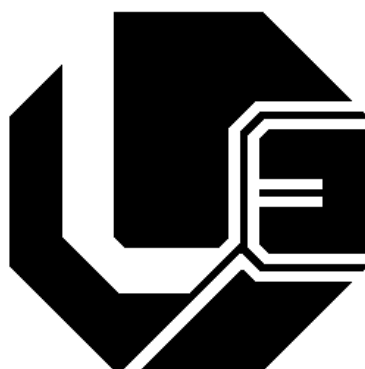


UNIVERSIDADE FEDERAL DE UBERLÂNDIA  
FACULDADE DE ENGENHARIA ELÉTRICA  
PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA



*Framework* para interface e  
gerenciamento computacional de  
sistemas de aquisição de dados

Ailton Luiz Dias Siqueira Junior

Março

2007

UNIVERSIDADE FEDERAL DE UBERLÂNDIA  
FACULDADE DE ENGENHARIA ELÉTRICA  
PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

*Framework* para interface e  
gerenciamento computacional de  
sistemas de aquisição de dados

Ailton Luiz Dias Siqueira Junior

Texto da dissertação apresentada à Universidade Federal de Uberlândia, perante a banca de examinadores abaixo, como parte dos requisitos necessários à obtenção do título de Mestre em Ciências.

Banca examinadora:

Dr. Alcimar Barbosa Soares (UFU) - Orientador

Dr. Alexandre Cardoso (UFU)

Dr. Luciano Vieira Lima (UFU)

Dr. Raimes Moraes (UFSC)

Uberlândia, março de 2007

Dados Internacionais de Catalogação na Publicação (CIP)

---

- S618f     Siqueira Junior, Ailton Luiz Dias, 1980-  
             Framework para interface e gerenciamento computacional de sistemas de aquisição de dados / Ailton Luiz Dias Siqueira Junior. - - 2007.  
             158 f. : il.
- Orientador: Alcimar Barbosa Soares.  
             Dissertação (mestrado) – Universidade Federal de Uberlândia, Programa de Pós-Graduação em Engenharia Elétrica.  
             Inclui bibliografia.
1. Engenharia elétrica - Teses. I. Soares, Alcimar Barbosa. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

CDU: 621.3

---

# **Framework para interface e gerenciamento computacional de sistemas de aquisição de dados**

**Ailton Luiz Dias Siqueira Junior**

Texto da dissertação apresentada à Universidade Federal de Uberlândia como parte dos requisitos para obtenção do título de Mestre em Ciências.

---

**Prof. Alcimar Barbosa Soares, PhD.**  
**Orientador**

*Tudo começa com um sonho...*

# Agradecimentos

A **DEUS** por tudo.

Aos meus pais, **Ailton** e **Moema**, por seus cuidados, carinho e atenção em todos os momentos, e por sempre me apoiarem incondicionalmente.

Aos meus irmãos **Homero** e **Gustavo**, por estarem sempre ao meu lado prontos para ajudar.

À minha noiva **Fábria**, pelo seu carinho, paciência e solidariedade nos momentos mais difíceis.

Ao professor **Alcimar**, que me orientou com paciência, dedicação, companheirismo e amizade durante todas as etapas deste trabalho.

Aos professores **Eduardo** e **Adriano Andrade** pela colaboração durante a revisão deste trabalho.

Ao meu amigo **Júlio**, por sempre indicar o método que faz tudo funcionar.

Ao meu colega e amigo **Cristiano**, pelas boas horas de conversa sobre engenharia, de onde surgiram várias idéias.

Aos meus queridos amigos do BioLab (Laboratório de Engenharia Biomédica e Automática), de ontem, hoje e sempre (por favor, perdoem-me o esquecimento de algum nome): Professor **Adriano Pereira, Alan, Aline, Ângela, Ana Paula, Camila**, professor **Destro, Éder, Eduardo Manço, Eduardo Silva, Fábio, Flávia, Gláucia, Guilherme**, professor **Henrique, Jeovane, Joaquim, José Geraldo, Karina, Kety, Laura, Letícia, Lílían**, professor **Luciano, Marcelle, Rogério, Rodrigo, Rafael, Saulo, Sérgio, Suzi, Thiago Caparelli, Thiago Grasiene, Thiago Finotti, Virgílio e Wilton**. A todos vocês meu muito obrigado!

À todos **professores e funcionários** da Universidade Federal de Uberlândia sempre dispostos a ajudar e a **Marly** pelo apoio de sempre na secretaria da pós-graduação.

Enfim, à **todos** aqueles que de alguma forma contribuíram para este trabalho.

Ao CNPq, pelo suporte financeiro a esta pesquisa.

# Resumo

SIQUEIRA, J. A. L. D. *Framework para interface e gerenciamento computacional de sistemas de aquisição de dados*. Dissertação de Mestrado. Faculdade de Engenharia Elétrica da Universidade Federal de Uberlândia. Uberlândia, 2007.

A maioria dos sistemas de aquisição de dados atuais é baseada em mecanismos de hardware e software dedicados. Tal estratégia dificulta consideravelmente tarefas como a expansão e a atualização do hardware, uma vez que impõem mudanças na estrutura do software, e vice-versa. Neste sentido, este trabalho descreve o desenvolvimento de um *framework* para interfaceamento e gerenciamento de dispositivos de aquisição de dados. Seu principal objetivo é aumentar a modularização do sistema, separando a implementação do hardware do equipamento da implementação do software associado ao mesmo. O *framework* foi desenvolvido com a utilização de técnicas de construção de software adaptativo, como polimorfismo, padrões de projeto, plug-ins e reflexão. Com isso, foi possível criar dispositivos de software capazes de detectar as características específicas de equipamentos conectados a um computador, aumentando assim a compatibilidade entre os aplicativos de software e os equipamentos de aquisição. Desta forma, tal estratégia, permite que um mesmo aplicativo possa ser utilizado para trabalhar com equipamentos distintos, sem que seja necessária qualquer mudança ou ajuste. Os passos para a utilização do *framework* em sistemas reais são apresentados através de exemplos e estudos de caso. Os resultados dos experimentos demonstraram que o *framework* permite a modularização de sistemas, separando, eficientemente, a implementação do hardware do software.

## Palavras-chave

Sistemas de aquisição de dados; Software adaptativo; Polimorfismo; Reflexão computacional; Padrões de projeto; Plug-ins

# Abstract

SIQUEIRA, J. A. L. D. *Framework for computational interfacing and management of data acquisition systems*. Master Dissertation. Faculty of electrical Engineering. University of Uberlândia. Uberlândia, 2007.

Most of the data acquisition systems are based on dedicated hardware and software. Such strategy leads to major difficulties when one needs to expand or update certain features of the hardware, as this generally leads to substantial changes in the software and vice-versa. This work proposes the development of a framework for interfacing and management of data acquisition systems. Its main purpose is to increase system modularization, detaching the development of the hardware from the development of the software. The framework was designed based on adaptive software techniques, such as polymorphism, design patterns, plug-ins and reflection. In so doing, it has been possible to create software devices capable of detecting specific features of the hardware connected to a computer, improving the compatibility between the software and various data acquisition devices. This has allowed a software application to be used with distinct data acquisition devices, without any change or adjustment. The basic steps for using the framework are presented by means of examples and case studies. The results have shown that the framework allows design modularization, separating the design and construction of both hardware and software.

## Keywords

Data acquisition systems; Adaptive software; Polymorphism; Computational reflection; Design patterns; Plug-ins

# Conteúdo

|          |                                                            |           |
|----------|------------------------------------------------------------|-----------|
| <b>1</b> | <b>Introdução</b>                                          | <b>1</b>  |
| 1.1      | Motivação do Trabalho . . . . .                            | 5         |
| 1.2      | Objetivos e metas do trabalho . . . . .                    | 6         |
| 1.3      | Estrutura do Trabalho . . . . .                            | 7         |
| <b>2</b> | <b>Construção de software adaptativo</b>                   | <b>9</b>  |
| 2.1      | Programação orientada a objeto . . . . .                   | 9         |
| 2.1.1    | Objetos e classes . . . . .                                | 10        |
| 2.1.2    | Herança . . . . .                                          | 11        |
| 2.1.3    | Polimorfismo . . . . .                                     | 13        |
| 2.2      | Padrões de projeto . . . . .                               | 15        |
| 2.3      | Reflexão e meta-programação . . . . .                      | 17        |
| 2.3.1    | Reflexão na plataforma <i>.net</i> . . . . .               | 18        |
| 2.4      | <i>Plug-ins</i> . . . . .                                  | 34        |
| <b>3</b> | <b>O desenvolvimento do <i>framework</i></b>               | <b>37</b> |
| 3.1      | Requisitos . . . . .                                       | 38        |
| 3.2      | Modelagem do módulo de controle . . . . .                  | 41        |
| 3.2.1    | A interface com os <i>SADs</i> . . . . .                   | 42        |
| 3.2.2    | O Gerenciador dos <i>plug-ins</i> . . . . .                | 45        |
| 3.3      | A implementação do <i>framework</i> . . . . .              | 47        |
| 3.4      | Desenvolvendo aplicativos com o <i>framework</i> . . . . . | 50        |

|          |                                                           |            |
|----------|-----------------------------------------------------------|------------|
| 3.5      | Criando os <i>plug-ins</i> . . . . .                      | 56         |
| <b>4</b> | <b>Estudos de casos</b>                                   | <b>62</b>  |
| 4.1      | O desenvolvimento do aplicativo <i>BioScope</i> . . . . . | 64         |
| 4.2      | O sistema virtual <i>Gerador Sinal</i> . . . . .          | 67         |
| 4.3      | O equipamento <i>BioDAQ</i> . . . . .                     | 69         |
| 4.4      | <i>Myosystem-Br1 P42</i> . . . . .                        | 79         |
| 4.5      | Atualizações no sistema . . . . .                         | 89         |
| 4.5.1    | Atualizando o <i>BioDAQ</i> . . . . .                     | 90         |
| 4.5.2    | Atualizando o <i>BioScope</i> . . . . .                   | 91         |
| <b>5</b> | <b>Conclusões e trabalhos futuros</b>                     | <b>97</b>  |
| 5.1      | Trabalhos futuros . . . . .                               | 99         |
| <b>A</b> | <b>Tutorial para o desenvolvimento de <i>plug-ins</i></b> | <b>101</b> |
| A.1      | Configurando o ambiente de desenvolvimento . . . . .      | 101        |
| A.2      | Desenvolvendo as classes do <i>plug-in</i> . . . . .      | 104        |
| A.3      | Iniciando e parando o processo de aquisição . . . . .     | 120        |
| A.4      | Armazenando o estado do equipamento . . . . .             | 128        |
| A.5      | Considerações finais . . . . .                            | 133        |
|          | <b>Referências</b>                                        | <b>135</b> |

# Lista de Figuras

|      |                                                                                                                                                                              |    |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 1.1  | Arquitetura típica de um instrumento biomédico . . . . .                                                                                                                     | 4  |
| 1.2  | Arquitetura típica do software de um instrumento biomédico . . . . .                                                                                                         | 4  |
| 2.1  | Exemplo de herança. . . . .                                                                                                                                                  | 12 |
| 2.2  | Diagrama de classes de uma estrutura polimórfica. . . . .                                                                                                                    | 13 |
| 2.3  | Diagrama de classes do padrão estratégia. . . . .                                                                                                                            | 16 |
| 2.4  | Padrão de projeto estratégia aplicado a um aplicativo “gráfico” genérico. .                                                                                                  | 17 |
| 2.5  | Pacotes da plataforma <i>.net</i> utilizados na reflexão. Cada bloco contém o nome do pacote na região superior e uma lista com as classes disponíveis neste pacote. . . . . | 19 |
| 2.6  | Saídas do console geradas pela execução do programa da Listagem 2.2 . . .                                                                                                    | 20 |
| 2.7  | Saídas do console geradas pela execução do programa da Listagem 2.3. . .                                                                                                     | 24 |
| 2.8  | Saídas do console geradas pela execução do programa da Listagem 2.4. . .                                                                                                     | 32 |
| 2.9  | Saída do console gerada pela execução do programa da Listagem 2.6. . . .                                                                                                     | 34 |
| 2.10 | Arquitetura da plataforma Eclipse. . . . .                                                                                                                                   | 36 |
| 3.1  | Diagrama de blocos exibindo os elementos que compõem um SAD típico. .                                                                                                        | 38 |
| 3.2  | Arquitetura de software geralmente empregada para a construção de sistemas de captura e processamento de dados. . . . .                                                      | 39 |
| 3.3  | Arquitetura proposta, baseada em <i>plug-ins</i> . . . . .                                                                                                                   | 40 |
| 3.4  | Diagrama de classes da interface com o SAD . . . . .                                                                                                                         | 42 |
| 3.5  | Diagrama de classes das interfaces com os canais . . . . .                                                                                                                   | 44 |
| 3.6  | Diagrama de atividades do método <i>ProcurarDispositivos</i> . . . . .                                                                                                       | 46 |

|      |                                                                                                                                                                                                                      |    |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 3.7  | Diagrama de classes do <i>framework DAQSystem</i> . . . . .                                                                                                                                                          | 48 |
| 3.8  | Exemplo de uma janela com um controle <i>PropertyGrid</i> . O controle está associado ao objeto que representa a janela, de forma que, ao alterar a propriedade <i>Text</i> , o título da janela é alterado. . . . . | 49 |
| 3.9  | Caixa de combinação mostrando os equipamentos encontrados após executar o código da Listagem 3.1. . . . .                                                                                                            | 51 |
| 3.10 | Exemplo de caixa de combinação mostrando os canais de aquisição disponíveis em um SAD. Os itens da caixa foram gerados através da propriedade <i>Nome</i> de cada canal presente no objeto <i>_daq</i> . . . . .     | 52 |
| 3.11 | <i>PropertyGrid</i> exibindo as propriedades de um SAD (a) e um canal (b). . .                                                                                                                                       | 53 |
| 3.12 | Classe <i>CanalGeradorDAQ</i> . . . . .                                                                                                                                                                              | 58 |
| 3.13 | Classe <i>GeradorDAQ</i> . . . . .                                                                                                                                                                                   | 59 |
| 4.1  | Diagrama esquemático de interconexão de um aplicativo ( <i>BioScope</i> ) a três equipamentos. . . . .                                                                                                               | 62 |
| 4.2  | Exemplo de interface para o aplicativo <i>BioScope</i> . . . . .                                                                                                                                                     | 64 |
| 4.3  | Diagrama de classes do software de visualização de sinais. . . . .                                                                                                                                                   | 65 |
| 4.4  | Aplicativo <i>BioScope</i> em execução. O detalhe mostra que nenhum equipamento foi encontrado. . . . .                                                                                                              | 67 |
| 4.5  | Diagrama de blocos do equipamento virtual acessado pelo <i>plug-in GeradorSinal</i> . . . . .                                                                                                                        | 68 |
| 4.6  | Ativando o equipamento <i>GeradorSinal #1</i> no aplicativo <i>BioScope</i> . . . . .                                                                                                                                | 69 |
| 4.7  | Configurando os canais do equipamento <i>GeradorSinal #1</i> no aplicativo <i>BioScope</i> . . . . .                                                                                                                 | 70 |
| 4.8  | Processo de aquisição no aplicativo <i>BioScope</i> utilizando o <i>plug-in GeradorSinal</i> . . . . .                                                                                                               | 71 |
| 4.9  | Foto da placa de circuito impresso principal do equipamento <i>BioDAQ</i> destacando seus principais componentes. . . . .                                                                                            | 72 |
| 4.10 | Diagrama de blocos mostrando como o <i>plug-in</i> acessa o equipamento. . . .                                                                                                                                       | 73 |
| 4.11 | Diagrama de classes do <i>plug-in BioDAQ</i> . . . . .                                                                                                                                                               | 74 |

|      |                                                                                                                                                                                                                                                                                             |    |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 4.12 | <i>BioScope</i> após a pesquisa por novos equipamentos. O detalhe mostra que o aplicativo detectou a presença do dispositivo <i>BioDAQ</i> . . . . .                                                                                                                                        | 76 |
| 4.13 | <i>BioScope</i> após ativar o equipamento <i>BioDAQ</i> . O detalhe mostra a lista com os canais do equipamento, totalizando 24 canais (Numerados de 0 até 23). . . . .                                                                                                                     | 76 |
| 4.14 | Conexão dos equipamentos durante o experimento com o <i>BioDAQ</i> . . . . .                                                                                                                                                                                                                | 77 |
| 4.15 | Montagem do experimento com o <i>BioDAQ</i> . . . . .                                                                                                                                                                                                                                       | 77 |
| 4.16 | Passos ilustrando a configuração do <i>BioDAQ</i> . (1) Configurando a frequência de aquisição, (2) selecionando o canal 8, (3) ativando o canal, (4) resultado final com o gráfico do canal 8 na região direita. . . . .                                                                   | 78 |
| 4.17 | a) Sinal obtido pelo <i>BioScope</i> através do sistema de aquisição <i>BioDAQ</i> . b) Sinal obtido pelo osciloscópio digital. . . . .                                                                                                                                                     | 80 |
| 4.18 | Comparação entre o sinal obtido com o <i>BioScope</i> e o osciloscópio digital ao variar a frequência (a) e a forma de onda (b) do sinal. . . . .                                                                                                                                           | 81 |
| 4.19 | Myosystem-Br1 P42. . . . .                                                                                                                                                                                                                                                                  | 81 |
| 4.20 | Diagrama de classes do <i>plug-in MyoSystem-Br1 P42</i> . . . . .                                                                                                                                                                                                                           | 82 |
| 4.21 | <i>MyoSystem-Br1 P42</i> na lista de equipamentos do aplicativo <i>BioScope</i> . . . . .                                                                                                                                                                                                   | 84 |
| 4.22 | Painéis apresentando as propriedades do <i>MyoSystem-Br1 P42</i> . . . . .                                                                                                                                                                                                                  | 84 |
| 4.23 | Editor de propriedades do canal no aplicativo <i>BioScope</i> . Em (a) foi selecionado um canal EMG e em (b) foi selecionado um canal auxiliar. . . . .                                                                                                                                     | 85 |
| 4.24 | Montagem do experimento com o equipamento <i>Myosystem-Br1 P42</i> . . . . .                                                                                                                                                                                                                | 85 |
| 4.25 | Passos de configuração do <i>Myosystem-Br1 P42</i> : (1) Configurando a frequência de aquisição em 2khz; (2) Selecionando o canal auxiliar; (3) Configurando o ganho do canal; (4) Ativando o canal. Ao final do processo, o gráfico do canal auxiliar é exibido na região direita. . . . . | 87 |
| 4.26 | Formas de ondas capturadas pelo <i>BioScope</i> através do <i>Myosystem-Br1 P42</i> comparadas com as obtidas pelo osciloscópio: a) Uma senóide pura; b) Variando a frequência da senóide; c) Variando a forma do sinal. . . . .                                                            | 88 |
| 4.27 | Eletrodos fixados no voluntário . . . . .                                                                                                                                                                                                                                                   | 89 |

|      |                                                                                                                                                                                                                                                                                                                                                                                 |     |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 4.28 | Aplicativo <i>BioScope</i> exibindo o Sinal EMG capturado através do equipamento <i>Myosystem-Br1 P42</i> . . . . .                                                                                                                                                                                                                                                             | 90  |
| 4.29 | Classe <i>BioDAQ</i> com a propriedade <i>UsarGatilho</i> . . . . .                                                                                                                                                                                                                                                                                                             | 91  |
| 4.30 | Propriedades do equipamento <i>BioDAQ</i> antes(a) e depois (b) da modificação. Note a mudança na versão do firmware retornada (foi modificada de 1.00 para 1.01) e a nova propriedade, <i>UsarGatilho</i> . . . . .                                                                                                                                                            | 91  |
| 4.31 | Exemplo de painel para seleção de dispositivos utilizados. . . . .                                                                                                                                                                                                                                                                                                              | 92  |
| 4.32 | Diagrama de classes do aplicativo <i>BioScope</i> após a modificação. . . . .                                                                                                                                                                                                                                                                                                   | 92  |
| 4.33 | Aplicativo <i>BioScope</i> sendo executado após a modificação. O seu novo painel apresenta os vários equipamentos encontrados. . . . .                                                                                                                                                                                                                                          | 93  |
| 4.34 | Conectando vários equipamentos ao aplicativo <i>BioScope</i> . . . . .                                                                                                                                                                                                                                                                                                          | 94  |
| 4.35 | Painéis do aplicativo <i>BioScope</i> sendo atualizados de acordo com o equipamento selecionado. Selecionando o equipamento <i>BioDAQ</i> (a), o painel passa a exibir as propriedades únicas desse equipamento (b). Ao selecionar o equipamento <i>MyoSystem Br1 P42</i> (c), o painel se modifica, passando a exibir as propriedades do <i>MyoSystem Br1 P42</i> (d). . . . . | 95  |
| 4.36 | Aplicativo <i>BioScope</i> coletando sinais de dois equipamentos diferentes. . . . .                                                                                                                                                                                                                                                                                            | 96  |
| A.1  | Criando um novo projeto no <i>Visual C# Express</i> . . . . .                                                                                                                                                                                                                                                                                                                   | 102 |
| A.2  | Janela de configuração do projeto. Em destaque, a opção para onde são enviados os arquivos gerados pelo projeto. . . . .                                                                                                                                                                                                                                                        | 103 |
| A.3  | Passos para adicionar as referências do <i>framework</i> ao projeto. . . . .                                                                                                                                                                                                                                                                                                    | 104 |
| A.4  | Diagrama de blocos do equipamento virtual acessado pelo <i>plug-in GeradorSinal</i> . . . . .                                                                                                                                                                                                                                                                                   | 105 |
| A.5  | Passos para criar uma subclasse de <i>DAQ</i> utilizando o gabarito fornecido com o <i>Framework</i> . . . . .                                                                                                                                                                                                                                                                  | 106 |
| A.6  | Aplicativo <i>BioScope</i> exibindo as propriedades do equipamento <i>GeradorSinal</i> , ainda incompleto. . . . .                                                                                                                                                                                                                                                              | 114 |

- A.7 Aplicativo *BioScope* exibindo as propriedades do equipamento *GeradorSinal*. O painel exibe: (a) A propriedade *FreqAquisicao*, adicionada à classe *GeradorSinal*. (b) A propriedade *Ganho*, adicionada à classe *GeradorCanal*. 119
- A.8 Aplicativo *BioScope* exibindo os sinais coletados através do *plug-in GeradorSinal*. . . . . 128

# Lista de Tabelas

|     |                                                                             |    |
|-----|-----------------------------------------------------------------------------|----|
| 2.1 | Propriedades da classe <i>Type</i> . . . . .                                | 21 |
| 2.2 | Métodos da classe <i>Type</i> e seus respectivos tipos de retorno . . . . . | 21 |

# Lista de Listagens

|     |                                                                                                                           |     |
|-----|---------------------------------------------------------------------------------------------------------------------------|-----|
| 2.1 | Desenhando diversas figuras utilizando polimorfismo em C#.                                                                | 14  |
| 2.2 | Obtendo as referências de <i>Type</i> no C#.                                                                              | 19  |
| 2.3 | Utilizando os objetos “Info” no C#.                                                                                       | 22  |
| 2.4 | Carregando uma classe dinamicamente no C#.                                                                                | 25  |
| 2.5 | Associando um atributo a uma classe.                                                                                      | 33  |
| 2.6 | Listando os atributos de uma classe.                                                                                      | 33  |
| 3.1 | Código para listar os equipamentos conectados ao sistema.                                                                 | 50  |
| 3.2 | Criando um objeto DAQ.                                                                                                    | 51  |
| 3.3 | Utilizando a propriedade <i>Canal</i> .                                                                                   | 51  |
| 3.4 | Modificando um <i>Canal</i> .                                                                                             | 52  |
| 3.5 | Modificando um <i>Canal</i> .                                                                                             | 53  |
| 3.6 | Iniciando a aquisição de dados.                                                                                           | 54  |
| 3.7 | Obtendo as amostras do SAD.                                                                                               | 55  |
| 3.8 | Parando a aquisição de dados.                                                                                             | 55  |
| A.1 | Definição da classe <i>GeradorSinal</i> .                                                                                 | 107 |
| A.2 | Método <i>Scan</i> gerado pelo gabarito <i>Novo DAQ</i> .                                                                 | 107 |
| A.3 | Exemplo de implementação do método <i>Scan</i> , procurando por equipamentos conectados à portas seriais.                 | 108 |
| A.4 | Código do método <i>Scan</i> do equipamento <i>GeradorSinal</i> .                                                         | 110 |
| A.5 | Código do método <i>Scan</i> do equipamento <i>GeradorSinal</i> .                                                         | 110 |
| A.6 | Definindo os valores iniciais para os membros herdados da classe <i>DAQ</i> no construtor da classe <i>GeradorSinal</i> . | 111 |

|      |                                                                                                                                          |     |
|------|------------------------------------------------------------------------------------------------------------------------------------------|-----|
| A.7  | Código do construtor padrão da classe <i>GeradorCanal</i> criado pelo gabarito.                                                          | 113 |
| A.8  | Código do método <i>Clone</i> gerado pelo gabarito <i>Novo Canal</i> .                                                                   | 113 |
| A.9  | Código da propriedade <i>FreqAquisicao</i> , adicionado à classe <i>GeradorSinal</i> , para fornecer acesso aos recursos do equipamento. | 115 |
| A.10 | Código da propriedade <i>Ganho</i> incluído na classe <i>GeradorCanal</i> .                                                              | 117 |
| A.11 | Código do método <i>Clone</i> , com o código para a cópia do campo <i>_ganho</i> na linha 11.                                            | 117 |
| A.12 | Código do método <i>AtualizaCanais</i> da classe <i>GeradorSinal</i> gerado pelo gabarito.                                               | 118 |
| A.13 | Código do método <i>AtualizaCanais</i> da classe <i>GeradorSinal</i> gerado pelo gabarito.                                               | 119 |
| A.14 | Código do método <i>IniciarAquisicao</i> da classe <i>GeradorSinal</i> gerado pelo gabarito.                                             | 120 |
| A.15 | Exemplo de implementação do código do método <i>IniciarAquisicao</i> utilizando a comunicação serial.                                    | 121 |
| A.16 | Laço criado no método <i>ThreadAquisicao</i> para executar o processo de aquisição.                                                      | 123 |
| A.17 | Implementação do método <i>ThreadAquisicao</i> no <i>plug-in GeradorSinal</i> .                                                          | 125 |
| A.18 | Código do método <i>PararAquisicao</i> .                                                                                                 | 127 |
| A.19 | Código do método <i>GetObjectData</i> da classe <i>GeradorSinal</i> gerado pelo gabarito.                                                | 129 |
| A.20 | Implementação do método <i>GetObjectData</i> da classe <i>GeradorSinal</i> .                                                             | 130 |
| A.21 | Construtor utilizado na serialização da classe <i>GeradorSinal</i> gerado pelo gabarito.                                                 | 131 |
| A.22 | Implementação do construtor utilizado na serialização da classe <i>GeradorSinal</i> .                                                    | 132 |
| A.23 | Implementação da serialização na classe <i>GeradorCanal</i> .                                                                            | 133 |

# Lista de Acrônimos

**ECG** - Eletrocardiógrafo ou Eletrocardiografia

**EEG** - Eletroencefalógrafo ou Eletroencefalografia

**EMG** - Eletromiógrafo ou Eletromiografia

**SAD** - Sistema de Aquisição de Dados

**USB** - Universal Serial Bus

**POO** - Programação Orientada a Objeto

**UML** - Unified Modeling Language

**IDE** - Integrated Development Environment ou Ambiente de desenvolvimento integrado

**AD** - Analógico-Digital

# Capítulo 1

## Introdução

Luigi Galvani demonstrou, no século dezanove, a natureza bioelétrica do corpo humano (GESELOWITZ; GESELOWITZ, 1999). Galvani, durante um de seus experimentos, observou que a eletricidade era capaz de realizar a contração de um músculo retirado da perna de um sapo e afirmou ter descoberto a existência da eletricidade animal.

A partir deste momento, diversos cientistas dedicaram-se aos estudos da relação entre a eletricidade e os processos fisiológicos. O discípulo de Galvani, Carlo Matteucci, determinou que organismos vivos produzem pequenas correntes elétricas e, conseqüentemente, pequenos campos magnéticos. Essa descoberta atraiu a atenção de diversos cientistas que iniciaram uma série de esforços no sentido de buscar explicações para estes fenômenos. As pesquisas evoluíram até que, em 1853, Hermann von Helmholtz publicou suas descobertas sobre diversos aspectos físicos da condução volumétrica explicando alguns fenômenos elétricos que ocorriam no corpo humano (GESELOWITZ; GESELOWITZ, 1999).

A partir destas investigações, surgiu o interesse no desenvolvimento de novos instrumentos capazes de capturar e armazenar sinais elétricos emanados do corpo humano. O coração por exemplo, é um músculo bastante ativo e apresenta uma atividade elétrica bastante significativa. Em 1878, Theodore Wilhelm criou o eletrocardiógrafo (ECG), um instrumento capaz de detectar a atividade elétrica produzida pelo coração. Este primeiro instrumento era baseado em um galvanômetro de corda e não permitia o registro da atividade. Dez anos depois, Augustus Desir Waller foi capaz de realizar o primeiro

registro de ECG utilizando uma pena inscritora e um cilindro de papel que movia-se a uma velocidade constante. Mas quem realmente alavancou as pesquisas sobre o ECG, foi Willem Einthoven que, em 1900, publicou seu trabalho no artigo intitulado “On the Normal Human Electrocardiogram and on the Capillary – Electrometer Examination of Several Cardiac Patients” associando padrões de sinais ECG com diversas patologias.

Durante a década de 30, o psiquiatra alemão Hans Berger publicou uma série de estudos sobre a atividade elétrica do córtex cerebral humano, captado através de um novo instrumento denominado eletroencefalógrafo (EEG). Por seu trabalho, Hans Berger é hoje reconhecido como o pai da eletroencefalografia (KAISER, 2005).

Já no final da segunda guerra mundial, devido ao grande número de pessoas feridas nos combates, surgiu a necessidade de entender melhor os músculos e os movimentos humanos para a auxiliar na reabilitação destes combatentes. Foi neste contexto que o ortopedista Vern Inman começou a estudar a caminhada humana e a captar sinais elétricos provindos da atividade muscular através de um instrumento denominado eletromiográfico (EMG). O Dr. Inman influenciou vários de seus colegas que continuaram suas pesquisas desenvolvendo novos instrumentos, metodologias e técnicas para captação de sinais eletromiográficos (SUTHERLAND, 2001).

Neste mesmo cenário pós-guerra, nos laboratórios da AT&T, os pesquisadores J. Bardeen, W. Brattain e W. Shockley trabalhavam com dispositivos eletrônicos baseados em semicondutores. Em 1947, eles demonstraram um novo dispositivo semicondutor, o transistor, que anos mais tarde revolucionaria a eletrônica reduzindo o tamanho e o consumo elétrico de diversos equipamentos tornando-se a base da eletrônica moderna (MORTON, 1999).

Logo após a invenção do transistor, diversas empresas desenvolveram os primeiros circuitos integrados, que possibilitavam a condensação de complexos circuitos eletrônicos em pequenas pastilhas de material semicondutor. Os engenheiros passaram a sintetizar cada vez mais circuitos em uma única pastilha e logo surgiu a idéia de desenvolver um microprocessador em um único chip. Em 1971, a Intel disponibilizou no mercado seu chip 4004, o primeiro microprocessador encapsulado em uma única pastilha (ABBATE, 1999).

Nos anos 70, diversos hobistas da computação começaram a construir computadores em suas residências. Um dos primeiros microcomputadores foi o Altair 8800, desenvolvido por uma pequena empresa chamada Micro Instrumentation Telemetry Systems (MITS). O Altair 8800 conseguiu algum sucesso na época, de modo que logo surgiram novos microcomputadores, dentre eles o Apple II, que foram ganhando espaço no mercado devido ao seu baixo preço e tamanho reduzido. A IBM, ao criar o seu microcomputador baseado numa arquitetura aberta, popularizou o termo computador pessoal (PC) tornando a tecnologia acessível a um grande número de pessoas.

O aumento da disponibilidade de PCs gerou uma evolução da informática e também, influenciou na instrumentação médica. Cada vez mais, os instrumentos médicos passaram a ser conectados a computadores para um processamento sistemático de dados, informação e conhecimento. Com isso, foi criada uma nova área de pesquisa: a informática médica. A informática médica é hoje composta por diversas sub-áreas como: sistemas de informação para área de saúde, documentação médica, processamento de sinais médicos, processamento de imagens médicas, sistemas de suporte a diagnóstico e terapias e bioinformática molecular (HAUX, 1997).

Um dos desafios da informática médica está no desenvolvimento de instrumentos biomédicos. A Figura 1.1 apresenta um exemplo de arquitetura típica de um instrumento biomédico, no caso, um eletromiógrafo. Este instrumento é composto por sensores (eletrodos) responsáveis por captar os potenciais de ação na superfície do corpo e transformá-los em uma grandeza elétrica mensurável. Os sensores são conectados a um equipamento dotado de uma unidade de condicionamento que trata o sinal proveniente do sensor em etapas de amplificação, para adequá-lo ao sistema de aquisição de dados (SAD), e de filtragem para eliminar ruídos presentes no ambiente. O sistema de aquisição de dados é um dispositivo eletrônico que converte sinais elétricos em valores numéricos binários, para serem processados e armazenados nos computadores. Um software comunica-se com o sistema de aquisição de dados e processa os sinais captados para extrair informações diversas.

A Figura 1.2 mostra a arquitetura típica de um software que acompanha um instru-



Figura 1.1: Arquitetura típica de um instrumento biomédico

mento biomédico. Nela, pode-se verificar a presença de um módulo de processamento que irá extrair as informações do sinal captado pelo instrumento, este módulo comunica-se com o módulo de controle do SAD e obtém o sinal através de uma bloco de memória. O módulo de controle tem a função de atender os comandos do módulo de processamento e preencher o buffer circular com os valores de cada ponto do sinal captado pelo SAD. Para realizar estas operações, o módulo de controle comunica-se com o SAD utilizando as rotinas do sistema operacional para acesso ao hardware.

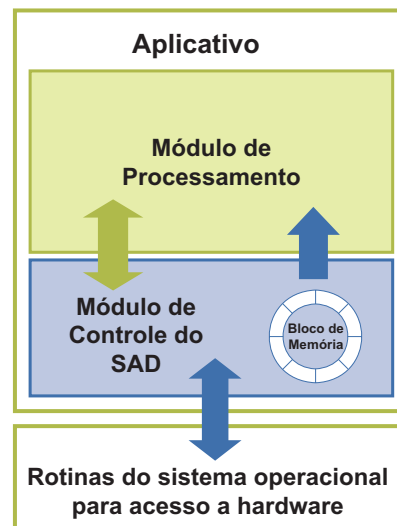


Figura 1.2: Arquitetura típica do software de um instrumento biomédico

Esta arquitetura de software, apesar de amplamente utilizada em diversos projetos de

instrumentos biomédicos, apresenta algumas limitações que serão apresentados no decorrer deste trabalho. Como alternativa, apresenta-se também, uma proposta de arquitetura para minimizar aqueles problemas.

## 1.1 Motivação do Trabalho

No Laboratório de Engenharia Biomédica e Automática da Faculdade de Engenharia Elétrica pertencente a Universidade Federal de Uberlândia, durante seus oito anos de existência, foram desenvolvidos diversos instrumentos biomédicos para auxiliar às pesquisas ali realizadas. Dentre estes sistemas, pode-se citar alguns exemplos como o eletromiógrafo Myosystem-Br (ANDRADE, 2000) ([www.datahominis.com.br](http://www.datahominis.com.br)), uma plataforma de força uniaxial (NAVES, 2001) e o sistema de biofeedback Selfcontrol (SÁ, 2004). Os aplicativos destes instrumentos foram construídos seguindo uma arquitetura semelhante a apresentada na Figura 1.2. Apesar de todos estes sistemas operarem de maneira satisfatória, ao final do seu projeto, apresentaram algumas limitações não observadas inicialmente.

O Myosystem-Br foi um dos primeiros sistemas e o que mais evoluiu com o passar do tempo. Atualmente, ele possui quatro versões diferentes, onde diversas unidades do sistema foram alteradas de maneira significativa. Nas suas primeiras versões, o SAD utilizado era uma placa de aquisição padrão PCI proprietária. Nas versões mais recentes, todo o hardware foi redesenhado e o SAD foi substituído por um novo modelo com tecnologia USB desenvolvido internamente. Cada passo dessa evolução gerou um impacto no software do sistema, resultando em diversas alterações no mesmo, principalmente no módulo de controle do SAD. Entretanto, algumas mudanças foram tão complexas que exigiram modificações também nos módulos de processamento gerando um gasto de tempo e recursos. Além disso, hoje o Myosystem-Br possui quatro softwares diferentes, um para cada versão do sistema, o que dificulta bastante a manutenção e a expansão das funcionalidades desse software.

Com a plataforma de força ocorreu um caso semelhante ao Myosystem-Br. A con-

cepção inicial do sistema foi realizada utilizando uma placa de aquisição de dados proprietária, que algum tempo depois, foi danificada. Como o laboratório já possuía um protótipo do novo módulo de aquisição USB, este módulo substituiu a placa de aquisição, mas foi necessário modificar o software que acompanhava a plataforma de força.

O sistema de biofeedback Selfcontrol apresenta uma característica incomum na instrumentação biomédica que é a necessidade de obtenção de sinais relativos a diversos parâmetros fisiológicos diferentes como, temperatura, oximetria, frequência respiratória, sinais EEG e sinais EMG. Para isso, foi construído um equipamento único com diversos canais, cada canal preparado para um tipo de sinal, exigindo um hardware complexo para atender os requisitos do sistema Selfcontrol.

Ao observar a história desses equipamentos, percebe-se algumas necessidades que não são tão aparentes durante o desenvolvimento do projeto. Assim algumas questões foram levantadas: Existe alguma maneira de minimizar o impacto das alterações do sistema, como aquelas ocorridas com o Myosystem-Br e a plataforma de força? O Myosystem-Br realmente precisa de uma versão do software para cada versão do equipamento? O software da plataforma de força de alguma forma poderia ser utilizado com outra plataforma, diferente da desenvolvida em conjunto com o software? O sistema Selfcontrol realmente necessita ser um equipamento único com todos os canais, ou poderia ser conectado a vários equipamentos diferentes e já existentes para aquisição de cada tipo de sinal?

Tendo em vista as questões acima e observando a maneira como estes instrumentos trabalham, percebe-se a necessidade de alteração na arquitetura do software, principalmente no módulo de controle do SAD, para contornar algumas das limitações descritas anteriormente.

## 1.2 Objetivos e metas do trabalho

Este trabalho tem como objetivo desenvolver um *framework* para interfaceamento de equipamentos dedicados à aquisição sinais, visando diminuir o impacto de alterações do sistema e permitir a conexão de equipamentos diferentes a um mesmo software, de maneira

simples e dinâmica.

O *framework* será implementado na forma de uma biblioteca de software e avaliada em uma série de estudos de caso.

Para desenvolver esta arquitetura, as seguintes metas foram estabelecidas:

- Desenvolvimento do *framework* para aquisição de dados
- Desenvolvimento de um aplicativo baseado no *framework*
- Conexão de equipamentos reais com *framework*
- Avaliação do sistema desenvolvido.

## 1.3 Estrutura do Trabalho

Este capítulo inicial abordou o surgimento e a evolução da instrumentação biomédica, apresentando a arquitetura típica de alguns instrumentos. Mostra ainda, alguns problemas dessas arquitetura.

O Capítulo 2 apresenta algumas técnicas para construção de sistema adaptativos que podem ser utilizadas para amenizar os problemas mencionados. Nele são apresentadas algumas ferramentas de engenharia de software, além de recursos presentes na plataformas de programação atuais, que permitem construir softwares capazes de modificar seu comportamento de acordo com o contexto de execução do sistema.

Em seguida, o Capítulo 3 apresenta a proposta de um *framework* para aquisição de dados. Apresenta também as várias etapas do desenvolvimento, o protótipo desenvolvido e a documentação necessária para a utilização do mesmo.

O Capítulo 4, apresenta a avaliação do sistema, onde o *framework* é utilizado no desenvolvimento de um aplicativo de visualização de sinais e conectado a diversos equipamentos diferentes. Apresenta também alguns experimentos para avaliar as capacidades do *framework*.

Finalmente, o Capítulo 5, apresenta as conclusões obtidas a partir da análise dos resultados. São também apresentadas algumas sugestões para trabalhos futuros que poderão ser realizados para estender as contribuições fornecidas por este documento.

## Capítulo 2

# Construção de software adaptativo

Um dos objetivos da engenharia de software é a construção de softwares que sejam facilmente modificados e estendidos. Diversas técnicas foram, e ainda são, desenvolvidas, buscando atingir um nível de modularização tal que uma mudança em uma decisão do projeto gere efeitos isolados em apenas uma pequena área do programa (GRAY, 2004).

De forma geral, pode-se projetar softwares que sejam facilmente modificáveis durante a fase de desenvolvimento e que permitam alterações durante a execução. No primeiro caso, são utilizados recursos disponíveis na programação orientada a objeto, como herança e polimorfismo, além de soluções estruturais como as apresentadas em diversos padrões de projeto. Para a construção de sistemas que se adaptem em tempo de execução, recursos como a reflexão e arquiteturas baseadas em plug-ins são utilizados. Este capítulo discute algumas destas técnicas que serão utilizadas no desenvolvimento do “*framework*” central deste trabalho.

### 2.1 Programação orientada a objeto

As bases da Programação Orientada a Objeto (POO) foram definidas na década de 60. A popularidade de linguagens baseadas nesse paradigma tem crescido nos dias atuais, e isso ocorre devido aos seus recursos que permitem um desenvolvimento mais rápido através do compartilhamento de componentes e reutilização de código (KOLDITZ; BAUER,

2004).

### 2.1.1 Objetos e classes

O principal elemento da programação orientada a objeto é, como o próprio nome diz, o objeto. Um objeto é um modelo de uma entidade do mundo real (como por exemplo um carro, uma pessoa, ou livro), incluindo entidades mais abstratas como um som ou um sinal elétrico (VINCENZI, 2004). A idéia é que os objetos funcionem como uma caixa preta, dessa forma é possível utilizar os objetos sem conhecer os seus mecanismos internos. As pessoas fazem isso todos os dias quando, por exemplo, dirigem um carro. Elas conhecem a interface (o volante e os pedais), mas não sabem todos os detalhes de como o motor movimenta o carro. Isto as permite dirigir carros diferentes, com motores diferentes, uma vez que a interface para todos os carros é similar.

Ao escrever programas utilizando a POO, escreve-se classes e não objetos. As classes são representações genéricas de vários objetos semelhantes, uma espécie de gabarito, enquanto que os objetos são as representações de entidades específicas. Por exemplo, a classe *torre* descreve os atributos<sup>1</sup> presentes todas as torres existentes no planeta, mas o objeto *Torre Eiffel* modela uma torre específica com valores dos atributos da Torre Eiffel, como a altura de 317 metros.

As classes são compostas por atributos e métodos. Os atributos mantêm os dados utilizados pelo modelo, como por exemplo, a *altura* da *torre* do exemplo anterior. Os métodos definem as funcionalidades dos objetos, ou seja, as ações que estes objetos podem executar. Por exemplo, o método *Acelerar* de um objeto *Carro* realiza a ação de aumentar a velocidade do veículo.

Para que os objetos consigam ser vistos como uma caixa preta, os atributos e os métodos devem possuir um controle de acesso privado. Os atributos e métodos públicos da classe podem ser acessados por outras classes e fornecem sua interface. Já os atributos e

---

<sup>1</sup>Na plataforma *.net* os atributos das classes costumam ser chamados de campos, já o termo atributo recebe outro significado, sendo utilizado para definir metadados que podem ser adicionados a diversos elementos da plataforma, e acessados por reflexão.

métodos privados, só podem ser acessados pela própria classe, de modo a serem utilizados em seu mecanismo interno, e que as outras classes não precisam ter conhecimento.

Ao executar um programa desenvolvido com a POO, um objeto é criado a partir de uma classe num processo chamado instanciação. Este objeto é capaz de instanciar outros objetos e executar seus métodos de forma que a semântica do programa é gerada pela interação entre os diversos objetos.

### 2.1.2 Herança

As linguagens orientadas a objeto possuem uma boa capacidade de reutilização de código, sendo que um dos recursos que contribuem para essa característica é a herança. A herança é uma forma de reutilização de código na qual uma nova classe é derivada a partir de outras classes bases, absorvendo seus atributos e métodos e adicionando novos elementos (DEITEL et al., 2001).

Para entender melhor como o recurso de herança funciona, considere o seguinte exemplo: a Figura 2.1 mostra um diagrama de classes de acordo com as especificações da *Unified Modeling Language* (UML). Neste diagrama, cada classe é representada por um bloco dividido em três partes, a parte superior contém o nome da classe, a intermediária contém os atributos da classe e a parte inferior, os métodos da classe. O sinal de “+” ou “-” no início do texto dos atributos e dos métodos indicam a visibilidade do membro da classe, o sinal “-” indica que o membro tem acessibilidade privada, enquanto que o sinal “+” indica que o membro tem acessibilidade pública. Os atributos são apresentados na forma *nome:tipo*, onde o nome é um identificador para a variável contida no objeto, o tipo é um dos formatos básicos da linguagem ou uma classe. Os métodos são representados pelo seu nome, seguidos de seus argumentos entre parênteses e o tipo de retorno após os dois pontos.

A figura apresenta três classes com uma seta conectando algumas delas. Esta seta modela a herança, portanto, a classe *Circulo* herda da classe *Ponto* e a classe *Cilindro* herda da classe *Circulo*. Dessa forma, de acordo com o conceito de herança, a classe *Circulo* possui sete métodos, os três representados no seu bloco, adicionados dos quatro

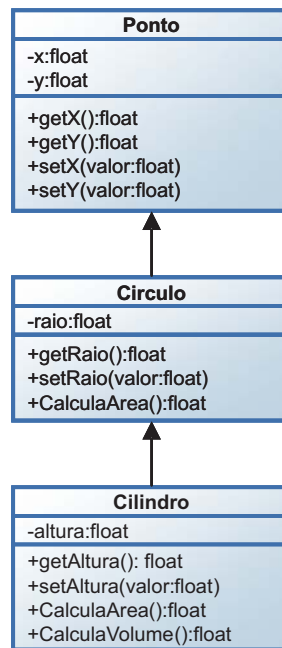


Figura 2.1: Exemplo de herança.

que herdou da classe *Ponto*. A classe *Circulo* também possui os atributos  $x$  e  $y$  da classe *Ponto*, entretanto não possui acesso direto a eles, pois são atributos privados. De maneira análoga, a classe *Cilindro* possui todos os atributos e métodos da classe *Circulo*, inclusive aqueles herdados da classe *Ponto*.

Quando uma classe herda de outra, ela também pode modificar a implementação de um método existente para adequá-lo as suas necessidades. Isso pode ser observado no bloco da classe *Cilindro* que descreve o método *CalculaArea*, já existente na classe *Circulo*, da qual a classe *Cilindro* herda. Neste caso, a classe *Cilindro* fornece uma nova implementação para o método, sobrescrevendo-o com um novo algoritmo, já que a maneira de calcular a área muda com o tipo da figura.

A herança facilita a vida do programador, economizando tempo durante o desenvolvimento ao evitar que ele reescreva código. Ela também facilita a manutenção do código, pois as correções de erros na classe base são automaticamente propagadas para as classes derivadas.

### 2.1.3 Polimorfismo

O recurso de polimorfismo, presente na POO, permite a escrita de programas que manipulam uma grande variedade de classes de maneira genérica, facilitando a adição de novas funcionalidades ao sistema. Através do polimorfismo, é possível projetar e implementar sistemas que são facilmente extensíveis (DEITEL et al., 2001).

A melhor maneira para entender o polimorfismo é através de um exemplo. Imagine que um aplicativo necessite desenhar diversas figuras em uma área de desenho. As figuras são retângulos, círculos ou polígonos. O programa só irá “saber” quais figuras serão desenhadas em tempo de execução, de acordo com o contexto.

Uma solução para o problema descrito seria criar uma classe para cada tipo de figura, e um dos métodos dessas classes realizaria a ação de desenhar a figura. Para tornar essa solução mais flexível, pode-se criar uma classe chamada *Figura* que as demais herdariam, e utilizar o polimorfismo para facilitar o desenho de um conjunto de figuras.

A Figura 2.2 apresenta um diagrama de classes que modela a solução anterior. A classe *Figura* contém um método chamado *Desenha* e é utilizada como classe base para as diversas figuras. O método *Desenha* não executa nenhuma função na classe *Figura*, mas é sobrescrito em cada uma das classes derivadas, implementando a maneira como cada figura é desenhada.

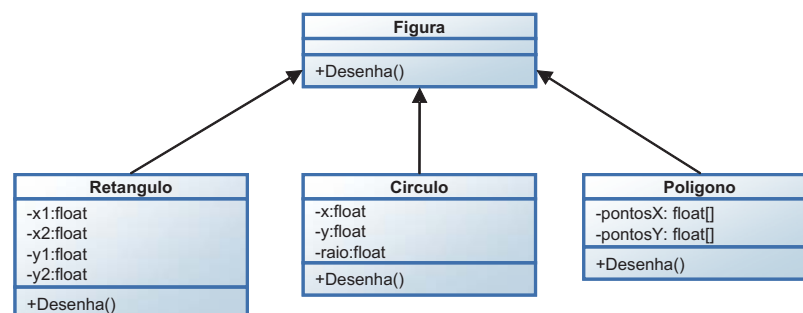


Figura 2.2: Diagrama de classes de uma estrutura polimórfica.

Os compiladores das linguagens orientadas a objeto são capazes de identificar a sobrecarga dos métodos, de forma que, ao realizar uma chamada de um método em uma

referência para uma classe base, o compilador executa a implementação presente no nível mais alto da hierarquia de classes. Por exemplo, se associarmos um objeto da classe *Retângulo* a uma referência para uma *Figura*, ao chamarmos o método *Desenha* dessa referência um retângulo é desenhado. Isso ocorre porque o compilador detecta que a referência aponta para um objeto de uma classe num nível maior na hierarquia de classes e sobrescreve o método *Desenha*. O compilador então executa a chamada da sobrecarga em substituição ao método da referência.

Através de chamadas polimórficas é possível simplificar a implementação do problema anterior criando um vetor de objetos *Figura* que é preenchido com objetos de suas diversas classes derivadas. Para que o programa desenhe todas as figuras é necessário apenas percorrer o vetor, executando a chamada do método *Desenha* de cada elemento, conforme mostrado na Listagem 2.1.

---

Listagem 2.1: Desenhando diversas figuras utilizando polimorfismo em C#.

---

```
1 //Cria uma matriz de objetos do tipo figura
2 Figura [] figs = new Figura[3];
3 //Cada elemento da matriz é associado
4 //a um tipo de objeto diferente
5 figs[0] = new Retangulo();
6 figs[1] = new Circulo();
7 figs[2] = new Poligono();
8 //Percorre a matriz
9 for(int i = 0; i < 4; i++)
10 {
11     //Chamada polimórfica, o compilador detecta
12     //se existem sobrecargas do método desenha
13     //na hierarquia de classes e desenha cada
14     //figura de maneira diferente.
15     figs[i].Desenha();
16 }
```

---

Além de simplificar a implementação, o polimorfismo facilita a expansão do sistema. No exemplo anterior, é possível adicionar novos tipos de figuras ao aplicativo escrevendo

novas classes que herdaram da classe *Figura* ou de uma de suas subclasses. Observe que as demais partes do código sofreriam poucas modificações, ou, no melhor caso, nenhuma alteração.

## 2.2 Padrões de projeto

Padrões de projeto são soluções bem concebidas para um grande número de problemas. Estas soluções são criadas por engenheiros de software experientes para auxiliar os programadores novatos nos seus projetos (PAPAJORGJI, 2005). Os trabalhos atuais nessa área são inspirados pelos trabalhos de Christopher Alexander e seus colegas que apresentavam guias para programadores novatos na linguagem Smalltalk (KURIAN; PONT, 2007).

A técnica é bastante difundida atualmente, em boa parte devido à influência do trabalho de Gamma e seus colegas. Eles publicaram um livro (GAMMA et al., 1995) com diversos padrões de projeto para uso geral no desenvolvimento de softwares orientados a objeto. Neste trabalho, cada padrão de projeto nomeia, descreve e apresenta a solução de um problema genérico, além de indicar quais as conseqüências de utilizar a solução proposta. Um padrão de projeto costuma ser acompanhado de guias e exemplos de implementação. Dessa forma, os engenheiros de software com pouca experiência podem encontrar soluções já testadas para seus problemas, obtendo resultados mais previsíveis. Já os engenheiros de software com maior experiência ganham um vocabulário único, facilitando a comunicação com os membros da equipe.

Um exemplo de padrão de projeto descrito no livro é o padrão *Estratégia* (*Strategy*). Este padrão é utilizado quando se tem um grupo de algoritmos diferentes, os quais não devem ser embutidos na classe que os utiliza. Isso pode ser necessário para diminuir o tamanho e, conseqüentemente, a complexidade da classe que contém os algoritmos. A adição de novos algoritmos, além de variações dos existentes, pode também ser necessária no futuro. Assim, a inclusão dos algoritmos em uma única classe dificulta essa operação.

Uma das soluções para o problema é apresentada na Figura 2.3 que descreve o padrão estratégia. Nela, os diferentes algoritmos são encapsulados em classes diferentes (Es-

estratégias concretas) e todos herdam de uma classe base abstrata (Estratégia). A classe *Estratégia* fornece a interface de acesso a todos os algoritmos que são utilizados pelo *Contexto*. O *Contexto* mantém uma referência para um objeto *Estratégia* que é previamente configurado com uma das estratégias concretas. O *Contexto* pode também definir uma interface por onde seus dados podem ser acessados pela *Estratégia*.

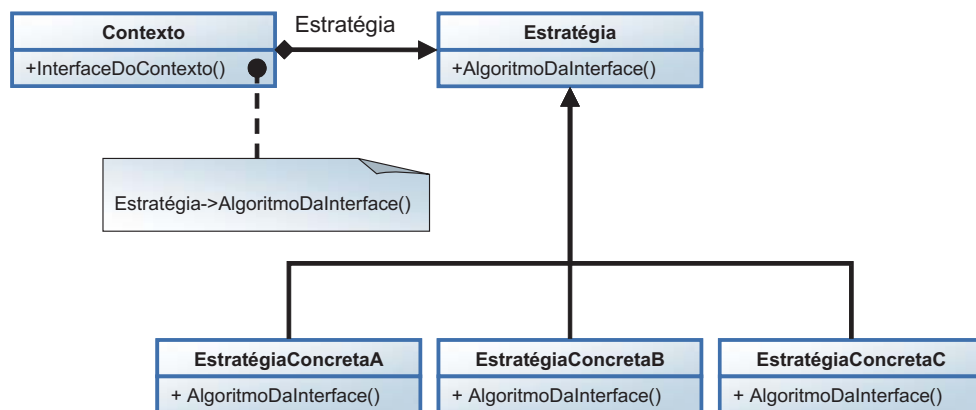


Figura 2.3: Diagrama de classes do padrão estratégia.

Para entender melhor o padrão estratégia, considere o seguinte exemplo: Um aplicativo de edição de imagens necessita salvar a imagem gerada em diversos formatos de arquivo diferentes (Figura 2.4). Alguns desses formatos de arquivo utilizam algoritmos de compressão que recebem o mapa de bits em memória e geram o arquivo compactado. Os algoritmos de compressão são diferentes de acordo com o formato de arquivo utilizado, mas realizam basicamente a mesma função: recebem o mapa de bits e geram o arquivo comprimido. Observe que, neste caso, tem-se uma família de algoritmos que utilizam uma mesma interface e se deseja torná-los intercambiáveis, de acordo com a vontade do usuário. Para resolver esse problema, pode-se aplicar a estrutura definida no padrão estratégia.

A classe que modela o aplicativo corresponderia ao *Contexto*. Essa classe possui um método *Salvar* que apresenta uma janela onde o usuário pode escolher o tipo de arquivo, e conseqüentemente, o tipo de compressão que será utilizado. Esse método utiliza uma referência para um *Compactador* (a *Estratégia*) para comprimir o arquivo. Os diversos algoritmos de compressão são encapsulados em diferentes classes que herdam da classe

*Compactador*, podendo-se assim, trocar o algoritmo de compressão, simplesmente modificando a referência para a qual o *Compactador* aponta. O diagrama de classes dessa estrutura é apresentada na Figura 2.4.

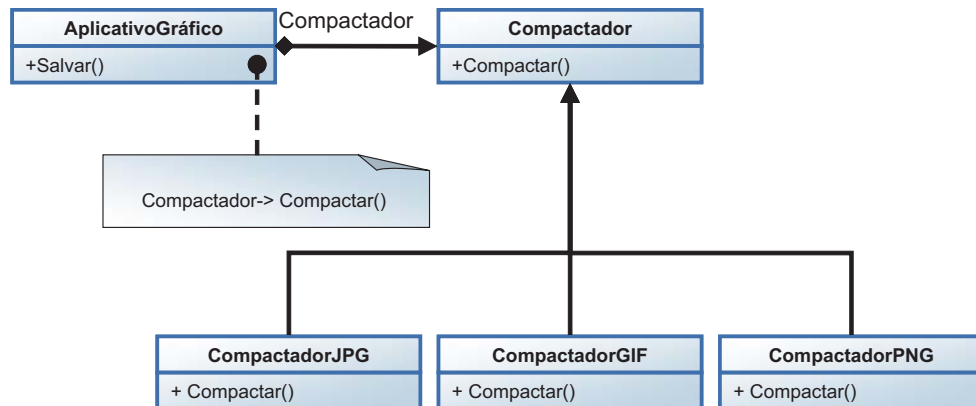


Figura 2.4: Padrão de projeto estratégia aplicado a um aplicativo “gráfico” genérico.

Essa estrutura permite a criação de hierarquias de classes que podem facilitar a expansão do aplicativo. Por exemplo, se for necessário adicionar novos métodos de compressão ou gerar variações dos algoritmos existentes, basta criar novas classes que herdam da classe *Compactador* ou de uma de suas classes derivadas. O padrão também elimina estruturas condicionais, já que utiliza o polimorfismo para realizar a mesma função. Em contrapartida, o padrão gera uma certa sobrecarga para acessar os diversos algoritmos devido ao uso do polimorfismo, além de aumentar o número de objetos existentes no projeto.

## 2.3 Reflexão e meta-programação

Reflexão e meta-programação são duas poderosas técnicas que permitem a criação de sistemas adaptativos (GRAY, 2004). Meta-programação é o ato de escrever programas que escrevem ou manipulam outros programas ou a si mesmos. Um exemplo de meta-programação é a reflexão. Segundo Maes (1987), um sistema computacional reflexivo é aquele que possui uma representação exata de si mesmo e, além disso, essa representação

é causal, ou seja, está sempre em conformidade com o sistema real. Dessa forma, modificações realizadas nesse modelo são refletidas no sistema real e vice-versa. Se o sistema possuir acesso ao modelo, ele pode modificar a si mesmo alterando a sua semântica em tempo de execução.

Através da reflexão é possível inferir sobre o estado (introspecção) e modificar a semântica do sistema em tempo de execução (intervenção). Em plataformas orientadas a objeto, a introspecção permite ao sistema descobrir quais as classes que compõem o código, e os atributos e métodos contidos nas mesmas. A intervenção permite criar instâncias dessas classes e executar seus métodos. A reflexão também permite a criação de novas classes e realizar modificações nas existentes em tempo de execução.

A reflexão pode ser implementada em diversas linguagens de programação (DEMERS; MALENFANT, 1995), estando disponível nas plataformas atuais de programação orientada a objeto, Java e *.net*. A seguir, são apresentados os princípios da reflexão na plataforma *.net* que serão utilizados no desenvolvimento do *framework*.

### 2.3.1 Reflexão na plataforma *.net*

A plataforma *.net* (MICROSOFT, 2007a) é uma das plataformas de programação atuais que suportam o recurso de reflexão. Para implementar essa funcionalidade, quando um programa é compilado gerando um código de máquina na forma de um arquivo *.dll* ou *.exe*, o compilador inclui neste meta-dados com informações sobre as classes. A plataforma fornece classes que são capazes de acessar esses meta-dados, permitindo a introspecção e a intervenção. A Figura 2.5 apresenta os pacotes da plataforma através dos quais são acessados os recursos de reflexão e algumas das classes contidas nesses pacotes.

A principal classe utilizada para acesso aos recursos de reflexão da plataforma *.net* é a classe *Type* do pacote *System*. *Type* é na verdade uma classe base abstrata, ou seja, ela é utilizada apenas como interface para suas classes derivadas. Dessa forma, não é possível gerar uma instância dela diretamente, mas a plataforma fornece meios de obter referências para instâncias das subclasses de *Type*. Na linguagem C#, essa operação é realizada através do operador *typeof*, do método *GetType*, que é herdado por todos os

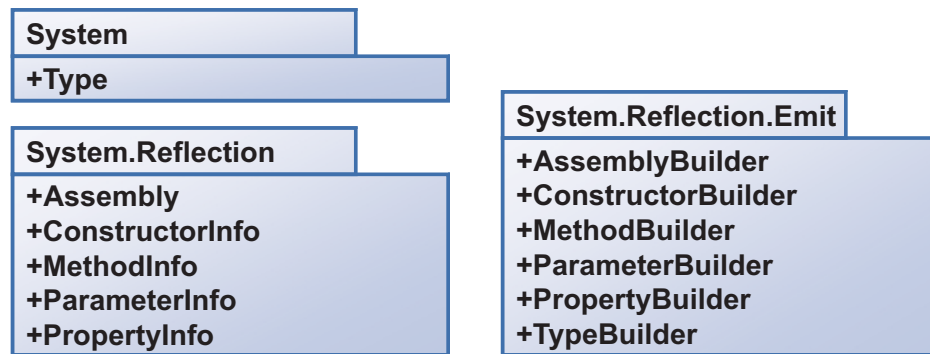


Figura 2.5: Pacotes da plataforma *.net* utilizados na reflexão. Cada bloco contém o nome do pacote na região superior e uma lista com as classes disponíveis neste pacote.

objetos da linguagem, ou ainda, a partir do método estático *GetType* da classe *Type*. O código da Listagem 2.2 demonstra a utilização de três métodos para obtenção da referência a um objeto *Type* que modela a classe *Double*.

Listagem 2.2: Obtendo as referências de *Type* no C#.

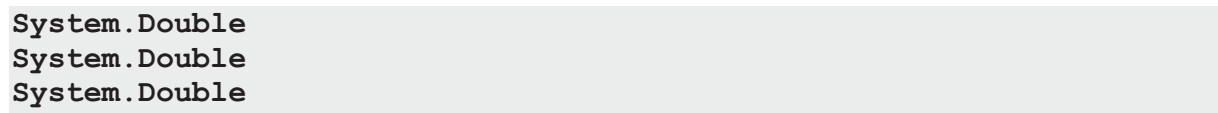
---

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace CreateType
6  {
7      class Program
8      {
9          //Demonstrando diversas maneiras de obter as informações sobre o
              tipo double
10         static void Main(string[] args)
11         {
12             //Utilizando o operador typeof
              Type tipoDouble1 = typeof(double);
13
14
15             //A partir de um objeto da classe
              double d = 10.0;
16             Type tipoDouble2 = d.GetType();
17

```

```
18
19      //Usando o método estático da classe Type
20      Type tipoDouble3 = Type.GetType("System.Double");
21
22      Console.WriteLine(tipoDouble1.ToString());
23      Console.WriteLine(tipoDouble2.ToString());
24      Console.WriteLine(tipoDouble3.ToString());
25
26  }
27 }
28 }
```



```
System.Double
System.Double
System.Double
```

Figura 2.6: Saídas do console geradas pela execução do programa da Listagem 2.2

As saídas do código acima são três linhas com o nome completo da classe (Figura 2.6). Este nome é formado pelo nome do pacote ao qual a classe pertence seguido pelo nome da classe, separados por um ponto. No caso, a classe *double* pertence ao pacote *System*, de modo que o nome completo da classe é *System.Double*.

*Type* possui diversas propriedades que permitem acessar as informações relativas a uma determinada classe. Através dessas propriedades, é possível, por exemplo, descobrir o nome da classe, de qual classe ela herda e se a classe é abstrata. A Tabela 2.1 resume algumas das propriedades contidas na classe *Type*.

A maioria dos métodos de *Type* são utilizados para obter informações sobre os membros da classe (construtores, atributos, métodos). Existem vários métodos com essa função, mas eles seguem um padrão. Por exemplo, o método *GetConstructor* é utilizado para obter um objeto do tipo *ConstructorInfo* que contém informações sobre um dos construtores da classe. Da mesma forma, o método *GetProperty* retorna um objeto do tipo *PropertyInfo* com informações sobre uma das propriedades do objeto. Existem ainda versões desses

Tabela 2.1: Propriedades da classe *Type*.

| Propriedade | Descrição                                                                    |
|-------------|------------------------------------------------------------------------------|
| Name        | Retorna o nome da classe.                                                    |
| FullName    | Retorna o nome completo da classe, incluindo o pacote que a classe pertence. |
| BaseType    | Obtém a classe da qual o tipo herda diretamente.                             |
| IsAbstract  | Retorna verdadeiro se a classe for abstrata.                                 |
| IsPublic    | Retorna verdadeiro se a classe for pública.                                  |

métodos cujo os nomes encontram-se no plural (*GetConstructors* para *GetConstructor*). Estes métodos são utilizados para obter todos os membros da classe de um determinado tipo. Por exemplo, para descobrir quais são os construtores da classe, basta utilizar o método *GetConstructors*, o qual retornará um vetor de objetos do tipo *ConstructorInfo*. A Tabela 2.2 apresenta alguns dos métodos da classe *Type* associados aos seus tipos de retorno.

Tabela 2.2: Métodos da classe *Type* e seus respectivos tipos de retorno

| Tipo de Membro | Métodos utilizados                                 | Tipo do Retorno        |
|----------------|----------------------------------------------------|------------------------|
| Construtores   | <i>GetConstructor()</i> , <i>GetConstructors()</i> | <i>ConstructorInfo</i> |
| Atributos      | <i>GetField()</i> , <i>GetFields()</i>             | <i>FieldInfo</i>       |
| Métodos        | <i>GetMethod()</i> , <i>GetMethods()</i>           | <i>MethodInfo</i>      |
| Propriedades   | <i>GetProperty()</i> , <i>GetProperties()</i>      | <i>PropertyInfo</i>    |

Semelhante à classe *Type*, as classes do tipo “*Info*” possuem propriedades e métodos para obter informações sobre cada um dos membros. Por exemplo, as classes *ConstructorInfo* e *MethodInfo* possuem o método chamado *GetParameters*, que retorna informações sobre os parâmetros desses membros através de um vetor de objetos do tipo *ParameterInfo*. As classes *ConstructorInfo* e *MethodInfo* possuem ainda, o método *Invoke* que realiza a chamada do membro correspondente. Quando este método é chamado em um

objeto da classe *ConstructorInfo*, uma nova instância da classe é gerada. Caso realize a chamada da função em um *MethodInfo*, o método é executado.

O programa da Listagem 2.3 demonstra a utilização da classe *Type*. Ao iniciar o programa, é criado um objeto *Type* com informações sobre a classe *Random*, utilizada para gerar números aleatórios. O programa utiliza as propriedades de *Type* para descobrir o nome completo da classe e de qual classe ela herda. Em seguida, o método *GetConstructors* é utilizado para obter as informações sobre os construtores da classe, gerando um vetor de objetos *ConstructorInfo*. O programa entra então em um laço, no qual a representação *string* de cada *ConstructorInfo* é exibida no console, apresentando o esquema do construtor. Por fim, o programa realiza a mesma operação com os métodos da classe. As saídas do programa são apresentadas na Figura 2.7.

Listagem 2.3: Utilizando os objetos “Info” no C#.

---

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4 using System.Reflection;
5
6 namespace UsandoOsInfos
7 {
8     class Program
9     {
10         static void Main(string[] args)
11         {
12             //Obtém a referência para o objeto da classe Type
13             Type tipo = typeof(System.Random);
14
15             //Utiliza as propriedades de Type para fornecer
16             //informações sobre a classe
17             Console.WriteLine("Nome da classe: " + tipo.Name);
18             Console.WriteLine("Nome completo: " + tipo.FullName);
19             Console.WriteLine("A classe herda de: " + tipo.BaseType);
20 }
```

```
21      //Lista os construtores da classe
22      Console.WriteLine("-Construtores
          -----");
23
24      //Utiliza a função GetConstructors() para obter as informações
25      //sobre os construtores da classe
26      ConstructorInfo[] construtores = tipo.GetConstructors();
27
28      foreach (ConstructorInfo construtor in construtores)
29      {
30          //Exibe no console uma representação string do construtor
31          //com os seus parâmetros
32          Console.WriteLine(construtor.ToString());
33      }
34
35      //Lista os métodos da classe
36      Console.WriteLine("-Métodos
          -----");
37
38      //Utiliza a função GetMethods() para obter as informações sobre
39      //os métodos da classe
40      MethodInfo[] metodos = tipo.GetMethods();
41
42      foreach (MethodInfo metodo in metodos)
43      {
44          //Exibe no console uma representação string do método
45          //com seu nome, parâmetros e tipo de retorno
46          Console.WriteLine(metodo.ToString());
47      }
48
49      }
50
51      }
52  }
```

---

```
Nome da classe: Random
Nome completo: System.Random
A classe herda de: System.Object
-Construtores-----
Void .ctor()
Void .ctor(Int32)
-Métodos-----
Int32 Next()
Int32 Next(Int32, Int32)
Int32 Next(Int32)
Double NextDouble()
Void NextBytes(Byte[])
System.Type GetType()
System.String ToString()
Boolean Equals(System.Object)
Int32 GetHashCode()
```

Figura 2.7: Saídas do console geradas pela execução do programa da Listagem 2.3.

O pacote *System.Reflection* contém uma classe chamada *Assembly*, utilizada para manipular os arquivos executáveis e bibliotecas da plataforma. Ela também possui métodos que permitem ao programador carregar, ou até mesmo executar um *assembly*, caso este seja um executável. Assim como a classe *Type*, esta classe também possui uma série de propriedades e métodos que permitem ao programa descobrir informações sobre um *assembly*.

Antes de realizar qualquer operação com uma instância da classe *Assembly*, é preciso carregar a informação de um arquivo *assembly*. Isso pode ser feito através dos métodos estáticos *Load* e *LoadFrom*, que criam um novo objeto da classe e carregam a informação do *assembly*. Caso se deseje obter uma referência para um objeto que represente o *assembly* do programa que está sendo executado, basta utilizar o método *GetExecutingAssembly*.

Após carregar um *assembly*, é possível utilizar o método *GetTypes*, para obter um vetor de objetos *Type* referentes às classes disponíveis no *assembly*. A partir desse momento, é possível gerar uma instância de uma dessas classes através dos métodos descritos anteriormente.

O programa da Listagem 2.4 demonstra como uma aplicação pode gerar uma instância

de uma classe contida em um *assembly* qualquer e executar seus métodos. Ao ser executado, o programa recebe dois argumentos, o primeiro deve ser um caminho para um arquivo *assembly*, enquanto o segundo é o nome de uma classe contida no *assembly*.

Listagem 2.4: Carregando uma classe dinamicamente no C#.

---

```
1  using System;
2  using System.Text;
3  using System.Reflection;
4
5  namespace ExReflection
6  {
7      class Program
8      {
9          static void Main(string[] args)
10         {
11             //Carrega o assembly
12             Assembly assembly = Assembly.LoadFrom(args[0]);
13
14             //Obtém as informações sobre a classe contida no Assembly
15             Type tipo = assembly.GetType(args[1]);
16
17             //Obtém as informações sobre os construtores da classe
18             ConstructorInfo[] construtores = tipo.GetConstructors();
19
20             //Imprime no console uma lista com os construtores públicos
21             for (int i = 0; i < construtores.Length; i++)
22             {
23                 //Gera uma string com o construtor e seus parâmetros
24                 Console.WriteLine(i.ToString() + " - " + construtores[i].
25                                     ToString());
26
27                 //Obtém do usuário qual o construtor será utilizado
28                 Console.Write("Digite o número do construtor a ser utilizado: "
29                               );
```

```
29      int nroDoConstrutor = int.Parse(Console.ReadLine());
30
31      ParameterInfo informacoesDosParametros = construtores[
          nroDoConstrutor].GetParameters();
32
33      //Obtém os parâmetros necessários pelo construtor através da
34      //função auxiliar
35      object [] parametros = ObterValoresDosParametros(
          informacoesDosParametros);
36
37      //Cria uma instância do objeto utilizando a reflexão
38      //invocando o construtor da classe
39      object objeto = construtores[nroDoConstrutor].Invoke(parametros
          );
40
41      //Exibe a representação string do objeto no console
42      Console.WriteLine("Objeto: " + objeto.ToString());
43
44      //Obtém as informações sobre os métodos da classe
45      MethodInfo [] metodos = tipo.GetMethods();
46
47      //Imprime no console uma lista com os métodos públicos
48      for (int i = 0; i < metodos.Length; i++)
49      {
50          //Lista os diversos métodos com os seus parâmetros
51          Console.WriteLine(i.ToString() + " - " + metodos[i].
              ToString());
52      }
53
54      //Obtém do usuário qual o método que ele deseja executar
55      Console.Write("Digite o número do método a ser executado :");
56      int nroDoMetodo = int.Parse(Console.ReadLine());
57
58      //Obtém os parâmetros necessários do método
59      informacoesDosParametros = metodos[nroDoMetodo].GetParameters()
```

```
        ;
60         parametros = ObterValoresDosParametros(informacoesDosParametros
        );
61
62         //Executa o método
63         object retorno = metodos[nroDoMetodo].Invoke(objeto , parametros
        );
64
65         //Exibe a representação string do retorno do método
66         Console.WriteLine("Retorno do método: " + retorno.ToString());
67
68         //Exibe a representação string do objeto , após a execução
69         //do método
70         Console.WriteLine("Objeto após a execução do método: " + objeto
        .ToString());
71
72     }
73
74
75     //Função auxiliar para preenchimento dos parâmetros de um
76     //construtor ou método.
77     //Esta função recebe a lista de parâmetros e pergunta o
78     //valor de cada um deles ao usuário. Ao final , a função
79     //retorna um vetor de objetos com os valores dos parâmetros.
80     private static object [] ObterValoresDosParametros(ParameterInfo []
        informacoesDosParametros)
81     {
82         //Cria a variável que irá manter os valores dos parâmetros
83         object [] valoresDosParametros = new object []
        informacoesDosParametros.Length];
84
85         //Para cada parâmetro
86         for (int i = 0; i < valoresDosParametros.Length; i++)
87         {
88             //Pergunta ao usuário o valor do parâmetro
```

```
89         Console.WriteLine("Entre com o parâmetro " +
90             informacoesDosParametros[i].Name + ": ");
91
92         //Obtém uma entrada do usuário
93
94         string respostaDoUsuario = Console.ReadLine();
95
96         //Verifica o tipo do parâmetro, se ele for uma string
97         //o parâmetro é a própria entrada do usuário
98         if (informacoesDosParametros[i].ParameterType.Equals(typeof
99             (string)))
100         {
101             valoresDosParametros[i] = respostaDoUsuario;
102         }
103         //Caso contrário tenta converter a string para o tipo
104         parâmetro
105         else
106         {
107             Type tipoDoParametro = informacoesDosParametros[i].
108                 ParameterType;
109             //Verifica se o parâmetro tem um construtor que
110             //recebe uma string
111             ConstructorInfo informacoesDoConstrutor =
112                 tipoDoParametro.GetConstructor(new Type[] { typeof(
113                     string) });
114
115             //Se existe o construtor, constrói o objeto e
116             //o adiciona à lista de parâmetros
117             if (informacoesDoConstrutor != null)
118             {
119                 valoresDosParametros[i] = informacoesDoConstrutor.
120                     Invoke(new object[] { respostaDoUsuario });
121                 continue; //Passa para a próxima interação
122             }
123
124             //Se não possuir um construtor verifica se existe
```

```

117         //um método de conversão
118         MethodInfo informacoesDoMetodo = tipoDoParametro.
            GetMethod("Parse", new Type[] { typeof(string) });
119
120         if (informacoesDoMetodo != null && informacoesDoMetodo.
            IsStatic)
121         {
122             valoresDosParametros[i] = informacoesDoMetodo.
                Invoke(null, new object[] { respostaDoUsuario })
                ;
123             continue;
124         }
125
126         //Se não conseguir converter o parâmetro retorna uma
127         //referência nula
128         return null;
129     }
130
131 }
132 return valoresDosParametros;
133 }
134 }
135 }

```

Ao ser executado, o programa gera um objeto *Assembly* e carrega as informações do arquivo passado pelo argumento através do método estático *LoadFrom* (Linha 12). Em seguida, o programa gera um objeto *Type* através da chamada do método *GetType* que recebe como argumento o nome da classe. Nesse momento, o software lista todos os construtores da classe e pede ao usuário que escolha um deles. Para realizar essa função, o programa utiliza o método *GetConstructors* da classe *Type*.

O programa segue varrendo o vetor de *ConstructorInfos*, gerado pelo método *GetConstructors*, exibindo uma linha com o protótipo de cada construtor na saída do console. Após imprimir todos os construtores, o programa solicita que o usuário escolha um deles. O usuário deve então, digitar o número que se encontra à esquerda de um dos construtores.

Nesse momento, o programa novamente infere sobre o construtor escolhido, pesquisando os seus parâmetros. Os parâmetros do construtor são obtidos através da chamada do método *GetParameters* da classe *ConstructorInfo*.

O vetor, com os parâmetros do construtor, é processado pela função auxiliar *ObterValoresDosParametros*. Essa função pergunta ao usuário qual o valor que será associado a cada parâmetro e o usuário responde com uma *string*. Para realizar essa pergunta é utilizada a propriedade *Name* da classe *ParameterInfo*.

A função tem ainda que converter a resposta do usuário, na forma de uma *string*, para o tipo de dado do parâmetro. Para realizar essa conversão, a função primeiro verifica se o tipo do parâmetro é uma *string*, o que tornaria a conversão desnecessária. Isso é feito através da propriedade *ParameterType* que retorna o tipo do parâmetro. O tipo é comparado com o tipo *string* através do método *Equals*. *Equals* retorna verdadeiro se o tipo do parâmetro é igual ao tipo *string*.

Caso o tipo do argumento seja diferente do tipo *string*, é necessária alguma forma de conversão. Uma maneira de realizar essa conversão é procurar um construtor da classe que receba como argumento uma única *string*. Isso é realizado, logo em seguida, através do método *GetConstructor*, que pesquisa a existência de construtores na classe que possuam argumentos com os tipos repassados ao método. Caso o método encontre um construtor compatível, uma referência para um objeto *ConstructorInfo* é retornada. Se não for encontrado nenhum construtor, o método retorna uma referência nula. Quando um construtor é encontrado, o método gera uma instância da classe através da chamada do método *Invoke*. O parâmetro é então adicionado a um vetor que será passado para o construtor da classe.

Se não existem construtores que recebem uma única *string* como argumento, pode-se pesquisar a classe por algum método que realiza a conversão. As classes correspondentes aos tipos base da linguagem costumam apresentar o método estático *Parse* para esta função. O programa procura então, a existência desse método através da chamada *GetMethod*. O método *GetMethod* recebe como argumentos o nome do método e os seus parâmetros e, de maneira análoga ao método *GetConstructor*, pesquisa a classe à procura

de métodos com essas características. Caso encontre um método compatível, *GetMethod* retorna um objeto do tipo *MethodInfo* com as informações sobre o método. Uma referência nula é retornada quando nenhum método é encontrado.

A conversão para o tipo do parâmetro é então realizada através da chamada do método *Invoke* da classe *MethodInfo*. O objeto é então criado pelo método e adicionado ao vetor de parâmetros. Se ainda assim não for possível converter o objeto, será gerada uma referência nula como resposta.

Após obter os valores dos parâmetros do usuário, a classe é instanciada através da chamada da função *Invoke* e sua representação na forma de texto é exibida no console com a chamada do método *ToString*.

O programa apresenta ao usuário os métodos públicos presentes na classe instanciada, sendo requisitado do usuário a escolha de um dos métodos. Após a resposta, a função *ObterValoresDosParametros* é novamente utilizada para obter os valores dos parâmetros do método selecionado.

Por fim, o método é executado através da chamada de *Invoke* e o retorno do método é escrito no console na forma de texto. Em seguida, a representação na forma de *string* do objeto também é exibida no console, encerrando a execução do programa.

A Figura 2.8 mostra a saída no console referente a execução do programa da Listagem 2.4. Na figura, os valores inseridos pelo usuário encontram-se sombreados. Os argumentos passados para o programa foram o assembly *mscorlib.dll* e a classe *Random* do pacote *System*. Após listar os construtores da classe, o usuário optou por utilizar o construtor de número um, que recebe um inteiro. O programa então pergunta o valor do parâmetro *Seed* ao usuário e, após a resposta, gera uma instância da classe e exibe uma *string* com o tipo do objeto. Em seguida, são listados os protótipos dos métodos disponíveis na classe, e, novamente, o programa pede uma intervenção do usuário. A terceira versão do método *Next* (opção de número 2) é selecionada e o programa pergunta o valor do parâmetro *maxValue*. Por fim, o programa executa a chamada do método exibindo na tela o valor do retorno e a representação *string* do objeto.

O pacote *System.Reflection.Emit* contém classes que permitem criar novos elementos

```
C:\>ExReflection
c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\mscorlib.dll
System.Random
0 - Void .ctor()
1 - Void .ctor(Int32)
Digite o número do construtor a ser utilizado: 1
Entre com o parâmetro Seed: 2000
Objeto: System.Random
0 - Int32 Next()
1 - Int32 Next(Int32, Int32)
2 - Int32 Next(Int32)
3 - Double NextDouble()
4 - Void NextBytes(Byte[])
5 - System.Type GetType()
6 - System.String ToString()
7 - Boolean Equals(System.Object)
8 - Int32 GetHashCode()
Digite o número do método a ser executado :2
Entre com o parâmetro maxValue: 10000
Retorno do método: 5768
Objeto após o método: System.Random
```

Figura 2.8: Saídas do console geradas pela execução do programa da Listagem 2.4.

da plataforma e alterar os existentes em tempo de execução. Para isso, existem uma série de classes “*Builder*” que realizam essa função. Esse recurso não será utilizado no *framework* objeto deste trabalho, de modo que não será descrito neste documento. Maiores informações podem ser obtidas na documentação da plataforma *.net* (MICROSOFT, 2007d).

A reflexão também permite a inclusão de marcadores especiais em elementos da linguagem (classes, atributos, métodos, parâmetros, *assemblies*). Esses marcadores são chamados de *Atributos*, e são inseridos na forma de meta-dados que não influem diretamente no membro ao qual foram associados. Entretanto, os atributos podem ser obtidos através da reflexão, fornecendo informações especiais sobre um membro.

Um exemplo de atributo disponível na plataforma é o *Serializable*. Este atributo, quando associado a uma classe, indica que ela pode ser transformada em um vetor de bytes utilizado para gravação em arquivo ou transmissão em rede. Para associar um

atributo a um membro, basta colocar o nome do atributo entre colchetes junto a sua declaração. A Listagem 2.5 demonstra a associação do atributo *Serializable* à uma classe (linha 8).

---

Listagem 2.5: Associando um atributo a uma classe.

---

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace Atributos {
6
7     //Associando o atributo Serializable à classe Ponto
8     [Serializable]
9     public class Ponto
10    {
11
12    }
13 }
```

---

Para descobrir se um membro possui um atributo, basta utilizar a propriedade *Attributes* ou o método *GetCustomAttributes*, disponíveis na classe *Type* e nas classes “*Infos*”. Por exemplo, é possível listar os atributos da classe *Ponto* da Listagem 2.5 utilizando o código da Listagem 2.6.

---

Listagem 2.6: Listando os atributos de uma classe.

---

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace Atributos {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             Type tipo = typeof(Ponto);
```

```
11
12         //Obtém os atributos da classe
13         object[] atributos = tipo.GetCustomAttributes(false);
14
15         foreach (object atributo in atributos)
16         {
17             //Imprime a representação string de cada atributo
18             Console.WriteLine(atributo.ToString());
19         }
20
21     }
22 }
23 }
```

---

**System.SerializableAttribute**

Figura 2.9: Saída do console gerada pela execução do programa da Listagem 2.6.

A função *GetCustomAttributes* retornou um objeto do tipo *System.SerializableAttribute*. Isso ocorre porque um atributo é, na verdade, uma classe que herda de *System.Attributes*. Além disso, toda classe que representa um atributo possui uma nomenclatura especial, sendo seu nome terminado com o termo *Attribute*. Dessa forma, o atributo *Serializable* é na verdade uma classe cujo o nome é *SerializableAttribute*.

A reflexão é cada vez mais utilizada em diversos projetos, incluindo várias partes da própria plataforma *.net*. Seus recursos facilitam a criação de sistemas adaptativos, permitindo ao programador criar sistemas genéricos cujas funcionalidades só serão conhecidas e acessadas em tempo de execução.

## 2.4 *Plug-ins*

Um dos desafios da engenharia de software é o de criar softwares que sejam facilmente expansíveis. Uma maneira de fazer isso é utilizar uma arquitetura de *plug-ins*. *Plug-ins*

são bibliotecas que são carregadas e executadas por uma aplicação principal, de acordo com a sua necessidade, fornecendo funções específicas que esta não suporta inicialmente.

Os *plug-ins* são utilizados atualmente em diversas aplicações, como por exemplo, nos navegadores da internet (para exibir animações e vídeos em páginas *html*), em reprodutores multimídia (para reproduzir novos formatos), e em editores de imagem (para adição de novos efeitos). Vários sistemas, de diferentes áreas, desenvolvidos utilizando arquiteturas de *plug-ins* podem ser encontrados na literatura (HOLY et al., 2006; PAPAJORGJI, 2005; MULLICK et al., 1999).

A Plataforma Eclipse é um bom exemplo do quanto a utilização de *plug-ins* pode facilitar a expansão do sistema. O Eclipse foi criado para prover uma plataforma comum de desenvolvimento de IDEs<sup>2</sup> capaz de acomodar as diversas ferramentas de desenvolvimento necessárias para as aplicações atuais (RIVIÈRES; WIEGAND, 2004).

Para tal, o Eclipse foi desenvolvido utilizando um sistema de *plug-ins* bastante eficiente, cuja a arquitetura é mostrada na Figura 2.10. A figura mostra que a plataforma é composta apenas de componentes básicos, sendo eles uma bancada (Workbench) que utiliza as bibliotecas gráficas *JFace* e *SWT* para gerar os componentes visuais relativos à interface com o usuário, uma área de trabalho (Workspace) que gerencia os arquivos e projetos em desenvolvimento, um sistema de ajuda ao usuário e um sistema de colaboração para desenvolvimento em equipe. As demais funcionalidades da IDE são adicionadas através de *plug-ins* que estendem os recursos básicos da plataforma, adicionando, por exemplo, editores para diversas linguagens com verificação de sintaxe, editores gráficos, navegadores de objetos, integração com servidores de aplicação dentre outros. Com isso é possível transformar o Eclipse em uma IDE Java, em uma ferramenta de desenvolvimento para web, ou em uma suíte para modelagem em UML, modificando apenas o conjunto de *plug-ins* utilizados pelo sistema.

Arquiteturas de *plug-ins*, juntamente com os recursos de reflexão e uma boa estrutura de código, que aproveite os recursos de herança e polimorfismo, permitem a criação de sistemas adaptativos e expansíveis. No próximo capítulo, essas técnicas serão utilizadas

---

<sup>2</sup>Integrated development environment ou ambiente de desenvolvimento integrado.

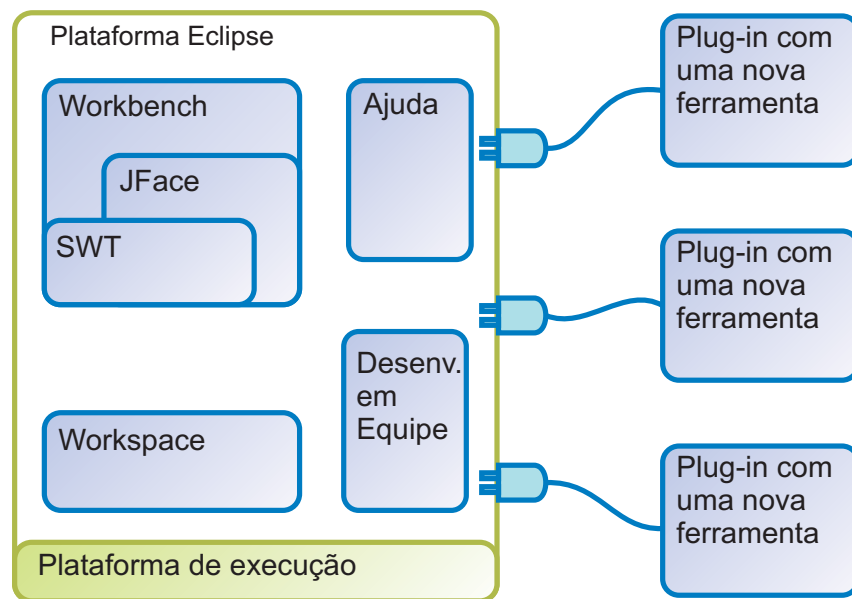


Figura 2.10: Arquitetura da plataforma Eclipse.

para criar um *framework* para aquisição de dados e interfacemento de hardware que facilite a criação de sistemas de processamento, compatíveis com diversos equipamentos.

## Capítulo 3

# O desenvolvimento do *framework*

No capítulo anterior, foram apresentadas as técnicas para construção de sistemas adaptativos. Neste capítulo, estas técnicas serão utilizadas no desenvolvimento de um *framework* para aquisição de dados. Este *framework* poderá ser utilizado no desenvolvimento de softwares de coleta e processamento sinais, permitindo que operem com diversos equipamentos diferentes.

Um *framework* orientado a objeto é um conjunto de classes e interfaces que provê uma solução para uma família de problemas semelhantes. O conjunto de classes deve ser flexível e extensível para permitir a construção de várias aplicações com pouco esforço, especificando apenas as particularidades de cada uma.

Apesar de muito semelhante a uma biblioteca de classes, um *framework* diferencia-se desta por fornecer uma estrutura de colaboração de classes. Nas bibliotecas de classes, as funcionalidades estão embutidas e implementadas de maneira isolada. Dessa forma, é possível acessar uma determinada funcionalidade da biblioteca utilizando apenas uma única classe. O *framework* impõe um modelo de colaboração, onde as funcionalidades são acessadas através de grupos de classes e suas relações. A aplicação utiliza essas classes estendendo as suas funcionalidades para acomodar os requisitos particulares do problema.

*Frameworks* também se parecem muito com padrões de projeto. Entretanto, se diferenciam destes por serem soluções mais específicas e complexas. Padrões de projeto são soluções mais abstratas, para problemas menores. Os *frameworks* encontram-se num nível

menor de abstração, são maiores e mais complexos. É muito comum um *framework* ser construído a partir de vários padrões de projeto (JOHNSON, 1997).

### 3.1 Requisitos

O diagrama de blocos de um equipamento de coleta e processamento típico é apresentado na Figura 3.1. Nele, o sinal analógico é conectado ao SAD através de um de seus canais, onde cada canal pode possuir recursos de condicionamento, como por exemplo, amplificação e filtragem. Após a etapa de condicionamento, o sinal é enviado para um conversor analógico-digital (AD) que converte o sinal analógico para um valor binário. Em equipamentos com vários canais, o conversor AD pode ser compartilhado através de um sistema de multiplexação. Os SADs também possuem uma unidade de controle responsável por comandar todos os elementos do equipamento, e se comunicam com o computador através de um canal de comunicação digital.

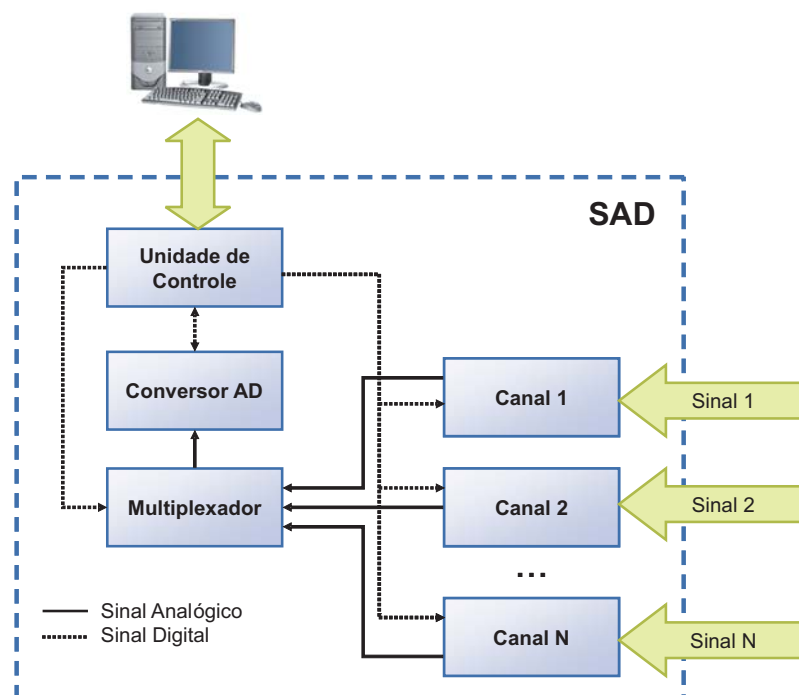


Figura 3.1: Diagrama de blocos exibindo os elementos que compõem um SAD típico.

Diferentes SADs podem possuir elementos e recursos diferentes. Por exemplo, dois SADs podem diferenciar-se no tipo de interface de comunicação digital, resolução do conversor AD, taxa de amostragem e número de canais. Inclusive, alguns desses elementos podem ser programáveis, por exemplo, um SAD pode permitir ao usuário definir a sua taxa de aquisição, o ganho de cada canal, ou a frequência de corte de um filtro.

Talvez, esse grande número de diferenças entre equipamentos seja o motivo de utilização da arquitetura de software apresentada na introdução deste trabalho (Figura 3.2). Inicialmente, parece mais simples construir o aplicativo de processamento vinculado ao equipamento de aquisição, onde é possível acessar todos os seus recursos diretamente. Entretanto, esta arquitetura possui problemas, conforme já descrito.

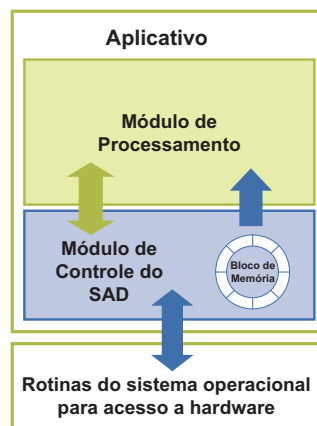


Figura 3.2: Arquitetura de software geralmente empregada para a construção de sistemas de captura e processamento de dados.

Dessa forma, este trabalho propõe a utilização de outra arquitetura, apresentada na Figura 3.3. Nesta, dois aplicativos de processamento estão se comunicando com diversos equipamentos. Assim como na arquitetura clássica, cada aplicativo possui um bloco responsável pelo processamento do sinal e um bloco responsável pelo acesso e controle do sistema de aquisição de dados. Entretanto, o módulo de controle não fornece acesso direto a um sistema de aquisição de dados real e sim, a uma interface genérica para diversos SADs.

Essa interface é única para todos os SADs compatíveis com o aplicativo, criando uma

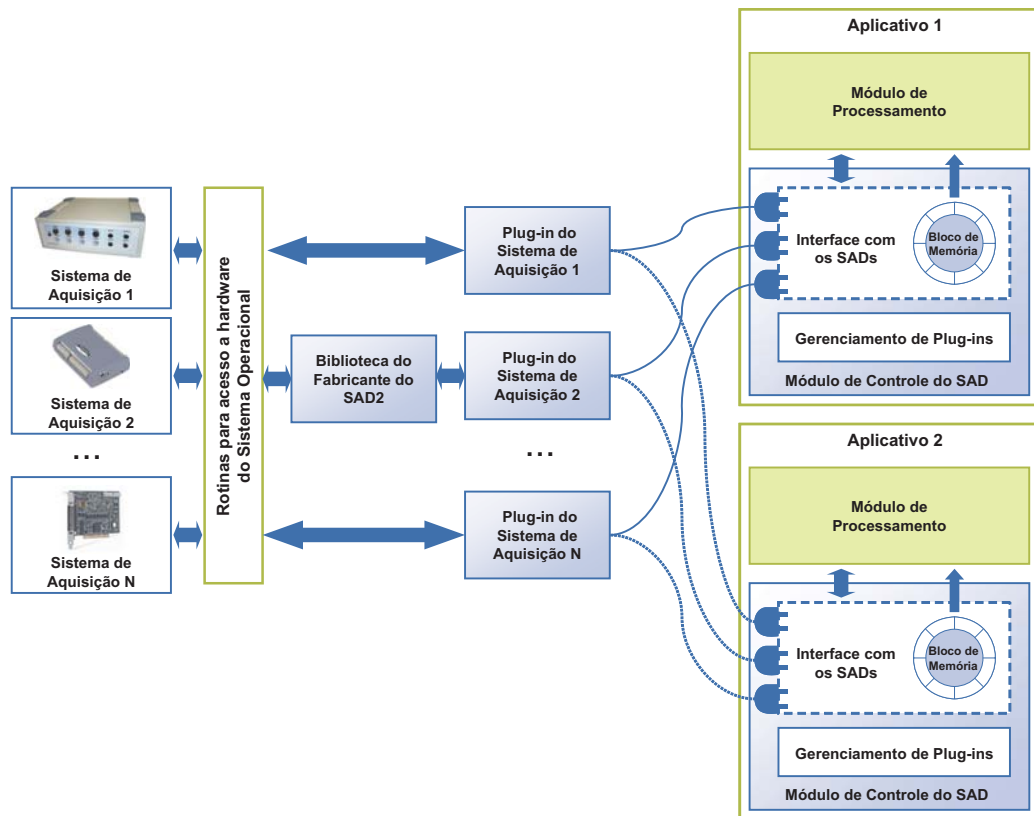


Figura 3.3: Arquitetura proposta, baseada em *plug-ins*.

camada de acesso intermediária. Dessa forma, o aplicativo torna-se independente do SAD utilizado, podendo, durante a sua operação, alternar entre vários equipamentos diferentes de maneira transparente para o módulo de processamento. A interface fornece acesso aos blocos de memória, contendo os dados coletados pelo equipamento, além de métodos para controle do processo de aquisição de dados. Mesmo possuindo uma interface única, o módulo de controle fornece meios para acessar os recursos específicos de cada equipamento.

Cada sistema de aquisição possui um *plug-in* baseado na interface descrita anteriormente. Esse *plug-in* funciona como um “tradutor” entre a interface e o equipamento, e é capaz de identificar a presença de equipamentos do seu tipo no sistema. Os *plug-ins* comunicam-se com o equipamento para o qual foram projetados de diversas formas. Alguns utilizam as bibliotecas fornecidas pelo fabricante do equipamento, enquanto outros podem acessar o equipamento diretamente, através de rotinas do sistema operacional. O

módulo de controle do SAD também possui um bloco para gerenciamento dos *plug-ins*. Este bloco é responsável pelo controle do ciclo de vida dos *plug-ins*, fornecendo métodos para carregar os *plug-ins* em tempo de execução, conectando a interface ao equipamento. O módulo de gerenciamento também utiliza os *plug-ins* para pesquisar o computador, fornecendo uma lista dos equipamentos disponíveis para utilização.

Se vários aplicativos utilizarem o módulo de controle do SAD descrito acima, um equipamento poderá ser automaticamente compatível com diversos softwares e vice-versa. Esse módulo de controle será desenvolvido na forma de um *framework*, sobre o qual poderão ser criados os diversos aplicativos de captura e processamento de dados, e os *plug-ins* para conexão com equipamentos.

Para o desenvolvimento do *framework*, será utilizada uma plataforma de programação que permita suporte a diversas linguagens de programação (C#, C++, Visual Basic, J#), possua os recursos de multitarefa, permita a linkagem dinâmica e reflexão, além de possuir um sistema de gerenciamento de memória. A plataforma *.net* vem sendo bastante aceita no mercado, sendo compatível com diversas ferramentas, como por exemplo, o LabView (National Instruments, Co., Austin, TX). A plataforma também permite a conexão com outros sistemas através do C++ (TOUB, 2004).

O processo de desenvolvimento de um aplicativo que utilize o *framework* requer, portanto, que a linguagem utilizada seja compatível com a plataforma *.net*, e que o *.net framework* esteja presente no ambiente de execução do aplicativo. As próximas seções apresentam os passos do desenvolvimento do *framework* utilizando a arquitetura e os requisitos descritos anteriormente.

## 3.2 Modelagem do módulo de controle

No diagrama de blocos da arquitetura proposta (Figura 3.3), observa-se que o módulo de controle do SAD possui duas unidades. A primeira corresponde a uma interface que o módulo de processamento irá utilizar para acessar os diversos SADs. A outra unidade corresponde a um sistema de gerenciamento do ciclo de vida dos *plug-ins*. As próximas

seções descrevem o desenvolvimento dessas unidades.

### 3.2.1 A interface com os *SADs*

Para construir a interface com os diversos SADs, optou-se por utilizar uma classe abstrata. Esta classe deve possuir apenas os atributos e métodos comuns a todos os SADs, e associa-se com a classe do programa principal de forma semelhante ao padrão de projeto estratégia, conforme o diagrama de classes da Figura 3.4.

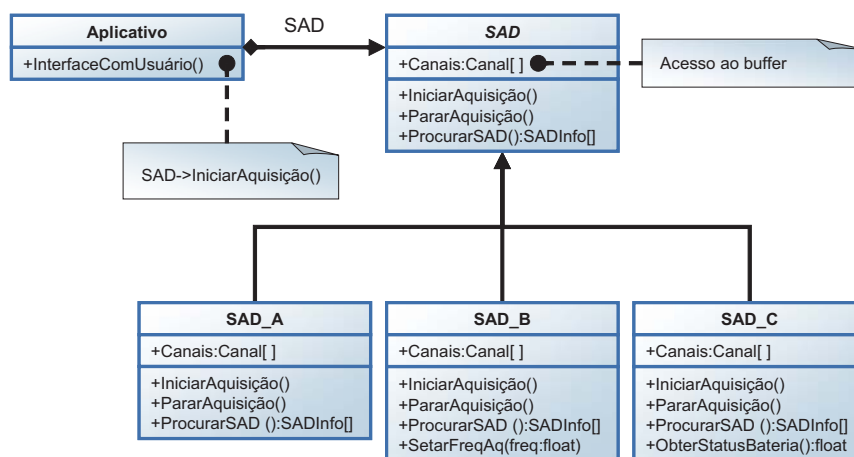


Figura 3.4: Diagrama de classes da interface com o SAD

No diagrama de classes, tem-se a interface *SAD*, uma classe abstrata que modela os diversos equipamentos que o sistema irá utilizar. Essa classe possui um membro chamado *Canais* que modela os diversos canais presentes no equipamentos, na forma de um vetor. Cada canal é representado através de um objeto da classe *Canal* que será apresentada em detalhes no decorrer do texto.

Para controlar o processo de aquisição de dados, a interface precisa de métodos para iniciar e parar o processo de coleta. Portanto, foram adicionados os métodos *IniciarAquisição* e *PararAquisição* à interface. A interface também deve ser capaz de pesquisar o sistema a procura de SADs. Esta função é realizada pelo método *ProcuraSAD* que retorna uma série de informações sobre os dispositivos de aquisição de dados conectados ao sistema. Apesar de conter os métodos descritos anteriormente, a classe não fornece

uma implementação deles. Estas implementações serão executadas nas classes derivadas, e serão acessadas por polimorfismo.

Os diversos equipamentos são modelados através de classes derivadas da classe *SAD*. Estas classes implementam as funções *IniciarAquisição*, *ParaAquisição* e *ProcurarSAD*. As classes derivadas também podem implementar novas funções, para acessar recursos únicos do equipamento em questão.

Além da interface com os SADs, o sistema necessita de uma segunda interface para controlar os aspectos individuais de cada canal. De maneira análoga, esta interface deve conter apenas os atributos e métodos presentes no maior número possível de canais, mantendo a compatibilidade com uma grande variedade de dispositivos.

Observando os diversos SADs, percebe-se que os canais necessitam de um *buffer* para manter os dados coletados pelos mesmos. O aplicativo principal deve então possuir um método de acesso a este buffer, de modo que possa realizar leituras dos seus dados, obtendo as amostras necessárias para processamento. Os canais devem também possuir a possibilidade de serem habilitados ou desabilitados, conforme a necessidade do usuário.

A classe abstrata *Canal*, apresentada no diagrama de classes da Figura 3.5, modela os recursos descritos anteriormente. A classe possui um *buffer circular* que é utilizado para armazenar os dados coletados pelo equipamento, até que a unidade de processamento os utilize. A classe também fornece para o aplicativo funções para acesso ao *buffer* e modificação do estado de ativação do canal.

Os canais de cada SAD podem ser implementados utilizando a classe *Canal* como base e adicionando novos métodos de acordo com as funcionalidades de cada um deles. Isso é demonstrado no diagrama da Figura 3.5, onde as classes *Canal\_B* e *Canal\_C* acrescentam novos métodos para configurar seus recursos individuais.

Na execução do sistema, ao criar uma instância de uma classe derivada de *SAD*, esta classe irá preencher o vetor de *Canais* com objetos de classes derivadas de *Canal*. A unidade de processamento pode então iniciar a aquisição de dados através da chamada do método *IniciaAquisição*. Quando isso ocorrer, o objeto *SAD* irá criar uma nova tarefa concorrente. Esta tarefa irá comunicar-se com o equipamento, obtendo as amostras

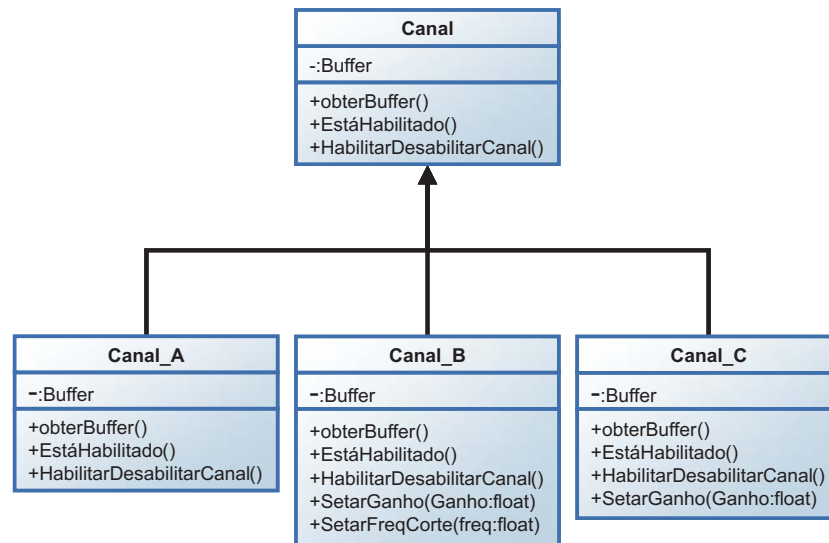


Figura 3.5: Diagrama de classes das interfaces com os canais

de cada sinal e preenchendo o *buffer* de cada canal. A unidade de processamento, em contrapartida, deve criar uma tarefa para retirar as amostras do *buffer* e realizar o seu processamento. Quando a unidade processamento decidir parar o processo de aquisição de dados, ela deve chamar a função *PararAquisição*. Esta função comunica-se com o equipamento, interrompendo o processo de aquisição, e termina a tarefa criada pela função *IniciaAquisição*.

Para manter a compatibilidade com diversos sistemas, os dados fornecidos pelo canal já são transformados para a sua unidade final, sendo representados na forma de um número em ponto flutuante. Quando estes dados são enviados ao *buffer* do canal, é utilizada uma estrutura de dados que armazena o tempo e o valor do sinal, tornando o sistema independente de variações de frequência. A interface do canal fornece ainda uma *string* indicando qual a unidade está sendo utilizada, e os valores máximos e mínimos que ele é capaz de medir.

A definição dessa interface já facilita a adição novos SADs ou a modificação dos existentes no código, gerando uma adaptação em tempo de desenvolvimento. Porém, o *framework* ainda não é capaz de realizar algumas tarefas importantes, como por exemplo, acessar as funções individuais dos diversos SADs ou canais, e não possui um sistema de

*plug-ins*.

Uma maneira de acessar esses recursos únicos, é através da reflexão. No Capítulo 2, onde foram apresentados alguns princípios da reflexão, verifica-se que é possível obter informações sobre os métodos de uma classe e executá-los dinamicamente. Utilizando esse recurso, é possível acessar as funções individuais de cada equipamento, implementadas através de métodos que não fazem parte da interface principal. Para isso o sistema pesquisa esses métodos e pergunta ao usuário o valor dos seus parâmetros, de maneira análoga ao exemplo da Listagem 2.4. Após a resposta do usuário, o sistema executa o método, acessando as funções específicas de cada sistema.

A utilização da reflexão possui um custo adicional de processamento, mas neste caso as propriedades que são modificadas por reflexão são acessadas apenas na fase de configuração do equipamento, onde o tempo de resposta não é um fator crítico. As funções que necessitam de maior velocidade são acessadas através da interface principal, por polimorfismo, que possui um custo computacional muito menor.

### 3.2.2 O Gerenciador dos *plug-ins*

A reflexão permite o acesso às funções únicas de cada SAD. Através dela, é ainda possível implementar um sistema de gerenciamento de *plug-ins* utilizando o carregamento dinâmico das classes. No *framework*, os *plug-ins* são na verdade bibliotecas que devem estar disponíveis em um diretório especificado pela aplicação. Estas bibliotecas contém classes derivadas da classe *SAD* do *framework* que permitem a comunicação com os equipamentos de aquisição de dados.

O sistema de gerenciamento facilita o carregamento dinâmico dessas classes e é implementado na classe denominada *FabricaSAD*. Esta classe tem a função de “fabricar” objetos que representem os diversos SADs conectados ao sistema. A classe *FabricaSAD* possui basicamente dois métodos estáticos. O primeiro é o *ProcurarDispositivos*, que pesquisa o sistema à procura de equipamentos de aquisição de dados. O segundo método, *CriarSAD*, gera uma instância de um objeto que representa um SAD específico.

A Figura 3.6 apresenta o diagrama de atividades do método *ProcurarDispositivos*.

Este método recebe como argumento um diretório, onde estão localizados os *plug-ins*. A primeira atividade do método é pesquisar todos os arquivos deste diretório a procura de bibliotecas. Quando uma biblioteca é encontrada, o método lista as classes presentes na biblioteca e verifica se elas correspondem a um SAD. Se a classe for um SAD, o método *ProcurarSAD* dessa classe é executado, retornando informações sobre os dispositivos daquela classe que estão conectados ao sistema. Estes dispositivos são adicionados a uma lista de dispositivos encontrados e o método continua a sua pesquisa até percorrer todos os arquivos. O método encerra então, a sua execução, retornando a lista com informações sobre os dispositivos encontrados.

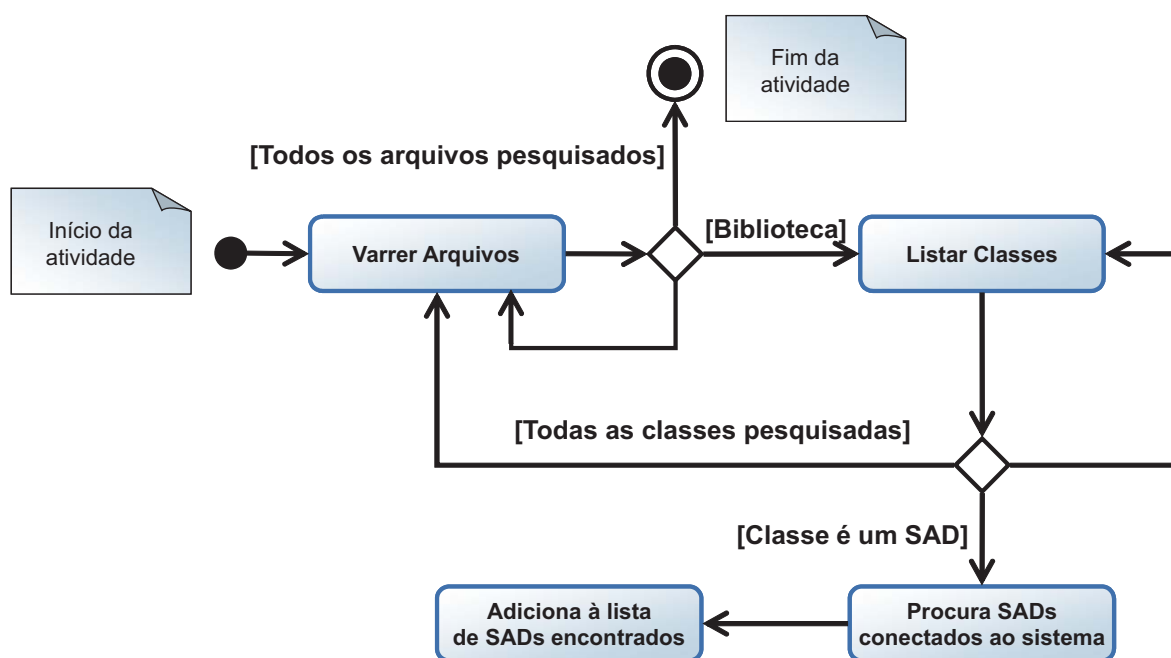


Figura 3.6: Diagrama de atividades do método *ProcurarDispositivos*.

O método *CriarSAD* recebe como parâmetro um dos objetos com informações sobre os SADs, gerado pelo método *ProcurarDispositivos*. Este objeto contém o nome da classe que é utilizada para manipular o SAD, a localização da biblioteca na qual a classe se encontra, além de conter informações necessárias para gerar uma instância dessa classe. O método utiliza essas informações para gerar uma instância da classe utilizando os recursos de reflexão descritos no Capítulo 2, retornando uma instância dessa classe que pode ser

utilizada pelo sistema de processamento.

### 3.3 A implementação do *framework*

O *framework* foi implementado na forma de uma biblioteca, utilizando a versão 2.0 da plataforma *.net*, recebendo o nome *DAQSystem*. A Figura 3.7 mostra o diagrama de classes gerado pela IDE.

*DAQSystem* implementa as classes descritas nas seções anteriores, modificando apenas a sua nomenclatura. Por exemplo, a classe *SAD* recebe o nome de *DAQ* no *framework*, já a classe *DAQFactory* representa a classe *FabricaSAD*. Além das classes descritas, existem outras classes auxiliares. Algumas delas são atributos que o sistema utiliza para identificar métodos especiais, outras modelam exceções que são utilizadas para indicar falhas em alguma parte do processo.

A classe *DAQDesconectadoException* é utilizada para gerar uma exceção caso o equipamento seja desconectado durante a aquisição. A classe *DAQException* é utilizada para gerar uma exceção no código em caso de alguma falha no processo. A classe fornece membros que informam detalhes sobre o erro ocorrido. A estrutura *DAQInfo* mantém as informações sobre os equipamentos conectados ao sistema e é utilizada pela classe *DAQFactory* para fornecer a lista de equipamentos encontrados e “fabricar” um novo *DAQ*. A classe *DAQConstructorAttribute* gera o atributo que será utilizado no desenvolvimento dos *plug-ins* para indicar qual o construtor da classe será utilizado pela “fabrica de objetos”. Por fim, a classe *DAQScanMethodAttribute* define o atributo que marca nas subclasses de *DAQ* qual o método utilizado para procurar por equipamentos conectados.

As funcionalidades únicas de cada equipamento são acessadas através das propriedades das classes derivadas de *DAQ*. Uma propriedade, na plataforma *.net*, é um recurso criado para permitir a alteração de campos de uma classe mantendo o encapsulamento. Em outras linguagens orientadas a objeto, como o C++, isso é feito através de dois métodos. Um método realiza a atribuição, recebendo um valor como parâmetro. Este valor é verificado antes de modificar o campo da classe, mantendo a consistência do objeto. O

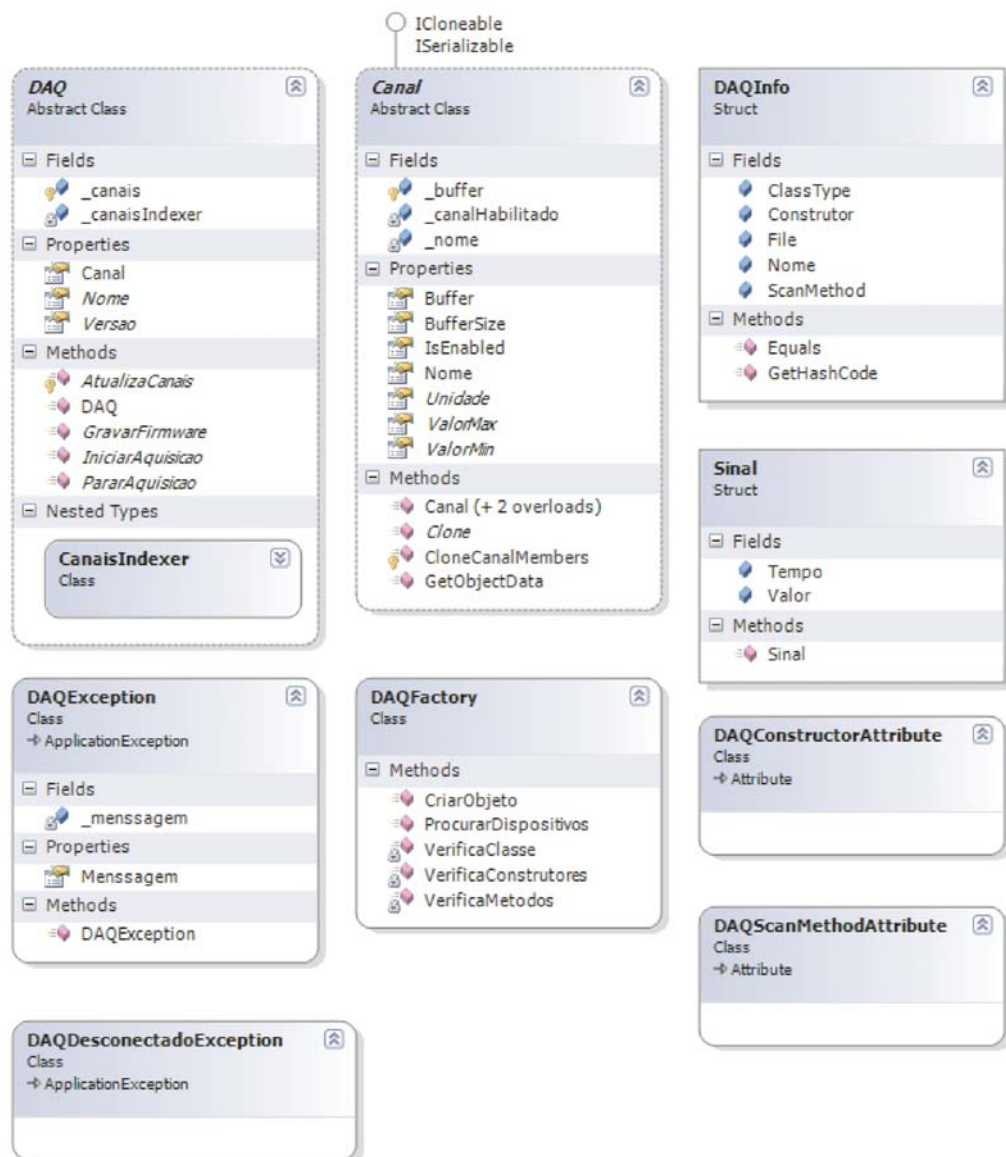


Figura 3.7: Diagrama de classes do *framework DAQSystem*.

outro método é utilizado para a obtenção do estado e, geralmente, retorna uma cópia do valor do membro. Na plataforma *.net*, esses dois métodos são associados em um único membro, simplificando a sintaxe de utilização.

A plataforma *.net* apresenta ainda mais uma facilidade ao utilizar as propriedades. Ela fornece um controle gráfico para visualização e edição de propriedades, chamado *PropertyGrid*. Este controle possui uma propriedade, denominada *SelectedObject*, que recebe um objeto e, através da reflexão, gera uma lista com todas as propriedades do objeto. Esta lista é exibida de maneira gráfica e os campos com os valores que podem ser editados. A Figura 3.8 mostra uma janela, na qual foi incluído um controle *PropertyGrid*. O objeto do qual o *PropertyGrid* está editando as propriedades é a sua janela. A propriedade *Text* é utilizada para obter ou modificar o título da janela. Dessa forma, ao alterar o valor da propriedade *Text*, observa-se a modificação do título da janela.

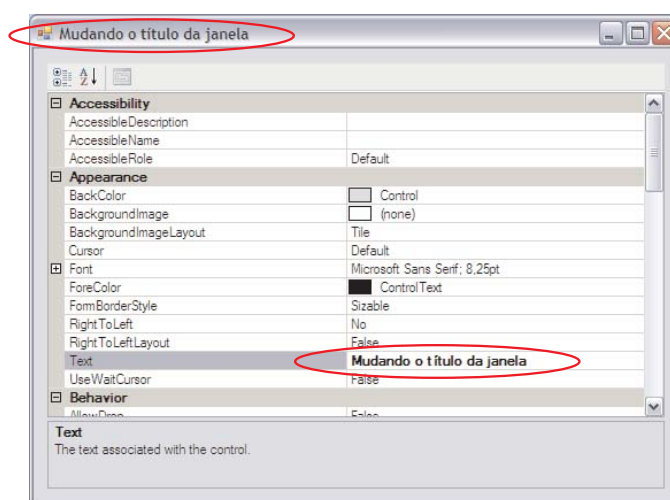


Figura 3.8: Exemplo de uma janela com um controle *PropertyGrid*. O controle está associado ao objeto que representa a janela, de forma que, ao alterar a propriedade *Text*, o título da janela é alterado.

### 3.4 Desenvolvendo aplicativos com o *framework*

A construção de um aplicativo que utilize o *framework* deve ser feita utilizando uma das linguagens da plataforma *.net*. A biblioteca *Biolab.DAQSystem.dll* deve ser adicionada ao projeto, permitindo acesso às classes do *framework*.

O primeiro passo é descobrir quais equipamentos estão conectados no computador. Para isso, é utilizado o método *ProcurarDispositivos* da classe *DAQFactory*. Este método retorna um vetor com as informações sobre os sistemas de aquisição conectados ao computador. Este vetor contém estruturas do tipo *DAQInfo*. Uma das informações contidas nesta estrutura é o nome do equipamento. Pode-se utilizar esse nome para gerar uma lista para usuário dos equipamentos disponíveis, e pedir para que ele escolha qual deles será utilizado. O código da Listagem 3.1 exemplifica essa operação. Nele, o nome dos equipamentos encontrados são exibidos em um controle *ComboBox* (Figura 3.9).

---

Listagem 3.1: Código para listar os equipamentos conectados ao sistema.

---

```
1 //Remove todos os items do ComboBox
2 cBoxEquipamentosEncontrados.Items.Clear();
3
4 //Obtém a lista com os equipamentos conectados ao sistema
5 //_daqList é um vetor de objetos DAQInfo
6 DAQInfo [] _daqList = DAQFactory.ProcurarDispositivos(@"c:\plugins");
7
8 //Adiciona os nomes dos equipamentos à caixa
9 //de combinação
10
11 foreach (DAQInfo daq in _daqList)
12 {
13     cBoxEquipamentosEncontrados.Items.Add(daq.Nome);
14 }
```

---

Após o usuário selecionar um dos equipamentos disponíveis, o sistema precisa criar um instância do objeto DAQ, referente ao *plug-in* do equipamento. Isso é realizado com uma única linha de código, utilizando o método *CriarObjeto*. Este método recebe como

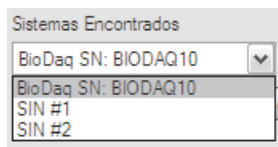


Figura 3.9: Caixa de combinação mostrando os equipamentos encontrados após executar o código da Listagem 3.1.

argumento um dos objetos *DAQInfo* gerados pelo método *ProcurarDispositivos* descrito anteriormente. A Listagem 3.2 exemplifica esta operação.

---

Listagem 3.2: Criando um objeto DAQ.

---

```

1 //Gera uma instância de uma das classes derivadas
2 //de DAQ. Para isso utiliza o método CriarObjeto
3 //de DAQFactory. O objeto DAQInfo, correspondente
4 //ao equipamento selecionado pelo usuário, é passado
5 //como parâmetro para o método
6 DAQ _myDaq = DAQFactory.CriarObjeto( _daqList [ cBoxEquipamentosEncontrados .
    SelectedIndex ] ) ;
```

---

Após gerar a instância de um DAQ, o sistema pode começar a utilizar os recursos do sistema de aquisição. Para obter informações sobre os canais conectados ao sistema, basta utilizar a propriedade *Canal*. Esta é uma propriedade indexada, e comporta-se como um vetor da linguagem. Dessa forma, é possível obter o número de canais que o equipamento possui através da propriedade *Length*, ou então, acessar a referência de um dos canais fornecendo um índice. O código da Listagem 3.3 demonstra como isso pode ser feito. Ele preenche um *ComboBox* com os nomes dos diversos canais disponíveis no equipamento, conforme mostrado na Figura 3.10.

---

Listagem 3.3: Utilizando a propriedade *Canal*.

---

```

1 //Apaga todos os itens existentes no ComboBox
2 cbxCanaIs.Items.Clear() ;
3 //Gera um laço que varre todos os canais ,
4 //observe a utilização da propriedade Length
```

```

5  for (int i = 0; i < _myDaq.Canal.Length; i++)
6  {
7      //Adiciona o nome do canal ao ComboBox
8      cbxCanais.Items.Add(_myDaq.Canal[i].Nome);
9  }

```

---

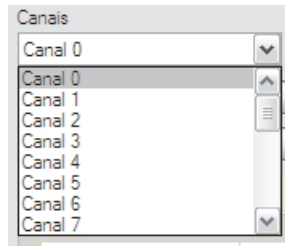


Figura 3.10: Exemplo de caixa de combinação mostrando os canais de aquisição disponíveis em um SAD. Os items da caixa foram gerados através da propriedade *Nome* de cada canal presente no objeto *\_daq*.

Um cuidado deve ser tomado ao utilizar essa propriedade, já que ela retorna uma cópia do objeto referente ao canal. Ao se modificar um dos elementos, modifica-se a cópia e não o objeto utilizado pelo *DAQ*. Portanto, após qualquer modificação, deve-se reenviar o objeto para o *DAQ* utilizando o mesmo índice. O código da Listagem 3.4 apresenta as duas situações, exemplificando como habilitar um dos canais do equipamento de maneira correta.

---

Listagem 3.4: Modificando um *Canal*.

---

```

1  //Maneira errada, não modifica o canal, modifica apenas a cópia.
2  _myDaq.Canal[0].IsEnabled = true;
3
4  //Maneira correta, realiza a modificação na cópia, e
5  //repassa para o DAQ.
6  Canal canal = _myDaq.Canal[0];
7  canal.IsEnabled = true;
8  _myDaq.Canal[0] = canal;

```

---

Para acessar as funções individuais de cada *DAQ* ou *Canal*, é utilizado um controle *PropertyGrid*. O objeto *DAQ* é passado para o *PropertyGrid* e suas propriedades são exibidas, permitindo sua edição. O mesmo pode ser feito com um dos canais do objeto, lembrando que após a edição, a cópia do objeto *Canal* deve ser repassada ao objeto *DAQ*. O código da Listagem 3.5 exemplifica esse processo. A Figura 3.11 mostra os controles exibindo as propriedades de um SAD e um canal.

Listagem 3.5: Modificando um *Canal*.

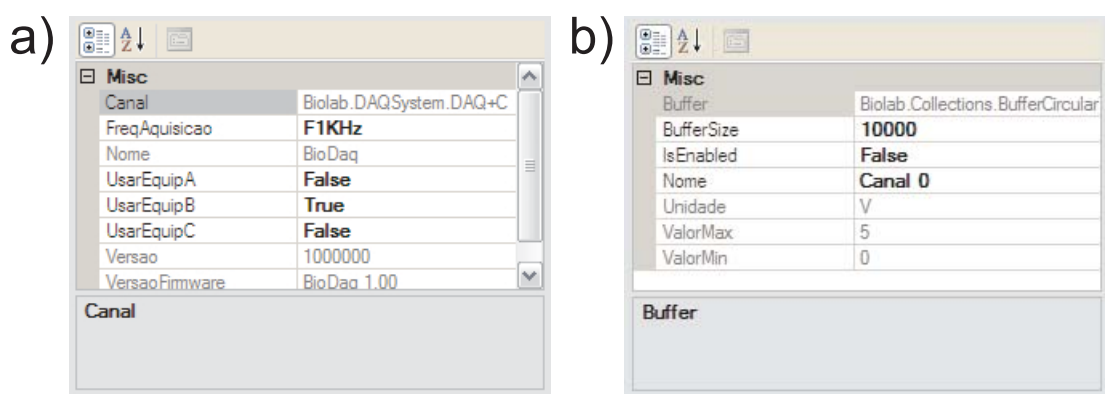
---

```

1  //Enviando o objeto DAQ para edição de suas propriedades
2  propertyGrid.SelectedObject = _myDaq;
3
4  //Enviando um dos canais para edição das propriedades
5  Canal canal = _myDaq.Canal[0];
6  propertyGrid.SelectedObject = canal;
7
8  //Após o usuário editar todas as propriedades deve realizar
9  //a seguinte chamada
10 _myDaq.Canal[0] = canal;

```

---

Figura 3.11: *PropertyGrid* exibindo as propriedades de um SAD (a) e um canal (b).

Para iniciar o processo de aquisição de dados, o sistema deve criar uma nova tarefa concorrente que realiza a coleta e processamento dos dados, e executar a chamada do método *IniciarAquisicao* do objeto *DAQ*. A tarefa de processamento pode obter os dados

do SAD através do *Buffer* de cada canal. O código da Listagem 3.6 demonstra esse processo, que se inicia com a modificação do valor da variável *\_estaAquisicionando* para verdadeiro. Esta variável é utilizada para controlar a tarefa de processamento. Quando é colocado o valor falso nessa variável, a tarefa de processamento encerra. A tarefa de processamento é criada (*\_consumidor*) utilizando os recursos de multitarefa da plataforma *.net* (DEITEL et al., 2001; AITKEN, 2002). Note que, apesar de criada, essa tarefa ainda não será executada. Neste momento, o sistema realiza a chamada do método *IniciarAquisicao* de *DAQ* e, por fim, inicia a tarefa de processamento.

---

Listagem 3.6: Iniciando a aquisição de dados.

---

```
1 //Variável utilizada para parar a tarefa _consumidor
2 _estaAquisicionando = true;
3
4 //Cria a nova tarefa
5 _consumidor = new Thread(new ThreadStart(this.Aquisicao));
6 try
7 {
8     //Inicia a aquisição do plugin
9     myDaq.IniciarAquisicao();
10    //Inicia a tarefa consumidora que realiza o processamento
11    //do aplicativo
12    _consumidor.Start();
13 }
14 //Se o sistema estiver desconectado, gera uma mensagem de
15 //erro.
16 catch (DAQDesconectadoException)
17 {
18     MessageBox.Show("Não existe nenhum equipamento conectado");
19 }
```

---

A tarefa *\_consumidor* deve retirar os dados contidos no *buffer* de cada canal e processá-los. A Listagem 3.7 apresenta um exemplo de como isso pode ser feito, onde o código inicia com um laço controlado pela variável *\_estaAquisicionando*, definida com o valor verdadeiro ao criar a tarefa. Todos os canais são então pesquisados a procura dos que

estão ativos. Quando um canal ativado é encontrado, utiliza o método *Remove* do *buffer* é chamado para obter as amostras que são processadas logo em seguida.

---

Listagem 3.7: Obtendo as amostras do SAD.

---

```
1 public void Aquisicao()
2 {
3     //Variável de controle da execução, quando for modificada
4     //para o valor falso, a tarefa termina.
5     while (_estaAquisicionando)
6     {
7         //Processa todos os canais
8         for(int i = 0; i < _myDaq.Canal.Length; i++)
9         {
10             //Obtém a referência do canal
11             Canal canal = _myDaq.Canal[i];
12             if(canal.IsEnabled)
13             {
14                 //Retira os valores do buffer
15                 Sinal[] valores = canal.Buffer.Remove();
16
17                 //Realiza algum processamento com o sinal
18                 Processa(Sinal);
19             }
20         }
21     }
22 }
```

---

Para parar o processo de aquisição deve-se executar o método *PararAquisicao* de *DAQ* e terminar a tarefa consumidora. O código da Listagem 3.8 exemplifica essa operação. Nele, o método *ParaAquisicao* é executado e, em seguida, o valor da variável *\_estaAquisicionando* é modificado para *false*. Quando isso ocorre, o fluxo de execução sai do laço principal da Listagem 3.7, terminado a tarefa.

---

Listagem 3.8: Parando a aquisição de dados.

---

```
1 //Para o DAQ
```

```
2 _myDaq.PararAquisicao();  
3 //Para o consumidor  
4 _estaAquisicionando = false;
```

---

Os métodos descritos acima fornecem a base para a construção de softwares de coleta e processamento de dados. O software pode utilizar essas classes de acordo com as suas necessidades. Por exemplo, um software que necessite obter dados de vários equipamentos diferentes poderia criar um vetor de objetos do tipo *DAQ* e utilizar as funções do framework para gerenciar os diversos equipamentos.

Para que qualquer software desenvolvido, utilizando o *framework*, realize suas funções, é necessário que ele possua um ou mais *plug-ins*. A próxima seção apresenta os detalhes da criação de *plug-ins* compatíveis com o *DAQSystem*.

### 3.5 Criando os *plug-ins*

A criação de *plug-ins* para o *DAQSystem* consiste no desenvolvimento de uma biblioteca da plataforma. Esta biblioteca deve conter uma ou mais classes derivadas de *DAQ* e *Canal*. Estas classes devem seguir algumas regras na sua construção.

As subclasses de *DAQ* devem:

- Conter um construtor que receba uma string. Este construtor será utilizado para gerar a instância da classe, e também deve conter o atributo *DAQConstructor*.
- Implementar todos os métodos abstratos da classe *DAQ*. A classe também deve implementar as propriedades abstratas herdadas.
- Deve possuir um método de pesquisa de dispositivos, e conter o atributo *DAQScanMethod*.
- Possuir o atributo *Serializable* e permitir salvar e recuperar o estado do equipamento em um arquivo.
- Utilizar propriedades para implementar as funções de configuração únicas do equipamento.

As subclasses de *Canal* devem:

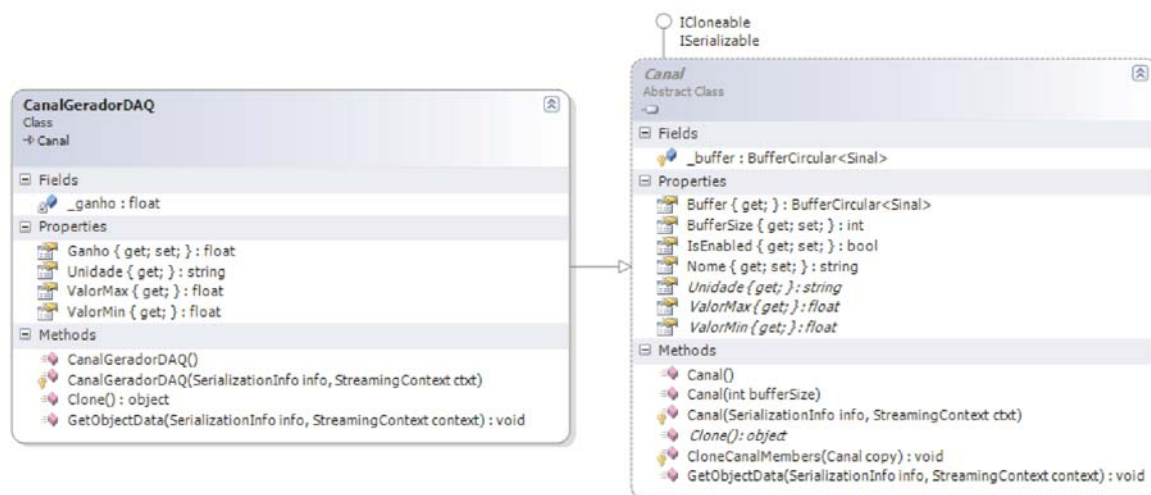
- Implementar as propriedades abstratas da classe *Canal*.
- Possuir o atributo *Serializable* e sobrescrever a implementação da interface *ISerializable*, herdada da classe *Canal*.
- Sobrescrever o método *Clone*, para implementar a interface *ICloneable*, herdada da classe *Canal*. Esta interface permite gerar cópias de um objeto e é utilizada pelo *framework*.
- Adicionar novas propriedades, com as configurações dos métodos únicos do canal.

Para ilustrar o processo de criação de um *plug-in*, será apresentado o exemplo de um *plug-in* que controla um SAD virtual. Este *plug-in* irá simular um SAD conectado no computador, gerando ondas senoidais.

O primeiro passo na criação do *plug-in* é criar uma classe que modele os seus diversos canais. Para isto basta criar uma classe derivada da classe *Canal*. A Figura 3.12 apresenta a classe *CanalGeradorDAQ* que herda de *Canal*. Esta classe irá modelar os canais específicos do equipamento virtual, seu código completo pode ser encontrado no CD que acompanha este documento.

O segundo passo é implementar as propriedades abstratas da classe *Canal*. Observe que a classe *CanalGeradorDAQ* sobrescreve as propriedades *Unidade*, *ValorMax* e *ValorMin* da classe *Canal*. Essas propriedades devem fornecer informações sobre o sinal que o canal está preparado para captar. A propriedade *Unidade* retorna uma *string* com a unidade de medida do sinal do canal. As propriedades *ValorMax* e *ValorMin* retornam os valores máximos e mínimos que o canal é capaz de medir. Por exemplo, se o canal estiver conectado a um sensor de temperatura, a propriedade unidade poderia retornar a *string* “°C”. Se o canal é capaz de medir temperaturas entre 0 e 100°C, as propriedades *ValorMin* e *ValorMax* devem retornar 0 e 100 respectivamente.

O próximo requisito é que o canal possua o atributo *Serializable* e implemente a interface *ISerializable*. Este atributo e esta interface estão relacionados com o sistema

Figura 3.12: Classe *CanalGeradorDAQ*.

de serialização da plataforma *.net*. Serializar um objeto é transformá-lo em um vetor de bytes para que sua representação possa ser enviada para um fluxo de dados. Um fluxo de dados pode ser um arquivo ou uma conexão de rede, ou seja, a serialização é utilizada para gravar um objeto em um arquivo em disco ou transmiti-lo por uma conexão de rede. Existem vários tutoriais (WILLIAMS, 2001; CHARTIER, 2002) e livros (DEITEL et al., 2001) que demonstram como implementar a serialização de objetos na plataforma *.net*.

A classe *CanalGeradorDAQ* possui o atributo *Serializable*. A implementação da interface *ISerializable* é realizada ao sobrescrever o método *GetObjectData*. A interface ainda exige que a classe possua um construtor que receba objetos do tipo *SerializationInfo* e *StreamingContext* como parâmetros. Detalhes desta implementação encontram-se no código disponível no CD que acompanha este documento.

A classe também sobrescreve o método *Clone* para implementar a interface *ICloneable* requerida pelo *framework*. Esta interface é utilizada para “clonar” um objeto, ou seja, gerar uma nova instância com a mesma representação do objeto. Detalhes sobre esta interface podem ser encontrados em diversos tutoriais (NESS, 2002) e na referência da plataforma (MICROSOFT, 2007b).

Por fim, a classe acrescenta suas propriedades únicas. No diagrama, tem-se a pro-

propriedade *Ganho* que simula um ganho programável no equipamento. Associada a esta propriedade está o campo *\_ganho* que irá armazenar o valor do ganho. O canal pode ter várias propriedades de acordo com o equipamento, sendo que cada uma deles configura um parâmetro do canal.

Após a criação do canal, é necessário desenvolver a classe derivada de *DAQ* que irá modelar o equipamento. O diagrama da Figura 3.13 mostra a classe *GeradorDAQ* que irá realizar esta função. O código completo desta classe também encontra-se no CD que acompanha este documento.

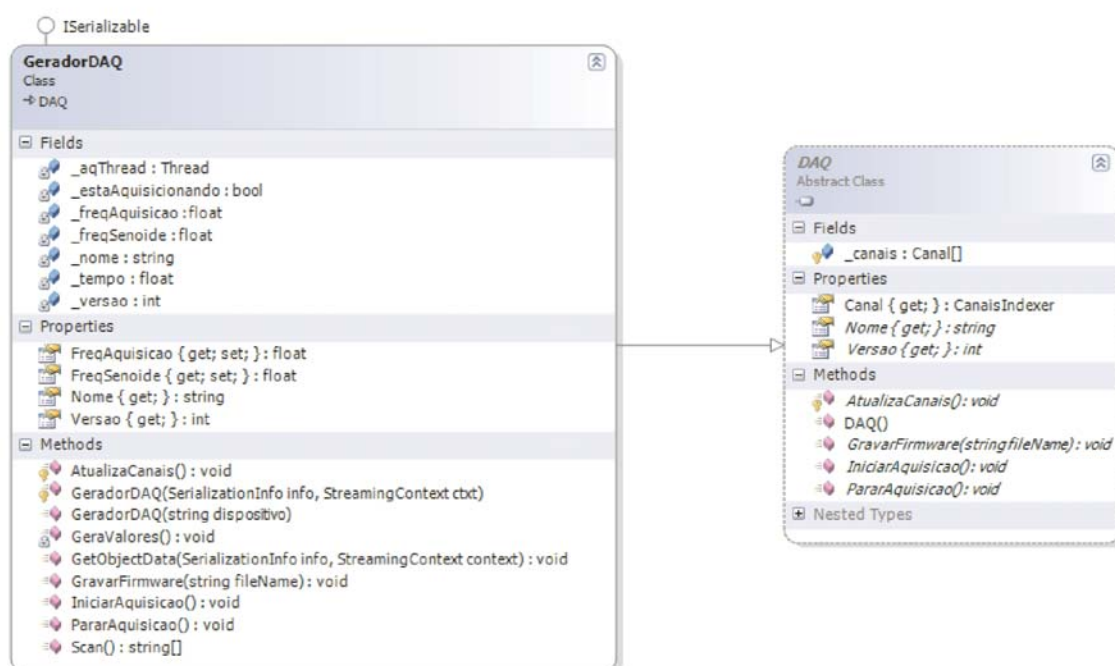


Figura 3.13: Classe *GeradorDAQ*.

Conforme mostra a Figura 3.13, a classe *GeradorDAQ* herda da classe *DAQ*. Geralmente, o primeiro passo no desenvolvimento de uma classe é o seu construtor. As classes que modelam os equipamentos do *plug-in* devem possuir um construtor especial, marcado com o atributo *DAQConstructor*. Este construtor recebe uma *string* como parâmetro, e cria a instância do objeto que representa o equipamento. Esta *string* é gerada pelo mesmo *plug-in* no seu método de pesquisa de equipamentos e pode conter informações sobre o

equipamento.

Como todo construtor, este deve gerar um valor inicial para as variáveis membro do objeto. Além dos membros incluídos na classe, este construtor deve gerar os valores da variável membro *\_canaís* herdado da classe *DAQ*. Esta variável é um vetor de objetos do tipo canal. O construtor então adiciona os seus canais a este vetor. Por exemplo, a classe *GeradorDAQ* preenche este vetor com objetos da classe *CanalGeradorDAQ*. Isto é possível devido à relação de herança entre a classe *CanalGeradorDAQ* e *Canal*. Mais detalhes podem ser obtidos no código do construtor contido no CD.

A classe também implementa as propriedades *Nome* e *Versao*, herdadas de *DAQ*. A propriedade *Nome* deve retornar uma *string* com o nome do equipamento enquanto que a propriedade *Versao* deve retornar um número com a versão do *plug-in*.

Após implementar as propriedades, é preciso implementar os métodos, também abstratos, herdados da classe *DAQ*. Estes métodos são *AtualizaCanais*, *GravarFirmware*, *IniciarAquisicao* e *PararAquisicao*.

O método *AtualizarCanais* é executado quando ocorre uma alteração em um dos canais da classe, e é utilizado para repassar as alterações para o equipamento. Para isso, a função deve pesquisar todos os canais à procura de alterações e, quando encontrá-las, enviar comandos para o equipamento, efetivando estas alterações. A classe *GeradorDAQ* não possui código nesta função por estar modelando um equipamento virtual.

O método *GravarFirmware* é utilizado quando o equipamento possui algum tipo de software embutido no hardware e permite a sua atualização. Este método recebe um arquivo com a nova versão do programa e utiliza o canal de comunicação para atualizar o *firmware* do equipamento. A maneira como isto é feita varia com cada equipamento. De maneira análoga ao método *AtualizarCanais*, a classe *GeradorDAQ* não possui código nesta função por estar modelando um equipamento virtual.

O método *IniciarAquisicao*, como o próprio nome indica, é utilizado para dar início ao processo de aquisição. Sua implementação requer que o método crie uma nova tarefa concorrente. Esta tarefa irá se comunicar com o equipamento, obter as amostras dos diversos sinais, convertê-las para a unidade final e adicioná-las aos *buffers* de seus respectivos ca-

nais. O código disponível no CD demonstra como criar essa tarefa concorrente. O código da tarefa concorrente encontra-se no método *GeraValores* que realiza as funções descritas acima, com exceção da comunicação com o equipamento e conversão para a unidade final, já que *GeradorDAQ* modela um equipamento virtual.

O método *PararAquisicao* tem como função enviar comandos ao equipamento para interromper o processo e terminar a tarefa concorrente, criada no método *IniciarAquisicao*.

Após a implementação dos métodos herdados da classe *DAQ*, deve-se criar o método estático procurar equipamentos conectados. Este método deve ser marcado com o atributo *DAQScanMethod*. Na classe *GeradorDAQ*, foi criado o método *Scan* para realizar esta função, este método retorna um vetor de *strings* que serão utilizadas pelo construtor para gerar as instância da classe. A classe *GeradorDAQ* sempre retorna duas *strings*, simulando a presença de dois equipamentos no computador.

Por fim, a classe deve possuir o atributo *Serializable* e permitir a sua serialização. No caso da classe *GeradorDAQ*, isso ocorre implementando a interface *ISerializable* de maneira análoga à classe *CanalGeradorDAQ*.

Por fim, a classe deve implementar as suas funcionalidades únicas através de propriedades. A classe *GeradorDAQ* cria duas propriedades: *FreqAquisicao* e *FreqSenoide*. A propriedade *FreqAquisicao* define o intervalo de tempo entre duas amostras, enquanto que a propriedade *FreqSenoide* define a frequência do sinal senoidal gerado pelo equipamento virtual. Elas fazem isso alterando variáveis membros da classe que são utilizadas pela função *GeraValores* da tarefa de aquisição.

O próximo capítulo apresenta a construção de um sistema de visualização de sinais, envolvendo desde o equipamento até o software, utilizando o *framework DAQSystem*.