

UNIVERSIDADE REGIONAL DE BLUMENAU
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
CURSO DE CIÊNCIAS DA COMPUTAÇÃO
(Bacharelado)

**REUTILIZAÇÃO DE SOLUÇÕES COM *PATTERNS* E
FRAMEWORKS NA CAMADA DE NEGÓCIO**

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE
REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA
DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA
COMPUTAÇÃO — BACHARELADO

MARCIO CARLOS GROTT

BLUMENAU, JUNHO/2003

2003/1-48

REUTILIZAÇÃO DE SOLUÇÕES COM *PATTERNS* E *FRAMEWORKS* NA CAMADA DE NEGÓCIO

MARCIO CARLOS GROTT

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO
PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE
CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

Prof. Marcel Hugo — Orientador na FURB

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

Prof. Marcel Hugo

Prof. Everaldo Artur Grhal

Prof. Ricardo Alencar Azambuja

DEDICATORIA

Este trabalho é dedicado a professora de informática, Sirlei de Fátima Albino da Escola Agrotécnica Federal de Rio do Sul, que desde suas primeiras aulas de informática que me incentivou a fazer um curso superior relacionado à informática.

Dedico também a professora de matemática Fátima Perez Zago de Oliveira da Escola Agrotécnica Federal de Rio do Sul, com a qual tive a oportunidade de poder lapidar mais os conhecimentos da aplicação da matemática em diversas áreas do conhecimento e principalmente na informática, por se constituir em ciência exata.

Este trabalho é dedicado em especial a Carlos Eugênio, Wilma Rossa e demais familiares que sempre acreditaram que podemos ser cada vez melhores e mais capazes de desenvolver nossas habilidades pessoais, intelectuais e profissionais.

AGRADECIMENTOS

Primeiramente agradeço a Deus por permitir que eu chegasse até aqui. Tantos foram os obstáculos, mas cheguei.

Ao meu orientador Marcel Hugo que tenho muito que agradecer. Obrigado pelos ensinamentos, pela objetividade e coerência com que conduziu este trabalho. Obrigado ainda pela amizade, confiança e consideração que teve por mim.

Agradeço à banca pelo cuidado e atenção que tiveram ao ler esta monografia e também pelas correções e comentários sugeridos.

A toda minha família, sobretudo meus pais Carlos e Wilma grandes incentivadores ao longo da minha vida e deste curso que nos últimos meses de faculdade deram muita ajuda para concluir este trabalho.

Ao senhor Oscar e Nora que me acolheram em seu lar quando vim morar em Blumenau até o presente momento, tratando-me como se eu fosse filho, neto e bisneto pois ambos já são tataravôs.

A todos os amigos que compartilharam comigo alguns momentos de tristeza e os vários momentos de alegria ao longo dos últimos anos do curso.

Aos funcionários da empresa Totall.com pela disposição em ajudar e ter concedido tempo para a concepção deste trabalho.

Agradeço a todos que de alguma forma contribuíram para o sucesso deste trabalho.

“Devia ter amado mais.
Ter chorado mais.
Ter visto o sol nascer.

Devia ter me arriscado mais,
e até errado mais.
Ter feito o que eu queria fazer.

Queria ter acreditado
as pessoas como elas são.
Cada um sabe a alegria,
e a dor que traz no coração.”

Epitáfio.

Sergio Britto

RESUMO

Este trabalho objetiva estudar padrões de projeto (*design patterns*) e *frameworks* para o desenvolvimento de um *framework* para cálculo de impostos incidentes em vendas de mercadorias. O aumento da complexidade dos sistemas, prazo menores de entrega e redução de custo tem contribuído para aplicação de padrões de projeto que descrevem soluções para problemas recorrentes e contribuem na formação de *frameworks*, sendo flexíveis e genéricos podendo ser reutilizados em diversas aplicações. Como resultado, chegou-se a um conjunto de informações que podem ser utilizados por desenvolvedores interessados em estudar reutilização e aplicação de padrões de projeto e *frameworks* para a melhoria de seu desenvolvimento de software. Além disto, foi desenvolvido um *framework* em Java para cálculo de impostos, aplicando diferentes padrões de projeto: *Singleton*, *Factory Method*, *Flyweight* e *Strategy*.

ABSTRACT

This work aims to study design patterns and frameworks for the development of a framework for calculating taxes in sales. The increase of systems complexity, short deadlines and cost reduction have contributed for application of design patterns that describe solutions for recurrent problems and contribute in the construction of frameworks. As result, it was reached a set of information that can be used by developers interested in studying reuse, design patterns and frameworks for improving software development. Moreover, it was developed a Java framework for calculating taxes in sales, applying different design patterns: Singleton, Factory Method, Flyweight and Strategy.

LISTA DE TABELAS

<i>Tabela 1: Requisito para reutilização no processo de desenvolvimento</i>	<i>50</i>
<i>Tabela 2: Descrição das vantagens do reuso de software</i>	<i>50</i>
<i>Tabela 3: Demonstração das desvantagens de reuso.</i>	<i>51</i>
<i>Tabela 4: Categorias de padrões mais importantes.....</i>	<i>60</i>
<i>Tabela 5: Obrigações de recolhimento de ICMS.....</i>	<i>89</i>
<i>Tabela 6: Situações em que a emissão da nota fiscal gera valores de tributação fiscal</i>	<i>90</i>
<i>Tabela 7: Itens relevantes na composição do pattern Singleto.....</i>	<i>94</i>
<i>Tabela 8: Demonstração do itens relevantes da composição do Factory Method.....</i>	<i>97</i>
<i>Tabela 9: Descrição dos itens relevantes do pattern Flyweight</i>	<i>100</i>
<i>Tabela 10: Descrição dos itens relevante ao pattern Strategy</i>	<i>102</i>

LISTA DE FIGURAS

<i>Figura 1: Arquitetura três camadas</i>	42
<i>Figura 2: Camada de Serviços de Apresentação (Interface com o Usuário)</i>	44
<i>Figura 3: Exemplo de uma coleção de padrões</i>	64
<i>Figura 4: Classificação de padrões</i>	64
<i>Figura 5: Níveis de projeto de software</i>	67
<i>Figura 6: Inversão de controle</i>	72
<i>Figura 7: Percepção de um framework entre aplicações de um mesmo domínio</i>	74
<i>Figura 8: Ilustra um framework caixa-branca, também com um único hot-spot R</i>	74
<i>Figura 9: Framework de caixa-preta</i>	75
<i>Figura 10: Framelet inspirado em um padrão de análise</i>	79
<i>Figura 11: Desenvolvimento de aplicações a partir de framelets</i>	80
<i>Figura 12: Demonstração esquemática do Pattern Singleton</i>	95
<i>Figura 13: Demonstração de um fragmento de código</i>	95
<i>Figura 14: Diagrama de classes da classe Emissor implementando o patterns Singleton</i>	96
<i>Figura 15: Demonstração de pattern Factory Method</i>	98
<i>Figura 16: Diagrama da classe abstrata Imposto</i>	99
<i>Figura 17: Demonstração da estrutura do pattern Flyweight</i>	101
<i>Figura 18: Diagrama de classe demonstrando o pattern Flyweight</i>	101
<i>Figura 19: Demonstração da estrutura do pattern Strategy</i>	103
<i>Figura 20: Classe abstrata imposto demonstrando a aplicação de pattern Strategy</i>	103
<i>Figura 21: Modelo da interface IntefaceCliente</i>	105
<i>Figura 22: Modelo da interface InterfaceProduto</i>	105
<i>Figura 23: Diagrama de classes do Framework</i>	106
<i>Figura 24: Diagrama de seqüência da criação de um objeto imposto(createImposto)</i>	107
<i>Figura 25: Diagrama de seqüência do cálculo de um imposto(getValorImposto)</i>	107
<i>Figura 26: Diagrama de Caso de Uso</i>	109
<i>Figura 27: Diagrama de classe do protótipo</i>	110
<i>Figura 28: Diagrama de Seqüência</i>	111
<i>Figura 29: Cadastro de informações dos Estados</i>	112
<i>Figura 30: Cadastro das informações de Clientes</i>	112
<i>Figura 31: Cadastro de informações do Produto</i>	113
<i>Figura 32: Cadastro de Informações do Emissor</i>	114

<i>Figura 33: Tela de atendimento ao cliente na compra de mercadorias.</i>	<i>115</i>
<i>Figura 34: Tela de atendimento a um cliente Micro Empresa.....</i>	<i>116</i>
<i>Figura 35: Cliente com tributação de ICMS e IPI.....</i>	<i>116</i>

SUMÁRIO

RESUMO.....	17
ABSTRACT.....	18
1 INTRODUÇÃO.....	27
1.1 Motivação do Trabalho.....	29
1.2 Objetivos do trabalho.....	29
1.3 Estrutura do trabalho	30
2 ARQUITETURA DE SOFTWARE.....	32
2.1 Introdução	32
2.2 Definição.....	33
2.3 Arquitetura de Aplicação.....	35
2.3.1 Introdução.....	35
2.3.2 Definição	35
2.3.3 Necessidades Atuais no Desenvolvimento de Aplicações	35
2.4 Mainframe.....	36
2.4.1 Aplicações Mainframe.....	36
2.4.2 Características.....	36
2.4.3 Ferramentas de métodos de desenvolvimento	36
2.4.4 Cacterísticas das Aplicações.....	37
2.4.5 Vantagens	37
2.4.6 Desvantagens	37
2.5 Aplicações em Rede	37
2.5.1 Características.....	38
2.5.2 Ferramentas e métodos de desenvolvimento.....	38

		23
2.5.3	Características das Aplicações.....	38
2.5.4	Vantagens	38
2.5.5	Desvantagens	38
2.6	Aplicações Cliente-Servidor	39
2.6.1	Características.....	39
2.6.2	Ferramentas e métodos de desenvolvimento	39
2.6.3	Características das Aplicações.....	40
2.6.4	Vantagens	40
2.6.5	Desvantagens	40
2.6.6	Cliente Gordo + Servidor Magro.....	40
2.6.7	Vantagens	40
2.6.8	Devantagens.....	40
2.6.9	Cliente Magro + Servidor Gordo.....	41
2.6.10	Desvantagens	41
2.6.11	Vantagens	41
2.6.12	Cliente Gordo + Servidor Gordo	41
2.7	Aplicações MULTICAMADAS (<i>Multi-tier</i>)	41
2.7.1	Motivação para uma arquitetura multicamadas.....	43
2.7.2	Camada de Interface	43
2.7.3	Camada de Regra de Negócio	44
2.7.4	União da tecnologia de <i>framework</i> e regras de negócios	46
2.7.5	Camada de Dados	47
2.7.6	Vantagens	47
2.7.7	Desvantagens	48
3	REUTILIZAÇÃO	49
3.1	Introdução	49
3.2	Requisito para Reutilização	49
3.3	As Vantagens da Reutilização	50
3.4	As Desvantagens da Reutilização	51

	24
3.5 Reutilização de Código.....	52
3.6 Reutilização de Herança	52
3.7 Reutilização de <i>Templates</i> (Modelos).....	53
3.8 Reutilização de Componente	53
3.9 Reutilização de <i>Frameworks</i>	54
3.10 Reutilização de Artefatos	55
3.11 Reutilização de Padrões	56
3.12 Reutilização de Componente de Domínio.....	56
4 PATTERNS.....	58
4.1 História	58
4.2 Definição de <i>Patterns</i>	59
4.3 Categorias de <i>Patterns</i>	59
4.4 O Processo de Aprendizagem	61
4.5 Componentes de um padrão	61
4.6 AntiPadrões.....	63
4.7 Catálogos de padrões.....	63
4.8 Processo de Desenvolvimento com Padrões	65
4.9 Projeto de software com padrões	67
4.10 Vantagens na Utilização de Padrões	68
4.11 Desvantagens na Utilização de Padrões.....	68
5 FRAMEWORKS.....	70

	25
5.1	Introdução 70
5.2	Definição 70
5.3	Inversão de Controle 71
5.4	Classificação dos <i>Frameworks</i> quanto À sua utilização 72
5.4.1	Horizontal e vertical 72
5.4.2	<i>Framework</i> de caixa branca x caixa preta 73
5.5	Desenvolvimento de aplicações a partir de <i>frameworks</i> 75
5.6	<i>Framelets</i> 76
5.6.1	Características..... 77
5.6.2	Desenvolvimento de <i>framelets</i> 78
5.7	Desenvolvimento de aplicações..... 79
6	IMPOSTOS 81
6.1	Descrição normativa dos Impostos 81
6.2	Estrutura do Imposto 82
6.3	Estrutura Normativa do IPI 82
6.3.1	Hipóteses Normativas de Incidência 83
6.3.2	Conseqüências Normativas..... 85
6.4	Estrutura Normativa do Imposto sobre Circulação de Mercadorias e Serviços (ICMS)..... 85
6.4.1	Incidências 85
6.4.2	Não incidências..... 86
6.4.3	Local da operação ou da prestação, ocorrência do fato gerador..... 87
6.4.4	Contribuinte 88
6.4.5	Conseqüência Normativa..... 88
6.4.6	Substituição Tributária 89
6.4.7	Retenção Micro-Empresa (M.E.)..... 91
6.4.8	Base Reduzida no Estado 92

	26
6.4.9 Redução em estados 7% e 12%	92
6.4.10 Cliente especial	93
7 DESENVOLVIMENTO DO FRAMEWORK.....	94
7.1 Patterns utilizados no desenvolvimento do Framework.....	94
7.1.1 Singleton	94
7.1.2 Factory Method	97
7.1.3 Flyweight	100
7.1.4 Strategy.....	102
7.2 Demais Interfaces e Classes que compõem o framework	104
7.3 Estudo de caso do Protótipo que utilizará o framework de cálculo de imposto ..	108
7.4 Diagramas do estudo de caso	108
7.5 Protótipo Implementado com o Framework.....	112
8 CONCLUSÃO	117
8.1 Limitações	118
8.2 Extensões	118
REFERÊNCIAS BIBLIOGRÁFICAS	119

1 INTRODUÇÃO

Nos últimos anos tem-se verificado o grande avanço no uso da tecnologia de orientação a objetos. Desde o início da década de 90 tem-se acompanhado, a princípio na comunidade acadêmica e após nas empresas, a crescente adoção da orientação a objetos para o desenvolvimento de aplicações. Este avanço deve-se, principalmente, às soluções propostas por esta tecnologia, as quais amenizam os efeitos dos problemas gerados pela crise de software. No começo, algumas empresas viam com ressalvas os benefícios trazidos pela orientação a objetos, devido a pouca produtividade inicial conseguida por seus desenvolvedores; entretanto, passados agora alguns anos, não existem mais dúvidas quanto à sua adoção, sendo apenas uma questão de tempo para completa utilização de seus princípios (GAMA, 2000; SINTES, 2002).

Como conseqüência da adoção da tecnologia orientada a objetos para o desenvolvimento de aplicativos, houve uma explosão de métodos, linguagens e ferramentas que implementam, de uma forma ou de outra, todos os conceitos apresentados por esta tecnologia. Hoje, na indústria de software, existe uma tentativa de padronizar uma linguagem para especificação de sistemas orientados a objetos, a *Unified Modeling Language* (UML). Em linguagens de programação tem-se uma série delas que favorecem a implementação dos requisitos da aplicação; só para citar alguns casos, tem-se C++, SmallTalk, e mais recente Java.

Para que os softwares possam atender os requisitos cada vez mais complexos, a utilização de orientação a objetos tem proporcionado aos desenvolvedores o uso de herança, encapsulamento e polimorfismo que permite ser feita a reutilização de soluções, o que tem sido procurado por toda a comunidade envolvida no desenvolvimento de novas soluções na implementação de sistemas (SINTES, 2002).

Os projetistas de sistemas devem encontrar os objetos e fatorá-los em classes no nível correto de granularidade, definindo as interfaces das classes e as hierarquias de herança estabelecendo as relações chaves entres eles, criando projetos específicos para resolver o problema, mas também genérico o suficiente para atender futuros problemas e requisitos (GAMA, 2000). Os objetos bem modelados fazem com que os programadores ganhem mais tempo na implementação. Depois de modeladas as classes bases e testadas, os

desenvolvedores precisam apenas se preocupar com a implementação da especialização da subclasse criada (ALUR, 2002).

Para se poder reutilizar uma maior quantidade de soluções, não basta ter escrito classes que descrevem toda a complexidade do projeto. Os projetistas novatos tendem a usar técnicas não orientadas a objetos pela sobrecarga de opções disponíveis. Sistemas cada vez mais complexos tendem a ter um número de classes cada vez maior, dificultando o desenvolvedor na codificação. Com o passar dos anos de experiência dos arquitetos em projetar soluções, observou-se que a essência dos problemas é semelhante e que quando especializavam as classes bases, o que é difícil conseguir na primeira vez, ganhavam maior produtividade no desenvolvimento, fazendo a reutilização de soluções corretas e que comprovadamente funcionavam (GAMA, 2000; SUN MYCROSYSTEM, 2002).

Observando que na essência os problemas assemelham-se, os projetistas começaram a documentar as soluções criadas durante os anos. Essa documentação foi sendo aprimorada e formaram um *pattern* de desenvolvimento. Os *patterns*, segundo Alur (2002), permitem documentar um problema conhecido e recorrente (que surgem várias vezes durante o projeto) e sua solução em um contexto específico e comunicar esse conhecimento para outras pessoas. Cada *pattern* tem uma aplicação específica dentro da arquitetura do software composto de multicamadas que separam a lógica de negócio, da interface do usuário e do acesso aos dados persistentes.

Sistemas em multicamadas é uma evolução dos sistemas monolíticos, que concentravam as regras de negócio, os dados e as interfaces do usuário em uma única máquina que predominantemente eram os *Mainframes*. Surgindo as redes locais que conectavam as máquinas, a arquitetura de sistema Cliente/Servidor difundiu-se separando as camadas de regra de negócio e interface dos dados. Conceitualmente uma aplicação pode ter múltiplas camadas, mas a mais popular é a em três camadas (*tree-tier*); constituindo em camada de interface do usuário, a camada de regras de negócio e acesso à banco de dados. Cada camada de abstração do sistema tem soluções especiais para problemas recorrentes em um contexto e, dessa forma, os *patterns* foram capturados em muitos níveis de abstração e em inúmeros domínios (GAMA, 2000; ALUR, 2002).

Os *patterns* podem ser agrupados em um *framework* (PREE, 1995). Booch (1994) define “*frameworks* como coleção de classes que fornece um conjunto de serviços para um

domínio particular”, contém um conjunto de classes no qual foram aplicados um ou vários *patterns* na sua modelagem. Os *frameworks* proporcionam um alto grau de reutilização, pois a partir do mesmo outros sistemas podem herdar funcionalidades reutilizando a solução desenvolvida e testada em vários outros sistemas já implementados ou que futuramente sejam implementados.

Hoje se tem um bom número de *patterns* catalogados onde uma análise destes permitirá compreendê-los, avaliá-los nos aspectos que tangem a reutilização da solução seguindo as características do projeto, elucidando sua aplicabilidade na resolução de problemas recorrentes e exemplificá-los com a implementação de soluções visando o reaproveitamento da solução implementada.

1.1 MOTIVAÇÃO DO TRABALHO

“Estamos constantemente resolvendo os mesmos problemas. Deve existir um padrão aqui em algum lugar” (Ambler, 1998). Este pensamento juntamente com a necessidade constante de economia de tempo no desenvolvimento faz com os desenvolvedores precisem reutilizar constantemente problemas já solucionados utilizando um padrão que se tornará parte de uma biblioteca de soluções utilizadas no desenvolvimento de aplicativos.

1.2 OBJETIVOS DO TRABALHO

O objetivo deste trabalho consiste em implementar um *framework* para cálculo de impostos em projetos de automação comercial utilizando a linguagem Java 2 SDK, verificando a melhor forma de solucionar um problema recorrente através de *patterns* para que proporcione um maior índice de reutilização da solução no desenvolvimento de sistemas.

Os objetivos específicos do trabalho:

- a) exemplificar o uso do *pattern* e *frameworks*;
- b) criar um *framework* com os *patterns* selecionados;
- c) desenvolver um material que ajudará os arquitetos de sistemas a decidirem qual a melhor solução a ser tomada no desenvolvimento de um projeto.

1.3 ESTRUTURA DO TRABALHO

O trabalho está composto em oito capítulos descritos a seguir.

O primeiro capítulo contém a explanação da visão geral do trabalho, mostrando a motivação, os objetivos, resumo e abstract do presente trabalho.

O segundo capítulo contém um apanhado histórico da evolução da arquitetura de sistemas desde os sistemas monolíticos passando para arquitetura sistema cliente/servidor e mostrando o que está sendo mais utilizado, a arquitetura em três camadas composta de camada de interface, regras de negócios e acesso à banco de dados. Descreve também as vantagens e desvantagens na alocação de camadas de sistema tanto no lado cliente quando servidor.

O terceiro capítulo retrata as formas de reutilização de código mais tradicionais. Descreve a reutilização mais primitiva e mais usual que é a reutilização de código sendo o tradicional copiar e colar o código que se pretende reutilizar. Discute a reutilização por herança, *templates*, componentes, *frameworks*, artefatos, *patterns* e finalizando com a reutilização de componente de domínio.

O quarto capítulo descreve como os *patterns* são criados a partir de problemas recorrentes encontrados durante o desenvolvimento de sistemas. Os *patterns* são categorizados de acordo com a área do problema que resolvem. Para cada *patterns* existe uma regra de formação seguindo alguns critérios usados para identificar e melhorar o vocabulário dos desenvolvedores.

O quinto capítulo mostra a criação de *frameworks*. Inicia com uma discussão sobre o que é um *framework*, como são formados e sua classificação de acordo com a visibilidade. A generalização junto com a flexibilidade constitui dois pontos que devem ser considerados quando se desenvolve um *framework*, pois o mesmo deve ser ao mesmo tempo genérico o suficiente para poder ser reutilizado e flexível para ser utilizado no desenvolvimento de aplicações. O capítulo demonstra a construção de *framelets* que são pequenos *frameworks* desenvolvidos para uma pequena parte de um sistema que poderá compor o *framework* principal.

O sexto capítulo mostra a formação dos impostos, como são utilizados no cotidiano quando um material troca de propriedade e há incidência de imposto. Demonstrem-se as fórmulas utilizadas para realização dos cálculos.

No sétimo capítulo é implementado um *framework* mostrando a utilização de quatro padrões de projeto que são modelados e implementados. O caso de uso hipotético utilizado para desenvolver um protótipo com a utilização do *framework* de cálculo de impostos demonstrando a utilização suas maneiras de reutilizar em outros projetos.

E por fim, a conclusão, mostrando os resultados obtidos com o trabalho, as suas limitações e possíveis melhoramentos.

2 ARQUITETURA DE SOFTWARE

2.1 INTRODUÇÃO

Inicialmente a programação de computadores era feita exclusivamente através de linguagem de máquina. Para programar um computador em linguagem de máquina é preciso conhecer bem a fundo a arquitetura de computadores, sendo necessário conhecer detalhes do processador para o qual os programas estão sendo feitos. Os recursos para programação em linguagem de máquina são bastante limitados, exigindo um grande esforço para solucionar problemas simples, sem falar na dependência que o programa em linguagem de máquina tem em relação à arquitetura de hardware para o qual ele foi escrito. Cada família de processador possui características próprias e instruções específicas, impossibilitando a execução de um mesmo programa em linguagem de máquina em processadores de famílias diferentes (RIOCCINI, 2000; SANTOS, 2003).

A programação de computadores diretamente em linguagem de máquina é uma tarefa extremamente complexa e pouco produtiva, o que limitava a disseminação da utilização dos computadores em ambiente que não possuíssem “cientistas” da computação.

Dois acontecimentos permitiram que a utilização dos computadores em larga escala fosse possível: o surgimento das linguagens de programação de alto nível e dos microcomputadores. As linguagens de programação de alto nível tornaram a programação de computadores uma tarefa extremamente mais simples e produtiva, ao mesmo tempo em que, os microcomputadores reduziram “astronomicamente” o preço do hardware.

Estes dois fatores permitiram que o uso da tecnologia da informação fosse disseminado nos níveis observados hoje. Historicamente a programação de computadores sempre foi definida como uma tarefa difícil, mesmo diante dos avanços tecnológicos alcançados, tais como: Programação Estruturada, Programação Orientada a Objetos, Ferramentas CASE, Sistemas Gerenciadores de Banco de Dados, Componentes, *Frameworks*, Arquitetura Cliente Servidor, Arquitetura Multicamadas, etc.

Pode-se ter uma sensação que a evolução tecnológica não reduziu a complexidade na programação de computadores, mas a questão é que, na mesma medida (ou talvez mais) que os recursos oferecidos pela tecnologia da informação evoluíam, a aplicação desta tecnologia também aumentava, tornando necessária uma constante evolução tecnológica a fim de

permitir que as novas necessidades para a tecnologia da informação fossem atendidas satisfatoriamente. Semelhantemente a um cachorro tentando pegar o rabo.

No início os profissionais de informática faziam PROGRAMAS, em seguida passaram a fazer SISTEMAS e atualmente fazem APLICAÇÕES (PRESSMAN, 1995).

PROGRAMAS eram executados em um único computador e atendiam a apenas uma pessoa, o usuário do computador (PRESSMAN, 1995; SOMMERVILLE, 2003).

SISTEMAS eram executados em vários computadores, um sendo o servidor e os demais clientes, localizados numa mesma empresa. O computador servidor fornecia serviços aos computadores clientes, normalmente armazenamento de dados. Os sistemas representavam aplicações departamentais e atendiam normalmente a centenas de usuários (PRESSMAN, 1995; SOMMERVILLE, 2003).

APLICAÇÕES são executadas em diversos computadores, possuindo diversos tipos de servidores e clientes, localizados geograficamente dispersos e atendendo normalmente a milhares de pessoas. As APLICAÇÕES normalmente representam o funcionamento das corporações integrando Clientes, Fornecedores, Parceiros e Funcionários. O bom funcionamento das aplicações é fundamental para a realização dos negócios da corporação, conseqüentemente para a sua existência, competitividade e evolução (PRESSMAN, 1995; SOMMERVILLE, 2003). É como se no início os profissionais de informática construíssem “*casinhas de cachorro*” e hoje precisam construir “*idades inteiras*”.

A evolução do nível de integração exigido pelas aplicações e a escala na casa dos milhares de usuário exigiram que a construção de software fosse levada mais a sério, pois a economia tornou-se dependente da tecnologia da informação para realização de negócios. Sendo necessário agora pensar em investir bastante no Projeto da Arquitetura de Aplicações.

2.2 DEFINIÇÃO

A palavra arquitetura está associada a arte e ciência de projetar e construir, levando em consideração os métodos, ferramentas e padronizações utilizadas no projeto e na construção. Não existe uma definição universalmente aceita para Arquitetura de Software. A Arquitetura de Software integra a disciplina Engenharia de Software e hoje tem sido o centro das atenções de vários estudos e pesquisas, devido à complexidade e importância que os softwares

alcançaram. Booch, Rumbaugh, e Jacobson (1999) definem arquitetura como um conjunto de decisões significativas sobre a organização de um sistema de software, a seleção de elementos estruturais e suas interfaces, assim como o relacionamento entre estes componentes.

Garlan e Shaw (1993) além dos algoritmos e estruturas de dados de um sistema; o projeto e especificação de todas as estruturas do sistema surgem como um novo tipo de problema. Aspectos estruturais incluem a organização e controle global das estruturas, protocolos de comunicação, sincronização, acesso a dados, elementos funcionais, distribuição física; projeto dos componentes, escalabilidade e performance e a seleção das alternativas de projeto.

Kazman (1997) define “a arquitetura de software de um programa ou sistema de computador é a estrutura ou estruturas do sistema, que abrange os componentes de software, as propriedades visíveis externamente desses componentes e o relacionamento entre eles”.

Por “propriedades visíveis externamente”, refere-se sobre as suposições que outros componentes podem fazer sobre componente, tais como: serviços fornecidos, características de performance, tratamento de erros, uso de recursos compartilhados, entre outras.

A arquitetura do software deve abstrair algumas informações do sistema (caso contrário não existe um ponto de vista da arquitetura e somente seria possível visualizar todo o sistema) e ainda assim fornecer informações suficientes para servir de base para análises, tomadas de decisões e redução de riscos.

Primeiro a arquitetura define componentes. A arquitetura incorpora informações sobre como os componentes interagem entre si. Isto significa que a arquitetura omite informações sobre os componentes que não se referem a suas interações (informações internas do componente) (LARMANN, 2000; PRESSMAN, 1995).

Segundo, a definição deixa claro que sistemas podem englobar mais de uma estrutura e que a arquitetura não define aspectos tecnológicos dos componentes e de seus relacionamentos. Para a arquitetura não importa se o componente de software é um processo, um objeto, uma biblioteca, um banco de dados, um produto comercial (LARMANN, 2000; PRESSMAN, 1995).

Terceiro, todo sistema de software possuem uma arquitetura, pois todo sistema é constituído por componentes e seus relacionamentos (LARMANN, 2000; PRESSMAN, 1995).

Quarto, o comportamento de cada componente é parte da arquitetura, desde que este comportamento seja relevante para utilização do componente. Este comportamento vai permitir que os componentes interajam entre si, o que claramente faz parte da arquitetura. Um componente é como uma caixa preta que não é necessário conhecer seu funcionamento interno e sim como utilizá-lo (LARMANN, 2000; PRESSMAN, 1995).

2.3 ARQUITETURA DE APLICAÇÃO

2.3.1 INTRODUÇÃO

Os modernos sistemas de informação em larga escala são muito complexos, são influenciados pelas constantes mudanças de padrões, combinam computação distribuída, diversas tecnologias e muitos sistemas e plataformas. Essa complexidade é intrínseca e não pode ser totalmente evitada, mas pode ser administrado com uma boa Arquitetura de Aplicação.

2.3.2 DEFINIÇÃO

Arquitetura de Aplicações define a organização estática dos sistemas aplicativos em um alto nível, lidam com a forma como esses sistemas são decompostos em elementos e como os sistemas interagem entre si. Boas arquiteturas são tolerantes a mudanças, compreensíveis e capacitam a funcionalidade do desenho dos sistemas desejados com uma performance aceitável.

2.3.3 NECESSIDADES ATUAIS NO DESENVOLVIMENTO DE APLICAÇÕES

Cada vez mais o desenvolvimento de sistema exigem necessidades que necessitam serem contempladas para atingir os objetivos tanto dos usuários quando no processo de desenvolvimento de uma empresa produtora de sistemas. A seguir alguns aspectos que devem ser considerados no desenvolvimento de sistemas segundo Santos (2003):

- a) produtividade no desenvolvimento (desde a concepção à implantação);
- b) agilidade na manutenção;

- c) integração com sistemas legados;
- d) escalabilidade;
- e) redução do custo de produção, treinamento e suporte ao usuário;
- f) agilidade e baixo custo na atualização de versões;
- g) segurança.

2.4 MAINFRAME

2.4.1 APLICAÇÕES MAINFRAME

Os sistemas inicialmente estavam centrados em uma única máquina, chamada de *Mainframe*, que tinha o controle total da aplicação concentrando as regras de negócio ou ambiente de negócio, o banco de dados e as interfaces do usuário. A aplicação era disponibilizada através de terminais mudos (burros, sem poder de processamento). Tinha um alto custo e atendiam a um grande número de usuários todos controlados pelo *Mainframe*. Os softwares escritos para esta tecnologia são freqüentemente monolíticos, isto é, a interface do usuário, regras de negócio e acesso aos dados estão todas contidas em uma única aplicação (RICCIONI, 2000; SANTOS, 2003).

2.4.2 CARACTERÍSTICAS

O processamento das informações de uma aplicação em um *Mainframe* caracteriza-se por ser processada exclusivamente em uma única máquina, o *Mainframe*. Os terminais que acessam a aplicação são designado terminais burros, pois não efetuam nenhum tipo de processamento local servindo apenas aos usuários para fazer iterações com o *Mainframe*. As iterações efetuadas pelos usuários causam grande tráfego de rede deixando-a muitas vezes lenta (SANTOS, 2003; RICCIONI, 2000).

2.4.3 FERRAMENTAS DE MÉTODOS DE DESENVOLVIMENTO

A utilização da linguagem de programação Cobol para o desenvolvimento de sistemas evoluindo para linguagens de 4ª geração. A persistência de dados feita em um banco de dados hierárquica acessado pelos terminais. A análise estruturada clássica utilizada para modelagem da aplicação junto com programação estruturada (SANTOS, 2003; RICCIONI, 2000).

2.4.4 CACTERÍSTICAS DAS APLICAÇÕES

As características significativas da aplicação podem-se destacar o processamento em lote (*batch*) e até *on-line* com alguns milhares de usuários e interfaces baseadas em caracteres.

2.4.5 VANTAGENS

Algumas vantagens de aplicações em *Mainframe* são descritas a seguir segundo Santos (2003):

- a) facilidade de gerência da segurança;
- b) facilidade de gerência de usuários;
- c) facilidade de gerência de aplicações;
- d) facilidade de integração de sistemas.

2.4.6 DESVANTAGENS

Algumas desvantagens das aplicações em *Mainframe* são descritas a seguir segundo Santos (2003):

- a) processamento centralizado permitindo apenas escalabilidade vertical;
- b) alto custo (milhões de dólares/ano);
- c) arquiteturas de hardware, software e comunicação totalmente proprietárias levando à dependência do fornecedor;
- d) interface da aplicação limitada a caractere;
- e) usuário sem autonomia.

2.5 APLICAÇÕES EM REDE

Com introdução das redes locais (LAN) os PC's (computador pessoal) se conectaram e resolveram alguns problemas, mas criaram outros. Ficou mais fácil alguma tarefa do escritório como o compartilhamento de arquivos, impressoras, surgindo o trabalho em colaboração, formando grupo de trabalho próximos ou espalhados pelo mundo trocando informações de um modo confiável, mas tendo sérias dificuldades de administração de gerenciamento da LAN.

2.5.1 CARACTERÍSTICAS

As características mais evidentes em uma aplicação em rede são disponibilidades de servidores de serviços compartilhados (ex. arquivos, impressão) com boa capacidade de processamento com uma boa velocidade de rede (10, 100 Mbps). A formação de estações de trabalho com bom poder de processamento em aplicações locais e interface gráfica.

2.5.2 FERRAMENTAS E MÉTODOS DE DESENVOLVIMENTO

O desenvolvimento de sistemas em rede utiliza análise essencial para especificação do sistema. Desenvolvimento em linguagens que dêem subsídio ao gerenciamento de arquivos específicos como, por exemplo, Clipper, Pascal, Cobol.

2.5.3 CARACTERÍSTICAS DAS APLICAÇÕES

Caracteriza-se por aplicações departamentais *on-line* com até dezenas de usuários. A interface da aplicação baseada em caracteres, mas com mais recursos visuais e processamento local. As aplicações acessam servidores através do mapeamento de unidades de disco ou protocolos proprietários.

2.5.4 VANTAGENS

O processamento espalhado permite o uso de aplicações nas estações, servidores e estações de baixo custo, arquiteturas de software e comunicação proprietária, mas com recursos de integração entre plataformas, interface com o usuário montado localmente e com recursos gráficos e usuário com autonomia total são algumas das vantagens proporcionadas por esta arquitetura de aplicação.

2.5.5 DESVANTAGENS

As dificuldades de gerenciamento de segurança, usuário, aplicações são algumas das desvantagens apresentadas por este modelo de arquitetura de sistema. Pode-se destacar ainda a dificuldade de integração de sistemas e aplicações sendo executadas nas estações de forma independente, podendo acessar arquivos no servidor, mas sobrecarregando a rede.

2.6 APLICAÇÕES CLIENTE-SERVIDOR

A arquitetura Cliente/Servidor foi sobre vários aspectos, uma revolução na computação. Apesar de resolver os problemas das aplicações baseadas em *Mainframes*, a arquitetura Cliente/Servidor não está livre de falhas. Por exemplo, a funcionalidade de acesso à banco de dados (como consulta (*Query*)) e regras de negócios, estavam embutidas no componente Cliente, e qualquer alteração nas regras de negócio, ou no acesso ao banco de dados, ou mesmo no próprio banco, freqüentemente obrigavam a atualização do componente para todos os usuários da aplicação, resultando em uma redistribuição da aplicação segundo Raccioni (2000).

Com este novo modelo Cliente/Servidor, tem-se a distribuição dos componentes da aplicação, dividindo os componentes da seguinte maneira: os componentes de base de dados ficam no servidor; os componentes da interface do usuário ficam no cliente e os da regra de negócio em um ou no outro, ou em ambos (RICCIONI, 2000; LARMAN, 2000).

Uma mudança na parte do componente do cliente requer a troca total ou parcial (executável ou pacotes de executáveis) e tem que ser distribuída para cada usuário. Este modelo Cliente/Servidor ficou conhecido como arquitetura de duas camadas (*two-tier*). Outra desvantagem da utilização deste modelo de duas camadas é a incapacidade de representar a lógica da aplicação em componentes separados, o que impede a reutilização do software. Também não é possível distribuir a lógica da aplicação para um computador distinto (LARMAN, 2000).

2.6.1 CARACTERÍSTICAS

Caracteriza-se por aplicações divididas em dois módulos: o Cliente e o Servidor. O cliente se comunica com o servidor através de protocolos de comunicação proprietários voltados para o serviço.

2.6.2 FERRAMENTAS E MÉTODOS DE DESENVOLVIMENTO

A metodologia utilizada para especificação de sistema na arquitetura cliente/servidor é a análise essencial e orientada a objetos. A persistência de dados utiliza banco de dados relacional e linguagem de programação como, por exemplo, Delphi e Visual Basic que

disponibilizam componentes nativos da linguagem para acesso a banco de dados e são orientadas a objetos.

2.6.3 CARACTERÍSTICAS DAS APLICAÇÕES

Caracteriza-se por aplicações departamentais *on-line* e com dezenas de usuários. A interface gráfica é mais amigável ao usuário possibilitando a configuração personalizada da interface.

2.6.4 VANTAGENS

Esta arquitetura diminui o tráfego de rede, pois no lado Cliente faz a solicitação de dados ao servidor e o processamento da informação se dá no lado Cliente.

2.6.5 DESVANTAGENS

As desvantagens encontram-se na dificuldade de gerenciamento de segurança, usuário, aplicações e integração de sistemas. Aplicações são executadas nas estações de forma independente, podendo acessar serviços no servidor, mas sobrecarregando a rede.

2.6.6 CLIENTE GORDO + SERVIDOR MAGRO

No Cliente concentra todas as regras de negócio, tornando o código confuso juntamente com a consistência de dados no Cliente. No Servidor o banco de dados utilizado apenas para guardar os dados.

2.6.7 VANTAGENS

Uso de ferramentas de desenvolvimento visual são muito mais produtivas, pois já se tem componentes nativos que abstraem a persistência de dados, protocolos de comunicação e acesso à banco de dados. Exemplos de linguagens de programação visual pode-se citar Delphi, Visual Basic, Visual Fox Pro, Visual C++.

2.6.8 DEVANTAGENS

Algumas desvantagens da aplicação Cliente/Servidor podem ser destacadas segundo Riccioni (2002) e Santos (2003):

- a) atualização de versão do programa cliente;

- b) performance (rede torna-se gargalo);
- c) interface, regra de negócio e acesso a dados muito acoplados.

2.6.9 CLIENTE MAGRO + SERVIDOR GORDO

Aplicação cliente sem regra de negócio servindo para entrada e saída de dados e servidor de banco de dados com regra de negócio em *stored procedures*, *packages* e *triggers*.

2.6.10 DESVANTAGENS

Este modelo apresenta desvantagens, pois as soluções de *stored procedures* são proprietárias, o que amarra a aplicação ao banco de dados, perda de escalabilidade horizontal, desenvolvimento em linguagens de programação proprietárias dos bancos de dados, normalmente procedural ou de scripts, com baixa reutilização de código e maior dificuldade de implementação.

2.6.11 VANTAGENS

Com a estratégia de colocar as regras de negócio no servidor, proporciona maior performance, já que a regra de negócio está no próprio banco de dados e geralmente esta em uma máquina equipada e configurada para vários acessos simultâneos com alta capacidade de processamento.

2.6.12 CLIENTE GORDO + SERVIDOR GORDO

Une todas as desvantagens das duas soluções.

2.7 APLICAÇÕES MULTICAMADAS (*MULTI-TIER*)

A arquitetura Cliente/Servidor (duas camadas, “*two-tier*”) foi substituída por múltiplas camadas. Conceitualmente uma aplicação pode ter qualquer número de camadas, mas a mais popular das arquiteturas multicamadas é a “*tree-tier*”, a qual divide o sistema em três camadas lógicas: a camada de interface de usuário, a camada de regras de negócios e a camada de acesso ao banco de dados mostrado na fig. 1. É possível acrescentar camadas adicionais e decompor ainda mais as existentes. Por exemplo, uma camada de Serviço pode ser dividida em serviços de nível mais alto e de nível mais baixo (por exemplo, geração de relatórios versus entrada/saída de arquivo) (LARMAN, 2000; SANTOS, 2003).

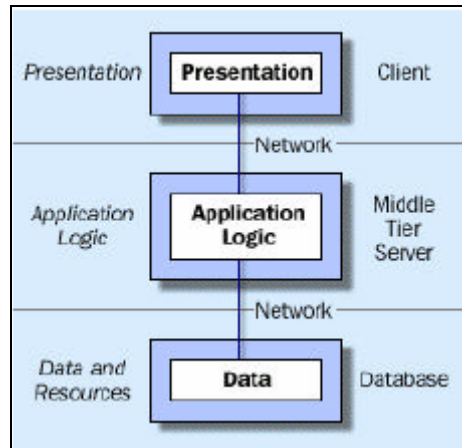


Figura 1: Arquitetura três camadas

A arquitetura Cliente/Servidor multicamadas adiciona à arquitetura de duas camadas importantes pontos; talvez o mais importante seja de tornar a aplicação menos frágil, isolando as alterações do cliente do restante da aplicação e também, permitindo maior flexibilidade no desenvolvimento de qualquer aplicação.

A arquitetura multicamadas reduz a fragilidade da aplicação, fornecendo maior isolamento entre as camadas. A camada de interface do usuário comunica-se somente com a camada de regras de negócio, nunca diretamente com a camada de acesso a dados. A camada de regra de negócio, por sua vez, comunica-se por um lado com a camada de interface de usuário e por outro lado com a camada de acesso a banco de dados. Deste modo, alterações feitas na camada de acesso a dados não afetarão a camada de interface com o usuário, pois são isoladas uma da outra. Esta arquitetura permite que alterações sejam feitas na aplicação com menor probabilidade de afetar o componente Cliente (o qual lembra-se, tem de ser redistribuído sempre que for alterado) (RACCIONI, 2000; LARMAN, 2000).

Segundo Raccioni (2000) a outra vantagem da arquitetura multicamadas é que os componentes de regra de negócios, acesso a dados, e o próprio banco de dados pode rodar em diferentes máquinas, distribuindo melhor a carga das aplicações, e tornando-as mais robustas e escaláveis, isto é, havendo a distinção entre cliente e servidor, sendo que o cliente pode fornecer componente, tal como um servidor.

2.7.1 MOTIVAÇÃO PARA UMA ARQUITETURA MULTICAMADAS

Os motivos para uma arquitetura multicamadas incluem segundo Larman (2000) e Riccioni (2000):

- a) isolar a lógica da aplicação em componentes separados que podem ser reutilizados em outros sistemas;
- b) distribuição de camadas em diferentes nós físicos de processamento e/ou diferentes processos. Isso pode melhorar o desempenho e aumentar a coordenação e o compartilhamento de informações em um sistema cliente/servidor;
- c) alocação de desenvolvedores para a construção de camadas específicas, tal como ter uma equipe trabalhando exclusivamente para a camada de apresentação. Isso suporta a especialização de conhecimentos, em termos das habilidades de desenvolvimento e a possibilidade de poder ter trabalhos executados em paralelo.

2.7.2 CAMADA DE INTERFACE

Os desenvolvedores podem atender a esse requerimento beneficiando-se no largo alcance da família dos ambientes operacionais gráficos, nos serviços contidos com o conceito dessa arquitetura. As famílias de ambientes operacionais gráficos, com sua interface de usuário e modelos de programação consistentes, permitem que os desenvolvedores construam sistemas que sejam executados nas mais diferentes configurações de hardware. Em casa, no escritório, na rua e na mesa de operações, existindo dispositivos baseados no padrão PC para atender a cada necessidade. Verifica-se na fig. 2 a camada de interface que interage com o usuário da aplicação recebendo as entradas de dados e fornecendo as respostas das requisições.

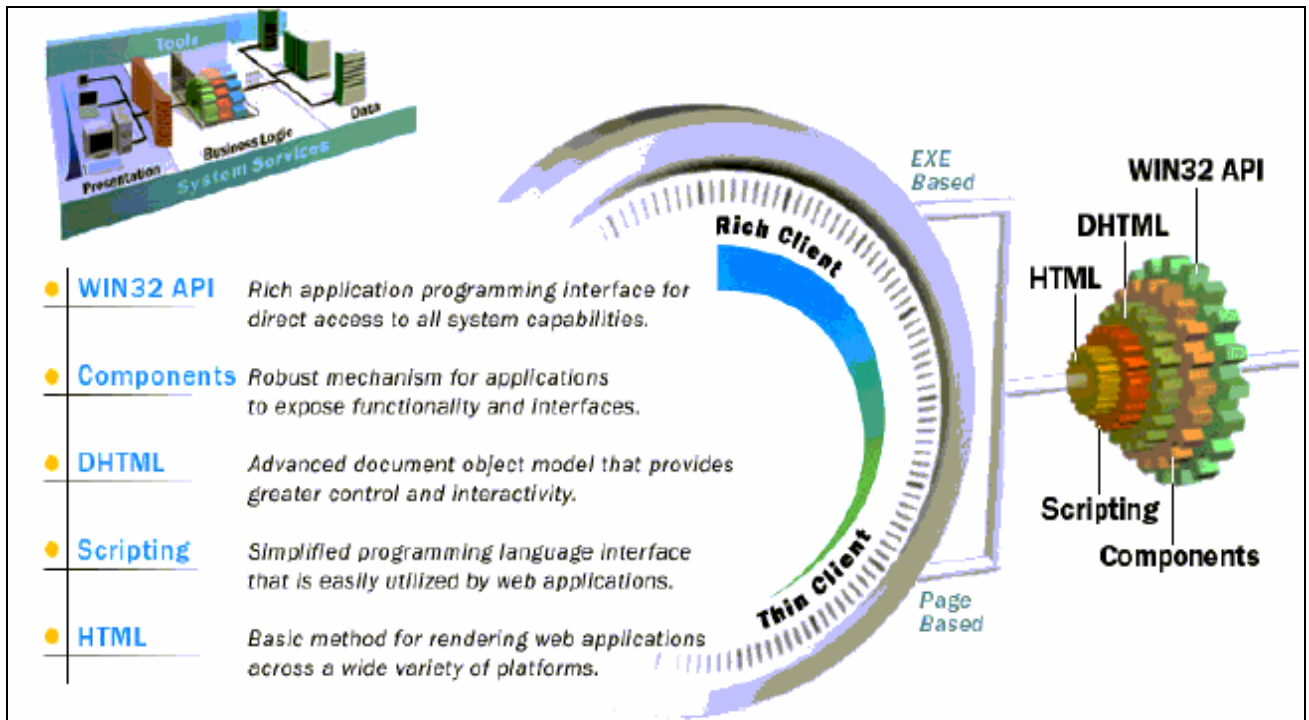


Figura 2: Camada de Serviços de Apresentação (Interface com o Usuário)

Os desenvolvedores podem construir interfaces de usuário utilizando HTML básico para atingir a maior variedade de *clients* - PC, Macintosh e UNIX - executando em diferentes *browsers*. Pode-se ainda construir interfaces de usuário usando HTML Dinâmico e *scripting* (Visual Basic® Scripting Edition ou JScript™), proporcionando a rica, flexível e interativa experiência que os usuários esperam dos sistemas baseados em interfaces gráficas. Em razão do suporte a diversas plataformas, fornecido pelos diversos *browsers* da Internet, essas interfaces de usuário podem ainda atender à grande variedade de *clients*. Os desenvolvedores podem embutir a lógica de negócio em componentes utilizando scripts que podem então ser colocados no *client* ou no servidor, conforme o dispositivo *client*.

2.7.3 CAMADA DE REGRA DE NEGÓCIO

O desafio de toda empresa no século 21 é como se tornar mais adaptativas às mudanças. Essas mudanças, em parte, significam alterações de regras do negócio. A alteração de regras de negócio implementadas em um sistema de informação se for feito de uma maneira amigável e parametrizada, dá um alto poder de flexibilização ao sistema. Esse recurso é viável caso uma arquitetura do sistema tenha sido bem definida e implementada, de modo que as regras tenham uma localização fixa e acessível. Entretanto, muitas das regras de negócio das empresas estão, de certo modo, perdidas, embutidas em códigos de programas de incontáveis aplicações legadas ou estão escondidas em antigos hábitos e processos

corporativos. Encontrá-las e entender os seus significados é uma operação difícil. Além disso, as empresas estão sempre correndo o risco de perder, por motivos diversos, as poucas pessoas que detêm as regras de negócios em suas mentes ou em antigos documentos e manuais.

A origem das regras de negócio praticamente coincide com a invenção do computador, pois sempre que existir um computador será necessário haver regras para regê-lo na forma de uma linguagem que o computador compreenda. O que mudou de lá pra cá, foi à tecnologia de como estas regras foram capturadas, criadas, armazenadas e interpretadas. O código fonte de uma aplicação se constitui de instruções, no formato de uma linguagem de programação, que especificam a lógica para se automatizar um procedimento. Esse procedimento pode se refletir sobre vários aspectos de uma aplicação como: especificação e controle de entradas e saídas de dados; transações e processamentos de dados; definição de interfaces e segurança; configurações de ambientes; manipulação de propriedades, atributos e métodos, etc. As regras de negócio estão distribuídas entre todos esses aspectos. O que diferencia um código fonte de regra de negócio de um código fonte comum, é que as regras de negócio devem conter conhecimento pertinente ao negócio abordado pela aplicação. A técnica de desenvolvimento utilizando regras de negócio implica em isolar o código fonte que contém regras de negócio, do código fonte comum, dando à aplicação um alto nível de flexibilidade.

Um bom exemplo de quando e como começou a se perceber as vantagens de isolar as regras de negócio de uma aplicação é descrito em DATE (2000). Foi com o surgimento das primeiras planilhas eletrônicas. As planilhas eletrônicas interpretavam as regras, na forma de macro-instruções, sendo suficiente para solucionar problemas da magnitude, por exemplo, de uma folha de pagamento de uma empresa. De acordo com HAY & HEALY (2003), regras de negócio podem ser conceituadas como: “Uma declaração que define alguns aspectos do negócio. Ela deve ser uma condição ou um fato (construída a partir de uma afirmação), uma limitação (construída a partir de uma ação), ou uma derivação. Ela deve ser atômica não podendo ser quebrada ou decomposta em regras de negócios mais detalhadas”. Perkins (2003) complementa o conceito de regras de negócio especificando melhor o seu domínio de atuação. Ele afirma que as regras de negócio são usadas para capturar e implementar uma lógica de negócio precisa em termos de processos, procedimentos e sistemas de informação nas organizações. Perkins relaciona o que regras de negócio podem ser:

- a) definições de termos do negócio;
- b) restrições de integridade de dados;
- c) funções matemáticas;

- d) inferências lógicas;
- e) seqüências de processo;
- f) relacionamentos entre fatos do negócio.

A forma mais comum de implementar a tecnologia de regras de negócio é delegar a responsabilidade de processamento aos Sistemas Gerenciadores de Banco de Dados (SGBD) através de implementações de mecanismos de gatilhos, procedimentos e restrições. Isso dificulta a utilização dessa funcionalidade por usuários leigos e restringe o campo de influência das regras. A alternativa é a utilização de ferramentas que implementam mecanismos de regras em ambientes e linguagens de desenvolvimento orientado a objeto. Mas o que as ferramentas fazem é gerar código fonte a partir da especificação das regras de negócio que depois devem ser compilados para gerar a aplicação. Estes mecanismos estão disponível no mercado, exemplo: RULEMACHINES (2003). Mas essa solução castra a propriedade de flexibilidade da aplicação finalizada. Outras ferramentas permitem uma configuração das regras em tempo de execução, mas são aplicações diferentes. É como se houvesse uma terceirização da gestão das regras de uma aplicação por outra aplicação.

Uma solução muito interessante seria a de que a própria aplicação possuísse meios de gerenciar suas próprias regras de uma forma independente sobre os aspectos de SGBD, ferramentas extras ou profissionais técnicos, sem que haja necessidade de manutenção do código fonte e re-compilação. As abordagens que descrevem como capturar, documentar, modelar e implementar regras de negócio casadas com um modelo de desenvolvimento de sistemas com qualidade resulta em um processo de desenvolvimento altamente produtivo e em um produto, no caso um software, flexível, altamente focado no negócio, que minimiza os custos de desenvolvimento e manutenção. Tudo isso ajuda às organizações que adotam essa tecnologia, a melhorar progressivamente e alcançar maiores níveis de qualidade.

2.7.4 UNIÃO DA TECNOLOGIA DE *FRAMEWORK* E REGRAS DE NEGÓCIOS

A idéia de reunir em um *framework* funcionalidades para gerenciar regras de negócios chega a ser uma tendência pela compatibilidade de ambas tecnologias. Propostas têm surgido, demandadas naturalmente, pela necessidade das organizações por melhores soluções para aprimorar o processo desenvolvimento de sistemas de informação. O desconhecimento e não

adoção em massa desse tipo de solução vem em parte em função da ainda não padronização dessa metodologia de desenvolvimento.

2.7.5 CAMADA DE DADOS

É utilizada pela camada de lógica de negócio para persistir o estado permanentemente. O núcleo da camada de dados é um ou mais banco de dados que hospedam e armazenam estados. Mais recentemente os desenvolvedores têm levado a combinação entre a camada de lógica de negócio com a camada de dados.

Pensando na primeira camada como um cliente e na segunda como um servidor, esta arquitetura efetivamente faz um cliente “magro” e um servidor “gordo”. Neste cenário tipicamente coloca-se alguma parte da camada de lógica de negócio (frequentemente à parte da lógica relacionada à persistência) dentro do banco de dados. Bancos de dados permitem executar lógica dentro do contexto do banco de dados, escrevendo um conjunto de módulos conhecidos como procedimentos armazenados (*stored procedures*). Colocando partes selecionadas da lógica de negócio em *stored procedures*, ganhando uma boa performance e escalabilidade. Assim, como se está colocando uma parte da lógica do negócio dentro do banco de dados, o tráfego na rede é reduzido.

Porém, os vários outros problemas da arquitetura em duas camadas permanecem. E enquanto o desenvolvimento de *stored procedures* representa um passo adiante, eles não resolvem todos os problemas. De fato, elas até acrescentam alguns novos problemas. Por exemplo, linguagens de *stored procedures* geralmente são proprietárias e a mudança a outro banco de dados requer o aprendizado da linguagem de *stored procedures* do outro banco e na migração desta parte da lógica para o outro banco.

2.7.6 VANTAGENS

Pode-se destacar algumas das vantagens oferecidas pela implementação da camada de banco de dados:

- a) grande facilidade de gerência de segurança;
- b) grande facilidade de gerência de aplicações;
- c) grande facilidade de gerência de usuários;
- d) grande facilidade de integração de sistemas: administração de componentes;
- e) Independência de banco de dados;

- f) grande escalabilidade horizontal através da criação de clusters de servidores de aplicação;
- g) boa performance;
- h) maior flexibilidade para a estruturação da arquitetura da aplicação;
- i) maior flexibilidade para incorporar o uso de outras tecnologias.

2.7.7 DESVANTAGENS

Maior dificuldade de implementação, pois requer um estudo mais aprofundado da estrutura de banco de dados e suportar os vários usuários concorrentes realizando operações simultâneas, tudo de maneira segura e eficiente para preservar a integridade dos dados.

3 REUTILIZAÇÃO

3.1 INTRODUÇÃO

Há um grande número de soluções implementadas a cada dia para resolver problemas específicos de situações de nosso cotidiano. Os projetistas tentam ser os mais práticos possíveis na implementação devido aos curtos prazos de entrega da solução, construindo implementações que são tão específicas ao problema que não podem ser reutilizadas em aplicações futuras.

O reuso tem como objetivo principal reaproveitar módulos de projetos que já tenham sido desenvolvidos, testados e implantados anteriormente com sucesso, visando uma diminuição considerável no tempo de desenvolvimento e uma facilidade maior de depuração, que, por consequência, desencadeiam os demais objetivos a serem alcançados.

A utilização da metodologia de orientação a objetos na análise de sistemas é um dos grandes avanços na construção de soluções que podem ser reutilizáveis. Infelizmente nota-se que as empresas não estão ainda acostumadas a utilizá-la em sua plenitude e ainda encontra-se resistência na sua adoção devido a projetos mal sucedidos ou por falta de conscientização da equipe de desenvolvimento.

O uso da metodologia de orientação a objetos em projetos de sistemas é uma tarefa difícil, mas a elaboração de projetos orientados a objetos reutilizáveis é ainda mais difícil (GAMMA, 2000). A reutilização não acontece no primeiro projeto, não acontece por acaso e nem pelo uso de ferramentas orientadas a objetos; é algo que precisa ser trabalhado arduamente e cuidadosamente na definição dos requisitos de objetos juntamente com a conscientização de toda a equipe de desenvolvimento.

Reutilização não compreende somente em reaproveitar código fonte. Isto não significa que está errado, muito pelo contrário, mas é a forma menos produtiva de reutilização. Sistemas não somente são feitos de código fonte; portanto, deve-se ser capaz de reutilizar mais do que código.

3.2 REQUISITO PARA REUTILIZAÇÃO

Existem três requisitos fundamentais para o projeto e o desenvolvimento de software baseado em reuso segundo Sommerville (2003) descrito na tabela 1.

Tabela 1: Requisito para reutilização no processo de desenvolvimento

Explicação
<p>Deve ser possível encontrar componentes reutilizáveis apropriados. As organizações necessitam de uma base de componentes reutilizáveis adequadamente catalogados e documentados. Deve ser fácil encontrar componentes nesse catálogo, se ele existir.</p>
<p>O responsável pelo reuso dos componentes precisa ter certeza de que os componentes se comportarão como especificado e de que serão confiáveis. Idealmente, todos os componentes no catálogo de uma organização devem estar certificados, a fim de confirmar que atingiram determinados padrões de qualidade. Na prática, essa situação não é realista e as pessoas em uma empresa aprendem de maneira informal sobre componentes confiáveis.</p>
<p>Os componentes devem ter a documentação associada para ajudar o usuário a compreendê-los e adaptá-los a uma nova aplicação. A documentação deve incluir informações sobre onde os componentes foram reutilizados e sobre quaisquer problemas de reuso que tenham sido encontrados.</p>

3.3 AS VANTAGENS DA REUTILIZAÇÃO

Uma das grandes vantagens no reuso de software é a diminuição do custo geral de desenvolvimento. Menos componentes de software precisam ser especificados, projetados, implementados e validados. Outras vantagens descritas na tabela 2.

Tabela 2: Descrição das vantagens do reuso de software

Benefícios	Explicação
Maior confiabilidade	Os componentes reutilizados que são empregados nos sistemas em operação devem ser mais confiáveis do que os componentes novos. Eles já foram experimentados e testados em diferentes ambientes.
Redução dos riscos de processo	Se recorrermos a um componente já existente, serão menores as incertezas sobre os custos relacionados ao reuso desse

	componente do que sobre custos de desenvolvimento.
Uso efetivo de especialistas	Em vez dos especialistas em aplicações fazerem o mesmo trabalho em diferentes projetos, ele podem desenvolver componentes reutilizáveis, que englobam seu conhecimento.
Conformidade com padrões	Alguns padrões, como os padrões de interface com o usuário, podem ser implementados como um conjunto de componentes-padrão.
Desenvolvimento acelerado	De modo geral, é mais importante fornecer um sistema para mercado o mais rápido possível do que se prender ao custo geral de desenvolvimento.

Fonte:Sommerville (2003).

3.4 AS DESVANTAGENS DA REUTILIZAÇÃO

O reuso não é sistemático e nem acontece por acaso devendo ser planejado e introduzido por meio de um programa de reuso empregado por toda a organização. Contudo, existem alguns problemas associados com o reuso que podem inibir a introdução desse método, significando reduções no custo total de desenvolvimento com o reuso serão menores do que as previstas conforme tabela 3.

Tabela 3: Demonstração das desvantagens de reuso.

Problema	Explicação
Aumento nos custos de manutenção	Se o código-fonte de componente não estiver disponível, então os custo de manutenção poderão aumentar, uma vez que os elementos reutilizados no sistema podem ser tornar crescentemente incompatíveis com as mudanças do sistema.
Falta de ferramentas de apoio	Os conjuntos de ferramentas CASEM não apóiam o desenvolvimento com reuso. Pode ser difícil ou impossível integrar essas ferramentas com um sistema de biblioteca de componentes.

Síndrome do “não-foi-inventado-aqui”	Alguns engenheiros de software às vezes preferem reescreverem componentes, porque acreditam que podem fazer melhor que o componente reutilizável.
Manutenção de uma biblioteca de componentes	Implementar uma biblioteca de componentes e assegurar que os desenvolvedores de software utilizem essa biblioteca pode ser dispendioso. As técnicas atuais de classificação, catalogação e recuperação de componentes de software são imaturas.
Encontrar e adaptar componentes reutilizáveis	Os componentes de software devem ser encontrados em uma biblioteca, compreendidos e, algumas vezes, adaptados, a fim de trabalharem em um novo ambiente.

Fonte: Sommerville (2003).

3.5 REUTILIZAÇÃO DE CÓDIGO

Reutilização de código é o tipo mais comum encontrado nas empresas de software, e refere-se a reutilização de código-fonte dentro de seções de uma aplicação e potencialmente de múltiplas aplicações. Isto é feito copiando e colando o código em outras seções e após modificando o código existente.

Este tipo de reutilização permite ao desenvolvedor ter acesso ao código-fonte a ser reutilizado e o mesmo, após o entendimento, decidir se quer ou não reutilizar.

A vantagem em reutilizar código-fonte é a diminuição da quantidade real de código que se necessita escrever, diminuindo os custos de desenvolvimento e prazo de entrega da aplicação. O efeito negativo é a limitação do efeito somente dentro da programação e geralmente aumentando o acoplamento dentro da aplicação.

3.6 REUTILIZAÇÃO DE HERANÇA

Um dos conceitos fundamentais em orientação a objetos é a herança, que se refere a utilização de comportamento implementado em outras classes existentes. A herança pode ser do tipo “é um”, “é como” e “é do tipo”. Por exemplo, uma classe Mamífero pode iniciar sua implementação herdando comportamento implementado na classe Animal.

Isto faz com que se tenha um maior grau de reutilização devido à parte da implementação já estar pronta, diminuindo os custos de desenvolvimento do sistema. Outra vantagem é que a classe a ser herdada já foi previamente testada. A desvantagem na reutilização através de herança é a perda da utilização de componentes que oferece um maior grau de reutilização. Os programadores novatos ao implementarem herança dificilmente testam a regressão da herança, teste de superclasses em subclasses, que geralmente resulta uma fraca hierarquia de difícil manutenção.

3.7 REUTILIZAÇÃO DE *TEMPLATES* (MODELOS)

Caracteriza-se pela reutilização da documentação. Documentos que auxiliam na elaboração de *layouts* de interfaces, código-fonte, casos de uso, relatórios de status, cronogramas de atividades, solicitações de mudanças, requisitos de usuários, arquivos de classes e cabeçalhos de documentação de métodos. Suas principais vantagens estão na organização, consistência e qualidade de artefatos de desenvolvimento utilizados pela empresa para documentação, manutenção e padrões internos. Sua desvantagem está quando os desenvolvedores modificam em benefício próprio os documentos. Estes modelos devem estar acessíveis, sempre atualizados e divulgados para que os desenvolvedores trabalhem com eles.

3.8 REUTILIZAÇÃO DE COMPONENTE

O desenvolvimento de sistemas baseados em componentes emergiu no final da década de 90 com uma abordagem baseada no reuso para o desenvolvimento de sistemas de software, motivado pela frustração do desenvolvimento orientado a objetos não tinha conduzido a níveis satisfatórios de reuso como originalmente foi sugerido. Segundo Sommerville (2003), as classes de objetos individuais eram muito detalhadas e específicas e tinham de estar associadas a uma aplicação em tempo de compilação ou quando o sistema estivesse conectado.

Reutilização de componentes refere-se ao uso de componentes pré-construídos e totalmente encapsulados no desenvolvimento de aplicações. Componentes são tipicamente auto-suficientes, encapsulam um único conceito, são mais abstratos do que as classes de objetos e podem ser considerados provedores de serviços *stand-alone* (SOMMERVILLE, 2003). Quando um sistema precisa de algum serviço, ele chama um componente para fornecer este serviço sem preocupar-se de onde este componente está sendo executado ou com a

linguagem de programação utilizada para desenvolvê-lo. A reutilização de componentes difere da reutilização de código pelo fato de que você não tem acesso ao código fonte. Diferença da reutilização de herança porque não faz criação de subclasses. Exemplos comuns de componentes são os *Java Beans* e componentes *ActiveX*

Há muitas vantagens na reutilização de componentes. Primeiro, ela oferece um escopo maior de reusabilidade do que o da reutilização de código ou de herança, porque os componentes são auto-suficientes - você literalmente os “pluga” e eles fazem o trabalho. Segundo, a utilização ampla de plataformas comuns, como sistemas operacionais e a *Java Virtual Machine* (JVM), provê um mercado amplo o bastante para que terceiros criem e vendam componentes a baixo custo. A principal desvantagem da reutilização de componente é que os componentes são pequenos e encapsulam um único conceito, se acaba precisando de uma grande biblioteca deles.

O caminho mais fácil para o trabalho com componentes é começar com objetos (*widgets*) da interface de usuário – barras de deslocamento, componentes gráficos e botões gráficos (para nomear alguns). Mas não se esqueça que há muito mais em uma aplicação do que a interface de usuário. Você pode ter componentes que encapsulam recursos do sistema operacional, tal como acesso à rede, ou que encapsulam recursos de persistência, tais como componentes que acessam bancos de dados relacionais. Se você está construindo seus próprios componentes, tenha o cuidado de que eles façam somente uma coisa. Por exemplo, um componente de interface de usuário para edição de endereços é bastante reutilizável por que você pode usá-lo em várias telas de edição. Um componente que edita endereços, endereços de e-mail e um número de telefone já não é tão reutilizável – não há muitas situações onde você queira todos esses três recursos simultaneamente. Em vez disso, é melhor construir três componentes reutilizáveis e usar cada um quando necessário. Quando um componente encapsula apenas um conceito, é um componente coeso.

3.9 REUTILIZAÇÃO DE FRAMEWORKS

Segundo Sommerville (2003), os primeiros proponentes do desenvolvimento orientado a objetos sugeriam que os objetos eram a mais adequada abstração para o reuso. Como explicado na seção anterior, os objetos nem sempre são de granularidade adequada e são específicos para uma aplicação em particular.

Reutilização de *framework* refere-se ao uso de coleções de classes que implementam em conjunto a funcionalidade básica de um domínio técnico ou de negócios comum. Os desenvolvedores usam *frameworks* como alicerce, a partir do qual eles constroem uma aplicação; os 80% comuns já estão no lugar, eles apenas necessitam acrescentar o restante, 20% específicos da aplicação. *Frameworks* que implementam os componentes básicos de uma interface (o mais conhecido *frameworks* de interface é o Modelo-Visão-Controle (MVC)) são muito comuns. Há *frameworks* para seguros, recurso humano, manufatura, bancos e comércio eletrônico. Reutilização de *framework* representa um alto nível de reutilização no nível do domínio. Os *frameworks* fornecem uma solução inicial para um domínio de problema e freqüentemente encapsulam uma lógica complexa que levaria anos para ser desenvolvida desde o rascunho. Infelizmente, a reutilização de *framework* sofre de diversas desvantagens. A complexidade dos *frameworks* torna-os difíceis de serem dominados, exigindo um longo processo de aprendizagem por parte dos desenvolvedores. *Frameworks* geralmente são de plataformas específicas, amarrando você a um único fornecedor, aumentando o risco de sua aplicação. Embora os *frameworks* implementem 80% da lógica necessária, são geralmente os 80% mais fáceis; à parte difícil, a lógica de negócio e os processos que são únicos de sua organização, ainda são deixados para o desenvolvedor da aplicação implementar. *Frameworks* raramente trabalham juntos a menos que eles venham de um mesmo fornecedor ou consórcio de fornecedores. Eles sempre exigem que você modifique seu negócio para adequar-se ao *framework* em vez de outro caminho.

3.10 REUTILIZAÇÃO DE ARTEFATOS

Reutilização de artefatos refere-se ao uso de artefatos de desenvolvimentos previamente criados – casos de uso, documentos-padrão, modelos específicos de domínio, procedimentos e diretrizes, e outras aplicações – para dar o chute inicial em um novo projeto (FURLAN, 1998). Há diversos níveis de reutilização de artefatos, indo dos 100% de reutilização pura, onde toma um artefato tal qual ele é e o utiliza em um novo projeto, quando apenas observa o artefato para ter uma idéia sobre como proceder. Por exemplo, documentos-padrão, tais como padronização de código e de interface, são artefatos valiosos para reutilização entre projetos, tanto como documentos de notação de modelagem e documentos de resumo de metodologia. A reutilização de artefatos promove consistência entre projetos e reduz a carga de gerenciamento para cada novo projeto.

Uma outra vantagem é poder freqüentemente adquirir vários artefatos ou encontrá-los *on-line*: padrões de interface de usuário são comuns para muitas plataformas, padrões de codificação para as principais linguagens estão geralmente disponíveis e metodologias padronizadas de orientação a objetos e notações de modelagem têm estado disponíveis há anos. A principal desvantagem da reutilização de artefato é que ela é freqüentemente percebida como reutilização superficial por programadores “*hard-core*” – você simplesmente passa os padrões e conectores de procedimentos de uma mesa para outra. O ponto (*baseline*) é que a reutilização de artefatos é uma técnica importante e viável que não deveria ser ignorada.

3.11 REUTILIZAÇÃO DE PADRÕES

Refere-se ao uso de abordagens publicamente documentadas para solução de problemas comuns. Padrões são freqüentemente documentados como um simples diagrama de classes e tipicamente compreendem de uma a cinco classes. Na reutilização de padrão, você não está reutilizando código; ao contrário, você está reutilizando a inteligência que está por trás do código. Padrões são uma forma de reutilização muito alta, que experimentará uma longa expansão de vida – pelo menos entre as linguagens de computação que você está atualmente utilizando e potencialmente pelo próprio paradigma de orientação a objeto.

3.12 REUTILIZAÇÃO DE COMPONENTE DE DOMÍNIO

A reutilização de componente de domínio refere-se à identificação e ao desenvolvimento de componentes de negócios reutilizáveis de larga escala. Um componente de domínio é uma coleção relacionada de classes de domínio e de negócios que trabalham juntas para suportar um conjunto coeso de responsabilidades. Um diagrama de componentes para uma empresa de telecomunicações tem diversos componentes de domínio, cada qual encapsulando muitas classes. O componente “OfertaDeServiços” encapsula mais de 100 classes indo desde uma coleção de classe de modelagem de planos de chamadas interurbanas até de pacotes de televisão a cabo ou de pacotes de serviços de Internet. Os componentes de domínio são inicialmente identificados dando-se uma abordagem dirigida à arquitetura para a modelagem, são modificados e incrementados durante o desenvolvimento de novas aplicações para a organização.

Componentes de domínio fornecem o maior potencial de reutilização, porque eles representam pacotes coesos de larga escala de comportamento de negócio que são comuns a

muitas aplicações. Tudo o que você produz durante o desenvolvimento de domínio deveria poder ser reutilizado. Componentes de domínio são efetivamente escaninhos arquiteturais nos quais os comportamentos de negócio são organizados e depois reutilizados.

As boas notícias sobre essas abordagens de reutilização é que você pode usá-las simultaneamente. Reutilização de *framework* freqüentemente prende você à arquitetura daquele *framework* e aos padrões e diretrizes que ele usa, mas ainda pode-se tirar vantagem das outras abordagens de reutilização com *frameworks*. As reutilizações de artefato e de componente são o ponto mais fácil para começar; com um pouco de pesquisa, pode-se rapidamente encontrar itens reutilizáveis.

4 PATTERNS

4.1 HISTÓRIA

Na área da informática, a evolução das capacidades computacionais e os seus requisitos crescem em ritmo acelerado. Os computadores duplicam sua capacidade de processamento a cada 18 meses (segundo a lei de Moore) e as novas tecnologias de transmissão de dados surgem a cada dia, de que modo e que a Engenharia de Software é capaz de responder a este crescente desenvolvimento?

A primeira solução aparece com a reutilização de código e de soluções anteriormente implementadas, com a intenção de aumentar a rapidez com que são criadas e a sua qualidade final. Mas mesmo assim, não é evitado um problema: “Quem desenvolve software tem muitas vezes a sensação de que está reinventando a roda“. É principalmente nas grandes empresas e sociedades de pesquisa e desenvolvimento que este fato toma uma dimensão maior. Dentro da mesma empresa pode estar um empregado tentando resolver um problema, enquanto que o colega no cubículo ao lado já tem a solução há meses. A principal razão desta questão é a inexistência de um veículo que facilite a comunicação e a partilha de conhecimentos por entre os vários indivíduos. Então, o que fazer para facilitar essa partilha de conhecimento? (ALUR, 2002, MESTKER, 2002, GAMMA, 2000).

A solução criada com a adaptação dos conceitos de padrões de arquitetura, desenvolvida por Christopher Alexander à área da Engenharia de Software. Os padrões aparecem como um método para facilitar essa comunicação de conhecimentos entre entidades e indivíduos. Devido à sua crescente popularidade, se pode encontrar documentada uma enorme quantidade de soluções de qualidade para problemas que surgem no dia-a-dia de uma empresa. Na área da Engenharia de Software, existem também diversos problemas comuns dos quais poderá depender a qualidade final do projeto, e por isso devem ser compreendidos e estudados com muito cuidado. Porém, se alguém já arranjou uma solução para o problema e conseguiu implementar com sucesso e qualidade, pode-se reutilizar o seu conhecimento.

Os padrões de software aparecem como uma forma de documentação em que os seus autores partilham experiências e soluções com as quais se viram confrontados no decorrer dos seus trabalhos, constituída por um problema, o seu contexto e a respectiva solução. Os padrões para serem reconhecidos seguem uma estrutura formal para serem posteriormente validados. Após o documento ter sido reconhecido como algo importante, este pode ser

publicado para que um principiante possa usar em seu proveito próprio os muitos anos de experiência que outras pessoas possam ter adquirido.

Existem alguns padrões bem conhecidos entre os engenheiros de software, mesmo que eles não tenham consciência disso. Por exemplo, um *proxy* num servidor é criada para facilitar os acessos e permitir uma maior eficiência, um *iterador* facilita a exploração de uma estrutura de dados, todos os *packages* e bibliotecas existentes para as diversas linguagens de comunicação, etc.

4.2 DEFINIÇÃO DE PATTERNS

Cada padrão descreve um problema que ocorre repetidas vezes em nosso ambiente, e então descreve o núcleo da solução para aquele problema, de tal maneira que se pode usar essa solução milhões de vezes sem nunca fazê-la da mesma forma duas vezes segundo Christopher Alexander (apud GAMMA, 2000).

Assim, um padrão descreve uma solução para um problema que ocorre com frequência durante o desenvolvimento de software, podendo ser considerado como um par “problema/solução”. Os projetistas e programadores que estão familiarizados com determinados padrões podem aplicá-los de imediato a problemas concretos, sem perder tempo e recursos a redescobri-los. Um padrão é, portanto, um conjunto de informações instrutivas que possui um nome e capta a estrutura essencial e o raciocínio de uma família de soluções já comprovadas para um problema repetido que ocorre num determinado contexto e para um conjunto de repercussões.

Segundo Gamma (2000) os padrões de projeto são descrições de objetos que se comunicam e classes que são customizadas para resolver um problema genérico de projeto em um contexto específico.

4.3 CATEGORIAS DE PATTERNS

Os padrões abrangem diferentes níveis de abstração, resultantes da enorme aplicabilidade prática que possuem, podendo classificá-los em diversas categorias, de modo a facilitar a sua recuperação e utilização num determinado problema. No entanto, esta classificação é um pouco flexível, podendo haver padrões que nitidamente se encaixam em

várias categorias. Na tabela 4 são descritas as categorias mais importantes de *patterns* segundo Alur (2002).

Tabela 4: Categorias de padrões mais importantes

Padrão	Explicação
Padrões de processo	Definem soluções para os problemas encontrados nos processos envolvidos na engenharia de Software: desenvolvimento, controle de configuração, testes, etc.
Padrões arquiteturais	Exprimem a organização das estruturas fundamentais de sistemas de software ou hardware.
Padrões de padrão (do Inglês, <i>patterns on patterns</i>)	São padrões que descrevem como um padrão deve ser escrito, ou seja, padrões que uniformizam o modo como estes devem ser apresentados aos seus utilizadores. Padrões de análise descrevem soluções para os problemas de análise de sistemas, incluindo conhecimento sobre o domínio específico de aplicação.
Padrões de projeto	Definem soluções para diversos problemas de projeto de software. Padrões de interface define soluções para problemas comuns no projeto de interfaces de sistemas. É um caso particular dos padrões de projeto.
Padrões de programação	Descrevem soluções particulares de programação numa determinada linguagem ou regras gerais de estilo de programação.
Padrões de persistência	Descrevem soluções para problemas de armazenamento de informações em arquivos ou bancos de dados.
Padrões de Hipertexto	Descrevem soluções para problemas que se encontram no projeto de aplicação direcionadas para a <i>World Wide Web</i> (WWW).

4.4 O PROCESSO DE APRENDIZAGEM

Os padrões para poderem ser aplicados e utilizados de forma coerente necessitam de um tempo de aprendizado seguindo alguns passos:

- a) aceitação;
- b) reorganização;
- c) divulgação.

Primeiro, aceita-se que os padrões são importantes para o trabalho. Então, se pesquisa o que é necessário conhecer sobre *patterns* a fim de utilizá-los. Finalmente, divulgam-se os detalhes suficientes sobre os *patterns* que poderão ajudar a resolver um problema.

4.5 COMPONENTES DE UM PADRÃO

Apesar de existirem diversos padrões de padrão, padrões têm sido descritos em diferentes formatos. Porém, existem componentes essenciais que devem ser claramente identificados ao se ler um padrão.

- a) o **nome do padrão** deve ser uma referencia concisa, significativa que se pode usar para nomear o padrão de projeto, descrevendo o conhecimento ou estrutura descrito pelo mesmo. A nomeação de padrões enriquece o vocabulário de projetistas, facilitando a comunicação entre projetistas e desenvolvedores, permitindo projetar em um nível mais alto de abstração;
- b) o **problema** descreve aonde se deve aplicar e objetivos do padrão perante o contexto. O problema pode representar a resolução de um algoritmo com a representação de classes. O problema poderá incluir as condições necessárias para que sejam satisfeitas;
- c) o **contexto** é pré -condição em que o problema e sua solução costumam ocorrer e para qual a solução é desejada, refletindo a aplicabilidade do padrão. Pode ser a configuração inicial do sistema antes da aplicação do padrão;
- d) as **forças** são descrições dos impactos, influências e restrições que são relevantes ao problema. Descrevem como interagem ou são conflitantes entre si e quais os objetivos que devem ser alcançados;
- e) a **solução** é um conjunto de relacionamentos estáticos, responsabilidades, colaborações e regras estabelecidas dinamicamente entre os elementos que compõem o projeto. A solução é uma descrição abstrata de como pode ser resolvido

- o problema sendo que a mesma não constituiu de um projeto concreto e muito menos de uma implementação em particular;
- f) o **exemplo** é uma ou mais maneiras que ilustram a utilização do padrão em um contexto inicial e transforma aquele contexto em um contexto final onde é aplicado;
- g) o **resultado do contexto** é o estado ou configuração do sistema após a aplicação do padrão, podendo ter conseqüências tanto positivas quanto negativas;
- h) as **conseqüências** são análises das vantagens e desvantagens da aplicação do padrão no projeto. As conseqüências poucas vezes mencionadas dão uma maior visão das alternativas de projetos, dos custos e benefícios. As conseqüências podem estar relacionadas a aspectos da linguagem de programação e implementação. Um dos aspectos considerados é o fator de reutilização de um projeto orientado a objetos, onde as conseqüências de um padrão incidem sobre a flexibilidade, a extensibilidade ou a portabilidade de um sistema;
- i) os **relatos** são as explicações das regras ou passos do padrão que explicam como e porque ele trata suas influências contrárias definidas nas forças, para alcançar os objetivos, princípios e filosofias propostas. Isso diz realmente como o padrão funciona, porque funciona e porque ele é bom.
- j) o **relacionamento** de padrões tanto pode ser estático e/ou dinâmico com outros padrões dentro da mesma linguagem ou sistema de padrões. Padrões relacionados geralmente compartilham as mesmas influências. São possíveis os seguintes tipos padrões relacionados:
- padrões predecessores, cuja aplicação conduza a esse padrão;
 - padrões sucessores, que devem se aplicados após esse;
 - padrões alternativos, que descrevem uma solução diferente para o mesmo problema, mas diante de influências e restrições diferentes; e
 - padrões dependentes, que podem (ou devem) ser aplicados simultaneamente com esse padrão.
- l) as **aplicações conhecidas** (*know uses*) são ocorrências já conhecidas do padrão e sua aplicação em sistemas existentes. Isso ajuda a validar o padrão, verificando se ele é realmente uma solução aprovada para um problema recorrente.

4.6 ANTIPADRÕES

Antipadrões representam uma “lição aprendida”, ao contrário dos padrões, que representam a “melhor prática” (HILLSITE, 2003). Os antipadrões podem ser de dois tipos:

- a) aqueles que descrevem uma solução ruim para um problema que resultou em uma situação ruim;
- b) aqueles que descrevem como escapar de uma situação ruim e como proceder para, a partir dela, atingir uma boa solução.

Os antipadrões são necessários porque é tão importante ver e entender soluções ruins como soluções boas. Se por um lado é útil mostrar a presença de certos padrões em sistemas mal sucedidos, por outro lado é útil mostrar a ausência dos antipadrões em sistemas bem sucedidos.

Segundo uma outra definição (CMG 98) um antipadrão é:

- a) um padrão que diz como ir de um problema para uma solução ruim ou;
- b) um padrão que diz como ir de uma solução ruim para uma solução boa.

A primeira parece inútil, exceto se considerar o padrão como uma mensagem “Não faça isso”. O segundo é definitivamente um padrão positivo, no qual o problema é a solução ruim.

4.7 CATÁLOGOS DE PADRÕES

Um catálogo de padrões é uma coleção de padrões relacionados (possivelmente relacionados fraca ou informalmente). Em geral, subdivide os padrões num pequeno número de categorias abrangentes e pode incluir algumas referências cruzadas entre os padrões.

A fig. 3 mostra o catálogo de padrões feito por Gamma (2000).

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

Figura 3: Exemplo de uma coleção de padrões.

Metsker (2002) classifica os padrões em cinco grupos distintos por sua intenção (problema a ser solucionado). Os grupos são:

- a) oferecer uma **interface**;
- b) atribuir uma **responsabilidade**;
- c) realizar a **construção** de classes ou objetos;
- d) controlar formas de **operação**;
- e) implementar uma **extensão** da aplicação.

A fig. 4 demonstra o catálogo de padrões definidos Metsker (2002).

Intenção	Padrões
1. Interfaces	Adapter, Facade, Composite, Bridge
2. Responsabilidade	Singleton, Observer, Mediator, Proxy, Chain of Responsibility, Flyweight
3. Construção	Builder, Factory Method, Abstract Factory, Prototype, Memento
4. Operações	Template Method, State, Strategy, Command, Interpreter
5. Extensões	Decorator, Iterator, Visitor

Figura 4: Classificação de padrões

Um catálogo de padrões pode oferecer um esquema de classificação e recuperação dos seus padrões, já que eles estão subdivididos em categorias, e adiciona uma certa quantidade de estrutura e organização a uma coleção de padrões, mas tipicamente não vai além de mostrar apenas as estruturas e relações mais superficiais (quando o faz).

Na fig. 3, o critério escopo decide se o padrão atua sobre uma classe ou sobre objetos da classe. Os padrões da classe lidam com as relações entre classes e as suas subclasses, por meio do uso de herança, sendo, portanto estabelecidas de forma estática (durante a compilação). Os padrões do objeto lidam com as relações entre objetos, que podem ser modificados durante a execução e apresenta maior dinamismo. A maioria dos padrões utiliza herança de alguma forma e, portanto, os únicos padrões classificados na categoria classe são os que se concentram em relações de classes. No critério proposto existem padrões de criação, padrões estruturais e padrões comportamentais.

Fazendo uma referência cruzada entre esses conceitos, chega-se às seguintes conclusões:

- a) os *patterns* com propósito de criação e escopo de classe delegam algumas partes da criação dos objetos para subclasses, enquanto os *patterns* com propósito de criação e escopo de objeto delegam algumas partes da criação dos objetos para outros objetos.
- b) os *patterns* estruturais com escopo de classe usam herança para compor classes, enquanto os *patterns* estruturais com escopo de objeto descrevem maneiras de “juntar” objetos.
- c) os *patterns* comportamentais com escopo de classe usam herança para descrever algoritmos e fluxos de controle, já os *patterns* comportamentais com escopo de objeto descrevem como um grupo de objetos coopera para realizar uma tarefa que nenhum deles seria capaz de suportar sozinho.

Os padrões de criação concentram-se no processo de criação de objetos, os estruturais lidam com a composição de classes ou objetos, e os comportamentais caracterizam as formas pelas quais as classes ou objetos interagem e distribuem responsabilidades.

4.8 PROCESSO DE DESENVOLVIMENTO COM PADRÕES

Segundo Buschmann (1996), os padrões não definem um novo método para o desenvolvimento de software que substitua os já existentes. Eles apenas complementam os

métodos de análise e projeto gerais e independentes do problema, por exemplo, Booch, OMT, Shlaer/Mellor, etc, com diretrizes para resolver problemas específicos e concretos. Buschmann (1996), sugeriu os seguintes passos para desenvolver um sistema usando padrões de software:

- a) utilizar o método preferido para o processo de desenvolvimento de software em cada fase do desenvolvimento;
- b) utilizar um sistema de padrões adequado para guiar o projeto e a implementação de soluções para problemas específicos, isto é, sempre que se encontrar um padrão que resolva um problema do projeto presente no sistema, utilizar os passos da implementação associados a esse padrão. Se esses se referirem a outros padrões, aplicam-se recursivamente;
- c) se o sistema de padrões não incluir um padrão para o problema do projeto, tenta-se encontrar um padrão noutras fontes conhecidas;
- d) se nenhum padrão estiver disponível, aplica-se às diretrizes de análise e projeto do método que se está usando. Essas diretrizes fornecem pelo menos algum apoio útil para resolver o problema do projeto em questão.

Segundo Buschmann (1996), essa abordagem simples evita que se criem outros métodos do projeto. Ela combina a experiência de desenvolvimento de software captada pelos métodos de análise e projeto com as soluções específicas para problemas de projeto descrito pelos padrões.

4.9 PROJETO DE SOFTWARE COM PADRÕES

No projeto de software, pode-se destacar os seguintes níveis de projeto demonstrado na fig. 5:

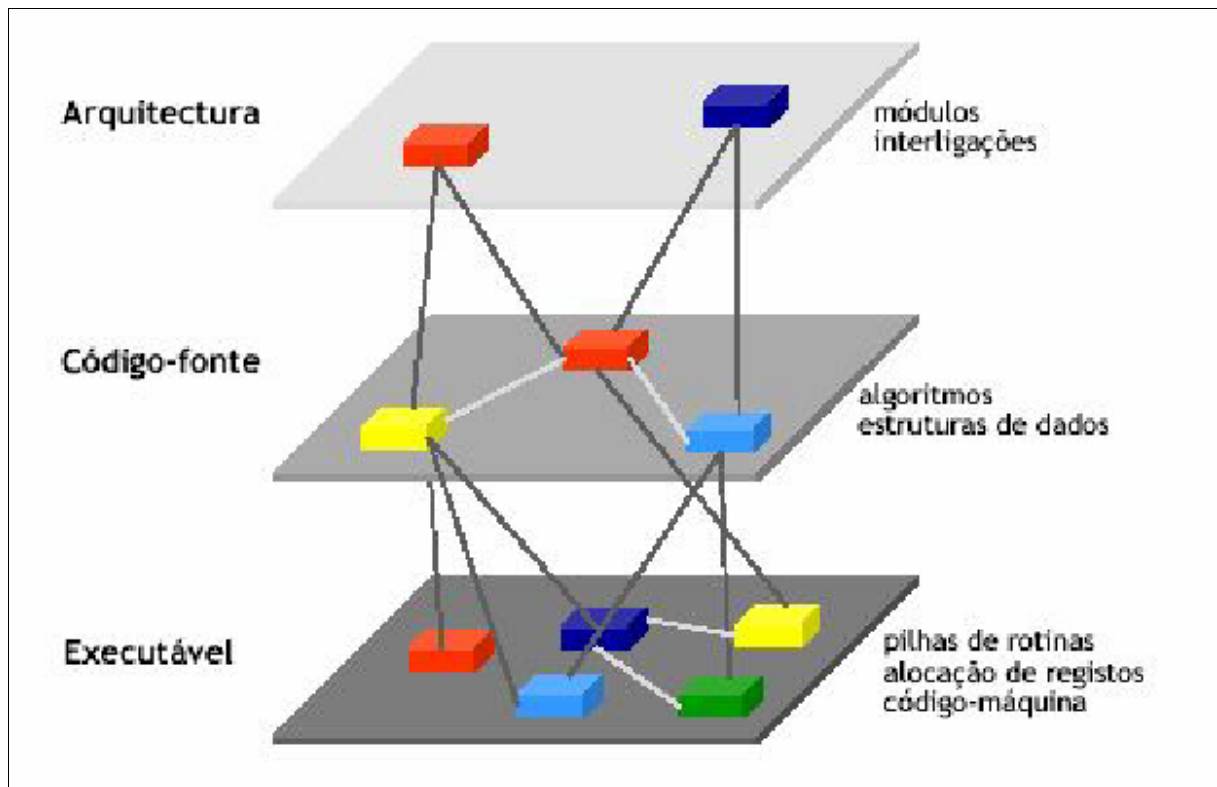


Figura 5: Níveis de projeto de software

No projeto de software com padrões, incluem-se geralmente os seguintes passos, numa análise *top-down*:

- identificar** o problema a resolver,
- selecionar** os padrões adequados;
- verificar** a aplicabilidade dos padrões ao problema em questão;
- instanciar** o padrão na situação concreta;
- avaliar** os compromissos de projeto envolvidos.

Buschmann (1996), propôs algumas regras para a derivação de novos padrões num dado projeto *Pattern Mining*, a destacar:

- encontrar pelo menos três exemplos nos quais um problema é resolvido efetivamente usando a mesma solução;
- extrair a solução, o problema e as influências;
- declarar a solução como candidata a padrão;

- e) executar um *writer's workshop* para melhorar a descrição do padrão-candidato e compartilhá-lo com outros projetistas e engenheiros de software (este passo é tipicamente de grande importância);
- f) aplicar o padrão-candidato noutra projeto de desenvolvimento de software;
- g) declarar o padrão-candidato como padrão, se a sua aplicação for bem sucedida, após a sua aplicação. Caso contrário, tentar procurar uma solução melhor.

4.10 VANTAGENS NA UTILIZAÇÃO DE PADRÕES

Os padrões de software oferecem um leque de vantagens a quem os utiliza. O aspecto principal é a sua capacidade de oferecer uma base de comunicação entre indivíduos, permitindo que estes partilhem experiências e soluções para problemas comuns. Para esse fim fornecem um vocabulário formal e especializado que serve como plataforma de bom entendimento (AGUIAR, 2003; SCHMIDT, 2003).

Reduzem a complexidade dos assuntos tratados, permitindo com que as pessoas possam dedicar mais tempo tentando tornar o seu produto original do que começarem de novo tudo. Levado ao extremo, permite, por exemplo, que uma empresa que forneça serviços de rede e, que esteja desenvolvendo um *web-browser*, não tenha que reinventar toda a Internet só para por o seu *browser* em funcionamento. Ao reduzirem a complexidade dos assuntos através de um nível superior de abstração, os padrões permitem uma maior capacidade de comunicação e de uma maneira mais simples e rápida, permitindo que sirvam para partilhar boas idéias beneficiando tanto o mais experiente na matéria como aquele que tenha iniciado a sua aprendizagem. A abstração também permite com que os padrões de software sejam independentes do ambiente em que são utilizados; por outras palavras, são independentes da plataforma, do sistema operacional, da linguagem de programação limitando-se ao problema em si e ao contexto em que se apresenta (AGUIAR, 2003; CETUS, 2003).

4.11 DESVANTAGENS NA UTILIZAÇÃO DE PADRÕES

Os padrões poderiam parecer como a solução ideal na comunicação de soluções, porém não são perfeitos. Têm os seus defeitos e limitações. Por exemplo, devido a sua capacidade de abstração relativamente às classes e aplicações, não suportam uma reutilização rápida de código fonte. Isto leva, embora seja mais fácil saber como fazer o trabalho, o mesmo não é reduzido. Outro problema é que devido a sua ligação ao contexto sobre o qual é

aplicado, caso não exista nenhum padrão para um novo contexto, leva a que seja necessário consultar um elevado número de padrões já existentes para conseguir descobrir como fazer o que se pretende. Este mesmo problema também pode resultar da sua elevada simplicidade, uma vez que explora muito superficialmente para poder ser aplicada a diversas situações parecidas.

Para finalizar, o fato de alguém ter escrito um padrão sobre um determinado assunto, não quer dizer que a solução que a outra pessoa descobriu seja a melhor. Significa sim, que a solução previamente descoberta funciona. Isto limita de certa forma o desenvolvimento de novas tecnologias, devido ao fato de estar aproveitando os conhecimentos de outras pessoas.

5 FRAMEWORKS

5.1 INTRODUÇÃO

Após a popularização da linguagem Smalltalk, no início da década de 80, paralelamente às bibliotecas de classes, começaram a ser construídos *frameworks* de aplicação, que acrescentam às bibliotecas de classes os relacionamentos e interação. Com os *frameworks*, reutilizam-se não somente as linhas de código, como também o projeto abstrato envolvendo o domínio de aplicação. Segundo Gamma (2000), um *framework* captura as decisões de projeto que são comuns ao seu domínio de aplicação. Assim, *frameworks* enfatizam reutilização de projetos em relação á reutilização de código, embora *framework*, geralmente, inclua subclasses concretas que você pode utilizar imediatamente.

Framework é definido por Coad (1988) como um esqueleto de classes, objetos e relacionamentos agrupados para construir aplicações específicas (Coad, 1988) e por Johnson como um conjunto de classes abstratas e concretas que provê uma infra-estrutura genérica de soluções para um conjunto de problemas. Essas classes podem fazer parte de uma biblioteca de classes ou podem ser específicas da aplicação. *Frameworks* possibilitam reutilizar não só componentes isolados, como também, toda a arquitetura de um domínio específico.

Diversos *frameworks* têm sido desenvolvidos nas duas últimas décadas, visando o reuso de software e conseqüentemente a melhoria da produtividade, qualidade e manutenibilidade. Neste capítulo são definidos os *frameworks de software*, bem como diversos conceitos básicos necessários para entender sua finalidade.

5.2 DEFINIÇÃO

Os *frameworks* procuram minimizar o esforço no desenvolvimento de aplicações, por conterem a definição da sua arquitetura, liberando, assim, o desenvolvedor de software dessa preocupação (LARMAN, 1999).

Um *framework* é o projeto de um conjunto de objetos que colaboram entre si para execução de um conjunto de responsabilidades. Um *framework* reusa análise, projeto e código. Ele reusa análise porque descreve os tipos de objetos importantes e como um problema maior pode ser dividido em problemas menores. Ele reusa projeto porque contém algoritmos abstratos e descreve a interface que o programador deve implementar e as

restrições a serem satisfeitas pela implementação. Ele reusa código porque torna mais fácil desenvolver uma biblioteca de componentes compatíveis e porque a implementação de novos componentes pode herdar grande parte de seu código das superclasses abstratas. Apesar de todos os tipos de reuso serem importantes, o reuso de análise e de projeto são os que mais compensam em longo prazo (Johnson, 1999).

Segundo Gamma (2000), *framework* é um conjunto de classes cooperantes que constróem um projeto reutilizável para uma categoria específica de software. Por exemplo, um *framework* pode ser orientado para a construção de editores gráficos para diferentes domínios, tais como desenho artístico, composição musical e sistemas CAD para mecânica. Um outro *framework* pode ajudar a construir compiladores para diferentes linguagens de programação e diferentes processadores.

Nota-se que nas três definições os *frameworks* são um conjunto de classes abstratas de um domínio específico modelando a arquitetura principal da aplicação. A customização de *framework* para uma aplicação específica faz-se através da criação de subclasses, sendo estas subclasses específicas das classes abstratas do *framework*.

5.3 INVERSÃO DE CONTROLE

Uma característica importante dos *frameworks* é que os métodos definidos pelo usuário para especializá-lo são chamados de dentro do próprio *framework* ao invés de serem chamados do código da aplicação do usuário. A reutilização em nível de *framework* leva a uma inversão de controle entre a aplicação e os softwares podendo ser observado na fig. 6. Quando se utiliza uma biblioteca convencional de sub-rotinas, escreve-se o corpo principal da aplicação e chama o código que quer reutilizar. Quando se utiliza *framework* reutiliza-se o corpo principal e escreve o código que este chama (GAMMA, 2000).

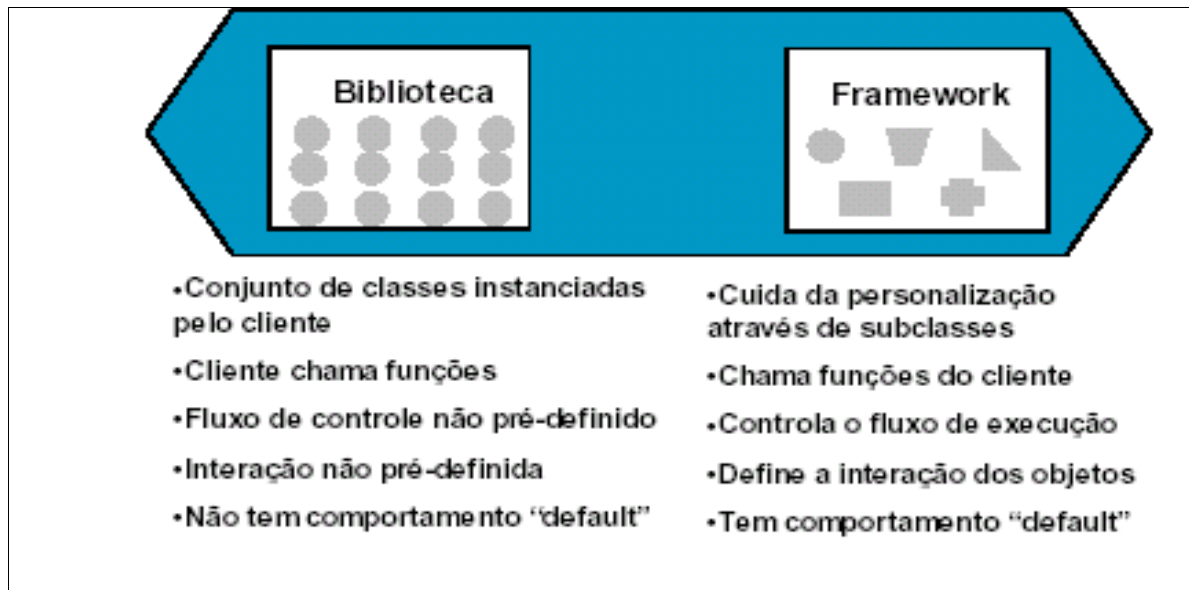


Figura 6: Inversão de controle

5.4 CLASSIFICAÇÃO DOS *FRAMEWORKS* QUANTO À SUA UTILIZAÇÃO

5.4.1 HORIZONTAL E VERTICAL

Podem classificar-se de acordo com sua aplicabilidade, em horizontais ou verticais (BUCH,1995, FAYAD, 1999).

Frameworks horizontais são voltados a implementações de aplicações relacionadas a infra-estruturas de comunicação de dados, de interface de usuários e de gerenciamento de dados.

Os *frameworks* verticais são voltados ao desenvolvimento com orientação para um domínio de aplicação específico. Procuram atender a características comuns do domínio e deixar as específicas para serem implementadas em aplicações construídas a partir deles.

Os *frameworks* de domínio de aplicação, ou verticais, são de maior interesse neste estudo. Procura-se trabalhar com um determinado domínio de uma aplicação para, a partir dela facilitar o desenvolvimento de novas aplicações.

5.4.2 FRAMEWORK DE CAIXA BRANCA X CAIXA PRETA

O comportamento específico de um *framework* de aplicação é geralmente definido adicionando-se métodos às subclasses de uma ou mais de suas classes.

Existem dois tipos de *frameworks*: **caixa branca** e **caixa preta** (Johnson, 1988). No *framework* caixa branca o usuário pode visualizar como foi implementado a partir de classes já existentes. O reuso é provido por herança, ou seja, o usuário deve criar subclasses das classes abstratas contidas no *framework* para criar aplicações específicas. Para tal, ele deve entender detalhes de como o *framework* funciona para poder usá-lo. Já no *framework* caixa preta o reuso é por composição, ou seja, o usuário combina diversas classes concretas existentes no *framework* para obter a aplicação desejada. Assim, ele deve entender apenas a interface (assinatura de métodos públicos), os serviços que são por eles disponibilizados e requisitados para poder usá-lo (PREE, 1997; BOCH, 1997; GAMMA, 2000).

No *framework* de caixa preta o usuário não é totalmente livre para criar ou alterar funcionalidades a partir da criação de classes, uma vez que o controle da aplicação é definido nas classes do *framework* e não nas classes do usuário. São elaborados para oferecer soluções prontas para determinados domínios, possibilitando, ainda, através de composição e adaptações de novas características (PREE, 1997; BOCH, 1997).

Um aspecto variável de um domínio de aplicação é chamado de “ponto de especialização” (em inglês, “*Hot spot*”) (BUSCHMAN, 1996). Diferentes aplicações dentro de um mesmo domínio são distinguidas por um ou mais *hot spots*. Eles representam as partes do *framework* de aplicação que são específicas de sistemas individuais. Os *hot-spots* são projetados para serem genéricos – podem ser adaptados às necessidades da aplicação conforme visto na fig. 7.

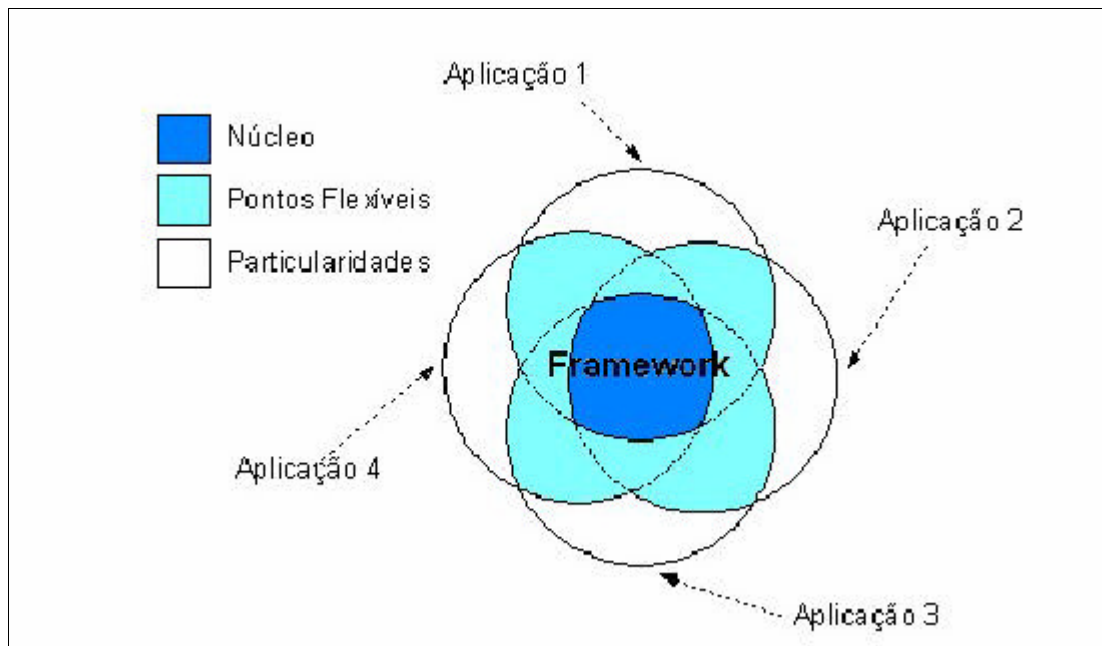


Figura 7: Percepção de um *framework* entre aplicações de um mesmo domínio

“Pontos fixos” (em inglês, “*Frozen-spots*”) definem a arquitetura geral de um sistema de software – seus componentes básicos e os relacionamentos entre eles. Os *frozen spots* permanecem fixos em todas as instanciações do *framework* de aplicação. A fig. 8 ilustra um *framework* caixa branca com um único *hot-spot* R (SCHMID, 1997). Para utilização do *framework* é necessário fornecer a implementação referente a esse *hot-spot*, que no caso da fig. 8 é R3.

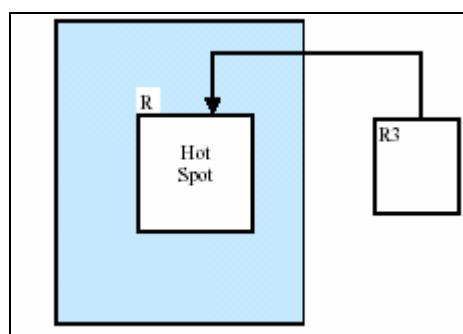


Figura 8: Ilustra um *framework* caixa-branca, também com um único hot-spot R

Na fig. 9, que ilustra um *framework* de caixa-preta, existem três alternativas (R1, R2 e R3) para implementação da responsabilidade R. O usuário deve escolher uma delas para obter sua aplicação específica. Note que R1, R2 e R3 fazem parte do *framework* e são as únicas alternativas possíveis de implementação do *hot-spot*. Já no caso da fig. 9, R3 não fazia parte

do *framework*, mas, em compensação, qualquer alternativa de implementação do *hot-spot* seria possível.

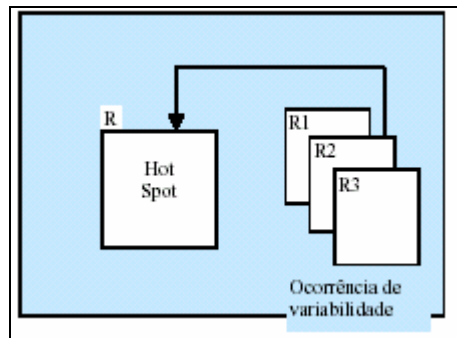


Figura 9: *Framework* de caixa-preta

Resumindo, um *framework* caixa branca é mais fácil de projetar, pois não há necessidade de prever todas as alternativas de implementação possíveis. Já o *framework* caixa preta é mais difícil de projetar por haver a necessidade de fazer essa previsão. Por outro lado, o *framework* caixa preta é mais fácil de usar, pois basta escolher a implementação desejada, enquanto no caixa branca é necessário fazer essas implementações completas.

Frameworks caixa branca podem evoluir para se tornar cada vez mais caixa preta (Johnson, 1997). Isso pode ser conseguido de forma gradativa, implementando-se várias alternativas que depois são aproveitadas na instânciação do *framework*. Ao mesmo tempo não se fecha totalmente o *framework*, permitindo ao usuário continuar usando-o como caixa branca. Após um certo tempo, estarão disponíveis diversas alternativas e então se pode decidir tornar o *framework* caixa preta de fato. A medida em que o *framework* vai se tornando mais caixa preta, diminui o número de objetos criados, embora aumente a complexidade das suas interconexões. Além disso, o “*scripting*” fica mais importante, por permitir a especificação das classes que farão parte da aplicação e sua interconexão. Pode-se também, construir ferramentas de apoio que ajudem na especificação da aplicação. Essas ferramentas têm como objetivo facilitar a instânciação do *framework*, por meio do uso de componentes visuais que sejam mais fáceis de usar do que os comandos do “*script*”.

5.5 DESENVOLVIMENTO DE APLICAÇÕES A PARTIR DE FRAMEWORKS

A finalidade básica de um *frameworks* é poder ser reutilizado na produção de diferentes aplicações, minimizando tempo e esforço requeridos para isto. Desta forma,

procura ser uma descrição aproximada do domínio, construída a partir das informações até então disponíveis.

Uma das vantagens do desenvolvimento de aplicações a partir de *frameworks* é a possibilidade de reutilização, desde a etapa de elaboração da arquitetura da aplicação, permitindo, ainda, ao usuário adaptar as características de sua aplicação.

O desenvolvimento de aplicações inicia com o entendimento do sistema contido no projeto do *framework* e segue, no detalhamento das particularidades da aplicação específica, o que é definido por seu usuário. Assim, a implementação de uma aplicação a partir do *framework* é feita pela adaptação de sua estrutura de classes, fazendo com que esta inclua as particularidades da aplicação.

A utilização ideal de desenvolvimento de aplicações a partir de *frameworks* consiste em completar ou alterar procedimentos e estruturas de dados neles presentes. Sob esta ótica, uma aplicação gerada por um *framework* não deveria incluir classes que não fossem subclasses de classes do *framework*. Todavia, como um *framework* nunca é uma descrição completa de um domínio, é possível que a construção de aplicações por um *framework* leve à obtenção de novos conhecimentos do domínio tratado, indisponíveis durante a sua construção.

Cabe observar que, além do desenvolvimento de aplicações a partir de *frameworks*, estes também podem servir de exemplos para o desenvolvimento de outros *frameworks* ou fazer parte do desenvolvimento de novos (LARMAN, 1999).

Além destas características dos *frameworks*, de flexibilidade e generalização, outro fator, que influencia na complexidade de desenvolvimento de novas aplicações a partir dos *frameworks*, é o fato de que eles podem agrupar diferentes quantidades de classes. Isto os torna mais ou menos complexos. *Frameworks* tipicamente possuem centenas de classes que devem estar semanticamente relacionadas e interconectadas (BRAGA, 2000).

5.6 FRAMELETS

Os *frameworks* procuram resolver e fazer compreender problemas muitas vezes complexos. Apesar de sua descrição detalhada e clara, podem apresentar um grau de dificuldade que acabam levando muitos desenvolvedores a não utilizá-los.

Além da dificuldade de compreender um *framework*, sua tendência de possuir uma grande quantidade de classes torna-o complexo e dificulta o desenvolvimento de seu projeto, de seu reuso no desenvolvimento de aplicações e implementações do seu comportamento (classes e métodos abstratos) (PREE, 2000).

Buscando manter as vantagens dos *frameworks* e viabilizar uma solução quanto à dificuldade de sua utilização, os *framelets* procuram disponibilizar uma estrutura de classes inter-relacionadas, semelhante à dos *frameworks*, mas com uma quantidade menor de classes.

Framelets são pequenas arquiteturas direcionadas a um problema específico de um domínio. São elaborados como unidades independentes, em blocos que podem ser facilmente compreendidos, modificados e combinados (PREE, 1999; PREE, 2000). Um *framelet* também pode prover a decomposição de *frameworks* maiores, de domínio do problema. O *framelet* procura manter as características do *framework*, adicionando maior flexibilidade e facilidade de reuso, além de facilitar o processo de compreensão e possibilitar o desenvolvimento de *frameworks* maiores.

5.6.1 CARACTERÍSTICAS

A principal característica do *framelet* é de ser um *framework* pequeno e independente, que pode ser acoplado a outros *framelets* para o desenvolvimento de uma aplicação (PREE, 1998, PREE, 2000). O *framelet* possui algumas características diferenciadas do *framework* (PREE, 1998; PREE, 2000):

- a) não assume o controle principal de uma aplicação;
- b) é pequeno em tamanho, normalmente possui até doze classes apenas;
- c) possui uma interface definida de forma simples e clara;
- d) pode ser auto-explicativo em um domínio de uma aplicação; e,
- e) usável isoladamente, independentemente de outros *framelets*.

Outras características do *framelet* ainda podem ser identificadas (TALIGENT, 2003):

- a) proporciona implementações concretas que podem ser usadas diretamente;
- b) possui pequena quantidade de classes e funções que necessitam ser derivadas ou herdadas;
- c) consolida e implementa funcionalidades similares em uma abstração comum;
- d) divide grandes abstrações em diversas menores, onde cada pequena abstração deve possuir um conjunto de responsabilidades pequeno e focado;

- e) possibilita implementar cada variação da abstração em um objeto; e,
- f) utiliza composição sempre que possível, reduzindo, assim, a quantidade de classes e a complexidade para o *framework* cliente.

Simplificando, os *framelets* procuram solucionar o problema de complexidade e dificuldade de utilização dos *frameworks* no desenvolvimento de aplicações, sem, no entanto, perder a característica da flexibilidade. Com a abordagem de *framelets* podem-se identificar, ainda, algumas vantagens, como a facilidade de composição para o desenvolvimento de aplicações, devido à sua característica de independência.

5.6.2 DESENVOLVIMENTO DE FRAMELETS

Quanto ao desenvolvimento, procuram eles estar com sua arquitetura construída de tal forma que possam disponibilizar seus serviços para serem utilizados por outro *framelet* ou diretamente por uma aplicação. Deve-se, ainda, analisar sua reusabilidade em um domínio de problema (BRAGA, 2000; PREE, 2000).

Identificar e dividir partes de um domínio de problema em um *framelet* é uma etapa essencial de seu projeto, para tanto se pode:

- a) identificar as abstrações chave de uma aplicação e mapeá-las em objetos num projeto convencional de uma aplicação;
- b) identificar agrupamentos de requerimentos que podem ocorrer em diversas aplicações;
- c) mapear pequenos problemas, em uma pequena arquitetura;
- d) identificar uma forma particular de uma variância de comportamentos.

Um *framelet* com estas características de desenvolvimento pode ser inspirado em um padrão de análise (COAD, 1997). Um exemplo de um *framelet* assim concebido é apresentado na fig. 10.

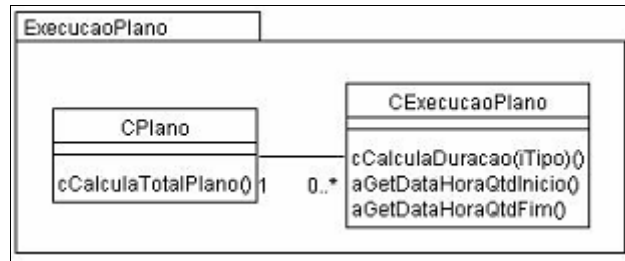


Figura 10: *Framelet* inspirado em um padrão de análise

O *framelet* *ExecucaaoPlano*, ilustrado na fig. 10, trata de um problema no qual um plano é formado por fases, etapas. Este *framelet* é composto pelas classes *CPlano* e *CExecucaaoPlano*. Por exemplo, a classe *CPlano* pode representar o plano de execução de uma obra, a linha de produção de uma fábrica ou o planejamento de atividades para algum objetivo. A classe *CExecucaaoPlano* representa as fases da execução de uma obra, as fases da linha de produção ou as atividades de um planejamento. O método *CPlano.cCalculaTotalPlano()* calcula o custo total de um plano. O método *CExecucaaoPlano.cCalculaDuracao()* calcula a duração de cada fase do plano. Os métodos abstratos *CExecucaaoPlano.aGetDataHoraInicio()* e *CExecucaaoPlano.aGetDataHoraFim()* são métodos tais que, uma vez implementados, devem retornar um valor referente ao início e ao fim de uma determinada fase.

O desenvolvimento do *framelet* requer ainda, que se determinem suas interfaces para que possa, através delas exportar seus serviços, e determinar as partes abstratas que devem ser implementadas pela aplicação que será desenvolvida a partir dele (PREE, 2000).

5.7 DESENVOLVIMENTO DE APLICAÇÕES

O desenvolvimento de uma aplicação a partir de *framelets* é realizado através de combinações de *framelets* entre si. Esta sistemática visa simplificar desenvolvimento e extensão de software, facilitando também a interação de um domínio de problema com outros softwares (PREE, 2000).

Algumas vantagens no desenvolvimento de aplicações a partir de *framelets* podem ser observadas (PREE, 2000):

- a) proporcionam um meio de gerenciamento da complexidade quantitativa de classes de um domínio de problema, dividindo esse problema em partes menores e flexíveis que podem ser combinadas;
- b) permitem, com maior facilidade, a extensão de um domínio de problema através da possibilidade de adicionar mais *framelets* a este; e,
- c) permitem facilmente a composição de vários *framelets* em um domínio de problema sem, no entanto, assumir o controle da execução de uma aplicação.

Conforme pode ser visto na fig. 11, os *framelets* disponibilizam três tipos de construções (PREE, 2000):

- a) componentes unidades binárias pré-definidas e configuráveis que podem ser usadas. São objetos concretos e prontos para serem utilizados;
- b) interfaces, um conjunto de operações que disponibilizam e requisitam serviços. Idem aos *frameworks*; e,
- c) padrões de projeto, soluções de arquiteturas de um domínio específico de um problema. Soluções abstratas de arquiteturas para problemas de projetos que devem ser implementados por outros componentes.

Os *framelets* permitem sua interação com outros *framelets* ou aplicações. Componentes e interfaces são exportados “horizontalmente” para outros *framelets* e “verticalmente” para o desenvolvimento de aplicações como pode ser visto na fig. 12. Padrões de projeto são apenas exportados “horizontalmente” entre os *framelets*. Assume-se que não existem no nível da aplicação (PREE, 2000).

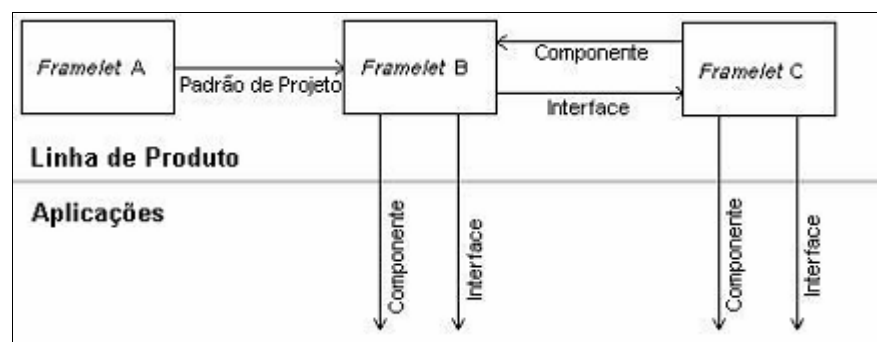


Figura 11: Desenvolvimento de aplicações a partir de *framelets*

É através dos tipos de construções que os *framelets* possibilitam e disponibilizam um desenvolvimento rápido de aplicações (PREE, 2000).

6 IMPOSTOS

A implementação de *frameworks* para cálculos de impostos, devido a complexidade existe nas leis tributárias vigentes, constitui-se de uma importante área de aplicação de *patterns* para facilitar a modelagem, o entendimento e manutenção da implementação de um *framework*.

Os imposto são tributos que toda a pessoa física, jurídica e outros determinados por lei devem efetuar o pagamento dos tributos incidentes sobre as operações mercantis realizadas (BORGES, 2000; CARRAZZA, 2002).

Dentro da carga tributária brasileira destaca-se dois impostos que estão presentes nas operações mercantis de maneira mais visível. O Imposto sobre Circulação de Mercadorias e Serviços (doravante denominado ICMS) que consistem em um tributo cobrado sobre operações mercantis que ocorre a transferência dos direitos de posse da mercadoria. O Imposto sobre Produto Industrializado (doravante denominado IPI) consistem na tributação da mercadoria quando da transferência dos direitos de posse da mercadoria no instante da saída do produto da industria ou quando da importação de mercadorias industrializadas (CARRAZZA, 2002).

As informações sobre tributação fiscal e cálculos de imposto fiscal neste trabalho não são conclusivas deixando claro ao leitor que somente constitui-se da explanação de informações necessárias para o desenvolvimento do mesmo. Para maiores informações deve-se consultar a legislação vigente nacional e as leis complementares.

6.1 DESCRIÇÃO NORMATIVA DOS IMPOSTOS

Dois fatores determinam a relevância dos impostos no atual contexto de negócios. O primeiro é a convicção de que tais impostos representam o maior ônus fiscal das empresas. O segundo é a consciência empresarial do significativo grau de sofisticação e complexidade da legislação presente segundo Borges (2000).

A efetiva importância desses impostos no mundo dos negócios vem exigindo das organizações um grande investimento de energia e recursos visando obter idéias e planos que possibilitem uma perfeita conciliação dos aspectos industriais, comerciais e fiscais voltados à anulação, redução ou adiantamento do ônus tributário segundo Borges (2000).

Essas idéias e planos fundamentam-se na adoção de ações que resultam em conseqüências fiscais menos onerosas.

6.2 ESTRUTURA DO IMPOSTO

A identificação dos elementos estruturais de qualquer imposto do nosso elenco tributário não estão definidos de forma consolidada ou agregada. Os critérios e aspectos se encontram mencionados de forma bastante dispersa em vários contextos legislativos inseridos nos diversos graus da ordem jurídica. Os critérios e aspectos integrantes da gama de impostos do sistema tributário nacional poderão verificar a existência de imposto em que o material objeto de sua incidência encontra-se prevista no plano constitucional, a área geográfica que lhe é relevante, definida pela atividade legiferantes ordinária e o instante na qual se consolida a concretização da sua hipótese de incidência, mencionada pela legislação complementar (CARRAZZA, 2002):

- a) hipótese: consiste na descrição legal, por conseqüência abstrata e genérica de um ato ou estado de efeito;
- b) critério material: identifica-se a ação, conduta ou o comportamento de uma pessoa, física ou jurídica, sujeita à tributação, por exemplo: industrializar produtos e, em seguida, negociá-los; comercializar mercadorias;
- c) critério espacial: investiga-se o local escolhido para materialização do comportamento, ação ou conduta relevante ao imposto;
- d) critério temporal: obtém-se o conjunto de dados que possibilita o conhecimento da ocasião ou instante em que se considera concretizado o fato gerador de tributação;
- e) base de cálculo: constitui na expressão econômica do fator gerador. É sobre a base de cálculo que o tributo incide. É a matéria necessária para o cálculo do tributo devido. Sobre essa base incide um percentual (alíquota). O resultando de tal incidência é o atributo devido;
- d) alíquota: fator aplicável à base de calculo para se conseguir o montante que o sujeito passivo deve ao sujeito ativo.

6.3 ESTRUTURA NORMATIVA DO IPI

O imposto sobre produtos industrializados (IPI) incide sobre produtos industrializados, nacionais e estrangeiros. Suas disposições estão regulamentadas pelo Decreto 2637 de 1998 (doravante denominado de RIPI/98).

O campo de incidência do imposto abrange todos os produtos com alíquota, ainda que zero, relacionados na Tabela de Incidência do IPI (doravante denominado de TIPI), observadas as disposições contidas nas respectivas notas complementares, excluídos aqueles a que corresponde a notação "NT" (não-tributado) (BORGES, 2000, CARRAZZA, 2002).

6.3.1 HIPÓTESES NORMATIVAS DE INCIDÊNCIA

Produto industrializado é o resultante de qualquer operação definida no RIPI/98 como industrialização, mesmo incompleta, parcial ou intermediária (Carrazza, 2002).

Caracteriza industrialização qualquer operação que modifique a natureza, o funcionamento, o acabamento, a apresentação ou a finalidade do produto, ou o aperfeiçoamento para consumo, tal como (BORGES, 2000, CARRAZZA, 2002):

- a) a que, exercida sobre matéria-prima ou produto intermediário, importe na obtenção de espécie nova (transformação);
- b) a que importe em modificar, aperfeiçoar ou, de qualquer forma, alterar o funcionamento, a utilização, o acabamento ou a aparência do produto (beneficiamento);
- c) a que consista na reunião de produtos, peças ou partes e de que resulte um novo produto ou unidade autônoma, ainda que sob a mesma classificação fiscal (montagem);
- d) a que importe em alterar a apresentação do produto, pela colocação da embalagem, ainda que em substituição da original, salvo quando a embalagem colocada se destine apenas ao transporte da mercadoria (acondicionamento ou acondicionamento);
- e) a que, exercida sobre produto usado ou parte remanescente de produto deteriorado ou inutilizado, renove ou restaure o produto para utilização (renovação ou acondicionamento).

Não se considera industrialização (BORGES, 2000, CARRAZZA, 2002):

- a) o preparo de produtos alimentares, não acondicionados em embalagem de apresentação;
- b) na residência do preparador ou em restaurantes, bares, sorveterias, confeitarias, padarias, quitandas e semelhantes, desde que os produtos se destinem a venda direta ao consumidor;

- c) em cozinhas industriais, quando destinados à venda direta a corporações, empresas e outras entidades, para consumo de seus funcionários, empregados ou dirigentes;
- d) o preparo de refrigerantes, à base de extrato concentrado, por meio de máquinas, automáticas ou não, em restaurantes, bares e estabelecimentos similares, para venda direta a consumidor;
- e) a confecção ou preparo de produto de artesanato;
- f) confecção de vestuário, por encomenda direta do consumidor ou usuário, em oficina ou na residência do confeccionador;
- g) o preparo de produto, por encomenda direta do consumidor ou usuário, na residência do preparador ou em oficina, desde que, em qualquer caso, seja preponderante o trabalho profissional;
- h) a manipulação em farmácia, para venda direta a consumidor, de medicamentos oficiais e magistrais, mediante receita médica;
- i) a moagem de café torrado, realizada por comerciante varejista como atividade acessória;
- j) a operação efetuada fora do estabelecimento industrial, consistente na reunião de produtos, peças ou partes e de que resulte:
 - edificação (casas, edifícios, pontes, hangares, galpões e semelhantes, e suas coberturas);
 - instalação de oleodutos, usinas hidrelétricas, torres de refrigeração, estações e centrais telefônicas ou outros sistemas de telecomunicação e telefonia, estações, usinas e redes de distribuição de energia elétrica e semelhantes;
 - fixação de unidades ou complexos industriais ao solo;
 - a montagem de óculos, mediante receita médica;
 - o conserto, a restauração e o acondicionamento de produtos usados, nos casos em que se destinem ao uso da própria empresa executora ou quando essas operações sejam executadas por encomenda de terceiros não estabelecidos com o comércio de tais produtos, bem assim o preparo, pelo consertador, restaurador ou acondicionador, de partes ou peças empregadas exclusiva e especificamente naquelas operações;
 - o reparo de produtos com defeito de fabricação, inclusive mediante substituição de partes e peças, quando a operação for executada gratuitamente, ainda que por concessionários ou representantes, em virtude de garantia dada pelo fabricante;

- a restauração de sacos usados, executada por processo rudimentar, ainda que com emprego de máquinas de costura;
- a mistura de tintas entre si, ou com concentrados de pigmentos, sob encomenda do consumidor ou usuário, realizada em estabelecimento varejista, efetuada por máquina automática ou manual, desde que fabricante e varejista não sejam empresas interdependentes, controladora, controlada ou coligada.

6.3.2 CONSEQÜÊNCIAS NORMATIVAS

Estabelecimento de um elo jurídico-tributário em que cuja extremidade ativa situa-se a União, e na extremidade passiva, o industrial ou importador (BORGES, 1999).

O IPI representa um valor obtido através da aplicação do percentual informado pelo governo sobre o preço da operação realizada (BORGES, 1999).

6.4 ESTRUTURA NORMATIVA DO IMPOSTO SOBRE CIRCULAÇÃO DE MERCADORIAS E SERVIÇOS (ICMS)

A composição do ICMS esta integrada por várias hipóteses normativas de incidência. Para análise dessa questão deve-se investigar a hipótese. Realização de atos ou negócios jurídico-mercantil de transferência do direito de posse ou propriedade de mercadorias. Aplicam-se nos limites do Estado ou do Distrito Federal no momento da saída ou entrada do referido bem (CARRAZZA, 2002, FABRETTI, 1999).

6.4.1 INCIDÊNCIAS

O imposto incide sobre (CARRAZZA, 2002):

- a) operações relativas à circulação de mercadorias, inclusive o fornecimento de alimentação e bebidas em bares, restaurantes e estabelecimentos similares;
- b) prestações de serviços de transporte interestadual e intermunicipal, por qualquer via, de pessoas, bens, mercadorias ou valores;
- c) prestações onerosas de serviços de comunicação, por qualquer meio, inclusive a geração, a emissão, a recepção, a transmissão, a retransmissão, a repetição e a ampliação de comunicação de qualquer natureza;
- d) fornecimento de mercadorias com prestação de serviços não compreendidos na competência tributária dos Municípios;

- e) fornecimento de mercadorias com prestação de serviços sujeitos ao imposto sobre serviços, de competência dos Municípios, quando a lei complementar aplicável expressamente o sujeitar à incidência do imposto estadual;
- f) sobre a entrada de mercadoria importada do exterior, por pessoa física ou jurídica, ainda quando se tratar de bem destinado a consumo ou ativo permanente do estabelecimento;
- g) sobre o serviço prestado no exterior ou cuja prestação se tenha iniciado no exterior;
- h) sobre a entrada, no território do Estado destinatário, de petróleo, inclusive lubrificantes e combustíveis líquidos e gasosos dele derivados, e de energia elétrica, quando não destinados à comercialização ou à industrialização, decorrentes de operações interestaduais, cabendo o imposto ao Estado onde estiver localizado o adquirente.

6.4.2 NÃO INCIDÊNCIAS

O imposto não incide sobre (CARRAZZA, 2002):

- a) operações com livros, jornais, periódicos e o papel destinado a sua impressão;
- b) operações e prestações que destinem ao exterior mercadorias, inclusive produtos primários e produtos industrializados semi-elaborados, ou serviços;
- c) operações interestaduais relativas à energia elétrica e petróleo, inclusive lubrificantes e combustíveis líquidos e gasosos dele derivados, quando destinados à industrialização ou à comercialização;
- d) operações com ouro, quando definido em lei como ativo financeiro ou instrumento cambial;
- e) operações relativas a mercadorias que tenham sido ou que se destinem a ser utilizadas na prestação, pelo próprio autor da saída, de serviço de qualquer natureza definido em lei complementar como sujeito ao imposto sobre serviços, de competência dos Municípios, ressalvadas as hipóteses previstas na mesma lei complementar;
- f) operações de qualquer natureza de que decorra a transferência de propriedade de estabelecimento industrial, comercial ou de outra espécie;
- g) operações decorrentes de alienação fiduciária em garantia, inclusive a operação efetuada pelo credor em decorrência do inadimplemento do devedor;

- h) operações de arrendamento mercantil, não compreendida a venda do bem arrendado ao arrendatário;
- i) operações de qualquer natureza de que decorra a transferência de bens móveis salvados de sinistro para companhias seguradoras.

Equipara-se às operações de que trata a saída de mercadoria realizada com o fim específico de exportação para o exterior, destinada a:

- a) empresa comercial exportadora, inclusive *tradings* ou outro estabelecimento da mesma empresa;
- b) armazém alfandegado ou entreposto aduaneiro.

6.4.3 LOCAL DA OPERAÇÃO OU DA PRESTAÇÃO, OCORRÊNCIA DO FATO GERADOR

Considera-se ocorrido o fato gerador do imposto no momento (BORGES, 1999, BORGES, 2000, CARRAZZA, 2002):

- a) da saída de mercadoria de estabelecimento de contribuinte, ainda que para outro estabelecimento do mesmo titular;
- b) do fornecimento de alimentação, bebida e outras mercadorias por qualquer estabelecimento;
- c) da transmissão a terceiro de mercadoria depositada em armazém geral ou em depósito fechado, no Estado do transmitente;
- d) da transmissão de propriedade de mercadoria, ou de título que a represente, quando a mercadoria não tiver transitado pelo estabelecimento transmitente;
- d) do início da prestação de serviços de transporte interestadual e intermunicipal, de qualquer natureza;
- e) do ato final do transporte iniciado no exterior;
- f) das prestações onerosas de serviços de comunicação, feita por qualquer meio, inclusive a geração, a emissão, a recepção, a transmissão, a retransmissão, a repetição e a ampliação de comunicação de qualquer natureza;
- g) do fornecimento de mercadoria com prestação de serviços;
- h) não compreendidos na competência tributária dos Municípios;
- i) compreendidos na competência tributária dos Municípios e com indicação expressa de incidência do imposto da competência estadual, como definido na lei complementar aplicável;

- j) do desembaraço aduaneiro das mercadorias importadas do exterior;
- l) do recebimento, pelo destinatário, de serviço prestado no exterior;
- m) da aquisição em licitação pública de mercadorias importadas do exterior apreendidas ou abandonadas;
- n) da entrada no território do Estado de lubrificantes e combustíveis líquidos e gasosos derivados de petróleo e energia elétrica oriundos de outro Estado, quando não destinados à comercialização ou à industrialização;
- o) da utilização, por contribuinte, de serviço cuja prestação se tenha iniciado em outro Estado e não esteja vinculada a operação ou prestação subsequente.

6.4.4 CONTRIBUINTE

Contribuinte é qualquer pessoa, física ou jurídica, que realize, com habitualidade ou em volume que caracterize intuito comercial, operações de circulação de mercadoria ou prestações de serviços de transporte interestadual e intermunicipal e de comunicação, ainda que as operações e as prestações se iniciem no exterior (BORGES, 1999, FABRETTI, 1999).

É também contribuinte a pessoa física ou jurídica que, mesmo sem habitualidade (BORGES, 2000, CARRAZZA, 2002):

- a) importe mercadorias do exterior, ainda que as destine a consumo ou ao ativo permanente do estabelecimento;
- b) seja destinatária de serviço prestado no exterior ou cuja prestação se tenha iniciado no exterior,
- c) adquira em licitação de mercadorias apreendidas ou abandonadas;
- d) adquira lubrificantes e combustíveis líquidos e gasosos derivados de petróleo e energia elétrica oriunda de outro Estado, quando não destinados à comercialização ou à industrialização.

6.4.5 CONSEQÜÊNCIA NORMATIVA

Em decorrência de um ato ou negócio cria-se uma relação jurídico-tributária em cuja extremidade ativa encontra-se o Estado ou Distrito Federal e, na extremidade passiva, o comerciante, industrial produtor ou uma das pessoas que lhes são equiparadas (BORGES, 2000).

Representarão uma quantia correspondente à aplicação de uma alíquota definida pelo governo sobre o valor da operação realizada em harmonia com as referidas condições. Um exemplo de cálculo do ICMS pode ser observado com o seguinte exemplo: Um comerciante vende canetas com preço de venda sem a dedução de ICMS é de R\$ 1,50 sendo definida a alíquota de 17% de ICMS conforme legislação em vigor. Logo o valor de venda da caneta é obtido aplicando-se a alíquota e o valor resultante de R\$ 0,3825 é somando ao preço de venda da caneta modificando-se para R\$ 1,8825. O cálculo executado:

Preço da caneta sem aplicação de ICMS.....	R\$ 1,50
17% de ICMS sobre o valor da caneta resulta em.....	R\$ 0,3825
Preço de venda da caneta com ICMS.....	R\$ 1,8825

6.4.6 SUBSTITUIÇÃO TRIBUTÁRIA

A substituição tributária é atribuída a determinados contribuinte de ICMS por força de lei, de reter o recolhimento do tributo devido por outrem (BORGES, 2000).

A substituição e diferimento são termos análogos. A substituição diz respeito a operações que, se presumem, ainda irão se realizar. O diferimento ou suspensão refere-se a operações anteriores segundo Borges (2000).

A substituição o legislador determina que se antecipe uma incidência tomando-se por ocorrido o que iria ocorrer, isto é, cobrando o ICMS antes da eclosão do fator gerador segundo Borges (2000).

Quando se efetua uma venda com esse produto, algumas alterações poderão acontecer no valor líquido da nota fiscal. O valor informado para o percentual da margem de lucro é fornecido pelo governo para cada linha de produto de acordo com as respectivas margens de lucro. A tabela 5 mostras as obrigações de recolhimento de ICMS pela industria, atacadistas e/ou varejista.

Tabela 5: Obrigações de recolhimento de ICMS

Industria deve	Atacadista – Varejista
- pagar ICMS pelas próprias operações que promove;	Dispensados do pagamento de ICMS ao fisco pelas operações que realizarem com

- reter e recolher o ICMS que será devido nas operações subseqüentes, a serem realizadas pelos atacadistas e varejistas adquirentes.	mercadorias recebidas com ICMS retido.
--	--

Fórmula de cálculo da substituição:

100,00	Valor do produto
* 1,20	20% referente à margem de lucro sendo utilizada para este produto
<hr/>	
120,00	Valor final da Base de cálculo para substituição
* 0,17	17% de ICMS do produto
<hr/>	
20,40	Valor do ICMS substituído
100,00	Valor do produto
* 0,17	17% de ICMS do produto
<hr/>	
17,00	Valor do ICMS normal
20,40	Valor do ICMS substituído
- 17,00	Valor do ICMS normal
<hr/>	
3,40	Valor final do ICMS Substituído.

A tabela 6 representa as situações na qual uma nota fiscal gera valores de tributação fiscal dependendo de localização geográfica do cliente e do tipo de cliente.

Tabela 6: Situações em que a emissão da nota fiscal gera valores de tributação fiscal

Norma interno	se Aliquota Interna	Onde Calcula	Pessoa Física fora do estado	Pessoa Física no estado	Pessoa Jurídica no estado	Pessoa Jurídica fora do estado
N	N	Fora	-	-	-	X
N	N	Dentro	-	-	X	-
N	N	Sempre	-	-	X	X

S	N	Fora	-	-	-	X
S	N	Dentro	-	-	-	-
S	N	Sempre	-	-	X	X
N	S	Fora	-	-	-	X
N	S	Dentro	-	-	X	-
N	S	Sempre	-	-	X	X
S	S	Fora	-	-	-	X
S	S	Dentro	-	-	-	-
S	S	Sempre	-	-	-	X

Fonte: TOTALL.COM (2003)

6.4.7 RETENÇÃO MICRO-EMPRESA (M.E.)

Quando for efetuada uma venda para um cliente M.E. (clientes que pagam ICMS por uma quota determinada pelo governo), calcula-se um valor de retenção. As alíquotas informadas para este produto, correspondem ao percentual que devera ser retido sobre o ICMS devido. Somente para vendas dentro do estado.

Fórmula: depois de apurados o valor do ICMS produto a produto, aplica-se o percentual de retenção informado para o produto.

Exemplo:

100,00	Valor base do produto para calculo do ICMS
* 0,17	Alíquota de ICMS do produto
<hr/>	
17,00	Valor apurado de ICMS

17,00	Valor apurado de ICMS
* 0,20	% Redução M.E.
<hr/>	
3,40	Valor da retenção. Este valor será somado ao valor líquido da NF.

6.4.8 BASE REDUZIDA NO ESTADO

Alguns produtos, especificados pelo governo, possuem uma BASE REDUZIDA no valor, que dever ser aplicada sobre a base de ICMS. Geralmente é uma espécie de subsídio a empresa que o produz por se tratar, por exemplo, de produtos VERDES, não nocivos à natureza. Isto varia muito de estado para estado. Somente para vendas dentro do estado.

Fórmula: depois de apurados o valor do ICMS produto a produto, aplica-se o percentual de redução cadastrado.

Exemplo:

100,00	Valor base do produto para cálculo do ICMS
* 0,20	% Base reduz. no estado
<hr/>	
80,00	Valor REDUZIDO para base de cálculo de ICMS
80,00	Valor base de cálculo do ICMS REDUZIDO
* 0,17	Alíquota de ICMS do produto
<hr/>	
13,60	Valor final apurado de ICMS

6.4.9 REDUÇÃO EM ESTADOS 7% E 12%

Quando for efetuada uma venda para um cliente de um estado cuja alíquota de ICMS do estado seja 12% ou 7%, existe a possibilidade da redução da base de cálculo do ICMS.

Fórmula: depois de apurados o valor do ICMS produto a produto, aplica-se o percentual de redução cadastrado, de acordo com o estado pra onde se está vendendo.

Exemplo:

100,00	Valor base do produto para cálculo do ICMS
* 0,20	% cadastrado em “Redução estados 7% e 12%”
<hr/>	
80,00	Valor REDUZIDO p/ base de cálculo de ICMS

80,00	Valor REDUZIDO p/ base de cálculo de ICMS
* 0,12	Alíquota de ICMS do produto (Deve ser 12 ou 7%)
<hr/>	
9,60	Valor final apurado de ICMS

6.4.10 CLIENTE ESPECIAL

Quando for efetuada uma venda para um cliente especial definido e cadastrados pelo governo classificado de acordo com determinadas circunstâncias, como por exemplo um deficiente físico, o governo define uma alíquota redutora da base de cálculo do ICMS. Para maiores informações sobre a classificação de cliente especial e alíquotas deve-se consultar a legislação vigente.

Fórmula: depois de apurados o valor do ICMS produto a produto, aplica-se o percentual de redução cadastrado, de acordo com o estado pra onde se está vendendo.

Exemplo:

100,00	Valor base do produto para cálculo do ICMS
* 0,20	% cadastrado em “Cliente Especial”
<hr/>	
80,00	Valor para base de cálculo de ICMS
80,00	Valor para base de cálculo de ICMS
* 0,12	Alíquota de ICMS do produto
<hr/>	
9,60	Valor final apurado de ICMS

7 DESENVOLVIMENTO DO *FRAMEWORK*

O desenvolvimento de um *framework* requer uma modelagem que abranja de forma concisa e precisa o domínio da utilização de *framework*. Uma modelagem abrangente geralmente é constituída de várias classes utilizadas para realização de tarefas específicas do domínio da aplicação. Com a iteração de várias classes em uma modelagem inicia-se a aplicação de padrões de projetos.

Os padrões de projeto utilizados no decorrer deste capítulo são extraídos do livro do Gamma (2000) denominado popularmente de GoF (*gang of four*, “ganguê dos quatro” devido aos quatro autores do livro que marcou o início de estudos mais aprofundados da aplicação de padrões na computação). Este livro tem um catálogo formado por vinte e três padrões formando a base de inspiração de desenvolvimento de novos padrões. O *framework* de cálculo de imposto utiliza-se de quatro padrões, descritos nos tópicos a seguir.

7.1 PATTERNS UTILIZADOS NO DESENVOLVIMENTO DO *FRAMEWORK*

7.1.1 SINGLETON

O *pattern Singleton* deve garantir que uma classe tenha somente uma instância e fornecer um ponto de acesso para a mesma. Na fig. 12 demonstra-se de um fragmento de código (GAMMA, 2000).

No tabela 7 demonstra-se o componente relevante do *pattern Singleton* segundo GAMMA (2000). Para obter mais informações sobre o *patterns* sugere-se a leitura do livro do Gamma (2000).

Tabela 7: Itens relevantes na composição do *pattern Singleton*

Itens	Descrição
Nome	<i>Singleton</i> – criação de Objetos
Motivação	Deve-se garantir que uma classe tenha somente uma instância. Por exemplo, embora possam existir inúmeras impressoras em um sistema, deveria haver somente um <i>spooler</i> de impressora.
Conseqüências	a) acesso controlado à instância única;

	b) espaço de nomes reduzido; c) permite refinamento de operações e representação; d) permite um número variável de instancias; e) mais flexível do que operações de classes;
Implementação	a) garantindo uma única instância mostrando na fig. 11; b) criando subclasses da classe <i>Singleton</i> .

Fonte: Gamma (2000).

A fig. 12 mostra o diagrama de classe do *pattern Singleton*. Também deve ser observado que o *pattern* possui uma variável estática e um método que retorna uma instância do objeto que implementa o *pattern*.

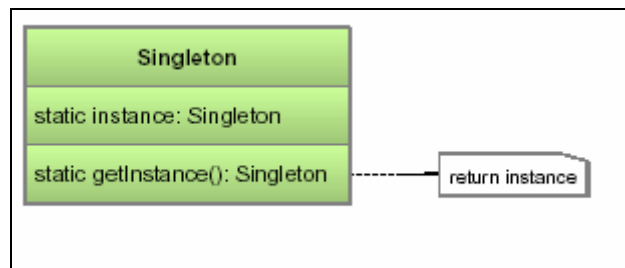


Figura 12: Demonstração esquemática do *Pattern Singleton*

Na fig. 13 demonstra-se um trecho de código com a implementação do *pattern* (à esquerda com o nome SingletonImp.java) e a outro mostrando a sua utilização (à direita UsoDoSingletonImp.java):

```

// SingletonImp.java
public final class SingletonImp {
    private static SingletonImp instance = null;

    private SingletonImp() { ... }

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

// UsoDoSingletonImp.java
public class UsoDoSingletonImp {
    :
    SingletonImp obj;
    :
    obj = SingletonImp.getInstance();
    :
}
  
```

Figura 13: Demonstração de um fragmento de código

No *framework* de cálculo de Impostos somente existe a necessidade de criação de uma única instância da classe Emissor pois a mesma uma vez instanciada será utilizada pelo restante da aplicação.

O diagrama de classes da classe Emissor pode ser visto na fig. 13 demonstrando a utilização do *pattern Singleton* na implementação do *framework*.

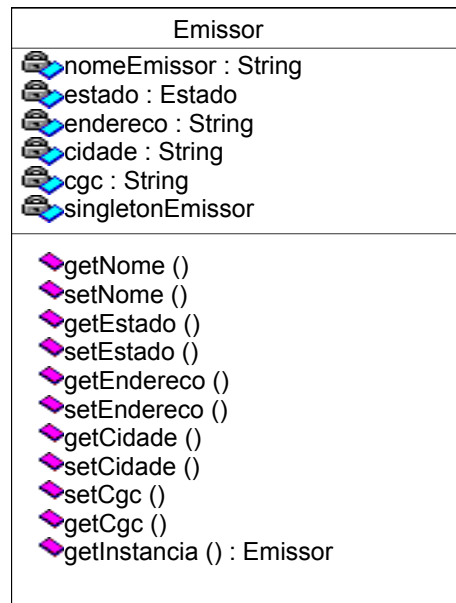


Figura 14: Diagrama de classes da classe Emissor implementando o *patterns Singleton*

Pode-se observar que retorno do último método é um objeto da classe Emissor. Este método verifica se a variável estática `singletonEmissor` já possui uma referência a um objeto Emissor e retorna a referência, caso contrario é instanciado um objeto Emissor.

A fig. 14 demonstra a classe Emissor que contém métodos e atributos que um sistema como, por exemplo, de automação comercial necessita para dispor as classes que realizam a tarefa de determinar o valor da tributação de um produto. Outras classes que são utilizadas como *Singleton* são as classes que fazes efetivamente os cálculos de `Icms`, `RetencaoME`, `SubsTributaria`, `BaseReducao` e `ReducaoEspecial`. O quadro 1 mostra o código que implementa o *Singleton* da classe `Estado` :

Quadro 1: Implementação da classe Estado com *Singleton*

```

public final class Emissor {
    private static Emissor singletonEmissor = null;
    private String nomeEmissor, cidade, endereco;
    private IEstado estado;
    private int cgc;
    public static Emissor getInstance(){
        if (singletonEmissor == null ){
            singletonEmissor = new Emissor(); }
        return singletonEmissor; }
}

```

7.1.2 FACTORY METHOD

O *pattern Factory Method* é uma interface para instanciação de objetos que mantém isoladas as classes concretas usadas na requisição da criação destes objetos. A separação de classes em uma “família” dotadas de uma mesma interface (“produtos”) e uma classe (“fábrica”) que possui um método (o *factory method*) que cria tais objetos.

Na tabela 8 demonstra-se o componente relevante do *pattern Factory Method* segundo Gamma (2000). Para obter maiores informações sugere-se a leitura do livro do Gamma (2000).

Tabela 8: Demonstração do itens relevantes da composição do *Factory Method*

Itens	Descrição
Nome	<i>Factory Method</i> – criação de classes.
Motivação	Os <i>frameworks</i> usam classes abstratas para definir e manter relacionamentos entre objetos. Um <i>framework</i> é frequentemente responsável também pela criação destes objetos. O <i>pattern Factory Method</i> oferece uma solução. Ele encapsula o conhecimento sobre a subclasse que deve ser criada e move este

	conhecimento para fora do <i>framework</i> .
Conseqüências	<ul style="list-style-type: none"> a) fornece ganchos para subclasses; b) conecta hierarquias de classes paralelas.
Implementação	<ul style="list-style-type: none"> a) duas variedades principais: <ul style="list-style-type: none"> - a classe <code>Creator</code> é uma classe abstrata e não fornece uma implementação para o método fabrica que ela declara. - quando a classe <code>Creator</code> é uma classe concreta e fornece uma implementação por omissão para o método fabrica. b) Métodos <i>factory</i> parametrizadas.

Fonte: GAMMA (2000)

Na fig. 15 demonstra-se a estrutura de classes que formam o *pattern Factory Method*.

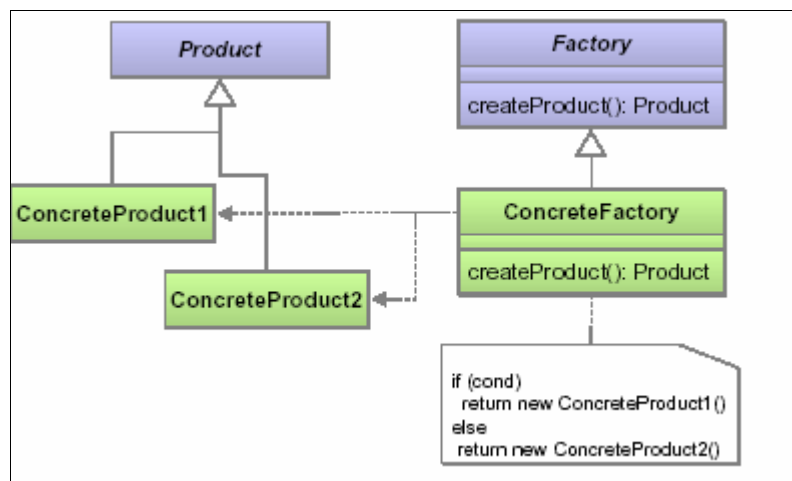


Figura 15: Demonstração de *pattern Factory Method*

O *framework* de cálculo de imposto utilizará quando fizer a criação das classes de imposto como IPI, ICMS, ect. Quando da instanciação da classe de Imposto será passado um parâmetro que identificará qual classe de imposto deverá ser criada.

Na fig. 16 demonstra-se a interface `Imposto` que modela os métodos necessários para utilizar nas classes concretas `Ipi` e `Icms` que a implementa. Nesta classe estão definidos os métodos que são necessários quando se desejar implementar algum novo tributo designado pelo governo. Caso seja necessário implementar métodos de um novo tributo faz-se na nova classe que implementa a classe abstrata `Imposto`. No diagrama da fig. 16 pode-se observar que se tem um método `createImposto` da classe abstrata `ImpostoFactory` que é

responsável pela instanciação de objetos que implementam a interface `Imposto`. Cada nova classe que implementar a interface `Imposto` e desejar incluir-se na `ImpostoFactory` deve sobrescrever o método `create`. Isto cria uma fábrica de objetos que funciona de forma polimórfica para inclusão e instanciação de novos objetos `Imposto`.

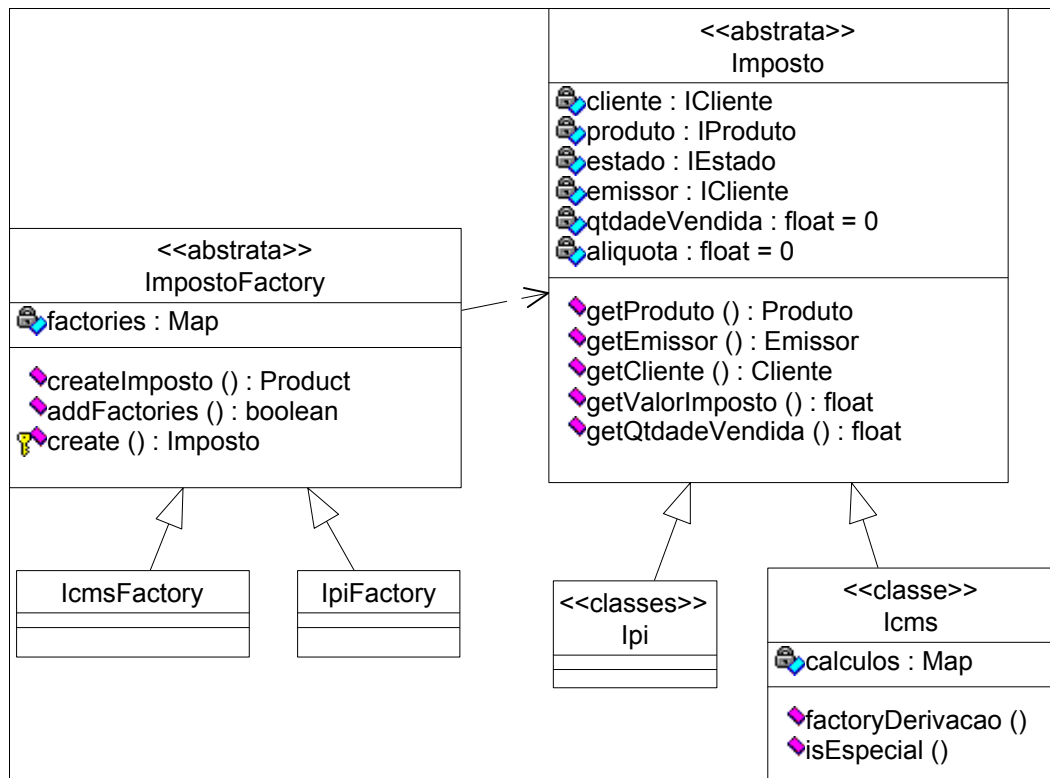


Figura 16: Diagrama da classe abstrata `Imposto`

Para que uma nova classe que calcule imposto seja utilizada na `ImpostoFactory`, ela deve se auto-registrar implementando uma classe privada, incluindo e chamando estaticamente a sua inclusão na `ImpostoFactory`. O quadro 2 demonstra o código exemplificando:

Quadro 2: Implementação da `ImpostoFactory`

```

private static class Factory extends ImpostoFactory {
    protected Imposto create() {
        return new Icms(); } }
static {
    ImpostoFactory.addFactory(
        "Icms", new Factory()); }
  
```


A fig. 16 demonstra os métodos que se fazem necessário quando se deseja implementar um cálculo de imposto já existente ou outro que ao governo venha a decretar. Nele encontra-se especificados os métodos básicos que quase todas as classes - como Ipi, ICMS, (implementadas no *framework*) e outras que implementam os cálculos de tributação - necessitam para determinar o valor da tributação.

7.1.3 FLYWEIGHT

O *pattern Flyweight* consiste em suportar grande quantidade de objetos de granularidade fina onde o custo de armazenamento é alto devido à grande quantidade de objetos que podem ser agrupados (GAMMA, 2000). Às vezes pode-se reduzir bastante o número de diferentes classes que se necessita instanciar se puder reorganizar quais instâncias são fundamentalmente de mesma forma exceto por alguns poucos parâmetros.

Na tabela 9 demonstra-se item relevante ao *pattern Flyweight* para sua formação e aplicação. Para maiores informações da utilização do *Flyweight* recomenda-se a leitura do livro de Gamma (2000).

Tabela 9: Descrição dos itens relevantes do *pattern Flyweight*

Itens	Descrição
Nome	<i>Flyweight</i> – estrutural de objetos.
Motivação	Quando se tem uma grande quantidade de objetos de fina granularidade podendo-se agrupá-los em famílias de objetos conforme as afinidades encontradas.
Conseqüências	<ul style="list-style-type: none"> a) redução de número de instâncias; b) economia de armazenamento e espaço de compartilhamento de objetos; c) o tempo de execução aumenta porque o estado de cada objeto deve ser calculado.
Implementação	<ul style="list-style-type: none"> a) remoção de estados extrínsecos; b) a gerencia dos objetos compartilhados.

Fonte: Gamma (2000).

Na fig. 17 demonstra-se a estrutura de classes que formam o *pattern flyweight*. Pode-se notar que o mesmo possui uma fábrica de objetos. Tem-se também uma interface que contém os métodos comuns às classes que implementam a interface.

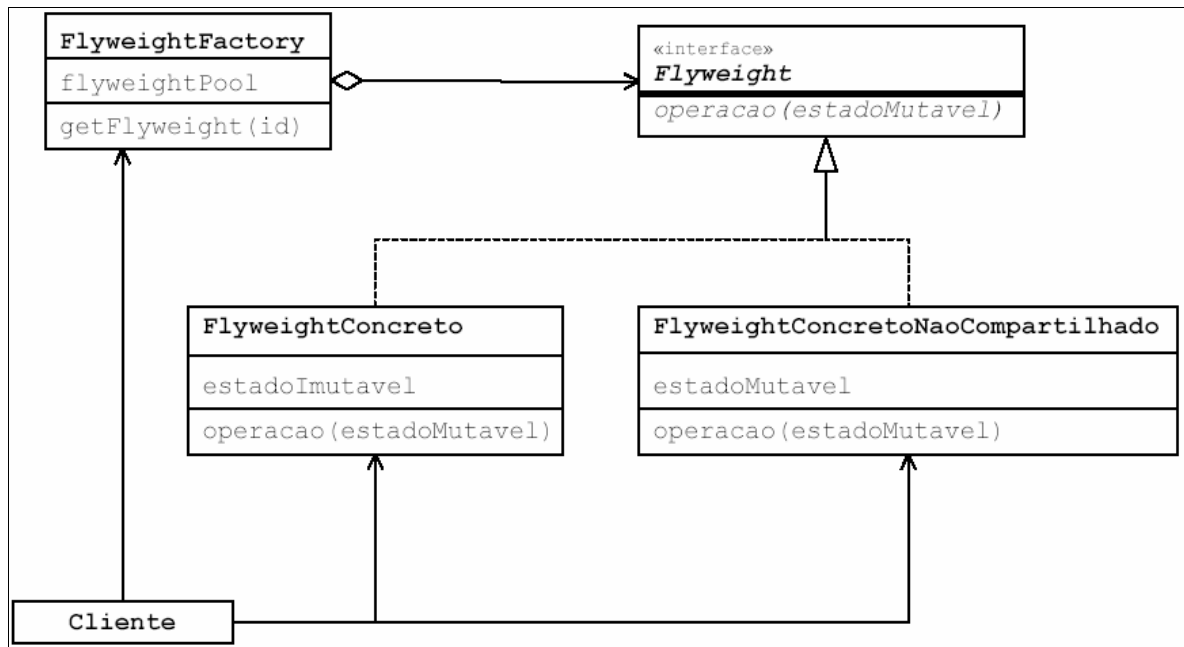


Figura 17: Demonstração da estrutura do *pattern Flyweight*

Na fig. 18 demonstra-se o diagrama de classes da estrutura do *pattern flyweight* sendo modelado no *framework* de cálculo de imposto. Observa-se que a modelagem abriga dois *patterns* que é um *Factory Method* na classe *Icms* e um *Strategy* com relação à implementação do cálculo de imposto.

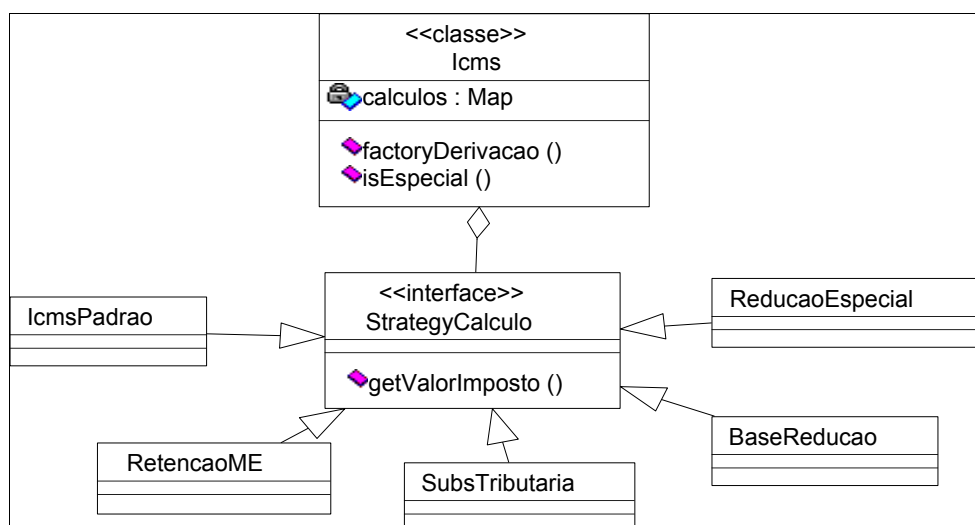


Figura 18: Diagrama de classe demonstrando o *pattern Flyweight*

7.1.4 STRATEGY

O *pattern Strategy* consiste na definição de uma família de algoritmos, encapsula cada um e faz com que eles possam ser permutáveis. Isto permite que o algoritmo varie independentemente do cliente que o utilizar (GAMMA, 2000).

Na tabela 10 demonstra-se os itens relevantes ao *pattern Strategy* para sua formação e aplicação. Para maiores informações da utilização do *Strategy* recomenda-se a leitura do livro de Gamma (2000).

Tabela 10: Descrição dos itens relevante ao *pattern Strategy*

Itens	Descrição
Nome	<i>Strategy</i> – comportamental de objetos.
Motivação	Quando se necessita de um algoritmo que trata de diferentes os dados submetidos a ele. Pode-se citar o tratamento de quebra de linha mencionado o livro do Gamma (2000) onde se modela um tratamento diferenciado para o problema de quebra de linhas de um <i>stream</i> .
Conseqüências	<ul style="list-style-type: none"> a) famílias de algoritmos relacionados; b) uma alternativa ao uso de subclasses; c) <i>strategies</i> eliminam comandos condicionais; d) uma escolha de implementações; e) os clientes devem estar cientes dos diferentes <i>strategies</i>; f) aumento do número de objetos.
Implementação	<ul style="list-style-type: none"> a) estratégias de implementação passadas como parâmetro; b) torna os objetos <i>Strategies</i> opcionais;

Fonte: Gamma (2000)

Na fig. 19 observa-se a composição estrutural do *pattern*. Nota-se a criação de uma interface *Strategy* que contém um método que é implementado de diferentes maneiras nas classes concretas que implementam a interface *Strategy*.

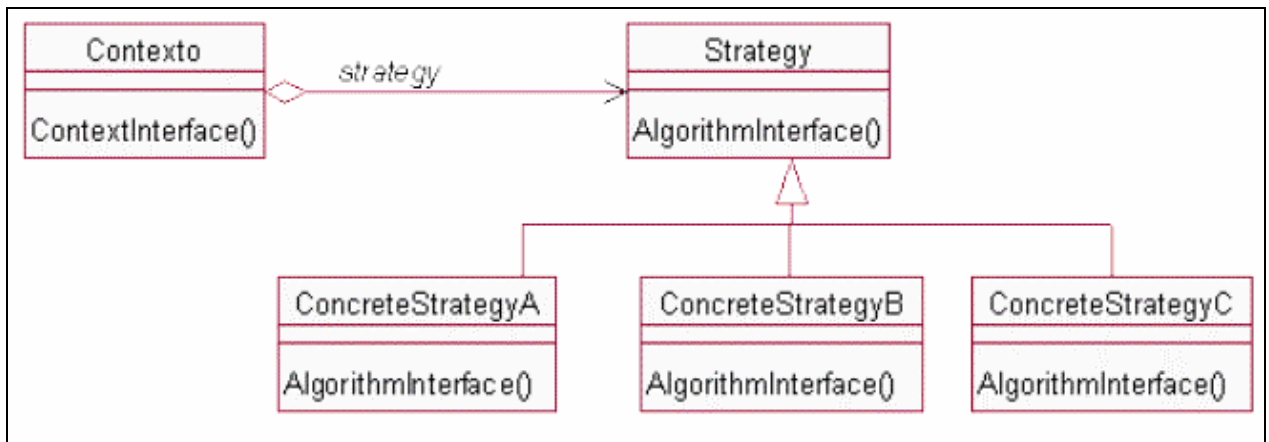


Figura 19: Demonstração da estrutura do *pattern Strategy*

No *framework* de cálculo de imposto verifica-se a aplicação deste *pattern* quando da definição da classe abstrata `Imposto` que tem um método chamado `getValorImposto()`. Este método é implementado diferentemente nas classes concretas que implementam a classe `Imposto` consistindo em um exemplo de aplicação do *pattern Strategy*. Pode-se observar o *Strategy* no diagrama de classe da fig. 20 que demonstra a modelagem da classe `Imposto`.

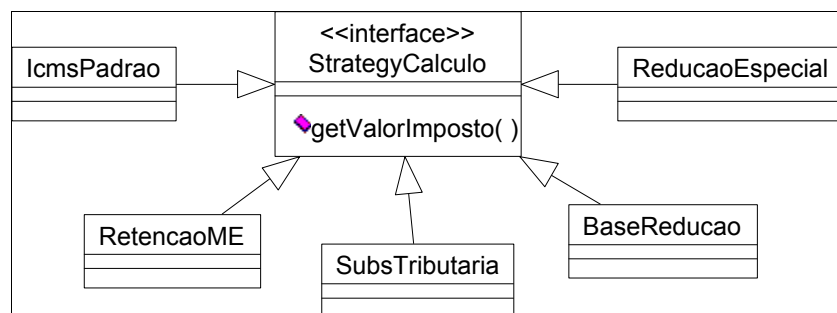


Figura 20: Classe abstrata imposto demonstrando a aplicação de *pattern Strategy*

Na fig. 20 observa-se que a modelagem utiliza dois *patterns Factory Method* e *Strategy*. O *Factory Method* que define uma fábrica de criação de objetos que fazem o cálculo de impostos e o *Strategy* que define uma implementação diferenciada para cada objeto criado pelo *Factory* no método que retorna o valor do imposto. O quadro 3 mostra o código implementado na classe `BaseReducao` que implementa a interface `StrategyCalculo`.

Quadro 3: Demonstração do código implementando um *Strategy*

```

/**
 *
 * @param imp Um objeto instanciado a partir da classe Icms utilizado no calculo
 * @return Retorna o valor da Base de Redução do produto especificado na classe Icms
 */
public float getValorImposto(Icms imp) {

    ICliente cliente = imp.getCliente();
    Emissor emitente = Emissor.getInstance();
    if ( cliente.getEstado().getUnidadeFederativa() == emitente.getEstado().getUnidadeFederativa() ) {
        valorBaseReducao = imp.getProduto().getValorUnitario() * imp.getQtdVendida() *
        imp.getProduto().getBaseReducao();
        valorBaseReducao *= imp.getProduto().getTxICMS();
    }
    return valorBaseReducao;
}

```

7.2 DEMAIS INTERFACES E CLASSES QUE COMPÕEM O *FRAMEWORK*

O *framework* de cálculo de imposto é modelado com um diagrama de classes como se mostra na fig. 23 que possui as interfaces Cliente e Produto, que até o presente momento não foi feita nenhuma explanação de seu funcionamento.

A interface `InterfaceCliente` define métodos necessários para que possa ser implementada uma classe Cliente que supre as necessidades de informação necessárias à realização dos cálculos de impostos. A modelagem desta interface constitui-se dos métodos e atributos necessários para realizar a tributação como por exemplo o método que define o tipo de cliente podendo ser um cliente pessoa física, jurídica, micro empresa e especial. Na fig. 21 a seguir o diagrama demonstra os métodos e atributos inerentes à interface.

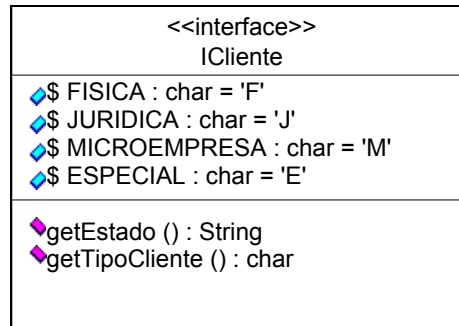


Figura 21: Modelo da interface IntefaceCliente

A outra interface necessária para composição do *framework* é a de `InterfaceProduto`. Esta se constitui de uma interface que é responsável por dispor de métodos necessários para realização dos devidos cálculos dos imposto auferidos a uma determinada classe `Produto` que modela um objeto produto do mundo real. A fig. 22 demonstra a modelagem da interface denominada de `InterfaceProduto`.

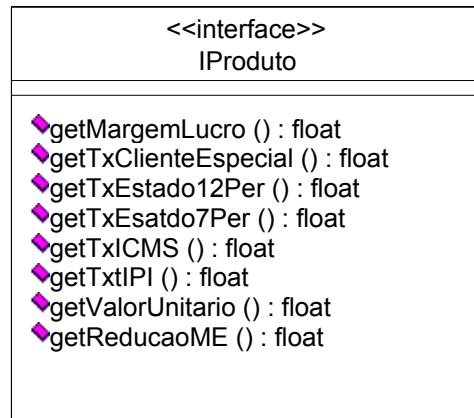


Figura 22: Modelo da interface InterfaceProduto

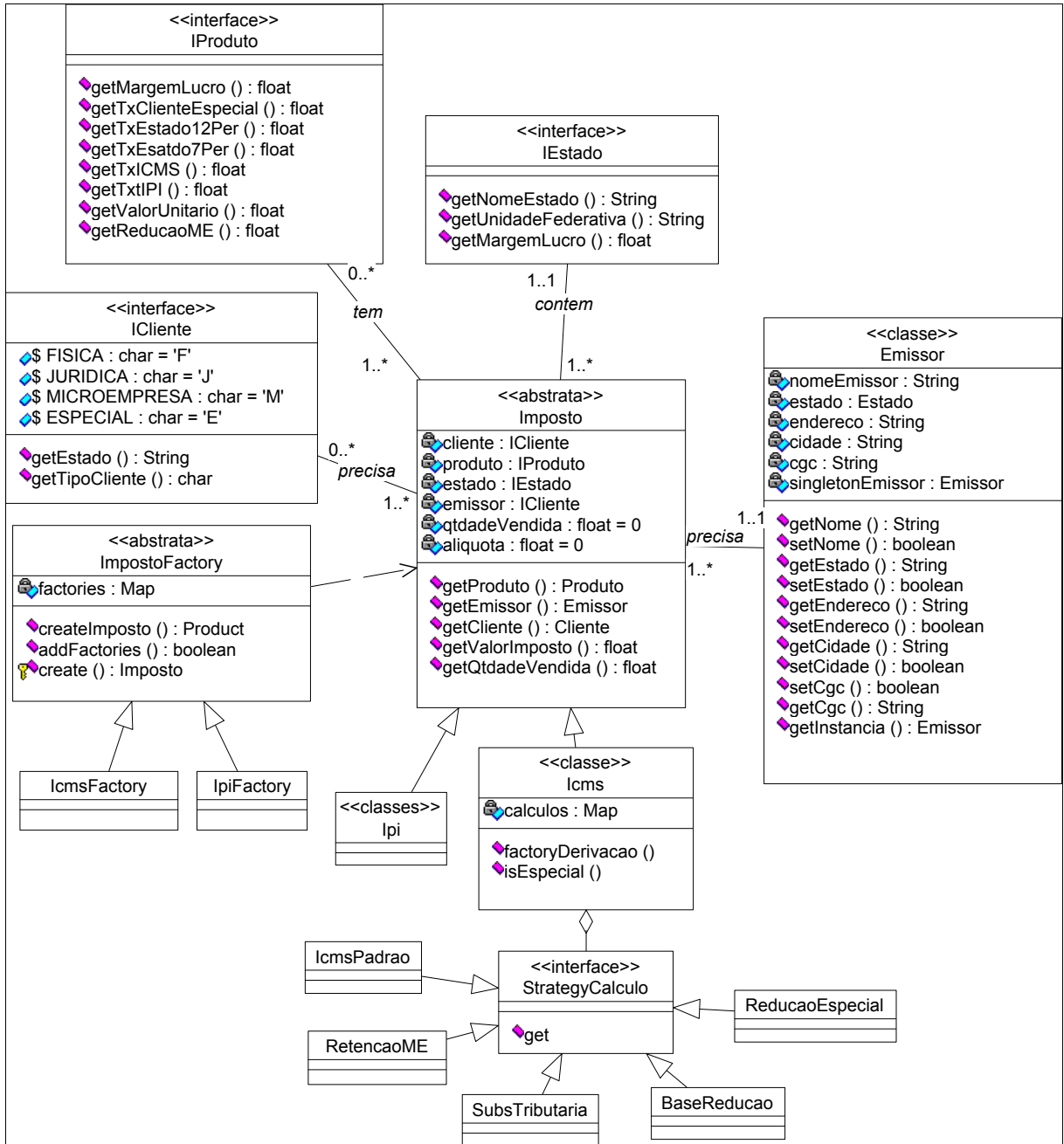


Figura 23: Diagrama de classes do *Framework*

Na fig. 23 demonstra-se o diagrama de classes do *framework*. Nele pode-se ter uma visão global de todos os padrões mencionados neste capítulo aplicados no contexto do trabalho. Pode-se visualizar a localização dos padrões dentro do diagrama visualizando onde estão sendo usados para resolver problemas de criação de objetos e estratégias de cálculos.

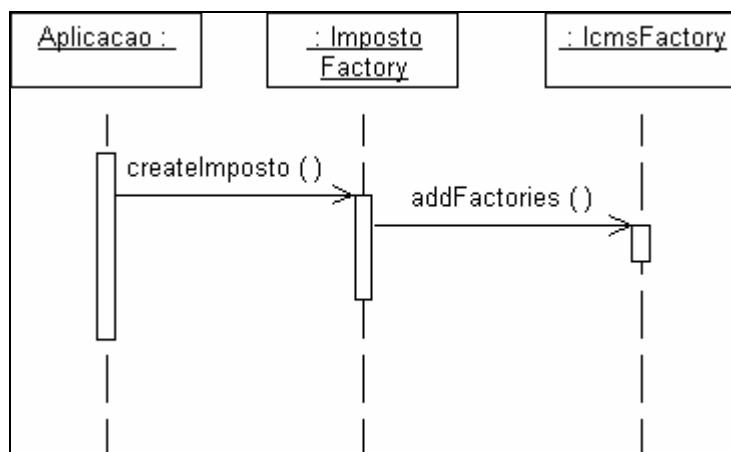


Figura 24: Diagrama de seqüência da criação de um objeto imposto(createImposto)

O diagrama de seqüência da fig. 24 mostra a seqüência de desencadeamento das chamadas dos métodos para criar um novo objeto imposto. Neste exemplo está sendo criado um objeto `Icms` que é criado dentro da classe `ImpostoFactory`.

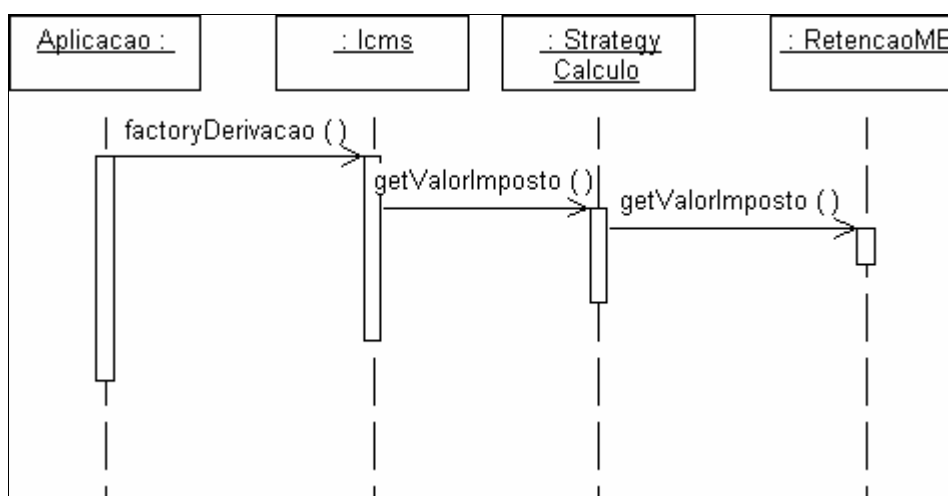


Figura 25: Diagrama de seqüência do cálculo de um imposto(getValorImposto)

O diagrama de seqüência demonstrado na fig. 25 verifica-se o desencadeamento das chamadas dos métodos que calculam o valor de um imposto. O diagrama mostra o `factoryDerivacao` que cria todos os impostos que são derivados da classe `Icms` e chama o método `getValorImposto` implementado nas classes que implementam a estratégia de cálculo de cada derivação do ICMS.

7.3 ESTUDO DE CASO DO PROTÓTIPO QUE UTILIZARÁ O *FRAMEWORK* DE CÁLCULO DE IMPOSTO

Neste tópico é apresentado um estudo de caso hipotético, utilizado para exemplificar o uso do *framework*.

Existe um aumento crescente da informatização de estabelecimentos comerciais em todo o país para ajudar na tomada de decisões, melhoria na qualidade de serviços prestados aos clientes e no controle da arrecadação de imposto devido a União.

A loja PaperOfficer S.A. que comercializa material de escritório deseja informatizar o registro de entrada e saída das mercadorias que comercializar. A loja vende para diversos clientes como pessoa física, jurídica, para empresas em diferentes estados com alíquotas de imposto diferenciados.

O cliente, após ter selecionado as mercadorias, dirige-se a um operador de caixa para efetuar a compra. O operador de caixa informa ao sistema o código do cliente que estão previamente cadastrados. Após ter informado os itens através da leitura de código finaliza a venda e destaca o cupom fiscal da impressora fiscal. O usuário do sistema deverá cadastrar as alíquotas de imposto para todos os estados brasileiros, informando a margem de lucro concedida pelo governo no respectivo estado de destino da mercadoria, os clientes e os produtos informando as alíquotas que incidem em cada produto da loja para os devidos impostos na qual o produto é tributado.

7.4 DIAGRAMAS DO ESTUDO DE CASO

Para o estudo de caso do protótipo há os seguintes casos de uso (fig. 26):

- a) o cliente após selecionar as mercadorias que deseja dirige-se ao caixa para efetuar o pagamento das mercadorias. O caixa registra os itens no sistema efetuando. Depois de informados todos os itens o sistema realiza o cálculo dos impostos incidentes sobre os produtos. Os clientes estarão previamente cadastrados no sistema juntamente com os produtos;
- b) o usuário cadastrará as margens de lucro concedidas pelo governo aos receptivos estados brasileiros;

- c) o usuário cadastrará os clientes;
- d) o usuário cadastrará os produtos.

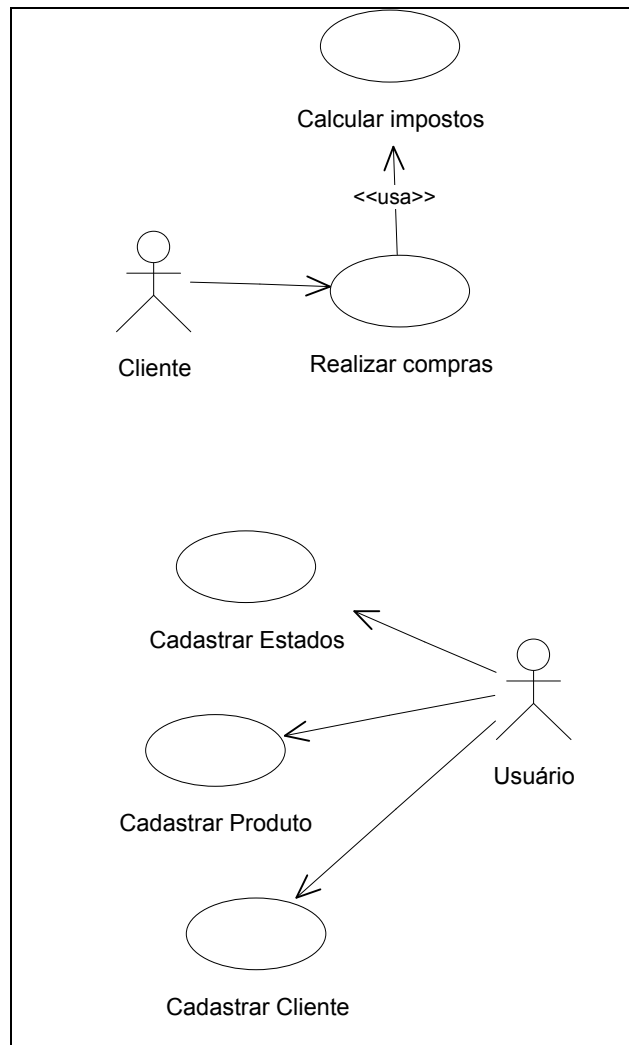


Figura 26: Diagrama de Caso de Uso

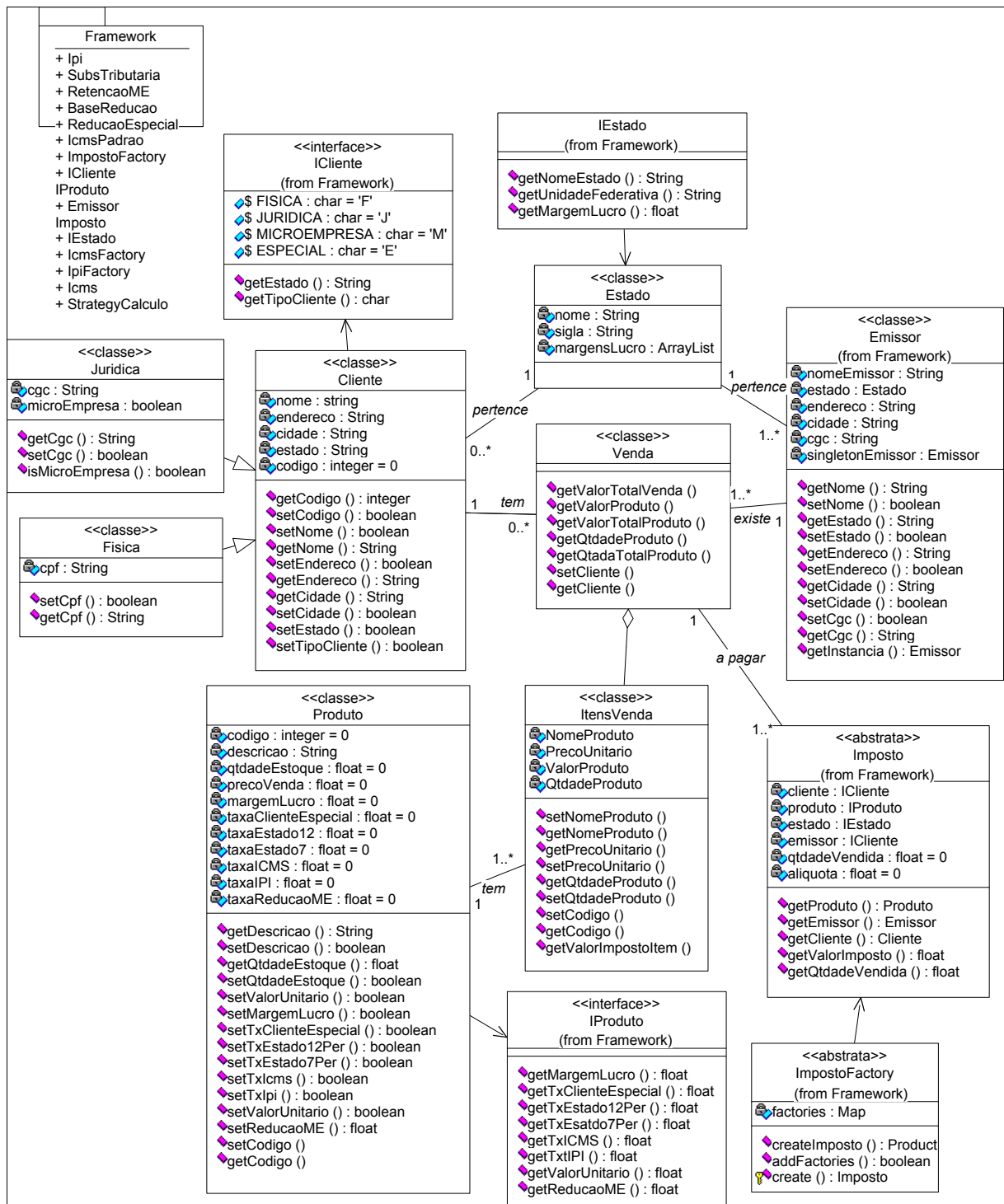


Figura 27: Diagrama de classe do protótipo.

A fig. 27 demonstra o diagrama de classe do protótipo que utilizará a *framework* de para realizar os cálculos de imposto. Pode-se notar que as classes pertencentes ao *framework* são destacadas com usando a nomenclatura “(from framework)” feita pela ferramenta CASE Rational Rose.

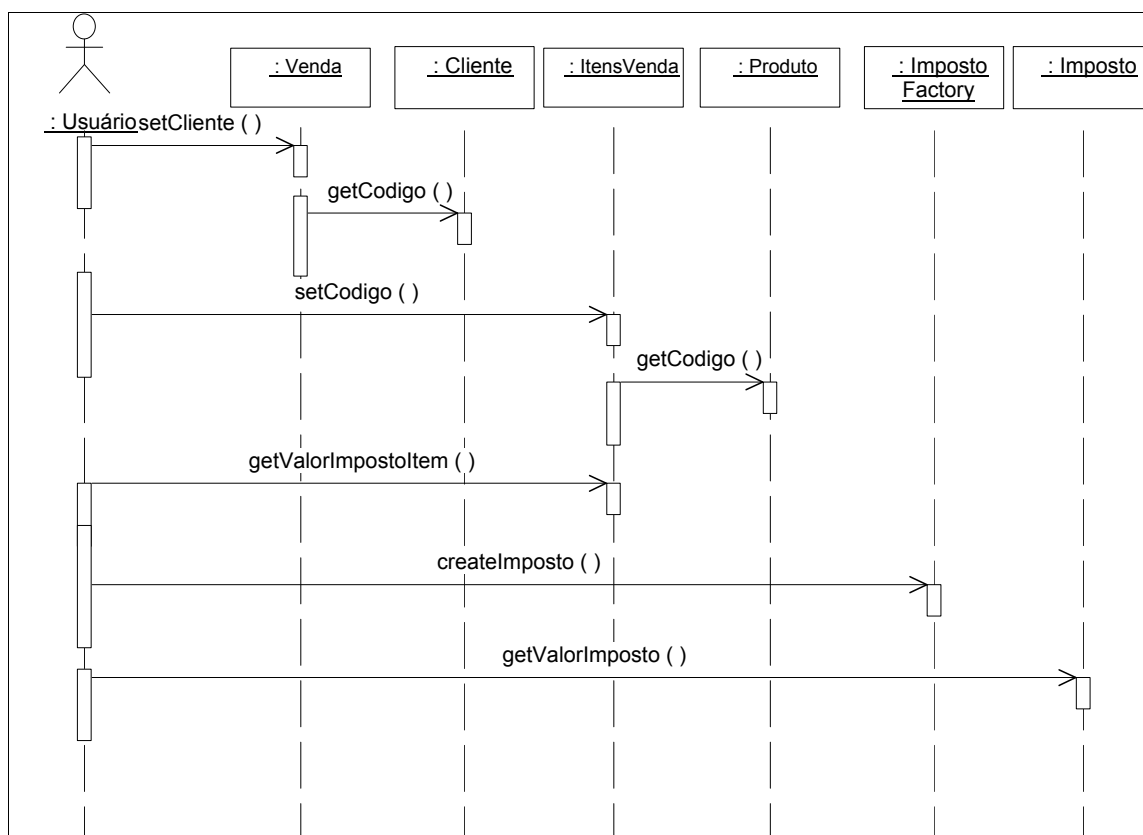


Figura 28: Diagrama de Seqüência

A fig. 28 demonstra o diagrama de seqüência do protótipo modelando a seqüência de ações quando se está fazendo o atendimento ao cliente no caixa do estabelecimento.

Depois de encerrada a modelagem do protótipo com os diagramas das fig. 26, 27 e 28, implementou-se na linguagem de programação Java, utilizando a ferramenta Borland JBuilder 8 Enterprise 8.0.140.0 para elaboração das telas que o usuário poderá interagir com a aplicação.

7.5 PROTÓTIPO IMPLEMENTADO COM O *FRAMEWORK*

Neste tópico são demonstradas as telas que são utilizadas no protótipo para mostrar a utilização da implementação do *framework* de cálculo de imposto.

Figura 29: Cadastro de informações dos Estados

A fig. 29 demonstra a tela de cadastro das informações dos estados. Nesta tela constam os seguintes itens:

- código: número interno do protótipo informado neste caso pelo operador;
- nome do estado;
- unidade federativa (UF): sigla da UF;
- taxa de Icms (ICMS): alíquota padrão do Estado;
- margens de lucros: informadas pelo governo.

Figura 30: Cadastro das informações de Clientes.

A fig. 30 demonstra a tela de cadastro das informações do cliente. Nela constam os seguintes itens:

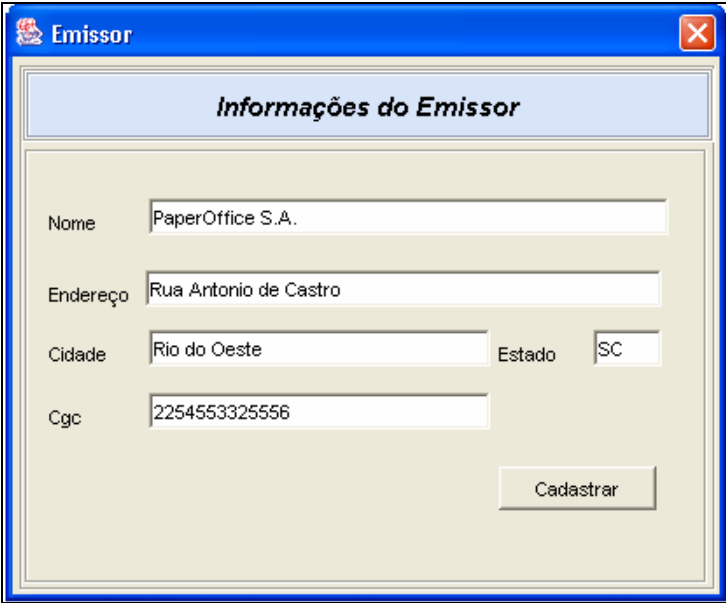
- a) código: número interno informado neste caso pelo operador;
- b) nome;
- c) sexo;
- d) tipo: qualificação dada ao cliente que depende da situação jurídica na qual pertence;
- e) documento: depende a situação jurídica poderá ser o CPF, CGF, Inscrição Estadual;
- f) estado civil;
- g) endereço;
- h) cidade;
- i) estado: unidade federativa na qual a pessoa é oriunda.

Figura 31: Cadastro de informações do Produto.

A fig. 31 demonstra a tela onde se informa o dado do produto. Destacam-se os seguintes itens:

- a) código: número interno do protótipo informando neste caso pelo operador;
- b) nome do produto;
- c) preço : preço de venda da mercadoria;
- d) ICMS: alíquota que incide sobre o produto;
- e) IPI: alíquota de IPI que incide sobre o produto;

- f) Cliente Especial: alíquota que incide quando venda para cliente do cadastrado como especial;
- g) Margem Subst.: margem de substituição;
- h) Retenção M.E.: alíquota de retenção quando o produto é vendido para cliente Micro Empresa;
- i) Base de Redução: base de redução informada pelo governo sobre alguns produtos;
- j) Retenção Estado 12%: retenção para Estado com alíquota de ICMS de 12%;
- l) Retenção Estado 7%: retenção para Estado com alíquota de ICMS de 7%;



A imagem mostra uma janela de software com o título "Emissor" e o subtítulo "Informações do Emissor". O formulário contém os seguintes campos:

- Nome: PaperOffice S.A.
- Endereço: Rua Antonio de Castro
- Cidade: Rio do Oeste
- Estado: SC
- Cgc: 2254553325556

Um botão "Cadastrar" está visível na parte inferior direita do formulário.

Figura 32: Cadastro de Informações do Emissor

A fig. 32 demonstra o cadastramento de informações referente ao estabelecimento comercial. Nesta tela constam as seguintes informações:

- a) nome;
- b) endereço;
- c) cidade;
- d) estado: deve-se levado em consideração o estado emissor para efeitos de tributação;
- e) cgc;

Registro de Vendas

Registro de Vendas

Cliente

Epitafio da Cunha

Itens de Venda

Descrição	Quantidade	Preço
ProdutoClienteEspecial	1.0	100.0

Impostos tributados

ICMS: 17.0 IPI:10.0 Cliente Especial:14.45

Iniciar Venda Calcular Impostos Código

Figura 33: Tela de atendimento ao cliente na compra de mercadorias.

A fig. 33 demonstra a tela onde o operador faz o atendimento ao cliente durante uma venda. Nela o operador informa o cliente informando a letra “C” seguida do código do cliente conforme informado em seu cadastro. Depois de informado o cliente fará a digitação dos itens selecionados pelo cliente. Para informar os produtos deve-se digitar a letra “P” seguida do código do produto conforme informado em seu cadastro. Para saber os imposto que incidiram sobre o produto o operador clica no botão “Calcular Imposto”.

Nas fig. 34 e 35 estão alguns exemplos da tela de atendimento ao cliente demonstrando a utilização do *framework* para realizar os cálculos de impostos dos produtos.

Registro de Vendas

Registro de Vendas

Cliente

Empresa de Alimentos S.A.

Itens de Venda

Descrição	Quantidade	Preço
ProdutoMicroEmpresa	1.0	100.0

Impostos tributados

ICMS: 17.0 IPI:10.0 Retenção ME: 3.4

Iniciar Venda Calcular Impostos Código

Figura 34: Tela de atendimento a um cliente Micro Empresa

Registro de Vendas

Registro de Vendas

Cliente

Antonio Carlos

Itens de Venda

Descrição	Quantidade	Preço
Produto	1.0	100.0

Impostos tributados

ICMS: 17.0 IPI:10.0

Iniciar Venda Calcular Impostos Código

Figura 35: Cliente com tributação de ICMS e IPI

8 CONCLUSÃO

Através deste trabalho foi possível aprimorar os conceitos referentes à modelagem e programação orientada a objetos e ter uma nova visão dos benefícios obtidos quando se implementa utilizando uma ou mais técnicas de representação do modelo de negócio do software na codificação.

A utilização de uma arquitetura de software em camadas proporciona uma melhor acomodação das funcionalidades na qual o sistema se destina. A modelagem de uma camada de negócios faz com que os bancos de dados livres-se de tratar informações que não fazem parte do contexto de um banco de dados e na camada de interface libera o desenvolvedor a preocupar-se com o desenho e funcionalidades da interface não poluindo o código da interface com o da camada de negócios. Outro ponto a ser considerado é o baixo acoplamento existente entre as camadas do sistema, pois todas são independentes umas das outras facilitando a modificação, manutenção, reutilização e portabilidade.

A construção de um software modelado em camadas, que tratam exatamente das funcionalidades para a qual são desenvolvidas, aumenta a quantidade de itens que podem ser reutilizados - desde linhas de códigos até a reutilização de toda uma camada através de um componente, que no caso deste trabalho pode-se reutilizar toda a camada de negócio que é responsável pelos cálculos de impostos. Dentro desta camada existem várias formas de reutilização desde as linhas de códigos - como por exemplo os cálculos referentes ao ICMS - à própria modelagem servindo de base para incrementar novos cálculos de impostos, até os padrões implementados e a reutilização da camada como um todo, pois a mesma constitui-se de um *framework* para cálculos de impostos.

A utilização de padrões de projeto dentro da modelagem e codificação cria micro unidades que podem ser alteradas com pouca ou nenhuma interferência no software. Os padrões selecionados, *Singleton*, *Factory Method*, *Flyweight* e *Strategy* criam um alto nível de abstração da maneira como será utilizada a implementação do padrão.

Com a modelagem e implementação de cálculos de impostos foi possível utilizar padrões de projetos, que solucionaram desde o problema da não necessidade de criação de várias instâncias de um mesmo objeto, que é único para todo o sistema; até a criação de objetos tendo somente uma classe responsável pela instanciação, tornando-se uma fábrica de

objetos; passando pela utilização de um padrão que dita uma estratégia a ser adotada quando há algoritmos muito semelhantes.

A implementação dos padrões resultou na formação de um *framework* facilitando as tarefas do desenvolvedor e sendo extensível quando houver necessidade de acrescentar novas classes ou implementar novos padrões. Para o desenvolvedor de uma aplicação o funcionamento dos cálculos será totalmente abstraído pelo conhecimento agregado ao *framework*, fazendo com que o desenvolvedor tenha mais tempo nas tarefas inerentes a aplicação. Outra vantagem apresentada é o baixo acoplamento aliado à facilidade de utilização, pois o *framework* necessita da implementação de três interfaces IProduto, ICliente e IEstado.

Com o término do trabalho concluiu-se que o mesmo pode ser utilizado por todas as pessoas que venham a necessitar ter o conhecimento da aplicação de padrões e desenvolvimento de *frameworks*, facilitando a implementação de sistemas mais complexos, presentes cada vez mais no cotidiano de uma empresa de desenvolvimento de sistemas.

8.1 LIMITAÇÕES

O *framework* de cálculo de imposto é limitado a calcular o IPI e ICMS juntamente com as derivações do ICMS. O *framework* não tem a implementação de *pattern Facade* que implementa uma forma de acesso. O cálculo de substituição tributária não leva em consideração algumas flexibilidades de lei devido a algumas diferenças existentes para alguns estados brasileiros.

8.2 EXTENSÕES

Como sugestão para continuação deste trabalho inclui-se:

- a) implementação do *pattern Facade* para acesso ao *framework*;
- b) implementação de outros impostos;
- c) implementação das variações da lei para substituição tributária;
- d) analisar a possibilidade de aplicação de outros padrões para mensuração de qualidade, reusabilidade e performance do *framework*;
- e) geração de uma linguagem de definição de interface (IDL) para portar o *framework* para outras linguagens de programação como por exemplo Delphi.

REFERÊNCIAS BIBLIOGRÁFICAS

AGUIAR, A., **Software Patterns: uma forma de reutilizar conhecimento**, [S.l],[200?]. Disponível em <http://www.fe.up.pt/_aaguiar/patterns>. Acesso em: 4 mar. 2003.

ADVISOR. **BLAZE Advisor Solutions Suíte**, [S.l],[200?]. Disponível em: <www.atsolutions.com.br/Blaze/elementsj.htm>. Acesso em: 01 fev. 2003.

AMARO, Luciano. **Direito tributário brasileiro**. 8 ed. São Paulo: Saraiva, 2002.

AMBLER, Scott W. **Análise e projeto orientado a objeto - seu guia para desenvolver sistemas robustos com tecnologia de objetos**. Tradução Oswaldo Zanelli. Rio de Janeiro: Infobook, 1998.

ALUR, Deepk; CUPRI, John; MALKS, Dan. **As melhores práticas e estratégias de design**. Rio de Janeiro: Campus, 2002.

BORGES, Humberto Bonavides. **Planejamento tributário: IPI, ICMS, ISS: economia de impostos, racionalização de procedimentos fiscais**. 3 ed. São Paulo: Atlas, 1999.

BORGES, Humberto Bonavides. **Gerência de Impostos: IPI, ICMS, ISS**. 3 ed. São Paulo: Atlas, 2000.

BOOCH, Grady. **Object-oriented analysis and design with applications**. Santa Clara: Addison-Wesley Publishing Company, 1994.

BRAGA, Rosana Teresinha Vaccare. **Um processo para Construção e Instanciação de Frameworks baseada em uma linguagem de Padrões para um Domínio Específico**. 2002. 224 f. Tese de Doutorado do Instituto de Ciências Matemáticas e de Computação, São Paulo.

CARRAZZA, Roque Antonio. **ICMS**. 9 ed. São Paulo: Malheiros Editores Ltda, 2002.

CETUS Team, **Architecture & Design: Patterns**, [S.l],[200?]. Disponível em <http://www.cetus-links.org/oo_patterns.html>. Acesso em 4 fev.2002.

CMG, Component Management Group (CMG). **Anti-patterns**, [S.l],[200?].Disponível em <<http://160.79.202.73/Resource/AntiPatterns/index.html>> Acesso em 15 fev. 2003.

COOPER, James W. **The Design Patterns Java Companion.**

CUNHA, Roberto Silvino da. **Ferramenta para a geração de sistemas baseado nas técnicas de *pattern* e *framework* a partir do System Architect.** 2001. 99 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

DEITEL, Harry M.; DEITEL, Paul J. **Java como programa.** 3. ed. Tradução Edson Furnankiewicz. Porto Alegre: Bookman, 2001.

FABRETTI, Laúdio Camargo. **Prática tributária da micro e pequena empresa.** 3 ed. São Paulo: Atlas, 1999.

FAYAD. **Object-oriented application frameworks,** [S.l],[199?]. Communications of the ACM 40. Disponível em: <www.cs.wustl.edu/~schmidt/CACM-frameworks.html>. Acesso em: 10 mar. 2003.

FURLAN, José Davi. **Modelagem de objetos através de UML – the unified modeling language.** São Paulo: Makron Books, 1998.

GAMA, Erich et al. **Padrões de projeto: soluções reutilizáveis de software orientado a objetos.** Porto Alegre: Bookman, 2000.

HAY, D.; HEALY, K. A. **Defining business rules - what are they really? Guide project by business rules,** [S.l],[200?]. Disponível em: <www.guide.org/ap/apbrules.htm>. Acesso em: 12 jan. 2003.

HILLSITE. **Patterns,** [S.l],[200?]. Disponível em <<http://hillside.net/patterns/>>. Acessado em: 15 maio 2003.

LARMAN, Craig. **Utilizando UML e padrões: uma introdução à análise e projeto orientado a objetos.** Tradução Luiza A. Meireless Salgado. Porto Alegre: Bookman, 2000.

MARTIN, James. **Princípios de análise e projeto baseado em objetos.** Tradução Cristina Bazán. Rio de Janeiro: Campus, 1994.

MORRISON, Michael. **Java 2 para leigos passo a passo.** Rio de Janeiro: Ciência Moderna Ltda., 2000.

PERKINS, A. **Business Rules are Meta Data**. [S.l],[200?]. Disponível em:<<http://www.brcommunity.com>>. Acesso em: 30 jan. 2003.

PREE, Wolfgang. **Design patterns for object-oriented software development**. Nova York: Addison-Wesley, 1995.

PRESSMAN, Roger S. **Engenharia de software**. Tradução Jose Carlos Barbosa dos Santos, revisão técnica Jose Carlos Maldonado, Paulo César Masieiro e Rosely Sanches. São Paulo: Makron Books, 1995.

RATIONAL ROSE. **Using Rose (Windows/UNIX)**, [S.l],[200?]. Disponível em: <<http://www.rational.com/rose>>. Acesso em: 26 mar. 2003.

RICCIONI, Paulo Roberto. **Introdução a Objetos Distribuídos com CORBA**. Florianópolis: Visual Books, 2000.

RULEMACHINES. **Visual Rule Studio**, [S.l],[200?]. Disponível em: <<http://www.rulemachines.com>>. Acesso em: 01 fev. 2003.

RUMBAUGH, James et al. **Modelagem e projetos baseados em objetos**. Rio de Janeiro: Campus, 1994.

SANTOS, José Maria Rodrigues Santos Júnior. **Arquitetura de aplicações**, [S.l],[200?].Disponível em< www.unit.br/zemaria>, Acesso em 14 mar. 2003,

SILVA, Ricardo Pereira. **Suporte ao desenvolvimento e uso de frameworks e componentes**. 2000. 262 f. Tese de Doutorado da Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre.

SINTES, Tony. **Aprenda programação orientada a objetos em 21 dias**. Tradução João Eduardo Nóbrega Ascencio. São Paulo: Pearson Education do Brasil, 2002

SCHMITD, Roger Anderson. **Ferramenta de auxílio ao processo de desenvolvimento de software integrando tecnologias otimizadas**. 2001. 115 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SCHMIDT et al. **Software Patterns**, [S.l],[200?]. Disponível em <<http://www.cs.wustl.edu/~schmidt/CACM-editorial.html>>. Acesso em: 5 abr. 2003.

SOMMERVILLE, Ian. **Engenharia de Software**. Tradução André Maurício de Andrade Ribeiro, revisão técnica Kechi Hiramã. São Paulo: Addison Wesley, 2003.

SUN MICROSYSTEM. **Java blueprints guidelines, *patterns*, and code for end-to-end Java applications**, [S.l],[200?]. Disponível em: <<http://java.sun.com/blueprints/patterns/index.html>>. Acesso em: 10 nov. 2002.

TOTAL.COM S.A. **Manual de utilização do sistema Total Lite**, Blumenau, [2002?]. Disponível em: <<http://www.totall.com.br>>. Acesso em: 12 mar. 2003.

TALIGENT Inc. Building Object-Oriented Frameworks. **A Taligent White Paper**, [S.l],[200?], disponível em: <<http://www.ibm.com/java/education/oobuilding/index.html>>. Acessado em 10 maio de 2003.