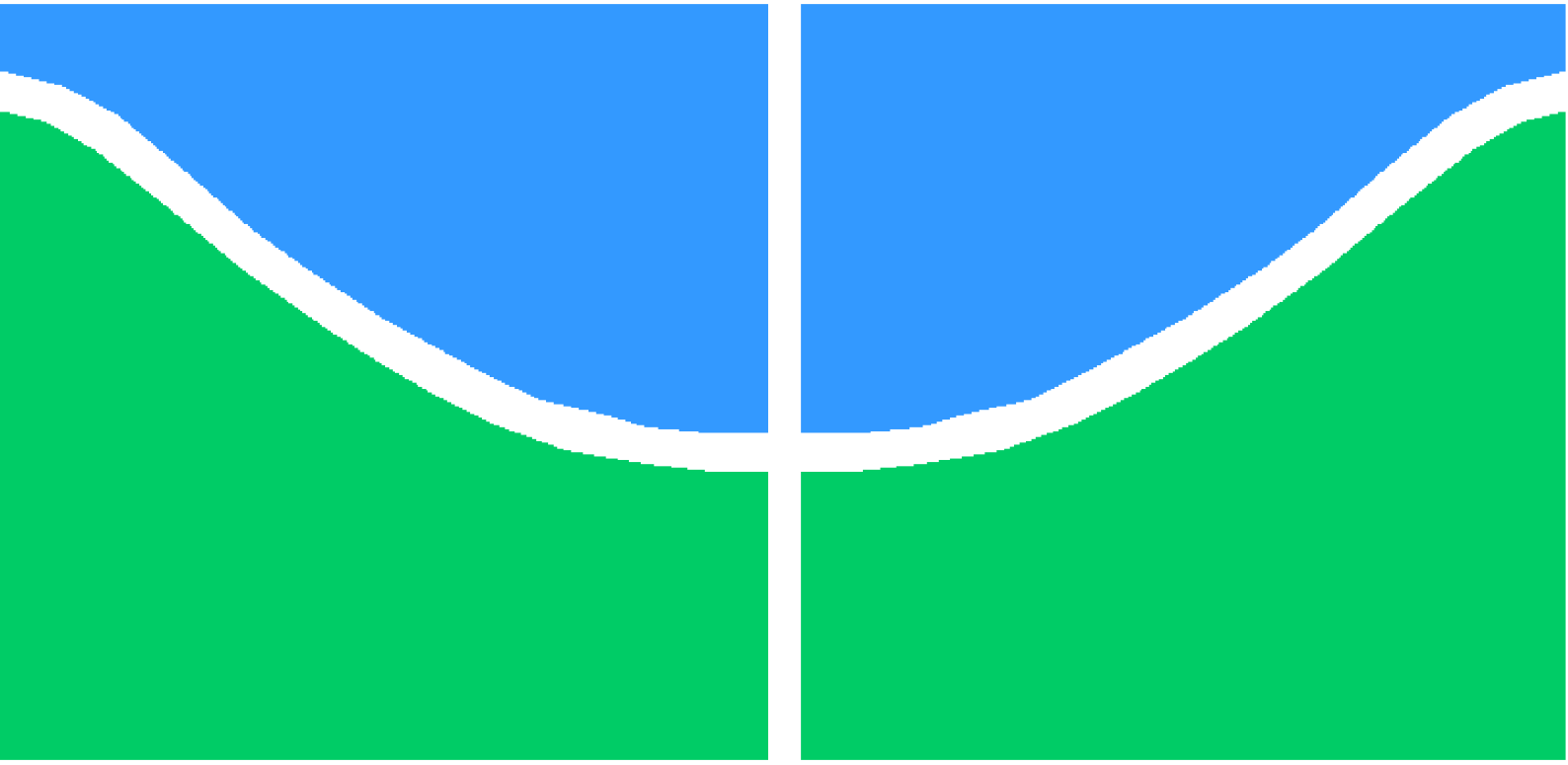

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Ferramenta Multiparadigma para o Mercado de Moedas apoiada por Métodos Matemáticos

Autor: Cleiton da Silva Gomes, Vanessa Barbosa Martins

Orientador: Drª. Milene Serrano

Brasília, DF

2014

Cleiton da Silva Gomes, Vanessa Barbosa Martins

Ferramenta Multiparadigma para o Mercado de Moedas apoiada por Métodos Matemáticos

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Dr^{a}. Milene Serrano

Coorientador: Dr. Ricardo Matos Chaim

Brasília, DF

2014

Cleiton da Silva Gomes, Vanessa Barbosa Martins

Ferramenta Multiparadigma para o Mercado de Moedas apoiada por Métodos Matemáticos/ Cleiton da Silva Gomes, Vanessa Barbosa Martins. – Brasília, DF, 2014-

55 p. : il. (algumas color.) ; 30 cm.

Orientador: Dr^{a}. Milene Serrano

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2014.

1. Forex. 2. Paradigmas de Programação. I. Dr^{a}. Milene Serrano. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Ferramenta Multiparadigma para o Mercado de Moedas apoiada por Métodos Matemáticos

CDU 02:141:005.6

Cleiton da Silva Gomes, Vanessa Barbosa Martins

Ferramenta Multiparadigma para o Mercado de Moedas apoiada por Métodos Matemáticos

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Trabalho aprovado. Brasília, DF, 01 de junho de 2013:

Drª. Milene Serrano
Orientador

Dr. Ricardo Matos Chaim
Coorientador

Titulação e Nome do Professor Convidado 01
Convidado 1

Titulação e Nome do Professor Convidado 02
Convidado 2

Brasília, DF
2014

Dedicamos esse trabalho aos que fazem ao invés de apenas reclamar.
Olhar pelo retrovisor e dizer o outro está errado é muito fácil. Dedicamos esse trabalho à aqueles que aprendem errando

Agradecimentos

Eu, Cleiton Gomes, agradeço ao meu pai que estudou até a 7ª série do ensino fundamental, mas sempre incentivou seus filhos a estudarem mesmo diante de tantas dificuldades e hoje seus três filhos estudam na UnB. Agradeço também aos meus irmãos por toda a experiência de vida que obtivemos juntos em várias aventuras de infância/adolescência. Por fim, agradeço também a Vanessa Barbosa por todo esforço e dedicação que teve ao longo deste TCC.

Eu, Vanessa Barbosa, agradeço à todos de que alguma maneira torceram por mim ao longo destes 5 anos, meus amigos: Álex, André Cruz, André Mateus, Hebert, Jefferson, Luanna, Mônica, Ramon,Sarah. Agradeço também aos meus pais e ao Cleiton Gomes que ficaram ao meu lado nos momentos mais críticos. Por fim agradeço aquele que meu deu força para chegar até aqui, Deus.

Agradecemos aos nossos orientadores Ricardo Chaim e Milene Serrano por todas as diversas reuniões construtivas, por todas as dicas maravilhosas, por todas as revisões de alto nível, por todo carinho e por todo o capricho que tiveram em nos orientar. Agradecemos também aos professores Maurício Serrano, Fabiana Freitas e Wander Cleber por todas as observações pertinentes e construtivas que fizeram ao longo do trabalho. Qualquer possível erro de semântica ou sintaxe desse TCC são de nossa inteira responsabilidade, mas foi uma honra ter cinco professores doutores de alto nível auxiliando no nosso trabalho.

"Sei que o meu trabalho é uma gota no oceano, mas sem ele,
o oceano seria menor."
(Madre Teresa de Calcutá)

Resumo

Faze-lo ao fim de toda a escrita.

Palavras-chaves: FOREX, Paradigmas de Programação, Qualidade de Software, Métodos Númericos.

Abstract

This is the english abstract.

Key-words: latex. abntex. text editoration.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

Fig.	Area of the i^{th} component
456	Isto é um número
123	Isto é outro número
lauro cesar	este é o meu nome

Lista de símbolos

Γ	Letra grega Gama
Λ	Lambda
ζ	Letra grega minúscula zeta
$\in$	Pertence

Sumário

1 Introdução

Este capítulo aborda o contexto, o problema de pesquisa, a justificativa, os objetivos e como o trabalho está organizado.

1.1 Contextualização

O Mercado de Moedas é constituído por transações entre as corretoras que operam no mesmo e são negociados, diariamente, contratos representando volume total entre 1 e 3 trilhões de dólares. É possível operar nesse mercado de forma manual ou automatizada (CVM, 2009).

No contexto de operações automatizadas, no Mercado de Moedas estão inseridos os produtos de software conhecidos como Experts, que processam métodos numéricos para gerar critérios de entrada e saída para comprar ou vender nesse mercado.

1.2 Problema de Pesquisa

Este TCC procurará responder a seguinte questão: é possível desenvolver uma ferramenta multiparadigma que substitua os Experts convencionais do Mercado de Moedas?

1.3 Justificativa

Operar no Mercado de Moedas de forma manual é muito arriscado e, portanto, não recomendado, uma vez que esse mercado é não previsível, o que pode provocar a perda do capital de um investidor em apenas alguns minutos. Em diversas situações, o mercado varia as cotações em apenas um minuto, sendo que a mesma variação pode ser feita durante horas. Para contornar esse problema, a plataforma MetaTrader[1] oferece as linguagens MQL4 (Paradigma Estruturado) e MQL5 (Paradigma Orientado a Objetos) para construir Experts que operem de forma automatizada.

A plataforma MetaTrader não oferece suporte de ferramentas de teste unitário para as linguagens MQL4 e MQL5. Após implementar um Expert, não é possível implementar testes de unidade para verificar se as instruções programadas estão de acordo com o esperado. A única forma de verificar se o Expert está seguindo as estratégias programadas é usar uma conta real ou demo na plataforma e operar durante um período específico de tempo.

[1] *http://www.metatrader4.com/*

Não foi possível encontrar na literatura investigada até o momento ferramentas que realizem a análise estática de código fonte em MQL4 e MQL5. Portanto, torna-se difícil obter uma análise de critérios de aceitabilidade (ou orientada a métricas) no nível de código fonte dos Experts programados nessas linguagens.

Adicionalmente, o código da plataforma MetaTrader é fechado. Dessa forma, não é possível a colaboração da comunidade de desenvolvedores no que tange a evolução das funcionalidades da ferramenta anteriormente.

Diante das preocupações acordadas até o momento, acredita-se que o desenvolvimento de uma ferramenta de código aberto para investimento no Mercado de Moedas, orientada a modelos conceituais de diferentes Paradigmas de Programação bem como às boas práticas da Engenharia de Software como um todo, irá conferir ao investidor maior segurança e conforto em suas operações. Portanto, a ferramenta proposta será implementada em diferentes linguagens de programação, padrões de projeto adequados, testes unitários orientados à uma abordagem multiparadigmas e análise qualitativa de código fonte. Um Expert (implementado em linguagem MQL4 ou MQL5) ou um conjunto de Experts serão substituídos pela ferramenta. Nesse último caso, a ferramenta terá a propriedade de controlar e/ou monitorar um ou mais Experts.

1.4 Objetivos

1.4.1 Objetivo Geral

Desenvolver uma ferramenta multiparadigma para operar de forma semi-automatizada no Mercado de Moedas.

1.4.2 Objetivos Específicos

Considerando o Mercado de Moedas e os Paradigmas de Programação Estruturado, Orientado a Objetos, Funcional, Lógico e Multiagentes, são objetivos específicos desse TCC:

1. Selecionar Métodos Matemáticos para serem implementados na ferramenta MVC através de um protocolo de experimentação;

2. Caracterizar as implementações dos códigos da ferramenta MVC através dos Paradigmas de Programação;

3. Comparar resultados financeiros obtidos pelo código no Paradigma Estruturado com códigos produzidos nos demais Paradigmas;

4. Realizar análise estática do código fonte dos produtos de software a partir de métricas de qualidade previamente definidas;

5. Apurar a cobertura de código por meio de ferramentas que implementam testes unitários nos Paradigmas Estruturado (linguagem C), Multiagentes (linguagem Java), Lógico (linguagem Prolog) e Funcional (linguagem Haskell);

6. Desenvolver testes de integrações e funcionais para a ferramenta MVC.

1.5 Organização do Trabalho

No capítulo 2, é apresentado o Referencial Teórico quanto ao Contexto Financeiro, Métodos Numéricos e aos Paradigmas de Programação. No Contexto Financeiro são tratados os atributos atrelados ao Mercado de Moedas como alavancagem, suporte e resistência. Em Métodos Numéricos, é realizada uma descrição dos métodos de Fibonacci, Correlação de Pearson e Mínimos Quadrados. Em Paradigmas de Programação são descritos os paradigmas: Estruturado, Orientado a Objetos, Lógico, Funcional e Multiagentes. Em Testes de Software, são evidenciados quais os tipos de testes serão utilizados neste TCC e também quais linguagens terão testes. Por fim, em Qualidade de Software são externalizadas as métricas de qualidade de código fonte, critérios para interpretação das métricas e ferramentas de análise estática.

2 Referencial teórico

O referencial teórico revela o momento de levantar o embasamento teórico sobre o tema de pesquisa. No contexto desse TCC, faz-se necessário, dentros outros aspectos pesquisar sobre os entendimentos existentes do problema de pesquisa e analisar quais mecanismos devem ser adotados para se propor uma solução (BELCHIOR, 2012).

O referencial teórico deste TCC irá levantar o embasamento teórico sobre Contexto Financeiro, Métodos Matemáticos, Paradigmas de Programação e Testes estáticos/dinâmicos para que seja possível propor uma solução para o problema de pesquisa.

2.1 Contexto Financeiro

Esta seção irá tratar os atributos aliados ao contexto financeiro como Alavancagem, Suporte e Resistência. Esses atributos são insumos para que se possa compreender melhor a dinâmica das estratégias para negociação no Mercado de Moedas.

2.1.1 Mercado de Moedas

Mercado de Moedas ou FOREX (abreviatura de Foreign Exchange) é um mercado interbancário onde as várias moedas do mundo são negociadas. O FOREX foi criado em 1971, quando a negociação internacional transitou de taxas de câmbio fixas para flutuantes. Com o resultado do seu alto volume de negociações, o Mercado de Moedas tornou-se o principal mercado financeiro do mundo (MARKET, 2011).

A operação no Mercado de Moedas envolve a compra de uma moeda e a simultânea venda de outra. As moedas são negociadas em pares, por exemplo: euro e dólar (EUR-USD). O investidor não compra ou vende euro e dólares fisicamente, mas existe uma relação monetária de troca entre eles. O FOREX é um mercado em que são negociados, portanto, derivativos de moedas. O investidor é remunerado pelas diferenças entre a valorização (se tiver comprado) ou desvalorização (se tiver vendido) destas moedas (CVM, 2009, pág. 3).

O Mercado de Moedas é descentralizado, pois as operações são realizadas por vários participantes do mercado em vários locais. É raro uma moeda manter uma cotação constante em relação a outra moeda. O câmbio entre duas moedas muda constantemente (FXCM, 2011, pág. 5).

O Mercado de Moedas é constituído por transações entre as corretoras que operam no mesmo e são negociados, diariamente, contratos representando volume total entre 1 e

3 trilhões de dólares. As transações são realizadas diretamente entre as partes (investidor e corretora) por telefone e sistemas eletrônicos, desde que tenham conexão à internet. As operações ocorrem 24 horas por dia, durante 5 dias da semana (abrindo às 18h no domingo e fechando às 18h na sexta; horário de Brasília), negociando os principais pares de moedas, ao redor do mundo (CVM, 2009, pág. 4).

2.1.2 Alavancagem

Alavancagem no contexto de mercado,deriva do significado de alavanca na Física, relacionado com a obtenção de um resultado final maior do que ao esforço empregado (DANTAS; MEDEIROS; LUSTOSA, 2006, pág. 3).

> O conceito de alavancagem é similar ao conceito de alavanca comumente empregado em física. Por meio da aplicação de uma força pequena no braço maior da alavanca, é possível mover um peso muito maior no braço menor da alavanca (BRUNI; FAMÁ, 2011, pág. 232).

A Alavancagem possui a propriedade de gerar oportunidades financeiras para empresas que possuem indisponibilidade de recursos internos e/ou próprios (ALBUQUERQUE, 2013, p 13).

No mercado FOREX, o investidor pode negociar contratos de taxas de câmbio e usar a Alavancagem para aumentar suas taxas de lucro. Se o investidor, por exemplo, realizar uma operação de compra apostando 0.01 por ponto e o mercado subir 1000 pontos, ele ganha 10 dólares (0.001x1000). Usando a técnica de Alavancagem, o investidor pode realizar a mesma operação de compra colocando sua operação alavancada a 1.0, realizando o lucro de 1000 dólares (1.0x1000) (EASYFOREX, 2014).

2.1.3 Suporte

Segundo MATSURA (2006, pág. 22), Suporte é o nível de preço no qual a pressão compradora supera a vendedora e interrompe o movimento de baixa. Pode-se identificar o Suporte por uma linha reta horizontal conforme a figura 1.

Suporte é uma região gráfica que após uma queda, os preços param e revertem no sentido contrário. É uma área em que os investidores tendem a comprar (DEBASTINI, 2008, p 97).

Os níveis de Suporte indicam as cotações em que os investidores acreditam que vão subir. À medida em que as cotações se deslocam para a zona de Suporte, os investidores estão mais confiantes para comprar (COLLINS et al., 2012).

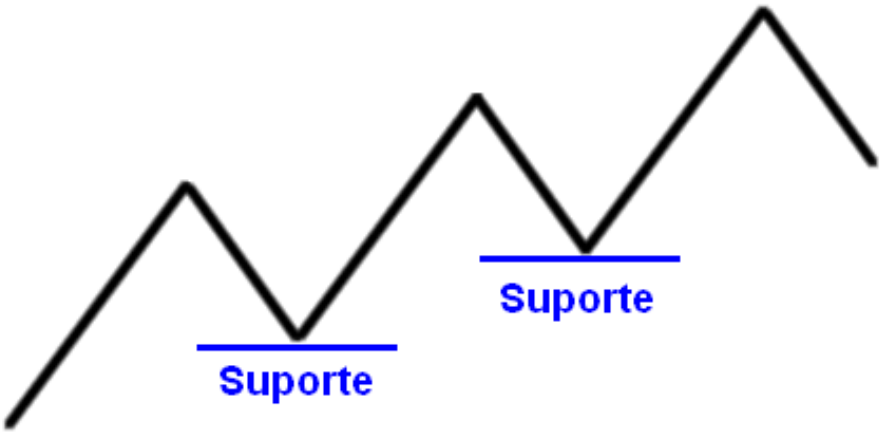

Figura 1 – Reta de Suporte.

Fonte: MATSURA (2006, pág. 22).

2.1.4 Resistência

Resistência é a região do gráfico em que após um movimento de alta, os preços param e revertem no sentido contrário. É um ponto em que os investidores tende a vender para ter o maior lucro possível (DEBASTINI, 2008, pág. 98).

Segundo MATSURA (2006, pág. 23), Resistência representa o nível de preço no qual a pressão vendedora supera a compradora e interrompe o movimento de alta. A Resistência é identificada por uma linha reta horizontal, conforme a figura 2.

A Resistência indica os níveis das cotações em que os investidores acreditam que as mesmas vão descer. À medida que as cotações se deslocam para a zona de Resistência, os investidores estão mais confiantes para vender (COLLINS et al., 2012).

2.2 Métodos Matemáticos

Métodos Matemáticos é qualquer método que se utiliza da matemática para resolver um problema. É possível citar alguns exemplos desses métodos como equações polinomiais, identidades trigonométricas, geometria coordenada, frações parciais, expansões binomiais, entre outros (RILEY; HOBSON; BENCE, 2011).

Métodos Matemáticos são aplicados a área de finanças. Cálculo e álgebra linear são fundamentais para o estudo de matemática financeira e ajuda a compreender a dinâmica de mercado (KONIS, 2014).

Este capítulo irá abordar sobre os Métodos Matemáticos de Mínimos Quadrados,

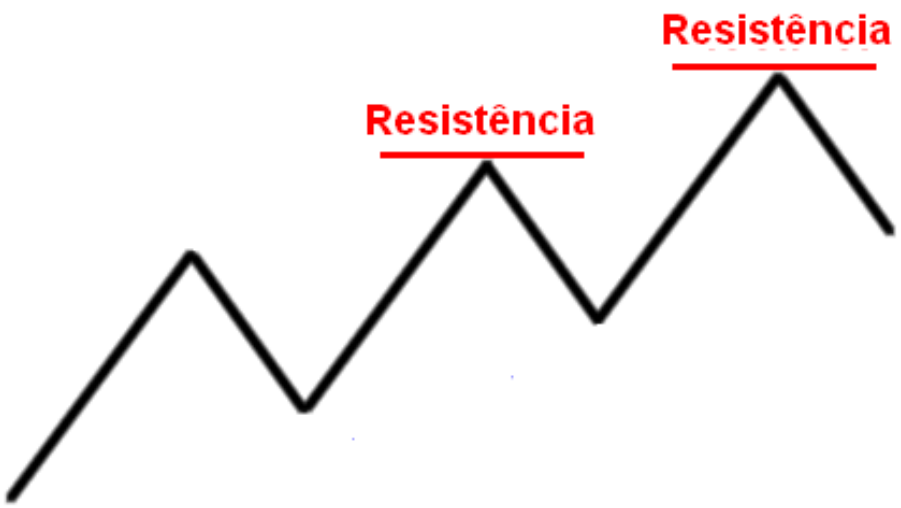

Figura 2 – Reta de Resistência.

Fonte: MATSURA (2006, pág. 23)

Fibonacci e Correlação Linear de Pearson.

2.2.1 Método de Mínimos Quadrados

O método de Mínimos Quadrados determina o valor mais provável de quantidades não conhecidas em que a soma dos quadrados das diferenças entre valores observados e computados é mínimo (INÁCIO, 2010, pág. 72).

Usa-se o método de Mínimos Quadrados para determinar a melhor linha de ajuste que passa mais perto de todos os dados coletados, no intuito de obter a melhor linha de ajuste, de forma que minimize as distâncias entre cada ponto de consumo (DIAS, 1985, pág. 46).

A aplicação do método de Mínimos Quadrados visa deduzir a melhor estimativa de mensurações de n medições idênticas (em condições de "repetitividade") e não idênticas (em condições de "reprodutividade"). Dessa forma o peso estatístico de um resultado é definido (VUOLO, 1996, pág. 149).

O desvio vertical do ponto:

$$(x_i, y_i) \tag{2.1}$$

da reta:

$$Y = B0 = B1 * X_i \tag{2.2}$$

é a altura do ponto menos altura da reta. A soma dos desvios quadrados verticais dos pontos:

$$(x_1, y_1)...(x_i, y_i) \tag{2.3}$$

à reta é portanto:

$$f(B0, B1) = \sum y_i - (B0 + B1 * X_i)^2 \tag{2.4}$$

$$0 <= i > \infty \tag{2.5}$$

As estimativas pontuais de C0 e C1, representadas por K0 e K1 e denominadas estimativa de Mínimos Quadrados, são aquelas que minimizam f(B0, B1). Em suma, para qualquer B0 e B1, K0 e K1 são tais que:

$$f(K0, K1) <= f(B0, B1) \tag{2.6}$$

A reta de Regressão Estimativa ou de Mínimos Quadrados é, por conseguinte, a reta cuja equação é :

$$Y = K0 + K1X \tag{2.7}$$

Como mostrado na figura 3(DEVORE, 2006, pág. 441).

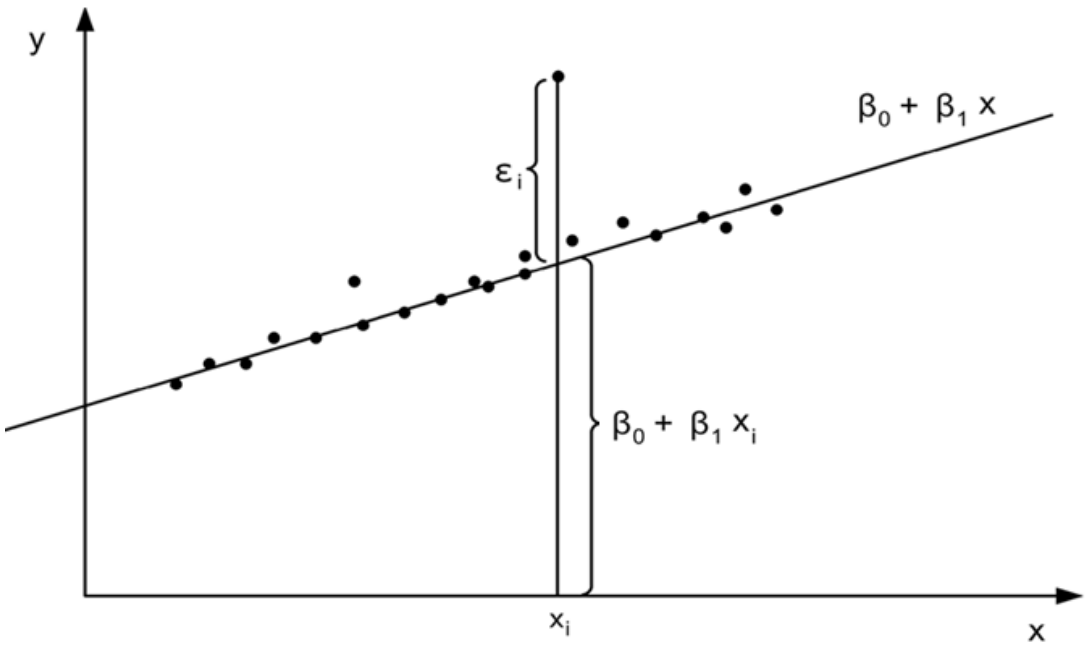

Figura 3 – Equação da reta Mínimos Quadrados.

Fonte: Devore (2006, pág. 443).

2.2.2 Método de Correlação Linear

Em estudos que envolvem duas ou mais variáveis, é comum o interesse em conhecer o relacionamento entre elas, além das estatísticas descritivas normalmente calculadas. A medida que mostra o grau de relacionamento entre as variáveis é chamada de Coeficiente de Correlação ou Correlação Linear ou Correlação Linear de Pearson. A Correlação Linear também é conhecida como medida de associação, interdependência, intercorrelação ou relação entre as variáveis (LIRA, 2004, pág. 62).

O Coeficiente de Correlação Linear de Pearson (r) é uma estatística utilizada para medir força, intensidade ou grau de relação linear entre duas variáveis aleatórias (FERREIRA, 2009, pág. 664).

Segundo (LOPES, 2005, pág. 134), a Correlação Linear indica a relação entre duas variáveis. De modo a interpretar o coeficiente de correlação (r), podem ser utilizados os seguintes critérios para classificar os resultados obtidos:

- De 0 a 0,50: fraca correlação;
- De 0,51 a 0,84: moderada correlação;
- A partir de 0,85: forte correlação.

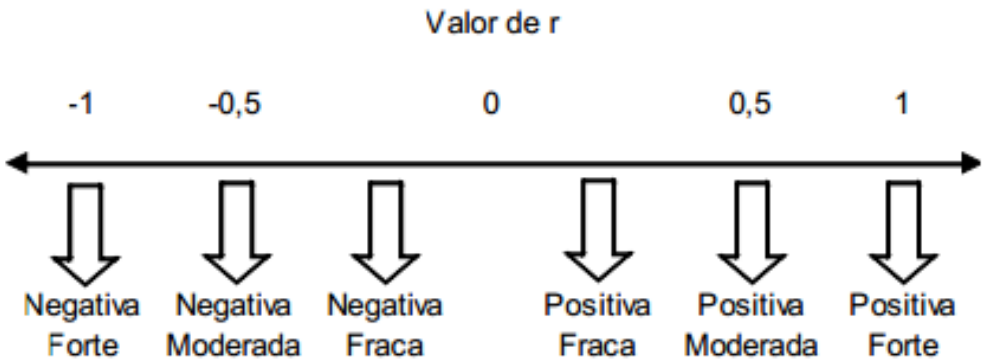

Figura 4 – Classificação da Correlação Linear.

Fonte: Lopes (2005, pág. 134).

Segundo Regra (2010, pág. 47) a Correlação Linear revela o grau de associação entre duas variáveis aleatórias. A dependência de duas variáveis X e Y é dada pelo Coeficiente de Correlação Amostral, conhecido também por coeficiente r-de-Pearson. Designa-se, normalmente, por r e é determinado de acordo com a figura 5.

Segundo VIALI (2009, pág. 8) as propriedades mais importantes do Coeficiente de Correlação são:

1. O intervalo de variação vai de -1 a +1.
2. O coeficiente é uma medida adimensional, isto é, independente das unidades de medida das variáveis X e Y.
3. Quanto mais próximo de +1 for “r”, maior o grau de relacionamento linear positivo entre X e Y, ou seja, se X varia em uma direção, Y variará no mesmo sentido.

$$r = \frac{Cov(X, Y)}{s_X . s_Y}$$

$$Cov(X, Y) = \frac{\sum_{i=1}^{n} XY}{n} - \overline{X}\,\overline{Y} \ ,$$

$$s_X = \sqrt{\frac{\sum_{i=1}^{k} f_i \left(X_i - \overline{X}\right)^2}{n}} \quad e \quad s_Y = \sqrt{\frac{\sum_{i=1}^{k} f_i \left(Y_i - \overline{Y}\right)^2}{n}}$$

Figura 5 – Determinação da Correlação Linear.

Fonte: Regra (2010).

4. Quanto mais próximo de -1 for “r”, maior o grau de relacionamento linear negativo entre X e Y, isto é, se X varia em um sentido, Y variará na direção inversa.

5. Quanto mais próximo de zero estiver “r”, menor será o relacionamento linear entre X e Y. Um valor igual a zero indicará ausência apenas de relacionamento linear.

A análise da Correlação Linear fornece um número, indicando como duas variáveis variam conjuntamente e mede a intensidade e a direção da relação linear ou não-linear entre duas variáveis. Essa análise também é um indicador que atende à necessidade de estabelecer a existência ou não de uma relação entre essas variáveis sem que, para isso, seja preciso o ajuste de uma função matemática. Em suma, o grau de variação conjunta entre X e Y é igual ao de Y e X (LIRA, 2004, pág. 65).

2.2.3 Método de Fibonacci

A sucessão ou sequência de Fibonacci é uma sequência de números naturais, na qual os primeiros dois termos são 0 e 1, e cada termo subsequente corresponde à soma dos dois precedentes. A sequência tem o nome do matemático pisano do século XIII Leonardo de Pisa, conhecido como Leonardo Fibonacci, e os termos são chamados números de Fibonacci. Os números de Fibonacci compõem a seguinte sequência de números inteiros: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, . . . , (GAGLIARDI, 2013, pág. 6).

Devido a sequência de Fibonacci ser recursiva, é possível determinar uma fórmula capaz de encontrar o valor de qualquer número de Fibonacci, Fn, se seu lugar na sequência, n, for conhecido. Esta propriedade garante que para obter todas as soluções da equação recursiva de Fibonacci: Fn+1 = Fn-1 + Fn, para qualquer n > 1 (ALVES., 2012, pág. 12).

Segundo Rocha (2008, pág. 32), a fórmula de Fibonacci, sem uso da recorrência, é dada de acordo com a figura 6. Essa fórmula também é conhecida como fórmula de Binet.

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Figura 6 – Fórmula de Fibonacci sem uso de recorrência.

Fonte: Rocha (2008, pág. 32).

2.3 Paradigmas de Programação

A palavra paradigma significa aquilo que pode ser utilizado como padrão, de forma que é um modelo a ser seguido (**??**). Segundo Normak (2013) Kurt Normark (2013), professor da Universidade de Aalborg na Dinamarca, paradigma de programação é um padrão que serve como uma escola de pensamentos para a programação de computadores.

Uma linguagem de programação multiparadigma é uma linguagem de programação que suporta mais de um paradigma de programação. O objetivo de tais linguagens é permitir que programadores usem a melhor ferramenta para o trabalho, admitindo que nenhum paradigma resolve todos os problemas da maneira mais fácil ou mais eficiente (PAQUET; MOKHOV, 2010, pág. 21). Seguem noções preliminares sobre cada paradigma, os quais serão investigados ao longo desse TCC.

2.3.1 Paradigma Procedural

Programação procedural é um paradigma de programação, derivado da programação estruturada, com base no conceito da chamada de procedimento. Procedimentos, também conhecidos como rotinas, sub-rotinas, métodos ou funções, contêm uma série de passos computacionais a serem realizados. Qualquer procedimento pode ser chamado a qualquer momento durante a execução de um programa, inclusive por outros procedimentos ou a si mesmo (PAQUET; MOKHOV, 2010, pág. 22).

A linguagem de programação procedural fornece ao programador um meio de definir com precisão cada passo na execução de uma tarefa e é muitas vezes uma escolha melhor em situações que envolvem complexidade moderada ou que requerem significativa facilidade de manutenção (PAQUET; MOKHOV, 2010, pág. 22).

As linguagens procedurais foram desenvolvidas em torno da arquitetura de computadores prevalentes na época, chamada de arquitetura von Neumann, criada pelo matemático húngaro John von Neumann (SEBESTA, 2012, pág. 18).

Em um computador de von Neumann, ambos os dados e programas são armazenados na mesma memória. A unidade central de processamento (CPU), que executa as instruções, é separada da memória. Portanto, as instruções e os dados devem ser transmitidos da memória para a CPU. Os resultados das operações na CPU devem ser devolvidos à memória, conforme ilustra a figura 7.

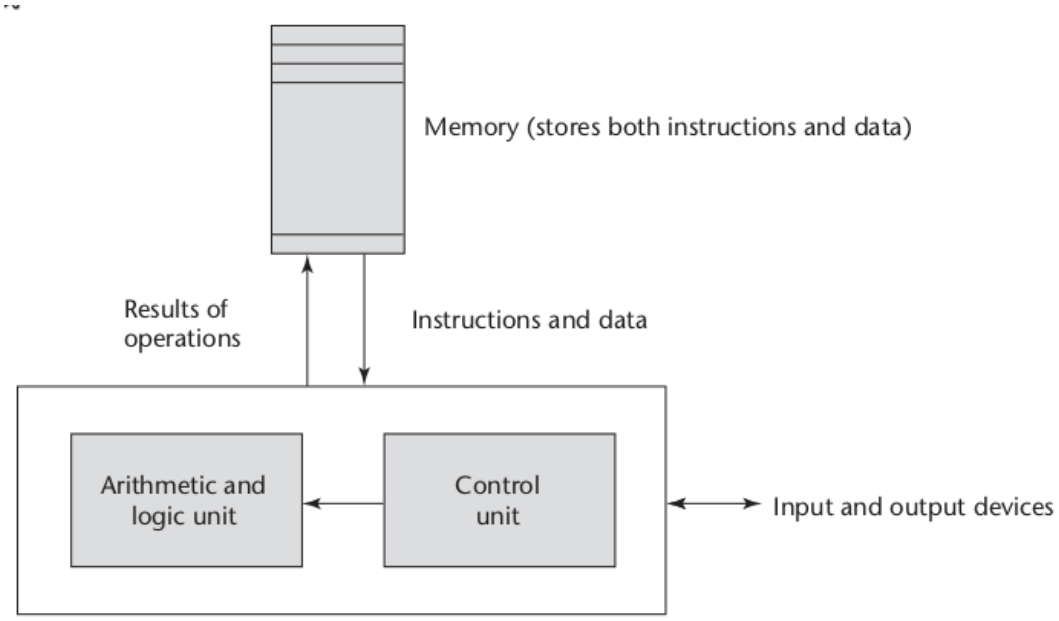

Figura 7 – Arquitetura de von Neumann.

Fonte: SEBESTA (2012, pág. 19).

Devido ao uso da arquitetura de von Neumann, os recursos centrais das linguagens imperativas são as variáveis, as quais modelam as células de memória, as instruções de

atribuição, baseadas na operação de canalização (piping) e na forma interativa de repetição, o método mais eficiente dessa arquitetura. A iteração é rápida nos computadores de von Neumann uma vez que as instruções são armazenadas em células adjacentes da memória. Esta eficiência desencoraja o uso de recursão para repetição, embora a recursão seja frequentemente mais natural (SEBESTA, 2012, pág. 18).

Linguagens imperativas contêm variáveis e valores inteiros, operações aritiméticas básicas, comandos de atribuição, sequenciamentos de comandos baseados em memórias, condições e comandos de ramificação. Essas linguagens suportam determinadas características comuns, que surgiram com a evolução do paradigma, tais como: estruturas de controle, entrada/saídas, manipulação de exceções e erros, abstração procedural, expressões e atribuição, e suporte de biblioteca para estruturas de dados (TUCKER; NOONAN, 2009, pág. 278-279).

Fortran (FORmula TRANslation) foi a primeira linguagem de alto nível a ganhar ampla aceitação, sendo esta uma linguagem imperativa. Projetada para aplicações científicas, essa linguagem conta com notação algébrica, tipos, subprogramas e entrada/saída formatada. Foi implementada em 1956 por John Backus na IBM especificamente para a máquina IBM 704. A execução eficiente foi uma grande preocupação, consequentemente, sua estrutura e comandos têm muito em comum com linguagens de montagem (BROOKSHEAR, 2003, pág. 458).

Outra linguagem de programação é o C Ritchie (1996), um dos criadores da linguagem, ela se tornou uma linguagem dominante na década de 90. Seu sucesso deve-se ao sucesso do Unix, um sistema operacional implementado em C.

> Apesar de alguns aspectos misteriosos para o iniciante e ocasionalmente até mesmo para o adepto, a linguagem C permanece uma simples e pequena linguagem, traduzível com simples e pequenos compiladores. Seus tipos e operações são bem fundamentados naquelas fornecidas por máquinas reais, e para pessoas que usam o OO computador para trabalhar, aprender a linguagem para gerar programas em tempo – e espaço – eficientes não é difícil. Ao mesmo tempo a linguagem é suficientemente abstrata dos detalhes da máquina de modo que a portabilidade de programa pode ser alcançada (RITCHIE, 1996).

2.3.2 Paradigma Orientado a Objetos

Um objeto é a abstração de uma "coisa"(alguém ou algo), e esta abstração é expressa com a ajuda de uma linguagem de programação. A coisa pode ser um objeto real, ou algum conceito mais complicado. Tomando um objeto comum, como um gato, por exemplo, você pode ver que ele tem certas características (cor, nome, peso) e pode executar algumas ações (miar, dormir, esconder, fugir). As características do objeto são chamados de propriedades em Orientação a Objetos (OO) e as ações são chamadas de métodos (STEFANOV, 2008, pág. 13).

O conceito de objeto não surgiu do paradigma orientado a objetos. Pode-se dizer ainda que OO foi uma evolução das práticas já existentes da época. O termo objetos surgiu quase simultaneamente em 1970 em vários ramos da ciência da computação, algumas áreas que influenciaram o paradigma OO foram: sistemas de simulação, sistemas operacionais, abstração de dados e inteligência artificial, como ilustra a figura 8 (CAPRETZ, 2003, pág. 1).

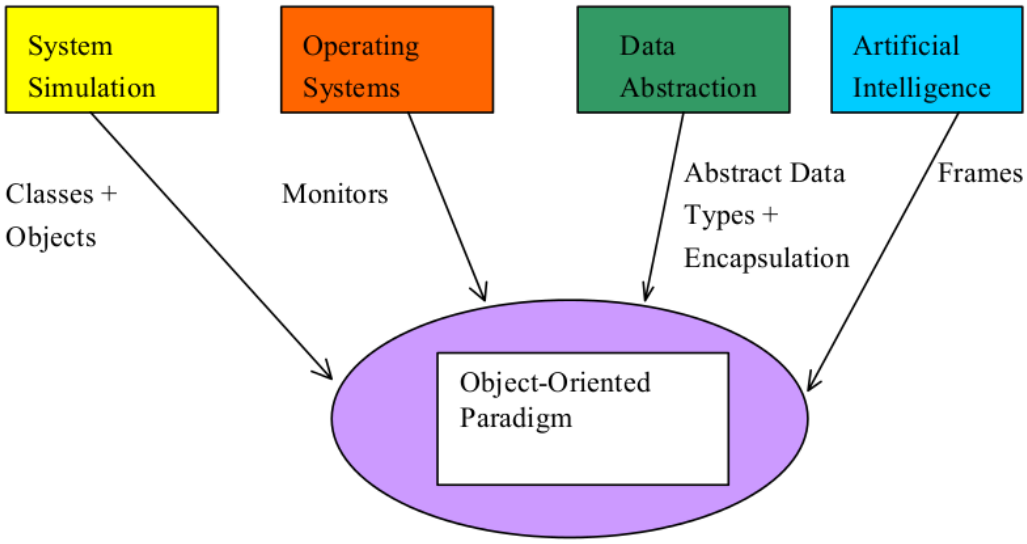

Figura 8 – A essência do Paradigma Orientado a Objetos.

Fonte: CAPRETZ (2003, pág. 1).

A programação orientada a objetos fornece um modelo no qual um programa é uma coleção de objetos que interagem entre si, passando mensagens que transformam seu estado. Neste sentido, a passagem de mensagens permite que os objetos dados se tornem ativos em vez de passivos (TUCKER; NOONAN, 2009, pág. 310).

Classes servem como modelos a partir dos quais objetos podem ser criados. Elas tem precisamente as mesmas variáveise operações de instâncias dos objetos, mas sua interpretação é diferente: onde um objeto representa variáveis reais, variáveis de classe são, em potencial, instanciadas apenas quando um objeto é criado (WEGNER, 1990, pág. 10).

Uma das principais vantagens do uso de orientação a objetos é que o objeto não precisa revelar todos os seus atributos e comportamentos (métodos). Em um bom design OO um objeto só deve revelar as interfaces necessárias para interagir com outros objetos. Os detalhes não pertinentes para a utilização do objeto deve ser ocultado de todos os outros objetos. Por exemplo, um objeto que calcula o quadrado de um número deve fornecer uma interface para obter o resultado. No entanto, as propriedades internas e algoritmos utilizados para calcular o quadrado não têm de ser disponibilizados ao objeto solicitante. O encapsulamento é definido pelo fato de que os objetos envolvem (quase que

como uma cápsula) atributos e métodos, e a ocultação de dados é uma das partes mais importantes do encapsulamento (WEISFELD, 2009, pág. 19).

Uma nova classe (designada por subclasse derivada) pode ser derivada de outra classe (designada por superclasse) por um mecanismo chamado de herança. A classe derivada herda todas as características da classe base: a sua estrutura e comportamento (resposta a mensagens). Além disso, o mecanismo de herança é permitido mesmo sem acesso ao código-fonte da classe base. Herança dá a OO seu benefício chefe sobre outros paradigmas de programação - a reutilização de código relativamente fácil sem a necessidade de alterar o código fonte existente (LEAVENS, 2014).

Tendo em vista o conceito de Herança, é possível entender o Polimorfismo , que etimologicamente significa "muitas formas", sendo a capacidade de tratar um objeto de qualquer subclasse de uma classe base, como se fosse um objeto da própria classe base. A classe base, portanto, tem muitas formas: a própria classe base, e qualquer uma de suas subclasses. Se você precisa escrever um código que lida com uma família de tipos, o código pode ignorar detalhes específicos do tipo e apenas interagir com o tipo base da família. Mesmo que o código esteja estruturado para enviar mensagens para um objeto da classe base, a classe do objeto poderia realmente ser a classe base ou qualquer uma de suas subclasses. Isso torna o código extensível, porque outras subclasses podem ser adicionadas mais tarde para a hierarquias de classes (VENNERS, 1996).

Se tratando de métodos polimórficos pode-se ter sobrecarga e a sobrescrita.Sobrecarga de método é um recurso que permite que uma classe tem dois ou mais métodos com mesmo nome, se suas listas de parâmetros se diferentes com: números de parâmetros passados, tipo de dados dos parâmetros e sequência dos dados passados como parâmetro. Já a sobrescrita é a declaração de um método na subclasse, que já está na classe mãe (SINGH, 2014b).

Os conceitos de sobrecarga e sobrescrita são constantemente confundidos. A sobrecarga acontece em tempo de compilação, enquanto a sobrescrita ocorre em tempo de execução. A sobrecarga feita na mesma classe, enquanto são necessárias classes mães e suas filhas para o uso sobrescrita, métodos privados sobrecarregados, mas eles não podem ser sobrescritos. Isso significa que uma classe pode ter mais de um método privado mesmo nome, mas uma classe filha não pode substituir os métodos privados sua classe mãe (SINGH, 2014a).

Reusabilidade é uma das grandes promessas da tecnologia orientada a objetos. Reutilização de código, o tipo mais comum de reuso, refere-se à reutilização de código-fonte dentro de seções de uma aplicação e potencialmente através de múltiplas aplicações. Em alguns casos, reutilização de código é alcançada compartilhando-se classes, coleções de funções e rotinas comuns. A vantagem do reuso de código é que ela reduz a quantidade real de código que você precisa escrever. Há ainda a reutilização de herança, que refere-se

ao uso de herança em sua aplicação para tirar vantagem de comportamento implementado em classes existentes (AMBLER, 1998).

2.3.3 Paradigma Orientado a Agentes

Um agente é qualquer coisa que pode perceber seu ambiente através de sensores e agir sobre esse ambiente através de atuadores. Um agente humano tem olhos, ouvidos e outros órgãos para sensores e como atuadores possui mãos, pernas, tato, voz, e assim por diante. Um agente robótico pode ter câmeras e localizadores, faixa do infravermelho como sensores e vários motores para atuadores. Um agente de software recebe as teclas digitadas, conteúdo de arquivos e pacotes de rede como entradas sensoriais e age sobre o meio ambiente por meio da exibição na tela, gravação de arquivos e envio de pacotes de rede (RUSSEL; NORVIG, 1995, pág. 34).

A exigência de continuidade e autonomia deriva nosso desejo de que um agente seja capaz de realizar atividades de uma forma flexível e inteligente, que é sensível a alterações no ambiente sem a necessidade de orientação constante humana ou de intervenção. Idealmente, um agente que funciona de forma contínua, num ambiente ao longo de um período de tempo seria capaz de aprender com sua experiência. Além disso, espera-se um agente que habita um ambiente com outros agentes e processos pode ser capaz de se comunicar e cooperar com eles (BRADSHAW, 1997, pág. 8).

O autor WOOLDRIDGE (2010, pág. 42) considera quatro tipos de arquitetura de agentes: baseados em lógica, reativos, em camadas e de crença-desejo-intenção.

> Nas arquiteturas baseadas em lógica os agentes contêm um modelo simbólico do ambiente do agente, explicitamente representado em uma base de conhecimento e a decisão da ação a executar é tomada a através de raciocínio lógico (GIRARDI, 2004, pág. 3).

Arquiteturas reativas deixam o raciocínio abstrato de lado e destinam-se a lidar com comportamentos básicos. Os agentes reagem às mudanças no ambiente como uma forma de resposta ao estímulo, executando as rotinas simples que corresponde a um estímulo específico (SCHUMACHER, 2001, pág. 13).

A arquiteturas em camadas, também conhecido como arquitetura híbrida, híbrida combina componentes da arquitetura baseada em lógica e da reativa. Esta arquitetura propõe um subsistema deliberativo que planeja e toma decisões da maneira proposta pela Inteligência Artificial Simbólica e um reativo capaz de reagir a eventos que ocorrem no ambiente sem se ocupar de raciocínios complexos (COSTA, 2004, pág. 44).

Na arquitetura BDI o esta do agente é representado por três estruturas: suas crenças (beliefs), que são o conhecimento do agente sobre seu ambiente; seus desejos (desires), representam objetivos ou situações que o agente gostaria de realizar ou trazer; por fim

tem-se suas intenções (intentions), são suas ações que têm decidido realizar(GIRARDI, 2004, pág. 3).

Um sistema multiagente (SMA) pode ser caracterizado como um grupo de agentes que atuam em conjunto no sentido de resolver problemas que estão além das suas habilidades individuais. Os agentes realizam interações entre eles de modo cooperativo para atingir uma meta (GIRARDI, 2004, pág. 6).

Para WOOLDRIDGE (2010, pág. 3) o uso de SMA se justifica uma vez que muitas tarefas não podem ser feitas por um único agente, e há ainda tarefas que são feitas de forma mais eficaz quando realizada por vários agentes. Para tal é essencial que o SMA seja capaz de: trabalhar em conjunto para alcançar um objetivo comum, monitorar constantemente o progresso do esforço da equipe como um todo, ajudar um ao outro quando necessário, coordenação das ações individuais de modo que eles não interfiram um com o outro, comunicar sucessos e fracassos, se necessário para a equipe para ter sucesso parcial.

SMA muitas vezes baseia-se em conceitos de outras disciplinas, como a psicologia, a ciência econômica, cognitiva, lingüística, inteligência artificial, etc. Por exemplo, analisar protocolos de interação e ações de comunicação entre os agentes com base na teoria dos atos de fala, vem da filosofia e da lingüística. A abstração da postura intencional foi emprestado da ciência cognitiva para analisar e raciocinar sobre os comportamentos autônomos de agentes. Recentemente, muitas metodologias e modelos de abstrações e conceitos de organização e sociologia foram propostas para modelar, analisar e projetar SMA (ODELL; GIORGINI; MÜLLER, 2005, pág. 1).

Agentes autônomos e SMA representam uma nova forma de analisar, projetar e implementar sistemas de software complexos. A visão orientada a agentes oferece um repertório poderoso de ferramentas, técnicas e metáforas que têm o potencial de melhorar consideravelmente a maneira como as pessoas conceituam e implementam muitos tipos de software. Agentes estão sendo usados em uma ampla variedade de aplicações, desde de pequenos sistemas, tais como filtrados de de e-mail personalizados, a sistemas grandes e complexos de missão crítica, como controle de tráfego aéreo. À primeira vista, pode parecer que tais tipos de sistema tem pouco em comum. E, no entanto este não é o caso: em ambos, a abstração chave usada é a de um agente (JENNINGS; SYCARA; WOOLDRIDGE, 1998).

Segundo JENNINGS e WOOLDRIDGE (1995) os agentes de software tem as seguintes propriedades: autonomia, competência social, reatividade e pró-atividade.

Os agentes mantêm uma descrição do seu próprio estado de processamento e o estado do mundo em torno deles, logo eles são ideais para aplicações de automação. Agentes autônomos são capazes de operar sem entrada ou intervenção do usuário. Podendo ser utilizados em como instalações e automação de processos (AGENTBUILDER, 2009b).

A empresa Acronymics e a Alternative Energy Systems Consultants (AESC) realizaram uma pesquisa afim de criar agentes de softwares capazes de comprar e vender energia eletricidade participando do mercado eletrônico. Cada agente apresentavam um comportamento único e individual determinado pelo seu próprio modelo econômico, esta pesquisa mostrou que os agentes podem ser usados para implementar mercados e leilões eletrônicos, e que um agente pode adotar os objetivos e intenções de seu stakeholder (AGENTBUILDER, 2009a).

2.3.4 Programação Declarativa

Linguagens declarativas permitem ao programador se concentrar na lógica de um algoritmo, diferentemente de linguagens imperativas que requerem do programador se concentrar tanto na lógica quanto no controle de um algoritmo, para tal se dá o nome de atribuição não-destrutiva. Nos programas declaritvos, os valores associados aos nomes de variáveis não podem ser alterados. Assim, a ordem na qual definições ou equações são chamadas, não interessa. Além disso, as definições declarativas não permitem efeitos secundários, isto é, o cálculo de um valor não irá afetar outro valor (COENEN, 1999).

De fato, em algumas situações, a especificação de um problema no formato adequado já constitui a solução para o problema algorítmico. Em outras palavras, a programação declarativa torna possível escrever especificações executáveis. Entretanto, na prática, os programas obtidos desde modo são frequentemente ineficientes, visto que esta abordagem de programação tem associado, ao uso adequado de transformações dos programas (APT, 1996, pág. 2).

2.3.4.1 Paradigma Funcional

O centro da programação funcional é a idéia de uma função. A linguagem de programação funcional dá um modelo simples de programação: dado um valor, o resultado é calculado com base em outros valores, as entradas da função. Devido à fundação simples, uma linguagem funcional dá uma visão mais clara das idéias centrais da computação moderna, incluindo abstração, polimorfismo e sobrecarga (THOMPSON, 1999, pág. 16).

A primeira linguagem de programação funcional foi inventada para oferecer recursos de linguagem para processamento de listas, cuja necessidade surgiu a partir das primeiras aplicações na área da inteligência artificial (TUCKER; NOONAN, 2009, pág. 361).

A programação funcional exige que as funções sefam cidadãos de primeira classe, oque significa que elas são tratadas como quaisquer outros valores e podem ser passadas como argumentos para outras funções ou serem retornadas como um resultado de uma função. Sendo cidadãos de primeira classe também significa que é possível definir e manipular funções dentro de outras funções (HOOGLE, 2013).

Uma linguagem de programação puramente funcional não usa variáveis ou instruções de atribuição. Isso libera o programador de preocupar-se com as células da memória do compilador no qual o programa é executado. Sem variáveis, construções interativas não são possíveis, porque elas são controladas por variáveis. A repetição deve ser feita por meio de recursão, não por meio de laços (SEBESTA, 2012, pág. 555).

Por ser programação declarativa o paradigma funcional não tem efeitos colaterais, uma função não faz nada que não seja retornar seu valor de resultado, com isso fica fácil trazer uma experiência matemática para a programação (PIPONI, 2006).

2.3.4.2 Paradigma Lógico

Na programação lógica, um programa consiste de uma coleção de declarações expressas como fórmulas da lógica simbólica. Existem regras de inferência da lógica que permitem uma nova fórmula derivada das antigas, com a garantia de que se as últimas fórmulas são verdadeiras, então a nova regra também será (SPIVEY, 1996, pág. 2).

Em outros paradigmas como o imperativo, uma pergunta terá sempre uma única resposta. A programação lógica é baseada na noção de que um programa implementa uma relação ao invés de um mapeamento. Desta forma, podemos fazer pedidos como: Dado A e B, determinar se a Relação(A, B) é verdadeira; dado A, encontrar todos os Bs tal que a Relação(A, B) é verdadeira; dado B, encontrar todos os As, tal que a Relação(A, B) é verdadeira; pesquisar os As e Bs para o qual a Relação(A, B) é verdadeira (PAQUET; MOKHOV, 2010, pág. 33).

Para garantir que o programa dará respostas corretas, o programador deve verificar se o programa contém apenas declarações verdadeiras, e em número suficiente para garantir que as soluções a serem derivadas resolvem todos os problemas que são de interesse. O programador também pode garantir que as derivações que a máquina realizará são bastante curtas, de modo que a máquina pode encontrar respostas rapidamente. No entanto, cada fórmula pode ser entendida no isolamento como uma verdadeira declaração sobre o problema a ser resolvido (SPIVEY, 1996, pág. 2).

A programação lógica surgiu como um paradigma distinto nos anos 70. Ela é diferente dos outros paradigmas porque requer que o programador declare os objetivos da computação, em vez dos algoritmos detalhados por meio dos quais esses objetivos podem ser alcançados (TUCKER; NOONAN, 2009, pág. 412).

2.4 Teste de Software

Teste de Software é um processo de execução de um programa elaborado para garantir que código fonte faz o que foi projetado para fazer e que não faz nada de ma-

neira não intencional, desta forma, seu objetivo encontrar erros (MYERS, 2004, pág. 8) (MYERS, 2004, pág. 8).

Para entender testes de software é fundamental que se conheça a diferença entre defeito, erro e falha. Segundo as definições estabelecidas pelo Institute of Electrical and Electronics Engineers (IEEE), defeito é uma instrução ou definição de dados incorretos; já o erro é qualquer estado intermediário incorreto ou resultado inesperado, ou seja, uma manifestação concreta de um defeito e por fim a definição de falha, é o comportamento operacional do software diferente do esperado pelo usuário (**??**).

> "Defeitos fazem parte do universo físico (a aplicação propriamente dita) e são causados por pessoas, por exemplo, através do mau uso de uma tecnologia. Defeitos podem ocasionar a manifestação de erros em um produto, ou seja, a construção de um software de forma diferente ao que foi especificado (universo de informação). Por fim, os erros geram falhas, que são comportamentos inesperados em um software que afetam diretamente o usuário final da aplicação (universo do usuário) e pode inviabilizar a utilização de um software"(NETO, 2005).

Figura 9 – Defeito x Erro x Falha.

Fonte: Neto (2005).

Teste e depuração são frequentemente confundidos, alguns acreditam até que são sinônimos, como foi mostrado anteriormente. A finalidade dos testes é mostrar que o programa tem erros, já o objetivo da depuração é encontrar o erro ou equívoco que levou ao fracasso do programa. Teste começa com condições conhecidas, utiliza procedimentos pré-definidos, e tem resultados previsíveis. A depuração começa a partir de condições iniciais, possivelmente desconhecidas (BEIZER, 1990).

Os testes podem ser projetados a partir de um ponto de vista funcional ou de um ponto de vista estrutural. Os testes funcionais são sujeitos a entradas, e suas saídas são verificadas afim de encontrar, ou não, conformidades com o comportamento especificado.

O usuário do software deve se preocupar apenas com as funcionalidade e características (BEIZER, 1990).

Caixa-preta ou teste funcional: “Teste de que ignora o mecanismo interno de um sistema ou componente e se concentra exclusivamente nas saídas geradas em resposta a entradas selecionadas e condições de execução” (**??**).

No método caixa-preta,como o próprio nome revela, vemos o programa como uma caixa preta. Seu objetivo é ser completamente indiferente sobre o comportamento interno e estrutura do programa. Em vez disso, se concentra em encontrar circunstâncias em que o programa não se comportam de acordo com as suas especificações(MYERS, 2004, pág. 13).

Caixa-branca ou teste estrutural: “Testes que leva em conta o mecanismo interno de um sistema ou componente” (**??**).

As conotações indicam adequadamente que você tem total visibilidade do funcionamento interno do produto de software, especificamente, a lógica ea estrutura do código (WILLIAMS, 2006, pág. 1).

Os testes são realizados em diferentes níveis, envolvendo o todo o sistema ou parte dele. São quatro níveis de teste: os teste de unidade, de integração, de sistema e de aceitação, como mostrado na figura 10 (NAIK; TRIPATHY, 2008, pág. 18).

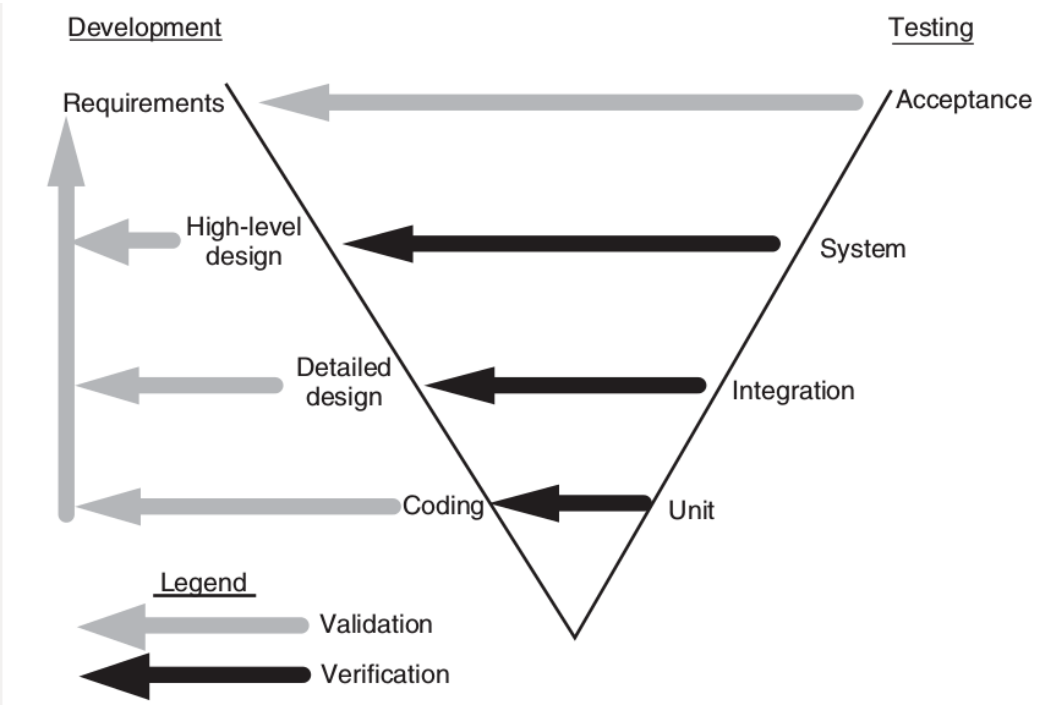

Figura 10 – Níveis de teste e seu desenvolvimento.

Fonte: Naik e TRIPATHY (2008, pág. 18).

2.4.1 Testes Unitários

"Os testes de hardware individuais ou unidades de software ou grupos de unidades relacionadas"(**??**).

Testa-se unidades individuais do programa, como as funções, métodos ou classes, de forma isolada. Depois de garantir que as unidades funcionam de forma satisfatória, os módulos são montados para a construção de sub-sistemas maiores, seguindo técnicas de teste de integração (NAIK; TRIPATHY, 2008, pág. 18).

Estes testes devem ser capazes de examinar o comportamento do código sob as mais variadas condições, ou seja, como o código deve se comportar se determinado parâmetro for passado (ou não), o que ele retorna se determinada condição for verdadeira, os efeitos colaterais que ele causa durante a sua execução, se determinada exceção é lançada (THIAGO, 2001).

O teste de unidade é importante para garantir que o código é sólido antes de ser integrado com outro código. Uma vez que o código está integrado na base de outro código, a causa de uma falha é mais difícil de ser encontrado (WILLIAMS, 2006).

2.4.2 Teste de integração

“Testes em que componentes de software, componentes de hardware, ou ambos são combinados e testados para avaliar a interação entre eles” (**??**).

Seu objetivo é a construção de um sistema razoavelmente estável que pode suportar o rigor dos testes de nível de sistema (NAIK; TRIPATHY, 2008, pág. 18).

O teste efetuado de modo incremental através da combinação de módulos em etapas. Em cada etapa um módulo é adicionado à estrutura do programa, assim o teste se concentra neste módulo recém-adicionado. Depois de ter sido demonstrado que um módulo integra adequadamente com a estrutura do programa, um outro módulo é adicionado, e o teste continua. Este processo é repetido até que todos os módulos foram integrados e testados (LEWIS, 2009, pág. 134).

Os casos de teste são escritos para examinar explicitamente as interfaces entre as diversas unidades. Estes casos de teste podem ser de caixa preta, em que o testador entende que um caso de teste requer várias unidades do programa para interagir. Como alternativa, têm-se os casos de teste caixa-branca que são escritos para exercer explicitamente as interfaces que são conhecidos para o testador (WILLIAMS, 2006).

2.4.3 Teste de Sistema

“Testes conduzidos em um sistema completo e integrado para avaliar a conformidade do sistema com os requisitos especificados” (IEEE 610, 1990).

Teste de sistema verifica se as funções foram realizados corretamente. Também verifica se certas características não funcionais estão presentes. Alguns exemplos incluem de características não funcionais: testes de usabilidade, testes de desempenho, testes de stress, testes de compatibilidade (LEWIS, 2009, pág. 134).

Uma pessoa que realiza teste de sistema deve ser capaz de pensar como um usuário final, o que implica uma profunda compreensão das atitudes e do ambiente do usuário final e de como o programa será usado. Então, se possível, um bom candidato a testador é um ou mais usuários finais. No entanto, o usuário final típico não têm a habilidade ou conhecimento para realizar muitas das categorias de testes, logo uma equipe de teste de sistema ideal pode ser composto por alguns especialistas de teste de sistema profissionais (pessoas que passam a vida realizando testes do sistema), um ou dois usuários representantes finais, um engenheiro de fatores humanos, e os principais analistas originais ou designers do programa (MYERS, 2004, pág. 104).

2.4.4 Teste de Aceitação

"Teste formal realizado para permitir que um usuário, cliente ou outra entidade autorizada, para determinar se a aceita um sistema ou componente" (**??**).

Antes de instalar e utilizar o software na vida real outro último nível de teste deve ser executado: o teste de aceitação. Aqui, o foco está no cliente de e na ótica do usuário. O teste de aceitação pode ser a única prova de que os clientes estão efetivamente envolvidos (SPILLNER; LINZ; SCHAEFER, 2014, pág. 61).

Geralmente, é realizada por um cliente ou usuário final do programa e, normalmente, não é considerada a responsabilidade da organização de desenvolvimento. No caso de um programa contratado, a organização contratante (usuário) realiza o teste de aceitação, comparando a operação do programa com o contrato original. Como é o caso de outros tipos de testes, a melhor maneira de fazer isso é criar casos de teste que tentam mostrar que o programa não atende o contrato, se esses casos de teste não forem bem sucedidos, o programa é aceito (MYERS, 2004, pág. 104).

2.5 Qualidade de Software

> "Qualidade é um termo que pode ter diferentes interpretações e para se estudar a qualidade de software de maneira efetiva é necessário, inicialmente, obter um consenso em relação à definição de qualidade de software que está sendo abordada" (BRAGA, 2012, pág. 11).

A qualidade de software é classificada em fatores internos e externos. Fatores externos são aqueles em que usuários comuns conseguem detectar, como por exemplo, a

velocidade do software ou a facilidade de uso. Os fatores externos agregam bastante valor ao usuário final, mas um bom caminho para assegurar que os fatores externos terão qualidade é caprichar na construção dos fatores internos. Portanto, técnicas de qualidade interna, como análise de qualidade de código fonte, são meios para atingir a qualidade externa (BUENO; CAMPELO, 2011, pág. 4).

2.5.1 Métricas de Qualidade de Código Fonte

Conceitos de qualidade são imprecisos e difíceis de serem aceitos em diversos contextos. No contexto de desenvolvimento de software, métricas de qualidade de código possuem o intuito de mensurar qualidade do produto de software (BUENO; CAMPELO, 2011, pág. 1).

As métricas de qualidade de código fonte podem ser objetivas ou subjetivas. As métricas subjetivas são obtidas através de regras bem definidas. As métricas subjetivas depende do sujeito que está realizando a medição. As métricas objetivas podem ser calculadas a partir de uma análise estática de código fonte de um software. Os resultados das métricas podem ser mapeados em intervalos com o intuito de serem interpretados quando analisados (MEIRELLES, 2013, pág. 14)

As métricas de qualidade de código fonte possuem papel fundamental no monitoramento e controle nas atividades de codificação e testes, realizados quase sempre por equipes que possuem diferentes formas de pensar e de realizar o trabalho criativo de codificação (SOMMERVILLE, 2011, pág. 341).

> "Ao contrário de outras engenharias, a engenharia de software não é baseada em leis quantitativas básicas, medidas absolutas não são comuns no mundo do software. Ao invés disso, tenta-se derivar um conjunto de medidas indiretas que levam a métricas que fornecem uma indicação de qualidade de alguma representação do software. Embora as métricas para software não sejam absolutas, elas fornecem uma maneira de avaliar qualidade através de um conjunto de regras definidas" (BUENO; CAMPELO, 2011, pág.).

2.5.2 Análise Estática de Código Fonte

A análise estática de código é um processo automatizado realizado por uma ferramenta sem a necessidade de execução do programa ou software a ser verificado (CHESS; WEST, 2007, pág. 64).

Na utilização da análise estática de código fonte, falhas são descobertas mais cedo no desenvolvimento, antes do programa ser executado, ainda em versões de testes. A real causa dos defeitos é revelada e não apenas suas consequências (MELO, 2011, pág. 19).

Filho (2013), define um intervalo de interpretação das métricas de qualidade de código fonte para se saber se a qualidade do código está preocupante, regular, boa ou excelente conforme a figura 11.

	ACC	ACCM	ANPM	DIT	NPA	SC
Excelente	[0, 2[	[0, 3[	[0, 2[	[0, 2[	[0, 1[	[0, 12[
Bom	[2, 7[	[3, 5[	[2, 3[	[2, 4[	[1, 2[	[12, 28[
Regular	[7, 15[	[5, 7[	[3, 5[	[4, 6[	[2, 3[	[28, 51[
Preocupante	[15, ∞[	[7, ∞[	[5, ∞[	[6, ∞[	[3, ∞[	[51, ∞[

Figura 11 – Intervalo para interpretação das métricas.

Fonte: Filho (2013).

2.5.3 Ferramentas de Análise Estática

Ferramentas de análise estática de código fonte varrem o código fonte e detectam possíveis anomalias. Também pode ser usados como parte de um processo de inspeção (SOMMERVILLE, 2011, pág. 345).

Obter os dados e extrai-los para análise estática de código fonte, não é uma tarefa trivial e requer a utilização de ferramentas automatizadas (MEIRELLES, 2013, pág. 2)

As métricas de qualidade de código fonte podem ser obtidas através do uso de uma ou mais ferramentas de extração de métricas. Existem diversas ferramentas para realização dessa atividade, mas não é possível afirmar que uma ferramenta é melhor que outras. Todas possuem pontos fortes e fracos. Para se escolher uma ferramenta de análise estática, deve-se analisar o contexto em que as mesmas estão inseridas como, por exemplo, a linguagem do projeto a ser analisado (MILLANI, 2013).

Referências

AGENTBUILDER. Auction agents for the electric power industry. 2009. Disponível em: <http://www.agentbuilder.com/Documentation/EPRI/index.html>. Citado na página 43.

AGENTBUILDER. Mercado forex: Série alertas. 2009. Disponível em: <http://www.cvm.gov.br/port/Alertas/mercadoForex.pdf>. Citado na página 42.

ALBUQUERQUE, A. A. *Alavancagem Financeira e Investimento*. Monografia (Especialização) — Faculdade de Administração, Universidade de São Paulo, São Paulo, 2013. Citado na página 30.

ALVES., T. *Um Passeio na Sequência de Fibonacci*. Monografia (Trabalho de Conclusão de Curso em Matemática) — Universidade Estadual da Paraíba, Paraíba, 2012. Citado na página 36.

AMBLER, S. *Uma Visão Realística da Reutilização em Orientação a Objetos*. [S.l.], 1998. Citado na página 41.

APT, K. R. *From Logic programming to Prolog*. 1. ed. [S.l.], 1996. Citado na página 43.

BEIZER, B. *Software Testing Techniques*. 2. ed. [S.l.], 1990. Citado 2 vezes nas páginas 45 e 46.

BELCHIOR, G. P. *Oficina de Metodologia Científica: Elaboração de Projetos de Pesquisa*. [S.l.], 2012. Citado na página 29.

BRADSHAW, J. M. *Software Agents*. [S.l.], 1997. Citado na página 41.

BRAGA, R. Qualidade de software: Avaliação de sistemas computacionais. 2012. Disponível em: <http://disciplinas.stoa.usp.br/pluginfile.php/57546/mod_resource/content/1/Aula8-QualidadeSoftware.pdf>. Citado na página 48.

BROOKSHEAR, J. G. *Ciência da Computação: Uma Visão Abrangente*. 3. ed. [S.l.], 2003. Citado na página 38.

BRUNI, A. L.; FAMÁ, R. *Gestão de custos e formação de preços*. São Paulo, 2011. Citado na página 30.

BUENO, C. S.; CAMPELO, G. B. *Qualidade de Software*. Monografia (Artigo) — Departamento de Informática,Universidade Federal de Pernambuco, Pernambuco, 2011. Citado na página 49.

CAPRETZ, L. F. *A Brief History of the Object-Oriented Approach*. [S.l.], 2003. Citado na página 39.

CHESS, B.; WEST, J. *Secure Programming with Static Analysis*. [S.l.], 2007. Citado na página 49.

COENEN, F. Tópicos de tratamento de informação: Linguagens declarativas. 1999. Disponível em: <http://cgi.csc.liv.ac.uk/~frans/OldLectures/2CS24/declarative.html#detail>. Citado na página 43.

COLLINS, V. et al. Support and resistance. 2012. Disponível em: <http://www.markets.com/pt/education/technical-analysis/support-resistance.html>. Citado 2 vezes nas páginas 30 e 31.

COSTA, S. L. *Uma Ferramenta e Técnica para o Projeto Global e Detalhado de Sistemas Multiagente*. Monografia (Mestrado em Engenharia de Eletricidade) — Departamento de Engenharia Elétrica, Universidade Federal do Maranhão, Maranhão, 2004. Citado na página 41.

CVM. Mercado forex: Série alertas. 2009. Disponível em: <http://www.cvm.gov.br/port/Alertas/mercadoForex.pdf>. Citado 3 vezes nas páginas 25, 29 e 30.

DANTAS, J. A.; MEDEIROS, O. R.; LUSTOSA, P. R. B. Reação do mercado à alavancagem operacional. 2006. Disponível em: <http://www.scielo.br/pdf/rcf/v17n41/v17n41a06.pdf>. Citado na página 30.

DEBASTINI, C. A. *Análise técnica de ações: identificando oportunidades de compra e venda*. 1. ed. [S.l.], 2008. Citado 2 vezes nas páginas 30 e 31.

DEVORE, J. L. *Probabilidade e Estatística para Engenharia e Ciências*. 6. ed. [S.l.], 2006. Citado na página 33.

DIAS, M. A. P. *Administração de materiais: uma abordagem logística*. 2. ed. [S.l.], 1985. Citado na página 32.

EASYFOREX. Leveraged forex trading: What is leverage in forex trading? 2014. Disponível em: <http://www.easy-forex.com/int/leveragedtrading/>. Citado na página 30.

FERREIRA, D. F. *Estatística básica*. 2. ed. [S.l.], 2009. Citado na página 34.

FILHO, C. *Kalibro: interpretação de métricas de código-fonte*. Monografia (Mestrado em Ciência da Computação) — Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2013. Citado 2 vezes nas páginas 49 e 50.

FXCM. Forex basic. 2011. Disponível em: <http://www.fxcm.com/forex-basics/>. Citado na página 29.

GAGLIARDI, J. D. *Fundamentos de Matemática*. Monografia (Monografia) — Universidade Federal de Campinas:, São Paulo, 2013. Citado na página 36.

GIRARDI, R. *Engenharia de Software baseada em Agentes*. [S.l.], 2004. Citado 2 vezes nas páginas 41 e 42.

HOOGLE. Functional programming. 2013. Disponível em: <http://www.haskell.org/haskellwiki/Functional_programming>. Citado na página 43.

INÁCIO, J. F. S. *Análise do Estimador de Estado por Mínimos Quadrados Ponderados*. Monografia (Monografia) — Universidade Federal do Rio Grande do Sul, Rio Grande do Sul, 2010. Citado na página 32.

JENNINGS, N. R.; SYCARA, K.; WOOLDRIDGE, M. *A Roadmap of Agent Research and Development*. [S.l.], 1998. Citado na página 42.

JENNINGS, N. R.; WOOLDRIDGE, M. *Intelligent agents: theory and practice*. [S.l.], 1995. Citado na página 42.

KONIS, K. Mathematical methods for quantitative finance. 2014. Disponível em: <https://www.coursera.org/course/mathematicalmethods>. Citado na página 31.

LEAVENS, G. T. Major programming paradigms. 2014. Disponível em: <http://www.eecs.ucf.edu/~leavens/ComS541Fall97/hw-pages/paradigms/major.html#object>. Citado na página 40.

LEWIS, W. E. *Software Testing and Continuous Quality Improvement*. 3. ed. [S.l.], 2009. Citado 2 vezes nas páginas 47 e 48.

LIRA, S. A. *Análise Correlação: Abordagem Teórica e de Construção dos Coeficientes com Aplicações*. Monografia (Dissertação Pós-Graduação) — Universidade Federal do Paraná, Paraná, 2004. Citado 2 vezes nas páginas 33 e 35.

LOPES, F. D. *Caderno didático: estatística geral*. [S.l.], 2005. Citado na página 34.

MARKET. About what is forex. 2011. Disponível em: <http://www.markets.com/pt/education/forex-education/what-is-forex.html>. Citado na página 29.

MATSURA, E. *Comprar ou Vender? Como investir na Bolsa Utilizando Análise Gráfica*. 7. ed. [S.l.], 2006. Citado 3 vezes nas páginas 30, 31 e 32.

MEIRELLES, P. *Monitoramento de métricas de código-fonte em projetos de Software Livre*. Monografia (Tese de Doutorado) — Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2013. Citado 2 vezes nas páginas 49 e 50.

MELO, J. R. F. *Análise Estática de Código*. Monografia (Monografia) — Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte, Natal, 2011. Citado na página 49.

MILLANI, L. F. G. *Análise de Correlação entre Métricas de Qualidade de Software e Métricas Físicas*. Monografia (Monografia) — Universidade Federal do Rio Grande do Sul, Rio Grande do Sul, 2013. Citado na página 50.

MYERS, G. J. *The art of Software Testing*. 2. ed. [S.l.], 2004. Citado 3 vezes nas páginas 45, 46 e 48.

NAIK, K.; TRIPATHY, P. *SOFTWARE TESTING AND QUALITY ASSURANCE Theory and Practice*. 2. ed. [S.l.], 2008. Citado 2 vezes nas páginas 46 e 47.

NETO, A. C. D. Introdução a teste de sofware. 2005. Disponível em: <http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035>. Citado na página 45.

NORMAK, K. Programming paradigms. 2013. Disponível em: <http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigms.html#paradigms_the-word_title_1>. Citado na página 36.

ODELL, J.; GIORGINI, P.; MÜLLER, J. P. *Agent-Oriented Software Engineering V*. [S.l.], 2005. Citado na página 42.

PAQUET, J.; MOKHOV, S. *Comparative Studies of Programming Languages*. [S.l.], 2010. Citado 3 vezes nas páginas 36, 37 e 44.

PIPONI, D. Eleven reasons to use haskell as a mathematician. 2006. Disponível em: <http://blog.sigfpe.com/2006/01/eleven-reasons-to-use-haskell-as.html>. Citado na página 44.

REGRA, C. M. *Tese de Mestrado em Estatística Computacional*. Monografia (Monografia) — Universidade Aberta, 2010. Citado 2 vezes nas páginas 34 e 35.

RILEY, F.; HOBSON, M. P.; BENCE, S. J. *Mathematical Methods for Physics and Engineering: A Comprehensive Guide*. [S.l.], 2011. Citado na página 31.

RITCHIE, D. M. O desenvolvimento da linguagem c. 1996. Disponível em: <http://cm.bell-labs.com/cm/cs/who/dmr/chistPT.html>. Citado na página 38.

ROCHA, R. A. *Algumas Evidências Computacionais da Infinitude dos Números Primos de Fibonacci*. Monografia (Trabalho de Conclusão de Curso em Ciência da Computação) — Universidade Federal do Rio Grande do Norte, Natal, 2008. Citado na página 36.

RUSSEL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 3. ed. [S.l.], 1995. Citado na página 41.

SCHUMACHER, M. *Objective Coordination in Multi-Agent System Engineering: design and implementation*. 1. ed. [S.l.], 2001. Citado na página 41.

SEBESTA, R. W. *Concepts of programming languages*. 10. ed. [S.l.], 2012. Citado 3 vezes nas páginas 37, 38 e 44.

SINGH, C. Difference between method overloading and overriding in java. 2014. Disponível em: <http://beginnersbook.com/2014/01/difference-between-method-overloading-and-overriding-in-java/>. Citado na página 40.

SINGH, C. Method overloading in java with examples. 2014. Disponível em: <http://beginnersbook.com/2013/05/method-overloading/>. Citado na página 40.

SOMMERVILLE, I. *Engenharia de Software*. 9. ed. [S.l.], 2011. Citado 2 vezes nas páginas 49 e 50.

SPILLNER, A.; LINZ tilo; SCHAEFER, H. *Software Testing Foundations*. 4. ed. [S.l.], 2014. Citado na página 48.

SPIVEY, M. *An introduction to logic programming through Prolog*. 1. ed. [S.l.], 1996. Citado na página 44.

STEFANOV, S. *Object-Oriented JavaScript*. [S.l.], 2008. Citado na página 38.

THIAGO, A. As vantagens do teste unitário. 2001. Disponível em: <http://andrethiago.wordpress.com/2011/04/06/as-vantagens-do-teste-unitario/>. Citado na página 47.

THOMPSON, S. *Haskell: The Craft of Functional Programming*. 2. ed. [S.l.], 1999. Citado na página 43.

TUCKER, A. B.; NOONAN, R. E. *Linguagens de programação : Princípios e paradigmas*. 2. ed. [S.l.], 2009. Citado 4 vezes nas páginas 38, 39, 43 e 44.

VENNERS, B. Polymorphism and interfaces. 1996. Disponível em: <<http://www.artima.com/objectsandjava/webuscript/PolymorphismInterfaces1.html>>. Citado na página 40.

VIALI, L. M. *Estatística Básica*. Monografia (Monografia) — Instituto de Matemática da Universidade Federal do Rio Grande do Sul, 2009. Citado na página 34.

VUOLO, J. H. *Fundamentos da Teoria de Erros*. 2. ed. [S.l.], 1996. Citado na página 32.

WEGNER, P. *Concepts and paradigms of object-oriented programming*. 1. ed. [S.l.], 1990. Citado na página 39.

WEISFELD, M. A. *The Object-Oriented Thought Process*. 3. ed. [S.l.], 2009. Citado na página 40.

WILLIAMS, L. *White-Box Testing*. [S.l.], 2006. Citado 2 vezes nas páginas 46 e 47.

WOOLDRIDGE, M. *Teamwork in Multi-agent systems: A Formal Approach*. 1. ed. [S.l.], 2010. Citado 2 vezes nas páginas 41 e 42.