

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/341327431>

Java and Design Patterns

Book · January 2016

CITATIONS

0

READS

668

1 author:



[Alvaro F. Pinheiro](#)

University of Pernambuco

29 PUBLICATIONS 5 CITATIONS

SEE PROFILE

Fundamentos da Engenharia de Software Java e Padrões de Projeto

PINHEIRO, Álvaro Farias
Autor

Fundamentos da Engenharia de Software

Java e Padrões de Projeto

Publicação 2015

O autor acredita que todas as informações aqui apresentadas estão corretas e podem ser utilizadas para qualquer fim legal. Entretanto, não existe qualquer garantia explícita ou implícita, de que o uso de tais informações conduzirá sempre ao resultado desejado. Os nomes de sites e empresas, por ventura, mencionados, foram utilizados apenas para ilustrar os exemplos, não tendo vínculo nenhum com o livro, não garantindo a sua existência nem divulgação. Eventuais erratas estarão disponíveis para download no site de publicação.

Algumas imagens utilizadas neste livro foram obtidas a partir da Internet.

Dados da Publicação

Pinheiro, Álvaro Farias

Fundamentos da Engenharia de Software: Java e Padrões de Projetos

1. ed. – Volume V

Recife, 2015

Selo Editorial: Publicação Independente

Bibliografia

ASIN B0174YQAUI

ISBN 9781520147130

1. Linguagem Java
2. Padrão Arquitetural MVC
3. Padrão de Projeto GRASP
4. Padrão de Projeto GoF

Fundamentos da Engenharia de Software Java e Padrões de Projeto

Publicação Independente
Livro em português com o título
Fundamentos da Engenharia de Software
Java e Padrões de Projeto
Volume 5
1ª Edição
Recife - Pernambuco – Brasil
Primeira revisão em Outubro de 2015
Segunda revisão em Junho de 2016

Fundamentos da Engenharia de Software Java e Padrões de Projeto

Para o meu amado filho Argel José Antas
Pinheiro que me trás energia.

Fundamentos da Engenharia de Software

Java e Padrões de Projeto

“Seja um padrão de qualidade. As pessoas não estão acostumadas a um ambiente onde o melhor é o esperado.”

Steve Jobs

SUMÁRIO

1 Paradigma Orientado a Objetos	1
1.1 Paradigmas de Programação	2
1.1.1 Paradigma Funcional	2
1.1.2 Paradigma Lógico	2
1.1.3 Paradigma Declarativo	3
1.1.4 Paradigma Imperativo	3
1.1.5 Paradigma Orientado a Eventos	3
1.1.6 Paradigma Orientado a Objetos	4
1.1.7 Paradigma Orientado a Aspectos.....	4
1.2 Linguagens de Programação.....	5
Projetos de Linguagens.....	7
1.3 Compilador	12
1.4 Interpretador	13
1.5 Máquina Virtual.....	13
1.6 Análises dos Compiladores	14
1.7 Princípios dos Paradigmas de Desenvolvimento.....	15
1.8 Princípios de Linguagens de Programação	17
1.9 Backus-Naur (BNF)	25
1.10 Conceitos Básicos do Paradigma Orientado a Objetos	25
Classe	25
Objeto.....	27
Herança.....	27
Polimorfismo	27
Sobreescreita (Override)	28
Sobrecarga (Overload).....	28
Encapsulamento.....	28
1.11 Exemplos.....	28
1.12 Ferramentas	29
1.13 Vídeos	29

SUMÁRIO

1.14 Sugestões de Leitura.....	29
1.15 Referências Web	29
1.16 Referências Bibliográficas	30
2 UML	31
2.1 Modelagem.....	31
2.2 Linguagem de Modelagem Unificada (UML)	32
2.2.1 Diagrama de Casos de Uso	34
2.2.2 Diagrama de Classes	39
2.2.3 Diagrama de Objetos	44
2.2.4 Diagrama de Sequência e Comunicação ou Colaboração	44
2.2.5 Diagrama de Atividades	46
2.2.6 Diagrama de Estados	47
2.2.7 Diagrama de Componentes	48
2.2.8 Diagrama de Distribuição	48
2.2.9 Diagrama de Tempo.....	49
2.3 Análise Orientada a Objetos.....	49
2.4 Modelagem Orientada a Objetos	52
2.5 Exemplos.....	59
2.6 Ferramentas	62
2.7 Apresentações (pptx)	62
2.8 Vídeos	62
2.9 Sugestões de Leitura.....	62
2.10 Referências Web	63
2.11 Referências Bibliográficas	64
3 Linguagem de Programação Java	66
3.1 Conceitos.....	66
3.2 Programação Orientada a Objetos com Java	68
3.2.1 Classes	69
3.2.2 Objetos.....	73

SUMÁRIO

3.2.3 Métodos	74
3.2.4 Pacotes	75
3.2.5 Relacionamentos.....	77
3.2.6 Polimorfismo.....	85
3.2.7 Super.....	90
3.2.8 Tratamento de Exceções	91
3.3 Tipos de Dados	94
3.3.1 Tipo Boolean	95
3.3.2 Tipos de dados inteiros:	97
3.3.3 Tipo Caractere.....	97
3.3.4 Tipos de Ponto Flutuante	98
3.3.5 Conversão de Dados.....	98
3.3.6 Operador instanceof.....	98
3.3.7 Operadores e Modificadores	101
3.3.8 Precedência	101
3.3.9 Diferença entre o operador de igualdade == e o método de comparação equals().....	102
3.4 Dicas.....	104
3.4.1 Usar a ferramenta Visual Class para desenhar formulários com Java.....	104
3.4.2 Usar a ferramenta JUDE <i>Community</i> para UML com Java	104
3.4.3 Instalar a ferramenta visual Green UML.....	105
3.4.4 Instalar a IDE Eclipse com <i>plugin</i> de modelagem	105
3.4.5 Instalar <i>plugin</i> de modelagem em outras versões do eclipse	105
3.4.6 Instalar UML2 <i>Tools</i>	105
3.4.7 Instalar eUML2 plugin para eclipse de modelagem.....	105
3.4.8 Instalar o Java Server Faces (JSF)	106
3.5 Exemplos.....	106
3.6 Atividades.....	124

SUMÁRIO

3.7 Ferramentas	127
3.8 Apresentações.....	128
3.9 Vídeos	128
3.10 Sugestões de Leitura.....	129
3.11 Referências Web	129
3.12 Referências Bibliográficas	129
4 Padrões de Projetos	130
4.1 Categorias dos Padrões de Projetos	131
4.2 Critérios dos Padrões de Projetos	131
4.3 <i>General Responsibility Assignment Software Patterns</i>	132
4.3.1 Creator Designer	132
4.3.2 Low Coupling.....	133
4.3.3 High Coesion.....	134
4.3.4 Polymorphism.....	135
4.3.5 Controller.....	136
4.3.6 Indirection.....	138
4.3.7 Protected Variations	139
4.3.8 Information Expert.....	140
4.3.9 Pure Fabrication	141
4.4 <i>Gang of Four (GoF)</i>	143
4.5 Padrão de Projeto GoF Criacional	145
4.5.1 Abstract Factory	145
4.5.2 Builder	146
4.5.3 Factory Method	148
4.5.4 Prototype.....	150
4.5.5 Singleton	151
4.6 Padrão de Projeto GoF Estrutural	152
4.6.1 Adapter.....	152
4.6.2 Bridge.....	153

SUMÁRIO

4.6.3 Composite	155
4.6.4 Decorator	156
4.6.5 Facade	158
4.6.6 Flyweight	159
4.6.7 Proxy	160
4.7 Padrão de Projeto GoF Comportamental	162
4.7.1 Chain of Responsibility	163
4.7.2 Template Method	164
4.7.3 Command	165
4.7.4 Strategy	167
4.7.5 State	168
4.7.6 Observer	169
4.7.7 Interpreter	171
4.7.8 Memento	172
4.7.9 Mediator	173
4.7.10 Visitor	174
4.7.11 Iterator	176
4.8 Padrão Arquitetural <i>Model View Control</i> (MVC)	177
4.9 Apresentações	177
4.10 Sugestões de Leitura	178
4.11 Referências Web	178
4.12 Referências Bibliográficas	178
5 Referências Bibliográficas	179
6 Referências Web	181
7 Biografias	183
8 Artefatos de <i>Software</i>	188
9 Ferramentas	190
10 Apresentações e Vídeos	192
11 Sugestões de Leitura	193

ILUSTRAÇÕES

Figura 1.1 Mapa Mental de Paradigmas de Desenvolvimento.....	2
Figura 1.2 Paradigmas.....	5
Figura 1.3 Linguagens.....	7
Figura 1.4 Compilador.....	13
Figura 1.5 Interpretador.....	13
Figura 1.6 Máquina Virtual.....	14
Figura 1.7 Conceitos Gerais de Paradigmas.....	20
Figura 1.8 Estrutura de Pilha.....	24
Figura 1.9 Estrutura de Fila.....	24
Figura 1.10 Símbolo de Classe.....	26
Figura 1.11 Símbolo de Objeto.....	27
Figura 2.1 Evolução da UML.....	33
Figura 2.2 Diagramas da UML.....	33
Figura 2.3 Elementos do Diagrama de Casos de Uso.....	34
Figura 2.4 Símbolos da UML.....	34
Figura 2.5 Diagrama de Caso de Uso.....	37
Figura 2.6 Diagrama de Casos de Uso com Inclusão.....	37
Figura 2.7 Diagrama de Casos de Uso com Extensão.....	37
Figura 2.8 Diagrama de Casos de Uso Realizar Matrícula.....	38
Figura 2.9 Elementos do Diagrama de Classes.....	40
Figura 2.10 Classe.....	40
Figura 2.11 Diagrama de Classes com Herança.....	41
Figura 2.12 Diagrama de Classes com Associação.....	41
Figura 2.13 Diagrama de Classes com Agregação.....	42
Figura 2.14 Diagrama de Classes com Composição.....	42
Figura 2.15 Diagrama de Classes com Realização.....	42
Figura 2.16 Diagrama de Classes com Dependência.....	43
Figura 2.17 Diagrama de Classes com Relacionamento.....	43
Figura 2.18 Diagrama de Classes com Comentário.....	44
Figura 2.19 Elementos do Diagrama de Sequência.....	44
Figura 2.20 Diagrama de Sequência.....	45
Figura 2.21 Diagrama de Comunicação.....	46
Figura 2.22 Elementos do Diagrama de Atividades.....	46
Figura 2.23 Diagrama de Atividades.....	46
Figura 2.24 Diagrama de Estados.....	47
Figura 2.25 Diagrama de Componentes.....	48
Figura 2.26 Diagrama de Distribuição.....	49
Figura 2.27 Especificação Resumida de Casos de Uso.....	60
Figura 2.28 Exemplo de Diagrama de Casos de Uso Consultar Cliente.....	61
Figura 2.29 Exemplo de Diagrama de Classes Consultar Cliente.....	61
Figura 2.30 Exemplo de Diagrama de Estados Consultar Cliente.....	61

ILUSTRAÇÕES

Figura 3.1 Programação Orientada a Objetos	66
Figura 3.2 Conceitos de Programação Orientada a Objetos com Java	68
Figura 3.3 Tipos de Métodos em POO	75
Figura 3.4 Relacionamento em Java	77
Figura 3.5 Polimorfismo	85
Figura 3.6 Exemplo de Relacionamentos na POO	87
Figura 3.7 Modificados em Java	104
Figura 3.8 Diagrama de Classes do Projeto Exemplo	106
Figura 4.1 GRASP <i>Creator</i>	133
Figura 4.2 GRASP <i>Low Coupling</i>	134
Figura 4.3 GRASP <i>High Coesion</i>	135
Figura 4.4 GRASP <i>Polymorphism</i>	136
Figura 4.5 GRASP <i>Controller</i>	137
Figura 4.6 GRASP <i>Indirection</i>	138
Figura 4.7 GRASP <i>Protected</i>	139
Figura 4.8 GRASP <i>Information</i>	140
Figura 4.9 GRASP <i>Fabrication</i>	142
Figura 4.10 Padrões GoF	143
Figura 4.11 GoF <i>Abstract Factory</i>	146
Figura 4.12 GoF <i>Builder</i>	147
Figura 4.13 GoF <i>Factory Method</i>	148
Figura 4.14 GoF <i>Prototype</i>	150
Figura 4.15 GoF <i>Adpater</i>	153
Figura 4.16 GoF <i>Bridge</i>	154
Figura 4.17 GoF <i>Composite</i>	155
Figura 4.18 GoF <i>Decorator</i>	156
Figura 4.19 GoF <i>Facade</i>	159
Figura 4.20 GoF <i>Flyweight</i>	160
Figura 4.21 GoF <i>Proxy</i>	161
Figura 4.22 GoF <i>Chain Responsibility</i>	163
Figura 4.23 GoF <i>Template Method</i>	164
Figura 4.24 GoF <i>Command</i>	166
Figura 4.25 GoF <i>Strategy</i>	167
Figura 4.26 GoF <i>Observer</i>	170
Figura 4.27 GoF <i>Interpreter</i>	172
Figura 4.28 GoF <i>Mediator</i>	173
Figura 4.29 GoF <i>Visitor</i>	175

SIGLAS e ACRÔNIMOS

JDBC - *Java DataBase Connectivity*
JSP - *Java Server Pages*
MVC - *Model–View–Controller*
OMG - *Object Management Group*
OOP - *Object Ooriented Programming*
UML - *Unified Modeling Language*

NOTAÇÕES

Este livre adota as seguintes notações, que devem ser observadas para o perfeito entendimento do material aqui produzido.

Notações Técnicas:

<..> - Menor que / Maior que, nos comandos SQL usado para metadados.

<..> - Menor / Maior, no texto indica que é um *hyperlink* (*link* para um endereço web).

[..] - Abre e fecha parênteses, no texto indica utilização de uma sigla.

(..) - Abre e fecha colchetes, nos comandos SQL indica opcionalidade.

| - Barra, para indicar “ou lógico”.

 - Sublinhado, nos comandos SQL indica o *default* (padrão).

 - Sublinhado, no texto indica que é um *hyperlink* (*link* para um endereço web).

Negritos - Títulos de capítulos, seções e subseções.

Itálico - Para palavras de língua estrangeira.

SOBRE O AUTOR



Álvaro Farias Pinheiro tem mais de 30 anos de experiência no campo da computação e nestes, mais de 15 anos na área acadêmica, lecionando disciplinas com foco em engenharia de software.

É mestre em Engenharia de Software pelo Centro de Estudos e Sistemas Avançados do Recife (CESAR) com a dissertação *Uso do Balanced Scorecard na Avaliação de Fábricas de Software* utilizadas como Instrumento de Ensino de Tecnologia da Informação Baseado em Problemas Reais. Especialista em Gestão Pública pela Universidade de Pernambuco (UPE)

e em Metodologias de Desenvolvimento pela União Brasileira de Tecnologia. É Bacharel em Sistemas de Informação pela Faculdade Integrada do Recife (FIR), iniciando sua vida acadêmica no Bacharelado em Ciência da Computação da Universidade Católica de Pernambuco (UNICAP).

Iniciou as atividades profissionais em 1986 como Programador na Elógica Microsistemas desenvolvendo para baixas e altas plataformas nas linguagens dBase, Clipper, FoxPro, Cobol e Natural. Como Analista de Sistemas passou pelas instituições: Centro de Prestação de Serviços Técnicos de Pernambuco (CETEPE), Empresa de Fomento da Informática do Estado de Pernambuco (FISEPE), Instituto de Tecnologia em Informática (ITECI), Apply Solutions e está hoje na Agência Estadual de Tecnologia da Informação do Estado de Pernambuco (ATI-PE) por concurso público, trabalhando como Analista em Gestão da Tecnologia da Informação e Comunicação, em exercício na Procuradoria Geral do Estado de Pernambuco (PGE-PE), atuando como Coordenador da Área de Sistemas.

Começou a lecionar em 1992 em cursos técnicos nas empresas ITECI e IBRATEC, e desde 2006 é professor de cursos de graduação e pós-graduação em computação nas instituições de ensino superior da região, a citar, Universidade Federal Rural de Pernambuco (UFRPE), Instituto Federal de Pernambuco (IFPE), Faculdade Estácio do Recife, Faculdade Integrada do Recife (FIR), Faculdade Joaquim Nabuco, Faculdade do Recife, Unibratrec e durante esses anos realizou diversos cursos de extensão como os de Análise de Ponto de Função, Administração de Redes, Linguagens de Programação Natural, Centura, Delphi e CSharp, Banco de Dados Adabas, MySQL, Postgre e Oracle, Análise Estruturada e Orientada a Objetos, entre outros.

SOBRE O AUTOR

Possui experiência na área de Engenharia de *Software* nas disciplinas de Teoria da Computação, Sistemas de Informação e Colaborativos, Desenvolvimento Estrutural e Orientado a Objetos para aplicações Desktop, Web e Mobile, Processos de Desenvolvimento IRUP, XP e Scrum, Banco de Dados SQLServer e Oracle, e Gestão de Projetos com PMBOK e SWEBOK. Trabalha atualmente na coordenação de projetos e com educação presencial e a distância, ministrando aulas e realizando pesquisas em ambientes virtuais de ensino e linguagens de programação.

<http://www.alvarofpinheiro.eti.br>

<http://br.linkedin.com/in/alvarofpinheiro>

<https://sites.google.com/site/alvarofpinheiro>

<http://www.facebook.com/alvarofpinheiroaulas>

PREFÁCIO

A motivação deste livro foi a dificuldade do entendimento sobre a disciplina Padrões de Projetos com Java por parte dos alunos que ao longo dos anos estudaram essa matéria nas aulas por mim ministradas. Assim a construção desse livro é a compilação das aulas produzidas com o intuito de facilitar o entendimento do assunto.

O livro aborda os seguintes temas:

- **Capítulo 1** *Paradigma Orientado a Objetos*. Introdução ao paradigma orientado a objetos com os seus principais conceitos.
- **Capítulo 2** *Linguagem de Modelagem Unificada*. Apresentação da linguagem de modelagem UML para diagramar projetos de *software*.
- **Capítulo 3** *Linguagem de Programação Java*. Introdução a linguagem de programação orientada a objetos Java abordando os seus principais recursos.
- **Capítulo 4** *Padrões de Projetos*. Apresentação dos principais padrões de projeto para o paradigma orientado a objetos.

Finalizando, este livro possui várias subseções que disponibilizam *hyperlink's* para *wiki's* sobre os assuntos abordados, como também *link's* para outros *ebook's* utilizados como referencial bibliográfico do conteúdo produzido e *links* para os sites oficiais dos temas abordados.

Álvaro Farias Pinheiro

1 Paradigma Orientado a Objetos

Neste capítulo será apresentado o paradigma de desenvolvimento orientado a objetos com os seus conceitos e benefícios.

Nos dicionários, paradigma significa um sistema de regras e regulamentos, que podem ser escritos ou não, e que servem para estabelecer fronteiras ou limites, determina como se comportar dentro dos limites para alcançar o sucesso. Simplificando, uma forma de pensar e agir.

Agora, um paradigma de desenvolvimento de *software* serve para fornecer uma visão sobre as estruturas da programação, isto é, as formas de se desenvolver um *software*. E para isso existem vários paradigmas, por exemplo, o imperativo, o lógico, o funcional, o orientado a objetos, o orientado a aspectos, entre outros.

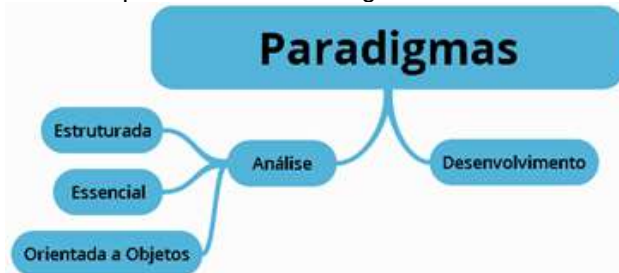
Dessa forma é possível se estabelecer um padrão de linguagem para um determinado nicho de problemas, por isso existem diversos paradigmas, além-claro, da própria evolução da ciência da computação, que de tempos em tempos, através de pesquisas, disponibilizam novas metodologias e paradigmas de programação.

Então se entende que certos grupos de linguagens foram e são desenvolvidas para determinado paradigma. Temos por exemplo Smalltalk para o Paradigma de Orientação a Objetos (POO), Scheme para o paradigma funcional. Mas, existem linguagens que suportam vários paradigmas. CSharp suporta tanto o POO como o Funcional, já o Java suporta o POO e o Aspecto, entre outras linguagens como Perl, Python, C++, etc.

A característica marcante dos paradigmas de programação são as restrições, isto é, o que pode ou não pode ser feito na linguagem. Por exemplo, a programação estruturada não aceita o comando de indireção *goto*, que é aceita no linear.

Para dar uma ideia do que veremos nessa disciplina segue na figura 2.1 um mapa mental que mostra sucintamente os vários tópicos abordados nessa disciplina.

Figura 1.1 Mapa Mental de Paradigmas de Desenvolvimento



Fonte: Próprio Autor

1.1 Paradigmas de Programação

Vamos aqui falar sobre alguns paradigmas. Entendido que as linguagens são criadas para atender os padrões definidos de um ou mais paradigmas, quais são as linguagens mais atuais e que paradigmas elas se baseiam.

1.1.1 Paradigma Funcional

É usado para criar programas cujo foco está na avaliação de funções matemáticas. Nesse paradigma existe a preocupação de se evitar estados ou dados mutáveis. Basicamente, a programação nas linguagens desse paradigma consiste em usar e criar funções, por exemplo: $f(x) = x + 1$, onde x é um parâmetro, e esse é um valor de entrada, que após ser processada a expressão matemática será fornecido o valor de saída da função. Vale salientar que as linguagens que atendem esse paradigma, são usadas na sua maioria no mundo acadêmico e de pesquisa, com pouquíssima aplicação comercial. Alguns exemplos de linguagens são: Lisp, Mathematica, Haskell e uma das mais recentes, encontrada na IDE Visual Studio Ultimate 2010 é o F#.

1.1.2 Paradigma Lógico

Também conhecido como Restritivo. De certa forma se assemelha ao funcional, pois o foco está na matemática, mais especificadamente na lógica matemática. As linguagens que fazem uso desse paradigma são orientadas a procedimentos, mas essas possuem limitações, tipo o armazenamento dos resultados dos procedimentos. Outra característica, tanto da lógica, como da funcional, é que elas não são uma sequência de afirmações e não possuem um estado global. Uma forma de entender as linguagens desses dois paradigmas, é que elas, normalmente definem o que é para ser processado,

mas não como processá-lo. A linguagem mais representativa desse paradigma é o Prolog, mas como o padrão anterior, esse é muito pouco usado para aplicações comerciais, tendo um uso maior no mundo científico.

1.1.3 Paradigma Declarativo

Esse paradigma é baseado nos paradigmas funcional e lógico. Ele foi criado para determinar a fronteira entre os paradigmas já falados do imperativo, que será explicado a seguir. Mas, podemos dizer que as linguagens declarativas, são aquelas que se descreve o que ela faz e não como seus procedimentos funcionam. Então, podemos entender que as linguagens de marcação, as que usam *tags*, são declarativas. Pois, elas descrevem o que são suas estruturas e não como elas são utilizadas. Um bom exemplo de linguagem declarativa é o HTML, outro exemplo muito utilizado é o SQL e mais recentemente o XML.

1.1.4 Paradigma Imperativo

É também denominado de modular ou procedural. Para entendê-lo, basta entender imperativo como comando, isto é, baseado em ações, que contrário aos três paradigmas anteriores, é focado na mudança de estado das variáveis. Exemplo: `if (vsexo == "F") { printf("Feminino"); }`. Observe que só haverá saída se o sexo for feminino, um comando. Ao contrário dos paradigmas anteriores, esse normalmente define como processar algo, mas não para que processá-lo, isso, dependerá da utilização do programa. As linguagens mais representativas desse paradigma são o C, Basic e o Pascal, porém uma muito utilizada no passado recente foi a Clipper e toda a linha das XBase.

1.1.5 Paradigma Orientado a Eventos

A maior característica desse paradigma é que no ato de programar não se determina o fluxo do controle, ficando a cargo do usuário do *software* desenvolvido a sua condução. Explicando melhor, nas linguagens do paradigma imperativo, tipo Clipper, Pascal,... O programador definia exatamente como a navegação iria acontecer. Para acessar tal opção, primeiro ir para o menu tal, depois submenu tal, e assim por diante. Já nas linguagens orientadas a eventos, o fluxo não era criado e sim as opções de acesso, deixando a cargo de o usuário acessar as opções, quando e como quiser, isto é, controle de fluxo de programas é definido por ocorrências externas ao *software*, e essas ocorrências são chamadas de eventos. Exemplo: um clique posiciona o cursor, um duplo clique seleciona uma palavra, um triplo clique seleciona um parágrafo, etc. Assim, o programador, define as ações, baseado nos eventos, ou melhor, quando elas devem acontecer. Mas, é o usuário que inicia o evento. Basicamente, toda a

linguagem que faz uso de interface gráfica é baseada nesse paradigma. Como exemplo temos as linguagens Visual Basic, Delphi, FoxPro, SQLWindows, entre tantas outras.

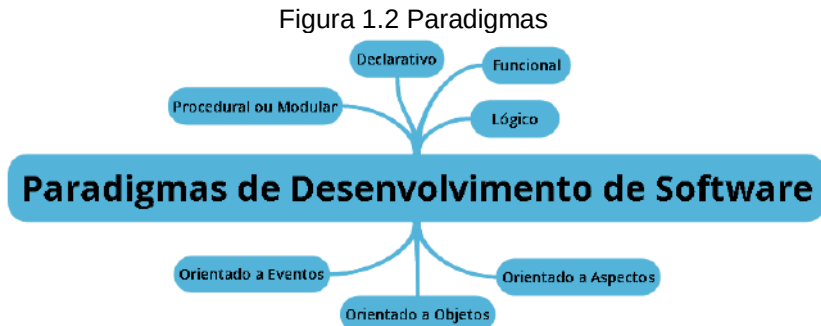
1.1.6 Paradigma Orientado a Objetos

Também conhecida como POO, teve a sua motivação de criação e evolução a partir do paradigma Orientado a Eventos e se deve a necessidade da área de engenharia de *software* realizar a manutenção corretiva, adaptativa e evolutiva de forma mais rápida em relação aos paradigmas anteriores. Focado na criação de classes e na instância de objetos dessas classes, a construção do *software* passa a ser mais criteriosa, pensando na possibilidade de reuso, isto é, reutilizar código pronto e testado. Além, de fazer as correções e implementações, de forma mais organizada e eficiente. Uma grande vantagem desse paradigma, é que ele une as três áreas básicas do desenvolvimento de aplicações, as quais são análise, programação e banco de dados. Permitindo que a mesma simbologia possa ser usada nessas três áreas. Nos dias atuais, quase todas as linguagens se baseiam nesse paradigma, umas criadas, outras adaptadas para ele. Como exemplos têm: C++, C#, Java, Object Pascal, Delphi, Delphi. NET, VB.NET, Python, Ruby e Smalltalk. Além do PHP a partir da versão quatro, do Perl a partir da versão cinco.

1.1.7 Paradigma Orientado a Aspectos

Também conhecida como POA, é um paradigma que permite a separação e organização do código de acordo com a sua importância para a aplicação, isso, passou a ser chamado de costurar o código. De certa forma, todas as aplicações desenvolvidas usando o paradigma POO, isto é, que possuem implementação de comportamento de um objeto, os chamados métodos, possuem um problema. Se não houver um cuidado na construção desse código, o mesmo pode ficar espalhado por toda a aplicação. É essa falha da POO que a POA entra, com o objetivo de resolvê-la. Permitindo que esse código seja encapsulado e modularizado. Para entender melhor, imagine um sistema escrito em Pascal, onde em várias partes do sistema é necessário fazer o registro da data, hora e usuário que acessou a rotina. Bom, em qualquer linguagem, seria criado um programa central que faria esse registro e onde fosse preciso, se criaria uma chamada para essa rotina. O problema que o uso dessa técnica, espalha a chamada da rotina, por todo o sistema. A proposta do POA é encapsular essas chamadas através de uma construção chamada de Aspect, onde esse se responsabilizaria pelo comportamento da aplicação que tem um comportamento adicional o chamado de *advice*, sobre um *joinpoint*. Atualmente a linguagem desse paradigma é o AspectJ, uma

versão Java para a orientação a aspecto. Na figura 2.2 pode ser observar os nomes dos principais paradigmas de desenvolvimento de software.



Fonte: Próprio Autor

1.2 Linguagens de Programação

Vamos agora fornecer alguns conceitos básicos sobre as linguagens de programação, que tem como objetivo escrever uma lógica que seja reconhecida por uma máquina para executar algum conjunto de comandos produzindo um resultado.

Existem três elementos importantes que devem ser entendidos em qualquer linguagem - a sintaxe, a semântica e a simbologia (o léxico), que podem ser definidos como:

- Sintaxe que vem do grego e quer dizer disposição consiste nas regras que constituem a formação das sentenças nas linguagens de programação (o que significa algo);
- Semântica que também vem do grego e tem a significação de sinal, trata do significado, (como expressa algo); e
- Léxico determina o conjunto de símbolos de uma linguagem (que simboliza algo).

Agora, sem dúvida, um dos itens mais usados, são os identificadores ou variáveis, que objetivam armazenar na memória um valor ou expressão.

Para se armazenar em uma variável, quando temos uma linguagem tipada, temos que definir os tipos de dados, que é uma combinação de valores que podem ser armazenados em um identificador e dependem da análise léxica, sintática e semântica do compilador ou interpretador da linguagem.

Entendido isso, pode-se perguntar para que segmentos as linguagens são construídas? Seguem alguns:

- Inteligência Artificial é um dos mais recentes e tem por objetivo descobrir *software* e *hardware* que possam simular a capacidade humana de pensar;
- Educação que objetiva auxiliar no processo cognitivo;
- Ciência e Engenharia que tem como objetivo criar meios que otimizem processos de criação e construção;
- Sistemas de Informação onde seu uso é mais abrangente, pois tem o foco de auxiliar as organizações se tornarem mais eficientes e eficazes;
- Redes objetivando a convergência dos recursos tecnológicos; e
- Internet permitir a colaboração de informações e atividades.

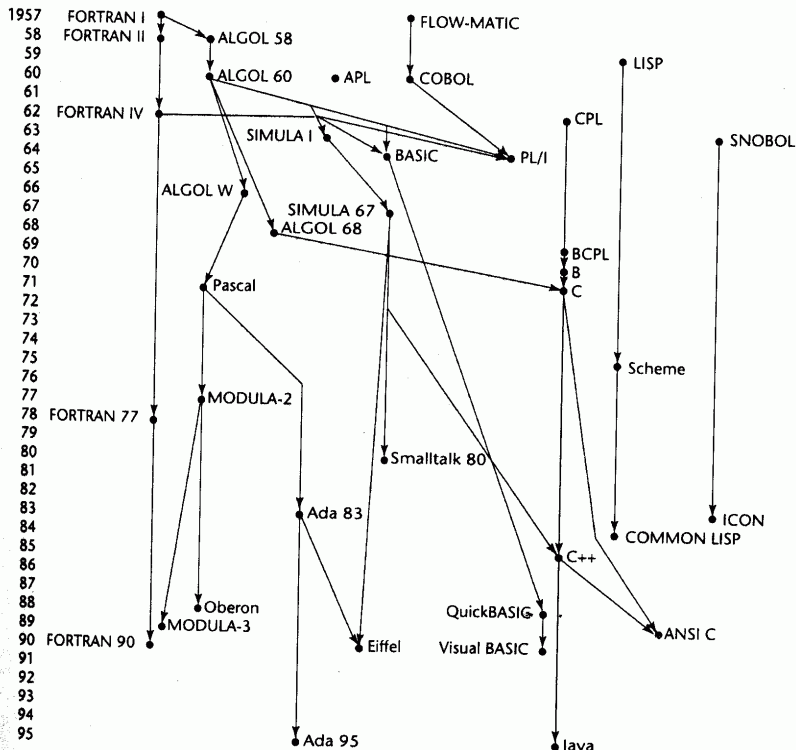
Bom, até aqui vimos o "b-a-ba" das linguagens, mas como elas evoluíram? Na figura 2.3, existe um resumo dessa evolução compreendendo o período de 1957 até 1995.

Agora, clique na [árvore genealógica das linguagens](#) para ter uma ideia mais completa de quando e de onde as principais surgiram. No site <http://www.levenez.com/lang/> é possível ter um detalhamento dessa evolução.

Como vocês podem observar são muitas linguagens. E dessas quais são as mais populares? Para responder essa pergunta, estou disponibilizando o resultado de uma pesquisa entre várias já realizadas sobre o tema. Essa foi realizada pela TIOBE que é uma organização que oferece uma avaliação periódica sobre qualidade de *software*. Assim esse ranking pode ser usado para se conhecer algumas das linguagens mais populares do momento. Esse outro site ohloh possibilita fazer uma comparação entre as principais linguagens. Mas, para ser mais justo, segue outro ranking com as

linguagens mais populares, realizado pelo site LangPop que divulgou mundialmente essa pesquisa na qual analisou a popularidade das principais linguagens de programação existentes no mercado, utilizando para isso o parâmetro de buscas realizadas nos principais sites de buscas da época como Google, Yahoo, Amazon, entre outros.

Figura 1.3 Linguagens



Fonte: <http://www.levenez.com/lang/>

Projetos de Linguagens

Como já vimos anteriormente neste livro os conceitos de projetos de linguagem, no capítulo anterior, seguem mais algumas informações.

Existe algumas características básicas necessárias serem trabalhadas para se projetar uma linguagem, essas são: a sua arquitetura, a configuração técnica, os seus padrões e a compatibilidade com o legado.

Então vamos entender basicamente o que é arquitetura de uma linguagem. Podemos dizer que, projetar uma linguagem é levar em consideração seu *hardware*, e o modelo que o define, o já citado modelo de von Neumann, trata-se do modelo, entrada-processamento-saída. Então, boa parte, das linguagens é projetada para essa arquitetura de máquina, mas existem outras.

Detalhando um pouco mais a questão de arquitetura, temos que as linguagens são projetadas para atendê-las, e nos dias atuais, são dois os projetos básicos de arquiteturas:

- *Complex Instruction Set Computer* (CISC), traduzindo Computador com um Conjunto Complexo de Instruções. Explicando basicamente o que é, trata-se de um padrão de construção de processadores capazes de executar centenas de instruções complexas. Mas, o que quer dizer isso? Uma forma simples de explicar, que dizer que. No processador, são armazenadas instruções chamadas de microcódigo, que se não o existissem, teriam que ser codificadas pelos programadores nas linguagens. Esse armazenamento de instruções nos processadores minimiza e simplifica o trabalho de desenvolvimento, já que muito código comum a qualquer tipo de programa já está embutido no processador. Como consequência direta as linguagens concebidas para essa arquitetura, tendem a ter um código executável bem menor. Mas, a sua desvantagem é que como a microprogramação realizada visa prever uma grande quantidade de funcionalidades comuns aos aplicativos, torna esse tipo de processador pesado em relação ao tempo de resposta. E para completar com o passar do tempo, notou-se que muitas dessas funcionalidades embutidas nos microcódigos nunca eram usadas na maioria das aplicações desenvolvidas. Um exemplo de padrão de processador CISC é o Intel 486. Resumindo temos: melhor código com pior desempenho; e
- *Reduced Instruction Set Computer* (RISC), traduzindo Computador com um Conjunto Reduzido de Instruções. Trata-se de outro padrão de construção de processadores, que como a própria sigla já diz, oferece um conjunto simples e pequeno de instruções armazenadas no processador. Simplificando, os processadores desse padrão não

possuem ou possuem muito pouca microcódigo. Consequência direta é que o esforço de desenvolvimento torna-se maior. Porém, como os processadores não possuem microcódigo desnecessário, prevendo funcionalidades comuns a diversos tipos de programas, que normalmente não são usadas, o tempo de resposta é muito superior. Grande parte dos projetos de processadores atuais segue esse modelo de arquitetura, a exemplo temos: DEC Alpha, o SPARC, o MIPS, o PowerPC, o os atuais x86 da Intel que usam uma arquitetura híbrida, meio CISC e meio RISC. Outro exemplo são as máquinas construídas para tipos específicos de aplicações como as de computação gráfica e de jogos. Resumindo temos: melhor desempenho com pior código.

Outra preocupação no projeto de uma linguagem de programação é em relação ao nicho que ela pretende atender, para resolver os seus problemas. Esse objetivo gera uma série de restrições, de natureza técnica ou de modelo, que são normalmente observados, durante o projeto de uma linguagem de programação. Essas restrições tecnológicas estão relacionadas aos níveis de abstração.

Também existe a preocupação no projeto de desenvolvimento de uma nova linguagem é a sua compatibilidade com os seus projetos anteriores, caso existam. Isso se deve pela necessidade de se poderem manter os códigos produzidos pelas versões anteriores com a nova linguagem. Uma tarefa muito complicada. Mas, a preocupação com o legado na área de TIC é uma constante.

Depois, que uma linguagem se torna bem aceita pelo mercado, mais uma preocupação se passa a ter que é a sua padronização. Essa padronização é necessária, para uniformizar o uso e a evolução da linguagem. Normalmente as entidades que formalizam a padronização são a ANSI e o ISO, veja os *Links* para saber mais.

Uma categorização importantíssima nos projetos de linguagens de programação é a Hierarquia de Chomsky, que classifica as linguagens em tipos, os quais podem ser vistos abaixo: Tipo 3: Gramática Regular > Linguagem Regular > Autômato Finito; Tipo 2: Gramática Livre de Contexto > Linguagem LC > Autômato com Pilha; Tipo 1: Gramática Sensível ao Contexto > Linguagem SC > Autômato Linear; e Tipo 0: Gramática Irrestrita > Linguagem Recursiva > Máquina de Turing.

No capítulo Teoria Geral da Computação já esses tipos já foram conceituados, mas para revisar os conceitos, vamos falar sucintamente sobre eles. Começando com a explicação de Linguagens Formais e Autômatos. Basicamente é o estudo de modelos matemáticos que permitam definir e reconhecer linguagens, possibilitando a sua classificação, estruturação e caracterização. É conhecida como Teoria da Ciência da Computação. A finalidade do estudo dessas teorias é fornecer fundamentos para a Ciência da Computação em relação à decidibilidade, computabilidade e complexidade computacional. Essas teorias fundamentam várias áreas da computação, como processamento de linguagens, reconhecimento de padrões, modelagem de sistemas.

Agora o que é Linguagem Formal? Sabemos que linguagem é uma forma de comunicação e que com a evolução do homem, essas ganharam especializações, do tipo léxico, sintático e semântico. Essas especializações servem tanto para linguagens naturais ou idiomas, como também para linguagens de programação e dos protocolos de comunicação.

Assim, podemos dizer que as Linguagens Formais consistem de regras para representação e especificação de linguagens. As representações são feitas por reconhecedores e geradores de linguagens, que nada mais é, que um recurso que verificam se uma sentença pertence ou não a uma determinada linguagem. Esses recursos são chamados de autômatos, que podem ser: Autômatos Finitos Determinísticos; Autômatos Finitos Não Determinísticos; Autômatos de Pilha; e Máquina de Turing.

Já os geradores são recursos que permitem a geração sistemática de todas as sentenças de uma linguagem. Esses recursos são as gramáticas, a qual se destaca a de Chomsky. Resumindo, as linguagens formais podem ser representadas de maneira finita e precisa através de teorias matemáticas, o que caracteriza a Ciência da Computação.

Resumindo o Tipo 3, o das linguagens regulares, que podem ser representados pelos autômatos finitos determinísticos e não determinísticos. Essas linguagens se baseiam nas Expressões Regulares. Essas expressões são representadas por três formas básicas: União, representada pelo símbolo (+) permitindo unir expressões; Concatenação, representada pelo símbolo (.) permitindo concatenar expressões; e Klensee, representada pelo símbolo (*) permitindo potencializar expressões.

Essas expressões podem ser representadas por Autômatos. E o que são? São máquinas de estados finitos, isto é, um autômato finito. Trata de uma modelagem de um comportamento, composto por estados, transições e ações. Um estado armazena informações sobre o passado, isto é, ele reflete as mudanças desde a entrada num estado, no início do sistema, até o momento presente. Uma transição indica uma mudança de estado e é descrita por uma condição que precisa ser realizada para que a transição ocorra. Uma ação é a descrição de uma atividade que deve ser realizada num determinado momento. Máquinas de estados finitos podem ser representadas por meio de um diagrama de estados ou diagrama de transição de estados.

Resumindo o Tipo 2, o das linguagens livre de contexto, essas são representadas por autômatos com pilhas. Que são idênticos aos autômatos finitos determinísticos, exceto pelo fato eles adicionalmente usam o recurso de memória sob a forma de pilhas. A função de transição δ irá depender agora dos símbolos que estão no topo da pilha, e especificarão como a pilha deve ser modificada a cada transição.

Resumindo o Tipo 1, o das linguagens sensíveis ao contexto, essas são representadas por uma forma restrita de uma máquina de Turing não determinística que satisfaça as duas condições seguintes: Seu alfabeto de entrada inclui dois símbolos especiais, servindo como marcadores de direção da esquerda e direita; e Necessitam de um ponteiro que se move para a esquerda levando além do marcador.

Resumindo o Tipo 0, o das linguagens recursivas, essas são representadas por máquinas como as de Turing, Moore e Mealy. A Máquina de Turing é um recurso teórico conhecido também como máquina universal, que foi concebido pelo matemático britânico Alan Turing, antes do surgimento dos modernos computadores digitais. É um modelo abstrato de computador, que trata apenas dos aspectos lógicos de funcionamento da memória, estados e transições. Porém com essa abstração pode-se modelar qualquer computador digital. A Máquina de Moore é outro recurso que utiliza apenas ações de entrada e a saída depende somente do estado. A vantagem do modelo de Moore é a simplificação do comportamento. Por exemplo, modelar um elevador com quatro estados possíveis para sua porta "Aberta", "Fechada", "Abrindo", "Fechando". A máquina de estados reconhece dois comandos: "comando_abrir" e "comando_fechar" que disparam a alteração de estado. A ação de entrada no estado "Abrindo" liga o motor que abre a porta, a ação de entrada no estado "Fechando" liga o motor na outra direção,

fechando a porta. Os estados "Aberta" e "Fechada" não desempenham nenhuma ação. E a Máquina de Mealy é mais um recurso onde a saída depende da entrada e do estado. O uso de uma máquina Mealy normalmente leva a uma redução no número de estados. O exemplo anterior nessa máquina seria: inicie o motor para fechar a porta se o "comando_fechar" chegar e inicie o motor na direção oposta para abrir a porta se o "comando_abrir" chegar.

Finalizando, os principais objetivos de um projeto de linguagem de programação são a obtenção da: Simplicidade; Legibilidade; Clareza; Confiabilidade; Suporte; Abstração; Ortogonalidade, e Eficiência.

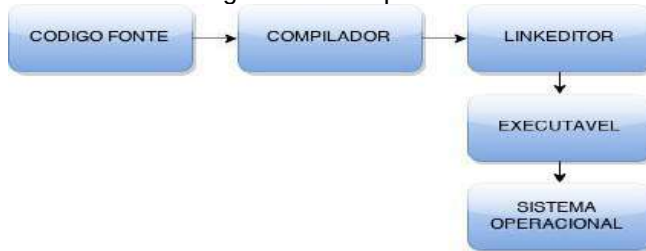
1.3 Compilador

O compilador é um programa que foi projetado para ler outro programa chamado de código fonte, escrito em uma linguagem específica para um compilador chamado de linguagem compilada.

A finalidade do compilador é gerar um terceiro programa semanticamente equivalente ao código fonte chamado de código objeto, que normalmente será o equivalente a linguagem de alto nível utilizada na sua versão em linguagem Assembly ou de máquina.

Durante a compilação são realizados vários processos chamados de análise e síntese. Na análise tem-se o objetivo de compreender o código fonte escrito, através das verificações léxica, sintática e semântica. Na síntese é gerado o código objeto, isto é, o código intermediário. Basicamente o processo é o seguinte: compila-se o código fonte e é realizada a análise léxica dessa é gerado o código simbólico (*tokens*), então se passa para o próximo passo que é a análise sintática e dessa é gerado o código analisado (parse), depois a análise semântica na qual é gerado o código qualificado, para daí ser gerado o código objeto, esse por fim é otimizado para gerar o código final.

Figura 1.4 Compilador



Fonte: Próprio Autor

Um programa que funciona como complemento do compilador é o Linkeditor, que possui a tarefa de linkar (unir) vários programas compilados em um programa chamado de executável. Permitindo assim que o programa rode no sistema operacional para qual o compilador / Linkeditor foi feito.

[Clique aqui para visualizar uma lista de linguagens compiladas.](#)

1.4 Interpretador

O Interpretador é um programa projetado para ler um código fonte escrito em uma linguagem específica para um interpretador chamado de linguagem interpretada.

O objetivo é ler linha a linha e converte em código objeto à medida que vai executando o programa, dessa forma o interpretador executa o código fonte linha a linha.

Figura 1.5 Interpretador



Fonte: Próprio Autor

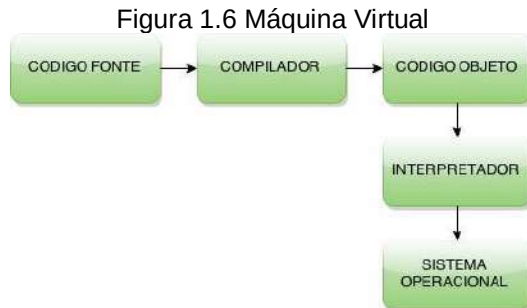
[Clique aqui para visualizar uma lista de linguagens interpretadas.](#)

1.5 Máquina Virtual

Nos dias atuais existem compiladores que geram código para o que é chamada máquina virtual, isto é, um misto de compilador e interpretador. As linguagens que se encaixam nessa categoria podem ser chamadas de

linguagens híbridas. Isso quer dizer que existem linguagens que são parte compilada e parte interpretada. Assim o compilador tem a função de converter o código fonte em um código intermediário chamado de *bytecode*, que nada mais é, do que uma linguagem de baixo nível e este será interpretada por uma máquina virtual, que funciona como um interpretador.

A principal diferença entre uma linguagem compilada para uma interpretada e essas duas para um híbrida é que na primeira todo o código fonte é lido pelo compilador para gerar de uma única vez um código final. Na segunda, a interpretada, o código fonte é lido linha a linha e o interpretador executa um comando por vez. E na última, temos um pouco de cada, já que o compilador analisa todo o código fonte, mas não gera o código final e sim um intermediário chamado de código objeto, e esse por sua vez é interpretado pela máquina virtual.



Fonte: Próprio Autor

1.6 Análises dos Compiladores

A análise léxica consiste de um processo de verificação das entradas dos caracteres com o objetivo de gerar uma sequência simbólica chamada de símbolos léxicos ou *lexicais tokens*, para que possam ser manipulados pelo analisador sintático chamado de *parser* que tem a finalidade de verificar as saídas. Resumindo a análise léxica tem a função de verificar um determinado alfabeto. Exemplo, quando analisamos uma palavra, pode definir através da análise léxica se existe ou não algum caractere que não faz parte do nosso alfabeto.

A análise sintática, mais conhecida como *parsing*, é o processo de verificar uma sequência de símbolos para determinar se sua estrutura gramatical segue as regras gramaticais definidas pela linguagem. Resumindo a análise

sintática transforma um texto na entrada em uma estrutura geralmente em forma de árvore, para que fique mais fácil o entendimento e o processamento, na verificação da hierarquia da linguagem. A grande maioria dos analisadores sintáticos utiliza alguma linguagem livre de contexto para fazer a análise e esses analisadores são classificados como *Left-Left* (LL) ou *Left-Right* (LR). O LL é um algoritmo de análise sintática que realiza a verificação no sentido cima para baixo ou *top-down*, já o LR faz a verificação de baixo para cima ou *bottom-up*.

Análise semântica tem a finalidade de verificar os erros semânticos. Exemplo, uma operação de soma entre operandos de tipos diferentes com um numérico e outro literal.

[Clique aqui para visualizar a especificação da Máquina Virtual Java.](#)
[Máquina Virtual \(wiki\).](#)

[Clique aqui para visualizar a especificação do .NET Framework.](#)
[DOT NET \(wiki\).](#)

1.7 Princípios dos Paradigmas de Desenvolvimento

As gramáticas que compõem as linguagens de programação são: léxica, sintática e semântica. Que respondem respectivamente por: analisar os símbolos, a escrita e o sentido das palavras de uma linguagem de programação.

As linguagens possuem o recurso básico de nomes de variáveis (identificadores) e tipos de dados. Respectivamente servem para denominar um local de memória que irá armazenar um dado e esse por sua vez deve possuir um tipo específico: número, literal, etc.

Os como já visto os paradigmas de desenvolvimento são padrões que determinam regras de construção de linguagens, tipo orientado a objetos, imperativos, etc.

Três conceitos especiais das linguagens de programação são: Eventos; Concorrência; e Correção. Que correspondem respectivamente por: Ações que ocorreram em um determinado momento; Quando ações podem ocorrer ao mesmo tempo de forma assíncrona; e Trata-se de desenvolver focado na qualidade, isto é, a manutenção corretiva e/ou evolutiva que tem o objetivo de gerar o menos impacto na construção do produto de *software*.

Existem diversos segmentos que os paradigmas de desenvolvimento são pensados para a área de desenvolvimento de *software*, mas podemos salientar os seguintes:

- Inteligência Artificial (IA) que tem como objetivo desenvolver *software* que possa aprender e assim evoluir por inferência, e uma forma de exemplificar essa lógica são com o conceito de redes neurais, que fazem uso do algoritmo MLP;
- Educação (EaD), nos dias atuais, ou melhor, na era da informação, um dos grandes problemas está no tempo, isto é, falta de tempo, nesse ponto a TIC também contribui possibilitando o uso de sistemas cooperativos ou colaborativos para disponibilizar conhecimento para quem não tem tempo para acessá-lo de forma tradicional, e o Ensino a Distância tem essa característica ;
- Ciência e Engenharia, inevitavelmente são nessa área que o emprego de TIC é mais evidente, muito devido a sua própria natureza científica;
- Sistemas de Informação (SI), é o emprego da TIC nas organizações buscando torná-las mais eficientes, eficazes e assim obter sua efetividade;
- Sistemas e Redes têm seu foco na tecnologia, possibilitando o grau de compartilhamento existente no mundo moderno; e
- Internet que é o foco do desenvolvimento atual, pois permite que o conhecimento e as necessidades sejam compartilhados e atendidos.

Para construir uma linguagem de programação algumas restrições ou necessidades devem ser observadas: Arquitetura, se o esforço de programar deve ser árduo em benefício ao poder de processamento - RISC ou se o esforço de programar deve ser atenuado em detrimento a desempenho da aplicação - CISC; Configurações Técnicas, levar em consideração a integração dos recursos da linguagem como, por exemplo, o uso de IDE's; Padrões, fazer uso de instituições ou organizações reguladoras, para padronizar a evolução da linguagem, exemplos: ISO, ANSI, OMG,...; e Sistemas Legados, que é um fator primordial na construção de novas linguagens, principalmente se for à evolução uma linguagem já existente, para que se mantenha a compatibilidade com que já foi criado.

Os objetivos que se almejam no desenvolvimento de linguagens são: Simplicidade e Legibilidade, para que se possa escrever pouco e fazer muito, sendo principalmente de fácil entendimento; Confiabilidade, para que o que se codifique gere o resultado esperado; Suporte, que a linguagem possua recursos para sanar dúvidas caso surjam; Abstração, que se possa focar no que é relevante, deixando as particularidades para as especializações da linguagem; Ortogonalidade, que o entendimento do código seja claro, caso a necessidade de leitura seja rápida, e isso só é possível se a linguagem for criada a partir de primitivas simples; e por fim a Eficiência, isto é a linguagem construir o que se propõe de forma rápida.

As linguagens de programação são ferramentas construídas para desenvolver outras ferramentas e essas precisam ser compreendidas pelos Sistemas Operacionais (SO's) que controlam a máquina (*hardware*). Para se chegar a um código-executável a partir de um código-fonte temos as seguintes formas:

- Linguagens Interpretadas: são aquelas que o código-fonte é analisado linha-a-linha pelo interpretador toda vez que o programa for executado, resultando em tempo de execução baixo;
- Linguagens Compiladas: são aquelas que todo o código-fonte é analisado gerando um código-final que toda vez que for executado não precisar ser mais analisado, resultando em um tempo de execução alto, porém com o código atrelado ao Sistema Operacional; e
- Linguagens Híbridas (compiladas-interpretadas) Máquinas Virtuais: são aquelas que todo o código-fonte é analisado e é gerado um código-objeto (*bytecode*) que para ser entendido pela máquina tem que ser interpretado por uma máquina-virtual, sendo assim mais rápido que o interpretado, mais lento que o compilado, porém portátil para qualquer Sistema Operacional.

1.8 Princípios de Linguagens de Programação

As linguagens de programação em sua maioria possuem alguns itens, ou melhor, recursos que são importantes o seu conhecimento. Um exemplo é saber distinguir o que é procedimento e função.

Um procedimento é um bloco de comando que não retorna um valor. Podemos exemplificar com o comando CLS do sistema operacional MS-DOS

que faz a ação de limpar a tela, mas não retorna nada. A própria ação já é o retorno, pois se não limpar a tela não funcionou, caso contrário tudo bem. Exemplificando, podemos observar nos trechos de código abaixo, feitos em linguagens diferentes representando um procedimento.

Exemplo em Pascal

```
procedure exProc(p1: integer; p2: integer);  
  // será exibido na tela o resultado da operação, mas não será retornado  
  nenhum valor  
  // para quem chamou esse procedimento  
  var  
    v1: integer;  
  begin  
    v1 := p1 + p2;  
    write('A soma é: ' + IntToStr(v1));  
  end;
```

Exemplo em Java

```
public static void exProc(int p1, int p2) {  
  // será exibido na tela o resultado da operação, mas não será retornado  
  nenhum valor  
  // para quem chamou esse procedimento (método sem retorno)  
  int v1;  
  v1 = p1 + p2;  
  System.out.print("A soma é: " + v1.ToString());  
}
```

Uma função também é um bloco de comando, porém que retorna um valor. Podemos exemplificar com a função DATE do sistema operacional MS-DOS que faz a ação de obter a data corrente e a retorna exibindo-a na tela. Exemplificando, podemos observar nos trechos de código abaixo, feitos em linguagens diferentes representando uma função.

Exemplo em Pascal

```
function exFunc(p1: integer; p2: integer): integer;  
  // será retornado para quem chamou esse procedimento o resultado da  
  operação  
  var  
    v1: integer;  
  begin  
    v1 := p1 + p2;
```

```
    exFunc = v1;  
end;
```

Exemplo em Java

```
public static int exFunc(int p1, int p2) {  
    // será retornado para quem chamou esse procedimento o resultado da  
    operação  
    int v1;  
    v1 = p1 + p2;  
    return (v1);  
}
```

Agora observem que em todos os exemplos tanto os procedimentos como as funções recebem valores em variáveis chamadas p1 e p2 do tipo inteiro, para serem processados. Essas variáveis recebem o nome de parâmetros e tem o objetivo de receber um valor passado por um programa chamador. Vejam o exemplo abaixo nas duas linguagens já utilizadas de programas que chamam as funções exFunc para exibir o resultado da operação.

Exemplo em Pascal

```
program ExPascal;  
var  
    vx: integer;  
    vy: integer;  
begin  
    vx := 2;  
    vy := 3;  
    write('A soma é: ' + exFunc(vx, vy));  
    // o resultado exibido é 5, já que foram passado 2 e 3 como parâmetros  
end.
```

Exemplo em Java

```
public static void main(String Args[]) {  
    int vx;  
    int vy;  
    vx = 3;  
    vy = 5;  
    System.out.print("A soma é: " + exFunc(vx, vy));  
    // o resultado exibido é 8, já que foram passado 3 e 5 como parâmetros  
}
```

Porém, as passagens de parâmetros basicamente podem ser de dois tipos. Um chamado de passagem por valor, que é o mais comum, e o outro de passagem por referência, menos usado. E qual a diferença? Por valor o procedimento ou a função recebem uma cópia da variável que é fornecida quando é invocada. O principal é que todas as alterações feitas dentro do bloco de comandos da função ou procedimento não vão afetar os valores originais da variável origem, já que o que foi passado foi uma cópia. Por referência a função ou o procedimento recebem é uma referência das variáveis utilizadas, e não uma cópia, isto quer dizer, que o que é passado é o endereço de memória da variável origem e decorrente disso as alterações realizadas nas variáveis dentro do bloco de comandos da função ou procedimento irão afetar os valores originais, já que se está usando o mesmo local de memória das rotinas chamadoras. Para entender melhor, observe a figura 2.7 e veja os exemplos abaixo feitos nas linguagens C, C++, Pascal e Java.

Figura 1.7 Conceitos Gerais de Paradigmas



Fonte: Próprio Autor

Passagem por valor em C

```
#include<stdio.h>
void troca(int v1, int v2) {
    int aux;
    aux = v1;
    v1 = v2;
    v2 = aux;
}
int main() {
    int v1 = 2, v2 = 3;
```

```
printf("Antes de chamar a função :\n v1 = %d\n v2 = %d\n",v1,v2);
troca(v1,v2);
printf("Depois de chamar a função:\n v1 = %d\n v2 = %d\n",a,b);
return 0;
// mesmo o procedimento troca ter feito a troca dos valores de v1 e v2 o
que é
// exibido antes é o mesmo que é exibido depois, pois a troca só ocorreu
dentro
// do procedimento, visto que a passagem foi por valor e sendo assim o que
é
// enviado é uma cópia das variáveis v1 e v2.
}
```

Passagem por referência em C

```
#include<stdio.h>
void troca(int *v1, int *v2) {
    int aux;
    aux = *v1;
    *v1 = *v2;
    *v2 = aux;
}
int main() {
    int v1 = 2, v2 = 3;
    printf("Antes de chamar a função :\n v1 = %d\n v2 = %d\n",v1,v2);
    troca(&v1,&v2);
    printf("Depois de chamar a função:\n v1 = %d\n v2 = %d\n",a,b);
    return 0;
    // a passagem por referência na linguagem C é representada pelo uso de
    // '&' = e comercial' o que faz com que os valores sejam passados através
da
    // referência de memória das variáveis, e assim o sendo, o que é exibido
antes
    // é diferente do que é exibido depois já que o procedimento troca,
    // inverteu os valores das variáveis. Um detalhe é que em C para se
manipular
    // variáveis passados por referência, deve-se usar o '*' = asterisco' antes
delas.
}
```

Passagem por valor em C++

```
#include<stdio.h>
```

```
void troca(int v1, int v2) {
    int aux;
    aux = v1;
    v1 = v2;
    v2 = aux;
}

int main() {
    int v1 = 2, v2 = 3;
    cout<<"Antes de chamar a função :\\na = " << v1<< " \\nb = " << v2 << endl;
    troca(a,b);
    cout<<"Depois de chamar a função :\\na = " << v1 << " \\nb = " << v2 << endl;
    return 0;
    // mesmo o procedimento troca ter feito a troca dos valores de v1 e v2 o
    que é
    // exibido antes é o mesmo que é exibido depois, pois a troca só ocorreu
    dentro
    // do procedimento, visto que a passagem foi por valor e sendo assim o que
    é
    // enviado é uma cópia das variáveis v1 e v2.
}
```

Passagem por referência em C++

```
#include<stdio.h>
void troca(int &v1, int &v2) {
    int aux;
    aux = v1;
    v1 = v2;
    v2 = aux;
}

int main() {
    int v1 = 2, v2 = 3;
    cout<<"Antes de chamar a função :\\na = " << v1<< " \\nb = " << v2 << endl;
    troca(a,b);
    cout<<"Depois de chamar a função :\\na = " << v1 << " \\nb = " << v2 << endl;
    return 0;
    // a passagem por referência na linguagem C++ é representada pelo uso
    de
    // '&' e comercial' o que faz com que os valores sejam passados através
    da
    // referência de memória das variáveis, e assim o sendo, o que é exibido
    antes
```

```
// é diferente do que é exibido depois já que o procedimento troca,  
// inverteu os valores das variáveis.  
}
```

O outro recurso importante nas linguagens de programação é a recursividade, porém nem todas possuem esse importante recurso. Agora o que é recursividade? Podemos dizer, que é a capacidade de uma função se auto-chamar. Isso otimiza a lógica de programação em muitos casos. O exemplo clássico é a criação da função fatorial. Veja o exemplo do uso da função matemática fatorial $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$ escrita em pascal:

Fatorial em Pascal

```
program TestaFatorial;  
var  
    num, fat: integer;  
function Fatorial (num:integer): integer;  
begin  
    if (num = 0) then  
        begin  
            Fatorial := 1;  
        end  
    else  
        begin  
            Fatorial := num * Fatorial(num - 1);  
        end;  
    end;  
begin  
    writeln('entre com um número inteiro');  
    readln(num);  
    fat := fatorial(num);  
    writeln('o fatorial de ',num,' é ',fat);  
end.
```

Fatorial em Java

```
public class ExemploFatorial {  
    public static void main (String args[]) {  
        for(long i=0; i<=10; i++) {  
            System.out.println(i + "! = " + fatorial(i));  
        }  
    }  
    public static long fatorial(long num) {
```

```
if (num <=1) {  
    return (1);  
}  
else {  
    return (num * fatorial(num - 1));  
}  
}
```

Para finalizar seguem mais dois recursos chamados pilhas e filas que serão detalhados na disciplina de Estrutura de Dados, mas para que se tenha um conhecimento breve segue suas descrições:

Pilha é uma estrutura computacional que funciona como uma pilha de anilhas, onde o Primeiro que Entra é o Último que Sai, que em inglês é chamado de *Last In First Out*, porém mais conhecido pelo acrônimo LIFO.

Figura 1.8 Estrutura de Pilha



Fonte: Internet

Fila é outra estrutura computacional que funciona como uma fila de banco, onde o Primeiro que Entra é o Primeiro que Sai, que em inglês é chamado de *First In First Out*, mas mais conhecido como FIFO.

Figura 1.9 Estrutura de Fila



Fonte: Internet

Esses conceitos são amplamente utilizados na computação, como, por exemplo, durante a execução de um programa, para o armazenamento de valores de variável local a um bloco e também para conter o endereço de retorno do trecho de programa que chamou a função ou procedimento atualmente em execução.

1.9 Backus-Naur (BNF)

Formalismo de Backus-Naur (BNF) é uma meta-sintaxe usada para expressar gramáticas livres de contexto, isto é, um modo formal de descrever linguagens formais.

A BNF é amplamente usada como uma notação para as gramáticas de linguagens de programação, conjuntos de instruções e protocolos de comunicação, e também como notação para representar partes de gramáticas de linguagens naturais.

[Clique aqui para ver exemplo \(BNF\) de Gramática da Linguagem Java](http://cui.unige.ch/isi/bnf/JAVA/BNFindex.html)

Disponível em: < <http://cui.unige.ch/isi/bnf/JAVA/BNFindex.html> >

1.10 Conceitos Básicos do Paradigma Orientado a Objetos

Classe O diagrama de classes na Programação Orientada a Objetos (POO), os problemas são pensados e expressados como objetos, ao contrário da análise tradicional os quais eram em rotinas e dados, que aqui foram substituídos por métodos (comportamento) e atributos (propriedades). Assim, quando é colocado o problema de desenvolver um sistema acadêmico na análise orientada a objetos, deve-se pensar como dividir esse problema em objetos.

Assim teríamos Alunos, Administradores, Professores, Cursos, Turmas, etc. E a melhor maneira de conceituar estes termos é considerar um objeto do mundo real e mostrar como podemos representá-lo em termos conceitos em POO. Assim, um objeto é um conceito que usamos para representar uma entidade do mundo real. Exemplificando, um Aluno possui nome, data de nascimento, identidade, etc. E além dessas características (propriedades), possuem ações (métodos) como frequentar as aulas, fazer as avaliações, etc. Em termos de POO para podermos tratar os objetos temos que criar classes.

Assim, uma classe representa um conjunto de objetos que possuem comportamentos e características comuns.

Na UML em primeiro lugar denomina-se uma classe e recomenda-se nomeá-la capitalizando-a e deixando no singular. E em segundo lugar denominam-se as propriedades, informações específicas relacionadas a uma classe de objeto, isto é características dos objetos que as classes representam.

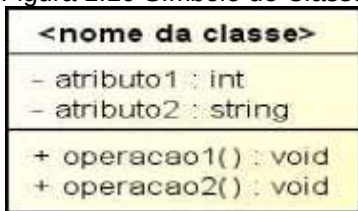
Em terceira e última parte, métodos, que são ações que os objetos de uma classe podem realizar. Assim tem-se um modelo para instanciar quantos objetos forem necessários. Dessa forma os objetos instanciados possuirão todas as características e comportamentos definidos pela classe. Então as classes especificam a estrutura (propriedades) e os comportamentos (operações) dos objetos, que são instâncias das classes.

Geralmente em um sistema de médio porte serão identificadas diversas classes que compõem o sistema. Neste contexto a UML surgiu como uma proposta de ser uma linguagem para modelagem de dados que usava diversos artefatos para representar o modelo de negócio; um destes artefatos é o diagrama de classes.

Os diagramas de classes registram atributos e operações de uma classe e as restrições de como os objetos podem ser conectados, descrevendo também os tipos de objetos no sistema e os relacionamentos entre eles e esse podem ser associações e abstrações. Para poder representar a visibilidade dos atributos e operações, a UML usa os seguintes símbolos:

- + público, visível em qualquer classe;
- - privado, visível somente dentro da classe;
- # protegido, na classe e suas subclasses.

Figura 1.10 Símbolo de Classe

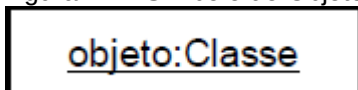


Fonte: Próprio Autor

Objeto Agora que sabemos o conceito e como codificar uma classe, vamos entender sobre objeto.

Segundo os pais da UML (Rumbaugh, Jacobson e Booch) um objeto é uma instância de uma classe, isto é, trata-se de uma cópia da classe na memória em tempo de execução (*runtime*), sendo assim, um elemento específico que possui valores (estados) nos atributos (campos).

Figura 1.11 Símbolo de Objeto



Fonte: Próprio Autor

A UML representa um objeto usando um retângulo que o representa, onde o nome da instância do objeto é seguido de dois pontos, seguido do nome da classe, com essa formação sublinhada.

Herança O relacionamento entre classes retrata as relações entre os objetos. Exemplo: um professor ministra uma disciplina para alunos numa sala. A UML reconhece três tipos mais importantes de relações: dependência, associação e generalização ou herança. Geralmente as classes não estão isoladas (coesas) e se relacionam entre si (acoplamento). O relacionamento e a comunicação entre as classes se definem em três tipos: Associações que podem ser de Agregação ou Composição; Generalização ou herança; e Dependências.

As associações são relacionamentos estruturais entre instâncias e especificam que objetos de uma classe estão ligados a objetos de outras classes, podendo ser unária, binária, etc. As associações podem existir entre classes ou entre objetos. Uma associação entre a classe Professor e a classe Disciplina (um professor ministra uma disciplina) significa que uma instância de Professor (um professor específico) vai ter uma associação com uma instância de uma Disciplina. Esta relação significa que as instâncias das classes são conectadas, seja fisicamente ou conceitualmente.

Polimorfismo É a capacidade de classes mais abstratas possuírem comportamentos diferentes das classes concretas, isto é, duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura), mas comportamentos distintos, especializados para uma das classes derivadas. A decisão sobre qual o

método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução, através do mecanismo de ligação tardia.

Mas para ser considerado polimorfismo, é necessário que os métodos tenham exatamente a mesma identificação, quer dizer, o mesmo nome de método, sendo utilizado o mecanismo de redefinição de métodos, isto é, sobrescrita dos métodos.

Sobrescrita (Override) a redefinição ocorre quando um método cuja assinatura já tenha sido especificada recebe uma nova definição, ou seja, uma nova implementação em uma classe derivada. O mecanismo de redefinição, juntamente com o conceito de ligação tardia, é a chave para a utilização do polimorfismo. É importante observar que, quando polimorfismo está sendo utilizado, o comportamento que será adotado por um método só será definido durante a execução.

Sobrecarga (Overload) um método aplicado a um objeto é selecionado para execução através da sua assinatura e da verificação a qual classe o objeto pertence. Dois ou mais métodos de uma mesma classe podem ter o mesmo nome, desde que suas listas de parâmetros sejam diferentes, constituindo assim uma assinatura diferente. Tal situação não gera conflito, pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos de argumentos do método.

Encapsulamento Consiste na separação de aspectos internos e externos de um objeto. Este mecanismo é utilizado amplamente para impedir o acesso direto ao estado de um objeto (seus atributos), disponibilizando externamente apenas os métodos que alteram estes estados.

1.11 Exemplos

BNF da Gramática do Java. Em <

<http://cui.unige.ch/isi/bnf/JAVA/BNFIndex.html> >

Exemplo de Compilador Fácil (em Java). Em < <http://goo.gl/PnHW7p> >

Exemplo de Compilador Simplíssimo (em CSharp). Em < <http://goo.gl/Dpf1wK> >

Exemplo de Compilador Simplíssimo (em Java). Em < <http://goo.gl/aikTAL> >

Exemplo de Compilador Muito Simples (em Java). Em < <http://goo.gl/8YQbik> >

Exemplo de Compilador Simples (em Java). Em < <http://goo.gl/8YQbik> >

Exemplo de Compilador Completo (em Java). Em < <http://goo.gl/Mglwlt> >

Exemplo de Autômatos (em Java). Em < <http://goo.gl/Vudw8L> >
Exemplo de Autômato com Pilha (em Java). Em < <http://goo.gl/l8yC8p> >

1.12 Ferramentas

Case-Tools. Em < <http://case-tools.org/uml.html> >
Astah. Em < <http://astah.change-vision.com/en/index.html> >
Simulação de Autômatos. Em < <http://goo.gl/kwf3lo> >

1.13 Vídeos

Compiladores. Em < <http://vimeo.com/112860457> >

1.14 Sugestões de Leitura

Introdução a Informática. Em < <http://goo.gl/LPT4Sc> >
Engenharia de Software. Em < <http://goo.gl/xk0CqE> >
Fundamentos de Programação. Em < <http://goo.gl/xwYOfC> >
UML Guia do Usuário. Em < <http://goo.gl/av5BTh> >
Análise e Projeto com UML. Em < <http://goo.gl/Q5Hp7q> >
Utilizando UML e padrões. Em < <http://goo.gl/1YPJ0l> >

1.15 Referências Web

A Theory of Object-Oriented Design. Em < <http://goo.gl/fq3q5> >
ANSI site oficial. Em < <http://www.ansi.org/> >
ANSI wiki. Em < <http://goo.gl/ZCrMxk> >
Aulas de UML. Em < <http://goo.gl/2TRbUF> >
Design Patterns Languages for OOP. Em < <http://www.vincehuston.org/dp/> >
Exemplo de Modelagem UML com a ferramenta ASTAH. Em < <http://goo.gl/JvqhSt> >
Exemplo de Modelagem UML. Em < <http://goo.gl/zw0YCF> >
Guia Básico sobre UML. Em < <http://goo.gl/eGhizM> >
IBM Rational Unified Process. Em < <http://goo.gl/uqI9H7> >
ISO site oficial. Em < <http://www.iso.org/iso/home.html> >
ISO wiki. Em < <http://goo.gl/vjWG> >
Langpop. Em < <http://www.langpop.com/> >
Mais sobre relacionamentos de Inclusão e Extensão. Em < <http://goo.gl/uH8X6Z> >

Manuais de Referência OMG-UML. Em < <http://www.omg.org/spec/UML/> >
Object Management Group (OMG). Em < <http://www.omg.org/> >
Object-Oriented Programming (OOP). Em < <http://goo.gl/AGVpPX> >
Ohloh. Em < <http://www.ohloh.net/> >
Rational Unified Process (RUP). Em < <http://www-01.ibm.com/software/br/rational/> >
Tiobe. Em < <http://www.tiobe.com/index.php/content/company/Home.html> >
Unified Modeling Language (UML). Em < <http://www.uml.org/> >
Wiki paradigma de desenvolvimento. Em < <http://goo.gl/rIYFIs> >
Wiki paradigma. Em < <http://pt.wikipedia.org/wiki/Paradigma> >

1.16 Referências Bibliográficas

AHO, Alfred; *et al.* Compilers: Principles, Techniques, and Tools. 2ªed. US: Pearson, 2006.

FERNANDES, Daniel. Análise de Sistemas Orientada ao Sucesso. 1ªed. Rio de Janeiro: Ciência Moderna, 2005.

LOPES, Filomena; *et al.* Desenvolvimento de Sistemas de Informação, Métodos e Técnicas, 2005.

LOUDEN, Kenneth. Compiladores: Princípios e Práticas. 1ªed. Pioneira Thomson Learning, 2004.

MEDEIROS, Ernani. Desenvolvendo Software com UML 2.0: Definitivo. 1ªed. São Paulo: Makron Books, 2004.

MELO, Ana. Desenvolvendo Aplicações com UML 2.2: do conceitual à implementação. 3ªed. RJ: Brasport, 2010.

RUMBAUGH, James; *et al.* Modelagem e Projetos Baseados em Objetos com UML. 2ªed. RJ: Campus, 2006.

RUMBAUGH, James; JACOBSON, Ivar; BOOCH, Grady. UML: Guia do Usuário. 1ªed. RJ: Campus, 2006.

SCHACH, Stephen. Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos. McGraw-Hill, 2008.

SOMMERVILLE, Ian. Engenharia de Software. 8ªed. Pearson, 2007.

TUCKER, Allen; NOONAN, Robert. Linguagens de Programação: Princípios e Paradigmas. 2ªed. McGraw, 2009.

WAZLAWICH, Raul. Análise e Projeto de Sistemas de Informação Orientados a Objetos. RJ: Campus, 2011.

2 UML

Para entender melhor os conceitos do paradigma de desenvolvimento orientado a objetos segue a apresentação da linguagem de modelagem unificada que padroniza a simbologia dos conceitos de POO.

2.1 Modelagem

Os projetos seguem processos de desenvolvimento e um dos processos mais maduros é o Processo Unificado (UP) que possui 4 marcos: Marco dos objetivos do ciclo de vida; Marco da arquitetura do ciclo de vida; Marco da capacidade operacional inicial; e Marco do lançamento do produto.

Passado pelo primeiro marco tem que começar a modelar os requisitos especificados. O cliente necessita entender o que os projetistas estão construindo e precisam ter condições de inferir os seus conhecimentos nos projetos para atender plenamente suas necessidades.

Para isso é necessário estabelecer um canal formal de interação onde a linguagem natural do cliente é transformada em linguagem técnica para a equipe de desenvolvimento. Com esse objetivo a UML se propõe a ser uma linguagem padrão aceita e compreendida por todos os *stakeholders*.

A modelagem utilizando a Linguagem de Modelagem Unificada, mais conhecida através da sigla em inglês UML que significa *Unified Modeling Language* é focada no paradigma Orientado a Objetos (OO) cujos conceitos são classe, objeto, herança, polimorfismo, encapsulamento de atributos e métodos, alta coesão e baixo acoplamento. É usado para a análise não focando na codificação do *software* ou *hardware* e sim no entendimento do problema (análise) e na sua solução (projeto).

A UML representa símbolos, esses usados em diagramas que assim representam uma linguagem simbólica com regras claras e precisas para utilização desses símbolos nos diversos diagramas. O objetivo dos diagramas é apresentar múltiplas visões do sistema chamado de modelo. Assim, um modelo UML é um conjunto de diagramas que servem para compreender e desenvolver um projeto de *software*, descrevendo o que o *software* deve fazer.

2.2 Linguagem de Modelagem Unificada (UML)

Durante a década de 80, várias linguagens de programação se popularizaram, entre elas uma que se destacou foi à chamada de Smalltalk, mas outras linguagens também participaram deste movimento, tais como, Objective C, C++ e Eiffel, todas projetadas para o paradigma orientado a objetos. Seguindo esta onda, os métodos orientados a objetos começaram a ser publicados. Como exemplo temos em 1988 os de Shlaer, em 1990 os de Wirfs-Brock, em 1991 os de Coad e Yourdon, ainda em 1991 de Booch, e também em 1991 os de Rumbaugh.

A primeira tentativa de sucesso de se criar uma linguagem unificada aconteceu quando Rumbaugh se juntou a Booch na *Rational Software Corporation* em 1994. Eles começaram a combinar conceitos de OMT com os métodos de Booch, produzindo um método unificado. Neste mesmo momento Jacobson também se juntou a *Rational* e começou a trabalhar com Booch e Rumbaugh. O trabalho conjunto resultou no que foi chamado de *Unified Modeling Language* (UML). Este foi um momento único, quando os autores dos três principais métodos trabalharam juntos para unificar suas propostas. Veja na figura 5.3 como se deu essa evolução nas suas versões.

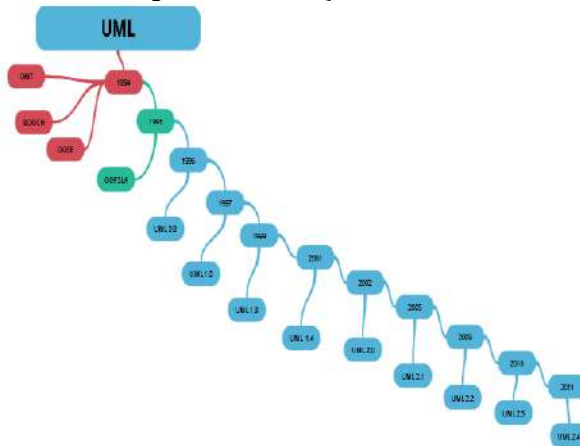
Em 1996, a *Object Management Group* (OMG) fez uma chamada por propostas com o objetivo de criar um padrão para modelagem orientada a objetos. Aos autores da UML começaram a trabalhar com metodologistas e desenvolvedores de outras empresas para produzir uma proposta atrativa para os membros da OMG, assim como para as empresas de ferramentas, metodologistas e desenvolvedores em geral. No final de 1997, se tornou um padrão da OMG e até hoje vem sendo mantida e evoluída, sendo hoje bem aceita por toda a comunidade de desenvolvimento de *software*.

A *Unified Modeling Language* (UML) é a sucessora de uma onda de métodos para análise e projeto orientados a objetos que surgiram no fim da década de 80. Ela unificou os métodos proposto por Booch, Rumbaugh e Jacobson. Atualmente a UML é um padrão mantido pela OMG, o que dá a ela um status de linguagem padrão para modelagem de sistemas, isto é, uma linguagem padrão para visualizar, especificar, construir e documentar artefatos de um sistema de *software* e combina diversos aspectos como:

- Modelagem de Dados
- Modelagem de Negócios
- Modelagem de Objetos

- Modelagem de Componentes

Figura 2.1 Evolução da UML

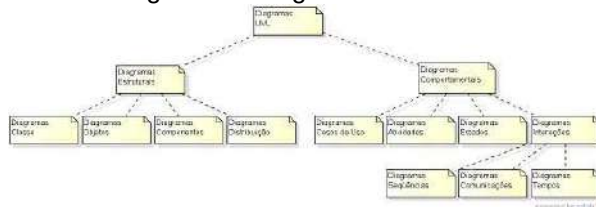


Fonte: Próprio Autor

É importante enfatizar que UML é uma linguagem de modelagem, ou seja, é uma forma de especificar um sistema de maneira visual. Ela é a linguagem padrão de modelagem adotada pelo *Rational Unified Process* (RUP) e seu grande sucesso se deu em conjunto com a popularização do RUP. Por isso, é muito comum que as pessoas associem UML ao RUP.

É importante distinguir muito bem, visto que as duas representam conceitos complementarmente diferentes. O RUP é um processo, ou seja, um conjunto de atividades, com seus respectivos artefatos e papéis, para se desenvolver produtos de *software* e a UML é uma linguagem para se modelar o comportamento e estrutura de um sistema. Na figura 5.4 se pode verificar quais são os principais diagramas dessa linguagem.

Figura 2.2 Diagramas da UML

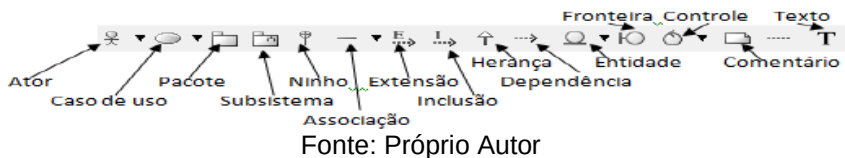


Fonte: Próprio Autor

2.2.1 Diagrama de Casos de Uso

O diagrama de caso de uso segundo Ivan Jacobson, um caso de uso é “um documento narrativo que descreve a sequência de eventos de um ator que usa um sistema para completar um processo”. Um caso de uso é uma técnica de modelagem que descreve o que o novo sistema deve fazer e se baseia em um processo de licitação de requisitos entre os *stakeholders* na condução de uma especificação na qual todos interagem. Na figura 5.5 temos os elementos utilizados nesse diagrama e na figura 5.6 os dois principais símbolos.

Figura 2.3 Elementos do Diagrama de Casos de Uso



Um caso de uso descreve quais comportamentos o sistema deverá responder para cada um dos usuários do mesmo, servindo de formalização das ações que precisam ser desenvolvidas. Retratando uma lista de eventos entre os usuários e o sistema em uma visão abstrata, onde essa lista de eventos relatados abstratamente descreve as interações desde o início da atividade até o fim da mesma.

Figura 2.4 Símbolos da UML



Casos de uso têm que ser compreensíveis pelos usuários por que só eles sabem o que o sistema precisa fazer. Os casos de uso permitem verificar se o desenvolvedor e o usuário concordam sobre o que o sistema deve fazer. Isso é um problema importante no desenvolvimento de *software*. No mesmo tempo, casos de uso podem servir de "contratos" entre os usuários e os desenvolvedores. Sendo assim os casos de uso devem:

- Descrever os requisitos funcionais do sistema;
- Fornecer uma descrição clara do que se deve fazer; e
- Definir os comportamentos das classes.

É importante salientar que casos de uso não são requisitos. E sim uma visão abstrata desses os quais são compostos pelos seguintes componentes:

- Ator, que é um papel que solicita eventos ao sistema e recebe feedbacks, onde cada ator pode participar de vários casos de uso; e
- Casos de uso são documentos narrativos que descrevem a sequência dos eventos feitos por um ator no uso do sistema.

Os elementos que descrevem esses papéis são exibidos abaixo. Vale salientar que o nome dos casos de uso deve ser composto por frases curtas, preferencialmente verbo + substantivo. Exemplo: matricular aluno.

Para especificar um caso de uso pode-se usar a estrutura que se segue, porém não existe uma estrutura rígida:

- Código do caso de uso;
- Nome do caso de uso;
- Descrição do caso de uso (um parágrafo);
- Lista dos nomes dos atores com descrição curta;
- Prioridade, caso de uso muito importante no projeto ou acessório;
- Pré-condições;
- Pós-condições;
- Fluxo de eventos principal;
- Fluxos de eventos alternativos;
- Fluxo de eventos de exceções;
- Cenários; e
- Dicionário de Dados.

Para identificar os atores que vão participar do modelo uma das técnicas é fazer as seguintes perguntas:

- Quem usa o sistema?
- Quem inicia o sistema?
- Quem fornece os dados?
- Quem usa as informações?

Na descrição dos atores geralmente se faz o uso de:

- Nome do caso de uso;
- Tipo de uso se é frequente, ocasional, etc;
- Descrição de seu papel no sistema.

Para identificar os casos de uso, normalmente a técnica usada é:

- Verificar as interações entre os atores e o sistema; e
- Verificar as ações do ator e do sistema.

É importante salientar que os atores sempre iniciam a ação. Para entendimento imagine o requisito em que um aluno necessita fazer um curso, onde existe a necessidade de um sistema de matrículas. Então:

- Quais são os atores?
- Quem usa o sistema?
- Como podemos identificar o caso de uso?

Podemos chamar este caso de uso de: Matricular Aluno (verbo+substantivo). Para fazer uma descrição textual do caso de uso, que possui dois atores Aluno e Administração. Sua especificação poderia ser:

- Caso de uso: Matricular Aluno.
- Atores: Aluno, Administrador.
- Descrição: Este caso de uso começa quando um aluno chega à administração que faz a matrícula do aluno no curso.

Fluxo:

- P1-O sistema registra a matrícula;
- P2-Recebe o pagamento;
- P3-Emite uma nota fiscal; e
- P4-O aluno é matriculado.

Na UML temos o diagrama de caso de uso que pode ser representado para o caso acima como mostrado na figura 5.7.

Figura 2.5 Diagrama de Caso de Uso



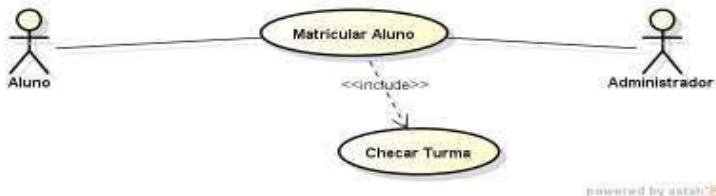
Fonte: Próprio Autor

Importante: o caso de uso deve começar com um verbo, enfatizando o processo. Exemplo: Matricular Aluno ou Cadastrar Curso. Deve-se identificar o ator que inicia o evento, começando a descrição da sequência de um caso de uso usando o esquema - este caso de uso começa quando um aluno faz uma matrícula em um curso. Verifique que um ator é um elemento externo ao sistema. Um ator é um papel que interage com o sistema, mas não faz parte dele.

O primeiro passo é identificar os casos de uso básicos, o próximo passo é identificar o relacionamento entre os casos de uso através das generalizações, inclusões e extensões.

As inclusões são um caso de uso que inclui o comportamento de outro. A finalidade básica da inclusão é realizar uma decomposição funcional e reduzir a complexidade de um caso de uso.

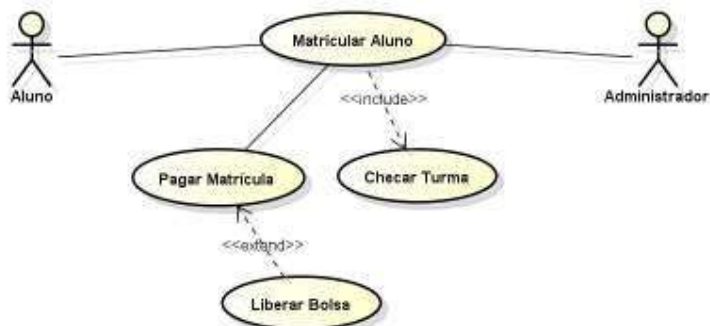
Figura 2.6 Diagrama de Casos de Uso com Inclusão



Fonte: Próprio Autor

As extensões definem pontos que adicionam um comportamento opcional, assim o caso de uso base pode ser executado mesmo que a extensão não o seja.

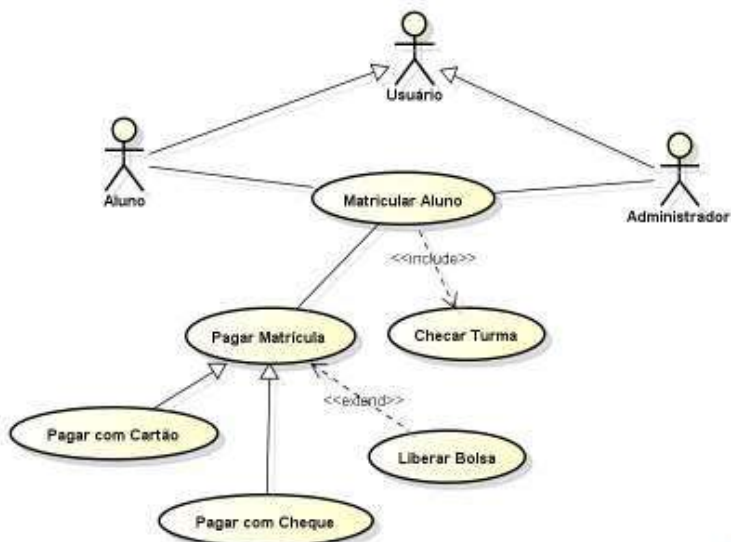
Figura 2.7 Diagrama de Casos de Uso com Extensão



Fonte: Próprio Autor

De princípio as generalizações são mais indicadas para as características dos atores, porém podem ser usadas com cautela para indicar especializações de comportamentos.

Figura 2.8 Diagrama de Casos de Uso Realizar Matrícula



Fonte: Próprio Autor

2.2.2 Diagrama de Classes

O diagrama de classes na Programação Orientada a Objetos (POO), os problemas são pensados e expressados como objetos, ao contrário da análise tradicional os quais eram em rotinas e dados, que aqui foram substituídos por métodos (comportamento) e atributos (propriedades). Assim, quando é colocado o problema de desenvolver um sistema acadêmico na análise orientada a objetos, deve-se pensar como dividir esse problema em objetos. Assim teríamos Alunos, Administradores, Professores, Cursos, Turmas, etc. E a melhor maneira de conceituar estes termos é considerar um objeto do mundo real e mostrar como podemos representá-lo em termos conceitos em POO.

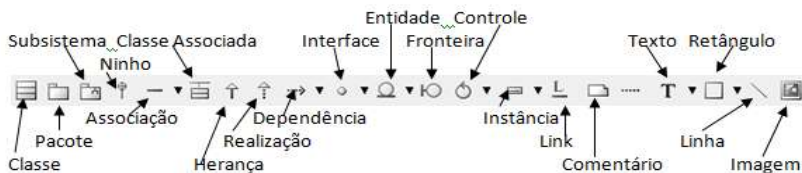
Assim, um objeto é um conceito que usamos para representar uma entidade do mundo real. Exemplificando, um Aluno possui nome, data de nascimento, identidade, etc. E além dessas características (propriedades), possuem ações (métodos) como frequentar as aulas, fazer as avaliações, etc. Em termos de POO para podermos tratar os objetos temos que criar classes. Assim, uma classe representa um conjunto de objetos que possuem comportamentos e características comuns.

Na UML em primeiro lugar denomina-se uma classe e recomenda-se nomeá-la capitalizando-a e deixando no singular. E em segundo lugar denominam-se as propriedades, informações específicas relacionadas a uma classe de objeto, isto é características dos objetos que as classes representam. Em terceiro e última parte, métodos, que são ações que os objetos de uma classe podem realizar. Assim tem-se um modelo para instanciar quantos objetos forem necessários. Dessa forma os objetos instanciados possuirão todas as características e comportamentos definidos pela classe. Então as classes especificam a estrutura (propriedades) e os comportamentos (operações) dos objetos, que são instâncias das classes.

Geralmente em um sistema de médio porte serão identificadas diversas classes que compõem o sistema. Neste contexto a UML surgiu como uma proposta de ser uma linguagem para modelagem de dados que usava diversos artefatos para representar o modelo de negócio; um destes artefatos é o diagrama de classes.

Observe a figura 5.11 para verificar os elementos que são utilizados nos diagramas de classes e na figura 5.12 a simbologia de um diagrama de classe definido na UML.

Figura 2.9 Elementos do Diagrama de Classes



Fonte: Próprio Autor

Os diagramas de classes registram atributos e operações de uma classe e as restrições de como os objetos podem ser conectados, descrevendo também os tipos de objetos no sistema e os relacionamentos entre eles e esse podem ser associações e abstrações. Para poder representar a visibilidade dos atributos e operações, a UML usa os seguintes símbolos: + público, visível em qualquer classe; - privado, visível somente dentro da classe; # protegido, na classe e suas subclasses.

Figura 2.10 Classe



powered by astah®

Fonte: Próprio Autor

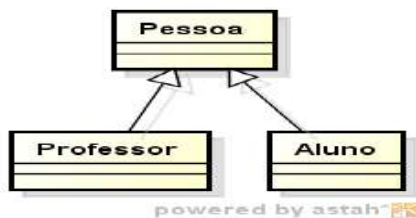
O relacionamento entre classes retrata as relações entre os objetos. Exemplo: um professor ministra uma disciplina para alunos numa sala. A UML reconhece três tipos mais importantes de relações: dependência, associação e generalização ou herança. Geralmente as classes não estão isoladas (coesas) e se relacionam entre si (acoplamento). O relacionamento e a comunicação entre as classes se definem em três tipos: Associações que podem ser de Agregação ou Composição; Generalização ou herança; e Dependências.

As associações são relacionamentos estruturais entre instâncias e especificam que objetos de uma classe estão ligados a objetos de outras classes, podendo ser unária, binária, etc. As associações podem existir entre

classes ou entre objetos. Uma associação entre a classe Professor e a classe Disciplina (um professor ministra uma disciplina) significa que uma instância de Professor (um professor específico) vai ter uma associação com uma instância de uma Disciplina. Esta relação significa que as instâncias das classes são conectadas, seja fisicamente ou conceitualmente. Dependências são relacionamentos de utilização no qual uma mudança na especificação de um elemento pode alterar a especificação do elemento dependente. A dependência entre classes indica que os objetos de uma classe usam serviços dos objetos de outra classe. Generalização ou herança, que pode ser simples ou múltipla (composta), serve para relacionar um elemento mais geral e um mais específico, onde o elemento mais específico herda as propriedades e métodos do elemento mais geral. Como a relação de dependência, ela existe só entre as classes. Um objeto particular não é um caso geral de outro objeto, apenas classes podem receber esse conceito. Agregação é um tipo de associação (é parte de ou todo-parte) onde o objeto parte é um atributo do todo, onde o objeto parte somente são criados se o todo ao qual estão agregados seja criado. Pedidos é composto por itens de pedidos. Composição é o relacionamento entre um elemento (o todo) e outros elementos (as partes) onde as partes só podem pertencer ao todo e são criadas e destruídas com ele. A seguir são mostradas as formas de relacionamento na UML.

Generalização é um relacionamento de especialização ou generalização, nos quais os objetos dos elementos especializados (os filhos) são substituíveis por objetos do elemento generalizado (os pais).

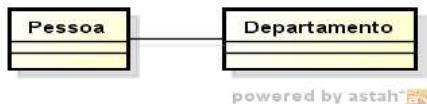
Figura 2.11 Diagrama de Classes com Herança



Fonte: Próprio Autor

Associação é um relacionamento estrutural que descreve um conjunto de ligações, em que as ligações são conexões entre objetos.

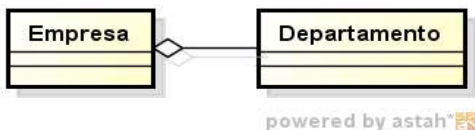
Figura 2.12 Diagrama de Classes com Associação



Fonte: Próprio Autor

Agregação é um tipo especial de associação representando um relacionamento estrutural entre o todo e sua parte.

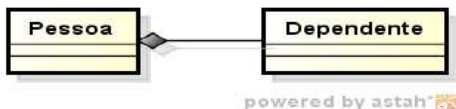
Figura 2.13 Diagrama de Classes com Agregação



Fonte: Próprio Autor

Composição é outro tipo especial de associação representando um relacionamento estrutural entre um elemento (o todo) e outros elementos (as partes) onde as parte só podem pertencer ao todo e são criadas e destruídas com ele.

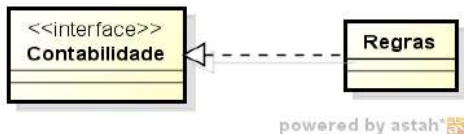
Figura 2.14 Diagrama de Classes com Composição



Fonte: Próprio Autor

Realização é um relacionamento semântico entre classes, em que uma classe especifica um contrato que outra classe garante executar.

Figura 2.15 Diagrama de Classes com Realização



Fonte: Próprio Autor

Dependência é um relacionamento semântico entre duas classes, nos quais a alteração de um (a classe independente) pode afetar a semântica da outra (a classe dependente).

Figura 2.16 Diagrama de Classes com Dependência



Fonte: Próprio Autor

O diagrama de classes lista todos os conceitos do domínio que serão implementados no sistema e as relações entre os conceitos. Ele é muito importante, pois define a estrutura do sistema a desenvolver. O diagrama de classes é consequência do prévio levantamento de requisitos, definição de casos de usos e classes. Como exemplo tem os passos de elicitação de requisitos com os *stakeholders* do sistema a ser desenvolvido, usando a técnica de entrevista com os administradores, professores, etc, que a partir desses os objetos do sistema são definidos: Alunos, Professores, Turmas, Cursos, etc. Definição dos atores do sistema: aluno, professor, administrador, etc. Definição e detalhamento dos casos de uso: Matricular Aluno, Pagar Matrícula, etc. Definição das classes: alunos, professor, etc.

Figura 2.17 Diagrama de Classes com Relacionamento



Fonte: Próprio Autor

Atributo representa uma propriedade que todos os objetos da classe têm, porém cada objeto terá valores particulares para seus atributos. A UML, o nome de um atributo é um texto que deve capitalizar todas as primeiras letras de cada palavra no nome menos a primeira palavra.

Todos os métodos têm que respeitar exatamente a assinatura que é composta pelo nome, número de parâmetros, tipos de dados e ordem. Um método não pode acrescentar ou cortar um parâmetro. Para mandar a mensagem corretamente, devesse saber qual é a classe do objeto, já que cada classe tendo método com assinatura diferente.

Mais uma informação. Devido aos questionamentos de alguns alunos sobre a classificação em relação ao tipo de generalização, complementei essa aula especificando essa tipificação. Ela pode ser basicamente de dois tipos: Total ou Parcial. Se Total temos para cada ocorrência da uma classe genérica sempre uma ocorrência em uma das classes especializadas. Se Parcial temos que nem sempre uma ocorrência da classe genérica possui uma ocorrência correspondente em uma classe especializada. Essa tipificação é muito interessante para a camada de persistência e pode ser também utilizada na estereotipação de entidades. Para melhor entender veja o exemplos na figura 5.20.



Fonte: Próprio Autor

2.2.3 Diagrama de Objetos

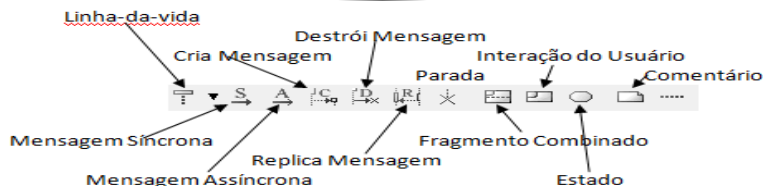
Segundo os pais da UML (Rumbaugh, Jacobson e Booch) um objeto é uma instância de uma classe, isto é, trata-se de uma cópia da classe na memória em tempo de execução (*runtime*), sendo assim, um elemento específico que possui valores (estados) nos atributos (campos).

A UML representa um objeto usando um retângulo que o representa, onde o nome da instância do objeto é seguido de dois pontos, seguido do nome da classe, com essa formação sublinhada.

2.2.4 Diagrama de Sequência e Comunicação ou Colaboração

O diagrama de sequência serve para exibir as interações entre os vários componentes de um sistema em especial os objetos e como seus métodos interagem entre si e em qual ordem (sequência).

Figura 2.19 Elementos do Diagrama de Sequência



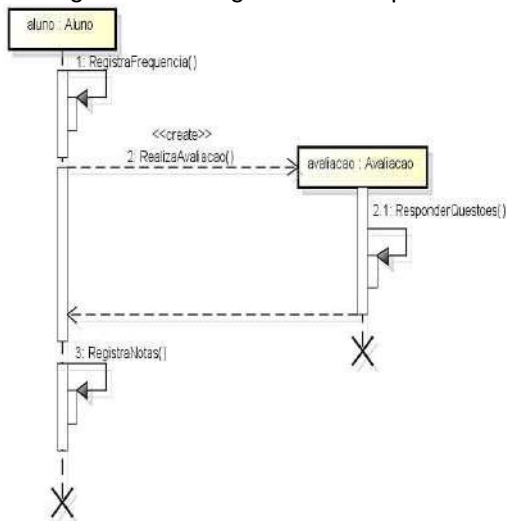
Fonte: Próprio Autor

O diagrama de comunicação serve para representar os elementos de um sistema que trabalham em conjunto. Esse diagrama expressa essa comunicação, mostrando a troca de mensagens entre os objetos.

Ele é similar ao diagrama de sequência, porém mais simples.

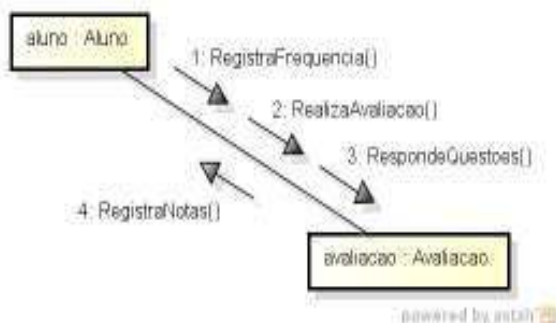
Os diagramas de sequência e de comunicação mostram as interações entre os objetos, por este motivo a UML se refere a estes diagramas como diagramas de interação.

Figura 2.20 Diagrama de Sequência



Fonte: Próprio Autor

Figura 2.21 Diagrama de Comunicação

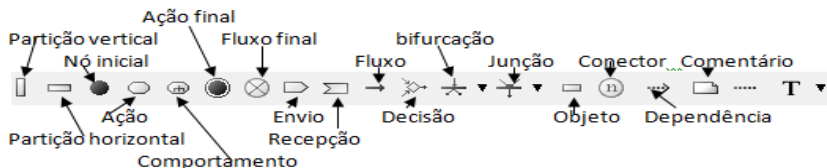


Fonte: Próprio Autor

2.2.5 Diagrama de Atividades

O diagrama de atividade exibe a forma que um objeto executa suas ações em um único processo, representando-os passo a passo, isto é, seu fluxo. As atividades que ocorrem em um caso de uso ou no comportamento do objeto seguem a sequência conforme descrito do diagrama.

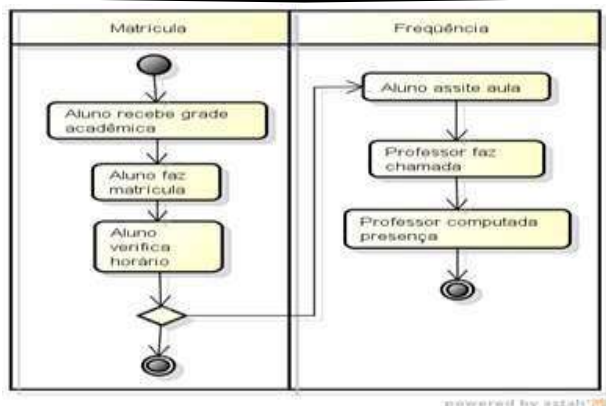
Figura 2.22 Elementos do Diagrama de Atividades



Fonte: Próprio Autor

Na figura de número 5.25 é exibido um exemplo de um diagrama de atividades que para os mais antigos podem observar o quanto é semelhante a um fluxograma, e isso se deve ao fato de ser uma herança deste último, pois a finalidade de ambos é a mesma. Representar a lógica de um processamento.

Figura 2.23 Diagrama de Atividades



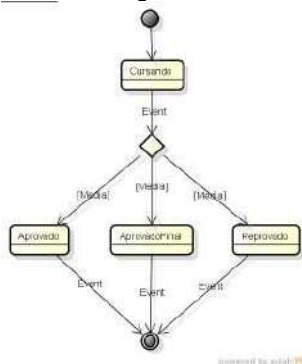
Fonte: Próprio Autor

2.2.6 Diagrama de Estados

O diagrama de estados tem a finalidade de exibir como um objeto realiza uma determinada operação num determinado momento da execução, representando um estado particular. Exemplo: um aluno pode estar com a situação de aprovado, aprovado na final ou reprovado.

O símbolo no topo da figura representa o início do estado e o símbolo na base da figura representa o fim do estado, entre elas são exibidas as transições de um estado para outro.

Figura 2.24 Diagrama de Estados

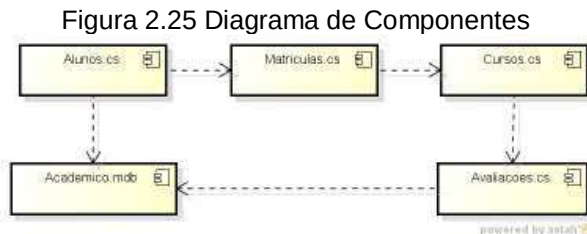


Fonte: Próprio Autor

2.2.7 Diagrama de Componentes

O diagrama de componentes mostra a organização dos componentes na implantação do sistema, mostrando os elementos (componentes) reutilizáveis de *software* e suas interdependências. Vale salientar que um componente é composto por um conjunto de classes, isto é, um *package* em Java ou *namespace* em C#. Na maioria das vezes um componente depende de outros, da mesma forma que as classes que ele possui, dependem das funcionalidades de outras classes do mesmo ou de outro componente. Desta forma o diagrama de componentes mostra esta dependência.

O diagrama de componentes também pode servir para exibir a configuração de um projeto de *software*, expondo a dependência entre os artefatos que compõem o projeto. Assim, as relações de dependência servem para informar a finalidade dos diversos artefatos que podem compor o projeto. E esses artefatos podem ser estereotipados para melhor identificá-los. Podendo ser um executável, um pacote, uma tabela, ou qualquer outro tipo de arquivo, estereotipado como documento ou às vezes arquivo.

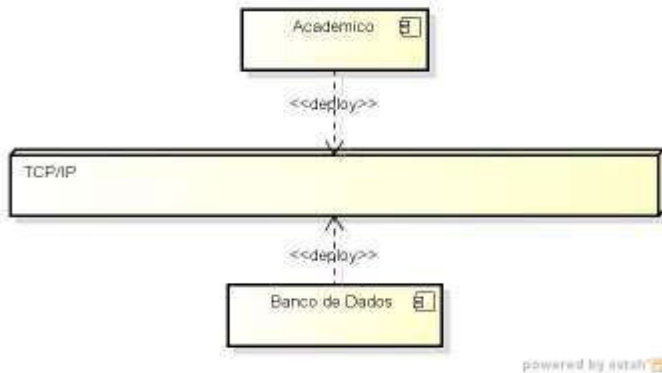


Fonte: Próprio Autor

2.2.8 Diagrama de Distribuição

O diagrama de implantação nos mostra a configuração física sobre qual o sistema será instalado. Esses diagramas servem para exibir a distribuição de *hardware* do projeto de *software*, identificando os recursos computacionais como pontos (nós) além da rede que os relaciona. Dessa forma, esse diagrama permite exibir a topologia de uma rede usada, como também os recursos (máquinas).

Figura 2.26 Diagrama de Distribuição



Fonte: Próprio Autor

2.2.9 Diagrama de Tempo

O diagrama de tempo serve para exibir a relação de tempo de execução dos objetos. Esse diagrama foi incluído na versão 2.0 para apresentar os comportamento dos objetos e suas interações em uma escala de tempo, focando as condições que mudam no decorrer desse período.

2.3 Análise Orientada a Objetos

Na Análise Orientada a Objetos os problemas são pensados e expressados como objetos, ao contrário da análise tradicional os quais eram em rotinas e dados e que foram substituídos por métodos (comportamento) e atributos (propriedades). Assim, quando é colocado o problema de desenvolver um sistema na análise orientada a objetos, deve-se pensar como dividir esse problema em objetos.

Tomando como exemplo um sistema acadêmico para uma análise e modelagem teria Alunos, Administradores, Professores, Cursos, Turmas, etc. E a melhor maneira de conceituar estes termos é considerar um objeto do mundo real e mostrar como podemos representá-lo em termos conceitos em POO. Assim, um objeto é um conceito que usamos para representar uma entidade do mundo real. Exemplificando, um Aluno possui nome, data de nascimento, identidade, etc. E além dessas características (propriedades), possuem ações (métodos) como frequentar as aulas, fazer as avaliações, etc.

Em termos de POO para podermos tratar os objetos temos que criar classes. Assim, uma classe representa um conjunto de objetos que possuem comportamentos e características comuns.

Na UML em primeiro lugar denomina-se uma classe e recomenda-se nomeá-la capitalizando-a e deixando no singular. E em segundo lugar denominam-se as propriedades, informações específicas relacionadas a uma classe de objeto, isto é características dos objetos que as classes representam. Em terceira e última parte, métodos, que são ações que os objetos de uma classe podem realizar. Assim tem-se um modelo para instanciar quantos objetos forem necessários. Dessa forma os objetos instanciados possuirão todas as características e comportamentos definidos pela classe. Então as classes especificam a estrutura (propriedades) e os comportamentos (operações) dos objetos, que são instâncias das classes.

Geralmente em um sistema de médio porte serão identificadas diversas classes que compõem o sistema. Neste contexto a UML surgiu como uma proposta de ser uma linguagem para modelagem de dados que usava diversos artefatos para representar o modelo de negócio; um destes artefatos é o diagrama de classes. Os diagramas de classes registram atributos e operações de uma classe e as restrições de como os objetos podem ser conectados, descrevendo também os tipos de objetos no sistema e os relacionamentos entre eles e esse podem ser associações e abstrações. Para poder representar a visibilidade dos atributos e operações, a UML usa os seguintes símbolos: + público, visível em qualquer classe; - privado, visível somente dentro da classe; # protegido, na classe e suas subclasses.

Agora que sabemos o conceito e como codificar uma classe, vamos entender sobre objeto. Segundo os pais da UML (Rumbaugh, Jacobson e Booch) um objeto é uma instância de uma classe, isto é, trata-se de uma cópia da classe na memória em tempo de execução (*runtime*), sendo assim, um elemento específico que possui valores (estados) nos atributos (campos). A UML representa um objeto usando um retângulo que o representa, onde o nome da instância do objeto é seguido de dois pontos, seguido do nome da classe, com essa formação sublinhada.

O relacionamento entre classes que retrata as relações entre os objetos, por exemplo, um professor que ministra uma disciplina para alunos numa sala, deve ser modelada em um dos tipos de relações da UML, as quais são: dependência, associação e generalização ou herança. Geralmente as classes não estão isoladas (coesas) e se relacionam entre si (acoplamento). O

relacionamento e a comunicação entre as classes se definem em três tipos: Associações que podem ser de Agregação ou Composição; Generalização ou herança; e Dependências. As associações são relacionamentos estruturais entre instâncias e especificam que objetos de uma classe estão ligados a objetos de outras classes, podendo ser unária, binária, etc. As associações podem existir entre classes ou entre objetos. Uma associação entre a classe Professor e a classe Disciplina (um professor ministra uma disciplina) significa que uma instância de Professor (um professor específico) vai ter uma associação com uma instância de uma Disciplina. Esta relação significa que as instâncias das classes são conectadas, seja fisicamente ou conceitualmente.

Outro aspecto importante da POO é o polimorfismo que é a capacidade de classes mais abstratas possuírem comportamentos diferentes das classes concretas, isto é, duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura), mas comportamentos distintos, especializados para uma das classes derivadas. A decisão sobre qual o método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução, através do mecanismo de ligação tardia. Mas para ser considerado polimorfismo, é necessário que os métodos tenham exatamente a mesma identificação, quer dizer, o mesmo nome de método, sendo utilizado o mecanismo de redefinição de métodos, isto é, sobrescrita dos métodos. A redefinição ocorre quando um método cuja assinatura já tenha sido especificada recebe uma nova definição, ou seja, uma nova implementação em uma classe derivada. O mecanismo de redefinição, juntamente com o conceito de ligação tardia, é a chave para a utilização do polimorfismo. É importante observar que, quando polimorfismo está sendo utilizado, o comportamento que será adotado por um método só será definido durante a execução. A outra forma é a sobrecarga, isto é, um método aplicado a um objeto é selecionado para execução através da sua assinatura e da verificação a qual classe o objeto pertence. Dois ou mais métodos de uma mesma classe podem ter o mesmo nome, desde que suas listas de parâmetros sejam diferentes, constituindo assim uma assinatura diferente. Tal situação não gera conflito, pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos de argumentos do método.

Por fim e não menos importante temos o encapsulamento, que consiste na separação de aspectos internos e externos de um objeto. Este mecanismo é utilizado amplamente para impedir o acesso direto ao estado de um objeto

(seus atributos), disponibilizando externamente apenas os métodos que alteram estes estados.

2.4 Modelagem Orientada a Objetos

O Paradigma Orientado a Objetos tem como processo mais completo para representá-lo o RUP e esse processo possui 4 marcos, os quais são: Marco dos objetivos do ciclo de vida; Marco da arquitetura do ciclo de vida; Marco da capacidade operacional inicial; e Marco do lançamento do produto.

Para começar a modelar os requisitos especificados. O cliente necessita entender o que os projetistas estão construindo e precisam ter condições de inferir os seus conhecimentos nos projetos para atender plenamente suas necessidades. Para isso é necessário estabelecer um canal formal de interação onde a linguagem natural do cliente é transformada em linguagem técnica para a equipe de desenvolvimento. Com esse objetivo a UML se propõe a ser uma linguagem padrão aceita e compreendida por todos os *stakeholders*.

A modelagem utilizando a Linguagem de Modelagem Unificada, mais conhecida através da sigla em inglês UML que significa *Unified Modeling Language* é focada no paradigma Orientado a Objetos (OO) cujos conceitos são classe, objeto, herança, polimorfismo, encapsulamento de atributos e métodos, alta coesão e baixo acoplamento. É usado para a análise não focando na codificação do *software* ou *hardware* e sim no entendimento do problema (análise) e na sua solução (projeto).

A UML representa símbolos, esses usados em diagramas que assim representam uma linguagem simbólica com regras claras e precisas para utilização desses símbolos nos diversos diagramas. O objetivo dos diagramas é apresentar múltiplas visões do sistema chamado de modelo. Assim, um modelo UML é um conjunto de diagramas que servem para compreender e desenvolver um projeto de *software*, descrevendo o que o *software* deve fazer.

A *Unified Modeling Language* (UML) é a sucessora de uma onda de métodos para análise e projeto orientados a objetos que surgiram no fim da década de 80. Ela unificou os métodos proposto por Booch, Rumbaugh e Jacobson. Atualmente a UML é um padrão mantido pela OMG, o que dá a ela um status de linguagem padrão para modelagem de sistemas, isto é, uma linguagem padrão para visualizar, especificar, construir e documentar artefatos de um

sistema de software e combina diversos aspectos como: Modelagem de Dados; Modelagem de Negócios; Modelagem de Objetos; e Modelagem de Componentes

É importante enfatizar que UML é uma linguagem de modelagem, ou seja, é uma forma de especificar um sistema de maneira visual. Ela é a linguagem padrão de modelagem adotada pelo *Rational Unified Process* (RUP) e seu grande sucesso se deu em conjunto com a popularização do RUP. Por isso, é muito comum que as pessoas associem UML ao RUP. É importante distinguir muito bem, visto que as duas representam conceitos complementarmente diferentes. O RUP é um processo, ou seja, um conjunto de atividades, com seus respectivos artefatos e papéis, para se desenvolver produtos de *software* e a UML é uma linguagem para se modelar o comportamento e estrutura de um sistema.

Na UML o diagrama de caso de uso segundo Ivan Jacobson, um caso de uso é "um documento narrativo que descreve a sequência de eventos de um ator que usa um sistema para completar um processo". Um caso de uso é uma técnica de modelagem que descreve o que o novo sistema deve fazer e se baseia em um processo de licitação de requisitos entre os *stakeholders* na condução de uma especificação na qual todos interagem.

Um caso de uso descreve quais comportamentos o sistema deverá responder para cada um dos usuários do mesmo, servindo de formalização das ações que precisam ser desenvolvidas. Retratando uma lista de eventos entre os usuários e o sistema em uma visão abstrata, onde essa lista de eventos relatados abstratamente descreve as interações desde o início da atividade até o fim da mesma.

Casos de uso têm que ser compreensíveis pelos usuários por que só eles sabem o que o sistema precisa fazer. Os casos de uso permitem verificar se o desenvolvedor e o usuário concordam sobre o que o sistema deve fazer. Isso é um problema importante no desenvolvimento de *software*. No mesmo tempo, casos de uso podem servir de "contratos" entre os usuários e os desenvolvedores. Sendo assim os casos de uso devem: Descrever os requisitos funcionais do sistema; Fornecer uma descrição clara do que se deve fazer; e Definir os comportamentos das classes.

É importante salientar que casos de uso não são requisitos. E sim uma visão abstrata desses os quais são compostos pelos seguintes componentes: Ator, que é um papel que solicita eventos ao sistema e recebe feedbacks, onde

cada ator pode participar de vários casos de uso; e Casos de uso são documentos narrativos que descrevem a sequência dos eventos feitos por um ator no uso do sistema. Os elementos que descrevem esses papéis são exibidos abaixo. Vale salientar que o nome dos casos de uso deve ser composto por frases curtas, preferencialmente verbo + substantivo. Exemplo: matricular aluno.

Para especificar um caso de uso pode-se usar a estrutura que se segue, porém não existe uma estrutura rígida: Código do caso de uso; Nome do caso de uso; Descrição do caso de uso (um parágrafo); Lista dos nomes dos atores com descrição curta; Prioridade, caso de uso muito importante no projeto ou acessório; Pré-condições; Pós-condições; Fluxo de eventos principal; Fluxos de eventos alternativos; Fluxo de eventos de exceções; Cenários; e Dicionário de Dados. Segue o *template* para especificação de Casos de Uso, [clique aqui](http://www.wthreex.com/rup/portugues/webtmpl/templates/req/rup_ucspec.htm) que está disponível no *link* que se segue < http://www.wthreex.com/rup/portugues/webtmpl/templates/req/rup_ucspec.htm > e para se obter o *template* mais detalhado da especificação de casos de uso, [clique aqui](http://goo.gl/5yKpBD) ou acesso o *link* disponível no site wthreex para baixá-lo < <http://goo.gl/5yKpBD> >

Para identificar os atores que vão participar do modelo uma das técnicas é fazer as seguintes perguntas: Quem usa o sistema? Quem inicia o sistema? Quem fornece os dados? Quem usa as informações? Na descrição dos atores geralmente se faz o uso de: Nome do caso de uso; Tipo de uso se é frequente, ocasional; e Descrição de seu papel no sistema. Para identificar os casos de uso, normalmente a técnica usada é: Verificar as interações entre os atores e o sistema; e Verificar as ações do ator e do sistema.

É importante salientar que os atores sempre iniciam a ação. Para entendimento imagine o requisito em que um aluno necessita fazer um curso, onde existe a necessidade de um sistema de matrículas. Então: Quais são os atores? Quem usa o sistema? Como podemos identificar o caso de uso? Podemos chamar este caso de uso de: Matricular Aluno (verbo+substantivo). Para fazer uma descrição textual do caso de uso, que possui dois atores Aluno e Administração. Sua especificação poderia ser: Caso de uso: Matricular Aluno. Atores: Aluno, Administrador. Descrição: Este caso de uso começa quando um aluno chega à administração que faz a matrícula do aluno no curso. Fluxo: P1-O sistema registra a matrícula; P2-Recebe o pagamento; P3-Emite uma nota fiscal e P4-O aluno é matriculado.

Importante: o caso de uso deve começar com um verbo, enfatizando o processo. Exemplo: Matricular Aluno ou Cadastrar Curso. Deve-se identificar o ator que inicia o evento, começando a descrição da sequência de um caso de uso usando o esquema - este caso de uso começa quando um aluno faz uma matrícula em um curso. Verifique que um ator é um elemento externo ao sistema. Um ator é um papel que interage com o sistema, mas não faz parte dele.

O primeiro passo é identificar os casos de uso básicos, o próximo passo é identificar o relacionamento entre os casos de uso através das generalizações, inclusões e extensões. As inclusões são um caso de uso que inclui o comportamento de outro. A finalidade básica da inclusão é realizar uma decomposição funcional e reduzir a complexidade de um caso de uso. As extensões definem pontos que adicionam um comportamento opcional, assim o caso de uso base pode ser executado mesmo que a extensão não o seja. De princípio as generalizações são mais indicadas para as características dos atores, porém podem ser usadas com cautela para indicar especializações de comportamentos.

No diagrama de classes na Programação Orientada a Objetos (POO), os problemas são pensados e expressados como objetos, ao contrário da análise tradicional os quais eram em rotinas e dados, que aqui foram substituídos por métodos (comportamento) e atributos (propriedades). Assim, quando é colocado o problema de desenvolver um sistema acadêmico na análise orientada a objetos, deve-se pensar como dividir esse problema em objetos. Assim teríamos Alunos, Administradores, Professores, Cursos, Turmas, etc. E a melhor maneira de conceituar estes termos é considerar um objeto do mundo real e mostrar como podemos representá-lo em termos conceitos em POO.

Assim, um objeto é um conceito que usamos para representar uma entidade do mundo real. Exemplificando, um Aluno possui nome, data de nascimento, identidade, etc. E além dessas características (propriedades), possuem ações (métodos) como frequentar as aulas, fazer as avaliações, etc. Em termos de POO para podermos tratar os objetos temos que criar classes. Assim, uma classe representa um conjunto de objetos que possuem comportamentos e características comuns.

Na UML em primeiro lugar denomina-se uma classe e recomenda-se nomeá-la capitalizando-a e deixando no singular. E em segundo lugar denominam-se as propriedades, informações específicas relacionadas a uma classe de

objeto, isto é características dos objetos que as classes representam. Em terceiro e última parte, métodos, que são ações que os objetos de uma classe podem realizar. Assim tem-se um modelo para instanciar quantos objetos forem necessários. Dessa forma os objetos instanciados possuirão todas as características e comportamentos definidos pela classe. Então as classes especificam a estrutura (propriedades) e os comportamentos (operações) dos objetos, que são instâncias das classes.

Geralmente em um sistema de médio porte serão identificadas diversas classes que compõem o sistema. Neste contexto a UML surgiu como uma proposta de ser uma linguagem para modelagem de dados que usava diversos artefatos para representar o modelo de negócio; um destes artefatos é o diagrama de classes.

Os diagramas de classes registram atributos e operações de uma classe e as restrições de como os objetos podem ser conectados, descrevendo também os tipos de objetos no sistema e os relacionamentos entre eles e esse podem ser associações e abstrações. Para poder representar a visibilidade dos atributos e operações, a UML usa os seguintes símbolos: + público, visível em qualquer classe; - privado, visível somente dentro da classe; # protegido, na classe e suas subclasses.

O relacionamento entre classes retrata as relações entre os objetos. Exemplo: um professor ministra uma disciplina para alunos numa sala. A UML reconhece três tipos mais importantes de relações: dependência, associação e generalização ou herança. Geralmente as classes não estão isoladas (coesas) e se relacionam entre si (acoplamento). O relacionamento e a comunicação entre as classes se definem em três tipos: Associações que podem ser de Agregação ou Composição; Generalização ou herança; e Dependências.

As associações são relacionamentos estruturais entre instâncias e especificam que objetos de uma classe estão ligados a objetos de outras classes, podendo ser unária, binária, etc. As associações podem existir entre classes ou entre objetos. Uma associação entre a classe Professor e a classe Disciplina (um professor ministra uma disciplina) significa que uma instância de Professor (um professor específico) vai ter uma associação com uma instância de uma Disciplina. Esta relação significa que as instâncias das classes são conectadas, seja fisicamente ou conceitualmente. Dependências são relacionamentos de utilização no qual uma mudança na especificação de um elemento pode alterar a especificação do elemento dependente. A

dependência entre classes indica que os objetos de uma classe usam serviços dos objetos de outra classe. Generalização ou herança, que pode ser simples ou múltipla (composta), serve para relacionar um elemento mais geral e um mais específico, onde o elemento mais específico herda as propriedades e métodos do elemento mais geral. Como a relação de dependência, ela existe só entre as classes. Um objeto particular não é um caso geral de outro objeto, apenas classes podem receber esse conceito. Agregação é um tipo de associação (é parte de ou todo-parte) onde o objeto parte é um atributo do todo, onde o objeto parte somente são criados se o todo ao qual estão agregados seja criado. Pedidos é composto por itens de pedidos. Composição é o relacionamento entre um elemento (o todo) e outros elementos (as partes) onde as partes só podem pertencer ao todo e são criadas e destruídas com ele.

Resumindo, temos as seguintes formas de relacionamentos: Generalização: É um relacionamento de especialização/generalização, nos quais os objetos dos elementos especializados (os filhos) são substituíveis por objetos do elemento generalizado (os pais); Associação: É um relacionamento estrutural que descreve um conjunto de ligações, em que as ligações são conexões entre objetos.

Agregação: É um tipo especial de associação representando um relacionamento estrutural entre o todo e sua parte; Composição: É outro tipo especial de associação representando um relacionamento estrutural entre um elemento (o todo) e outros elementos (as partes) onde as parte só podem pertencer ao todo e são criadas e destruídas com ele; Realização: É um relacionamento semântico entre classes, em que uma classe especifica um contrato que outra classe garante executar; e Dependência: Relacionamento semântico entre duas classes, nos quais a alteração de um (a classe independente) pode afetar a semântica da outra (a classe dependente).

O diagrama de classes lista todos os conceitos do domínio que serão implementados no sistema e as relações entre os conceitos. Ele é muito importante, pois define a estrutura do sistema a desenvolver. O diagrama de classes é consequência do prévio levantamento de requisitos, definição de casos de usos e classes. Como exemplo tem os passos de elicitação de requisitos com os *stakeholders* do sistema a ser desenvolvido, usando a técnica de entrevista com os administradores, professores, etc. que a partir desses os objetos do sistema são definidos: Alunos, Professores, Turmas, Cursos, etc. Definição dos atores do sistema: aluno, professor,

administrador, etc. Definição e detalhamento dos casos de uso: Matricular Aluno, Pagar Matrícula, etc. Definição das classes: alunos, professor, etc.

Atributo representa uma propriedade que todos os objetos da classe têm, porém cada objeto terá valores particulares para seus atributos. A UML, o nome de um atributo é um texto que deve capitalizar todas as primeiras letras de cada palavra no nome menos a primeira palavra.

Todos os métodos têm que respeitar exatamente a assinatura que é composta pelo nome, número de parâmetros, tipos de dados e ordem. Um método não pode acrescentar ou cortar um parâmetro. Para mandar a mensagem corretamente, devesse saber qual é a classe do objeto, já que cada classe tendo método com assinatura diferente.

Mais uma informação. Devido aos questionamentos de alguns alunos sobre a classificação em relação ao tipo de generalização, complementei essa aula especificando essa tipificação. Ela pode ser basicamente de dois tipos: Total ou Parcial. Se Total temos para cada ocorrência da uma classe genérica sempre uma ocorrência em uma das classes especializadas. Se Parcial temos que nem sempre uma ocorrência da classe genérica possui uma ocorrência correspondente em uma classe especializada. Essa tipificação é muito interessante para a camada de persistência e pode ser também utilizada na estereotipação de entidades.

Os demais diagramas são tão importantes quanto os anteriores, porém menos usados, pois sua aplicação se dá para esclarecer outros aspectos da modelagem e só quando se fazemos necessidade real são desenvolvidos. Esses são os diagramas de sequência que serve para exibir as interações entre os vários componentes de um sistema em especial os objetos e como seus métodos interagem entre si e em qual ordem (sequência). O diagrama de comunicação que serve para representar os elementos de um sistema que trabalham em conjunto. Esse diagrama expressa essa comunicação, mostrando a troca de mensagens entre os objetos. Ele é similar ao diagrama de sequência, porém mais simples. Os diagramas de sequência e de comunicação mostram as interações entre os objetos, por este motivo a UML se refere a estes diagramas como diagramas de interação. O diagrama de atividade que exibe a forma que um objeto executa suas ações em um único processo, representando-os passo a passo, isto é, seu fluxo. As atividades que ocorrem em um caso de uso ou no comportamento do objeto seguem a sequência conforme descrito do diagrama. O diagrama de estados que tem a finalidade de exibir como um objeto realiza uma determinada operação num

determinado momento da execução, representando um estado particular. Exemplo: um aluno pode estar com a situação de aprovado, aprovado na final ou reprovado. O diagrama de componentes que mostra a organização dos componentes na implantação do sistema, mostrando os elementos (componentes) reutilizáveis de *software* e suas interdependências. Vale salientar que um componente é composto por um conjunto de classes. Na maioria das vezes um componente depende de outros, da mesma forma que as classes que ele possui, dependem das funcionalidades de outras classes do mesmo ou de outro componente. Desta forma o diagrama de componentes mostra esta dependência. E o diagrama de implantação que nos mostra a configuração física sobre qual o sistema será instalado. Esses diagramas servem para exibir a distribuição de *hardware* do projeto de *software*, identificando os recursos computacionais como pontos (nós) além da rede que os relaciona. Dessa forma, esse diagrama permite exibir a topologia de uma rede usada, como também os recursos (máquinas).

2.5 Exemplos

Para melhor entender como realizar uma análise orientada a objetos fazendo uso da Linguagem de Modelagem Unificada, segue um exemplo baseado na necessidade de se controlar as solicitações de serviços da empresa ABC. Especificação de Requisitos Funcionais:

- RF01. A empresa ABC necessita de uma aplicação que controle: o seu cadastro de seus clientes; o seu cadastro dos técnicos; e as solicitações de serviços realizadas.
- RF02. Para cada cliente são cadastrados os seguintes dados: código (que deve ser gerado automaticamente); o nome; endereço completo com o logradouro, número, complemento, bairro, cidade, e estado; e telefones.
- RF03. Quando o cliente fizer a primeira chamada por telefone, seus dados devem ser atualizados.
- RF04. Para o técnico deve se cadastrar: o nome; o CPF; endereço residencial completo; telefones; e data de admissão. Quando o técnico for desligado da empresa, deve se cadastrar a data da rescisão contratual.

- RF05. Quando o cliente solicitar um atendimento, deve-se cadastrar a identificação do cliente; a data e hora da solicitação; e o problema a ser solucionado; o status da solicitação deve ficar em “S” que significa Solicitado.
- RF06. Após a solicitação realizada ela deve ser atendida por um dos técnicos colocando a mesma no status “A” de atendimento e informando qual o técnico está fazendo esse atendimento, sendo registrada a data e hora do início do atendimento.
- RF07. Depois de realizado o atendimento o cliente deve ser avisado sobre o parecer deste escrito pelo técnico e o status deverá passar para “E” de encerrado sendo registrada a data e hora do encerramento.
- RF08. Caso alguma irregularidade ocorra nessa solicitação, o status deverá passar para “C” que indica cancelado, registrando o motivo do cancelamento.

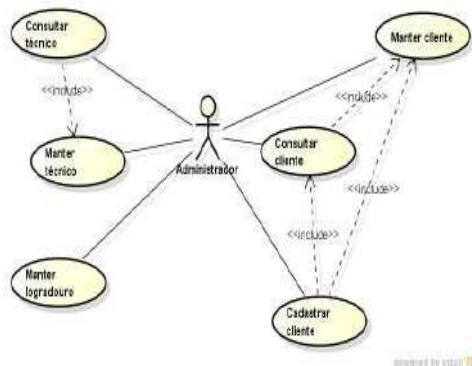
Figura 2.27 Especificação Resumida de Casos de Uso

Caso de Uso Consultar Cliente
Código: UC01
Nome: Consultar Cliente
Descrição: Tem a finalidade de exibir os clientes cadastrados possibilitando as opções de inclusão, alteração e exclusão.
Ator: Administrador
Fluxo Principal
1. Exibição de lista de todos os clientes cadastrados.
2. Oferece ao usuário:
2.1. Selecionar um cliente, para alterar seu cadastro;
2.2. Localizar um cliente ou conjunto de clientes por meio de pesquisa;
2.3. Escolher a opção de “ inserir cliente”.
3. Pesquisar. cliente
3.1. Encontrar cliente através do nome ou parte dele.
3.2. O sistema exibe a lista de clientes que satisfação ao critério, exibindo:
3.2.1. Código de identificação;
3.2.2. Nome do cliente;
3.2.3. Telefone.
4. Inserção de Cliente
4.1. Inclui Caso de Uso Manter Cliente
5. Seleção de Cliente

- 5.1. Selecionar um cliente habilitando as opções de Alterar Cliente e Excluir Cliente.
- 5.2. Se selecionado usar Include Caso de Uso Manter Cliente.

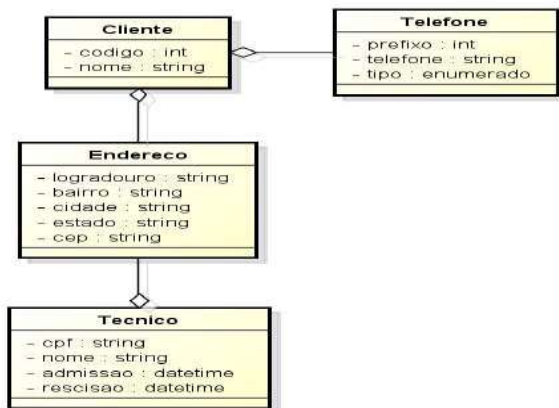
Fonte: Próprio Autor

Figura 2.28 Exemplo de Diagrama de Casos de Uso Consultar Cliente



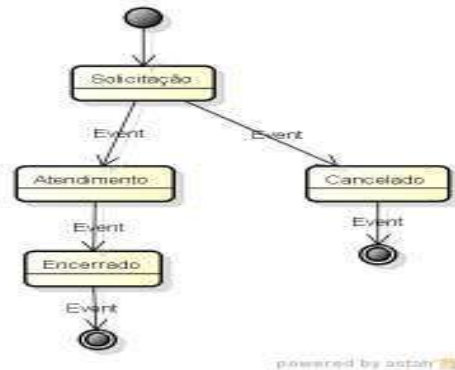
Fonte: Próprio Autor

Figura 2.29 Exemplo de Diagrama de Classes Consultar Cliente



Fonte: Próprio Autor

Figura 2.30 Exemplo de Diagrama de Estados Consultar Cliente



Fonte: Próprio Auto

2.6 Ferramentas

Astah Change-Vision. Em < <http://astah.change-vision.com/en/index.html> >

Astah Community. Em < <http://astah.change-vision.com/en/product/astah-community.html> >

Astah Profissional. Disp.:< <http://astah.change-vision.com/en/product/astah-professional.html> >

Astah Share. Em < <http://astah.change-vision.com/en/product/astah-share.html> >

Astah UML. Em < <http://astah.change-vision.com/en/product/astah-uml.html> >

CASE tools. Em < <http://case-tools.org/uml.html> >

Manual do Astah. Em < <http://goo.gl/z9GrFB> >

Plugin de modelagem UML no NetBeans 6.9. Em < <http://goo.gl/tYFMDL> >

Plugin diversos para NetBeans 6.9. Disp.:< <http://dlc.sun.com.edgesuite.net/netbeans/6.9/rc/zip.html> >

2.7 Apresentações (pptx)

UML. Em < <https://pt.slideshare.net/alvarofpinheiro/alvarofpinheirouml> >

2.8 Vídeos

Introdução a Compiladores. Em < <http://vimeo.com/112860457> >

2.9 Sugestões de Leitura

Análise e Projeto de Sistemas. Em < <http://goo.gl/kQVR7L> >
Desenvolvendo Aplicações com UML 2.2. Em < <http://goo.gl/vVW6IL> >
Engenharia de Software. Em < <http://goo.gl/qZVIYT> >
Exercitando Modelagem com UML. Em < <http://goo.gl/r7toX5> >
Fundamentos de Programação. Em < <http://goo.gl/xwYOfC> >
Introdução a Informática. Em < <http://goo.gl/LPT4Sc> >
O Homem A Administração A Sociedade. Em < <http://goo.gl/MSkGE9> >
UML Essencial. Em < <http://goo.gl/cdLkbT> >
UML Guia do Usuário. Em < <http://goo.gl/av5BTh> >
Utilizando UML e padrões. Em < <http://goo.gl/1YPJ0I> >

2.10 Referências Web

A Theory of Object-Oriented Design. Disp.:< [http://www.eden-study.org/articles/2002/isf4\(4\).pdf](http://www.eden-study.org/articles/2002/isf4(4).pdf) >
ANSI site oficial. Em < <http://www.ansi.org/> >
ANSI wiki. Em < http://pt.wikipedia.org/wiki/American_National_Standards_Institute >
Aulas de UML. Em < <http://goo.gl/2TRbUF> >
Design Patterns Languages for Object-Oriented Programs (DP). Disp.:< <http://www.vincehuston.org/dp/> >
Exemplo de Modelagem UML com a ferramenta ASTAH. Em < <http://goo.gl/JvqhSt> >
Exemplo de Modelagem UML. Em < <http://goo.gl/zw0YCF> >
Guia Básico sobre UML. Em < <http://goo.gl/eGhizM> >
Guia da UML. Em < <http://goo.gl/3Wtx0b> >
Guia de Modelagem de Casos de Uso. Em < <http://goo.gl/PtwH3L> >
Guia de Modelagem de Negócios. Em < <http://goo.gl/W2U0B6> >
Guia de Referência de Convenções de Codificação Java. Em < <http://goo.gl/PfvJcb> >
Guia de Design. Em < <http://goo.gl/OpvFEC> >
Guia de Programação. Em < <http://goo.gl/NL4sX9> >
Guia de Teste. Em < <http://goo.gl/hKmKE1> >
IBM Rational Unified Process. Em < ftp://public.dhe.ibm.com/software/pdf/br/RUP_DS.pdf >
ISO site oficial. Em < <http://www.iso.org/iso/home.html> >
ISO wiki. Em < http://en.wikipedia.org/wiki/International_Organization_for_Standardization >
Langpop. Em < <http://www.langpop.com/> >
Mais sobre relacionamentos de Inclusão e Extensão. Em < <http://goo.gl/uH8X6Z> >

Manuais de Referência OMG-UML. Em < <http://www.omg.org/spec/UML/> >
Modelagem UML. Em < <http://goo.gl/iwtZh3> >
Object Management Group (OMG). Em < <http://www.omg.org/> >
Object-Oriented Programming (OOP). Em < <http://goo.gl/rIYfIs> >
Ohloh. Em < <http://www.ohloh.net/> >
Paradigma (wiki). Em < <http://pt.wikipedia.org/wiki/Paradigma> >
Paradigma de Desenvolvimento (wiki). Em < <http://goo.gl/rIYfIs> >
Rational Unified Process (RUP). Em < <http://www-01.ibm.com/software/br/rational/> >
Standish Group Report Chaos 2009. Disp.: < http://www1.standishgroup.com/newsroom/chaos_2009.php >
Standish Group. Em < <http://www.standishgroup.com/> >
Template Casos de Uso. Disp.: < http://www.wthreex.com/rup/portugues/webtmpl/templates/req/rup_ucspec.htm >
Template Mais Detalhado para Especificação de Casos de Uso. Disp.: < <http://goo.gl/5yKpBD> >
Template RUP para Especificação de Casos de Uso. Em < <http://goo.gl/GwnXCn> >
Tiobe. Em < <http://www.tiobe.com/index.php/content/company/Home.html> >
Um Mundo Feito (quase completamente) de Software. Em < <http://goo.gl/YE26CS> >
Unified Modeling Language (UML). Em < <http://www.uml.org/> >

2.11 Referências Bibliográficas

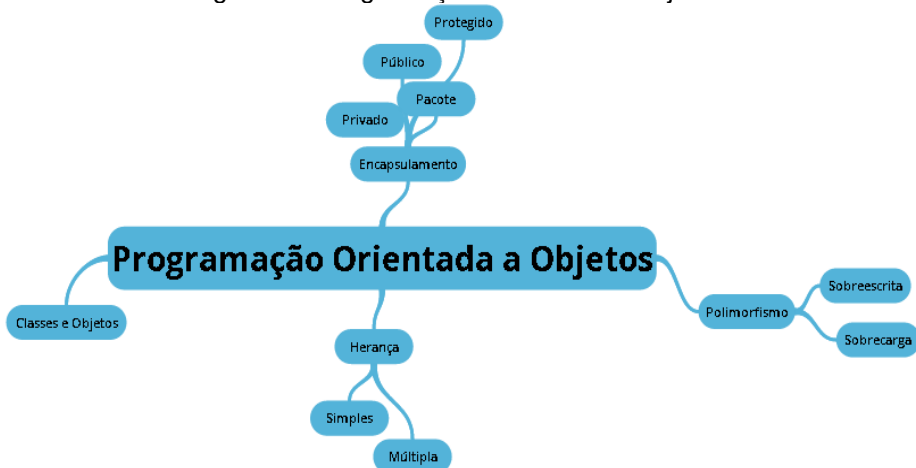
BENNETT, Simon; *et al.* Schaum's Outline of UML. 1ªed. UK: McGraw-Hill, 2005.
BEZERRA, Eduardo. Princípios de Análise e Projetos de Sistemas com UML. 1ªed. Rio de Janeiro: Campus, 2007.
CARDOSO, Caíque. UML na Prática: do problema ao sistema. 1ªed. São Paulo: CM, 2003.
DEMARCO, Tom; LISTER Timothy. Peopleware Productive Projects and Teams. US: Dorset House Pub, 1999.
DENNIS, Alan. Análise e Projeto de Sistemas. 2ªed. São Paulo: LTC, 2011.
DRUCKER, Peter. Management: Tasks, Responsibilities, Practices. 1ªed. US: Transaction Publishers, 2007.
FOWLER, MARTIN. UML Essencial. 2ªed. Porto Alegre: Bookman, 2004.
GAMMA, Erich; *et al.* Padrões de Projeto. Porto Alegre: Bookman, 2007.
KRUCHTEN, Philippe. Introdução ao RUP Rational Unified Process. 1ªed. Rio de Janeiro: Ciência Moderna, 2003.

- MEDEIROS, Ernani. Desenvolvendo Software com UML 2.0: Definitivo. 1ªed. São Paulo: Makron Books, 2004.
- MELO, Ana. Desenvolvendo Aplicações com UML 2.2: do conceitual à implementação. 3ªed. RJ: Brasport, 2010.
- PENDER, Tom. UML - A Bíblia. 1ªed. RJ: Campus, 2004.
- PRESSMAN, Roger. Engenharia de Software: uma abordagem profissional. 7ªed. Porto Alegre: Bookman, 2011.
- RUMBAUGH, James; *et al.* Modelagem e Projetos Baseados em Objetos com UML. 2ªed. RJ: Campus, 2006.
- RUMBAUGH, James; JACOBSON, Ivar; BOOCH, Grady. UML: Guia do Usuário. 1ªed. RJ: Campus, 2006.
- SOMMERVILLE, Ian. Engenharia de Software. 8ªed. Pearson, 2007

3 Linguagem de Programação Java

Java é uma linguagem de programação que usa o paradigma orientado a objetos e que foi criada na década de 90 por James Gosling da Sun Microsystems, que não gera código nativo, mas sim um *bytecode* que é executado por uma máquina virtual. A linguagem Java possui diversas características muito úteis para o desenvolvimento Web, as quais podemos citar a orientação a objetos baseado no Simula67; portabilidade; APIs para rede; segurança; sintaxe semelhante à linguagem C; e liberação de memória automática. Como sugestão de leitura para mais uma visão do que é a linguagem Java seguem dois bons *links* o [Wiki](#) e [WikiBook](#). Antes de começar é importante lembrar que este material é uma introdução ao Java e que se o objetivo é se aprofundar nesta linguagem aconselho o seguinte livro Java: Como Programar de Harvey Deitel e Paul Deitel. Antes de iniciarmos com o assunto observe a figura 10.1 para ter uma visão geral desse capítulo.

Figura 3.1 Programação Orientada a Objetos



Fonte: Próprio Autor

3.1 Conceitos

BYTECODE é um código intermediário que serve para ser interpretado pela Máquina Virtual Java (MVJ), essa característica faz com que os programas

Java sejam independentes de sistemas operacionais. Cada *opcode* tem o tamanho de um byte, por isso é chamado de *bytecode*.

MVJ é um programa que carrega e executa códigos Java, convertendo os *bytecodes* em código executável do sistema operacional instalado, sendo responsável pelo gerenciamento dos aplicativos, à medida que são executados. A arquitetura de uma MVJ permite um controle sofisticado sobre o código que está sendo executado.

APPLETS são códigos que rodam dentro de uma VM incorporada ao browser do usuário, executando código baixado de um servidor HTTP remoto. O código remoto roda em uma *sandbox*, que protege o usuário de códigos maliciosos. O autor do *applet* pode aplicar um certificado para assinar digitalmente o *applet* como "seguro", dando a ele permissão de sair do *sandbox* e acessar livremente a máquina onde está rodando.

J2SE (Java 2 *Standard Edition*) é uma ferramenta de desenvolvimento para a plataforma Java. Ela contém todo o ambiente necessário para a criação e execução de aplicações Java, incluindo a máquina virtual Java (JVM), o compilador Java, as APIs do Java e outras ferramentas utilitárias.

J2EE (Java 2 *Enterprise Edition*) inclui toda a funcionalidade existente na plataforma Java SE mais todas as funcionalidades necessárias para o desenvolvimento e execução de aplicações em um ambiente corporativo, adicionando de bibliotecas que fornecem funcionalidade para implementar *software* Java distribuído, tolerante a falhas e multicamada, baseada amplamente em componentes modulares executando em um servidor de aplicações.

J2ME (Java 2 *Micro Edition*) é uma tecnologia que possibilita o desenvolvimento de *software* para sistemas e aplicações embarcadas, ou seja, toda aquela que roda em um dispositivo de propósito específico, desempenhando alguma tarefa que seja útil para o dispositivo. É a plataforma Java para dispositivos compactos, como celulares, PDAs, controles remotos, e outra gama de dispositivos. Java ME é uma coleção de APIs do Java definidas através da JCP (Java *Community Process*).

Os *containers* Java são:

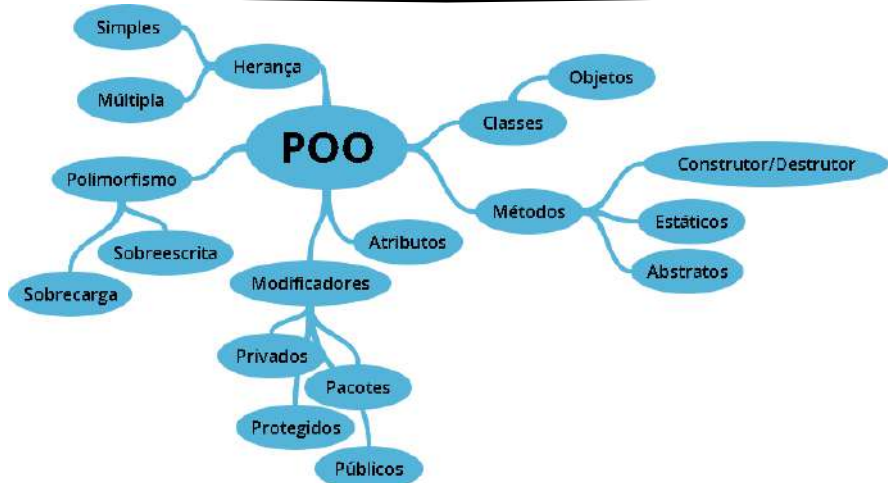
- JDBC (Java *Database Connectivity*), utilizado no acesso a bancos de dados;

- *Servlets* são utilizados para o desenvolvimento de aplicações Web com conteúdo dinâmico. Ele contém uma API que abstrai e disponibiliza os recursos do servidor Web de maneira simplificada para o programador;
- JSP (*Java Server Pages*), uma especialização do *servlet* que permite que conteúdo dinâmico seja facilmente desenvolvido;
- JTA (*Java Transaction API*) é uma API que padroniza o tratamento de transações dentro de uma aplicação Java;
- EJBs (*Enterprise Java Beans*), utilizados no desenvolvimento de componentes de *software*. Eles permitem que o programador se concentre nas necessidades do negócio do cliente, enquanto questões de infraestrutura, segurança, disponibilidade e escalabilidade é responsabilidade do servidor de aplicações;
- JCA (*Java Connector Architecture*) é uma API que padroniza a ligação a aplicações legadas; e
- JPA (*Java Persistence API*), é uma API que padroniza o acesso a banco de dados através de mapeamento Objeto/Relacional dos *Enterprise Java Beans*.

3.2 Programação Orientada a Objetos com Java

Para iniciarmos o estudo da programação Java vamos começar com o entendimento do conceito de classe. Na Programação Orientada a Objetos (POO), os problemas são pensados e expressados como objetos, ao contrário da análise tradicional os quais eram em rotinas e dados, que aqui foram substituídos por métodos (comportamento) e atributos (propriedades). Assim, quando é colocado o problema de desenvolver um sistema acadêmico na análise orientada a objetos, deve-se pensar como dividir esse problema em objetos. Observe a figura 10.2 para ter uma ideia abrangente dos conceitos.

Figura 3.2 Conceitos de Programação Orientada a Objetos com Java



Fonte: Próprio Autor

3.2.1 Classes

Assim teríamos Alunos, Administradores, Professores, Cursos, Turmas, etc. E a melhor maneira de conceituar estes termos é considerar um objeto do mundo real e mostrar como podemos representá-lo em termos conceituais em POO. Assim, um objeto é um conceito que usamos para representar uma entidade do mundo real. Exemplificando, um Aluno possui nome, data de nascimento, identidade, etc. E além dessas características (propriedades), possuem ações (métodos) como frequentar as aulas, fazer as avaliações, etc. Em termos de POO para podermos tratar os objetos temos que criar classes. Assim, uma classe representa um conjunto de objetos que possuem comportamentos e características comuns.

Na UML em primeiro lugar denomina-se uma classe e recomenda-se nomeá-la capitalizando-a e deixando no singular. E em segundo lugar denominam-se as propriedades, informações específicas relacionadas a uma classe de objeto, isto é características dos objetos que as classes representam. Em terceira e última parte, métodos, que são ações que os objetos de uma classe podem realizar. Assim tem-se um modelo para instanciar quantos objetos forem necessários. Dessa forma os objetos instanciados possuirão todas as características e comportamentos definidos pela classe. Então as classes especificam a estrutura (propriedades) e os comportamentos (operações) dos objetos, que são instâncias das classes.

Geralmente em um sistema de médio porte serão identificadas diversas classes que compõem o sistema. Neste contexto a UML surgiu como uma proposta de ser uma linguagem para modelagem de dados que usava diversos artefatos para representar o modelo de negócio; um destes artefatos é o diagrama de classes. Os diagramas de classes registram atributos e operações de uma classe e as restrições de como os objetos podem ser conectados, descrevendo também os tipos de objetos no sistema e os relacionamentos entre eles e esse podem ser associações e abstrações. Para poder representar a visibilidade dos atributos e operações, a UML usa os seguintes símbolos: + público, visível em qualquer classe; - privado, visível somente dentro da classe; # protegido, na classe e suas subclasses.

O relacionamento entre classes retrata as relações entre os objetos. Exemplo: um professor ministra uma disciplina para alunos numa sala. A UML reconhece três tipos mais importantes de relações: dependência, associação e generalização ou herança. Geralmente as classes não estão isoladas (coesas) e se relacionam entre si (acoplamento). O relacionamento e a comunicação entre as classes se definem em 3 tipos: Associações que podem ser de Agregação ou Composição; Generalização ou herança; e Dependências.

As associações são relacionamentos estruturais entre instâncias e especificam que objetos de uma classe estão ligados a objetos de outras classes, podendo ser unária, binária, etc. As associações podem existir entre classes ou entre objetos. Uma associação entre a classe Professor e a classe Disciplina (um professor ministra uma disciplina) significa que uma instância de Professor (um professor específico) vai ter uma associação com uma instância de uma Disciplina. Esta relação significa que as instâncias das classes são conectadas, seja fisicamente ou conceitualmente.

Dependências são relacionamentos de utilização no qual uma mudança na especificação de um elemento pode alterar a especificação do elemento dependente. A dependência entre classes indica que os objetos de uma classe usam serviços dos objetos de outra classe. Generalização ou herança, que pode ser simples ou múltipla (composta), serve para relacionar um elemento mais geral e um mais específico, onde o elemento mais específico herda as propriedades e métodos do elemento mais geral. Como a relação de dependência, ela existe só entre as classes. Um objeto particular não é um caso geral de outro objeto, apenas classes podem receber esse conceito.

Agregação é um tipo de associação (é parte de ou todo-parte) onde o objeto parte é um atributo do todo, onde o objeto parte somente são criados se o todo ao qual estão agregados seja criado. Pedidos é composto por itens de pedidos. Composição é o relacionamento entre um elemento (o todo) e outros elementos (as partes) onde as partes só podem pertencer ao todo e são criadas e destruídas com ele.

O diagrama de classes lista todos os conceitos do domínio que serão implementados no sistema e as relações entre os conceitos. Ele é muito importante, pois define a estrutura do sistema a desenvolver. O diagrama de classes é consequência do prévio levantamento de requisitos, definição de casos de usos e classes. Como exemplo tem os passos de elicitação de requisitos com os *stakeholders* do sistema a ser desenvolvido, usando a técnica de entrevista com os administradores, professores, etc. que a partir desses os objetos do sistema são definidos: Alunos, Professores, Turmas, Cursos, etc. Definição dos atores do sistema: aluno, professor, administrador, etc. Definição e detalhamento dos casos de uso: Matricular Aluno, Pagar Matrícula, etc. Definição das classes: alunos, professor, etc.

Atributo representa uma propriedade que todos os objetos da classe têm, porém cada objeto terá valores particulares para seus atributos. A UML, o nome de um atributo é um texto que deve capitalizar todas as primeiras letras de cada palavra no nome menos a primeira palavra. Todos os métodos têm que respeitar exatamente a assinatura que é composta pelo nome, número de parâmetros, tipos de dados e ordem. Um método não pode acrescentar ou cortar um parâmetro. Para mandar a mensagem corretamente, devesse saber qual é a classe do objeto, já que cada classe tendo método com assinatura diferente. Entendido o conceito de classe, segue uma codificação básica em Java:

```
/**
 * Classe de domínio ClassPessoa
 *
 * @author (Álvaro Pinheiro)
 * @version (01/03/2011)
 */
public class ClassPessoa {
    // variáveis de instância
    private int codigo;
    private String nome;
}
```

```
* Construtor da classe ClassPessoa
*/
public ClassPessoa() {
    // inicialização das variáveis de instância
    codigo = 0;
    nome = "";
}
/**
 * Método para atribuir valor ao campo código
 *
 * @param pcodigo
 * @return codigo
 */
public void setCodigo(int pCodigo) {
    codigo = pCodigo;
}

/**
 * Método para retornar valor do campo código
 *
 * @param nulo
 * @return codigo
 */
public int getCodigo() {
    return codigo;
}
/**
 * Método para atribuir valor ao campo nome
 *
 * @param pnome
 * @return nome
 */
public void setNome(String pNome)
    nome = pNome;
}
/**
 * Método para retornar valor do campo nome
 *
 * @param nulo
 * @return nome
 */
```

```
    public String getNome() {  
        return nome;  
    }  
}
```

3.2.2 Objetos

Agora que sabemos o conceito e como codificar uma classe, vamos entender sobre objeto. Segundo os pais da UML (Rumbaugh, Jacobson e Booch) um objeto é uma instância de uma classe, isto é, trata-se de uma cópia da classe na memória em tempo de execução (*runtime*), sendo assim, um elemento específico que possui valores (estados) nos atributos (campos). A UML representa um objeto usando um retângulo que o representa, onde o nome da instância do objeto é seguido de dois pontos, seguido do nome da classe, com essa formação sublinhada. Muito bem, temos o conceito de objeto agora como se instância um em Java? Segue o código exemplificando:

```
/**  
 * Classe de domínio ClassCliente  
 */  
 * @author (Álvaro Pinheiro)  
 * @version (01/03/2011)  
 */  
public class ClassCliente {  
    /**  
     * Construtor da classe ClassCliente  
     */  
    public ClassCliente() {  
    }  
    /**  
     * Método da classe cliente que exemplifica o instân-  
     * ciamento de dois objetos da classe Pessoa  
     */  
    public void exemplo() {  
        ClassPessoa objCliente1 = new ClassPessoa();  
        ClassPessoa objCliente2 = new ClassPessoa();  
        objCliente1.setCodigo(1);  
        objCliente1.setNome("abc");  
        objCliente2.setCodigo(2);  
        objCliente2.setNome("xyz");  
    }  
}
```

```
}
```

No exemplo acima, temos a seguinte sintaxe: `ClassPessoa objCliente1 = new ClassPessoa();`. O primeiro `ClassPessoa` nos diz de que classe o objeto será instanciado, o nome do objeto nesse exemplo é `objCliente1`, é importante salientar que não pode existir dois ou mais objetos com o mesmo nome. Em seguida temos a atribuição seguida da palavra reservada `new` que irá dar o comando que um novo objeto deve ser instanciado a partir do construtor `ClassPessoa()`. Em seguida com a sintaxe: `objCliente1.setCodigo(1);`. Passamos a atribuir o número 1 ao campo privado código da classe `Pessoa` por intermédio do método público `setCodigo(int pCodigo)` da classe `Pessoa`. Assim, é que se atribui novos valores (estados) aos campos privados de uma classe.

3.2.3 Métodos

Vamos aprender agora a chamar as classes e os métodos escritos em Java a partir de um ponto principal.

```
/**
 * Classe Principal:
 *
 */
class Principal {
/**
 * Método que será chamado pelo método principal
 */
    public void exemplo() {
        System.out.println("Método chamado!");
    }
/**
 * Método principal
 */
    public static void main(String[] args) {
        Principal principal = new Principal();
        principal.exemplo();
    }
}
```

Nessa classe principal temos um método principal "*main*" que por *default* recebe um *array* de caracteres que poderá ser preenchido quando o

programa principal for carregado. Ao carregá-lo uma mensagem irá aparecer na tela "Método chamado!", pois quando um objeto da classe é instanciado o método exemplo é invocado. Pode-se ver na figura 10.3 os tipos de métodos existentes na programação orientada a objetos com Java.

Figura 3.3 Tipos de Métodos em POO



Fonte: Próprio Autor

3.2.4 Pacotes

Um pacote em Java é a forma que se tem para permitir a organização de classes em módulos, isto é, colocar todas as classes de funcionalidades semelhantes e/ou equivalentes em um mesmo lugar físico, chamado de package. Os pacotes Java podem ser armazenados em arquivos compactados denominados arquivos JAR, permitindo assim que classes com mesmas funcionalidades possam ser utilizadas de uma maneira mais rápida e fácil, já que todas as classes estão agrupadas em um mesmo arquivo.

Outra vantagem do uso de pacotes é a utilização do modificador package que disponibiliza para todas as propriedades e comportamentos das classes agrupadas a sua visibilidade. Além de normalmente os pacotes serem utilizados para conter Classes, Interfaces e Tipos Enumerados - agrupando-os pelas suas semelhanças. Para se usar um pacote em uma classe Java deve-se importá-lo e isso é feito com a seguinte declaração `import javax.swing.*`; Com essa declaração é importado para a classe corrente todas as classes que estão dentro do pacote `javax.swing`. Mas, se for desejado pode-se importar somente a classe que se quer usar de um pacote específico, como no exemplo `import Java.awt.event.ActionEvent`; Para se criar um arquivo JAR se faz necessário utilizar um utilitário de linha de comando que se segue `JAR arquivo.class meu pacote.jar`, esse comando compacta todos os arquivos `.class` dentro do arquivo JAR denominado `meuPacote.jar`.

Uma característica importante na criação dos pacotes é a utilização de uma hierarquia baseada na separação dos nomes por ponto (.). Apesar de pacotes inferiores na hierarquia de nomeação são muitas vezes referidos como "subpacotes" dos pacotes correspondentes superior na hierarquia, não há nenhuma relação semântica entre pacotes. O *Java Language Specification* estabelece pacote de convenções de nomenclatura para evitar a possibilidade de dois pacotes publicado com o mesmo nome. As convenções de nomenclatura descrevem como criar nomes de pacote exclusivo, para que os pacotes que são amplamente distribuídos. Isso permite que os pacotes sejam separadamente, de forma fácil e automaticamente instaladas e catalogadas.

Segue o Core dos pacotes no Java SE 6:

- Java.lang - Funcionalidades básicas da língua e tipos fundamentais;
- Java.util - Utilitário;
- Java.io - Operações de entrada e saída;
- Java.math - Funções matemática;
- Java.net - Operações de rede;
- Java.security - Geração de chave de criptografia e descriptografia;
- Java.sql - Drivers JDBC para acessar bancos de dados;
- Java.awt - GUI nativa;
- javax.swing - GUI genérica; e
- Java.applet - Classes para a criação de applets.

Observe o exemplo abaixo, temos a importação do pacote javax.swing que permite entre outras classes, usar a JOptionPane, que disponibiliza os métodos showMessageDialog(null, "texto", "título", ícone) para mostrar uma mensagem na tela e o showInputDialog("texto") para entrar com um valor.

```
/**
 * Classe Principal com pacote swing:
 *
 */
import javax.swing.*;
class Principal
{
    /**
     * Método que será chamado pelo método principal
     */
    public void exemplo()
```

```
{
    System.out.println("Método chamado!");
    String sVar1 = JOptionPane.showInputDialog("Entre com um número:");
    String sVar2 = JOptionPane.showInputDialog("Entre com outro
número:");
    int nVar3 = Integer.parseInt(sVar1);
    int nVar4 = Integer.parseInt(sVar2);
    int nVar5 = nVar3 + nVar4;
    JOptionPane.showMessageDialog(null, "A soma é " + nVar5,
"Operação", JOptionPane.PLAIN_MESSAGE);
}
/**
 * Método principal
 */
public static void main(String[] args)
{
    Principal principal = new Principal();
    principal.exemplo();
}
}
```

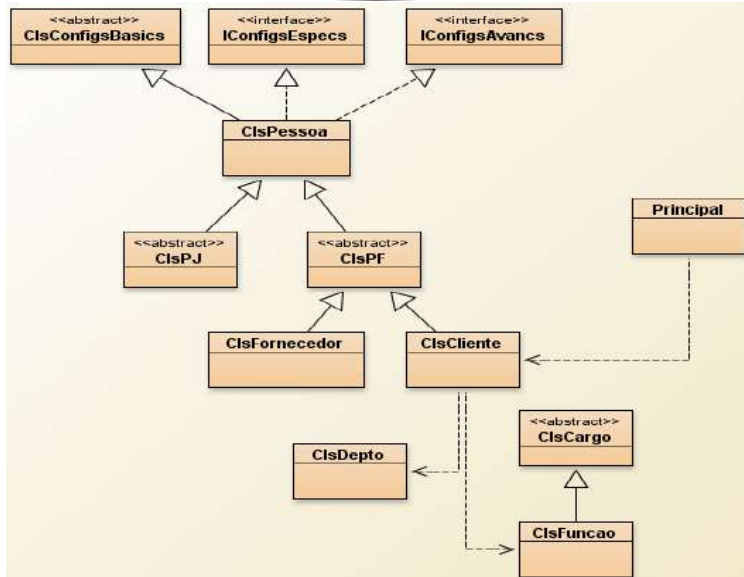
3.2.5 Relacionamentos

O que vamos estudar agora é como em Java se codifica os relacionamentos, isto é as associações e heranças. Também iremos codificar os três tipos de classes: Concreta, Abstrata e Interface. Observe o diagrama de class:

Temos a seguinte modelagem. A classe principal está associada à classe cliente, esta por sua vez está associada às classes departamento e função. A classe função estende a classe abstrata cargo. A classe cliente é uma subclasse da classe pessoa-física que também é pai da classe filha fornecedor.

Ambas as classes, cliente e fornecedor são concretas, assim permitindo que se instanciem objetos dessas classes.

Figura 3.4 Relacionamento em Java



Fonte: Próprio Autor

Já as classes pessoa-física e pessoa-jurídica são abstratas, assim sendo possuem métodos não implementados e por consequência não permitindo que se instanciem objetos diretamente dessas classes. Essas duas últimas classes são filhas da classe concreta pessoa que estende a classe abstrata configurações básicas e implementa (realiza) as interfaces configurações especiais e configurações avançadas. Vamos observar como fica a codificação dessas classes a seguir.

```

/**
 * Exemplo de Classe abstrata
 * @author (Álvaro Pinheiro)
 */
public abstract class ClsConfigsBasics
{
    // variáveis de instância
    private int configBasic1;
    /**
     * Exemplo de método implementado
     */
    public int retornaConfigBasic1()
  
```

```
{
    return (configBasic1);
}
/**
 * Exemplo de método não implementado (abstrato)
 */
public abstract int retornaConfigBasic2();
}
```

```
/**
 * Exemplo de Interface
 * @author (Álvaro Pinheiro)
 */
public interface IConfigsEspecs
{
    /**
     * Exemplo de assinatura de método
     */
    int retornaConfigEspec1(int y);
    /**
     * Exemplo de assinatura de método
     */
    void executaConfigEspec1();
}
```

```
/**
 * Exemplo de Interface
 * @author (Álvaro Pinheiro)
 */
public interface IConfigsAvancs
{
    /**
     * Exemplo de assinatura de método
     */
    int retornaConfigAvanc1(int y);
    /**
     * Exemplo de assinatura de método
     */
    void executaConfigAvanc1();
}
```

```
/**
 * Exemplo de classe que estende classe abstrata e implementa (realiza)
interface
 * @author (Álvaro Pinheiro)
 */
public class ClsPessoa extends ClsConfigsBasics implements
IConfigsEspecs, IConfigsAvancs
{
    // variável de instância
    private int x;
    /**
     * Construtor da classe pessoa
     */
    public ClsPessoa()
    {
        // inicializa variável de instância
        x = 0;
    }
    /**
     * Exemplo de método próprio da classe pessoa
     */
    public int metodoClassePessoa(int y)
    {
        return (x + y);
    }
    /**
     * Exemplo de sobreescrita do método da classe abstrada clsConfigsBasics
     */
    public int retornaConfigBasic2()
    {
        return (0);
    }
    /**
     * Exemplo de sobreescrita do método da interface IConfigsEspecs
     */
    public int retornaConfigEspec1(int y)
    {
        return (1+y);
    }
    public void executaConfigEspec1()
    {

```

```
//ação;
}
/**
 * Exemplo de sobreescrita do método da interface IConfigsAvancs
 */
public int retornaConfigAvanc1(int y)
{
    return (1+y);
}
public void executaConfigAvanc1()
{
    //ação;
}
}

/**
 * Exemplo de classe abstrata
 * @author (Álvaro Pinheiro)
 */
public abstract class ClsPJ extends ClsPessoa
{
    /**
     * Exemplo de método concreto
     */
    public void metodoConcreto()
    {
        //ação
    }
    /**
     * Exemplo de método abstrato
     */
    public abstract void metodoAbstrato();
}

/**
 * Exemplo de classe abstrata
 * @author (Álvaro Pinheiro)
 */
public abstract class ClsPF extends ClsPessoa
{
    /**
```

```
* Exemplo de método concreto
*/
public void metodoConcreto()
{
    //ação
}
/**
 * Exemplo de método abstrato
 */
public abstract void metodoAbstrato();
}

/**
 * Exemplo de classe fornecedor
 * @author (Álvaro Pinheiro)
 */
public class ClsFornecedor extends ClsPF
{
    /**
     * Construtor da classe fornecedor
     */
    public ClsFornecedor()
    {
    }
    /**
     * Método da classe fornecedor
     */
    public void metodoFornecedor()
    {
        //ação
    }
    /**
     * Realização do método abstrato da classe PF
     */
    public void metodoAbstrato()
    {
        //ação
    }
}

/**
```

```
* Exemplo de classe cliente
* @author (Álvaro Pinheiro)
*/
public class ClsCliente extends ClsPF
{
    //variável de instancia
    ClsDepto depto;
    ClsFuncao funcao;
    /**
     * Construtor da classe cliente
     */
    public ClsCliente()
    {
    }
    /**
     * Método da classe cliente
     */
    public void metodoCliente()
    {
        //ação
    }
    /**
     * Realização do método abstrato da classe PF
     */
    public void metodoAbstrato()
    {
        //ação
    }
}

/**
 * Exemplo de classe departamento
 * @author (Álvaro Pinheiro)
 */
public class ClsDepto
{
    /**
     * Construtor
     */
    public ClsDepto()
    {
    }
}
```

```
}
/**
 * Método
 */
public int metodoExemplo()
{
    return (0);
}

/**
 * Exemplo de classe funcao
 * @author (Álvaro Pinheiro)
 */
public class ClsFuncao extends ClsCargo
{
    /**
     * Construtor
     */
    public ClsFuncao()
    {
    }
    /**
     * Método
     */
    public int metodoExemplo()
    {
        return (0);
    }
    /**
     * Realização
     */
    public int metodoAbstrato()
    {
        return (0);
    }
}

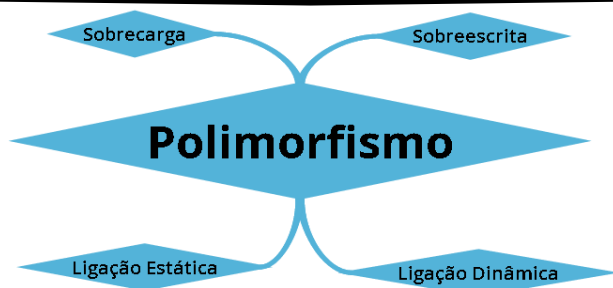
/**
 * Exemplo de classe abstrata cargo
 * @author (Álvaro Pinheiro)
```

```
*/  
public abstract class ClsCargo  
{  
    /**  
     * Método  
     */  
    public abstract int metodoAbstrato();  
}  
  
/**  
 * Classe Principal  
 */  
class Principal  
{  
    /**  
     * Método principal  
     */  
    public static void main(String[] args)  
    {  
        System.out.println("Método principal");  
        ClsCliente cliente = new ClsCliente();  
        cliente.metodoCliente();  
    }  
}
```

3.2.6 Polimorfismo

É a capacidade de classes mais abstratas possuírem comportamentos diferentes das classes concretas, isto é, duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura), mas comportamentos distintos, especializados para uma das classes derivadas. A decisão sobre qual o método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução, através do mecanismo de ligação tardia. Observe a figura 10.5.

Figura 3.5 Polimorfismo



Fonte: Próprio Autor

Mas para ser considerado polimorfismo, é necessário que os métodos tenham exatamente a mesma identificação, quer dizer, o mesmo nome de método, sendo utilizado o mecanismo de redefinição de métodos, isto é, sobrescrita dos métodos.

Override ou Sobreescrita, a redefinição ocorre quando um método cuja assinatura já tenha sido especificada recebe uma nova definição, ou seja, uma nova implementação em uma classe derivada. O mecanismo de redefinição, juntamente com o conceito de ligação tardia, é a chave para a utilização do polimorfismo. É importante observar que, quando polimorfismo está sendo utilizado, o comportamento que será adotado por um método só será definido durante a execução.

Overload ou Sobrecarga, um método aplicado a um objeto é selecionado para execução através da sua assinatura e da verificação a qual classe o objeto pertence. Dois ou mais métodos de uma mesma classe podem ter o mesmo nome, desde que suas listas de parâmetros sejam diferentes, constituindo assim uma assinatura diferente. Tal situação não gera conflito, pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos de argumentos do método.

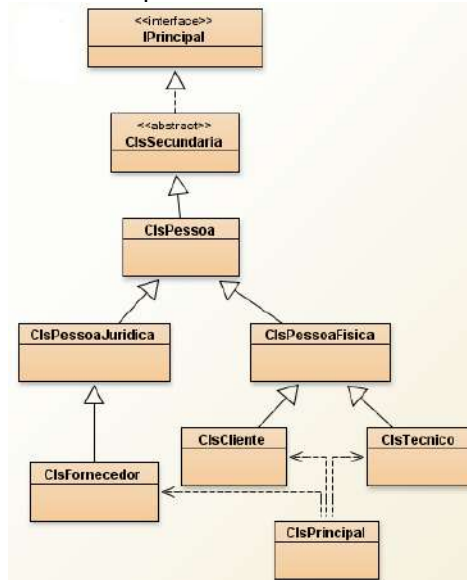
Early Binding ou Ligação Prematura, quando um método é invocado durante a compilação do programa, o mecanismo de ligação é prematura.

Late Binding ou Ligação Tardia, quando a definição do método que será efetivamente invocado só ocorre durante a execução do programa, o mecanismo de ligação usado é o de tardia também conhecido pelos termos *dynamic binding* ou *run-time binding*.

Em Java, todas as determinações de métodos a executar ocorrem através de ligação tardia exceto em dois casos: métodos declarados como `final` não podem ser redefinidos e, portanto não são passíveis de invocação polimórfica da parte de seus descendentes e métodos declarados como `private` são implicitamente finais.

Para melhor entender o polimorfismo, veja a figura 10.6:

Figura 3.6 Exemplo de Relacionamentos na POO



Fonte: Próprio Autor

```
public interface IPrincipal
{
    void verificaAcesso();
    boolean verificaCpf(String pCpf);
}

public abstract class ClsSecundaria implements IPrincipal
{
    public abstract void verificaAcesso();
    public boolean verificaCpf(String pCpf)
```

```
{
    if (true) {
        return true;
    }
    else
    {
        return false;
    }
}
}
```

```
public class ClsPessoa extends ClsSecundaria
{
    private String cpf;
    private String nome;
    public ClsPessoa()
    {
        cpf = "";
        nome = "";
    }
    public ClsPessoa(String pCpf) // exemplo de overload (sobrecarga)
    {
        cpf = pCpf;
        nome = "";
    }
    public ClsPessoa(String pCpf, String pNome) // exemplo de overload
(sobrecarga)
    {
        cpf = pCpf;
        nome = pNome;
    }
    public void verificaAcesso()
    {
        System.out.print("Acesso tipo 1");
    }
}
```

```
public class ClsPessoaJuridica extends ClsPessoa
{
    public ClsPessoaJuridica()
    {

```

```
}  
}  
  
public class ClsPessoaFisica extends ClsPessoa  
{  
    public ClsPessoaFisica()  
    {  
    }  
    public void verificaAcesso() // exemplo de override (sobrescrita)  
    {  
        System.out.print("Acesso tipo 2");  
    }  
}  
  
public class ClsFornecedor extends ClsPessoaJuridica  
{  
    public ClsFornecedor()  
    {  
    }  
}  
  
public class ClsCliente extends ClsPessoaFisica  
{  
    public ClsCliente()  
    {  
    }  
    // exemplo de herança e encapsulamento  
}  
  
public class ClsTecnico extends ClsPessoaFisica  
{  
    public ClsTecnico()  
    {  
    }  
    public void verificaAcesso() // exemplo de override (sobrescrita)  
    {  
        System.out.print("Acesso tipo 3");  
    }  
    public void verificaAcesso(int p1) // exemplo de overload (sobrecarga)  
    {  
        System.out.print("Acesso tipo 4");  
    }  
}
```

```
}  
}  
  
public class ClsPrincipal  
{  
    public ClsPrincipal()  
    {  
    }  
    public static void main(String Args[])  
    {  
        ClsFornecedor objFornecedor = new ClsFornecedor();  
        ClsCliente objCliente = new ClsCliente();  
        ClsTecnico objTecnico = new ClsTecnico();  
        objFornecedor.verificaAcesso(); // executa System.out.print("Acesso tipo  
1")  
        objCliente.verificaAcesso(); // executa System.out.print("Acesso tipo 2")  
        objTecnico.verificaAcesso(); // executa System.out.print("Acesso tipo 3")  
        objTecnico.verificaAcesso(1); // executa System.out.print("Acesso tipo  
4")  
    }  
}
```

3.2.7 Super

É um recurso para se invocar um construtor da classe pai. A palavra super disponibiliza o acesso a partes da classe pai a partir da classe filha, e é útil no polimorfismo da sobreposição de métodos. Veja o exemplo a seguir.

```
public class ClsPessoa  
{  
    public ClsPessoa() {} // construtor  
    public void verificaAcesso() {  
        System.out.print("Acesso tipo 1"); }  
    public void verificaAcesso(String pUsuario, String pSenha) {  
        System.out.print("Acesso tipo especial"); }  
}  
  
public class ClsPessoaJuridica extends ClsPessoa  
{  
    public ClsPessoaJuridica() {} // construtor  
    // herda verificaAcesso do pai sem sobrescrever executando Acesso tipo 1
```

```
}

public class ClsPessoaFisica extends ClsPessoa
{
    public ClsPessoaFisica() {} // construtor
    public void verificaAcesso() {
        System.out.print("Acesso tipo 2"); // sobreescreve o tipo de acesso do
        pai (override) }
    public void verificaAcesso(String p1) {
        super.verificaAcesso(); // executa System.out.print("Acesso tipo 1") da
        classe pai }
}

public class ClsTecnico extends ClsPessoaFisica
{
    public ClsTecnico() {} // construtor
    public void verificaAcesso(int p1) {
        System.out.print("Acesso tipo 3"); // exemplo de sobreescrita (override) }
    public void verificaAcesso(int p1, int p2) {
        System.out.print("Acesso tipo 4"); // exemplo de sobrecarga (overload) }
}

public class ClsPrincipal
{
    public ClsPrincipal() {} // construtor
    public static void main(String Args[]) {
        ClsTecnico objTecnico = new ClsTecnico();
        objTecnico.verificaAcesso("x"); // executa System.out.print("Acesso tipo
1")
        objTecnico.verificaAcesso(); // executa System.out.print("Acesso tipo 2")
        objTecnico.verificaAcesso(1); // executa System.out.print("Acesso tipo
3")
        objTecnico.verificaAcesso(1,2); // executa System.out.print("Acesso tipo
4")
        objTecnico.verificaAcesso("x","y"); // executa System.out.print("Acesso
tipo especial") }
}
```

3.2.8 Tratamento de Exceções

Em Java tanto a Máquina Virtual quanto o desenvolvedor podem "lançar" erros e exceções, através do comando *throw* e "apanhar" erros e exceções com o comando *try-catch*. As situações de erro e exceção ocorrem na forma de uma referência para uma instância de uma classe. Toda situação considerada de erro ou exceção corresponde ao arremesso ou lançamento (*throw*) de uma referência de uma instância de uma classe. E existem três classes, *String*, *Object* e *Throwable*. A classe *String* é a única classe que têm literais (representação de constantes). A classe *Object* é a única que não estende nenhuma classe em sua definição. A classe *Throwable* é a única classe cujas instâncias diretas ou indiretas podem ser usadas para serem lançadas. A classe *Throwable* é pré-definida na linguagem, juntamente com várias outras classes definindo um grande número de categorias de erro.

Quando se deseja criar novas exceções, essas devem estender a classe *Exception* que é uma subclasse de *Throwable*. Duas classes descendentes da classe *Throwable* podem ser consideradas especiais: a classe *Error* e a classe *RuntimeException*. A linguagem Java não obriga o programador a tratar os lançamentos de exceção envolvendo instâncias da classe *Error*, da classe *RuntimeException* e das classes delas derivadas. A Máquina Virtual Java lança referências para instâncias de maneira implícita utilizando as classes pré-definidas. O desenvolvedor pode usar o comando *throw* de forma explícita.

Quando o fluxo de controle atinge o comando *throw*, a expressão atribuída é avaliada e a partir desse ponto o fluxo de controle será desviado para uma cláusula *catch* correspondente do comando *try-catch*. A procura por um *try-catch* correspondente é propagada até o ponto em que exista um retorno para o método *main*, e nesse caso a execução do programa será finalizada, com mensagem de erro provida.

A linguagem Java classifica os erros e exceções em dois grupos. O que obriga a verificação (*checked*) e o que não obriga verificação (*nonchecked*). Para lançar uma exceção que seja instância das classes de verificação obrigatória à linguagem obriga o desenvolvedor a declarar no cabeçalho do método quais as classes de exceção podem ter instâncias lançadas. Portanto, o formato completo do cabeçalho de definição de um método é <modificadores> <tipo> <nome>(<parametros>) *throws* <classes> Veja o exemplo:

```
static void metodoX() throws ExcecaoTipo1, ExcecaoTipo2, ExcecaoTipo3 {  
    throw new ExcecaoTipo1();  
}
```

```
    throw new ExcecaoTipo2();  
    throw new ExcecaoTipo3();  
}
```

O desenvolvedor pode tratar as exceções de duas maneiras. A primeira é utilizar um comando que pode dar origem à exceção dentro do bloco *try* de um comando *try-catch* que contenha uma cláusula *catch* adequada. A segunda é utilizar a cláusula *throws* do cabeçalho de definição de um método. Podem até mesmo não tratar naquele método uma exceção que exige verificação, mas a linguagem obriga o programador a avisar isto através da cláusula *throws*. Segue abaixo a sintaxe básica do comando:

```
try  
{  
    <comandos que podem ou nao lançar exceções>  
}  
catch (ExcecaoTipo1 parâmetro)  
{  
    <tratamento da instância da classe ExcecaoTipo1>  
}  
catch (ExcecaoTipo2 parâmetro)  
{  
    <tratamento da instância da classe ExcecaoTipo2>  
}  
finally  
{  
    <estes comandos serão executados com ou sem lançamento de exceção>  
}
```

Para melhor entender o tratamento de exceção segue um exemplo:

```
public class Main {  
    public static void main(String[] args) {  
        int dividendo = 10;  
        int divisor = 0;  
        Calculadora calculadora = new Calculadora();  
        try  
        {  
            int quociente = calculadora.dividir(dividendo, divisor);  
            System.out.println(quociente);  
        }  
    }  
}
```

```
        catch (DivPorZero especifico)
        {
            System.out.println(especifico);
        }
        catch (Exception geral)
        {
            System.out.println(geral);
        }
    }
}

public class Calculadora {
    public int dividir(int dividendo, int divisor) throws DivPorZero {
        try
        {
            return (dividendo/divisor);
        }
        catch (Exception e)
        {
            throw new DivPorZero("Divisão por zero");
        }
    }
}

public class DivPorZero extends Exception {
    public DivPorZero(String mensagem) {
        super(mensagem);
    }
}
```

3.3 Tipos de Dados

A linguagem Java oferece diversos tipos de dados com os quais podemos trabalhar. Há basicamente duas categorias em que se encaixam os tipos de dados: tipos primitivos e tipos de referências. Os tipos primitivos correspondem a dados mais simples ou escalares, enquanto os tipos de referências consistem em *arrays*, classes e interfaces.

Seguem os Tipos:

- *boolean* - Pode assumir o valor *true* ou o valor *false*;

- *char* - Caractere em notação Unicode de 16 bits. Serve para a armazenagem de dados alfanuméricos. Também pode ser usado como um dado inteiro com valores na faixa entre 0 e 65535;
- *byte* - Inteiro de 8 bits em notação de complemento de dois. Pode assumir valores entre $-2^7=-128$ e $2^7-1=127$;
- *short* - Inteiro de 16 bits em notação de complemento de dois. Os valores possíveis cobrem a faixa de $-2^{15}=-32.768$ a $2^{15}-1=32.767$;
- *int* - Inteiro de 32 bits em notação de complemento de dois. Pode assumir valores entre $-2^{31}=-2.147.483.648$ e $2^{31}-1=2.147.483.647$;
- *long* - Inteiro de 64 bits em notação de complemento de dois. Pode assumir valores entre -2^{63} e $2^{63}-1$;
- *float* - Representa números em notação de ponto flutuante normalizada em precisão simples de 32 bits em conformidade com a norma IEEE 754-1985. O menor valor positivo representável por esse tipo é $1.40239846e-46$ e o maior é $3.40282347e+38$; e
- *double* - Representa números em notação de ponto flutuante normalizada em precisão dupla de 64 bits em conformidade com a norma IEEE 754-1985. O menor valor positivo representável é $4.94065645841246544e-324$ e o maior é $1.7976931348623157e+308$.

Ao contrário do que acontecem com outras linguagens de programação, as características dos tipos de dados listados acima independem da plataforma em que o programa deverá ser executado. Dessa forma, os tipos de dados primitivos são realmente únicos e garantem a capacidade de intercâmbio de informações entre diversos tipos de computadores, aliviando o programador da preocupação e da árdua tarefa de converter dados em formatos apropriados para a postagem.

3.3.1 Tipo Boolean

Este é o tipo de dado mais simples encontrado em Java. Uma variável booleana pode assumir apenas um entre dois valores: *true* ou *false*. Algumas operações possíveis em Java como $a \leq b$, $x > y$, etc. têm como resultado um valor booleano, que pode ser armazenado para uso futuro em variáveis

booleanas. Estas operações são chamadas operações lógicas. As variáveis booleanas são tipicamente empregadas para sinalizar alguma condição ou a ocorrência de algum evento em um programa Java. Por exemplo: `boolean fim_do_arquivo = false;` é a declaração de uma variável do tipo *boolean*, cujo nome é `fim_do_arquivo`. O valor *false* à direita do sinal "=" indica que a variável recebe esse valor como valor inicial. Sem essa especificação o valor de uma variável é impraticável, podendo ser qualquer um dos valores possíveis para seu tipo (neste caso *true* ou *false*).

Aproveitando o ensejo, há nessa linha a essência da declaração de qualquer variável em Java:

- Informar o tipo de dado que deseja declarar (*boolean*);
- Informar o nome que será usado para batizar a variável (`fim_do_arquivo`);
- Atribuir à variável um valor inicial (= *false*);
- Terminar a declaração com um ponto-e-vírgula; e
- Uma palavra sobre identificadores.

Na declaração acima usamos o nome `fim_do_arquivo` para designar a variável. Um nome de variável, assim como nome de um método, classe, rótulo e dezenas de outros itens lexicográficos, constitui o que é chamado um identificador. Uma vez criado, um identificador representa sempre o mesmo objeto a ele associado, em qualquer contexto em que seja empregado.

As seguintes regras regem a criação de identificadores:

- O primeiro caractere de um identificador deve ser uma letra. Os demais caracteres podem ser quaisquer sequências de numerais e letras;
- Não apenas os numerais e letras latinas podem ser empregadas, como também letras de quaisquer outros alfabetos;
- Devido a razões históricas, o underscore "_" e o sinal de dolar "\$" são considerados letras e podem ser usados nos identificadores;
- Assim como em outras linguagens, como C e C++, os identificadores distinguem o tipo de caixa das letras, isto é, as maiúsculas são consideradas distintas das minúsculas. Isso significa que `fim_de_arquivo` é um identificador diferente de `Fim_De_Arquivo`; e

- Os identificadores não podem ser palavras chave, como: *class*, *for*, *while*, *public*, etc.

3.3.2 Tipos de dados inteiros:

Os tipos de dados primitivos *byte*, *int*, *char*, *short* e *long* constituem tipos de dados inteiros. Isso porque variáveis desses tipos podem conter um valor numérico inteiro dentro da faixa estabelecida para cada tipo individual. Por exemplo, um *byte* pode conter um inteiro entre -128 e 127, enquanto um *short* pode conter um valor entre -32.768 e 32.767.

Há diversas razões para se utilizar um ou outro dos tipos inteiros em uma aplicação. Em geral, não é sensato declarar todas as variáveis inteiras do programa como *long*. Raramente os programas necessitam trabalhar com dados inteiros que permitam fazer uso da máxima capacidade de armazenagem de um *long*. Além disso, variáveis grandes consomem mais memória do que variáveis menores, como *short*.

Observação: Se alguma operação aritmética cria um resultado que excede um dos limites estabelecidos para o tipo inteiro empregado, não há qualquer indicação de erro para avisar sobre essa ocorrência. Ao invés disso, um complemento de dois do valor obtido será o resultado. Por exemplo, se a variável for do tipo *byte*, ocorrem os seguintes resultados: $127+1 = -128$, $127+9=-120$ e $127+127=-2$. Entretanto, uma exceção do tipo *ArithmeticException* é levantada caso ocorra uma divisão por zero.

Operações com inteiros

3.3.3 Tipo Caractere

Uma variável do tipo *char* armazena um caractere Unicode. Um caractere Unicode é um caractere de 16 bits, sendo que de 0 a 255 correspondem aos caracteres do código ASCII (a tabela ASCII é uma tabela padronizada internacionalmente de associações entre caractere e a sua representação numérica no computador). Uma constante do tipo caractere é representada colocando-se entre apóstrofes, ou pelo valor numérico correspondente na tabela Unicode (ou ASCII), ou ainda, pela sequência '\x' onde x especifica o caractere a ser referido. Esta especificação de sequência de escape obedece às mesmas convenções do C/C++. Por exemplo: 'a', 'f', '\n', etc., são constantes do tipo *char*.

3.3.4 Tipos de Ponto Flutuante

Em Java, existem duas categorias de variáveis de ponto flutuante: *float* armazena valores numéricos em ponto flutuante de precisão simples e *double* de precisão dupla. Ambas seguem norma: IEEE Standard for Binary Floating Point Arithmetic, ANSI/IEEE Std. 754-1985 (IEEE, New York). O fato de obedecer a essa norma é que torna os tipos de dados aceitos pela linguagem Java tão portátil. Esses dados serão aceitos por qualquer plataforma, independentemente do tipo de sistema operacional e do fabricante do computador. A representação dos valores em ponto flutuante pode ser feita usando a notação decimal (exemplo: -24.321) ou a notação científica (exemplo: 2.52E-31).

3.3.5 Conversão de Dados

O *cast* ou a conversão de objetos em Java é realizado de duas maneiras, de acordo com a conversão de tipos primitivos: implícito e explícito. Primeiramente, toda a classe filha pode ser associada a uma declaração de *class* ou *interface* pai. Esta é a forma implícita, pois não necessita de nenhum recurso para ser efetuado, como no exemplo abaixo:

```
...  
Pai objeto = new Filho();  
...
```

No exemplo, o objeto filho é declarado como uma instância de Pai. O inverso deste caso ocorre quando se tem um objeto pai e sabe-se que ele é uma instância de determinada classe filho. Esta forma é a explícita, pois é necessário informar ao compilador, qual a classe que se deseja fazer o *cast*:

```
...  
Pai objeto = new Filho();  
...  
Filho objFilho = (Filho)objeto;  
...
```

3.3.6 Operador instanceof

O operador *instanceof* é utilizado para *cast* explícitos que não se tem conhecimento da classe real do objeto. No exemplo, o objeto é declarado como sendo instância de Pai, mas é criada uma instância de Filho. Na linha seguinte, é associado a uma variável declarada como Filho o valor de objeto,

que foi declarado como *Pai*, mas é possível saber que ele é um *Filho*. Por exemplo:

```
...
public void write(Collection c) {
    if (c instanceof List) {
        write((List) c);
    } else {
        write((Set) c);
    }
}
...
```

No exemplo, a instância de *Collection* é testada para verificar se é um *List* ou um *Set*. De acordo com o valor dela, é chamada uma função *write* usando sobrecarga de métodos.

Ao trabalhar com expressões, salvo quando todos os operando são do mesmo tipo, é inevitável ter que considerar conversões entre um tipo de dado e outro. Há basicamente dois tipos de conversões de dados. O primeiro se refere à conversão implícita na qual, os dados são convertidos automaticamente, praticamente sem a preocupação do programador. Isto ocorre no caso de conversão de dados de tipo inteiro para real e de números para *Strings*. Por exemplo:

```
double x;
int i = 20;
x = i;
```

Neste caso o valor do inteiro *i* é convertido automaticamente para um *double* antes de ser armazenado na variável *x*. As regras de conversão implícita empregadas pela linguagem Java são as seguintes: os operadores unários ++ e -- convertem um tipo *byte* e *short* é convertido para um *int*, e os demais tipos não são afetadas para os operadores binários, as regras são um pouco mais complicadas. Para operações envolvendo apenas inteiros, se um dos operandos for *long*, o outro será convertido para um *long* antes de realizar a operação, a qual resultará num *long*. Caso contrário, ambos os operandos são convertidos para um *int* e o resultado será também um *int*, a menos que o resultado da operação seja grande demais para caber num *int*. Nesse caso, o resultado será convertido para um *long*. Para operações envolvendo números de ponto flutuante, se um dos operadores for *double*, o outro será convertido para *double* antes de realizar a operação e o resultado será um *double*. Do

contrário, ambos os operando são convertidos para *float*, e o resultado será um *float*.

Algumas vezes, porém, as conversões implícitas não são suficientes para garantir um resultado esperado em uma expressão. Nesses casos, é importante podermos controlar precisamente a ocorrência de uma conversão de tipo. Isto pode ser feito por meio de um operador unário de conversão. Por exemplo:

```
float eventos = 25.7;  
float dias = 7.2;  
x = (int)(eventos / dias);
```

O resultado da expressão acima será precisamente 3, isto é a parte inteira de 25.7 dividido por 7.2. A diferença é que o operador de conversão de tipo (int) transformou o valor do quociente 25.7/7.2 em um inteiro, truncando as casas decimais. Note que o operador de conversão é um operador unário de maior precedência, logo, tivemos de colocar a divisão entre parênteses para garantir que o resultado dessa divisão formasse um único operando. A conversão entre quaisquer inteiros e entre inteiros e números de ponto flutuante é permitida. Porém, não é permitido converter dados do tipo *boolean* para nenhum outro tipo, enquanto a conversão entre objetos somente é permitida para classes parentas.

Vejamos ainda outro exemplo de utilização do operador de conversão de tipo:

```
int a = 3;  
int b = 2;  
double x;  
x = a / b;
```

Neste exemplo, desejamos que a variável *x* tenha o valor do quociente entre *a* e *b*. Como *x* é um dado de ponto flutuante, presume-se que o resultado desejado neste exemplo seja, digamos, $x=3/2=1.5$. Como, porém os operandos da divisão são ambos inteiros, o resultado será também um inteiro, isto é, teremos $x = 1.0$. Para contornar essa situação, podemos converter explicitamente um dos operandos para um dado em ponto flutuante, fazendo:

```
x = (double)a / b;
```

Observamos que a conversão entre inteiros de maior comprimento para inteiros de menor comprimento pode causar perda de informação na representação dos valores. Caso o valor a ser convertido ocupe mais bits do que o tipo da variável recebendo esse valor, então o valor é truncado.

```
int l = 393;
byte b;
b = l;
System.out.println("O valor de b é " + b);
```

Neste exemplo teremos como resultado:

O Valor de b é -119

Por que isso aconteceu? Fica mais fácil de compreender isso se interpretarmos os números acima na notação binária. O número 393 é representado em base dois pelo número 110001001, que tem 9 bits de comprimento (omitimos os zeros não significativos). Como a variável *b* é do tipo *byte* e seu comprimento é de apenas 8 bits, teremos os 8 primeiros bits da direita para a esquerda atribuída a *b*, isto é, 10001001, que representa o número -119 em base dois, na notação de complemento de dois.

3.3.7 Operadores e Modificadores

Expressões são combinações ordenadas de valores, variáveis, operadores, parênteses e chamadas de métodos, permitindo realizar cálculos aritméticos, concatenarem *Strings*, comparar valores, realizar operações lógicas e manipular objetos. Sem expressões, uma linguagem de programação seria praticamente inútil. O resultado da avaliação de uma expressão é em geral um valor compatível com os tipos dos dados que foram operados.

A maneira como uma expressão é avaliada (ou calculada) obedece às mesmas regras familiares da Matemática. A regra mais simples é a da associatividade. Quando um mesmo operador aparece mais de uma vez em uma expressão, como em $a+b+c$, então o operador mais à esquerda é avaliado primeiro e em seguida o da direita, e assim por diante. Esta seria então equivalente a $((a+b)+c)$.

3.3.8 Precedência

Há situações em que é necessário juntar operadores diferentes numa mesma expressão. Nesse caso a associatividade não se aplica mais trivialmente. Nesse caso como descobrir quais são os operandos de um operador, e quais operadores são avaliados primeiro? A resposta está na precedência dos operadores. Por exemplo, escrevendo $a+b*c$, somos levados naturalmente a multiplicar primeiro b com c e em seguida o somar o produto com a . Isto porque a multiplicação tem precedência sobre a adição. A precedência é então um conjunto de regras que permitem determinar quais são os operandos de um dado operador.

Como a linguagem Java é rica em operadores, alguns pouco familiares, precisamos conhecer a precedência desses operadores.

Na tabela a seguir, está indicada a precedência dos operadores comumente usados em Java. A tabela apresenta os operadores ordenados da maior precedência para a menor. Observe que, existem alguns operadores que naturalmente não requerem preocupação quanto à sua precedência, devido à forma como são empregados.

Operador unário: positivo, negativo, negação (inversão bit a bit), não (lógico):
 $+ - \sim !$

Incremento, Decremento: $++ --$

Multiplicação, divisão e módulo (inteiros): $* / \%$

Adição, subtração: $+ -$

Translação (bit a bit) à esquerda, direita sinalizada, e direita não sinalizada (o bit de sinal será 0): $<<, >>, >>>$

Operador relacional: menor, menor ou igual, maior ou igual, maior: $<, <=, >=, >$

Igualdade: igual, diferente: $== !=$

Operador lógico e bit a bit: $\&$

Ou exclusivo (xor) bit a bit: $\wedge$

Operador lógico ou bit a bit: $|$

Operador lógico e condicional: $\&\&$

Operador lógico ou condicional: $||$

Condicional: if-then-else compacto: $?:$

Atribuição: $= op=$

3.3.9 Diferença entre o operador de igualdade $==$ e o método de comparação `equals()`

`equals()`

O método `public boolean equals(Object obj)` é um método proveniente da classe *Object*. Por default, sua comparação utiliza o operador `==` para comparar os dois objetos. Como todo objeto em JAVA possui a classe *Object* como sua *Superclass*, podemos sobrescrever seus métodos para comparar seus valores. Isso é o que acontece com a classe *String*. Ela sobrescreve seu método `.equals` para verificar se o valor de uma *String* é exatamente igual ao do objeto.

`==`

O operador `==` verifica se os dois objetos possuem a mesma referência, e ao contrario do que muitos pensam, ele não verifica se os dois objetos possuem o mesmo valor.

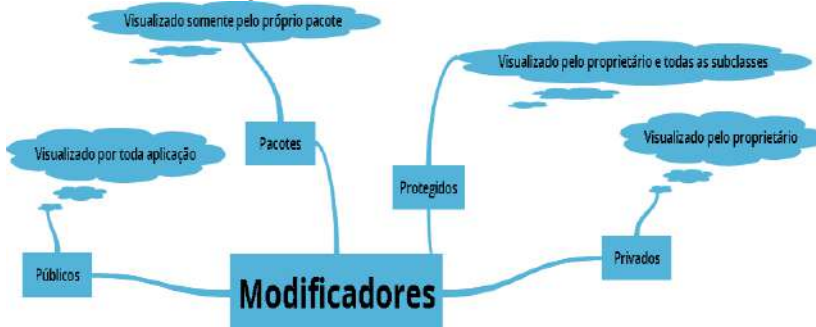
Verifique o seguinte exemplo: Repare que `texto1` refere-se ao objeto *"PogNotFound"*, e `texto2` refere-se a outro objeto que possui o mesmo texto. Ou seja, comparações de valores entre dois objetos do tipo *String* devem ser usados o método `.equals()` ou `.compareTo`

```
public class teste {
    public static void main(String[] args) {
        String texto1 = "teste";
        String texto2 = new String("teste");
        String texto3 = "teste";
        String texto4 = "testes";
        System.out.println("Valor de texto1: " + texto1);
        System.out.println("Valor de texto2: " + texto2 + "\n");
        System.out.println("Valor de texto3: " + texto3 + "\n");
        System.out.println("Valor de texto4: " + texto4 + "\n");
        if (texto1 == texto2)
            System.out.println("Valores Iguais!");
        else
            System.out.println("Valores Diferentes!");
        if (texto1 == texto3)
            System.out.println("Valores Iguais!");
        else
            System.out.println("Valores Diferentes!");
        if (texto1.equals(texto2))
            System.out.println("Valores Iguais!");
        else
            System.out.println("Valores Diferentes!");
    }
}
```

```
if (texto1.equals(texto4))    System.out.println("Valores Iguais!");  
else  
    System.out.println("Valores Diferentes!");  
}  
}
```

A figura 10.7 mostra como os modificadores são usados em Java.

Figura 3.7 Modificadores em Java



Fonte: Próprio Autor

3.4 Dicas

3.4.1 Usar a ferramenta Visual Class para desenhar formulários com Java

Como o desenvolvimento do *plugin* do Visual Class foi descontinuado para as novas versões do eclipse, indico usar o easyeclipse que já vem com esse recurso.

Acessar o *link* para baixar a versão desktop para windows 32bits: < <http://sourceforge.net/projects/easyeclipse/files/> >

Os *plugins* para o easyeclipse podem ser obtidos no *link*: < <http://www.easyeclipse.org/site/plugins/index.html> >

3.4.2 Usar a ferramenta JUDE Community para UML com Java

Com essa versão é possível se criar os diagramas UML e exportar para Java e vice-versa. Com as versões atuais do Astah (novo nome do Jude) na versão *community* não é possível fazer a exportação, apenas na versão *professional* é que está habilitada essa funcionalidade.

Acessar o *link* para baixar o Jude: < <http://goo.gl/AGruUa> >

3.4.3 Instalar a ferramenta visual Green UML

Green UML Diagramming Tool Green é um *plugin* simples e leve que permite a criação de diagramas UML dentro do ambiente do Eclipse, permitindo engenharia reversa. Acessar menu Help opção *Install New Software* no campo *Work with* digitar: < <http://www.cse.buffalo.edu/faculty/alphonce/green/> > e clicar em *Add*; Na caixa de diálogo *Add Site* digite Green UML e clique em *Ok*; Nas opções apresentadas com caixa de verificação marque a opção Green; Concorde com a licença finalize e reinicie o eclipse.

3.4.4 Instalar a IDE Eclipse com *plugin* de modelagem

Acessar o *link*: < <http://archive.eclipse.org/technology/epp/downloads/release/indigo/> > E por que essa versão? Já vem com os componentes visuais de modelagem integrados; Baixá-la e descompactá-la na pasta na qual deseja instalar o Eclipse.

3.4.5 Instalar *plugin* de modelagem em outras versões do eclipse

Com a versão desejada descompactada e acessada; Acessar o menu Help e a opção *Install New Software...*; No *combo Work with* selecionar o *link*: < <http://www.eclipse.org/downloads/> > Em *type filter text...* digitar UML e aguardar o filtro para os *plugins* disponíveis; Selecionar, finalizar e quando for pedido reiniciar o eclipse.

3.4.6 Instalar UML2 Tools

A UML2 Tools é uma ferramenta que permite edição dos diagramas UML versão 2.0 como classes, componentes, atividades, estados, casos de uso, sequências e de tempo. Esses diagramas estão disponíveis no menu *New / Other / UML 2.1 Diagrams* e são baseados no MDT UML2 metamodelo; Acessar o *link*: < <http://goo.gl/L0jv12> > e baixar o pacote Eclipse Galileo SR2 com Modeling Tools.

3.4.7 Instalar eUML2 plugin para eclipse de modelagem

A eUML2 Tools é uma ferramenta que permite edição dos diagramas UML versão 2.0 como classes, componentes, atividades, estados, casos de uso, sequências e de tempo;

Acessar o *link*: < <http://www.soyatec.com/euml2/installation/offline.php#offline> > baixar o pacote.

Para instalar a ferramenta acesse: < <http://www.soyatec.com/euml2/installation/galileo/index.php> >

Para trabalhar com a ferramenta acesse: < <http://www.soyatec.com/euml2/documentation/com.soyatec.euml2.doc/> >

3.4.8 Instalar o Java Server Faces (JSF)

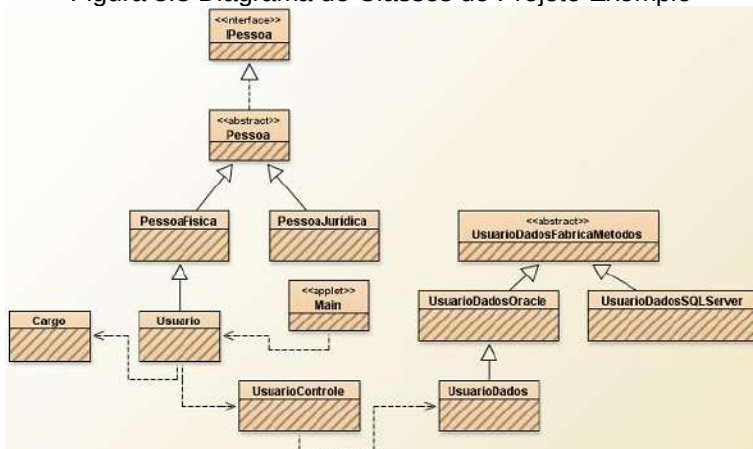
Instalar o eclipse o plugin JBoss pelo endereço: < <http://download.jboss.org/jbosstools/updates/nightly/trunk> > E baixar no endereço www.java2s.com as bibliotecas: slf4j-log4j2-1.5.3.jar; antlr-2.7.6.jar; dom4j-1.6.1.jar; ejb3-persistence.jar; hibernate3.jar; hibernate-annotations.jar; hibernate-commons-annotations.jar; hibernate-entitymanager.jar; javassist-3.4.GA.jar; jta-1.1.jar; log4j-1.2.15.jar; e slf4j-api-1.5.3.jar.

3.5 Exemplos

Para revisar os conceitos já estudados de Java segue abaixo codificação de um projeto.

Projeto Java SE Exemplo: Código-Fonte. Em < <http://goo.gl/RIMDX1> >
Projeto Java SE Agenda-Eletrônica com JDBC e MVC: Código-Fonte e Apresentação. Disp. < <http://goo.gl/i6lZXB> >
Projeto Java SE Acadêmicos com JDBC e MVC: Código-Fonte. Em < <http://goo.gl/ZfaAAu> >

Figura 3.8 Diagrama de Classes do Projeto Exemplo



Fonte: Próprio Autor

Classe Principal
// pacote
package exjava;

```
// importando todas as classes do pacote javax.swing
import javax.swing.*;

// classe principal
public class Main {

    // método principal
    public static void main(String[] args) {

        // tratamento de exceção
        try
        {
            // declarando variável alfanumérica
            String sRetorno;

            // método estático - pode ser invocado diretamente da classe
            sRetorno = PessoaFisica.dependentes();
            System.out.println(sRetorno);

            // entrando com dados
            String sCargoCodigo = JOptionPane.showInputDialog("Entre com o
código do cargo:");
            String sCargoNome = JOptionPane.showInputDialog("Entre com o
nome do cargo:");

            // convertendo alfanumérico para numérico
            int nCargoCodigo = Integer.parseInt(sCargoCodigo);

            // instanciando objeto cargo da classe Cargo
            Cargo cargo = new Cargo();

            // atribui valores (estados) aos campos do objeto
            cargo.setCargoCodigo(nCargoCodigo);
            cargo.setCargoDescricao(sCargoNome);

            // obtém os estados dos campos
            System.out.println("Cargo - código:" + cargo.getCargoCodigo() + "
descrição:" + cargo.getCargoDescricao());

            // entrando com dados
```

```
String sUsuarioCodigo = JOptionPane.showInputDialog("Entre com o  
código do usuário:");  
String sUsuarioNome = JOptionPane.showInputDialog("Entre com o  
nome do usuário:");  
String sUsuarioSexo = JOptionPane.showInputDialog("Entre com o  
sexo do usuário:");  
  
// convertendo alfanumérico para numérico  
int nUsuarioCodigo = Integer.parseInt(sUsuarioCodigo);  
  
// instanciando objeto usuario da classe Usuario  
Usuario usuario = new Usuario();  
  
// atribui valores (estados) aos campos do objeto  
usuario.setCodigo(nUsuarioCodigo);  
usuario.setNome(sUsuarioNome);  
usuario.setSexo(sUsuarioSexo);  
usuario.setCargo(cargo);  
  
// obtém os estados dos campos  
System.out.println("Usuário - código:" + usuario.getCodigo() + "  
nome:" + usuario.getNome() + " sexo:" + usuario.getSexo() + " cargo:" +  
usuario.getCargo().getCargoDescricao());  
  
// saindo os dados  
System.out.println(usuario.validaCPF());  
System.out.println(usuario.validaCNPJ());  
System.out.println(usuario.validaNome());  
  
// entrando com dados  
String sDividendo = JOptionPane.showInputDialog("Entre com o  
Dividendo:");  
String sDivisor = JOptionPane.showInputDialog("Entre com o  
divisor:");  
  
// convertendo alfanumérico para numérico  
int nDividendo = Integer.parseInt(sDividendo);  
int nDivisor = Integer.parseInt(sDivisor);  
  
// invoca método calcular  
int nQuociente = usuario.calcular(nDividendo, nDivisor);
```

```
}

// tratamento de exceção específico
catch (DivPorZero e)
{
    System.out.println(e.getMessage());
}

// tratamento de exceção genérico
catch (Exception e)
{
    System.out.println(e.getMessage());
}
}
```

Classe Exceção DivPorZero

// pacote

package exjava;

// subclasse exceção DivPorZero extendida da classe Exception

public class DivPorZero extends Exception {

// construtor da classe

public DivPorZero(String pMensagem) {

// invoca construtor da classe pai

super(pMensagem);

}

}

Classe Concreta Cargo

// pacote

package exjava;

// classe

public class Cargo {

// campos

private int cargoCodigo;

private String cargoDescricao;

```
// métodos get e set
public void setCargoCodigo(int pCargoCodigo) {
    this.cargoCodigo = pCargoCodigo;
}
public int getCargoCodigo() {
    return (this.cargoCodigo);
}
public void setCargoDescricao(String pCargoDescricao) {
    this.cargoDescricao = pCargoDescricao;
}
public String getCargoDescricao() {
    return (this.cargoDescricao);
}

// construtor padrão
public Cargo() {
}
}
```

Classe Concreta Usuário (subclasse de PessoaFisica e dependência de Cargo)

// pacote

package exjava;

// subclasse Usuario extendida da classe PessoaFisica

```
public class Usuario extends PessoaFisica {
```

```
    // campo da classe Usuário
    private Cargo cargo;
```

```
    // métodos get e set
    public void setCargo(Cargo pCargo) {
        this.cargo = pCargo;
    }
    public Cargo getCargo() {
        return (this.cargo);
    }
}
```

```
    // construtor padrão
    public Usuario() {
```

```
}

// construtor sobrecarregado
public Usuario(int pCodigo, String pNome, String pSexo, Cargo pCargo) {
    super(pCodigo, pNome, pSexo);
    this.cargo = pCargo;
}

// método de cálculo
public int calcular(int pDividendo, int pDivisor) throws DivPorZero {
    try
    {
        int quociente = pDividendo / pDivisor;
        return (quociente);
    }
    catch (Exception e)
    {
        throw new DivPorZero("exemplo de exceção específica");
    }
}
}
```

Classe Concreta PessoaFisica (subclasse de Pessoa)

// pacote

package exjava;

// subclasse PessoaFisica extendida da classe abstrata Pessoa

```
public class PessoaFisica extends Pessoa {
```

// campo da classe

```
    private String sexo;
```

// método get e set do campo sexo

```
    public void setSexo(String pSexo) {
```

```
        this.sexo = pSexo;
```

```
    }
```

```
    public String getSexo() {
```

```
        return (this.sexo);
```

```
    }
```

// construtor padrão

```
public PessoaFisica() {  
}  
  
// construtor sobrecarregado (overload)  
public PessoaFisica(int pCodigo, String pNome, String pSexo) {  
    super(pCodigo, pNome);  
    this.sexo = pSexo;  
}  
  
// sobreescrita (override) do método herdado abstrato  
public String validaCPF() {  
    return ("exemplo de método sobreescrito validaCPF");  
}  
  
// sobreescrita (override) do método herdado abstrato  
public String validaCNPJ() {  
    return ("exemplo de método sobreescrito validaCNPJ");  
}  
  
// método estático  
public static String dependentes() {  
    return ("exemplo de método estático");  
}  
}
```

Classe Concreta PessoaJuridica (subclasse de Pessoa)

// pacote

package exjava;

// subclasse PessoaJurídica estendida da classe abstrata Pessoa

```
public class PessoaJuridica extends Pessoa {
```

// campo da classe

```
private String inscricao;
```

// método get e set do campo inscricao

```
public void setInscricao(String pInscricao) {
```

```
    this.inscricao = pInscricao;
```

```
}
```

```
public String getInscricao() {
```

```
    return (this.inscricao);
```

```
}

// sobreescrita (override) do método herdado abstrato
public String validaCPF() {
    return ("exemplo de método sobreescrito validaCPF");
}

// sobreescrita (override) do método herdado abstrato
public String validaCNPJ() {
    return ("exemplo de método sobreescrito validaCNPJ");
}
}
```

Classe Abstrata Pessoa (realiza IPessoa)

// pacote

package exjava;

// classe abstrata Pessoa realizando a Interface IPessoa

public abstract class Pessoa implements IPessoa {

// campos da classe pessoa

private int codigo;

private *String* nome;

// métodos get e set

public void setCodigo(int pCodigo) {

this.codigo = pCodigo;

}

public int getCodigo() {

return (this.codigo);

}

public void setNome(*String* pNome) {

this.nome = pNome;

}

public *String* getNome() {

return (this.nome);

}

// construtor padrão

public Pessoa() {

}

```
// construtor sobrecarregado (overload)
public Pessoa(int pCodigo, String pNome) {
    this.codigo = pCodigo;
    this.nome = pNome;
}

// métodos abstratos
public abstract String validaCPF();
public abstract String validaCNPJ();

// método ativo da classe
public String validaNome() {
    return ("exemplo de método ativo");
}
}

Interface IPessoa
// pacote
package exjava;

// interface
public interface IPessoa {

    // métodos assinados
    public String validaCPF();
    public String validaCNPJ();
}

Classe abstrata UsuarioDadosFabricaMetodos
// pacote
package exjava;

// classe abstrata
public abstract class UsuarioDadosFabricaMetodos {

    // método ativo
    public static String StringConexao()
    {
        return ("String de conexão utilizado");
    }
}
```

```
// método abstrato
public abstract String Incluir(Usuario usuario) throws Exception;

// método abstrato
public abstract String Alterar(Usuario usuario) throws Exception;

// método abstrato
public abstract String Excluir(Usuario usuario) throws Exception;

// método abstrato
public abstract String Consultar(Usuario usuario) throws Exception;
}

Classe concreta UsuarioDadosOracle
// pacote
package exjava;

public class UsuarioDadosOracle extends UsuarioDadosFabricaMetodos {

    // método ativo
    public String Incluir(Usuario usuario) throws Exception
    {
        try
        {
            String objConexao = StringConexao();
            return ("Executa comando INSERT do ORACLE com a conexão ao
banco " + objConexao);
        }
        catch (Exception e)
        {
            throw new Exception("Exemplo de exceção");
        }
        finally
        {
            // Libera objetos de banco
        }
    }

    // método ativo
    public String Alterar(Usuario usuario) throws Exception
```

```
{
    try
    {
        String objConexao = StringConexao();
        return ("Executa comando UPDATE do ORACLE com a conexão ao
banco " + objConexao);
    }
    catch (Exception e)
    {
        throw new Exception("Exemplo de exceção");
    }
    finally
    {
        // Libera objetos de banco
    }
}
```

```
public String Excluir(Usuario usuario) throws Exception
{
    try
    {
        String objConexao = StringConexao();
        return ("Executa comando DELETE do ORACLE com a conexão ao
banco " + objConexao);
    }
    catch (Exception e)
    {
        throw new Exception("Exemplo de exceção");
    }
    finally
    {
        // Libera objetos de banco
    }
}
```

```
public String Consultar(Usuario usuario) throws Exception
{
    try
    {
        String objConexao = StringConexao();
```

```
        return ("Executa comando SELECT do ORACLE com a conexão ao
banco " + objConexao);
    }
    catch (Exception e)
    {
        throw new Exception("Exemplo de exceção");
    }
    finally
    {
        // Libera objetos de banco
    }
}
}
```

Classe concreta UsuarioDadosSQLServer

// pacote

package exjava;

```
public class UsuarioDadosSQLServer extends UsuarioDadosFabricaMetodos
{
```

// método ativo

```
    public String Incluir(Usuario usuario) throws Exception
```

```
    {
```

```
        try
```

```
        {
```

```
            String objConexao = StringConexao();
```

```
            return ("Executa comando INSERT do SQLServer com a conexão ao
banco " + objConexao);
```

```
        }
```

```
        catch (Exception e)
```

```
        {
```

```
            throw new Exception("Exemplo de exceção");
```

```
        }
```

```
        finally
```

```
        {
```

```
            // Libera objetos de banco
```

```
        }
```

```
    }
```

// método ativo

```
public String Alterar(Usuario usuario) throws Exception
{
    try
    {
        String objConexao = StringConexao();
        return ("Executa comando UPDATE do SQLServer com a conexão ao
banco " + objConexao);
    }
    catch (Exception e)
    {
        throw new Exception("Exemplo de exceção");
    }
    finally
    {
        // Libera objetos de banco
    }
}
```

```
public String Excluir(Usuario usuario) throws Exception
{
    try
    {
        String objConexao = StringConexao();
        return ("Executa comando DELETE do SQLServer com a conexão ao
banco " + objConexao);
    }
    catch (Exception e)
    {
        throw new Exception("Exemplo de exceção");
    }
    finally
    {
        // Libera objetos de banco
    }
}
```

```
public String Consultar(Usuario usuario) throws Exception
{
    try
    {
        String objConexao = StringConexao();
```

```
        return ("Executa comando SELECT do SQLServer com a conexão ao
banco " + objConexao);
    }
    catch (Exception e)
    {
        throw new Exception("Exemplo de exceção");
    }
    finally
    {
        // Libera objetos de banco
    }
}
}
```

Classe concreta UsuarioDados

// pacote

package exjava;

public class UsuarioDados extends UsuarioDadosOracle

```
{
    public UsuarioDados()
    {
        // herda as implementações de métodos correspondente ao banco
        utilizado
    }
}
```

Classe concreta UsuarioControle

// pacote

package exjava;

public class UsuarioControle {

// método estático para validação

public static *String* Validar(Usuario usuario) throws Exception

```
{
    try
    {
        if ((usuario.getCodigo() > 0) && (usuario.getCodigo() <= 100))
        {
            UsuarioDados dados = new UsuarioDados();
        }
    }
}
```

```
        return (dados.Incluir(usuario));
    }
    else
    {
        throw new Exception("Intervalo válido para código é de 1 a 100");
    }
}
catch (Exception e)
{
    throw new Exception("Exemplo de exceção");
}
}
```

Classe JFrame

package exjava;

public class ExemploJFrame extends javax.swing.JFrame {

```
    public ExemploJFrame() {
        initComponents();
    }
```

@SuppressWarnings("unchecked")

// <editor-fold defaultstate="collapsed" desc="Generated Code">

```
private void initComponents() {
```

```
    jLabel1 = new javax.swing.JLabel();
    jLabel2 = new javax.swing.JLabel();
    jLabel3 = new javax.swing.JLabel();
    jTextField1 = new javax.swing.JTextField();
    jTextField2 = new javax.swing.JTextField();
    jTextField3 = new javax.swing.JTextField();
    jLabel4 = new javax.swing.JLabel();
    jButton1 = new javax.swing.JButton();
```

```
setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
```

```
    jLabel1.setText("Código");
```

```
jLabel2.setText("Nome");

jLabel3.setText("Sexo");

jLabel4.setForeground(new Java.awt.Color(255, 0, 0));
jLabel4.setText("Mensagem");

jButton1.setText("Salvar");
jButton1.addActionListener(new Java.awt.event.ActionListener() {
    public void actionPerformed(Java.awt.event.ActionEvent evt) {
        jButton1ActionPerformed(evt);
    }
});

javax.swing.GroupLayout layout = new
javax.swing.GroupLayout(getContentPane());
getContentPane().setLayout(layout);
layout.setHorizontalGroup(

layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
    .addGroup(layout.createSequentialGroup()
        .addGap(41, 41, 41)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
    .addComponent(jButton1)
    .addComponent(jLabel4)
    .addGroup(layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
    .addComponent(jLabel1)
    .addComponent(jLabel2)
    .addComponent(jLabel3))
    .addGap(18, 18, 18)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
    .addComponent(jTextField2,
javax.swing.GroupLayout.PREFERRED_SIZE, 288,
javax.swing.GroupLayout.PREFERRED_SIZE)
```

```
        .addComponent(jTextField1,
javax.swing.GroupLayout.PREFERRED_SIZE, 53,
javax.swing.GroupLayout.PREFERRED_SIZE)
        .addComponent(jTextField3,
javax.swing.GroupLayout.PREFERRED_SIZE, 22,
javax.swing.GroupLayout.PREFERRED_SIZE))))
        .addContainerGap(20, Short.MAX_VALUE))
    );
    layout.setVerticalGroup(

layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
        .addGroup(layout.createSequentialGroup())
            .addGap(27, 27, 27)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.B
ASELINE)
        .addComponent(jLabel1)
        .addComponent(jTextField1,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
            .addGap(18, 18, 18)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.B
ASELINE)
        .addComponent(jLabel2)
        .addComponent(jTextField2,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
            .addGap(18, 18, 18)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.B
ASELINE)
        .addComponent(jLabel3)
        .addComponent(jTextField3,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
            .addGap(28, 28, 28)
            .addComponent(jLabel4)
```

```
.addGap(26, 26, 26)
.addComponent(jButton1)
.addContainerGap(86, Short.MAX_VALUE))
);

pack();
} // </editor-fold>

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    Usuario usuario = new Usuario();
    usuario.setCodigo(Integer.parseInt(jTextField1.getText()));
    usuario.setNome(jTextField2.getText());
    usuario.setSexo(jTextField3.getText());
    try
    {
        jLabel4.setText(UsuarioControle.Validar(usuario));
    }
    catch (Exception e)
    {
        jLabel4.setText(e.getMessage());
    }
}

public static void main(String args[]) {
    Java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new ExemploJFrame().setVisible(true);
        }
    });
}

// Variables declaration - do not modify
private javax.swing.JButton jButton1;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel2;
private javax.swing.JLabel jLabel3;
private javax.swing.JLabel jLabel4;
private javax.swing.JTextField jTextField1;
private javax.swing.JTextField jTextField2;
private javax.swing.JTextField jTextField3;
// End of variables declaration
```

```
}
```

3.6 Atividades

1) Considere uma classe `ContaBancária` e suas duas subclasses `ContaCorrente` e `ContaPoupança`, cada uma das três com o método `AplicarJuros`. A redefinição do método `AplicarJuros` nas duas subclasses, cada uma com detalhes de implementação diferentes com a mesma interface, caracteriza o princípio denominado na orientação a objetos como:

1. Abstração.
2. Encapsulamento.
3. Herança.
4. Coesão.
5. Polimorfismo.

2) Na orientação Objeto, quando uma rotina precisar ser modificada, basta o código do programa que define o objeto ser mudado que o comportamento da rotina é imediatamente atualizado. Esta característica é conhecida como:

1. Acoplamento.
2. Polimorfismo.
3. Herança.
4. Sobrecarga.
5. Encapsulamento.

3) No tocante à linguagem Java, a principal utilização da variável de classe *static* como uma única cópia de dados:

1. Compartilhamento de código.
2. Economia de memória.
3. Ocultamento de informação.
4. Minimização de manutenção.
5. Controle de acesso.

4) Analise as seguintes afirmações relacionadas à Programação Orientada a Objetos: I. Em um Programa Orientado a Objetos as instâncias de uma classe armazenam tipos diferentes de informações e apresentam comportamentos distintos; II. Em uma Aplicação Orientada a Objetos podem existir múltiplas instâncias de uma mesma classe; III. Em Programação Orientada a Objetos deve existir um e somente um objeto de uma mesma classe; e IV. Os serviços que podem ser solicitados a um objeto

são definidos pelos métodos. Indique a opção que contenha todas as afirmações verdadeiras:

1. II e IV
2. II e III
3. III e IV
4. I e III
5. I e II

5) Na Programação Orientada a Objetos podem-se definir as visibilidades dos métodos e atributos. Quanto a essa característica é correto afirmar que:

1. O acesso aos atributos e aos métodos privados só pode ser feito a partir dos métodos membros da classe.
2. Os métodos protegidos podem ser acessados a partir dos métodos de qualquer classe, desde que instanciada na mesma aplicação.
3. Os métodos públicos e os atributos protegidos só podem ser acessados a partir dos métodos da própria classe ou de classes derivadas.
4. Os métodos privados e os atributos protegidos podem ser acessados a partir dos métodos de qualquer classe.
5. O acesso aos atributos privados e aos métodos públicos só pode ser feito a partir dos métodos membros da classe.

6) Analise as seguintes afirmações relacionadas à Orientação a Objetos: I. Os atributos podem ser de instância ou de classe. Quando os atributos são de instância, cada objeto guarda a sua própria cópia de tais atributos; II. Os atributos compartilhados entre objetos de uma classe são chamados de atributos de objetos. Em Java, por exemplo, esses atributos são identificados com a palavra *static*; III. Existem métodos cujo código apenas acessa atributo de classe ou os parâmetros passados. Esses métodos são chamados de métodos de classe; e IV. Um mesmo nome de objeto pode ser usado para identificar diferentes objetos em uma mesma classe ou diferentes objetos em classes diferentes, evitando assim, que seja necessário usar nomes diferentes para objetos diferentes que realizam a mesma operação. A esta característica da Orientação a Objetos dá-se o nome de Polimorfismo. Indique a opção que contenha todas as afirmações verdadeiras:

1. I e II
2. II e III
3. III e IV
4. II e IV
5. I e III

7) Analise as seguintes afirmações relacionadas à Orientação a Objetos: I. As exceções que podem ser lançadas por um método são definidas na criação da classe abstrata que contém o método. As exceções normalmente provocam o encerramento indesejado do programa; II. Quando uma classe é definida, nem sempre é possível definir todos os seus métodos. Para resolver esta situação podem-se definir métodos abstratos, que podem ser instanciados a qualquer momento após a criação da classe; III. Da mesma forma que é necessária especificar o código a ser executado na criação de um objeto, pode ser necessário especificar um código para ser executado na destruição de um objeto. Isso é feito com a utilização de um método chamado destrutor ou finalizador. Nesses métodos é comum, por exemplo, inserir código para fechar arquivos ou encerrar conexões; e IV. Em uma classe podem existir métodos que não devem ser redefinidos em classes derivadas. Esses métodos são chamados métodos finais e uma tentativa de redefini-los resultam em um erro de compilação. Indique a opção que contenha todas as afirmações verdadeiras:

1. I e II
2. II e III
3. I e III
4. III e IV
5. II e IV

8) Na Programação Orientada a Objetos:

1. A definição dos objetos deve ser obrigatoriamente definida no corpo do construtor.
2. O código construtor é responsável por criar e inicializa os atributos dos objetos.
3. O construtor é uma função que tem um nome qualquer e, quando instanciada, faz referência aos atributos da classe a ser definida.
4. O código construtor é responsável por criar os métodos da classe.
5. Quando se instancia uma classe diversas vezes, o código construtor é responsável por armazenar e compartilhar os atributos das instâncias e os de classe utilizados ao longo de todo o processo.

9) Uma pizzaria fez uma ampliação de suas instalações e o gerente aproveitou para melhorar o sistema informatizado, que era limitado e não atendia a todas as funções necessárias. O gerente, então, contratou uma empresa para ampliar o *software*. No desenvolvimento do novo sistema, a empresa aproveitou partes do sistema antigo e estendeu os componentes de maneira a usar código validado, acrescentando as novas funções

solicitadas. Que conceito de orientação a objetos está descrito na situação hipotética acima?

1. Polimorfismo.
2. Herança.
3. Encapsulamento.
4. Abstração.
5. Mensagem.

10) Com relação a conceitos de orientação a objetos, julgue os seguintes itens. I. As variáveis ou métodos declarados com modificador de acesso *private* só são acessíveis a métodos da classe em que são declarados; II. Uma classe deve possuir uma única declaração de método construtor; III. Uma instância de uma classe abstrata herda atributos e métodos de sua superclasse direta; e IV. O polimorfismo permite substituir a lógica condicional múltipla. Estão certos apenas os itens:

1. I e II
2. I e III
3. I e IV
4. II e III
5. II e IV

Respostas: 1)5; 2)3; 3)2; 4)1; 5)1; 6)5; 7)4; 8)2; 9)2; 10)3.

3.7 Ferramentas

Bluej. Em < <http://www.bluej.org/> >

DrJava. Em < <http://drjava.org/> >

Easyecclipse. Em < <http://sourceforge.net/projects/easyecclipse/files/> >

Eclipse. Em < <http://www.eclipse.org/> >

Eclipse Java Galileo SR2 win32 com UML2.2 de may2010. Em < <http://goo.gl/xOKqSF> >

Green UML. Em < <http://green.sourceforge.net/index.html> >

Heidisql (front-end MySQL). Em < <http://goo.gl/u9zO29> >

Idea. Em < <http://www.jetbrains.com/idea/> >

JBuilder. Em < <https://downloads.embarcadero.com/free/jbuilder> >

JCreator. Em < <http://www.jcreator.com/> >

JEdit. Em < <http://www.jedit.org/> >

Jude Community. Em < <http://goo.gl/r2xvL2> >

Makeware. Em < <http://makeware.ueuo.com/inicio.php> >

MyEclipse. Em < <https://www.genuitec.com/products/myeclipse/download/> >

Pacote de componentes para Classes Visuais. Em < <http://goo.gl/zD9DuJ> >

MySQL. Em < <http://dev.mysql.com/downloads/mysql/> >
MySQL connector. Em < <http://dev.mysql.com/downloads/connector/j/> >
MySQL IDE. Em < <http://www.heidisql.com/download.php> >
Netbeans. Em < <https://netbeans.org/downloads/> >
StylusStudio. Em < http://www.stylusstudio.com/java_ide.html >
Visual Swing. Em < <http://code.google.com/p/visualswing4eclipse/> >
XAMPP (BD MySQL). Em < <http://sourceforge.net/projects/xampp/files/XAMPP%20Windows/> >

3.8 Apresentações

Java Completo - Programação Orientada a Objetos com Java. Em < <http://goo.gl/A3qO5w> >
Java Parte 01 - Apresentação do Java. Em < <http://goo.gl/e82UPz> >
Java Parte 02 - Introdução ao Java. Em < <http://goo.gl/vTJiUC> >
Java Parte 03 - Array e Comandos do Java. Em < <http://goo.gl/Yed6UY> >
Java Parte 04 - Orientação a Objetos com Java. Em < <http://goo.gl/t4tzb5> >
Java Parte 05 - Recursos do Java. Em < <http://goo.gl/CMG48U> >
Java Parte 06 - Tratamento de Exceções em Java. Em < <http://goo.gl/Yb8Q5t> >
Java Parte 07 - Exemplo de Codificação em Java. Em < <http://goo.gl/A9msHk> >
Java Parte 08 - Entrada e Saída de Dados com Java. Em < <http://goo.gl/i9Hc3q> >
Java Parte 09 - MVC e JDBC com Java. Em < <http://goo.gl/oP36SO> >
Java Parte 10 - Exemplo de Aplicação em Java com POO, MVC e JDBC. Em < <http://goo.gl/A5ryXo> >
Java Parte 11 - Conceitos Gerais de Java. Em < <http://goo.gl/BfstSF> >
Java Parte 12 - POO com Java. Em < <http://goo.gl/zd56Sq> >
Java Parte 13 - Exemplos. Em < <http://goo.gl/cTHpFB> >

3.9 Vídeos

Instalação do Eclipse Galileo (SR2). Em < <https://www.youtube.com/watch?v=95IkXJQXr0Y> >
Instalação do Eclipse Indigo Modeling. Em < <https://www.youtube.com/watch?v=95IkXJQXr0Y> >
Trabalhando com Java e Banco. Em < https://www.youtube.com/view_play_list?p=52761782814BF9F2 >

Usando Java com o MySQL. Em < <https://www.youtube.com/watch?v=U6uUUy4r7t0> >

3.10 Sugestões de Leitura

Aprenda em 21 dias Java 2. Em < <http://goo.gl/FcxgaF> >
Core Java 2: Fundamentals. Em < <http://goo.gl/sl2EOZ> >
Começando a Programar em Java para Leigos. Em < <http://goo.gl/YVOCJL> >
Programação de software em Java. Em < <http://goo.gl/7oiuPY> >
Conceitos de computação com Java. Em < <http://goo.gl/fCsK0V> >
Padrões e projetos orientados a objetos. Em < <http://goo.gl/oGf1Y2> >
Programação Orientada a Objetos com Java. Em < <http://goo.gl/w69GX6> >

3.11 Referências Web

BNF for Java. Em < <http://cui.unige.ch/isi/bnf/JAVA/BNFindex.html> >
Convenções de Java. Em < <http://www.oracle.com/technetwork/java/codeconventions-135099.html> >
Especificações API Java. Em < <http://download.oracle.com/javase/6/docs/api/> >
MySQL. Em < <http://dev.mysql.com/doc/refman/5.5/en/> >
Netbeans Aplicação. Em < <http://goo.gl/ND3QLK> >
Netbeans Quickstart. Em < http://netbeans.org/kb/docs/java/quickstart_pt_BR.html >
Referência de Convenções de Codificação em Java. Em < <http://goo.gl/PfvJcb> >
Tutorial Java. Em < <http://download.oracle.com/javase/tutorial/index.html> >

3.12 Referências Bibliográficas

DEITEL, Harvey; DEITEL, Paul. Java: Como Programar. 6ªed. US: Pearson, 2005.
DEITEL, Harvey; DEITEL, Paul. Ajax, Rich Internet Applications. 1ªed. US: Prentice Hall, 2008.
RUMBAUGH, James; *et al.* Modelagem e Projetos Baseados em Objetos com UML. 2ªed. RJ: Campus, 2006.
RUMBAUGH, James; JACOBSON, Ivar; BOOCH, Grady. UML: Guia do Usuário. 1ªed. RJ: Campus, 2006.

4 Padrões de Projetos

Um padrão é a solução para um determinado problema em um determinado contexto. Um padrão codifica conhecimento específico obtido em uma experiência em um determinado domínio. Um sistema bem estruturado estará cheio de padrões: de linguagem; de projeto; e de arquitetura. Segundo Fowler, podem ser utilizados para melhorar o entendimento ou comunicação dos problemas e decisões arquiteturais. Podem ser vistos como uma tentativa de criar um vocabulário comum para comunicação.

Um padrão que se deve ter conhecimento na orientação a objetos é o GRASP que significa *General Responsibility Assignment Software Patterns* e que descreve os princípios fundamentais do design orientado a objetos e a atribuição de responsabilidades. Outro padrão é o de Gamma, *et al* que descreve vinte e três padrões clássicos na orientação a objetos. O padrão de Gamma é mais conhecido como padrão da gangue dos quatro (*Gang of Four patterns*, ou apenas GoF).

Segue um exemplo de *template* dos Padrões de Projetos:

- Nome - o nome que serve como referencia para o padrão;
- Problema - explica o problema que ocorre em um contexto, com sintomas e em condições;
- Solução - elementos que constituem o design, seus relacionamentos, responsabilidades e colaborações;
- Consequências - resultados e compromissos decorrentes da aplicação do padrão. Impactos sobre a flexibilidade, extensibilidade, portabilidade ou desempenho do sistema. Fundamentam a escolha do padrão mais apropriado;
- Nome e Classificação - identificam unicamente o padrão e o classifica para catalogação;
- Intenção - uma breve descrição que deve responder o que o padrão faz qual sua intenção e que problema ele trata;
- Também Conhecido Como - outros nomes pelo qual o padrão é conhecido;
- Motivação - um cenário que ilustra o problema e como as classes deste padrão o solucionam;
- Aplicabilidade - em que situações o padrão pode ser aplicado;

- Estrutura - representação gráfica do padrão com suas classes e colaborações;
- Participantes - classes e objetos que participam no padrão, incluindo suas responsabilidades;
- Colaborações - como os participantes colaboram entre si;
- Consequências - como o padrão atende a seus objetivos e que “efeitos colaterais” seu uso pode provocar;
- Implementação - dicas, técnicas e erros comuns ao implementar este padrão;
- Exemplo de Código - fragmentos de código ilustrando o padrão;
- Usos Conhecidos - exemplos de uso do padrão em sistemas reais; e
- Padrões Relacionados - padrões que estão fortemente relacionados a este, incluindo suas diferenças, ou utilizados por este.

4.1 Categorias dos Padrões de Projetos

Existem basicamente três categorias, as criacionais, estruturais e comportamentais. Padrões Criacionais: auxiliam na criação de objetos, tornando o programa menos dependente do modo como os objetos são criados e compostos. Assim, estes padrões permitem que se mudem as classes dos objetos criados em tempo de execução com pouco esforço de programação; Padrões Estruturais: Descrevem formas flexíveis para compor classes e objetos; e Padrões Comportamentais: Estes padrões são relacionados a algoritmos e responsabilidades associados a cada objeto. Mudando-se os objetos e classes utilizados, pode-se mudar o comportamento do programa. Acoplando-se um objeto a outro, pode-se adicionar comportamento ao segundo objeto.

4.2 Critérios dos Padrões de Projetos

Os padrões de projeto são classificados por dois critérios, o de objetivo e o de escopo. Objetivo reflete as categorias (criação, de estrutura ou de comportamento). Escopo especifica quando o padrão é aplicado (classes ou a objetos).

Padrões com escopo de classe lidam com relacionamentos estabelecidos através de herança, ou seja, estáticos e definidos em tempo de compilação. Padrões com escopo de objeto lidam com relacionamentos de objeto, que podem ser modificados em tempo de execução e são mais dinâmicos. Praticamente todo padrão utiliza herança de alguma forma, por isto apenas os

padrões que focam em relacionamentos através de herança devem ser classificados com escopo de classe. Os padrões mais importantes estão em escopo de objeto.

4.3 *General Responsibility Assignment Software Patterns*

Os padrões GRASP descrevem os princípios fundamentais do design orientado a objetos e a atribuição de responsabilidades, expressos como padrões. Estes padrões servem como guia para a realização de um *design* baseado em atribuição de Responsabilidades (RDD).

Trata-se de uma solução geral e reutilizável para algum determinado problema que ocorre com frequência no desenvolvimento de um projeto, isto é, são as melhores práticas formalizadas para resolver problemas comuns no desenvolvimento. É designado para o paradigma orientado a objeto e mostram relacionamentos e interações entre classes e/ou objetos. Resumindo, consiste de um conjunto de práticas para atribuição de responsabilidades a classes e objetos em projetos orientados a objeto.

Existem nove padrões GRASP:

- *Creator Designer*
- *Low Coupling*
- *High Cohesion*
- *Polymorphism*
- *Controller*
- *Indirection*
- *Protected Variations*
- *Information Expert*
- *Pure Fabrication*

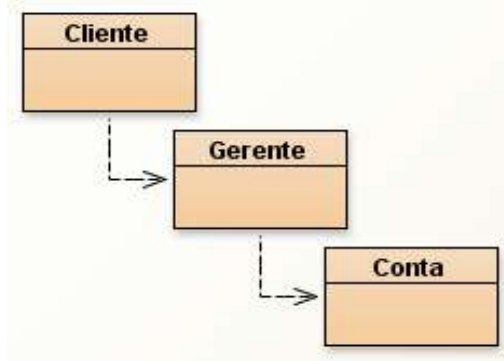
4.3.1 **Creator Designer**

Intenção: Designar responsável por criar uma nova instância de uma determinada classe. Problema: Quem deve ser o responsável por criar uma nova instância de uma determinada classe?

Aplicabilidade: Atribui-se a classe B a responsabilidade para criar uma instância da classe A se alguma das situações acontecer: B contém ou agrega objetos do tipo A; B armazena instâncias do tipo A; B, de forma

privada, usa objetos do tipo A; B tem os dados de inicialização que irão ser passados para A quando ele é criado; e Dessa forma, B é um *Expert* em relação à criação de A. Se mais de uma opção se aplica, dê preferência à classe que agrega ou contém instâncias de A.

Figura 4.1 GRASP Creator



Fonte: Próprio Autor

```
public class Cliente {
    public static String solicitarConta() {
        return (Gerente.abrirConta());
    }
}
public class Gerente {
    public static String abrirConta() {
        return (Conta.criarConta());
    }
}
public class Conta {
    public static String criarConta() {
        return ("Conta Criada");
    }
}
```

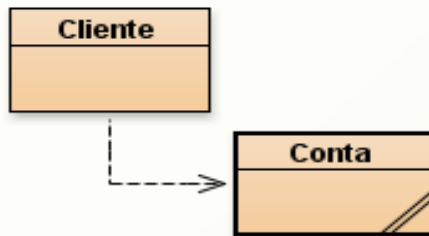
4.3.2 Low Coupling

Intenção: Atribuir à responsabilidade de forma que o acoplamento permaneça baixo. Problema: Como dar suporte a baixa dependência, redução de impacto de mudanças e aumento do reuso?

Aplicabilidade: Atribuir à responsabilidade de forma que o acoplamento desnecessário permaneça baixo. Mas o que é acoplamento? É a medida de o quão forte um elemento está conectado, tem conhecimento ou depende de outros elementos.

Um elemento (classe, subsistema, etc.) com baixo acoplamento não é dependente de muitos outros elementos, provendo aumento nas possibilidades de reuso, baixa manutenção, etc.

Figura 4.2 GRASP *Low Coupling*



Fonte: Próprio Autor

```
public class Cliente {
    public static String consultarConta() {
        return (Conta.consultarSaldo());
    }
}
public class Conta {
    public static String consultarSaldo() {
        return ("Saldo da Conta");
    }
}
```

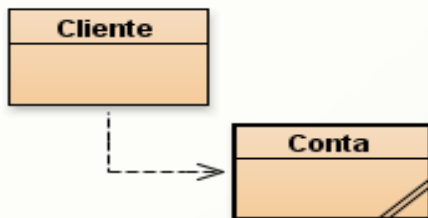
4.3.3 High Coesion

Intenção: Atribuir responsabilidade de forma que a coesão permaneça alta. Problema: Como manter os objetos focados, fáceis de entender, gerenciáveis e ainda pouco acoplados?

Aplicabilidade: Atribuir responsabilidade de forma que a coesão, ou, mais especificamente, coesão funcional, permaneça alta. E o que é coesão? É a

medida de quão forte estão relacionadas e focadas as responsabilidades de um elemento. Um elemento com alta coesão indica que suas responsabilidades são realizadas com pouco esforço. Por outro lado, um elemento com baixa coesão está 'muito distante', desassociado, dos outros elementos, gerando maior esforço para que ele cumpra suas responsabilidades.

Figura 4.3 GRASP *High Coesion*



Fonte: Próprio Autor

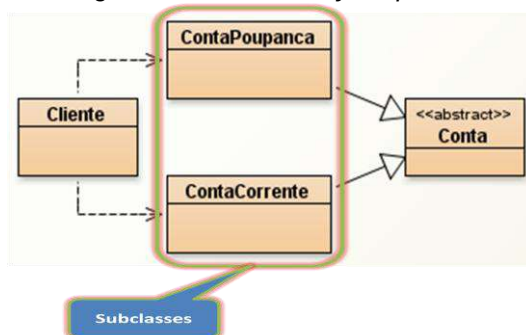
```
public class Cliente {  
    public static String consultarConta() {  
        return (Conta.consultarSaldo());  
    }  
}  
  
public class Conta {  
    public static String consultarSaldo() {  
        return ("Saldo da Conta");  
    }  
}
```

4.3.4 Polymorphism

Intenção: Quando comportamentos relacionados variam entre classes.
Problema: Como tratar alternativas baseadas em tipo? Como criar componentes de *software* plugáveis?

Aplicabilidade: Quando alternativas ou comportamentos relacionados variam por tipo (classe), atribuir a responsabilidade ao comportamento – usando operações polimórficas – para os tipos nos quais o comportamento varia. Nunca execute teste pelo tipo de um objeto e utilize lógica condicional para variar o comportamento baseado neste tipo.

Figura 4.4 GRASP Polymorphism



Fonte: Próprio Autor

```
public class Cliente {
    public static void solicitarSaldo() {
        new ContaCorrente().consultarSaldo();
        new ContaPoupanca().consultarSaldo();
    }
}

public abstract class Conta {
    public abstract String consultarSaldo();
}

public class ContaCorrente extends Conta {
    public String consultarSaldo() {
        return ("Saldo da Conta");
    }
}

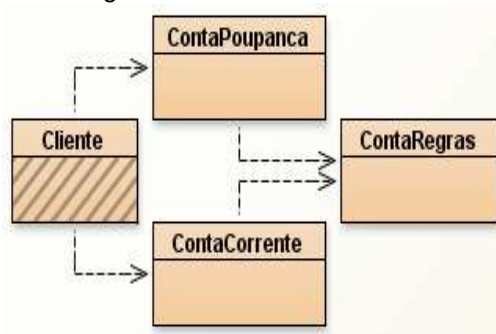
public class ContaPoupanca extends Conta {
    public String consultarSaldo() {
        return ("Saldo da Poupança");
    }
}
```

4.3.5 Controller

Intenção: Delegar trabalho a outros objetos, executando pouca lógica em si.
Problema: Qual o primeiro objeto após a Interface de Usuário que receberá e coordenará a operação do sistema?

Aplicabilidade: Atribuir essa responsabilidade a uma classe representando uma das opções: Classe que representa todo o sistema (aconselhável somente em situações em que existem poucas operações de sistema), serviços de um módulo ou subsistema ou qualquer outra variação de *controller*; e Classe que representa um cenário de caso de uso no qual o evento do sistema acontece. Normalmente o objeto definido pelo GRASP *controller* delega o seu trabalho a outros objetos, executando pouca ou nenhuma lógica em si. Adicionalmente este objeto não deve ser uma classe persistente do sistema.

Figura 4.5 GRASP Controller



Fonte: Próprio Autor

```
public class ContaRegras {
    public static String validarCliente() {
        return ("Cliente Validado");
    }
    public static String validarConta() {
        return ("Conta Validado");
    }
}

public class ContaCorrente {
    public static String consultarSaldo() {
        ContaRegras.validarCliente();
        ContaRegras.validarConta();
        return ("Saldo da Conta");
    }
}

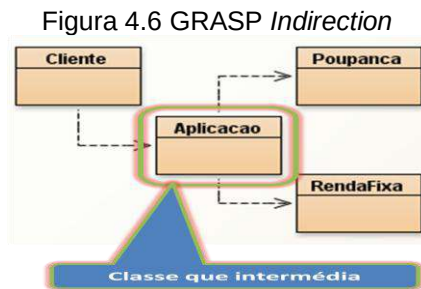
public class ContaPoupanca {
    public static String consultarSaldo() {
```

```
ContaRegras.validarCliente();  
ContaRegras.validarConta();  
return ("Saldo da Poupança");  
}  
}
```

4.3.6 Indirection

Intenção: Atribuir a responsabilidade a um intermediário que irá mediar entre os demais componentes de forma que eles não sejam diretamente acoplados. Problema: Onde atribuir uma responsabilidade para evitar acoplamento direto entre duas ou mais entidades? Como “desacoplar” objetos de forma que baixo acoplamento seja suportado e que o potencial de reuso permaneça alto?

Aplicabilidade: Atribuir a responsabilidade a um objeto intermediário que irá mediar entre os demais componentes ou serviços de forma que eles não sejam diretamente acoplados.



Fonte: Próprio Autor

```
public class Cliente {  
    public static void verificaAplicacao(double valor) {  
        Aplicacao.melhorAplicacao(valor);  
    }  
}  
  
public class Aplicacao {  
    public static String melhorAplicacao(double valor) {  
        if (valor > 1000) {  
            return (RendaFixa.aplicarMontante(valor));  
        } else {  
            return (Poupanca.aplicarMontante(valor));  
        }  
    }  
}
```

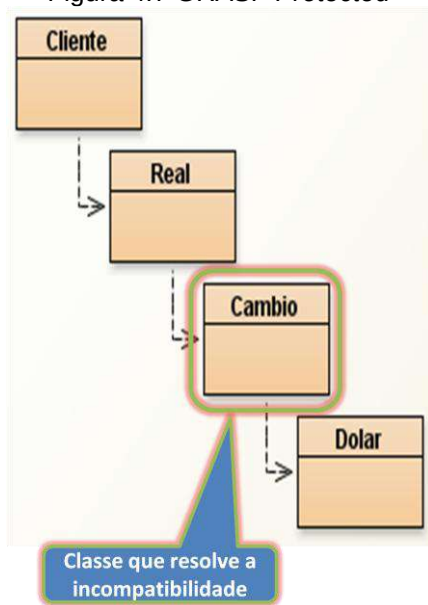
```
}  
}  
}
```

4.3.7 Protected Variations

Intenção: Identificar pontos de variações ou instabilidade previsíveis e atribuir responsabilidades para criação de uma interface estável em torno desses pontos. **Problema:** Como projetar objetos, subsistemas e sistemas de forma que as variações ou instabilidades nestes elementos não impliquem em impacto indesejado sobre outros elementos?

Aplicabilidade: Identificar pontos de variações ou instabilidade previsíveis. Atribuir responsabilidades para criação de uma interface estável em torno desses pontos.

Figura 4.7 GRASP *Protected*



Fonte: Próprio Autor

```
public class Cliente {  
    public static void aplicarDolar(String moedaReal) {
```

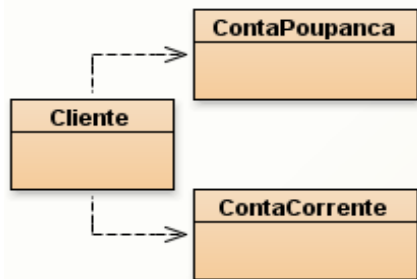
```
        Real.aplicarMontante(moedaReal);
    }
}
public class Real {
    public static String aplicarMontante(String moedaReal) {
        return (Cambio.converterMoeda(moedaReal));
    }
}
public class Cambio {
    public static String converterMoeda(String moedaReal) {
        String moedaDolar = realizarConversao(moedaReal);
        return (Dolar.aplicarMontante(moedaDolar));
    }
}
public class Dolar {
    public static String aplicarMontante(String moedaDolar) {
        return ("Valor Convertido para Dolar");
    }
}
}
```

4.3.8 Information Expert

Intenção: Atribuir responsabilidades para classes que possuem a informação necessária. Problema: Qual é o princípio básico para se atribuir responsabilidades a objetos?

Aplicabilidade: Atribuir responsabilidades para classes que possuem a informação necessária.

Figura 4.8 GRASP Information



Fonte: Próprio Autor

```
public class Cliente {
    public static String solicitarSaldo() {
        String corrente = (ContaCorrente.consultarSaldo());
        String poupanca = (ContaPoupanca.consultarSaldo());
        return ("Corrente:" + corrente + "\n Poupança:" + poupanca);
    }
}
public class ContaCorrente {
    public static String consultarSaldo() {
        return ("Saldo da Conta");
    }
}
public class ContaPoupanca {
    public static String consultarSaldo() {
        return ("Saldo da Poupança");
    }
}
```

4.3.9 Pure Fabrication

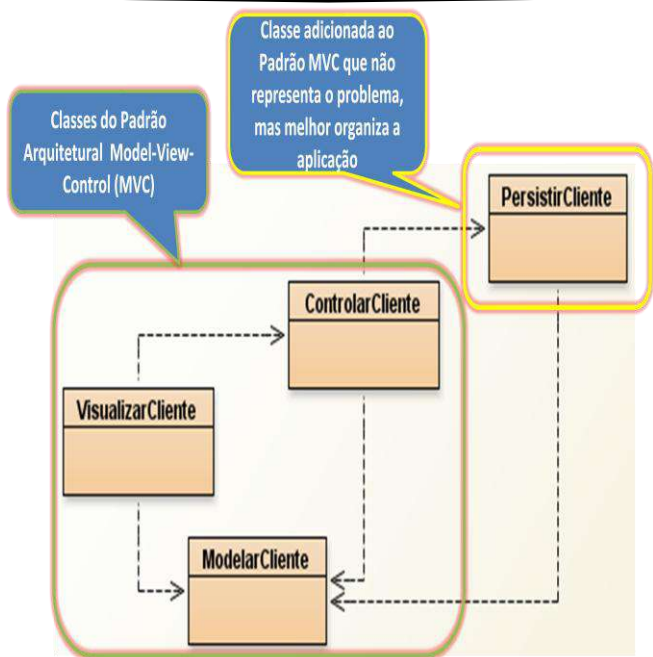
Intenção: Atribuir responsabilidades altamente coesas a uma classe que não representa um conceito no domínio do problema. **Problema:** Que objetos devem possuir responsabilidades quando não se quer violar os padrões *High Coesion* e *Low Coupling*, uma vez que a solução oferecida pelo *Information Expert* não é adequada?

Aplicabilidade: Atribuir um conjunto de responsabilidades altamente coesas a uma classe artificial ou conveniente que não representa um conceito no domínio do problema. Tal classe é uma criação da imaginação para suportar alta coesão, baixa acoplamento e reuso.

```
public class ModelarCliente {
    //Clase de Domínio da Aplicação
    private int codigo;
    private String nome;
    //gets...
    //sets...
}
public class VisualizarCliente {
    public static void cadastrarCliente() {
```

```
//Classe Visual de Entrada/Saída
ModelarCliente c = new ModelarCliente();
new ControlarCliente().validar(c);
}
}
public class ControlarCliente {
    //Classe de Regras de Negócio
    public static String validar(ModelarCliente c) {
        new PersistirCliente().incluir(c);
        return ("Realizar Regras de Negócio");
    }
}
public class PersistirCliente {
    //Classe Comandos SQL
    public static String incluir(ModelarCliente c) {
        return ("Realizar gravação no Banco de Dados");
    }
}
```

Figura 4.9 GRASP *Fabrication*



Fonte: Próprio Autor

4.4 Gang of Four (GoF)

Gamma *et al* descrevem vinte e três padrões que podem ser utilizados em praticamente qualquer área de programação. Estes padrões se tornaram clássicos da orientação a objetos e são chamados de padrões da gangue dos quatro (*Gang of Four patterns*, ou apenas GoF). Esse padrão se divide em dois critérios: Critério Objetivo que refletem as categorias de criação, estrutura e comportamento, por sua vez esse critério se divide em três categorias. Categoria dos Criacionais usados para determinar como os objetos são criados; Categoria dos Estruturais que descrevem formas flexíveis para compor classes e objetos; e a Categoria dos Comportamentais relacionados aos algoritmos e responsabilidades associados a cada objeto. O Critério Escopo que especifica quando o padrão é aplicado à classe e/ou objeto. Como pode ser visto logo abaixo.

Figura 4.10 Padrões GoF

Criacional	Estrutural	Comportamental
------------	------------	----------------

1. <i>Factory</i> (Classe)	<i>Method</i>	1. <i>Adapter</i> (Classe/Objeto)	1. <i>Interpreter</i> (Classe)
2. <i>Abstract</i> (Objeto)	<i>Factory</i>	2. <i>Bridge</i> (Objeto)	2. <i>Template Method</i> (Classe)
3. <i>Builder</i> (Objeto)		3. <i>Composite</i> (Objeto)	3. <i>Chain Responsibility</i> (Objeto)
4. <i>Prototype</i> (Objeto)		4. <i>Decorator</i> (Objeto)	4. <i>Command</i> (Objeto)
5. <i>Singleton</i> (Objeto)		5. <i>Facade</i> (Objeto)	5. <i>Strategy</i> (Objeto)
		6. <i>Flyweight</i> (Objeto)	6. <i>State</i> (Objeto)
		7. <i>Proxy</i> (Objeto)	7. <i>Observer</i> (Objeto)
			8. <i>Memento</i> (Objeto)
			9. <i>Mediator</i> (Objeto)
			10. <i>Visitor</i> (Objeto)
			11. <i>Iterator</i> (Objeto)

Fonte: Próprio Autor

Descrição Resumida dos Padrões GoF:

- *Abstract Factory*-Permitir que um cliente crie famílias de objetos sem especificar suas classes concretas;
- *Adapter*-Classe adaptadora entre a classe interna e um componente;
- *Bridge*-Desassociar a evolução de duas classes;
- *Builder*-Encapsular a construção de um comportamento e permitir que ele seja construído em etapas;
- *Chain Responsibility*-Define uma forma de passar uma solicitação entre objetos;
- *Command*-Encapsular um pedido de comando em um objeto;
- *Composite*-Criar uma hierarquia parte todo;
- *Decorator*-Estender a funcionalidade dinamicamente;
- *Facade*-Definir uma interface de alto nível;
- *Factory Method*-Subclasses decidem quais classes concretas serão criadas;
- *Flyweight*-Definir ajustes finos em subclasses;
- *Iterator*-Fornecer forma de acessar seqüencialmente uma coleção de objetos sem expor a sua implementação;
- *Interpreter*-Permitir a inclusão de elementos de linguagem em um aplicativo;
- *Mediator*-Definir comunicação simplificada entre classes;

- *Memento*-Salvar e restaurar o estado interno de um objeto;
- *Observer*-Definir um regime de notificação de objetos de mudanças para um outro objeto;
- *Prototype*-Permitir criar novas instancias simplesmente copiando instancias existentes;
- *Proxy*-Um classe que controla o acesso para outra classe;
- *Singleton*-Assegurar que somente um objeto de uma determinada classe seja criado em todo o projeto;
- *State*-Alterar o comportamento de um objeto quando seu estado muda;
- *Strategy*-Encapsular um algoritmo dentro de uma classe;
- *Template Method*-Permitir que subclasses redefinam os passos de um algoritmo; e
- *Visitor*-Define uma nova operação em uma classe sem trocá-la.

4.5 Padrão de Projeto GoF Criacional

Padrões criacionais são usados para determinar como os objetos serão criados ou instanciados. Padrões de projeto abstraem o processo de instanciação. Eles ajudam a tornar o sistema independente de como os objetos são criados, compostos e representados. Um padrão criacional de classe utiliza herança para variar a classe que será instanciada. Um padrão criacional de objeto irá delegar a instanciação de um objeto para outro objeto. Padrões criacionais tornam-se importantes à medida que os sistemas evoluem e passam a depender mais de composição de objetos do que de herança de classes. À medida que isto acontece, a ênfase migra da codificação de um conjunto de comportamentos para a codificação de conjuntos menores de comportamento, que podem ser combinados em vários conjuntos mais complexos.

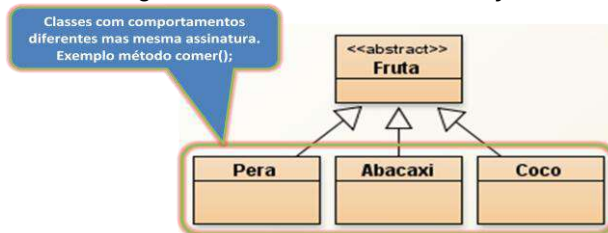
4.5.1 Abstract Factory

Intenção: Provê uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

Aplicabilidade: Este padrão deve ser utilizado quando o programa precisa de independência de como seus objetos são criados, compostos e representados. Ou quando o sistema precise ser configurado para uma ou muitas famílias de classes ou objetos. Ou quando uma família de objetos relacionados é projetada para ser utilizada em conjunto e esta premissa deve

ser garantida. Ou quando se deseja prover uma biblioteca de classes, e deseja-se disponibilizar apenas as interfaces, e não as implementações.

Figura 4.11 GoF *Abstract Factory*



Fonte: Próprio Autor

```
public abstract class Fruta {  
    public abstract String comer();  
}  
public class Pera extends Fruta {  
    public String comer() {  
        return ("pegar;mastigar;");  
    }  
}  
public class Abacaxi extends Fruta {  
    public String comer() {  
        return ("pegar;descascar;mastigar;");  
    }  
}  
public class Coco extends Fruta {  
    public String comer() {  
        return ("pegar;descascar;abrir;mastigar;");  
    }  
}
```

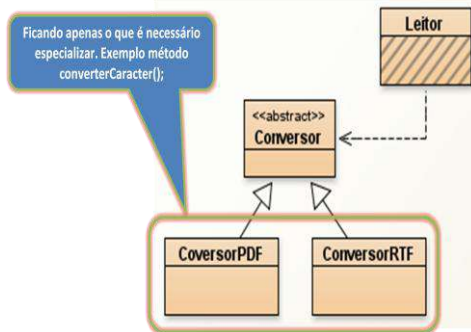
4.5.2 Builder

Intenção: Separa a construção de um objeto complexo de sua representação, de modo que o mesmo processo de construção possa criar diferentes representações.

Aplicabilidade: O padrão *builder* deve ser utilizado quando o algoritmo para criação de objetos complexos deve ser independente das partes que

compõem o objeto e da forma como este objeto é “montado”. Ou quando o processo de construção deve permitir diferentes representações para o objeto que está sendo construído.

Figura 4.12 GoF *Builder*



Fonte: Próprio Autor

```
public abstract class Conversor {
    public abstract String converterCaractere();
    public abstract String converterParagrafo();
    public abstract String converterTexto();
}

public class ConversorRTF extends Conversor {
    public String converterCaractere() {
        return ("Converte um caractere em PDF");
    }
    public String converterParagrafo() {
        return ("Converte um parágrafo em PDF");
    }
    public String converterTexto() {
        return ("Converte um texto em PDF");
    }
}

public class ConversorPDF extends Conversor {
    public String converterCaractere() {
        return ("Converte um caractere em PDF");
    }
    public String converterParagrafo() {
        return ("Converte um parágrafo em PDF");
    }
}
```

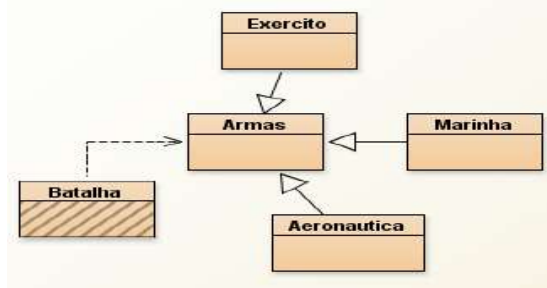
```
    public String converterTexto() {  
        return ("Converte um texto em PDF");  
    }  
}  
public class Leitor {  
    public static void main(String[] args) {  
        lerRTF("texto");  
    }  
    protected static void lerRTF(String txt) {  
        Conversor c = new ConversorRTF();  
        c.converterCaractere();  
    }  
    protected static void lerPDF(String txt) {  
        Conversor c = new CoversorPDF();  
        c.converterCaractere();  
    }  
}
```

4.5.3 Factory Method

Intenção: Define uma interface para criação um objeto, mas deixa as subclasses decidirem que classe instanciar. Permite que uma classe delegue a instanciação a suas subclasses.

Aplicabilidade: Este padrão deve ser utilizado quando uma classe não pode antecipar a classe dos objetos que deve criar. Ou quando a classe deseja que suas subclasses especifiquem o objeto que será criado. O quando a classe delega a responsabilidade de criação para um de muitas classes auxiliares, e deseja-se localizar o “conhecimento” de que classe auxiliar deve ser delegada.

Figura 4.13 GoF *Factory Method*



Fonte: Próprio Autor

```
public class Armas {
    public Armas() {
    }
}

public class Exercito extends Armas {
    public static void usarInfantaria() {
    }
}

public class Marinha extends Armas {
    public static void usarNavios() {
    }
}

public class Aeronautica extends Armas {
    public static void usarAvioes() {
    }
}

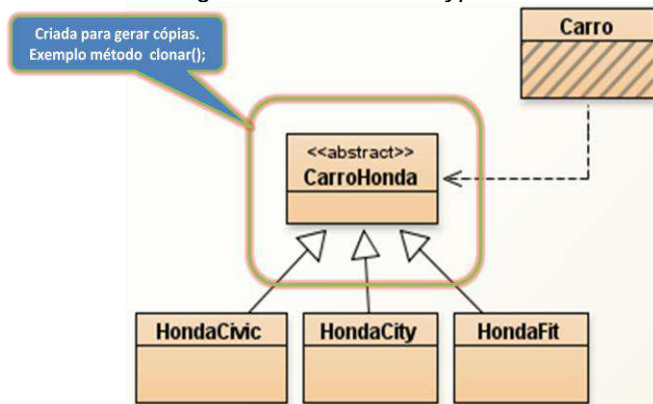
public class Batalha {
    public static void main(String[] args) {
        if (args[0].equals("terra")) {
            Armas arma = new Exercito();
        }
        else if (args[0].equals("mar")) {
            Armas arma = new Marinha();
        }
        else if (args[0].equals("ar")) {
            Armas arma = new Aeronautica();
        }
    }
}
```

4.5.4 Prototype

Intenção: Especifica que tipos de objetos criarem usando uma instância prototípica do objeto. Cria novos objetos através da cópia deste protótipo.

Aplicabilidade: Este padrão deve ser utilizado quando a aplicação deve ser independente de como os produtos são criados, compostos, e representados, e, adicionalmente: As classes a serem instanciadas são definidas em tempo de execução (por exemplo, *dynamic loading*); Deseja-se evitar criar uma hierarquia de fábricas paralelas a hierarquia de classes; Quando a instanciação da classe pode ter um de algumas poucas combinações diferentes de estado. Isto pode ser mais conveniente criar um número correspondente de protótipos e cloná-los ao invés de instanciar a classe manualmente a cada vez com o estado apropriado.

Figura 4.14 GoF *Prototype*



Fonte: Próprio Autor

```
public abstract class CarroHonda {  
    protected double valor;  
    public abstract CarroHonda clonar();  
}  
  
public class HondaCivic extends CarroHonda {  
    protected HondaCivic(HondaCivic civic) {  
    }  
    public HondaCivic() {  
        this.valor = 70000;  
    }  
}
```

```
        public CarroHonda clonar() {
            return new HondaCivic(this);
        }
    }
    public class HondaCity extends CarroHonda {
        protected HondaCity(HondaCity city) {
        }
        public HondaCity() {
            this.valor = 60000;
        }
        public CarroHonda clonar() {
            return new HondaCity(this);
        }
    }
    public class HondaFit extends CarroHonda {
        protected HondaFit(HondaFit fit) {
        }
        public HondaFit() {
            this.valor = 50000;
        }
        public CarroHonda clonar() {
            return new HondaFit(this);
        }
    }
    public class Carro {
        public static void main(String[] args) {
            HondaCivic civic = new HondaCivic();
            CarroHonda civicNovo = civic.clonar();
            HondaCity city = new HondaCity();
            CarroHonda cityNovo = city.clonar();
            HondaFit fit = new HondaFit();
            CarroHonda fitNovo = fit.clonar();
        }
    }
}
```

4.5.5 Singleton

Intenção: Garantir que uma classe possui apenas uma única instância. Provê um ponto de acesso global a ela.

Aplicabilidade: Este padrão deve ser utilizado quando deve haver apenas uma instância de cada classe e esta instância deve ser acessível a todos os

clientes a partir de um ponto de acesso conhecido. Ou quando uma única instância deve ser extensível apenas por subclasses, e os clientes devem apenas utilizar a instância estendida, sem modificar seu código.

```
public class Singleton {  
    private static Singleton instancia;  
    private Singleton() {  
    }  
    public static Singleton getInstancia() {  
        if (instancia == null)  
            instancia = new Singleton();  
        return instancia;  
    }  
}
```

4.6 Padrão de Projeto GoF Estrutural

Padrões estruturais que descrevem as formas para relacionar as classes e objetos. Padrões estruturais focam em como criar estruturas de maior porte através da composição de classes e objetos. Padrões estruturais de classe utilizam herança para compor interfaces ou implementações. Padrões estruturais de objeto utilizam composição de objetos para prover novas funcionalidades. A flexibilidade adicional da composição de objetos vem da habilidade de mudar esta composição em tempo de execução, o que é impossível na composição estática de classes.

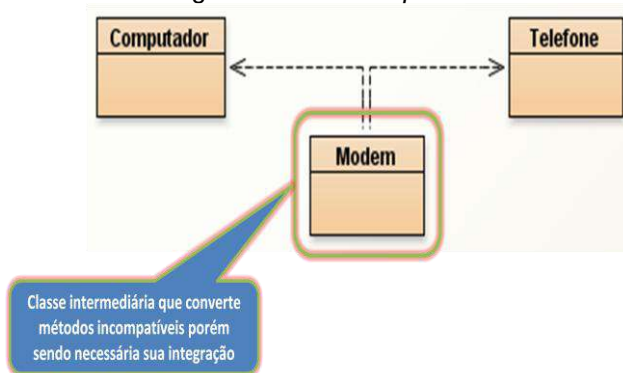
4.6.1 Adapter

Intenção: Converte a interface de uma classe na interface esperada pelo cliente. Compatibiliza classes de interfaces não compatíveis, permitindo que trabalhem em conjunto.

Aplicabilidade: Este padrão deve ser utilizado quando se deseja utilizar uma classe já existente, e sua interface não atende a interface que você precisa. Quando se deseja criar uma classe reusável que coopera com classes ainda não conhecidas ou não criadas, ou seja, classes que não necessariamente possui interfaces compatíveis. Necessita-se usar diversas subclasses já existentes, mas é impraticável adaptar suas interfaces através da criação de

subclasses para cada uma delas. Um objeto adaptador pode adaptar a interface da superclasse.

Figura 4.15 GoF Adapter



Fonte: Próprio Autor

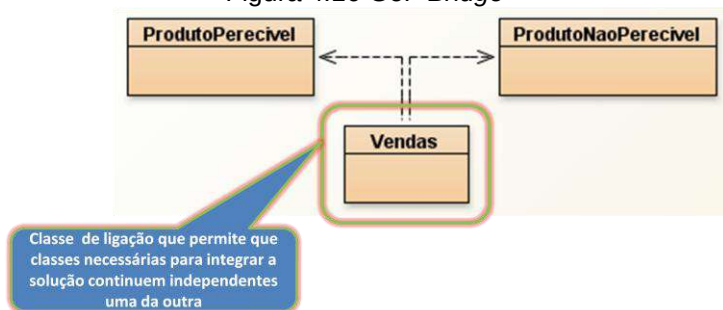
```
public class Computador {
    public static String enviaSinalDigital() {
        return ("Sinal Digital");
    }
}
public class Telefone {
    public static String enviaSinalAnalogico() {
        return ("Sinal Analogico");
    }
}
public class Modem {
    public static String modulaSinal(Computador c) {
        return ("Sinal modulado");
    }
    public static String demodulaSinal(Telefone t) {
        return ("Sinal demodulado");
    }
}
```

4.6.2 Bridge

Intenção: Desacopla uma abstração de sua implementação, de modo que as duas possam variar independentemente.

Aplicabilidade: Este padrão pode ser utilizado nas seguintes situações: Deseja-se evitar uma dependência direta entre a abstração e sua implementação. Este pode ser o caso, por exemplo, quando a implementação tiver de ser selecionada ou substituída em tempo de execução; Ambas as abstrações e suas implementações devem ser extensíveis através da criação de subclasses. Neste caso o padrão *bridge* permite que diferentes abstrações e implementações possam ser combinadas e estendidas independentemente. Mudanças na implementação de uma abstração não deve impactar em seus clientes, isto é, seu código não deve ser recompilado.

Figura 4.16 GoF Bridge



Fonte: Próprio Autor

```
public class ProdutoPerecivel {
    public void prazoValidade() {
    }
    public void definirRotulo() {
    }
}
public class ProdutoNaoPerecivel {
    public void realizarDepreciacao() {
    }
    public void definirUtilidade() {
    }
}
public class Vendas {
    private ProdutoPerecivel pp;
    private ProdutoNaoPerecivel pnp;
    private String nf;
    //gets...
```

```
//sets...
public void realizarVenda(ProdutoPerecivel pp,ProdutoNaoPerecivel pnp) {
}
}
```

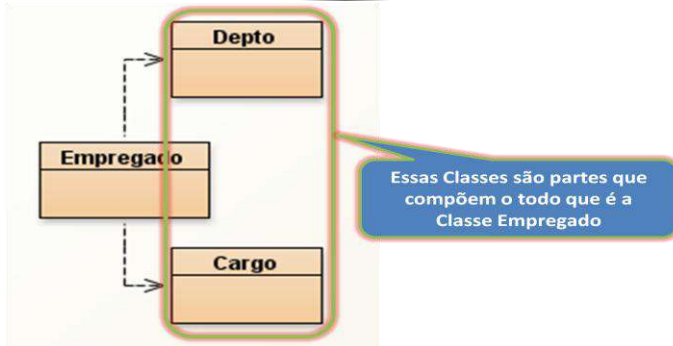
4.6.3 Composite

Intenção: Compõe objetos em estruturas de árvores para representar hierarquias parte-todo. Permite que clientes tratem de modo uniforme tanto objetos individuais quanto suas composições.

Aplicabilidade: Este padrão deve ser utilizando quando se deseja representar hierarquias parte-todo. Ou quando se deseja que clientes sejam capazes de ignorar as diferenças entre composições dos objetos e os objetos individualmente. Clientes irão tratar uniformemente todos os objetos na estrutura composta.

```
public class Depto {
    private int codigo;
    private String descricao;
    //gets...
    //sets...
}
public class Cargo {
    private int codigo;
    private String descricao;
    //gets...
    //sets...
}
public class Empregado {
    private int matricula;
    private String nome;
    private Depto departamento;
    private Cargo cargo;
    //gets...
    //sets...
}
```

Figura 4.17 GoF *Composite*



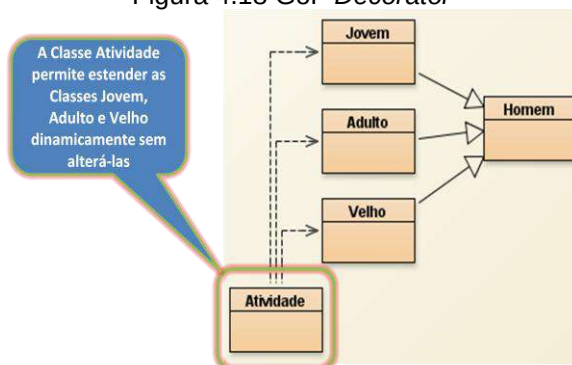
Fonte: Próprio Autor

4.6.4 Decorator

Intenção: Anexa dinamicamente responsabilidade adicional a um objeto. Provê uma alternativa flexível ao uso de herança como mecanismo de extensão de funcionalidade.

Aplicabilidade: Utilize este padrão para adicionar responsabilidades individuais a objetos dinamicamente e transparentemente, isto é, sem afetar outros objetos. Utilize este padrão para responsabilidades que podem ser "removidas". Ou ainda quando a extensão de funcionalidade através da criação de subclasses é impraticável. Em algumas situações um grande número de extensões independentes é possível e isto poderá produzir uma grande quantidade de subclasses para suportar cada uma das combinações.

Figura 4.18 GoF *Decorator*



Fonte: Próprio Autor

```
public class Homem {
    private int idade;
    //gets...
    //sets...
}
public class Jovem extends Homem {
    public String Atividade() {
        return("pouca");
    }
}
public class Adulto extends Homem {
    public String Atividade() {
        return("plena");
    }
}
public class Velho extends Homem {
    public String Atividade() {
        return("restrita");
    }
}
public class Atividade {
    public String definicaoEtaria(int idade) {
        if ((idade >= 0) && (idade < 4)) {
            Jovem h = new Jovem();
            return (h.Atividade());
            //novo_método()
        }
        else if ((idade >= 4) && (idade < 60)) {
            Adulto h = new Adulto();
            return (h.Atividade());
            //novo_método()
        }
        else {
            Velho h = new Velho();
            return (h.Atividade());
            //novo_método()
        }
    }
}
```

4.6.5 Facade

Intenção: Provê uma interface unificada para o conjunto de interfaces de um subsistema. Define uma interface de mais alto nível que torna um subsistema de mais fácil uso.

Aplicabilidade: Utilize este padrão quando: Deseja-se prover uma interface simples para um subsistema complexo. A fachada pode prover uma visão padrão do subsistema que é boa o suficiente para a maior parte dos clientes. Apenas clientes que necessitem de customização terão que olhar além da fachada. Existem muitas dependências entre clientes e as classes que implementam as abstrações. A introdução da fachada desacopla o subsistema dos clientes e outros subsistemas, promovendo a independência e portabilidade do subsistema. Deseja-se quebrar o sistema em camadas. Utilize a fachada para definir um ponto de entrada para cada um dos subsistemas. Se os subsistemas são dependentes, podem-se simplificar as dependências entre eles fazendo com que se comuniquem apenas através de suas fachadas.

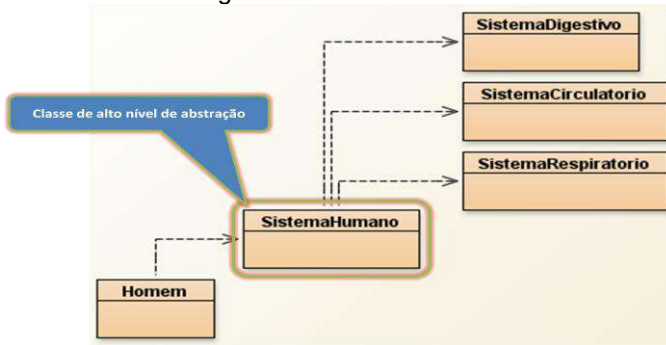
```
public class SistemaDigestivo {
    public static void realizarDigestao() {
    }
}
public class SistemaCirculatorio {
    public static void realizarCirculacao() {
    }
}
public class SistemaRespiratorio {
    public static void realizarRespiracao() {
    }
}
public class SistemaHumano {
    public void inicializacao() {
        SistemaRespiratorio nariz = new SistemaRespiratorio();
        nariz.realizarRespiracao();
        SistemaDigestivo intestino = new SistemaDigestivo();
        intestino.realizarDigestao();
        SistemaCirculatorio arteria = new SistemaCirculatorio();
        arteria.realizarCirculacao();
    }
}
```

```

public class Homem {
    public static void main(String[] args) {
        humano = new SistemaHumano();
        humano.inicializacao();
    }
}

```

Figura 4.19 GoF Facade



Fonte: Próprio Autor

4.6.6 Flyweight

Intenção: Usa compartilhamento para suportar um grande número de pequenos objetos de forma eficiente.

Aplicabilidade: Este padrão deve ser utilizado quando: Uma aplicação utiliza um grande número de objetos; Armazenamento tem custo elevado devido à grande quantidade de objetos; Muitos grupos de objeto podem ser substituídos por relativamente poucos objetos compartilhados. A aplicação não depende da identidade do objeto. Uma vez que os objetos podem ser compartilhados, testes de identidade irão retornar *true* para objetos conceitualmente distintos.

```

public class Cliente {
    private int codigo;
    private String nome;
    private String email;
    private String fone;
    //gets...
    //sets...
}

```

```
//métodos...
}  
public class ClienteVIP extends Cliente {  
    private boolean especial;  
    //gets...  
    //sets...  
    //métodos...  
}  
public class ClienteNegativado extends Cliente {  
    private boolean bloqueio;  
    //gets...  
    //sets...  
    //métodos...  
}
```

Figura 4.20 GoF Flyweight



Fonte: Próprio Autor

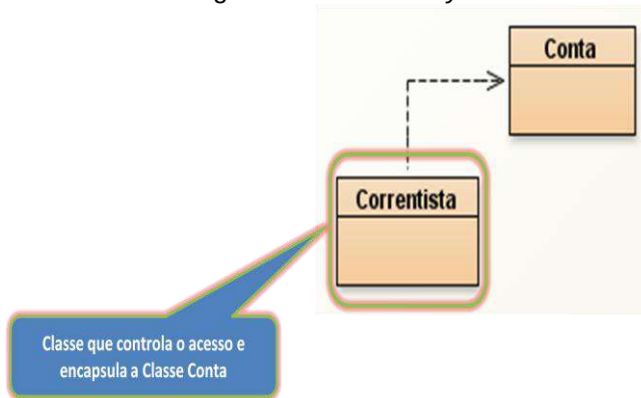
4.6.7 Proxy

Intenção: Provê um ponto de atendimento para que outro objeto possa controlar o acesso ao primeiro.

Aplicabilidade: Proxy é aplicável sempre que existe a necessidade de referências mais versáteis ou sofisticadas que um ponteiro para um objeto. Exemplos comuns são: Proxy remoto provendo um representante local para

um objeto em um espaço de memória diferente; Proxy virtual criando objetos custosos sobre demanda; Proxy de proteção controlando o acesso ao objeto original; Referência inteligente que executa ações adicionais quando o objeto original é acessado.

Figura 4.21 GoF Proxy



Fonte: Próprio Autor

```
public class Conta {
    private double saldo;
    private String tipo;
    //gets...
    //sets...
    public void movimentaSaldo(String tipo, double valor) {
        double aux = 0;
        if (tipo.equals("C")) {
            aux = getSaldo();
            aux = aux + valor;
            setSaldo(aux);
        } else {
            aux = getSaldo();
            aux = aux - valor;
            setSaldo(aux);
        }
    }
}

public class Correntista {
    private String usuario;
```

```
private String senha;  
private String conta;  
private String tipo;  
private float valor;  
//gets...  
//sets...  
public void acesarConta(String usuario, String senha) throws Exception {  
    try  
    {  
        //entrada de dados do usuário  
        if ((usuario.equals(getUsuario())) && (senha.equals(getSenha())) {  
            //entrada de dados da conta  
            Conta conta = new Conta();  
            conta.movimentaSaldo(valor)  
        } else {  
            throw new Exception("Acesso negado");  
        }  
    }  
    catch (Exception ex) {  
        throw (ex);  
    }  
}
```

4.7 Padrão de Projeto GoF Comportamental

Padrões comportamentais são relacionados aos algoritmos e responsabilidades associados a cada objeto. Padrões comportamentais estão relacionados com algoritmos e atribuição de responsabilidades entre objetos. Estes padrões não descrevem apenas os padrões de classes e objetos, mas também os padrões de comunicação entre estas classes e objetos. Caracterizam complexos fluxos de controle, difíceis de acompanhar em tempo de execução. Eles mudam o foco do fluxo de controle e permite que se concentre apenas na forma como os objetos estão interconectados. Padrões comportamentais de classes utilizam herança para distribuir o comportamento entre classes, que inclui os padrões *Template Method* – o mais simples deles, e o *Interpreter*. Padrões comportamentais de objetos utilizam composição de objetos, ao invés de herança, para realização de tarefas que um único objeto não poderia realizar. Estes objetos podem manter referências explícitas entre si, porém isto trás um maior acoplamento. Outros padrões permitem um

menor nível de acoplamento, como o *Memento*, *Chain of Responsibility* e *Observer*.

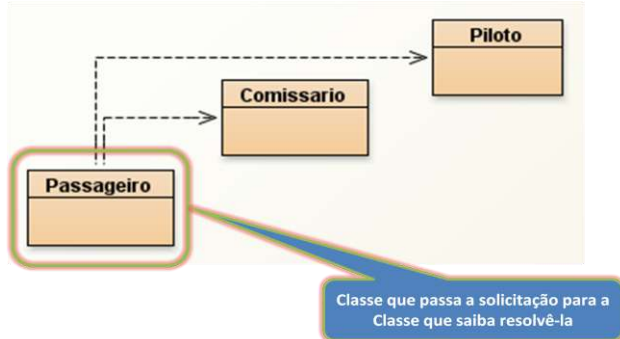
4.7.1 Chain of Responsibility

Intenção: Evita acoplamento entre solicitantes e atendentes permitindo que mais de um objeto tenha chance de tratar a solicitação. Encadeia os atendentes e passa a solicitação através desta cadeia permitindo que todos eles a tratem.

Aplicabilidade: Este padrão deve ser usado quando: mais de um objeto pode tratar uma solicitação e o objeto que a tratará não é conhecido a priori. O objeto que trata a solicitação deve ser escolhido automaticamente; deve-se emitir uma solicitação para um dentre vários objetos, sem especificar explicitamente o receptor; o conjunto de objetos que pode tratar uma solicitação deveria ser especificado dinamicamente.

```
public class Piloto {  
    public static void informacaoVoo() {  
    }  
}  
public class Comissario {  
    public void atendimentoBordo() {  
    }  
}  
public class Passageiro {  
    public void solicitacao(String tipo) {  
        if (tipo.equals("voo")) {  
            new Piloto().informacaoVoo();  
        } else if (tipo.equals("bordo")) {  
            new Comissario().atendimentoBordo();  
        }  
    }  
}
```

Figura 4.22 GoF *Chain Responsibility*



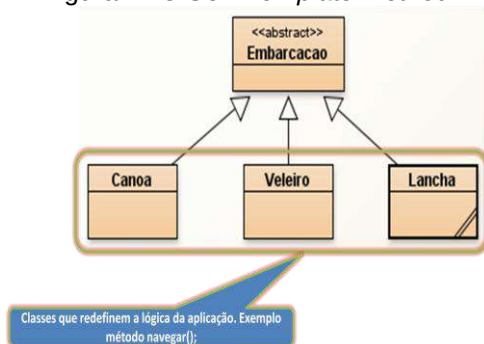
Fonte: Próprio Autor

4.7.2 Template Method

Intenção: Permite subclasses redefinir os passos de um algoritmo.

Aplicabilidade: Este padrão deve ser utilizado: Para implementar partes invariantes de um algoritmo uma única vez e deixar que as subclasses implementem o comportamento que varia. Quando o comportamento padrão entre subclasses devam ser fatorados e localizados em uma classe comum para evitar duplicação de código; Para controlar extensões por subclasses. Pode-se definir um *template method* que chama operações gancho em pontos específicos, permitindo extensões apenas nestes pontos.

Figura 4.23 GoF *Template Method*



Fonte: Próprio Autor

```
public abstract class Embarcacao {
```

```
    public abstract void fazerNavegacao();
    public void fazerAtracao() {
        //amarrar
    }
}
public class Canoa extends Embarcacao
{
    public void fazerNavegacao(){
        //remar
    }
}
public class Veleiro extends Embarcacao
{
    public void fazerNavegacao(){
        //velejar
    }
}
public class Lancha extends Embarcacao
{
    public void fazerNavegacao(){
        //pilotar
    }
}
```

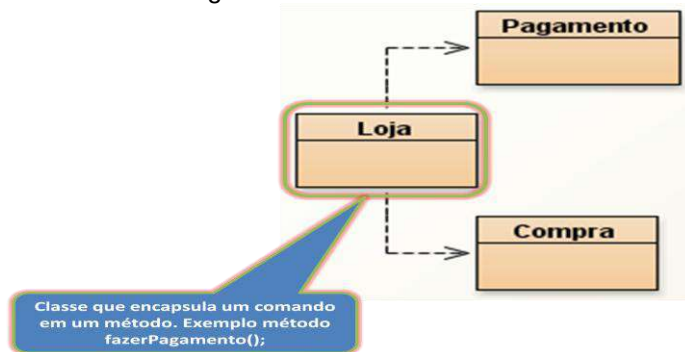
4.7.3 Command

Intenção: Encapsula uma solicitação em um objeto, permitindo que se parametrize clientes com diferentes solicitações, filas ou registros de solicitações, suportando ainda que operações sejam desfeitas.

Aplicabilidade: Utilize este padrão para: Parametrizar objetos por uma ação a ser executada. Você pode expressar tal parametrização numa linguagem procedural através de uma função callback, ou seja, uma função que é registrada em algum lugar para ser chamada em um momento mais adiante. Os *Commands* são uma substituição orientada a objetos para callbacks; Especificar, enfileirar e executar solicitações em tempos diferentes. Um objeto *Command* pode ter um tempo de vida independente da solicitação original. Se o receptor de uma solicitação pode ser representado de uma maneira independente do espaço de endereçamento, então você pode transferir um objeto *Command* para a solicitação para um processo diferente e lá atender a solicitação; Suportar desfazer operações. A operação *Execute*, de *Command*,

pode armazenar estados para reverter seus efeitos no próprio comando. A interface do *Command* deve ter acrescentada uma operação *Unexecute*, que o reverte efeitos de uma chamada anterior de *Execute*. Os comandos executados são armazenados em uma lista histórica. O nível ilimitado de desfazer e refazer operações são obtidos percorrendo esta lista para trás e para frente, chamando operações *Unexecute* e *Execute*, respectivamente.

Figura 4.24 GoF Command



Fonte: Próprio Autor

```
public class Compra {
    public static void fazerPagamento(double valor, String tipo) {
    }
}

public class Pagamento {
    public static String especie() {
        return ("dinheiro");
    }
    public static String credito() {
        return ("cartão de crédito");
    }
    public static String debito() {
        return ("cartão de débito");
    }
}

public class Loja {
    public static void main(String[] args) {
        Compra.fazerPagamento(100, new Pagamento().especie());
        Compra.fazerPagamento(100, new Pagamento().debito());
    }
}
```

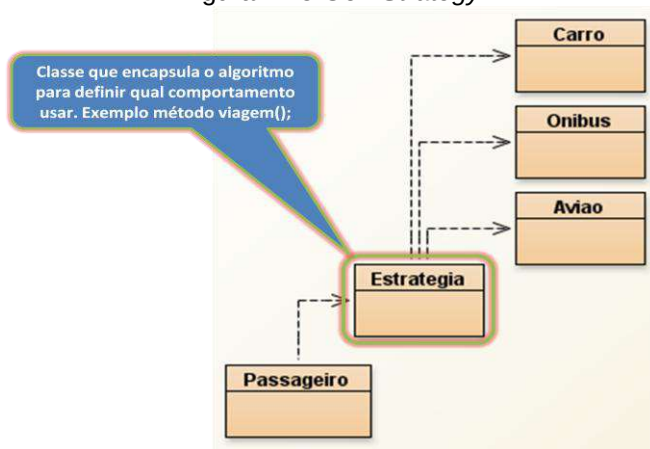
```
}  
}
```

4.7.4 Strategy

Intenção: Encapsular um algoritmo dentro de uma classe.

Aplicabilidade: Este padrão deve ser utilizado quando: Diversas classes relacionadas diferem apenas em comportamento. Estratégias provêm formas de configurar a classe com um dos muitos comportamentos; Necessita-se de diversas variações de um algoritmo; Um algoritmo utiliza dados que não devem ser conhecidos pelo cliente. Utilize este padrão para evitar expor estruturas de dados complexas e específicas do algoritmo. Um classe define diversos comportamentos, e estes aparecem como instruções condicionais múltiplas. Ao invés de manter estas condicionais, mova os trechos condicionais para sua própria classe de estratégia.

Figura 4.25 GoF Strategy



Fonte: Próprio Autor

```
public class Carro {  
    public void viajar() {  
        //custo médio & tempo médio  
    }  
}  
public class Onibus {
```

```
    public void viajar() {
        //custo baixo & tempo alto
    }
}
public class Aviao {
    public void viajar() {
        //custo alto & tempo baixo
    }
}
public class Estrategia {
    private static void viagem(float custo, int tempo) {
        if ((custo < 100) && (tempo > 10))
        {
            Onibus a = new Onibus();
        }
        else if ((custo > 1000) && (tempo < 10))
        {
            Aviao a = new Aviao();
        }
        else
        {
            Carro a = new Carro();
        }
    }
}
public class Passageiro {
    public static void main(String[] args) {
        Estrategia.viagem(80,8);
    }
}
```

4.7.5 State

Intenção: Alterar o comportamento de um objeto quando seu estado muda.

State Aplicabilidade: Este padrão deve ser utilizado quando: O comportamento de um objeto depende fortemente do seu estado e ele deve alterar o seu comportamento em tempo de execução dependendo do seu estado. Os métodos têm instruções condicionais grandes em que as condições dependem do estado do objeto. Este estado é normalmente

representado por uma ou mais constantes do tipo enumerado. Frequentemente, vários métodos contêm esta mesma estrutura condicional. O padrão State coloca cada ramo da instrução condicional numa classe separada. Desta forma, o estado do objeto pode ser tratado como um objeto ele próprio, o qual pode variar independentemente.

```
public class Oficial {
    private String patente;
    public Oficial(String patente)
    {
        this.patente = patente;
        verificaAtribuicoes(patente);
    }
    public void verificaAtribuicoes(String patente)
    {
        if (patente.equals("General")) {
            //executa atribuições
        }
        else if (patente.equals("Coronel")) {
            //executa atribuições
        }
        else if (patente.equals("Major")) {
            //executa atribuições
        }
        else if (patente.equals("Capitão")) {
            //executa atribuições
        }
        else if (patente.equals("Tenete")) {
            //executa atribuições
        }
    }
}
```

4.7.6 Observer

Intenção: Define um regime de notificação de objetos de mudanças para outro objeto.

Aplicabilidade: Use este padrão quando uma mudança em um objeto deve causar mudança em outros e não se sabem quais e quantos são os outros objetos; quando um objeto deve ser capaz de notificar outros objetos sem

assumir nada sobre que são estes outros objetos. Em outras palavras, você não quer que estes objetos estejam fortemente acoplados entre si.

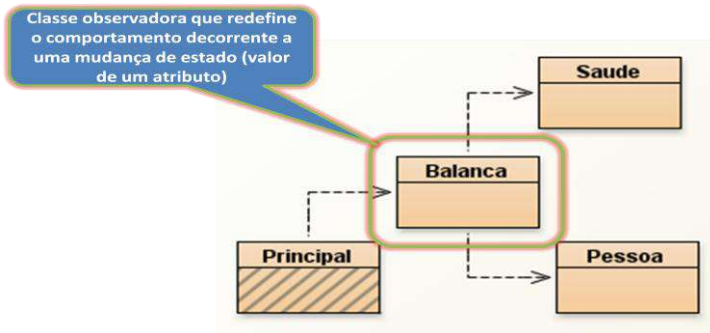
```
public class Pessoa {
    private int peso;
    public void setPeso(int peso) {
        this.peso = peso;
    }
    public int getPeso() {
        return (this.peso);
    }
}

public class Saude {
    public static void saudavel() {
    }
    public static void obeso() {
    }
}

public class Balanca {
    public static void medir(Pessoa p) {
        if (p.getPeso() < 80) {
            Saude.saudavel();
        }
        else {
            Saude.obeso();
        }
    }
}

public class Principal {
    public static void main(String[] args) {
        Pessoa p = new Pessoa();
        p.setPeso(60);
        Balanca.medir(p);
    }
}
```

Figura 4.26 GoF *Observer*



Fonte: Próprio Autor

4.7.7 Interpreter

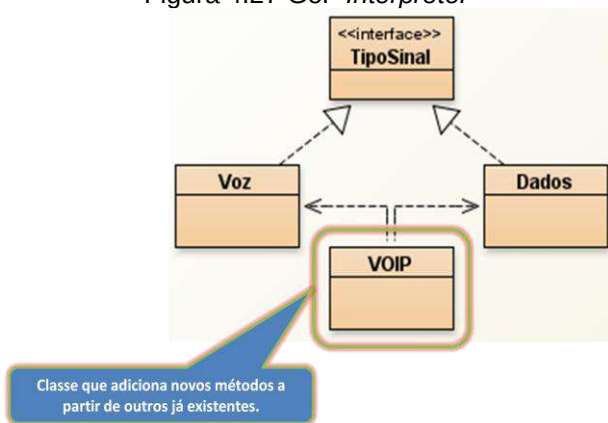
Intenção: Dada uma linguagem, cria uma representação de sua gramática, juntamente com um interpretador que utiliza esta representação para interpretar sentenças na linguagem.

Aplicabilidade: Este padrão deve ser utilizado quando existe uma linguagem a ser interpretada e é possível representar expressões nesta linguagem como árvores sintáticas abstratas. Esse padrão funciona melhor quando: A gramática é simples. Para gramáticas complexas, a hierarquia de classes se torna muito grande e não gerenciável. Outras ferramentas como geradores de parsers são melhores alternativas nestas situações, pois podem interpretar expressões sem construir árvores sintáticas abstradas, o que pode salvar espaço e possivelmente tempo; Eficiência não é crítico. Os interpretadores mais eficientes são usualmente não implementados através da interpretação de árvores de parser diretamente, mas primeiro é feita uma tradução para um outro formato.

```
public interface TipoSinal {  
    String tratarSinal();  
}  
  
public class Voz implements TipoSinal {  
    public String tratarSinal() {  
        return ("Analógico");  
    }  
}  
  
public class Dados implements TipoSinal {  
    public String tratarSinal() {
```

```
        return ("Digital");
    }
}
public class VOIP {
    public void usarSinal() {
        Voz v = new Voz();
        Dados d = new Dados();
        v.tratarSinal();
        d.tratarSinal();
    }
}
```

Figura 4.27 GoF Interpreter



Fonte: Próprio Autor

4.7.8 Memento

Intenção: Salvar e restaurar o estado interno de um objeto.

Aplicabilidade: Use este padrão quando uma parte ou todo o estado de um objeto deve ser guardado e possivelmente recuperado posteriormente; a obtenção direta do estado quebraria a proteção de informação expondo detalhes de implementação.

```
import Java.util.Arrays;
import Java.util.ArrayList;
public class Contatos
```

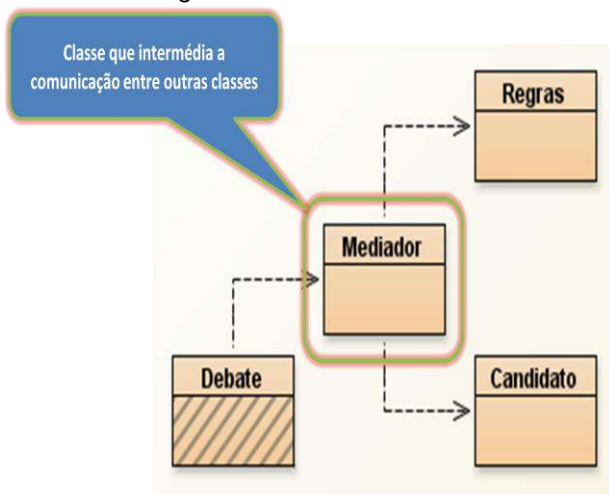
```
{  
    private int codigo;  
    private String nome;  
    private ArrayList<String> email;  
    private ArrayList<String> fones;  
    //gets...  
    //sets...  
}
```

4.7.9 Mediator

Intenção: Define um objeto que encapsula o modo como um conjunto de objetos interage. Promove um acoplamento fraco entre objetos, evitando que referenciem explicitamente um ao outro, e com isto permitindo que se possa variar independentemente a interação entre eles.

Aplicabilidade: Use este padrão quando um conjunto de objetos se comunica de maneira complexa; o reuso de objetos é dificultado porque este referencia e se comunica com muitos outros objetos; o comportamento operações deve ser customizável sem a criação de inúmeras subclasses.

Figura 4.28 GoF Mediator



Fonte: Próprio Autor

```
public class Regras {
```

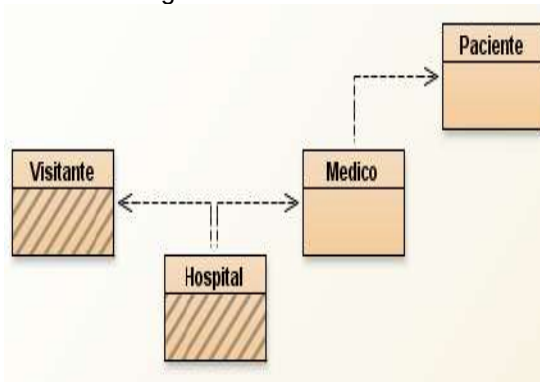
```
    public void defineRegraUm() {
    }
    public void defineRegraDois() {
    }
    public void defineRegraTres() {
    }
}
public class Candidato {
    public void explicaPropostaUm() {
    }
    public void explicaPropostaDois() {
    }
    public void explicaPropostaTres() {
    }
}
public class Mediator {
    public void mediarCandidado(Candidato candidato) {
    }
    public void aplicarRegras(Regras regra) {
    }
}
public class Debate {
    public static void main(String[] args) {
        Candidato x = new Candidato();
        Candidato y = new Candidato();
        Regras r = new Regras();
        Mediator m = new Mediator();
        m.mediarCandidado(x);
        m.aplicarRegras(r);
        m.mediarCandidado(y);
        m.aplicarRegras(r);
    }
}
```

4.7.10 Visitor

Intenção: Define uma nova operação em uma classe sem trocá-la.

Aplicabilidade: Este padrão deve ser utilizado quando: Uma estrutura de objetos contém muitas classes de objetos com interfaces distintas, e deseja-se executar operações nestes objetos que dependem de sua classe concreta;

Muitas operações distintas e não relacionadas precisam ser executadas em objetos em uma estrutura de objetos, e deseja-se evitar poluir estas classes com estas operações. *Visitor* permite que operações relacionadas sejam mantidas juntas definindo-as em uma classe. Quando a estrutura do objeto é compartilhada por muitas aplicações, utilize *Visitor* para colocar as operações apenas nas aplicações que necessitem dela; As classes que definem a estrutura do objeto raramente mudam, porém frequentemente deseja-se adicionar novas operações sobre a estrutura. Modificar a classe da estrutura de objetos requer redefinir a interface para todos os *visitors*, o que é potencialmente custoso. Se as classes da estrutura de objetos mudam constantemente, provavelmente é melhor definir as operações nestas classes.

Figura 4.29 GoF *Visitor*

Fonte: Próprio Autor

```
public class Paciente {
    public void solicitaTratamento() {
    }
}
public class Medico {
    public void realizaTratamento(Paciente p) {
    }
}
public class Visitante {
    public void visitaPaciente(Paciente p) {
    }
}
public class Hospital {
```

```
public static void main(String[] args) {  
    Paciente p = new Paciente();  
    Medico m = new Medico();  
    Visitante v = new Visitante();  
    p.solicitaTratamento();  
    m.realizaTratamento(p);  
    v.visitaPaciente(p);  
}  
}
```

4.7.11 Iterator

Intenção: Provê uma forma de acessar seqüencialmente os elementos de um agregado de objetos, sem expor a sua representação interna.

Aplicabilidade: O padrão *iterator* deve ser utilizado para acessar agregações de objetos sem expor sua representação interna; para suportar diversas “varreduras transversais” em agregados de objetos; para prover uma interface uniforme para varrer estruturas agregadas diferentes.

```
import Java.util.*;  
public class Iteracao {  
    public static void main(String args[]) {  
        ArrayList lista = new ArrayList();  
        lista.add("elemento1");  
        lista.add("elemento2");  
        lista.add("elemento3");  
        lista.add("elemento4");  
        lista.add("elemento5");  
        Iterator iterar = lista.iterator();  
        while(iterar.hasNext()) {  
            Object element = iterar.next();  
        }  
        ListIterator iterarLista = lista.ListIterator();  
        while(iterarLista.hasNext()) {  
            Object element = iterarLista.next();  
            iterarLista.set(element + "modificarLista");  
        }  
    }  
}
```

4.8 Padrão Arquitetural *Model View Control* (MVC)

Visa separar a lógica do negócio da lógica da apresentação, permitindo o desenvolvimento, teste e manutenção isolados. O *model* (domínio) é usado para definir e gerenciar o domínio da informação e notificar observadores sobre mudanças nos dados; O *view* (apresentação) apresenta o modelo num formato adequado ao utilizador; e O *controller* (regras) recebe a entrada de dados e inicia a resposta ao utilizador ao invocar objetos do modelo.

Exemplo: aluno, professor e turma fazem parte do domínio de um sistema acadêmico. Operações como calcular a média final do aluno ou o índice de faltas da turma fazem parte da lógica de domínio. A forma como o dado é armazenado é de responsabilidade do modelo.

Caso Prático: uma aplicação web em que a visão é um documento HTML. O controlador recebe uma entrada GET ou POST após um estímulo do utilizador e decide como processá-la, invocando objetos do domínio para tratar a lógica de negócio, e por fim invocando uma visão para apresentar a saída.

A associação direta é um relacionamento que descreve um conjunto de vínculos, onde cada vínculo é definido como uma conexão semântica entre objetos. A associação indireta é um relacionamento entre elementos, um independente e outro dependente.

Uma utilização do padrão arquitetural MVC seria da seguinte forma: a camada *View* seriam as páginas ASPX ou JSP, a camada *Model* seriam as classes de domínio da aplicação e a camada de *Controller* seriam as classes com as regras de negócio. Porém, não foi essa a utilização original idealizada para o MVC quando criado para ser aplicada na linguagem Smalltalk. Nessa o usuário interage com a interface de alguma maneira no *View*, o *Controller* manipula o evento do usuário, daí acessa o *Model* acessando as classes de cadastro.

4.9 Apresentações

Padrões. Em < <http://www.slideshare.net/alvarofpinheiro/fundamentos-de-padres-de-projeto-de-software> >

4.10 Sugestões de Leitura

GAMMA, Erich; *et al.* Padrões de Projeto. Em <
<http://c2.com/cgi/wiki?DesignPatternsBook> >

4.11 Referências Web

Categorias de Padrões de Projeto de Software (wiki). Em <
<http://qoo.gl/jOvwFs> >
Categorias de Programação Orientada a Objetos (wiki). Em <
<http://qoo.gl/Cu5QKh> >
Design Patterns (wiki). Disp.:<
[http://en.wikipedia.org/wiki/Design_pattern_\(computer_science\)](http://en.wikipedia.org/wiki/Design_pattern_(computer_science)) >
GRASP. Em < [http://en.wikipedia.org/wiki/GRASP_\(object-oriented_design\)](http://en.wikipedia.org/wiki/GRASP_(object-oriented_design)) >
MVC (wiki). Em < <http://pt.wikipedia.org/wiki/MVC> >
Padrões de Projeto (wiki). Disp.:<
[http://pt.wikipedia.org/wiki/Padr%C3%A3o de projeto de software](http://pt.wikipedia.org/wiki/Padr%C3%A3o_de_produto_de_software) >
Software Design Patterns.<
<http://commons.wikimedia.org/w/index.php?title=Software+design+patterns&uselang=pt> >

4.12 Referências Bibliográficas

FOWLER, Martin. UML Essencial. 2ªed. Porto Alegre: Bookman, 2004.
GAMMA, Erich; *et al.* Padrões de Projeto. Porto Alegre: Bookman, 2007.
PRESSMAN, Roger. Engenharia de Software: uma abordagem profissional. 7ªed. Porto Alegre: Bookman, 2011.
RUMBAUGH, James; *et al.* Modelagem e Projetos Baseados em Objetos com UML. 2ªed. RJ: Campus, 2006.
RUMBAUGH, James; JACOBSON, Ivar; BOOCH, Grady. UML: Guia do Usuário. 1ªed. RJ: Campus, 2006.
SOMMERVILLE, Ian. Engenharia de Software. 8ªed. Pearson, 2007.

5 Referências Bibliográficas

AHO, Alfred; *et al.* Compilers: Principles, Techniques, and Tools. 2ªed. US: Pearson, 2006.

BEDREGAL, B.; BENEDITO, A. Introdução à Teoria das Ling. Formais dos Autômatos e da C. RN: Edunp, 2010.

BENNETT, Simon; *et al.* Schaum's Outline of UML. 1ªed. UK: McGraw-Hill, 2005.

BEZERRA, Eduardo. Princípios de Análise e Projetos de Sistemas com UML. 1ªed. Rio de Janeiro: Campus, 2007.

CARDOSO, Caíque. UML na Prática: do problema ao sistema. 1ªed. São Paulo: CM, 2003.

DEITEL, Harvey; DEITEL, Paul. Java: Como Programar. 6ªed. US: Pearson, 2005.

DENNIS, Alan. Análise e Projeto de Sistemas. 2ªed. São Paulo: LTC, 2011.

DIVERIO, Tiarajú; *et al.* Teoria da Computação: Máquinas Universais e Comp. 3ªed. Porto Alegre: Bookman, 2011.

FERNANDES, Daniel. Análise de Sistemas Orientada ao Sucesso. 1ªed. Rio de Janeiro: Ciência Moderna, 2005.

FOWLER, Martin. UML Essencial. 2ªed. Porto Alegre: Bookman, 2004.

GAMMA, Erich; *et al.* Padrões de Projeto. Porto Alegre: Bookman, 2007.

HOPCROFT, John; *et al.* Introdução à Teoria de Autômatos, Linguagens e Computação. 1ªed. RJ: Campus, 2002.

KRUCHTEN, Philippe. Introdução ao RUP Rational Unified Process. 1ªed. Rio de Janeiro: Ciência Moderna, 2003.

LOUDEN, Kenneth. Compiladores: Princípios e Práticas. 1ªed. Pioneira Thomson Learning, 2004.

MEDEIROS, Ernani. Desenvolvendo Software com UML 2.0: Definitivo. 1ªed. São Paulo: Makron Books, 2004.

MELO, Ana. Desenvolvendo Aplicações com UML 2.2: do conceitual à implementação. 3ªed. RJ: Brasport, 2010.

MENEZES, Paulo. Linguagens Formais e Autômatos. 6ªed. Porto Alegre: Bookman, 2011.

PENDER, Tom. UML - A Bíblia. 1ªed. RJ: Campus, 2004.

PRESSMAN, Roger. Engenharia de Software: uma abordagem profissional. 7ªed. Porto Alegre: Bookman, 2011.

RUMBAUGH, James; *et al.* Modelagem e Projetos Baseados em Objetos com UML. 2ªed. RJ: Campus, 2006.

RUMBAUGH, James; JACOBSON, Ivar; BOOCH, Grady. UML: Guia do Usuário. 1ªed. RJ: Campus, 2006.

SCHACH, Stephen. Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos. McGraw-Hill, 2008.

SOMMERVILLE, Ian. Engenharia de Software. 8ªed. Pearson, 2007.

TUCKER, Allen; NOONAN, Robert. Linguagens de Programação: Princípios e Paradigmas. 2ªed. McGraw, 2009.

WAZLAWICH, Raul. Análise e Projeto de Sistemas de Informação Orientados a Objetos. RJ: Campus, 2011.

6 Referências Web

A Theory of Object-Oriented Design. Em < <http://goo.gl/fq3q5> >
Análise Orientada a Objetos wiki. Em < <http://goo.gl/N0Ktwf> >
Aulas de UML. Em < <http://goo.gl/2TRbUF> >
Autômatos Finitos Determinísticos wiki. Em < <http://goo.gl/3IJJxV> >
Autômatos Finitos Determinísticos. Em < <http://goo.gl/NZHyRP> >
Autômatos Finitos e Expressões Regulares. Em < <http://goo.gl/3Ox2WM> >
BNF for Java. Em < <http://cui.unige.ch/isi/bnf/JAVA/BNFindex.html> >
Categorias de Padrões de Projeto de Software (wiki). Em < <http://goo.gl/jOvwFs> >
Categorias de Programação Orientada a Objetos (wiki). Em < <http://goo.gl/Cu5QKh> >
Convenções de Java. Em < <http://www.oracle.com/technetwork/java/codeconventions-135099.html> >
Design Patterns (wiki). Disp.: < [http://en.wikipedia.org/wiki/Design_pattern_\(computer_science\)](http://en.wikipedia.org/wiki/Design_pattern_(computer_science)) >
Design Patterns Languages for Object-Oriented Programs (DP). Disp.: < <http://www.vincehuston.org/dp/> >
Design Patterns Languages for OOP. Em < <http://www.vincehuston.org/dp/> >
Engenharia de Software wiki. Em < http://pt.wikipedia.org/wiki/Engenharia_de_software >
Especificações API Java. Em < <http://download.oracle.com/javase/6/docs/api/> >
Exemplo de Modelagem UML com a ferramenta ASTAH. Em < <http://goo.gl/JvghSt> >
Exemplo de Modelagem UML. Em < <http://goo.gl/zw0YCF> >
Gramática BNF para SQL ANSI. Em < <http://savage.net.au/SQL/> >
GRASP. Em < [http://en.wikipedia.org/wiki/GRASP_\(object-oriented_design\)](http://en.wikipedia.org/wiki/GRASP_(object-oriented_design)) >
Guia Básico sobre UML. Em < <http://goo.gl/eGhizM> >
Guia da UML. Em < <http://goo.gl/3Wtx0b> >
Guia de Modelagem de Casos de Uso. Em < <http://goo.gl/PtwH3L> >
Guia de Modelagem de Negócios. Em < <http://goo.gl/W2U0B6> >
Guia de Referência de Convenções de Codificação Java. Em < <http://goo.gl/PfvJcb> >
Guia de Design. Em < <http://goo.gl/OpvFEC> >
Guia de Programação. Em < <http://goo.gl/NL4sX9> >
Langpop. Em < <http://www.langpop.com/> >

Mais sobre relacionamentos de Inclusão e Extensão. Em < <http://goo.gl/uH8X6Z> >

Manuais de Referência OMG-UML. Em < <http://www.omg.org/spec/UML/> >

Modelagem UML. Em < <http://goo.gl/iwtZh3> >

MVC (wiki). Em < <http://pt.wikipedia.org/wiki/MVC> >

Netbeans Aplicação. Em < <http://goo.gl/ND3QLK> >

Netbeans Quickstart. Em < http://netbeans.org/kb/docs/java/quickstart_pt_BR.html >

Object Management Group (OMG). Em < <http://www.omg.org/> >

Object-Oriented Programming (OOP). Em < <http://goo.gl/rIYFI5> >

Ohloh. Em < <http://www.ohloh.net/> >

Padrões de Projeto (wiki). Disp.: < [http://pt.wikipedia.org/wiki/Padr%C3%A3o de projeto de software](http://pt.wikipedia.org/wiki/Padr%C3%A3o_de_projeto_de_software) >

Paradigma (wiki). Em < <http://pt.wikipedia.org/wiki/Paradigma> >

Paradigma de Desenvolvimento (wiki). Em < <http://goo.gl/rIYFI5> >

Paradigma. Em < <http://pt.wikipedia.org/wiki/Paradigma> >

Referência de Convenções de Codificação em Java. Em < <http://goo.gl/PfvJcb> >

Software Design Patterns. < <http://commons.wikimedia.org/w/index.php?title=Software+design+patterns&uselang=pt> >

Template Casos de Uso. Disp.: < http://www.wthreex.com/rup/portugues/webtmpl/templates/reg/rup_ucspec.htm >

Template Mais Detalhado para Especificação de Casos de Uso. Disp.: < <http://goo.gl/5yKpBD> >

Template RUP para Especificação de Casos de Uso. Em < <http://goo.gl/GwnXCn> >

Tiobe. Em < <http://www.tiobe.com/index.php/content/company/Home.html> >

Tutorial Java. Em < <http://download.oracle.com/javase/tutorial/index.html> >

Unified Modeling Language (UML). Em < <http://www.uml.org/> >

7 Biografias

Alan Turing nasceu na Inglaterra em 1912, foi matemático, lógico, criptoanalista e cientista da computação. É um dos nomes importantes na Ciência da Computação. Formulou conceitos de algoritmo e computação com a máquina de Turing. Trabalhou no *National Physical Laboratory*. Saber mais. Disponível em < http://en.wikipedia.org/wiki/Alan_Turing >

Alfred Vaino Aho é cientista da computação canadense. Conhecido internacionalmente como co-autor da linguagem de programação AWK, junto com Peter Weinberger e Brian Kernighan, e também pela sua co-autoria no Livro do Dragão como é mais conhecido o livro, *Compilers: Principles, Techniques, and Tools*, escrito em parceria com Ravi Sethi e Jeffrey Ullman. Suas contribuições para área foram às versões iniciais das ferramentas Unix egrep e fgrep. Também possui livros nas áreas de algoritmos e estruturas de dados. Saber mais. Disponível em < http://pt.wikipedia.org/wiki/Alfred_Aho >

Avram Noam Chomsky nasceu nos Estados Unidos na Filadélfia em 1928, é um linguista e filósofo. É professor de Linguística no *Massachusetts Institute of Technology*. É autor de trabalhos fundamentais sobre as propriedades matemáticas das linguagens formais, sendo o seu nome associado à chamada Hierarquia de Chomsky. Seus trabalhos estão relacionados a uma abordagem matemática da linguagem. Saber mais. Disponível em < http://pt.wikipedia.org/wiki/Noam_Chomsky >

Christopher Date formou-se no Reino Unido pela High Wycombe Royal Grammar School em 1958 e recebeu o bacharelado em Matemática pela Universidade de Cambridge em 1962. Iniciou as atividades na área da computação como programador na Leo *Computers Ltd* em Londres onde também começou a lecionar. Em 1967 começou a trabalhar na IBM na Inglaterra também como programador e instrutor de computação, ficando entre os anos de 1969 e 1974 como o principal instrutor da área educacional da IBM. Saber mais. Disponível em < http://en.wikipedia.org/wiki/Christopher_J._Date >

Craig Larman é um canadense nascido em 1958 que trabalha como cientista da computação especializado em desenvolvimento iterativo e incremental, desenvolvimento de *software* ágil, análise orientada a objetos e modelagem ágil. Ele é o autor de vários textos sobre Scrum. Saber mais. Disponível em < http://en.wikipedia.org/wiki/Craig_Larman >

Erich Gamma é um dos quatro autores do livro *Design Patterns: Elements of Reusable Object-Oriented Software*. Ele é também o principal projetista do JUnit e do Eclipse. Nascido em 1961, em Zurique é cientista da computação e co-autor influente de livros didáticos sobre engenharia de *software*. Co-escreveu o JUnit framework de teste de *software* com Kent Beck e liderou o projeto do Eclipse da plataforma Java *Development Tools* (JDT). Também trabalhou na IBM Rational projeto Jazz. Saber mais. Disponível em < http://en.wikipedia.org/wiki/Erich_Gamma >

Grady Booch nasceu em 1955 nos Estados Unidos, é engenheiro de software e cientista-chefe da IBM Research. Obteve seu grau de bacharel pela *United States Air Force Academy* e título de mestre em engenharia elétrica pela Universidade da Califórnia em Santa Barbara. Saber mais. Disponível em < http://en.wikipedia.org/wiki/Grady_Booch >

Hal Varian, nascido em 1947, em Ohio, é economista especializado em microeconomia e economia da informação. É professor emérito nas universidades de Berkeley e Haas *School of Business*. Formou-se como bacharel em economia pelo e licenciado em matemática pelo MIT. Possui o doutorado em economia pela *Berkeley University*, e é doutor honorário pelo Instituto de Tecnologia de Karlsruhe, Alemanha. Já lecionou em *Stanford University*, *Oxford University*, *Michigan University*. Nos dias atuais é o economista-chefe da Google e publica regularmente artigos no *New York Times* e *Wall Street Journal*. Saber mais. Disponível em < http://en.wikipedia.org/wiki/Hal_Varian >

Harvey Deitel tem mais de 40 anos de experiência na área de computação. Graduou-se e fez o mestrado no *Massachusetts Institute of Technology* e o doutorado na *Boston University*. Possuindo mais de 20 anos de experiência no ensino universitário. Possuindo diversas publicações traduzidas em diversos idiomas. Saber mais.

Disponível em <
http://www.informit.com/authors/author_bio.aspx?ISBN=9780137058426 >

Ian Sommerville, inglês nascido em 1951 é professor titular de engenharia de software na Universidade de St. Andrews na Escócia. Destacado pesquisador no campo da engenharia de sistemas, segurança e informática social. Saber mais.

Disponível em <
http://en.wikipedia.org/wiki/Ian_Sommerville_%28academic%29 >

Idalberto Chiavenato, nascido em 1936 em São Paulo, é autor na área de administração de empresas e de recursos humanos. Atualmente, atua como conselheiro no Conselho Regional de Administração do Estado de São Paulo (CRA-SP) e é Presidente do Instituto Chiavenato de Educação. Saber mais.

Disponível em < http://pt.wikipedia.org/wiki/Idalberto_Chiaavenato >

Ivar Jacobson é um sueco nascido em 1939, se graduou no Instituto de Tecnologia de Chalmers, onde fez seu mestrado em Engenharia Elétrica e possuindo doutorado pela *Royal Institute of Technology de Estocolmo*. Saber mais.

Disponível em < http://en.wikipedia.org/wiki/Ivar_Jacobson >

James Gosling nasceu em Calgary no Canadá em 1955 é conhecido como o pai da linguagem de programação Java. Em 1977, formou Bacharel em Ciência da Computação pela Universidade de Calgary, e em 1983 tornou-se PhD em Ciência da Computação pela Universidade Carnegie Mellon. Enquanto estudava no doutorado, Gosling programou a versão original do editor de texto Emacs para UNIX, e antes de se juntar a Sun Microsystems ele construiu uma versão de UNIX para máquinas multiprocessadas, assim como diversos compiladores e sistemas de mensagens eletrônicas. Trabalhou de 1984 até 2010 na Sun Microsystems, atualmente está na Google. Saber mais.

Disponível em < http://pt.wikipedia.org/wiki/James_Gosling >

James Rumbaugh é Bacharel em Física pelo Massachusetts Institute of Technology (MIT), possui mestrado em Astronomia pela Caltech, e é Ph. D. em Ciência da Computação também pelo MIT. Trabalhou na *Digital Equipment Corporation* (DEC), como cientista. Foi um dos líderes na criação

da OMT e em 1994 se juntou a *Rational Software*, e junto com Ivar Jacobson e Booch de Grady desenvolveram a *Unified Modeling Language* (UML). Mais tarde, eles fundiram as suas metodologias de desenvolvimento de *software*, OMT, OOSE e Booch em um único processo batizado de *Rational Unified Process* (RUP). Saber mais.

Disponível em < http://en.wikipedia.org/wiki/James_Rumbaugh >

John von Neumann nasceu na Hungria em 1903 naturalizado como estadunidense, foi um matemático. Membro do Instituto de Estudos Avançados em Princeton, *New Jersey*, do qual também faziam parte Albert Einstein. Professor na Universidade de Princeton e um dos construtores do ENIAC. Trabalhou no desenvolvimento da Física Nuclear e contribuiu com diversas teorias, a exemplo as dos conjuntos, análise funcional, ergódica, mecânica quântica, ciência da computação, economia, teoria dos jogos, análise numérica, estatística, entre outras. Saber mais.

Disponível em < http://pt.wikipedia.org/wiki/John_von_Neumann >

Kenneth Laudon é professor de Sistemas de Informação na New York University's Stern School of Business. Bacharel em Economia pela *Stanford University* e Ph. D pela *Columbia University*. Autor de vários livros que tratam do tema Sistemas de Informação, Organizações e Sociedade. Saber mais.

Disponível em < <http://pages.stern.nyu.edu/~klaudon/> >

Patrick Naughton americano nascido em 1965 é um desenvolvedor de software conhecido como sendo um dos criadores originais da linguagem de programação Java, pois trabalhou na Sun Microsystems no Projeto Verde a origem do Java. Em 1994, saiu da Sun e foi para Starwave de propriedade de Paul Allen para desenvolver um servidor Java. É autor do livro *O Manual do Java*, Osborne, 1995 e co-autor do *Java: The Complete Reference*, com Herbert Schildt, Osborne, 1996. Em 1998, torna-se vice-presidente executivo da Infoseek. Saber mais.

Paul Deitel é graduado em Tecnologia da Informação e pós-graduação em Administração pela Massachusetts *Institute of Technology*. Trabalha como consultor em diversas organizações internacionais. Saber mais.

Disponível em < http://www.informit.com/authors/author_bio.aspx?ISBN=9780137058426 >

Peter Ferdinand Drucker, austríaco nascido 1909 e que faleceu recentemente em 2005, foi um filósofo e economista, que por suas contribuições a comunidade, passou a ser considerado o pai da administração moderna, se tornando um dos pensadores mais citados pela teoria da globalização em economia em geral. Saber mais. Disponível em < http://pt.wikipedia.org/wiki/Peter_Drucker >

Roger Pressman é Bacharel em Engenharia de Software pela University of Connecticut, com Mestrado pela *University of Bridgeport* e PhD pela *University of Connecticut*. Pertence ao conselho editorial da IEEE *Software* e do *The Cutter TI Jornal*. Autor e consultor na área de Engenharia de Software. Trabalha na área de engenharia de software a mais de 30 anos, atuando como administrador, professor e consultor. É presidente da R.S. Pressman & Associates, Inc., empresa de consultoria especializada em práticas de Engenharia de Software. Saber mais. Em < <http://www.cutter.com/meet-our-experts/rpbio.html> >

Stephen Schach é professor emérito do curso de Ciência da Computação e Engenharia de Computação do Departamento de Engenharia Elétrica e Ciência da Computação da Universidade de Vanderbilt. Possui mestrado em Física pelo *Weizmann Institute of Science* e doutorado em Matemática Aplicada pela *University of Cape Town*. Saber mais. Disponível em < <http://www.vuse.vanderbilt.edu/~srs/> >

Tim O'Reilly, nasceu em 1954 na Irlanda, é o fundador da O'Reilly Media e estudioso e fomentador de software livre, para muitos é o criador da expressão Web 2.0. É Bacharel em filologia clássica pela Harvard University em 1975. Saber mais. Disponível em < http://pt.wikipedia.org/wiki/Tim_O'Reilly >

Tom DeMarco, nascido em 1940 na Pensilvânia, é autor, professor e palestrante em engenharia de software e é reconhecido como um dos criadores da Análise Estruturada nos anos 1980. É Bacharel em Engenharia Elétrica pela Universidade de *Cornell University*, Mestre pela *Columbia University* e Doutor pela University of Paris at the Sorbonne. Saber mais. Disponível em < http://en.wikipedia.org/wiki/Tom_DeMarco >

8 Artefatos de Software

Avaliação da Organização de Desenvolvimento. Em < <http://goo.gl/n4Aqyg> > Acesso:Jan/15.

Avaliação da Organização-alvo. Em < <http://goo.gl/n5bKph> >

Avaliação de Iteração. Em < <http://goo.gl/YIRRBy> >

Avaliação de Status. Em < <http://goo.gl/546iYI> >

Caso de Desenvolvimento. Em < <http://goo.gl/xT7kXd> >

Caso de Negócios. Em < <http://goo.gl/9euZmQ> >

Documento de Arquitetura de Negócios. Em < <http://goo.gl/9AuSQa> >

Documento de Arquitetura de Software. Em < <http://goo.gl/BsuHyq> >

Especificação de Caso de Uso de Negócios Nome do Caso de Uso de Negócios. Disp.:< <http://goo.gl/XklXt> >

Especificação de Caso de Uso Nome do Caso de Uso. Em < <http://goo.gl/8OsCF8> >

Especificação de Realização de Casos de Uso Nome do Caso de Uso de Negócios. Disp.:< <http://goo.gl/PoKB2f> >

Especificação de Realização de Casos de Uso Nome do Caso de Uso. Em < <http://goo.gl/NHNreG> >

Especificação dos Requisitos de Software Para Subsistema ou Recurso. Em < <http://goo.gl/mnOYe5> >

Especificação dos Requisitos de Software. Em < <http://goo.gl/dfnJMy> >

Especificação Suplementar de Negócios. Em < <http://goo.gl/G52y77> >

Especificação Suplementar. Em < <http://goo.gl/aGdgmj> >

Glossário de Negócio. Em < <http://goo.gl/SRgCx8> >

Glossário de Negócios. Em < <http://goo.gl/yjDhRQ> >

Glossário. Em < <http://goo.gl/UVhQ2H> >

Lista de Materiais. Em < <http://goo.gl/A0sTGO> >

Lista de Riscos. Em < <http://goo.gl/Vyy1js> >

Modelo de Negócio. Em < <http://goo.gl/O3O6Ya> >

Notas de Release. Em < <http://goo.gl/e0FC0h> >

Plano de Aceitação do Produto. Em < <http://goo.gl/RH4JDj> >

Plano de Desenvolvimento de Software (Projeto Pequeno). Em < <http://goo.gl/J9qsvB> >

Plano de Desenvolvimento de Software. Em < <http://goo.gl/5KRFCJ> >

Plano de Garantia de Qualidade. Em < <http://goo.gl/D1M0Gp> >

Plano de Gerenciamento de Configuração. Em < <http://goo.gl/almYbn> >

Plano de Gerenciamento de Requisitos. Em < <http://goo.gl/1iwQYG> >

Plano de Gerenciamento de Riscos. Em < <http://goo.gl/kvwGyW> >

Plano de Implantação. Em < <http://goo.gl/zK4sDV> >
Plano de Integração do Build. Em < <http://goo.gl/J5eP3T> >
Plano de Iteração. Em < <http://goo.gl/1f4yLQ> >
Plano de Métricas. Em < <http://goo.gl/64uo2r> >
Plano de Negócio. Em < <http://goo.gl/Jfw2VK> >
Plano de Requisitos. Em < <http://goo.gl/sLM5Lu> >
Plano de Resolução de Problemas. Em < <http://goo.gl/pwvd6l> >
Plano de Teste. Em < <http://goo.gl/XM0cNO> >
Proposta Comercial. Em < <http://goo.gl/Dl17e4> >
Regras de Negócios. Em < <http://goo.gl/Zx5CWY> >
Requisitos de Software. Em < <http://goo.gl/Bi5piJ> >
Solicitações dos Principais Envolvidos. Em < <http://goo.gl/XGUuGR> >
Sumário de Avaliação de Testes. Em < <http://goo.gl/ZVLVb6> >
Termo de Abertura do Projeto. Em < <http://goo.gl/CYnUmT> >
Visão. Em < <http://goo.gl/liXVDn> >
Visão (Projeto Pequeno). Em < <http://goo.gl/x8YMS7> >
Visão do Negócio. Em < <http://goo.gl/wUKpZc> >

9 Ferramentas

Astah Change-Vision. Em < <http://astah.change-vision.com/en/index.html> >

Astah Community. Em < <http://astah.change-vision.com/en/product/astah-community.html> >

Astah Profissional. Em < <http://astah.change-vision.com/en/product/astah-professional.html> >

Astah Share. Em < <http://astah.change-vision.com/en/product/astah-share.html> >

Astah UML. Em < <http://astah.change-vision.com/en/product/astah-uml.html> >

Astah. Em < <http://astah.change-vision.com/en/index.html> >

Astah: modelagem UML. Em < <http://astah.change-vision.com/en/index.html> >

Auger. Em < <http://www.evoluma.com/auger/index.html> >

Bluej. Em < <http://www.bluej.org/> >

CASE tools. Em < <http://case-tools.org/uml.html> >

Case-Tools. Em < <http://case-tools.org/uml.html> >

DrJava. Em < <http://drjava.org/> >

Easyecclipse. Em < <http://sourceforge.net/projects/easyecclipse/files/> >

Eclipse Java Galileo SR2 win32 com UML2.2 de may2010. Em < <http://goo.gl/xOKgSF> >

Eclipse. Em < <http://www.eclipse.org/> >

Green UML. Em < <http://green.sourceforge.net/index.html> >

JBuilder. Em < <https://downloads.embarcadero.com/free/jbuilder> >

JCreator. Em < <http://www.jcreator.com/> >

JEdit. Em < <http://www.jedit.org/> >

JFLAP. Em < <http://www.jflap.org/> >

Jude Community. Em < <http://goo.gl/r2xvL2> >

Makeware. Em < <http://makeware.ueuo.com/inicio.php> >

Manual do Astah. Em < <http://goo.gl/oxXADk> >

MyEclipse. Em < <https://www.genuitec.com/products/myeclipse/download/> >

Netbeans. Em < <https://netbeans.org/downloads/> >

Pacote de componentes para Classes Visuais. Em < <http://goo.gl/zD9DuJ> >

Plugin de modelagem UML no NetBeans 6.9. Em < <http://goo.gl/cKByrP> >

Plugin diversos para NetBeans 6.9. Em < <http://dlc.sun.com.edgesuite.net/netbeans/6.9/rc/zip.html> >

StarUML: modelagem UML. Em < <http://staruml.sourceforge.net/en/> >

StylusStudio. Em < http://www.stylusstudio.com/java_ide.html >

Visual Swing. Em < <http://code.google.com/p/visualswing4eclipse/> >

10 Apresentações e Vídeos

Análise. < <http://www.slideshare.net/alvarofpinheiro/medida-de-esforo-de-software-com-anlise-de-ponto-de-funco> >

Compiladores. Em < <http://vimeo.com/112860457> >

Instalação do Eclipse Galileo (SR2). Em < <https://www.youtube.com/watch?v=95IkXJQXr0Y> >

Instalação do Eclipse Indigo Modeling. Em < <https://www.youtube.com/watch?v=95IkXJQXr0Y> >

Java Completo - Programação Orientada a Objetos com Java. Em < <http://goo.gl/A3qO5w> >

Java Parte 01 - Apresentação do Java. Em < <http://goo.gl/e82UPz> >

Java Parte 02 - Introdução ao Java. Em < <http://goo.gl/vTJjUC> >

Java Parte 03 - Array e Comandos do Java. Em < <http://goo.gl/Yed6UY> >

Java Parte 04 - Orientação a Objetos com Java. Em < <http://goo.gl/t4tzb5> >

Java Parte 05 - Recursos do Java. Em < <http://goo.gl/CMG48U> >

Java Parte 06 - Tratamento de Exceções em Java. Em < <http://goo.gl/Yb8Q5t> >

Java Parte 07 - Exemplo de Codificação em Java. Em < <http://goo.gl/A9msHk> >

Java Parte 08 - Entrada e Saída de Dados com Java. Em < <http://goo.gl/i9Hc3q> >

Java Parte 09 - MVC e JDBC com Java. Em < <http://goo.gl/oP36SO> >

Java Parte 10 - Exemplo de Aplicação em Java com POO, MVC e JDBC. Em < <http://goo.gl/A5ryXo> >

Java Parte 11 - Conceitos Gerais de Java. Em < <http://goo.gl/BfstSF> >

Java Parte 12 - POO com Java. Em < <http://goo.gl/zd56Sq> >

Java Parte 13 - Exemplos. Em < <http://goo.gl/cTHpFB> >

Padrões. Em < <http://www.slideshare.net/alvarofpinheiro/fundamentos-de-padres-de-projeto-de-software> >

Trabalhando com Java e Banco. Em < https://www.youtube.com/view_play_list?p=52761782814BF9F2 >

UML. Em < <https://pt.slideshare.net/alvarofpinheiro/alvarofpinheirouml> >

Usando Java com o MySQL. Em < <https://www.youtube.com/watch?v=U6uUUy4r7t0> >

11 Sugestões de Leitura

Análise e Projeto com UML. Em < <http://goo.gl/Q5Hp7q> >
Aprenda em 21 dias Java 2. Em < <http://goo.gl/FcxqaF> >
Começando a Programar em Java para Leigos. Em < <http://goo.gl/YVOCJL> >
Compiladores. Em < <http://goo.gl/KRL19c> >
Conceitos de computação com Java. Em < <http://goo.gl/fCsK0V> >
Core Java 2: Fundamentals. Em < <http://goo.gl/sl2EOZ> >
Desenvolvendo Aplicações com UML 2.2. Em < <http://goo.gl/vVW6lL> >
Engenharia de Software e Sistemas de Informação. Em < <http://goo.gl/3m2OIh> >
Engenharia de Software. Em < <http://goo.gl/K55qXq> >
Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos. Disp.: < <http://goo.gl/I5G7Y2> >
Exercitando Modelagem com UML. Em < <http://goo.gl/r7toX5> >
GAMMA, Erich; et al. Padrões de Projeto. Em < <http://c2.com/cgi/wiki?DesignPatternsBook> >
Gerenciando Projetos de Desenv. Software com PMI, RUP e UML. Disp.: < <http://goo.gl/GcXUe4> >
Guia prático de engenharia de software. Em < <http://goo.gl/ppe5ff> >
Padrões e projetos orientados a objetos. Em < <http://goo.gl/oGf1Y2> >
Programação de software em Java. Em < <http://goo.gl/7oiuPY> >
Programação Orientada a Objetos com Java. Em < <http://goo.gl/w69GX6> >
UML Essencial. Em < <http://goo.gl/cdLkbT> >
UML Guia do Usuário. Em < <http://goo.gl/av5BTh> >
Utilizando UML e padrões. Em < <http://goo.gl/1YPJ0I> >

Livros da série Fundamentos da Engenharia de Software



Fundament
os da
Engenhari
a de
Software:
Conceitos
Básicos é
uma coletânea

de disciplinas que integradas servem para fundamentar o entendimento da construção de projetos de software com qualidade, isto é, baseado em processos maduros e reconhecidos pela comunidade tecnológica. O objetivo deste livro é fornecer ao leitor as bases necessárias para o desenvolvimento de aplicações sejam Desktop, Web ou Mobile. Iniciando a leitura na Teoria da Computação, passando por Processos, Linguagens, Bancos de Dados e finalizando com Sistemas de Informação e Colaboração. Este livro pode ser lido capítulo a capítulo ou somente a disciplina desejada, pois sua elaboração consiste na compilação das disciplinas fundamentais da Engenharia de Software que são independentes, mas ao mesmo tempo se integram objetivando o desenvolvimento de aplicações.



Introdução
à Banco de
Dados.
Neste são
abordados
os
conceitos
básicos de
bancos de
dados e

seus sistemas gerenciadores, mas com o foco na arquitetura relacional, porque ainda hoje o mercado faz uso em larga escala desses bancos de dados, mesmo que o paradigma predominante seja o orientado a objetos e que, já existam a um bom tempo bancos orientados a objeto, até mesmo os bancos objetos-relacionais que são um híbrido entre essas duas arquiteturas, o que predomina ainda é o relacional, assim, este material é focado na linguagem de consulta estruturada para os SGBD-Rs do mercado, com foco na comparação de cinco dos mais utilizados bancos relacionais, os quais são: Oracle, SQLServer, MySQL, SQLBase e Interbase.



Este livro é
sobre
processos
de
desenvolvi
mento de
software,
evidencian
do a
necessidade

e de qualidade na construção de sistemas, conceituando a diferença entre desenvolvimento Adhoc e com processo. Para isso é realizado a introdução à engenharia de requisitos abordando as técnicas para a elicitação de requisitos que forneçam subsídios necessários para uma construção de software com maior qualidade, enfatizando a necessidade de se aplicar na construção de qualquer sistema as técnicas de análise e modelagem, evidenciando o uso da linguagem da Linguagem de Modelagem Unificada (UML) para diagramar um projeto de software, explicando a necessidade do uso de modelos na construção, entrando com detalhes na análise orientada a objetos, com o objetivo de explorar os seus conceitos de requisitos e modelagem integrados. Este material é finalizado com a introdução à medidas de esforço de desenvolvimento, técnica necessária para responder as perguntas básicas de qualquer desenvolvimento: Qual o prazo e custo? E para



Este livro aborda os sistemas que são classificados como informação, a exemplo, sistemas

de apoio à decisão, sistemas estratégicos, sistemas gerenciais e sistemas transacionais. A produção deste material que compõe o volume 4 da coleção Fundamentos da Engenharia de Software é resultado da compilação das aulas produzidas nas disciplinas que compõem os capítulos deste livro.



A motivação deste livro é exemplificar os conceitos de Padrões de Projetos

utilizando a linguagem de programação Java, sendo a construção uma compilação das aulas produzidas com o intuito de facilitar o entendimento do assunto abordando os seguintes temas: Paradigma Orientado a Objetos que introduz o leitor nos conceitos do POO; Linguagem de Modelagem Unificada para apresentar a simbologia UML dos conceitos de POO; Linguagem de Programação Java apresentando essa poderosa linguagem de programação orientada a objetos para exemplificar os padrões de projeto; e Padrões de Projetos que neste livro aborda os mais referenciados nas academias, sendo eles o GRASP e GoF.



Este livro é o resultado do uso da ferramenta MS Project da Microsoft utilizada na aplicação dos

conceitos de gestão de projetos do PMBOK com as premissas da engenharia de testes para aquisição de qualidade nos produtos de software.



Este livro aborda basicamente os conceitos básicos de programação como autômatos, tipos de



Este livro introduz nas tecnologias Web abordando os conceitos básicos para

linguagens, princípios dos compiladores, paradigmas de desenvolvimento e lógica de programação.

desenvolvimento para Internet com a apresentação da plataforma Dot Net e exibindo dicas de codificação para a linguagem de marcação ASPX, para a linguagem de script mais utilizada pelos navegadores o JavaScript com exemplos de CSS e principalmente dicas de código para a linguagem de programação CSharp e de banco de dados SQL com foco no SQLServer.