UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

MAC-0499 TRABALHO DE FORMATURA SUPERVISIONADO

Tail
A Java Technical Analysis Library

Alunos:

Alexandre Oki Takinami

Carlos Eduardo Manssuer

Márcio Vinicius dos Santos

Thiago Garutti Thies

Orientador:

Alfredo Goldman

Co-Orientadores:

Danilo Toshiaki Sato

Julian Monteiro

Paulo Silveira

Abstract

Technical Analysis studies forecasting future price trends with the objective of manage a best moment to buy and sell shares. The Tail's target is to develop a Java Open-Source library that abstracts the basic components of Technical Analysis, supplying tools for creation, manipulation and evaluation of strategies to buy and sell.

The objective of this paper is to present the Tail library, its methodology, evolution and results obtained until now.

Resumo

A Análise Técnica estuda a série temporal dos ativos com o objetivo de determinar o melhor momento de compra e venda. O projeto Tail tem como principal meta desenvolver uma biblioteca Java Open-Source que abstrai os componentes básicos da Análise Técnica Grafista, fornecendo ferramentas para manipulação, criação e avaliação de estratégias de compra e venda.

O objetivo dessa monografia é apresentar a biblioteca Tail, sua metodologia, evolução e resultados obtidos com seu uso até então.

Motivação

Poucas atividades humanas exercem tanto fascínio quanto o mercado de ações, retratado exaustivamente em filmes, livros e em toda a cultura contemporânea. O exercício de compra e venda de ações é praticado desde a Roma antiga, mas somente na cidade de Bruges, na Bélgica em 1487, houve o surgimento da primeira bolsa de valores, quando o comércio de papéis começou a se destacar a ponto de possuir direitos e um prédio próprio. Desde então, esta prática cresce não só por meio de negociações de grupos de investidores e instituições financeiras, mas também por investimentos de

pessoas físicas, que procuram obter um maior rendimento para seu dinheiro pessoal. Hoje, estima-se que mais de 60% das famílias norte-americanas têm uma parte significativa de suas economias voltadas para a compra de ações, e esta é uma tendência que também deve chegar ao Brasil.

Apesar da instabilidade do mercado e do medo de investir devido ao seu aspecto abstrato (pois não se vê moedas, nem papéis impressos e apenas números mudando ininterruptamente em um grande painel), a explicação para o crescimento desta área é simples: a propensão humana à troca, o desafio de se conseguir prever o futuro e o sentimento de se estar manipulando diretamente com dinheiro propenso à grande incerteza, formam a receita ideal para que uma pessoa tenha o sentimento de estar "jogando", assim como em um cassino. Também por estes motivos muitas pessoas criticam, afirmando que estão lidando apenas com o fator sorte e que seu dinheiro não está seguro, mas como em qualquer área, a compreensão da lógica e do funcionamento do mercado de ações é essencial para que se tenha sucesso.

Existem inúmeras ferramentas para a compra e venda de ativos, entretanto nenhuma delas tem como objetivo a visualização dos resultados da aplicação de uma técnica grafista em uma linha histórica de um ativo. Possibilitar este tipo de recurso não somente facilita o aprendizado da análise grafista para investidores casuais, mas também diminui a obscuridade do mercado acionário e facilita a criação de novas técnicas.

Capítulo 1

Introdução

A idéia do projeto Tail nasceu quando um grupo de alunos procurou o prof. Alfredo Goldman para orientá-los no desenvolvimento de um sofware para o projeto de conclusão de curso. Ele então teve a idéia de juntar ao grupo alguns alunos do mestrado através de um sistema de co-orientação, onde os mestrandos auxiliariam os graduandos na implementação, modelagem e metodologia do projeto. Somente então o grupo definiu o tema: o desenvolvimento de um software open-source de análise técnica.

Essa idéia, ainda vaga, foi gradativamente tomando a forma do projeto desenvolvido. O grupo se reunia semanalmente adaptando o projeto, atribuindo novas tarefas e objetivos. Os graduandos tiveram a oportunidade de trabalhar em conjunto com os mestrandos, que compartilharam suas experiências anteriores.

1.1 Objetivos do Projeto

- Implementar os componentes básicos da análise técnica grafista: série temporal, operações de compra e venda e indicadores técnicos;
- Implementar as estratégias de compra e venda mais utilizadas no mercado, assim como permitir o rápido desenvolvimento de novas estratégias;

- Implementar um algoritmo genérico para determinar um momento apropriado de compra e venda de um ativo, através da escolha da melhor estratégia aplicada a uma série temporal;
- Permitir que o critério de escolha da melhor estratégia seja trocado e desenvolvido facilmente;
- Criar relatórios que facilitem o estudo e a compreensão dos resultados obtidos pelo algoritmo;
- Criar uma interface gráfica, permitindo o uso das ferramentas implementadas de forma fácil, rápida e de simples entendimento, mas que não limite os recursos da biblioteca;
- Arquitetura orientada a objetos, com o objetivo de ser facilmente escalável e de simples entendimento;
- Utilizar práticas de XP, adaptando-as conforme as necessidades do grupo.
- Manter a cobertura de testes superior a 90
- Analisar o funcionamento do sistema de co-orientação, com o objetivo estendê-lo para projetos futuros.

1.2 Organização da Monografia

A monografia foi organizada da seguinte forma:

Introdução

Como surgiu o projeto, suas motivação e objetivos.

Conceitos e Tecnologias Estudadas

Nessa seção são explicados conceitos básicos de análise técnica grafista, assim como os conceitos utilizados para o desenvolvimento do software.

Desenvolvimento do software

Mostra passo a passo o processo de modelagem do projeto, alguns detalhes importantes e as tecnologias utilizadas na aplicação.

Conclusões

Análise dos resultados obtidos tanto no projeto quanto com a metodologia.

Sumário

1 Introdução **1**

1.1 Objetivos do Projeto 1

1.2 Organização da Monografia 2

2 Conceitos e Tecnologias Estudadas **5**

2.1 Definição do Projeto 5

2.2 Mercado Financeiro 6

2.2.1 Mercado de Ações a Vista 7

2.2.2 As duas faces do mercado 8

2.3 A Análise Técnica 8

2.3.1 A Escola Fundamentalista 9

2.3.2 A Escola Técnica Grafista 9

2.4 Programação Orientada a Objetos (P.O.O.) 11

2.4.1 Conceitos Fundamentais 12

2.5 Padrões de Projeto (Design Patterns) 18

2.6 Programação Extrema - A metodologia XP 26

2.6.1 A Programação Extrema 27

2.6.2 A metodologia aplicada no desenvolvimento do projeto 31

3 Desenvolvimento do Software **35**

3.1 Modelagem 35

3.1.1 Abstração dos componentes básicos da análise técnica 35
3.1.2 Abstração das operações de compra e venda 37
3.1.3 Escolhendo a melhor estratégia de compra e venda 42
3.2 Detalhes do Projeto 47
3.2.1 Exemplos de indicadores técnicos implementados 48
3.2.2 Exemplos de estratégias implementadas 50
3.2.3 Critérios implementados 52
3.3 As ferramentas utilizadas 53
3.4 O TailS (TAIL Simulator) 55
3.5 Domain Specific Language (DSL) 56
3.5.1 Interpretador de Scripts 57
3.5.2 Mini-IDE 57

4 Considerações Finais **58**
4.1 Resultados obtidos 58
4.1.1 Avaliação da Metodologia 58
4.1.2 Avaliação dos objetivos 58
4.1.3 Avaliação dos resultados com o uso da aplicação 59
4.2 Cobertura de teste superior a 90% 60
4.3 O futuro do projeto 60

Capítulo 2

Conceitos e Tecnologias Estudadas

2.1 Definição do Projeto

Como todo sistema desenvolvido de forma modular, é difícil determinar desde o início todas suas funcionalidades e necessidades, mesmo assim, as seguintes metas foram definidas:

- Ler uma série temporal de um ativo;
- Calcular indicadores técnicos;
- Viabilizar a aplicação dos indicadores técnicos sobre os ativos utilizando uma vasta gama de estratégias de compra e venda;
- Criar relatórios que facilitem o estudo e a compreensão dos resultados obtidos;
- Arquitetura orientada a objetos, com o objetivo de ser facilmente escalável e de simples entendimento;
- Utilizar práticas de XP, adaptando-as conforme as necessidades do grupo;
- Manter a cobertura de testes superior a 90%;

- Criar uma interface gráfica, permitindo o uso das ferramentas implementadas de forma fácil, rápida e de simples entendimento, mas que não limitasse os recursos da biblioteca.

2.2 Mercado Financeiro

O Mercado Financeiro é segmentado em 4 subdivisões:

- **Mercado Monetário**: envolve operações relacionadas com as taxas de juros pretendidas pelas políticas econômicas do governo. Os papéis são negociados tendo como parâmetro a taxa básica de juros. Exemplo: papéis do Tesouro Nacional, títulos públicos emitidos pelos Estados e Municípios, Certificado de Depósito Interbancário (CDI), Certificado de Depósito Bancário (CDB) e debêntures.

- **Mercado de Crédito**: visa suprir a necessidade dos vários agentes econômicos, concedendo crédito às pessoas físicas, empréstimos e financiamentos às empresas. Estas operações são tipicamente realizadas por instituições bancárias.

- **Mercado Cambial**: é o segmento em que ocorrem operações de compra e venda de moedas internacionais convertíveis. Reúne agentes econômicos que realizam transações monetárias com o exterior.

- **Mercado de Capitais**: é o mais relevante no processo de desenvolvimento econômico, pois consegue ligar aqueles que têm a capacidade de poupar (os investidores) aos carentes de recursos de longo prazo. Também oferece financiamento com prazo indeterminado, que envolve a emissão e subscrição de ações. Exemplo típico são as cadernetas de poupança e o mercado de ação a vista.

2.2.1 Mercado de Ações a Vista

A função do mercado de ações é de promover a riqueza nacional através de participações dos investimentos de agentes econômicos. Os principais ativos negociados neste mercado são as ações, que são pequenas frações do capital de uma empresa. O mercado de ações a vista é conhecido como aquele que paga todas as operações de compra e venda no prazo máximo de dois dias. As empresas que disponibilizam suas ações são chamadas de empresas de capital aberto e são registradas em um órgão público chamado CVM (Comissão de Valores Mobiliários), sendo obrigadas a periodicamente divulgar descrições sobre a empresa em caráter econômico, social e financeiro.

Ações trazem retorno ao investidor de duas maneiras: dividendo e valorização. Dividendo é a parte do lucro das empresas repassada aos acionistas, geralmente em dinheiro. A valorização das ações é consequência do comportamento do mercado e do desempenho da empresa em determinado período, podendo variar por caráter especulativo.

O mercado de ações pode ser segmentado em dois: primário e secundário. No mercado primário as empresas colocam seus papéis a venda. No mercado secundário ocorrem renegociações entre os agentes econômicos que detêm as ações adquiridas no mercado primário e os que desejam comprá-las. O resultado de uma negociação do mercado secundário, ao contrário do primário, não é repassado ao financiamento da empresa, sendo apenas simples transferência entre investidores.

Bolsas de Valores são associações civis sem fins lucrativos que visam manter um local em condições adequadas para a realização de operações de valores mobiliários, devem promover a segurança do capital, a liquidez nas negociações e cumprimento das normas legais.

As operações na Bolsa são dadas a partir de ordens de compra e venda por meio de uma Sociedade Corretora devidamente cadastrada. As negociações ocorrem em sua maioria durante o horário de funcionamento do mercado, entretanto após o fechamento ainda é possível executar ordens de compra e venda num curto horário

pré-estabelecido, conhecido como *after market*.

Os índices da Bolsa mostram a evolução do mercado em um período, refletindo o comportamento dos investimentos em ações e as tendências da economia. Índices são carteiras fictícias compostas por um conjunto de ações, calculados pelo desempenho médio de seu conteúdo. O índice mais importante é o Ibovespa (Índice Bovespa), composto pelos 60 ativos mais negociados na Bolsa de Valores de São Paulo. A cada quadrimestre é feita uma reavaliação do mercado, atualizando as ações participantes do Ibovespa.

2.2.2 As duas faces do mercado

Na bolsa de valores dois personagens se destacam: os investidores e os especuladores. O investidor preserva pela segurança de seu patrimônio e espera por uma rentabilidade estável, tentando saber antecipadamente quais serão os seus proventos. O especulador (do latim speculare, que significa vigiar, ficar atento) tem como principal fonte de lucro o provento das ações, focando em negociações com possibilidade de retorno financeiro maior, e consequentemente menos seguras.

2.3 A Análise Técnica

Todos os investidores que almejam lucro na Bolsa precisam adquirir ações em tendência de valorização antes dos demais acionistas e vendê-las antes que desvalorizem. Mas para adquirir o *timing* certo para operar é preciso conhecimento do mercado financeiro e do desempenho das empresas, tanto no âmbito brasileiro quanto global.

Escolas foram criadas com o objetivo de prever o comportamento da Bolsa, recomendando os melhores momentos de compra e venda. As principais escolas são a Fundamentalista e a Técnica Grafista.

2.3.1 A Escola Fundamentalista

A Escola Fundamentalista tem por objetivo determinar o preço da ação de uma empresa analisando todas as informações que podem alterar o seu valor no mercado. Informações como o patrimônio da empresa, seu desempenho, sua posição no setor de atuação, sua concorrência, fatores financeiros, dados econômicos, política monetária, mudança no quadro técnico, mudança na direção, programas de pesquisas e campanhas publicitárias são levadas em consideração. A imprensa diária, publicações setoriais e balanços das empresas são permanentes fontes de consulta para os analistas fundamentalistas.

Esta escola baseia-se no fato de que existe uma correlação lógica entre o valor da empresa e o preço de seu ativo. Quando o valor da empresa esta acima do preço da ação ocorre uma indicação de compra, por outro lado, quando o preço do ativo é maior do que o valor da empresa, ocorre uma indicação de venda.

Os fundamentalistas tentam prever a evolução das ações de uma empresa através da análise de seu prospécto, não se importando com a evolução histórica da ação.

A reação do mercado de ações sobre a descoberta da enorme reserva de petróleo na bacia de Santos impulsionando o valor das ações da Petrobras em mais de 10% é um exemplo de uso de análise fundamentalista.

2.3.2 A Escola Técnica Grafista

Visa avaliar o melhor momento para compra e venda de ações através da análise histórica e comportamental do ativo na Bolsa.

Suas principais características são:

- Análise dos dados gerados nas transações (preço, volume, etc).
- Uso de gráficos na busca de padrões.
- Análise de tendências.

Uma das várias formas de se aplicar as premissas da análise técnica grafista é através do uso de indicadores técnicos, fórmulas que analisam as transações através de seus dados (preço de abertura, preço de fechamento, volume, etc).

Uma vantagem do uso de indicadores é mostrar o comportamento do ativo, auxiliando na tomada de decisões de compra e venda. Entretanto é muito difícil determinar o melhor indicador e ajustá-lo o melhor possível para um determinado ativo.

A análise técnica surgiu no início do século XX com os trabalhos de Charles Dow e seu sócio Edward Jones divulgados em artigos para o *The Wall Street Journal*. Ambos formaram a empresa *Dow Jones & Company* e foram os primeiros a criarem índices para tentar prever o comportamento do mercado de ações. O primeiro índice era formado por 11 ações (9 empresas do setor ferroviário e 2 indústrias) e deram origem ao que hoje é conhecido como o índice Dow-Jones.

Todos estudos divulgados originaram a Teoria de Dow, a base da análise técnica. A teoria diz que toda oscilação de alta ou de baixa é anteriormente sinalizada e portanto passível de ser prevista. Toda análise é baseada no estudo de gráficos da ação e seus indicadores, sem levar em conta fatores externos. A teoria é baseada nos seguintes princípios:

- **Os índices descontam tudo**: Tudo que acontece fora do mercado é refletido pelos gráficos. Notícias econômicas, balanços patrimoniais, catástrofes naturais e acidentes são rapidamente absorvidos pelo mercado;

- **As três tendências do mercado**: Segundo Dow, o mercado sofre ação de três tendências: primária, secundária e terciária. A primária é a principal do mercado, nela são observados movimentos longos de alta ou de baixa e tem uma duração que pode variar de 1 a 2 anos. A secundária acontece dentro da primária e tem duração de alguns meses, nela são notados variações no valor do ativo. Uma valorização é sempre seguida por uma desvalorização, e o contrário também acontece. A terciária é análoga à secundária, porém tem duração de algumas se-

manas. Tendências são importantes para a análise técnica pois facilitam a avaliação do ativo;

- **As três fases dos movimentos**: Dow percebeu que um mercado em alta ou em baixa apresentam as mesmas características, movimentando-se em três fases. No caso da alta, a primeira fase é quando o mercado começa a crescer e os investidores mais qualificados são os primeiros a comprar, aproveitando o bom momento. A segunda fase é quando as informações começam a ser dissipadas e outros investidores compram, acelerando a valorização do ativo. A terceira fase é quando os preços estão mais elevados e o investidores mais qualificados começam a preparar um novo processo de venda. O mesmo processo ocorre para um mercado em baixa;

- **O princípio da confirmação**: Se um índice mostra rompimento de tendência ou rompimento da barreira de suporte e resistência, para que este seja válido, é preciso que exista um outro índice diferente que confirme o fato;

- **Volume deve confirmar a tendência**: Em uma tendência de alta, o volume de negociações aumenta caso o papel esteja se valorizando e diminui caso o papel desvalorize. Em uma tendência de baixa, o volume diminui caso o papel valorize e aumenta caso o papel desvalorize;

- **A tendência continua até surgir um sinal de reversão**: Uma tendência só se reverterá caso haja um sinal bem claro dessa alteração. Isso mostra que uma ação só para de subir, ou descer, caso haja algum motivo.

2.4 Programação Orientada a Objetos (P.O.O.)

Programação orientada a objetos é um paradigma de programação que usa classes, objetos e suas interações para abstrair componentes do mundo real.

2.4.1 Conceitos Fundamentais

Classe

Define as características básicas de um objeto (através de atributos) e seu comportamento (através de métodos), provendo uma estrutura modular para o programa.

Objeto

Consiste em uma instância particular de uma classe. O valor do conjunto de seus atributos é denominado estado do objeto. O estado inicial de um objeto é definido por sua classe.

Exemplo: A classe **SMAIndicator** representa o indicador técnico de média simples mais conhecido como *Simple Moving Average* (SMA), ela possui um objeto do tipo indicator e um inteiro como atributos e fornece o método *getValue(int index)* que dado um índice retorna o valor do indicador SMA.

```
class SMAIndicator
{
  Indicator indicator;
  int timeFrame;
  SMAIndicator(Indicator indicator, int timeFrame) {
    this.indicator = indicator;
    this.timeFrame = timeFrame;
  }
  Double getValue(int index) {
    return calculate(index);
  }
  Double calculate(int index) {
    double sum = 0.0;
```

```
    for (int i = Math.max(0, index - timeFrame + 1); i <= index; i++) {
      sum += indicator.getValue(i);
    }
    return sum / Math.min(timeFrame, index + 1);
  }
}
```

Encapsulamento

Esconde os detalhes de implementação de uma classe dos clientes. Um cliente somente tem acesso aos métodos e atributos declaradamente acessíveis, impedindo o uso de métodos e alterações de atributos indevidamente.

Exemplo: Em Java, uma das formas de usar o encapsulamento é através dos modificadores de acesso *private*, que somente permite que métodos da própria classe acessem o método/atributo e *public*, que permite que qualquer classe acesse o método/atributo. Vamos aplicar o encapsulamento no exemplo anterior.

```
public class SMAIndicator
{
  private Indicator indicator;
  private timeFrame;
  public SMAIndicator(Indicator indicator, int timeFrame) {
    this.indicator = indicator;
    this.timeFrame = timeFrame;
  }
  public Double getValue(int index) {
    return calculate(index);
  }
```

```
  private Double calculate(int index) {
    double sum = 0.0;
    for (int i = Math.max(0, index - timeFrame + 1); i <= index; i++) {
      sum += indicator.getValue(i);
    }
    return sum / Math.min(timeFrame, index + 1);
  }
}
```

Herança

Permite que características de um objeto sejam estendidas a outros objetos. Uma classe pode estender outra classe aproveitando seus métodos e atributos.

Exemplo: Um dos problemas que ocorreram durante o desenvolvimento foi a perda de desempenho da aplicação devido o recálculo de um mesmo índice de indicadores. Foi necessário implementar uma forma de memorizar os índices já calculados de um modo simples e organizado, evitando repetição de código. A solução encontrada foi desenvolver uma classe abstrata mãe de todos os indicadores cacheados chamada **CachedIndicator**, qualquer indicador que necessitasse dessa funcionalidade simplesmente precisaria estendê-la.

```
public abstract class CachedIndicator
{
  private List<Double> results;
  public CachedIndicator() {
    results = new ArrayList<Double>();
  }
  public T getValue(int index) {
```

```
    increaseLength(index);
    if (results.get(index) == null) {
      int i = index;
      while (i > 0 && results.get(i--) == null);
      for (; i <= index; i++) {
        results.set(i, calculate(i));
      }
    }
    return results.get(index);
  }
  private void increaseLength(int index) {
    if (results.size() <= index) {
      results.addAll(Collections.<Double>
nCopies(index - results.size() + 1, null));
    }
  }
  protected abstract Double calculate(int index);
}
```

O modificador de acesso *abstract* significa que a classe invoca métodos somente implementados pelos seus filhos e consequentemente não pode ser instanciada. Esses métodos recebem o modificador *abstract* em sua assinatura. O modificador de acesso *protected* permite que somente as classes filhas tenham acesso ao método/atributo.O método *getValue(int index)* verifica se já foi armazenado o valor calculado no ponto, armazenando em caso negativo e retorno o valor. O modo como o valor foi calculado (através da chamada do método *calculate(int index)*) é obrigação da classe filha.

Agora somente precisa fazer a classe **SMAIndicator** estender **CachedIndicator**.

```
public class SMAIndicator extends CachedIndicator
{
  private Indicator indicator;
  private timeFrame;
  public SMAIndicator(Indicator indicator, int timeFrame) {
    this.indicator = indicator;
    this.timeFrame = timeFrame;
  }
  private Double calculate(int index) {
    double sum = 0.0;
    for (int i = Math.max(0, index - timeFrame + 1); i <= index; i++) {
      sum += indicator.getValue(i);
    }
    return sum / Math.min(timeFrame, index + 1);
  }
}
```

Perceba que **SMAIndicator** não precisou mais implementar o método *getValue(int index)* (já implementado pelo **CachedIndicator**), somente precisou implementar sua versão própria do método *calculate(int index)*.

Polimorfismo

Objetos derivados de um mesmo objeto possuem métodos com assinaturas iguais, porém com comportamentos diferentes, dependendo da especialização do objeto. O uso do polimorfismo permite que esses objetos sejam tratados da mesma maneira.

Exemplo: No desenvolvimento do projeto, foi necessário implementar diversos indicadores, de uma forma simples e organizada, evitando necessidade de mais código para tratamento de indicadores sempre que um novo indicador fosse desenvolvido.

Foi criada com esse objetivo a interface ***Indicator***, o código abaixo mostra a Interface ***Indicator***.

```
public interface Indicator {
  public Double getValue(int index);
}
```

Falta obrigar a classe **CachedIndicator** a implementar ***Indicator*** (As classes filhas automaticamente implementam.)

```
public abstract class CachedIndicator implements Indicator
{
  private List<Double> results;
  public CachedIndicator() {
    results = new ArrayList<Double>();
  }
  public T getValue(int index) {
    increaseLength(index);
    if (results.get(index) == null) {
      int i = index;
      while (i > 0 && results.get(i--) == null);
      for (; i <= index; i++) {
        results.set(i, calculate(i));
        &amp;amp;nbsp;       }
    }
    return results.get(index);
  }
  private void increaseLength(int index) {
```

```
    if (results.size() <= index) {
      results.addAll(Collections.<Double>
nCopies(index - results.size() + 1, null));
    }
  }
  protected abstract Double calculate(int index);
}
```

Interfaces permitem o uso do polimorfismo, uma vez que todas as classes que a implementam possuem as mesmas chamadas de método.

```
    Indicator close = new ClosePriceIndicator(data);
    Indicator sma = new SmaIndicator(close, 3);
```

Todos indicadores implementam a mesma interface ***Indicator***. Existem inúmeras ferramentas para a compra e venda de ativos, entretanto nenhuma delas tem como objetivo a visualização dos resultados da aplicação de uma técnica grafista em uma linha histórica de um ativo. Possibilitar este tipo de recurso não somente facilita o aprendizado da análise grafista para investidores casuais, mas também diminui a obscuridade do mercado acionário e facilita a criação de novas técnicas, logo possuem as mesmas chamadas de método, facilitando o desenvolvimento e tratamento de novos indicadores.

2.5 Padrões de Projeto (Design Patterns)

Design Patterns são soluções, na modelagem do projeto, reutilizadas para problemas que ocorrem com freqüência no desenvolvimento do software. Os padrões geralmente são descrições ou *templates* de como solucionar um problema que podem ser usados

em diversas situações. Os *patterns* adotados no projeto agilizaram o desenvolvimento, facilitaram o entendimento do código e a utilização dos conceitos de orientação a objetos. Para cada padrão há uma documentação descrevendo quando o *pattern* é usado, para que serve , suas vantagens e desvantagens.

Alguns exemplos de *patterns* utilizados no projeto:

Observer

Usado para permitir que um ou mais objetos (*Observers*) sejam notificados caso exista mudança de estado de um outro objeto (*Subject*). Todos os *Observers* tratam a mudança de estado individualmente. Objetos podem ser adicionados ou removidos da lista de *ObserversSubject* que possui uma lista de *Observers*.

No projeto, o *pattern* foi utilizado nas classes **SeriesDataset** (*Subject*) e **SeriesChart** (*Observer*). **SeriesChart** é responsável por desenhar um gráfico e **SeriesDataset** contém os pontos a serem desenhados. Quando o usuário deseja ver outros pontos do gráfico (através das chamadas de método *moveLeft()* e *moveRight()*), o **SeriesDataSet** recalcula os pontos e notifica o **SeriesChart** através da assinatura *datasetChanged()*. O modo como os pontos serão redesenhados depende da implementação de **SeriesChart**. A figura 2.1 mostra o padrão *Observer* utilizado no Tail.

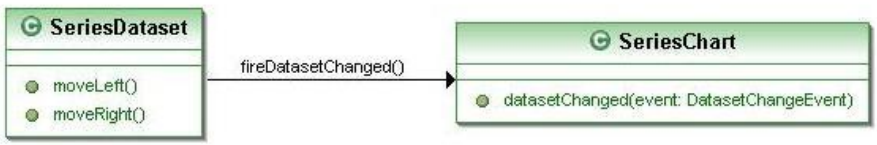


Figura 2.1: Padrão Observer no Tail

Template Method

Utilizado quando mais de um objeto contém algoritmos semelhantes. Consiste em uma classe mãe contendo a parte comum do algoritmo e classes filhas contendo implementações das partes diferentes. A classe mãe obriga que suas filhas implementem os métodos necessários para a completude do algoritmo. Este *pattern* evita duplicação de código, facilitando a manutenção e a escalabilidade da aplicação.

No projeto, o padrão foi utilizado na classe **AbstractStrategy**, como é mostrado no diagrama da figura 2.2.

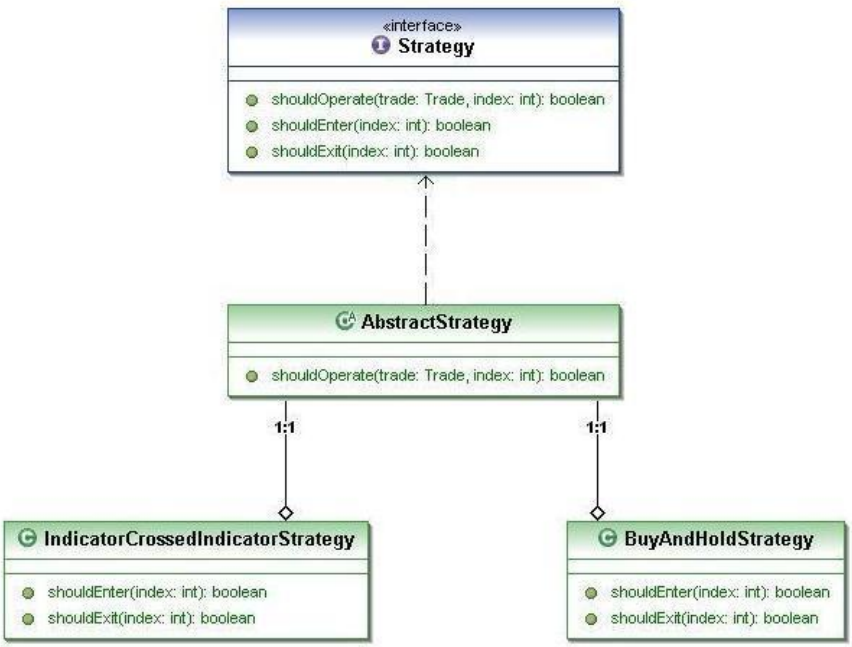


Figura 2.2: Padrão Template Method no Tail

A interface ***Strategy*** contém as chamadas de método *shouldEnter()*, *shouldExit()* e *shouldOperate()*. A classe **AbstractStrategy** implementa o *shouldOperate()* e as classes filhas implemetam o *shoudEnter()* e o *shouldExit()*. Apesar de não implementados

por **AbstractStrategy** os métodos *shouldEnter()* e *shouldExit()* são invocados durante a execução do método *shouldOperate()*.

Wrapper

Encapsula um objeto (*adaptable*) em um novo tipo, adaptando-o de um modo aceito por outro objeto (*client*). Consiste em uma classe (*wrapper*) que recebe *adaptable* e é do tipo aceito por *client*. Todas as invocações de métodos do *wrapper* são transformadas em invocações de métodos do adaptable.

Wrappers foram utilizados na modelagem de indicadores. Um exemplo são os **SimpleIndicators**.

SimpleIndicators são classes que recebem a série temporal (uma instância de ***TimeSeries***) e a transformam em um indicador. Um exemplo é a classe **ClosePriceIndicator** que recebe um ***TimeSeries*** no construtor e através da sua assinatura *getValue(int i)*, retorna o valor do preço de fachamento do ***TimeSeries*** no ponto i.

Decorator

Permite a adição de um novo comportamento a um objeto dinamicamente. O detentor do novo comportamento (*Decorator*) e o receptor são do mesmo tipo, permitindo que chamadas de responsabilidade do receptor sejam repassadas diretamente.

No projeto o *pattern* foi utilizado na modelagem dos indicadores. Um exemplo desse uso na figura 2.3.

A classe **ClosePriceindicator** retorna através da assinatura *getValue(int i)*, o preço de fechamento de uma ação no índice i. O **SMAIndicatorIndicator** recebe um ***Indicator*** no construtor e retorna através de *getValue(int i)*, o SMA do indicador no ponto i. Com o uso de *decorators* é fácil trocar o parâmetro do ativo a ser calculado.

```
/* calcula o sma do preço de fechamento */
```

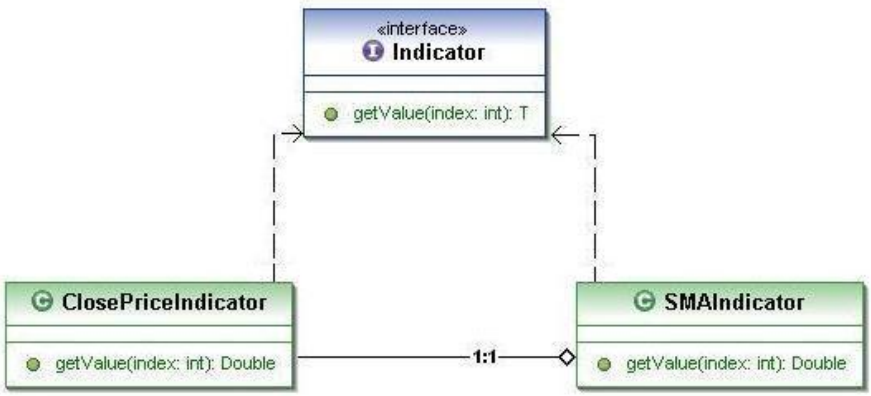


Figura 2.3: Padrão Decorator no Tail

```
Indicator indicator = new SMAIndicator( new ClosePriceIndicator(data) );
/* calcula o sma do preço de abertura */
Indicator indicator = new SMAIndicator( new OpenPriceIndicator(data) );
```

Factory Method

Define uma interface(***Creator***) para um objeto criador (**ConcreteCreator**) de objetos (**Product**). Permite determinar em tempo de execução qual tipo de *Product* será criado. Encapsula a construção, facilitando a instanciação de objetos com alta complexidade de criação.

No projeto o pattern foi utilizado para permitir a escolha do modo de avaliação de um critério em tempo de execução. Vide figura 2.4

As implementações de ***StrategyEvaluatorFactory*** criam diferentes implementações de ***StrategyEvaluator*** através do método create, permitindo que a criação dos ***StrategyEvaluators*** fique a cargo do algoritmo de avaliação de estratégias /textbfWalk-Forward.

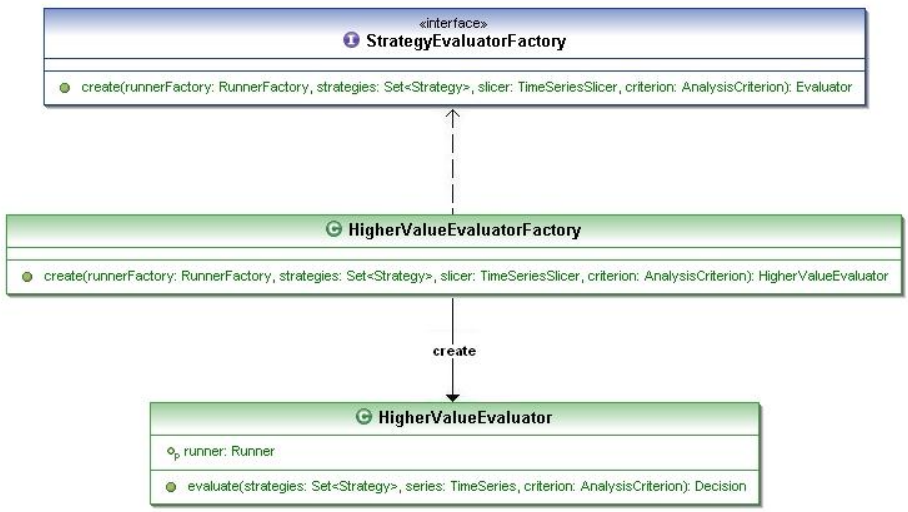


Figura 2.4: Exemplo de **Factory Method**

```
StrategyEvaluatorFactory evalFactory = new BestStrategyEvaluatorFactory();
WalkForward walker = new WalkForward(evalFactory);
```

Strategy Pattern

Define uma família de algoritmos, encapsulando cada um, e permite sua intercambialidade em tempo de execução. Permite a combinação de componentes gerando novos algoritmos. Consiste em um algoritmo genérico (*Context*) que utiliza implementações diferentes de uma estratégia (*Strategy*).

Coincidentemente um exemplo do *Strategy Pattern* no projeto é a interface ***Strategy***, que determina qual será a estratégia de compra e venda a ser aplicada em uma série temporal pela interface ***Runner***. Vide figura 2.5

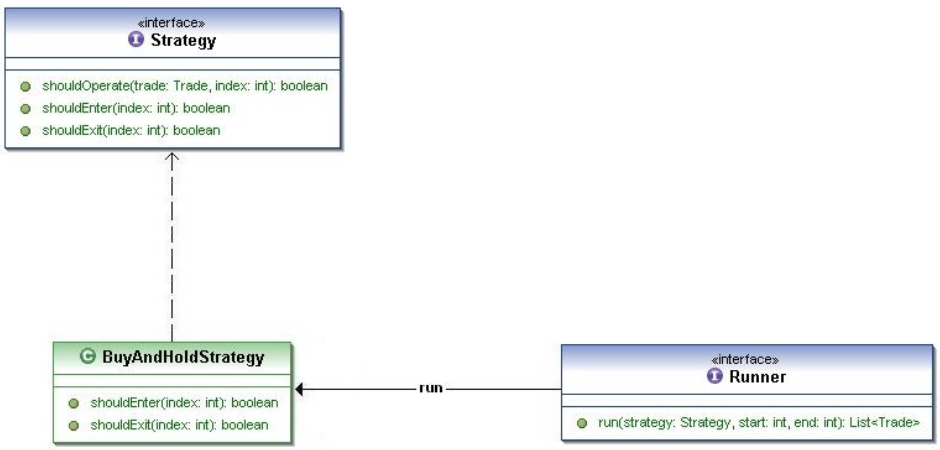


Figura 2.5: Exemplo de **Strategy Patten**

Uma instância de ***Runner*** recebe uma ***Strategy*** e uma série temporal no construtor, e através do método *run()*, determina os pontos de compra e venda da série através da ***Strategy***.

```
Strategy buyAndHold = new BuyAndHoldStrategy();
Runner runner = new HistoryRunner(buyAndHold);
```

Specification Pattern

Combina dois algoritmos de uma mesma família utilizando lógica booleana. Consiste em colocar na interface do algoritmo os métodos booleanos *And()*, *Or()*, *Not()* que recebem e retornam objetos do mesmo tipo da interface. Permite a criação de novos algoritmos através da combinação.

No projeto o pattern possibilitou a combinação de estratégias de compra e venda. Veja a figura 2.6.

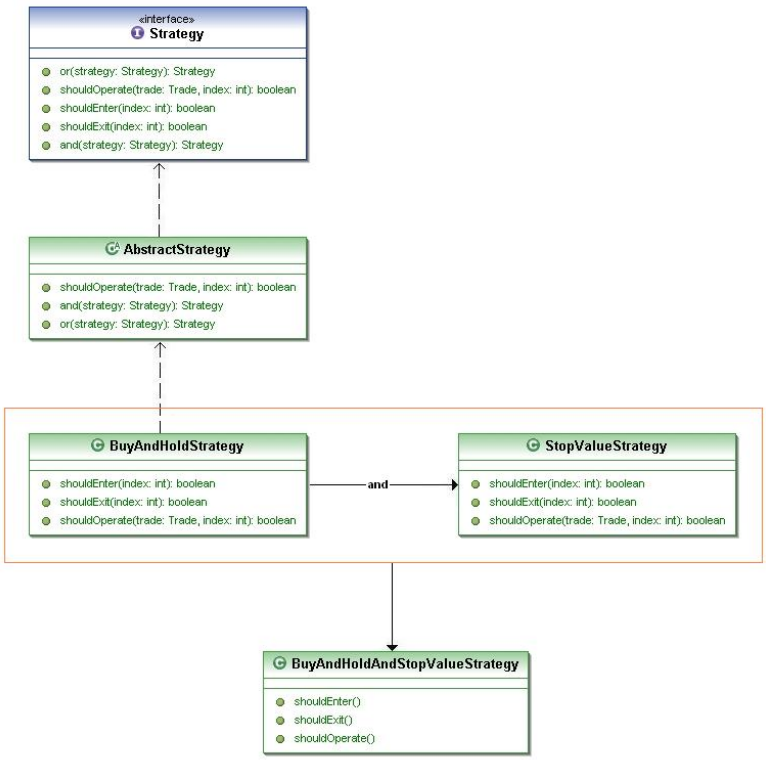


Figura 2.6: Exemplo de **Specification Pattern**

Foi implementada a classe abstrata **AbstractStrategy** que implementa os métodos *and(Strategy)* e *or(Strategy)* da interface ***Strategy***. **BuyAndHoldStrategy** é a estratégia que "segura"o ativo comprando no inicio e vendendo no final da série temporal. **StopValueStrategy** recebe no construtor um percentual, e somente vende caso a ação baixe mais que o percentual armazenado. Combinando as duas estratégias conseguimos uma estratégia que compra no primeiro e caso o valor não passe de um percentual venda no último.

```
Strategy buyAndHold = new BuyAndHoldStrategy();
```

```
/* valor não pode ser menor que 10 %*/
Strategy stopValue = new StopValueStrategy(0.10);
Strategy buyAndHoldAndStopStrategy = buyAndHold.and(stopValue);
```

2.6 Programação Extrema - A metodologia XP

Métodos ágeis são nova formas de desenvolvimento de software baseada em valores essenciais pré-estabelecidos pelo grupo de desenvolvimento, foi baseada no *Manifesto for Agile Software Development*[1], por desenvolvedores contrários às metodologias de desenvolvimento tradicionais.

Nesse manifesto estão descritos os princípios para o desenvolvimento ágil de software:

- A prioridade máxima é a satisfação do cliente;
- Entregas funcionais e freqüentes (algumas semanas ou meses);
- Mudanças no escopo acontecem e devem ser adicionadas ao projeto, isso mostra credibilidade no trabalho e é uma vantagem competitiva para a equipe;
- Equipe de desenvolvimento deve trabalhar em conjunto com a equipe de negócios;
- Confiança na equipe de desenvolvimento;
- Manter a equipe sempre motivada;
- Comunicação face-a-face entre os desenvolvedores;
- Simplicidade;
- Intervalos regulares;

[1] http://www.agilemanifesto.org/

- Auto-avaliação a cada iteração.

O desenvolvimento ágil surgiu a medida que a complexidade das aplicações cresciam e os prazos nunca eram cumpridos, trazendo um estouro enorme nos orçamentos da maioria das empresas. Com isso, surgiu a necessidade de se criar novas formas de gerenciar os membros de uma equipe e agilizar o processo de desenvolvimento. Uma dessas metodologia ficou conhecida como Programação Extrema (*Extreme Programming*).

2.6.1 A Programação Extrema

> *XP is a poweful set of concepts for simplifying the process of software design, development and testing. It is about minimalism and incrementalism, which are specially useful principles when tackling complex problems that require a balance of creativity and discipline.*
> ***Michal A. Cusumano, Professor do MIT.***

A programação extrema é um modelo de desenvolvimento de software baseado nos trabalhos realizados por Kent Beck. Em 1996, ele foi chamado para revisar o projeto C3, controle de pagamento da Chrysler. O projeto foi desenvolvido em Smalltalk e estava com sérios problemas. Kent Beck decidiu começar o projeto do zero pois já não haviam formas viáveis de evoluir com o desenvolvimento, então juntou desenvolvedores experientes e criou uma nova equipe de desenvolvimento seguindo suas práticas e levando-as ao extremo, o resultado foi que em 1997 o projeto já estava em produção.

A metodologia XP segue alguns princípios básicos que são definidos como práticas, valores e princípios.

Valores

São definidos como ações que devem ser tomadas freqüentemente com o objetivo de manter um bom desempenho e harmonia da equipe. Esses valores são:

- **Comunicação**: não perca tempo com problemas já resolvidos. A comunicação cria um senso de união na equipe e de um trabalho cooperativo;
- **Simplicidade**: faça as coisas com simplicidade, evite perder tempo com complexidades desnecessárias.
- **Feedback**: são respostas refletidas por alguém sobre algum assunto, pode ser por exemplo opinião sobre algumas idéias novas para o sistema, idéias sobre implementação, como estão escritos os testes e outros. Mudanças em um projeto são inevitáveis, mas para tê-las é preciso que se tenha um *feedback* sobre o que se deve mudar. O *feedback* é um valor que deve ser sempre praticado, e a resposta deve vir em minutos ou horas mas não dias ou meses. Quanto mais cedo se sabe da mudança mais fácil é a adaptação.
- **Coragem**: enfrente os problemas até achar uma solução. Tenha coragem para dizer para a equipe que o cliente está insatisfeito.
- **Respeito**: respeite os membros da sua equipe e o projeto.

A metodologia XP possibilita a criação de novos valores, dependendo da necessidade da equipe. Valores como segurança e qualidade de vida podem ser importantes. Valores se relacionam, dando suporte um ao outro. Exemplo: aumentando a comunicação, melhora a simplicidade.

Práticas

As práticas em XP são feitas diariamente, freqüêntemente. Uma característica importante é que elas são bem definidas e claras, ou um programador adota uma prática

ou não.

Princípios

Os princípios servem para guiar a equipe às práticas de XP levando em consideração os valores definidos pela equipe. Os princípios dão uma boa idéia sobre o que as práticas de XP querem realizar. Alguns princípios são:

- **Humanos**: o respeito por cada necessidade humada de cada membro da equipe;
- **Econômicos**;
- **Benefício mútuo**: todas as atividades deveriam beneficiar todos os interessados;
- **Similaridades**: soluções podem ser reutilizadas para resolver outros problemas.
- **Diversidade**: trazer para a equipes pessoas com diferentes habilidades, atitudes e especialidades.
- **Reflexão**: reflexões frequentes após cada iteração. Analisar o por quê do sucesso ou da falha.
- **Oportunidade**: transformar os problemas em oportunidades de melhorar o projeto.
- **Redundância**: problemas difíceis devem ser resolvidos de várias maneiras. O custo da redundância é bem menor do que ocorrer um erro grave no projeto
- **Qualidade**: sempre manter o código com clareza e simplicidade.
- **Passos pequenos**: Fazer iterações pequenas. Tentar evitar grandes mudanças em passos grandes, isso pode ser perigo para o desenvolvimento.

As práticas de XP sozinhas não funcionam muito bem, é preciso usá-las integradamente. Uma prática pode ser mais relevante do que outra. As principais práticas da metodologia são:

- **Equipe próxima (*Sit together*)**: Os programadores devem sentar juntos, de preferência numa mesma sala para facilitar a comunicação, sem paredes e sem barreiras no ambiente. A prática possibilita uma integração da equipe;
- **Senso de Equipe**:cada integrante da equipe de desenvolvimento deve trabalhar com sentimento de equipe e sempre mantê-la motivada para enfrentar o projeto juntos;
- **Informações espalhadas pela sala**: As informações como cartões de histórias deverão ser de fácil entendimento e de fácil acesso aos dados, gráficos podem ser usados para mostrar para a equipe e para o cliente o andamento do projeto de forma fácil e simples.
- **Trabalhe com Energia**: Trabalhe apenas o tanto de horas que for produtivo e o apenas o que for suportável, não adianta se esforçar se não haverá produção, é preferível que descanse e volte depois;
- **Programação pareada**: Nessa prática é recomendado que duas pessoas sentem lado a lado em um computador. Manter ambos focados em um mesma atividade cria *brainstorm*, gera discussões sobre soluções de problemas evitando erros banais. Essa prática recomenda uma troca de pares freqüente;
- **Histórias**: Determina que sejam escritas pequenas histórias de funcionalidades que devem ser adicionados ou corrigidas no projeto, é preciso escrever nas histórias um resumo do que deve ser feito e uma estimativa de tempo de implementação. Cada história deve ser colocada em um lugar visível para a equipe;

- **Ciclos semanais**: A semana pode começar com o planejamento dos cartões, revisar as datas de entrega, separar as histórias em tarefas e fazer testes para as novas funcionalidades. Ciclos semanais dão uma freqüência de trabalho para a equipe;
- ***Build* (configuração) de 10 minutos**: É recomendado fazer um *build* do sistema como um todo, rodar todos os testes automaticamente com uma duração de até 10 minutos. O *build* deve ser rápido para não perder a oportunidade do *feedback* e conferir se o sistema está consistente, ou seja, verificar se todos os testes estão rodando sem erros;
- **Integração contínua**: Como o projeto está sendo desenvolvido em uma equipe no qual cada dupla modifica um pedaço do código ou acrescenta novas funcionalidades e testes ao sistema continuamente, é necessário que todos tenham a versão atual do projeto. É recomendável sempre comitar as atualizações o mais breve possível;
- **Fazer os testes antes de programar**: Escrever testes que falham antes de fazer as modificações. Essa prática pode eliminar vários problemas como quebra do escopo, coesão e acoplamento, dúvidas no código e o ritmo de desenvolvimento;
- **Design Incremental**: Pensar no design diariamente para sempre deixá-lo em um ótimo estado para o sistema. Sempre manter em mente o design do projeto tentando melhorá-lo freqüentemente.

2.6.2 A metodologia aplicada no desenvolvimento do projeto

É importante salientar a necessidade da implantação de uma metodologia que fosse compatível com o grupo e com o projeto. Muitas práticas de XP foram adaptadas conforme as necessidades do grupo.

Desenvolvimento do projeto modularmente

Dividir o projeto em módulos pequenos diminuiu a complexidade e facilitou a compreensão e integração de cada componente.

Projeto integralmente testado

Todos as classes desenvolvidas foram exaustivamente testadas, o que facilitou muito a refatoração do código e a compreensão da aplicação como um todo.

Durante todo o projeto, foi almejada a criação dos testes de unidade em conjunto com o desenvolvimento das classes (muitas vezes foram criados os testes antes das classes, com resultados satisfatórios), também foram criados testes para todo novo problema encontrado, objetivando evitar reincidências de erros.

Desenvolvimento Incremental

A aplicação foi desenvolvida de forma incremental, adicionando novas funcionalidades às classes conforme sua necessidade. Toda vez que houve necessidade de refatoração do código, ela foi feita. Essa visão ágil do desenvolvimento impediu a implementação de componentes poucos utilizados. A refatoração do código, aliada ao fato das classes possuírem um grande número de testes de unidade, resultou em um código limpo e simples.

Código limpo e de fácil entendimento

Todo projeto em que mais de uma pessoa desenvolve corre o risco de haver heterogeneidade de código. O projeto utilizou a metodologia de nomenclatura da Sun para código java, além de usar um formater próprio para as classes. Houve sempre a preocupação de fazer algoritmos de fácil entendimento e em classes pequenas.

Testes de Unidade simples e coesos

Todos os testes de unidade foram desenvolvidos para serem entendidos facilmente, e sempre que existe dependência, foram criadas classes mais simples retornando sempre os fatores esperados, permitindo um maior entendimento do projeto através da execução e compreensão dos testes.

Uso de frameworks / componentes existentes

No projeto foram utilizados componentes da comunidade Java que se enquadrassem com as necessidades e filosofia *open-source* do projeto.

Desenvolvimento pareado

Ao criar novas classes complexas ou algoritmos elaborados foi utilizado o desenvolvimento pareado, facilitando a implementação.

Design evolutivo

Utilizando a metodologia de programação orientada à objetos, os comportamentos comuns dos objetos foram exteriorizados em assinaturas de interfaces, auxiliando o uso do polimorfismo.

Não Propriedade do código

O código da aplicação é coletivo: todos os participantes contribuíram com código, evitando ”especialistas”em determinadas partes. Como o código é coletivo as pessoas escrevem mais claramente, e os erros de programação são facilmente notados.

Uso de *patterns*

Sempre que foi possível o uso de *patterns*, estes foram utilizados. O uso desses *patterns* aumentou a qualidade e velocidade do desenvolvimento, assim como o uso do polimorfismo e encapsulamento.

Sofware Livre

Permitir o uso gratuito e transparente resulta em inúmeros benefícios tanto para o software em si quanto para a sociedade. Sua premissa de maior qualidade, confiabilidade e flexibilidade a um custo menor se enquadra perfeitamente com o projeto. Todos os componentes de terceiros utilizados também são softwares livres, permitindo uma maior compreensão e adaptação dos componentes utilizados.

Co-orientação

Juntar a vontade de fazer algo novo pelos graduandos com a experiência dos mestrandos permitiu grande produção aliada a qualidade. As diferentes especialidades dos mestrandos acrescentaram não só ao projeto, mas também aos graduandos.

Capítulo 3

Desenvolvimento do Software

3.1 Modelagem

3.1.1 Abstração dos componentes básicos da análise técnica

A modelagem inicial da biblioteca concentrou-se nos componentes básicos da análise grafista: série temporal e indicadores econômicos. Veja um exemplo na figura 3.1.

Uma série temporal de um ativo é um conjunto de valores do papel (preço de abertura, fechamento, variação real, variação de negócios, etc) extraídos da bolsa com um determinado intervalo (de 15 em 15 minutos ou diariamente por exemplo) dentro do período da série. Na modelagem do projeto cada um desses valores extraídos representam uma instância da class-e **Tick**, que possui um atributo **date** e todos os valores coletados naquela determinada data.

Com a abstração da série temporal desenvolvemos a interface ***TimeSeries***, que contém uma lista de **Ticks** e que dado um índice *i* devolve o respectivo **Tick** através da assinatura *getTick(int i)*.

Um indicador técnico é uma linha temporal que em cada ponto retorna o resultado de uma fórmula matemática utilizando um ou mais valores de um **Tick**. A interface ***Indicator*** através da assinatura *getValue(int i)* obriga as classes que a imple-

Figura 3.1: Série temporal de novembro de 1998 a fevereiro de 2000 da Time Warner (TWX)

mentam a se comportarem como indicadores técnicos. Como qualquer um dos valores de um **Tick** pode ser calculado por um indicador, foram criados indicadores simples (**SimpleIndicators**) que retornam o valor de um determinado atributo do **Tick** de qualquer ponto do ***TimeSeries***. Através do uso da *pattern Decorator*, indicadores recebem como parâmetro os **simpleIndicators**, viabilizando o uso de todos os atributos no cálculo de indicadores e permitindo o uso de indicadores sobrepostos.

```
//Uso de simpleIndicators e de indicadores sobrepostos

/* Instanciação de um SimpleIndicator do preço de fechamento da ação */
Indicator closePrice = new ClosePriceIndicator(timeSeries);
/* Instanciação de um SimpleIndicator do volume da ação*/
Indicator volume = new VolumeIndicator(timeSeries);
/* calcula sma do preço de fechamento */
```

```
Indicator smaClosePrice = new SMAIndicator(closePrice);
/* calcula sma do volume */
Indicator smaVolume = new SMAIndicator(volume);
```

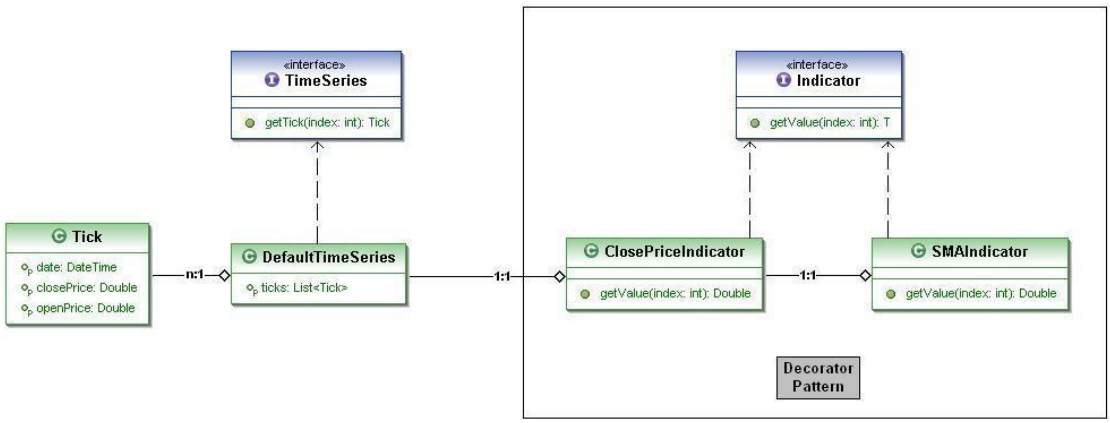


Figura 3.2: UML da modelagem básica, note que nesse exemplo o indicator **SMAIndicator** recebe como parâmetro um simpleIndicator **ClosePriceIndicator**, retornando através do método implementado *getValue(int i)* o valor do SMA aplicado no preço de fechamento do **Tick** i.

3.1.2 Abstração das operações de compra e venda

Desenvolvidos os indicadores, o próximo passo era criar ferramentas que interpretassem os dados resultantes desses indicadores, e através dessa interpretação executassem operações de compra e venda, a figura 3.3 exemplifica as operações.

A figura e a tabela acima demonstram as operações tomadas a partir da estratégia de cruzamento de médias móveis sobre a série temporal *interday* do ativo IBM. Perceba que a estratégia de compra é tomada sempre que o 50-day EMA cruza em sentido ascendente o 50-day SMA e estratégia de venda ocorre quando o cruzamento é descendente. Com o objetivo de simplificar a modelagem, todas as operações são absolutas,

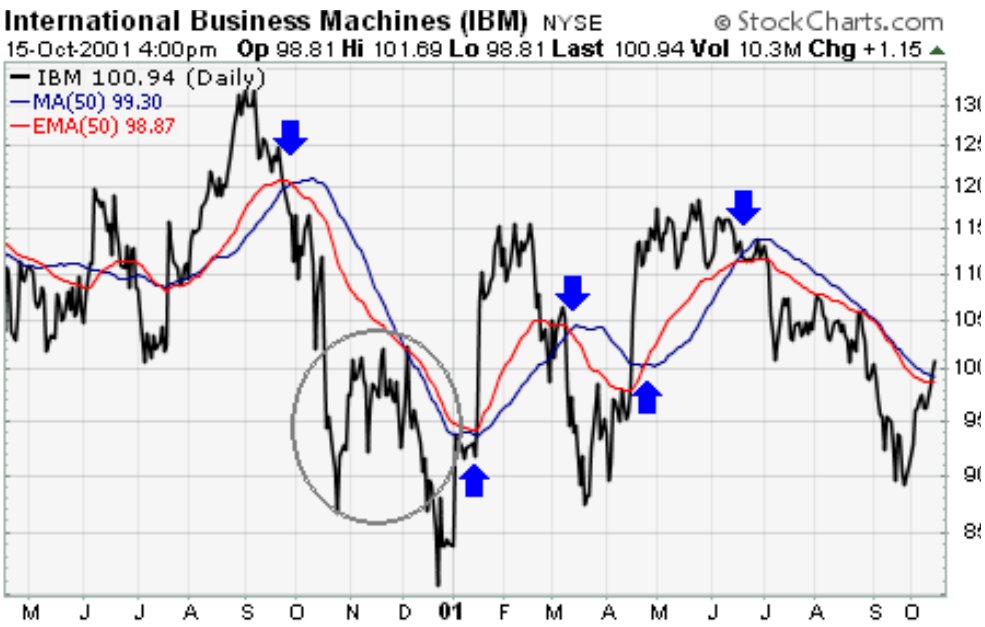


Figura 3.3: Gráfico do preço de fechamento interday do ativo IBM , plotado em conjunto com dois indicadores econômicos: media simples ponderada de 50 dias (50-day SMA) e media exponencial de 50 dias (50-day EMA). As setas azuis correspondem a operações de compra (seta para cima) e venda (seta para baixo). Repare que as operações são feitas quando os dois indicadores se cruzam imagem extraída do site http://www.stockcharts.com/.

ou seja, a compra/venda correspondem sempre a quantidade total de papel disponível para negócio.

No Tail, cada operação é representada através da instanciação de uma classe **Operation**, que possui como atributos um Enum **OperationType** (com os possíveis valores *BUY* e *SELL*) e um índice identificando o **Tick** em que ocorreu a operação. Como todas as operações são absolutas, uma operação de venda pode ser sucedida somente por uma operação de compra e vice-versa. Com o objetivo de abstrair esse comportamento e visando facilitar a análise das operações executadas, criamos a classe **Trade** que é um par de **Operations** distintas.

Uma característica interessante da análise técnica grafista é a interpretação subjetiva dos indicadores: profissionais diferentes eventualmente utilizam indicadores

Tipo da operação	Data da operação
Venda	22/09/00
Compra	15/01/01
Venda	10/03/01
Compra	20/04/01
Venda	15/06/01

Tabela 3.1: data e tipo das operações tomadas na Figura 3.2

iguais adotando estratégias diferentes de compra e venda. Um exemplo é utilizar o SMA como o *trigger* para uma tomada de decisão utilizando outro indicador, ou usar somente o SMA para a tomada de decisão (como visto anteriormente). Visando dar a abertura para o usuário criar a estratégia de decisão que mais condiz com suas necessidades foi criada a interface ***Strategy*** que através das assinaturas *ShouldEnter(int i)* e *ShouldExit(int i)* interpreta indicadores (geralmente carregados no construtor da Strategy) retornando um boolean identificando se devem ser criadas respectivamente operações de entrada ou de saída em um ponto de um ***TimeSeries***. Foram implementadas as estratégias mais comuns para os indicadores mais utilizados.

O próximo passo foi definir a interface ***Runner*** que através da assinatura *run(Strategy strategy, int start, int end)* percorre os **Ticks** de um ***TimeSeries*** a partir de start até end, e utilizando strategy retorna um **List**<**Trade**> com todas as Trades criadas e operadas.

```
/* o atributo a ser analisado do timeSeries é o preço de fechamento */
Indicator<Double> close = new ClosePriceIndicator(timeSeries);
/* criado novo indicador SMA-50 */
Indicator<Double> sma = new SMAIndicator(close, 50);
/* criada strategy que indica que se deve operar quando os indicadores
cruzam */
Strategy strategy = new IndicatorCrossedIndicatorStrategy(sma,close);
/* implementação padrão do runner, analisa uma série histórica */
```

```
Runner runner = new HistoryRunner();
/* resultado do uso da strategy sobre os indicadores */
List<Trade> trades = runner.run(strategy, timeSeries.getBegin(), timeSeries
```

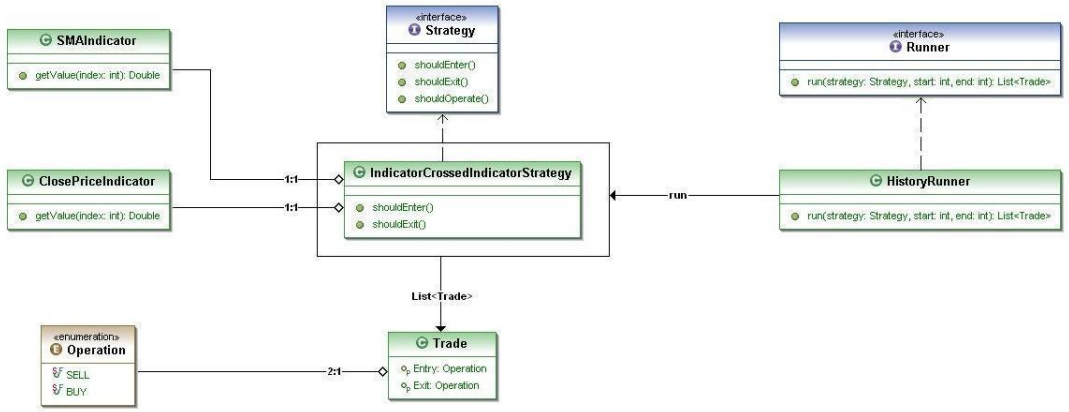


Figura 3.4: Modelagem do uso de um **HistoryRunner** para a interpretação da estratégia de cruzamento dos indicadores **ClosePriceIndicator** e **SMAIndicator**.

Com o uso de polimorfismo, a troca da estratégia de operação do timeSeries é simples. A seguir, um exemplo de como criar uma nova estratégia baseada em uma situação real:

Uma prática comum na análise técnica grafista é a venda baseada no conceito de **Resistência**. Resistência é um nível de preço onde as vendas são fortes o suficiente para inverter o processo de subida, gerando descida do valor do ativo. Resistências são ótimos *triggers* para a venda de ativos. Um exemplo de estratégia é um **ResistanceStrategy** que recebe uma ***Strategy*** no construtor, o valor da resistência e o indicador com os valores do ativo.

```
public class ResistanceStrategy implements Strategy {
  private final Strategy strategy;
  private final Indicator indicator;
```

```
  private double resistance;
  public ResistanceStrategy(Indicator indicator, Strategy strategy,
double resistance) {
    this.strategy = strategy;
    this.resistance = resistance;
    this.indicator = indicator;
  }
}
```

Agora é necessário implementar as assinaturas *ShouldEnter(int i)* e *ShouldExit(int i)* da interface ***Strategy***.

```
public boolean shouldEnter(int index) {
  // Compra quando a strategy comprar
  return strategy.shouldEnter(index);
}
public boolean shouldExit(int index) {
  //Se o valor do papel ultrapassar a resistência vende.
  if (indicator.getValue(index) >= resistance)
    return true;
  // Caso contrário usa strategy para tomar a decisão de venda.
  return strategy.shouldExit(index);
}
```

Agora suponhamos que seja necessário o uso de resistência no exemplo anterior.

```
Indicator<Double> close = new ClosePriceIndicator(timeSeries);
Indicator<Double> sma = new SMAIndicator(close, 50);
```

```
Strategy crossStrategy = new IndicatorCrossedIndicatorStrategy(sma,close);
//Utilização do ResistenceStrategy como decorator do
IndicatorCrossedIndicatorStrategy
Strategy strategy = new ResistanceStrategy(close, crossStrategy, 70);
Runner runner = new HistoryRunner();
//O runner não precisa saber qual a especialização de Strategy
graças ao uso do polimorfismo.
List<Trade> trades = runner.run(strategy, timeSeries.getBegin(),
timeSeries.getEnd());
```

3.1.3 Escolhendo a melhor estratégia de compra e venda

Criadas as estratégias e os indicadores, o próximo passo é determinar qual é a melhor estratégia de compra e venda dado um ativo. O método *Walking Forward* é uma das maneiras mais usadas para comparar estratégias. O algoritmo do método consiste em:

1. Formar hipóteses

   Consiste em criar hipóteses sobre o comportamento do mercado, através da lógica e observação. No Tail, toda a hipótese é representada através de um Strategy.

2. Obter os Dados

   Adquirir a maior quantidade possível de dados históricos que condizem com o ativo a ser estudado. No Tail esse conjunto de dados é representado pelo TimeSeries.

3. Segmentar os Dados

   Dividir os dados em intervalos, tais como anos, meses ou dias. O primeiro intervalo pode ser maior que os outros.

4. Determinar um Critério para avaliação da hipótese

   Determinar o critério de comparação que mais se adequa ao usuário, exemplos podem ser maior lucro total ou menor risco.

5. Escolha da melhor hipótese do primeiro segmento de dados

   Usar cada hipótese sobre o primeiro segmento de dados, determinando através do critério pré-determinado, a hipótese que melhor atende ao critério nesse intervalo.

6. Para cada um dos outros intervalos fazer

   - Utilizar a melhor hipótese do intervalo anterior no intervalo atual, guardando o resultado obtido.
   - Usar cada hipótese sobre o intervalo atual, determinando através do critério pré-determinado, a que melhor atende ao critério nesse intervalo.

7. Guardar os resultados obtidos.

O objetivo do *Walk-Forward* é tentar obter através da linha histórica do ativo qual é a melhor estratégia a se usar na próxima interação. Perceba que não necessariamente a melhor estratégia da rodada anterior será a melhor estratégia dessa rodada. Portanto resultados desse algoritmo devem ser exaustivamente testados e comparados com casos reais comuns. O *Walk-Forward* entretanto desenvolve um ótimo papel na filtragem e elaboração de estratégias de compra e venda.

Segmentando o TimeSeries

O próximo passo seguindo o algoritmo, era criar uma forma de dividir o ***TimeSeries*** em pedaços separados por algum critério. Com esse objetivo foi criada a interface ***TimeSeriesSlicer***, como na figura 3.1.3.

Toda a classe que implementa ***TimeSeriesSlicer*** é um *Wrapper* de uma instância de ***TimeSeries*** que devolve tanto o ***TimeSeries*** todo, através da assinatura *getSeries()*, quanto cada um de seus pedaços (slices) através da assinatura *getSlice(int position)*. O modo como ele separa os slices é determinado pela implementação da classe.

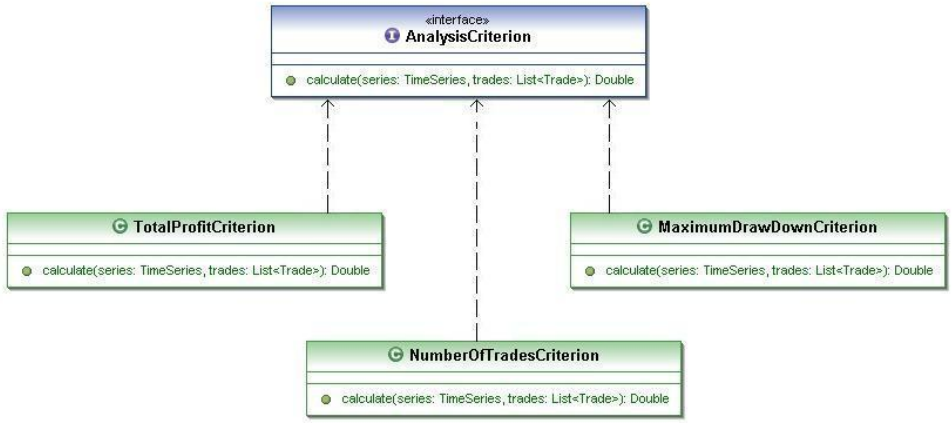
Criando critérios de comparação

Avaliar uma estratégia é quantificar seu resultado atráves de um critério. Com a meta de facilitar a criação de critérios de avaliação foi criada a interface ***AnalysisCriterion***.

A assinatura *calculate()* quantifica o valor de uma lista de trade.A implementação define o modo da quantificação.

Alguns exemplo de ***AnalysisCriterion*** implementados, veja a figura 3.1.3.

- **TotalProfitCriterion**: Retorna o lucro total acumulado;
- **NumberOfTradesCriterion**: Retorna o números de Trades efetuados;
- **MaximumDrawnDownCriterion**: Calcula a maior queda de um ativo.



Escolhendo a melhor Hipótese

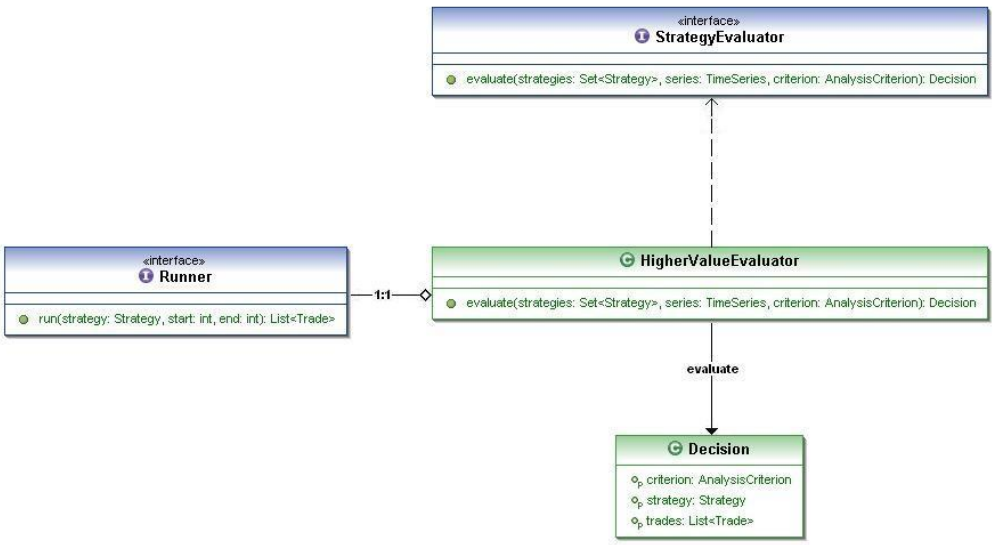
Decidido um conjunto de hipóteses e um critério de comparação, o próximo passo é criar uma ferramenta que utilize um ***Runner*** para rodar as hipóteses e saiba comparar através de um critério pré-determinado qual é a melhor. Para definir esse comportamento foi criada interface ***StrategyEvaluator***.

Uma classe que implementa ***StrategyEvaluator*** recebe um ***Runner*** no construtor e através da chamada do método evaluate retorna um objeto do tipo **Decision**, que possui a melhor ***Strategy*** do período, um **List**<**Trade**> contendo todas as trades geradas pelo runner respectivas a ***Strategy*** escolhida, o ***AnalysisCriterion*** aplicado e a pontuação obtida pela ***Strategy*** escolhida através desse mesmo ***AnalysisCriterion***. Veja a implementação de ***StrategyEvaluator*** na figura 3.1.3.

Um bom exemplo de classe que implementa ***StrategyEvaluator*** é **HigherValueEvaluator** que retorna a ***Strategy*** de maior pontuação referente a um ***AnalysisCriterion*** pré determinado.

Implementando o algoritmo WalkForward

Com todos os passos para a execução do algoritmo criados, agora o algoritmo WalkForward pode ser reescrito utilizando componentes Tail:

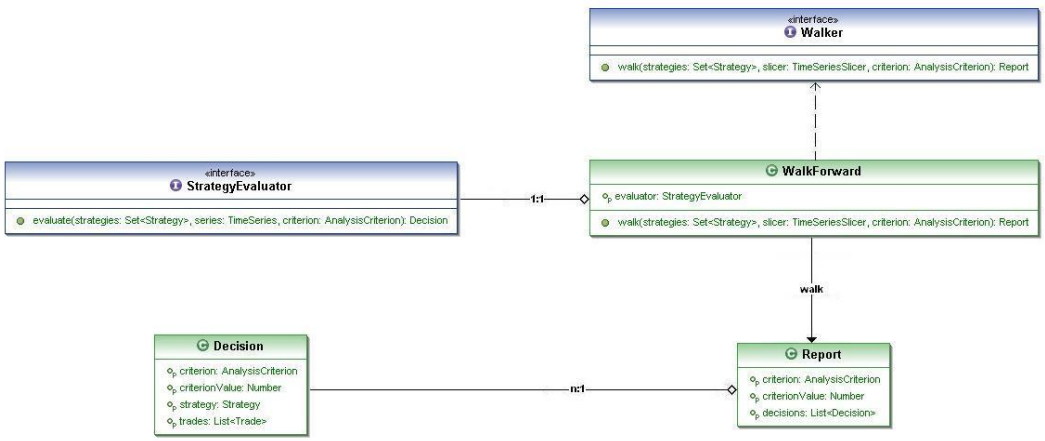


1. Criar um **Set**<**Strategy**> contendo as strategies que serão comparadas.
2. Instanciar um TimeSeries contendo os dados a serem estudados.
3. Instanciar um TimeSeriesSlicer configurado para dividir o TimeSeries em intervalos específicos.
4. Instanciar o AnalysisCriterion mais apropriado ao estudo.
5. Instanciar um StrategyEvaluator, invocar o método evaluate utilizando Set<Strategy>, AnalysisCriterion e o primeiro slice do TimeSeriesSlicer como parâmetros, recebendo como retorno um Decision com a melhor Strategy do primeiro período.
6. Para cada um dos outros intervalos fazer:
   - Utilizar a Strategy contida na Decision anterior no intervalo atual, guardando o resultado obtido.

- Invocar o método evaluate utilizando Set<Strategy>, AnalysisCriterion e o próximo slice do TimeSeriesSlicer como parâmetros, recebendo como retorno um Decision com a melhor Strategy do próximo período.

Armazenar os Decisions, sua pontuação obtida através do AnalysisCriterion e calcular sua pontuação geral.

Para simplificar execução das etapas 5,6 e 7, criamos a classe **WalkForward** que recebe um ***StrategyEvaluator*** no construtor e através da chamada de método *walk* retorna uma instância da classe **Report** que contém um **List**<**Decision**> com as melhores ***Strategy*** de cada período e a pontuação geral alcançada através do ***AnalysisCriterion***. O modelo do WalkForwad está ilustrado na figura 3.1.3.



3.2 Detalhes do Projeto

O objetivo dessa seção é apresentar exemplos de implementações do projeto, assim como suas necessidades, nuancias e dificuldades do desenvolvimento.

3.2.1 Exemplos de indicadores técnicos implementados

Simple Moving Average (SMA)

Cada ponto deste indicador significa a média dos n dias anteriores. O SMA suaviza o gráfico de uma ação, fazendo com que grandes oscilações tendem a ser lineares. Quanto maior o número de dias, mais suavizado é o resultado. Valores da linha temporal abaixo do SMA são bons momentos de compra, pois o valor da série está abaixo do valor médio, analogamente valores acima do SMA são bons momentos de venda. Veja um exemplo na figura 3.5.

Figura 3.5: gráfico do ativo SUNW, no período de abril de 1999 até janeiro de 2001, junto com dois indicadores: SMA de 50 dias e SMA de 200 dias (imagem extraída do site stockcharts.com).

Exponential Moving Average (EMA)

Parecido com o SMA, faz uma média ponderada dos n últimos dias, dando um peso maior aos dias mais recentes.

Bollinger Bands

É constituído por três indicadores: o SMA, o SMA mais duaz vezes o desvio padrão e o SMA menos duas vezes o desvio padrão. Ao desenhar o gráfico deste indicador, é possível perceber que um "tubo"é criado, os sinais são dados quando a ação entra ou sai deste "tubo". Veja um exemplo na figura 3.6.

Figura 3.6: o ativo Wal-Mart Stores junto com o indicador Bollinger (imagem extraída do site stockcharts.com).

Stochastic Oscillator

É composto por dois indicadores, um é uma fórmula matemática que utiliza os valores máximos e mínimos de um determinado período, além do preço de fechamento da ação e o outro é um simples SMA do primeiro. Seus sinais são dados quando uma linha cruza com a outra.

Parabolic SAR

Indicador muito bom quando a ação está em tendência de alta ou de baixa, segue o valor da ação com uma curva parabólica. Seu cálculo utiliza valores de máximo e

mínimo da ação, e seus sinais são dados quando cruza com o valor da ação.Veja um exemplo na figura 3.2.1.

Indicadores auxiliares

Muitos indicadores criados não são utilizados diretamente para dar sinais de compra ou venda, eles existem apenas para ajudar no cálculo de outros. Um exemplo é o Standard Deviation Indicator que foi criado para auxiliar no cálculo do Bollinger Bands.

Como os indicadores fazem seus cálculos com base em outros indicadores, também foram criados alguns que apenas retornam os valores de uma ação, como por exemplo o **ClosePriceIndicator**.

3.2.2 Exemplos de estratégias implementadas

IndicatorCrossedIndicatorStrategy

Estratégia de indicadores como o SMA e EMA, seu conceito é simples: dado dois indicadores, quando o primeiro está em cima, e cruza com o segundo é um sinal de

compra. Quando ocorre o contrário, é um sinal de venda.

PipeEnterStrategy

Criada para indicadores que formam um tubo, como o Bollinger Bands, dá o sinal de venda quando o valor da ação está acima e então cruza com a linha mais alta, entrando no tubo. Quando está abaixo e entra no tubo, é a hora de comprar.

AndStrategy e OrStrategy

Muitas vezes é necessário comprar uma ação quando duas estratégias diferentes sinalizarem, ou quando pelo menos uma sinalizar. **AndStrategyOrStrategy** permitem fazer combinações de estratégias.

```
Strategy s1 = new IndicatorCrossedIndicatorStrategy(sma,closePrice);
Strategy s2 = new IndicatorOverIndicatorStrategy(sar,closePrice);
/* Cria AndStrategy de s1 e s2 */
Strategy s3 = new AndStrategy(s1, s2);
/* Cria AndStrategy de s1 e s2 */
Strategy s4 = s1.and( s2);
```

Min Value Stopper Strategy

Sinaliza venda quando o valor da compra se encontra abaixo de um porcentual pré-determinado.

3.2.3 Critérios implementados

Total Profit

Lucro total da estratégia considerando que no primeiro dia foi investido R$ 1.

Average Profit

Média de Lucro por Tick da estratégia considerando que no primeiro dia foi investido R$ 1.

Brazilian Total Profit

Lucro total extraídos os impostos brasileiros de transação da estratégia considerando que no primeiro dia foi investido R$ 1.

Brazilian Transaction Cost

Gasto total obtido com as transações da estratégia, considerando o preço cobrado pelas transações brasileiras.

BuyAndHold

Lucro total obtido comprando no primeiro **Tick** e vendendo no último.

Maximum Draw Down

Relação entre o maior Pico e a maior Queda do lucro obtido, quanto maior indica mais risco.

Number of Ticks

Número total de ticks que o papel estava comprado.

Number of Trades

Número de negócios feitos.

RewardRiskRatio

Relação entre o o Total Profit e o Maximum Draw Down.

VersusBuyAndHold

Retorna o valor de um critério, mas calculado pelos resultados do BuyAndHold.

3.3 As ferramentas utilizadas

Java CSV

O Java CSV[18] é uma biblioteca open source para leitura e escrita de arquivos de extensão CSV (Comma Separated Values). A ferramenta foi utilizada para a geração de um TimeSeries a partir de um arquivo da base de dados, que geralmente são em CSV.

JUnit

O JUnit[16] é uma biblioteca open source que permite ao programador gerar testes automatizados com grande rapidez. A utilização da ferramenta permitiu o fácil desenvolvimento de testes, assim como sua fácil execução.

JFreeChart

O JFreeChart[15] é uma biblioteca de uso livre para geração de gráficos utilizando a linguagem Java. Permite a criação de *applets* e a exportação para arquivos de imagem PNG.

XStream

O XStream [14] é uma biblioteca open source que possibilita e serialização e deserialização de um objeto em formato XML. Foi utilizada para implementar o *save* e *load* de *reports* e *stock analisys*.

Freemarker

O Freemarker[12] é uma ferramenta open source que possibilita a utilização de templates para geração de arquivos nos formatos HTML, PS,TEX e outros. Foi utilizada na exportação os reports para HTML.

Apache POI

O Apache POI[13] é uma API Java open source que permite a geração e manipulação de diversos arquivos de formato Microsoft tais como XLS e PPT. Foi utilizada na exportação dos reports para XLS.

JRuby

JRuby[21]é uma implementação em Java da linguagem de programação Ruby. Permite a integração de código Ruby com Java, além da transformação de objetos. Foi utilizada para a instanciação de novos indicadores e estratégias durante a execução do programa.

Apache ANT

Apache Ant[20] é uma ferramenta open source em Java para *tracking*, resolução e manipulação de dependências de projeto. Foi utilizada para a criação dos *releases* do projeto.

Eclipse IDE

O Eclipse IDE[17] *open source* implementada em Java e C++ que permite a utilização e implementação de novos compontentes (*plugins*).

ECLEmma

ECLEmma [19] *Plugin* do Eclipse que permite a métrica de cobertura de código.

3.4 O TailS (TAIL Simulator)

O TailS é um programa de análise técnica que utiliza a biblioteca Tail, foi criado com os seguintes objetivos:

- Facilitar o uso do Tail;
- Disponibilizar análise dos dados obtidos pelo Tail;
- Disponibilizar a criação de novas estratégias de compra e venda;
- Interface gráfica agradável;
- Possibilidade de salvar um relatorio obtido para possíveis estudos futuros e comparação dos dados obtidos;
- Exportação para o formato HTML e XLS.

O desenvolvimento das telas utilizou o pattern model-view-control, separando lógica de negócios da lógica de visualização.

3.5 Domain Specific Language (DSL)

DSL é um tipo de linguagem desenvolvida para um particular tipo de problema. O uso de DSL's data desde os primórdios dos PC's, um exemplo de DSL são os Shells Scripts, linguagens desenvolvidas para a operação de sistemas operacionais. Outro exemplo é a linguagem que programas CAD utilizam para receber parâmetros do desenho.

A necessidade de se criar uma DSL para o Tail apareceu pelos seguintes motivos:

- fornecer criação e instanciação de indicadores e estratégias em tempo real através de uma interface gráfica simples e intuitiva;
- público alvo da aplicação não precisa possuir experiência com linguagens de programação;
- a linguagem Java é complicada para não desenvolvedores;
- simplificação na criação de indicadores e estratégias.

A linguagem escolhida para o desenvolvimento da DSL foi o ruby pelos seguintes motivos:

- possuir integração com o Java (através da implementação JRuby);
- ser linguagem de script, facilitando o uso em tempo real;
- ser intuitiva e de fácil entendimento;
- ser orientada a objetos.

Os componentes básicos desenvolvidos para a criação da DSL foram um interpretador de scripts que cria e instancia indicadores e estratégias e uma mini-IDE que facilita a criação desses scripts.

3.5.1 Interpretador de Scripts

O interpretador de scripts ruby foi implementado com uma classe java chamada RubyDSL que recebe no construtor um script em ruby com todos os métodos criados para facilitar o uso da biblioteca. O método *evalIndicator(String code)* recebe o script gerador de indicadores, retornando uma instância Java da classe Indicator. O método *evalStrategies(String code)* recebe o script que instancia ***Strategy***'s Java, e retorna uma Lista das ***Strategy***'s criadas. A implementação dos dois á análoga: o script recebido é interpretado com ajuda dos métodos recebidos no construtor como código ruby e depois portado para instâncias em Java.

```
/* Código DSLTail */
strategies << cross(sma(5), close);
/* Código Java Tail */
List<Strategy> strategies = new ArrayList<Strategy>();
strategies.add( new IndicatorCrossedIndicator(
new SmaIndicator( 5, new ClosePriceIndicator(timeSeries))));
```

3.5.2 Mini-IDE

Além de viabilizar a criação de scripts, é necessário facilitar o desenvolvimento deles. Foi desenvolvido dentro do TailS um mini-IDE que auxilia a escrita de scripts ruby através das seguintes funcionalidades:

- Auto-complete de código ruby e código DSLTail;
- *Sintax-Highlighting* separando invocação de estratégias, indicadores e código ruby;
- *Code-Snippets* fornecendo os scripts padrão mais comuns.

Capítulo 4

Considerações Finais

4.1 Resultados obtidos

4.1.1 Avaliação da Metodologia

A combinação de metodologia ágil, co-orientação e força de vontade foi muito bem sucedida. A metodologia XP permitiu que o projeto fosse desenvolvido gradualmente e com qualidade desde o começo, o uso de testes facilitou a manutenção do código, o acompanhamento de perto de mestrandos especialistas em áreas diferentes permitiu a visualização dos objetivos de modos diferentes.

A modelagem bem estruturada foi fundamental para a evolução projeto, refatorações freqüentes e a criação de novas funcionalidades são formas de manter um projeto sempre bem estruturado. Seguindo este raciocínio foi possível criar um projeto grande em um período curto.

4.1.2 Avaliação dos objetivos

Felizmente todos os objetivos definidos foram satisfatoriamente alcançados. O produto final desse trabalho de conclusão de curso foi a biblioteca Tail e o TailS. o Tail

disponibiliza um arcabouço preciso e flexível para a análise técnica, já o TailS utiliza os recursos da biblioteca para facilitar a análise dos resultados.

4.1.3 Avaliação dos resultados com o uso da aplicação

Alguns resultados interessantes foram obtidos, por exemplo com os dados diários das ações da Petrobrás , investindo R$ 1000,00 em 1991 e utilizando o indicador Parabolic SAR, hoje este valor seria cerca de R$ 1.322.699,00 segundo o critério **TotalProfit**. Uma boa forma de comparar o resultado de uma estratégia é com o BuyAndHold, neste caso, investindo os mesmos R$ 1000,00 e não operando mais, hoje este dinheiro valeria cerca de 12,3 vezes menos segundo o critério **VersusBuyAndHold**. A figura 4.1 mostra os valores.

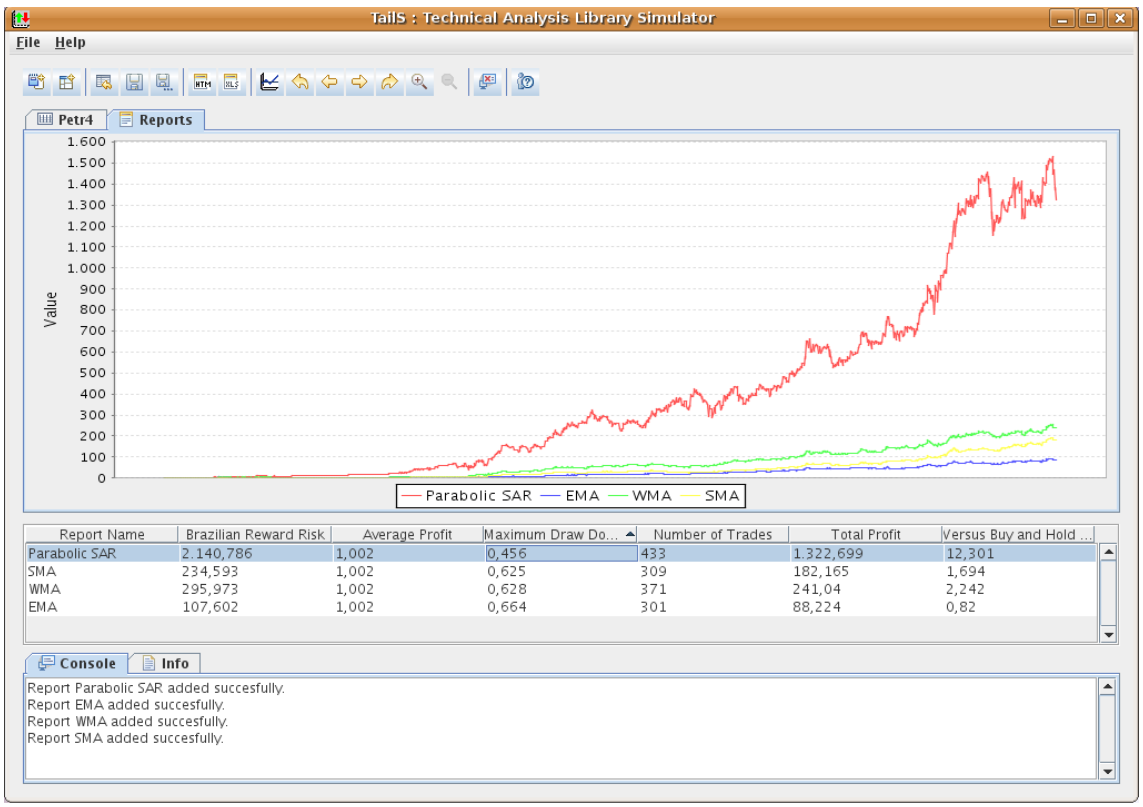


Figura 4.1: Relatório gerado pela ferramenta TailS

Inicialmente, a DSL parecia não ser muito simples, porém com alguns dias de uso é possível perceber o poder desta ferramenta. Alterando apenas alguns comandos, os resultados mudam completamente. A DSL facilitou o estudo dos resultados obtidos, sendo possível fazer pequenas mudanças ou usar uma estratégia totalmente diferente sem alterar diretamente o código.

4.2 Cobertura de teste superior a 90%

O desenvolvimento de testes automatizados foi um dos focos do projeto, é necessário ter garantia que refatorações no código não alterem o resultado final, além de ser importante garantir que indicadores e estratégias estejam se comportando de maneira esperada.

No início o objetivo era manter a cobertura próxima a 100%, mas alguns métodos não fazem sentido serem testados, como por exemplo o *getName()* que apenas retorna o nome de um indicador, ou então classes com mais de um construtor em que um chama o outro com valores padrões, até mesmo classes inteiras como os **SimpleIndicators** que apenas retornam valores de um **Tick**. A cobertura do projeto ficou em 94,2% das classes

4.3 O futuro do projeto

Ainda há muita coisa que deve ser criada no projeto Tail. Atualmente há a depêndencia de arquivos no formato csv que são exportados por outros programas. Futuramente o Tail deverá analizar os dados de uma ação em tempo real. Neste caso, será preciso criar outro ***Runner*** que rode todas as estratégias em tempo real.

A utilização do TailS é direcionada para usuário já acostumado com expressões ou jargões do mercado financeiro. O próximo passo seria simplificar o acesso a essas informações, criando um site que mostre claramente as sugestões de compra e venda,

explicando a razão do bom momento para negociar, além de permitir aos usuários que rodem suas próprias estratégias.

Referências Bibliográficas

[1] Extreme Programming Explained - Embrace Change. Autor: Kent Beck with Cynthia Andres foreword by Erich Gamma - Second Edition

[2] Technical Analysis Of The Financial Market. Autor: John J. Murphy

[3] http://www.bovespa.com.br/

[4] http://stockcharts.com/

[5] http://educaterra.terra.com.br/

[6] http://en.wikipedia.org/

[7] http://www.nelogica.com.br/tutoriais/introtec/introtec03.php

[8] http://www.rmac3.com.br/artigos/teoriadedow.htm

[9] http://comocomprarmeias.marcocarvalho.com/index.php/2007/07/01/teoria-de-dow-a-pedra-fundamental-para-ganhar-dinheiro-na-bolsa/

[10] http://en.wikipedia.org/wiki/Parabolic_SAR

[11] http://www.investopedia.com/

[12] http://freemarker.org/

[13] http://poi.apache.org/

[14] http://xstream.codehaus.org/

[15] http://www.jfree.org/jfreechart/

[16] http://www.junit.org/

[17] http://www.eclipse.org/

[18] http://sourceforge.net/projects/javacsv/

[19] http://www.eclemma.org/

[20] http://ant.apache.org/

[21] http://jruby.codehaus.org/