwares existentes, dentre elas a questão de um espaço de solução reduzido, uma vez que a maioria das técnicas utiliza funções objetivo simples. Isso reforça a idéia de se integrar em um único software de apoio um conjunto extenso de métodos e algoritmos (organizados de acordo com o tipo de arranjo físico, por exemplo) com um conjunto de rotinas de preparação e conversão de dados para auxiliar na análise e formação de diversos arranjos físicos alternativos. A integração desse software de apoio com um software de simulação pode auxiliar nos testes e na escolha do arranjo físico que mais atende aos objetivos estabelecidos. Da mesma maneira, a integração desse software de apoio com um software CAD pode auxiliar no detalhamento do arranjo físico.

CAPÍTULO 3. DESENVOLVIMENTO DE SOFTWARE ORIENTADO A OBJETO

3.1. Considerações Iniciais

Desde os anos 80, os conceitos de orientação a objeto (OO) vem sendo aplicados no desenvolvimento de software. Nos anos 90, a OO ganhou força impulsionada pelo grande volume de pesquisas e publicações a respeito de sua utilização e aplicação, e pelo interesse das empresas de desenvolvimento de software em suas vantagens e benefícios, tornando-se extensivamente utilizada na atualidade. Seus conceitos facilitam o desenvolvimento de software em todas as fases do processo, possibilitam mudanças relativamente rápidas e conduzem ao reuso. O reuso de componentes de software, por sua vez, conduz ao desenvolvimento mais rápido de softwares com melhor qualidade. Além disso, sistemas orientados a objeto são mais fáceis de manter, de adaptar, e de ampliar devido a sua estrutura desacoplada, necessidades estas muito desejadas pelos desenvolvedores de softwares corporativos cujas dimensões e complexidade demandam grande esforço de projeto. Mesmo propondo um novo paradigma para o desenvolvimento de software, a popularidade da OO vem aumentando rapidamente e a grande maioria dos sistemas computacionais atuais são orientados a objeto. Pressman (2002) atribui essa popularidade às vantagens oferecidas pela OO com relação às técnicas de desenvolvimento de software convencionais baseadas em processos e fluxos de dados.

Como a aplicação da OO conduz ao reuso, diversas técnicas, métodos e abordagens vêm sendo propostas para facilitar o desenvolvimento de software baseado em reuso. Dentre elas, pode-se destacar o desenvolvimento de padrões, como os apresentados por Gamma et al. (1995), o desenvolvimento de software baseado em componentes, descrito em Pressman (2002) e Sommerville (2001), e *frameworks* orientados a objeto, como os apresentados por Fayad, Schmidt e Johnson (1999), e

Fayad e Johnson (1999). Essas técnicas são muitas vezes aplicadas em conjunto e buscam formas cada vez mais eficientes de reuso.

Neste capítulo, serão discutidos esses tópicos e apresentados diversos conceitos, métodos e abordagens tecnológicas utilizadas no desenvolvimento do software proposto.

3.2. Princípios da Orientação a Objeto

Com a extensiva utilização da orientação a objeto nos últimos anos, seus conceitos e técnicas tornaram-se bem conhecidos e dispensam explicações detalhadas. No entanto, para dar melhor sustentação à abordagem tecnológica adotada para o desenvolvimento do software proposto, os conceitos mais relevantes a este trabalho são apresentados e comentados nesta seção.

A grande diferença entre uma abordagem convencional e uma orientada a objeto está na forma de visualização dos dados e processos que compõem o software. Na abordagem convencional, o software é visto como uma coleção de processos compostos por rotinas e sub-rotinas executadas de acordo com o fluxo dos dados dentro do sistema, enquanto que na abordagem orientada a objeto, essas rotinas e dados são encapsulados em uma única entidade chamada objeto.

Um **objeto** pode ser visto como uma entidade que possui um estado, representado por um conjunto de atributos, e um conjunto de operações que modificam este estado. Cada operação associada ao objeto fornece serviços a outros objetos que solicitam-nos quando algum processo computacional é requisitado. Os objetos são criados de acordo com as definições de uma **classe de objetos** que serve de modelo para a criação dos mesmos. Isso inclui as declarações de todos os atributos e operações que devem ser associados ao objeto dessa classe (SOMMERVILLE, 2001). Cada objeto criado durante a execução de um sistema computacional recebe uma identificação única que o distingue dos demais.

Um **software orientado a objeto** (sistema OO) é formado por um conjunto finito de objetos que se comunicam entre si através de mensagens. Cada mensagem é um sinal enviado de um objeto a outro solicitando um serviço. Esse serviço executa uma operação que utiliza os dados a seu alcance dentro do objeto e retorna uma mensagem contendo o resultado da operação ao objeto solicitante.

As definições acima mostram a visão diferenciada em um desenvolvimento orientado a objeto com relação ao desenvolvimento convencional. No entanto, o grande

diferencial está relacionado a três características fundamentais da OO conhecidas como encapsulamento, herança e polimorfismo.

O **encapsulamento** é inerente à definição de objeto e fornece diversos benefícios como: detalhes internos de implementação das operações são ocultados do mundo externo (isso possibilita modificações sem propagação de efeitos indesejáveis no sistema); as operações e dados são reunidas em uma única entidade (isso facilita o reuso de componentes); e as interfaces entre os objetos são simplificadas uma vez que um objeto não necessita conhecer detalhes da estrutura interna de outro objeto para enviar-lhe uma mensagem (isso favorece o acoplamento).

A **herança** é o mecanismo que possibilita a um objeto específico incorporar atributos, operações e relacionamentos de objetos genéricos sem repetir código, e definir novos atributos, operações e relacionamentos para si. Isso é feito em nível de classe e possibilita que classes especializadas (**subclasses**) compartilhem as mesmas características, isto é, tenham o mesmo ancestral comum (**superclasse**). É um dos diferenciadores-chave entre sistemas convencionais e sistemas OO, e também um dos conceitos mais importantes, pois possibilita forte encapsulamento, reutilização de código e fácil manutenção.

O **polimorfismo** é o mecanismo que possibilita a uma operação assumir vários comportamentos diferentes. É um dos conceitos mais poderosos associados ao mecanismo de herança e permite que uma operação definida em uma superclasse possa assumir o comportamento de uma operação definida em uma subclasse. Isso possibilita a um objeto da subclasse ser utilizado como se fosse um objeto da superclasse sem comprometer seu comportamento dentro do sistema.

Essas três características são peças-chave no desenvolvimento de sistemas OO e, quando bem empregadas, facilitam a manutenção e a modificação do mesmo, assim como a reutilização de seus componentes. Técnicas e métodos de reuso se valem dessas características para projetar padrões e *frameworks* que podem ser usados no desenvolvimento de sistemas OO, como será abordado nas seções seguintes.

Como um sistema OO é composto por vários objetos comunicantes, é conveniente mencionar que essa comunicação é feita apenas entre objetos relacionados. Tomando como base as definições da UML (BOOCH, RUMBAUGH e JACOBSON, 2000), um **relacionamento** é uma conexão semântica entre objetos que possibilita a troca de mensagens entre eles. Os mais utilizados são **dependência** (forma de relacionamento

em que modificações no objeto independente podem afetar o objeto dependente, mas não o contrário), **associação** (define uma conexão simples entre objetos), e **agregação** (ou todo/parte – especifica que um objeto toma outros como parte dele).

Uma outra forma de relacionamento é conhecida como **generalização** e caracteriza o mecanismo de herança. A generalização é a forma de relacionamento que permite aos objetos de classes especializadas (subclasses) serem utilizados como se fossem objetos da classe generalizada (superclasse).

Um último importante conceito relacionado à OO é a persistência dos objetos. **Persistência** é uma característica associada às propriedades de um objeto (valores de seus atributos) que determina se seus estados serão preservados após o término da execução do processo que o criou ou não. Objetos persistentes são recuperados e utilizados em uma nova execução do processo, enquanto objetos **transientes** são descartados após o término da execução desse processo.

3.3. Processo de Software

Processos de software são utilizados pelos engenheiros de software para controlar e coordenar projetos de desenvolvimento de software reais.

Segundo Sommerville (2001), um processo de software pode ser definido como um conjunto de atividades e resultados associados que conduzem à produção de um produto de software. Processos de software são complexos e dependem do julgamento humano como em qualquer processo intelectual. Por essa razão, existe uma grande diversidade de processos de software, nenhum ideal, desenvolvidos de maneiras diferentes por cada organização de acordo com suas necessidades. Muitas delas preferem utilizar processos *ad-hoc* em vez de utilizar algum processo padronizado. No entanto, em todo processo de software existem atividades fundamentais comuns:

1. Especificação do software: as funcionalidades e as restrições de operação do software são definidas.
2. Projeto e implementação do software: o software é concebido e codificado de acordo com as especificações.
3. Validação do software: o software é avaliado para verificar se está em conformidade com as necessidades do cliente.
4. Evolução do software: o software é modificado para atender as novas exigências do cliente.

Essas atividades definem o que é chamado de “ciclo de vida” do software e podem ser desenvolvidas de diferentes maneiras pelo engenheiro de software. A escolha da abordagem e da forma de desenvolvimento de cada atividade é uma importante tarefa designada ao engenheiro de software e depende de sua estratégia de desenvolvimento.

Ao longo dos anos, diversas tentativas para organizar o processo de software foram feitas, gerando diversos modelos de processo de software. Nesta seção, são apresentados alguns dos modelos de processo de software mais comuns encontrados na literatura e um processo para o desenvolvimento de software OO.

3.3.1. Modelos de Processo de Software

Segundo Sommerville (2001) e Pressman (2002) existem diversos modelos de processo de software. Os mais conhecidos são:

1. Modelo seqüencial linear ou modelo em cascata;
2. Modelos evolucionários: incremental e espiral; e
3. Modelo de desenvolvimento baseado em componentes.

O **modelo seqüencial linear** ou **modelo em cascata**, também conhecido como ciclo de vida clássico, é o modelo mais antigo e o mais amplamente utilizado. Ele sugere uma abordagem seqüencial de atividades para o desenvolvimento do software com ciclos de realimentação, isto é, pode-se retornar a qualquer atividade anteriormente executada e reiniciar o processo a partir dela. O processo é modelado segundo um ciclo convencional de engenharia e segue o seguinte fluxo de atividades: (1) análise e definição de requisitos, (2) projeto, (3) geração e teste de código, (4) implantação e teste do sistema, e (5) operação e manutenção. Apesar de sua grande utilização, ele apresenta alguns problemas, como sua inflexibilidade na partição do projeto em estágios distintos e “estados de bloqueios” nos quais alguns membros da equipe necessitam esperar que outros membros completem suas tarefas para que eles possam iniciar as suas. Isso pode levar a atrasos no projeto que comprometam a produtividade, o que costuma ser impraticável nos dias atuais, pois, a grande concorrência força a produção de software em um curto intervalo de tempo.

Modelos evolucionários são modelos interativos cujo objetivo é o refinamento sucessivo do software em direção a versões cada vez mais completas. Dois modelos são apresentados pelos autores anteriormente citados: (1) modelo incremental, e (2) modelo espiral.

O **modelo incremental** toma os elementos do modelo seqüencial linear e os aplica repetidamente. Ao final de cada interação, uma versão do software é produzida e avaliada pelo cliente e novos requisitos e definições são levantados para iniciar um novo ciclo. O processo é repetido até que o software esteja completo. Esse modelo oferece vantagens quando não há tempo ou mão de obra suficiente para se produzir um sistema completo. No entanto, como a cada incremento são adicionadas novas funcionalidades no sistema, se não houver um controle rigoroso, o projeto pode ser comprometido e gerar diversos desconfortos ao cliente, além de elevar seu custo e gerar atrasos.

O **modelo espiral** combina os aspectos controlados e sistemáticos do modelo seqüencial linear com aspectos de prototipagem, possibilitando o desenvolvimento rápido de versões incrementais de software. Suas atividades são executadas seqüencialmente na forma de uma espiral, partindo de seu centro e girando no sentido horário (Figura 3.1). A cada volta completa, uma nova interação é iniciada e um novo ciclo de tarefas é executado. A espiral é segmentada em regiões (de três a seis) seguindo a seguinte ordem:

1. Comunicação com o cliente: envolve atividades e tarefas para estabelecer a comunicação entre o desenvolvedor e o cliente.
2. Planejamento: envolve atividades e tarefas necessárias para definir recursos, prazos e outras informações relacionadas ao projeto.
3. Análise de risco: envolve atividades e tarefas para avaliar os riscos, tanto técnicos quanto gerenciais.
4. Engenharia: envolve atividades e tarefas para construir uma ou mais representações da aplicação.
5. Construção e liberação: envolve atividades e tarefas necessárias para construir, testar, instalar e fornecer apoio ao usuário.
6. Avaliação pelo cliente: envolve atividades e tarefas para obter realimentação do cliente, com base na avaliação das representações do software criadas durante o estágio de engenharia e implementas durante o estágio de instalação.

As voltas mais internas da espiral são dedicadas ao projeto de desenvolvimento conceitual do produto, cujo produto final pode ser um primeiro protótipo ou apenas a modelagem do sistema. As próximas voltas são dedicadas ao projeto de desenvolvimento do produto cujos resultados são versões cada vez mais complexas do

sistema. As voltas seguintes são dedicadas ao projeto de melhoria do produto, cujo objetivo é efetuar ajustes finais no sistema e concluí-lo. Por fim, as demais voltas são dedicadas ao projeto de manutenção e evolução do produto. O início de cada projeto é marcado na espiral e nomeado ponto de entrada do projeto. Como se trata de uma espiral, normalmente o ponto de entrada de um projeto é o término de outro. Sendo assim, um eixo é traçado na espiral determinando o local em que os pontos de entrada serão marcados.

Figura 3.1 – Modelo Espiral (PRESSMAN, 2002, p.35)

O modelo espiral é uma boa abordagem para desenvolvimento de sistema de grande porte e seu processo evolucionário facilita na compreensão das necessidades do cliente e na compreensão do domínio do problema. No entanto, a aplicação dessa abordagem deve levar em consideração os riscos técnicos em todos os estágios do projeto, pois, se um risco importante não for descoberto e gerenciado, fatalmente ocorrerão problemas que podem até comprometer todo o projeto. Apesar de suas vantagens, o modelo espiral ainda não é tão amplamente usado como os citados anteriormente.

O **modelo de desenvolvimento baseado em componentes** (PRESSMAN, 2002) parte do princípio de reuso e busca desenvolver um sistema a partir de componentes previamente desenvolvidos. Caracterizado naturalmente por um processo evolucionário, necessita de uma abordagem interativa, o que lhe confere muitas das características do modelo espiral. Como o desenvolvimento baseado em componentes conduz ao reuso, a aplicação desse método tende a reduzir prazos de desenvolvimento, custos de projeto,

riscos técnicos, e gera, conseqüentemente, um aumento na produtividade, quando comparado com a aplicação de outros modelos mais tradicionais. A diferença básica na aplicação desse modelo, como mostrado na Figura 3.2, é que na fase de engenharia do produto os componentes identificados no processo de análise dos requisitos são pesquisados em uma biblioteca. Os componentes encontrados são extraídos, enquanto os demais são desenvolvidos, utilizando o processo de desenvolvimento orientado a objeto mais adequado, e adicionados à biblioteca para reuso. Os componentes, então, são aplicados na construção ou melhoria do sistema em sua interação atual.

Figura 3.2 – Modelo de Desenvolvimento Baseado em Componentes (PRESSMAN, 2002, p.40)

3.3.2. Modelo de Processo Orientado a Objeto

Segundo Pressman (2002) qualquer modelo de processo de software apresentado na seção anterior pode ser facilmente adaptado para ser utilizado com a OO. No entanto, devido as suas características, um sistema OO tende a evoluir com o tempo e seus componentes podem ser utilizados na construção de outros sistemas (reuso). Sendo assim, o modelo apontado pelo autor para a engenharia de software orientada a objeto é o modelo de desenvolvimento baseado em componentes. A Figura 3.3 mostra esse modelo adaptado às necessidades da OO.

O processo OO mostrado na Figura 3.3 inicia-se com a comunicação com o cliente e move-se ao longo da espiral evolutiva. Durante essa conversa com o cliente, o domínio do problema é definido e suas classes básicas são identificadas. O planejamento e a análise de riscos estabelecem a base para o plano do projeto OO com

suas metas, objetivos, prazos e aspectos técnicos relevantes. Na região de engenharia, o trabalho iterativo (caixa cinza) enfatiza o reuso. Sendo assim, as classes identificadas são procuradas numa biblioteca de classes OO existente. As classes não encontradas serão então desenvolvidas aplicando-se a análise, o projeto, a programação, e o teste orientados a objeto. As novas classes são então colocadas na biblioteca para que possam ser reutilizadas no futuro. O software é construído, testado e entregue para a avaliação do cliente, reiniciando o ciclo.

Figura 3.3 – Modelo de Processo Orientado a Objeto (PRESSMAN, 2002, p.533)

3.4. Métodos Orientados a Objeto

No final dos anos 80 e durante os anos 90, vários autores publicaram métodos para análise orientada a objeto (OOA – *Object-Oriented Analysis*) e projeto orientado a objeto (OOD – *Object-Oriented Design*). OMT (*Object Modeling Technique*) (RUMBAUGH, 1991), OOSE (*Object-Oriented Software Engineering*) (JACOBSON, 1992), Coad e Yourdon (COAD e YOURDON, 1992, 1993), Booch (BOOCH, 1994), e Fusion (COLEMAN et al., 1996), são bem conhecidos e representam uma pequena amostra desses métodos. Apesar das diversas diferenças em suas terminologias e passos do processo, esses métodos abordam o processo de análise e projeto de maneira semelhante (PRESSMAN, 2002).

Cada um dos métodos apresentados possui vantagens e desvantagens com relação à abordagem, à quantidade de ferramentas gráficas e textuais diferentes que utilizam, e às limitações semânticas de seus modelos. Além disso, a transição da OOA para a OOD

nem sempre é tranqüila e, em geral, não há uma preocupação específica com a codificação do sistema, isto é, a transição da OOD para a OOP (*Object-Oriented Programming* – programação orientada a objeto) nem sempre é bem definida. Por fim, a natureza sistemática de alguns deles, limita o processo de desenvolvimento evolucionário e iterativo.

Numa tentativa de definir uma forma padronizada e unificada de modelagem para resolver vários dos problemas citados, Grandy Booch, James Rumbaugh e Ivar Jacobson combinaram as melhores características de seus métodos individuais e criaram a UML (*Unified Modeling Language*) (BOOCH, RUMBAUGH e JACOBSON, 2000).

A UML é uma linguagem de modelagem visual que utiliza vários tipos de diagramas para auxiliar o analista e o projetista a documentar parte ou todo o processo de software. Cada diagrama é uma apresentação gráfica de uma coleção de elementos de modelagem (símbolos gráficos), freqüentemente relacionados por arcos e vértices (relacionamentos), que ilustram partes distintas do software. Alguns desses diagramas são: diagrama de caso de uso, diagrama de classe, diagrama de seqüência, diagrama de colaboração, e diagrama de implantação. Nem todos os diagramas necessitam ser utilizados no desenvolvimento de um software, cabe ao responsável por cada etapa ou atividade escolher os diagramas a serem utilizados.

A UML possui também vários mecanismos que podem ser utilizados para facilitar a compreensão dos diagramas, como estereótipos, restrições e valores atribuídos. Esses mecanismos também podem ser utilizados para estender a sintaxe e a semântica da linguagem, possibilitando a criação de novos elementos de modelagem e notações.

A grande vantagem da UML com relação a outros métodos é que ela não é um método em si, mas sim, uma linguagem para representação de um sistema e pode ser aplicada de maneira independente em todas as fases do processo de software. Por se tratar de uma linguagem, a transição entre as fases de análise, projeto e codificação é natural, rápida, complementar e sem ambigüidade.

3.5. Processo de Software Utilizando a UML

Como mencionado na seção anterior, a UML é independente do processo de software e não se limita a um ciclo de vida. Sendo assim, seus digramas podem ser aplicados em quase todas as partes do processo para modelagem do sistema e documentação de suas atividades. Sua aplicação mais freqüente é nos processos de OOA

e OOD, por tratarem da concepção do sistema. No entanto, técnicas de cenários baseadas em casos de uso combinados com diagramas de seqüência, têm sido aplicadas para o levantamento de requisitos (PRESSMAN, 2002; SOMMERVILLE, 2001).

Em UML, um sistema é representado por cinco “visões” diferentes, cada qual definida por um conjunto de diagramas:

1. Visão de caso de uso: representa o sistema a partir da perspectiva do usuário utilizando diagramas de caso de uso. Ela não especifica a organização do sistema como um software, mas sim as forças que determinam a forma arquitetural do sistema. Os aspectos estáticos dessa visão são capturados nos diagramas de caso de uso, e os aspectos dinâmicos são capturados em diagrama de interação, gráfico de estados e diagramas de atividades.
2. Visão de projeto: abrange as classes, interfaces e colaborações que definem o sistema como software. Os aspectos estáticos dessa visão são capturados nos diagramas de classes e de objetos, e os aspectos dinâmicos são capturados em diagrama de interação, gráfico de estados e diagramas de atividades.
3. Visão de processo: representa os aspectos dinâmicos ou comportamentais do sistema. Os aspectos estáticos e dinâmicos dessa visão são capturados nos mesmos tipos de diagramas da visão de projeto, mas com foco nas classes ativas que representam cada processo.
4. Visão de implementação: representa os componentes e arquivos utilizados para a montagem e fornecimento do sistema físico. Os aspectos estáticos dessa visão são capturados nos diagramas de componentes, e os aspectos dinâmicos são capturados em diagrama de interação, gráfico de estados e diagramas de atividades.
5. Visão de implantação: representa o ambiente em que o sistema é executado (hardware e topologia). Os aspectos estáticos dessa visão são capturados nos diagramas de implantação, e os aspectos dinâmicos são capturados em diagrama de interação, gráfico de estados e diagramas de atividades.

Essas visões podem ser tratadas isoladamente, permitindo que participantes do processo de software tratem aspectos específicos do sistema. Sendo assim, a OOA baseada na UML costuma concentrar-se nas visões de caso de uso e de projeto, enquanto que OOD baseada na UML costuma tratar as visões restantes (PRESSMAN, 2002).

3.6. Engenharia de Software Baseado em Componentes

A engenharia de software baseada em componentes (CBSE – *Component-Based Software Engineering*) é um processo que enfatiza o projeto e a construção de sistemas de computador complexos a partir de componentes de software reusáveis. Para isso, ela possui duas atividades de engenharia que são conduzidas em paralelo e trabalham de forma complementar: engenharia de domínio e desenvolvimento baseado em componentes. Um modelo de processo para a CBSE, proposto por Pressman (2002), é ilustrado na Figura 3.4 e mostra como essas duas atividades de engenharia são desenvolvidas.

Figura 3.4 – Modelo de processo para a CBSE (PRESSMAN, 2002, p.707)

A **engenharia de domínio** é aplicada em domínios de aplicações para identificar componentes funcionais, comportamentais e de dados que possam ser reusados, e registrá-los em uma biblioteca de reuso. Ela é conduzida através de três atividades principais:

1. Análise de domínio: consiste da investigação de um domínio de aplicação e de amostras de aplicações desenvolvidas para esse domínio em busca de padrões funcionais e comportamentais que possuam características comuns. A partir disso, são construídos um modelo de domínio, uma arquitetura de software e um modelo estrutural.

2. Construção de domínio: consiste do desenvolvimento dos componentes reusáveis identificados durante a análise de domínio. Esses componentes são desenvolvidos com base no modelo estrutural e na arquitetura de software.
3. Disseminação de domínio: consiste na documentação e registro dos componentes reusáveis em uma biblioteca de reuso para que possam ser utilizados futuramente no desenvolvimento de novas aplicações.

O **desenvolvimento baseado em componentes** é aplicado para a análise, projeto e construção de sistemas utilizando os componentes desenvolvidos pela engenharia de domínio. Seu processo é conduzido de acordo com o modelo de desenvolvimento baseado em componentes descrito na seção 3.3.1, e consiste das seguintes atividades básicas mostradas na Figura 3.4: análise, projeto arquitetural, qualificação de componentes, adaptação de componentes, composição de componentes, engenharia de componentes, teste, e entrega do aplicativo.

A atividade de **análise** consiste de técnicas convencionais de levantamento de requisitos e definição do domínio do problema, buscando caracterizá-lo de acordo com um ou mais modelos de domínio existente. Com essas informações, dá-se início à atividade de **projeto arquitetural**, cujo objetivo é definir o estilo arquitetural mais adequado à criação do aplicativo.

Uma vez definida a arquitetura, a atividade de **qualificação de componentes** busca determinar os componentes mais adequados para preencher a estrutura dessa arquitetura. Esses componentes podem (1) existir na biblioteca de reuso e/ou (2) serem construídos por meio da engenharia de componentes.

A atividade de **adaptação de componentes** é executada quando existe conflito entre os componentes definidos para a aplicação e as características dos componentes existentes. Nesse caso, é verificada a possibilidade de modificar características do componente definido para a aplicação de forma que este se encaixe com os existentes. Se isso não for possível, é verificada a possibilidade de se modificar os componentes existentes ou criar novos componentes.

A atividade de **engenharia de componentes** é executada quando novos componentes necessitam ser criados ou componentes existentes necessitam ser modificados. Nesse caso, é executado um processo similar ao da engenharia de domínio para que esses novos componentes sejam desenvolvidos, classificados e registrados na biblioteca de reuso.

À medida que os componentes são disponibilizados, a atividade de **composição de componentes** os utiliza para compor a arquitetura estabelecida para a aplicação. Para isso, são usados modelos infra-estruturais (padrões de projeto ou *frameworks* de aplicação) que determinam como unir esses componentes. Esses modelos normalmente são artefatos de software registrados na biblioteca de reuso.

Construído a aplicação, este é testado e entregue ao cliente para avaliação. Como esse processo é interativo e evolutivo, a cada nova interação, uma nova versão da aplicação é entregue ao cliente para avaliação. Isso se repete até que a aplicação esteja completa.

Várias são as tecnologias aplicadas à CBSE para o desenvolvimento de software. Dentre elas, três são mais conhecidas:

1. Componentes reusáveis que são seus “blocos de construção”;
2. Padrões de projeto que são artefatos usados durante as fases de projeto; e
3. Frameworks de aplicação que são utilizados como modelos infra-estruturais.

Componentes reusáveis são produtos indiretos da aplicação do processo de CBSE e podem ser considerados seus “blocos de construção”. Na literatura, várias definições podem ser encontradas para componentes. No entanto, todas convergem para o mesmo ponto fundamental: componentes são partes independentes e substituíveis de um software. Sendo assim, um componente pode ser uma função, um módulo, uma classe ou um subsistema em código fonte, cujos arquivos são compilados juntamente com os arquivos da aplicação, ou, como vêm sendo utilizados mais recentemente, podem ser partes independentes de um software, na forma compilada, chamadas de **componentes binários** (PARRISH e DIXON e CORDES, 2001). Várias tecnologias são empregadas no desenvolvimento e uso de pacote de componentes binários, como DLL (*Dynamic Link Library*), COM (*Component Object Model*), CORBA (*Common Object Request Broker Architecture*) e JavaBeans.

A grande vantagem em se utilizar componentes no desenvolvimento de software é a redução no tempo, nos riscos, e nos custos desse desenvolvimento. No entanto, à medida que o número de componentes cresce em uma biblioteca de reuso, achar o componente mais adequado se torna uma tarefa difícil. Além disso, o gerenciamento e manutenção de uma biblioteca de reuso eficiente pode gerar custos indesejáveis à empresa.

Padrões de projeto (GAMMA et al., 1995) são estruturas que ocorrem diversas vezes dentro de seu contexto. Em outras palavras, um padrão de projeto especifica a estrutura e o comportamento de um conjunto de classes em um projeto de software. Em geral, podem ser identificados em um projeto de software dois ou mais padrões de projeto.

Padrões de projeto auxiliam na especificação, visualização, compreensão, construção e documentação de artefatos de um sistema complexo de software que facilitam a atividade de projeto arquitetural da CBSE e conseqüentemente na atividade de qualificação de componentes, agilizando o processo de desenvolvimento de software. Apesar disso, a identificação de padrões de projeto não é uma atividade simples e demanda muita experiência do engenheiro de software. Além disso, à medida que o número de padrões identificados cresce, a tarefa de selecionar o padrão mais adequado torna-se tão difícil quanto a tarefa de pesquisar componentes em uma biblioteca de reuso.

Frameworks de aplicação (FAYAD e SCHMIDT e JOHNSON, 1999) são aplicações reusáveis e semi-completas que podem ser especializadas para produzir uma aplicação personalizada. Eles são projetos reusáveis de sistemas computacionais que descrevem como esses sistemas são decompostos em um conjunto de objetos comunicantes e o controle de fluxo entre eles.

Frameworks de aplicação são referenciados como uma tecnologia promissora para auxiliar na implementação e no projeto de sistemas complexos, possibilitando a redução de custos e melhoria da qualidade de software. Eles são construídos para desenvolver aplicações em um domínio específico e caracterizam uma família de aplicações (SUMMMERVILLE, 2001).

A vantagem de um *framework* sobre as duas tecnologias anteriores é que ele fornece simultaneamente ao engenheiro de software o modelo arquitetural com seus padrões e estruturas, e uma implementação dessas estruturas composta por componentes genéricos (classes abstratas) que definem as interfaces de acoplamento e controle usadas pelo programa principal para definir o fluxo de controle da aplicação. O engenheiro de software necessita apenas definir as classes específicas (subclasses) necessárias a sua aplicação e conectá-las ao *framework*. Apesar dessas facilidades, sua complexidade e o tempo que leva para aprender a utilizá-lo têm dificultado seu uso extensivo. Além disso,

seu desenvolvimento é lento e depende de especialistas, o que pode gerar custos elevados e indesejáveis em um primeiro momento.

3.7. Desenvolvimento de Software Baseado em *Frameworks* de Aplicação

Frameworks de Aplicação, como descrito na seção anterior, são apontados como uma das tecnologias de reuso promissoras para auxiliar no projeto e na implementação de sistemas complexos (FAYAD e SCHMIDT e JOHNSON, 1999).

Nesta seção, são apresentados uma visão geral e diversos conceitos inerentes a *frameworks* de aplicação. Também é sugerido um processo de software para desenvolvimento de *frameworks* e de suas aplicações, o qual foi adotado para o desenvolvimento do software proposto.

3.7.1. *Frameworks* de Aplicação Orientados a Objeto (OOAF)

Um *Framework* de Aplicação Orientado a Objeto (OOAF – *Object-Oriented Application Framework*) define um conjunto de classes abstratas ou superclasses, suas interfaces básicas e seus relacionamentos, pré-definindo um padrão de comportamento para o software, assim como um programa principal que determina o controle de fluxo da aplicação (inversão de controle). Como classes abstratas não podem possuir instâncias, elas servem de modelo para criação das subclasses específicas para cada aplicação (classes concretas). Cada subclasse deve então definir o corpo para cada operação (método abstrato) e assim determinar o comportamento específico de cada objeto. Essa forma de construção utiliza as vantagens oferecidas por três características da OO: abstração de dados (ou encapsulamento), polimorfismo e herança, como apresentados na seção 3.2.

As classes abstratas compõem a parte variante (ou flexível) do OOAF e são chamadas de **pontos variáveis** (*hot spots*). Além delas, o OOAF também é composto por outras classes que não mudam e são usadas na implementação de seu núcleo (*kernel*) chamadas de **pontos fixos** (*frozen spots*) (MARKIEWICZ e LUCENA, 2001).

Inversão de código é uma das características que diferenciam o desenvolvimento baseado em *framework* de um baseado em componentes. Em um desenvolvimento baseado em componentes, o desenvolvedor toma um conjunto de componentes de uma biblioteca de reuso e escreve um programa principal que chama cada componente

quando este é necessário. Com isso, o desenvolvedor decide quando cada componente será chamado e é responsável pela estrutura e fluxo de controle do programa. Em um *framework*, o programa principal é reusado e cabe ao desenvolvedor apenas decidir quais componentes serão conectados a ele. O código do desenvolvedor é então chamado pelo *framework* que determina a estrutura e o fluxo de controle do programa.

Como inicialmente é difícil conhecer todo o domínio da aplicação, a primeira versão do *framework* é construída para poder implementar exemplos e costuma ser um *framework* caixa-branca. Feito isso, ele começa a ser utilizado no desenvolvimento de aplicações. À medida que vai sendo utilizado, o *framework* vai sendo melhorado sistematicamente até ser transformado em um *framework* caixa-preta.

Frameworks caixa-branca são chamados assim porque seus pontos variáveis são implementados por herança, isto é, o desenvolvedor cria classes concretas derivadas das classes abstratas do *framework* para implementar a aplicação. Já, **frameworks caixa-preta** são chamados assim porque seus pontos variáveis são implementados por composição, isto é, o desenvolvedor seleciona componentes dentre as diversas classes concretas existentes e os “encaixa” no *framework* para implementar a aplicação. *Frameworks* caixa-preta são pouco flexíveis, mas mais fáceis de serem utilizados, uma vez que o desenvolvedor necessita apenas conhecer as interfaces de seus pontos variáveis e selecionar os componentes mais adequados a sua aplicação. No entanto, *frameworks* caixa-branca são mais flexíveis e mais poderosos nas mãos de um especialista, mas mais difíceis de serem utilizados. Como um *framework* é transformado de caixa-branca para caixa-preta à medida que evolui, eles são tratados não como uma dicotomia, mas como extremos de um espectro. Os *frameworks* intermediários desse espectro são chamados de caixa-cinza. **Frameworks caixa-cinza** são aqueles cujos pontos variáveis são implementados por herança e por composição, dependendo das necessidades da aplicação.

Além da forma de classificação de *frameworks* em caixa-branca, caixa-preta, ou caixa-cinza, Fayad, Schmidt e Johnson (1999) apresentam três outras formas de classificação considerando seus escopos:

1. *Frameworks de infra-estrutura de sistemas*: auxiliam no desenvolvimento da infra-estrutura de sistemas, como sistemas operacionais, interfaces com o usuário e compiladores. Normalmente são usados internamente na organização do software e não são vendidos separadamente.

2. Frameworks de integração de middleware: consistem de um conjunto de classes de objetos padronizadas que suportam a comunicação de componentes e a troca de informações. Eles estendem a infra-estrutura de um software para que este possa operar em um ambiente distribuído. Exemplos desse tipo de *framework* são CORBA, COM e DCOM da Microsoft, Java Beans e bancos de dados transacionais.
3. Frameworks de aplicação empresarial: são desenvolvidos para um domínio de aplicações, como telecomunicações, sistemas financeiros, aviação e manufatura. Eles incorporam o conhecimento sobre o domínio da aplicação e auxiliam na criação de aplicações finais voltadas para o usuário, e podem utilizar internamente *frameworks* de infra-estrutura e/ou de integração *middleware*.

Uma aplicação pode utilizar vários *frameworks* integrados e ser desenvolvida de diversas maneiras. Sua composição básica consiste de três partes: o *framework*, as subclasses concretas das classes abstratas do *framework*, e algo mais (FAYAD, SCHMIDT e JOHNSON, 1999). Algo mais costuma incluir um roteiro (*script*) que especifica quais subclasses concretas serão usadas e como elas serão interconectadas. Ele também pode incluir objetos que não têm relacionamentos com o *framework* ou que usam um ou mais objetos do *framework*, mas que não são chamados por eles. Objetos chamados por objetos do *framework* devem participar do modelo colaborativo do *framework*, pois são considerados parte dele.

A forma mais simples de usar um *framework* para desenvolver uma aplicação é conectando componentes existentes. Isso é semelhante à construção de uma casa de brinquedos com Lego® ou uma placa de circuito com componentes eletrônicos. Os programadores devem saber apenas que objetos do tipo A são conectados a objetos do tipo B, sem necessitar conhecer a especificação exata de A e B. No entanto, nem sempre isso é possível. Algumas vezes é necessário construir novas subclasses concretas do *framework* para desenvolver uma nova aplicação. Isso requer mais conhecimento sobre as classes abstratas que na forma anterior. Sendo assim, o programador deve entender detalhadamente das interfaces do *framework*.

Outra forma de usar um *framework* é modificando algumas de suas classes abstratas para adequá-lo às necessidades da nova aplicação. Isso requer um grande conhecimento sobre o *framework* e é considerada a forma mais difícil de uso. Apesar

disso, ela é também a forma mais poderosa de uso quando colocada nas mãos de especialistas. Por outro lado, modificações nas classes abstratas podem danificar classes concretas existentes, não sendo a forma mais adequada de uso para *frameworks* empregados na construção de sistemas abertos.

A primeira forma usa *frameworks* caixa-preta para desenvolver aplicações, pois não é necessário conhecer detalhes de sua implementação. A segunda forma usa *frameworks* caixa-cinza, pois requer conhecimento mais detalhado da implementação do *framework* para desenvolver novos pontos variáveis. A terceira forma usa *frameworks* caixa-branca, pois requer muitos detalhes da implementação do *framework*.

Apesar de suas vantagens, OOAF's são tão difíceis de desenvolver quanto softwares complexos e requerem habilidades que só um especialista possui. Seu aprendizado não é rápido e costuma levar de 6 a 12 meses até que um desenvolvedor possa utilizá-lo intensivamente e eficientemente. Sua manutenção é esporádica, mas normalmente as mudanças são feitas muito mais em partes funcionais do que em partes não funcionais, o que requer um conhecimento profundo sobre o *framework* para que essas mudanças não invalidem componentes existentes ou prejudiquem sistemas em operação. Fayad, Schmidt e Johnson (1999) apresentam algumas alternativas para contornar esses problemas sugerindo a criação de uma documentação adequada e um número razoável de exemplos.

3.7.2. Processo de Desenvolvimento de OOAF's

Markiewicz e Lucena (2001) comentam que o desenvolvimento de *frameworks* é composto por três grandes estágios: análise de domínio, projeto do *framework* e instanciação do *framework*.

A **análise de domínio** tenta descobrir os requisitos de um domínio, bem como seus possíveis requisitos futuros. Durante este estágio, os pontos variáveis e fixos do *framework* são parcialmente definidos.

O **projeto do *framework*** define as abstrações do *framework* e modela seus pontos variáveis e fixos (às vezes utilizando diagramas UML), prevendo a extensibilidade e a flexibilidade propostas na análise de domínio. Neste estágio, padrões de projeto são usados para auxiliar na modelagem.

A **instanciação do *framework*** consiste da implementação dos pontos variáveis do *framework* que são encaixados no mesmo para gerar uma aplicação. Os pontos fixos do *framework*, por não serem alterados, são comuns a todas as aplicações geradas.

Segundo Fayad, Schmidt e Johnson (1999), o processo de desenvolvimento de um OOAF é semelhante ao de qualquer processo de reuso e visa construí-lo iterativamente. Bosch (1999) identifica três fases distintas no processo de desenvolvimento de um *framework*: fase de desenvolvimento do *framework*, fase de uso do *framework*, e fase de evolução e manutenção do *framework*. As atividades envolvidas em cada uma dessas fases assemelham-se as descritas na seção 3.6, apenas sendo orientadas ao desenvolvimento de *frameworks*.

Sendo assim, a Engenharia de Domínio do processo de CBSE descrita na seção 3.6 pode ser adaptada para o desenvolvimento de *frameworks* (análise de domínio e projeto do *framework*), e o Desenvolvimento Baseado em Componentes pode ser adaptado para desenvolver as aplicações (instanciação do *framework*).

Nesse processo, a engenharia de domínio desenvolve tanto o *framework* quanto alguns componentes para seus pontos variáveis que podem ser tomados como exemplos ou usados no desenvolvimento de aplicações. A biblioteca de reuso mantém o *framework*, seus componentes e seus artefatos. Essas informações são então utilizadas pela engenharia de desenvolvimento baseada em componentes que qualifica e adapta componentes para encaixá-los nos pontos variáveis do *framework*, podendo, se necessário, modificá-los.

3.8. Banco de Dados

Bancos de dados e sistemas de bancos de dados são tecnologias antigas que datam do final da década de 60 e evoluíram sistematicamente ao longo dos últimos 30 anos. Nesse período, várias formas de armazenamento e organização de dados foram pesquisadas e utilizadas, como o sistema hierárquico e o modelo em rede. No entanto, um dos modelos mais bem aceitos e difundidos na atualidade é o modelo relacional.

Bancos de dados baseados no modelo relacional, conhecidos como bancos de dados relacionais (BDR), surgiram na década de 70, sendo a primeira implementação de grande porte concluída em 1976 pela IBM sob o nome de Sistema R, e o primeiro sistema relacional comercial lançado em 1979 pela Oracle (JACKSON, 1999). Sua forma simples e genérica de organização associada a uma linguagem padronizada de

acesso aos dados (SQL - *Structured Query Language*) motivaram sua adoção e utilização extensiva, atingindo mais de 90% de todas as aplicações de bancos de dados atualmente em operação (RAHAYU et al., 2001). Com a evolução da informática nos últimos 20 anos, os BDR's também evoluíram e se tornaram tecnologias maduras, seguras, e de alto desempenho.

Com o surgimento da metodologia de desenvolvimento orientada a objeto, surgiu também a necessidade de se armazenar objetos em bancos de dados, nascendo assim os chamados bancos de dados orientados a objeto (BDOO). Os primeiros sistemas de bancos de dados de objetos surgiram na década de 80. No entanto, somente em 1991 surgiram os primeiros BDOO's padronizados (JACKSON, 1999). O interesse em desenvolver BDOO's não surgiu apenas como uma resposta natural ao novo paradigma imposto pela OO, mas sim, pela necessidade de armazenar dados complexos e relacionamentos comuns em objetos, mas não suportados pelos BDR's. Sua baixa utilização nos dias atuais se deve a sua imaturidade, à falta de padronização, e às dificuldades em utilizá-los extensivamente, quadro este que o ODMG (*Object Data Management Group*) vêm tentando reverter. O ODMG é formado por representantes das maiores empresas comerciais de BDOO's que juntos buscam definir padrões e incorporá-los a seus produtos. Os BDOO's definidos pelo ODMG implementam uma linguagem para definição de objetos (ODL – *Object Definition Language*), uma linguagem de consulta a objetos (OQL – *Object Query Language*), e às vezes suportam uma linguagem para manipulação de objetos (OML – *Object Management Language*). O OML é dependente da forma como o BDOO suporta as linguagens de programação, como Smalltalk, C++ e Java. Já, o ODL é um mecanismo independente de linguagem de programação que define os atributos e métodos de uma classe no banco de dados. Os métodos são escritos em OQL que é similar ao SQL e foi projetado para ser muito parecido com o padrão SQL 92, mas inclui extensões para orientação a objeto.

Apesar dos esforços do ODMG, a substituição de BDR's tradicionais por BDOO's é uma tarefa difícil, complexa, de custos elevados, e muitas vezes inviável, considerando a quantidade de sistemas BDR's em operação, sua complexidade, e o grande volume de dados que armazenam. Porém, a metodologia de OO tem se mostrado mais eficiente em modelar o mundo real que a metodologia relacional (RAHAYU et al., 2001), assim como a grande quantidade de estudos, técnicas e abordagens OO desenvolvidos na última década propiciaram um aumento expressivo no

desenvolvimento de sistemas OO cada vez mais complexos. Numa tentativa de resolver esse dilema, vários autores têm proposto formas de estender o modelo relacional para que este suporte a complexidade inerente à OO. Esse movimento, chamado de *Terceiro Manifesto* (DATE e DARWEN, 1998; JACKSON, 1999), deu origem aos chamados bancos de dados objetos-relacionais (BDO-R).

BDO-R's tentam unir o melhor da tecnologia relacional com o melhor da metodologia OO: o mecanismo eficiente de gerenciamento dos BDR's com a capacidade de modelagem do mundo real da OO. A idéia básica por trás disso é estender o modelo relacional para que ele possa trabalhar com o mecanismo de herança, a semântica dos relacionamentos agregação e associação, e atributos complexos. Isso induz a várias modificações na linguagem SQL como: suporte a tipos de dados e funções definidas pelo usuário, suporte a objetos complexos, e suporte a herança (JACKSON, 1999). Apesar de parecer uma forma rápida de migrar sistematicamente da plataforma relacional para a plataforma OO ou de trabalhar com soluções híbridas (sistemas OO e convencionais), essa abordagem ainda enfrenta alguns problemas relacionados ao desempenho dos sistemas gerenciadores de bancos de dados relacionais (SGBDR) existentes. A habilidade dos SGBDR's em transformar consultas SQL submetidas em consultas otimizadas é citada como um dos maiores benefícios da abordagem relacional. A heurística necessária para isso depende muito do conhecimento embutido nas operações da álgebra relacional. Consultas objetos-relacionais envolvem tanto dados complexos quanto funções definidas pelo usuário, cuja otimização é difícil de ser planejada. Isso pode gerar consultas muito lentas e comprometer o desempenho das aplicações, acarretando em custos operacionais indesejados pelas empresas. Enquanto essa e outras questões não são resolvidas, outros trabalhos de pesquisa propõem abordagens diferentes para o armazenamento de objetos em BDR's dentro da linha objeto-relacional. Rahayu et al. (2001) comentam ter identificado cinco categorias de abordagens objetos-relacionais diferentes:

1. Metodologias de transformação objeto-relacional: consistem da utilização da OO para modelagem do mundo real e transformação da mesma em um esquema relacional (tabelas) para implementação. Fong (2002) e Rahayu et al. (2000) são exemplos de trabalho nessa categoria.
2. Camada de objeto sobre o sistema relacional: consiste da construção de uma camada de software sobre o mecanismo de controle do BDR que simula as

características da OO. Essa camada pode transformar consultas OQL em consultas SQL, ou mapear objetos e seus relacionamentos em tabelas relacionais combinadas (operação *join*). Exemplos dentro dessa categoria são apresentados por Stoimenov et al. (1999), que propõem uma arquitetura para aplicações GIS (*Geographic Information Systems*), e Fong (2002), que comenta sobre a utilização de um *gateway* para transformação de consultas OQL em SQL com base em seu método de transformação.

3. Sistemas relacionais estendidos: consistem da extensão de sistemas relacionais para suportarem identificadores de objetos (OID – *Object Identifier*), herança, tipos de dados complexos, e operações definidas pelo usuário.
4. Coexistência de sistemas OO e SGBDR: consiste da implementação de interfaces de encapsulamento de sistemas relacionais que permitem aos sistemas OO acessarem e manipularem tabelas e consultas dentro de objetos. Um exemplo disso é a API *Borland Database Engine* (BDE) da Borland que permite aos programas escritos em linguagens C++ e Delphi acessarem dados em Paradox, dBase, Interbase e outros sempre da mesma maneira.
5. Interoperação de SGBDOO e SGBDR: consiste da utilização de múltiplos tradutores (ou *gateways*), um para cada banco de dados remoto, que possibilitam a uma aplicação utilizar simultaneamente um BDOO e um ou mais DBR's.

Como pôde ser observado, existem muitas abordagens objetos-relacionais, cada qual desenvolvida para um domínio do problema ou como uma resposta rápida às necessidades do mercado. No entanto, as três primeiras categorias parecem o caminho mais lógico a seguir, uma vez que possibilitam a migração sistemática e contínua de sistemas relacionais para sistemas OO sem comprometer a base de sistemas relacionais em operação, ou a utilização conjunta de sistemas OO e convencionais.

3.9. Considerações Finais

A adoção de uma abordagem orientada a objeto (OO) para o desenvolvimento de software oferece diversas vantagens sobre as abordagens convencionais. O conceito de objeto associado às três características fundamentais que a diferenciam das demais abordagens (encapsulamento, herança e polimorfismo) confere às estruturas OO maior

modularidade, flexibilidade, adaptabilidade e reusabilidade que as estruturas convencionais. Por essa razão, a OO conduz naturalmente ao reuso de componentes de software que é considerada uma das formas mais eficientes de produção de software. Ela proporciona um desenvolvimento mais rápido, reduz riscos de projeto e falhas na codificação, e conseqüente aumento na qualidade dos softwares produzidos a um custo menor.

Apesar das vantagens e aparentes facilidades de um desenvolvimento de software baseado em reuso, este possui atividades complexas apoiadas por tecnologias ainda em desenvolvimento. Dentre elas pode se destacar *Frameworks* de Aplicação Orientados a Objeto (OOAF).

OOAF é apontado como uma das tecnologias promissoras de reuso para auxiliar no projeto e na implementação de sistemas complexos. Sua grande vantagem sobre outras tecnologias, como componentes reusáveis e padrões de projeto, é que ele, por ser uma aplicação semi-completa, fornece simultaneamente ao desenvolvedor: (1) o modelo arquitetural com seus padrões e estruturas; (2) uma implementação dessas estruturas composta por componentes genéricos (classes abstratas) que definem as interfaces de acoplamento e controle para os componentes específicos (classes concretas); e (3) um programa principal que define o fluxo de controle da aplicação. No entanto, ele possui algumas desvantagens sobre as outras tecnologias, como: (1) redução em sua flexibilidade e adaptabilidade (projetado para aplicações de domínio específico); o tempo de aprendizagem e uso é mais longo (estrutura mais complexa); e (3) seu desenvolvimento é mais demorado e difícil, devido ao seu grau de generalização e dificuldades nos testes e validação (é necessário construir uma aplicação ao menos para isso).

Muitos avanços foram feitos na área de desenvolvimento de software a partir da adoção de abordagens e tecnologias OO. No entanto, a questão da persistência dos objetos ainda é muito discutida. Bancos de dados orientados a objeto (BDOO) ainda estão caminhando em busca de padrões e melhorias de desempenho. Por outro lado, a grande maioria dos sistemas computacionais em operação na atualidade utiliza bancos de dados relacionais (BDR), mais maduros, padronizados, e eficientes que o BDOO's, mas que não suportam o armazenamento de dados complexos e parte da semântica dos relacionamentos OO. Nesse contexto, a utilização de bancos de dados objetos-relacionais (BDO-R) tem demonstrado ser uma alternativa viável para resolver essa

questão a curto e médio prazo, pois combinam as necessidades da OO com a eficiência dos BDR's tradicionais.

Um software de apoio com as características sugeridas na seção 2.5, necessita de uma estrutura que suporte a inclusão não só de um conjunto de métodos e algoritmos para análise e formação de arranjos físicos, mas também de novos destes elementos, à medida que novas necessidades de projeto de arranjo físico forem surgindo, sem sofrer modificações ou gerar alterações significativas no código fonte.

Devido a suas características, a tecnologia de OOAF's demonstra ser mais adequada ao desenvolvimento desse tipo de software de apoio, pois: (1) as estruturas usadas para armazenar os dados dos projetos, os algoritmos e as rotinas de preparação e conversão de dados e integração com outros sistemas podem ser desenvolvidos como componentes e acoplados facilmente a seus pontos variáveis; e (2) a inversão de código pré-define um ambiente de trabalho para o projetista e a lógica de utilização desses componentes. Além disso, esses componentes podem ser registrados em uma biblioteca de componentes e usados para compor diversas aplicações diferentes, cada uma voltada para uma necessidade de projeto. Quanto ao gerenciamento adequado dos dados obtidos e gerados durante o desenvolvimento dos projetos, este pode ser obtido armazenando os objetos que os contêm em um BDO-R, a partir de uma das abordagens descritas na seção 3.8.

CAPÍTULO 4. ANÁLISE DO PROBLEMA E SOLUÇÃO ADOTADA

4.1. Considerações Iniciais

Neste capítulo, é apresentada a solução computacional adotada para o desenvolvimento do software proposto e os principais elementos que compõem essa solução, definidos a partir da análise dos elementos e dificuldades no desenvolvimento de projetos de arranjo físico apresentados no Capítulo 2.

O software proposto foi classificado como pertencente ao domínio das aplicações tipo pacote de software ou *toolkit* (mala de ferramentas) usadas para auxiliar em atividades de análise matemática, estatística, financeira, formação de células de manufatura, dentre outras, cuja principal característica é a presença de um pacote de rotinas para cálculos e análises, controladas por um conjunto de mecanismos que interagem com o usuário para executar essas rotinas e exibir seus resultados. No caso do software proposto, esse pacote de rotinas é usado para análise e formação de arranjos físicos, e o diferencial em seu desenvolvimento foi a definição de uma abordagem tecnológica que permite a inclusão de novos algoritmos e suas estruturas de dados sem necessitar modificar sua estrutura ou efetuar alterações significativas em seu código fonte.

A abordagem tecnológica adotada combina diversas tecnologias orientadas a objeto apresentadas no Capítulo 3. A utilização de um *framework* de aplicação orientado a objeto (OOAF) combinado a técnicas de reuso e um banco de dados objeto relacional (BDO-R) conferiram ao software proposto a escalabilidade desejada.

Deste capítulo em diante, as considerações descritas nos parágrafos seguintes foram feitas para facilitar a compreensão do texto descritivo da solução e de seus componentes.

A palavra Aplicação, com a letra inicial maiúscula, denota uma instância do *framework*.

A palavra aplicação, com a letra inicial minúscula, denota uma das muitas aplicações do software proposto como ferramenta de apoio ao projeto de arranjo físico. Quando associada a outros termos técnicos, confere a eles a mesma propriedade. Por exemplo, componentes da aplicação são todos os componentes do software proposto utilizados no apoio a um projeto de arranjo físico ou desenvolvidos com essa finalidade.

A palavra componente foi adotada como um termo genérico para referir-se a qualquer elemento da solução implementado como parte do software proposto (convertido em código). Por exemplo, um componente da aplicação refere-se a uma implementação computacional de um dado ou algoritmo usado em uma aplicação.

O nome da superclasse que representa um dos pontos variáveis do *framework* será usado com as iniciais maiúsculas para referir-se a qualquer uma de suas subclasses. Por exemplo, Objeto de Dados refere-se à própria classe Objeto de Dados ou a qualquer uma de suas derivadas.

O nome da superclasse que representa um dos pontos variáveis do *framework* será usado com as iniciais minúsculas para referir-se a qualquer instância de qualquer uma de suas subclasses. Por exemplo, objeto de dados refere-se a uma instância de qualquer subclasse derivada da classe Objeto de Dados.

A mesma convenção adotada para as superclasses será adotada para referir-se a qualquer classe, objeto ou conceito que defina formas genéricas ou classificatórias (iniciais maiúsculas) e seus elementos específicos (iniciais minúsculas).

4.2. Processo de Software Adotado

Normalmente, a adoção de um processo de software depende da abordagem tecnológica adotada para o desenvolvimento de um software. Essa abordagem é definida com base no domínio do problema, que é caracterizado durante a análise dos dados iniciais colhidos junto ao cliente. Por essa razão, todos os modelos de processo de software, encontrados na literatura, iniciam com a comunicação com o cliente e a especificação do software, como descrito na seção 3.3, p. 38.

No caso do software proposto, a análise inicial do domínio do problema levou a adoção de um processo de software baseado em um modelo de processo orientado a objeto, como o apresentado na seção 3.3.2, p. 42. Após a primeira volta da espiral, o projeto conceitual do software proposto revelou a abordagem tecnológica necessária

para seu desenvolvimento, caracterizando um processo de software voltado ao desenvolvimento e uso de *frameworks* de aplicação.

Sendo assim, o processo de software adotado para o desenvolvimento do software proposto foi o sugerido por Pressman (2002) para CBSE (seção 3.6, p. 46), modificado para acomodar o desenvolvimento baseado em OOAF, como sugerido na seção 3.7.2, p. 53. As principais modificações no processo são descritas a seguir:

- Eliminação das atividades relacionadas ao planejamento e à análise de risco. Essas atividades normalmente estão associadas a sistemas de grande porte, que envolvem tanto desenvolvimento de software quanto mudanças no hardware e envolvimento do pessoal da empresa. No caso do software proposto, desenvolvido com finalidades de pesquisa, essas atividades não são relevantes e podem ser tratadas de maneira informal.
- Adaptação do ciclo de voltas da espiral. A engenharia de domínio da CBSE, sugerida para desenvolver o *framework* (análise de domínio e projeto do *framework*), é realizada primeiro e ocupa as voltas mais internas da espiral, que correspondem às voltas da fase de desenvolvimento conceitual, e as primeiras voltas da fase de desenvolvimento de novos produtos. O desenvolvimento baseado em componentes da CBSE, sugerido para desenvolver as Aplicações (instanciação do *framework*), é realizada em seguida e ocupa as voltas intermediárias da espiral, que correspondem às voltas restantes da fase de projeto de desenvolvimento de novos produtos. As voltas das duas fases restantes (melhoria do produto e manutenção do produto) são compartilhadas por ambas as engenharias da CBSE desenvolvidas em paralelo para efetuar melhorias e mudanças nas Aplicações e no *framework*.
- Utilização da UML como ferramenta de modelagem do processo. A UML foi adotada para homogeneizar a forma de documentação do processo, tanto das atividades de análise, projeto, programação e teste orientados a objeto quanto das atividades para análise de domínio e documentação do *framework*, pois proporciona uma forma ampla e clara de representação do software e facilita a transição entre as diversas atividades do processo, definindo um padrão gráfico único para visualização do software e de seus componentes.

4.3. Levantamento e Análise do Problema

O levantamento dos requisitos revela o domínio do problema, enquanto que sua análise revela o domínio da aplicação e define a solução computacional mais adequada. Na literatura, podem ser encontradas diversas técnicas para se desenvolver essas atividades. As mais tradicionais buscam investigar as necessidades do cliente, avaliá-las tecnicamente em busca de necessidades adicionais em nível de sistema, e transformá-las nas listas de requisitos do usuário e do sistema respectivamente. Isso revela o domínio da aplicação que orienta na definição da arquitetura do software e na escolha da abordagem tecnológica mais adequada ao seu desenvolvimento. Os requisitos, por sua vez, determinam as principais funcionalidades do software e os detalhes de sua estrutura.

No caso do software proposto, como se trata de um projeto com finalidades acadêmicas e de pesquisa, os requisitos do usuário foram definidos com base nas dificuldades no desenvolvimento de projetos de arranjo físico descritas na seção 2.5. A Tabela 4.1 mostra uma síntese dessas dificuldades na forma de necessidades do usuário com os requisitos definidos para satisfazê-las.

A análise técnica das necessidades e requisitos listados na Tabela 4.1 revelou o domínio da aplicação e um novo conjunto de necessidades do software proposto, sintetizadas na Tabela 4.2 juntamente com os requisitos para satisfazê-las.

Tabela 4.1 – Necessidades do usuário × requisitos do usuário.

	Necessidade do Usuário	Requisito do Usuário
1	Auxílio na escolha do arranjo físico capaz de satisfazer os objetivos do projeto.	U.1. O software proposto deve disponibilizar um conjunto de ferramentas para análise dos dados da empresa, como análises P-Q ou P-Q-L.
2	Auxílio na escolha do método e dos algoritmos mais adequados para a análise e formação de um tipo de arranjo físico.	U.2. O software proposto deve disponibilizar um número razoável de métodos e algoritmos diferentes organizados de maneira orientada ao processo ou tipo de arranjo físico para facilitar a escolha e a aplicação dos mesmos.

	Necessidade do Usuário	Requisito do Usuário
3	Auxílio na organização, preparação e interpretação dos dados usados nas análises.	U.3. O software proposto deve possuir formas de organização e gerenciamento eficientes dos dados dos projetos; e U.4. O software proposto deve possuir rotinas de obtenção, preparação e conversão de dados para auxiliar na aplicação mais rápida dos métodos e algoritmos, assim como na interpretação dos resultados.
4	Auxílio na escolha do resultado capaz de melhor satisfazer os objetivos do projeto.	U.5. O software proposto deve disponibilizar rotinas para comparação dos resultados através de critérios predeterminados; U.6. O software proposto deve disponibilizar rotinas para comparação visual dos resultados através de uma interface amigável; e U.7. O software proposto deve possuir meios de integração com sistemas de simulação para avaliação dos resultados obtidos.
5	Auxílio no detalhamento do arranjo físico escolhido.	U.8. O software proposto deve possuir meios de integração com sistemas CAD para o detalhamento gráfico do arranjo físico.

De acordo com os requisitos do usuário listados na Tabela 4.1, o software pertence ao domínio das aplicações tipo pacote de software ou *toolkit*, usadas para auxiliar em atividades de análises matemática, estatística, financeira, formação de células de manufatura, dentre outras. Alguns exemplos de aplicações pertencentes a esse domínio são o MiniTab® e o MatLab®, que reúnem diversas rotinas para cálculos matemáticos e estatísticos, e o PFAST descrito na seção 2.4.

Tabela 4.2 – Necessidades do sistema X requisitos do sistema.

	Necessidade do Sistema	Requisito do Sistema
1	<p>Capacidade para incorporar diferentes tipos de <u>algoritmos</u> e rotinas (<u>operações auxiliares</u>) para:</p> <ul style="list-style-type: none"> – obtenção, preparação e conversão de dados; – comparação e visualização de resultados; e – integração com softwares de simulação e CAD. 	S.1. O software proposto deve possuir a flexibilidade estrutura necessária para incorporar e controlar diversos tipos diferentes de algoritmos e operações auxiliares.
2	Capacidade para permitir fácil inclusão de novos algoritmos e operações auxiliares para atender a necessidades futuras ou não previstas do projetista.	S.2. Além de flexível, a estrutura do software proposto deve permitir a inclusão de novos algoritmos e operações auxiliares sem sofrer modificações significativas em seu código fonte.
3	Formas de organização e disponibilização dos algoritmos e das operações auxiliares de acordo com as necessidades de cada método ou projeto para facilitar a escolha do projetista.	S.3. O software proposto deve criar um ambiente de trabalho em que os algoritmos e operações auxiliares sejam organizados e disponibilizados de acordo com as necessidades de cada método, projeto ou tipo de arranjo físico.
4	Capacidade para organizar os dados dos projetos de acordo com as necessidades do mesmo.	S.4. O software proposto deve possuir um banco de dados cujo gerenciamento atenda as necessidades de organização dos projetos.
5	Capacidade para armazenar e recuperar os diversos tipos de dado diferentes usados no desenvolvimento dos projetos.	S.5. A estrutura do banco dados do software proposto deve ser flexível o bastante para conseguir armazenar e recuperar dados de diversos tipos diferentes e complexos.

Softwares pertencentes a esse domínio possuem características estruturais e funcionais bem definidas. Em geral, eles possuem: (1) um conjunto de rotinas que implementam funções de cálculo e análise; (2) um conjunto de mecanismos para seleção, controle e execução dessas rotinas a partir de critérios e comandos pré-definidos; (3) um conjunto de operações para aquisição, tratamento, conversão, gravação, visualização e impressão dos dados tomados como entrada por essas rotinas; e (4) um conjunto de interfaces para interação do usuário com o software e com as rotinas, os mecanismos e as operações mencionados. No caso do software proposto, essas quatro características fundamentais foram complementadas com base nos requisitos listados na Tabela 4.1 e na Tabela 4.2, e sintetizadas na Tabela 4.3 que mostra, da direita para esquerda, as características do software proposto com os requisitos que as definiram, e o tipo das mesmas (funcional ou estrutural).

A estrutura arquitetural básica observada nos softwares pertencentes a esse domínio é composta por: (1) uma espécie de pacote ou biblioteca de rotinas; (2) um núcleo formado pelos mecanismos de seleção, controle e execução; (3) um módulo que cuida da interação do usuário com esse núcleo; e (4) algum tipo de repositório de dados. No caso do software proposto, essa estrutura foi complementada e detalhada de acordo com outros aspectos relevantes à solução adotada, como será apresentado na seção 4.5.1.

As características listas na Tabela 4.3 determinam aspectos importantes para a escolha da abordagem tecnológica mais adequada ao desenvolvimento do software proposto. No entanto, outros elementos abstraídos durante o processo de análise contribuíram para a escolha das tecnologias que compuseram essa abordagem. Esses elementos também conferiram características importantes à estrutura do software proposto. Os elementos mais importantes são apresentados nos parágrafos a seguir e descritos em detalhes na seção 4.5.

Tabela 4.3 – Características do software proposto.

	Requisitos	Características
Características Funcionais	U.1, U.2, U.5, U.6, U.7 e U.8	C.1. O software proposto deve disponibilizar um número razoável de algoritmos e operações auxiliares organizados de maneira orientada ao projeto, tipo de arranjo físico, ou método para facilitar a escolha e a aplicação dos mesmos.
	S.2	C.2. O software proposto deve possuir meios de incorporar novos algoritmos e operações auxiliares para atender às necessidades futuras ou não previstas do projetista sem prejudicar os projetos desenvolvidos ou em desenvolvimento.
	S.3	C.3. O software proposto deve possuir um conjunto de mecanismos capazes de selecionar, controlar e executar quaisquer algoritmos e operações auxiliares a partir de um conjunto de critérios e uma lógica pré-definida para interação dos mesmos com o sistema e com o projetista.
	U.3, U.4 e S.4	C.4. O software proposto deve armazenar, recuperar e organizar os dados de acordo com as necessidades dos projetos.
Características Estruturais	S.1	C.5. O software proposto deve possuir uma estrutura que suporte quaisquer estruturas para os dados dos projetos, algoritmos e operações auxiliares sem necessitar modificações.
	S.2	C.6. A estrutura do software proposto deve permitir a inclusão ou remoção de quaisquer estruturas para os dados dos projetos, algoritmos e operações auxiliares sem sofrer modificações ou gerar alterações significativas no código fonte.
	S.5	C.7. O software proposto deve possuir um banco de dados com uma estrutura flexível o bastante para armazenar e recuperar diversos tipos de dados comuns e complexos que variam de simples números e textos a matrizes, vetores e tabelas.

Durante o processo de análise, foram identificados quatro elementos essenciais para o uso do software proposto como um pacote, denominados elementos da aplicação:

1. **dados do projeto:** quaisquer dados obtidos para a aplicação dos diversos algoritmos e métodos usados nos processos de análise e formação de arranjo físico de um projeto, assim como quaisquer resultados gerados pelos mesmos;
2. **algoritmos:** quaisquer algoritmos aplicados nos processos de análise e formação de arranjo físico de um projeto;
3. **operações auxiliares:** quaisquer rotinas usadas para obtenção, preparação e conversão de dados, comparação e visualização de resultados, integração com sistemas de simulação e CAD, dentre outras; e
4. **interfaces com o usuário:** quaisquer interfaces de software usadas para interação do projetista com os algoritmos e operações auxiliares ou com outras funções definidas para o software proposto.

Esses quatro elementos compõem a base para as aplicações do software proposto como software de apoio ao projeto de arranjo físico. Seus relacionamentos formam a estrutura básica do software proposto, e suas interações determinam a maneira na qual são controlados pelos mecanismos do núcleo.

Outro elemento importante são as **Estruturas de Projeto**, descritas na seção 4.5.2. Estruturas de Projeto são estruturas dinâmicas usadas para organizar os elementos das aplicações de acordo com as necessidades dos métodos, projetos ou tipos de arranjo físico. As Aplicações utilizam essas estruturas para disponibilizar ao projetista os elementos da aplicação mais apropriados ao desenvolvimento de um método, projeto ou tipo de arranjo físico, satisfazendo os requisitos S.2, S.3 e S.4, e as características C.3, C.4 e C.5.

O penúltimo elemento importante são as **Estruturas das Aplicações**, descrita na seção 4.5.3. Estruturas das Aplicações são estruturas similares às Estruturas de Projeto e são usadas para determinar quais Estruturas de Projeto podem ser usadas em uma Aplicação e quais elementos das aplicações serão disponibilizados inicialmente ao projetista para este iniciar seu trabalho. Os mecanismos do núcleo usam essas estruturas para criar e executar dinamicamente uma Aplicação.

O último elemento importante, esboçado durante a análise e detalhado durante o projeto do *framework*, é o Metamodelo, descrito na seção 4.5.4. O Metamodelo é um modelo abstrato composto por superclasses que descrevem os elementos das aplicações,

e um conjunto de classes auxiliares cujos objetos são considerados elementos estruturais, utilizados para descrever os parâmetros dos algoritmos e operações auxiliares, e estruturar as Estruturas de Projeto e Estruturas das Aplicações. Os relacionamentos entre as superclasses determinam os relacionamentos que suas subclasses podem estabelecer entre si, assim como os relacionamentos das superclasses com as classes auxiliares determinam os relacionamentos que suas subclasses podem estabelecer com os objetos das classes auxiliares. Esses relacionamentos são descritos na forma de roteiros que são utilizados pelos mecanismos do núcleo para criar e executar dinamicamente as aplicações, e para solicitar ao projetista os parâmetros dos algoritmos e operações auxiliares. As subclasses que implementam os elementos das aplicações foram denominadas genericamente de componentes das aplicações.

4.4. Abordagem Tecnológica Adotada

O levantamento e a análise dos requisitos definiram aspectos importantes para a escolha das tecnologias que compuseram a abordagem tecnológica adotada para o desenvolvimento do software proposto. Em princípio, uma abordagem tradicional, baseada em procedimentos ou até orientada a objeto, seria suficiente para desenvolver um software tipo pacote ou *toolkit*, uma vez que sua principal funcionalidade é a execução das rotinas do pacote escolhidas pelo usuário. No entanto, quatro aspectos importantes inviabilizam essa abordagem:

1. Execução dos algoritmos e operações auxiliares. Seus parâmetros variam em tipo de dados e quantidade. Em uma abordagem tradicional, seria necessário implementar uma rotina de execução para cada algoritmo ou operação auxiliar. Essas rotinas solicitariam os parâmetros necessários, chamariam o algoritmo ou operação auxiliar passando os parâmetros solicitados, e exibiriam os resultados. No caso do software proposto, isso representaria efetuar modificações em sua estrutura e alterar significativamente seu código fonte, não satisfazendo os requisitos S.1e S.2.
2. Controle sobre os dados dos projetos. É similar ao aspecto 1. Existem diversos tipos de dados diferentes tomados como parâmetro pelos algoritmos. Em uma abordagem tradicional, seria necessário efetuar modificações na lógica e na estrutura do software, inclusive no banco de dados, para

implementar, armazenar e recuperar cada dado do projeto. No caso do software proposto, isso não satisfaz os requisitos S.4e S.5.

3. Disponibilização dos algoritmos e operações auxiliares. Isso depende de restrições definidas para cada método, projeto ou tipo de arranjo físico. De fato, não faz sentido disponibilizar ao projetista um algoritmo para *layouts* de bloco se ele está desenvolvendo um arranjo físico celular, por exemplo. No entanto, em uma abordagem tradicional, seria necessário definir essas restrições, de alguma maneira, no código fonte e alterar as interfaces do sistema para disponibilizar um novo algoritmo ou rotina auxiliar ao projetista. No caso do software proposto, isso não satisfaz o requisito S.3.
4. Inclusão ou remoção de quaisquer componentes das aplicações. Em uma abordagem tradicional, isso implicaria em algumas modificações estruturais e em alterações significativas em seu código fonte. No caso do software proposto, isso não satisfaz quaisquer requisitos do sistema.

Esses quatro aspectos propuseram o maior desafio para o desenvolvimento do software proposto: definir uma solução tecnológica que satisfizesse os requisitos do usuário e do sistema e ainda facilitasse a inclusão ou remoção de componentes das aplicações.

A solução para esse desafio foi pensar no software proposto como um “brinquedo” feito com blocos de montar que utiliza uma base para encaixar um conjunto de blocos. Isso equivale à forma mais simples de uso de um *framework* que busca construir uma Aplicação a partir de componentes existentes, como descrito na seção 3.7.1, p. 50. Para suportar essa solução, foram combinadas diversas tecnologias orientadas a objeto, como mostrado na Figura 4.1. Detalhes da utilização de cada uma dessas tecnologias no desenvolvimento do software proposto são fornecidos nos parágrafos a seguir.

Em uma abordagem OO, os quatro tipos de elementos da aplicação, apresentados na seção 4.3, podem ser representados por classes genéricas (superclasses) que definem uma interface padronizada para a seleção, controle e execução dos mesmos pelos mecanismos do núcleo do software proposto. Os conceitos de herança e polimorfismo permitem que essas superclasses assumam o comportamento de suas classes derivadas (subclasses), criadas para implementar elementos específicos da aplicação (componentes da aplicação). Com isso, os mecanismos do núcleo podem controlar quaisquer

componentes da aplicação sem necessitar rotinas adicionais para isso. Essa forma de desenvolvimento é característica de *frameworks* de aplicação orientados a objeto (OOAF). As classes genéricas correspondem aos pontos variáveis do *framework* e o núcleo composto de um conjunto de mecanismos corresponde à inversão de controle. Por essa razão, o primeiro passo foi definir um OOAF para o desenvolvimento do software proposto que fornecesse a base para a construção das aplicações do software proposto.

Figura 4.1 – Síntese da abordagem tecnológica adotada

O segundo passo foi definir uma forma de “empacotar” todos os componentes da aplicação e encontrar um meio para que os mecanismos pudessem selecioná-los, encaixá-los aos pontos variáveis e executá-los dinamicamente. Vale a pena lembrar que pode haver um número indefinido desses componentes que devem ser encaixados em um número fixo de pontos variáveis (existe apenas um para cada tipo de componente no *framework*). A solução adotada foi criar um tipo de biblioteca (BCA – Biblioteca de Componentes das Aplicações) que contém todos os componentes das aplicações na forma de classes de objetos e um conjunto de roteiros de uso desses componentes que descrevem as Estruturas das Aplicações, Estruturas de Projeto e os parâmetros de cada algoritmo ou operação auxiliar. Através de uma lógica pré-definida, os mecanismos percorrem esses roteiros para criar e executar dinamicamente as Aplicações (instâncias do *framework*) e suas aplicações (utilização das Aplicações como software de apoio) a

partir das interações do projetista com as interfaces do software proposto. A utilização de estruturas dinâmicas descritas na forma de roteiros elimina a necessidade de efetuar quaisquer modificações estruturais ou no código fonte do software proposto e permite ao desenvolvedor criar e modificar Aplicações rapidamente, podendo personalizá-las de acordo com as necessidades de um projeto ou projetista de arranjo físico.

O terceiro passo foi adotar uma forma de alimentar a BCA sem demandar muito esforço. A maior dificuldade nessa tarefa é construir e registrar as Estruturas das Aplicações e as Estruturas de Projeto. Elas são complexas e envolvem muitos componentes das aplicações. Se não forem construídas corretamente, elas podem gerar diversos problemas durante a execução do software proposto. Para auxiliar o desenvolvedor nessa tarefa, foram adotadas algumas técnicas da CBSE para desenvolver uma Biblioteca de Componentes Reusáveis (BCR) que contém todos os componentes das aplicações e seus relacionamentos. O desenvolvedor administra e gerencia essa biblioteca por meio de um software de suporte, denominado Software Gerenciador de Componentes das Aplicações (SGCA). O SGCA auxilia no registro dos componentes e na construção das Estruturas de Projeto que compõem as diversas aplicações do software proposto. Ao final desse processo, o desenvolvedor utiliza uma função do SGCA para converter os elementos contidos na BCR nos elementos contidos na BCA, alimentando-a com os roteiros e componentes utilizados nas Aplicações.

O quarto passo foi adotar uma tecnologia de banco de dados que satisfizesse os requisitos do usuário e do sistema levantados na seção 4.3. A maior dificuldade encontrada foi definir um meio de fazer o gerenciador desse banco de dados reconhecer automaticamente um tipo novo de dados dos projetos sem necessitar criar rotinas ou novas estruturas de armazenamento para isso. A solução para esse problema foi a adoção de uma abordagem objeto-relacional como as citadas na seção 3.8. Essa abordagem utiliza uma camada de software, denominada Repositório de Objetos de Dados (ROD), sobre um SGBD para mapear as propriedades dos objetos que contêm os dados dos projetos (objetos de dados) em campos de uma ou mais tabelas relacionais. Cada objeto possui métodos específicos usados pelo ROD para obter as informações que ele necessita para criá-lo e armazená-lo no banco de dados ou recuperá-lo para a memória, onde é mantido durante sua utilização pelos algoritmos e operações auxiliares.

Para finalizar essa abordagem, a UML foi adotada para criar gabaritos (*templates*) utilizados nos processos de modelagem dos componentes das aplicações, Estruturas das

Aplicações e Estruturas de Projeto. Esses gabaritos auxiliam o desenvolvedor na utilização do *framework* e facilitam sua compreensão. Para criação desses gabaritos, foram utilizados os mecanismos de extensibilidade da UML, como estereótipos, valores atribuídos e restrições que ajudam o desenvolvedor a entender a origem de cada componente e pré-definir suas propriedades.

4.5. Elementos da Solução Adotada

Vários elementos compuseram a solução adotada para o desenvolvimento do software proposto. Os principais foram apresentados nas seções anteriores. Nesta seção, esses elementos são descritos juntamente com outros elementos definidos ao longo do processo de projeto do software proposto.

4.5.1. Arquitetura do Software

A definição de uma arquitetura de software é parte do resultado da análise de domínio. Para o software proposto, foi definida uma arquitetura que busca satisfazer os requisitos do sistema, descritos na seção 4.3, e que incorpora os elementos estruturais discutidos na seção 4.4: ROD e BCA.

A visão geral da arquitetura do software proposto é mostrada na Figura 4.2 na forma de blocos dispostos em camadas. Os blocos Aplicação 1, Aplicação 2 e Aplicação n representam as instâncias do *framework*, o bloco Núcleo de Controle representa o núcleo do *framework*, e os demais blocos representam o ROD, a BCA e um Sistema Gerenciador de Banco de Dados (SGBD). Os parágrafos seguintes descrevem esses blocos.

Figura 4.2 – Visão geral da arquitetura do software proposto

Cada **Aplicação** é formada por uma Interface Padrão, um conjunto de Estruturas de Projeto e um conjunto de operações auxiliares iniciais. A **Interface Padrão** é a interface principal da Aplicação. Ela define o ambiente de trabalho do projetista e determina a forma de organização e disponibilização dos componentes das aplicações ligados as Estruturas de Projeto relacionadas à Aplicação. As operações auxiliares iniciais são disponibilizadas ao projetista para que este possa iniciar seu trabalho, como abrir um projeto existente ou criar um projeto novo. Dentro da lógica de operação definida para o *framework*, uma Aplicação é formada basicamente por componentes das aplicações organizados segundo uma Estrutura da Aplicação, que é usada pelos mecanismos do *framework* para criar e executar essa Aplicação. Isso será melhor discutido no Capítulo 5. Sendo assim, pode haver várias Aplicações registradas na BCA. Nesse caso, a escolha de qual Aplicação deve ser executada pelo *framework* pode ser feita pelo projetista ou pré-configurada no software proposto.

O **Núcleo de Controle** contém um conjunto de mecanismos que definem a lógica básica de operação do *framework* e são implementados como métodos dos pontos fixos do *framework*. Ele encapsula o ROD e a BCA, e os utiliza, respectivamente para: garantir a persistência dos objetos que contém os dados dos projetos; e obter os componentes das aplicações e seus roteiros de uso, definidos a partir das Estruturas das Aplicações, Estruturas de Projeto e descrições dos parâmetros dos algoritmos e operações auxiliares.

O **Repositório de Objetos de Dados (ROD)** é uma camada de software sobre um SGBD, cuja função é mapear as propriedades dos objetos que contém os dados dos projetos (objetos de dados) para o formato definido pelo banco de dados e vice e versa. Isso define uma interface de acesso única e independente ao SGBD em uso. O ROD foi elaborado para encapsular um SGBDR, um SGBDOO ou um SGBDO-R comerciais, mas sua independência funcional é suficiente para encapsular qualquer sistema de armazenamento e recuperação de dados, inclusive sistemas de arquivos. O ROD também fornece ao *framework* e suas Aplicações conjuntos de objetos de dados que são utilizados pelas Aplicações para exibir os projetos com seus dados organizados na forma de uma estrutura tipo árvore, denominada Árvore de Projeto.

A **Biblioteca de Componentes das Aplicações (BCA)** é a BCR convertida pelo SGCA. Para encapsular detalhes dessa conversão, foi projetada uma camada de software sobre ela que desempenha papel similar ao do ROD sobre o SGBD. Isso define uma interface única e independente de acesso aos componentes das aplicações e seu roteiros

de uso, encapsulando detalhes de implementação. A BCA foi elaborada para conter um conjunto de tabelas estáticas em código fonte com referências as classes das aplicações e informações sobre as Estruturas das Aplicações e Estruturas de Projeto interpretadas como roteiros de uso. No futuro, ela poderá encapsular outras tecnologias com a mesma finalidade, como componentes binários (DLL's, JavaBeans, ActiveX, etc.) e algum tipo de linguagens para descrição dos roteiros – por exemplo XML (*Extendible Markup Language*), dispensando a necessidade de recompilar o software proposto para incorporar a BCA toda vez que esta for modificada (gerada pelo SGCA).

4.5.2. Estruturas de Projeto

As Estruturas de Projeto foram elaboradas para “encaixar” os componentes das aplicações ao *framework*. Elas são estruturas dinâmicas definidas a partir de um conjunto de classes e objetos estruturais relacionados entre si registrados na BCR que determinam quais componentes das aplicações podem ser utilizados em um determinado método, projeto ou tipo de arranjo físico. Esses elementos serão descritos e exemplificados na seção 4.5.4.

Uma Estrutura de Projeto é descrita como uma forma de organização estrutural hierárquica que relaciona um projeto aos componentes que descrevem seus dados. Esses componentes, por sua vez, são relacionados aos componentes que implementa os algoritmos e operações auxiliares que podem ser utilizados para processar os dados desse projeto. Um projeto pode ser dividido em vários subprojetos. Subprojeto é uma denominação genérica utilizada para representar um método, tipo de arranjo físico ou outro projeto de arranjo físico específico. Um subprojeto tem as mesmas propriedades de um projeto e é composto da mesma maneira, podendo ser dividido em outros subprojetos. Essa forma de organização genérica permite representar desde um simples método a um projeto complexo, dividido em vários tipos de arranjo físico com suas diversas visões gerais e detalhadas. A Figura 4.3 mostra um exemplo de uma Estrutura de Projeto para um projeto de arranjo físico fictício.

As Aplicações usam as Estruturas de Projeto como um modelo ou gabarito para gerar dinamicamente um projeto de arranjo físico que pode tomar diferentes configurações dependendo das necessidades do projetista. À medida que o projetista vai interagindo com o software, os projetos com seus subprojetos vão sendo criados e os dados obtidos e gerados vão sendo vinculados a eles, formando uma estrutura de dados hierárquica denominada Árvore de Projeto.

Figura 4.3 – Exemplo de uma Estrutura de Projeto

Uma **árvore de projeto** pode ser vista como uma instância de uma Estrutura de Projeto na qual os dados obtidos e gerados durante o desenvolvimento de um projeto são vinculados a ela. Ela é formada por um conjunto de objetos vinculados entre si que contêm informações sobre: (a) o projeto em si (raiz da árvore); (b) cada um de seus subprojetos (ramos da árvore ou sub-árvores); e (c) os dados obtidos e gerados durante o desenvolvimento desse projeto ou de um de seus subprojetos (nós folha da árvore). A Figura 4.4 mostra um exemplo de árvore de projeto construída a partir da Estrutura de Projeto mostrada na Figura 4.3 com os elementos descritos neste parágrafo indicados pelas letras que os precedem.

Figura 4.4 – Exemplo de uma árvore de projeto

4.5.3. Estruturas das Aplicações

Estruturas das Aplicações são similares às Estruturas de Projeto e são construídas da mesma maneira. Elas foram elaboradas para estruturar uma instanciiação do *framework*.

Assim como os projetos, as Aplicações foram definidas para serem construídas a partir dos componentes das aplicações, conferindo a elas os mesmos aspectos dinâmicos relacionados à construção dos projetos. Isso permite registrar uma ou mais Aplicações na BCR e gerar uma BCA específica para cada Aplicação, ou uma BCA com todas as Aplicações.

Uma Estrutura da Aplicação é usada para relacionar uma Aplicação às Estruturas de Projeto que podem ser utilizadas dentro do ambiente definido por sua Interface Padrão, que as utiliza para gerar as Árvores de Projeto de acordo com as solicitações do projetista. Ela também é usada para relacionar um conjunto de operações auxiliares disponibilizadas inicialmente ao projetista para que este possa interagir com essas Estruturas de Projeto e com as Árvores de Projeto armazenadas no banco de dados, como operações para abrir um projeto existente ou criar um novo projeto. A Figura 4.5 mostra um exemplo de uma Estrutura da Aplicação que usa a Estrutura de Projeto mostrada na Figura 4.3 e três operações auxiliares: Criar, Abrir e Fechar Projeto Piloto.

Figura 4.5 – Exemplo de uma Estrutura da Aplicação

4.5.4. O Metamodelo

Durante a fase de levantamento e análise do problema, e parte do projeto conceitual do software proposto, os elementos descritos nas seções anteriores foram abstraídos na forma de classes de objetos. Essas classes e seus relacionamentos foram modelados na forma de um diagrama de classes UML, denominado Metamodelo, mostrado na Figura 4.6. As classes à esquerda representam os elementos usados para compor as Estruturas das Aplicações e as Estruturas de Projeto; as classes ao centro representam os componentes das aplicações; e as classes a direita são usadas para descrever os parâmetros dos algoritmos e operações auxiliares.

O Metamodelo é um modelo abstrato empregado de diferentes maneiras no desenvolvimento do *framework*, na construção das Aplicações, e na construção de novos componentes das aplicações. As classes do Metamodelo são organizadas em três grupos:

1. grupo de classes que representam os pontos variáveis do *framework* (classes em amarelo claro na Figura 4.6);
2. grupo de classes que representam os pontos variáveis de suporte do *framework* (classes em azul claro na Figura 4.6); e
3. grupo de classes auxiliares que estruturam as Aplicações e são considerados pontos fixos do *framework* (classes em verde claro na Figura 4.6).

Os relacionamentos entre essas classes determinam como os componentes das aplicações são conectados entre si e às Estruturas das Aplicações e Estruturas de Projeto. Para isso, cada relacionamento recebeu um nome, o qual é utilizado no processo de modelagem das Aplicações para que o desenvolvedor possa identificá-los e registrá-los na BCR corretamente.

Como mencionado, as classes ao centro da Figura 4.6 são as superclasses que representam os três principais pontos variáveis do *framework*. A partir delas são derivados os componentes das aplicações:

1. dados dos projetos: são descritos pelas derivadas concretas da superclasse **Objeto de Dados**, cujos objetos são considerados persistentes;
2. algoritmos e operações auxiliares: são implementados pelas derivadas concretas da superclasse **Procedimento Operacional**; e

3. interfaces de controle: são implementadas pelas derivadas concretas da superclasse **Interface com o Usuário**.

Figura 4.6 – O Metamodelo

Essa composição básica possui um alto grau de abstração e oferece ao desenvolvedor a flexibilidade necessária para criar quaisquer componentes das aplicações e adicioná-los ao software proposto sem necessitar modificar sua estrutura ou alterar seu código fonte.

As classes na parte inferior da Figura 4.6 – **Variável** e **Objeto de Projeto** – representam um tipo específico de dado e são tratadas de maneira distinta pelo núcleo do *framework*. Ambas são superclasses e suas derivadas podem ser tomadas como parâmetro pelos Procedimentos Operacionais. Por isso, foram modeladas como derivadas da superclasse **Objeto de Dados**. A superclasse **Variável** é usada para derivar classes concretas que descrevem dados simples, como números e cadeias de caracteres (*strings*), que não são considerados dados dos projetos, mas são utilizados pelos algoritmos e operações auxiliares em seus processamentos, como coeficientes, valores de máximo e mínimo, etc.. A superclasse **Objeto de Projeto** é usada para derivar classes concretas que descrevem um projeto ou subprojeto, cujos objetos são usados para vincular os objetos de dados que contém os dados desse projeto ou subprojeto.

As classes à direita da Figura 4.6 (em azul claro) – **Procedimento de Seleção** e **Procedimento de Visualização** – são as superclasses que representam os dois pontos

variáveis de suporte do *framework*. Elas assim foram denominadas por não serem componentes das aplicações, mas serem necessárias ao processo de execução dos procedimentos operacionais. As derivadas concretas da superclasse **Procedimento de Seleção** são usadas pelo núcleo do *framework* para solicitar parâmetros de entrada de um procedimento operacional ao projetista, quando estes forem necessários. Elas podem ser utilizadas para selecionar um ou mais parâmetros e podem utilizar uma Interface com o Usuário para interação com o projetista. A superclasse **Procedimento de Visualização** representa um conjunto de Procedimentos Operacionais que podem ser utilizados pelo núcleo do *framework* para exibir os resultados de um processamento ao projetista. São opcionais e podem estar relacionadas a um ou mais parâmetros de saída de um Procedimento Operacional. Não existe, de fato, uma superclasse Procedimento de Visualização, todas são derivadas concretas da superclasse Procedimento Operacional são reconhecidas pelos mecanismos do *framework* como tal. Sua representação no Metamodelo é semântica e foi definida para facilitar a automação de métodos de análise e formação de arranjo físico e a exibição automática de resultados, uma vez que permite a execução em cascata de algoritmos e/ou operações auxiliares.

As classes em verde claro à esquerda e à direita da Figura 4.6 – **Aplicação, Contexto, Processa, e Parâmetro** – representam as quatro classes auxiliares que estruturam as Aplicações e são reconhecidas como pontos fixos do *framework*. Suas instâncias são consideradas “elementos estruturais” e são usadas para descrever as Estruturas das Aplicações, as Estruturas de Projeto, e os parâmetros de um Procedimento Operacional. Essas instâncias são criadas pelo SGCA, durante o cadastramento dos componentes das aplicações, das Estruturas das Aplicações, e das Estruturas de Projeto, e registradas na BCR para depois serem convertidas em elementos dos roteiros de uso da BCA. Uma explicação mais detalhada sobre isso é dada na seção 5.3.2.

A classe **Parâmetro** é usada para descrever os parâmetros de uma derivada concreta da superclasse Procedimento Operacional. Seus objetos relacionam essa classe derivada às derivadas concretas da superclasse Objeto de Dados tomadas como seus parâmetros de entrada e/ou saída, e determinam quais derivadas concretas da superclasse Procedimento de Seleção serão usadas para selecionar os objetos de dados de entrada e quais derivadas concretas da superclasse Procedimento Operacional serão

usados como Procedimentos de Visualização para exibir os objetos de dados de saída durante a execução de suas instâncias.

A classe **Aplicação** é usada para representar as instâncias do *framework*. Ela é uma derivada concreta da superclasse Objeto de Dados e cada um de seus objetos descreve uma Aplicação. Os relacionamentos desse objeto com outros elementos descrevem a Interface Padrão, as Estruturas de Projeto e as operações auxiliares iniciais dessa Aplicação. O relacionamento Processa a Aplicação associa uma Aplicação ao Procedimento Operacional com sua Interface com o Usuário de controle que implementa a Interface Padrão. O relacionamento Usa associa uma Aplicação a seu contexto da aplicação, que descreve quais são suas Estruturas de Projeto e operações auxiliares iniciais.

A classe de associação **Processa** é usada para relacionar uma derivada concreta da superclasse Objeto de Dados a uma derivada concreta da superclasse Procedimento Operacional formando um par denominado **par processa**. Seus objetos são utilizados para “encaixar” esse par às Estruturas das Aplicações e às Estruturas de Projeto. Em termos semânticos, esse relacionamento determina quais Procedimentos Operacionais podem processar quais Objetos de Dados em um determinado projeto. Em uma Aplicação, ele é usado para indicar quais são suas operações auxiliares, e quais algoritmos e operações auxiliares podem ser disponibilizados ao projetista para processar um determinado dado de um projeto. Essa lógica é baseada na visão do projetista, que constrói seus arranjos físicos a partir dos dados dos projetos. Nessa lógica, os primeiros elementos das aplicações disponibilizados ao projetista são os dados dos projetos. Quando ele seleciona um dado, ele espera que os algoritmos e operações auxiliares disponíveis atuem sobre aquele dado. Para o desenvolvedor, os Procedimentos Operacionais relacionados a um Objeto de Dados, passam a ser vistos como métodos ligados dinamicamente a esse Objeto de Dados. Essa visão auxilia na construção das Estruturas das Aplicações e das Estruturas de Projeto, pois um Objeto de Dados passa a “carregar” consigo todos os Procedimentos Operacionais que o processam com suas Interface com o Usuário de controle.

A classe **Contexto** é o principal elemento estrutural usado na construção das Estruturas das Aplicações e Estruturas de Projeto. Seus objetos são usados para relacionar às derivadas concretas da classe Objeto de Projeto (projetos de subprojetos) os componentes das aplicações usados em seu desenvolvimento, assim como, relacionar

a uma Aplicação suas Estruturas de Projeto e operações auxiliares iniciais. Em termos semânticos cada projeto, subprojeto ou Aplicação possui um contexto que determina quais componentes da aplicação podem ser utilizados por ele. Em uma Aplicação, eles são utilizados como filtros que determinam quais pares [Objeto de Dados, Procedimento Operacional] (pares processa) podem ser usados no desenvolvimento de um projeto ou subprojeto, ou no ambiente da Aplicação. Para o desenvolvedor, um objeto da classe Contexto (contexto) quando relacionado a um conjunto de pares processa determina: (1) quais Objetos de Dados descrevem os dados de um projeto ou subprojeto; e (2) a visibilidade dos Procedimentos Operacionais ligados como métodos dinâmicos a esses Objetos de Dados, isto é, somente os Procedimentos Operacionais relacionados aos pares processa contidos nesse contexto podem ser disponibilizados ao projetista (visibilidade pública), os demais são escondidos do mesmo (visibilidade privada). A Figura 4.7 mostra um exemplo de uma parte da Estrutura de Projeto da Figura 4.3, p.76, modelada na forma de um diagrama classe-objeto com todos os elementos estruturais mencionados. Essa forma de modelagem é descrita no Apêndice A.

Figura 4.7 – Exemplo de contexto de projeto

Para facilitar o desenvolvimento das Estruturas de Projeto e Estruturas das Aplicações, os objetos da classe Contexto foram classificados como:

- **contexto de projeto:** denominação genérica dada aos objetos da classe Contexto usados para relacionar um projeto ou subprojeto a seus componentes das aplicações; e
- **contexto da aplicação:** denominação genérica dada aos objetos da classe Contexto usados para relacionar uma Aplicação a seus componentes das aplicações.

As Estruturas de Projeto possuem ramificações que podem ser vistas como outras Estruturas de Projeto, definindo uma forma recursiva de construção (forma fractal). Isso é obtido quando um contexto contém um par processa que liga um Objeto de Projeto a um de seus Procedimentos Operacionais, normalmente usado para criar ou abrir um projeto ou subprojeto. A mesma técnica é usada para conectar uma Estrutura de Projeto a um contexto da aplicação. Nesse caso, o Objeto de Projeto que representa o elemento raiz da Estrutura de Projeto é relacionado ao par processa contido no contexto da aplicação.

Essa forma genérica de construção pode gerar alguns problemas que devem ser evitados. O primeiro é a referência cíclica que pode ocorrer entre Objetos de Projeto. O segundo é que todos os componentes das aplicações passam a necessitar de um contexto para existir. Isso limita o uso livre de operações auxiliares, como edição, visualização e impressão de objetos de dados que, em geral, independem de um contexto.

Para auxiliar na resolução desse segundo problema, foi definido um elemento chamado **contexto geral**. O contexto geral é um objeto da classe Contexto que contém todos os pares processa que podem ser disponibilizados livremente ao projetista. Esse objeto é reconhecido pelos mecanismos do *framework* como um de seus pontos fixos e, ao contrário dos demais contextos, ele não é usado por nenhum projeto, subprojeto ou Aplicação, mas sim pelos contextos de projeto e contextos das aplicações, que o utilizam para definir a visibilidade dos Procedimentos Operacionais ligados como métodos dinâmicos a um determinado Objeto de Dados. Quando um contexto aceita utilizar o contexto geral, os mecanismos do *framework* tornam públicos os Procedimentos Operacionais ligados a um Objeto de Dados que pertence a esse contexto e ao contexto geral. A Figura 4.8 mostra um exemplo de uso do contexto geral. Note que os Procedimentos Operacionais Visualizar Matriz e Editar Matriz, relacionados ao Objeto de Dados Matriz Peça-Máquina 0-1, estão contidos no contexto geral (Figura 4.8b), mas não no contexto Arranjo Físico Celular (Figura 4.8a). Nesse caso, esses Procedimentos Operacionais são vistos como públicos no contexto geral e como privados no contexto Arranjo Físico Celular. Caso o contexto Arranjo Físico Celular utilize (aceite) o contexto geral, esses Procedimentos Operacionais tornam-se públicos (Figura 4.8c) e passam a ser vistos como contidos no contexto Arranjo Físico Celular.

Figura 4.8 – Exemplo de uso do contexto geral

4.6. Considerações Finais

O levantamento e a análise dos requisitos descritos na seção 4.3 definiram elementos e características importantes para o desenvolvimento do software proposto. Apesar de sua classificação como um software do tipo pacote ou *toolkit* definir um comportamento e uma estrutura arquitetural fáceis de implementar a partir de uma abordagem tecnológica tradicional, alguns aspectos definidos durante a fase de análise inviabilizam essa abordagem e exigem uma estrutura de software que suporte a inclusão de novos componentes das aplicações sem sofrer modificações ou gerar alterações significativas no código fonte do software proposto.

A solução adotada para conferir à estrutura do software proposto essa flexibilidade, apresentada na seção 4.4, é baseada na forma de uso mais simples de um OOAF que busca construir uma Aplicação a partir de componentes existentes. Os elementos-chave dessa solução são as Estruturas das Aplicações e Estruturas de Projeto, implementadas na forma de roteiros, que descrevem aos mecanismos de controle do OOAF como utilizar os componentes das aplicações para construir e executar dinamicamente uma Aplicação com suas aplicações. Esses elementos combinados a uma ferramenta de apoio ao desenvolvimento (SGCA) permitem ao desenvolvedor compor novas aplicações para o software proposto de maneira simples e rápida, sem necessitar modificações estruturais ou alterações em seu código fonte. A Figura 4.9

ilustra essa solução cujos elementos, apresentados neste capítulo, podem ser compreendidos como:

- Framework de Aplicação Orientado a Objeto (OOAF): base da solução na qual são acoplados dinamicamente os componentes das diversas aplicações definidas para o software proposto. Seus mecanismos utilizam as informações contidas na BCA para determinar os componentes que devem ser acoplados a seus pontos variáveis para executar as Aplicações. A interação do projetista com o software proposto (Aplicação) gera uma configuração diferente a cada momento (aplicação) cujo estado de seus objetos de dados é preservado pelo ROD.
- Biblioteca de Componentes das Aplicações (BCA): pacote de componentes das aplicações com seus roteiros de uso que descrevem aos mecanismos de controle do OOAF como construir dinamicamente as Aplicações (instâncias do OOAF) e suas aplicações (utilização da Aplicação como software de apoio). A BCA é gerada pelo SGCA a partir das informações contidas na BCR.
- Software Gerenciador de Componentes das Aplicações (SGCA): software de apoio usado pelo desenvolvedor para registrar os componentes das aplicações, as Estruturas das Aplicações, e as Estruturas de Projeto em uma Biblioteca de Componentes Reusáveis (BCR). O SGCA utiliza essas informações de registro para gerar a BCA por solicitação do desenvolvedor.
- Repositório de Objetos de Dado (ROD): camada de software sobre um Sistema Gerenciador de Bancos de Dados (SGBD) utilizada para garantir a persistência dos objetos de dados das aplicações e seus relacionamentos. Pode ser visto como o repositório de dados das Aplicações.
- Estrutura da Aplicação: estrutura dinâmica usada pelos mecanismos de controle do OOAF para criar e executar uma Aplicação. Sua composição básica inclui: (1) a interface principal do software proposto – Interface Padrão; (2) um conjunto de operações auxiliares iniciais; e (3) um conjunto de Estruturas de Projeto. É modelada pelo desenvolvedor usando diagramas UML e registrada na BCR via SGCA para depois ser convertida nos roteiros de uso de componentes das aplicações da BCA.

- Estrutura de Projeto: estrutura dinâmica cuja composição determina quais componentes das aplicações podem ser empregados em um projeto de arranjo físico. É modelada pelo desenvolvedor usando diagramas UML e registrada na BCR via SGCA para depois ser convertida nos roteiros de uso de componentes das aplicações da BCA. É utilizada pela Aplicação para construir dinamicamente suas aplicações em resposta às solicitações do projetista.
- Componente da aplicação: denominação genérica dada a qualquer Objeto de Dados, Procedimento Operacional ou Interface com o Usuário construído pelo desenvolvedor para uma aplicação do software proposto. O desenvolvedor usa gabaritos e diagramas UML para a modelagem desses componentes e os codifica com a ajuda de modelos pré-definidos (*templates*) e exemplos de componentes existentes.
- Objeto de Dados: denominação genérica dos componentes usados para armazenar quaisquer dados dos projetos.
- Procedimento Operacional: denominação genérica dos componentes que implementam os algoritmos e operações auxiliares.
- Interface com o Usuário: denominação genérica dos componentes que implementam as interfaces de interação do projetista com os Procedimentos Operacionais.
- Desenvolvedor: representa qualquer pessoa responsável pela criação de componentes ou modelagem das estruturas dinâmicas com conhecimento suficiente da solução para gerar novas Aplicações ou efetuar modificações nas Aplicações existentes.
- Projetista: é o projetista de arranjo físico que atua como usuário do sistema.

Figura 4.9 – Síntese da solução computacional adotada.

CAPÍTULO 5. O *FRAMEWORK* ORIENTADO A OBJETO

5.1. Considerações Iniciais

No Capítulo 4, foram apresentados o problema e a solução computacional adotada com todos os seus elementos. Nessa solução, um *framework* de aplicação é utilizado como elemento tecnológico base que, conectado a dois outros elementos tecnológicos (BCA e ROD), é capaz de construir dinamicamente as Aplicações e preservar o estado de processamento dos objetos de dados.

O *framework* de aplicação utilizado no desenvolvimento do software proposto pode ser classificado como um *framework* de aplicação empresarial e foi pensado, inicialmente, para ser usado como um *framework* caixa-cinza. No entanto, seu projeto arquitetural e estrutural permitem que ele possa ser utilizado no futuro como um *framework* caixa-preta, efetuando poucas alterações.

Neste capítulo é apresentado o projeto desse *framework*, e são detalhados o ROD, a BCA, o SGCA e a BCR. Também é descrita a forma de uso desse *framework* e são apresentados alguns aspectos de sua implementação para auxiliar na compreensão dos Capítulos 6 e 7.

Deste capítulo em diante, algumas considerações devem ser feitas para compreender detalhes do projeto e da implementação do software proposto.

Toda a codificação do software proposto foi feita usando termos e palavras da língua inglesa, adotada como linguagem padrão no desenvolvimento do software proposto. Isso se reflete nos processos de modelagem, uma vez que a UML foi adotada como padrão e seus diagramas de classes, construídos durante a fase de projeto, são complementados durante a fase de implementação. Sendo assim, esses diagramas mostram nomes de classes com seus atributos e métodos na língua inglesa. Por outro lado, durante a análise do problema e fase de projeto conceitual do software proposto, vários termos e elementos foram definidos na língua portuguesa e sua utilização torna

mais fácil a compreensão dos textos descritivos da solução. Para facilitar a relação entre os elementos descritos na língua portuguesa e suas implementações na língua inglesa, o mecanismo de estereotipagem da UML foi utilizado nos diagramas de classes para mostrar o nome do elemento em português.

5.2. Estrutura do *Framework*

O *framework* de aplicação usado no desenvolvimento do software proposto é dividido em duas partes: uma denominada ***framework de operação*** e outra denominada ***framework da Aplicação***.

O ***framework de operação*** é o *framework* principal. Todas as Aplicações são instanciadas a partir dele. Como seus pontos variáveis são muito genéricos e difíceis de usar, o desenvolvedor pode ter dificuldade para implementar uma Aplicação sem o auxílio de exemplos e gabaritos. Por outro lado, uma vez estruturada uma Aplicação, apenas suas Estruturas de Projeto costumam sofrer alterações, pois, sua Interface Padrão e suas operações auxiliares iniciais, uma vez construídas, pouco irão mudar. Pensando nisso, foi elaborado um segundo *framework*, denominado ***framework da Aplicação***, que pode ser visto como uma generalização de uma Aplicação. Seus pontos variáveis são definidos a partir de especializações das classes dos pontos variáveis do *framework* de operação, e suas derivadas concretas são reconhecidas e disponibilizadas pela Interface Padrão de acordo com sua lógica de organização.

O *framework* de operação implementa os blocos Núcleo de Controle, ROD, SGBD e BCA da arquitetura do software proposto mostrada na Figura 4.2, p.73. Ele recebeu esse nome por desempenhar três funções operacionais básicas do software proposto: (1) execução segura de procedimentos operacionais; (2) instanciação e persistência dos objetos de dados tomados como parâmetro por esses procedimentos operacionais; e (3) disponibilização dos componentes das aplicações encaixados na Estrutura de Projeto em uso. Essas três funções são complexas e utilizam vários mecanismos de controle. Por essa razão, o Núcleo de Controle foi decomposto em quatro módulos funcionalmente independentes, cada qual responsável por desempenhar uma parte dessas funções. Esses módulos são detalhados na seção 5.3.

A Figura 5.1 mostra a estrutura do *framework* de operação na forma de blocos sobrepostos em camadas e a relação desses blocos com os blocos da arquitetura do software proposto. Os blocos CIU, MCO, GOD e GPI representam os quatro módulos

que compõem o Núcleo de Controle da arquitetura e formam o núcleo do *framework* de operação. Os demais blocos representam o ROD, o SGBD e a BCA apresentados na seção 4.5.1. Além dos módulos, a Figura 5.1 também mostra, de maneira simbólica, os três tipos de componentes das aplicações, dentro do bloco da BCA, ligados às Estruturas das Aplicações e Estruturas de Projeto, descritas na forma de roteiros de uso.

Figura 5.1 – Estrutura do *framework* de operação.

O *framework* da Aplicação possui uma estrutura básica definida a partir dos elementos que compõem uma Estrutura da Aplicação. Essa estrutura básica é composta pela Interface Padrão e um conjunto de classes genéricas e específicas derivadas dos pontos variáveis do *framework* de operação. As classes genéricas são tratadas como seus pontos variáveis e são reconhecidas pela Interface Padrão que utiliza suas descrições para disponibilizar suas derivadas concretas (componentes das aplicações) ao projetista. As classes específicas definem os pontos fixos do *framework* da Aplicação. Elas implementam as operações auxiliares iniciais da Aplicação e um conjunto de componentes das aplicações que são compartilhados pelas Estruturas de Projeto utilizadas nas aplicações do software proposto.

A Figura 5.2 mostra a estrutura de um *framework* da Aplicação, na forma de blocos sobrepostos, e sua relação com os blocos Aplicação da arquitetura do software proposto. O bloco *Framework* da Aplicação *i* representa o *framework* da Aplicação definido para a Aplicação *i*, que, como descrito anteriormente, contém a Interface Padrão da Aplicação *i* e um conjunto de classes genéricas e específicas que representam seus pontos variáveis e fixos. Seus pontos fixos, por serem componentes das aplicações, são encaixados à Estrutura da Aplicação *i*, descrita na forma de roteiros de uso. Todos esses elementos são ilustrados na Figura 5.2 pelos retângulos dentro do bloco Pontos

Variáveis e Fixos do *Framework* da Aplicação *i*. Os blocos Estrutura de Projeto representam as Estruturas de Projeto descritas para a Aplicação *i*. Os componentes das aplicações encaixados nessas Estruturas de Projeto são derivadas concretas dos pontos variáveis do *framework* da Aplicação *i*. Esses componentes são ilustrados na Figura 5.2 pelos retângulos dentro do bloco Estrutura de Projeto *i,j*.

Figura 5.2 – Estrutura de um *framework* da Aplicação.

5.3. Projeto do *Framework* de Operação

Nesta seção, será apresentado o projeto conceitual do *framework* de operação com todos os seus módulos e funcionalidades, descritos nas seções subsequentes. Alguns diagramas da UML criados durante a fase de projeto foram utilizados para ilustrar a modelagem e a descrição funcional de cada módulo, bem como seu comportamento.

5.3.1. Visão Geral

Os módulos que compõem o *framework* de operação citados na seção 5.2 foram projetados para serem implementados como classes de objetos, cujo diagrama de classes é mostrado na Figura 5.3. O SGBD é um sistema completo adquirido de terceiros apenas para armazenar dados. Por essa razão, ele aparece no diagrama como uma classe de utilidades estereotipada. O pacote Classes das Aplicações foi colocado nesse diagrama apenas para ilustrar o conteúdo da BCA mostrado na Figura 5.1.

Figura 5.3 – Diagrama de Classes dos módulos do *framework* de operação.

A classe *TUserInterfaceController* implementa o módulo Controlador de Interfaces com o Usuário (CIU). O CIU é responsável pelo controle e execução de todas as interfaces com o usuário usadas para interação do projetista com o software de apoio. Ele foi projetado para encapsular aspectos e detalhes de interação do núcleo com o projetista ou sistemas externos, definindo uma interface padronizada para isso.

A classe *TOperatingControlEngine* implementa o módulo Mecanismo de Controle Operacional (MCO). O MCO é o módulo de controle principal do *framework*. Ele é responsável pela execução dos procedimentos operacionais, e pelo controle das Aplicações em execução. O GOD e o GPI são agregados por valor ao MCO por serem encapsulados totalmente por ele, e são considerados como extensões de seus mecanismos. Isso foi pensado para permitir que o núcleo de controle do *framework* possa ser, no futuro, separado do CIU, permitindo acesso seguro e simultâneo aos dados dos projetos em diversos tipos de interfaces diferentes, como interfaces gráficas, texto, HTML, etc.

A classe *TDataObjectManager* implementa o módulo Gerenciador de Objetos de Dados (GOD). O GOD é responsável pela persistência e controle dos estados de processamento de todos os objetos de dados. Ele utiliza a BCA para instanciar os objetos de dados solicitados pelo MCO, e o ROD para recuperá-los do banco de dados

ou para criá-los. Somente o MCO tem acesso ao GOD, no entanto, existem alguns Objetos de Dados reservados que também podem acessá-lo e são usados pelos procedimentos operacionais e procedimentos de seleção para obter as árvores dos projetos e conjuntos de objetos de dados. Existem também alguns Objetos de Dados reservados que são controlados pelo GPI, mas, como o MCO não os distingue, são solicitados ao GOD que os encaminha para o GPI. O GOD encapsula totalmente o ROD, por isso ele foi agregado por valor.

A classe *TProcedureAndInterfaceManager* implementa o módulo Gerenciador de Procedimentos e Interfaces (GPI). O GPI é responsável pela instanciação dos procedimentos operacionais, procedimentos de seleção, e interfaces com o usuário usados pelo MCO e CIU. Ele também gera listas e árvores de componentes das aplicações usadas pela Aplicação para disponibilizar ao projetista os componentes que podem ser empregados no desenvolvimento de um projeto ou subprojeto. Para isso, ele utiliza as informações contidas na BCA na forma de roteiros de uso.

A classe *TDataObjectRepository* implementa o módulo Repositório de Objetos de Dados (ROD) descrito na seção 4.5.1. O ROD é responsável pela persistência de todos os objetos de dados e define uma forma padronizada de acesso ao SGBD em uso.

A classe *TApplicationComponentLibrary* implementa a camada de software sobre a BCA, como descrito na seção 4.5.1. Como as informações contidas na BCA são integradas ao código fonte do software proposto, essa camada de software foi considerada como um módulo do núcleo do *framework* e recebeu o mesmo nome. A BCA é responsável pela instanciação de todos os componentes das aplicações e define uma forma padronizada de acesso aos roteiros que independe de implementação.

Cada módulo, assim como os pontos variáveis do *framework*, possuem métodos e propriedades que definem suas interfaces, cujos detalhes são apresentados no Apêndice B. Para garantir a integridade e segurança do núcleo do *framework* e assegurar seu bom funcionamento, foram adotadas duas formas de comunicação, ou interação, entre objetos:

1. **direta:** um objeto acessa diretamente os métodos e propriedades do outro objeto;
2. **por mensagem:** um objeto envia uma mensagem, via sistema operacional, a outro objeto solicitando um serviço.

Por via de regra, um componente da aplicação não tem acesso direto ao núcleo do *framework*. Isso é feito via mensagens que são controladas pelo núcleo do *framework* e direcionadas de maneira segura aos módulos do *framework*. Com exceção dessa regra, módulos e componentes utilizam mais freqüentemente a forma de comunicação direta, reservando a comunicação por mensagem para envio de mensagens de controle e atualizações de estado, ou em situações nas quais a comunicação direta pode gerar instabilidade na Aplicação ou comprometer sua integridade.

Para que o núcleo do *framework* possa reconhecer todos os objetos, foi definida uma superclasse chamada *TObjectClass* da qual são derivadas todas as classes que compõem o *framework* e suas Aplicações. Essa superclasse define a interface usada pelo núcleo do *framework* para recepção e envio de mensagens. A Figura 5.4 mostra o diagrama de classes da superclasse *TObjectClass* com suas duas superclasses derivadas *TModule* e *TSysObject*. A superclasse *TModule* é usada para derivar todos as classes que implementam os módulos do *framework* e define duas propriedades de controle entre módulos: *Active* e *LastError* que indicam, respectivamente, se o módulo foi inicializado e a mensagem do último erro ocorrido após a chamada de um serviço não realizado. A superclasse *TSysObject* (Objeto do Sistema) é usada para derivar todas as classes descritas no Metamodelo e será descrita na seção 5.3.2.

Figura 5.4 – Superclasses do *framework*.

Nas seções subsequentes, cada módulo e sua interação com os demais módulos e com os componentes das aplicações são descritos em detalhes. Para facilitar a compreensão dessa interação com os mecanismos e a lógica de controle do *framework* de operação, uma descrição de seu funcionamento básico é feita a seguir.

A função principal do *framework* de operação é executar, de maneira segura, os procedimentos operacionais, fornecendo-lhes os parâmetros necessários, executando suas interfaces com o usuário e garantindo a persistência dos objetos de dados utilizados em sua execução. A Figura 5.5 ilustra essa função, de maneira simplificada, através de um diagrama de sequência que mostra a interação dos módulos do *framework* de operação com a Interface Padrão e o Projetista.

Figura 5.5 – Diagrama de sequência para a execução de um procedimento operacional.

Quando o projetista seleciona um procedimento operacional via Interface Padrão, um serviço do MCO é chamado para executá-lo. O MCO solicita ao GPI uma instância desse procedimento operacional, uma instância de sua interface com o usuário de

controle, e instancias dos procedimentos de seleção. Os objetos de dados tomados como parâmetros de entrada são solicitados através dos procedimentos de seleção cujas interfaces com o usuário são enviadas ao CIU. Feito isso, os objetos de dados selecionados são solicitados ao GOD e relacionados a seus parâmetros. Em seguida, os objetos de dados tomados como parâmetros de saída são criados, via GOD, e relacionados a seus parâmetros. Por fim, o MCO envia a interface com o usuário de controle ao CIU para ser exibida enquanto o procedimento operacional é executado. Ao final da execução, os históricos de processamento são registrados, os objetos de dados modificados são atualizados, e as mudanças de estado comunicadas ao ambiente operacional via MCO. Se existirem procedimentos de visualização associados aos parâmetros de saída, estes serão executados, seguindo os mesmos passos descritos neste parágrafo.

5.3.2. Aplicação do Metamodelo no Projeto do *Framework*

O metamodelo foi utilizado no projeto do *framework* de operação para modelar seus pontos variáveis, e definir a lógica básica de controle dos mecanismos do núcleo e sua interação com os componentes das aplicações. Os pontos variáveis do *framework* de operação foram modelados a partir das classes: Objeto de Dados, Procedimento Operacional, Interface com o Usuário, Procedimento de Seleção, Variável, e Objeto de Projeto.

As classes Contexto, Objeto da Aplicação e Parâmetro, também foram modeladas, e seus objetos são utilizados pelos mecanismos de controle do núcleo como pontos fixos. Toda vez que um objeto de projeto é instanciado, automaticamente a BCA associá-o ao seu contexto de projeto. A classe Processa é considerada apenas pelo SGCA. No processo de conversão da BCR para BCA ela é tratada como um relacionamento triplo entre Contexto, Objeto de Dados e Procedimento Operacional registrado em uma única tabela. Os objetos dessas classes são criados pelo SGCA, durante o cadastramento dos componentes das aplicações, das Estruturas das Aplicações e das Estruturas de Projeto, e registradas na BCR para depois serem convertidos em elementos dos roteiros de uso da BCA. Esse processo pode ser melhor compreendido a partir do exemplo ilustrado pela Figura 5.6 e pela Figura 5.7.

Figura 5.6 – Exemplo de um diagrama classe-objeto para o algoritmo ROC.

A Figura 5.6 mostra um diagrama classe-objeto criado para registrar o algoritmo ROC. Os estereótipos (nomes entre << >>) são usados para indicar o nome da classe ancestral do componente da aplicação e o nome da classe do objeto. Os números que aparecem nos relacionamentos Agrega indicam a ordem dos parâmetros. Esse diagrama é registrado na BCR via SGCA para depois ser convertido em um conjunto de informações depositadas na tabelas da BCA para serem usadas pelos mecanismos de controle do *framework* na forma de roteiros. A Figura 5.7 mostra essas tabelas com as informações do diagrama da Figura 5.6. Alguns campos não relevantes a essa exemplificação foram omitidos para facilitar a compreensão.

Na Figura 5.7, a tabela *Class* contém os registros de todas as classes que descrevem os componentes das aplicações, cuja coluna *AncestorClassID* é usada para estabelecer os relacionamento de herança. As demais tabelas são usadas para registrar os elementos estruturais e seus relacionamentos. A tabela *Control* contém os registros do relacionamento Controla. A tabela *Parameter* contém o registro de todos os objetos da classe *TParameter* e seus relacionamentos Aceita (*DataObjectClassID*), Seleciona (*SelprocObjectID*) e Exibe Seleção (*SelprocObjectID*). A tabela *SelViewProcedure* contém o registro dos Procedimentos de Seleção e Visualização relacionados aos parâmetros. A tabela *Aggregate* contém os registros do relacionamento Agrega. A tabela *Contain* contém os registros dos relacionamentos Contém e Processa combinados na forma de um relacionamento triplo. A tabela *Context* contém todos os objetos da classe *TContext*. Por fim, a tabela *Use* contém os registros do relacionamento Usa entre Objetos de Projeto e seus contextos de projeto.

Class		
ClassID	Name	AncestorClassID
1.0.0.0.150	Matriz Peça-Maquina 0-1	1.0.0.0.0
1.1.0.0.160	Arranjo Físico Celular	1.1.0.0.0
2.0.0.0.170	Visualizar Matriz 0-1	2.0.0.0.0
2.0.0.0.180	ROC	2.0.0.0.0
3.0.0.0.181	Interface ROC	3.0.0.0.0

Control	
OpPrococedureClassID	UserInterfaceClassID
2.0.0.0.180	3.0.0.0.181

Parameter					
ObjectID	Name	Type	DataObjectClassID	SelProcObjectID	ViewProcObjectID
101	Matriz 0-1 de Entrada	{in}	1.0.0.0.150		
102	Matriz 0-1 de Saída	{out}	1.0.0.0.150		103

SelViewProcedure		
ObjectID	ClassID	Name
103	2.0.0.0.170	Visualizar Matriz 0-1

Aggregate		
OpProcedureClassID	ParamObjectID	Order
2.0.0.0.180	101	1
2.0.0.0.180	102	2

Contain		
ContextObjectID	DataObjectClassID	OpProcedureClassID
100	1.0.0.0.150	2.0.0.0.180

Context	
ObjectID	Name
100	Arranjo Físico Celular

Use	
ProjectClassID	ContextObjectID
1.1.0.0.160	100

Figura 5.7 – Tabelas da BCA com as informações sobre o diagrama classe-objeto do algoritmo ROC da Figura 5.6.

Apenas a BCA tem acesso a essas tabelas que as utiliza para fornecer aos outros módulos do *framework* os roteiros de uso e as instâncias dos componentes das aplicações. Esses roteiros são disponibilizados na forma de uma estrutura, denominada **Estrutura de Roteiro**, cujos detalhes de sua implementação e construção são descritos na seção 5.3.8. Essas estruturas são organizadas na forma de árvores cujos nós contêm os elementos do Metamodelo (pontos fixos e variáveis) na forma de objetos de referência, e cujas ramificações representam os relacionamentos entre esses elementos.

Um **objeto de referência** é um objeto instanciado pela BCA que contém informações sobre uma classe ou um objeto e é usado pelos mecanismos de controle do *framework* para construir dinamicamente uma Aplicação com suas diversas aplicações. Objetos de referência contêm apenas informações sobre os elementos do Metamodelo e suas derivadas. Para facilitar a implementação das Estruturas de Roteiro, dos objetos de referência e dos mecanismos do *framework*, todos os elementos do Metamodelo foram

modelados como classes derivadas de uma única superclasse chamada **Objeto do Sistema** (*TSysObject*). A Figura 5.8 mostra o diagrama de classes dos pontos variáveis e classes auxiliares do *framework* de operação. Os objetos de referência são instâncias da classe *TObjectRef* que é uma derivada concreta de *TSysObject* e implementa apenas seus métodos abstratos e redefine o comportamento do método *IsReference*. Os mecanismos do *framework*, em geral, trabalham com a interface definida por *TSysObject* e utilizam o método *IsReference* para determinar se foi enviado como parâmetro um objeto ou um objeto de referência. Os demais métodos *IsClassRef*, *IsObjectRef* e *IsObject*, utilizam o valor retornado por *IsReference* para determinar se é uma referência a uma classe ou objeto, ou se é um objeto.

Além dos métodos *Is*, os mecanismos do *framework* também utilizam os atributos de *TSysObject* para identificar as classes e seus objetos. Os atributos *Name* e *Description* são definidos durante o registro dos componentes das aplicações na BCR e são ignorados pelo *framework* de operação, mas foram definidos para que as Aplicações possam disponibilizar seus componentes ao projetista como, por exemplo, itens de um menu em uma interface gráfica padrão Windows[®], como comando de linha em uma Aplicação tipo console, *hiperlinks* HTML em Aplicações WWW, dentre outros.

De todos, o atributo *ClassID* merece destaque. Ele foi projetado para ser um identificador baseado em código hierárquico composto por cinco números separados por pontos, como comentado na Figura 5.8. Os primeiro quatro números variam de 0 a 15 e são utilizados para identificar as classes dos pontos variáveis do *framework* e suas derivadas, e as classes auxiliares do Metamodelo. Cada número representa um nível de generalização. Níveis com valor igual a 0 não possuem classes. Por exemplo, a classe *TDataObject* possui identificador igual a 1.0.0.0.0; isso significa que ela está no primeiro nível e seu número é 1. Por outro lado, a classe *TVariable* possui identificador igual a 1.2.0.0.0; isso significa que ela está no segundo nível, seu número é 2, e seu ancestral é a classe *TDataObject*. O último nível identifica a classe concreta de um ponto variável (componente da aplicação) e seu número pode variar de 100 a 65535 – os valores de 0 a 99 são reservados para uso do *framework*. Cada componente da aplicação recebe um identificador (*ID*) único, gerado pelo SGCA, que o distingue dos demais componentes independente dos quatro primeiros identificadores hierárquicos. O *ClassID* definido para os pontos fixos e variáveis do *framework* são mostrados na

Figura 5.8 como valores atribuídos. O caractere “/” à esquerda do atributo *ClassID* indica que ele é herdado da classe ancestral *TSysObject*.

A formação do valor do atributo *ClassID* propicia ao *framework* de operação e suas Aplicações identificarem os pontos variáveis, e define uma forma de organização e classificação que pode ser utilizada pela Interface Padrão para disponibilizar os componentes das aplicações ao projetista.

Figura 5.8 – Diagrama de classes dos pontos variáveis e classes auxiliares do *framework*.

A classe *TSysObject* define também três métodos abstratos utilizados pelos mecanismos do *framework* para estabelecer os relacionamentos entre os componentes das aplicações: *GetRelationship*, *RelationshipCount* e *SetRelationship*. Esses métodos definem uma forma genérica para adicionar ou remover relacionamentos entre componentes das aplicações que independe de implementações específicas, e dá liberdade ao desenvolvedor para poder interferir nesse processo e definir suas próprias propriedades. Esses relacionamentos são definidos pelo Metamodelo, e cada derivada da

superclasse *TSysObject* pré-define um comportamento para esses métodos que aceita os relacionamentos que ele pode estabelecer. Por exemplo, a superclasse *TOperatingProcedure* pré-define propriedades para aceitar os relacionamentos Controla, Agrega, e Processa; enquanto que a classe *TParameter*, sendo um ponto fixo, define propriedades para aceitar os relacionamentos Aceita, Selecciona e Exibe Resultado. Em geral, esses métodos são muito utilizados pelos mecanismos do GPI e da BCA. Já o MCO, os utiliza apenas com os componentes das aplicações, pois as classes auxiliares, por serem consideradas pontos fixos do *framework*, definem propriedades para isso.

5.3.3. Controlador de Interfaces com o Usuário (CIU)

O CIU foi projetado para controlar a execução das interfaces com o usuário e encapsular detalhes do ambiente em que as interfaces serão executadas. Ambientes gráficos, como os oferecidos pela maioria dos sistemas operacionais (SO) atuais (Windows® da Microsoft, o MacOS® da Apple, Linux®, dentre outros) possuem interfaces gráficas baseadas em janelas que definem uma forma padronizada de interação do usuário com um sistema computacional. Em geral, esses ambientes gráficos trabalham com o conceito de eventos. Todas as ações do usuário sobre o ambiente são transformadas em eventos que são enviados às rotinas do programa em execução que as traduzem em chamadas a rotinas de processamento de dados. Essa lógica pode ser implementada de muitas maneiras diferentes e muitas das linguagens de programação visuais, como VB e VC++ da Microsoft e C++ Builder e Delphi da Borland, possuem componentes prontos que encapsulam essa lógica e chamam automaticamente rotinas definidas pelo programador para tratar os eventos. Por essa razão, o papel do CIU é encapsular esses e outros detalhes de programação, e definir um padrão próprio para execução e tratamento de mensagens que independa do ambiente gráfico ou de detalhes da linguagem de programação utilizada. Por essa razão, a forma de interação entre o CIU e as Interfaces com o Usuário pode variar de um ambiente para outro. Assim sendo, suas interfaces definem apenas os métodos estritamente necessários ao controle do *framework*. Esses métodos são mostrados na Figura 5.9.

A interação do CIU e dos demais módulos com as interfaces com o usuário é direta. Já o contrário é sempre por mensagens. A interação entre as interfaces com o usuário e seus procedimentos operacionais em geral é feita por mensagem, uma vez que costumam ser mensagens de controle. No entanto, algumas interfaces, como as das

operações de edição, visualização e impressão, costumam utilizar a interação direta para evitar problema de fluxo de mensagens devido ao volume de dados que devem tratar. Por fim, a interação do CIU com o MCO é sempre por mensagens, uma vez que apenas mensagens de controle são enviadas ao MCO.

Figura 5.9 – Diagrama de classes do CIU com a Interface com o Usuário.

Ao contrário dos demais módulos, o CIU é instanciado pela Aplicação e associado ao MCO. Isso foi definido para que a Aplicação possa definir os parâmetros de inicialização necessários à interação do CIU com o ambiente no qual as interfaces com o usuário serão executadas.

5.3.4. O Mecanismo de Controle Operacional (MCO)

O MCO foi projetado para ser o núcleo (*kernel*) do *framework* de operação. Seu papel principal é executar os procedimentos operacionais solicitados pelo projetista via Interface Padrão. Para isso ele utiliza o GPI, o GOD e a CIU. A Figura 5.10 mostra esses módulos com os métodos usados pelo MCO para executar os procedimentos operacionais. O GPI é usado para obter o procedimento operacional a ser executado, seus parâmetros e procedimentos de seleção e visualização, e sua interface com o usuário de controle. O GOD é usado para criar os objetos de dados tomados como parâmetros de saída, recuperar os objetos de dados tomados como parâmetros de entrada, e armazenar todos os objetos de dados modificados durante a execução do procedimento operacional. A CIU, como mencionada na seção anterior, é usado para executar as interfaces com o usuário, inicialmente para obter os parâmetros de entrada, depois para controlar a execução do procedimento operacional, e por fim para exibir os resultados do processamento.

Uma das características do núcleo do *framework* é a execução simultânea de vários procedimentos operacionais. Isso possibilita ao projetista executar várias análises simultaneamente e poder comparar seus resultados. Para que isso ocorra de maneira

segura, foram criadas as sessões seguras, e adotado o princípio de compartilhamento entre objetos de dados, que é assegurado pelos mecanismos do GOD.

Figura 5.10 – Diagrama de classes do MCO com o CIU, o GOD e o GPI.

As sessões seguras garantem a integridade dos dados dos projetos e possibilitam a execução simultânea de vários procedimentos operacionais. Existem quatro tipos de sessões seguras que são implementadas pelas classes:

1. ***TSecuritySession*** é a classe base para a implementação de todas as sessões seguras e é utilizada para instanciar a Sessão Geral.
2. ***TApplicationSession*** é usada para garantir a segurança das Aplicações em execução. Ela permite que mais que uma Aplicação possa ser executada ao mesmo tempo. Suas instâncias são denominadas genericamente de sessão da Aplicação.
3. ***TProjectSession*** é usada para garantir a segurança de um projeto. Ela permite que vários projetos sejam desenvolvidos ao mesmo tempo ou que o mesmo projeto seja desenvolvido simultaneamente em várias Aplicações, compartilhando suas alterações. Suas instâncias são denominadas genericamente de sessão de projeto.
4. ***TProcessingSession*** é usada para garantir a segurança de um procedimento operacional em execução. Suas instâncias são denominadas genericamente de sessão de processamento e cada uma é executada como um *Thread*.

A Figura 5.11 mostra o diagrama de classes com essas sessões seguras, com seus métodos e propriedades, relacionadas ao MCO e à classe Procedimento Operacional, também com seus métodos e propriedades utilizados nos processos de execução.

Figura 5.11 – Diagrama de classes do MCO com suas sessões seguras e o Procedimento Operacional.

O MCO trabalha com as sessões seguras de maneira hierárquica agregando uma sessão segura a outra. Cada sessão segura é responsável pelos componentes das aplicações e sessões seguras em seu interior. Quando ela é fechada, ela fecha automaticamente as sessões seguras em seu interior, e libera todos os componentes das aplicações sob seu controle, preservando os estados atuais de processamento dos objetos de dados. Esse processo garante que ao fechar uma sessão de projeto, por exemplo, todos os procedimentos operacionais, interfaces com o usuário, e objetos de dados que estão sendo usados por esse projeto serão liberados. A Figura 5.12 mostra a hierarquia definida para as sessões seguras.

A **Sessão Geral** é a primeira sessão segura criada pelo MCO. Ela é usada para organizar e manter todas as demais sessões criadas para executar as Aplicações. Está sessão só é finalizada quando o MCO é finalizado.

Uma **sessão da Aplicação** é criada pelo método *Run()* do MCO para executar uma Aplicação. Ela é usada pelo MCO para manter informações relativas a uma Aplicação e controlar as sessões de projeto e de processamento executadas por solicitação de sua Interface Padrão. Esta sessão é finalizada quando a sessão de processamento na qual a Interface Padrão está sendo executada termina.

Figura 5.12 – Organização hierárquica das sessões seguras.

Uma **sessão de projeto** é criada sempre que um projeto é aberto pelo método *MCO.OpenProject()*. Ela é usada pelo MCO para manter informações relativas ao projeto e controlar as sessões de processamento usadas para executar os procedimentos operacionais utilizados no desenvolvimento desse projeto. Esta sessão é finalizada quando o projeto é fechado pelo método *MCO.CloseProject()*.

Uma **sessão de processamento** é criada pelo método `MCO.Execute()` para executar um procedimento operacional. Ela permite que vários procedimentos operacionais sejam executados em seu interior, mas somente um de cada vez. Em princípio, todos os objetos no interior de uma sessão de processamento podem ser compartilhados entre seus procedimentos operacionais, com exceção dos próprios procedimentos operacionais. Isso significa que a mesma interface com o usuário pode ser utilizada por vários procedimentos operacionais sem necessitar ser fechada, destruída e recriada. Isso permite ao desenvolvedor definir uma Interface com o Usuário que possa ser utilizada em vários processamentos subseqüentes mantendo sempre a mesma interface com o projetista, como é o caso das operações de visualização e edição em que a troca de uma para outra em uma aplicação não afeta o estado visual dos dados. Para que isso seja feito, existem regras definidas pela sessão de processamento e pelo MCO. Suponha que uma sessão de processamento S_1 tenha sido criada para executar um procedimento operacional P_1 para processar o objeto de dados O_1 utilizando a interface com o usuário I_1 . Se I_1 solicitar que um outro procedimento operacional (P_2) seja executado dentro de S_1 , o MCO tentará suspender a execução de P_1 via $S_1.Suspend()$ que encapsula $P_1.Suspend()$. Se P_1 não aceita ser suspenso, uma nova sessão de processamento (S_2) é criada para executar P_2 . Nesse caso, todos os objetos de dados necessários à execução de P_2 serão novamente solicitados e/ou criados e uma nova interface com o usuário será criada. No entanto, se P_1 aceita ser suspenso, seu estado atual é armazenado por S_1 e P_2 é executado, passando a compartilhar todos os objetos no interior de S_1 , inclusive a interface com o usuário aberta (I_1). Quando P_2 termina sua execução, P_1 tem seu estado de processamento restaurado pelo MCO via $S_1.Restore()$, e é reiniciado via $S_1.Resume()$ que encapsula $P_1.Resume()$. Nesse processo, uma sessão de processamento só é finalizada quando todos os procedimentos operacionais em seu interior finalizarem suas atividades. Nesse momento, todos os objetos alocados para ela são liberados.

Em geral, não há restrições sobre a sessão segura na qual uma sessão de processamento será agregada. No entanto, como um objeto de dado deve estar sempre vinculado a um projeto ou subprojeto para poder ser usado, é conveniente que um procedimento operacional processe esse objeto de dados dentro do ambiente do projeto. No entanto, quando se deseja abrir um projeto ou criar um projeto novo, é usado o ambiente da sessão da Aplicação. A Sessão Geral também pode ser usada para agregar

uma sessão de processamento, no entanto, ela é reservada para agregar as sessões das Aplicações.

Outra utilização importante das sessões seguras, é a difusão de alterações em objetos de dados. Como os objetos de dados são compartilhados, quando um deles (**O₁**) é modificado em alguma sessão de processamento, todas as sessões de processamento que utilizam **O₁** necessitam ser comunicadas para que seus procedimentos operacionais recebam essas modificações e atualizem seus estados de processamento. Esse processo é controlado pelo GOD que recebe como parâmetro em seus métodos a sessão segura na qual um objeto de dados será utilizada. Como regra geral, toda vez que o MCO necessita de um objeto de dados, digamos **O₁**, ele o solicita ao GOD via *LoadDataObject()*, *EditDataObject()* ou *NewDataObject()*, enviando-lhe a sessão segura na qual ele será utilizado. O GOD mantém uma lista com todas as sessões seguras para as quais **O₁** foi solicitado. Quando **O₁** é atualizado pelo MCO via *GOD.PostDataObject()*, o GOD envia um mensagem de atualização a todas as sessões seguras na lista comunicando as alterações em **O₁**.

O principal método do MCO é *Execute()*. É ele que executa todos os procedimentos operacionais. Nesse processo duas derivadas da superclasse Objeto de Dados recebem tratamento diferenciado: Objeto de Projeto e Variável.. O MCO utiliza os objetos de projeto para vincular os objetos de dados criados para a execução de um procedimento operacional. Isso é feito pelo próprio GOD quando um novo objeto de dados é cria via *NewDataObject()*. Com isso , um objeto de dados é automaticamente vinculado a uma árvore de projeto. Por outro lado, variáveis não são consideradas dados de um projeto, elas são utilizadas apenas para fornecer valores simples como parâmetro de um procedimento operacional, e como extensões dos objetos de dados. No entanto, elas são vinculadas aos históricos de processamento registrados pelo método *GOD.RegisterHistory()*. Por essa razão, toda vez que um procedimento operacional cujos parâmetros incluem derivadas da superclasse Variável é executado, o MCO cria essas variáveis independente de serem tomadas como parâmetros de entrada e/ou de saída.

Os demais métodos do MCO, *Run()*, *NewProject()*, *NewSubproject()*, *OpenProject()* e *CloseProject()*, encapsulam o método *Execute()* em seus processos. Apesar desses métodos serem públicos para a Aplicação, por questões de segurança, eles não podem ser acessados diretamente por seus componentes. Por essa razão, a

chamada desses métodos pelos procedimentos operacionais, principalmente pela Interface Padrão, é feito via sessão de processamento. Quando um procedimento operacional necessita executar um outro procedimento operacional, ou criar, abrir ou fechar um projeto, ele envia uma mensagem a sua sessão de processamento que a trata e redireciona para o MCO diretamente ou via sistema operacional. Isso garante que não ocorrerá qualquer tipo de instabilidade na execução da Aplicação. Convém lembrar que o *framework* é multitarefa. Isso significa que vários processos podem estar concorrendo pelos mesmos recursos. Se essa concorrência não for bem sincronizada, um procedimento operacional pode gerar travamento do sistema ou *deadlock* (como é mais conhecido na área de sistemas operacionais), ou modificar dados de maneira desordenada, comprometendo a integridade dos mesmos.

5.3.5. O Gerenciador de Objetos de Dados (GOD)

O GOD foi projetado para gerenciar os objetos de dados e seus relacionamentos de vínculo (ver seção 5.3.9, pg. 121), e para controlar suas mudanças de estado. Nesse processo, o GOD utiliza o ROD, o GPI e a BCA. O ROD é encapsulado pelo GOD que o considera como uma forma de interface padronizada de acesso aos objetos de dados e seus relacionamentos armazenados em um banco de dados. O GPI é utilizado para encaminhar solicitações de criação e carga de objetos de dados reservados de acesso aos componentes. A BCA é usada pelo GOD para instanciar objetos de dados e objetos de referência aos mesmos. A Figura 5.13 mostra esses módulos com seus métodos utilizados pelo GOD em seus processos. Nesta seção será apresentada apenas a forma de gerenciamento dos objetos de dados. A forma de gerenciamento dos relacionamentos de vínculo será apresentada na seção 5.3.9 para facilitar a compreensão da mesma.

O GOD mantém os objetos de dados, em uso pelos procedimentos operacionais em execução, na memória da máquina e controla seu compartilhamento e alterações de estado, comunicando quaisquer mudanças às sessões seguras para as quais foram alocados. As sessões seguras, por sua vez, comunicam os procedimentos operacionais em seu interior para que estes possam solicitar atualização do objeto de dados modificado. Quando não há mais nenhuma sessão segura utilizando um objeto de dados, este é automaticamente liberado da memória, passando a ser mantido apenas no banco de dados. Isso é feito para agilizar o acesso às propriedades do objeto de dados e ao mesmo tempo para poder compartilhá-lo.

Cada objeto de dados possui uma identificação única que é utilizada para armazená-lo e recuperá-lo do banco de dados. Quando o MCO solicita um objeto de dados ao GOD, este chama a BCA para criar uma instância transiente desse objeto de dados e a envia ao ROD para que este carregue suas propriedades, cujos valores estão armazenados no banco de dados. De maneira similar, quando um objeto de dados for modificado, o GOD envia-o ao ROD para que este atualize os valores de suas propriedades no banco de dados.

Figura 5.13 – Diagrama de classes do GOD com o ROD, GPI e BCA.

O GOD disponibiliza três formas básicas de acesso aos objetos de dados armazenados no banco de dados:

- 1º) Acesso direto: é o modo de acesso normal ao GOD feito via seus métodos;
- 2º) Acesso via árvore de objetos de dados: permite acesso a vários objetos de dados vinculados que são disponibilizados na forma de uma estrutura tipo árvore;
- 3º) Acesso via conjunto de objetos de dados: permite acesso a vários objetos de dados, selecionados por algum tipo de critério, que são disponibilizados na forma de uma lista semelhante a uma tabela relacional.

O modo de **acesso direto** é mais usado pelo MCO, uma vez que somente ele pode interagir diretamente com o GOD. No entanto, as Aplicações necessitam obter

informações sobre os objetos de dados armazenados no ROD para que possam disponibilizar ao projetista um projeto com seus dados (árvores de projeto), assim como os procedimentos de seleção necessitam exibir listas de objetos de dados para que o projetista possa escolher o objeto de dados que deseja processar. Isso também ocorre com as informações e componentes contidos na BCA que devem ser disponibilizados ao projetista pela Aplicação em uso. Para resolver este problema, foram criadas quatro superclasses derivadas da superclasse Objeto de Dados, cujas derivadas são consideradas extensões do GOD e do GPI e podem ser tomadas como parâmetro por qualquer procedimento operacional. Essas superclasses são: *TAppComponentList*, *TAppComponentTree*, *TDataObjectSet*, *TDataObjectTree*. Suas derivadas concretas são denominadas classes reservadas do framework e são registradas na BCR, podendo ser utilizadas pelo desenvolvedor para desenvolver novas classes ou relacioná-las aos parâmetros dos procedimentos operacionais.

Como o MCO não distingue estas classes reservadas, considerando-as como uma derivada qualquer de Objeto de Dados, ele as solicita ao GOD via acesso direto. O GOD, por sua vez, reconhece essas classes e cria ou carrega seus objetos e os disponibiliza ao MCO. O GOD lida diretamente com as derivadas das superclasses *TDataObjectSet* e *TDataObjectTree*, usadas para implementar os modos de acesso via conjunto de objetos de dados e via árvores de objetos de dados, consecutivamente. As derivadas das superclasses *TAppComponentList* e *TAppComponentTree* são extensões do GPI. O GOD chama os métodos do GPI *CreateComponentList()* ou *CreateComponentTree()* para criá-las e *LoadComponentList()* ou *LoadComponentTree()* para carregá-las, e passa a gerenciar suas instâncias como ele faz com qualquer objeto de dados. O GOD trata as instâncias dessas classes reservadas como transientes. Sendo assim, apenas os métodos *NewDataObject()* e *LoadDataObject()* têm efeito sobre elas. Os demais métodos as ignoram.

A forma de interação entre as classes reservadas com o GOD, o ROD e o GPI é complexa e, por envolver três módulos do *framework* e entrar em muitos detalhes de implementação, foge do escopo descritivo desse capítulo. Isso é feito no Apêndice C.

Os métodos de acesso do GOD são divididos em duas categorias: métodos para leitura (*LoadDataObject()*) e métodos para escrita (*NewDataObject()*, *EditDataObject()* e *PostDataObject()*). A diferença está na forma como o GOD disponibiliza um objeto de dados. Operações somente para leitura compartilham o mesmo objeto de dados

carregado na memória. Já, operações de escrita, criam uma cópia do objeto de dados, quando este existir, bloqueiam o mesmo para impedir que outras operações possam solicitar acesso de escrita, e disponibiliza a cópia ao solicitante (MCO ou objeto de uma classe reservada). Quando o objeto de dados for atualizado, todos as sessões seguras que o estão utilizando são notificadas das mudanças ocorridas. À medida que os procedimentos operacionais em seu interior solicitam atualização, a cópia do objeto de dados vai sendo enviada a eles até que todos a tenham. Nesse instante, o objeto de dados original é descartado e a cópia toma seu lugar. Isso garante que as modificações no objeto de dados não irão afetar os resultados de uma impressão ou de um algoritmo em execução, uma vez que o MCO pode executar vários procedimentos operacionais simultaneamente.

5.3.6. O Gerenciador de Procedimentos e Interfaces (GPI)

O GPI foi projetado para fornecer ao MCO instancias de procedimentos operacionais com seus parâmetros, procedimentos de seleção, e interfaces com o usuário. Ele também fornece às Aplicações estruturas contendo os componentes que ela pode disponibilizar ao projetista. Em seus processos, ele utiliza a BCA para instanciar esses componentes e obter seus roteiros de uso. A Figura 5.14 mostra a BCA com seus métodos utilizados pelo GPI. Ele foi elaborado para encapsular a complexidade dos mecanismos de busca sobre esses roteiros que consideram aspectos de herança e visibilidade sobre dois relacionamentos do Metamodelo: Processa e Agrega. Esses conceitos foram definidos para facilitar a codificação e o registro de novos Objetos de Dados e Procedimentos Operacionais com seus parâmetros.

Figura 5.14 – Diagrama de classes do GPI com a BCA.

Para o desenvolvedor, o relacionamento Processa associa um Procedimento Operacional como um método dinâmico de um Objeto de Dados. Sendo um método, é

natural que as derivadas desse Objeto de Dados herdem esse Procedimento Operacional. Da mesma maneira, os parâmetros definidos para um Procedimento Operacional também devem ser herdados pelo Procedimento Operacional derivado, respeitando a semântica definida nos diagramas de classes UML. Os mecanismos do GPI garantem isso ao desenvolvedor, que pode registrar um Objeto de Dados ou Procedimento Operacional novo sem necessitar estabelecer todos os relacionamentos Processa ou Agrega de seus ancestrais novamente. No entanto, podem ocorrer situações em que um desses relacionamentos pode não ser mais desejado pelo novo componente. Nesse caso, é utilizado o conceito de visibilidade. Em princípio, todo relacionamento Processa ou Agrega é visto pelo GPI como sendo público, isto é, pode ser herdado por todas as classes derivadas. Quando uma destas classes deseja inibir um relacionamento herdado, ela redefine este relacionamento marcando-o como privado.

Outro aspecto importante considerado pelos mecanismos do GPI é a sobrecarga de relacionamentos. Quando uma classe redefine um relacionamento já existente em uma de suas classes ancestrais, esse relacionamento é considerado uma sobrecarga do relacionamento ancestral.

Para compreender melhor os conceitos de herança e visibilidade descritos nos parágrafos anteriores, considere uma classe concreta **O**, derivada de Objeto de Dados, e sua derivada **O'**, ambas relacionadas a Procedimentos Operacionais, cujos roteiros são representados pelos conjuntos **R** e **R'** sucessivamente:

$$\mathbf{R} = \{\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3\} \text{ e}$$

$$\mathbf{R}' = \{\mathbf{P}_2', -\mathbf{P}_3\}, \text{ onde}$$

$\mathbf{P}_i$ representa o Procedimento Operacional **i**;

$\mathbf{P}_i'$ representa o Procedimento Operacional derivado do
Procedimento Operacional **i**;

$-\mathbf{P}_i$ marca o Procedimento Operacional **i** como privado;

Quando aplicados os mecanismos de busca sobre **O**, o resultado é o próprio conjunto **R**. No entanto, quando aplicados esses mecanismos sobre **O'**, eles retornam o seguinte conjunto:

$$\mathbf{R}^* = \mathbf{R} \cup^* \mathbf{R}' = \{\mathbf{P}_1, \mathbf{P}_2'\}, \text{ onde}$$

$\cup^*$ é a operação de união matemática entre conjuntos respeitando as restrições de herança e visibilidade dos relacionamentos.

A primeira restrição respeitada por U^* é a visibilidade dos relacionamentos. Durante o processo de união, o primeiro passo é eliminar todos os relacionamentos privados para poupar esforço computacional durante a união dos relacionamentos sobrecarregados. Isso pode ser visto claramente em R^* com relação a P_3 que aparece como público em R , mas em R' é marcado como privado. Sendo assim, P_3 é eliminado do conjunto R^* .

A segunda restrição respeitada por U^* é a sobrecarga dos relacionamentos, que é o segundo passo no processo de união. Esse é o caso de P_2' e P_2 . P_2' está relacionado a O' , enquanto seu ancestral P_2 está relacionado ao ancestral O . Nesse caso, P_2' é mantido no conjunto R^* , enquanto P_2 é descartado.

Com relação aos parâmetros de um procedimento operacional, os mecanismos de busca se comportam de maneira idêntica, diferenciando-se apenas na forma de comparação. No caso dos Procedimentos Operacionais, a comparação usa a identificação da classe (propriedade *ClassID*), enquanto que para os parâmetros, a comparação é feita usa o nome dos parâmetros (propriedade *Name*). Cada parâmetro recebe um nome distinto. Se ele aparecer em mais que um relacionamento, apenas um será considerado.

Assim como foi definido para o GOD, os procedimentos operacionais também não têm acesso direto ao GPI, como descrito na seção anterior. Por essa razão ele disponibiliza classes reservadas derivadas das superclasses *TAppComponentList* e *TAppComponentTree* para que os procedimentos operacionais possam disponibilizar ao projetista os componentes das aplicações necessários ao desenvolvimento de um projeto ou subprojeto. Como mencionado na seção anterior, a forma de interação entre essas classes reservadas e o GPI é complexa e foge do escopo descritivo deste capítulo. Isso é feito no Apêndice C.

5.3.7. O Repositório de Objetos de Dados (ROD)

O ROD foi projetado para ser uma camada de software sobre um SGBD, desenvolvido a partir de uma abordagem objeto-relacional. Ele define uma interface única de acesso aos objetos de dados e seus relacionamentos de vínculo armazenados no banco de dados, seja ele relacional, orientado a objeto, ou objeto-relacional. Como bancos de dados relacionais suportam apenas armazenar dados em suas tabelas, o ROD

utiliza o banco de dados apenas para armazenar as propriedades dos objetos de dados, independente de sua tecnologia. Nesta seção, são apresentadas apenas as formas de armazenamento e recuperação dos objetos de dados. As formas de armazenamento e recuperação dos relacionamentos de vínculo são apresentadas na seção 5.3.9 para facilitar a compreensão das mesmas.

O GOD pode solicitar ao ROD operações para leitura, atualização ou inserção de objetos de dados na base de dados. Nesse processo, o ROD mapeia as propriedades do objeto de dados para as estruturas internas do banco de dados e efetua a operação solicitada. No caso de bancos de dados relacionais e objetos-relacionais, essas propriedades são mapeadas para campos de uma ou mais tabelas do banco de dados. No caso de bancos de dados orientados a objetos, essas propriedades são mapeadas para as propriedades do objeto armazenado no banco de dados. Além das operações básicas sobre objetos de dados, o ROD também gera conjuntos de objetos de dados somente para consulta (classes reservadas derivadas da superclasse *TDataObjectSet*), que são usados para acesso ao GOD via conjunto de objetos de dados, e na construção das árvores de objetos de dados (classes reservadas derivadas da superclasse *TDataObjectTree*). Detalhes sobre esses conjuntos são fornecidos no Apêndice C.

A característica chave da abordagem adotada para o ROD é a forma de mapeamento das propriedades dos objetos de dados. A Figura 5.15 ilustra como é feito esse mapeamento. A técnica aplicada para isso foi inspirada nos métodos de transformação apresentados por Fong (2002) e Rahayu et al. (2000) em que transações e relacionamentos de herança orientados a objeto são transformados em transações e tabelas relacionais. A utilização de um repositório de metadados e uma camada de software para efetuar o mapeamento foi inspirada no trabalho desenvolvido por Stoimenov et al. (1999), no qual um mediador é utilizado para transformar objetos em esquemas relacionais.

O ROD divide a base de dados em duas partes, denominadas: repositório de metadados e repositório de objetos. O **repositório de metadados** contém as informações de mapeamento que relacionam uma classe e suas propriedades a uma entidade e seus atributos dentro do banco de dados. A entidade pode ser uma tabela em um banco de dados relacional ou objeto-relacional, ou uma classe em um banco de dados orientado a objeto. Da mesma maneira, o atributo pode ser um campo de uma tabela em um banco de dados relacional ou objeto-relacional, ou um atributo de uma

classe em um banco de dados orientado a objeto. O **repositório de objetos** contém as propriedades dos objetos de dados que foram armazenadas de acordo com as estruturas definidas pelos metadados. Essas estruturas podem ser tabelas ou contêineres de objetos.

Figura 5.15 – Esquema de mapeamento das propriedades dos objetos de dados.

O ROD efetua todo o processo de mapeamento a partir dos três métodos virtuais da superclasse Objeto de Dados mostrados na Figura 5.15. Esses métodos são sobrecarregados pelas classes derivadas para definir, armazenar e carregar suas propriedades. Detalhes sobre esses métodos são descritos no Apêndice B.

Toda vez que o ROD é inicializado pelo sistema, ele chama o método **BCA.GetDataObjectClasses()** para verificar se existem Objetos de Dados novos que ainda não foram mapeados. Se houver, ele solicita uma instância de cada Objeto de Dados, via **BCA.InstanceOf()**, e chama seus métodos **DefineProperties()** para obter a lista de propriedades a serem mapeadas. Em seguida, ele registra as propriedades no repositório de metadados e gera os esquemas para criação das entidades e atributos que iram receber os valores dessas propriedades. Ao final desse processo, ele cria as novas entidades no repositório de objetos. A Figura 5.16 mostra a BCA com os métodos utilizados nesse processo.

Figura 5.16 – Diagrama de classes do ROD com a BCA.

Quando o GOD envia ao ROD um objeto de dados para ser carregado, o ROD utiliza as informações do repositório de metadados para efetuar as consultas necessárias ao repositório de objetos e montar uma lista com todos os valores das propriedades desse objeto de dados. Essa lista é enviada ao objeto de dados via método *LoadProperties()* que a utiliza para carregar suas propriedades para a memória. De maneira similar, quando o GOD envia ao ROD um objeto de dados para ser atualizado, o ROD utiliza as informações do repositório de metadados para construir uma lista de propriedades com ligações com os atributos das entidades no banco de dados que é enviada ao objeto de dados via método *UpdateProperties()*.

O processo de mapeamento é exemplificado pela Figura 5.17. Observe que as classes e seus relacionamentos de generalização são mapeados para uma tabela denominada *TGeneralization*, enquanto que suas propriedades são mapeadas para uma tabela denominada *TPropertyMap*. A tabela *TGeneralization* indica quais são as classes ancestrais de uma classe mapeada. Isso auxilia o sistema na localização de classes já mapeadas, evitando repetir estruturas de entidades já existentes. A tabela *TPropertyMap* indica em quais atributos de quais entidades estão armazenados os valores das propriedades de um determinado objeto. Para cada nova classe é gerada uma entidade que contém um atributo chamado *ObjectID* e os atributos para suas propriedades. Todo objeto de dados armazenado no banco de dados recebe uma identificação única, definida pelo ROD, que é atribuída ao *ObjectID* e permite ao ROD localizar todas as propriedades desse objeto de dados em todas as entidades nas quais elas foram armazenadas. A única classe ancestral que não é mapeada nesse processo é a superclasse Objeto de Dados (*TDataObject*) por ser a superclasse base e suas propriedades sempre estarem presentes em todos os objetos de dados. O ROD reconhece seus atributos e os mantém na entidade *TDataObject*.

Apesar do ROD gerar entidades separadas para cada classe derivada, os relacionamentos de generalização (herança) são tratados como mutuamente exclusivos (disjuntos), ou seja, um objeto instanciado de uma classe não pode ser compartilhado com qualquer objeto instanciado de uma de suas classes derivadas. Nesses casos, é comum criar uma entidade para cada classe, repetindo os atributos das classes derivadas. No entanto, a forma de estruturação adotada facilita as consultas para os conjuntos de objetos de dados, pois, como todas as classes são derivadas de Objeto de Dados a entidade *TDataObject* contém a referência a todos os objetos de dados armazenados.

Figura 5.17 – Exemplo de mapeamento de um Objeto de Dados.

5.3.8. A Biblioteca de Componentes das Aplicações (BCA)

O módulo BCA foi projetada para implementar uma camada de software sobre os dados da BCR convertida em código fonte pelo SGCA. O módulo BCA, assim como o SGCA, considera cada classe do Metamodelo e suas derivadas registradas na BCR como elementos do Metamodelo. Seus métodos definem uma forma padronizada de acesso a esses elementos e seus relacionamentos, sendo divididos em dois grupos:

1. Métodos para instanciação e liberação de objetos: criam e destroem objetos e objetos de referência desses elementos.

2. Métodos de obtenção de roteiros: usados para gerar roteiros a partir dos relacionamentos entre os elementos registrados.

Na seção 5.3.2 foram apresentados alguns detalhes sobre a organização dos dados da BCR na forma de código fonte, e como a BCA os utiliza para fornecer ao GPI as instâncias das classes das aplicações e as listas de seus possíveis relacionamentos com base nos roteiros de uso. Na Figura 5.7, p.98, são mostradas oito das nove tabelas estáticas que armazenam os dados convertidos da BCR. A nona tabela, denominada *Application*, é mostrada na Figura 5.18 e contém os registros dos objetos da classe *TApplicationObject* usados para identificar uma Aplicação e relacioná-la a seu contexto da aplicação (*ContextObjectID*) e ao Procedimento Operacional de sua Interface Padrão (*OProcClassID*).

<i>Application</i>				
<i>ObjectID</i>	<i>Name</i>	<i>Description</i>	<i>ContextObjectID</i>	<i>OProcClassID</i>
201	Aplicação padrão do SAPAF	Aplicação padrão desenvolvida para o SAPAF.	200	180

Figura 5.18 – Tabela *Application* da BCA.

O termo elemento do Metamodelo foi adotado como uma forma genérica de referencia aos três tipos de elementos registros pelo SGCA na BCR definidos a partir das classes do Metamodelo. Esses elementos são ilustrados pela Figura 5.19, que mostra as classes do Metamodelo, e seus relacionamentos, coloridas de acordo com o tipo de elemento que representam. Esses elementos são descritos como:

1. Componentes das aplicações – classes concretas derivadas das classes abstratas que representam os pontos variáveis do Metamodelo – classes em amarelo claro na Figura 5.19.
2. Objetos de referência – objetos usados para referenciar um Procedimento de Seleção ou Procedimento de Visualização – classes em azul claro na Figura 5.19; isso é necessário para que o GPI possa reconhecê-los e associá-los corretamente aos parâmetros de um procedimento operacional, pois o MCO pode executar um deles para obter um ou vários parâmetros simultaneamente.
3. Objetos – instâncias dos pontos fixos do Metamodelo – classes em verde claro na Figura 5.19.

Figura 5.19 – Tipos de elementos do Metamodelo.

Os métodos de instanciação criam os elementos e os inicializam de acordo com seu tipo. Se o elemento é uma classe ou objeto de referência eles criam uma instância dessa classe ou da classe referenciada e inicializam suas propriedades básicas, definidas pela classe *TSysObject*, com os dados contidos nas tabelas da BCA. Se o elemento é um objeto, eles o recuperam da tabela da BCA que o contém.

Os mecanismos-chave do módulo BCA são os empregados na criação dos roteiros de uso. Eles interpretam as informações contidas nas tabelas estáticas da BCA e geram uma estrutura na forma de árvore, denominada Estrutura de Roteiro, cujos nós contém os elementos do Metamodelo na forma de objetos de referência, e cujas ramificações representam os relacionamentos entre esses elementos. Um roteiro pode ser visto como uma representação dinâmica dos diagramas de classe-objeto gerados nos processos de modelagem das Estruturas das Aplicações, Estruturas de Projeto e descrição dos parâmetros dos Procedimentos Operacionais.

As Estruturas de Roteiro são geradas a partir dos relacionamentos entre objetos instanciados de duas classes *TScript* e *TRelationshipList* cujo diagrama de classes é mostrado na Figura 5.20. A classe *TScript* representa o nó da árvore. Ela é considerada um descritor, pois descreve um elemento do Metamodelo representado na forma de um objeto de referência (*Element*), sua visibilidade (*Public*) e seus relacionamentos com outros elementos (listas de objetos *TRelationshipList*). A classe *TRelationshipList* faz a

ligação entre os ramos da árvore. Seus objetos contêm uma lista com todos os elementos que estabelecem um tipo de relacionamento (*RelType*) específico com *Element*, descritos na forma de objetos da classe *TScript*.

Figura 5.20 – Diagrama de classes para construção das Estruturas de Roteiro.

O módulo BCA disponibiliza um método, denominado *GetScript()*, que implementa os mecanismos para geração dos roteiros. Esse método toma como parâmetros a identificação de um elemento contido na BCA e um conjunto com os tipos de relacionamentos desejados, e gera um roteiro para esse elemento com todos os elementos relacionados a ele. O conjunto com os tipos de relacionamentos desejados age como um filtro, determinando quais relacionamentos, estabelecidos pelo elemento, devem ser considerados na estruturação do roteiro.

GetScript() é implementado para ser um método recursivo. Quando solicitado, cria um objeto *TScript* para descrever o elemento enviado como parâmetro (*AElement*). Para cada tipo de relacionamento que *AElement* pode estabelecer e que pertence ao conjunto com os tipos de relacionamentos desejados, ele cria um objeto *TRelationshipList*, agrega-o ao objeto *TScript*, e busca nas tabelas estáticas os elementos que estabeleceram esse tipo de relacionamento com *AElement*. Cada elemento relacionado tem seu próprio roteiro que também é descrito da mesma maneira. Sendo assim, *GetScript()* solicita a si mesmo para obter o roteiro desse elemento e agregá-lo ao objeto *TRelationshipList* criado. Para que esses mecanismos funcionem corretamente, foi estabelecida uma rota que determina uma única direção de navegação entre os relacionamentos. Essa rota é ilustrada pela Figura 5.21. Os relacionamentos, na forma de setas, indicam a direção percorrida. *GetScript()* percorre os roteiros como se estes fossem um grafo, cujos nós são os elementos do metamodelo e os caminhos são os relacionamentos entre esses elementos.

princípio, essas quatro formas de relacionamento cobrem as necessidades de relacionamentos persistentes do software proposto, mas sua implementação é genérica o bastante para suportar outras formas de relacionamento futuras. Essas quatro formas de relacionamento são gerenciadas pelo GOD que as reconhece e mantém no banco de dados via ROD.

Figura 5.22 – Diagramas de classes dos tipos de vínculos entre Objetos de Dados.

O relacionamento **Vínculo de Parentesco** foi criado especialmente para vincular objetos de dados a seus projetos e subprojetos e não é usado pelos mecanismos de controle do *framework* de operação com outra finalidade.

O relacionamento **Vínculo de Processamento** foi criado para registrar os históricos de processamento dos objetos de dados. Com eles, o projetista pode verificar a origem de um dado do projeto e como esse dado foi transformado. Para efetuar o registro, o GOD utiliza uma classe de objetos auxiliar, denominada *THistoryObject*, que é usada para armazenar as informações de processamento, e a lista de parâmetros utilizados no processamento de um objeto de dados. *THistoryObject* é uma derivada concreta da classe *TDataObject* e seus objetos são armazenados no banco de dados como qualquer objeto de dados, podendo ser consultados como qualquer um.

O relacionamento **Vínculo de Extensão** foi criado com duas finalidades básicas: (1) estender os atributos de um Objeto de Dados anteriormente codificado sem necessitar destruir suas definições na base de dados ou criar uma nova classe derivada; e (2) definir listas de atributos complexos ou grandes demais para serem armazenados com o objeto de dados ou trazidos simultaneamente para a memória.

O relacionamento **Vínculo de Compartilhamento** foi criado para permitir o compartilhamento de dados entre um ou mais objetos de dados. São definidas duas formas de compartilhamento denominadas **Compartilhamento por Uso** e **Compartilhamento por Dependência**. **Compartilhamento por Uso** é utilizada para definir relacionamentos entre objetos de dados que compartilham dados comuns, cuja modificação pode afetar o estado de ambos. **Compartilhamento por Dependência** é usado para definir relacionamento entre um objeto de dados que necessita dos dados de um ou mais objetos de dados, cujas modificações afetam o objeto de dados dependente, mas as modificações no objeto de dados dependente não os afetam.

Vínculos de parentesco e vínculos de processamento são estabelecidos pelos mecanismos de controle do *framework* de operação como parte dos requisitos levantados para sua concepção, enquanto vínculos de extensão e vínculos de compartilhamento são definidos internamente pelos objetos de dados ou estabelecidos pelos procedimentos operacionais, e fazem parte dos *frameworks* das Aplicações e de suas instâncias. No entanto, independente de como são estabelecidos os relacionamentos de vínculo, o GOD os reconhece e os armazena ou recupera da base de dados. Para isso, ele utiliza os sete métodos mostrados pela Figura 5.23, cujos detalhes são descritos no Apêndice B.

Figura 5.23 – Métodos do Objeto de Dados usados para controle dos vínculos.

Com exceção dos vínculos de processamento, os demais vínculos podem ser alterados pelos procedimentos operacionais, como foi mencionado anteriormente. Como os procedimentos operacionais não têm acesso direto ao GOD, eles devem utilizar outros meios para isso. Uma forma simples é tomar como parâmetro de entrada um conjunto de objetos de dados de qualquer tipo, e tomar o objeto de dados (**A**), a ter seus vínculos modificados, como parâmetro de saída ou entrada e saída. Dessa maneira, ele pode usar o conjunto de objetos de dados para obter os objetos de dados (**B**) desejados e vinculá-los ao objeto de dados **A**.

Apesar de todo o controle sobre esses quatro tipos de vínculo ser feito pelo GOD, é o ROD que garante a persistência dos mesmos. Para isso, ele utiliza uma entidade cujos atributos descrevem os relacionamentos de vínculo entre esses objetos de dados, seus tipos, e as propriedades dos relacionamentos. Essa entidade, denominada *LinkTable*, é ilustrada na Figura 5.24. O atributo **A_ObjectID** identifica o objeto de dados que recebe os vínculos, enquanto o atributo **B_ObjectID** identifica o objeto de dados vinculado. Já o atributo **LinkType** identifica o tipo de vínculo estabelecido; seu tipo é numérico, mas foi mostrado o nome do tipo para efeito ilustrativo. O último atributo, **LinkProperties**, contém uma lista de valores utilizada para armazenar as propriedades dos relacionamentos de vínculo, como é o caso do vínculo parâmetro.

O ROD disponibiliza métodos específicos para acesso à entidade de vínculos que permitem ao GOD inserir (*Link()*), remover (*Unlink()*), consultar (via derivada de *TDataObjectSet*), e carregar (*LoadLinks()*) vínculos de um objeto de dados.

LinkTable			
A_ObjectID	B_ObjectID	LinkType	LinkProperties
707	1127	Parentesco	
1127	1129	Processamento	
1129	974	Parâmetro	Name=Object
1176	972	Dependência	
707	1180	Parentesco	Name=Matrix
1180	1182	Processamento	
1182	972	Parâmetro	Name=GenCount

Figura 5.24 – Entidade de vínculos do ROD.

5.4. Uso do *Framework* de Operação

O *framework* de operação é exclusivamente usado para o desenvolvimento do *framework* da Aplicação, cuja estrutura básica foi descrita na seção 5.2. Para isso, as seguintes atividades devem ser desenvolvidas:

1. Defina o tipo de Aplicação desejada. O tipo da Aplicação determina como devem ser a Interface Padrão e o comportamento básico esperado da Aplicação. Por exemplo, sistemas especialistas definem um comportamento linear para a execução das tarefas e limitam as ações do projetista. Em contra partida, aplicações multitarefa possibilitam a execução de vários processos ao mesmo tempo e definem um comportamento não linear para a execução das tarefas, dando ao projetista maior liberdade no desenvolvimento das atividades.
2. Defina a Interface Padrão. Após a escolha do tipo da Aplicação, o passo seguinte é definir a aparência da Interface Padrão e o comportamento esperado de cada componente da aplicação em seu ambiente de trabalho. Um resultado natural do desenvolvimento dessa atividade é a definição de diversos componentes genéricos da aplicação que formaram a base do *framework* da Aplicação, ou seja, serão implementados como sendo seus pontos variáveis.
3. Projete o *framework* da Aplicação. Nessa atividade, os componentes genéricos da aplicação, definidos na atividade anterior, são projetados para se tornarem os pontos variáveis do *framework* da Aplicação. Em seguida, a Interface Padrão é projetada para reconhecer as classes concretas desses pontos variáveis, e também são projetados os pontos fixos do *framework* da Aplicação que darão suporte a Interface Padrão e a execução dos objetos das classes dos pontos variáveis.
4. Implemente o *framework* da Aplicação. Uma vez projetado o *framework* da Aplicação, implementá-lo significa implementar a Interface com o Usuário e o Procedimento Operacional para a Interface Padrão, e implementar os pontos variáveis e fixos, derivando-os das classes de ponto variável do *framework* de operação.
5. Teste o *framework* da Aplicação. Nessa atividade, uma estrutura de projeto deve ser criada e incorporada à Aplicação para que seja verificado se a Interface padrão esta respondendo como esperado e se sua aparência atende as definições feitas anteriormente. Também é verificado o funcionamento dos pontos fixos e variáveis do *framework*.

6. Documente o uso do *framework* da Aplicação. Esta é a última atividade. Nela são criados os gabaritos e descritas as diversas formas de aplicação dos mesmos para a criação das Estruturas de Projeto e seus elementos. Com isso, o *framework* da Aplicação está pronto para ser usado no desenvolvimento da aplicação, que consiste na criação e implementação das Estruturas de Projeto e de seus elementos.

Para o desenvolvimento do *framework* da Aplicação, existe um conjunto de gabaritos e regras para construção da Interface Padrão e dos componentes para abertura, criação e fechamento de projetos, definidos no Apêndice A.

5.5. Uso do *Framework* da Aplicação

O *framework* da Aplicação estende e simplifica o uso do *framework* de operação. Sua estrutura define um conjunto de pontos variáveis mais específicos que possibilitam ao desenvolvedor criar mais facilmente novos Objetos de Dados, Procedimentos Operacionais, e Interfaces com o Usuário que comporão as Estruturas de Projeto a serem incorporadas à Aplicação.

O uso do *framework* da Aplicação consiste, basicamente, da aplicação dos diversos gabaritos para a construção das Estruturas de Projeto e de seus elementos. Para isso, foi definido um modelo de processo baseado na *engenharia de desenvolvimento baseado em componentes* da CBSE apresentado na seção 3.6. Esse modelo é mostrado na Figura 5.25. As atividades desse modelo são similares às atividades do modelo de processo para a CBSE, só que foram definidas para construir Estruturas de Projeto e seus elementos, e registrá-los na BCR para que sejam incorporados à Estrutura da Aplicação.

Figura 5.25 – Modelo de processo para uso do *framework* da Aplicação.

A atividade de **análise** busca identificar os novos elementos do domínio da aplicação com base nos componentes genéricos definidos para a aplicação, como os tipos e estruturas de dados, os algoritmos e ferramentas de apoio ao projeto, os contextos de projeto, os projetos e subprojetos, etc.

A atividade de **projeto** toma os elementos identificados pela atividade de análise e aplica os gabaritos para modelá-los na forma de componentes das aplicações que comporão a Estrutura de Projeto a ser implementada e incorporada à Aplicação.

A atividade de **qualificação** efetua buscas na BCR para tentar localizar os componentes das aplicações existentes ou que possam ser adaptados para compor a Estrutura de Projeto.

A atividade de **adaptação** toma os componentes das aplicações que foram selecionados para serem adaptados e efetua as mudanças necessárias ou cria componentes derivados para efetuar essas mudanças.

A atividade de **composição** toma as classes da aplicação implementadas e registra-as na BCR de acordo com seus diagramas de classes modelados na atividade de projeto. Ao final desta atividade, a BCA é gerada pelo SGCA e o software proposto é recompilado para que a nova Estrutura de Projeto e seus elementos sejam incorporados à Aplicação.

A atividade de **engenharia** é aplicada toda vez que um ou mais componentes das aplicações necessitam ser criados ou mesmo modificados.

A atividade de **teste** verifica se os novos componentes das aplicações estão funcionando corretamente e se as árvores de projeto estão sendo geradas como previsto. Feitos todos os testes necessários, o software proposto é liberado para uso e avaliação do projetista.

A atividade de **atualização** toma as solicitações de mudança e adequação sugeridas pelo projetista e modifica os componentes das aplicações e a Estrutura de Projeto para satisfazer a essas solicitações.

Como pôde ser observado, os pontos-chave na aplicação desse modelo de processo são o uso dos gabaritos para modelar os componentes das aplicações e o uso da BCR para reaproveitar componentes das aplicações existentes e registrar os novos componentes das aplicações desenvolvidas durante o processo. Uma amostra desses gabaritos pode ser obtida no Apêndice A.

5.6. O Software Gerenciador de Componentes das Aplicações (SGCA)

O SGCA foi elaborado para auxiliar na construção das Aplicações. Ele possui quatro funcionalidades básicas:

- registro dos componentes das aplicações e seus relacionamentos
- registro das Estruturas das Aplicações e Estruturas de Projeto;
- geração da BCA;
- controle e gerenciamento de componentes voltado para reuso.

Em todos os seus processos, o SGCA utiliza a BCR para manter as informações de registro, poder efetuar consultas para gerar a BCA, e disponibilizar quaisquer itens registrados para reuso. A BCR foi projetada na forma de um banco de dados relacional, cujas tabelas mantêm os registros dos elementos do Metamodelo e dos seus relacionamentos, descritos na seção 5.3.8. A Figura 5.26 mostra o ME-R (Modelo Entidade-Relacionamento) utilizado na definição dessas tabelas. Nesse ME-R, as entidades foram definidas a partir dos elementos do Metamodelo e seus relacionamentos e cardinalidade foram definidos a partir dos relacionamentos das classes do Metamodelo que representam cada elemento.

O SGCA foi projetado com base em técnicas de reuso para ser, inicialmente, um gerenciador de componentes reusáveis para a construção das Aplicações. No entanto, a flexibilidade estrutural obtida com a adoção de estruturas dinâmicas (Estruturas das Aplicações e Estruturas de Projeto) para compor as Aplicações que são descritas na forma de roteiros interpretados pelo *framework*, possibilitou a união da própria BCR com o *framework*. Como a BCR é apenas um repositório cujos componentes não estão necessariamente na forma binária, foi projetada a BCA, que pode ser considerada uma versão binária ou em código fonte da BCR. Isso possibilita ao desenvolvedor trabalhar com um repositório completo, recheado de componentes e Aplicações, e converter para a BCA apenas a Aplicação desejada.

No primeiro estágio de desenvolvimento do software proposto, a preocupação maior residia na geração da BCA e em sua utilização pelo *framework*. Por essa razão, o SGCA foi implementado para alimentar a BCR e gerar a BCA, não havendo uma preocupação com suas interfaces e formas de registro amigáveis como, por exemplo, a utilização de uma interface gráfica voltada para a modelagem das estruturas dinâmicas diretamente no SGCA, ou alguma forma mais intuitiva de alimentar a BCR com essas estruturas.

Figura 5.26 – Modelo Entidade-Relacionamento da BCR.

A Figura 5.27 mostra a tela principal do SGCA com seus menus e a janela de cadastro de componentes. O menu Arquivo contém os comandos para geração da BCA. O menu Registro contém os comandos para registrar: os componentes das aplicações; procedimentos de seleção e visualização; objetos das classes Contexto, Objeto da Aplicação e Parâmetro; e os relacionamentos múltiplos Contém, Processa, Agrega e Controla. Nessa versão, não há nenhum tipo de interpretação semântica por parte do SGCA, isto é, ele não verifica inconsistências ou referências cíclicas que podem ocorrer no registro dos relacionamentos, assim como não orienta no registro das estruturas dinâmicas. Para os registros, o desenvolvedor necessita utilizar os diagrama de modelagem para fazer o acompanhamento.

Figura 5.27 – Tela principal do SGCA.

5.7. Aspectos da Implementação do *Framework* de Operação

Para a implementação do software proposto, foi adotada a plataforma PC com sistema operacional Windows[®] 2000 da Microsoft. A linguagem de programação adotada foi o Object Pascal estendido pelo Delphi[®] da Borland que é baseada na linguagem Pascal com instruções de suportar a orientação a objeto e desenvolvimento visual.

Para a implementação do ROD foi adotado o gerenciador de banco de dados relacional Cliente-Servidor Interbase[®] versão 5.0 da Borland por fazer parte do pacote de desenvolvimento profissional do Delphi, e por possuir comandos para criação de tabelas via software cliente – o ROD necessita desse recurso para criar as tabelas dos Objetos de Dados automaticamente.

O Delphi trabalha com um conjunto de conceitos e padrões de programação que foram adotados para a codificação do software proposto. O primeiro está relacionado à nomenclatura dos identificadores. Os nomes das classes e tipos sempre iniciam com a letra “T” seguida pelo nome com as iniciais maiúsculas e todas as palavras juntas: exemplo *TDataObject* é o nome da classe Objeto de Dados ou *Data Object* em inglês. Nomes de propriedades e métodos de uma classe são escritos com as iniciais maiúsculas e todas as palavras juntas: exemplo *ClassName* é a propriedade nome da classe, ou *class name* em inglês. O nome das variáveis utilizadas para armazenar os valores de uma propriedade iniciam com a letra “F” seguida pelo nome da propriedade. Também existem outros padrões que, por não aparecerem nos textos desta tese, não necessitam ser citados.

O segundo está relacionado à estruturação de um programa. O Delphi trabalha com o conceito de *Unit* que são módulos auto-executáveis usados na forma de bibliotecas. Esses módulos possuem sessões para declaração das interfaces do módulo (declaração de tipos, constantes, classes, variáveis, etc.), implementação dos métodos das classes e procedimentos e funções, e uma sessão para inicialização e outra para finalização do módulo. Isso permite trabalhar com objetos instanciados em nível de sistema. Esse recurso foi utilizado para inicializar o núcleo do *framework*, o módulo BCA e um objeto, denominado *SystemObject*, que encapsula o sistema operacional e disponibiliza seus recursos ao *framework* a partir de uma interface padronizada. Esses recursos envolvem: configurações de ambiente, conexão com o banco de dados, controle

de mensagens, relatórios (*logs*) de inicialização e notificações de erro com solicitações ao usuário.

O terceiro, e último, foi à utilização de classes dinâmicas do Delphi para instanciar objetos de maneira genérica. O compilador do Delphi cria automaticamente tabelas que são usadas por mecanismos em código de máquina para instanciar objetos a partir de uma referência (ponteiro) da classe. Esse mecanismo foi utilizado na geração da BCA e possibilitou vincular as tabelas estáticas a referência à classe que implementa um componente, dispensando a criação de rotinas específicas para isso. Convém lembrar que em linguagens como C++ a instanciação é sempre explícita, isto é, um objeto é instanciado a partir do comando *new* que solicita explicitamente o nome da classe. Isso implicaria na criação de funções para criação de objetos para cada nova classe incorporada à BCA.

Um último aspecto relevante à implementação do *framework* é o uso de *Threads* para executar os procedimentos operacionais. Convém lembrar que o *framework* é multitarefa, e isso é obtido com a execução simultânea de processos. *Threads* são característicos de sistemas operacionais (SO) multitarefa como Unix, Linux, Windows dentre outros, e sua utilização depende de um conjunto de regras e outros recursos disponibilizados pelo SO, como semáforos e mutex. As sessões seguras de processamento e o *SystemObject* encapsulam essas regras e recursos para tornar a execução dos procedimentos operacionais mais seguras. Mesmo assim, um procedimento operacional deve seguir algumas regras de execução. Essas regras prevêm duas formas de execução para procedimentos operacionais: uma por eventos e outra por processamento. Na primeira, o procedimento operacional chama *SystemObject.WaitMessage()* que o coloca em estado de espera até que uma mensagem seja enviada a ele. Quando isso ocorrer, *WaitMessage()* retira o procedimento operacional do estado de espera e retorna para ele a mensagem a ser processada. No segundo modo, o procedimento operacional executa trechos de código repetidamente até obter o resultado desejado e encerra suas atividades. Nessa situação, ele pode consumir muitos recursos de máquina e dificultar a execução de outros procedimentos. Sendo assim, é conveniente que ele chame *SystemObject.ProcessMessages()* periodicamente para que o sistema possa liberar recursos para outros procedimentos.

CAPÍTULO 6. O *FRAMEWORK* DA APLICAÇÃO (SAPAF)

6.1. Considerações Iniciais

O desenvolvimento de uma Aplicação a partir do *framework* descrito no Capítulo 5 é dividido em duas partes. A primeira é a definição e estruturação de uma Aplicação utilizando o *framework* de operação, que resulta em um segundo *framework* denominado *framework* da Aplicação. A segunda é a estruturação das aplicações dessa Aplicação em projetos de arranjo físico que resulta na modelagem e construção das Estruturas de Projeto. Sendo assim, o primeiro passo para desenvolver o software proposto a partir do *framework* descrito no Capítulo 5 é o desenvolvimento de um *framework* da Aplicação para ele.

O software proposto foi denominado SAPAF – Software de Apoio ao Projeto de Arranjo Físico – e o desenvolvimento de seu *framework* da Aplicação seguiu as atividades descritas na seção 5.4. Neste capítulo, são fornecidos alguns detalhes do projeto e da forma de uso desse *framework* denominado *framework* do SAPAF.

6.2. Tipo da Aplicação e sua Interface Padrão

O SAPAF foi projetado para ser uma aplicação baseada em janelas sob o sistema operacional Windows[®] da Microsoft. Sendo assim, sua janela principal (Interface Padrão), é composta por um menu e uma área de trabalho. O menu contém os algoritmos e as operações auxiliares que o projetista pode utilizar para processar um objeto de dados dentro de um determinado contexto de projeto. Esses algoritmos e operações são trocados toda vez que um objeto de dados é selecionado pelo projetista, disponibilizando a ele somente os algoritmos e as operações que ele pode utilizar para processar o objeto de dados selecionado. A área de trabalho contém um espaço para as janelas implementadas pelas Interfaces com o Usuário e componentes gráficos que

fornece informações sobre o objeto de dados selecionado e exibem a árvore do projeto em desenvolvimento.

Nessa interface, a árvore de projeto é aberta, isto é, o projetista pode acessar qualquer um de seus objetos de dados e processá-lo, ou entrar em qualquer subprojeto e desenvolvê-lo. Como o *framework* de operação é multitarefa, o projetista pode processar vários objetos de dados simultaneamente, por exemplo: ele pode editar um objeto de dados, enquanto outro está sendo processado pelo algoritmo DCA, e um terceiro está sendo impresso. Para que a Interface Padrão “saiba” para qual destes objetos de dados ela deve disponibilizar os algoritmos e operações auxiliares em seus menus, foi adotado um conceito chamado de **objeto de dados ativo**. É considerado ativo, o objeto de dados que está visível ou sendo manuseado pelo projetista em um determinado momento.

A Figura 6.1 mostra a Interface Padrão como descrito nos dois parágrafos anteriores. O menu aparece na parte superior da janela. A área de trabalho compreende toda a parte da janela abaixo do menu. Ela exibe a árvore de projeto à esquerda e o espaço para as interfaces com o usuário à direita. Na parte inferior da árvore de projeto, são mostradas algumas informações sobre o objeto selecionado na árvore, enquanto que na parte inferior do espaço para as interfaces com o usuário, são mostradas algumas informações sobre o objeto de dados ativo.

Figura 6.1 – Interface Padrão do SAPAF.

Os itens no menu foram organizados de acordo com a funcionalidade básica de cada procedimento operacional. O menu Objeto mostra as operações sobre o projeto ou subprojeto ao qual o objeto ativo está vinculado, e as operações que podem ser feitas sobre ele, como imprimir, editar, visualizar, etc. O menu Algoritmos mostra todos os algoritmos que podem ser aplicados para transformar o objeto de dados ativo. O menu Ferramentas mostra todas as operações auxiliares utilizadas para obtenção de dados externos ou integração com outros sistemas que podem ser aplicadas sobre o objeto de dados ativo. O menu Janela contém a lista de todas as janelas das interfaces com o usuário abertas na área de trabalho e podem ser utilizadas pelo projetista para localizar uma janela específica.

A janela de projeto mostra a árvore do projeto em desenvolvimento e permite ao projetista navegar pelo projeto e visualizar todos os seus dados. Toda vez que um objeto de dados ou projeto é selecionado, alguns de suas propriedades são mostrados na parte inferior dessa janela.

6.3. Modelagem do *Framework* da Aplicação

Na seção anterior, foram definidos: o tipo da aplicação, a Interface Padrão, e a forma de disponibilização dos componentes das aplicações. Para que a Interface Padrão reconheça os procedimentos operacionais definidos para as aplicações e os disponibilize em seus menus de maneira correta, foram criados três pontos variáveis genéricos, derivados da classe Procedimento Operacional, e vários outros pontos variáveis mais específicos, derivados desses três pontos variáveis genéricos. A Figura 6.2 mostra o diagrama de classes com esse três pontos variáveis genéricos e suas derivadas específicas.

As classes *TObjectOperation*, *TAlgorithm* e *TToolOperation* foram definidas para classificar os três tipos básicos de Procedimentos Operacionais reconhecidos pela Interface Padrão. Elas definem métodos abstratos e concretos que determinam o comportamento específico de cada Procedimento Operacional derivado das mesmas. A classe ***TObjectOperation*** é usada para definir as operações básicas sobre os Objetos de Dados. Na Figura 6.2 são mostradas quatro dessas operações reconhecidas pela Interface Padrão. Essas classes e quaisquer outras derivadas da classe *TObjectOperation*, são organizadas no menu Objeto. A classe ***TAlgorithm*** é usada para implementar todos os algoritmos usados no desenvolvimento dos projetos. As classes derivadas da classe

TAlgorithm são organizadas no menu Algoritmos. A classe *TToolOperation* é usada para implementar Procedimentos Operacionais auxiliares, como rotinas de conversão, importação e exportação de dados. Essas rotinas são mostradas na Figura 6.2 para exemplificação. As classes derivadas da classe *TToolOperation* são organizadas no menu Ferramentas.

Figura 6.2 – Pontos variáveis do *framework* do SAPAF: Procedimentos Operacionais.

O processamento das operações básicas normalmente ocorre muito mais em nível de interface do que em nível de procedimento operacional, enquanto que o processamento dos algoritmos e ferramentas ocorrem no sentido oposto. Por essa razão, a classe *TObjectOperation* pré-define um comportamento para os métodos abstratos de *TOperatingProcedure* voltado para operar baseado em eventos. Com isso, suas derivadas necessitam apenas redefinir o método *TObjectOperation.HandleMessage()* para receber eventos específicos de suas interfaces.

Assim como um comportamento específico foi definido para as operações, a Interface Padrão também define um comportamento esperado de cada Interface com o Usuário criada para um CIU voltado para o SO Windows®. Por essa razão, foram definidos pontos variáveis específicos derivados da superclasse Interface com o Usuário para implementar esse comportamento. A Figura 6.3 mostra essas classes.

Figura 6.3 – Pontos variáveis do *framework* do SAPAF: Interfaces com o Usuário.

A superclasse *TWinInterface*, implementa um comportamento específico para as interfaces com o usuário que serão executadas no ambiente Windows. Ela também implementa o comportamento esperado pela Interface Padrão, como notificação do objeto de dados ativo, solicitação de visibilidade, notificação de abertura e fechamento, dentre outros eventos de notificação e controle utilizadas pela Interface Padrão para operar corretamente. A classe *TOperationInterface* é usada para derivar todas as Interfaces com o Usuário utilizadas pelas operações. Ela implementa o comportamento pré-definido pela classe *TObjectOperation*. A classe *TControlInterface* é usada para derivar todas as Interfaces com o Usuário utilizadas pelos algoritmos e ferramentas. Ela implementa um comportamento pré-definido para as classes *TAlgorithm* e *TToolOperation* que é esperado pela Interface Padrão.

As classes *TViewObject*, *TEditObject* e *TSaveObject*, apesar de serem consideradas pontos variáveis, são derivadas concretas e podem ser utilizadas para visualizar, editar ou salvar qualquer objeto de dados. Elas foram projetadas para trabalhar em conjunto usufruindo da mesma interface com o usuário. Isso é obtido encapsulando a chamada de *TEditObject* e *TSaveObject* a partir de *TViewObject*.

A classe *TViewObject* possui quatro métodos utilizados pela Interface Padrão para definir as operações de edição e gravação. *TViewObject.CanEdit()* e *TViewObject.CanSave()* indicam à Interface Padrão se o procedimento de visualização ativo encapsula a chamada aos procedimentos *TEditObject* e *TSaveObject* respectivamente. Isso determina a visibilidade dos comandos Editar e Salvar do menu Objeto. Quando o projetista seleciona um desses comandos, a Interface Padrão chama métodos específicos de *TViewObject* para realizar a operação desejada.

TViewObject.Edit() coloca o objeto ativo em modo de edição e *TViewObject.Salve()* atualiza o objeto de dados e retorna ao modo de visualização. Ambos os métodos encapsulam chamadas ao método *MCO.Execute()* para executar um procedimento *TEditObject* e *TSaveObject* ou uma de suas derivadas.

Esses pontos variáveis simplificam bastante a implementação dos componentes das aplicações para o SAPAF, uma vez que encapsulam todo o comportamento definido para a Aplicação com sua Interface Padrão. Para completar o pacote do *framework* do SAPAF, foram projetados alguns pontos fixos.

O primeiro ponto fixo é uma derivada concreta da superclasse *TObjectProject*, denominada *TGeneralProject*, usada para registrar qualquer objeto de projeto. Com ela, também foram projetadas operações para abrir, fechar, criar, visualizar, editar e salvar um projeto, que são reconhecidas pela Interface Padrão como suas operações auxiliares iniciais. Também foram projetados um Procedimento de Seleção e sua Interface de Seleção para selecionar o projeto a ser aberto.

Outro ponto fixo é uma derivada concreta da classe *TVariable*, denominada *TCommentary*, usada para adicionar comentários aos objetos de dados em geral. Com ela, também foram projetadas operações para visualizar, editar e salvar comentários. Os comentários são adicionados aos Objetos de Dados utilizando vínculos de extensão.

6.4. Uso do *Framework* da Aplicação

O *framework* da Aplicação é utilizado, basicamente, para construir as Estruturas de Projetos que são encaixadas à Estrutura da Aplicação. No Apêndice A, são mostrados vários gabaritos que podem ser aplicados para isso. No entanto, existem alguns “truques” que podem ser aplicados nesse processo.

Um deles é o registro dos Objetos de Projeto para compor a Estrutura de Projeto. O SGCA, assim com o *framework* de operação, identificam os componentes pelo seu *ClassID*, e não pelo seu *ClassName*, que é o nome de implementação da classe. Isso possibilita registrar várias vezes a mesma classe com identificações diferentes e relacioná-las com outros elementos do Metamodelo como se fosse uma nova implementação ou derivada de Objeto de Projeto. Caso queira que o novo Objeto de Projeto herde os relacionamentos da classe original, basta colocá-la como ancestral. Esse mesmo truque pode ser utilizado para registrar operações de visualização, edição,

etc., em que o mesmo Procedimento Operacional é registrado várias vezes e relacionado a várias Interfaces com o Usuário diferentes.

Outro truque é o relacionamento das operações para criação de objetos de dados. A lógica de operação definida para a Interface Padrão solicita que exista um objeto de dados ativo para que os procedimentos operacionais possam ser disponibilizados. Isso significa, que uma operação de criação só estará disponível ao projetista quando houve um objeto de dados ativo. O problema em torno disso é: como disponibilizar uma operação para criar um objeto de dados, se ele ainda não existe? A resposta para isso é relacionar a operação de criação não ao Objeto de Dados que ela gera, mas a um Objeto de Projeto ou Objeto da Aplicação, pois estes sempre estarão, em algum momento, ativos.

CAPÍTULO 7. UMA INSTANCIACÃO DO *FRAMEWORK* (SAPAF)

7.1. Considerações Iniciais

No Capítulo 6, foi apresentado o *framework* do SAPAF, e foram definidas sua Interface Padrão e sua lógica de operação. Para que o SAPAF possa ser utilizado como software de apoio, é necessária a criação de pelo menos uma aplicação para ele, isto é, uma Estrutura de Projeto.

Neste capítulo, é apresentada a Estrutura de Projeto construída para testar o SAPAF e validar a solução adotada para seu desenvolvimento. Essa estrutura envolve três tipos de arranjos físicos com seus algoritmos e tipos de dados, que serão descritos a seguir. No final deste capítulo, são mostrados alguns dados obtidos com a execução dos algoritmos implementados.

7.2. Modelagem da Estrutura do Projeto Piloto

Para os testes do SAPAF foram selecionados quatro algoritmos empregados em três tipos de arranjos físicos. Esses algoritmos, assim como algumas operações auxiliares, foram escolhidos com a seguinte finalidade: testar o *framework* e suas funcionalidades, mostrar a versatilidade do software proposto e validar a solução.

A Figura 7.1 mostra, de maneira simplificada, a Estrutura de Projeto para os testes, denominada Estrutura do Projeto Piloto. À direita dessa estrutura, é mostrada uma outra estrutura que descreve as operações e ferramentas básicas criadas para cada tipo de dado utilizado pelos algoritmos. Observe que, em ambas as estruturas, os estereótipos indicam o nome da classe de origem ou classe ancestral. No caso dos Objetos de Projeto, como todos são registrados utilizando a classe *TGeneralProject*, o nome da classe é o nome de registro de cada um (propriedade *Name*). O Objeto de Dados *TCommentary* e suas operações, *TAddCommentary*, *TViewCommentary*, *TEditCommentary* e *TSaveCommentary*, fazem parte do pacote de componentes do

framework do SAPAF, assim como *TGeneralProject* e suas operações *TOpenProject* e *TCloseProject*. Esses componentes foram colocados na estrutura para indicar sua associação com os Objetos de Dados que implementam os tipos de dados mostrados na Estrutura do Projeto Piloto. As demais operações e ferramentas são genéricas e representam os pontos variáveis do *framework* do SAPAF. Elas foram colocadas na estrutura para indicar quais são as operações necessárias para cada Objeto de Dados, sendo que as ferramentas são opcionais.

Figura 7.1 – Estrutura do Projeto Piloto e dos Objetos de Dados.

A utilização de comentários mostra como podem ser adicionados e manuseados vínculos de extensão. As classes *TBinaryTable* e *TTimeTable* são tabelas que compartilham um mesmo tipo de dados: uma matriz numérica implementada pela classe *TFloatMatriz*, que é uma derivada concreta de Objeto de Dados. Os objetos

TBinaryTable compartilham os objetos *TFloatMatriz* por dependência, enquanto que os objetos *TTimeTable* compartilham os objetos *TFloatMatriz* por uso. Nesse caso em particular, os dados contidos nos objetos *TFloatMatriz* não podem ser alterados pelos objetos *TBinaryTable* ou seus procedimentos operacionais. Por essa razão eles são gerados somente por conversão de um objeto *TTimeTable* em um objeto *TBinaryTable*.

Todos os algoritmos, com exceção de *TGeneticAlgorithm*, possuem saídas relacionadas a um Procedimento de Visualização que é uma operação derivada de *TViewObject*, e demonstram essa funcionalidade da solução.

O algoritmo DCA (*TDCAAlgorithm*) aparece em dois lugares processando dois tipos de dados diferentes. Internamente, ele é capaz de trabalhar com os dois tipos de dados, mas para o *framework* ele só pode processar um tipo de dados. Por essa razão, ele foi cadastrado duas vezes e recebeu dois *ClassID*'s diferentes, mas com os mesmos dados informativos.

O algoritmo *TGeneticAlgorithm* implementa o algoritmo genético proposto por Gonçalves Filho (2001) e utiliza a mesma Interface com o Usuário para selecionar seus parâmetros, monitorar seu processamento, e exibir seus resultados.

7.3. Resultados Obtidos

Alguns resultados obtidos com a execução desses algoritmos e operações são mostrados a seguir, ilustrados pelas interfaces do SAPAF.

A Figura 7.2 mostra a Interface Padrão do SAPAF com a interface de visualização de objetos *TBinaryTable* e a janela de visualização de comentários mostrando um comentário feito para o objeto *TBinaryTable*.

A Figura 7.3 mostra a Interface Padrão do SAPAF com duas interfaces de visualização de objetos *TBinaryTable* abertas exibindo a matriz 0-1 original e a matriz 0-1 diagonalizada pelo DCA.

A Figura 7.4 mostra a Interface com o Usuário implementada para o algoritmo genético que é utilizada para selecionar parâmetros, controlar a execução e exibir resultados. Ela foi mostrada separada do SAPAF devido ao seu tamanho.

A Figura 7.5 mostra a Interface Padrão do SAPAF com duas interfaces de visualização: uma mostrando o diagrama de-para, e a outra mostrando um layout de blocos gerado pelo algoritmo MST a partir desse diagrama.

Figura 7.2 – Visualização de dados e comentários.

Figura 7.3 – Comparação visual de matrizes binárias.

Figura 7.4 – Interface de seleção, controle e visualização de resultados do algoritmo genético.

Figura 7.5 – Diagrama de-para e o resultado produzido pelo algoritmo MST.

CAPÍTULO 8. CONCLUSÕES

8.1. Resumo do Trabalho Efetuado

Diversas são as dificuldades verificadas no desenvolvimento de um projeto de arranjo físico. Dentre elas, podem ser citadas:

- dificuldade na escolha do tipo de arranjo físico capaz de satisfazer os objetivos do projeto;
- dificuldade na escolha do método e dos algoritmos mais adequados para a análise e formação do tipo de arranjo físico escolhido;
- dificuldade na obtenção, preparação e interpretação dos dados;
- dificuldade na escolha do resultado capaz de melhor satisfazer os objetivos do projeto.

Na tentativa de auxiliar o projetista a superar essas dificuldades, muitos autores têm proposto ferramentas computacionais desenvolvidas para auxiliar na aplicação de métodos ou na utilização de algoritmos e ferramentas gráficas empregados nos processos de análise e formação de arranjos físicos. No entanto, essas ferramentas possuem limitações em seu escopo de aplicação. Isso acaba dificultando também a escolha e aquisição dessas ferramentas.

Uma alternativa para minimizar essas dificuldades, seria a utilização de um software de apoio que se adaptasse às necessidades de um projeto de arranjo físico ou do próprio projetista. Pensando nisso, este trabalho de pesquisa buscou desenvolver uma solução computacional que conferisse a um software de apoio essa característica. A solução adotada toma como base um *framework* de aplicação orientado a objeto (OOAF) que utiliza componentes e estruturas dinâmicas armazenadas em uma biblioteca (BCA) para executar as Aplicações (instâncias do *framework*), e um banco de dados para manter o estado de processamento dos objetos que contém os dados obtidos e gerados em um projeto de arranjo físico.

Os elementos-chave dessa solução são as estruturas dinâmicas (Estruturas das Aplicações e Estruturas de Projeto), usadas para conectar e organizar os componentes das aplicações (tipos de dados, algoritmos, ferramentas gráficas, operações de suporte, etc.) empregados nos projetos de arranjo físico. Essas estruturas são modeladas e registradas em uma Biblioteca de Componentes Reusáveis (BCR), juntamente com seus componentes, por meio de um Software Gerenciador de Componentes das Aplicações (SGCA). As informações contidas na BCR são utilizadas pelo SGCA para gerar a BCA, que pode ser considerada uma versão binária da BCR. Com isso, as estruturas dinâmicas contidas agora na BCA são interpretadas na forma de roteiros que descrevem aos mecanismos do núcleo de controle do *framework* como utilizar os componentes para executar as Aplicações.

Essa forma de construção das Aplicações permite que novos componentes sejam adicionados ao software proposto sem necessitar de modificações estruturais ou gerar alterações em seu código fonte atual. Ao mesmo tempo, permite que diversas Aplicações diferentes sejam construídas rapidamente ou adaptadas para atender às solicitações e necessidades de quaisquer projetos de arranjo físico.

Para completar essa solução, o banco de dados adotado também seguiu a mesma idéia. Ele foi projetado com base em uma abordagem objeto-relacional que define uma camada de software sobre um banco de dados relacional que mapeia as propriedades dos objetos para campos em tabelas relacionais. Essa camada de software possui um conjunto de mecanismos que, ao detectarem que um novo componente de dados foi adicionado à BCA, mapeia suas propriedades para um repositório de metadados e gera automaticamente as estruturas (tabelas) necessárias para armazenar essas propriedades.

8.2. Conclusões Principais

A flexibilidade estrutural do software proposto permite a ele se adaptar as necessidades de um projeto de arranjo físico ou do projetista a partir da inclusão, em sua estrutura, de novos componentes de software empregados na análise e formação de arranjos físicos sem prejudicar o desenvolvimento atual do projeto ou obrigar o projetista a aprender e utilizar uma nova ferramenta.

Essa característica permite ao software proposto atuar em um escopo de aplicação mais amplo que o das ferramentas computacionais de sua categoria, utilizadas como a

mesma finalidade. A prova disso é a Aplicação apresentada no Capítulo 7 que permite trabalhar com três tipos de arranjos físicos diferentes.

A abordagem tecnológica adotada, assim com as tecnologias e técnicas empregadas no desenvolvimento do software proposto, podem ser facilmente adaptadas para desenvolver softwares em outros domínios de aplicação. A utilização de estruturas dinâmicas, armazenadas em uma biblioteca de componentes e interpretadas na forma de roteiros, permite a adaptação rápida de um software sem modificações estruturais, e propõem uma alternativa para construção de softwares muito desejada nos dias de hoje: construção rápida de softwares a partir de componentes reusáveis.

Outra vantagem apresentada por essa solução é a capacidade adaptativa conferida ao software desenvolvido, que permite configurá-lo de acordo com as necessidades atuais e futuros do cliente sem desenvolver dezenas de versões que acabam gerando diversas modificações estruturais, muitas vezes difíceis de serem controladas, principalmente em nível de banco de dados cujas mudanças podem gerar diversos tipos de efeitos colaterais e indesejáveis em um sistema computacional complexo.

8.3. Limitações do Trabalho

A primeira limitação deste trabalho reside na plataforma adotada para seu desenvolvimento. O módulo CIU e todas as Interfaces com o Usuário estão intimamente ligadas à plataforma Windows[®]. No futuro, se houver necessidade de desenvolver suas Aplicações para a Internet, tanto o CIU quanto as Interfaces com o Usuário necessitaram ser reconstruídas.

A segunda limitação deste trabalho é que, apesar do MCO ter sido inicialmente projetado para trabalhar com vários CIU's, ele foi implementado para trabalhar apenas com uma. No futuro, se houver a necessidade de compartilhar os dados dos projetos com várias aplicações simultâneas, seja em um ambiente mono usuário ou distribuído, ele necessitará ser modificado para poder trabalhar com um CIU por Aplicação em execução.

A terceira, e maior, limitação deste trabalho é a utilização da BCA em código fonte. Isso significa que toda vez que uma alteração for necessária, seja em alguma estrutura dinâmica ou em algum componente, o software deverá ser recompilado. Isso restringe a solução adotada e obriga o desenvolvedor a gerar versões diferentes do software proposto para cada Aplicação.

A quarta, e última, limitação deste trabalho é o SGCA. Em sua versão atual, ele auxilia a gerar a BCA mas não facilita muito o registro e a consulta das Estruturas das Aplicações e Estruturas de Projeto. Como as Estruturas de Projeto tendem a crescer, o grau de dificuldade para gerenciá-las pode aumentar a ponto de comprometer o desenvolvimento das Aplicações.

8.4. Trabalhos Futuros

O primeiro trabalho que pode ser desenvolvido é efetuar melhorias no SGCA para torná-lo mais interativo, desenvolvendo mecanismos e interfaces voltados para a modelagem das estruturas dinâmicas diretamente em seu ambiente.

Um outro trabalho prioritário e importante é a adoção de tecnologias que permitam gerar uma BCA que não necessite ser compilado juntamente com o *framework*. Isso permite desenvolver uma ferramenta variante do SGCA para registrar as Aplicações diretamente na BCA. Com isso, o cliente passa a ter uma Aplicação totalmente personalizada e adaptada às suas necessidades.

Outro trabalho menos prioritário é a criação de uma versão desse software para Internet para disponibilizar, por exemplo, resultados de um trabalho de análise de um cliente via WWW (*World Wide Web*), quando este for obtido em laboratório.

Por fim, um último trabalho sugerido é a aplicação da solução adotada em outros domínios de aplicação para explorar novas possibilidades de pesquisa e desenvolvimento de software baseado em reuso de componentes.

REFERÊNCIAS

- APPLE, J.M. (1977). *Plant Layout and Material Handling*. 3.ed. New York: John Wiley & Sons.
- ASKIN, R.G.; STANDRIDGE, C.R. (1993). *Modeling and Analysis of Manufacturing Systems*. New York: John Wiley & Sons.
- BAASE, S. (1978). *Computer algorithms: Introduction to design and analysis*. Reading, Massachusetts: Addison-Wesley Pub. Co.
- BALAKRISHNAN, J.; CHENG, C.H.; WONG, K.F. (2003). FACOPT: a user friendly FACility layout OPTimization system. *Computers & Operations Research*. v.30, p.1625-41.
- BILLO, R.E.; BIDANDA, B.; TATE, D. (1994). A genetic cluster algorithm for the machine-component grouping problem. *Journal of Intelligent Manufacturing*. v.7, p.229-41.
- BOOCH, G. (1994). *Object-Oriented Analysis and Design with Applications*. 2.ed. Redwood City, California: Benjamin Cummings.
- BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. (2000). *UML: Guia do Usuário*. Tradução de Fábio Freitas. Rio de Janeiro: Campus.
- BOSCH, J. et al. (1999). Building Application Frameworks: Object-Oriented Foundations of Framework Design. In: *Framework Problems and Experiences*. New York: John Wiley & Sons. Cap. 3, p.55-82.
- BROWN, E.C.; SUMICHRIST, R.T. (2001). CFGGA: a grouping genetic algorithm for cell formation problem. *International Journal of Production Research*. v.39, n.16, p.3651-69.
- COAD, P.; YOURDON, E. (1992). *Análise Baseada em Objetos*. São Paulo: Campus.
- COAD, P.; YOURDON, E. (1993). *Object-Oriented Design*. New Jersey: Prentice Hall.
- COLEMAN et al. (1996). *Desenvolvimento Orientado a Objeto: O Método Fusion*. 3.ed. Rio de Janeiro: Campus.
- DATE, C.J.; DARWEN, H. (1998). *Foundation for Object/Relational Databases: The Third Manifesto*. Reading, Massachusetts: Addison-Wesley.
- DE LIT, P. et al. (2001). Assembly planning with an ordering genetic algorithm. *International Journal of Production Research*. v.39, n.16, p.3623-40.
- DE LIT, P.; FALKENAUER, E.; DELCHAMBRE, A. (2000). Grouping genetic algorithms: an efficient method to solve the cell formation problem. *Mathematics and Computers in Simulation*. v.51, p.257-71.

- DRISCOLL, J.; THILAKAWARDANA, D. (2001). The definition line balancing difficulty and evaluation of balance solution quality. *Robotics and Computer Integrated Manufacturing*. v.17, p.81-6.
- FAYAD, M.E.; JOHNSON R.E. (1999). *Domain-Specific Application Frameworks: Framework Experience by Industry*. New York: John Wiley & Sons.
- FAYAD, M.E.; SCHMIDT, D.C.; JOHNSON, R.E. (1999). *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. New York: John Wiley & Sons.
- FONG, J. (2002). Translating object-oriented database transactions into relational transactions. *Information and Software Technology*. v.44, p.41-51.
- FOULDS, L.R. (1997). LayoutManager: A microcomputer-based decision support system for facilities layout. *Decision Support System*. v.20, p.199-213.
- GAMMA, E. et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. New York: Addison-Wesley.
- GHOSH, S.; GAGNON, R.J. (1989). A comprehensive literature review and analysis of the design, balancing and scheduling of assembly systems. *International Journal of Production Research*. v.27, n.4, p.637-70.
- GONÇALVES FILHO, E.V. (2001). *Arranjo físico de fábrica: um modelo para o processo de projeto e um algoritmo genético para a formação de células de fabricação*. 133p. Tese (livre docência) – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos. 2001.
- HASSAN, M.D.M. (2000). Toward re-engineering models and algorithms of facility layout. *Omega: The International Journal of Management Science*. V.28, p.711-23.
- HERAGU, S. (1997). *Facilities Design*. Boston: PWS Publishing Company.
- HUANG, H.; IRANI, S.A. (1999). Design of Facility Layouts Using Layout Modules: A Numerical Clustering Approach. Proceeding of the 8th Annual Industrial Engineering Research Conference, 1999, Phoenix, AZ, May 23-26.
- IRANI, S.A. (1999). *Handbook of Cellular Manufacturing Systems*. New York: John Wiley & Sons.
- IRANI, S.A. et al (2000). Production Flow Analysis and Simplification Toolkit (PFAST). *International Journal of Production Research*. v.38, n.8, p.1855-74.
- IRANI, S.A.; HUANG, H. (1998). Layout Modules: A Novel Extension of Hybrid Cellular Layouts. In: ASME INTERNATIONAL MECHANICAL ENGINEERING CONGRESS & EXPOSITION, Winter Annual Meeting of the ASME, 1998, Anaheim, CA, November 15-20.
- JACKSON, M. (1999). Thirty years (and more) of databases. *Information and Software Technology*. v.41, p.969-78.

- JACOBSON, I. (1992). *Object-Oriented Software Engineering*. New York: Addison-Wesley.
- KUSIAK, A.; CHO M. (1992). Similarity coefficient algorithms for solving the group technology problem. *International Journal of Production Research*. v.30, n.11, p.2633-46.
- LEE, Y.H.; LEE, M.H. (2002). A shape-based block approach to facility layout problems using hybrid genetic algorithm. *Computers & Industrial Engineering*. v.42, p.237-48.
- LIGGETT, R.S. (2000). Automated facilities layout: past, present and future. *Automation in Construction*. v.9, p.197-215.
- LUONG, L. et al. (2002). A decision support system for cellular manufacturing system design. *Computer & Industry Engineering*. v.42, p.457-70.
- MARKIEWICZ, M.E.; LUCENA, C.J.P. (2001). Object Oriented Framework Development. *ACM Crossroads Student Magazine*. Disponível em: <www.acm.org/crossroads/xrds7-4/frameworks.html> Acesso em: 26 mar. 2003.
- MONTREUIL B.; VENKATADRI, U.; RARDIN, R.L. (1999). Fractal layout organization for job shop environments. *International Journal of Production Research*. v.37, n.3, p.501-21.
- MUTHER, R. (1978). *Planejamento do Layout: Sistema SLP*. Vários tradutores. São Paulo: Edgard Blücher.
- OH, K.H. (1997). Expert Line Balancing System (ELBS). *Computer & Industry Engineering*. v.33, p.303-6.
- PARRISH, A.; DIXON B.; CORDES D. (2001). A conceptual foundation for component-based software development. *The Journal of Systems and Software*, v.57, p.193-200.
- PRESSMAN, R.S. (2002). *Engenharia de Software: Uma Abordagem Prática*. Tradução de Mônica Maria G. Travieso. 5.ed. Rio de Janeiro: McGraw-Hill.
- RAHAYU, J.W. et al. (2000). A methodology for transforming inheritance relationships in an object-oriented conceptual model to relational tables. *Information and Software Technology*. v.42, p.571-92.
- RAHAYU, J.W. et al. (2001). Performance evaluation of object-relational transformation methodology. *Data & Knowledge Engineering*. v.38, p.265-300.
- RUMBAUGH, J. et al. (1991). *Object-Oriented Modeling and Design*. Englewood Cliffs: Prentice-Hall.
- SLACK, N. et al. (1997). Arranjo Físico e Fluxo. In: *Administração da Produção*. Vários tradutores. São Paulo: Atlas. Cap.7, p.210-50.

SOMMERVILLE, I. (2001). *Software Engineering*. 6.ed. New York: Addison-Wesley.

STOIMENOV, L. et al. (1999). Bridging objects and relations: a mediator for a OO front-end to RDBMSs. *Information and Software Technology*. v. 41, p.57-66.

TAHARA, C.S. (2001). *Arquitetura para integração de métodos para apoio a decisão em formação de células de manufatura*. 156p. Tese (Doutorado) – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos. 2001.

TOMPKINS, J.A. et al (1996). *Facilities Planning*. 2.ed. New York: John Wiley & Sons.

VENKATADRI, U.; RARDIN, R.L.; MONTREUIL. B. (1997). A design methodology for fractal layout organization. *IIE Transactions*. v.29, p.911-24.

VOSNIAKOS, G.C.; ZIAAIE-MOAYYED, M.; MAMALIS A.G. (1997). Design of a system for computer-aided engineering of manufacturing facilities. *Computer Integrated Manufacturing Systems*. v.10, n.1, p.1-7.

GLOSSÁRIO

Componentes da aplicação – elemento da aplicação que foi implementado, de alguma maneira, para ser utilizado nas aplicações do software proposto.

Contexto da Aplicação – denominação genérica dada aos objetos da classe *TContext* usados para relacionar uma Aplicação a seus componentes das aplicações.

Contexto de projeto – denominação genérica dada aos objetos da classe *TContext* usados para relacionar um projeto ou subprojeto a seus componentes das aplicações

Contexto geral – objeto da classe *TContext* que contém todos os componentes da aplicação que podem ser disponibilizados livremente ao projetista.

Dados do projeto – dados obtidos e gerados durante o desenvolvimento de um projeto de arranjo físico.

Diagrama classe-objeto – diagrama que mistura representações gráficas de classes e objetos, relacionados entre si, usado para descrever as Estruturas das Aplicações, Estruturas de Projeto e parâmetros dos Procedimentos Operacionais.

Elemento da aplicação – denominação genérica dada aos quatro tipos de elementos essenciais identificados para o uso do software proposto como ferramenta de apoio: dados dos projetos, algoritmos, operações auxiliares e interface com o usuário.

Elementos do Metamodelo – forma genérica de referencia aos três tipos de elementos registros pelo SGCA na BCR definidos a partir das classes do Metamodelo

Estrutura da Aplicação – estrutura dinâmica usada pelos mecanismos de controle do *framework* para criar e executar uma Aplicação.

Estrutura de Projeto – estrutura dinâmica cuja composição determina quais componentes das aplicações podem ser empregados em um projeto de arranjo físico.

Estruturas de Roteiro – são estruturas na forma de árvores cujos nós contém os elementos do Metamodelo na forma de objetos de referência e suas ramificações representam os relacionamentos entre esses elementos.

Gabarito – modelagem padronizada definida de acordo com cada aplicação, criada para auxiliar na definição de novos componentes das aplicações.

Interface Padrão – interface principal da Aplicação que define o ambiente de trabalho do projetista e determina a forma de organização e disponibilização dos componentes das aplicações ligados as Estruturas de Projeto de uma Aplicação.

Interfaces com o usuário – representam quaisquer interfaces de software usadas para interação do projetista com os procedimentos do projeto ou com outras funções definidas para o software proposto

Operações auxiliares – rotinas de preparação e conversão de dados, e integração como outros sistemas.

Roteiro – representação dinâmica dos diagramas de classe-objeto gerados nos processos de modelagem das Estruturas das Aplicações, Estruturas de Projeto e descrição dos parâmetros dos Procedimentos Operacionais.

APÊNDICE A – Diagramas e gabaritos para utilização do
Framework.

Diagrama Classe-Object

O diagrama classe-objeto foi criado como uma ferramenta gráfica para auxiliar no processo de modelagem das estruturas dinâmicas. Ele recebeu esse nome porque combina classes e objetos, relacionados entre si, em um único diagrama. As classes representam os componentes das aplicações, seus ancestrais, ou procedimentos de seleção e visualização, e são descritas na forma de classes UML. Os objetos representam os elementos estruturais, isto é, objetos das classes auxiliares do Metamodelo (*TApplicationObject*, *TContext*, *TProcess* e *TParameter*), e são representados na forma de objetos UML. Os relacionamentos estabelecidos entre as classes e os objetos são determinados pelos relacionamentos estabelecidos entre as classes do Metamodelo das quais eles se originam. Por isso, recebem o mesmo nome. A Figura A.1 exemplifica esses elementos. Os retângulos de cantos vivos representa as classes e os retângulos de cantos arredondados representam os objetos. Em todos eles aparece um nome estereotipado que representa o nome da classe ancestral, ou o nome da classe do objeto. Os nomes que aparecem nos objetos são atribuídos a suas propriedades *Name*, as demais propriedades são mostradas explicitamente, como é o caso de *Type* nos objetos da classe *TParameter*. Note que os nomes dos relacionamentos são os mesmos que aparecem no Metamodelo. Isso auxilia no processo de registro das estruturas dinâmicas.

Figura A.1 – Exemplo de diagrama classe-objeto.

Algumas convenções foram adotadas para a utilização desse diagrama e com relação à nomenclatura das classes e dos objetos. Como pode ser observado na Figura A.1, o nome do par processa é sempre o nome da classe do objeto de dados (*TBinaryTable*) seguido do nome da classe do procedimento operacional

(*TROCAAlgorithm*), separados por ponto (“.”). Isso ajuda na identificação desses relacionamentos, pois lembra uma classe com seus métodos. Os nomes dos parâmetros, *NewTable* e *SourceTable*, são os nomes sugeridos para registrar os parâmetros. Isso facilita a implementação do procedimento operacional, pois ele depende desse nome para identificar o parâmetro e usá-lo. Os números ao lado dos relacionamentos *Agrega* indicam a ordem dos parâmetros. Note que alguns nomes de classes e de objetos aparecem na língua inglesa e outros na língua portuguesa. Existe uma lógica por trás disso que é muito utilizada nos gabaritos. As classes são codificadas usando palavras e termos da língua inglesa. Quando um nome de classe aparece em inglês, precedido pela letra “T” e com todas as palavras juntas, ele representa o nome verdadeiro da classe que será atribuído à propriedade *ClassName* dos objetos de referência. Quando ele aparece em português ele representa um nome fictício que deve ser substituído pelo nome real, ou é o nome que aparece na propriedade *Name* dos objetos de referência. No caso dos objetos, independente da língua, o nome que aparece é o nome que será atribuído à propriedade *Name* dos objetos de referência. Nesse caso a língua adotada depende da utilização da propriedade *Name*.

Os gabaritos a seguir auxiliam o desenvolvedor a construir uma Aplicação e a compreender como ela pode ser estruturas (Estrutura da Aplicação).

Gabarito para Interface Padrão

A Interface Padrão é o primeiro procedimento a ser executado em uma aplicação e ela necessita da Estrutura da Aplicação para poder disponibilizar suas operações

auxiliares iniciais. Sendo assim, é comum que ela tome como parâmetro de entrada o objeto da Aplicação para a qual ela está sendo executada e como saída, a Estrutura dessa Aplicação, que será enviada a ela por meio da árvore de componentes *TApplicationStructure* (ver Apêndice C). Essa forma de construção permite a uma interface padrão ser utilizada por várias Aplicações.

Gabarito para Abrir e Fechar um Projeto

Os gabaritos para abrir e fechar um projeto relacionam os Procedimentos Operacionais com suas Interface com o Usuário que são enviados como parâmetros aos métodos do MCO para abrir e fechar um projeto: *OpenProject()* e *CloseProject()*, respectivamente. Em ambos os casos, o par processo é relacionado ao contexto da Aplicação. A Interface Padrão pode utilizar as propriedades dos Procedimentos

Operacionais para identificá-los e utilizá-los corretamente, como por exemplo o seu nome (*Name*) ou sua identificação de classe (*ClassID*).

Gabarito para Criar Objetos de Dados

Para criar um Objeto de Dados, é necessário utilizar um truque. Em geral, os algoritmos e operações auxiliares são disponibilizados ao projetista à medida que os objetos de dados que eles processam são selecionados. Isso significa que o objeto de dados deve existir e estar visível ao projetista para que os algoritmos e operações auxiliares sejam disponibilizados. Quando ele ainda não existe, não há como disponibilizar uma operação para criá-lo, segundo essa lógica. A alternativa, é relacionar a operação para criar um objeto de dados com um objeto de dados que já existe, em outras palavras, um objeto de dados “parente”. Existem dois tipos de objetos de dados

que se encaixam nessa categoria, objetos das Aplicações e objetos de projeto. Por essa razão, os gabaritos mostrados para criar objetos e dados relacionam um objeto de dados genérico à classe *TApplicationObject* e a uma derivada da classe *TProjectObject*.

Essa forma de ligação também pode ser usada para ligar uma Estrutura de Projeto às Estruturas das Aplicações, ou como uma ramificação de outras Estruturas de Projeto.

Gabarito para Visualizar e Editar Objetos de Dados

Os gabaritos para visualizar e editar objetos de dados são simples e podem ser aplicados a quaisquer outros tipos de operações com funções similares, como é o caso da operação de impressão que pode ser modelada exatamente igual a uma operação de visualização. Note que nos dois gabaritos, as operações estão ligadas (contidas) no contexto geral, como descrito no texto da tese.

APÊNDICE B – Descrição das interfaces dos módulos e pontos
variáveis do *Framework*

Considerações Iniciais

A descrição a seguir mostra a declaração das classes com seus métodos e propriedades utilizados para interação entre objetos. Métodos e propriedades internos ou reservados não foram descritos por não serem acessíveis ou serem ignorados pelos demais objetos.

Para cada classe descrita são mostradas: sua declaração em Object Pascal do Delphi e uma descrição rápida de cada método ou propriedade.

Superclasse *TObjectClass*

Classe abstrata da qual são derivadas todas as classes que implementam os módulos do *framework* e as classes descritas no Metamodelo.

Declaração

```
TObjectClass = class
protected
  procedure HandleMessage(var Message: TObjectMessage); virtual;
  abstract;
  procedure PostMessage(var Message: TObjectMessage); virtual;
  function SendMessage(var Message: TObjectMessage): Integer;
  virtual;
public
  property Name: String read FName write FName;
  property ObjectType: TObjectType read FObjectType;
end;

TObjectMessage = record
  Sender,                //Remetente da mensagem
  Receiver: TObjectClass; //Destinatário da mensagem
  Param: Pointer;        //Parâmetro adicional
  Result: Integer;        //Valor retornado após tratamento da msg
case MessageType: TMessageType of
  smtLog,
  smtQuery: (
    SysMsgType: TSysMsgType;
    Message: String[255]; //Mensagem a ser registrada ou exibida
  );
  smtMessage,
  smtResult: (
    Msg: Integer;           //Número ou identificador da mensagem
    Data: array [0..255] of byte; //Dados associados à mensagem
  );
end;
```

Descrição dos Métodos

TObjectClass.HandleMessage

Método abstrato. É definido pelas derivadas de *TObjectClass* para recebe e tratar as mensagens enviadas por outros objetos via sistema operacional.

TObjectClass.PostMessage

Método virtual. É usado para postar uma mensagem para outro objeto. Mensagens postadas retornam imediatamente, não interrompendo a execução do serviço que o chamou. Caso haja resposta (*Message.MessageType = smtQuery*) ela será retornada via *HandleMessage*.

TObjectClass.SendMessage

Método virtual. É usado para enviar uma mensagem para outro objeto. Nesse caso, o serviço que o chamou aguarda o tratamento da mensagem pelo objeto receptor. Retorna o valor em *Message.Result* após a mensagem ter sido processada pelo destinatário.

Descrição das Propriedades

<i>Name</i>	Nome do objeto. É usado com diversas finalidades para identificação visual do objeto.
<i>ObjectType</i>	Tipo do objeto. É utilizado como uma forma de classificação que auxilia as rotinas de tratamento de eventos a identificarem os objetos e poderem obter propriedades específicas de cada um.

Classe *TUserInterfaceController*

Implementa o módulo CIU.

Declaração

```
TUserInterfaceController = class(TModule)
public
  procedure CloseInterface(AUInterface: TUserInterface);
  procedure HideInterface(AUInterface: TUserInterface);
  procedure ShowInterface(AUInterface: TUserInterface);

  property hOCE: HModule write FHOCR;
end;
```

Descrição dos Métodos

TUserInterfaceController.CloseInterface

Força o fechamento de uma interface com o usuário e o término do processamento.

TUserInterfaceController.HideInterface

Esconde uma interface do usuário e solicita a suspensão do processamento.

TUserInterfaceController.ShowInterface

Abre e exibe uma interface com o usuário.

Descrição das Propriedades

hOCE Recebe o manipulador do módulo MCO quando associado a ele.

Classe *TOperatingControlEngine*

Implementa o módulo MCO.

Declaração

```
TOperatingControlEngine = class(TModule)
public
  procedure CloseProject(AHProjectSession: HSession;
    ACloseProcedure, ACloseInterface: TSysObject);
  function Execute(AHSession: HSession; AProject, ADOject,
    AOPcedure, AUInterface: TSysObject): HSession;
  function NewProject(AHAppSession: HSession; ACreateProcedure,
    ACreateInterface: TSysObject): TSysObject;
  function NewSubproject(AHProjectSession: HSession; AProject,
    ACreateProcedure, ACreateInterface: TSysObject): TSysObject;
  function OpenProject(AHAppSession: HSession; AProject,
    AOpenProcedure, AOpenInterface: TSysObject): HSession;
  procedure Run(AppID: TObjectID);
  procedure SetUIC(AUIC: TUserInterfaceController);
end;
```

Descrição dos Métodos

TOperatingControlEngine.CloseProject

Fecha um projeto aberto pelo método OpenProject. Ele encapsula uma das formas de uso do método *Execute*, aguardando o final de sua execução para determinar se a sessão do projeto deve ou não ser fechada. Em geral, o procedimento de encerramento é usado para solicitar confirmação da ação. Se terminar normalmente, a sessão é fechada, senão, nada é feito.

TOperatingControlEngine.Execute

Executa um procedimento operacional para processar um objeto de dados. É a única maneira disponibilizada pelo núcleo do *framework* de operação para executar um procedimento operacional, inclusive a Interface Padrão para cada aplicação. Com exceção do parâmetro *AOProcedure*, os demais podem ser nulos.

Retorna a sessão de processamento aberta ou usada para executar o procedimento operacional.

TOperatingControlEngine.NewProject

Cria um novo objeto de projeto que será o objeto raiz de uma árvore de projeto. Ele encapsula uma das formas de uso do método *Execute* e aguarda seu término para solicitar ao GOD uma referência ao objeto de projeto criado.

Retorna um objeto de referência para o novo objeto de projeto que pode ser usado para solicitar sua abertura.

TOperatingControlEngine.NewSubproject

Cria um novo subprojeto (objeto de projeto que será vinculado a um outro objeto de projeto). Ele encapsula uma das formas de uso do método *Execute* e aguarda seu término para solicitar ao GOD uma referência ao objeto de projeto criado.

Retorna um objeto de referência para o novo objeto de projeto.

TOperatingControlEngine.OpenProject

Abre um projeto armazenado na base de dados. Ele cria uma sessão de projeto e encapsula uma das formas de uso do método *Execute*. Não aguarda o fim de sua execução. Isso permite que o Procedimento de Abertura possa ser executado indefinidamente, exibindo por exemplo a árvore do projeto.

Retorna a sessão segura do projeto aberto.

TOperatingControlEngine.Run

Executa uma Aplicação cuja identificação é dada por AppID. Este método cria uma sessão para a Aplicação e executa sua Interface Padrão via método *Execute*.

TOperatingControlEngine.SetUIC

Associa ao MCO um CIU. Caso já exista um CIU, finaliza-o e associa o novo CIU.

Classe *TDataObjectManager*

Implementa o módulo GOD.

Declaração

```
TDataObjectManager = class(TModule)
public
    function CancelDataObject(AHSession: HSession;
        AObject: TSysObject): TDataObject;
    function EditDataObject(AHSession: HSession;
        AObject: TSysObject): TDataObject;
    procedure FreeDataObject(AHSession: HSession;
        AObject: TSysObject);
    procedure Initialize;
    function LoadDataObject(AHSession: HSession;
        AObject: TSysObject): TDataObject;
    function NewDataObject(AHSession: HSession; AProject: TSysObject;
        AClassID: TClassID): TDataObject; overload;
    function NewDataObject(AHSession: HSession; AProject,
        AObject: TSysObject): TDataObject; overload;
    function PostDataObject(AHSession: HSession;
        AObject: TSysObject): TDataObject;
    function RegisterHistory(AHSession: HSession;
        AObject: TDataObject;
        AOPProcess: TOperatingProcedure): THistoryObject;

    property PIM: TProcedureAndInterfaceManager read FPIM
        write SetPIM;
end;
```

Descrição dos Métodos

TDataObjectManager.CancelDataObject

Cancela a edição de um objeto de dados dentro da sessão identificada por HSession, descartando todas as alterações efetuadas. Retorna o objeto de dados com os valores dos atributos anteriores a edição.

TDataObjectManager.EditDataObject

Carrega um objeto de dados, se necessário, e o coloca em modo de edição dentro da sessão identificada por HSession. Retorna o objeto de dados em modo de edição.

TDataObjectManager.FreeDataObject

Libera um objeto de dados de dentro da sessão identificada por HSession. Se o objeto de dados estiver em modo de edição, ele será automaticamente cancelado.

TDataObjectManager.LoadDataObject

Carrega um objeto de dados para ser usado dentro da sessão identificada por HSession. Retorna o objeto de dados carregado.

TDataObjectManager.NewDataObject

Cria um novo objeto de dados para ser usado dentro da sessão identificada por HSession e o vincula ao projeto identificado por AProject. Retorna o objeto de dados criado.

TDataObjectManager.PostDataObject

Atualiza as propriedades do objeto de dados em modo de edição dentro da sessão identificada por HSession na base de dados. Retorna o objeto de dados atualizado.

TDataObjectManager.RegisterHistory

Registra um histórico de processamento para o objeto de dados ADOject que foi processado por AOProcess. Retorna o histórico de processamento.

Descrição das Propriedades

PIM Contém o módulo GPI que foi associado a ele pelo MCO.

Classe *TProcedureAndInterfaceManager*

Implementa o módulo GPI.

Declaração

```
TProcedureAndInterfaceManager = class(TModule)
public
  function CreateComponentList(
    AClassID: TClassID): TAppComponentList; overload;
  function CreateComponentList(
    AObject: TSysObject): TAppComponentList; overload;
  function CreateComponentTree(
    AClassID: TClassID): TAppComponentTree; overload;
  function CreateComponentTree(
    AObject: TSysObject): TAppComponentTree; overload;
  function CreateOperatingProcedure(
    AOPcedure: TSysObject): TOperatingProcedure;
  function CreateSelectionProcedure(ASProcedure: TSysObject;
    AOPcedure: TOperatingProcedure): TSelectionProcedure;
  function CreateUserInterface(
    AUInterface: TSysObject): TUserInterface;
  procedure FreeObject(AObject: TSysObject);
  function GetApplication(AppID: TObjectID): TApplicationObject;
  function LoadComponentList(AResList: TAppComponentList): Boolean;
  function LoadComponentTree(AResTree: TAppComponentTree): Boolean;
end;
```

Descrição dos Métodos

TProcedureAndInterfaceManager.CreateComponentList

Cria a lista de componentes das aplicações identificada por AClassID ou AObject. Retorna a lista criada.

TProcedureAndInterfaceManager.CreateComponentTree

Cria a árvore de componentes das aplicações identificada por AClassID ou AObject. Retorna a árvore criada.

TProcedureAndInterfaceManager.CreateOperatingProcedure

Cria o procedimento operacional identificado por AOpcedure. Retorna o procedimento operacional criado.

TProcedureAndInterfaceManager.CreateSelectionProcedure

Cria o procedimento de seleção identificado por ASProcedure. Retorna o procedimento seleção criado.

TProcedureAndInterfaceManager.CreateUserInterface

Cria a interface com o usuário identificada por AUInterface. Retorna a interface com o usuário criada.

TProcedureAndInterfaceManager.FreeObject

Libera um objeto e o destrói.

TProcedureAndInterfaceManager.GetApplication

Retorna o objeto da Aplicação identificado por AppID com seu contexto da Aplicação e o procedimento operacional e a interface com o usuário de sua interface padrão.

TProcedureAndInterfaceManager.LoadComponentList

Carrega uma lista de componentes criada pelo método *CreateComponentList()*. Retorna True se teve sucesso, False se houve erro.

TProcedureAndInterfaceManager.LoadComponentTree

Carrega uma árvore de componentes criada pelo método *CreateComponentTree()*. Retorna True se teve sucesso, False se houve erro.

Classe *TDataObjectRepository*

Implementa o módulo ROD.

Declaração

```
TDataObjectRepository = class(TModule)
public
  function Insert(AObject: TDataObject): Boolean;
  function NewObjectID(AObject: TDataObject): TObjectID;
  function Select(AObject: TDataObject): Boolean;
  function Update(AObject: TDataObject): Boolean;
  function Link(AObject, ALinkedObject: TSysObject;
    ALinkState: TLinkState; ALinkProps: TLinkProperties): Boolean;
  function LoadLinks(AObject: TSysObject; ALinkList:
    TDataObjectLinkList; ALinkTypes: TLinkTypeSet): Boolean;
  function Unlink(AObject, ALinkedObject: TSysObject): Boolean;
  function Open(AObjectSet: TDataObjectSet): Boolean;
  function Close(AObjectSet: TDataObjectSet): Boolean;
end;
```

Descrição dos Métodos

TDataObjectRepository.Insert

Inseri um novo objeto de dados no banco de dados. Retorna True se teve sucesso, False se o objeto já existe ou ocorreu algum erro de acesso ao banco de dados

TDataObjectRepository.NewObjectID

Gera um novo identificador de objetos e associa-o a propriedade ObjectID de AObject, se este estiver instanciado (<> nil). Retorna o novo identificador de objetos se teve sucesso. Caso contrário, retorna 0.

TDataObjectRepository.Select

Carrega as propriedades de um objeto de dados armazenadas no banco de dados. Retorna True se teve sucesso, False se o objeto não existe ou ocorreu algum erro de acesso ao banco de dados.

TDataObjectRepository.Update

Atualiza as propriedades de um objeto de dados no banco de dados. Retorna True se teve sucesso, False se o objeto não existe ou ocorreu algum erro de acesso ao banco de dados.

TDataObjectRepository.Link

Inseri ou atualiza um registro de relacionamento de vínculo entre objetos de dados. Retorna True se teve sucesso, False se ocorreu algum erro de acesso ao banco de dados.

TDataObjectRepository.LoadLinks

Carrega a lista de relacionamentos de ALinkList com os objetos vinculados a AObject cujos tipos de vínculo estabelecidos são definidos em ALinkTypes. Retorna True se teve sucesso, False se ocorreu algum erro de acesso ao banco de dados.

TDataObjectRepository.Unlink

Remove um registro de relacionamento de vínculo entre objetos de dados. Retorna True se teve sucesso, False se ocorreu algum erro de acesso ao banco de dados.

TDataObjectRepository.Open

Estabelece a conexão de um conjunto de objetos de dados com o banco de dados de acordo com os critérios definidos em AObjectSet.Criteria. Retorna True se teve sucesso, False se ocorreu algum erro de acesso ao banco de dados.

TDataObjectRepository.Close

Fecha a conexão de um conjunto de objetos de dados com o banco de dados. Retorna True se teve sucesso, False se ocorreu algum erro de acesso ao banco de dados.

Classe TApplicationComponentLibrary

Implementa o módulo ROD.

Declaração

```
TApplicationComponentLibrary = class(TModule)
public
  function AncestorOf(AClassID: TClassID): TObjectRef; overload;
  function AncestorOf(AObject: TSysObject): TObjectRef; overload;
  function ClassIDOf(const AClassName: String): TClassID;
  function ClassNameOf(AClassID: TClassID): String;
  function GetScript(AClassID: TClassID; AObjectID: TObjectID;
    RelFilter: TRelTypeSet): TScript; overload;
  function GetScript(AObject: TSysObject; RelFilter:
    TRelTypeSet = cNormalRelTypes): TScript; overload;
  function GetAppScript(AppID: TObjectID): TScript;
  function GetDataObjectClasses(StartClassID: TClassID;
    ClassList: TSysObjectList): TSysObjectList;
```

```

procedure FreeObject(AObject: TSysObject);
procedure FreeScript(AScript: TScript);
function InstanceOf(AObject: TObjectRef): TSysObject; overload;
function InstanceOf(AClassID: TClassID;
  AObjectID: TObjectID = 0): TSysObject; overload;
function IsAncestor(AClassID, DClassID: TClassID): Boolean;
  overload;
function IsAncestor(AObject, DObject: TSysObject): Boolean;
  overload;
function ReferenceOf(AObject: TSysObject): TObjectRef; overload;
function ReferenceOf(AClassID: TClassID;
  AObjectID: TObjectID = 0): TObjectRef; overload;

property LastObjectID: TObjectID read FLastObjectID;
property FirstClassID: TClassID index 1 read GetClassID;
property LastClassID: TClassID index 2 read GetClassID;
end;

```

Descrição dos Métodos

TApplicationComponentLibrary.AncestorOf

Retorna um objeto de referência para a classe ancestral da classe identificada por AObject ou AClassID. *Nil* se não existe ancestral.

TApplicationComponentLibrary.ClassIDOf

Retorna o identificador hierárquico da classe cujo nome é dado por AClassName. 0 se não encontrou.

TApplicationComponentLibrary.ClassNameOf

Retorna o nome da classe cujo identificador hierárquico é dado por AClassID. '' se não encontrou.

TApplicationComponentLibrary.GetAppScript

Retorna o roteiro da aplicação contendo o contexto de projeto da mesma e o procedimento operacional e a interface com o usuário da interface padrão. Retorna *nil* se ocorrer um erro ou a classe não existir.

TApplicationComponentLibrary.GetDataObjectClasses

Método usado pelo ROD para obter a lista de Objetos de Dados novos. Retorna uma lista de objetos de referência com todas as classes de Objetos de Dados cujo ID é maior ou igual a StartClassID.

TApplicationComponentLibrary.FreeObject

Libera um objeto e o destrói. AObject pode ser uma instância ou um objeto de referência.

TApplicationComponentLibrary.FreeScript

Libera um roteiro e o destrói, incluindo seu objetos associados e agregados.

TApplicationComponentLibrary.InstanceOf

Retorna uma instância do elemento identificado por AClassID e AObjectID, ou AObject.

TApplicationComponentLibrary.IsAncestor

Retorna True se a classe AClassID ou objeto AObject for ancestral de DClassID ou DObject. False se não.

TApplicationComponentLibrary.ReferenceOf

Retorna um objeto de referência de AObject ou AClassID e AObjectID.

Descrição das Propriedades

<i>LastObjectID</i>	Contém o identificador do último objeto gerado por <i>InstanceOf()</i> .
<i>FirstClassID</i>	Contém o identificador da primeira classe da aplicação não reservada.
<i>LastClassID</i>	Contém o identificador da última classe da aplicação não reservada.

Superclasse *TDataObject*

Implementa a classe base para todos os objetos de dados.

Declaração

```

TDataObject = class(TSysObject)
protected
  procedure ChangeLink(AOldObject, ANewObject: TSysObject;
    ANewState: TLinkState); virtual;
  procedure DefineLinks(ALinkList: TDataObjectLinkList); virtual;
    abstract;
  procedure DefineProperties(APropList: TPropertyList); virtual;
  procedure EditLinks(ALinkList: TDataObjectLinkList); virtual;
    abstract;
  procedure LinkHistory(AHistory: TSysObject); virtual;
  procedure LinkProjectObject(AProject: TSysObject); virtual;
  procedure LoadLinks(ALinkList: TDataObjectLinkList); virtual;
  procedure LoadProperties(APropList: TPropertyList); virtual;
  procedure UpdateLinks(ALinkList: TDataObjectLinkList); virtual;
  procedure UpdateProperties(APropList: TPropertyList); virtual;

  property LinkList: TDataObjectLinkList read FLinkList;
end;

```

Descrição dos Métodos

TDataObject.ChangeLink

Modifica um vínculo existente. Usado pelo GOD para reassociar objetos de vínculos modificados ou desatualizados.

TDataObject.DefineLinks

Retorna em ALinkList uma lista de objetos de referência às classes dos Objetos de Dados que estabelecem vínculo de extensão e compartilhamento. Método usado pelo GOD para criar e relacionar os objetos de dados de um novo objeto de dados. Para cada elemento da lista, o GOD cria um novo objeto de dados que será carregado quando o método LoadLinks for chamado. As classes derivadas devem redefinir esse método para inserir seus vínculos.

TDataObject.DefineProperties

Método usado pelo ROD para obter em APropList a lista de propriedades do objeto de dados que deverão ser mantidas no banco de dados. Cada elemento da lista especifica uma propriedade. As classes derivadas devem redefinir esse método para inserir suas propriedades.

TDataObject.EditLinks

Método usado pelo GOD para obter os objeto de dados vinculados que devem ser colocados em modo de edição juntamente com o próprio objeto de dados quando este for colocado em modo de edição. Quando o GOD for atualizar o objeto de dados na base de dados, ele utilizará o método UpdateLinks para obter os objeto de dados vinculados que deverão ou não ser atualizadas juntamente com o objeto de dados.

TDataObject.LinkHistory

Método usado pelo GOD para relacionar um objeto de dados a seu histórico de processamento (vínculo de processamento).

TDataObject.LinkProjectObject

Método usado pelo GOD para relacionar um objeto de dados a seu objeto de projeto (vínculo de parentesco).

TDataObject.LoadLinks

Método usado pelo GOD para enviar ao objeto de dados a lista de objetos de dados vinculados por extensão e compartilhamento. A lista de vínculos enviada é agregada por valor para ser usada pelo objeto de dados para controlar seus vínculos. As classes derivadas devem redefinir esse método para obter seus vínculos.

TDataObject.LoadProperties

Método usado pelo ROD para enviar ao objeto de dados a lista de propriedades armazenadas no banco de dados. As classes derivadas devem redefinir esse método para obter suas propriedades.

TDataObject.UpdateLinks

Método usado pelo GOD para obter a lista de objetos de dados vinculados que devem ser atualizados no banco de dados. As classes derivadas devem redefinir esse método para atualizar seus vínculos.

TDataObject.UpdateProperties

Método usado pelo ROD para obter as propriedades a serem atualizadas no banco de dados. As classes derivadas devem redefinir esse método para atualizar suas propriedades.

Descrição das Propriedades

LinkList Contém a lista de objetos vinculados.

Superclasse *TOperatingProcedure*

Implementa a classe base para todos os procedimentos operacionais.

Declaração

```
TOperatingProcedure = class(TSysObject)
public
  procedure MakeVisible; virtual;
  function Resume: Boolean; virtual; abstract;
  function Execute: Boolean; virtual; abstract;
  function Suspend: Boolean; virtual; abstract;
  function Terminate: Boolean; virtual; abstract;

  property Aborted: Boolean read FAborted;
  property Canceled: Boolean read FCanceled;
  property Error: String read FError;
  property ParamCount: Integer read GetParamCount;
  property Params[Index: Integer]: TParameter read GetParams;
```

```

property SessionHandle: HSession read FhSession write FhSession;
property Suspended: Boolean read FSuspended;
property Terminated: Boolean read FTerminated;
end;

```

Descrição dos Métodos

TOperatingProcedure.MakeVisible

Envia uma mensagem à interface com o usuário de controle solicitando que fique visível ao usuário.

TOperatingProcedure.Resume

Deve ser implementado pelas classes derivadas para reiniciar um processamento suspenso. Retorna True se o processamento foi reiniciado com sucesso, False se não.

TOperatingProcedure.Execute

Deve ser implementado pelas classes derivadas para processar os objetos de dados. Retorna True se o processamento terminou normalmente, False se não.

TOperatingProcedure.Suspend

Deve ser implementado pelas classes derivadas para suspender o processamento. Retorna True se o processamento foi suspendido com sucesso, False se não.

TOperatingProcedure.Terminate

Deve ser implementado pelas classes derivadas para encerrar o processamento. Retorna True se o processamento foi encerrado com sucesso, False se não.

Descrição das Propriedades

<i>Aborted</i>	Indica se o processamento foi encerrado de maneira forçada.
<i>Canceled</i>	Indica se o processamento foi encerrado pelo usuário.
<i>ParamCount</i>	Contém o número de parâmetros na listas.
<i>Params</i>	Lista de parâmetros.
<i>SessionHandle</i>	Contém o manipulador de sua sessão de processamento.
<i>Suspended</i>	Indica se o processamento está suspenso.
<i>Terminated</i>	Indica se o processamento terminou normalmente.

Superclasse ***TUserInterface***

Implementa a classe base para todas as interfaces com o usuário.

Declaração

```
TUserInterface = class(TSysObject)
protected
  procedure DoClose; virtual;
  procedure DoInvisible; virtual;
  procedure DoOpen; virtual;
  procedure DoVisible; virtual;

public
  procedure Close;
  procedure Hide;
  procedure Open(AhUIC: HModule);
  procedure Show;

  property Active: Boolean read FActive;
  property Visible: Boolean read FVisible;
end;
```

Descrição dos Métodos

TUserInterface.DoClose

Deve ser implementado pelas classes derivadas para fechar ou destruir a interface.

TUserInterface.DoInvisible

Deve ser implementado pelas classes derivadas para tornar a interface invisível ao usuário.

TUserInterface.DoOpen

Deve ser implementado pelas classes derivadas para abrir/criar a interface.

TUserInterface.DoVisible

Deve ser implementado pelas classes derivadas para tornar a interface visível ao usuário.

TUserInterface.Close

Fecha e finaliza uma interface com o usuário e comunica a ação ao procedimento operacional vinculado.

TUserInterface.Hide

Torna uma interface invisível ao usuário e comunica a ação ao procedimento operacional vinculado.

TUserInterface.Open

Abre e inicializa uma interface com o usuário e comunica a ação ao procedimento operacional vinculado.

TUserInterface.Show

Torna uma interface visível ao usuário e comunica a ação ao procedimento operacional vinculado.

Descrição das Propriedades

Active Indica se a interface com o usuário está aberta.

Visible Indica se a interface com o usuário está visível.

Superclasse *TSelectionProcedure*

Implementa a classe base para todos os procedimentos de seleção.

Declaração

```
TSelectionProcedure = class(TSysObject)
protected
  function CreateDataObject(AClassID: TClassID): TDataObject;
    overload;
  function CreateDataObject(AClassName: String): TDataObject;
    overload;
  procedure FreeDataObject(AObject: TDataObject);
  function LoadDataObject(AObject: TSysObject): TDataObject;
public
  function Execute: Boolean; virtual; {abstract;}

  property Canceled: Boolean read FCanceled;
  property SelParamCount: Integer read GetSelParamCount;
  property SelParams[Index: Integer]: TParameter read GetSelParams;
  property SessionHandle: HSession read FhSession write FhSession;
end;
```

Descrição dos Métodos

TSelectionProcedure.CreateDataObject

Encapsula o acesso ao GOD para solicitar objetos de dados. Só pode criar objetos de dados das classes reservadas.

TSelectionProcedure.FreeDataObject

Encapsula o acesso ao GOD para solicitar a liberação de um objeto de dados criado por *CreateDataObject()* ou carregado por *LoadDataObject()*.

TSelectionProcedure.LoadDataObject

Encapsula o acesso ao GOD para solicitar a carga de um objeto de dados qualquer.

TSelectionProcedure.Execute

Deve ser implementado pelas classes derivadas para efetuar a seleção dos objetos de dados. Retorna True se selecionou os objetos de dados, False se não.

Descrição das Propriedades

<i>Canceled</i>	Indica se a seleção foi cancelada pelo usuário.
<i>SelParamCount</i>	Contém o número de parâmetros na lista de seleção.
<i>SelParams</i>	Lista de parâmetros selecionados.
<i>SessionHandle</i>	Contém o manipulador de sua sessão de processamento..

APÊNDICE C – Classes Reservadas do *Framework*

Visão Geral

As classes reservadas foram criadas para que as Aplicações pudessem acessar objetos de dado armazenados no ROD e aos roteiros e componentes contidos na BCA. Elas são derivadas das quatro superclasses mostradas na Figura C.1. Essas superclasses são reconhecidas pelo ROD, pelo GOD e pelo GPI que as instanciam e constroem.

Figura C.1 – Superclasses das classes reservadas.

As quatro superclasses mostradas na Figura C.1 possuem alguns métodos e atributos comuns cujas funções são basicamente as mesmas. A propriedade *Criteria* é usada para definir os critérios de seleção utilizados pelo ROD para filtrar objetos de dados e pelo GPI para obter os roteiros com os componentes. Sua forma de uso será descrita mais adiante. As propriedades *hPIM* e *hDOM* são referências aos módulos GPI e GOD, respectivamente, usadas pelos métodos dessas superclasses para enviar mensagens de comando para eles. A maioria desses métodos pré-definido o acesso ao GPI e ao GOD, enviando-lhes mensagens de comando, e podem ser redefinidos pelo desenvolvedor para definir comportamentos mais específicos. O método *Load()*, é usado para carregar uma lista ou árvore. Quando o GOD ou o GPI criam um objeto de uma das classes concretas dessas superclasses, elas ficam vazias, pois ainda não foram definidos os critérios de seleção necessários a sua carga. Os procedimentos operacionais então

inicializam as propriedades de *Criteria* e chamam *Load()* para efetuar a carga. No caso da superclasse *TAppComponentList* ela é construída pelo GPI que utiliza os critérios de seleção para obter os roteiros e adicionar seus objetos de referência à lista *Items* via método *Add()*. No caso das superclasses *TAppComponentTree* e *TDataObjectTree* elas se auto-constroem. Seus métodos *Load()* primeiro carregam a lista de componentes ou conjunto de objetos que iram usar e, em seguida, adicionam esses objetos como seus nós folhas ou novos ramos. Novos ramos são solicitados ao GOD ou GPI para que estes os instanciem. Em seguida o método *Load()* em execução define os critérios de seleção do novo ramo, e chama seu método *Load()* para que ele se auto-carregue. A superclasse *TDataObjectSet*, por ser tratada como uma extensão das tabelas de objetos contidas no banco de dados, define métodos similares aos de acesso a tabelas. Por essa razão, em vez de um método *Load()*, ela tem os métodos *Open()* e *Close()*. Esses métodos enviam comandos ao GOD solicitando a abertura ou o fechamento da conexão com essas tabelas. O GOD, por sua vez, chama os métodos do ROD *Open()* e *Close()* para executar o comando solicitado.

As superclasses *TAppComponentTree* e *TDataObjectTree* são usadas para obter listas de componentes e conjuntos de objetos de dados organizados na forma de uma árvore, cujo nó raiz contém o objeto que define um critério de seleção e seus nós folhas ou ramos contém os objetos selecionados por esse critério. Elas definem uma forma recursiva de construção que é determinada por suas derivadas concretas, cujas instâncias descrevem um componente ou objeto de dados e uma lista de outras instâncias da mesma classe. A regra básica para essa estruturação é bem simples. Os objetos instanciados de suas derivadas são classificados de duas maneiras: objeto ramo e objeto folha. Objetos folha descrevem apenas um componente ou objeto de dados e não possuem outros objetos agregados, dependendo de um objeto ramo para existir. Um objeto ramo, por sua vez, descreve um componente ou objeto de dados e contém uma lista de objetos agregados. Os objetos agregados podem ser objetos folha ou outros objetos ramo. O componente ou objeto de dados descrito pelo objeto ramo é o objeto que determina o critério de seleção para os objetos que serão agregados. Por exemplo, em uma árvore de projeto, um objeto ramo descreve um objeto de projeto, enquanto que os objetos agregados descrevem os objetos de dados vinculados ao objeto de projeto. Para obter os objetos que serão agregados, o objeto ramo utiliza uma instância de uma

derivada da classe *TAppComponentList* ou classe *TDataObjectSet* para selecionar os objetos desejados.

A Classe Abstrata *TCriteria* e suas Derivadas

A classe *TCriteria* é uma classe abstrata utilizada para definir critérios de seleção. Suas derivadas são utilizadas pelo ROD e pelo GPI para selecionar objetos de dados e obter roteiros respectivamente. A Figura C.2 mostra as derivadas de *TCriteria* com seus atributos. Apenas a classe *TOQLCriteria* não foi implementada nessa primeira versão por não ser necessária.

Figura C.2 – Superclasses *TCriteria* e suas derivadas.

TCriteria define dois atributos que são utilizadas pelo ROD e pelo GPI para determinar o tipo de critério que devem utilizar. O atributo *CriteriaType* determina o tipo de critério definido. O ROD o utiliza para determinar o conteúdo do atributo *Criteria* e o GPI para identificar se sua derivada é a classe *TScriptCriteria*. O atributo *Criteria* é uma *String* que contém uma expressão lógica ou comando de seleção usado pelo ROD para montar uma seleção (comando SQL ou OQL). O conteúdo de *Criteria* é definido pelas classes derivadas de *TCriteria* a partir dos valores de seus atributos. Como essas expressões lógicas ou comandos dependem do banco de dados, essas derivadas, com exceção de *TScriptCriteria*, são definidas como parte do ROD. O GPI ignora *Criteria* e utiliza os valores dos atributos da classe *TScriptCriteria* que são utilizados como parâmetros do método *BCA.GetScript()*.

A Superclasse *TAppComponentList* e suas Derivadas

A superclasse *TAppComponentList* é usada para obter listas de componentes. Ela descreve a interface usada pelo GPI para carregá-la e de acesso aos componentes em seu interior, que são armazenados como objetos de referência.

Para essa primeira implementação do *framework*, foi previsto apenas um tipo de lista que contém os componentes das aplicações disponibilizados ao projetista. Essa lista é implementada pela derivada *TAppClassList*, que redefine os métodos *Add()* e *Insert()* para filtrar os componentes que irá armazenar a partir do critério *ClassLevel*.

A Superclasse *TAppComponentTree* e suas Derivadas

A superclasse *TAppComponentTree* é usada para obter listas de componentes estruturados na forma de árvore, como descrito anteriormente. Ela descreve a interface de acesso aos componentes e ramos em seu interior.

Para essa primeira implementação do *framework*, foram previstos dois tipos de árvores: um para obter os componentes ligados às Estruturas das Aplicações (*TApplicationStructure*) e outro para obter os componentes ligados às Estruturas de Projeto (*TObjectStructure*). Ambas redefinem os métodos *Load()* para se auto-construírem e o fazem de maneira similar.

Elas foram implementadas para conter suas estruturas por completo, como todos os componentes ou apenas parte deles. Isso é determinado pelos critérios de filtragem definidos nos atributos de *TScriptCriteria*, que permitem filtrar esses componentes de acordo com seus códigos de classificação hierárquicos (*ClassID* e *ClassLevel*) e seus relacionamentos (*RelationSet*).

A Superclasse *TDataObjectSet* e suas Derivadas

A superclasse *TDataObjectSet* é usada para obter conjunto de objetos de dados. Ela descreve a interface de acesso aos objetos de dados contidos em seus conjuntos.

Para essa primeira implementação do *framework*, foram previstos dois tipos de conjuntos: um para consultas somente para leitura (*TDBQuerySet*) e outro para acesso direto (*TDBDataObjectSet*).

A classe *TDBQuerySet* é a classe básica de acesso ao objeto de dados que é reconhecida e controlada pelo ROD. Quando solicitada à abertura de um de seus objetos

via *ROD.Open()*, o ROD utiliza seus critérios de seleção para montar um comando SQL de seleção (*SELECT*) e executá-o através de um componente do Delphi definido para isso (*TQuery*) que é associado ao objeto *TDBQuerySet* para que seus métodos possam percorrer seu conteúdo e disponibilizá-lo como objetos de referência. Quando solicitado o fechamento desse objeto via *ROD.Close()*, este remove o objeto *TQuery* associado a ele e o destrói.

A classe *TDBObjectSet* é derivada da classe *TDBQuerySet*. Ela utiliza os mecanismos implementados pela classe *TDBQuerySet* para acessar os objetos de dados e redefine os métodos de acesso direto que são ignorados por ela (*Cancel()*, *Edit()*, *Insert()* e *Post()*) para permitir acesso direto aos objetos de dados no conjunto. *TDBObjectSet* é considerada uma classe básica de acesso direto aos objetos de dados e pode ser utilizada para definir novas classes com comportamentos mais específicos.

A Superclasse *TDataObjectTree* e suas Derivadas

A superclasse *TDataObjectTree* é usada para obter conjuntos de objetos de dados estruturados na forma de árvore, como descrito anteriormente. Ela descreve a interface de acesso aos objetos de dados e ramos em seu interior.

Para essa primeira implementação do *framework*, foram previstos três tipos de árvores: uma para obter objetos vinculados (*TLinkTree*), outra para obter as árvores de projeto (*TProjectTree*), e outra para obter os históricos de processamento (*THistoryTree*).

A classe *TLinkTree* foi projetada para ser a classe base de acesso a quaisquer objetos vinculados, independente do tipo de vínculo estabelecido. Ela encapsula e simplifica o processo de construção, tomando como conjunto de objetos uma instância da classe *TDBQuerySet* com critérios definidos por um objeto da classe *TLinkCriteria*.

As classes *TProjectTree* e *THistoryTree* são derivadas de *TLinkTree* e redefinem apenas seus critérios de filtragem.