

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/353273690>

ERCEMAPI 2007 versao final completa com isbn

Book · July 2007

CITATIONS

0

READS

357

3 authors:



Bernadette Farias Lóscio
Federal University of Pernambuco

76 PUBLICATIONS 605 CITATIONS

[SEE PROFILE](#)



Pedro Porfirio Muniz Farias
Universidade de Fortaleza

69 PUBLICATIONS 176 CITATIONS

[SEE PROFILE](#)



Raimir Holanda Filho
Universidade de Fortaleza

96 PUBLICATIONS 547 CITATIONS

[SEE PROFILE](#)

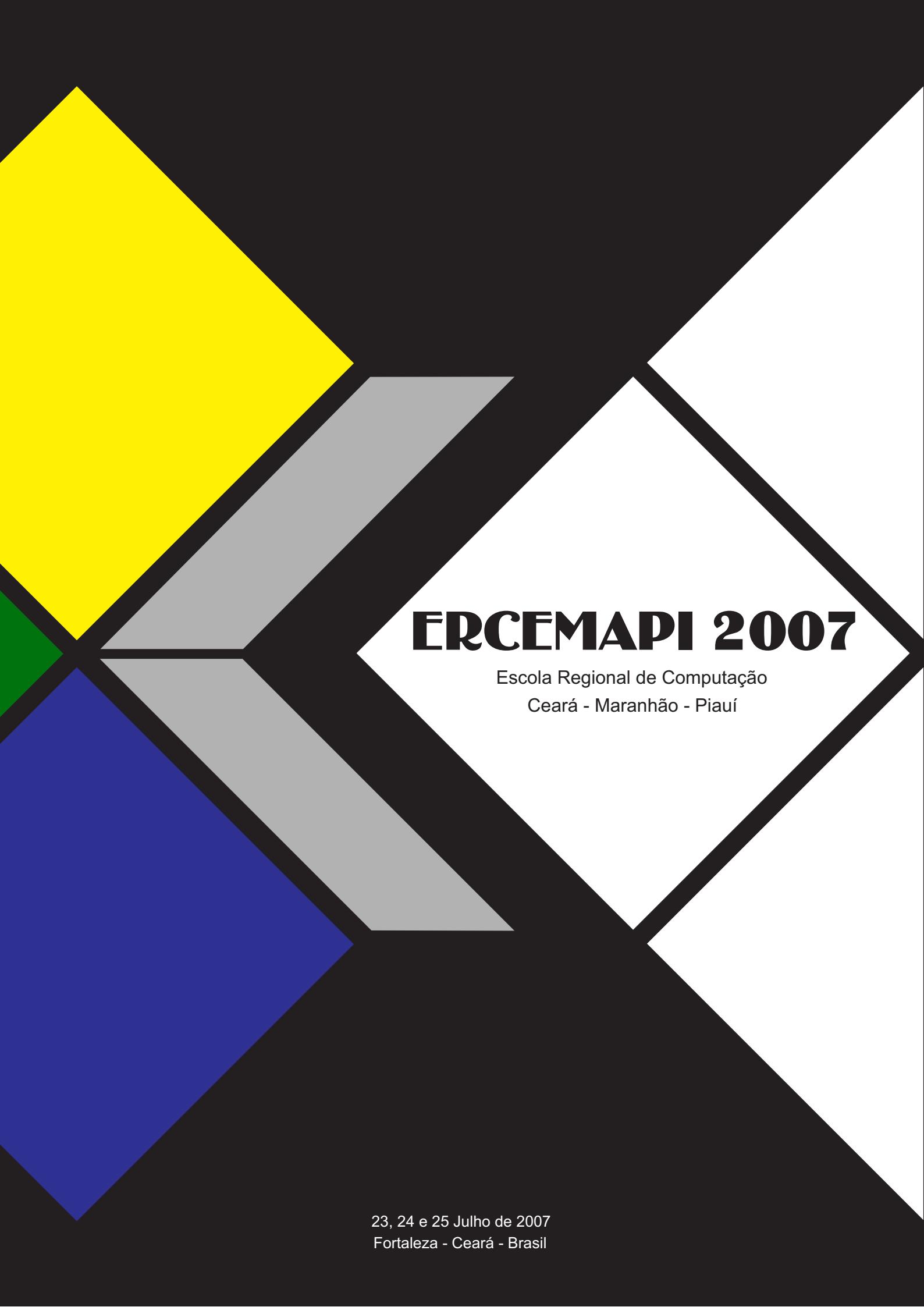
Some of the authors of this publication are also working on these related projects:



Semantic Based information Retrieval [View project](#)



Search-based Testing [View project](#)



ERCEMAPI 2007

Escola Regional de Computação

Ceará - Maranhão - Piauí

23, 24 e 25 Julho de 2007
Fortaleza - Ceará - Brasil



Escola Regional de Computação – Ceará, Maranhão e Piauí
Fortaleza, 23 a 25 de julho de 2007

LIVRO – TEXTO

Edição

Sociedade Brasileira de Computação

Organizadores

Bernadette Farias Lóscio
Pedro Porfírio Muniz Farias
Raimir Holanda Filho

Promoção

Sociedade Brasileira de Computação

Patrocínio

Secretaria de Ciência e Tecnologia do Estado de Ceará (SECITECE)
Fundação Cearense de Apoio ao Desenvolvimento Científico e Tecnológico (FUNCAP)
Fundação de Amparo à Pesquisa do Estado do Maranhão (FAPEMA)
Instituto Atlântico

Realização

Universidade de Fortaleza (Unifor)
Universidade Federal do Ceará (UFC)
Centro Federal de Educação Tecnológica do Ceará (CEFET-CE)
Faculdade Integrada do Ceará (FIC)

Apoio

Universidade Estadual do Ceará (UECE)
Universidade Federal do Piauí (UFPI)
Universidade Federal do Maranhão (UFMA)

Copyright © de Sociedade Brasileira de Computação
Todos os direitos reservados

Xxxxx Escola Regional de Computação - Ceará , Maranhão e Piauí
(1.:2007:Fortaleza,CE)
Anais / I Escola Regional de Computação - Ceará , Maranhão e Piauí –
ERCEMAPI; Editores : Bernadette Farias Lóscio, Pedro Porfírio Muniz
Farias, Raimir Holanda Filho – Fortaleza : Sociedade Brasileira de
Computação, 2007

287p.; il.

ISBN: 978-85-7669-161-7

1. Ciência da Computação. I. Bernadette Farias Lóscio, II Pedro
Porfírio Muniz Farias, Raimir Holanda Filho

CDU : xxxxxxxxxxxxxxxx

Boas Vindas à ERCEMAPI 2007

É com grande satisfação que apresentamos este livro-texto da primeira ERCEMAPI (Escola Regional de Computação – Ceará, Maranhão e Piauí) que se realiza em Fortaleza no Campus da UNIFOR.

As Escolas Regionais de Computação (ERC), promovidas pela SBC (Sociedade Brasileira de Computação) através de suas secretarias regionais, são eventos sem fins lucrativos que têm como objetivos principais:

1. Disseminar o conhecimento científico acerca de temas emergentes e relevantes ligados a computação.
2. Incentivar a produção de textos didáticos e técnicos de alta qualidade.

A primeira ERCEMAPI está sendo organizada pela UNIFOR, UFC, CEFET-CE e FIC, conta com os patrocínios da SECITECE, FUNCAP, FAPEMA e Instituto Atlântico e com o apoio da UECE, UFMA e UFPI.

Nesta edição, a ERCEMAPI, conta com mini-cursos elaborados por dez professores doutores ligados a Universidades da Região. O leitor encontrará aqui os textos de cada um dos mini-cursos ministrados.

Cumpre salientar o elevado espírito de colaboração com que preside a realização da ERCEMAPI. Além da colaboração das instituições citadas ressalte-se a colaboração pessoal dos professores, que ministram os cursos sem remuneração, bem como dos diversos alunos que colaboraram na organização do evento.

Finalmente, gostaríamos de expressar formalmente o nosso agradecimento a todos os que contribuíram para tornar a ERCEMAPI um evento de sucesso.

Fortaleza, Julho de 2007

Pedro Porfírio Muniz Farias
Bernadette Farias Lóscio
Raimir Holanda Filho

ERCEMAPI 2007

I Escola Regional de Computação - Ceará, Maranhão e Piauí

Comitê de Organização

Bernadette Farias Lóscio (UFC)

Flavio Horacio Souza Vieira (UNIFOR)

Itamar de Souza Lima (CEFET-CE)

Milton Escossia Barbosa Neto (FIC)

Pedro Porfírio Muniz Farias (UNIFOR)

Raimir Holanda Filho (UNIFOR)

Sumário

Capítulo 1 - Administração Avançada de Redes	1
Raimir Holanda Filho (Unifor) e José Everardo Bessa Maia (UECE)	
Capítulo 2 - Teste de Software	25
Pedro de Alcântara dos Santos Neto (UFPI) (UFPI), Francisco Vieira de Sousa (UFPI) e Rodolfo Resende (UFPI)	
Capítulo 3 - Desenvolvimento de Software utilizando Integração Contínua ...	53
Francisco Nauber Bernardo Góis (Serpro), Rafael Braga de Oliveira (Serpro) e Pedro Porfírio Muniz Farias (Unifor)	
Capítulo 4 - Introdução a Engenharia Dirigida por Modelos	81
Denivaldo Lopes (UFMA)	
Capítulo 5 - Introdução aos Padrões de Software.....	111
Jerffeson Teixeira de Souza (UECE) e Tarciane de Castro Andrade (UECE)	
Capítulo 6 - Introdução à Programação em TV Digital Interativa.....	143
Cidcley Teixeira de Souza (CEFET-CE)	
Capítulo 7 - Processamento de Consultas em redes de Sensores sem Fio	173
Angelo Brayner (Unifor) e Aretusa Lopes (Unifor)	
Capítulo 8 - Programação Funcional Usando Haskell	203
Francisco Vieira de Souza (UFPI) e Pedro de Alcântara dos Santos Neto (UFPI)	
Capítulo 9 - Programação Paralela em Clusters de Multiprocessadores com MPI e Open MP	233
Francisco Heron de Carvalho Junior (UFC)	
Capítulo 10 - XML: Conceitos e Aplicações	265
Bernadette Farias Lóscio (UFC) e Fernando Cordeiro Lemos (UFC)	

Capítulo

1

Administração Avançada de Redes

Raimir Holanda Filho e José Everardo Bessa Maia

Abstract

This book chapter presents some network administration concepts that are not covered in traditional documents related to this area. Initially, we make a review of the main tasks of network administration, emphasizing traffic management. Then, we show a traffic measurement infrastructure for large networks which supports advanced applications of administration. Considering the importance of the traffic pattern knowledge to network administrators, we describe two applications of traffic administration: attack and P2P traffic detection. These tasks are part of a major activity named traffic classification.

Resumo

Este capítulo apresenta aspectos de administração de redes que não são normalmente tratados em textos tradicionais do assunto. Inicialmente, é feita uma revisão das principais tarefas de administração de redes, com ênfase em gerência de tráfego. Em seguida, apresenta-se uma infra-estrutura de medição de tráfego para grandes redes que suporta aplicações avançadas de administração. Considerando a importância do conhecimento do perfil do tráfego para o administrador da rede, são descritas duas aplicações de administração de tráfego: detecção de tráfegos de ataque e de P2P. Essas duas tarefas são parte de uma atividade maior chamada classificação de tráfego.

1.1. Introdução

Administração de redes e sistemas é um ramo da computação que trata da gerência operacional. As tarefas da administração de redes podem todas ser vistas como atividades de uma ou mais das 5 áreas funcionais de gerência: falhas, desempenho, configuração, contabilização e segurança.

A gerência de falhas trata da detecção, isolamento e correção de condições anormais de operação. A gerência de configuração permite identificar, controlar, coletar e prover dados para a gerência dos objetos da rede. A gerência de performance fornece

as facilidades para avaliar o comportamento dos elementos da rede. A gerência de segurança trata dos aspectos de autenticação e autorização de uso e a gerência de contabilização trata do registro dos recursos utilizados.

Este capítulo estará concentrado em medição e administração de tráfego. Mais especificamente apresenta uma infra-estrutura de medição e a aplicação de técnicas de agrupamento e discriminantes estatísticos para classificação de tráfego. Administração de tráfego é uma atividade transversal a todas as áreas funcionais descritas anteriormente.

1.2. Classificação de Tráfego

Classificação e identificação do tráfego Internet por tipo de aplicação constitui-se em uma importante atividade em administração de redes. O conhecimento do perfil do tráfego é de uma grande relevância para atividades tais como: monitoramento de falhas, priorização de tráfego, planejamento de rede, qualidade de serviço (QoS – *Quality of Service*) e segurança.

Muitos esforços têm sido dedicados a gerência de redes de computadores e a identificação do tráfego que flui através da rede. A identificação e monitoramento do tráfego são importantes para a melhor utilização dos recursos das redes de computadores.

Pesquisas, relacionadas à gerência de segurança em redes de computadores, têm sido alvo de grande interesse nos últimos anos. Este fato decorre do aumento considerável com que as atividades de ordem pessoal, empresarial e governamental dependem das redes de computadores. Um ataque a uma rede de computadores pode implicar em diferentes níveis de ameaças, desde a perda de privacidade até enormes prejuízos de ordem financeira. Um ataque pode ser considerado, portanto, como a utilização de uma determinada rede com o propósito de comprometer a segurança das informações armazenadas ou transportadas nesta rede.

O controle do acesso aos recursos de uma rede de computadores permite uma maior disponibilidade e confiabilidade da rede, além de minimizar os transtornos causados aos usuários. Gerenciar a segurança de uma rede de computadores permite controlar o acesso aos recursos desta rede e oferecer um aumento da qualidade dos serviços ofertados aos usuários.

Neste capítulo, tratamos o problema da gerência de segurança através da identificação de fluxos de ataque presentes no tráfego Internet. Consideramos que a identificação da presença de ataques de forma precisa e confiável consiste no primeiro passo em direção a uma consistente gerência de segurança. Existem várias abordagens na literatura para identificação de tráfego de ataques, dentre elas, detecção baseada em assinaturas, detecção baseada em comportamento, anomalias e propriedades estatísticas multivariadas.

Uma segunda classe de tráfego que cresce constantemente na Internet é o tráfego P2P. Aplicações P2P surgem a cada dia: redes de distribuição de conteúdo, sistemas de processamento compartilhado, entre outros. Por isso, no trabalho de classificação que descrevemos a seguir, este é o segundo tipo de tráfego, além de ataques, que escolhemos para implementar um algoritmo de detecção.

1.3. Infra-estrutura de Medição

A RNP, através do grupo de trabalho de medições (GT-Medições <http://www.rnp.br/pd/gts2004-2005/medicoes.html>), especificou, desenvolveu e implantou uma arquitetura para a medição de desempenho orientada a serviços, denominada piPEs-BR [Sampaio et al. 2006], seguindo os princípios da arquitetura piPEs (colocar referência).

A arquitetura piPEs-BR é composta por um conjunto de módulos que visam contemplar funcionalidades de teste, armazenamento, agendamento, autorização, interface e detecção/aconselhamento. Estes módulos interagem entre si através do uso de serviços Web, fornecendo e solicitando serviços com funcionalidades específicas e bem definidas. Além dos módulos, a arquitetura prevê interações com três tipos de usuários: usuários finais, avançados e administradores do ambiente. A Figura 1.1 apresenta a arquitetura, seus módulos e usuários. Neste capítulo apresentamos um projeto que propõe o desenvolvimento de duas aplicações que interagem com a arquitetura piPEs-BR na condição de usuários avançados conforme o bloco destacado a esquerda da figura abaixo.

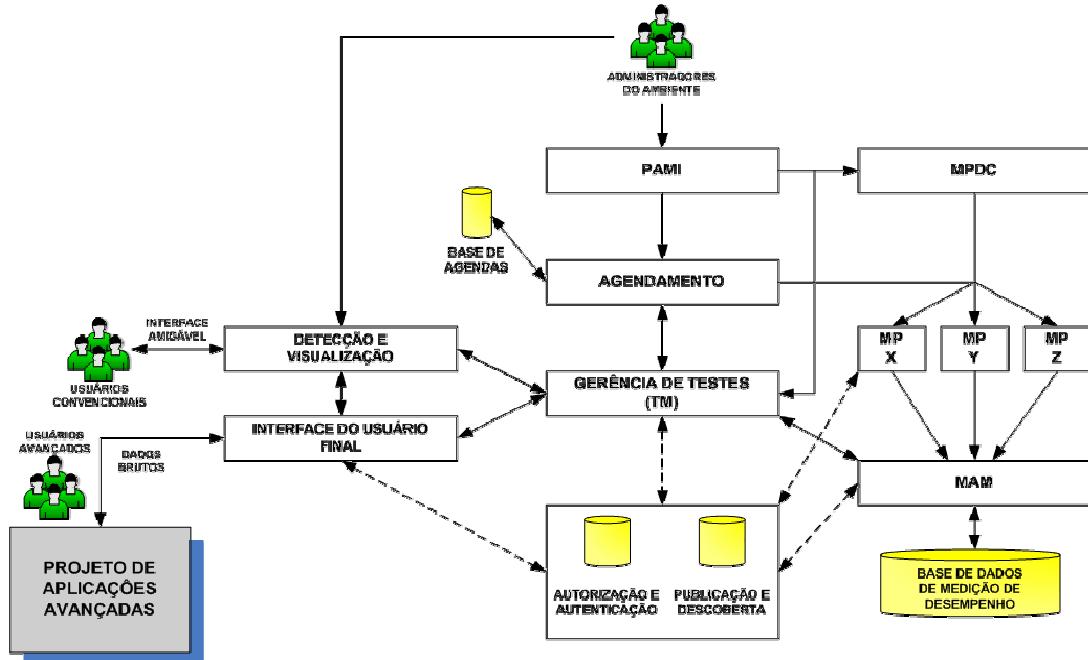


Figura 1.1. Arquitetura piPEs-BR

As aplicações avançadas que utilizarão a infra-estrutura do piPEs-BR são: identificação de aplicações e alocação dinâmica de banda passante. A Figura 1.2 apresenta o diagrama de blocos do projeto proposto e sua interação com a arquitetura piPEs-BR.

O contexto desta proposta é composto inicialmente por três módulos assim definidos: Pré-processamento, Identificação de Aplicações e Alocação dinâmica de banda passante (BandWidth - BW).

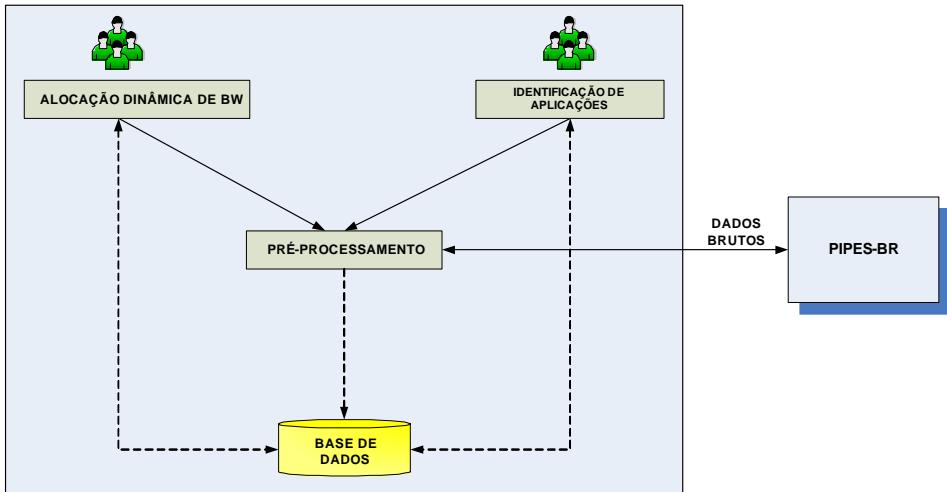


Figura 1.2. Interação das Aplicações Avançadas com a Arquitetura piPEs-BR

A interação com a arquitetura piPEs-BR se dará através da interface de usuários avançados, conforme mostrado na Figura 1.1. A partir dessa interface, são disponibilizados os dados relativos à coleta do tráfego que alimentará o sistema de aplicações. A primeira etapa no desenvolvimento das aplicações consiste no pré-processamento dos dados brutos com o objetivo de otimizar o processamento das aplicações. Para tanto, deve-se acessar os dados coletados através da arquitetura piPEs-BR conforme apresentado na Figura 1.2. Neste pré-processamento, é realizada uma seleção do conjunto de dados que são relevantes para as nossas análises.

Com o contínuo crescimento do tráfego em redes de backbone, faz-se necessária a implementação de um ambiente que, confiavelmente, identifique e classifique cada tipo de aplicação. A identificação das aplicações em um tráfego é necessária para diversos fins, incluindo operações de gerência de redes, aplicações específicas de engenharia de tráfego, contabilidade (*accounting*), segurança, planejamento de capacidade (*capacity planning*), provisionamento, diferenciações de serviço e redução de custos.

Existe, portanto, um esforço de pesquisa para identificação e classificação das aplicações, as quais têm demandado uma análise dos dados mais aprofundada.

Para cada tipo de aplicação, surge a necessidade de fornecer diferentes níveis de qualidade de serviço (QoS). Este fato é decorrente dos seguintes fatores:

- Os requisitos de cada aplicação variam (banda de transmissão de dados, atrasos, etc.);
- As importâncias das aplicações variam de acordo com as organizações. Por exemplo, em uma empresa, transações de banco de dados são consideradas críticas, de alta prioridade, enquanto tráfego web tem menos importância;
- Devido aos recursos serem limitados, existe a necessidade de otimizar seu uso para garantir um bom desempenho para as aplicações.

A abordagem comumente usada para identificação de aplicações em uma rede IP (*Internet Protocol*) consiste em associar determinado tráfego com um determinado número de porta TCP (*Transmission Control Protocol*) ou UDP (*User Datagram Protocol*). Esta abordagem tem se tornado inadequada com o passar do tempo, pois:

- a) Algumas aplicações utilizam erroneamente as portas tornando errônea também sua classificação;
- b) Na IANA (*Internet Assigned Numbers Authority*), algumas aplicações não têm portas definidas, como é o caso do Napster e do Kazaa;
- c) Uma aplicação pode usar outras portas para burlar as restrições de controle de acesso, como servidores web em portas diferentes da 80, que é restrita a usuários com privilégios na maioria dos sistemas operacionais;
- d) O uso de *firewalls* e de bloqueios de aplicações desconhecidas tem feito com que aplicações, como é o caso do ICQ, busquem portas que estão aceitando conexões ou utilizem a própria porta 80 para terem acesso a uma determinada rede;
- e) Ataques via *trojans* (cavalos de tróia) geram um grande volume de tráfego em uma determinada porta que realmente não estão associadas com aplicações válidas;
- f) Identificação de aplicações via acesso a carga útil (*payload*) é altamente desaconselhável;
- g) Um percentual considerável do tráfego atual utiliza criptografia.

Portanto, identificar e medir tal tráfego constitui uma tarefa de difícil solução, se métodos baseados unicamente em portas TCP forem utilizados. Por outro lado, métodos baseados na leitura de *payloads* são indesejáveis por questão de privacidade e muito do tráfego hoje está sendo criptografado, aumentando a dificuldade de acesso às informações das mensagens. Nossa proposta consiste em realizar essa identificação baseando-se na caracterização do tráfego, modelando o comportamento de cada aplicação e mapeando esse comportamento em função dos fluxos de pacotes e cabeçalhos TCP/IP.

O módulo de alocação dinâmica de BW tem como objetivo encontrar soluções para os problemas de exigências de banda passante em alguns ambientes de rede. O atual modo de funcionamento da Internet não permite oferecer garantias com relação à banda passante efetiva, sendo, portanto, necessário encontrar maneiras alternativas para fornecer melhores garantias de QoS.

As propostas de alocação de banda passante devem lidar com dois tipos de problemas majoritários: ou a banda alocada para um usuário final é insuficiente para escoar o volume de tráfego que está passando pela sua rede em um dado momento ou, por outro lado, a banda passante pode estar sendo subutilizada (*overprovisioning*). Caso a banda seja insuficiente, a qualidade do serviço fornecido ao cliente no momento do congestionamento cairá, pois muitos pacotes serão perdidos. Caso a banda passante esteja sendo subutilizada, o cliente terá prejuízo pois estará pagando ao seu provedor por um recurso que ele não utiliza plenamente. Para resolver estes dois problemas, o projeto ora apresentado propõe a utilização de mecanismos de banda passante sob demanda - BoD (*Bandwidth on Demand*).

Os algoritmos de alocação podem ser reativos, através dos quais a banda é alocada apenas seguindo as variações de uso do enlace, ou proativos, onde tenta-se prever a utilização do enlace no próximo intervalo de tempo.

Também se percebe uma classificação entre os métodos de implementação de BoD. Alguns métodos, para prover a largura de banda em um enlace, utilizam-se simplesmente da capacidade de transmissão dos meios de comunicação; um exemplo é o tipo de BoD outrora usado nas redes digitais de serviços integrados (<http://www.azinet.com/azinet/isdninfo.htm>). Outros métodos fazem um controle diferenciado da largura de banda: existe uma capacidade física de transmissão de dados no enlace que é constante, mas há uma limitação “lógica” (em geral feita usando-se políticas de fila) que não permite que a largura de banda “lógica” seja ultrapassada.

Inicialmente, a limitação “lógica” não parece fazer muito sentido. Por que não se usa toda a capacidade que o meio dispõe? É neste ponto onde entra o fator de QoS. Ao limitar a banda “logicamente”, o ISP (*Internet Service Provider*) mantém um controle maior sobre as necessidades de seus clientes, provendo melhores garantias de taxas de vazão (*throughput*). Caso haja problemas de disponibilidade de recursos, o ISP pode rejeitar um possível aumento de largura de banda de um cliente para continuar provendo um serviço de qualidade ao conjunto de seus outros clientes.

1.4. Detecção dos tráfegos de Ataque e P2P

Nesta seção, inicialmente, iremos descrever os *traces* utilizados para validar nossa proposta e em seguida a metodologia aplicada na detecção dos tráfegos de ataque e P2P. As subseções finais relatam os resultados obtidos na etapa de detecção para ambos os tipos de tráfego.

1.4.1. Descrição dos *Traces*

Para a detecção dos tráfegos de ataque e P2P utilizamos uma abordagem baseada em fluxo. Os fluxos são identificados como uma seqüência de pacotes que apresentam o mesmo conjunto de valores contidos nos seguintes campos do cabeçalho TCP/IP: endereço IP de origem, endereço IP de destino, porta TCP de origem, porta TCP de destino e tipo de protocolo.

Qualquer trabalho relacionado com a identificação de tráfego requer a utilização de dados. A disponibilidade de *traces* (arquivos de coleta de fluxos TCP/IP) previamente identificados é uma parte substancial do trabalho a ser realizado. Os dados utilizados aqui são dados reais e foram disponibilizados em [Moore et al. 2005]. Os *traces* foram utilizados nas pesquisas de [Moore e Papagiannaki 2005], [Moore et al 2005], [Moore e Zuev 2005] e [Auld e Moore 2007]. Esses dados foram coletados de uma rede com aproximadamente 1.000 usuários conectados a Internet através de uma conexão *full-duplex* Gigabit Ethernet e referem-se a um período de 24 horas. Foi gerado um conjunto de 10 arquivos sendo cada um referente a um período de 1.680 segundos (28 minutos), e disponibilizados para a comunidade científica. O método de coleta dos dados é descrito em [Moore et al. 2003].

As análises partem de dados previamente processados e apresentados em [Moore et al. 2005]. Neste pré-processamento, para cada fluxo coletado foi identificada a aplicação na qual ele está associado. A tabela 1.1. mostra as aplicações encontradas nos *traces* e suas classes correspondentes.

Tabela 1.1. Tipos de aplicações contidas em cada classe de tráfego.

Classes	Aplicações presentes nos <i>traces</i>
BULK	ftp
DATABASE	postgres, sqlnet oracle, ingres
INTERACTIVE	ssh, klogin, rlogin, telnet
MAIL	imap, pop2/3, smtp
SERVICES	X11, dns, ident, ldap, ntp
WWW	www
P2P	KaZaA, BitTorrent, GnuTella
ATTACK	Internet worm and virus attacks
GAMES	Half-Life
MULTIMEDIA	Windows Media Player, Real

Ao conjunto de fluxos foi aplicado um complexo procedimento para aumentar a sua confiabilidade na detecção precisa de cada aplicação. O procedimento proposto em [Moore e Papagiannaki 2005] e utilizado em [Moore et al 2005], [Auld e Moore 2007] e [Moore e Zuev 2005], consiste em análise de portas, assinaturas, protocolos, análise dos primeiros 1Kbytes de cada fluxo de dados, análise das estações acessadas através do DNS (*Domain Name Service*) de cada estação e finalmente na intervenção humana nos fluxos os quais não conseguiram ser classificados pelos procedimentos acima descritos. Esta tarefa resultou em uma precisão de identificação das aplicações de 99,99%. Na tabela 1.2 temos uma descrição geral dos nove métodos aplicados ao conjunto de *traces*.

Tabela 1.2. Métodos de identificação aplicados nos fluxos.

Métodos de Identificação
I. Classificação das portas
II. Cabeçalho dos pacotes
III. Assinatura do pacote
IV. Protocolo do pacote
V. Assinatura do primeiro Kbyte
VI. Protocolo do primeiro Kbyte
VII Busca de fluxos de protocolos específicos
VIII. Protocolos de todos os fluxos
IX. Histórico das estações acessadas

Além das categorias de aplicações, durante o pré-processamento foi gerado para cada fluxo dos 377.526 um conjunto de estatísticas relacionadas ao fluxo que em [Moore et al. 2005] são chamados discriminantes. Um total de 249 discriminantes foi gerado incluindo estatísticas simples sobre o tamanho do pacote e o tempo entre pacotes, e informações derivadas do protocolo de transporte TCP tais como contadores de pacotes *Syn* e *Ack*. Algumas das 249 estatísticas geradas estão descritas na tabela 1.3.

Tabela 1.3. Algumas das 249 variáveis examinadas de cada fluxo de dados.

Campos	Descrição
Variável 1	Porta de origem
Variável 2	Porta de destino
Variável 17	mínimo do total de bytes do pacote IP
Variável 45	Número de todos os pacotes com no mínimo um byte de dados
Variável 123	RTT - tempo de estabelecimento da conexão (cliente-servidor)
Variável 158	Máximo de bytes no pacote ethernet
Variável 159	Variância dos bytes no pacote ethernet
Variável 165	Máximo do total de bytes no pacote IP
Variável 166	Variância do total de bytes no pacote IP
Variável 211	Quantidade de tempo gasta em transferência de dados
Variável 212	Duração da conexão

As informações estatísticas geradas foram derivadas a partir das informações contidas nos cabeçalhos dos pacotes enquanto que na definição da classe da aplicação foi utilizada uma análise baseada em conteúdo. Portanto, nossas análises têm como

ponto de partida estes dados pré-processados nos quais para cada fluxo foram gerados um conjunto de estatísticas e uma classe que define a aplicação.

O uso de traces pré-classificados é um avanço na precisão dos métodos que buscam a identificação do tráfego Internet, pois conseguem classificar com maior exatidão todas as aplicações presentes nos *traces*. Conforme apresentado em [Auld e Moore 2007] e [Moore e Papagiannaki 2005] o uso de dados não pré-classificados leva a uma precisão máxima de 50%-70% do total de fluxos.

1.4.2. Metodologia

A metodologia aplicada a este trabalho constitui-se em duas etapas: seleção dos discriminantes e formação dos agrupamentos de fluxos de pacotes. A seleção dos discriminantes constitui uma das etapas mais importantes e difíceis no processo de identificação de componentes de tráfego.

1.4.2.1. Seleção de Discriminantes

A tarefa de identificação de tráfego é de fato uma tarefa de classificação. A etapa de identificação dos discriminantes que serão utilizados na fase de classificação é provavelmente a de maior importância. A qualidade da classificação está diretamente relacionada com os discriminantes escolhidos para sua elaboração. Quando se considera a utilização de variáveis discriminantes, é essencial que se tenha medido nos elementos amostrais variáveis que possam realmente distinguir as populações, caso contrário a qualidade da classificação estará comprometida. Um equívoco bastante comum consiste em se pensar que quanto maior o número de discriminantes, melhor será a solução alcançada. Um dos métodos de detecção de discriminante bastante disseminado é baseado em análise de variância [Anderson 1958]. Nesse trabalho aplicamos análise de variância univariada. A partir do *trace* classificado, cada variável é examinada individualmente e independentemente e sua distribuição F é calculada. As variáveis são então ordenadas pela distribuição F e entre aquelas com maiores valores são então escolhidos os discriminantes. Na análise de variância univariada, uma comparação é feita através da análise de variância de cada variável candidata a discriminante, separadamente. Aquelas variáveis com valores da distribuição F mais significativos estão relacionadas às variáveis mais importantes para a discriminação dos grupos e, portanto, serão consideradas discriminantes. A distribuição F utiliza a razão de duas estimativas, a média da variância dos elementos inter-grupos S_E^2 , pela variância média dos elementos intra-grupos S_D^2 , assim definida:

$$\text{Distribuição F} = \frac{S_E^2}{S_D^2} \quad (1)$$

onde,

$$S_E^2 = n \frac{\sum (\bar{x}_j - \bar{\bar{x}})^2}{(k-1)} \quad (2)$$

e

$$S_D^2 = \frac{1}{k(n-1)} \left[\sum (x_i - \bar{x}_1)^2 + \dots + \sum (x_i - \bar{x}_k)^2 \right] \quad (3)$$

sendo, k o número de amostras e n o número de observações em cada amostra.

Na distribuição F, existe uma distribuição diferente para cada combinação de tamanho da amostra n e número de amostras k . A distribuição é contínua em todo o intervalo de 0 a +8. Além disso, grandes diferenças entre médias amostrais juntamente com pequenas variâncias amostrais podem resultar em valores de F extremamente grandes.

A forma de cada distribuição amostral teórica F depende do número de graus de liberdade associado. Tanto o numerador como o denominador tem graus de liberdade correspondentes. Os graus de liberdade, tanto do numerador como do denominador da distribuição F, se baseiam nos cálculos necessários para deduzir cada estimativa da variância populacional. Para o numerador o número de graus de liberdade é ($k - 1$) e para o denominador $k(n - 1)$.

1.4.2.2. Técnicas de Análise de Agrupamentos

A análise de agrupamento pertence a um conjunto de técnicas para análise estatística multivariada. Análise estatística multivariada é apropriada para qualquer conjunto de dados onde múltiplas medidas são realizadas com possíveis correlações entre essas medidas. Técnicas multivariadas em geral, analisam a estrutura de correlação entre diversas variáveis, podendo revelar resultados mais completos do que se as variáveis fossem analisadas separadamente [Johnson 1998]. Análise de agrupamento, portanto, pode ser utilizada para encontrar grupos nos dados sob análise [Kaufman and Rousseeuw 1990]. A técnica de análise de agrupamento compreende um conjunto de diferentes algoritmos e métodos para agrupar objetos de tipos similares em respectivas categorias. O problema enfrentado por muitos pesquisadores em diferentes áreas consiste exatamente em como organizar os dados sob análise em estruturas que sejam suficientemente representativas. Em outras palavras, análise de agrupamento é uma ferramenta exploratória que busca particionar os componentes em diferentes grupos tal que membros de um mesmo grupo sejam os mais similares possíveis e membros de diferentes grupos sejam os mais diferentes possíveis [Jain 1991].

Estatisticamente, isso implica que a variância intra-grupo deve ser a menor possível e que a variância inter-grupo deve ser a maior possível. Cada agrupamento então descreve, em termos dos dados coletados, a classe a qual seus membros pertencem.

Análise de agrupamento é, portanto uma ferramenta de descoberta. Esta análise pode revelar associações nos dados sob análise, ainda que essas associações não sejam evidentes, porém são úteis uma vez que possam ser descobertas. Os resultados obtidos com a análise de agrupamento podem contribuir para a definição de um esquema de classificação mais formal.

Análise de agrupamento tem sido descrita na literatura através de várias técnicas. Entretanto, todas essas técnicas basicamente pertencem a duas classes: hierárquica e não hierárquica. Na abordagem não hierárquica, inicia-se com um conjunto arbitrário de agrupamentos (semente) e os membros dos agrupamentos são movidos até que a variância intra-grupo seja mínima. A abordagem hierárquica pode ser implementada de duas maneiras: divisiva ou aglomerativa. Utilizando a forma hierárquica aglomerativa, dados n componentes, o método inicia com n agrupamentos (cada agrupamento tendo um componente). Então agrupamentos são unidos sucessivamente até se obter um desejado número de agrupamentos. Na forma hierárquica divisiva, inicia-se com um único agrupamento (de n componentes) e então divide-se os agrupamentos sucessivamente até se obter um número desejado de agrupamentos. Diversos conceitos de distância têm sido utilizados para formar os agrupamentos. Os mais conhecidos são: distância euclidiana, distância ponderada, distância de Minkowski e o coeficiente de concordância de Jaccard. Neste trabalho utilizamos a distância euclidiana dada por:

$$dist(x, y) = \sqrt{\sum (x_i - y_i)^2} \quad (4)$$

onde x_i e y_i são as coordenadas dos pontos x e y .

Análise de agrupamento foi utilizada nesse trabalho para dividir os fluxos de tráfego nos seguintes grupos: ataque e não ataque; P2P e não P2P; usando abordagem hierárquica divisiva e distância Euclidiana.

1.4.3. Detecção de Tráfego de Ataque

Diversos métodos têm sido utilizados na detecção do tráfego de ataque. Os métodos de detecção, por exemplo, baseados em assinatura extraem os dados da rede e identificam os ataques usando seqüências de dados conhecidas e que estão presentes no conteúdo dos pacotes. Tais métodos além de ineficientes, pois não se adaptam a novos tipos de ataques, apresentam sérias restrições relacionadas à privacidade dos dados que estão trafegando pela rede e nem sempre assinaturas estão disponíveis para todos os tipos de ataques.

Uma segunda abordagem empregada utiliza uma massa de dados com ataques previamente identificados para treinar algoritmos de aprendizagem quanto ao comportamento dos ataques. Essa abordagem apresenta a vantagem de que o algoritmo pode ser novamente treinado para aprender sobre novos tipos de ataques. Entretanto, para que isso seja possível, nós devemos inserir instâncias desses novos ataques no arquivo de treinamento, e o método automaticamente reajustaria seu conjunto de regras para que a detecção possa ser realizada.

As duas abordagens apresentadas anteriormente possuem sérias limitações, pois ambas necessitam que os ataques sejam previamente conhecidos e, portanto, novos tipos de ataques não serão detectados. Para superar essas restrições, outras abordagens têm sido aplicadas.

O método de detecção de anomalias detecta comportamentos anormais nos dados, ou seja, detecta desvios do comportamento considerado normal. Esta abordagem

apresenta a grande vantagem de ser possível além de detectar os ataques conhecidos, ser capaz de detectar novos tipos de ataques, pois esses novos ataques provocarão desvios no comportamento normal da rede. Normalmente, métodos de detecção de anomalias necessitam de um conjunto de dados considerado limpo, ou seja, sem a presença de ataques para que se conheça o comportamento normal da rede.

Esse trabalho baseia-se na utilização de métodos de estatística multivariada para identificação do tráfego de ataque. A abordagem apresentada utiliza um reduzido número de discriminantes estatísticos e análise de agrupamento para identificação de tráfego de ataque com resultados superiores aos encontrados até então na literatura relacionada. Análise de agrupamentos por ser uma técnica não supervisionada, permite que novos ataques sejam detectados. O método apresentado foi validado utilizando *traces (offline)* reais. Uma vez identificado o tráfego de ataque nos *traces offline* a abordagem pode ser utilizada para a identificação em tráfego *online*.

1.4.3.1. Trabalhos Relacionados

Identificação de tráfego de ataque tem recebido considerável atenção nos últimos anos constituindo-se em uma importante área de pesquisa. Entretanto, muitos dos trabalhos publicados em identificação de tráfego de ataque têm se restringido a tipos específicos de ataques tais como *DoS attacks* [Hussain et al. 2003], *port scan* [Jung et al. 2004] e *worms* [Kim e Karp 2004], [Schechter et al. 2004].

Em [Jung et al. 2002], os autores apresentam uma metodologia para identificar *flash crowds* e ataques de negação de serviço (*DoS – Denial of Service*). Foram estudadas as propriedades de ambos os tipos de eventos com uma especial atenção para as características que distinguem os dois.

Uma abordagem comumente utilizada para detectar tais ataques tem sido tratar anomalias como desvios de volume de tráfego [Barford et al. 2002], [Brutlag 2000], [Lakhina et al. 2005], [Roughan et al. 2004]. Em [Lakhina et al. 2004a], são tratadas anomalias em redes de *backbone* analisando a quantidade de bytes através de um enlace enquanto que em [Lakhina et al. 2004b] são analisados o volume de tráfego em fluxos de Origem-Destino (OD). A abordagem de detecção de anomalias baseada em volume tem tido sucesso em identificar grandes mudanças no perfil do tráfego tal como ataques conhecidos como *bandwidth flooding attacks*, entretanto, existem várias classes de anomalias que não causam alterações significativas no volume de tráfego. Outras abordagens têm sido utilizadas baseadas na exploração de correlação de padrões entre diferentes variáveis da MIB SNMP [Cabrera et al. 2002], [Rhoden et al. 2002],[Thottan e Ji. 2003], ou baseadas em heurísticas para identificar tipos específicos de anomalias em fluxos de pacotes IP [Kim et al. 2004].

Em [Portnoy et al. 2001] é apresentado um método de detecção automática de intrusões onde é possível detectar ataques ainda desconhecidos, entretanto, é aplicado a um escopo reduzido de tipos de ataques. Um método baseado em anomalias [Taylor e Alves-Foss 2000] tem mostrado alta eficiência na operação por implicar em um baixo custo para a rede. Usando *traces* reais, em [Taylor e Alves-Foss 2001] é apresentado uma análise de eventos anormais de tráfego.

Recentemente técnicas de aprendizado de máquina baseada em redes neurais Bayesianas foram utilizadas para discriminar dados em categorias derivadas de

informações contidas nos pacotes provendo uma classificação dos mesmos sem, contudo acessar o conteúdo dos pacotes [AULD et al. 2007].

Todos esses métodos apresentados acima utilizam métricas baseadas em volume de tráfego. Nós acreditamos, entretanto, que todos eles têm um alcance limitado na identificação de tráfego de ataques uma vez que não possuem um conjunto suficiente de informações para definir comportamentos anormais.

Por outro lado, nós consideramos que métodos que são capazes de examinar individualmente propriedades estatísticas dos fluxos são bastante mais eficientes.

1.4.3.2. Resultados e Discussão

Neste trabalho, para a seleção dos discriminantes, aplicamos análise de variância univariada. Esse trabalho difere da referência [Zuev e Moore 2005] em pelo menos dois aspectos. Primeiro, estamos interessados em discriminar um único tipo de tráfego enquanto [Zuev e Moore 2005] tenta obter uma classificação mais ampla em dez categorias diferentes. Segundo, utilizamos um método mais simples na seleção dos discriminantes. Em [Zuev and Moore 2005], os autores usam o método Naïve Bayes para a seleção dos discriminantes. Nosso método consiste na seleção independente baseado na distribuição F. A partir do *trace* classificado, cada variável é examinada individualmente e independentemente e sua importância é analisada a partir dos valores da distribuição F. Por último, nós usamos um número reduzido de variáveis em comparação a [Zuev e Moore 2005].

Nossa análise utilizou os *traces* previamente descritos. A quantidade de fluxos de ataque e não ataque em cada *trace* é descrita na tabela 1.4. a seguir.

Tabela 1.4. Fluxos por trace.

Fluxos	Trace 1	Trace 2	Trace 3	Trace 4	Trace 5	Trace 6	Trace 7	Trace 8	Trace 9	Trace 10	Total
Ataque	122	19	41	324	122	134	89	129	367	446	1793
Não Ataque	24741	23782	22891	21961	21526	19250	55746	55365	65881	64590	375733
Total	24863	23801	22932	22285	21648	19384	55835	55494	66248	65036	377526
% Ataque	0,49%	0,08%	0,18%	1,45%	0,56%	0,69%	0,16%	0,23%	0,55%	0,69%	5,09%

A seleção dos discriminantes foi baseada em dois critérios: o prévio conhecimento do comportamento comum aos ataques e aquelas variáveis que apresentaram grandes valores para a distribuição F. A tabela 1.5. apresenta as doze variáveis candidatas a discriminante baseado nos grandes valores da distribuição F.

Tabela 1.5. Variáveis candidatas a discriminantes.

Número da Variável	Variável Candidata
1	Porta do Servidor
2	Número Mínimo do Total de Bytes em um Pacote IP <i>cliente para servidor</i>
3	Tamanho Máximo da Janela de Anúncio <i>cliente para servidor</i>
4	Média dos Bytes de Controle no Pacote <i>cliente para servidor</i>
5	Média do Tamanho do Segmento <i>servidor para cliente</i>
6	Tamanho Máximo da Janela de Anúncio <i>servidor para cliente</i>
7	Mediana dos Bytes de Dados IP <i>cliente para servidor</i>
8	Pacotes de Dados Atuais <i>cliente para servidor</i>
9	Tamanho Mínimo da Janela de Anúncio <i>servidor para cliente</i>
10	Variância dos Bytes de Dados <i>servidor para cliente</i>
11	Variância dos Bytes de Controle no Pacote <i>cliente para servidor</i>
12	Tamanho Máximo do Segmento <i>cliente para servidor</i>

Entre as doze variáveis listadas na tabela, algumas delas expressam informações redundantes. Foram, portanto, selecionadas as cinco variáveis que melhor explicavam o comportamento conhecido dos ataques. Estas variáveis são: Tamanho Máximo do Segmento *cliente para servidor* (D1), Tamanho Mínimo da Janela de Anúncio *servidor para cliente* (D2), Número Mínimo do Total de Bytes em um Pacote IP *cliente para servidor* (D3), Média dos Bytes de Controle no Pacote *cliente para servidor* (D4), Variância dos Bytes de Controle no Pacote *cliente para servidor* (D5).

Utilizando os cinco discriminantes descritos acima, agrupamentos de fluxos foram gerados utilizando técnicas hierárquicas e distância Euclidiana. A qualidade da separação dos fluxos em agrupamentos de ataque e não ataque está diretamente relacionada com os resultados do trabalho proposto. Para tanto, foram utilizados os seguintes parâmetros: *precisão média*, *precisão média de ataque* e *confiança*.

Os termos *precisão média*, *precisão média de ataque* e *confiança* são definidos seguir:

$$\text{Precisão Média} = \frac{\text{nº de fluxos corretamente classificados nos clusters}}{\text{total de fluxos do trace}}$$

$$\text{Precisão Média de Ataque} = \frac{\text{nº de fluxos corretamente classificados nos clusters}}{\text{total de fluxos de ataques no trace}}$$

$$\text{Confiança} = \frac{\text{nº de fluxos corretamente classificados nos clusters de ataque}}{\text{total de fluxos nos clusters de ataque}}$$

As tabelas 1.6, 1.7 e 1.8 mostram respectivamente a precisão média de identificação, a precisão média na identificação de ataques e a confiança usando as variáveis discriminantes D1, D2, D3, D4 e D5. O campo D1-5 representa as cinco variáveis analisadas em conjunto. Foram utilizados os 10 *traces* descritos na seção 1.4.1 aplicando-se os cinco discriminantes escolhidos. Nas últimas três colunas foram anotados os valores mínimos, médios e máximos obtidos no conjunto dos *traces*.

Tabela 1.6. Precisão média de identificação por trace e por discriminante.

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Mín	Méd	Máx
D1	96,40	98,20	95,90	94,20	97,50	97,30	91,90	93,40	92,30	87,70	87,70	94,48	98,20
D2	98,30	98,50	97,30	99,90	99,70	98,90	96,80	99,20	95,10	91,70	91,70	97,54	99,90
D3	88,30	98,10	95,90	100,00	86,00	81,70	91,10	96,70	98,10	70,80	70,80	90,67	100,00
D4	97,50	98,50	96,10	99,80	97,30	98,10	94,70	96,60	97,80	94,30	94,30	97,07	99,80
D5	96,30	98,20	96,30	99,80	96,80	98,00	95,10	96,80	97,90	93,70	93,70	96,89	99,80
D1-5	97,60	98,40	97,00	100,00	99,20	98,50	94,40	96,80	98,00	72,00	72,00	95,19	100,00

Tabela 1.7. Precisão média na identificação de ataques por trace e por discriminante.

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Mín	Méd	Máx
D1	74,59	5,26	4,88	82,41	95,08	85,07	8,99	57,36	79,84	73,77	4,88	56,73	95,08
D2	86,07	47,37	36,59	99,69	98,36	94,78	64,04	93,80	89,92	91,48	36,59	80,21	99,69
D3	4,10	0,00	0,00	100,00	0,00	0,00	0,00	98,45	99,46	100,00	0,00	40,20	100,00
D4	90,98	21,05	17,07	99,38	85,25	90,30	50,56	97,67	98,91	88,12	17,07	73,93	99,38
D5	76,23	5,26	17,07	99,38	90,16	91,79	52,81	96,12	98,91	87,00	5,26	71,47	99,38
D1-5	96,72	21,05	39,02	100,00	95,90	91,79	37,08	98,45	99,46	37,22	21,05	71,67	100,00

Tabela 1.8. Confiança por trace e por discriminante.

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Mín	Méd	Máx
D1	94,79	100,00	50,00	99,63	85,93	94,21	100,00	87,06	98,99	98,21	50,00	90,88	100,00
D2	100,00	64,29	93,75	100,00	99,17	96,95	100,00	100,00	96,49	90,07	64,29	94,07	100,00
D3	100,00	0,00	0,00	100,00	0,00	0,00	0,00	80,38	95,55	60,43	0,00	43,64	100,00
D4	88,80	100,00	58,33	100,00	92,04	95,28	83,33	80,25	95,28	98,99	58,33	89,23	100,00
D5	92,08	100,00	70,00	100,00	84,62	93,18	87,04	82,12	95,53	98,73	70,00	90,33	100,00
D1-5	85,51	80,00	76,19	100,00	97,50	96,85	100,00	80,89	95,30	100,00	76,19	91,22	100,00

As figuras 1.3, 1.4 e 1.5 foram construídas a partir dos dados dessas tabelas e ilustram o poder de separação dos cinco discriminantes selecionados e a variabilidade média entre os *traces* obtidos, usando como medida, respectivamente, a precisão média de identificação, a precisão média de identificação de ataques e a confiança média de identificação. Essa variabilidade é mostrada pelos valores máximos e mínimos de cada discriminante apresentados nas tabelas 1.6., 1.7. e 1.8.. A última barra refere-se ao processamento conjunto dos cinco discriminantes.

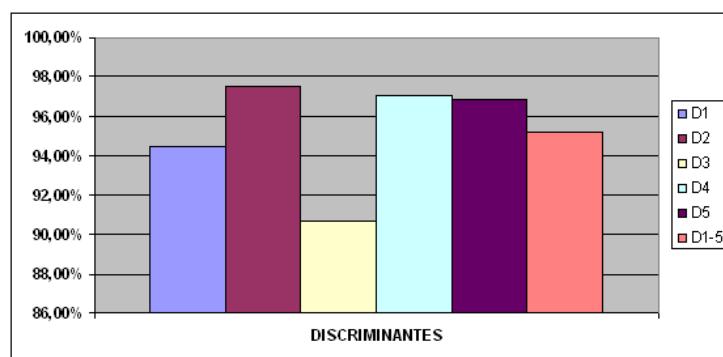


Figura 1.3. Precisão da identificação por discriminantes.

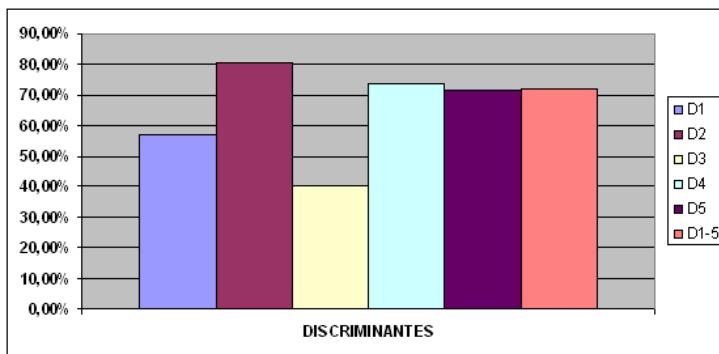


Figura 1.4. Precisão na identificação de ataques por discriminantes isolados e em conjunto.

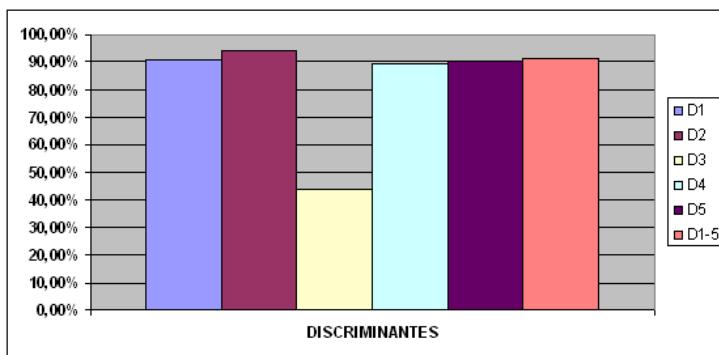


Figura 1.5. Confiança da identificação por discriminante e em conjunto.

Como pode se observar nessas figuras, há uma grande variabilidade entre os *traces*. A análise detalhada deste fenômeno revelou que quando poucos fluxos de ataque estão presentes no *trace*, a precisão da separação diminui. Esta é uma constatação de certa forma esperada pelo reduzido volume de informações presentes nos dados.

A tabela 1.9. a seguir apresenta os resultados para precisão média e confiança, em comparação com os mesmos resultados obtidos em [Zuev e Moore 2005]. Nesta tabela aparecem as seguintes abreviações: NB para *Naïve Bayes*, FCBF para *Fast Correlation-based Filter* e Kernel para *Kernel Density Estimation*. A última linha, *Cluster(5)* mostra os resultados deste trabalho. Devemos ressaltar que entre os cinco discriminantes selecionados pelo método deste trabalho, três deles não coincidem com aqueles utilizados em [Zuev e Moore 2005]. Este fato aliado ao objetivo de selecionar um só tipo de aplicação explica os resultados alcançados notadamente melhores.

Tabela 1.9. Precisão média e confiança dos métodos.

Método	Precisão Média (%)	Confiança (%)
NB	65,26	1,10
NB + Kernel	93,50	8,52
FCBF + NB	94,29	10,38
FCBF + NB + Kernel	96,29	13,46
Cluster (5)	95,19	91,22

Como pode ser observado na tabela 1.9., o melhor resultado obtido em [Zuev e Moore 2005] foi de 13,46% de confiança para identificação de tráfego de ataque e

96,29% de precisão média de identificação. Por outro lado, a seleção de discriminantes e a técnica de agrupamento aplicados neste trabalho resultaram em uma confiança de 91,22% para identificação de tráfego de ataque e uma precisão média de identificação de 95,19%. Ou seja, em conclusão, pode-se ver que apesar da precisão média de identificação ter aproximadamente o mesmo valor, a confiança na identificação de tráfego de ataque aumentou consideravelmente, atingindo um percentual viável de aplicação prática.

1.4.3.3. Conclusões

Nesta seção, apresentamos a aplicação da metodologia descrita anteriormente para identificação de tráfego de ataques. A metodologia aplicada baseia-se na seleção de variáveis discriminantes e posterior agrupamento dos fluxos em ataques e não ataques. Identificação de tráfego de ataque é uma tarefa para a qual a taxa de sucessos atual está entre as mais baixas.

Os resultados encontrados mostram que a metodologia utilizada é superior à principal referência utilizada no desenvolvimento de nosso trabalho. O melhor resultado obtido em [Zuev and Moore 2005] foi 13,46% de confiança para identificação de ataques e 96,29% de precisão média de identificação. Em comparação, os resultados deste trabalho alcançam 91,22% de confiança para identificação de ataques e 95,19% de precisão média de identificação.

Esta é uma pesquisa em andamento. Como continuidade, estamos planejando aplicar a metodologia apresentada neste trabalho a *traces* próprios, coletados em dois ISPs, para a identificação *offline* e *online* do tráfego de ataque.

1.4.4. Detecção de Tráfego de P2P

Nos últimos anos tem-se verificado uma crescente utilização de aplicações P2P refletindo atualmente em uma parcela significativa do tráfego total da Internet. Dada a sua crescente presença, caracterizá-lo e identificá-lo constitui-se em uma importante tarefa. Entretanto, esta identificação tem-se constituído em uma tarefa de difícil execução dadas as características de muitas aplicações P2P de se camuflarem para fugir ao controle exercido sobre tais aplicações.

Tem-se procurado investir em novos métodos que tornem mais precisa e eficiente a identificação dos diferentes tipos de tráfego presentes em uma rede. Diversas abordagens foram sendo desenvolvidas. A partir de 2005, uma nova abordagem, baseada em análise estatística dos fluxos de dados, tem tido resultados bastante satisfatórios. A análise estatística é baseada em informações coletadas durante as conexões entre as estações que compõe a rede e a Internet.

Nos métodos estatísticos, uma etapa crítica é a escolha dos discriminantes a serem utilizados na separação do tráfego (veja seção 1.4.2.1). Conforme descrito na seção 1.4.1, para os *traces* gerados em [Zuev e Moore 2005], foram computados 249 variáveis e o método apresentado neste trabalho parte deste conjunto de 249 variáveis que armazenam estatísticas a respeito do tráfego da rede.

Assim como para o problema de identificação de tráfego de ataque, o problema ao se lidar com tráfego P2P refere-se a seleção de um pequeno número destas variáveis que sejam capazes de discriminar o tráfego em P2P e não P2P. Para tanto, cada um dos

valores das 249 variáveis foi examinada de maneira individual e independente, sendo sua razão F calculada. As variáveis foram então ordenadas pela razão F e aquelas com maiores valores foram escolhidas como discriminantes. Uma vez selecionados os discriminantes, procede-se a uma análise de agrupamentos.

1.4.4.1. Trabalhos Relacionados

Classificação e identificação de aplicações têm recebido considerável atenção nos últimos anos constituindo-se, portanto, em importantes áreas de pesquisa em razão de seu comprovado crescimento [Karagiannis et al 2004a]. Muitos dos trabalhos publicados nessa área, tem sido baseados em portas TCP [Moore et al 2001], [Logg e Cottrell 2003], assinaturas [Sen et al 2004], em heurísticas aplicadas à camada de transporte [Karagiannis et al 2004], [Karagiannis et al 2005] e mais recentemente em abordagens utilizando aprendizagem de máquina [McGregor et al 2004] e em estatísticas do tráfego [Moore et al 2005], [Erman et al 2006], [Auld e Moore 2007].

As técnicas de classificação inicialmente adotadas para a identificação de aplicações através do monitoramento de tráfego, basearam-se no uso exclusivo das portas TCP [Moore et al 2001], [Logg e Cottrell 2003]. Este processo gerava estimativas imprecisas visto que protocolos como HTTP eram usados por outras aplicações, como VLAN sobre HTTP. Também os novos serviços disponíveis na Internet evitavam o uso de portas definidas pelo IANA (*Internet Assigned Numbers Authority*) como as aplicações P2P. Conforme observado em [Hussain et al 2003], os fluxos de ataque também são responsáveis por uma falsa estimativa de tráfego em portas bem conhecidas como a porta 80 (HTTP) e a porta 21 (FTP).

Uma abordagem baseada em assinaturas dos protocolos P2P foi utilizada em [Sen et al 2004] conseguindo identificar as seguintes aplicações: Gnutella, Edonkey, DirectConnect, BitTorrent e Kazaa, porém sendo necessário para a identificação a análise da carga útil dos pacotes.

Em [Zander et al. 2005] o autor busca caracterizar o tráfego utilizando aprendizagem de máquina, porém não utilizando *traces* que foram pré-classificados. Conforme análise realizada em [Auld e Moore 2007] esta abordagem leva a imprecisões na classificação, pois não analisa a presença de fluxos que não utilizam a lista IANA (como os fluxos P2P) e fluxos de ataques.

Conforme apresentado em pesquisas mais recentes por [Auld e Moore 2007] e [Moore e Papagiannaki 2005], a abordagem heurística de [Karagiannis et al. 2004], que apresenta valores de 95% na identificação de tráfego P2P, propõe um esquema de classificação mais genérico que, ao invés de examinar os fluxos, olha para padrões de fluxos dentro dos conjuntos de portas e endereços IPs, sendo inadequado quando uma dessas informações ou ambas estiverem indisponíveis [Auld e Moore 2007]. Em [Moore e Papagiannaki 2005] foi desenvolvido um método mais preciso de identificação das aplicações, baseado em uma seqüência de operações sobre os fluxos de dados que permite não somente a identificação do tráfego P2P como em [Karagiannis et al. 2004], mas de todas as aplicações presentes na rede.

Para a análise das variáveis e escolha daquelas que melhor se adequam para a classificação dos fluxos de dados P2P foram utilizados métodos estatísticos que encontram os valores que melhor identificam uma população de elementos e a

diferenciam de outra população [Mingoti 2005]. Estes métodos estatísticos são conhecidos como distribuição F e análise de agrupamentos [Mingoti 2005] (veja seção 1.4.2). Utilizando então cada uma das 249 variáveis estatísticas, cada uma das quais contendo 377.526 valores referentes a cada um dos fluxos de dados, nossa meta será descobrir quais campos são os melhores para se identificar o tráfego P2P, os quais serão chamados de discriminantes após sua escolha, totalizando 94.003.974 valores de variáveis estatísticas a serem analisados, procurando então separar os fluxos em dois grupos distintos: fluxos P2P e fluxos não P2P.

1.4.4.2. Resultados e Discussão

A seleção das variáveis discriminantes foi baseada em dois critérios: o prévio conhecimento do comportamento comum das aplicações P2P e variáveis que apresentaram grandes valores para a distribuição F [Mingoti 2005].

Foram, portanto, selecionados as cinco variáveis que melhor explicavam o comportamento conhecido das aplicações P2P através do uso da distribuição F. Estas variáveis foram: Porta do Servidor (D1), Tamanho Máximo da Janela de Anúncio *cliente para servidor* (D2), Tamanho Máximo da Janela de Anúncio *servidor para cliente* (D3), Tamanho Mínimo da Janela de Anúncio *servidor para cliente* (D4) e Tamanho Mínimo do Segmento *cliente para servidor* (D5).

Utilizando as cinco variáveis discriminantes descritas acima, agrupamentos de fluxos (com 377.526 variáveis cada) foram gerados utilizando técnicas hierárquicas e distância Euclidiana. A qualidade da separação dos fluxos em agrupamentos P2P e não P2P está diretamente relacionada com os resultados do trabalho proposto. Para tanto, foram utilizados os seguintes parâmetros: *confiança*, *precisão*, *precisão P2P*. Os mesmos parâmetros foram utilizados para medir os resultados em [Moore et al 2005], [Auld e Moore 2007] e [Moore e Zuev 2005]:

$$\text{Confiança} = \frac{\text{nº de fluxos corretamente classificados nos clusters P2P}}{\text{total de fluxos nos clusters P2P}}$$

$$\text{Precisão Média} = \frac{\text{nº de fluxos corretamente classificados nos clusters}}{\text{total de fluxos do trace}}$$

$$\text{Precisão P2P} = \frac{\text{nº de fluxos corretamente classificados nos clusters}}{\text{total de fluxos P2P no trace}}$$

A tabela 1.10 mostra a confiança, a precisão média e a precisão média P2P na classificação das aplicações. A linha D1-5 na tabela refere-se ao uso das cinco variáveis discriminantes juntas. Foram utilizados os 10 *traces* descritos anteriormente aplicando-se os cinco campos discriminantes escolhidos individualmente e em conjunto. Nas colunas podem-se notar os valores mínimos, médios e máximos obtidos no conjunto dos *traces*.

Tabela 1.10. Precisão e Precisão P2P na classificação dos *traces*.

Confiança %				Precisão Média %			Precisão Média P2P %				
	Mín	Méd	Máx		Mín	Méd	Máx		Mín	Méd	Máx
D1	44,79	79,00	87,48	D1	29,67	83,93	100,00	D1	32,00	88,06	96,99
D2	34,79	61,00	83,00	D2	99,63	85,93	94,21	D2	42,50	87,06	98,99
D3	24,79	86,05	90,50	D3	39,63	85,93	100,00	D3	85,07	92,36	98,99
D4	54,79	87,60	94,07	D4	49,54	85,93	94,21	D4	47,35	87,06	95,99
D5	64,79	93,00	100,00	D5	19,63	84,93	93,45	D5	45,00	85,06	93,96
D1-5	47,38	86,12	91,49	D1-5	72,06	96,79	100,00	D1-5	55,63	82,47	100,00

As figuras 1.6 a 1.11 foram construídas a partir dos dados da tabela 1.10. As figuras mostram o poder de separação das cinco variáveis discriminantes selecionadas e a variabilidade média entre os *traces* obtidos. Essa variabilidade é mostrada pelos valores máximos e mínimos de cada discriminante. A última barra refere-se ao processamento conjunto das cinco variáveis discriminantes.

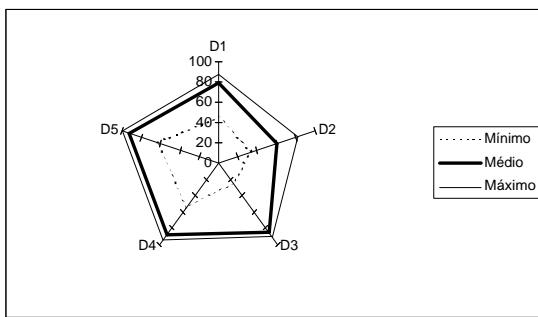


Figura 1.6. Gráfico de Kiviat para a confiança média na classificação.

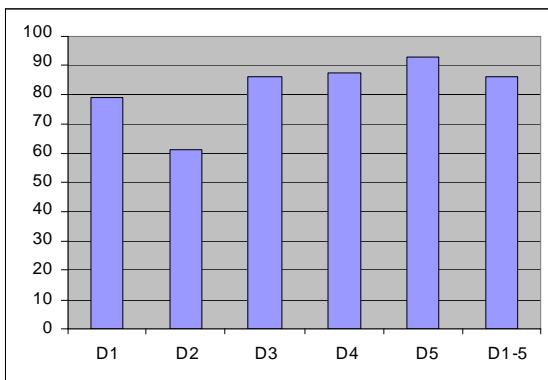


Figura 1.7. Confiança média para a classificação por discriminante e em conjunto.

Como pode ser observado nas figuras 1.7, 1.9 e 1.11 há certa variabilidade entre os *traces*. A análise precisa deste fenômeno revelou que quando poucos fluxos de aplicações P2P estão presentes no *trace*, a precisão da separação diminui. Esta é uma constatação de certa forma esperada pelo reduzido volume de informações presentes nos dados.

Uma forma diferente e visualmente mais intuitiva de analisar estes dados é através do gráfico de Kiviat mostrado nas figuras 1.6, 1.8 e 1.10. Num gráfico de Kiviat eixos radiais eqüidistantes representam as dimensões consideradas para análise. Neste gráfico cada eixo representa um campo discriminante. Em cada eixo são marcados os valores mínimos, médios e máximos e estes pontos são ligados. A figura assim constituída dá uma idéia visual do poder de separação de cada variável discriminante. Nessas figuras, a linha cheia representa os valores médios e as linhas pontilhadas e contínuas representam os valores mínimos e máximos respectivamente.

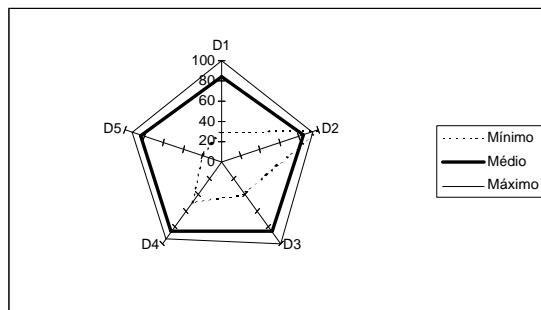


Figura 1.8. Gráfico de Kiviat para a precisão média na classificação.

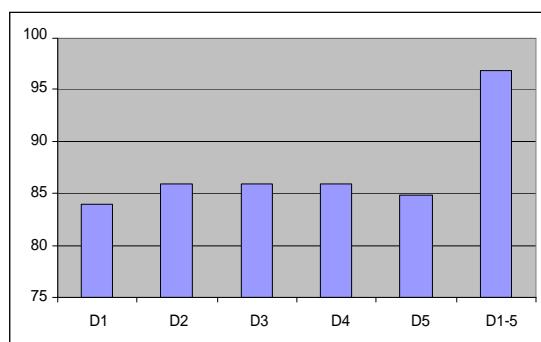


Figura 1.9. Precisão média na classificação por discriminante e em conjunto.

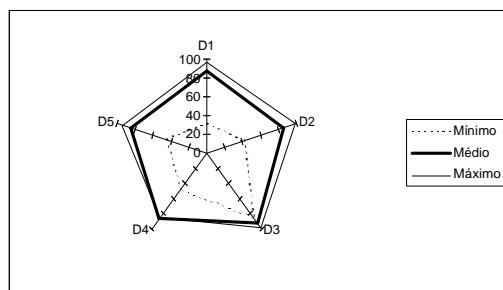


Figura 1.10. Gráfico de Kiviat para a precisão média na classificação P2P.

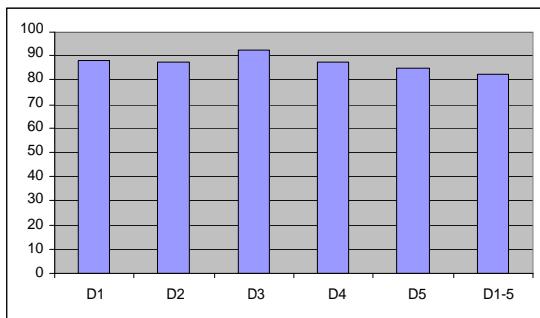


Figura 1.11. Precisão média na classificação de aplicações P2P por discriminante e em conjunto

Foram executados para cada um dos *traces*, a análise de agrupamento com cada discriminante (D1, D2, D3, D4 e D5) individualmente e em conjunto (D1-5). Por questões de espaço, preferimos apresentar os resultados mínimos, médios e máximos para cada discriminante. Algumas das variáveis discriminantes tiveram um resultado acima da média de 86,12%, chegando mesmo a identificar 100% dos fluxos P2P presentes sem a necessidade do conhecimento dos endereços IPs de origem e destino nem do uso das informações contidas nos conteúdos dos pacotes.

Preferimos agrupar os cinco discriminantes e apresentar o resultado médio para não dependermos de um único discriminante, melhorando a confiança do método ao aplicá-lo a outros conjuntos de fluxos.

1.4.4.3. Conclusões

Esta seção apresentou uma metodologia para identificação de aplicações P2P. A metodologia baseia-se na seleção de discriminantes e posterior identificação dos fluxos P2P. Os resultados encontrados mostram a viabilidade da proposta para ser utilizada na prática. Os resultados foram de 96,79% de precisão média, 86,12% de confiança e 82,47% de precisão para aplicações P2P.

Esta é uma pesquisa em andamento. Como continuidade, estamos planejando aplicar a metodologia apresentada neste trabalho a *traces* próprios, coletados em dois ISPs.

1.5. Considerações Finais

Este capítulo abordou algumas tarefas avançadas de administração de redes com ênfase em administração de tráfego. No atual estágio de evolução da Internet, a imprevisibilidade das novas aplicações e das novas estratégias de ameaças à integridade dos sistemas em redes exige que o administrador disponha de ferramentas que possam monitorar, mesmo que *offline*, o comportamento do tráfego na sua rede.

Uma das dificuldades inerentes a esse processo é a construção de uma infra-estrutura de medição e armazenamento que possa cobrir a rede de forma ampla. Foram apresentados um modelo de infra-estrutura de medição e um conjunto de técnicas de monitoramento de duas classes de tráfego de interesse para o administrador e de difícil detecção: tráfego de ataque e tráfego de aplicações P2P. O modelo e os algoritmos aqui descritos podem ser estendidos para detecção de outras classes de tráfego.

Referências

- Anderson, T. W. (1958). An Introduction to Multivariate Statistical Analysis. *Ed. John Wiley Sons, NY.*
- Auld, T. e Moore A. (2007). A Bayesian Neural Networks for Internet Traffic Classification, IEEE Transactions on Neural Networks, Vol 18, No 1.
- Barford, P., Kline, J., Plonka, D. e Ron, A. (2002). A signal analysis of network traffic anomalies. *In Internet Measurement Workshop.*
- Brutlag, J. (2000). Aberrant behavior detection in timeseries for network monitoring. *In USENIX LISA.*
- Cabrera, J. B. D., Lewis, L., Xinzhou, Q., Lee, W., Prasanth, R., Ravichandran, B. e Mehra, R. (2002). Proactive Intrusion Detection and Distributed Denial of Service Attacks—A Case Study in Security Management. *Journal of Network and Systems Management.*
- Erman, J., Arlitt, M., e Mahanti, A. (2006). Traffic classification using Clustering Algorithms. In SIGCOMM 2006 MineNet Workshop, Pisa, Italy.
- Hussain, A., Heidemann, J. e Papadopoulos, C. (2003). A Framework for Classifying Denial of Service Attacks. *In ACM SIGCOMM, Karlsruhe.*
- IANA, Internet Assigned Numbers Authority. <http://www.iana.org/assignments/port-numbers>.
- Jain, R. (1991). The Art of Computer Systems Performance Analysis. *In John Wiley Sons, Inc.*
- Johnson, D. (1998). Applied Multivariate Methods for Data Analysis. In Brooks/Cole Publishing Co.
- Jung, J., Krishnamurthy, B. e Rabinovich, M. (2002). Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. *In Proceedings of ACM WWW.*
- Jung, J., Paxson, V., Berger, A. e Balakrishnan, H. (2004). Fast Portscan Detection Using Sequential Hypothesis Testing. *In IEEE Symposium on Security and Privacy.*
- Karagiannis, T., Broido, A., Faloutsos, M., e Claffy, K. (2004). Transport Layer Identification of P2P Traffic. In Proceedings of IMC'04.
- Karagiannis, T., Broido, A., Brownlee, N., Claffy, K. e Faloutsos, M. (2004a). Is P2P dying or just hiding ? In IEEE Globecom 04.
- Karagiannis, T., Papagiannaki, K. e Floutsos, M. (2005) BLINC: Multilevel Traffic Classification in the Dark. In Proceedings of the SIGCOMM'05.
- Kaufman, L. e Rousseeuw, P. (1990). Finding Groups in Data: An Introduction to Cluster Analysis. *In Wiley and Sons, Inc.*
- Kim, H. A. e Karp B. (2004). Autograph: Toward Automated, Distributed Worm Signature Detection. *In Usenix Security Symposium, San Diego.*

- Kim, M. S., Kang, H. J., Hung, S. C., Chung, S. H. e Hong, J. W. (2004). A Flow-based Method for Abnormal Network Traffic Detection. In *IEEE/IFIP Network Operations and Management Symposium, Seoul*.
- Lakhina, A., Crovella, M. e Diot, C. (2004a). Characterization of Network-Wide Anomalies in Traffic Flows. *Technical Report BUCS-2004-020, Boston University*.
- Lakhina, A., Crovella, M. e Diot, C. (2004b). Diagnosing Network-Wide Traffic Anomalies. In *ACM SIGCOMM, Portland*.
- Lakhina, A., Crovella, M. e Diot, C. (2005). Mining anomalies using traffic feature distributions. In *Proceedings of ACM SIGCOMM*.
- Logg C. e Cottrell. (2003). Characterization of the traffic between SLAC and the Internet.
- McGregor, A., Hall, M., Lorier, P. e Brunskill, J. (2004). Flow clustering using machine learning techniques. In *PAM 2004, Antibes Juan-lesPins, France*.
- Mingoti, S. (2005). Análise de dados através de métodos de estatística multivariados, uma abordagem aplicada. Editora UFMG, páginas 213-256.
- Moore, D., Keys, K., Koga, R., Lagache, E. e Claffy, K. (2001). CoralReef software suite as a tool for system and network administrators In Proceedings Large Installation System Administration Conference LISA, páginas 133-144.
- Moore, A., Hall, J., Kreibich, C., Harris, E., e Pratt, I. (2003). Architecture of a Network Monitor. In *Passive & Active Measurement Workshop (PAM)*.
- Moore, A., Zuev, D., e Crogan, M. (2005). Discriminators for use in flow-based classification. RR-05.13 Department of Computer Science, University of London, 2005.
- Moore, A. e Papagiannaki, K. (2005). Toward the Accurate Identification of Network Applications, In Proceedings of the Sixth Passive and Active Measurement Workshop (PAM 2005), volume 3431, Springer-Verlag LNCS.
- Moore, A. e Zuev, D. (2005). Internet Traffic Classification Using Bayesian Analysis Techniques. In Proceedings of the 2005 ACM Sigmetrics International Conference on Measurements and Modeling of Computer Systems.
- Paxson, V. (1999). “BRO: A system for detecting network intruders in real-time” Computer Networks (Amsterdam), vol 31, no 23-24, páginas 2435-2463.
- Portnoy, L., Eskin, E. e Stolfo, S. (2001). Intrusion detection with unlabeled data using clustering. In *ACM Workshop on Data Mining Applied to Security (DMSA)*.
- Rhoden, G. E., Melo, E. T. L. e Westphall, C. B. (2002). Detecção de Intrusões em Backbones de Redes de Computadores através da Análise de Comportamento com SNMP, *II Workshop de Segurança, Búzios, Brasil*.

- Roesch, M. (1999). “SNORT – lightweight intrusion detection for networks”, in USENIX 13th, Large Installation System Administration Conference (LISA), Seattle, WA, páginas 229-238.
- Roughan, M., Griffin, T., Mao, Z. M., Greenberg, A. e Freeman, B. (2004). Combining Routing and Traffic Data for Detection of IP Forwarding Anomalies. *In ACM SIGCOMM NetS Workshop, Portland*.
- Sampaio, L. , Koga, Ivo. K., Souza, H., KOGA, Ivan K., Rhoden, G. E., Vetter, F., Leiria, G., Monteiro, J. A. S., Melo, E. (2006). piPEs-BR: Uma arquitetura para a medição de desempenho em redes IP. In: Anais do 24º Simpósio Brasileiro de Redes de Computadores. Curitiba, PR. v. 1, p. 523-538.
- Schechter, S., Jung, J. e Berger, A. (2004). Fast Detection of Scanning Worm Infections. *In Seventh International Symposium on Recent Advances in Intrusion Detection (RAID), Sophia Antipolois, France*.
- Sen, S., Spatscheck, O., e Wang, D. (2004). Accurate, Scalable In-Network Identification on P2P Traffic using Application Signatures. WWW'04: Proceedings of the 13th International Conference on World Wide Web.
- Taylor, C. e Alves-Foss, J. (2000). Low Cost Network Intrusion Detection.
- Taylor, C. e Alves-Foss, J. (2001). NATE: Network Analysis of Anomalous Traffic Events. *In Proceedings New Security Paradigms Workshop*.
- Thottan, M. e Ji., C. (2003). Anomaly Detection in IP Networks. *In IEEE Trans. Signal Processing (Special issue of Signal Processing in Networking)*, pages 2191.2204.
- Zander, S., Nguyen, T., e Armitage, G. (2005). Automated Traffic Classification and Application Identification using Machine Learning. *In 30th Annual IEEE Conference on Local Computer Networks (LCN 30)*, pages 220–227, Sydney, Australia.
- Zuev, D. e Moore, A. (2005). Traffic Classification using a Statistical Approach. In Passive and Active Measurement Workshop, Boston, USA.

Capítulo

2

Teste de Software

Pedro Santos Neto, Francisco Vieira de Sousa, Rodolfo Resende

Abstract

Testing is one of the activities related to quality assurance in software development. The cost to perform test activities is usually high. These activities can consume from 30% to 40% of the overall effort in a project. The main goal of this lecture is to present the test concepts, test levels, test techniques, and related measures. During this lecture we exhibit a test discipline inside a software development process. We expect to help the execution of testing activities in the software development organizations, decreasing the related costs and improving the quality of the developed products.

Resumo

A atividade de teste é uma das atividades relacionadas à garantia da qualidade no desenvolvimento de software. O custo para realização das atividades de teste é geralmente alto, não sendo raro que organizações gastem de 30% a 40% do esforço total de um projeto nessa atividade. O principal objetivo deste curso é apresentar os conceitos relacionados à área de teste, desde seus fundamentos, passando pelos níveis de teste existentes, as possíveis técnicas empregadas e medidas relacionadas à sua execução. Durante essa explanação exibiremos o funcionamento do fluxo de teste dentro de um processo de desenvolvimento. Esperamos com isso, auxiliar a adoção das atividades de teste pelas empresas desenvolvedoras de software, diminuindo os custos relacionados à sua execução e aumentando a qualidade dos produtos desenvolvidos.

2.1. Introdução

A cada dia nos tornamos mais dependentes de sistemas de software. Muitos dos produtos que utilizamos incorporam, de alguma forma, processadores e softwares de controle. Precisamos, cada vez mais, de mecanismos que garantam a qualidade desses sistemas, evitando problemas durante sua utilização.

O teste constitui um elemento crítico na garantia de qualidade, representando a revisão final da especificação, projeto e geração de código [23]. A realização dessa

atividade é geralmente bastante onerosa em um desenvolvimento. Dependendo do tipo de sistema a ser desenvolvido, ela pode ser responsável por mais de 50% dos custos [23]. Isso ocorre porque uma parte significativa dessas atividades ainda é executada de forma manual, com a geração de casos de teste baseada nos documentos de requisitos ou em outros documentos produzidos durante o processo de desenvolvimento. Para se ter uma idéia dos custos envolvidos, de acordo com um relatório publicado pelo NIST [22], U\$59.500.000.000,00 é o valor relativo ao custo de falhas em softwares desenvolvidos nos Estados Unidos, apenas em 2002. Esse mesmo relatório estima que mais de 37% desse custo (U\$22.200.000.000,00) poderia ter sido eliminado se a infraestrutura para teste fosse melhorada.

A qualidade é um importante aspecto do software, uma vez que os custos relacionados a falhas podem ser bastante altos. A Garantia de Qualidade é um importante investimento. De acordo com Pressman [23], qualidade é "a conformidade a requisitos funcionais e de desempenho, padrões de desenvolvimento explicitamente documentados e outras características implícitas que são esperados por todos os softwares desenvolvidos profissionalmente".

Os requisitos do software podem ser utilizados para aferir a qualidade de um produto. A não conformidade do produto aos requisitos é considerada falta de qualidade. Dependendo do tipo de software, a prioridade dos requisitos pode variar. A Figura 2.1 mostra uma possível configuração de prioridades de requisitos para dois tipos de aplicações. Para um sistema interativo, como um Sistema de Controle de Mercearias, a portabilidade, manutenibilidade e usabilidade podem ser requisitos prioritários, enquanto que em um software embarcado de satélite o desempenho pode ser uma questão fundamental. A medida de qualidade desses dois tipos de software leva em consideração aspectos diferentes.

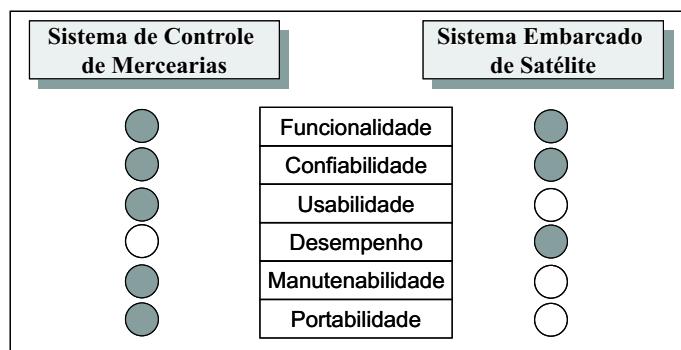


Figura 2.1. Características de qualidade para diferentes tipos de software.

As técnicas para Validação e Verificação (V&V) são algumas das atividades relacionadas à garantia da qualidade. Embora essas atividades sejam consideradas importantes por quase todas as organizações, boa parte delas ainda têm dúvidas sobre executar ou não tais atividades, dentre outros fatores, em função de uma percepção de altos custos [23].

A Figura 2.2 [23] mostra o custo estimado para corrigir erros em diferentes estágios do desenvolvimento. Quanto mais cedo um defeito for descoberto, menor serão os custos para corrigir os erros associados. A utilização de atividades de garantia da

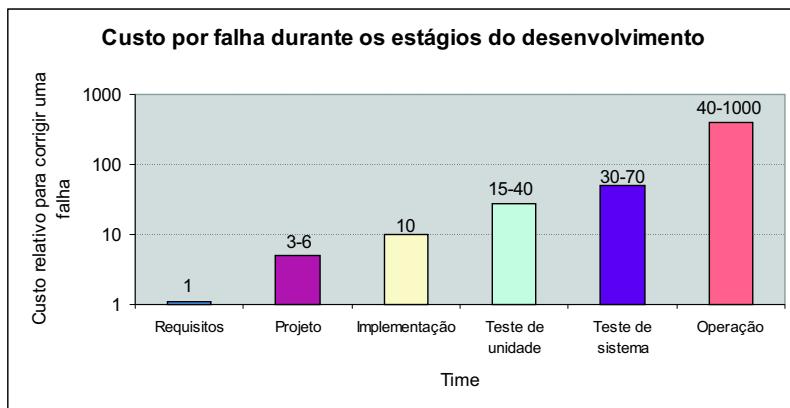


Figura 2.2. Custo para correção de erros durante o desenvolvimento de software.

qualidade no desenvolvimento de software aumenta os custos devido a introdução de atividades adicionais no desenvolvimento, mas propicia a descoberta de erros mais cedo, o que normalmente gera uma diminuição geral de custos.

Existem diversas técnicas de V&V. Dentre elas, existem as técnicas que não exigem a execução do software. Elas tentam determinar propriedades que devem ser válidas independentemente da execução. São exemplos dessas técnicas a verificação de modelos, análise estática, revisões e inspeções. Por outro lado, existem técnicas que tentam encontrar falhas a partir da análise do produto durante sua execução, como por exemplo, a simulação, execução simbólica e o teste [16].

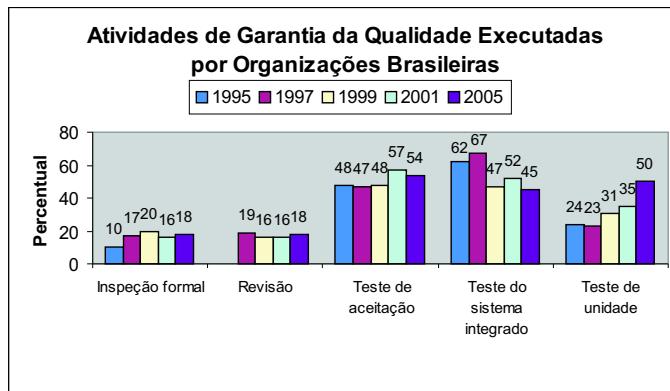


Figura 2.3. Atividades de V&V executada pelas organizações brasileiras desenvolvedoras de software.

Embora a exigência por produtos com maior qualidade seja cada vez maior, muitas organizações desenvolvedoras de software ainda não executam essas atividades. A Figura 2.3 mostra as atividades de V&V executadas pelas organizações brasileiras. Esses dados foram extraídos do Programa Brasileiro da Qualidade e Produtividade em Software [19]. Conforme exibido na figura, ainda existem diversas organizações brasileiras que não utilizam técnicas de V&V. Os principais problemas relacionadas a não execução das atividade de teste são a falta de cultura nessa disciplina e os custos relacionados à sua execução.

Identificação	Procedimento de Teste Login
Objetivo	Efetuar o login de um usuário no Merci.
Requisitos especiais	A TelaPrincipal deve estar no estado SEM USUARIO.
Fluxo	1. Preencher o campo login e senha 2. Pressionar o botão login.

Figura 2.4. Procedimento de Teste Login.

Embora as inspeções (revisões técnicas) sejam mais eficazes na detecção e remoção de defeitos [1, 17], os testes são importantes para complementar as revisões e aferir o nível de qualidade conseguido. A realização de testes é limitada por restrições de cronograma e orçamento, uma vez que esses fatores determinam quantos testes poderão ser executados. É importante que os testes sejam bem planejados e desenhados, para conseguir-se o melhor proveito possível dos recursos alocados para eles [3].

O *teste de software* é a verificação *dinâmica* do funcionamento de um programa utilizando um *conjunto finito* de casos de teste, adequadamente *escolhido* dentro de um domínio de execução infinito, contra seu *comportamento esperado* [16]. Nesse conceito existem alguns elementos chaves para este trabalho:

- dinâmica: o teste exige a execução do produto, embora algumas de suas atividades possam ser realizadas antes do produto estar operacional;
- conjunto finito: o teste exaustivo é geralmente impossível, mesmo para produtos simples;
- escolhido: é necessário selecionar testes com alta probabilidade de encontrar defeitos, preferencialmente com o mínimo de esforço;
- comportamento esperado: para que um teste possa ser executado é necessário saber o comportamento esperado, para que ele possa ser comparado com o obtido.

Neste trabalho apresentamos uma visão geral do teste de software, apresentando alguns conceitos básicos, os níveis de teste, objetivos e critérios de teste definidos pelo IEEE [16], as atividades de teste, juntamente com os principais documentos associados à sua execução, e uma classificação de ferramentas de apoio aos testes.

2.2. Conceitos básicos

Nesta seção apresentamos alguns dos conceitos básicos relacionados a área de teste. A descrição desses conceitos foi baseado no Glossário de Terminologia de Engenharia de Software da IEEE [14] e no Guia do Corpo de Conhecimento em Engenharia de Software [16].

Um *caso de teste* especifica como deve ser testado uma parte do software. Essa especificação inclui as entradas, saídas esperadas, e condições sob as quais os testes devem ocorrer. Um exemplo para um caso de teste para um login inválido é apresentado

Identificação	Login inválido com caracteres não permitidos	
Itens a testar	Verificar se a tentativa de login utilizando um identificador inválido, contendo caracteres não permitidos, exibe a mensagem apropriada.	
Entradas	Campo	Valor
	Login	pasn!
Saídas esperadas	Campo	Valor
	Login	pasn!
	Senha	aaaa (exibida com *'s)
	Mensagem	O campo login deve ter no mínimo 2 e no máximo 8 caracteres alfabéticos.
Estado alcançado	SEM USUARIO	
Ambiente	Banco de dados de teste.	
Procedimentos	Procedimento de Teste Login	
Dependências	Não aplicável.	

Figura 2.5. Caso de teste para login inválido.

na Figura 2.5. Um *procedimento de teste* detalha as ações a serem executadas em um determinado caso de teste. A Figura 2.4 exibe o Procedimento de Teste de Login. Observe que um caso de teste só faz sentido se for especificado o(s) procedimento(s) de teste associado(s). Sem essa associação não é possível determinar o que deve ser feito com as entradas especificadas no caso de teste. Observe também que o procedimento de teste é independente dos dados utilizados nos casos de teste.

Um *oráculo* é qualquer agente (humano ou não) que decide se um software funcionou corretamente em um determinado teste. A geração de um oráculo é provavelmente a parte mais difícil e cara da automação de testes.

Um *critério de teste* define quais propriedades de um programa devem ser exercitadas para constituir um teste completo [12]. Muitas vezes esse termo é utilizado como sinônimo para *técnica de teste*, uma vez que critérios podem ser utilizados para selecionar os casos de teste. Critérios também podem ser utilizados para decidir se o teste pode ou não ser considerado completo e, por consequência, finalizado.

Existem basicamente duas maneiras de se construírem testes [8]: (i) método da caixa branca: tem por objetivo determinar defeitos na estrutura interna do produto, a partir do desenho de testes que exercitem suficientemente os possíveis caminhos de execução; e (ii) método da caixa preta: tem por objetivo determinar se os requisitos foram total ou parcialmente satisfeitos pelo produto. Os testes de caixa preta não verificam como ocorre o processamento, mas apenas os resultados produzidos.

2.3. Níveis e objetivos de teste

Existem dois conceitos de teste ortogonais que muitas vezes são considerados correlatos: o alvo do teste (target) e seu objetivo (objective) [16]. O alvo do teste apresenta a subdivisão geralmente utilizada em produtos de software complexos. O alvo de um teste pode ter diferentes níveis: pode ser um módulo único, correspondendo ao teste de unidade, um agrupamento de módulos, correspondendo ao teste de integração, ou o sistema completo,

correspondendo ao teste de sistema.

2.3.1. Teste de Unidade

O Teste de Unidade é um teste de Caixa Branca. Testes de Unidade têm por objetivo verificar um elemento que possa ser logicamente tratado como uma unidade de implementação. Em produtos implementados com tecnologia orientada a objetos, uma unidade é tipicamente uma classe. Em alguns casos, pode ser conveniente tratar como unidade um grupo de classes correlatas. Normalmente, os testes de unidade são feitos pelos próprios desenvolvedores, durante a implementação de uma unidade [8].

De uma forma simples, o teste de unidade nada mais é do que escrever um pedaço de código para exercitar um outro pedaço de código, com o objetivo de determinar se esse outro pedaço de código está se comportando conforme o esperado. Para isso devemos realizar uma série de assertivas, que nada mais são que a verificação de condições associadas ao funcionamento do pedaço de código a ser testado.

O teste de unidade pode ocorrer de forma paralela para os vários módulos que compõe o sistema. Assim, se existirem dez programadores codificando o sistema, esses dez programadores podem também codificar testes de unidade para verificar o que foi desenvolvido. As principais características avaliadas no teste de unidade são:

- Interface: verifica-se se os dados entram e saem corretamente, se número e tipo de parâmetros são apropriados.
- Compatibilidade: verifica-se operações sobre variáveis, cálculos incorretos contendo *over/underflow*, uso de índices impróprios, maiores que o possível ou menores que o permitido.
- Caminhos de execução: verifica-se se os caminhos importantes são corretamente executados.
- Atendimento a erros: verifica-se se a rotina de erro corresponde ao erro encontrado, se as mensagens elucidam a causa do problema.
- Condições de contorno: testes são realizados com valores máximo, mínimo, imediatamente abaixo e acima de itens de dados e de variáveis de controle de iterações, verifica-se erros de precedência de operadores, comparações de tipos de dados diferentes, terminação inexistente de ciclos e erros de precisão em cálculos.

De uma maneira geral, todos os desenvolvedores realizam testes durante o desenvolvimento. Geralmente se codifica algo e se verifica, a partir da execução do que foi implementado, se seu comportamento ocorre conforme o esperado. O grande problema é que esses testes são feitos de forma manual. Assim, mesmo que uma certa condição tenha sido verificada durante sua implementação, nada garante que no futuro ela estará funcionando, uma vez que parte do código pode ser alterado sem a posterior verificação do funcionamento.

Para resolver esses problemas, os testes de unidade devem ser criados e sua execução deve ser automatizada, devendo ser executado periodicamente. Assim, caso haja

uma mudança em alguma parte do código com testes de unidade desenvolvidos, gerando um comportamento incorreto do produto, essa mudança será revelada quando esses testes forem executadas.

2.3.1.1. Ferramentas para o Teste de Unidade

A família xUnit [20] é o arcabouço mais utilizado atualmente para criação de testes de unidade. Para a linguagem Java, existe o JUnit [13], que foi desenvolvido por Kent Beck e Erich Gamma. A maioria das ferramentas de desenvolvimento Java existentes no mercado, e.g. JBuilder, Eclipse, NetBeans, incorporam o JUnit dentro de seu ambiente, facilitando assim o seu uso.

Utilizaremos aqui um exemplo de uma implementação de uma classe representando um triângulo. O código dessa classe é exibido na Tabela 2.1, assim como sua classe de teste, que será detalhada a seguir. A versão atual do JUnit sofreu algumas mudanças significativas, uma vez que o arcabouço foi reformulado para utilizar anotações. Neste trabalho descrevemos o JUnit 3, uma vez que seu entendimento não requer o conhecimento de anotações em Java.

Para criar uma classe de teste, utilizando o JUnit, basta criar uma classe, tornando-a uma herdeira da classe `JUnit.framework.TestCase`. No código exibido na Tabela 2.1, temos a classe *Triangulo*, juntamente com sua classe de teste, *TestTriangulo*, herdando de *TestCase*.

As classes de teste devem conter um ou mais métodos públicos responsáveis pela execução dos testes. Qualquer método iniciado com a palavra *test* é considerada pelo JUnit um teste. Normalmente utilizamos o padrão *testXXX()* para denotar um teste para *XXX*. Assim, para testarmos se a classe triângulo corretamente classifica triângulos equiláteros, podemos criar o método *testEquilatero()*. Da mesma forma, existem outros testes na classe *TestTriangulo*: *testIsosceles*, *testEscaleno*, *testInvalido*.

Conforme pode ser visto na classe *TestTriangulo*, e já mencionado anteriormente, testar um pedaço de código é escrever outro pedaço de código que o utilize, fazendo invocações ao pedaço a ser testado, sempre verificando os resultados obtidos. Vamos analisar com maiores detalhes um dos métodos de teste: o método *testInvalido*. Nesse método temos na sua primeira linha a criação de um triângulo com valores 1,2 e 3, juntamente com a chamada a um método *assertEquals*. Esse método verifica se um determinado valor é igual a outro. Nesse caso, está sendo verificado se "INVALIDO" é o resultado obtido ao se invocar o método *classifica()*, do objeto triângulo criado no teste. Se os valores do *assert* forem iguais, é dito que o teste "passou", em caso contrário dizemos que o teste "falhou".

Uma vez criado a classe de teste, ela se torna executável, visto que a classe *TestCase*, do JUnit, contém métodos para a execução dos testes. Como a classe criada herda de *TestCase*, ela se torna executável automaticamente. Basta saber como executar uma classe de teste no seu ambiente de desenvolvimento Java. Na maioria dos ambientes, ao se clicar com o botão direito do mouse em cima da classe de teste, surge um menu contendo a opção de execução da classe. A Figura 2.6 mostra a tela do JUnit com o resultado da

```

package triangulo;

/**
 * Classe representando um triangulo.
 */
public class Triangulo {
    int ladoZero, ladoUm, ladoDois;

    final String INV = "INVALIDO";
    final String EQU = "EQUILATERO";
    final String ISO = "ISOSCELES";
    final String ESC = "ESCALENO";

    /**
     * Construtor recebendo os lados do triangulo.
     * @param l0 int
     * @param l1 int
     * @param l2 int
     */
    public Triangulo(int l0, int l1, int l2) {
        this.ladoZero = l0;
        this.ladoUm = l1;
        this.ladoDois = l2;
    }

    /**
     * Classifica um triangulo de acordo com seus lados,
     * retornando uma string com a classificacao.
     * @return String
     */
    public String classifica() {
        if (ladoZero == 0 || ladoUm == 0 || ladoDois == 0)
        {
            return INV;
        }
        if (ladoZero + ladoUm <= ladoDois || 
            ladoDois + ladoUm <= ladoZero ||
            ladoDois + ladoZero <= ladoUm)
        {
            return INV;
        }
        if (ladoZero != ladoUm && ladoUm != ladoDois &&
            ladoZero != ladoDois)
        {
            return ESC;
        }
        if (ladoZero == ladoUm && ladoUm == ladoDois)
        {
            return EQU;
        }
        return ISO;
    }
}

package triangulo;

import junit.framework.TestCase;

/**
 * Testa a classe triangulo.
 */
public class TestTriangulo extends TestCase {

    /**
     * Construtor padrao.
     */
    public TestTriangulo() {
    }

    /**
     * Testa triangulo invalido, com soma de dois
     * lados igual ao terceiro.
     */
    public void testInvalido()
    {
        Triangulo t = new Triangulo(1, 2, 3);
        assertEquals("INVALIDO", t.classifica());
    }

    /**
     * Testa se a classificacao de triangulo equilatero é
     * feita corretamente.
     */
    public void testEquilatero()
    {
        Triangulo t = new Triangulo(2, 2, 2);
        assertEquals("EQUILATERO", t.classifica());
    }

    /**
     * Testa se a classificacao de triangulo escaleno é
     * feita corretamente.
     */
    public void testEscaleno()
    {
        Triangulo t = new Triangulo(5, 6, 7);
        assertEquals("ESCALENO", t.classifica());
    }

    /**
     * Testa se a classificacao de triangulo isosceles é
     * feita corretamente.
     */
    public void testIsosceles()
    {
        Triangulo t = new Triangulo(3, 3, 5);
        assertEquals("ISOSCELES", t.classifica());
    }
}

```

Tabela 2.1. Exemplo de uma classe e sua classe de teste.

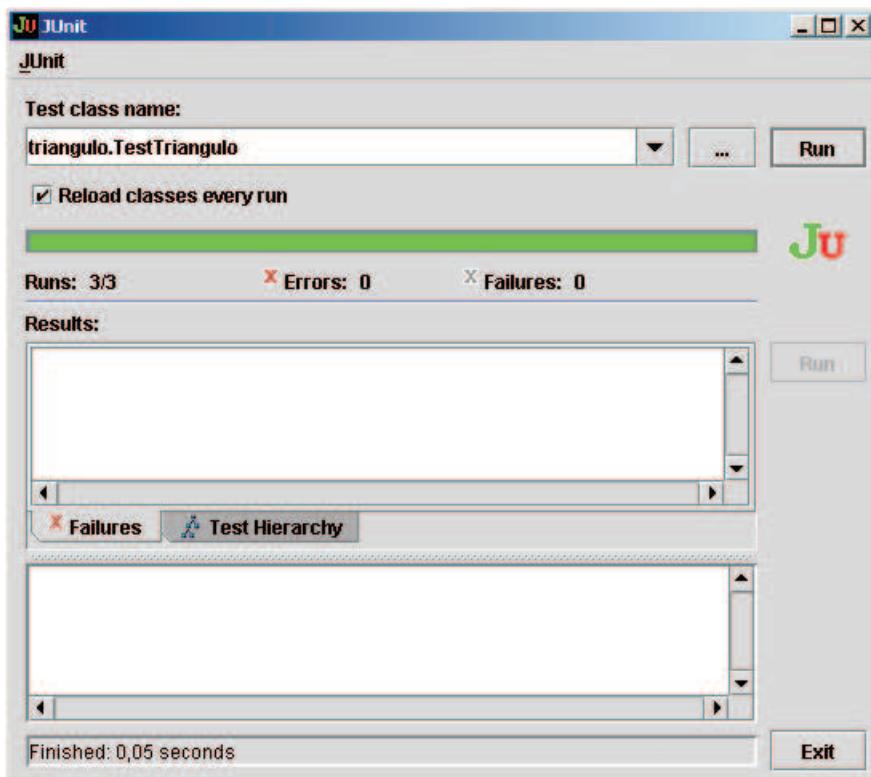


Figura 2.6. Exemplo de execução da classe de teste com sucesso.

execução de uma classe de teste sem falhas. A barra verde na tela indica que não houve falhas.

A Figura 2.7 apresenta a mesma tela, porém com o registro de falhas na execução dos testes. Para cada falha registrada existe um resumo da falha encontrada, e um detalhamento sobre sua origem e onde ela se manifestou. Os detalhes da falha exibem o resultado esperado, o resultado obtido, e a informação sobre onde essa falha se manifestou no teste. A partir desses dados é possível realizar a depuração do código para se descobrir exatamente que trecho do código está associado à falha encontrada.

Uma importante questão sobre o teste de unidade é saber quando parar de criar testes. Essa pergunta é bem interessante e a resposta está diretamente ligada ao critério de teste utilizado na sua organização. Assim, se o critério utilizado for cobertura de código (100% de cobertura de linhas), poderemos finalizar a criação dos testes quando ele exercitar todas as linhas existentes na unidade sendo avaliada. Para isso precisaremos de uma outra ferramenta, muito útil para avaliação de testes de unidade, denominada de Ferramenta de Análise de Cobertura. Essas ferramentas criam relatórios de execução, normalmente indicando a cobertura alcançada, tanto em termos de porcentagem, quanto em termos de linhas existentes. Um exemplo desse relatório é exibido na Figura 2.8. Nessa figura podemos notar as classes executadas, juntamente com a porcentagem de cobertura alcançada. Clicando no nome da classe, podemos ver o código com diferenciação de cores para linhas executadas e não executadas.

De forma similar ao JUnit, boa parte dos ambientes de desenvolvimento Java pos-

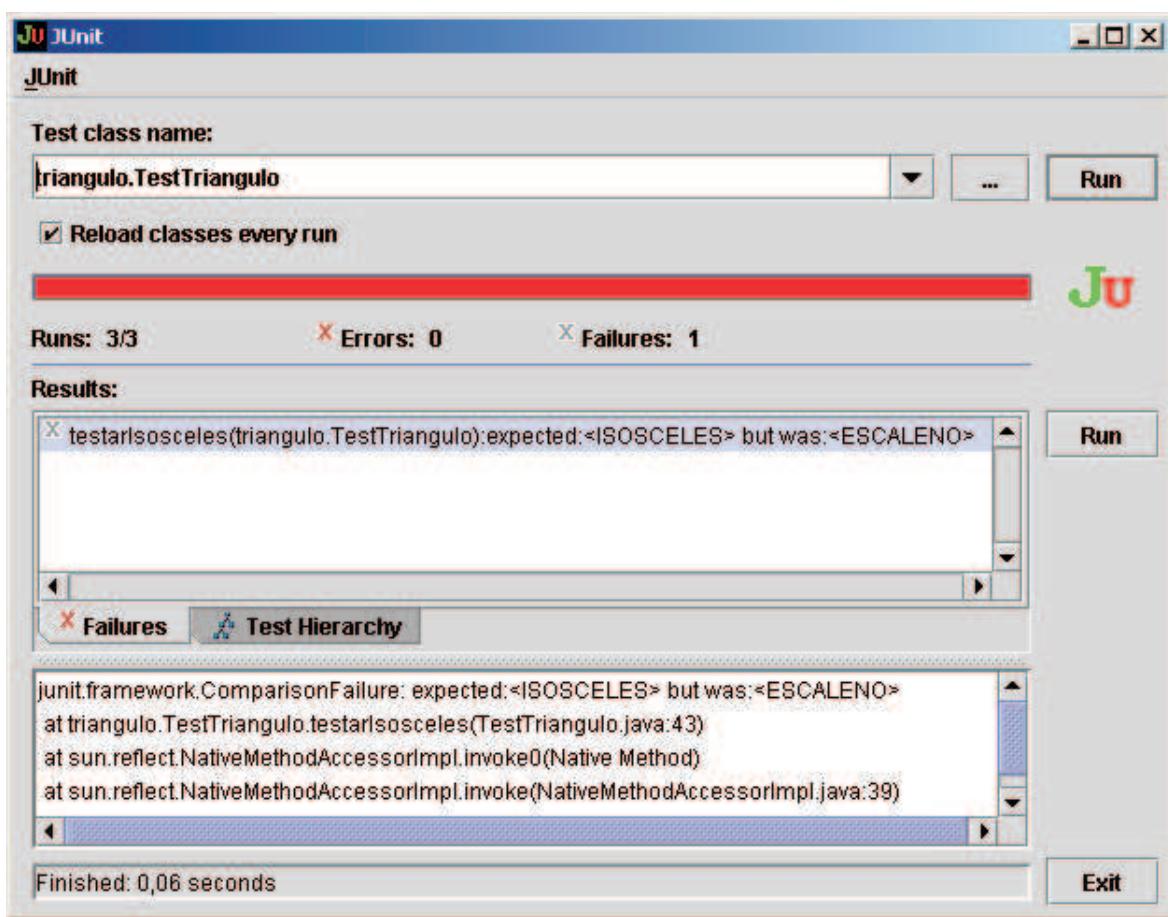


Figura 2.7. Exemplo de execução da classe de teste com a presença de uma falha.

suem suporte para ferramentas de análise de cobertura. Embora o suporte dessas ferramentas sejam importantes, o bom senso é algo fundamental. Os desenvolvedores têm uma boa noção quando a criação de teste pode ser finalizada.

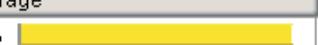
Class name	Coverage	Diff.	Missed lines
triangulo.Triangulo	94.44% 	94.44	1/18
triangulo.TestTriangulo	100% 	100.0	0/14

Figura 2.8. Trecho de uma tela mostrando o relatório de cobertura para o teste do triângulo.

2.3.1.2. Considerações Finais sobre o Teste de Unidade

Os testes de unidade podem auxiliar bastante o desenvolvimento de software com qualidade. Para isso os testes de unidade devem ser [18]:

- Automáticos. Os testes de unidade devem ser executados automaticamente e de forma periódica.
- Completos. Os testes de unidade devem testar todos os caminhos possíveis de serem executados em um objeto de teste.
- Independentes. Um teste de unidade não deve depender de outro, podendo ser executado individualmente, a qualquer hora e em qualquer ordem.
- Reproduzíveis. Os teste de unidade devem ser independentes do ambiente sobre o qual executam, devendo produzir os mesmos resultados em cada execução.
- Criados de forma profissional. Os testes de unidade devem ser criados de forma profissional, seguindo os mesmos critérios de qualidade utilizados para o código fonte do projeto.

As características apresentadas acima estão diretamente relacionadas à qualidade do teste de unidade. Existem diversos desenvolvedores que concordam que o desenvolvimento de testes de unidade são fundamentais para a criação de produtos de qualidade. Mas ainda existem muitos mitos associados aos testes de unidade, que precisam ser eliminados. Dentre eles destacamos dois principais :

- Escrever testes atrasa o desenvolvimento. Muitas vezes, por causa de prazos apertados, os testes são deixados de lado no desenvolvimento. No entanto, código gerado sem o apoio dos testes de unidade normalmente apresenta mais falhas e, quanto mais tarde se descobre uma falha, mais caro é a sua correção, atrasando ainda mais os prazos definidos.
- É melhor e mais fácil desenvolver testes depois do código pronto. Normalmente, todos os desenvolvedores testam seus códigos durante o desenvolvimento. O grande

problema é que nem todos criam esses testes de forma automatizada e reproduzível. O custo adicional para tornar um teste manual em um teste automático é pequeno e o benefício associado é muito grande. Além disso, o ideal é fazer os testes de unidade durante o desenvolvimento, visto que nesse momento, todas as regras associadas à parte do produto sendo desenvolvido devem estar bem assimiladas. Algumas técnicas, como o Desenvolvimento Dirigido por Testes [2], prescrevem até que os testes devem ser criados antes mesmo do código.

Os testes de unidade representam uma importante ferramenta para auxílio ao desenvolvimento de software com qualidade. De forma resumida, podemos destacar como principais vantagens relacionadas ao seu uso [10]:

- Testes de unidade auxiliam a descoberto de falhas mais cedo no ciclo de desenvolvimento, reduzindo os custos relacionados à correção dessas falhas.
- Testes de unidade validam o código ao longo do tempo, uma vez que, ao escrever um teste para parte de um código, essa parte estará sempre sendo validada, garantindo assim sua corretude (desde que o teste seja adequado!).
- Testes de unidade facilitam a refatoração de código. Por mais que seja desenvolvido um código enxuto e correto para um produto de software, fatalmente será necessário realizar alterações nele, seja devido a uma evolução do produto ou até mesmo correção de eventuais problemas. Os testes de unidade auxiliam essa tarefa, garantindo que o software continua funcionando, mesmo após grandes alterações.
- O conjunto de testes de unidade pode ser considerado uma documentação executável do código, uma vez que eles descrevem como o desenvolvedor espera que o código funcione.
- Testes de unidade aumentam a confiança dos desenvolvedores e facilitam o trabalho em equipe, pois sabendo que existem testes para as diversas unidades desenvolvidas, os desenvolvedores ficam mais seguro ao alterar trecho de códigos escrito por outras pessoas. Isso favorece o trabalho em equipe.

2.3.2. Teste de Integração

Testes de Integração têm por objetivo verificar as interfaces entre as partes de uma arquitetura de produto. De forma mais geral, esses testes têm por objetivo verificar se as unidades implementadas, em cada iteração, funcionam corretamente, em conjunto com as unidades já implementadas e testadas a partir de testes de unidade, em iterações anteriores.

O teste de integração é uma atividade cujo objetivo é descobrir erros associados às interfaces entre os módulos quando esses são integrados para construir a estrutura do software que foi planejada.

Para realização do teste de integração é necessário definir uma estratégia de integração. De forma geral, a estratégia de integração determina como os diversos componentes do sistemas serão agrupados e como a verificação do funcionamento, agora em conjunto, será realizada.

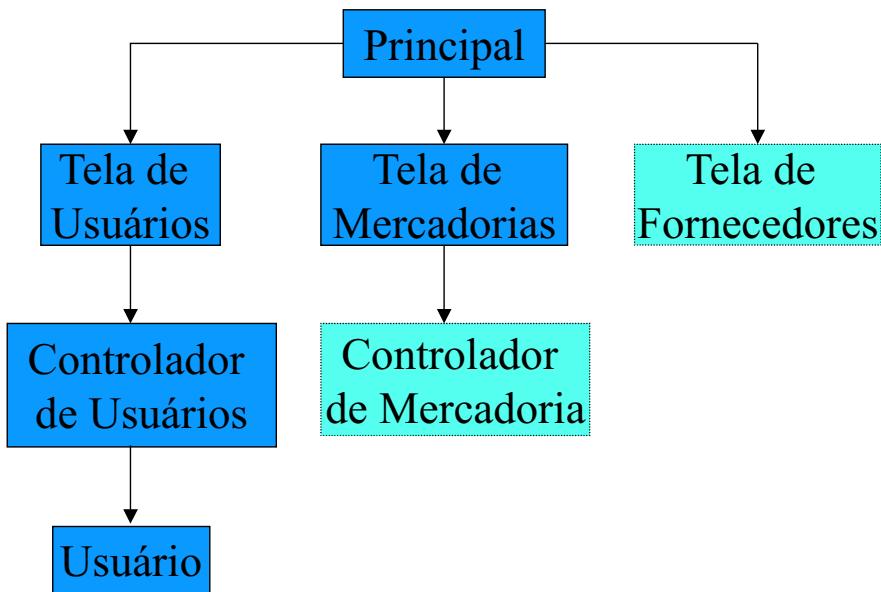


Figura 2.9. Exemplo de integração utilizando a técnica Top-Down.

Alguns desenvolvedores utilizam uma "estratégia" denominada "big-bang", que pode ser resumida em: não utilize estratégia alguma! Ou seja, as unidades são integradas sem qualquer ordem, seguindo a seguinte prescrição: "vamos juntar tudo para ver se funciona!".

O mais indicado para o desenvolvimento profissional de produtos de software é utilizar uma estratégia de Integração Incremental, em que o programa seja construído e testado em pequenos segmentos. Com isso, a correção de erros torna-se mais fácil e as interface entre as unidades tem maior probabilidade de serem testadas completamente.

Existem diversas estratégias incrementais de integração que podem ser consideradas em um projeto. Não existe um consenso sobre qual a melhor técnica: na maioria dos casos cada organização determina o que é melhor para seus projetos.

A Integração Top-down é uma estratégia incremental em que os módulos são integrados de cima para baixo. Assim, em um típico Sistema de Informação, como por exemplo um Sistema de Controle de Mercearias [8], iniciariam a integração a partir do módulo principal do sistema, verificando seu funcionamento integrado com cada uma das telas de acesso a funções do sistema, para em seguida verificar a integração dessas telas às unidades responsáveis pelas regras de negócio, e assim sucessivamente. Isso é apresentado na Figura 2.9. A cor mais escura significa que a classe já foi integrada. Classes representadas por retângulos mais claros indicam que ainda necessitam ser integradas ao sistema.

Na Integração Bottom-Up utilizamos o caminho inverso. Iniciamos pelas unidades localizadas inferiormente na hierarquia de unidades. Módulos inferiores são combinados em grupos e drivers são escritos para testar esses grupos. À medida que as classes superiores estão concluídas e testadas, iniciamos a integração com os grupos inferiores. Isso é feito a partir da junção da classe a ser integrada às classes inferiores agrupadas, conforme apresentado na Figura 2.10. Nessa figura, devemos considerar que as classes inferiores,

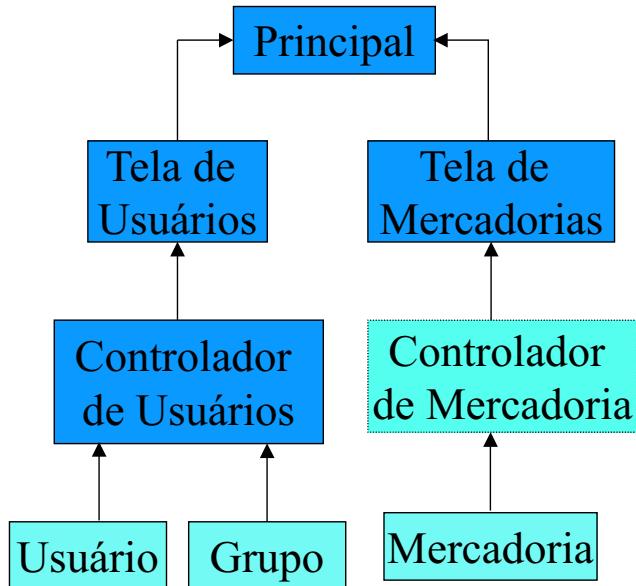


Figura 2.10. Exemplo de integração utilizando a técnica Bottom-Up.

com cores mais claras, já foram testadas e integradas.

As ferramentas para o teste de integração são as mesmas utilizadas para o teste de unidade. No entanto, durante o teste de integração, pode ser maior a necessidade de se criar elementos que simulem outros, uma vez que diversos módulos ainda não implementados podem ser necessários em um teste. Para isso, e também para os testes de unidade, existem ferramentas para criação de módulos falsos, que simulam módulos reais ainda não implementados. Para o mundo orientado a objetos, existem os Mock Objects [9]. Como o próprio nome indica, Mock Objects são objetos que simulam, ou imitam, o comportamento de objetos reais do sistema, de uma maneira controlada.

A execução periódica dos testes de integração, de forma sistematizada, é conhecida como Integração Contínua [7]. Existem diversos arcabouços para a integração contínua, como por exemplo o Ant e o Cruise Control. Essas ferramentas auxiliam na execução periódica desses testes, enviando diferentes formas de mensagens para grupos pré-configurados. Isso auxilia bastante o desenvolvimento, pois mantém toda uma equipe informada sobre a possível existência de falhas no estado atual do software em construção.

2.3.3. Teste de Sistema

O objetivo do Teste de Sistema é executar o sistema sob ponto de vista de seu usuário final, verificando o correto atendimento a todos os requisitos funcionais e não-funcionais. Esses testes devem ser executados em condições similares - de ambiente, interfaces sistêmicas e massas de dados - àquelas que um usuário utilizará no seu dia-a-dia. Por causa disso é que esse nível de teste é o ideal para a verificação dos requisitos não-funcionais.

Assim, durante o teste de sistema precisamos realizar um teste funcional, para saber se os diversos requisitos funcionais foram implementados corretamente, assim como diversos testes relacionados a requisitos não-funcionais, como por exemplo, teste de desempenho e estresse, teste de segurança e teste de usabilidade. Nas próximas seções

falaremos sobre os objetivos de teste, detalhando um pouco mais os testes funcionais e os teste de desempenho e estresse.

2.4. Objetivos do Teste

O objetivo do teste está associado às condições e propriedades específicas do software sob teste. Testes podem ser criados para verificar se as especificações funcionais estão corretamente implementadas (teste funcional), podendo ser executados diretamente pelos usuários finais, para decidir sobre a aceitação do produto desenvolvido (teste de aceitação); teste podem verificar se o desempenho do software está dentro do aceitável (teste de desempenho); se ele funciona sob condições anormais de demanda (teste de estresse); se o software é adequado ao uso (teste de usabilidade); teste podem ter como objetivo mostrar que o software não regrediu, e continua funcionando conforme estava anteriormente a uma alteração (teste de regressão); se os procedimentos de instalação são corretos e bem documentados (teste de instalação); para verificar o seu nível de segurança (teste de segurança); ou para verificar seu funcionamento, a partir da liberação do produto para pequenos grupos de usuários trabalhando em um ambiente controlado (teste alfa) ou em um ambiente não controlado (teste beta).

Esses objetivos citados acima não constituem todos os objetivos existentes - ainda existem outros. Neste trabalho detalhamos um pouco mais três tipos de teste: o teste funcional, o teste de desempenho e o teste de estresse.

2.4.1. Teste Funcional

Os testes funcionais devem verificar a consistência entre o produto implementado e os respectivos requisitos funcionais. Esses testes geralmente são executados por uma equipe independente de testes, agindo como se fosse o usuário final do produto e tentando encontrar inconsistências entre o que foi solicitado e o que foi desenvolvido.

Um importante aspecto relacionado ao teste funcional é a questão: devemos ou não automatizar esse tipo de teste? A resposta pode ser obtida pelo leitor a partir da análise da seguinte situação: imagine que você está testando um software e, para isso, projetou 48 testes diferentes. Você iniciou a execução dos testes, sempre anotando os resultados obtidos. Cada teste, durou, em média, três minutos para ser realizado, visto que precisavam de alguns dados iniciais e alguns trechos do programa possuíam dependências com outras parte do software. Ou seja, em aproximadamente 144 minutos (+2h30) você conseguiria testar o software por completo. Agora imagine que, ao executar o 43º teste, você descobriu um problema. Feito isso, uma ação normal a ser tomada é avisar aos desenvolvedores o fato. Após isso, os desenvolvedores provavelmente devem corrigir os problemas, mas existem grandes chances de que outras partes do software sejam afetadas. Assim, você terá que refazer os 48 testes novamente. Normalmente, tais testes devem ser executados, dezenas, centenas e até milhares de vezes novamente. A automação de um conjunto de testes normalmente demanda bem mais esforço que sua execução manual, mas uma vez automatizado, sua execução é bastante simples, se atendo a execução de um script ou o clique de alguns botões.

Resumindo: automatizar testes funcionais é, na maioria dos casos, um bom investimento!



Figura 2.11. Tela de Login do Merci sem usuário logado.

2.4.1.1. Ferramentas para o Teste Funcional

Existem diversas ferramentas para a automação do teste funcional. A maioria dessas ferramentas se baseia na gravação de todas as ações executadas por um usuário, gerando um script dos passos executados. As ações gravadas podem ser facilmente re-executadas, permitindo assim a automação do teste. O script gerado pode ser facilmente alterado, permitindo uma modificações nos testes gravados sem a necessidade de re-execução do software e nova captura. Dentre as facilidades existentes nesse tipo de ferramenta, existe a possibilidade de acessar o valor dos diversos elementos existentes nas telas do software, com o intuito de verificar os dados existentes. A partir disso é que os testes são criados.

Para explicar a partir de um exemplo, utilizaremos nesta parte do texto a Tela de Login do software Merci, que é o exemplo contido no livro texto que descreve o processo Praxis [8]. Nas figuras 2.11 e 2.12 apresentamos a imagem da tela sem usuário e com usuário logado. Podemos notar que ao logar no software a tela de login muda seu estado, exibindo uma configuração diferente de campos e comandos ativos. Por exemplo, na Figura 2.11 podemos notar que apenas o botão Login encontra-se ativo, enquanto que na Figura 2.12 o botão Login não está ativo, mas os botões Logoff e Alterar Senha estão. Os testes que vamos criar precisam verificar essas habilidades, pois a não ativação de um botão é uma falha e pode gerar grandes problemas para os usuários de um produto.

Como exemplo de ferramenta de apoio para o teste funcional, utilizaremos o Abbot [24], que é uma ferramenta de captura e re-execução para software desenvolvidos utilizando a linguagem Java. Na Figura 2.13 apresentamos a tela principal do Abbot.

Para se executar o Abbot, basta executar o arquivo Abbot.jar, utilizando a seguinte linha de comando: `java -jar Abbot.jar`. Em seguida, será necessário criar um novo script, a partir da execução dos comandos *File→New Script* para em seguida informar os dados para executar o software a ser testado, indicando o nome da classe principal do sistema (*target classe name*) e o diretório onde essa classe se encontra (*classpath*). Para o Merci, a classe principal é *br.ufmg.dcc.merci10.MerciPrincipal*, e o diretório dela



Figura 2.12. Tela de Login do Merci com usuário logado.

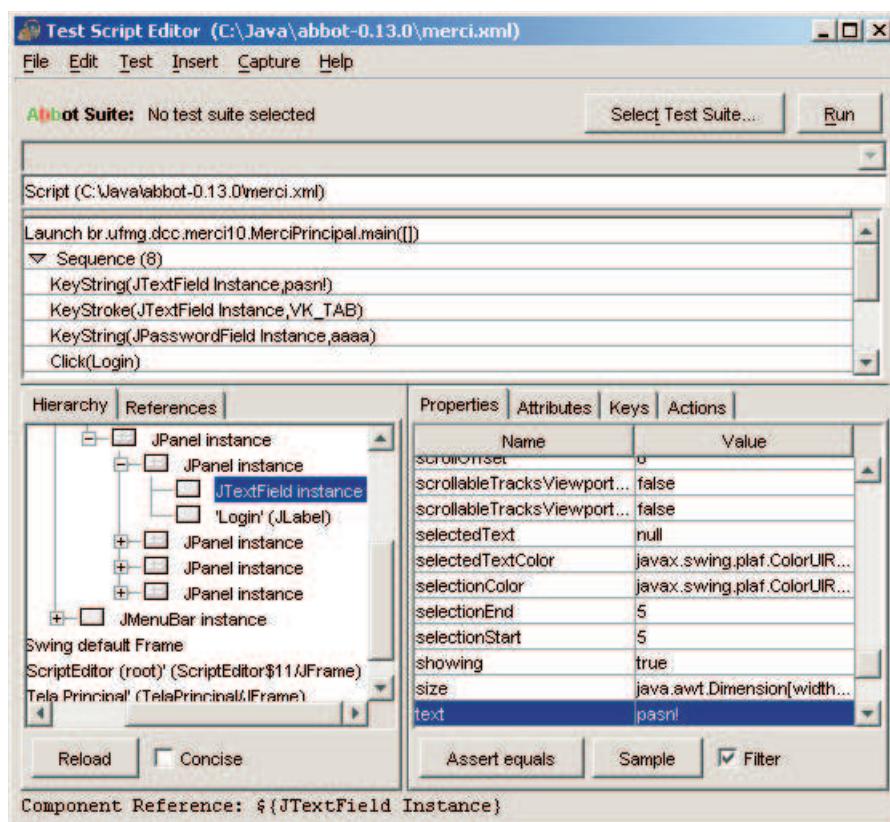


Figura 2.13. Tela principal do Abbot.

é `c:\temp\MerciPortOdbc`.

A execução da aplicação pode ser feita a partir da opção Test→Launch. Com isso temos o Abbot iniciado, assim como a aplicação a ser testada.

Para criar um teste, precisamos primeiro definir quais ações precisam ser realizadas (procedimento de teste), quais dados serão usados e quais resultados devem ser obtidos (caso de teste). Por exemplo, podemos automatizar o caso de teste exibido na Figura 2.5. Nesse caso, estamos fazendo um teste cujo objetivo é verificar se ao tentar fazer login, utilizando um login inválido (contendo caracteres não permitidos), o sistema exibe a mensagem de erro especificada no projeto do Merci.

Precisamos então automatizar o teste descrito anteriormente. Isso requer a gravação das ações associadas ao teste. Para iniciar a gravação, devemos selecionar a opção Capture→All actions (no motion). Essa opção não captura as informações de movimentação do mouse na tela, visto que isso gera scripts muito grandes. Mas em determinadas situações pode ser necessário utilizar a captura de movimentos. Iniciando a gravação, basta executar o Merci, utilizando os dados previstos no teste. Ao finalizar as ações do teste, basta retornar para a tela do Abbot para que o script seja concluído. Nesse momento irá aparecer na tela do Abbot uma seqüência de ações. Essas ações correspondem aos passos gravados. A Figura 2.13 mostra a tela do Abbot com o script gravado. Nela podemos notar, por exemplo, a existência do passo `KeyString(JTextField Instance,pasn!)`. Esse comando é responsável pela entrada do valor `pasn!` no campo `JTextField Instance` da Tela de Login. Esse campo é, na verdade, o campo login da tela.

Para se executar o teste criado, basta selecionar a opção Test→Run. Nesse caso todas as ações serão executadas novamente. Isso que fizemos até agora não é um teste pelo simples fato de não verificar muita coisa. Uma das ações que temos que fazer é verificar se a mensagem exibida está de acordo com o que foi especificado. Precisamos então verificar propriedades dos componentes da tela. Para isso temos que acessar os dados do componente. Isso pode ser feito da seguinte forma: devemos deixar o cursor do mouse em cima do componente a ser verificado e em seguida pressionar as teclas `Alt+Shift+F1`. Isso seleciona o componente na tela do Abbot, conforme exibido na Figura 2.13 (componente `JTextField Instance`). Em seguida, você deve pesquisar, no lado direito da tela do Abbot, a propriedade a ser verificada. Por exemplo, a propriedade relacionada ao valor contido em um campo do tipo `JTextField` é a propriedade `text`, assim, devemos fazer uma verificação nesse valor. Conforme exibido na Figura 2.13, o texto presente nesse campo é `pasn!`. Da mesma forma, poderíamos capturar o componente relacionado à exibição de mensagem para verificar seu valor. Um ponto de verificação é inserido no script quando clicamos no botão `Assert equals` localizado próximo a tabela de propriedades do componente. Ou seja, durante a re-execução do script, podemos inserir a verificação de certos valores nos componentes da tela.

2.4.2. Testes de Desempenho e Estresse

Testes de desempenho consistem em testar um sistema quanto aos seus requisitos de desempenho. Alguns exemplos desses requisitos são: latência, que é o tempo entre uma requisição e a completude e resposta da operação requisitada; vazão (*throughput*), o número de operações que o sistema é capaz de completar em um dado período de tempo;

escalabilidade, a quantidade de usuários simultâneos que o sistema pode lidar; uso de recursos de máquina, como memória e processamento, dentre outros [6]. Como supracitado, é necessário que haja um conjunto bem definido de objetivos para esses requisitos, do contrário, esses testes serão medições cegas [25]. Apesar de um teste de desempenho completo e ideal depender da existência de um sistema totalmente integrado e funcional, no contexto sobre o qual o mesmo irá funcionar, testes de desempenho são freqüentemente aplicados em todos os passos do processo de teste [23]. Muitos autores afirmam que isso é uma boa prática, baseando-se na máxima que diz que quanto mais cedo uma falha é detectada, mais eficiente e barata é a sua solução, principalmente levando em consideração o fato de que a maioria dos problemas críticos de desempenho advém de decisões feitas em estágios iniciais do ciclo de desenvolvimento do software [6].

Testes de estresse normalmente são feitos juntamente com testes de desempenho, não raro havendo confusão entre os mesmos. Enquanto o teste de desempenho tem como objetivo testar se determinado sistema é capaz de lidar com a carga definida nos requisitos, o teste de estresse consiste em submeter o sistema a situações anormais de uso [21], como grandes quantidades de carga, comportamento anormal de portas em um servidor, redução dos recursos computacionais disponíveis, entradas não realistas de dados. Assim, é possível observar o comportamento do sistema durante essas situações críticas, identificando falhas potencialmente difíceis de serem encontradas em situações normais, mas não toleráveis, como o vazamento de informações confidenciais de um banco de dados em mensagens de erro, por exemplo [3].

Testes de desempenho e estresse geralmente são onerosos. Eles exigem uma equipe dedicada a testes, diferente da equipe de desenvolvimento. Isso acontece por que os desenvolvedores, via de regra, não são capazes de exercer o afastamento necessário do seu produto e nem capazes de pensar como o usuário final, o que é uma necessidade para a realização de tais testes [21]. Além disso, esses testes demandam instrumentação de software e hardware, além de testadores hábeis e com bom conhecimento nas ferramentas escolhidas. São testes caros, porém importantes, pois em softwares de médio a grande porte, os principais problemas relatados após a entrega do software são relativos a degradação de desempenho ou a incapacidade do sistema de lidar com a vazão exigida. Isso acontece porque é comum o software ser largamente testado quanto à funcionalidade, mas não ter seu desempenho apropriadamente testado [25]. Além disso, existem problemas que não são detectados em testes funcionais que são bem mais visíveis quando o sistema é testado em situações de estresse, mas que podem acontecer em situações consideradas normais [21]. Além disso, com a consolidação dos sistemas Web, questões relativas ao desempenho e comportamento sob estresse ganharam ainda mais visibilidade. O desempenho é um fator de vital importância em uma aplicação Web, por afetar diretamente o tempo de resposta para o usuário, ponto central no sucesso de público de uma aplicação Web. Além disso, a previsibilidade das condições de funcionamento de uma aplicação Web é algo ainda mais difícil de obter, o que torna o sistema ainda mais suscetível a enfrentar situações de estresse, por mais irrealistas que essas situações sejam, tornando imprescindível a realização de testes de estresse nesse tipo de sistema [21].

2.5. Técnicas de teste

Um dos objetivos do teste é revelar falhas e para isso muitas técnicas foram desenvolvidas. O princípio básico por trás de uma técnica é ser sistemática, ao ponto de identificar um conjunto representativo de comportamentos dos programas.

Essas técnicas são freqüentemente classificadas como caixa branca ou caixa preta. Técnicas de caixa branca têm por objetivo determinar defeitos na estrutura interna do produto, a partir do desenho de testes que exercitem suficientemente os possíveis caminhos de execução (esses testes também são conhecidos como teste estrutural ou teste baseado no código). As técnicas de teste caixa preta têm por objetivo determinar se os requisitos foram totais ou parcialmente satisfeitos pelo produto (esses testes também são conhecidos como teste funcional ou teste baseado na especificação).

Existem diversas classificações para as técnicas existentes. Neste trabalho utilizamos a classificação proposta pelo IEEE [16], nos atendo a algumas técnicas baseadas na especificação e técnicas baseadas em erros.

2.5.1. Técnicas baseadas na especificação

As duas principais técnicas baseadas na especificação são: o particionamento de equivalência e a análise de valor limite. A seguir descrevemos tais técnicas, utilizando como base a descrição contida no livro do processo Praxis [8].

No **Particionamento de equivalência** o domínio de entrada é subdividido em uma coleção de subdomínios, ou classes equivalentes, e um conjunto de testes representativos é extraído a partir de cada classe identificada. Cada categoria tem a capacidade de revelar uma classe de erros, permitindo que casos de teste na mesma categoria sejam eliminados sem que se prejudique a cobertura dos testes. Para cada entrada do sistema, são identificados os conjuntos de valores válidos e inválidos associados que definem as classes de equivalência para essa entrada.

Em um programa de gestão de pessoal, por exemplo, a idade de um funcionário pode variar entre 15 e 80 anos (de acordo com a especificação de requisitos do programa). As classes de equivalência para essa entrada são todos os valores inteiros menores que 15, valores inteiros entre 15 e 80, e valores inteiros maiores que 80. Para cada uma dessas classes, qualquer valor tem, potencialmente, a mesma capacidade de detectar erros, sendo dispensável a execução de vários testes para valores pertencentes à mesma classe de equivalência.

Na **Análise de valor-limite** os casos de teste são escolhidos próximo ou nas fronteiras dos domínios de entrada, uma vez que boa parte das falhas tendem a se concentrar próximo a esses valores. O emprego dessa técnica deve ser complementar ao emprego da partição de equivalência. Assim, em vez de se selecionar um elemento aleatório de cada classe de equivalência, selecionam-se os valores nas extremidades de cada classe.

Utilizando o exemplo anterior, relacionado à idade de um funcionário em um sistema de gestão de pessoal, devemos selecionar como valores representantes das partições, os valores limítrofes. Nesse caso, para representar a partição de valores abaixo de 15 anos utilizaríamos 14, para representar valores entre 15 e 80 utilizaríamos o valor 15 e 80, e para representar valores na partição acima de 80 utilizaríamos o valor 81. Esses valores

1. if ($x > 0$) 2. doThis(); 3. if ($x > 10$) 4. doThat();	1. if ($x < 0$) 2. doThis(); 3. if ($x > 10$) 4. doThat();
1. if ($x > 0$) 2. doThis(); 3. if ($x < 10$) 4. doThat();	1. if ($x \geq 0$) 2. doThis(); 3. if ($x > 10$) 4. doThat();

Tabela 2.2. Exemplo de mutações.

atendem tanto o particionamento em classes de equivalência quanto a análise de valor limite.

2.5.2. Técnicas baseadas em erros

As técnicas de teste baseadas em erros utilizam informação sobre os erros mais comumente detectadas no processo de desenvolvimento e sobre tipos específicos de erros que se deseja revelar. Duas técnicas bem conhecidas baseadas em erros são: a semeadura de erros [4] e análise de mutantes [5].

Na **Semeadura de Erros** uma quantidade específica de falhas é inserida artificialmente no programa. Após o teste, do total de erros encontrados, verifica-se quais são naturais e quais são artificiais. Usando estimativas de probabilidade, o número de erros naturais ainda existentes no programa pode ser estimado [11].

A **Análise de Mutantes** é uma técnica que utiliza um conjunto de programas ligeiramente modificados (mutantes), obtidos a partir do programa original (P), para avaliar o quanto um conjunto de testes (T) é adequado ao programa (P). O objetivo é encontrar um conjunto de testes (T) capaz de revelar todas as diferenças de comportamento entre P e seus mutantes. A Tabela 2.2 apresenta o exemplo de um programa original (acima e à esquerda) e três mutantes. Observe que a mutação gera programas diferentes e que deveriam ser detectados por casos de teste para testar o programa original. Caso os mutantes gerados tenham um comportamento idêntico ao programa original diríamos que os mutantes são equivalentes. Se o conjunto de testes (T) detecta a diferença existente no mutante, dizemos que T matou o mutante. A partir dos mutantes mortos e do total de mutantes não equivalentes podemos obter o escore de mutação (Em), que é o número de mutantes mortos (Mm) dividido pelo número total de mutantes gerados (Tm) menos os mutantes equivalentes (Me), ou seja: $Em = Mm / (Tm - Me)$.

2.6. Documentos de teste

Independentemente do processo de software utilizado durante o desenvolvimento, existem artefatos diretamente relacionados à atividade de teste [15]. Dentre esses artefatos destacamos:

- Plano de Teste. Documento que detalha o escopo, abordagem, recursos e agenda das atividades de teste. Além disso, identifica os ítems de teste, construções a serem

testadas, tarefas a serem realizadas, pessoal para realização e riscos associados. Esse documento é produzido durante a atividade *Planejamento*.

- Especificação de Casos de Teste. Documento especificando entradas, resultados esperados e um conjunto de condições de execução para um item de teste. Esse documento é produzido durante a atividade *Desenho*.
- Especificação dos Procedimentos de Teste. Documento contendo os passos para execução de um conjunto de casos de teste. Esse documento também é produzido durante a atividade *Desenho*.
- Relatório de Incidentes. Documento relatando qualquer evento ocorrido durante o processo de teste que requer investigação. Esse documento é produzido durante a atividade *Execução*.
- Log de testes. Registro cronológico de detalhes sobre a execução dos testes. Esse documento também é produzido durante a atividade *Execução*.
- Relatório de Testes. Documento resumindo as atividades de teste os resultados obtidos, incluindo uma avaliação dos testes executados. Esse documento é produzido durante a atividade *Verificação de término* e *Balanço final*.

2.7. Ferramentas de apoio às atividades de teste

Nesta seção apresentamos as principais categorias de ferramentas de teste existentes no mercado. Não encontramos uma terminologia amplamente difundida para essas ferramentas, assim, tentamos utilizar aquilo que achamos mais apropriado. Nossa classificação é apresentada a seguir:

- Gestão de testes: ferramentas para organizar o conjunto de testes, facilitando sua administração e documentação.
- Geração de testes: ferramentas que auxiliam a geração de dados de testes, a partir de modelos representando o formato das entradas permitidas. Apenas as entradas dos testes são geradas, a geração dos resultados esperados exige a presença de um oráculo.
- Captura e re-execução: ferramentas que gravam todas as ações executadas durante a execução de um software e que permitem sua execução automática posterior sem supervisão humana.
- Integração contínua: ferramentas que aceitam seqüências de teste manualmente criadas, automaticamente geradas ou pré-gravadas e as executam sem supervisão humana, durante intervalos de tempo pré-definidos.
- Teste de desempenho e estresse: ferramentas para submeter um software a certas condições funcionamento, utilizando recursos de quantidade, freqüência ou volume configuráveis, permitindo assim analisar o atendimento aos requisitos não-funcionais existentes, assim como o funcionamento do sistema em situações anormais, com sobrecarga.

- Análise de perfil: ferramentas que realizam medições no software em execução, gerando um perfil contendo informação sobre memória utilizada, métodos invocados, tempo de execução e outros parâmetros úteis para a descoberta de eventuais gargalos no sistema.
- Teste de segurança: ferramentas que realizam uma varredura em portas do sistema para detectar pontos de vulnerabilidades. Existem também ferramentas para interceptação de requisições permitindo sua alteração, para tentar burlar regras do sistema.
- Acompanhamento de defeitos: ferramentas para acompanhamento de qualquer defeito em um software, desde a sua criação até sua solução, possibilitando a visualização, pesquisa, impressão e administração de mudanças no software, relacionadas aos erros gerenciados.
- Avaliação dos testes: ferramentas que auxiliam a avaliação da qualidade dos testes, seja a partir da análise na cobertura atingida, seja a partir da utilização de técnicas baseadas em erros. Essas ferramentas geralmente implementam diferentes critérios de avaliação.
- Ferramentas de TBM: geram os dados de entrada dos testes, as saídas esperadas e também os dados exigidos para a execução dos testes, além de outras condições necessárias. Algumas ferramentas possuem suporte para criação de novos testes, ambiente para gestão desses testes, incluindo execução automática, e avaliação da cobertura atingida. Essas ferramentas representam o que existe de mais avançado com relação a automação de testes, mas exigem que modelos descrevendo o software sejam fornecidos como entrada.
- Ferramentas de Teste de Unidade: essas ferramentas contém funções que auxiliam a escrita de testes para exercitar unidades do sistema, tornando fácil a criação de programas que verificam outros programas.

2.8. Atividades de Teste

Neste trabalho descrevemos o Fluxo de Teste do Processo Praxis [8]. As atividades contidas no fluxo são muito similares às atividades prescritas pelo IEEE [15]. O Fluxo de Teste (Figura 2.14) tem dois grandes grupos de atividades para cada bateria a ser desenvolvida: preparação e realização. O plano da bateria é elaborado durante a preparação, juntamente com o desenho das especificações de cada teste. Durante a realização, os testes são executados, os erros encontrados são, se possível, corrigidos e os relatórios de teste são redigidos. A preparação e realização de cada bateria correspondem a uma execução completa do Fluxo de Teste.

O grupo de preparação compreende as atividades de Planejamento e Desenho. No Planejamento normalmente é produzido o Plano de Teste, com ênfase na parte gerencial necessária para execução das atividades de teste. As ferramentas de Gestão de Testes são utilizadas para auxiliar a execução dessa atividade.

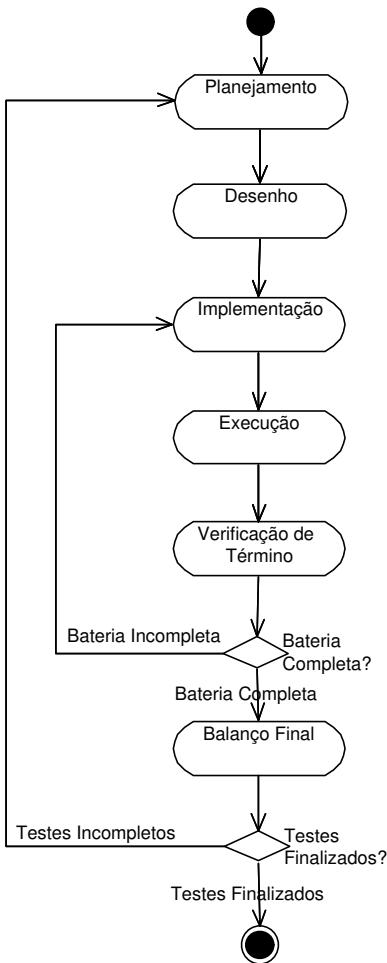


Figura 2.14. O Fluxo de Teste.

Em seguida é realizado o Desenho, devendo ser produzido o documento contendo as especificações dos testes. Isso inclui o documento com os Procedimentos de Teste e os Casos de Teste. Algumas ferramentas de geração de testes podem ser utilizadas nessa atividade, facilitando assim a tarefa de criar os testes a serem executados.

O grupo de realização é formado pelas demais atividades: Implementação, que monta o ambiente de testes; Execução, que os executa efetivamente, produzindo os principais relatórios de testes; Verificação do término, que determina se a bateria pode ser considerada como completa; e Balanço final, que analisa os resultados da bateria, produzindo um relatório de resumo.

Durante a Implementação, os testes especificados no Desenho, devem ser implementados. Uma série de ferramentas podem ser utilizadas, tais como as ferramentas de captura e re-execução, ferramentas de teste de unidade, ferramentas para o teste de desempenho e estresse e ferramentas para o teste de segurança.

Na Execução são utilizadas ferramentas que auxiliem a automação da execução dos testes, como por exemplo, as ferramentas de integração contínua. Paralelamente, são utilizadas ferramentas de avaliação dos testes e de análise de perfil, para facilitar a execução da próxima atividade. Ainda durante a execução, os erros encontrados são registrados, para que as devidas providências sejam tomadas. Geralmente são utilizadas as ferramentas de acompanhamento de defeitos nessa tarefa, visto que elas possuem uma série de facilidades que auxiliam o controle de defeitos e correções.

Na Verificação do término os relatórios produzidos durante a execução são analisados, para se verificar se o teste pode ser considerado finalizado. São analisados os relatórios de cobertura, o perfil gerado durante a execução dos testes, assim como os erros registrados. Com base nesses dados é que decidimos se o teste deve ou não continuar.

No Balanço final devemos registrar as lições aprendidas de forma a facilitar a realização dos testes em outros produtos. Se possível, são identificados artefatos reutilizáveis em outros testes, inclusive procedimentos, casos e componentes de teste.

2.9. Conclusão

Neste trabalho apresentamos uma visão geral do teste de software. Apresentamos alguns conceitos relacionados, os níveis de teste, e algumas técnicas de teste. Apresentamos também as atividades de teste, relacionando-as aos principais documentos de teste, bem como às principais categorias de ferramentas de testes existentes.

O teste de software é fundamental para o desenvolvimento de produtos de qualidade. Os dois principais problemas relacionados à não execução das atividades de teste nas organizações desenvolvedoras de software são a falta de cultura nessa disciplina e os custos associados com a execução dessas atividades. Neste trabalho demonstramos como realizar alguns tipos de teste, tentando facilitar a adoção de testes no desenvolvimento de software. Além disso, é importante frisar que a execução de testes favorece a qualidade, uma vez que os requisitos são verificados constantemente. O teste também permite encontrar falhas mais cedo no desenvolvimento, reduzindo, de forma geral, os custos do desenvolvimento. Os testes ajudam a tornar os requisitos mais estáveis, uma vez que, quanto mais cedo se inicia o planejamento dos testes, mais cedo podemos verificar a consistência dos requisitos. Os testes ainda permitem o acompanhamento contínuo da qualidade e da produtividade, pois testando podemos ter uma idéia mais precisa sobre o percentual do sistema que já foi concluído, evitando falsas expectativas. Isso facilita o gerenciamento e auxilia a todos os profissionais envolvidos com o desenvolvimento de software.

Em resumo, o teste, embora aparentemente introduza custos no desenvolvimento, é capaz de reduzir o custo total do projeto. Além disso, no mercado cada vez mais competitivo atual, entregar produtos com baixa qualidade pode ser destrutivo para algumas empresas, valendo a pena o investimento nessa disciplina.

Referências

- [1] V. Basili, R. Selby, and D. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, 12(7):733–743, July 1986.

- [2] K. Beck. *Test-Driven Development by Example*. Addison-Wesley Signature Series, 2002.
- [3] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.
- [4] T. Budd. *Mutation Analysis: Ideas, Examples, Problems and Prospects*. North-Holand Publishing Company, 1981.
- [5] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.
- [6] G. Denaro, A. Polini, and W. Emmerich. Early performance testing of distributed software applications. In *WOSP '04: Proceedings of the 4th international workshop on Software and performance*, pages 94–103, New York, NY, USA, 2004. ACM Press.
- [7] P. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.
- [8] W. P. Filho. *Engenharia de Software: Fundamentos, Métodos e Padrões*. LTC, 2nd edition, 2003.
- [9] S. Freeman, T. Mackinnonm, N. Pryce, and J. Walnes. Companion to the 19th annual acm sigplan conference on object-oriented programming, systems, languages, and applications, oopsla 2004, october 24-28, 2004, vancouver, bc, canada. In *OOPSLA Companion*, pages 236–246. ACM, 2004.
- [10] A. Gazola and E. David. Testes unitários para camadas de negócios no mundo real. *Mundo Java*, (23):54–61, 2007.
- [11] A. L. Goel. Software reliability models: Assumptions, limitations, and applicability. *IEEE Transactions on Software Engineering*, 11(12):1411–1423, December 1985.
- [12] J. Goodenough and S. Gerhart. Towards a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, June 1975.
- [13] T. Husted and V. Massol. *JUnit in Action*. Manning Publications, 2003.
- [14] IEEE. *IEEE Standard Glossary of Software Engineering Terminology - IEEE Std.610.12-1990*. IEEE Computer Society, 1990.
- [15] IEEE. *IEEE Standard for Software Test Documentation - IEEE Std 829-1998*. IEEE Computer Society, 1998.
- [16] IEEE. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2004.
- [17] C. Lott and D. Rombach. Repeatable software engineering experiments for comparing defect-detection techniques. volume 1, pages 241–277, 1996.

- [18] B. Marick. *The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing*. Prentice Hall, 1995.
- [19] MCT. *Programa Brasileiro da Qualidade e Produtividade de Software*. Ministério da Ciência e Tecnologia, 3rd edition, September 2004.
- [20] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Professional, 2007.
- [21] G. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.
- [22] NIST. Planning Report 02-3, National Institute of Standards and Technology, <http://www.nist.gov/director/prog-ofc/report02-3.pdf>, 2002.
- [23] R. Pressman. *Engenharia de Software*. McGraw-Hill, 6th edition, 2006.
- [24] T. Wall. Abbot Java GUI Test Framework. Available Online, <http://abbot.sourceforge.net/>, last access on May 2005.
- [25] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26(12):1147–1156, 2000.

Capítulo

3

Desenvolvimento de Software Utilizando Integração Contínua

Francisco Nauber Bernardo Góis, Rafael Braga de Oliveira e Pedro Porfírio Muniz Farias

Abstract

Continuous integration is a software engineering practice where the cycle including development, test and build generation is frequently executed. Due to the high execution frequency of these tasks is necessary to do this work with the support of automated tools. Generally, but not necessarily, continuous integration is associated with Test Driven Development (TDD). In this chapter, besides continuous integration and TDD, are presented some tools that automate the software development process.

Resumo

O Desenvolvimento utilizando integração continua é uma prática onde o ciclo envolvendo o desenvolvimento, testes e a geração de novas versões do sistema é freqüentemente executado. Em virtude da freqüência com que tais tarefas são realizadas é necessário contar com o suporte de várias ferramentas que automatizam o processo de desenvolvimento. Normalmente, mas não necessariamente, o desenvolvimento utilizando integração continua está associado ao desenvolvimento dirigido a testes (TDD). Neste capítulo, além dos conceitos de integração contínua e TDD, discorreremos sobre algumas ferramentas que auxiliam a automatização do desenvolvimento.

3.1. Introdução

A lógica de vários processos de desenvolvimento de *software* pressupõe a completa construção de unidades isoladas que, uma vez concluídas, devem ser testadas através de testes unitários. Após o sucesso desses testes, estas unidades devem ser integradas. Em seguida, serão realizados os testes de integração e, finalmente, deve ser gerada uma versão estável do sistema.

Nestes processos, as unidades de código são construídas sem que o sistema seja testado como um todo. Assim, devido a esta construção isolada das unidades, a integração entre as mesmas normalmente apresenta uma grande quantidade de erros.

Integração Contínua - IC [1, 3, 4, 12] é a prática de desenvolvimento de software onde a equipe de desenvolvimento integra seu trabalho constantemente. A prática de Integração Contínua gera freqüentes versões do produto desenvolvido (*builds*), que são testadas, verificadas e documentadas, a cada ciclo de integração [2].

O termo *build* é utilizado “*tanto para designar o processo de conversão de arquivos de código fonte em artefatos de software que podem ser executados isoladamente na forma de um produto de software acabado como para designar o resultado deste processo*” [18]. Ou seja, para designar a versão do *software* produzido neste processo.

O principal objetivo da integração contínua é encontrar rapidamente falhas produzidas pela integração dos componentes desenvolvidos individualmente. Como podem ser produzidas várias *builds* no mesmo dia, as falhas de integração podem ser identificadas e corrigidas mais facilmente, vez que relativamente poucas alterações no código devem ter sido realizadas desde a última *build* produzida com sucesso [3].

A prática de integração contínua pressupõe a utilização coordenada de várias classes de ferramentas de automação [1]. Conforme a Figura 3.1, a coordenação destas ferramentas cabe ao servidor de integração contínua. Entre as principais ferramentas utilizadas atualmente com este propósito, estão as ferramentas *CruiseControl* [6], *Anhil* [14], *Apache Continuum* [17].

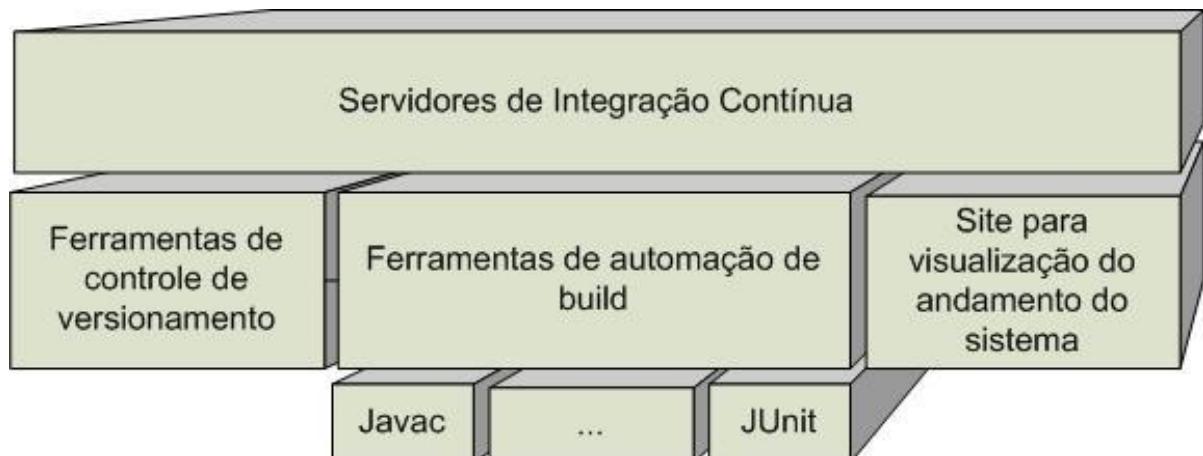


Figura 3.1: Hierarquia de ferramentas na prática de Integração Contínua

O servidor de integração contínua aciona ferramentas de controle de versão e ferramentas de automação de *build*. Além disto, publica num site a situação atual e o histórico do desenvolvimento do sistema, provendo visibilidade e permitindo o acompanhamento do projeto por todos os envolvidos.

As ferramentas de automação de construção de *software*, automação de *build*, como *Ant* e *Maven*, executam seqüências de tarefas como a compilação, e a execução de testes. Para cada tarefa a ser executada, outras ferramentas específicas devem ser

acionadas. Por exemplo, o *Javac* é acionado para a compilação de programas Java, enquanto o *JUnit* é acionado para realização de testes unitários em Java.

As ferramentas de controle de versão, como o *CVS* e o *Subversion*, disponibilizam um repositório centralizado para armazenamento e controle de arquivos. Versionamento é a ação de guardar diversas versões dos artefatos armazenados e disponibilizar quaisquer das versões quando for solicitado.

A ação de duas pessoas trabalhando em um mesmo artefato ao mesmo tempo é comum em projetos que utilizam ferramentas de versionamento. Duas formas de tratar esta situação podem ser utilizadas. A primeira forma consiste no desenvolvedor solicitar autorização para alteração do artefato (*check-out*) a ser desenvolvido, de forma que nenhum outro desenvolvedor tenha acesso ao artefato até que as alterações necessárias sejam realizadas. A segunda forma, mais utilizada em processos que fazem uso da Integração Contínua, consiste no desenvolvedor realizar o *check-out* de todo o projeto em sua máquina sem se preocupar se os arquivos estão ou não sendo utilizados por outras pessoas.

O controle dos artefatos no repositório fica a cargo da ferramenta de versionamento, de forma que esta deve verificar conflitos de artefatos alterados por dois desenvolvedores e verificar junto a cada desenvolvedor se deve ser realizada a junção das alterações realizadas nos arquivos.

Existem dois tipos básicos de Integração Contínua, a saber: Integração Contínua Síncrona ou Integração Contínua Assíncrona.

A Integração Contínua Síncrona consiste em cada desenvolvedor, um de cada vez, integrar seu conjunto de artefatos de forma que novas integrações somente possam ser realizadas após a confirmação de finalização da anterior.

Neste tipo de integração, todas as verificações e testes devem ser realizados antes da confirmação do envio dos artefatos para o repositório. Os artefatos somente devem ser submetidos ao repositório se todos os testes forem bem sucedidos.

Este tipo de integração garante que toda *build* armazenada no repositório está íntegra e funcionando de acordo com os testes disponibilizados.

A grande desvantagem desta abordagem é que alterações nos artefatos podem demorar horas ou até dias para serem efetivadas numa nova versão da *build* se estas alterações não forem bem sucedidas nos testes executados.

A Integração Contínua Assíncrona consiste em integrar quaisquer alterações ao repositório sem se preocupar se estas alterações irão passar nos testes disponibilizados. Todo o controle de consistência da *build* ficará a cargo da ferramenta de Integração Contínua utilizada, que deve realizar os testes e afirmar se a *build* está íntegra e disponibilizá-la ou não no ambiente de produção.

A Figura 3.2 retrata a prática de Integração Contínua Assíncrona ilustrando o relacionamento entre as classes de ferramentas envolvidas.

Neste capítulo, o objetivo não é detalhar exaustivamente as ferramentas apresentadas, mas promover uma visão de suas funcionalidades e como as mesmas se relacionam.

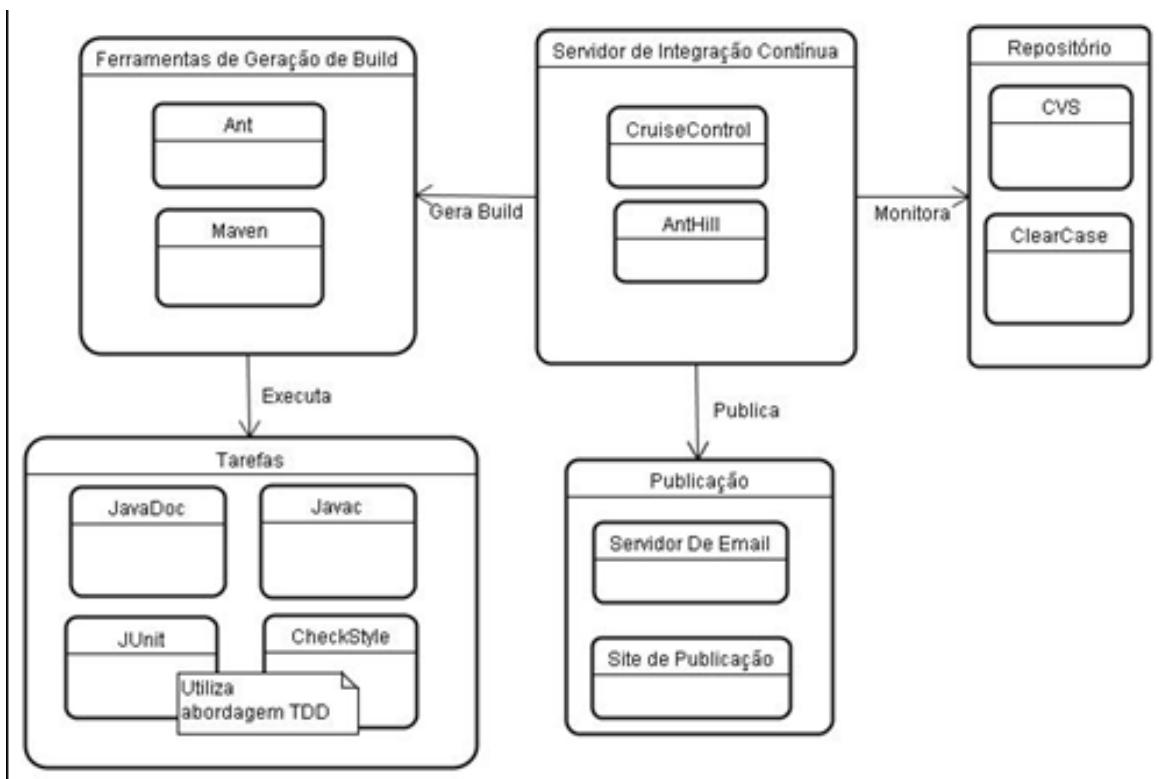


Figura 3.2: Relacionamento entre ferramentas utilizadas Integração Contínua Assíncrona

O servidor de integração monitora o repositório gerenciado pelas ferramentas de controle de versão. A cada alteração, aciona as ferramentas de automação de *build* e também publica os resultados obtidos utilizando diversos meios, como *web site* e correio eletrônico.

Entre outros benefícios obtidos com a prática de Integração Contínua [2, 3], estão:

- Automação na geração de novas versões do sistema (*builds*);
- Automação na distribuição das novas versões;
- Automação na execução dos testes unitários;
- Obtenção de métricas do código fonte gerado;
- Avaliação da qualidade do código gerado;
- Geração automática da documentação.

O restante do capítulo está organizado como segue. A segunda seção apresenta um conjunto de práticas essenciais para o sucesso do processo de integração contínua. A execução freqüente dos testes está no cerne da integração contínua. Assim, é comum a prática da integração contínua ser associada ao desenvolvimento orientado a testes. A terceira seção aborda o Desenvolvimento Orientado a Testes descrevendo sucintamente a ferramenta JUnit que é utilizada para criação de testes unitários em Java. A seção 4

apresenta ferramentas de automação de *build*, abordando aspectos da utilização da Ferramenta *Ant* e da Ferramenta *Maven*. A quinta seção descreve o funcionamento da ferramenta de integração contínua *CruiseControl*. Por fim, na seção 7, é apresentada a conclusão do capítulo.

3.2. Práticas Necessárias à Integração Contínua

Segundo Fowler [1], as dez principais práticas necessárias para o sucesso da aplicação da Integração Contínua são:

- Manter um único repositório de artefatos;
- Automatizar a geração de *builds*;
- Testar *builds* automaticamente: O processo de geração de *builds* deve possuir testes automáticos;
- Atualizar artefatos diariamente: todos os envolvidos devem incluir seus artefatos no repositório diariamente;
- Gerar nova *build* a cada disponibilização de artefatos;
- Agilizar a disponibilização de novas builds;
- Executar testes em ambiente similar ao ambiente de produção;
- Permitir fácil acesso à última versão dos arquivos da aplicação: qualquer envolvido autorizado deve ter fácil acesso aos arquivos executáveis da aplicação;
- Prover visibilidade da situação atual do projeto: deve existir um mecanismo sinalização visual que permita a todos ter ciência a cerca da situação atual do projeto;
- Gerar automaticamente versões para distribuição.

Estas práticas serão detalhadas nas seções seguintes.

3.2.1. Manter um Único Repositório de Artefatos

Projetos de *software* possuem um conjunto de artefatos que necessitam ser organizados de forma a produzir uma *build*. A complexidade e o esforço necessários aumenta dependendo do número de pessoas envolvidas.

A prática de possuir um único repositório onde estão todos os artefatos do projeto parte do princípio de que existe uma ferramenta de gerenciamento destes artefatos, onde todos os envolvidos no projeto possuem conhecimento e acesso à ferramenta.

A ferramenta deve permitir versionamento dos artefatos, ou seja, diversas versões de cada artefato devem ser armazenadas. A ferramenta deve resolver conflitos no versionamento dos artefatos. O repositório da ferramenta deve ser a única fonte de artefatos para a integração, evitando dualidades e reduzindo a complexidade do gerenciamento.

Todos os artefatos necessários para a criação da *build* devem estar no repositório e não apenas o código fonte do aplicativo. Artefatos com imagens ou manuais de utilização devem também estar no repositório.

3.2.2. Automatizar a Geração de *Builds*

O processo de gerar um sistema executável a partir do código fonte geralmente não é uma tarefa simples e envolve mover arquivos, compilar o código fonte ou carregar *schemas* no banco de dados, dentre outras tarefas. A grande maioria das tarefas descritas pode ser realizada de forma automatizada e este é objetivo das ferramentas de automação de *build*.

A prática “Automatizar a Geração de *Builds*” envolve automatizar todo o processo de geração de versões executáveis do sistema, codificando um ou mais *scripts*. *Scripts* são aplicativos que executam seqüências de tarefas de forma automatizada. Após a produção dos *scripts*, o processo de construção de novas versões do aplicativo deve ser executado de forma automática.

A execução dos *scripts* pode ser acionada através da linha de comando (*Commanded automation*), pode ser acionada em horários ou intervalos de tempo pré-determinados por um servidor de integração contínua (*Scheduled automation*) ou pode ser acionada por eventos como a disponibilização no repositório de uma nova versão do código fonte a ser construída (*Triggered automation*).

Na seção 4, serão descritas algumas ferramentas de automação de *build*.

3.2.3. Testar *Builds* Automaticamente

As formas tradicionais de geração de novas versões do aplicativo envolvem compilar o código fonte, adicionar os componentes necessários e gerar o executável do programa.

Esta forma de geração de versões não garante que o aplicativo gerado está correto em relação às suas funcionalidades. Portanto, é fundamental testar o sistema.

O processo de Integração Contínua prevê como prática chave a realização de testes automatizados antes de quaisquer ciclos de geração de novas *builds*. Abordagens como o *Test Driven Development – TDD* [15, 16] preconizam a realização de testes automatizados do código fonte de todo o sistema.

Ferramentas livres que utilizam a abordagem TDD permitem realizar este tipo de automação. Dentre elas, podemos citar:

- *JUnit*;
- *Mock Objects*;
- *FIT*;
- *Selenium*.

3.2.4. Atualizar Artefatos Diariamente

A integração permite que desenvolvedores se comuniquem entre si, informando as mudanças realizadas. O processo de submeter as mudanças realizadas é vital para que conflitos e erros de implementação sejam encontrados precocemente. A prática de

submeter o código à integração freqüentemente habitua os desenvolvedores a dividir seu trabalho em pequenos módulos. Esta prática também tem como o objetivo dar visibilidade do progresso do projeto.

3.2.5. Gerar Nova *Build* a Cada Disponibilização de Artefatos

O principal objetivo desta atividade é garantir que a cada disponibilização de artefatos uma *build* seja gerada e suas verificações de integridade realizadas. A geração da *build* pode ser realizada de forma manual ou automática.

A abordagem manual consiste em o desenvolvedor, a cada disponibilização de artefatos, acessar a máquina de integração, iniciar o processo de geração de uma *build*, acompanhar o progresso das verificações e, se estas forem bem sucedidas, confirmar a geração da *build*.

A segunda abordagem consiste em utilizar um servidor de integração contínua que fica monitorando o repositório. A cada mudança, a própria ferramenta dispara o processo de geração de uma nova versão.

3.2.6. Agilizar a Disponibilização de Novas *Builds*

Um dos principais objetivos da Integração Contínua é dar uma resposta rápida quanto à existência ou não de falhas na integração. A geração rápida de *builds* é primordial para que o processo de desenvolvimento não seja frustrante com a espera de horas para receber resultados das verificações a cada disponibilização de artefatos.

Existem duas opções que devem ser levadas em consideração no que diz respeito a verificações envolvendo banco de dados. Na primeira opção, mais comumente utilizada, não são imediatamente realizadas verificações no banco de dados, por serem longas e demorarem horas para terminar. As únicas verificações efetuadas são no código fonte. Portanto, podem ser realizadas em poucos minutos. Neste caso, os erros de integração com o banco de dados serão descobertos tarde, por não ter sido realizada nenhuma verificação.

A segunda opção inclui todas as verificações possíveis, o que torna o trabalho tedioso para o desenvolvedor, obrigando-o a esperar durante horas a cada disponibilização de artefato. A maneira ideal é projetar a geração de versões em estágios diferentes, onde o primeiro estágio é executado a cada disponibilização de artefatos e o segundo uma vez ao dia.

3.2.7. Executar Testes em Ambiente Similar ao Ambiente de Produção

O objetivo dos testes é identificar riscos e falhas que podem acontecer no ambiente de produção do aplicativo. Desta forma, o ambiente de testes deve ser uma cópia similar do ambiente de produção, de forma que as falhas encontradas nos testes sejam as mesmas encontradas caso o aplicativo estivesse na produção, evitando que alguma falha ou risco que ocorreria na produção não seja encontrada durante os testes.

Um das formas de simular o ambiente de produção é utilizar produtos de virtualização, que permitem simular máquinas ou até redes de computadores.

3.2.8. Permitir Fácil Acesso à Última Versão dos Arquivos da Aplicação

O acesso à versão atual do projeto, ou ao endereço onde a aplicação está hospedada, deve ser fácil e de conhecimento geral. Dessa forma, todos os envolvidos podem acessar e, consequentemente, podem testar e verificar se o sistema, em sua atual versão, atende aos requisitos especificados.

3.2.9. Prover Visibilidade da Situação Atual do Projeto

Esta prática consiste em fornecer a todos a visibilidade do real estágio do projeto. Devem ser desenvolvidos mecanismos simples e eficazes de sinalização que representem o estado atual da *build*. Como exemplo, podem ser utilizados *leds*, lâmpadas, painéis, calendários, objetos (*tokens*), *sites* etc, de modo a tornar explícito tanto o estado atual como o histórico de andamento do projeto.

Em [19], é ilustrado um exemplo de *token* (galinha de borracha) utilizado para sinalizar quem está, no presente instante, construindo a nova versão do sistema. Os demais membros da equipe deverão aguardar o término da geração da nova versão para que possam, da mesma forma, requisitar o *token* e passar a incorporar as suas alterações na versão corrente do sistema.



Figura 3.3: Exemplo de *token* utilizado para indicar quem está alterando a versão corrente do sistema [19]

3.2.10. Gerar Automaticamente Versões para Distribuição

Normalmente os testes da integração contínua são realizados em diversos ambientes, o que torna muito comum a movimentação de arquivos tanto entre os ambientes de teste como entre os ambientes de testes e o ambiente de produção. Para facilitar tal movimentação, é conveniente automatizar a geração de versões para distribuição. Esta prática, além de agilizar o processo, reduz a possibilidade de ocorrência de erros.

Além de produzir a instalação de versões para distribuição de forma automática, é conveniente também produzir *scripts* para desinstalação automática, permitindo retornar o sistema à versão estável anterior caso ocorram falhas que justifiquem este retorno.

Na seção seguinte, será abordado o Desenvolvimento Orientado a Testes (TDD) e detalhadas ferramentas *JUnit* e *Mock Objects*. A Integração Contínua, bem como o TDD, supõe o uso intensivo de testes. É comum se utilizar em conjunto as duas abordagens.

3.3. Desenvolvimento Orientado a Testes (TDD)

Segundo [24], *TDD* “é uma técnica de programação centrada em escrever os testes para ajudar a dirigir o projeto de uma classe”. Escrever um teste, no presente contexto, significa desenvolver um código capaz de verificar o comportamento de outro código de maneira automática.

O ciclo de vida do TDD está representado na Figura 3.4. O ciclo apresentado indica que os testes e o código a ser testado são escritos de maneira incremental. A cada novo teste escrito, o código a ser testado é construído em seguida. Assim, quando não houver mais testes a serem escritos, o código estará pronto. Segundo [16], nesta abordagem, os testes determinam o código a ser escrito [16].

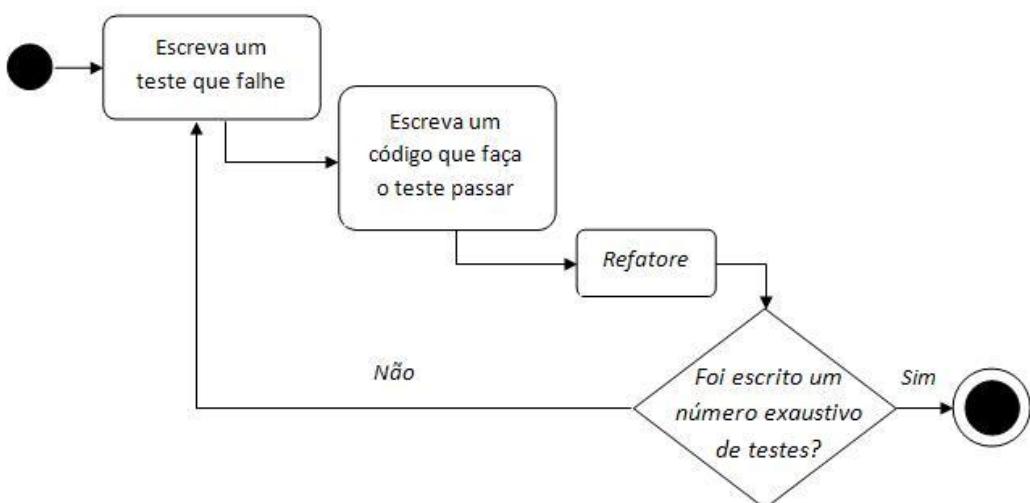


Figura 3.4: Ciclo de Vida da Abordagem TDD

Primeiramente, é escrito um teste que falhe. De fato, um teste não poderia passar antes que o código a ser testado fosse escrito. Em seguida, cria-se uma porção mínima de código que faça o referido teste passar. O código então é evoluído a partir de uma prática denominada refatoramento, a qual será detalhada na seção 3.3.1. Refatorar significa aperfeiçoar o código escrito.

Durante a etapa de refatoramento, é verificado constantemente se o código continua passando no teste. Caso seja introduzida alguma falha, ajustes devem ser providenciados até que o teste volte a passar. Concluída a etapa de refatoramento, caso ainda exista algum teste a ser escrito, o ciclo volta à primeira etapa. Caso contrário, o código estará concluído.

Podem ser destacados alguns benefícios do *TDD*:

- código fonte legível, simples e de fácil manutenção;

- ampla cobertura dos testes;
- quantidade exaustiva de testes;
- falhas introduzidas durante a evolução do código são mais facilmente identificadas;
- direciona o desenvolvimento em pequenos passos, provendo rápido *feedback* em relação à qualidade do código gerado;
- garante contínua testabilidade;
- mantém o código e o teste sincronizados;
- todos os testes são automatizados.

Apesar de aumentar a quantidade de código do sistema devido à implementação dos testes unitários, o TDD reduz código desnecessário ao projeto, tornando-o modularizado, flexível e extensível.

3.3.1. Refatoramento

Refatoramento [20] é a prática de se aperfeiçoar o código fonte sem alterar o seu comportamento externo. Parte integral do processo de desenvolvimento de metodologias ágeis, como *Extreme Programming* [21], tem como principal objetivo dar clareza e consistência ao código gerado. Refatoramento não tem como objetivo corrigir falhas ou acrescentar novas funcionalidades, mas apenas melhorar a qualidade da funcionalidade previamente desenvolvida.

Seguem alguns objetivos do refatoramento:

- Melhorar o projeto de *software*;
- Tornar o código do *software* mais legível e comprehensível;
- Eliminar código duplicado;
- Reduzir o tamanho do código;
- Simplificar o código produzido;
- Remover redundâncias;
- Melhorar o reuso do código;
- Otimizar o desempenho do *software*.

O processo de refatoramento é levado a cabo através de um conjunto de pequenas alterações sobre o código do sistema. Ao término de cada uma dessas alterações, é produzida uma nova versão. Como as mudanças no código são de pequena monta, a probabilidade de introdução de erros diminui.

A técnica TDD é facilitada pela utilização de ferramentas que automatizam os testes. A seguir, serão descritas a ferramentas *JUnit* e *Mock Objects*.

3.3.2. JUnit

Para verificar o funcionamento de um programa, é comum a utilização de ferramentas de depuração de código ou a inclusão da impressão de mensagens que indiquem o andamento da execução e o valor assinalado a certas variáveis. Nos dois casos o programador deve verificar visualmente se os valores atribuídos às variáveis estão corretos. Por exemplo, o programador poderia verificar se o valor de uma variável é 4, como deveria ser.

Ocorre que a verificação visual só é viável para pequenos testes realizados imediatamente depois da codificação. Se o mesmo teste for repetido uma semana depois da sua codificação não será fácil lembrar o valor que a variável deveria assumir.

Outro problema desta abordagem é que os testes não são decomponíveis. As mensagens de cada teste ficam distribuídas no código e se misturam com as mensagens de outros testes.

Um terceiro problema é que as mensagens relacionadas com os testes se misturam com as próprias saídas do programa que está sendo testado.

Finalmente a verificação visual não é apropriada quando se desejar automatizar testes, uma vez que necessita da interferência do programador para julgar o acerto ou o erro dos testes.

Todos estes problemas assumem proporções maiores se centenas de testes estiverem sendo executados.

No contexto da automação de testes, a solução apropriada é a adoção de um *framework* que permita a construção incremental de testes. Os testes construídos devem ser componíveis, repetíveis e auto-verificáveis. Devem ser “repetíveis”, pois a finalidade é realizar os testes sempre que qualquer mudança ocorrer no programa. Devem ser “auto-verificáveis”, pois o valor resultante que deve ser comparado na realização do teste deve estar indicado explicitamente na chamada do teste. *JUnit* é um *framework open-source* que preenche os requisitos acima citados e tem como principal objetivo a criação de testes unitários automatizados na linguagem *Java*.

As principais características do *JUnit* incluem:

- Aserções para testar os resultados esperados.
- Compartilhamento do processo de inicialização e finalização dos dados comuns a vários testes (*test fixtures*).
- Execução dos testes através de *tests runners*.

Anotações introduzidas no código fonte indicam os métodos que serão utilizados no *framework*. A anotação `@Test` indica os métodos de teste, a anotação `@Before` indica métodos que serão executados nos processos de inicialização e a anotação `@After` indica métodos que serão utilizados nos processos de finalização. Os métodos anotados com `@Before` deverão ser executados uma única vez no inicio dos testes enquanto os anotados com `@After` executarão uma única vez ao final dos testes.

No exemplo do Quadro 3.1, é apresentado o código da classe *Operacao* e o método *soma()*, que retorna a soma de dois números. A classe de teste *TestOperacao* implementa o método *testSoma()* que testa o método *soma()* da classe *Operacao*.

```
//classe operação
public class Operacao{
    ...
    public int soma(int a, int b){
        return a+b;
    }
    ...
}

//classe de teste
public class TestOperacao{
    ...
    @Test public void testSoma(){
        assertEquals("Soma",4,soma(2,2));
    }
    ...
}
```

Quadro 3.1: Teste utilizando JUnit (versão 4.0)

A verificação realizada no exemplo acima utiliza o método *assertEquals()*, que verifica se o retorno da chamada *soma(2,2)* é igual a quatro. Métodos como o *assertEquals()* são disponibilizados pela classe *junit.framework.Assert* e têm como principal objetivo verificar se um método se comporta conforme o esperado. Entre os métodos de verificação disponibilizados pela classe *Assert*, temos:

- *assertEquals()*: verifica se dois inteiros, caracteres, booleanos ou reais são iguais;
- *assertNotNull()*: verifica se o valor de uma variável é diferente de *null*;
- *assertNull()*: verifica se o valor de uma variável é *null*;
- *assertSame()*: verifica se dois objetos referenciam o mesmo objeto;
- *assertTrue()*: verifica se uma condição é verdadeira;
- *fail()*: provoca falha no teste.

Para o caso em que se presume que o teste deverá lançar uma exceção, a versão 4.0 do *framework* disponibiliza um parâmetro chamado *expected* na anotação *@Test* (Quadro 3.2). As versões anteriores à 4.0 utilizam o método *fail()* para verificar falhas.

```
//Verifica falhas na versão 4.0
@Test(expected=ArithmaticException.class)

    public void testDivisaoPorZero () {
        int n = 2 / 0;
    }.....
```

Quadro 3.2: Verificação de falhas utilizando JUnit

Testes podem ser agrupados em suítes utilizando o *JUnit* através da classe *TestSuite*. Uma suíte é um conjunto de testes automáticos agrupados para serem executados conjuntamente. O Quadro 3.3 apresenta um exemplo de utilização de uma suíte para testar as classes *Operação* e *Algoritmo*. É importante observar que a suíte é instanciada e os casos de teste são acrescentados utilizando o método *addTest()*.

```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new OperacaoTest());
    suite.addTest(new AlgoritmoTest());
    return suite;
}
```

Quadro 3.3: Criando suíte de testes utilizando JUnit

A grande maioria dos ambientes *J2EE* disponíveis ainda não está pronta para executar a versão 4.0 do *framework*. Um adaptador chamado *JUnit4TestAdapter* permite que testes criados na versão 4.0 possam ser executados nas versões anteriores do *JUnit*.

Ao construir os testes de forma incremental é comum ser necessário inicializar objetos que ainda não estão totalmente definidos ou que estão parcialmente codificados. Os mock Objects, descritos a seguir, são utilizados nestes casos.

3.3.3. Mock Objects

Mock Objects são componentes simulados que imitam o comportamento de componentes reais. Este tipo de teste é utilizado quando o objeto real não está disponível ou este é muito complexo para ser implementado no teste. *Mock Objects* possuem a mesma interface do objeto real a ser simulado. O principal objetivo de se utilizar *Mock Objects* é isolar a classe a ser testada de modo que nenhuma outra classe seja envolvida nos testes.

No exemplo abaixo, são apresentadas duas classes: *Conta* e *LinhaItem*. A classe *Conta* é formada por um conjunto de elementos da classe *LinhaItem*, de forma que a realização de um teste com a classe *Conta* envolve instanciar elementos da classe *LinhaItem*.

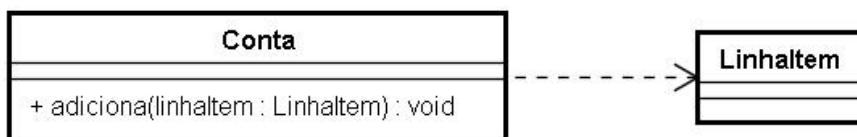


Figura 3.5: Dependência entre as classes Conta e LinhaItem.

Não é apropriado testar a classe *Conta* sem que estejam criado objetos da classe *LinhaItem* que a compõe. Todavia se a classe *LinhaItem* não estiver codificada, é possível utilizar *Mock Objects* para simula-la. No exemplo do Quadro 3.4, será utilizado o *framework* *EasyMock* para criar o objeto *linhaMock*.

```

LinhaItem linhaMock =
    createStrictMock(LinhaItem.class);

expect(linhaMock.getValor()).andReturn("10");
Conta conta=new Conta();
Conta.adiciona(linhaMock);
assertEquals(conta.calculaSomatorio(),10);

```

Quadro 3.4: Utilizando *Mock Objects* para eliminar dependências.

O objeto *linhaMock* é criado através do método *createStrickMock()*, que recebe como parâmetro a classe do objeto a ser criado. Os valores retornados pelos métodos do objeto simulado são definidos utilizando-se o método *expect()*, o qual recebe como parâmetro a chamada ao método *getValor()* do objeto *linhaMock* e executa a chamada ao método *andReturn*, passando como parâmetro o valor a ser retornado. De acordo com o exemplo, quando o método *getValor()* da classe *LinhaItem()* for chamado pela classe *Conta*, o valor a ser retornado será 10.

Nesta seção foram apresentados conceitos referentes a TDD, bem como as ferramentas JUnit e MockObjects. Na seção seguinte, será indicado como estas ferramentas terão sua execução acionada a partir das ferramentas de automação de build.

3.4. Ferramentas de Automação de Build

São ferramentas cujo principal objetivo é automatizar o processo de geração de *builds*, compilando o código fonte, testando-o, criando a documentação e disponibilizando a aplicação. Entre as ferramentas mais utilizadas com este propósito temos:

- *Ant*;
- *Apache Maven*;
- *NAnt*.

3.4.1. Apache Ant

O *Apache Ant* é uma ferramenta escrita na linguagem *Java* para automatizar a construção de *builds*. A ferramenta é flexível e facilmente extensível, permitindo a inserção de tarefas customizadas.

```

<?xml version="1.0"?>
<project name="integra" default="compila" basedir="c:\>
    <property name="src" value="c:\src"/>
    <target name="init">
        <mkdir dir="c:\build"/>
    </target>
    <target name="compila" depends="init">
        <!-- -Compila o código -->
        <javac src="${src}" destdir="c:\build"/>
    </target>
</project>

```

Quadro 3.5: Arquivo build.xml

O funcionamento da ferramenta consiste basicamente em definir um arquivo no formato XML com a relação de tarefas a serem executadas. Cada tarefa é chamada de *target*. Cada *target* possui um conjunto de subtarefas chamadas de *task*. Normalmente, o arquivo de tarefas é denominado build.xml. No Quadro 3.5, é ilustrado um exemplo de configuração deste arquivo.

O arquivo build.xml é formado normalmente pelas *tags*:

- *project*;
- *property* (opcional);
- *target*;

O elemento principal do arquivo build.xml é a *tag* denominada *project*. Esta *tag* possui três atributos fundamentais:

- *name*: Nome do projeto;
- *default*: Define a target padrão a ser executada quando nenhuma target for especificada;
- *basedir*: Diretório principal que referencia todos os diretórios relativos utilizados.

A *tag* *property* permite a criação de propriedades que podem ser utilizadas em todo o escopo do projeto. Cada propriedade possui os atributos *name* e *value*.

O elemento *target* é utilizado para definir um conjunto de ações a serem executadas. Cada ação está especificada através de um elemento *task*.

O elemento *target* também inclui alguns atributos. Dentre eles, estão:

- *depends*: define uma seqüência de outros *targets* que precisam ser executados como pré-requisito. Assim, o uso do atributo *depends* define uma árvore que especifica a ordem de execução dos *targets*. No exemplo do Quadro 3.6, a tarefa compile somente será executada após o término das tarefas *init* e *document*;
- *if*: Define condição inicial para que a tarefa possa ser executada;
- *unless*: Define uma condição que inibe a execução do *target* quando satisfeita;
- *description*: Define a descrição da tarefa.

```
<target name="compile" depends="init, document">
    <!-- Compile the java code -->
    <javac srcdir="${src}" destdir="${build}" />
</target>
```

Quadro 3.6: Uso do elemento target para compilação

O *target* inicial pode ser executado diretamente, via linha de comando, ou ter sua execução disparada a partir de uma ferramenta de integração contínua, como o *CruiseControl*.

Cada elemento *target* é composto por uma seqüência de tarefas indicadas por elementos *task*. Segundo a *Apache*, “task é um pedaço de código que pode ser executado”. O elemento *task* possui a seguinte estrutura:

```
<nomedatask atributo1=valor atributo2=valor ...../>
```

Quadro 3.7: Estrutura do elemento task

No Quadro 3.7, o *nomedatask* indica o nome da tarefa a ser executada. Os atributos da *task* definem suas configurações. O Quadro 3.6 exemplifica uma *task* destinada a compilar um código fonte em *Java*. Neste caso, a *task* foi definida como `<javac srcdir="${src}" destdir="${build}" />`, onde *javac* é o nome da *task* e os atributos *srcdir* e *destdir* são o diretório fonte e o diretório destino, respectivamente.

A ferramenta *Ant* possui um conjunto de *tasks* pré-definidas para representar tarefas mais freqüentes. Entre estas *tasks* estão:

- *ant*: executa um subprojeto do *Ant*;
- *checksum*: gera um *checksum* para os arquivos definidos;
- *concat*: concatena um ou mais arquivos;
- *copy*: copia um arquivo para um novo arquivo ou diretório;
- *cvs*: retorna pacote ou modulo do *CVS*;
- *delete*: exclui arquivo ou diretório;
- *ear*: gera um arquivo *ear* da aplicação;
- *get*: retorna um arquivo de uma determinada *url*;
- *jar*: gera um arquivo *jar* da aplicação;
- *java*: executa uma classe *java*;
- *javadoc/javadoc2*: gera documentação de código utilizando o *javadoc*;
- *mail*: envia e-mail com arquivos escolhidos;
- *mkdir*: cria diretório;
- *sql*: Executa um conjunto de instruções *SQL* num banco de dados a partir de um arquivo texto.

Além das *tasks* pré-definidas, o *Ant* permite a criação de tarefas customizadas através do elemento *taskdef* (Quadro 3.8). Este elemento possui os atributos *classname* e *classpath*. O atributo *classname* define a classe a ser executada enquanto o atributo *classpath* define a localização da os arquivos utilizados pela classe.

```
<target name="usa" description="Usando a Task" depends="jar">
    <taskdef name="aloMundo" classname="AloMundo"
             classpath="${ant.project.name}.jar" />
    <alomundo/>
</target>
```

Quadro 3.8: Criando task customizada.

3.4.1.1. Automatizando Tarefas com o *Ant*

Esta seção ilustra a utilização da ferramenta *Ant* para exemplificar a automação das seguintes tarefas:

- Compilação de Classes;
- Testes;
- Verificações e Revisões de Código.

3.4.1.1.1. Automatizando a Compilação de Classes

A compilação de classes utilizando o *Ant* usa uma *task* chamada *javac*. Os principais atributos da *task* são:

- *srcdir*: diretório onde fica localizado o código fonte a ser compilado;
- *destdir*: atributo que define o diretório onde as classes compiladas devem ser armazenadas;
- *includes*: lista separada por vírgulas que inclui os artefatos a serem compilados. Se este atributo não for informado, todas as classes do diretório são compiladas.

O Quadro 3.9 apresenta um exemplo de utilização da *task javac* em um arquivo de *build* do *Ant*.

```
<javac srcdir="${src}" destdir="${build}" classpath="meujar.jar" />
```

Quadro 3.9: Utilizando a *task javac* no *Ant*.

3.4.1.1.2. Automatizando Testes

O *Ant* possibilita a realização automática de testes a partir de uma integração com alguma ferramenta de automação de testes, como o *JUnit*. Esta subseção apresenta uma visão geral do processo de configuração do *Ant* para integrá-lo ao *JUnit*. Neste caso, é utilizada uma *task* chamada *junit*. Os principais atributos da *task junit* são:

- *printsummary*: define se serão gerados estatísticas *online* para cada caso de teste;
- *fork*: define se os testes serão disparados em uma *JVM* diferente.

A *task* utiliza algumas *tags* de configuração, como:

- *classpath*: define o *classpath* para execução dos testes;
- *formatter*: define o formato do relatório a ser gerado;
- *test*: especifica um teste específico a ser executado;
- *batchtest*: define um conjunto de casos de teste a serem executados.

O exemplo do Quadro 3.10 ilustra a configuração do *Ant* para integração com o *JUnit*.

```
<junit printsummary="yes" haltonfailure="yes">
  <classpath>
    <pathelement location="${build.tests}" />
    <pathelement path="${java.class.path}" />
  </classpath>
  <formatter type="html" />
  <test name="TestCase" haltonfailure="no" outfile="result">
    <formatter type="xml" />
  </test>
  <batchtest fork="yes" todir="${reports.tests}">
    <fileset dir="${src.tests}">
      <include name="**/*Test*.java" />
      <exclude name="**/AllTests.java" />
    </fileset>
  </batchtest>
</junit>
```

Quadro 3.10: Utilizando a task *junit*

Os relatórios especificados na tag *formatter* podem ser do tipo XML ou HTML. A tag *batchtest* pode incluir testes baseados em uma máscara utilizada para filtrar os arquivos fonte. No Quadro 3.10, a tag *include* e a tag *exclude* apresentam respectivamente o uso das máscaras “**/*Test*.java” e “**/AllTests.java”.

3.4.1.1.3. Automatizando Verificações e Revisões de Código

Revisões e verificações de código podem ser realizadas utilizando ferramentas integradas ao *Ant*, como a ferramenta *Checkstyle* [11].

A ferramenta *CheckStyle* permite a verificação do código desenvolvido, informando se o código está ou não aderente às convenções definidas. A *task* da ferramenta tem como principais atributos:

- *file*: determina o arquivo que deve ser verificado;
- *config*: define o arquivo responsável pelas convenções de código;
- *maxErros*: O número máximo de erros tolerados por uma *build*.

A ferramenta não possui *task* pré-definida no *Ant*. Desta forma, é necessário a utilização do elemento *taskdef* para a definição da *task* (Quadro 3.11)

```
<taskdef resource="checkstyletask.properties"
         classpath="/path/to/checkstyle-all-4.3.jar"
```

Quadro 3.11: Utilizando *taskdef* para definição da *task* do *CheckStyle*.

Todas as verificações realizadas são reportadas no formato definido por uma *task* chamada de *formatter*, que possui um atributo chamado *tofile*, responsável por definir o arquivo com os resultados das verificações (Quadro 3.12).

```
<checkstyle config="docs/sun_checks.xml">
  <fileset dir="src/checkstyle" includes="**/*.java"/>
  <formatter type="plain"/>
  <formatter type="xml" toFile="build/checkstyle_erros.xml"/>
</checkstyle>
```

Quadro 3.12: Utilizando a task checkstyle.

Outra ferramenta de automação de *build* é o *Maven*. Sua utilização é mais simples quando comparada ao uso do *Ant*. Todavia, por ser mais antigo, o *Ant* tem uma base instalada bem maior. Na seção seguinte, veremos algumas de suas características.

3.4.2. Maven

A ferramenta *Maven* oferece uma estrutura padronizada para os projetos, a qual facilita tanto a automação de tarefas, como compilação, testes e análise de dependência, como facilita também o acompanhamento do projeto, permitindo visualização via *Web* do histórico e da situação atual do projeto.

Com relação ao *Ant*, uma diferença é que a configuração do *Maven* dispensa a criação de boa parte das *tasks*, uma vez que estas estão pré-definidas na estrutura padronizada da ferramenta.

Graças a esta estrutura padronizada, o esforço para configuração da *build* é menor. Como exemplo, as tarefas nativas da ferramenta não precisam ser configuradas previamente.

O *Maven* trabalha com uma estrutura de diretório padrão onde os artefatos devem ser colocados para que se possa executar o conjunto de tarefas automatizadas, conforme ilustrado no Quadro 3.13.

```
pom.xml
/src
  /main
    /java
    /resources
    /webapp
  /test
    /java
    /resources
/site
```

Quadro 3.13. Estrutura de diretórios padrão do Maven

Após todos os arquivos estarem no seu diretório definido, a automação de tarefas é simples, utilizando um dos comandos abaixo:

- *mvn test*: executa testes unitários;
- *mvn package*: cria o arquivo *jar* do projeto;
- *mvn install*: adiciona arquivo *jar* ao repositório e usa as dependências necessárias;
- *mvn site*: gera o *website* do projeto;
- *mvn eclipse*: gera o projeto do *Eclipse*.

Entre as principais funcionalidades providas pela ferramenta, podemos citar:

- Geração de *builds*;
- Documentação;
- Relatórios;
- Análise de dependências;
- Releases;
- Distribuição.

A ferramenta é formada por um modelo que, combinado com algumas entradas dos usuários, produz um portal de informações (Figura 3.6).

The screenshot shows a web-based interface for the Apache Maven Project. At the top, there's a navigation bar with links like 'Overview', 'Build', 'Report', 'FAQ', and 'Examples'. Below the navigation, a sidebar on the left contains sections for 'Overview', 'Examples', 'Project Documentation', and 'built by maven'. The main content area is titled 'File Activity Report' and displays a table of file changes between July 3, 2006, and August 3, 2006. The table has two columns: 'Filename' and 'Number of Times Changed'. Some examples of files listed include 'maven/plugins/trunk/maven-checkstyle-plugin/src/site/apt/examples/multi-module-config.apt', 'maven/plugins/trunk/maven-checkstyle-plugin/src/site/apt/index.apt', and 'maven/plugins/trunk/maven-clean-plugin/pom.xml'. The table shows that many files were changed multiple times, with some files having up to 5 changes.

Filename	Number of Times Changed
/maven/plugins/trunk/maven-checkstyle-plugin/src/site/apt/examples/multi-module-config.apt	5
/maven/plugins/trunk/maven-checkstyle-plugin/src/site/apt/index.apt	5
/maven/plugins/trunk/maven-checkstyle-plugin/src/site/apt/examples/custom-developed-checkstyle.apt	4
/maven/plugins/trunk/maven-checkstyle-plugin/src/site/apt/examples/custom-property-expansion.apt	4
/maven/plugins/trunk/maven-checkstyle-plugin/src/site/apt/examples/suppressions-filter.apt	4
/maven/plugins/trunk/maven-checkstyle-plugin/src/site/apt/usage.apt	4
/maven/plugins/trunk/maven-checkstyle-plugin/src/site/fml/faq.fml	4
/maven/plugins/trunk/maven-checkstyle-plugin/src/site/apt/examples/custom-checker-config.apt	3
/maven/plugins/trunk/maven-checkstyle-plugin/src/site/site.xml	3
/maven/plugins/trunk/maven-checkstyle-plugin/pom.xml	2
/maven/plugins/trunk/maven-checkstyle-plugin/src/main/java/org/apache/maven/plugin/checkstyle/Locator.java	2
/maven/plugins/trunk/maven-checkstyle-plugin/src/main/java/org/apache/maven/plugin/checkstyle/CheckstyleViolationCheckMojo.java	2
/maven/plugins/trunk/maven-ant-plugin/pom.xml	1
/maven/plugins/trunk/maven-ant-plugin/pom.xml	1
/maven/plugins/trunk/maven-antrun-plugin/pom.xml	1
/maven/plugins/trunk/maven-assembly-plugin/pom.xml	1
/maven/plugins/trunk/maven-clean-plugin/pom.xml	1
/maven/plugins/trunk/maven-clover-plugin/pom.xml	1
/maven/plugins/trunk/maven-compiler-plugin/pom.xml	1
/maven/plugins/trunk/maven-dependency-plugin/pom.xml	1
/maven/plugins/trunk/maven-deploy-plugin/pom.xml	1
/maven/plugins/trunk/maven-dao-plugin/pom.xml	1
/maven/plugins/trunk/maven-ear-plugin/pom.xml	1
/maven/plugins/trunk/maven-ejb-plugin/pom.xml	1
/maven/plugins/trunk/maven-help-plugin/pom.xml	1
/maven/plugins/trunk/maven-idea-plugin/pom.xml	1
/maven/plugins/trunk/maven-jar-plugin/pom.xml	1
/maven/plugins/trunk/maven-javadoc-plugin/pom.xml	1
/maven/plugins/trunk/maven-jxr-plugin/pom.xml	1
/maven/plugins/trunk/maven-one-plugin/pom.xml	1
/maven/plugins/trunk/maven-plugin-plugin/pom.xml	1
/maven/plugins/trunk/maven-pmd-plugin/pom.xml	1
/maven/plugins/trunk/maven-project-info-reports-plugin/pom.xml	1
/maven/plugins/trunk/maven-rar-plugin/pom.xml	1

Figura 3.6. Portal gerado pela ferramenta Maven [15]

A ferramenta utiliza o conceito de *Project Object Model (POM)*, onde todos os artefatos são gerados a partir do modelo definido para o projeto. Enquanto o *Ant*, para a realização de tarefas, necessita que *tasks* sejam codificadas, para a mesma finalidade, o *Maven* necessidade da definição de um modelo para o projeto.

O arquivo pom.xml (Quadro 3.14) descreve o modelo do projeto [8]. O nó principal do arquivo é o elemento <project>. Nele estão todas as informações do projeto.

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>br.teste</groupId>
    <artifactId>aprendendo-maven</artifactId>
    <packaging>war</packaging>
    <version>0.0.1</version>
    <name>Aprendendo o usar o maven </name>
        <dependencies>
            <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>4.0</version>
                <scope>test</scope>
            </dependency>
        </dependencies>
</project>
```

Quadro 3.14. Arquivo pom do Maven [8]

O elemento *project* possui como filhos:

- *modelversion*: descreve a versão do arquivo de configuração;
- *groupid*: indica o grupo ao qual o projeto pertence;
- *artifactid*: descreve o módulo dentro do grupo de projetos descrito no elemento *ModelVersion*;
- *packaging*: define o tipo de pacote que a *build* deve gerar. No caso de aplicações *Web*, é comum ser gerado um arquivo do tipo *WAR*;
- *version*: descreve a versão em que o projeto se encontra enquanto o elemento;
- *name*: descreve o nome do projeto;
- *build*: é responsável pela geração da *build* do sistema e tem como elemento filho o elemento *plugin*, responsável pela inserção de novas funcionalidades na ferramenta;
- *dependencies*: O elemento *dependencies* descreve o conjunto de bibliotecas que o *Maven* precisa em alguma fase do projeto. O elemento *dependency* tem os seguintes filhos:

§ *groupid*: identifica o grupo de projeto da dependência;

§ *artifactid*: define o módulo da dependência;

§ *version*: identifica a versão da dependência;

§ *scope*: define em que escopo do ciclo da *build* a dependência pode estar disponível. Este elemento pode assumir os valores: *compile*, *provided*, *test* e *system*.

- *compile*: a dependência fica disponível em todas as fases do projeto, desde a compilação até a instalação do sistema;
- *provided*: a dependência fica disponível para compilação. Em tempo de execução, ela deve ser disponibilizada pelo ambiente no qual a aplicação está executando;
- *test*: a dependência fica disponível para a execução dos testes do sistema e não é enviada junto com a aplicação;
- *system*: a dependência fica disponível no repositório e sua localização deve ser fornecida.

Nesta seção, foram descritas brevemente duas ferramentas de automação de *build*: o *Ant* e o *Maven*. Estas ferramentas tanto podem ser utilizadas isoladamente como podem ser acionadas por uma ferramenta de integração contínua. Na próxima seção, é apresentada a ferramenta de Integração Contínua chamada *CruiseControl*.

3.5. Ferramentas de Integração Contínua

As ferramentas de integração contínua têm como principal objetivo oferecer um framework para geração de builds, incluindo seus testes e também a visualização do estado presente e do histórico do desenvolvimento do sistema. Em seguida, a medida em que for sendo apresentada a ferramenta *CruiseControl*, verificar-se-a como estes como estes objetivos podem ser atendidos. .

3.5.1. *CruiseControl*

A ferramenta *CruiseControl* tem como propósito realizar a integração contínua, permitindo a publicação e a notificação de resultados.

A ferramenta observa e detecta mudanças no repositório. Quando uma mudança ocorre, o repositório local é atualizado, iniciando o processo de geração de build. Após a geração da *build*, o *CruiseControl* publica vários artefatos e informa o resultado (Figura 3.7).

A chamada ao script de geração de build é feita utilizando elementos que integram o *CruiseControl* a a ferramentas de automação como o *Ant* e o *Maven*. Uma diversidade de ferramentas de versionamento são suportadas pelo *CruiseControl*. O monitoramento do repositório pode acontecer em intervalos regulares ou disparado a qualquer momento pelo usuário da ferramenta. Ao final de todo o processo, é gerado um website com todos os resultados obtidos no processo de geração de build.

A principal vantagem do uso do *CruiseControl* é garantir que as builds são geradas a partir de quaisquer mudanças no repositório.

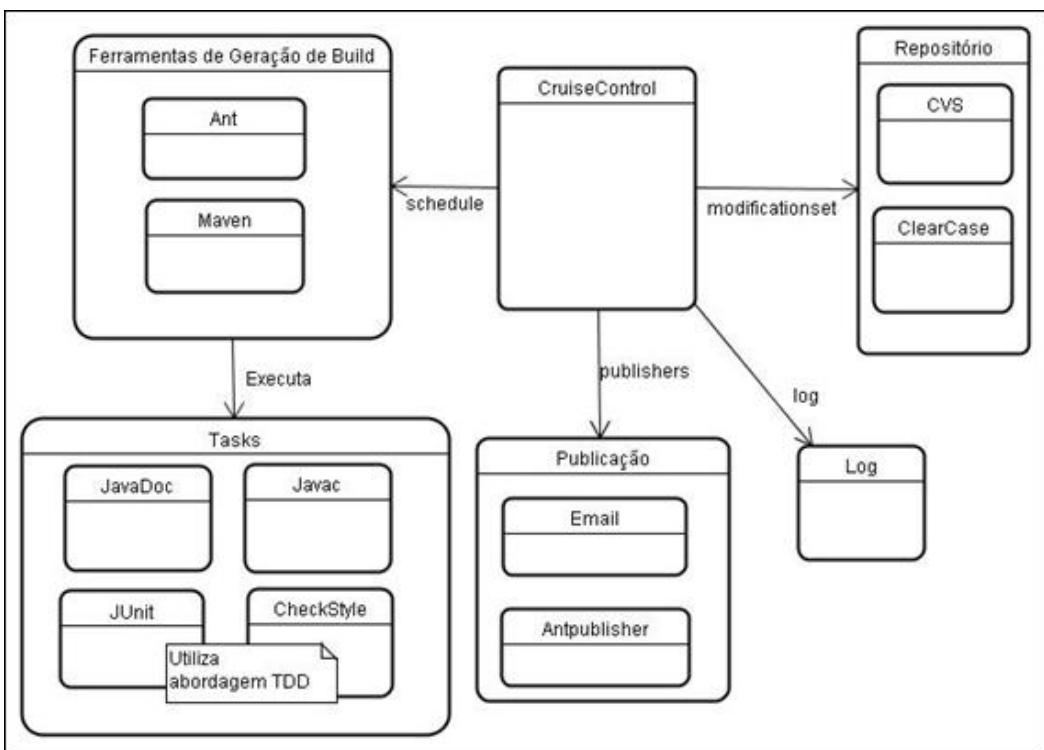


Figura 3.7. Integração Contínua Utilizando o *CruiseControl*

O primeiro passo para utilização da ferramenta é a criação dos diretórios de trabalhos:

- checkout: Diretório onde o Cruisecontrol realiza checkout do projeto na ferramenta de versionamento;
- logs: Diretório onde a ferramenta irá disponibilizar os relatórios;
- artifacts: Diretório onde a ferramenta irá disponibilizar quaisquer arquivos necessários para a geração e verificação da build.

A ferramenta utiliza um arquivo de configuração chamado config.xml. Este arquivo de configuração padrão do CruiseControl é chamado de config.xml. O elemento principal do arquivo config.xml é o cruisecontrol (Quadro 3.15), o qual possui os seguintes elementos filhos:

- project: Define projeto de integração contínua;
- plugin: Registra novas funcionalidades para a ferramenta;
- property: Define uma ou mais propriedades para o projeto.

```

<cruisecontrol>
  <project name="MEUPROJETO" buildafterfailed="true">
    <listeners>
      <currentbuildstatuslistener
        file="logs/PROJETOS/status.txt"/>
    </listeners>

    <!-- Bootstrappers são executados após cada execução de build
    antes da checagem de modificação -->
    <bootstrappers>
    </bootstrappers>

    <!--Define onde e em quanto tempo o CruiseControl busca por
    mudanças -->
    <modificationset quietperiod="10">
      <cvs localworkingcopy="checkout/MEUPROJETO"/>
    </modificationset>

    <!--Configura a geração de build -->
    <schedule interval="60">
      <ant anthome="C:\apache-ant-1.6.5"
          buildfile="build-MY_PROJECT_1.xml"
          target="build"
          uselogger="true"
          usedebug="false"/>
    </schedule>

    <!--Diretório do log -->
    <log logdir="logs/MEUPROJETO"/>
    <!--Publica resultados -->
    <publishers>
    </publishers>
  </project>
</cruisecontrol>

```

Quadro 3.15. Arquivo de configuração do CruiseControl

O elemento `project` define as tarefas de integração contínua, e tem como principais atributos:

- `name`: Identificador do projeto;
- `buildafterfailed`: Determina se a build deve continuar o processo caso seja encontrada alguma falha. O valor padrão para este campo é `true`;
- `requireModification`: Determina se é necessário alguma mudança no projeto para geração de uma nova build;
- `forceOnly`: Determina se uma nova build deve ser gerada apenas quando for solicitada;

O elemento `project` tem como elementos filhos:

- `Modificationset`;
- `Schedule`;
- `Log`;
- `Publisher`.

O elemento *Modificationset* (Quadro 3.16) indica como serão observadas as mudanças no repositório. O elemento possui um atributo chamado de *quietperiod*, que determina de quanto em quanto tempo a ferramenta irá ao repositório escolhido à procura por mudanças. O elemento *Modification* possui elementos filhos responsáveis por informar qual ferramenta de versionamento será utilizada. Entre estas ferramentas, podemos destacar:

- ClearCase: Monitora a ferramenta *ClearCase* usando o comando *cleartool lshistory*;
- CVS: Monitora o repositório da ferramenta CVS usando o comando *cvs log*;
- FileSystem: Monitora um diretório ou arquivo específico.

```
<modificationset quietperiod="10">
    <cvs localworkingcopy="checkout/MEUPROJETO" />
</modificationset>
```

Quadro 3.16. Monitoramento do CVS a partir do *CruiseControl*.

O elemento *Schedule* (Quadro 3.17) determina a geração de build de tempos em tempos de acordo com o valor preenchido no atributo *interval*. Através deste elemento, a ferramenta se integra com as ferramentas de geração de builds. A integração é realizada através dos elementos *ant* e *maven*. O elemento *ant* define qual arquivo de build o *CruiseControl* deve executar.

```
<schedule interval="30" >
    <ant antscript="build.sh" target="compile" />
    <ant antscript="build.sh" target="test" />
</schedule>
```

Quadro 3.15. Realizando chamada do *Ant* a partir do *CruiseControl*.

O elemento *Log* define onde o *CruiseControl* deve armazenar resultados alcançados pela execução da build . O elemento *log* possui os seguintes atributos:

- *dir*: Define o diretório de armazenamento do *log*. O valor padrão deste atributo é *./logs/[nome do projeto]*;
- *encoding*: Define o formato de *encoding* dos arquivos de *log*.

O elemento *publisher* os resultados de execução. Os principais publicadores disponíveis são:

- *Antpublisher*: executa um *script ant* responsável por uma publicação customizada;
- *Clearcasebaselinepublisher*: Cria uma *baseline* na ferramenta de versionamento *ClearCase*;
- *Cms synergy baseline publisher*: Cria uma *baseline* intermediária no banco CM Synergy;
- Email: Envia email com o link do resultado da *build*;

- *Ftppublisher*: Copia o arquivo de *log* e os artefatos para um servidor de *FTP*;
- *Jabber*: Envia mensagem instantânea para o *Jabber*;
- *Onfailure*: Executa um ou mais publicadores quando a build falha;
- *Onsuccess*: Executa um ou mais publicadores quando o processo de geração de builds foi bem sucedido.

O arquivo de execução do servidor chamado de *cruisecontrol.bat* está localizado na pasta *bin* da ferramenta. Após a execução da ferramenta, um portal é gerado para apresentar o resultado de todas as verificações (Figura 3.8).

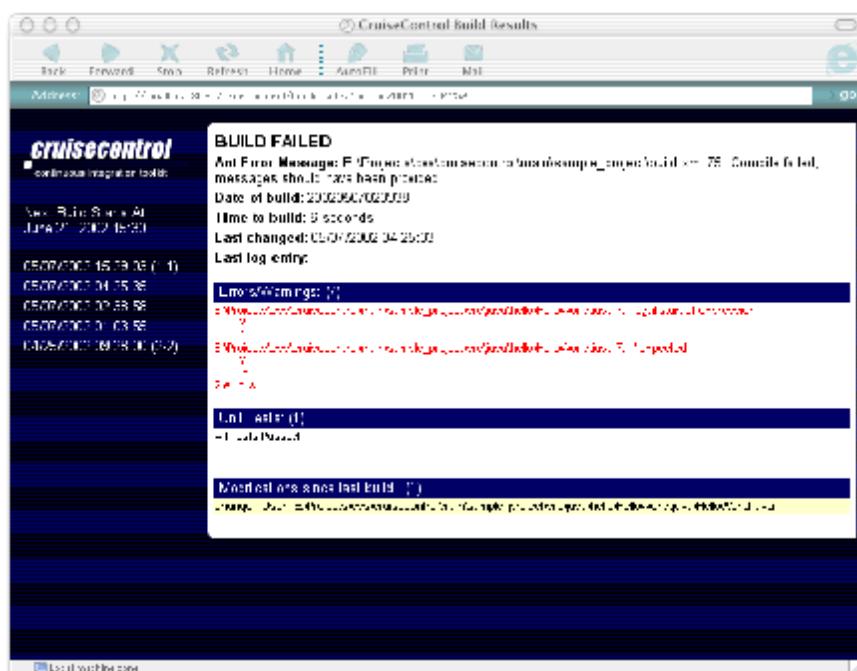


Figura 3.8. Portal gerado pela ferramenta CruiseControl

Nesta seção foi apresentada uma descrição sucinta da ferramenta *Cruisecontrol* com o objetivo de ilustrar o escopo das ferramentas de integração contínua.

3.6. Conclusão

No processo de Integração Contínua são produzidas tantas *builds* quantas modificações forem feitas no sistema. A cada construção de build todo o sistema é testado o que permite a verificação e correção de erros de integração num curto prazo de tempo.

Neste capítulo, inicialmente, foram apresentadas um conjunto de práticas normalmente utilizadas no processo de integração contínua. Em seguida foram apresentados conceitos de TDD e refatoramento. Entre as ferramentas que auxiliam a implementação da abordagem TDD foram apresentadas as ferramentas *JUnit* e *Mock Objects*.

A aplicação da integração contínua pressupõe a coordenação de várias ferramentas. Servidores de integração, como o *CruiseControl*, consultam o repositório

de arquivos gerenciado por ferramentas de controle de versionamento, disparam seqüências de tarefas através de ferramentas de automação de *build* e publicam sites para visualização e acompanhamento da evolução do sistema. Foram apresentadas duas ferramentas de automação de *build*: Ant e Maven.

A ferramenta Ant possui um conjunto de tarefas chamadas de targets, formada por subtarefas chamadas de tasks. Foram exibidos exemplos da utilização do Ant para tarefas de Compilação, Testes e Verificações de Código.

A ferramenta Maven oferece uma estrutura padronizada permitindo visualização via Web do histórico e da situação atual do projeto. A ferramenta utiliza o conceito de Project Object Model (POM), onde todos os artefatos são gerados a partir do modelo definido. A diferença em relação ao Ant é a não necessidade de criar boa parte das tasks, uma vez que estas estão pré-definidas na estrutura padronizada do Maven.

Finalmente foram apresentados detalhes da ferramenta de integração contínua CruiseControl que, além de disparar a geração de builds e seus respectivos testes, também publica o resultado do processamento de através do correio eletrônico ou em um portal.

Como vimos, a prática de integração contínua, apesar de simples em seus princípios, requer o conhecimento de uma gama de ferramentas. O conjunto de ferramentas aqui mencionado é apenas uma seleção dentre as várias opções disponíveis no mercado e é certamente muito mutável. Tanto as ferramentas abordadas devem evoluir incorporando gradativamente novas características como também novas ferramentas devem suplantar e substituir as ferramentas atuais.

Neste capítulo não se objetivava ser exaustivo em relação às características de nenhuma das ferramentas utilizadas. Desejava-se, ao contrário, ser sucinto o suficiente para elucidar os objetivos de cada ferramenta, exemplificar sua utilização e esclarecer os relacionamentos entre elas.

Enfim, foram fornecidos os conceitos intrínsecos e os objetivos da prática da integração contínua, e exemplificou-se como ela pode ser implementada.

Referências

- [1] Fowler, Martin (2006). “Continuous Integration”. Disponível em <http://www.martinfowler.com/articles/continuousIntegration.html>.
- [2] The Agile System Development Lifecycle (SDLC). Disponível em <http://www.ambyssoft.com/essays/agileLifecycle.html#Figure5TestingDuringDevelopment>.
- [3] Wikipedia (2007). Continuous Integration. Disponível em http://en.wikipedia.org/wiki/Continuous_Integration.
- [4] Duvall, Paul M, Glover, Andrew. (2007). Continuous Integration: Improving Software Quality And Reducing Risk. Addison-Wesley.
- [5] Lee, Kevin A (2007). IBM Rational ClearCase, Ant, and CruiseControl. IBM Press.
- [6] CruiseControl getting started, disponível em <http://cruisecontrol.sourceforge.net/>

- [7] TELES, Vinícius Manhães (2004). Extreme Programming: aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade. São Paulo: Novatec.
- [8] Junior, Maurício Linhares de Aragão. Automatizando seus projetos com o Maven 2. Disponível em <http://repo.maujr.org/artigos/maven2-guj/maven-2-guj.pdf>.
- [9] Severo, Carlos Emilio Padilla. NetBeans IDE refactoring – Parte I. Disponível em <http://www.devmedia.com.br/articles/viewcomp.asp?comp=1778>.
- [10] Ambler, Scott W. Introduction to Test Driven Design (TDD). Disponível em <http://www.agiledata.org/essays/tdd.html>.
- [11] Burn, Oliver. Checkstyle 4.3. Disponível em <http://checkstyle.sourceforge.net/>.
- [12] Test Driven Development – Part III: Continuous Integration. Disponível em <http://www.agiledeveloper.com/articles/TDDPartIII.pdf>
- [13] Teles, Vinícius Manhães (2007) Integração Contínua. Disponível em <http://www.improveit.com.br/xp/praticas/integracao>.
- [14] <http://www.urbancode.com/projects/anthill/default.jsp>
- [15] Beck, Kent (2002). Test-Driven Development By Example. Addison Wesley.
- [16] Astels, David (2003). Test-Driven Development: A Practical Guide. Prentice Hall PTR.
- [17] Apache Maven Project (2007). Welcome to Continuum. Disponível em: <http://maven.apache.org/continuum/>.
- [18] Wikipedia. Software build. Disponível em: http://en.wikipedia.org/wiki/Software_build.
- [19] Shore, James (2006). Successful Software: Continuous Integration on a Dollar a Day. Disponível em <http://www.jamesshore.com/Blog/Continuous-Integration-on-a-Dollar-a-Day.html>.
- [20] Fowler, Martin, Beck, Kent, Brant, John, Opdyke, William, Roberts, Don (1999). *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional.
- [21] Beck, Kent (1999). *Extreme Programming Explained: Embrace Change*. Beck, Kent.
- [22] Beck, Kent (2004). *JUnit Pocket Guide*. O'Reilly.
- [23] Massol, Vicent (2003). *JUnit in Action*. Manning Publications Company: 2003.
- [24] Rainsberger, J.B (2005). *JUnit Recipes: Practical Methods for Programmer Testing*. Manning Publications Co.

Capítulo

4

Engenharia Dirigida por Modelos

Denivaldo Lopes

Universidade Federal do Maranhão, São Luís - MA

Abstract

Model Driven Engineering (MDE) has been proposed to make face to the complexity of development, maintenance and evolution of software. In MDE, formal models can be manipulated and transformed between them and constitute the main artifact to assure the longevity, quality and low cost of production of software. MDE combines process and analysis with architecture. The Object Management Group's (OMG) Model Driven Architecture (MDA), the Eclipse Project's Eclipse Modeling Framework (EMF) and Microsoft's Software Factories are examples of MDE. MDE is a promising research trend and can comprise: code generation, transformation language and engine, metadata management, tools for mapping specification between models, algorithms of metamodel matching, processes and methodologies.

Resumo

MDE (Model Driven Engineering) tem sido proposta para fazer face à complexidade do desenvolvimento, manutenção e evolução de software. Em MDE, os modelos formais podem ser manipulados e transformados entre si e constituem o artefato principal para garantir a longevidade, a qualidade e o baixo custo de produção de um software. MDE combina processo e análise com a arquitetura. MDA (Model Driven Architecture) da OMG (Object Management Group), o framework de modelagem do projeto “Eclipse” e o software factories da Microsoft são exemplos de MDE. MDE é uma linha de pesquisa promissora, podendo abranger: geração de código a partir de um modelo de domínio, linguagens e motores de transformação, gerenciamento de metadados, ferramentas para a especificação de correspondências entre modelos, algoritmos de metamodel matching, processos e metodologias.

4.1. Introdução

Nas últimas décadas, a complexidade de desenvolvimento, manutenção e evolução de software tem crescido muito. A utilização isolada da tecnologia orientada a objetos, componentes e aspectos é insuficiente para fazer face à crescente complexidade. Aplicações na Internet, sistemas móveis, sistemas embarcados, uma variedade de sistemas operacionais, várias linguagens de programação, diferentes *middlewares* (por exemplo, CORBA, DCOM, Serviços Web [W3C 2004a] e JXTA [JXTA 2007]) são elementos deste conturbado contexto. Assim, pode-se observar que sistemas distribuídos heterogêneos estão mais difundidos do que nunca. Além disto, as empresas de software são desafiadas a reagir rapidamente a mudanças na lógica do negócio ou mudanças na tecnologia usada para implementar os sistemas de softwares. A fim de fornecer soluções racionais para suportar o gerenciamento desta complexidade, muitas organizações e empresas têm proposto abordagens baseadas em modelos. Um modelo pode ser definido como “uma abstração de um sistema físico que distingue o que é pertinente do que não é com o intuito de simplificar a realidade” [Muller 2004]. Esta nova tendência tem resultado em MDE (*Model Driven Engineering*) [Favre 2004, Fondement 2004, Kent 2002]. MDA (*Model Driven Architecture*) da OMG (*Object Management Group*) [OMG 2003a], Software Factories da Microsoft [Greenfield 2006] e EMF (*Eclipse Modeling Framework*) do projeto Eclipse [Budinsky 2003] são exemplos de MDE.

Já faz muito tempo que modelos estão sendo usados no desenvolvimento de software. Booch, OMT, UML e ferramentas CASE baseadas em modelos foram propostas nos anos 80 e 90 para promover o aperfeiçoamento do processo de desenvolvimento de software. Entretanto, neste período, os modelos foram frequentemente usados nas fases iniciais de desenvolvimento de software como análise de requisitos e projeto. As fases de implementação, testes e implantação foram frequentemente realizadas manualmente por programadores ou com uma inexpressiva utilização de modelos.

Com o advento de MDE, a utilização de modelos passou a ter uma dimensão mais ampla. Em MDE, modelos são usados em todas as fases do processo de desenvolvimento de software. Assim, cada modelo que é usado em uma fase pode ser projetado em outro modelo na fase subjacente. Por exemplo, um modelo sem aspectos de plataforma pode ser usado na fase de projeto. Em seguida, uma transformação pode traduzir este modelo em um outro modelo com aspectos de uma plataforma. Consequentemente, este modelo com aspectos da plataforma pode ser usado para gerar o código fonte na fase de implementação.

Na realidade, MDE não é uma nova abordagem que pretende substituir os processos de desenvolvimento de software como *Waterfall*, mas aprimorá-los com uma maneira racional de mover informações de uma fase para a outra fase do desenvolvimento de software.

Em MDE, modelos se tornam o centro do processo de desenvolvimento de software, separando aspectos independentes da plataforma de aspectos dependentes da plataforma. Outros conceitos e técnicas são importantes em MDE como metamodelos, linguagens de transformação, correspondências, processos e metodologias. UML (*Unified Modeling Languages*) [OMG 2007] e EDOC (*Enterprise Distributed Object Computing*) [OMG 2004] são linguagens de modelagem cuja especificação contém um

metamodelo usado para permitir a criação de modelos. ATL (*Atlas Transformation Language*) [Atlas group 2006], YATL (*Yet Another Transformation Language*) [Patrascoiu 2004] e *QVT transformation language* [OMG 2005a] são exemplos de linguagens de transformação. A especificação dos relacionamentos entre o metamodelo de UML e um metamodelo de WSDL é um exemplo de especificação de correspondência. Várias metodologias conforme uma abordagem MDE são propostas na literatura. Em [Lopes 2005a e Lopes 2005b], uma abordagem é apresentada propondo a separação entre especificação de correspondência da definição de transformação. Em [Lopes 2006], a ferramenta MT4MDE é proposta para permitir a modelagem de correspondência (i.e., a criação de especificação de correspondência), a determinação de correspondência entre modelos (i.e. dados dois metamodelos, a ferramenta determina uma possível especificação de correspondência) e a geração de definições de transformação a partir de um modelo de correspondência (i.e. especificação de correspondência).

Neste capítulo, as principais tecnologias e especificações que fundamentam MDE, algumas ferramentas que suportam MDE e as tendências de pesquisa que abordam MDE serão apresentadas e discutidas.

4.2. Engenharia Dirigida por Modelos

Antes de apresentar MDE em detalhes, faz-se necessário discorrer sobre vários conceitos de base sobre MDE.

4.2.1. Conceitos de base

Antes de apresentar alguns conceitos sobre MDE, faz-se necessário discutir o conceito de modelo e linguagem de modelagem.

Segundo o dicionário Hachette, um modelo é “algo proposto à imitação”. Esta definição é muito generalista. As definições seguintes são encontradas na literatura técnica da informática e fornecem mais detalhes sobre o termo modelo na informática.

Um modelo “é uma abstração de um sistema físico que distingue o que é pertinente do que não é com o intuito de simplificar a realidade. Um modelo contém todos os elementos necessários à representação de um sistema real” [Muller 2004].

Um modelo é “uma simplificação de um sistema e é criado com uma finalidade específica. O modelo deve ser capaz de responder as perguntas em lugar de um sistema em estudo. As respostas fornecidas pelo modelo devem ser as mesmas que aquelas fornecidas pelo sistema, a condição que as perguntas e respostas restem no limite do domínio definido pelo objetivo geral do sistema” [Bézivin 2001].

Um modelo é “uma descrição de um (ou de uma parte de) sistema expresso em uma linguagem bem definida, isto é, respeitando um formato preciso (uma sintaxe) e um significado (uma semântica). Esta descrição deve ser conveniente para uma interpretação automatizada por computador” [Kleppe 2003].

Um modelo é “uma abstração de um sistema” [Frankel 2003]. “Uma abstração é uma descrição de qualquer coisa omitindo certos detalhes não pertinentes para o objetivo da abstração” [OMG 2001].

Um modelo é “uma descrição de um sistema físico com certo objetivo, por exemplo, a descrição de aspectos lógicos e comportamentais de um sistema físico para certa categoria de leitores. Assim, um modelo é uma abstração de um sistema físico que é especificado a partir de um ponto de vista (*viewpoint*), isto é, para certa categoria de *stakeholders*, por exemplo, os conceptores, os utilizadores, os clientes de um sistema, e em respeito a certos níveis de abstrações, de acordo com os objetivos do modelo” [OMG 2007].

Um modelo é “uma simplificação de qualquer coisa e permite ver, manipular e raciocinar sobre o assunto em estudo, e que ajuda a compreender a complexidade inerente ao assunto” [Mellor 2004].

Na literatura, várias definições do termo modelo podem ser encontradas. Entretanto, elas transmitem a mesma idéia: imitação do real, representação, simplificação, abstração e descrição limitada de um sistema. Remarca-se também que um modelo é criado com um objetivo específico e dentro de um contexto particular. Assim, o objetivo e o contexto são os elementos que irão dirigir as escolhas feitas durante a criação de um modelo. Um modelo não é uma descrição completa de um sistema, mas ele permite a elaboração de ponderações sobre o sistema em estudo dentro de um contexto limitado. Os modelos são necessários justamente por que eles permitem simular os sistemas físicos antes de sua construção, fornecendo uma maneira simples, manipulável e econômica de prever os elementos de um sistema e de suas relações. Um modelo pode também ser deduzido de um sistema existente ou de um sistema em desenvolvimento. Os modelos são essenciais para estabelecer uma comunicação eficaz entre os membros de uma equipe e entre as equipes implicadas em um projeto, e por assegurar a robustez arquitetural de um sistema em concepção ou em análise.

Diferentes tipos de modelos existem, por exemplo, os modelos físicos, os modelos matemáticos e os modelos na informática. Neste capítulo, os modelos na informática constituem o foco deste estudo, em particular modelos no contexto de MDE.

A criação de modelos é feita utilizando uma linguagem bem definida, isto é, com uma sintaxe e uma semântica especificadas para regulamentar a criação dos elementos e suas relações. Assim, um modelo pode ser, por exemplo, concebido em uma linguagem matemática ou em uma linguagem gráfica.

Uma linguagem de modelagem é uma especificação formal bem definida que contém os elementos de base para construir modelos. Além disto, uma linguagem de modelagem é concebida dentro de um domínio limitado e com objetivos específicos. Assim, a existência de várias linguagens de modelagem parece ser razoável, cada uma sendo adaptada a um domínio específico. UML e EDOC são exemplos de linguagem de modelagem. A linguagem concebida para criar modelos é frequentemente descrita por um metamodelo, isto é, o que precede o modelo.

Na literatura, várias definições sobre metamodelo podem ser encontradas.

Um metamodelo é “um modelo de uma linguagem de modelagem” [Favre 2004].

Um metamodelo é “um modelo que define a linguagem para exprimir um modelo” [OMG 2007].

Um metamodelo é criado usando outras linguagens conhecidas como linguagem de metamodelagem. Cada linguagem de metamodelagem corresponde a um metametamodelo, isto é, o que precede um metamodelo.

Um metametamodelo é “um modelo que define a linguagem para expressar metamodelos. A relação entre um memetamodelo e um metamodelo é similar à relação entre um metamodelo e um modelo” [OMG 2007]. Dentre as linguagens de metamodelagem, destacam-se MOF (*Meta-Object Facility*) da OMG [OMG 2006] e Ecore do Projeto Eclipse [Budinsky 2003].

A relação entre um metametamodelo, um metamodelo, um modelo e uma informação está bem definida na arquitetura de metamodelagem (chamado também de *framework* de metamodelagem). Esta relação é baseada na arquitetura a quatro níveis apresentada na Figura 4.1.

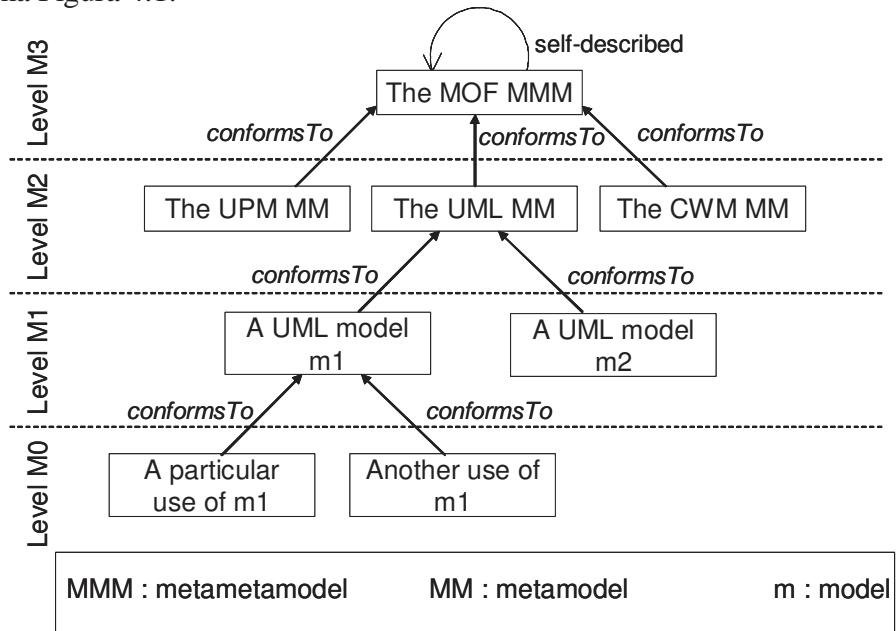


Figura 4.1. A arquitetura a quarto níveis

Neste documento, define-se que as relações entre o nível M3 e M2, o nível M2 e M1 e o nível M1 e M0 são do tipo ConformTo (conforme à). Os níveis podem ser descritos como a seguir [Frankel 2003 e Mellor 2004]:

- M3 (metametamodelo): o nível M3 constitui a base da arquitetura de metamodelagem. A função primordial deste nível é definir linguagens para especificar metamodelos. Um metametamodelo define um modelo de mais alto nível de abstração que o metamodelo, e este primeiro é tipicamente mais compacto que o segundo;
- M2 (metamodelo): um metamodelo é conforme a um metametamodelo. A função principal do nível de metamodelo é definir uma linguagem para especificar modelos. Os metamodelos são tipicamente mais elaborados que os metametamodelos;

- M1 (modelo): um modelo é conforme a um metamodelo. A função principal do nível de modelo é definir uma linguagem para descrever um domínio da informação;
- M0 (informação): os objetos de usuários representam as informações finais. A principal responsabilidade dos objetos de usuários é descrever um domínio da informática em específico em uma plataforma final.

Nesta arquitetura, deve-se notar a existência de poucos metametamodelos (por exemplo, MOF e Ecore), vários metamodelos (por exemplo, UML, EDOC e UEML), um grande número de modelos e uma infinidade de informações.

Na literatura, vários aspectos de MDE foram identificados, por exemplo:

- “MDE tem sua ênfase nas pontes entre espaços tecnológicos e na integração de campos de conhecimento desenvolvido por diferentes comunidades” [Favre 2004];
- “MDE é mais abrangente em escopo do que MDA. MDE combina processo e análise com arquitetura” [Kent 2002];
- MDE propõe “um *framework* (1) para claramente definir metodologias, (2) para desenvolver sistemas em qualquer nível de abstração, (3) para organizar e automatizar as atividades de teste e validação. Além disto, esta técnica assegura que qualquer especificação deve ser expressa por modelos, que são compreendidos por humanos e computadores” [Fondement 2004].

Muitos benefícios fornecidos por MDE foram identificados tais como a economia de tempo e de recursos e a proteção contra erros. MDE ainda está evoluindo e necessita de mais tempo para estar estabilizada.

4.2.2. MOF

MOF foi adotado em 1997 pela OMG. A especificação de MOF define uma linguagem abstrata e um *framework* para a especificação, a construção e o gerenciamento de metamodelos neutros de tecnologia. MOF também define um framework para a implementação de repositórios que contêm os metadados (i.e. modelos) descritos pelos metamodelos. Este framework utiliza as correspondências de tecnologias normalizadas para transformar os metamodelos MOF em uma das API de metadados [OMG 2006].

A especificação MOF comporta:

- Uma definição formal do metamodelo de MOF;
- Uma correspondência dos metamodelos MOF em IDL CORBA que produz as interfaces IDL para gerar um metadado;
- Um conjunto de interfaces IDL CORBA para representar e manipular metamodelos MOF;
- Um formato XMI para a troca de metamodelos MOF.

4.2.3. Ecore

Ecore é uma linguagem de metamodelagem que faz parte de EMF (*Eclipse Modeling Framework*) que é o resultado dos esforços do projeto Eclipse (*Eclipse Tools Project*). EMF é um *framework* de modelagem e de geração de código que suporta a criação de ferramentas e de aplicações dirigidas por modelos [Budinsky 2003].

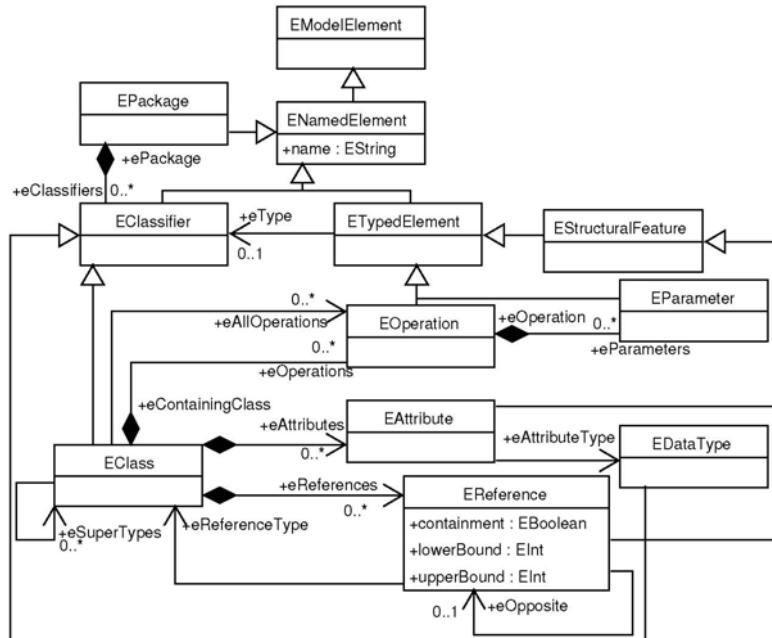


Figura 4.2. O metamodelo Ecore (fragmento)

A Figura 4.2 apresenta um fragmento do metamodelo de Ecore [Budinsky 2003]. Este fragmento de Ecore apresenta os elementos seguintes:

- ENamedElement: é uma generalização para EClassifier e ETypedElement;
- EPackage: agrupa as classes (i.e. EClass) e os tipos de dados (i.e. EDataType);
- EClassifier: é um elemento comum que é especializado por um EClass ou um EDataType;
- EClass: é identificado por um nome e contém os atributos, as referências e as operações;
- EAttribute: contém os dados que pertencem a uma classe;
- EReference: permite a criação de associações entre classes;
- EOperation: contém o comportamento de uma classe.

Algumas ferramentas permitem definir e manipular metamodelos conforme à Ecore. Dentre eles, pode-se citar:

- O *plug-in* de EMF que contém um editor para a criação de metamodelos utilizando Ecore (<http://www.eclipse.org/emf>);

- A ferramenta Omondo (<http://www.omondo.com>).

4.2.4 XMI

XMI (*XML Metadata Interchange*) permite a troca de modelos serializados em XML. XMI é focalizado na troca de metadados conforme o MOF [OMG 2005b]. Embora XMI já esteja na versão 2.1, as ferramentas de modelagem em sua grande maioria ainda trabalham com XMI versão 1.2.

O objetivo de XMI é permitir a serialização e troca de metamodelos conforme ao MOF e de modelos conforme a estes metamodelos na forma de arquivos utilizando dialetos XML. XMI permite também serializar metamodelos que são criados com outras linguagens de metamodelagem como Ecore. Isto é possível porque XMI não define um dialeto XML único, mas um conjunto de regras que permitem criar um DTD ou XML schema para diferentes metamodelos.

Assim, os metamodelos conforme ao MOF ou ao Ecore e seus modelos respectivos podem ser portados entre ferramentas e repositórios usando XMI.

A Figura 4.3 ilustra um simples modelo UML, e a Figura 4.4 apresenta sua representação correspondente em XMI.

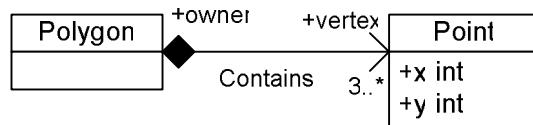


Figura 4.3. Um modelo de polígono.

```

<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Tue Nov 23 21:00:33 CET 2004'>
  ***
  <XMI.content>
    <UML:Model xmi.id = 'a1' name = 'model 1' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
      ***
      <UML:Class xmi.id = 'a4' name = 'Polygon' visibility = 'public' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
        </UML:Class>
        <UML:Class xmi.id = 'a6' name = 'Point' visibility = 'public' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
          ***
          <UML:Classifier.feature>
            <UML:Attribute xmi.id = 'a8' name = 'x' visibility = 'public' isSpecification = 'false' ownerScope = 'instance'>
              ***
              </UML:Attribute>
              <UML:Attribute xmi.id = 'a12' name = 'y' visibility = 'public' isSpecification = 'false' ownerScope = 'instance'>
              ***
              </UML:Attribute>
            </UML:Classifier.feature>
          </UML:Class>
        <UML:Association xmi.id = 'a15' name = 'Contains' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
          ***
          </UML:Association>
        ***
      </XMI.content>
    </XMI>
  
```

Figura 4.4. A representação correspondente em XMI do modelo polígono

4.3. Arquitetura Dirigida por Modelo (MDA)

Devido à limitação de números de páginas deste documento, aborda-se somente MDA já que é uma abordagem representativa de MDE.

Nesta seção, aborda-se MDA considerando as seguintes questões: O que é MDA? Por que MDA? Quais são os benefícios de MDA? Por quem MDA foi concebido e normalizado?

4.3.1. O que é MDA?

Na literatura, várias definições podem ser encontradas para MDA. Nos parágrafos seguintes, apresentam-se algumas destas definições.

MDA é “uma evolução do OMA para assegurar a integração e a interoperabilidade durante o ciclo de vida de um sistema desde o modelo do negócio (*business modeling*) e sua concepção até a construção de componentes, a montagem, a integração, a implantação, a gestão e a evolução” [OMG 2001]. MDA é fundamentada na experiência adquirida pela criação de especificações (padrões) como UML, MOF, CWM, XMI e IDL. MDA define uma arquitetura para estruturar modelos a fim de permitir a integração, a interoperabilidade e a portabilidade no nível de modelos.

MDA é uma abordagem que aprimora o potencial de modelos no desenvolvimento de sistemas. MDA é dirigida por modelos “porque ela fornece uma abordagem de utilização de modelos para dirigir a compreensão, a concepção, a implantação, a operação, a manutenção e a modificação de sistemas” [OMG 2003].

MDA é “uma abordagem para a especificação de sistemas de tecnologia de informação. Ela separa a especificação das funcionalidades da especificação da implementação em uma plataforma tecnológica específica” [OMG 2003].

O padrão MDA da OMG é “um caso particular da abordagem MDE” [Favre 2004].

Em suma, define-se MDA como uma abordagem de desenvolvimento baseado em modelos e um conjunto de especificações (padrões) da OMG. Esta abordagem permite separar as especificações das funcionalidades de um sistema (o PIM – *Platform Independent Model*) das especificações de plataforma de um sistema (o PSM – *Platform Specific Model*). As especificações fundamentais de MDA são IDL, UML, MOF, CWM e XMI. No caso de MDA, a OMG sustenta a transformação como o operador fundamental devendo ser aplicado entre modelos.

Na abordagem MDA, tudo é considerado como um modelo, tanto um XML *schema* quanto um código fonte ou um binário. O modelo independente de plataforma (PIM) e o modelo específico da plataforma (PSM) constituem os principais tipos de modelos em MDA. Os PIMs são modelos que não têm dependência de uma plataforma tecnológica (i.e. eles são então independentes de CORBA, EJB, Serviços Web e dotNET). Os PSMs são modelos dependentes de plataformas tecnológicas e servem de base para a geração de código fonte.

Uma plataforma é definida como “*a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns*,

which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented” [OMG 2003].

As operações de transformação constituem a base da abordagem MDA, podendo ser agrupadas em:

- PIM a PIM: estas transformações se efetuam para acrescentar ou subtrair informações de modelos. Por exemplo, a passagem da fase de análise à fase de concepção permite acrescentar informações. Tais transformações não são naturalmente automatizadas e requerem a intervenção do desenvolvedor para acrescentar ou subtrair informações. Outro exemplo é o refinamento do PIM com o intuito de deixá-lo mais completo;
- PIM a PSM: estas transformações se efetuam quando um PIM está suficientemente enriquecido com as informações do negócio para poder ser transformado em uma plataforma tecnológica. Um exemplo deste tipo é a transformação de um modelo UML contendo a lógica de um negócio em um outro modelo UML que agrega uma plataforma como os serviços Web;
- PSM à PIM: estas transformações são úteis para separar uma lógica do negócio da sua plataforma tecnológica. Por exemplo, uma transformação de um PSM contendo perfis CORBA que suprime estes perfis, retornando apenas a lógica do negócio. Particularmente, este tipo de transformação é mais requisitado no processo de engenharia reversa;
- PSM a PSM: estas transformações são necessárias nas fases de refinamento de plataformas tecnológicas, de implantação, de otimização, de configuração ou mudança de plataforma tecnológica.

As relações entre PIM, PSM, metamodelos, infra-estrutura (i.e. plataforma) e outras tecnologias são descritas pelo metamodelo MDA descrito na Figura 4.5 [OMG 2001].

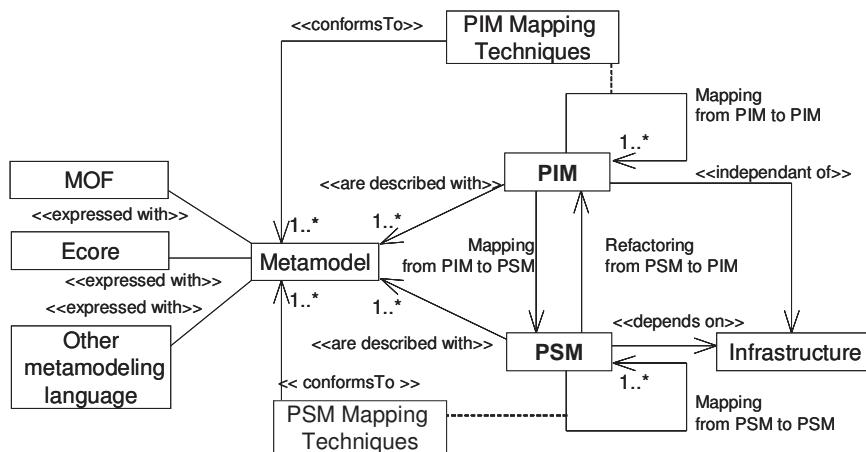


Figura 4.5. A descrição do metamodelo MDA [OMG 2001]

Analizando a Figura 4.5, pode-se observar que MDA fornece o processo pelo qual um modelo é transformado em um outro modelo. A Figura 4.6 apresenta uma outra ilustração do processo de transformação em MDA [Bézivin 2004].

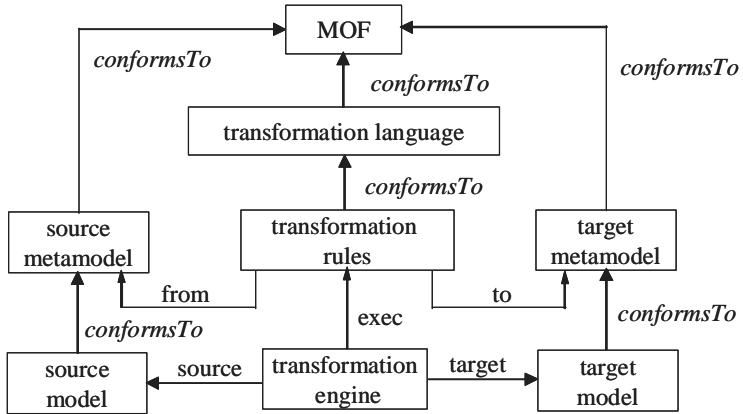


Figura 4.6. A transformação de modelos em MDA [Bézivin 2004]

4.3.2. Por que MDA?

A OMG compreendeu que um *middleware* como CORBA não pode ser uma solução suficiente para responder as novas exigências de um novo contexto no qual os sistemas de software se tornaram mais complexos devido a diversos fatores como:

- A integração de novos aspectos: a segurança, a disponibilidade e a confiabilidade devem ser consideradas desde o início do ciclo de desenvolvimento (análise e concepção) de um sistema;
- A chegada de novas tecnologias: atualmente, os sistemas evoluem mais rapidamente do que no passado, não somente porque as exigências do negócio e das aplicações mudam, mas também porque as plataformas tecnológicas estão constante e rapidamente evoluindo. Sendo assim, a proteção dos investimentos em softwares contra a obsolescência devido a integração de novas plataformas tecnológicas é um importante fator a ser considerado;
- A compatibilidade com as tecnologias legadas (*legacy systems*): novas tecnologias aparecem a uma alta velocidade, mas as tecnologias anteriores não desaparecem e são agregadas em um sistema legado, gerando um problema de heterogeneidade entre as tecnologias;
- O grande número de tecnologias: os sistemas computacionais distribuídos são geralmente constituídos de várias tecnologias. Além disto, estes sistemas devem se comunicar de uma maneira transparente.

Estes fatores unidos tornam a produtividade e a qualidade de sistemas de softwares difíceis de serem asseguradas. Além disto, o tempo de entrega dos sistemas de softwares deve ser cada vez mais curto em consequência da necessidade que as empresas têm de responderem rapidamente as novas exigências e mudanças.

Assim, a OMG tomou a decisão de investir em uma abordagem dirigida por modelos para fazer face à complexidade de desenvolvimento, manutenção e evolução de sistemas de software.

A idéia principal de MDA pode ser ilustrada por seu logotipo na Figura 4.7. Os modelos são entidades capazes de unificar e suportar o desenvolvimento de sistemas de informática, assegurando a interoperabilidade e a portabilidade. Os modelos representam o núcleo (e.g. MOF, UML e CWM), os *middlewares* representam o nível de execução das aplicações (e.g. CORBA, Serviços Web e dotNET) e os serviços padronizados fornecessem o suporte à execução destas aplicações (e.g. segurança, transação e eventos). Todo este conjunto de tecnologias é usado para suportar diferentes domínios tais como o financeiro, o comércio eletrônico e as telecomunicações. MDA surgiu não somente para gerenciar a complexidade, mas também para harmonizar as diversas tecnologias. Não se deve considerar que as tecnologias utilizadas no passado, e.g. objeto, componentes e *middlewares*, falharam, mas que elas não foram criadas para resolver o nível atual de complexidade dos sistemas de software. A disseminação da Internet e utilização em massa de sistemas de software resultaram em novas exigências jamais imaginadas. Assim, estas tecnologias atingiram seus limites e, atualmente, não são mais capazes de fornecer o nível requerido de abstração.

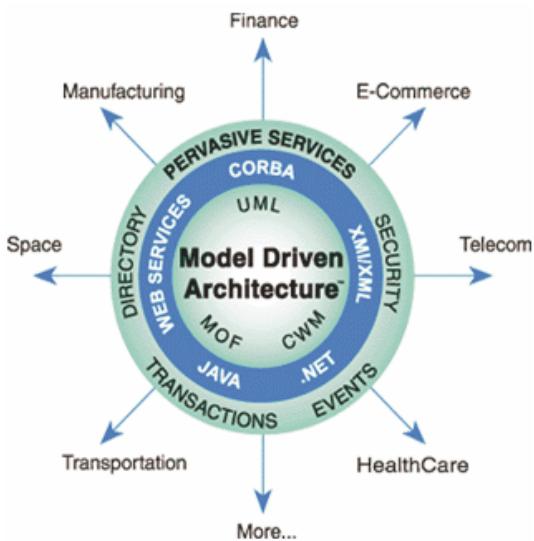


Figura 4.7. A transformação de modelos em MDA [OMG 2001]

4.3.3. Quais são os benefícios da abordagem MDA?

Certos benefícios da abordagem MDA não foram ainda comprovados, mas os resultados atuais são encorajadores. Na literatura, os seguintes benefícios foram identificados:

- O mesmo PIM pode ser utilizado várias vezes para gerar os modelos em plataformas diferentes (PSMs) [Gerber 2002];
- Os diferentes pontos de vista de um mesmo sistema, i.e. vários níveis de abstração ou detalhes de implementação [OMG 2001];
- O aumento da portabilidade e da interoperabilidade de sistemas através de modelos [OMG 2003];
- A proteção da lógica do negócio contra as mudanças ou a evolução das tecnologias envolvidas [OMG 2001];
- A evolução simultânea dos modelos do negócio e das tecnologias;

- A proteção contra os fatores inerentes ao desenvolvimento manual de sistemas;
- O aumento no retorno do investimento em tecnologias;
- O suporte a reengenharia e engenharia reversa que torna possível a recuperação da lógica do negócio a partir de código fonte ou dos ambientes de implementação;
- O aumento do nível de abstração, garantindo que os desenvolvedores possam se comunicar de uma forma mais produtiva;
- A reutilização e a composição de modelos tornam possível a construção de sistemas complexos;
- O aumento da interação e da migração entre os diferentes espaços tecnológicos [Kurtev 2002].

4.3.4. Por quem MDA foi concebida e normalizada?

MDA é uma iniciativa da OMG a partir do constato de que a complexidade existente no desenvolvimento, manutenção e evolução de sistemas não podia ser mais gerenciada unicamente pela utilização de CORBA. Atualmente, uma especificação MDA não existe, mas sim um conjunto de especificações tais como UML, CWM, XMI, MOF, OCL e, mais recentemente, MOF QVT. Estas especificações fornecem a base e o direcionamento para a criação e a utilização da abordagem MDA.

4.4. Linguagens de transformação de modelos

Várias linguagens de transformação de modelos foram propostas, dentre estas se destacam MOF QVT da OMG e ATL da Universidade de Nantes.

4.4.1. MOF QVT

Em 2002, a OMG iniciou um processo de padronização através de seu RFP MOF 2.0 Query/View/Transformation (QVT) para estimular a criação de uma linguagem de transformação de modelos padronizada. Uma dezena de submissões foi apresentada a OMG em resposta a este RFP, sendo que a proposta do grupo QVT-Merge foi retida como versão preliminar. Atualmente, a especificação MOF QVT 2.0 já possui uma versão adotada [OMG 2005a].

A especificação de MOF QVT 2.0 é baseada em MOF e OCL e possui os seguintes elementos principais:

- *Query*: uma requisição é uma expressão que é avaliada sobre um modelo. O resultado de uma requisição é uma ou várias instâncias de tipos definidos no modelo fonte ou definidos pela linguagem de requisições. No contexto de MDA, OCL é a linguagem mais utilizada para fazer requisições;
- *View*: uma vista é um modelo que é completamente derivado de um outro modelo. Uma vista não pode ser modificada separadamente de seu modelo do qual ela foi derivada. Uma requisição é um tipo restrito de vista gerado por transformações;

- *Transformations*: uma transformação gera um modelo alvo a partir de um modelo fonte.

A especificação MOF QVT tem uma natureza híbrida, pois possui partes declarativas e imperativas.

A parte declarativa pode ser dividida em dois níveis de arquitetura:

- *Relations* é constituído de um metamodelo e uma linguagem que suportam a combinação de padrões de objetos e a criação de *templates* de objetos;
- *Core* é um metamodelo e uma linguagem definidos usando extensões mínimas de EMOF e OCL.

A parte imperativa é constituída de *operational mappings* e *black-box MOF operations*.

Operational mappings é uma linguagem padrão para fornecer implementações imperativas para a parte da linguagem fornecida por *relations*. Com esta finalidade, extensões OCL são fornecidas para permitir um estilo procedural e uma sintaxe concreta que o torna familiar a uma linguagem imperativa.

Implementações de *Black-box* devem explicitamente implementar uma *Relation*, que é responsável por manter traços entre os elementos do modelo relacionados pelas implementações de operações.

A Figura 4.8 ilustra os relacionamentos entre os metamodelos de QVT.

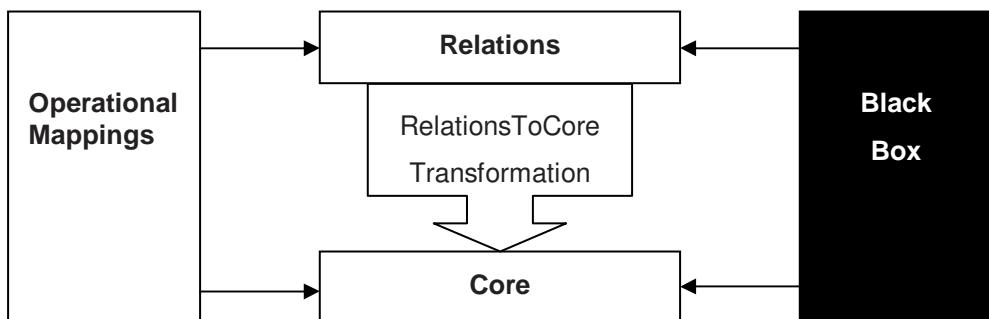


Figura 4.8. Relacionamentos entre metamodelos de QVT [OMG 2005a]

MOF QVT não é uma implementação, mas uma especificação. Atualmente, não existe uma linguagem de transformação que seja completamente conforme a MOF QVT. Entretanto, várias linguagens apresentam diferentes aspectos de MOF QVT tal como ATL.

4.4.2. ATL

ATL (*Atlas Transformation Language*) é uma linguagem para a realização de transformações de modelos no contexto de MDE [Bézivin 2003]. ATL é uma linguagem de transformação independente de repositório, podendo trabalhar tanto com MDR (*MetaData Repository*) [NetBeans 2007] quanto EMF [Budinsky 2003].

Uma requisição em ATL é uma expressão em OCL que pode retornar tipos primitivos, elementos de um modelo, coleções, *n-uplets* ou uma combinação destes tipos (e.g. coleções de *n-uplets*). A versão atual de ATL não suporta a transformação

incremental e nem a bidirecionalidade. Entretanto, os desenvolvedores de ATL preconizam a utilização do suporte de traçabilidade para realizar esta característica em ATL. A abordagem imperativa em ATL contém instruções que explicitam as etapas de execução de um procedimento denominado de *helpers*.

Em ATL, uma definição de transformação começa pela declaração do nome do módulo que permite a criação de espaços de nomes. Deve-se utilizar a palavra reservada `create` seguida do nome do metamodelo de saída e da palavra reservada `from` seguida do metamodelo de entrada. Assim, definem-se os metamodelos usados no processo de transformação. ATL também permite o uso de bibliotecas de definições de transformação pelo uso da palavra reservada `uses` que funciona similarmente a `import` de Java. Uma regra de transformação puramente declarativa usa a palavra reservada `from` para especificar o elemento de entrada, a palavra `to` para especificar o elemento de saída e um conjunto de ligações (*bindings*) representadas pelo símbolo `<-`. A Figura 4.9 ilustra um fragmento de uma definição de transformação em ATL para transformar uma classe em UML em outra classe em Java.

```
module UML2JAVA ;
create OUT : JAVA from IN : UML ;
rule Class2JClass{
    from uclass : UML !Class
    to jclass : JAVA !JClass(
        uclass.name <- jclass.name
    )
}
```

Figura 4.9. Uma regra completamente declarativa em ATL

As instruções imperativas em ATL podem ser agrupadas em:

- Expressões: as expressões são baseadas em OCL;
- Variáveis: uma declaração de variável é feita pela palavra reservada `let`;
- Atribuições: o operador de *bindings* `<-` pode ser utilizado com esta finalidade;
- Manipulação de instâncias: as instâncias podem ser criadas de maneira implícita graças à abordagem declarativa. Além disto, a criação e destruição de uma instância podem ser feitas usando o operador `new` e `delete`, respectivamente;
- Declarações condicionais como `if ... then ... else ... endif`, `switch`, `while`, `do... while` e `foreach`.

4.5. Uma abordagem para MDE

Na literatura sobre modelagem de sistemas, duas tendências complementares e não exclusivas a respeito das técnicas de transformação podem ser percebidas. A primeira consiste em estudar os problemas ligados a criação de correspondências entre metamodelos (ou modelos). A segunda consiste na concepção e na realização de uma linguagem de transformação segundo as recomendações da OMG. Entretanto, na literatura sobre MDA, estas duas tendências são frequentemente misturadas. Em [Lopes

2005a], uma separação explícita entre especificação de correspondência e a definição de transformação é proposta conforme a Figura 4.10.

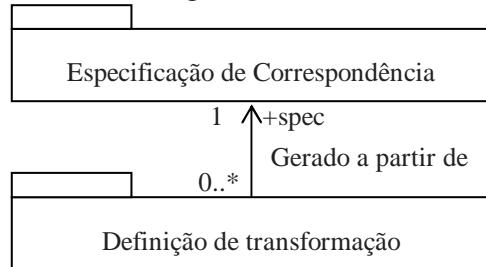


Figura 4.10. Relação entre especificação de correspondência e definição de transformação

A especificação de correspondência apresenta a lógica utilizada para estabelecer as relações entre dois metamodelos, enquanto a definição de transformação contém regras precisas para transformar um modelo em um outro modelo. Na realidade, a especificação de correspondência pode ser considerada como um PIM e a definição de transformação como um PSM.

Esta separação explícita tem implicações significativas resultando em uma nova arquitetura tipo para a transformação de modelos ilustrada na Figura 4.11. Nesta arquitetura tipo, o modelo de correspondência (i.e. a especificação de correspondência – mapping M) é separado do modelo de transformação (i.e. a definição de transformação – transformation M).

Esta proposta de arquitetura tipo contém as seguintes entidades:

- **MM** (um metamodelo): por exemplo, MOF ou Ecore;
- **Source MM e target MM** (um metamodelo fonte e um alvo): por exemplo, o metamodelo UML ou EDOC;
- **Source M e target M** (um modelo fonte e um alvo): por exemplo, um modelo de uma agência de viagem em UML que deve ser transformado em um modelo Java;
- **mapping MM** (um metamodelo de correspondência): a linguagem para modelagem de correspondências entre os elementos de um metamodelo fonte e os elementos de um metamodelo alvo;
- **mapping M** (um modelo de correspondência): o modelo contendo as correspondências entre dois metamodelos;
- **transformation MM** (um metamodelo de transformação): o formalismo que permite a criação precisa de transformações de um modelo fonte em um modelo alvo;
- **transformation M** (um modelo de transformação): descreve a transformação de modelos;
- **transformation program** (um programa de transformação): um programa executável para realizar a transformação;

- transformation engine (um motor de transformação): executa um programa de transformação.

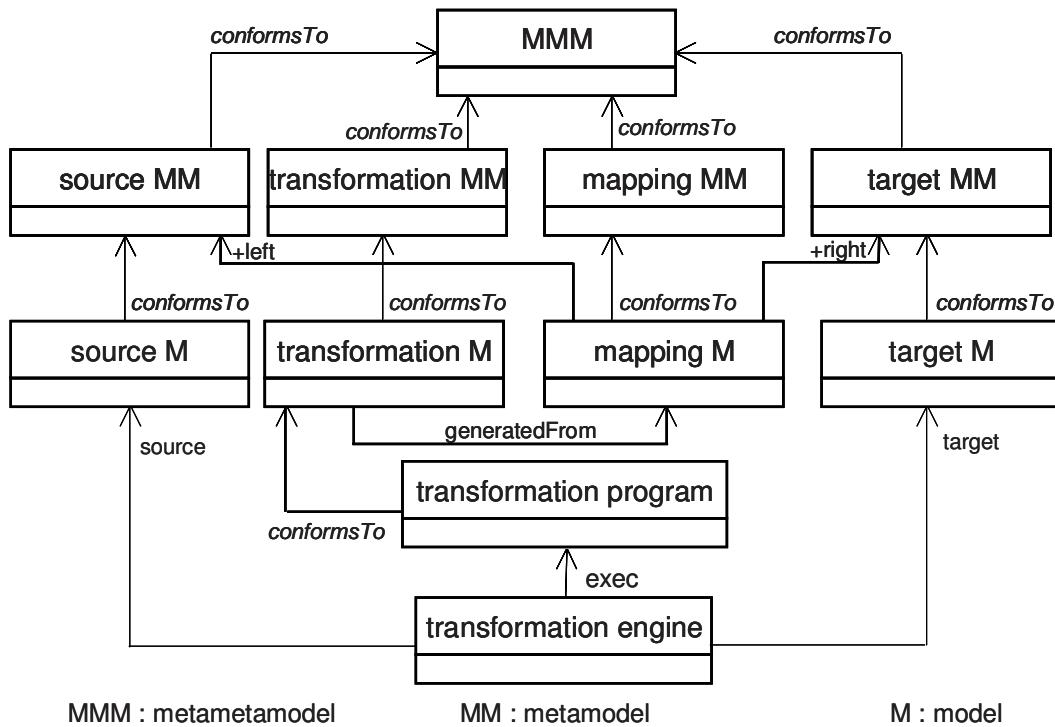


Figura 4.11. Uma proposta de arquitetura tipo para transformação de modelos [Lopes 2005b]

Esta arquitetura tipo para transformação de modelos resulta em uma metodologia que pode ser ilustrada pelo diagrama de atividade apresentado na Figura 4.12.

O processo da Figura 4.12 pode ser organizado nas seguintes etapas:

1. A escolha ou criação de um metamodelo para construir PIMs;
2. A escolha ou criação de um metamodelo para construir PSMs;
3. A especificação de correspondência entre o metamodelo do PIM e o metamodelo do PSM;
4. A geração de definição de transformação a partir da especificação de correspondência;
5. A aplicação da definição de transformação para transformar um PIM em um PSM;
6. A verificação do PSM: a passagem a etapa 7 se estiver incompleto, senão passagem a etapa 8;
7. A intervenção do utilizador para completar o PSM;
8. A geração do código fonte, dos scripts, dos arquivos de implantação.

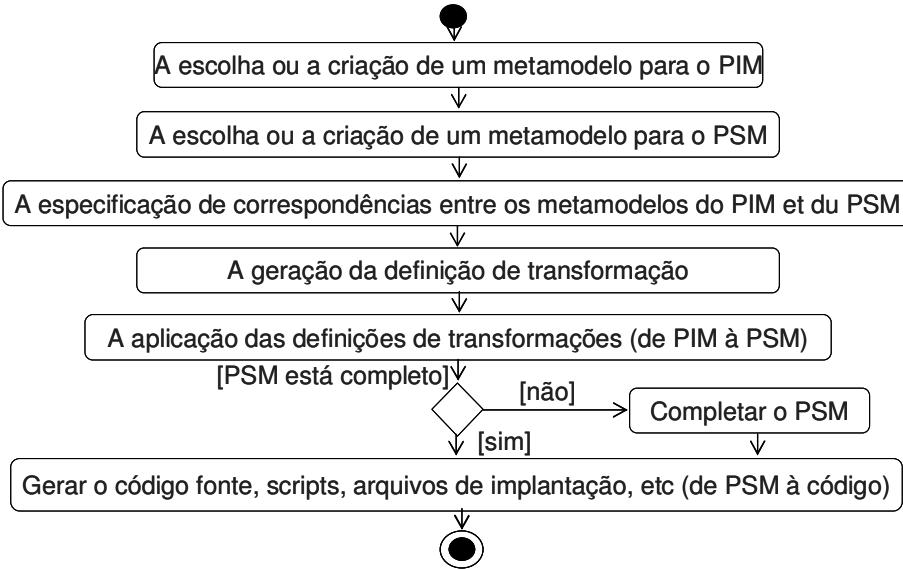


Figura 4.12. Diagrama de atividades para aplicar MDE [Lopes 2005a]

4.6. Exemplos com UML, EDOC, Serviços Web, J2EE e JXTA

A Figura 4.13 ilustra alguns exemplos de utilização da abordagem apresentada na seção 4.4. Inicialmente, o modelo fonte (PIM) deve ser construído conforme um metamodelo como UML ou EDOC. Em seguida, uma transformação modelo-à-modelo passa as informações do modelo fonte (PIM) para um modelo alvo (PSM). Esta transformação modelo-à-modelo é realizada por uma definição de transformação gerada a partir de uma especificação de correspondência entre o metamodelo usado para construir o PIM e o metamodelo usado para construir o PSM. Uma vez que o PSM foi gerado, uma transformação de modelo-à-código é usada para gerar um código fonte ou um arquivo de configuração ou uma implantação ou um documento XML.

Em [Lopes 2005a], os exemplos com UML, EDOC, Serviços Web, J2EE e dotNET são descritos. Em [Sousa 2007], o exemplo com UML e JXTA é ilustrado.

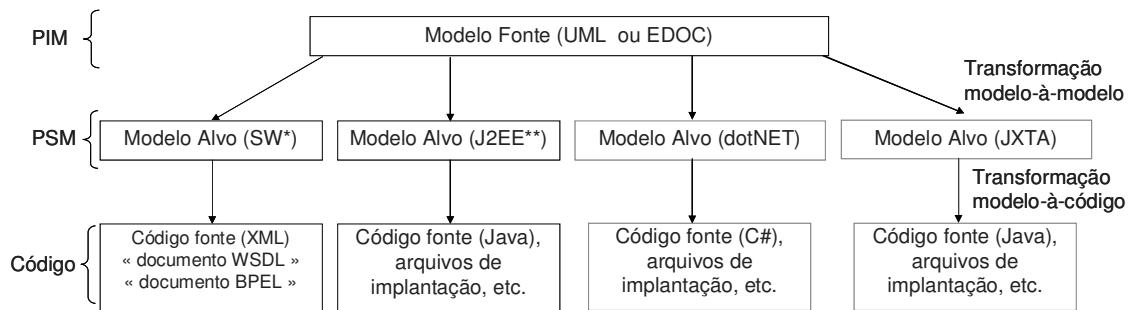


Figura 4.13. Estudos de caso de utilização de MDE [Lopes 2005a][Sousa 2007]

4.6.1. Serviços Web

Para aplicar a abordagem proposta, um metamodelo de serviços Web deve ser fornecido. Dentre as tecnologias que constituem os serviços Web, destacam-se WSDL, SOAP, UDDI e BPEL. A Figura 4.14 apresenta um metamodelo de WSDL (Web Service Description Language).

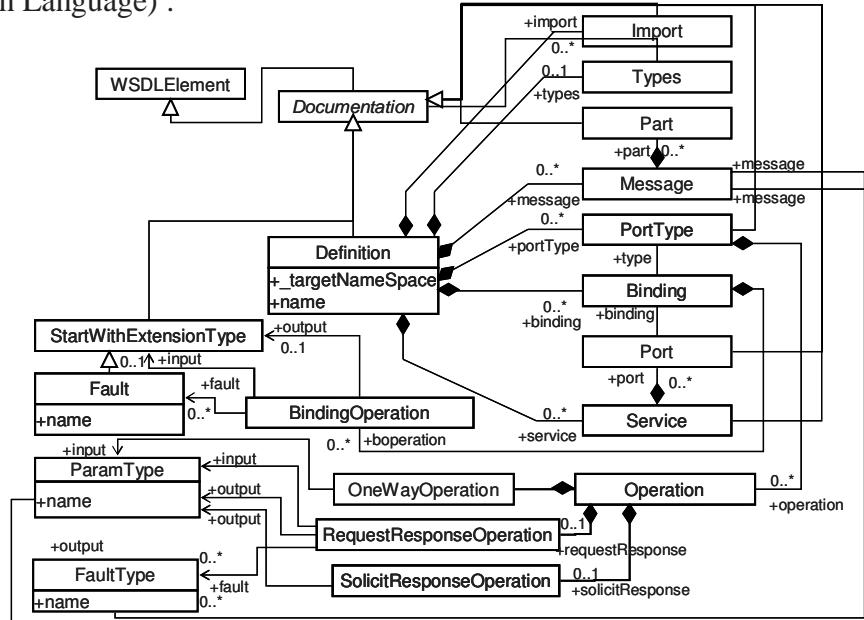


Figura 4.14. Um metamodelo de WSDL [Lopes 2005a]

Uma proposta de correspondência entre o metamodelo de UML e WSDL está descrita pela Figura 4.15. A partir destas correspondências, definições de transformação podem ser geradas.

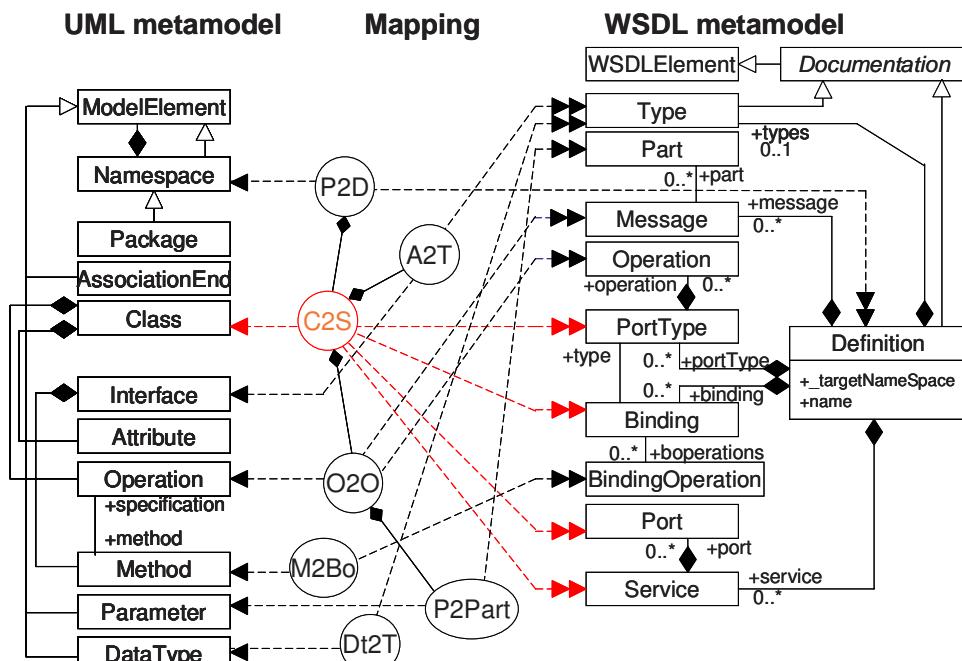


Figura 4.15. Uma especificação de correspondência entre o metamodelo UML e WSDL [Lopes 2005a]

A Figura 4.16 apresenta a definição de transformação gerada a partir da correspondência C2S.

```

module UML2WSDL;
create OUT : WSDL from IN : UML;
rule C2S{
    from c : UML!Class
    to s : WSDL!Service(
        name <- 'Service' + c.name,
        owner <- c.namespace,
        port <- pt
    ),
    pt : WSDL!Port(
        name <- c.name + 'Port',
        binding <- bd,
        documentation <-"t\l<soap:address location=""+ ***
    ),
    bd : WSDL!Binding(
        name <- c.name + 'Binding',
        owner <- c.namespace,
        type <- pType,
        boperations <- c.feature ->select (e|e.ocllsTypeOf(UML!Method)),
        soapBinding <- soapB
    ),
    pType: WSDL!PortType(name <- c.name,***)
}
***
```

Figura 4.16. Uma definição de transformação de UML em WSDL [Lopes 2005a]

4.6.2. J2EE

A plataforma J2EE para serviços Web pode ser constituída da linguagem Java + WSDP (*Web Service Developer Pack*).

A Figura 4.17 apresenta um metamodelo para a linguagem Java.

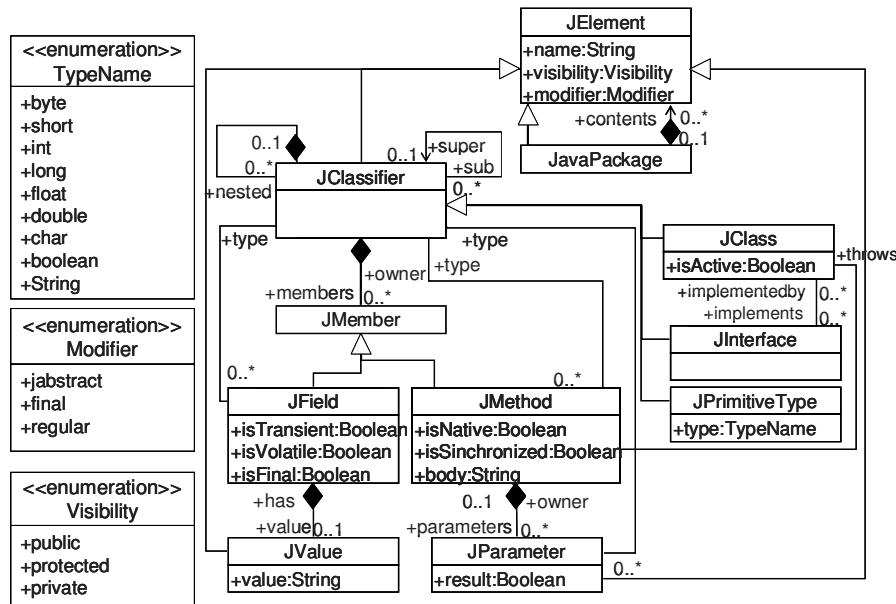


Figura 4.17. Um metamodelo de Java [Lopes 2005a]

A Figura 4.18 ilustra um *template* que mostra como a API de WSDP deve ser utilizada para implementar serviços Web.

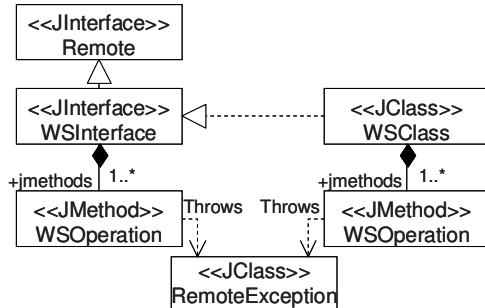


Figura 4.18. Um *template* para usar a API de WSDP [Lopes 2005a]

A Figura 4.19 ilustra uma correspondência entre UML e Java.

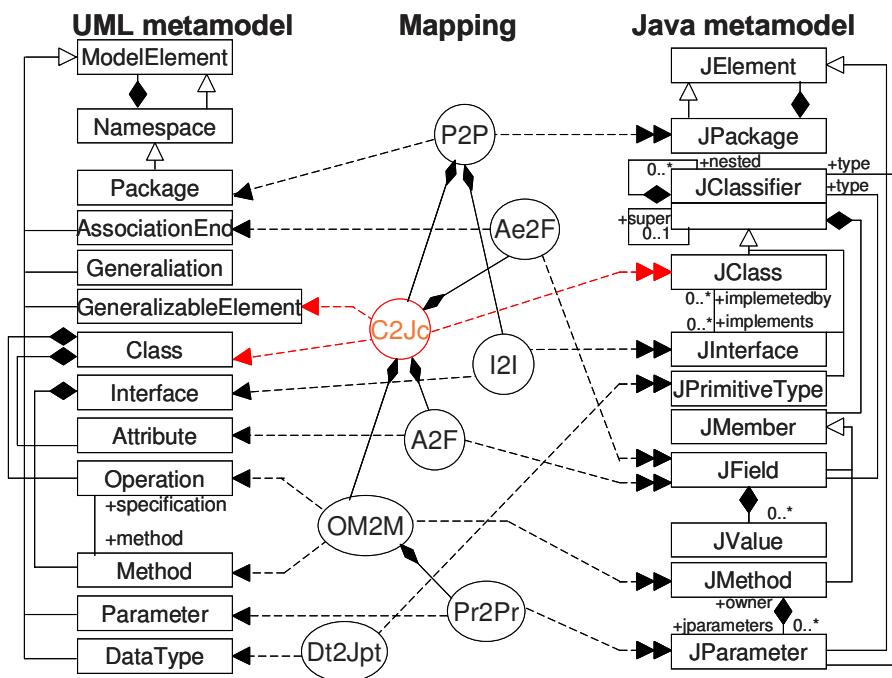


Figura 4.19. Uma especificação de correspondência entre UML e Java [Lopes 2005a]

A Figura 4.20 mostra uma definição de transformação em ATL gerada a partir da especificação de transformação da Figura 4.19.

```

module UML2JAVAM;
create OUT : JAVAM from IN : UML;
rule C2Jc{
    from c : UML!Class
    to jc : JAVAM!JClass(
        name <- c.name,
        visibility <- if c.visibility = #vk_public then
        #public
        else if c.visibility = #vk_private then
        #private
    )
}
  
```

```

else
#protected
endifendif,
isActive <- c.isActive,
jpackage <- c.namespace,
super <- if c.generalization -> select(e|e.parent <> c)->size() > 0 then
c.generalization -> select(e|e.parent <> c)-> first().parent
else JAVAMIJClass
endif,
implements <- c.clientDependency -> select(e|e.stereotype->
exists(e|e.name='realize'))->collect(e|e.supplier),
***  

) } ***

```

Figura 4.20. Uma definição de transformação de UML em Java [Lopes 2005a]

A Figura 4.21 ilustra o refinamento do PSM gerado nesta primeira etapa para agregar as características apresentadas no *template*.

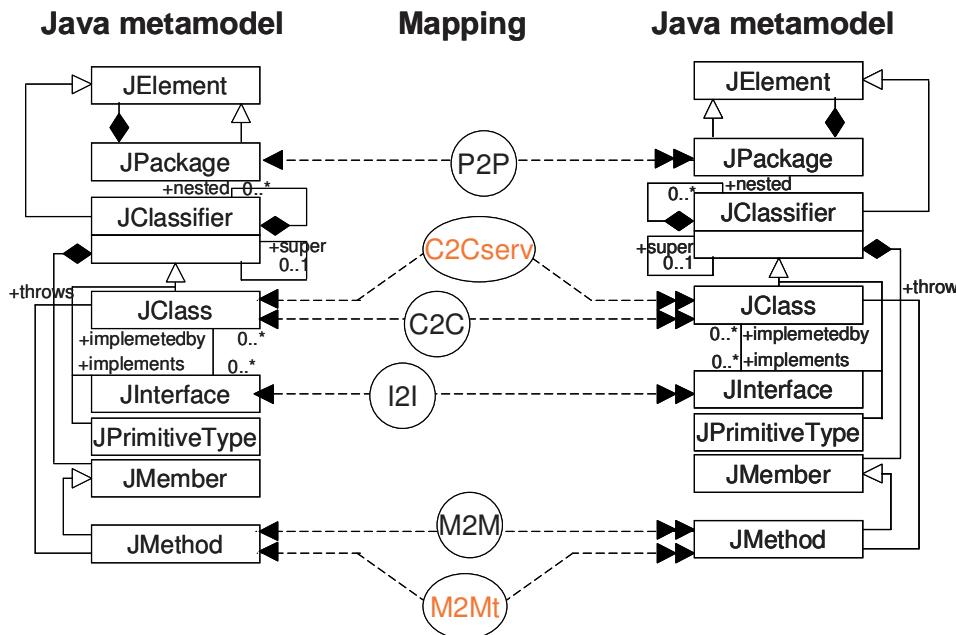


Figura 4.21. Especificação de correspondência para inserção da API de WSDP

A Figura 4.22 mostra uma definição de transformação em ATL que refina o modelo Java introduzindo a plataforma WSDP segundo o *template* da Figura 4.18.

```

module Java2Java;
create OUT : JAVAMR from IN : JAVAM;
uses strings;
rule C2Cserv{
    from c : JAVAM!JClass(
        c.name.startsWith('Service'))
    to jc : JAVAMR!JClass(
        name <- c.name,
        ***  

        implements <- c.implements->asSequence()->including(ITF)
    ),
    ITF : JAVAMR!JInterface(
        name <- 'ITF_'+c.name,

```

```

super <- JAVAMR!JInterface.allInstances()->select(e|e.name='Remote')->first(),
)
}
rule M2Mt{
    from m : JAVAMR!JMethod (m.owner.name.startsWith('Service'))
    to jm : JAVAMR!JMethod(
        name <- m.name,
        throws <- JAVAMR!JClass.allInstances()->
            select(e|e.name='RemoteException')->first()
        ), ***}
***
```

Figura 4.22. Uma definição de transformação de Java em Java+WSDP [Lopes 2005a]

Em [Lopes 2005a], outros detalhes sobre os exemplos usando UML, EDOC, J2EE e dotNEt são fornecidos. Em [Sousa 2007], um exemplo de JXTA no contexto de MDE é ilustrado. Em [Lima 2007], uma abordagem MDA para suportar a difusão e compartilhamento de documentos hipermídia é fornecida.

4.7. Ferramentas

Várias ferramentas suportam uma abordagem MDE. Dentre as ferramentas destacam-se:

- Editores de modelos: permitem a criação de modelos usando uma representação gráfica como Poseidon for UML, Rational Rose, ArgoUML e Omondo for UML;
- Editores de metamodelos: o *plug-in* de EMF possui um editor em forma de árvore para criação de metamodelos conforme eCore;
- Editores, compiladores e *debuggers* de definições de transformação: o ADT (*Atlas Development Toolkit*) fornece um *plug-in* para Eclipse que permite editar, compilar e fazer o *debug* de definições de transformação usando a linguagem ATL;
- Conversores de modelos conforme a UML para metamodelos conforme a MOF: a ferramenta uml2mof do projeto netBeans permite que um modelo conforme a UML seja transformado em um metamodelo conforme a MOF;
- Projeções de repositórios em plataformas específicas: MOF é uma especificação que pode ser projetada em várias plataformas. A projeção de MOF em Java resulta na especificação JMI (*Java MetaData Interchange*) que é implementada pelo MDR (*MetaData Repository*).

Nas seções subsequentes, chama-se a atenção para MT4MDE que permite a criação de modelos de correspondência entre dois metamodelos e a geração da definição de transformação em ATL. SAMT4MDE é uma outra ferramenta que completa MT4MDE com a funcionalidade de determinar as correspondências entre dois metamodelos de forma semi-automática. Além disto, SAMT4MDE permite criar um modelo de correspondência baseado nas correspondências determinadas.

A Figura 4.23 mostra a arquitetura de MT4MDE e de SAMT4MDE [Lopes 2006].

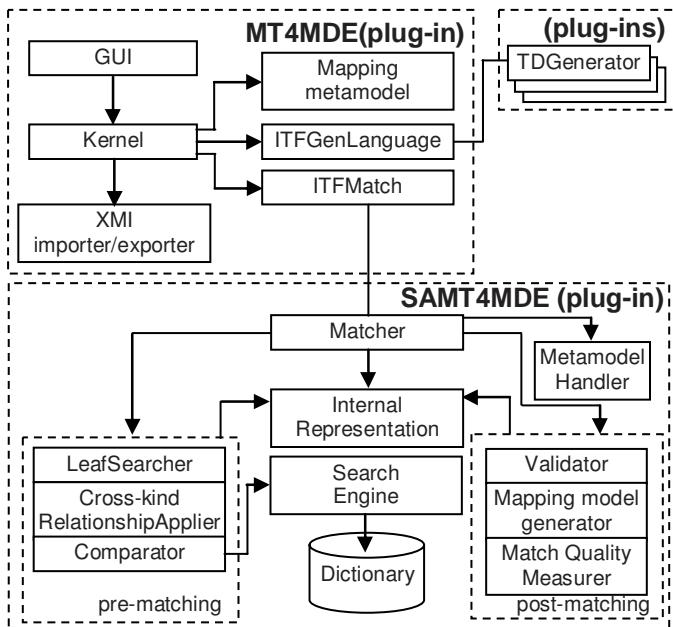


Figura 4.23. MT4MDE e SAMT4MDE: Arquitetura do *plug-in* para Eclipse

MT4MDE é composto de:

- GUI é a interface gráfica com o usuário de MT4MDE;
- Kernel executa todas as funcionalidades básicas de MT4MDE;
- XMI importer/exporter toma um metamodelo no formato XMI e traduz para o mesmo metamodelo conforme a eCore. Este também permite tomar um metamodelo conforme a eCore e traduzi-lo no formato XMI;
- Mapping metamodel é um metamodelo usado para criar modelos de correspondência;
- ITFGenLanguage é uma interface para manipular os geradores de definições de transformação que criam definições de transformação a partir do modelo de correspondência.

A arquitetura de SAMT4MDE é composta de:

- Matcher implementa a interface ITFMatch e coordena a criação de correspondências entre dois metamodelos;
- Internal Representation é uma representação mais adaptada para a criação de correspondências como sendo um conjunto de elementos inter-relacionados;
- MetamodelHandler permite a navegação em metamodelos;
- LeafSearch realiza a busca de metaclasse que são folhas na estrutura do modelo;

- Cross-kind relationship applier aplica as relações definidas em [Pottinger 2003];
- Search Engine busca por similaridades entre os elementos de metamodelos;
- Dictionary é uma base de dados que armazena dicionários de domínios;
- Validator recebe os elementos correspondentes e interage como usuário a fim de validá-los;
- Mapping model generator cria um modelo de correspondência a partir dos elementos validados;
- Match quality measurer avalia os resultados e fornece medidas de qualidade de correspondência.

A Figura 4.24 apresenta MT4MDE e o resultado da execução de SAMT4MDE para determinar as correspondências entre o metamodelo UML e Java.

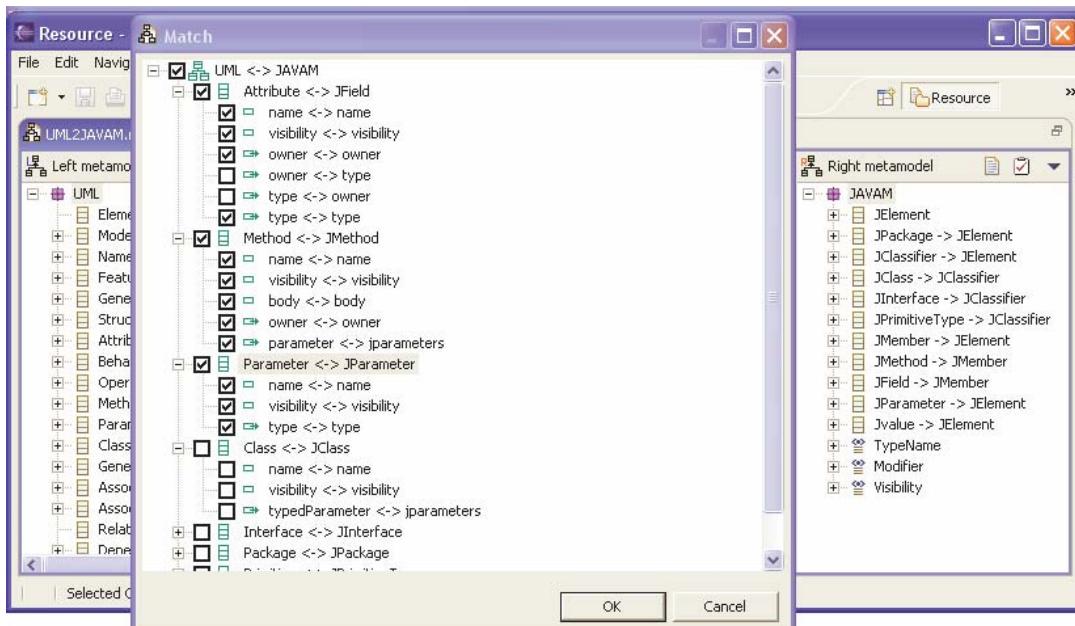


Figura 4.24. MT4MDE e SAMT4MDE: execução

A Figura 4.25 mostra o modelo de correspondência gerado a partir das correspondências encontradas na etapa ilustrada pela Figura 4.24.

A Figura 4.26 mostra a definição de transformação em ATL gerada a partir do modelo de correspondência.

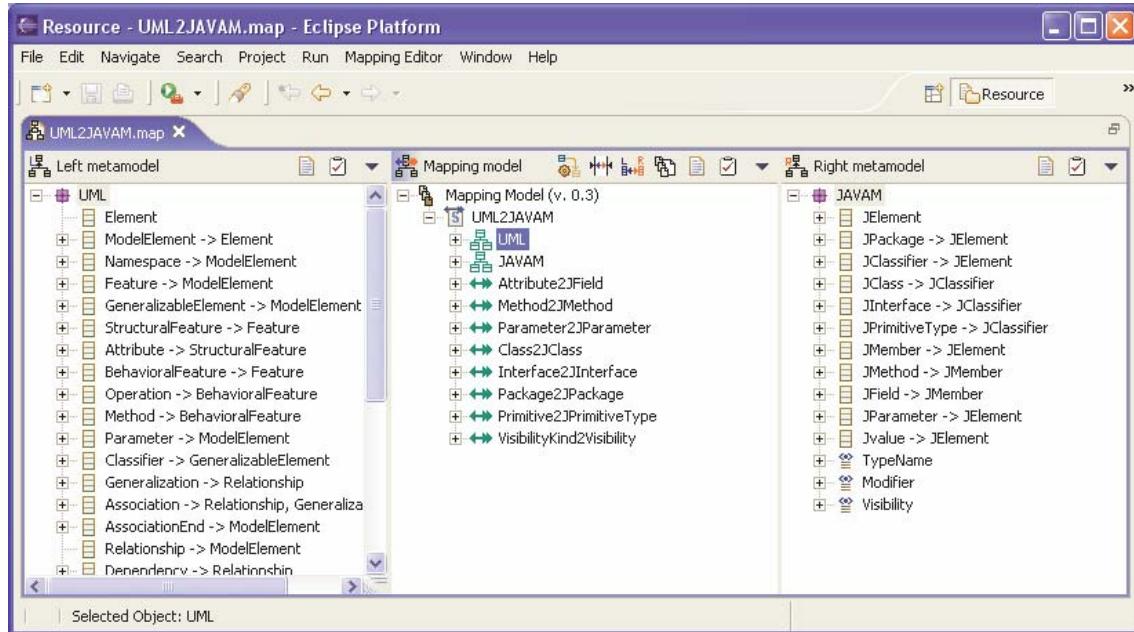


Figura 4.25. Modelo de correspondência gerado a partir das correspondências

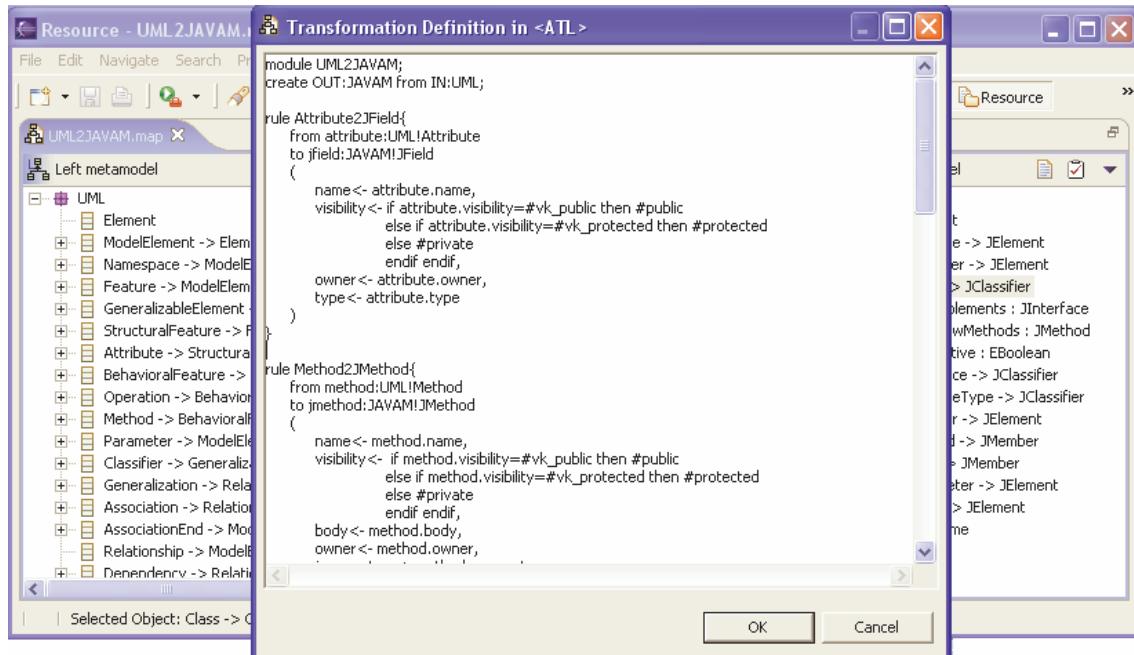


Figura 4.26. Definição de transformação em ATL gerado a partir do modelo de correspondência

4.8. Conclusões

Neste capítulo, MDE foi apresentada destacando-se o desenvolvimento de sistemas de software, alguns exemplos, algumas especificações e algumas ferramentas.

Dentre as abordagens MDE, destacou-se MDA da OMG. MDA é um *framework* constituído por várias especificações dentre elas: UML, MOF, XMI e MOF QVT.

Embora MDA seja uma abordagem promissora, vários mitos ainda perturbam o entendimento sobre este *framework*:

- A independência total de plataforma: a noção de plataforma é relativa e não absoluta. Assim, pode-se ter um modelo independente de uma plataforma, mas dependente de uma outra plataforma;
- A utilização de UML para modelagem de todo tipo de sistema: UML é uma linguagem de modelagem para uso genérico que fornece perfis como mecanismo de extensão. Entretanto, mesmo perfis UML, não são suficientemente capazes de atender todas as necessidades de todos os domínios;
- A geração automática e completa do código a partir de modelos: em regra geral, um código pode ser gerado a partir de um modelo se, e somente se, a informação necessária para gerar o código estiver presente no modelo. Atualmente, não é possível gerar integralmente o código, pois as linguagens de modelagem não são capazes de expressar toda a semântica e a lógica com o nível de detalhes necessários;
- Os problemas da abordagem MDA serão resolvidos com uma linguagem de transformação normalizada: é verdade que a linguagem de transformação normalizada irá impulsionar a difusão de MDA, mas outras questões ainda precisam ser mais estudadas como o *gap* entre metamodelos;
- A utilização de modelos irá simplificar os sistemas complexos: modelos permitem abstrair os detalhes não pertinentes em uma determinada etapa de desenvolvimento de um sistema, mas não podem tornar um sistema complexo em simples. Na realidade, MDA permite que a complexidade seja melhor gerenciada e mascarada do desenvolvedor.

Neste trabalho, propõe-se MT4MDE e SAMT4MDE para suportar MDE.

Enfim, muitos conceitos de MDE ainda não foram implementados por uma ferramenta e outros ainda precisam ser mais aprimorados antes que possam ser usados eficientemente. Além disto, outros conceitos devem ser apresentados para resolver alguns problemas ainda sem solução em MDE como o preenchimento de *gaps* entre metamodelos [Lopes 2005a][Lopes 2005b].

Agradecimentos

O autor agradece ao **Fundo Setorial de Tecnologia da Informação (CT-Info), MCT, CNPq (CT-Info/MCT/CNPq)** pela bolsa de pesquisa e auxílio financeiro fornecidos.

O autor também agradece a **FAPEMA** pelo auxílio financeiro fornecido para participação da primeira Escola Regional de Computação – Ceará, Maranhão e Piauí (ERCEMAPI).

Referências

- Atlas group, LINA e INRIA (2006), ATL: Atlas Transformation Language - ATL User Manual version 0.7, 2006.
- Bézivin, J. e Gerbé, O. (2001), “Towards a Precise Definition of the OMG/MDA Framework”, In Proceedings of the 16th International Conference on Automated Software Engineering, 2001, páginas 273–280.
- Bézivin, J., Dupé, G., Jouault, F., Pitette, G., e Rougui, J. E. (2003), “First Experiments with the ATL Model Transformation Language: Transforming XSLT into XQuery”, 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture, 2003.
- Bézivin, J., Hammoudi, S., Lopes, D. e Jouault, F. (2004), “Applying MDA Approach for Web Service Platform”, 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004), 2004, páginas 58–70.
- Budinsky, F., Steinberg, D., Merks, E. et al (2003), Eclipse Modeling Framework: A Developer's Guide, Addison-Wesley Pub Co, 1st, August, 2003, 704 páginas.
- Favre, J. M. (2004), “Towards a Basic Theory to Model Driven Engineering”, UML 2004 - Workshop in Software Model Engineering (WISME 2004), 2004,
- Fondement, F. e Silaghi, R. (2004), “Defining Model Driven Engineering Processes”, UML 2004 - Workshop in Software Model Engineering (WISME 2004), 2004.
- Frankel, D. S. (2003), Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley and OMG Press, 2003.
- Gerber, A., Lawley, M., Raymond, K., Steel, J. e Wood, A. (2002), “Transformation: The Missing Link of MDA”, First International Conference on Graph Transformation (ICGT2002), 2002.
- Greenfield, J., Short, K., Cook, S., Kent, S. e Crupi, J. (2006), Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley, 2006, 696 páginas.
- JXTA Community (2007), JXTA v2.0 Protocols Specification, Revision 2.5.2, Janeiro 2007, Disponível em <https://jxta-spec.dev.java.net/nonav/JXTAProtocols.html>, Data de acesso: 02/07/2007.
- Kent, S. (2002), “Model Driven Engineering”, Integrated Formal Methods - IFM, 2002, páginas, 286-298.
- Kleppe, A., Warmer, J. e Bast, W. (2003), MDA Explained: The Model Driven Architecture: Practice and Promise, Addison-Wesley, 1st edition, August 2003.
- Kurtev, I., Bézivin, J. e Arksit, M. (2002), “Technological Spaces: An Initial Appraisal”, CoopIS, DOA'2002 Federated Conferences, Industrial track, 2002.
- Lima, B., Sousa, J. G. e Lopes, D. (2007), “Using MDA to Support Hypermedia Document Sharing”, International Conference on Software Engineering Advances (ICSEA 2007), IEEE Press, 2007.

- Lopes, D. (2005a), Study and Applications of the MDA Approach in Web Service Platforms, Ph.D. thesis (written in French), University of Nantes, 2005.
- Lopes, D., Hammoudi, S., Bézivin, J. e Jouault, F. (2005b), “Generating Transformation Definition from Mapping Specification: Application to Web Service Platform”, The 17th Conference on Advanced Information Systems Engineering (CAiSE'05), Springer, LNCS 3520, 2005, páginas 309-325.
- Lopes, D., Hammoudi, S., Abdelouahab, Z. (2006), “Schema Matching in the Context of Model Driven Engineering: From Theory to Practice”, Advances in Systems, Computing Sciences and Software Engineering, Springer, 2006.
- Mellor, S. J., Scott, K., Uhl, A. e Weise, D. (2004), MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley, 1st edition, March 2004.
- Muller, P., Gaertner, N. (2004), Modélisation Objet avec UML, Eyrolles, 2004, 514 páginas.
- NetBean.org (2007), MetaData Repository, Disponível em <http://mdr.netbeans.org>, Data de acesso 02/07/2007.
- OMG (2001), Model Driven Architecture (MDA)- document number ormsc/2001-07-01, July 2001.
- OMG (2003), MDA Guide Version 1.0.1, Document Number: omg/2003-06-01, June, 2003.
- OMG (2004), UML Profile for Enterprise Distributed Object Computing Specification, 2004, Disponível em <http://www.omg.org/technology/documents/formal/edoc.htm>, Data de acesso 02/07/2007.
- OMG (2005a), Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification ptc/05-11-01, 2005.
- OMG (2005b), XML Metadata Interchange (XMI), v2.1, formal/05-09-01, 2005.
- OMG (2006), Meta Object Facility Core Specification version 2.0, formal/2006-01-01, 2006.
- OMG (2007), UML Superstructure, v2.1.1, formal/07-02-03, 2007.
- Patrascoiu, O. (2004), “Mapping EDOC to Web Services using YATL”, 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004), September, 2004, páginas 286-297.
- Pottinger, R. A. e Bernstein, P. A. (2003), “Merging Models Based on Given Correspondences”, Proceedings of the 29th VLDB Conference, 2003, páginas 826-873.
- Sousa, J. G. e Lopes, D. (2007), “Developing Peer-to-Peer Applications with MDA e JXTA”, Advances in Systems, Computing Sciences and Software Engineering, Springer, 2007.
- W3C (2004a), Web Services Architecture (WSA), 2004, Disponível em <http://www.w3.org/TR/ws-arch/>, Data de acesso: 02/07/2007.

Capítulo

5

Introdução aos Padrões de Software

Jerffeson Teixeira de Souza e Tarciane de Castro Andrade

Resumo

Padrões de Software têm se tornado um instrumento fundamental de garantia de qualidade e produtividade no desenvolvimento de projetos de software. Além de fornecer um vocabulário comum para expressar soluções e uma linguagem para relacioná-las, os padrões permitem a criação de uma biblioteca de soluções para ajudar na resolução de problemas recorrentes, incentivando uma cultura de documentação e reuso de “boas práticas” no processo de desenvolvimento de sistemas computacionais. Nesse mini-curso apresentaremos os conceitos fundamentais a cerca dessa tecnologia, abordando desde aspectos básicos como a definição do que são padrões, seu histórico, onde têm sido aplicados, como podem ser classificados e combinados, passando pela discussão dos diversos formatos de padrões existentes atualmente. Finalmente, os participantes serão estimulados a documentar suas próprias experiências com a utilização de padrões com uma apresentação de dicas de escrita de padrões.

Abstract

Software patterns have become a fundamental tool in the attempt to guarantee the quality and productivity of software development projects. In addition to providing a common vocabulary to express solutions and a language to relate them, patterns allow for the development of a solution library which can aid in the resolution of recurring problems, stimulating a culture of documentation and reuse of “best practices” in the development process of computational systems. In this lecture we will discuss the fundamental concepts of this technology, from basic aspects as to how patterns can be defined, their history, where they have been applied, how they can be classified and combined, to the discussion of different patterns formats used nowadays. Finally, the attendees will be stimulated to document their own experiences with patterns with the presentation of tips on how to write good pattern.

5.1. Introdução

Todas as pessoas que lidam com a construção de software, sejam elas analistas, projetistas ou desenvolvedores, certamente já se depararam com um problema que de alguma forma já foi resolvido, de forma que a solução para esse problema poderia beneficiá-las. O reuso consiste na utilização de quaisquer soluções pré-existentes em novos sistemas e tem sido largamente adotado por todos aqueles que produzem software. Porém, tornar uma solução reutilizável e aproveitar todo seu potencial não são tarefas fáceis, pois a reutilização requer a documentação adequada do conhecimento necessário para a resolução do problema.

Diante desse contexto, na última década, padrões de software têm sido estudados e documentados como uma forma promissora de propiciar o reuso, não somente na fase de projeto, mas em todas as outras fases do desenvolvimento do software, como, análise, implementação e testes. Por intermédio dos padrões podem ser documentadas soluções para diferentes tipos de problemas que em quaisquer que sejam seus níveis de abstração, podem ser reusadas por aqueles que se deparam com problemas similares.

Este mini-curso tem como objetivo introduzir os conceitos fundamentais de padrões de software para todos aqueles que desejam conhecer sobre essa forma de reuso, ao discutir em detalhes os conceitos, classificações e formatos de padrões, além de fornecer dicas de como documentar bons padrões entre outras informações relevantes.

O mini-curso está organizado da seguinte forma: na seção 5.1.1 são apresentados os conceitos de padrões de forma geral, padrões de software em especial e os benefícios e consequências negativas de utilizá-los. Na seção 5.1.2 encontram-se a história e evolução dos padrões de software, como eles surgiram e como eles continuam sendo disseminados. Na seção 5.1.3 são mostrados os parâmetros de qualidade de um padrão, o quê o faz ser considerado um bom padrão. Na seção 5.1.4 discutem-se os tipos de padrões com suas respectivas áreas de atuação. Na seção 5.1.5 são apresentadas as diversas formas de agrupamento de padrões. Já na seção 5.2 encontram-se os mais variados e conhecidos formatos de documentação de padrões. Adicionalmente, na seção 5.3 é mostrado como a comunidade de padrões contribui para disseminar a cultura de documentar e utilizar padrões. Na seção 5.4 são fornecidas diretrizes de como escrever padrões. E, finalmente, nas seções 5.5 e 5.6 encontram-se, respectivamente, as considerações finais a respeito deste trabalho e os agradecimentos.

5.1.1. O que são Padrões?

Os primeiros conceitos de padrões (seção 5.1.2) surgiram na área de arquitetura, por meio do trabalho de Christopher Alexander na década de 70 [Alexander 1977], [Alexander et. al. 1979]. Ele criou as primeiras definições para os termos padrão e linguagem de padrões, usadas hoje na área de software. De acordo com Alexander, padrões podem ser definidos da seguinte forma:

“Um padrão é uma entidade que descreve um problema que ocorre repetidamente em um ambiente e então descreve a essência de uma solução para este problema, de tal forma que você possa usar essa solução milhões de vezes, sem nunca utilizá-la do mesmo modo”, [Alexander et. al. 1977].

Ou ainda como:

“Como um elemento no mundo, cada padrão é uma relação entre um certo contexto, um certo sistema de forças que ocorrem repetidamente naquele contexto, e uma certa configuração espacial que permite que estas forças se solucionem por si só”, [Alexander 1979].

Como podemos observar nas definições de Alexander, o conceito de padrões não é aplicável apenas à arquitetura, podendo ser usado ainda para documentar soluções recorrentes para problemas na área de desenvolvimento de software.

Existem inúmeras definições para o conceito de padrões de software, aqui vamos citar algumas:

“Um padrão é uma solução para um problema em um contexto”, [Coplien 1994].

“Um padrão é um pedaço da literatura que descreve um problema de projeto e uma solução geral para o problema em um contexto particular”, [Coplien 2000].

“Cada padrão é uma regra de três partes, que expressa a relação entre um certo contexto, um certo sistema de forças que ocorre repetidamente naquele contexto, e certa configuração de software que permite solucionar as forças”, [Gabriel 1998].

De fato, todas as definições de padrões levam para o mesmo objetivo: documentar uma solução recorrente para um problema que ocorre em um determinado contexto. Vale destacar que os padrões são documentações de boas soluções, ou seja, soluções realmente comprovadas pela experiência prática.

A disseminação da utilização de padrões de software (seção 5.1.2), em especial, padrões de projeto, iniciou através dos padrões publicados no livro do GoF¹ [Gamma et. al.

¹ Gang of Four: nome do grupo formado pelos quatro autores do livro.

1995] que retratam diversos tipos de soluções para problemas de projeto em sistemas orientados a objetos. Segundo GoF, os padrões de projeto possuem a seguinte definição:

“Os padrões de projeto são descrições de objetos que se comunicam e classes que são customizadas para resolver um problema de projeto genérico em um contexto específico”, [Gamma et. al. 1995].

Dessa forma, o uso de padrões de software, em geral, permite não apenas a reutilização de soluções já existentes e documentadas por especialistas, mas também a captura de estruturas, práticas e técnicas em um determinado domínio. A seguir, listaremos outros benefícios da utilização de padrões, tais como [Vlissides 1990], [Schmidt 2000], [Andrade et al 2003]:

- Melhorar a comunicação entre os engenheiros de software ao oferecer um vocabulário comum, conciso e compartilhado;
- Otimizar o tempo ao concentrar esforço de desenvolvimento em aspectos inéditos e particulares do sistema;
- Facilitar o entendimento de sistemas e a evolução do código. Quando utilizados desde o início do desenvolvimento, padrões podem gerar menor retrabalho nas etapas mais avançadas do projeto;
- Melhorar a distribuição de responsabilidades e consequente diminuição dos efeitos colaterais causados por mudanças;
- Facilitar a reestruturação de um sistema.

No caso especial de padrões de projetos, podemos citar as seguintes vantagens:

- Independência, ao permitir a adoção de soluções abstratas e independentes de implementação;
- Possibilitar a criação de frameworks contendo padrões de projeto já implementados a fim de facilitar o reuso dos padrões;
- Modularidade, ao permitir maior autonomia funcional entre os módulos com consequente diminuição da complexidade por quebrar o problema em problemas menores.

Segundo Coplien [Coplien 2000], a utilização de padrões no desenvolvimento de software contribui indiretamente para:

- Maior produtividade, ao permitir a disponibilidade de soluções prontas, evitando o retrabalho;
- Diminuição do tempo de desenvolvimento, ao possibilitar o reuso das soluções e consequente diminuição do custo do software ao cliente;
- Satisfação do cliente derivado do fator anterior.

Ainda segundo Coplien, a utilização de padrões, por si só, não oferece nenhum desses benefícios. Eles são um instrumento que aumenta a habilidade humana e o poder de aplicação de tecnologias, gerência de boas práticas e oportunidades [Coplien 2000].

Contudo, o uso de padrões de projeto pode ocasionar consequências negativas ao projeto, como as citadas em [Andrade 2003]:

- Menor eficiência, ao exigir, em alguns casos, a incorporação de novas classes ou camadas de aplicação, aumentando a quantidade de métodos executados, tornando o sistema mais lento. Assim, se a prioridade em um determinado sistema é o alto desempenho, deve-se tomar cuidado ao optar por utilizar certos padrões;
- Menor legibilidade/manutenibilidade ao permitir, por exemplo, o aumento de níveis hierárquicos de classes, podendo tornar o código do sistema menos legível e consequentemente, elevando o custo de manutenção do mesmo, principalmente para desenvolvedores que não tenham conhecimento técnico sobre os padrões aplicados.

Em relação aos padrões de software em geral, uma desvantagem da sua utilização seria a falta de motivação da equipe devido à ausência de conhecimento prévio sobre padrões, causando resistência ao seu uso.

5.1.2. História e Evolução dos Padrões de Software

Os primeiros conceitos de padrões surgiram no final da década de 70 por meio do trabalho do arquiteto Christopher Alexander. Alexander documentou no livro “The Timeless Way of Building” [Alexander 1979] cerca de 250 padrões para construção de edifícios, jardins, praças, casas, entre outros.

Padrões de software, por sua vez, têm uma história bem mais recente (início da década de 90). Enquanto estudante de graduação na Universidade de Oregon, o americano Kent Beck foi apresentado ao trabalho do Alexander por estudantes do curso de arquitetura. Anos depois, já trabalhando como consultor na empresa Tektronix, Kent se deparou mais uma vez com o livro do Alexander e só então percebeu que as idéias do arquiteto poderiam ser aplicadas à área de engenharia de software visto sua semelhança com projetos arquiteturais. Em 1987, Kent e Ward Cunningham estavam trabalhando em um projeto de interface gráfica quando decidiram aplicar as idéias de Alexander, desenvolvendo uma pequena linguagem com cinco padrões com o objetivo de otimizar o uso de Smalltalk nesse tipo de projeto. Esses padrões foram então usados pelos próprios usuários do sistema no projeto daquela interface. Tendo em vista os ótimos resultados obtidos pela aplicação desses padrões, os dois autores decidiram apresentar essa experiência de sucesso em um artigo intitulado “Using Pattern Languages for Object Oriented Programs” [Beck et. al. 1987] que foi publicado no congresso Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), realizado em outubro de 1987, em Orlando, Estados Unidos.

Em 1988, Jim Coplien iniciou o catálogo de padrões específicos para a linguagem C++, chamados por ele de idioma (padrões de programação específicos para a linguagem em questão). O resultado deste trabalho foi publicado como livro em setembro de 1991, intitulado “Advanced C++ Programming Styles an Idioms” [Coplien 1991]. Peter Coad

[Coad 1992], em paralelo, documentou e publicou sete padrões de análise de orientação a objetos no jornal *Communications of the ACM* intitulado de “Object-oriented patterns”.

Ao mesmo tempo, o então estudante de Doutorado da Universidade de Zurich, Erich Gamma, trabalhava na sua tese de doutorado com o projeto de sistemas orientados a objeto desenvolvendo o framework ET++. Em 1990, Erich Gamma e Richard Helm, se encontraram pela primeira vez na conferência conjunta European Conference on Object-Oriented Programming/Object-Oriented Programming, Systems, Languages, and Applications (ECOOP/OOPSLA), em Ottawa, Canadá, onde aconteceu uma vasta discussão, liderada pelo professor Bruce Anderson, sobre a aplicabilidade de padrões no projeto de software. Nesse momento, Erich e Richard descobriram interesses semelhantes quanto à possível aplicabilidade dessa técnica na criação de sistemas orientados a objeto reusáveis. Um ano depois, os dois tiveram outra oportunidade de discutirem suas idéias quando da realização da OOPSLA’91. Novos adeptos como Ralph Johnson e John Vlissides estavam lá e se juntaram aos dois para escrita dos primeiros padrões de projeto. Em 1992, nesse mesmo evento, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides se uniram para criar o grupo conhecido como Gang of Four (GoF). O fruto do trabalho do GoF foi primeiramente publicado no ECOOP’93 com o título “Design Patterns: Abstraction and Reuse of Object-Oriented Design” [Gamma et. al. 1993]. Os padrões publicados nessa conferência serviram como base para o mais consagrado livro da área de padrões de software, “Design Patterns: Elements of Reusable Object-Oriented Software” [Gamma et. al. 1995], publicado em 1995 pelos mesmos autores.

Ainda em 1993, um grupo de interessados na recém surgida área se reuniu no estado do Colorado, Estados Unidos. Surgia desse encontro o grupo Hillside (www.hillside.net), com o intuito de organizar e alavancar a área de padrões de software. Estavam entre os participantes desse encontro Ward Cunningham, Kent Beck, Grady Booch, Ralph Johnson, James O. Coplien e outros. No ano seguinte, o grupo Hillside se reuniria novamente, mas agora para organizar o primeiro congresso da área de padrões de software, o PLoP (Pattern Languages of Programs).

Esta primeira conferência da série PLoP, “Pattern Languages of Programs” aconteceu na cidade de Monticello, no estado de Illinois, EUA. Desde então vem se repetindo anualmente nesse mesmo local. A partir de 1996, surgiu a edição européia do PLoP, chamada EuroPLoP, seguida da ChilliPLoP, no Arizona, EUA, em 1998. Em 2000, surgiu a série KoalaPLoP na Austrália, em 2001 o MensorePLoP no Japão e em 2002, o VikingPLoP na Escandinávia. A organização geral de todos estes eventos é supervisionada pelo grupo Hillside. Acompanhando essa tendência de disseminação dos PLoPs, a primeira versão latino-americana, denominada SugarLoafPLoP, foi organizada em 2001 como um evento satélite do Simpósio Brasileiro de Engenharia de Software (SBES). Nos anos seguintes, o SugarLoafPLoP ganhou espaço e se tornou um evento isolado. Desde então, a comunidade latino-americana de padrões se reúne anualmente nessa conferência para discutir seus trabalhos, estando atualmente na 6^a edição.

5.1.3. Parâmetros de Qualidade

Segundo Alexander, o objetivo da arquitetura é satisfazer as necessidades humanas. A Qualidade sem Nome (“The Quality Without a Name”) é um conceito pregado por Alexander [Alexander 1979], cuja idéia principal está no fato de que todo padrão deve ter certas características difíceis de serem nomeadas que o fazem satisfazer às necessidades do ser humano. Fazendo uma analogia à qualidade subjetiva de Alexander, padrões de software também precisam satisfazer às necessidades daqueles que constroem software e para tal necessitam possuir qualidade. Mas o que faz com que um padrão seja realmente bom, com qualidade?

As premissas básicas que tornam um padrão bom, de acordo com Coplien [Coplien 2000], são:

- **Autenticidade.** Os padrões capturam soluções comprovadas na prática por meio de usos conhecidos, ao invés de teorias ou postulados. A solução do padrão deve dizer ao leitor o quê fazer de forma específica e concreta. Um bom padrão descreve exemplos reais, que possam ser usados facilmente na prática;
- **O padrão resolve o problema.** Os padrões capturam soluções para o problema em um determinado contexto. Um bom padrão deve ajudar o leitor a focar no problema ao extrair as “Forças” (seção 5.2) importantes que apresentam elementos favoráveis e desfavoráveis à adoção da solução para o problema. Essas forças devem ser balanceadas pela solução, de forma que o problema seja resolvido da melhor maneira possível diante do contexto existente;
- **A solução do padrão não é óbvia.** O poder dos padrões está na documentação de soluções que levaram esforço e tempo para serem concebidas e que podem ser reusadas por pessoas inexperientes quando deparadas com o mesmo problema. Um bom padrão não possui soluções simplistas e triviais;
- **O padrão descreve um relacionamento.** Um bom padrão não descreve apenas os módulos que compõem a solução, mas descreve também os seus relacionamentos através de estruturas dinâmicas e mecanismos de funcionamento da solução;
- **O padrão possui um componente humano.** Um bom padrão deve ser elegante e útil para prover ao ser humano o conforto e a facilidade de compreendê-lo e utilizá-lo de forma correta e estimulante.

É importante lembrar que nem toda solução ou boa prática constitui um padrão. Mesmo que tenha algo que se deseja documentar com todos os requisitos de um padrão, ele não deve ser considerado um padrão até que seja verificada a sua recorrência preferivelmente em no mínimo três casos existentes. A esse fato, dá-se o nome de Teste de Paternidade do padrão [Appleton 1997]. Porém, a recorrência por si só não é uma característica tão importante para um padrão, além disso, é necessário mostrar que o padrão é adequado e útil para o seu propósito. Enfim, para que um padrão tenha qualidade, todos esses fatores devem ser levados em consideração. Na dúvida, deve-se documentar o padrão e submetê-lo à análise da comunidade de padrões (seção 5.1.2 e seção 5.3).

5.1.4. Áreas de Aplicação e Tipos de Padrões

Apesar da disseminação dos tão conhecidos padrões de projeto GoF, os padrões de software estão presentes em todas as áreas e domínios específicos da computação, incluindo: sistema operacional, segurança, processos de software, análise, projeto, desenvolvimento, entre outros. Para quaisquer que sejam as áreas de atuação, um padrão de software pode ser documentado e reutilizado por aqueles do domínio em questão. Para isso, é necessária a existência de uma boa solução recorrente para um determinado problema do domínio, conforme citado nas seções 5.1.1 e 5.1.3.

Existem inúmeros tipos de padrões com diversas áreas de atuação, aqui vamos listar alguns tipos de padrões mais conhecidos:

- **Padrões Arquiteturais:** padrões que expressam a organização estrutural fundamental ou esquema para o software. Eles provêem o conjunto de subsistemas pré-definidos, especificam suas responsabilidades, e incluem regras e guias para organizar os relacionamentos entre as partes. Diversos tipos de padrões arquiteturais podem ser encontrados, por exemplo, no livro do POSA [Buschmann et. al 1996];
- **Padrões de Projeto:** padrões que provêem soluções para refinar os subsistemas ou componentes do software ao nível de projeto, bem como o relacionamento entre eles. Descrevem aspectos do projeto do software, como objetos, classes, herança, agregação, entre outros. Diversos tipos de padrões de projeto podem ser encontrados no livro GoF [Gamma et.al 1995];
- **Padrões de Análise:** padrões que descrevem soluções para problemas da fase análise de sistemas e são padrões específicos de domínio. Alguns exemplos de padrões de análise podem ser encontrados no livro de Fowler [Fowler 1997];
- **Padrões de Programação:** padrões que descrevem soluções de programação particulares de determinada linguagem ou regras gerais de programação. Alguns padrões de programação podem ser encontrados no livro “Core J2EE Patterns” [Alur et. al 2003];
- **Padrões Organizacionais:** padrões voltados para soluções de crescimento, melhoria ou maturidade das organizações, a fim de estruturá-las da melhor forma. Padrões organizacionais podem ser encontrados no livro “Organizational Patterns of Agile Software Development” [Coplien and Harrison 2004];
- **Padrões de Processo:** padrões que definem soluções para os problemas nos processos da engenharia de software: desenvolvimento, controle de configuração, testes, etc. Alguns padrões de processo podem ser encontrados no livro “Process Patterns: Building Large-Scale Systems Using Object Technology”, [Ambler 1998] e no artigo de Coplien [Coplien 1994];
- **Anti-padrões:** os anti-padrões, ao contrário dos padrões, descrevem uma “lição aprendida” a cerca de uma solução ruim para um determinado problema [Appleton 1997]. Eles descrevem como escapar de uma situação ruim e como, a partir de tal situação, proceder para alcançar uma boa solução. Alguns anti-padrões estão

documentados no livro intitulado “AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis” [Brown et. al. 1998].

5.1.5. Coletâneas de Padrões

Os padrões de software, como citados anteriormente, tratam de resolver problemas de software através de soluções já existentes, possibilitando assim, o reuso dessas soluções. Os padrões quando da sua aplicação podem gerar outros problemas que necessitam de outros padrões para ser resolvidos, e assim por diante. Em geral os padrões não funcionam isoladamente: um padrão pode depender da aplicação de outro padrão anterior, ou pode conter como parte da sua solução outro padrão, ou ainda pode ser uma combinação de vários padrões. De forma geral, um padrão e seus padrões relacionados descrevem soluções similares, que se completam ou se encaixam dependendo das influências envolvidas.

Portanto, os padrões podem ser agrupados em categorias que permitem claramente expor, se existirem, os seus relacionamentos, dependências ou combinações. Isso pode ser feito por meio de coleções de padrões, catálogos de padrões, sistemas de padrões ou linguagens de padrões.

- **Coleção de Padrões:** é uma coletânea de padrões onde os padrões não possuem nem um tipo de ligação, nem relacionamento semântico entre si, funcionam completamente isolados e independentes. Os padrões contidos em uma coleção podem ter sido apresentados no mesmo congresso ou documentados pelo mesmo autor. Por exemplo, uma coleção de padrões pode ser encontrada nos anais da conferência SugarLoafPLoP 2007;
- **Catálogo de Padrões:** é uma coletânea de padrões com fraco relacionamento entre eles ou relacionados informalmente. Em geral, agrupa os padrões de forma abrangente e pode possuir referências cruzadas entre eles. Um catálogo de padrões atribui um pouco de estrutura e organização a uma coleção de padrões, mas usualmente não vai além de mostrar apenas relações mais superficiais (se, de fato mostrar alguma delas) [Appleton 1997]. Como exemplo, os padrões descritos no GoF [Gamma et. al. 1995] estão agrupados no formato de catálogo de padrões;
- **Sistema de Padrões:** é um conjunto coeso de padrões sobre o mesmo tema. Um sistema de padrões possui padrões relacionados entre si e que juntos apóiam a construção e evolução de temas completos. Um sistema de padrões descreve todas as relações entre os padrões e como eles podem ser combinados ou conectados a outros padrões do sistema para resolver problemas mais complexos [Appleton 1997]. Como exemplo, os padrões descritos no POSA [Buschmann et. al. 1996] estão agrupados no formato de sistema de padrões;
- **Linguagem de Padrões:** é uma coleção estruturada de padrões que se apóiam uns nos outros para transformar requisitos e restrições numa arquitetura [Coplien 2000]. Uma linguagem de padrões é uma forma de subdividir um problema geral e sua solução complexa em problemas menores relacionados e suas respectivas soluções. Uma linguagem de padrões possui um contexto comum compartilhado entre todos os padrões pertencentes a ela. Os padrões podem ser usados isoladamente ou com

demais padrões relacionados na linguagem, porém, cada um deles resolve seu próprio problema. Como exemplo, o artigo “MetaPatterns: A Pattern Language for Pattern Writing” [Meszaros and Doble 1996] descrevem uma linguagem de padrões para auxiliar na escrita de padrões.

5.2. Formato de Padrões

Um padrão descreve uma solução para um problema em um contexto específico. Para que possa ser facilmente encontrado e utilizado, necessita ser documentado de forma adequada.

Os padrões são documentados textualmente e organizados em seções ou componentes que definem o *template* ou formato do padrão. Não existe um consenso dentro da comunidade de padrões sobre qual a melhor forma de documentar um padrão. Diferentes autores adotam diferentes estilos para formalizar seus padrões. Alguns preferem ser mais textuais e menos estruturados, enquanto outros preferem o contrário. Não existe um formato adequado para todas as ocasiões, mas os formatos já existentes podem e devem ser usados como base.

A seguir serão descritos os formatos de padrões mais conhecidos e utilizados por pesquisadores da área. Apesar de cada formato ter sido criado para atender a padrões de áreas específicas, diferentes formatos podem, a princípio, ser utilizados na documentação de qualquer padrão. Cabe então a cada autor utilizar o formato que mais se adeque ao seu propósito, ou ainda, definir o seu próprio formato. No entanto, segundo [Meszaros et. al. 1996], a documentação dos elementos “nome”, “contexto”, “problema”, “forças” e “solução” são obrigatórios para a documentação de um padrão, mesmo que não estejam explicitamente visíveis.

5.2.1. Formato Alexandriano

O arquiteto Alexander, por meio do trabalho intitulado “The Timeless Way of Building” [Alexander 1979], foi o primeiro a introduzir o conceito de padrões, a criar um formato comum para todos os padrões e a documentá-los utilizando este formato.

Para Alexander, cada padrão é uma regra de três partes que expressa uma relação entre o contexto, o problema (seguido do sistema de forças que ocorre repetidamente neste contexto) e a solução, que permite que estas forças se resolvam. As seções do formato Alexandriano não são explicitamente delimitadas, a única exceção é a separação sintática clara através da palavra *Therefore* que precede a solução do padrão. Os demais elementos que compõem o padrão, como contexto, problema e forças, também estão presentes, mas são simplesmente destacados no corpo geral do texto.

Para documentar no formato Alexandriano o autor deve estruturar o padrão com informações na seguinte ordem:

- O nome do padrão seguido de uma indicação de confiança correspondente a uma, duas ou três estrelas. Padrões com três estrelas são aqueles em que o autor possui maior grau de segurança e embasamento. Menos estrelas, portanto, significam menor confiança;
- Uma figura que expressa um exemplo do padrão descrito;
- Um parágrafo introdutório, o qual descreve a situação em que o padrão pode ser aplicado e enumera eventuais padrões que possivelmente já foram aplicados anteriormente;

- Três diamantes (◊◊◊) para marcar o início do problema atacado pelo padrão;
- Descrição da essência do problema, destacado em negrito, em uma ou duas sentenças;
- Descrição detalhada e fundamentada do problema, juntamente com sua motivação e com as forças que influenciam na resolução do problema declarado;
- A palavra “Therefore.” para separar o problema da solução;
- A descrição da solução do padrão, também destacada em negrito. Esta solução deve ser documentada sempre no formato de instruções, para que o usuário saiba o que é necessário fazer para construir o padrão;
- Um diagrama que ilustra a solução, com comentários para indicar os principais componentes;
- Outros três diamantes (◊◊◊) para mostrar a finalização do corpo principal do padrão;
- E, finalmente, um parágrafo descrevendo os outros padrões relacionados da linguagem.

5.2.2. Formato Canônico

O formato canônico é assim chamado por ser uma forma de compor padrões, ou seja, estruturá-los em seções, onde cada seção é composta por um cabeçalho e um texto contendo a sua descrição. Ao contrário do formato Alexandriano, o formato Canônico possui seções explicitamente delimitadas.

Para documentar no formato Canônico, o autor deve estruturar o padrão com as seguintes seções:

- **Nome:** palavra ou frase que nomeie o padrão. O nome do padrão deve ser significativo no sentido de refletir de forma sucinta a sua essência;
- **Problema:** descreve o problema que o padrão resolve diante de um certo contexto;
- **Contexto:** circunstâncias iniciais dentro das quais o problema emerge e para as quais a solução é desejável. É uma situação que ocorre antes da aplicação do padrão;
- **Forças:** descrição dos aspectos relevantes para o problema e que influenciam na escolha de uma solução para esse problema;
- **Solução:** instruções de como solucionar o problema;
- **Exemplo:** descreve um exemplo de aplicação do padrão. Ilustra como o padrão pode ser utilizado;
- **Contexto Resultante:** descreve as circunstâncias que ocorrem após a aplicação do padrão, podendo ter uma subseção chamada de “Conseqüências” (tanto boas, quanto ruins);

- **Racional:** mostra porque a solução é uma boa solução e como as “Forças” foram priorizadas. Esta seção afirma realmente como o padrão funciona, porque funciona e porque ele é bom;
- **Padrões Relacionados:** descreve os relacionamentos desse padrão com outros dentro da mesma linguagem ou fora dela;
- **Usos Conhecidos:** descreve ocorrências conhecidas do padrão e sua aplicação em sistemas existentes. Esta seção ajuda a validar o padrão, verificando se ele é realmente uma solução provada para um problema recorrente.

5.2.3. Formato GoF

O formato GoF, “Gang of Four”, presente nos padrões de projeto para orientação a objetos do livro “Design Patterns: Elements of Reusable Object-Oriented Software” [Gamma et. al 1995], utiliza seções bem definidas e destacadas no texto. Como é utilizado para documentação de padrões relacionados ao projeto de sistemas Orientados a Objetos, o formato GoF está voltado para padrões que necessitam descrever as soluções através de diagramas de classes e códigos de implementação, tornando-o bastante extenso. Embora tenha este direcionamento, o autor pode usá-lo, com algumas adaptações, para formalizar outros padrões de software.

Para documentar no formato GoF, o autor deve estruturar o padrão com as seções abaixo:

- **Nome e Classificação:** o nome do padrão deve conter a sua essência, deve ser usado para descrever o problema e a solução em uma ou duas palavras. A classificação do padrão pode ser: padrões de criação (estão relacionados à criação de objetos), padrões estruturais (tratam das associações entre classes e objetos) ou padrões comportamentais (tratam das interações e divisões de responsabilidades entre as classes ou objetos);
- **Intenção:** apresenta um resumo do que o padrão faz, levando em conta quais são as abordagens do projeto e do problema que são tratadas por ele;
- **Também Conhecido Como:** expõe outros nomes conhecidos para o padrão, se existirem;
- **Motivação:** descreve um cenário que ilustra um problema de projeto e como as estruturas de classes e objetos no padrão resolvem o problema. O cenário ajuda a entender a descrição mais abstrata do padrão que vem a seguir.
- **Aplicabilidade:** descreve quais as situações que o padrão pode ser aplicado. Aqui, podem ser incluídos exemplos de projetos onde o padrão pode ser apropriado, além de dicas de como reconhecer tais situações;
- **Estrutura:** representação gráfica da estrutura básica da solução por meio de diagramas de classes e/ou seqüência na notação UML [Booch et. al. 2000], por exemplo;

- **Participantes:** descrevem as classes, objetos ou componentes participantes do padrão com suas respectivas responsabilidades;
- **Colaborações:** descreve como os participantes colaboram entre si para cuidarem de suas responsabilidades;
- **Consequências:** mostra quais são os resultados (positivos e negativos) de se utilizar o padrão permitindo analisar os custos e benefícios da aplicação do padrão antes mesmo de aplicá-lo;
- **Implementação:** descreve técnicas, dicas ou questões específicas da linguagem para implementar o padrão;
- **Código Exemplo:** fragmentos de códigos que ilustram como implementar o padrão;
- **Usos Conhecidos:** exemplos encontrados do padrão em sistemas reais para diferentes domínios;
- **Padrões Relacionados:** esta seção descreve como o padrão está relacionado com outros padrões que se referem ao mesmo problema, quais as diferenças entre eles e com quais outros padrões deveriam ser utilizados.

5.2.4. Formato POSA

O formato POSA presente no livro “Pattern Oriented Software Architecture” [Buschmann et. al. 1996], similarmente ao GoF, é muito bem estruturado e possui extensas seções.

Para documentar no formato POSA, deve-se estruturar o padrão com as seguintes seções:

- **Nome:** o nome do padrão e, logo abaixo em destaque, breve resumo do padrão;
- **Também conhecido como:** outros nomes para o padrão, se existirem;
- **Exemplo:** um exemplo do mundo real demonstrando a existência do problema e a necessidade para a documentação do padrão. Resume a essência do padrão e serve como introdução às seções “Contexto”, “Problema” e “Solução”;
- **Contexto:** a situação na qual o padrão deve ser aplicado;
- **Problema:** o problema que o padrão resolve, incluindo a descrição das forças associadas. As forças são expostas em destaque no texto;
- **Solução:** descreve o princípio fundamental da solução;
- **Estrutura:** descreve especificação detalhada dos aspectos estruturais do padrão;
- **Dinâmica:** cenários típicos que descrevem o comportamento de execução do padrão. Prepara o leitor para a seção seguinte;
- **Implementação:** dicas de implementação do padrão. Se possível, oferecer fragmentos de códigos em diferentes linguagens;

- **Exemplo Resolvido:** discussão de aspectos importantes para resolução do exemplo que ainda não foram descritos nas seções “Solução”, “Estrutura”, “Dinâmica” e “Implementação”;
- **Variantes:** breve descrição das variantes ou especializações do padrão;
- **Usos Conhecidos:** exemplos de usos do padrão em sistemas conhecidos reais;
- **Consequências:** descrição dos benefícios que o padrão provê e seus problemas potenciais;
- **Veja também:** referências para padrões que resolvem problemas similares e para padrões que ajudam a refinar esse padrão.

5.2.5. Formato J2EE

O formato J2EE descrito no livro “Core J2EE Patterns: Best Practices and Design Strategies” [Alur et. al. 2003] é menor se comparado à quantidade de seções que os formatos GoF e POSA possuem. Entretanto, possui seções igualmente extensas em virtude da necessidade de exemplificar a solução com códigos de implementação.

Para documentar no formato J2EE, o autor deve estruturar o padrão com as seguintes seções:

- **Problema:** descreve o problema de projeto encontrado pelo desenvolvedor;
- **Forças:** lista as razões e motivações que afetam o problema e a solução do padrão. Essa lista de forças enfatiza os motivos pelos quais alguém deve decidir por que aplicar esse padrão em detrimento de outro e provê uma justificativa para o uso do padrão;
- **Solução:** descreve brevemente a abordagem da solução e os elementos da solução em detalhes:
 - Estrutura: utiliza diagramas de classe UML [Booch et. al. 2000] para mostrar a estrutura básica da solução;
 - Estratégias: descreve as diferentes maneiras que o padrão pode ser implementado;
- **Consequências:** descreve as vantagens e desvantagens resultantes da aplicação do padrão;
- **Código Exemplo:** exemplos de implementação para o padrão e suas estratégias. Essa seção pode ser opcional caso a solução já tenha sido completamente contemplada na subseção “Estratégias” da solução;
- **Padrões Relacionados:** descreve a lista de outros padrões relacionados do livro e eventuais fontes externas, como os padrões do GoF.

5.2.6. Formato Anti-Padrões

A documentação de um anti-padrão difere da documentação de um padrão em virtude da necessidade de seções que comprovem que a solução recorrente gera consequências

negativas, além de seções de como proceder no sentido de resolver os problemas gerados a partir do anti-padrão. Brown et. al. no livro intitulado “AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis” [Brown et. al. 1998] define um formato bastante utilizado para documentar anti-padrões, para tal, deve-se estruturá-lo de acordo com as seções a seguir:

- **Nome:** identifica unicamente o anti-padrão, deve ser usado para referências futuras, portanto seu significado deve ser condizente com seu propósito;
- **Também Conhecido Como:** identifica outros nomes para a anti-padrão;
- **Nível mais Freqüente:** o nível é indicado por apenas uma palavra que identifica onde o anti-padrão se encaixa dentro do modelo de projeto de software descrito no livro. O nível pode ser: micro-arquitetura, frameworks, aplicação, sistemas, empresarial, global ou industrial;
- **Nome da Solução Refatorada:** identifica o nome do padrão da nova solução, ou seja, da solução reconstruída;
- **Tipo da Solução Refatorada:** o tipo da solução é formado por apenas uma palavra que identifica o tipo de ação que resulta da solução do anti-padrão. O tipo pode ser: software, tecnologia, processo ou papel. Software indica que um novo software é criado na solução. Tecnologia indica que a solução exige a aquisição de nova tecnologia ou produto. Processo indica que a solução exige a adoção de um processo. Papel indica que a solução exige atribuir responsabilidades a um membro ou equipe;
- **Causas Principais:** descrevem as causas gerais desse anti-padrão que podem ser: precipitação, indiferença, mentalidade fechada, preguiça, ganância, falta de conhecimento, orgulho ou falta de responsabilidade;
- **Forças Desbalanceadas:** identifica as forças principais que são ignoradas, pouco consideradas ou excessivamente consideradas nesse anti-padrão;
- **Evidências Verbais:** seção opcional que indica frases freqüentemente ouvidas que estão relacionadas a esse anti-padrão;
- **Background:** seção opcional que contém considerações adicionais de onde e como o problema ocorre;
- **Forma Geral:** pode-se incluir diagramas para identificar as características gerais desse anti-padrão. Esta seção é importante já que a **Solução Refatorada** procura afirmar o que estiver dito aqui;
- **Sintomas e Conseqüências:** lista os sintomas e conseqüências que resultam desse anti-padrão. A lista deve ser numerada ou com marcadores;
- **Causas Típicas:** identifica outras causas desse anti-padrão. Esta seção complementa a seção de **Causas Principais**;
- **Exceções Conhecidas:** identifica as principais exceções para a anti-padrão;

- **Solução Refatorada:** explica a solução refatorada que resolve as forças identificadas na seção **Forma Geral**;
- **Variações:** lista as variações conhecidas desse anti-padrão, ou seja, as soluções alternativas, se existirem;
- **Exemplo:** demonstra com um exemplo como a solução é aplicada ao problema em sintonia com os detalhes da solução;
- **Soluções Relacionadas:** identifica citações e referências cruzadas que sejam apropriadas, inclusive outros anti-padrões relacionados;
- **Aplicabilidade para Outros Pontos de Vista e Níveis:** analisa esse anti-padrão sobre outros pontos de vista: gerencial, arquitetural e do desenvolvedor. Também descreve a relevância do padrão para outros níveis;
- **Referências e Fontes:** referencia padrões relacionados com esse anti-padrão.

5.3. O Workshop de Escritores

O Writers' Workshop (workshop de escritores, em inglês) é um dos eventos que ocorre durante as conferências PLoP. Ele pode ser considerado um dos momentos mais importantes dessas conferências. Na ocasião, onde ocorre a discussão de artigos que documentam padrões, o principal objetivo é o melhoramento do trabalho feito pelo(s) autor(es). O processo de apresentação de cada artigo, portanto, diferencia-se do processo da maioria das conferências existentes atualmente, onde somente o autor expõe os resultados de seu trabalho de pesquisa. Como será detalhado a seguir, este processo possui regras que devem ser seguidas para a boa condução do mesmo.

5.3.1. Motivação

Uma maneira simples para definir padrões se baseia na sua utilidade: padrões de software são soluções documentadas para problemas recorrentes na indústria de software como um todo. Essa visão de padrões nos faz perceber que uma das características que deve estar associada a qualquer padrão de software é a sua forma de solução, ou seja, se a solução apresentada por determinado padrão (seja ele de projeto, processo ou análise) constitui-se de uma solução comprovada para os problemas que ele se propõe a resolver.

Com o intuito de maximizar a contribuição que os padrões darão com sua documentação deve-se fazer com que os padrões sejam escritos com clareza, e que, de forma geral, possam ser usados corretamente quando necessário.

O objetivo dos padrões seria, portanto, conseguir promover a utilidade deles próprios, ou seja, não adiantaria a documentação de uma boa solução para um problema de desenvolvimento, por exemplo, se ela não é feita de forma que possa ser repetida em outras ocasiões.

Neste ponto, o encontro que acontece nas conferências PLoP, o qual este texto se propõe a expor, encontra sua razão de ser. O Writers's Workshop (doravante WW) consiste da discussão centrada na melhoria de artigos que documentem padrões. Esse processo foi apresentado por Richard Gabriel [Gabriel 2002], um especialista em padrões e poeta.

Richard, como freqüentador de reuniões de recitação e discussão de poesias, sugeriu que os padrões fossem revisados e criticados tais quais as poesias e com o mesmo intuito: melhorar o trabalho final. No caso das conferências PLoPs, essa melhoria servirá para a elaboração de uma versão final que será de fato publicada nos anais finais do evento (vale aqui destacar que todos os artigos selecionados para a conferência já passaram pelo processo de *shepherding*²), ou seja, o processo WW funciona como a última forma de melhoria do artigo. A maneira como a reunião acontece promove a participação de pessoas externas na tentativa tanto de agregar melhorias a cerca tanto do conteúdo quanto da forma do artigo. Segundo Coplien, em “A Pattern Language for Writers’ Workshops” [Coplien 1997], o WW provê uma boa prática de revisão de padrões e permite aos autores obter comentários diretos sobre a avaliação do seu trabalho. Coplien, baseado nisso e em diversas experiências em PLoPs, documenta, nesse artigo, uma linguagem de padrões descrevendo todo o processo do WW.

5.3.2. O Círculo WW: grupos e participantes

As sessões WW possuem uma forma particular de realização. A primeira característica que pode ser observada é a organização de todos os participantes na forma de um círculo. É fácil ver o porquê desta escolha: dessa forma todos têm a mesma visibilidade, e assim não existiria um ponto ou um local onde houvesse melhores oportunidades para se expressar (como aconteceria em uma disposição habitual de sala de aula, por exemplo). Este fato pode até parecer sem importância para o real objetivo da reunião, mas, na verdade, o círculo WW possui o importante objetivo de possibilitar que todos que queiram contribuir possam fazê-lo em igualdade de condições.

Os membros que participam de uma sessão são determinados pelo grupo referente àquela sessão, onde tudo acontece da seguinte forma: os artigos aceitos para a conferência são previamente separados em grupos. O grupo ao qual o artigo de sua autoria pertence determinará quais são as sessões que você deverá participar. Os participantes da conferência que não possuem artigos destinados ao WW devem escolher um dos grupos para participar. Cada participante só pode pertencer a um desses grupos. As sessões que ele irá participar (seja como “autor” ou como “demais participantes”) serão determinadas pelos artigos do grupo ao qual pertence.

Antes de descrever as regras da discussão e o próprio processo de análise que acontece nos WW’s (assunto das duas seções seguintes) devemos conhecer quem faz parte desse círculo e saber suas respectivas responsabilidades nas sessões:

- Autor(es): obviamente o(s) autor(es) participa(m) da sessão na qual o artigo de sua autoria está sendo analisado. Na verdade, os autores participam de todas as sessões que pertencem ao grupo pré-determinado que contenha seu artigo, mas de forma diferente. Na reunião de discussão de um artigo escrito pelo autor, por exemplo, sua participação ativa será apenas a leitura de uma parte do texto, algo como um parágrafo ou frase que resuma o artigo como um todo e consequentemente o(s) padrão(s) documentado(s). No decorrer dessa sessão o autor não mais participa das

² Processo que ocorre durante um período antes da realização da conferência, onde o(s) autor(es) do artigo tem a possibilidade de melhorá-lo de acordo com sugestões provenientes de um especialista.

discussões, pois o artigo e o(s) padrão(ões) nele documentado(s) devem estar claros o suficiente para atingir o seu objetivo sem a participação do(s) autor(es), o artigo deve “falar por si só”. Nas outras sessões do grupo, ele participa como demais participantes (ver abaixo);

- Moderador: dentre os participantes da conferência, uma pessoa com elevada experiência na área de padrões é colocada para ser o moderador das sessões WW, sendo um moderador por grupo de padrões. Essa pessoa geralmente já participou de outras sessões de WW, consequentemente conhecendo bem o processo. Seu papel, como será contextualizado na seção 5.3.4, é basicamente fazer com que as sessões do seu grupo ocorram da forma correta: na seqüência previamente definida e de acordo com as regras;
- Demais participantes: são considerados todos os participantes do grupo, com exceção dos autores do artigo em análise na sessão corrente e do moderador do grupo. Estes demais participantes possuem grande responsabilidade. Na verdade, a principal e mais importante parte do tempo será tomada por esses demais participantes que são os responsáveis pelas sugestões que possibilitarão a melhoria do padrão em discussão;
 - Resumidor: pessoa responsável por dar uma visão geral do artigo que será analisado. Esta pessoa resumirá o artigo, mas assim como os outros participantes, poderá no decorrer da sessão discutir o padrão que esteja sendo analisado.

5.3.3. Regras Gerais

Em relação às regras, o nosso objetivo não é despender muito tempo com a explicação das mesmas ou até mesmo justificar o motivo de sua existência. Como toda regra, o mais importante é segui-las. Veja, então, as regras necessárias de como proceder durante uma sessão de WW:

- O(s) autor(es) só participa(m)ativamente do início e do final de cada sessão. Ele(s) não deve(m) interferir com os comentários dos outros participantes durante a discussão propriamente dita do artigo. O(s) autor(es), no momento das discussões, é dito ficar como uma Mosca na Parede (veja padrão “Fly on The Wall” [Coplien 1997]). O fato de o autor não poder participar das discussões se justifica pelo fato de que o artigo deve “falar por si só”, visto que o autor não estará presente sempre que um leitor estiver interessado em seu artigo. A atividade recomendada para o autor no momento da discussão é somente observá-la e anotar o que é dito pelos outros membros do grupo;
- Quando existir a necessidade de se referenciar ao(s) autor(es) do artigo, deve-se chamá-lo(s) exatamente dessa forma – como ‘autor(es)’. Não se deve citar o nome de quem escreveu o artigo. Apesar de ser difícil (principalmente quando você conhece quem escreveu o padrão) este ato trará mais impessoalidade à discussão;
- Além de não citar nomes, não se deve fazer gestos, como apontar para o autor. Deve-se ainda evitar contato visual com o mesmo;

- No momento da discussão deve-se ter bom senso no sentido de não repetir o que já foi dito. Isto é relevante, pois é comum acontecer de outras pessoas terem pensado da mesma forma que você pensou, ou ainda, após alguma discussão polêmica, chegarem num acordo. Evite repetições. Apenas concorde. Para essas ocasiões deve-se pronunciar a palavra “Gosh!” (lê-se góshi).

É apropriado destacar que essas regras costumam ser explicadas e demonstradas antes do início das sessões WW que ocorrerão em determinada conferência PLoP. Todos os participantes devem seguir-las, pois, como já foi dito, elas representam, em conjunto, uma forma para a condução das sessões que permita o melhor aproveitamento tanto para o(s) autor(es) quanto para os outros participantes.

5.3.4. O Processo

Até este momento já foram dadas algumas informações sobre as sessões WW. Com o conhecimento sobre os participantes assim como sobre sua organização em grupos e sobre as principais regras para as sessões, apresentadas acima, já é possível fazer, de fato, uma descrição da sessão: sua estruturação completa, a sua seqüência pré-estabelecida, os acontecimentos mais comuns, entre outros. A seguir é mostrado em seqüência temporal as etapas que ocorrem durante uma sessão WW nas conferências PLoP:

1. **Apresentação do Padrão**: todos os encontros começam com a apresentação, pelo moderador, do artigo a ser discutido. Como formalidade, se é lido o nome do artigo e também o(s) nome(s) do(s) autor(es);
2. **Breve leitura**: após a devida apresentação pelo moderador do artigo que será posto em discussão naquela sessão, o autor (um dos, no caso de um trabalho não-individual) é convidado a ler uma parte de seu artigo. Não é tão importante qual parte será lida, ficando esta escolha com o próprio autor. Este passo é importante no sentido de relacionar o trabalho ao autor;
3. **Resumo por participante**: nesta etapa, como o nome indica, o resumidor deve de forma objetiva resumir o artigo referente àquela sessão. Apenas resumi-lo! Não se deve começar ainda a crítica ao artigo. Um membro do grupo, geralmente um voluntário, fará esse resumo para todos os outros participantes. A pessoa responsável por resumir deve passar uma visão ampla do artigo sem especificar todos os seus pontos. Apesar da existência desta parte nas sessões, todos os participantes devem ter lido previamente e atenciosamente os artigos do grupo a qual pertencem;
4. **Aspectos positivos**: a discussão sobre o artigo realmente começa aqui. Sob a gerência do moderador, os demais participantes (incluindo o moderador) procuram elencar quais são as características observadas no artigo sob crítica que o fazem um bom padrão, um padrão bem documentado. Além disso, essa etapa funciona para mostrar ao(s) autor(es) quais são as características que devem permanecer para a versão final do artigo. Dentre esses “pontos positivos” podem ser citados desde a formatação/organização das seções até a existência de recursos que facilitem o entendimento do padrão. Nesta fase, assim como também durante o decorrer da sessão, o moderador deve estar atento. Algumas vezes, acontece de alguém mencionar sobre os aspectos positivos e, simplesmente,

começar a citar alguns problemas encontrados no artigo (assunto para a próxima etapa). Outras vezes, percebe-se que alguns participantes começam a falar sobre os mesmos temas, mas com palavras diferentes, por exemplo. Nessas horas, o moderador deve, educadamente, tomar a palavra e tentar apurar uma opinião geral e concisa dos participantes para que seja possível continuar com a sessão sem nenhum desperdício de tempo. O modo como o artigo será analisado acaba dependendo do moderador. Uma forma já consagrada constitui-se da análise em seqüência do artigo ao invés de deixar que os membros dêem suas opiniões sobre quaisquer partes do padrão sem nenhuma seqüência;

5. **Sugestões de Melhoria**: funcionando de forma semelhante à etapa anterior, essa parte da sessão também serve como período para a real discussão do artigo. Esta é uma das partes mais extensas das sessões. Nela, os demais participantes devem não apenas mostrar os erros, mas também, e principalmente, propor sugestões para a correção destes. Em contraste à etapa anterior, aqui deverão ser mostradas as características que deverão ser adicionadas/modificadas para a versão final do artigo. É válido destacar que, apesar de continuar sendo o responsável pela execução da sessão, o moderador também pode expressar sua opinião nesta etapa do processo. Dentre os tipos de participação, se costuma sugerir que alguns elementos mais elucidativos sejam adicionados, ou que algumas partes sejam mais bem explicadas, ou ainda idéias novas para melhoria do trabalho. Além disso, um melhoramento efetivo do padrão também é conseguido através da discussão sobre a solução documentada e a usabilidade da mesma. Novamente, aqui o moderador deve estar ciente da necessidade de não estender as discussões que certamente levariam muito tempo. Nestas ocasiões procura-se entrar em um consenso, e, caso este não seja ou pareça possível, o moderador deve propor que os membros que não estejam conseguindo um ponto em comum tentem tratar do assunto com o(s) autor(es) após a sessão;
6. **Esclarecimentos do(s) Autor(es)**: podemos observar que o(s) autor(es) não participa(m) ativamente da reunião desde a leitura de um trecho do artigo (etapa 2 acima). Ele(s) permanece(m) fisicamente fora do círculo dos demais participantes, anotando as observações, observando o andamento das discussões, analisando os pontos de vista dos membros de seu grupo. Após a conclusão da fase anterior, o moderador finaliza a discussão e, nessa etapa, o(s) autor(es) tem a oportunidade de esclarecer dúvidas que tenham surgido durante esse período de observação com os demais participantes. É importante destacar que este não é um momento para a retomada da discussão. A atividade a qual o(s) autor(es) deve(m) se limitar é a solicitação de esclarecimentos sobre alguma opinião dada durante a discussão. Lembre-se que deve estar escrito no artigo tudo que é necessário para o completo entendimento e posterior utilização do(s) padrão(ões) documentado(s);
7. **Fechamento Positivo**: neste momento, os participantes reconhecem o trabalho do(s) autor(es). Para isso, normalmente um dos demais participantes enfatiza as contribuições do artigo. Essa sessão serve para que a discussão a cerca dos padrões acabe de forma positiva.

8. **Saudações:** uma maneira formal de encerramento das sessões. O(s) autor(es) pode(m) agradecer em relação às questões levantadas durante a discussão e, finalmente, é aplaudido pelos demais participantes (incluindo aqui, o moderador).

5.4. Dicas de Como Escrever Bons Padrões

Nessa seção discutiremos algumas “boas práticas” de escrita de padrões de software. Apesar de não garantirem sozinhas a produção de padrões de qualidade, essas dicas apresentam práticas que podem ajudar, especialmente a iniciantes, na documentação de bons padrões que podem ser posteriormente refinados.

Discutiremos 10 dicas. Por restrições de espaço, as 5 últimas dicas serão apresentadas de forma reduzida. Em cada uma delas, o texto em negrito e itálico apresentado entre linhas descreve a dica de forma simplificada. No texto que aparece antes dessa descrição, no caso das dicas completas, apresentamos a motivação por trás dessa dica. Finalmente, no texto após a descrição resumida da dica, mostramos detalhes sobre como implementá-la e referências de outras publicações que discutem a mesma solução.

Use e Abuse do Conhecimento de Especialistas (É grátis !!)

Desde o início da década de 1990 onde os primeiros padrões de software foram documentados, a comunidade de padrões já conseguiu acumular grande quantidade de conhecimento a cerca de boas práticas de escrita de padrões. Como consequência, vários trabalhos já foram publicados por especialistas que lidam especificamente sobre como escrever bons padrões [Sane 1995] [Meszaros et. al. 1996] [Vlissides 1996] [Harrison 2000] [Almeida et. al. 2004].

Antes mesmo de iniciar na escrita do seu padrão, leia os diversos trabalhos já publicados que apresentam “boas práticas” de documentação. Incorpore essas práticas sempre que possível.

Indiscutivelmente, a publicação com maior impacto na forma como padrões de software são documentados atualmente é o artigo “Metapatterns – A Pattern Language for Pattern Writing” [Meszaros et. al. 1996], de Gerard Meszaros e Jim Doble, publicado na conferência PLoP de 1996. Nesse artigo, os autores descrevem 22 boas práticas, documentadas na forma de padrões, que abordam desde a escolha do nome para o padrão até a discussão de como fazer o padrão fácil de entender. Para iniciantes na arte da escrita de padrões, essa é sem dúvida a referência ideal por discutir aspectos básicos e fundamentais que garantem a qualidade na documentação.

A linguagem de padrões “The Language of Shepherding: A Pattern Language for Shepherds and Sheep”, descrita por Neil Harrison em [Harrison 2000], tem como objetivo melhorar o processo de *shepherding* que acontece sempre antes das conferências PLoP. Apesar de ter sido desenvolvido com o intuito de descrever boas práticas de *Shepherds*, o artigo traz vários padrões com dicas importantes para beneficiar aqueles que desejam documentar padrões.

Em seu artigo “Elements of Pattern Style” [Sane 1995], o autor Aamod Sane apresenta algumas dicas de como evitar erros comuns na documentação de padrões. O artigo é organizado no sentido de discutir dicas gerais de escrita para cada parte de um padrão, i.e., seu Título, Intenção, Motivação/Problema, Solução, entre outras.

O autor John Vlissides em “Seven Habits of Successful Pattern Writers” [Vlissides 1996], apresenta sete hábitos desenvolvidos pelo autor durante o processo de escrita do livro “Design Patterns” [Gamma et. al. 1995]. Os hábitos, apesar de não garantirem sucesso no processo de documentação de padrões, têm o potencial de agilizar e facilitar a obtenção desse desejado sucesso.

O artigo “Student’s PLoP Guide: A Pattern Family to Guide Computer Science Students during PLoP Conferences” [Almeida et. al. 2004] tem como objetivo principal apresentar como novos participantes em conferências PLoP devem se comportar. Para isso, são apresentados treze padrões que descrevem desde a determinação se uma dada solução é realmente um padrão e deve ser documentada como tal, até a discussão do processo de Workshop de Escritores (seção 5.3) que o autor irá participar na conferência. Apesar de não ter esse propósito, o artigo discute e incentiva o desenvolvimento iterativo dos padrões, de forma que pode ser visto como ferramenta indireta de garantia de qualidade na escrita de padrões.

Finalmente, essa seção é um exemplo de tentativa de divulgação de práticas reconhecidamente boas para a documentação de padrões. Use-a.

Comece Pequeno (Mas Não Tão Pequeno !!)

Como discutido anteriormente, a documentação de padrões de software envolve a análise de diversos aspectos relevantes para o entendimento da sua solução e que permitem a reusabilidade do mesmo. Alguns desses aspectos são essenciais para o entendimento do padrão. Sem eles, a própria usabilidade da solução descrita no padrão estaria comprometida.

Adicionalmente, vemos que padrões evoluem de forma iterativa. Percebe-se claramente que o próprio processo de divulgação de padrões em conferências especializadas (conferências PLoP) estimula a iteração entre autores e membros da comunidade de padrões no sentido de permitir o melhoramento contínuo desses padrões.

Comece a documentação de seu padrão com uma versão resumida do mesmo. Essa versão deve conter somente os elementos fundamentais que caracterizam o padrão, ou seja, seu Nome, Contexto, Problema, Forças e Solução.

O padrão “Mandatory Elements Present” [Meszaros et. al. 1996] sugere exatamente os componentes acima como elementos fundamentais para a comunicação da solução descrita no padrão e do processo de escolha dessa solução. Adicionalmente, os autores desse artigo

lemboram que, apesar de frequentemente não estarem facilmente visíveis, esses componentes estão presentes em basicamente todos os padrões descritos até hoje. Finalmente, como forma de enfatizar o uso desses componentes fundamentais, o padrão “Optional Elements When Helpful”, apresentado no mesmo artigo, descreve outras seções, como Contexto Resultante, Padrões Relacionados, Exemplos, Exemplos de Código, Racional e outros, como componentes opcionais que devem ser incluídos somente quando fizer o padrão mais fácil de entender.

Frequentemente, autores de padrões iniciantes focam excessivamente na documentação de elementos não essenciais (como Exemplos de Código), perdendo muitas vezes a atenção nas práticas mais básicas de escrita de bons padrões. O padrão “Matching Problem and Solution” [Harrison 2000] trata dessa problemática, sugerindo a leitura das seções Problema e Solução em conjunto, no sentido de garantir que esses dois componentes fundamentais combinem, ou seja, que a solução resolva o problema por completo e somente ele. Nesse sentido, o desenvolvimento de uma versão inicial enxuta do padrão, facilita com que o autor se concentre nesses aspectos básicos de escrita de padrões de qualidade.

Na comunidade de padrões, uma versão inicial e resumida como a descrita acima, mas que normalmente não descreve as Forças é comumente conhecido como *Patlet*. O desenvolvimento de *Patlets* irá permitir que o autor foque no início exclusivamente nos elementos fundamentais que descrevem o padrão e, ainda, o desenvolvimento incremental do mesmo.

Defina, Descreva e Foque no PÚBLICO ALVO (Eles são seus clientes !!)

Para serem reusáveis, soluções e padrões, têm que primeiro ser entendíveis. Além disso, o autor de um padrão deve lembrar que leitores diferentes, com *backgrounds* diferentes, naturalmente requerem níveis de detalhamento e estratégias de apresentação diferentes. Alguns leitores serão impreterivelmente mais familiarizados com uma certa linguagem de programação do que outros, ou uma notação matemática, ou mesmo uma expressão própria de uma tecnologia desconhecida por alguns. Com isso, um autor dificilmente conseguirá descrever seu padrão de forma ideal para todos os tipos de leitores potenciais.

Adicionalmente, e mais do que freqüentemente, autores esquecem o perfil do leitor potencial do padrão e acabam por escrever como se eles mesmos fossem utilizar o padrão posteriormente. Esse erro pode tornar inútil até mesmo a melhor das soluções.

Defina um público alvo para seu padrão. Para garantir que o padrão será entendível por um membro típico desse público alvo, se coloque na posição desse membro. Use tecnologias, terminologias e notações conhecidas por membros desse público. Valide seu padrão o apresentando a um participante do seu público alvo. Caso necessário, realize revisões corretivas.

Os autores do artigo “Metapatterns – A Pattern Language for Pattern Writing” [Meszaros et. al 1996] discutem vastamente essa problemática. No padrão “Clear Target Audience”, os autores sugerem a identificação e descrição no artigo do público alvo principal para o padrão. Os autores do artigo sugerem ainda que o autor do padrão mantenha o foco no público alvo durante a escrita do padrão e a sua posterior validação com algum membro desse público. Como outro ponto positivo, esse artigo lembra que a descrição do público alvo ajuda com que as pessoas possam determinar o significado de algum termo ambíguo presente no padrão. Ainda no mesmo artigo, o padrão “Terminology Tailored to Audience” discute o uso de termos que sejam conhecidos por membros típicos do público alvo. Ainda nesse padrão, os autores sugerem que na introdução do artigo, sejam indicadas as referências da terminologia adotada. Já no padrão “Understood Notation”, os autores incentivam o uso de notações conhecidas por membros do público alvo. No caso do autor ter alguma dúvida sobre o conhecimento dessa notação que ele está utilizando, este deve incluir uma explicação clara e concisa sobre essa notação. Finalmente, o padrão “Glossary” sugere o uso de um glossário no sentido de garantir o entendimento de certos termos, caso exista alguma dúvida sobre a familiarização com esses termos em particular ou para ajudar leitores que não se encaixam diretamente no público alvo definido para o padrão.

Escolha um Formato (mas Não Morra Abraçado com Ele !!)

Diversos formatos diferentes têm sido utilizados na descrição de padrões de software. Existem formatos mais estruturados, como a Formato Canônico, GoF e POSA, e formatos mais descritivos, como o Formato Alexandriano. No entanto, percebe-se facilmente que não existe um único formato ideal para todas as situações, visto que padrões podem documentar soluções fundamentalmente diferentes em sua estrutura.

Adicionalmente, autores iniciantes comumente conhecem somente um ou dois formatos. Além disso, autores tendem a preferir o formato usado por padrões consagrados, como por exemplo, o formato utilizado do livro do GoF.

Finalmente, é importante lembrar que a definição de um formato é um componente importante no processo de escrita, pois torna tanto a escrita do padrão como seu entendimento posterior mais fácil.

Antes mesmo de iniciar a escrita do seu padrão, escolha um formato para ele. Use esse formato desde o início da documentação dos elementos essenciais do padrão (veja a dica “Comece Pequeno (Mas Não Tão Pequeno !!)”). Mantenha essa escolha enquanto não perceber dificuldades ao inserir novas informações no padrão. Caso isso aconteça, tente outro formato conhecido ou crie o seu.

O padrão “Form Follows Function” [Harrison 2000] enfatiza a necessidade de se definir um formato que seja coerente com o padrão. Comumente, autores tendem a forçar um

formato que não se adequa ao padrão que está sendo descrito, e acabam por desfigurar o padrão. Nesse sentido, os autores tratam de como alterar o formato de um padrão de forma gradual, pelas introduções de novas seções e remoção de informações duplicadas.

Jonh Vlissides [Vlissides 1996] discute a escolha do formato ideal para um padrão quando descreve a boa prática “Adhering to a Structure”. Nele, a autor discute a existência de vários formatos de documentação de padrões e a necessidade de escolha de um formato tanto para o padrão como para o seu autor.

Infelizmente, a única forma de descobrir que um formato não é adequando para o padrão que está sendo descrito é através de sua utilização. Caso isso aconteça com o formato que você escolheu, não insista nesse formato. Tente outro formato. Lembre-se que a escrita de padrões é um processo iterativo, sendo assim, a mudança do formato usado na documentação do padrão faz parte do processo evolutivo do padrão.

Forças, Forças e Forças (Sempre Elas !!)

Sem qualquer dúvida, o componente mais difícil de se documentar em um padrão são suas forças. Forças são naturalmente difíceis de se encontrar e descrever. No entanto, sua importância para o entendimento do padrão é tamanha que o autor deve dedicar o tempo e esforço necessário na sua documentação.

De forma geral, as forças irão descrever os diversos aspectos que são levados em consideração na escolha de uma solução para o problema discutido no padrão. Cada uma delas “empurra” para uma solução ou um conjunto de soluções. Se, para um dado problema, existisse somente uma força, sempre escolheríamos a solução que melhor resolve essa dada força. No entanto, forças são conflitantes por natureza. Percebe-se que para cada força que nos empurra para uma certa solução, existe uma ou várias outras que nos leva para não escolher essa solução. O que cada uma das soluções faz é escolher uma forma de balancear as forças que emergem do problema a ser resolvido, priorizando umas em detrimento de outras. Dentre todas essas soluções, aquela que resolve as forças de forma coerente com o que indica um dado contexto, naturalmente se tornará uma solução padrão.

No sentido de ilustrar esses conceitos, façamos uso de um exemplo simplificado. Suponha que alguém esteja interessado em comprar um meio de transporte (carro, moto, bicicleta) que possa levá-lo ao trabalho diariamente. O **problema** seria então “Qual, entre os vários meios de transporte disponíveis no mercado, comprar?“. Dentre os diversos aspectos que devem ser considerados na escolha de um meio de transporte para comprar, temos o “Custo Financeiro”. Essa seria, sem dúvida, uma das nossas **forças**, que inclusive está presente em basicamente todas as decisões que tomamos. Essa força nos “empurra” para **soluções** mais baratas. Se esse fosse o único aspecto a ser considerado na resolução de um dado problema, sempre escolheríamos a solução mais barata. No entanto, forças contraditórias sempre estão presentes. No nosso caso, podemos pensar em uma força “Conforto”. Essas duas forças são naturalmente conflitantes. Como dito anteriormente, o que as diversas soluções fazem é balancear as forças de uma certa forma. A escolha por um Carro, por exemplo, daria claramente preferência a força ”Conforto” em detrimento do

“Custo Financeiro”. Já a escolha de uma Bicicleta traria um balanceamento inverso. Mas, apenas, uma dessas soluções pode ser considerada um padrão. Para que isso seja determinado, precisamos analisar em que **contexto** nosso comprador se encontra. Caso o comprador possua uma boa quantidade de recursos financeiros, podemos ver que as soluções com maior “Custo Financeiro” se tornam menos problemáticas de forma que a força “Conforto” é priorizada por esse contexto, ou seja, o comprador irá provavelmente escolher a solução que gera maior “Conforto”. No nosso exemplo, o comprador irá possivelmente escolher um Carro, pois essa solução é a que prioriza as forças de forma coerente com o contexto. Essa solução, para esse problema, nesse contexto é provavelmente um padrão. Se tivéssemos um contexto diferente, uma pessoa sem qualquer reserva financeira, a força no contexto exigida para priorização seria o “Custo Financeiro”. Assim, é fácil perceber que entre as soluções possíveis, a escolha da Bicicleta seria a mais adequada e possivelmente se tornaria uma solução padrão.

De fato, sem uma discussão sobre as forças, um padrão seria meramente a descrição de uma solução para um problema.

Para descrever as forças em um padrão, considere soluções alternativas para o problema tratado pelo padrão. Documente como forças os diversos aspectos que são levados em consideração na escolha de uma solução em detrimento da outra. Defina um nome para cada força. Explique essas forças claramente e indique de que forma elas são conflitantes entre si.

O padrão “Visible Forces” [Meszaros et. al. 1996] também sugere que cada força receba um nome e que essas sejam claramente indicadas no texto, com formatação diferente (negrito, sublinhado ou itálico).

Em [Harisson 2000], o autor discute, no padrão “Forces Define Problem”, a forte ligação entre o problema e as forças. Ele afirma que frequentemente padrões trazem como descrição de um problema, uma questão que aponta diretamente para a solução descrita no padrão. As forças irão, então, dar substância ao problema e ajudam a descrever a complexidade da escolha de uma solução particular para esse problema.

Receba e Incorpore Sugestões via Shepherding e PLoPs

Sempre que possível, coloque seu padrão para discussão. Em especial, submeta seu padrão para uma conferência PLoP e utilize o processo de Shepherding e o Workshop de Escritores para obter sugestões de como melhorar seu padrão.

Escolha um Nome Represetativo

Escolha um nome para seu padrão que descreva tanto o problema que o mesmo resolve como a essência de sua solução.

Mostre que seu Padrão é Realmente um Padrão

Descreva claramente e de forma detalhada onde a solução descrita no padrão acontece no mundo.

Capriche nos Exemplos

Use exemplos concretos para ilustrar a aplicabilidade do padrão. Escolha exemplos de fácil entendimento para as pessoas.

Descreva Padrões Relacionados

Identifique e descreva outros padrões já documentados que de alguma forma são relacionados ao seu. Mostre claramente que tipo de relacionamento é esse.

5.5. Conclusões

Padrão de software é um dos temas mais excitantes na computação atualmente. O muito que já foi produzido em termos de padrões é muito pouco diante de todo conhecimento acumulado em décadas de desenvolvimento de sistemas computacionais. Nesse sentido, esperamos que esse tutorial tenha trazido não somente conhecimento suficiente que possibilite o uso de padrões, mas que de alguma forma tenhamos alimentado o sentimento altruísta de documentação de soluções no formato de padrões.

Em meados do próximo ano acontecerá no Ceará a Sétima Conferência Latino-Americana em Linguagens de Padrões para Programação (SugarLoafPLoP'2008). Essa será uma grande oportunidade para documentar novos padrões, aprimorar o conhecimento e discutir a aplicação de padrões. Nós, do Grupo de Padrões de Software da UECE

(GPS.UECE), nos colocamos à disposição de todos os interessados nessa área e esperamos encontrar todos no SugarLoafPLoP'2008.

5.6. Agradecimentos

Gostaríamos de agradecer a todos que de alguma forma colaboraram para que pudéssemos desenvolver um mini-curso de qualidade. Em especial, agradecemos as contribuições do aluno Fabrício Gomes de Freitas, bolsista de Iniciação Científica da FUNCAP e membro do Grupo de Padrões de Software da UECE (GPS.UECE). Gostaríamos também de agradecer as sugestões dos alunos da disciplina Padrões de Software do Mestrado Acadêmico em Computação da UECE (MACC), Mário Alves, Renato Lenz, Robson Feitosa e Viviane Santos.

Referências

- Alexander, C., Ishikawa, S., Silverstein, M., Angel, S. and Abrams, D. (1975) "The Oregon Experiment", Oxford University Press.
- Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. and Angel, S. (1977) "A Pattern Language: Towns, Buildings, Construction", Oxford University Press, New York, NY.
- Alexander, C. (1979) "The Timeless Way of Building", Oxford University Press, New York.
- Almeida, S., Alvaro, A., Lucrédio, D., Garcia, C. and Piveta, K. (2004) "Student's PLoP Guide: A Pattern Family to Guide Computer Science Students during PLoP Conferences", In the 4th Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP), Writers' Workshop, Porto das Dunas-CE, Brazil.
- Alur, D., Crupi, J. and Malks, D. (2003) "Core J2EE Patterns: Best Practices and Design Strategies", Prentice Hall.
- Ambler, S. (1998) "Process Patterns: Building Large-Scale Systems Using Object Technology", Cambridge University Press.
- Andrade, R., Marinho, F., Santos, M. e Nogueira, R. (2003) "Uma Proposta de um Repositório de Padrões Integrado ao RUP", Session Pattern Application (SPA), SugarLoafPLoP 2003, The Third Latin American Conference on Pattern Languages of Programming, Porto de Galinhas, PE.
- Appleton, B. (1997) "Patterns and Software: Essential Concepts and Terminology", <http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html>, Junho.
- Beck, K. and Cunningham, W. (1987) "Using Pattern Languages for Object-Oriented Programs", Technical Report nº CR-87-43, <http://c2.com/doc/oopsla87.html>, Junho.
- Booch, G., Rumbaugh, J. and Jacobson, I. (2000) "UML, Guia do Usuário: tradução", Campus, Rio de Janeiro;
- Brown, W., Malveau, R., McCormick, H. and Mowbray, T. (1995) "AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis", Wiley.

- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996) "Pattern-Oriented Software Architecture Volume 1: A System of Patterns", Wiley.
- Coad, P. (1992) "Object-Oriented Patterns", Communications of the ACM, V. 35, n°9, pages 152-159.
- Coplien, J. (1991) "Advanced C++ Programming Styles and Idioms", Reading-MA, Addison-Wesley.
- Coplien, J. (1994) "A Development Process Generative Pattern Language", Pattern Languages of Programs, Monticello, EUA.
- Coplien, J. (1994) "Generative pattern languages: An emerging direction of software design", C++ Report, pages 18-22, 66-67.
- Coplien, J. and Schmidt, D. (1995) "Pattern Language of Program Design", Addison-Wesley.
- Coplien, J. (1997) "A Pattern Language for Writers' Workshops", PLoP.
- Coplien, J. (2000) "Software Patterns: A White Paper", SIGS Publication.
- Coplien, J. and Harrison, N. (2004) "Organizational Patterns of Agile Software Development", Prentice Hall.
- Fowler, M. (1997) "Analysis Patterns: Reusable Object Models", Addison-Wesley.
- Freeman, E., Sierra, K. and Bates, B. (2004) "Head First: Design Patterns", O'Reilly.
- Gabriel, R. (1998) "Patterns of Software: Tales from the Software Community ", Oxford University Press.
- Gabriel, R. (2002) "Writers' Workshops & the Work of Making Things: Patterns, Poetry...", Pearson Education.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1993) "Design Patterns - Abstraction and Reuse of Object-Oriented Design", LNCS, n° 707, pages 406-431.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) "Design Patterns - Elements of Reusable Object-Oriented Software", Addison-Wesley.
- Harrison, B. (2000) "The Language of Shepherding: A Pattern Language for Shepherds and Sheep", 7th. Pattern Languages of Programs Conference, August, Allerton Park, Monticello, Illinois.
- Johnson, R. (1992) "Documenting Frameworks Using Patterns", OOPSLA '92, pages 63-76.
- Meszaros, G. and Doble, J. (1996) "MetaPatterns: A Pattern Language for Pattern Writing", Patterns Languages of Program Design, PLoP.
- Rising, L. (1998) "The Patterns Handbook: Techniques, Strategies, and Applications", Sigs.
- Sane, A. (1995) "The elements of pattern style", Proceedings of the Second Pattern Languages of Programs Conference, Addison-Wesley, Menlo Park, California.
- Schmidt, D., Stal, M., Rohnert, H. and Buschmann, F. (2000) "Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects", Wiley.

Vlissides, J. (1990) "Patterns: The Top Ten Misconceptions," Object Magazine,
<http://hillside.net/patterns/papersbibliographys.htm>, Junho.

Vlissides, J., Coplien, J. and Kerth, N. (1996) "Pattern Languages of Program Design 2", Addison-Wesley.

Vlissides, J. (1996) "Pattern Hatching: Seven Habits of Successful Pattern Writers", C++ Report, <http://hillside.net/patterns/papers/7habits.html>, November.

Capítulo

6

Introdução à Programação em TV Digital Interativa

Cidcley Teixeira de Souza

Resumo

Nos últimos anos, o interesse apresentado pela mídia em TV Digital Interativa (TVDI) vem demonstrando a importância dessa tecnologia. Esse interesse pode ser explicado pelo impacto esperado dessa tecnologia em todos os setores da cadeia produtiva da TV, no qual o desenvolvimento de software interativo pode ser destacado. Entretanto, devido às peculiaridades desse ambiente, o desenvolvimento desses novos produtos de software não segue os mesmos métodos de programação de aplicações convencionais para computadores. Nesse contexto, esse trabalho tem como objetivo principal fornecer os conhecimentos básicos necessários para a compreensão dos novos conceitos relacionados à programação de aplicações em TVDI.

Abstract

In the last years, the interest shown by media companies in Interactive Digital TV (IDTV) demonstrates the importance of this technology. Such interest can be explained by the expected impact this technology will have in all sectors of the TV production chain, in which the development of interactive software can be highlighted. However, due to the peculiarities of this environment, the development of such new software products does not follow the same programming methods of conventional computer applications. In this context, the main goal of this work is to provide the basic knowledge required for the understanding of the new concepts related to the programming of applications in IDTV.

6.1. Introdução à TV Digital Interativa (TVDI)

6.1.1 Breve Histórico da Televisão

A televisão é fruto do desenvolvimento científico e tecnológico de diversos pesquisadores. “É importante ressaltar que, em nenhum momento, um estágio evolutivo da televisão substituiu o estágio anterior; a evolução sempre foi lenta e gradual, se agregando paulatinamente ao modelo anterior” [Becker e Montez 2004].

Em 1817, o primeiro passo para a invenção da televisão foi dado através da descoberta do selênio como material capaz de transformar energia luminosa em energia elétrica. Com a possibilidade de transmitir imagens através da corrente elétrica, pesquisas sobre a transmissão de imagens foram iniciadas, até que, em 1932, a RCA - Radio Corporation of America -

apresentou a primeira televisão eletrônica. A partir desta descoberta, mais pesquisas em sistemas eletrônicos foram estimuladas em diversas partes do mundo, de forma que, alguns anos depois, a televisão já operava em países como a França, a Alemanha, a Rússia e os Estados Unidos.

Com o advento da Segunda Guerra Mundial (1939-1945), houve uma relativa desaceleração nas pesquisas, pois somente a Alemanha manteve a televisão funcionando. Porém, após a guerra, a televisão se desenvolveu tão rapidamente que passou a rivalizar com outros meios de comunicação da época, tanto no plano da informação, quanto no plano da publicidade. Na década de 1950, surgiu a televisão a cabo, também conhecida como TV por assinatura, oferecendo uma melhor qualidade de recepção em áreas longínquas. Neste mesmo ano, o Brasil colocou no ar a sua primeira estação de televisão, sendo o 50º país do mundo a ter televisão.

Alguns anos depois, a televisão preto-e-branco foi substituída pela televisão colorida e foi criado o videotape. Com a técnica do videotape ocorreu uma verdadeira revolução na televisão. Antes de sua descoberta, o conteúdo televisivo era transmitido ao vivo. Havia uma câmera que gerava um sinal diretamente para uma antena principal que, por sua vez, era responsável pela transmissão para as antenas domésticas, isto é, para a casa dos telespectadores. Quando surgiu o videotape, uma tecnologia capaz de armazenar as imagens captadas pelas câmeras, tornou-se possível o procedimento de edição das imagens, resultando em uma melhor elaboração técnica das cenas e na criação de efeitos especiais.

Nos anos 60, houve um considerável crescimento tecnológico que proporcionou o aperfeiçoamento dos meios de transmissão, principalmente devido ao surgimento dos satélites artificiais. Em 1962, os Estados Unidos realizaram a primeira transmissão oficial de televisão via satélite. Dez anos depois, a televisão em cores foi inaugurada no Brasil, após passar por um grande dilema referente ao padrão de televisão em cores que seria adotado. Existiam duas opções. A primeira opção seria adotar um dos três padrões já criados até então: o americano NTSC - *National Television System Committee*, o alemão PAL - *Phase Alternation Line* ou o francês SECAM - *Système Electronique Couleur Avec Mémoire*. A segunda opção seria desenvolver uma combinação destes padrões. Após uma série de discussões e estudos sobre os três padrões, as autoridades brasileiras concluíram que o mais adequado seria a combinação dos padrões NTSC e PAL.

A próxima fase da evolução da televisão corresponde ao surgimento da tecnologia digital. As emissoras de televisão substituíram seus equipamentos analógicos por similares digitais, obtendo uma melhor qualidade dos sinais audiovisuais. Além disso, o aumento do número de canais resultou na criação de um componente digital integrado ao aparelho receptor do sinal televisivo: o controle remoto, o qual vem proporcionando uma maior comodidade para o telespectador, que não precisa se locomover quando desejasse mudar de canal, ligar/desligar a televisão, aumentar/diminuir volume, etc.

No final da década de 80, os Japoneses iniciaram estudos com o objetivo de desenvolver novos conceitos relativos aos serviços de tecnologia digital. Então, apresentaram o HDTV - *High Definition Television*. O HDTV é um sistema com qualidade de imagem e som comparável à qualidade dos cinemas, proporcionando ao telespectador um maior realismo e, até mesmo, um maior envolvimento em relação ao conteúdo televisivo.

Porém, somente nos anos 90 a tecnologia de televisão digital se consolidou devido ao lançamento de três padrões mundiais. Lançado em Setembro de 1993, o primeiro padrão corresponde ao consórcio europeu DVB - *Digital Vídeo Broadcasting*, um resultado do trabalho de entidades da indústria relacionadas à televisão, dentre elas as emissoras, os fabricantes, as operadoras de rede, os desenvolvedores de software e os órgãos reguladores [DVB 2007]. Diversos países implementaram o padrão DVB como modelo de TV digital,

como foi o caso da França, Inglaterra, Portugal, Espanha, Suíça, Bélgica, Austrália, Grécia, Dinamarca, Áustria, Finlândia, dentre outros. O outro padrão é o norte-americano ATSC - *Advanced Television Systems Committee*. Ele foi lançado nos Estados Unidos no dia 1º de Novembro de 1998 pela ATSC, como uma organização internacional sem fins lucrativos para o desenvolvimento de padrões de televisão digital [ATSC 2007]. Países como Canadá, México, Coréia do Sul e Argentina já adotaram este padrão. Em 1999, o grupo DiBEG - *Digital Broadcasting Experts Group* - apresentou o padrão japonês de televisão digital ISDB - *Integrated Services Digital Broadcasting* [DiBEG 2004]. Até o momento, este padrão só foi adotado no Japão.

Seguindo essa tendência, o projeto do Decreto do sistema brasileiro de televisão digital terrestre foi instituído em 26 de novembro de 2003. De maneira similar ao que aconteceu durante a transição da televisão preto-e-branco para a televisão colorida, as autoridades brasileiras também tiveram que escolher entre os três padrões de televisão digital já existentes (DVB, ATSC, ISDB) ou optar pela criação de um padrão nacional. Desta vez, foi escolhida a criação de um novo padrão de televisão digital. Segundo [MC 2007]: “O programa visa realizar estudos técnico-econômicos de viabilidade para as tecnologias e soluções, subsidiando o Governo Federal nas decisões sobre o tema e disponibilizando o conhecimento gerado no decorrer do mesmo para os diversos agentes envolvidos – governo, emissoras, indústrias, empresas de software e de serviços e instituições de pesquisa”.

6.1.2 Conceitos Básicos sobre TVDI

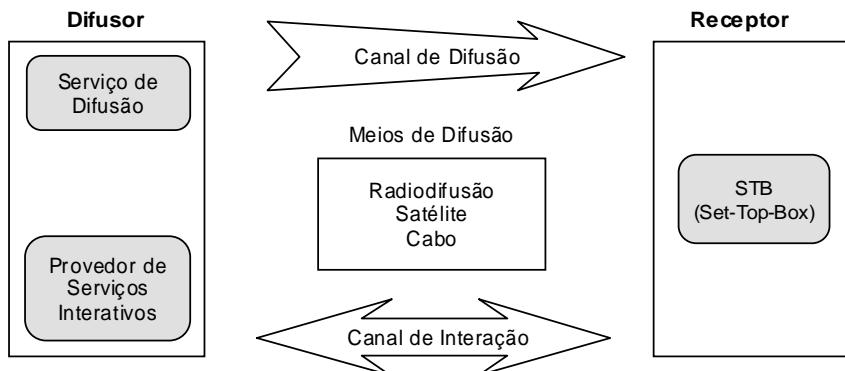
A Televisão Digital Interativa (TVDI) surge como uma ferramenta de comunicação baseada na transmissão digital de sinais audiovisuais. A imagem é formada por uma seqüência de códigos binários, isto é, de zeros (0) e uns (1). Sendo formado por apenas dois valores de bits, o sinal em uma transmissão digital pode ser reconstituído com uma qualidade superior ao sistema analógico devido a sua forma singular: sinal captado ou sinal não captado. Enquanto o sinal em uma transmissão analógica é quase sempre recebido, mesmo apresentando ruídos e se degradando devido ao afastamento do emissor, em uma transmissão digital isso não ocorre. Quando captado, o sinal não degrada enquanto puder ser recebido, além de poder ser regenerado eficientemente pelo receptor caso o sinal caia de vez.

A TVDI traz várias novidades positivas em relação à televisão analógica, como por exemplo:

- Som com qualidade de *Compact Disk* (CD);
- Imagem com qualidade de *Digital Vídeo Disk* (DVD) com 525 linhas;
- Incremento nas transmissões de áudio associadas aos programas, como dublagem e legenda em várias línguas;
- Transmissão de dados associados aos programas;
- Otimização de cobertura;
- Serviços adicionais de telecomunicações (comércio eletrônico);
- Maior número de canais;
- Maior diversidade de programação;
- Nova gama de aplicações e serviços interativos englobando texto, vídeo, áudio e os mais variados tipos de elementos gráficos.

6.1.3 Componentes de um Sistema de TVDI

Um sistema de TVDI pode ser decomposto em três componentes principais: um difusor, um meio de difusão e uma recepção doméstica [Fernandes et alii 2004]. A Figura 6.1. apresenta um modelo de sistema de TVDI com a representação desses componentes interagindo entre si.



Figuras 6.1. Componentes Básicos de um Sistema de TVDI.

Difusor

O difusor, também conhecido por emissor, gera o sinal dos programas de televisão produzidos nos estúdios dos provedores de serviços das emissoras de televisão. Assim como apresentado na Figura 6.1, o difusor é composto por dois provedores de serviço: provedor de serviço de difusão, responsável pelo envio (unidirecional) do sinal de áudio e vídeo para o canal de difusão, e o provedor de serviço de interação, responsável pelo envio e recepção (bidirecional) de dados para o canal de retorno, dando suporte às interações dos telespectadores.

O sinal pode ser transmitido ao vivo ou pode ser gravado antes de ser transmitido. Ao ser gravado, o conteúdo televisivo pode passar por subsistemas de produção (gravar, editar e criar programas) e armazenamento.

Meio de Difusão

O meio de difusão transmite o sinal gerado pelo difusor para o receptor doméstico. A transmissão pode ocorrer por radiodifusão (terrestre), satélite ou cabo. O conteúdo que será enviado pode ser de fluxos de áudio e vídeo, assim como pode ser de fluxos de dados.

Recepção Doméstica

A recepção doméstica é composta por dois sub-componentes:

- **Antena** – existente para as tecnologias de radiodifusão e satélite, captando o sinal difundido antes dele ser processado pelo receptor digital. No caso da tecnologia via cabo, o sinal vai direto para o receptor digital.
- **Receptor digital** – para as tecnologias de radiodifusão e satélite, o sinal é recebido da antena e enviado para o monitor do telespectador. O receptor digital é um equipamento que pode estar embutido no aparelho de televisão ou pode ser encontrado à parte. Nesse último caso, o equipamento é tecnicamente conhecido por STB – *Set Top Box* (o nome surgiu pelo fato do aparelho ser semelhante a uma caixa que normalmente fica por cima do televisor). O STB é um aparelho com uma tecnologia semelhante à dos decodificadores de TV por assinatura (cabo ou satélite). Consiste em um hardware adaptador agregado à televisão analógica que tem a função de converter o sinal digital recebido em um sinal analógico compatível com o sinal dos aparelhos de televisão convencionais. Para que seja possível um nível de interatividade mais

intenso entre o telespectador e o provedor de serviços é necessário um canal de retorno, também conhecido como canal de interação. Nesse caso, quando ocorre interação, o STB é conhecido como “STB interativo”.

A Figura 6.2 apresenta as etapas do processamento do sinal em um STB interativo. Inicialmente, a antena capta o sinal difundido por radiodifusão ou satélite (quando o meio de difusão é via cabo esta etapa não existe). Em seguida, o sintonizador digital (*tuner*) recebe corretamente o sinal difundido, isolando um canal particular, para que o demodulador extraia o fluxo de transporte MPEG-2. Este fluxo é carregado pelo demultiplexador, que extrai todos os fluxos elementares. O decodificador converte tudo para o formato apropriado ao equipamento televisivo.

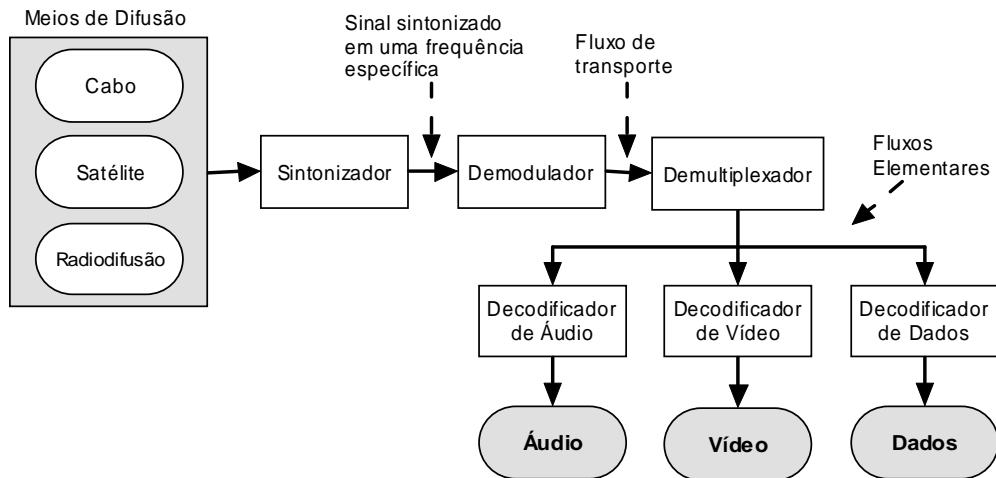


Figura 6.2. Diagrama Simplificado de Recepção de um STB.

Uma TVDI completa não precisa de um STB, pois, neste caso, considera-se que a televisão já possui internamente antena, sintonizador, demodulador, demultiplexador e decodificador. Porém, um aparelho de televisão completamente digital ainda não é uma tecnologia acessível devido ao seu alto custo.

6.1.4 Arquitetura de um Sistema de TVDI

A arquitetura de um sistema de TVDI pode ser representada através de uma arquitetura em camadas (Figura 6.3). Neste tipo de arquitetura, cada camada oferece serviços para uma camada superior e utiliza os serviços que são oferecidos por uma camada inferior.



Figura 6.3. Arquitetura de um Sistema de TVDI.

Esta arquitetura está presente nas duas extremidades do sistema de TVDI, ou seja, no ambiente do difusor (emissoras de televisão) e no ambiente do receptor doméstico (usuário dos programas de televisão). As camadas em um sistema de TVDI são:

- **Camada de Transmissão** – agrupa três subsistemas:
 - Transmissão e recepção – responsável pelo levantamento do sinal no difusor e pela sintonia do sinal no receptor.
 - Modulação e demodulação – responsável pela modulação e demodulação do fluxo de transporte codificado.
 - Codificação e decodificação – responsável pela codificação e decodificação (codec) do fluxo de transporte.
- **Camada de Transporte** – no ambiente da emissora é responsável pela multiplexação de vários programas em um único fluxo de transporte. No ambiente do usuário, esta camada realiza a demultiplexação do fluxo de transporte de acordo com o programa selecionado pelo telespectador.
- **Camada de Compressão** – realiza os processos de compressão de sinais de áudio e vídeo (A/V) no ambiente da emissora (difusor) e descompressão de sinais de A/V no ambiente do telespectador.
- **Camada de Middleware** – camada de software que oferece um serviço padronizado para a camada de aplicação, escondendo as peculiaridades e heterogeneidades das camadas de compressão, transporte e transmissão.
- **Camada de Aplicativos** – corresponde à camada visível para o usuário e que fará a interação direta com o mesmo, sendo suportada pelas camadas inferiores. É responsável pela execução dos aplicativos (Internet, filmes, etc).

6.1.5 Cenário de um Sistema de TVDI Terrestre

A Figura 6.4 apresenta um cenário de um sistema de TVDI. Neste cenário são apresentados os componentes descritos no item 6.1.3 (difusor, meio de difusão e recepção doméstica) em conjunto com a arquitetura em camadas detalhada no item 6.1.4 (transmissão, transporte, compressão, *middleware* e aplicativos), de forma que é possível identificar os fluxos envolvidos na transmissão e recepção do sistema: áudio, vídeo e dados (A/V/D). Além disso, pode-se observar que a composição do sistema é formada por uma série de subsistemas relativamente independentes que trabalham em conjunto e garantem a transmissão e recepção dos fluxos de A/V/D e um determinado nível de interatividade (intermitente ou permanente) entre os pontos comunicantes, posto que existe um canal de interação.

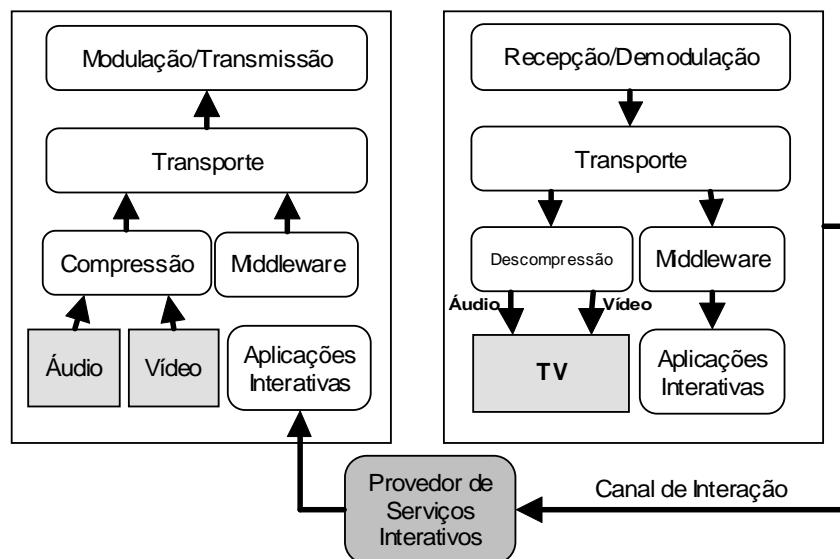


Figura 6.4. Cenário de um Sistema de TVDI.

Para prover os programas em formato digital (camada de aplicativos), as emissoras comprimem a informação de acordo com a largura de banda alocada para o canal de televisão (camada de compressão). No caso do envio de vários programas em um mesmo canal, após a compressão individual, ocorre a multiplexação dos vários programas em um único fluxo de transporte (camada de transporte). O fluxo de transporte é codificado (1º subsistema da camada de transmissão), de forma que sua reconstrução no ambiente do receptor doméstico apresente o menor número de erros possíveis. Após a codificação, o fluxo é modulado em uma portadora de alta frequência (2º subsistema da camada de transmissão) e transmitido no ar por uma antena apropriada (3º subsistema da camada de transmissão).

Para receber os programas oferecidos pelas emissoras, o ambiente de recepção doméstica realiza o mesmo processo descrito anteriormente, porém, de forma inversa. Inicialmente, ocorre a sintonização do sinal captado por radiodifusão pela antena do STB interativo (1º subsistema da camada de transmissão). Em seguida, ocorre o processo de demodulação e decodificação (2º e 3º subsistemas da camada de transmissão). Desta forma, o sinal resultante é um fluxo de bits equivalente ao fluxo de transporte original. De acordo com o programa selecionado pelo usuário, a demultiplexação (camada de transporte) é realizada, seguida da descompressão (camada de compressão). Todo este processo resulta em um programa que já pode ser visualizado pelo telespectador através da tela da televisão (camada de aplicativos).

6.1.6 Sistemas Abertos de Televisão Digital

Atualmente três importantes sistemas abertos para TVDI encontram-se em desenvolvimento. O sistema predominantemente europeu, *Digital Vídeo Broadcasting*(DVB), o sistema norte-americano, *Advanced Television Systems Committee*(ATSC), e o sistema japonês, conhecido como *Integrated Services Digital Broadcasting*(ISDB).

Esses sistemas, DVB, ATSC e ISDB, especificam as etapas apresentadas neste texto com relação à produção, difusão e recepção de serviços de TVDI, através da adoção de padrões para cada uma dessas etapas. Esses sistemas possuem várias similaridades entre si, porém divergem consideravelmente com relação aos padrões adotados na modulação do sinal, além das especificações referentes ao *middleware*.

DVB - Digital Vídeo Broadcasting

O grupo DVB foi fundado em 1993 através da união de diversas empresas públicas e privadas de vários países europeus, além da Austrália, que o adotou em 2001. O DVB-T, especificação para a radiodifusão, utiliza uma modulação do tipo COFDM (*Coded Orthogonal Frequency Division Multiplexing*), e a taxa de transmissão pode chegar a 31 Mbps, dependendo dos parâmetros utilizados na modulação do sinal. A largura de banda original é de 8 MHz, com a possibilidade de ser escalonado para 6 MHz.

A compressão e codificação de vídeo são feitas sobre o padrão *MPEG-2 Video*. A codificação de áudio é realizada através do padrão *MPEG-2 Layer II Audio*. A multiplexação segue o padrão *MPEG-2 Systems*. O *middleware* utilizado é conhecido como *Multimedia Home Platform*, ou MHP.

Uma de suas vantagens, principalmente sobre o padrão ATSC, é uma maior imunidade aos problemas de multi-percurso, além da possibilidade de recepção móvel. Uma desvantagem, porém, é a baixa imunidade à interferência causada por equipamentos eletrônicos.

O país destaque no uso do DVB é a Inglaterra, o qual já possui mais de um milhão de usuários. Neste e nos demais países, parte da televisão digital terrestre é um serviço pago, e os terminais de acesso são subsidiados pelas operadoras televisivas.

ATSC - Advanced Television Systems Committee

Esse sistema se consolidou em novembro de 1998, e além dos Estados Unidos, foi adotado pelo Canadá, México, Coréia do Sul e Taiwan. Utiliza a modulação 8-VSB (*Vestigial Side Band*) em uma largura de banda de 6 MHz (com a possibilidade de ser escalonado para 8 MHz), alcançando uma taxa de transmissão de 19,4 Mbps.

A codificação de vídeo é feita sobre o padrão MPEG-2 Video. Já a codificação de áudio é realizada através do padrão Dolby AC-3. A multiplexação segue o padrão MPEG-2 Systems. O middleware utilizado é o *DTV Application Software Environment*, ou DASE.

O ATSC foi projetado para a recepção por antenas externas, voltado para a televisão de alta definição (HDTV), e não permite em sua versão original recepção móvel. Outra desvantagem é a necessidade de uso de equalizadores para a recepção de sinais em situações de multi-percurso.

Até o momento, este sistema não obteve sucesso basicamente devido aos altos preços dos aparelhos de alta definição e da maioria dos terminais de acesso. Outro fator negativo é o fato de que mais de 80% da população norte-americana possui TV por assinatura via cabo ou via satélite, onde os difusores adotam seus próprios padrões para a migração da tecnologia analógica para a digital.

ISDB - Integrated Services Digital Broadcasting

Estabelecido em 1999 por várias empresas e operadoras de televisão japonesas, o ISDB-T é o sistema de transmissão terrestre adotado somente por esse país no momento. Utiliza o padrão de modulação COFDM, com algumas variações, e possui uma taxa de transferência de 24 Mbits/s, e uma largura de banda de 6 MHz (escalonável a 8 MHz).

A codificação de vídeo, como nos dois sistemas anteriores, segue o padrão *MPEG-2 Vídeo*. A codificação de áudio utiliza a especificação *MPEG-2 AAC (Advanced Audio Coding)*. A multiplexação segue o padrão *MPEG-2 Systems*. O middleware é conhecido como ARIB, da *Association of Radio Industries and Businesses*. As suas maiores vantagens são a grande flexibilidade de operação e seu maior potencial para transmissões móveis e portáteis.

6.2. Interatividade

A evolução das telecomunicações e da Internet tem despertado nos usuários uma expectativa de também poder utilizar a televisão como um meio de interação com os programas, conteúdos e serviços, de maneira semelhante à interação oferecida aos usuários da Internet.

Neste contexto, cria-se uma nova demanda no mercado mundial por aplicações interativas via televisão. Podendo usar esses programas e serviços interativos diretamente pela televisão, seja com a ajuda de um controle remoto ou de um teclado, o usuário tem a chance de se tornar mais ativo e seletivo que antes, concretizando o diálogo do telespectador com os programas, conteúdos e serviços.

De acordo com [Peng et alli 2001], todos os serviços interativos podem ser introduzidos em um ambiente de TVDI. Com novas ferramentas, as emissoras também dispõem de mais possibilidades de aumentar a audiência e, consequentemente, obter maiores retornos financeiros com os anunciantes. É importante ressaltar que o resultado do investimento na tecnologia digital depende do sucesso do modelo de negócio adotado por cada emissora, já que é preciso inovação na elaboração de novos serviços que atraiam e incentivem o usuário de televisão a consumir os serviços oferecidos. Todo esse cenário vem transformando a indústria da televisão em um negócio cada vez mais lucrativo e cheio de novas possibilidades nas áreas de educação, cultura, comércio, indústria, política, entretenimento, etc.

A interatividade tornou-se um recurso essencial para uma televisão digital de qualidade, criando-se assim, através da união destas, uma TVDI. Através da TVDI, o telespectador abandona o papel meramente passivo para se tornar ativo frente à programação, com o poder de interagir com a televisão e elaborar o seu próprio conteúdo televisivo. Analisando do ponto de vista técnico, o grau de interação do usuário com as aplicações, serviços e conteúdo interativos pode ser classificado em três categorias: local, intermitente e permanente [Fernandes et alli, 2004].

6.2.1 Interatividade Local

A interatividade local pode ser considerada como a mais básica das três categorias. O difusor é composto pelo provedor de serviço de difusão, que gera o sinal dos programas de televisão para que o canal de difusão transmita os fluxos de áudio e vídeo unidirecionalmente para o receptor doméstico. A antena doméstica recebe estes fluxos, enquanto o receptor digital fica responsável pelo armazenamento, ou seja, pelas aplicações que estão sendo executadas e que permitem ao telespectador a interação propriamente dita. Porém, o telespectador não consegue realizar o envio de dados em direção ao emissor, pois não possui um canal de retorno no receptor digital. As informações enviadas pelo difusor são de caráter geral para todos os telespectadores.

Como exemplos de aplicações para este nível de interatividade pode-se citar a configuração de legendas, jogos residentes, guias de programação *Electronic Program Guides* (EPGs), entre outros. A Figura 6.5 apresenta uma tela de uma aplicação EPG, ilustrando um guia de televisão que disponibiliza ao usuário a programação precisa de cada emissora associada ao gênero (comédia, drama, suspense, documentário, etc), horário, duração, resumo de cada programa, dentre outras informações.



Figura 6.5. Aplicação EPG utilizando Interatividade Local. Fonte: [Freevo 2007].

6.2.2 Interatividade Intermitente

Na interatividade intermitente, mudanças significativas são realizadas, de forma que, nesta categoria, a comunicação do usuário em direção ao difusor seja possível. O difusor apresenta, além do provedor do serviço de difusão, um outro provedor denominado provedor de serviço de interação.

O provedor do serviço de difusão continua gerando o sinal dos programas de televisão para que o canal de difusão unidirecional transmita os fluxos de áudio e vídeo para o receptor doméstico. A antena doméstica também continua exercendo a função de receptora dos fluxos de áudio e vídeo. O receptor digital apresenta uma mudança significativa para o aumento da interatividade: um canal de retorno. Este canal permite que o telespectador transmita fluxos de dados ao difusor, que trata estes fluxos no provedor de serviço de interação. Porém, o canal de interação é unidirecional, de forma que o difusor não consegue enviar respostas ao telespectador. Por isso, o canal de retorno é considerado não-dedicado.

Por ser uma comunicação de dados unidirecional, essa categoria de interatividade é muito utilizada em aplicações como votações, pesquisas de opinião e *quiz*, onde o usuário envia alguma informação, mas não espera nenhuma resposta do difusor pelo canal de retorno. A Figura 6.6 apresenta uma tela da aplicação SMS, onde o usuário pode utilizar o controle remoto para enviar uma mensagem ao difusor.



Figura 6.6. Aplicação SMS utilizando Interatividade Intermittente. Fonte: [Jucá 2005].

6.2.3 Interatividade Permanente

A interatividade permanente é uma evolução da interatividade intermitente, na qual a comunicação dos dados no canal de interação deixa de ser unidirecional para se tornar bidirecional, existindo para isso um canal de retorno dedicado no receptor digital.

Através deste nível de interatividade, é possível ter o acesso às funções básicas de um computador conectado à Internet e usufruir aplicações como navegação, e-mail, *chat*, competições interativas (jogos multi-usuários em tempo real), compras, *homebanking*, educação à distância, etc. Outra característica importante nesta categoria de interatividade consiste no fato da comunicação fluir também entre os telespectadores.

A Figura 6.7 apresenta uma tela da aplicação AOLTV [Davenport 2000], onde o usuário pode enviar, receber, armazenar e apagar mensagens de e-mail utilizando um controle remoto apropriado ou, até mesmo, um teclado.



Figura 6.7. Aplicação de e-mail com Interatividade Permanente. [Davenport 2005].

6.3 Programação de Aplicações em TVDI

6.3.1 O DVB-MHP

Há alguns anos, a comunidade que desenvolvia as tecnologias para TVDI (emissoras, provedores de conteúdos, fabricantes de receptores, dentre outros) percebeu que para obter sucesso comercial seria preciso desenvolver um ambiente independente de *hardware* e

software específicos, aberto e interoperável. Assim, seria preciso desenvolver sistemas interativos que fossem portáteis em receptores e STBs de diferentes fabricantes, explorando a tendência mundial do movimento da convergência digital, principalmente entre a transmissão *broadcast*, a Internet e o consumo eletrônico.

Desta forma, no ano de 1997, o grupo europeu DVB iniciou a especificação da camada de *middleware* do projeto, definindo um padrão aberto que apresentasse um conjunto de tecnologias para implementação de serviços digitais multimídia interativos. Em junho do ano 2000, esta especificação deu origem à plataforma *Multimedia Home Platform*, mais conhecida por sua sigla MHP, tendo, por isso, sua especificação denominada de DVB-MHP.

Nos anos seguintes foram lançadas novas especificações: no ano de 2001 a versão MHP 1.1, no ano de 2003 a versão MHP 1.1.1 e em maio 2005 a versão atual MHP 1.1.2 [MHP 2005].

O MHP define uma interface genérica entre as aplicações digitais interativas e o terminal (*hardware* e sistema operacional) no qual as aplicações são executadas [MHP 2005]. É justamente essa interface a responsável pelo cumprimento do requisito que foi citado anteriormente: a portabilidade. Desta maneira, não cabe aos desenvolvedores de aplicações a preocupação com detalhes específicos de *hardware* e *software* dos terminais nos quais suas aplicações serão executadas.

Este padrão estende suas funcionalidades para serviços interativos em todas as redes de transmissão definidas pelo projeto DVB: satélite (DVB-S), cabo (DVB-C), terrestre (DVB-T) e microondas (DVB-MC e DVB-MS). Já é utilizado por países como a Itália, a Finlândia, a Coréia do Sul, a Alemanha, a Espanha e a Suécia. O projeto espera ainda o início da adoção do MHP na Dinamarca, Noruega, Austrália, Bélgica, Áustria, Malta, Hungria e República Checa.

Por conta da versatilidade e da disponibilidade de ferramentas gratuitas de produção de software, além de dispor de uma especificação aberta com um grande número de funcionalidades e também grande aceitação na pesquisa e desenvolvimento de sistemas de TVDI, o MHP é adotado nesse texto como base para o tratamento de aplicações. Contudo, apesar de pertencerem a outros sistemas de TVDI, as outras duas especificações de *middleware* possuem várias similaridades com a especificação européia.

Camadas Básicas do MHP

O DVB-MHP foi inicialmente construído para cenários de transmissão digital, onde o receptor doméstico recebe o conteúdo via um canal de difusão unidirecional e um canal adicional para a interação, o canal de retorno. A interação do usuário ocorre através da tela da televisão, na qual o MHP é operado via controle remoto. Desta forma, o DVB-MHP está definido em termos de três camadas básicas:

- A camada mais baixa da arquitetura é a camada *Resources* (camada de recursos), que reúne os elementos de *hardware* e *software* para servirem de base para o padrão MHP. Os mais comuns são os dispositivos de entrada e saída (I/O), memória, CPU, sistema gráfico e decodificador MPEG [MHP 2005].
- A segunda camada é conhecida por *System Software*, que corresponde a uma camada de software do sistema que realiza o isolamento das aplicações interoperáveis da camada de recursos de *hardware* e *software* [Peng et alii 2001], provendo uma visão abstrata da plataforma para estas aplicações. Desta forma, as aplicações não acessam diretamente os recursos, sendo um conjunto de APIs usadas para isso. Essa camada também apresenta um gerenciador de aplicações

que controla o ciclo de vida das aplicações MHP. Ela também tem suporte para os protocolos básicos de transporte.

- A última camada é denominada *Applications*. Esta camada inclui um gerenciador de aplicações, também conhecido por navegador, que, além de controlar o *middleware*, também controla as aplicações que estão sendo executadas nele. As aplicações acessam as plataformas através das APIs.

MHP Profiles

O MHP introduziu o conceito de *Profile* para ajudar na implementação das padronizações. Um *profile* define as potencialidades do STB em operar com os diferentes tipos de funcionalidades habilitadas pelo DVB, sendo que essa especificação apresenta três *profiles*:

O *Enhanced Broadcast Profile* foi definido desde a primeira versão da especificação (MHP 1.0). Pode ser considerado o mais básico dos *profiles* em termos de desempenho no STB, pois não fornece suporte para conexão IP (*Internet Protocol*). Além disso, não possui (ou apresenta limitado) um canal de retorno para o envio de fluxos de dados em direção ao difusor. Oferece suporte para a execução de aplicações locais no STB ou recebidas pelo canal de difusão.

O *Interactive TV Profile* também foi definido desde a primeira versão da especificação. Pode ser considerado mais avançado que o *profile* anterior em termos de desempenho no STB, pois apresenta no terminal um canal de interação mais significativo. Uma das diferenças consiste no fato de ser possível realizar, neste *profile*, *downloads* através do canal de retorno, o que no *Enhanced Broadcast Profile* só era possível através do canal de difusão. Também há um suporte mais intenso ao canal de interação com APIs apropriadas.

O *Internet Access Profile* foi definido na versão MHP 1.1. É o *profile* com mais destaque no padrão MHP, pois tem como alvo STBs mais sofisticados, com poder de processamento e memória maior do que os *profiles* citados anteriormente. Oferece suporte às aplicações com acesso a conteúdos da Internet. Além disso, esse *profile* contém o elemento DVB-HTML, no qual a informação é apresentada através de conteúdo hipermídia.

Modelo de Aplicações

O MHP define o modelo e ciclo de vida das aplicações, como também os protocolos e os mecanismos de distribuição de dados em ambientes de televisão pseudo-interativa (com interatividade somente local) e interativa [Fernandes et alli 2004].

Uma característica importante do MHP consiste no suporte a execução de aplicações JavaTV, denominadas DVB-J [Zhang 2003], que inclui uma máquina virtual Java denominada JVM, definida na especificação Java Virtual Machine da empresa Sun Microsystems. Um conjunto de pacotes de software (bibliotecas de códigos reusáveis e específicos para TVDI) provê APIs genéricas e as aplicações só acessam a plataforma através destas APIs, pois elas possibilitam que os programas escritos na linguagem Java tenham acesso aos recursos e facilidades do receptor digital de forma padronizada.

As APIs JavaTV permitem a produção de serviços para televisão tradicional e interativa, oferecendo, nesta última, níveis avançados de interatividade, além de gráficos de qualidade e um processamento local no STB. [JavaTV 2007]. É importante ressaltar que cada serviço é formado por um conjunto de conteúdos para a exibição (áudio, vídeo, dados sincronizados). Determinados pacotes de software oferecem algumas funcionalidades para manipulação destes serviços, principalmente por parte do usuário: seleção de um serviço, busca de informações sobre um serviço, filtragem de informações de um serviço, controle da apresentação de um serviço, acesso às informações entregues pelo canal de difusão.

Além destas funcionalidades, é possível gerenciar o ciclo de vida de toda aplicação, que o JavaTV nomeou de Xlet. Um Xlet pode vir previamente armazenado no STB, assim como pode ser enviado pelo canal de difusão. O conceito de Xlet será detalhado mais adiante.

6.3.2 Tecnologias Relacionadas ao Padrão DVB

O padrão de DVB para a transmissão digital é muito extenso e complexo - não é apenas um padrão, há muitos padrões que definem como o sinal é construído, como é transmitido e o que fazer quando ele é recebido no STB. Esta seção fornece uma breve visão geral de algumas partes do padrão DVB. Uma descrição mais detalhada dos padrões de DVB pode ser encontrada em [DVB 2007].

O MPEG-2

O DVB usa fluxos (*streams*) MPEG-2 para a transmissão do sinal de TVDI. Cada fluxo, denominado Fluxo de Transporte (*Transport Stream*), ou simplesmente TS, é transmitido através de um canal e tem tipicamente uma taxa de dados em torno de 40 Mbit/s (para satélite e cabo, e de 25 Mbit/s para a transmissão terrestre). Esta taxa da transmissão é suficiente para fornecer de sete a oito canais de TV separados. Um canal é geralmente uma faixa de freqüência, que é transmitida usando satélite, cabo ou transmissão terrestre.

Um TS consiste em um conjunto de sub-fluxos chamados Fluxos Elementares (*elementary streams*). Cada fluxo elementar pode conter áudio MPEG-2 codificado, vídeo MPEG-2 codificado ou dados encapsulados. Um TS é formado por pacotes que carregam esses fluxos elementares multiplexados no tempo. Para identificar que parte do fluxo de transporte pertence a que fluxo elementar cada pacote contém um identificador do pacote (PID) de seu fluxo elementar. Figura 6.8 mostra uma visão esquemática de um fluxo de transporte.

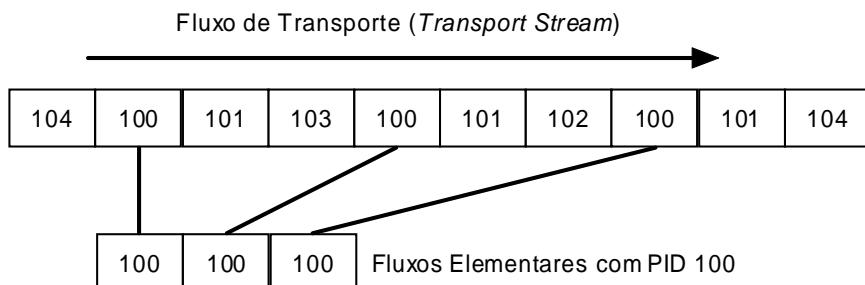


Figure 6.8. Um Fluxo de Transporte DVB.

Cada fluxo elementar pode ter uma taxa de bits diferente. Os fluxos vídeo geralmente necessitam uma taxa de bits mais elevada, enquanto os fluxos de dados usam taxas mais baixas. As taxas de bits diferentes resultam em uma freqüência diferente para fluxos elementares no fluxo de transporte. Por exemplo, o fluxo de transporte da Figura 6.8 contém mais pacotes do fluxo elementar com PID 101 do que o fluxo elementar com PID 104.

Um STB que recebe estes fluxos necessita de informações suplementares para decodificá-los da maneira correta, de modo a responder questões como: que fluxo contém que tipo de dados e que fluxos devem permanecer juntos. Esses dados são fornecidos pelas informações de serviço do fluxo. As informações de serviço podem ser vistas como uma base de dados simples que contém tabelas para programas, serviços e outros dados. Estas tabelas são transmitidas repetidamente no fluxo de transporte.

A tabela da associação de programas (*Program Association Table - PAT*) é a tabela fundamental para as informações de serviço. Um fluxo de transporte contém somente uma

PAT, que contém os PIDs da PMT (*Program Map Table*) de cada serviço. A PAT é transmitida com PID fixo 0.

O conjunto de alguns fluxos elementares é chamado de serviço. Um serviço pode ser um canal de TV, um canal de rádio, ou um canal de dados. A PMT descreve de quais fluxos um serviço consiste e de que tipo eles são. Há somente uma PMT por serviço em um fluxo de transporte. A relação entre os fluxos elementares, entre a PAT e as PMTs, pode ser visualizada no exemplo da Figura 6.9. O exemplo mostra uma parte do fluxo de transporte e como as tabelas de informação de serviço são usadas para a decodificação do fluxo.

As setas na Figura 6.9 indicam os dados de um canal de TVDI específico. Primeiramente, o STB recebe a PAT, localizada no PID 0. A PAT aponta para o PID 200, onde a PMT do serviço 1 é encontrada. O receptor procura os tipos e PIDs dos fluxos elementares que formam o serviço. O PID 100 carrega o vídeo, o PID 102 o áudio, e o PID 105 uma aplicação de TVDI. O receptor pode agora montar os diferentes fluxos elementares e decodificar o vídeo, o áudio e os dados.

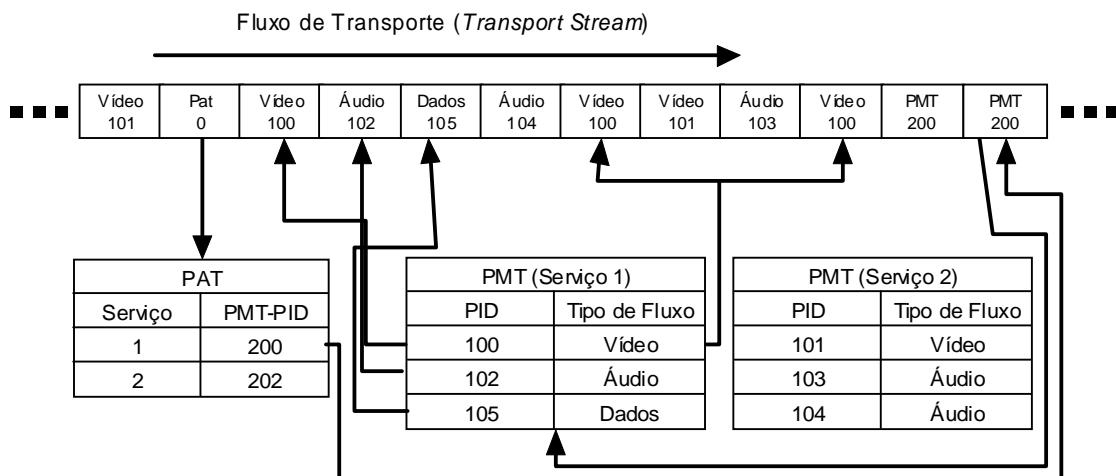


Figura 6.9. Tabelas PAT e PMT.

As informações de serviço dos fluxos MPEG incluem mais tabelas do que a PAT e a PMT. A seguir são apresentadas as tabelas geralmente encontradas em um fluxo de transporte DVB[DVB 2007]:

- *Network Information Table (NIT)*: A Tabela da Informação da Rede descreve como os fluxos de transporte são organizados sobre a rede atual. Uma rede no sentido do DVB descreve um radiodifusor. A NIT contém também o nome e a identificação da rede e das informações sobre outros fluxos de transporte na transmissão atual.
- *Conditional Access (CA) Table*: A tabela de Acesso Condicional define qual sistema de acesso condicional deve ser usado para decodificar um fluxo. Os sistemas de acesso condicionais (CA) são usados nas TVs pagas. O sistema de CA garante que somente os usuários autorizados possam usar um serviço específico.
- *Service Description Table (SDT)*: A Tabela da Descrição do Serviço, ao contrário da PMT, há somente uma SDT por fluxo de transporte. Essa tabela contém as informações para cada serviço. A SDT tipicamente contém informações tal como o nome do serviço, o ID do serviço e o status do serviço.
- *Event Information Table (EIT)*: A Tabela da Informação de Eventos contém nomes dos programas, tempos de começo, durações e outras informações.

- *Application Information Table* (AIT): A Tabela de Informações de Aplicação (AIT) é a tabela central para a TVDI. Essa tabela é transmitida para cada serviço que contém uma aplicação MHP. Ela contém uma entrada para cada aplicação MHP válida para esse serviço. A AIT contém todas as informações que o receptor necessitará para executar a aplicação e para avisar ao usuário que aplicações estão disponíveis no momento. Isto inclui elementos como o nome da aplicação, a posição de seus arquivos e alguns argumentos que devem ser passados à aplicação quando essa iniciar.[MHP 2007]

6.3.3 APIs de Programação para o MHP

As aplicações MHP usam uma API para acessar os recursos do STB. De fato, esse padrão não utiliza somente uma API, mas diversas APIs com funções específicas. O padrão MHP define e referencia muitas APIs que são suportadas por essa plataforma. Segue uma descrição sucinta de algumas APIs importantes no desenvolvimento de aplicações para esse padrão.

A API AWT (*Abstract Window Toolkit*)

A API de Java 1.1 define o núcleo das APIs de apresentação do MHP com o pacote AWT (`java.awt`). Mas nem todas as partes do AWT foram utilizadas pela especificação MHP, visto que o AWT foi projetado para interfaces baseadas em PC, que seguem uma estrutura completamente diferente da estrutura das interfaces de aplicações de TVDI.

Atualmente, a maioria dos ambientes para interfaces utiliza a idéia do WIMP (*Windows, Icons, Menus, Pointers*). As interfaces WIMP, como o nome sugere, utilizam um ponteiro para mover e selecionar janelas, ícones e menus. O ponteiro é manipulado quase sempre pelo *mouse* do computador. Para a entrada de texto e para atalhos um teclado é usado.

A interação na televisão é diferente, visto que o dispositivo de entrada primário é o controle remoto. A especificação MHP define apenas um conjunto muito básico de teclas para o controle remoto. Estas teclas incluem as teclas numéricas, teclas de navegação (para cima, para baixo, para esquerda, para direita), uma tecla de entrada, uma tecla para *teletext*, e quatro teclas de cores. Como os modelos de interação são diferentes do WIMP, apenas algumas classes gráficas básicas (como `Graphics`, `Component` e `Image`) do AWT foram usadas pela API do MHP.

O Pacote DVB UI

A parte básica do AWT é estendida por algumas classes novas com foco em TVDI. Essas classes estão contidas no pacote `org.dvb.ui`. `DVBGraphics` que estende as funcionalidades gráficas do AWT adicionando, por exemplo, o suporte a fundos transparentes e semi-transparentes, que são usadas pelas aplicações que não querem obstruir os vídeos em execução.

Sistemas MHP suportam somente um tipo de fonte, a Tirésias. Se as aplicações necessitam mais do que esse tipo de fonte elas podem usar a classe `FontFactory` do pacote `org.dvb.ui.FontFactory` para instanciar novas fontes que não são fornecidas no sistema, sendo que essas fontes têm que ser transmitidas através de difusão ou pelo canal de interatividade.

O HAVi (*Home Audio / Video Interoperability*)

Os componentes do HAVi (*Home Audio / Video Interoperability*)[HAVi 2007] substituem os componentes do AWT como *framework* de interfaces no MHP. Todas as classes começam com um H maiúsculo, que é normalmente a única diferença entre os nomes das classes do AWT. Os componentes do HAVi ajudam a fornecer um melhor visual para aplicações de TVDI. As duas principais diferenças para o mundo dos PCs são consideradas por estes

componentes: a baixa definição da TV e a falta dos dispositivos de entrada usuais (mouse e teclado).

O HAVi também introduz seus próprios componentes e *containers*. Pelo fato de não haver nenhuma janela em uma aplicação MHP um novo *container* de alto nível é fornecido no HAVi: a cena (classe `org.havi.ui.HScene`). Uma cena tem basicamente as mesmas propriedades de uma janela, mas não utiliza os conceitos de *frame*, *menu* ou barra de status. Além disso, o HAVi fornece diversos componentes que definem o comportamento usual de uma interface, como por exemplo:

- Botões com texto ou figuras: `HGraphicButton`, `HtextButton`
- Labels: `HstaticText`
- Lista e entradas de texto: `HListGroup`, `Htext`
- Gráficos e animações: `HStaticIcon`, `HIcon`, `HStaticAnimation`, `HAnimation`

Embora o HAVi forneça diversos componentes gráficos, estações de TV, assim como provedores de serviços MHP, freqüentemente utilizam seu próprio toolkit de interfaces, a partir de subclasses de componentes HAVi ou AWT, como forma de criar uma identidade própria.

JMF (Java Media Framework)

Muitas aplicações MHP podem modificar o vídeo atual com relação ao seu tamanho ou forma, por exemplo, um guia de programação poderá mostrar uma versão menor do programa atual na área de aplicação. Uma aplicação pode também apresentar um outro fluxo de vídeo, além do fluxo do programa atual. Para esta finalidade foi desenvolvida uma API Java: a API JMF (*Java Media Framework*).

A API JMF permite que áudio, vídeo ou outras mídias baseadas no tempo sejam adicionadas às aplicações e aos *applets* construídos com a tecnologia Java. Este pacote opcional pode capturar, mostrar e decodificar múltiplos formatos de mídias, e estende a plataforma de Java 2 *Standard Edição* (J2SE) para desenvolvimento multimídia. Na programação MHP os dois cenários principais do uso de JMF são os seguintes:

- Redefinir o tamanho atual do canal de vídeo que está sendo assistido pelo usuário e colocar alguma informação adicional sobre esse vídeo redefinido. Isto é usado freqüentemente em diversos programas através da inclusão de notícias e informações sobre o tempo. Uma outra aplicação que modifica o vídeo é usada em programas de esportes. Um exemplo seria o PIP (*Picture-In-Picture*) através do uso de diferentes câmeras durante um jogo de futebol.
- Apresentar trechos de vídeo como parte de uma aplicação, isso é usado em guias de programação para mostrar cenas de filmes e outros programas presentes na grade de programação.

A API DAVIC (*Digital Audio Visual Council*)

A API definida pelo DAVIC (*Digital Audio Visual Council*) [DAVID 2007] fornece acesso direto ao fluxo de transporte e aos fluxos elementares. O DAVIC foi uma associação audiovisual sem fins lucrativos composta de aproximadamente 200 companhias, agências de governo, e organizações de pesquisa em torno do mundo. A finalidade principal do DAVIC foi criar padrões da indústria para a interoperabilidade fim-a-fim de informações audiovisuais digitais interativas. A associação terminou seu trabalho em 2001 e foi fechada. Os padrões e

as publicações podem ser encontrados no site do DAVIC [DAVID 2007]. As classes para o acesso direto do fluxo DVB-MPEG podem ser encontradas no pacote de `org.davic.mpeg`.

A sintonização de um novo canal (modificar a faixa de freqüência de recepção) também é feita através de uma API do DAVIC (`org.davic.net.tuning`).

API de Acesso às Informações de Serviço

As informações de serviço podem ser acessadas através de duas APIs: a API DVB-SI (*Service Information*) e a API JavaTV. A maioria das informações podem ser encontradas em ambas APIs. Contudo, a API JavaTV fornece o acesso de uma forma independente do protocolo, visto que a API DVB-SI tem informações específicas do MHP/DVB. Todos os conceitos definidos em um fluxo DVB tais como PAT, PMT e outras tabelas de informação de serviço, podem ser acessadas com essas duas APIs.

API de Acesso ao Canal de Interatividade

Os STBs compatíveis com o MHP 1.1 para os *profiles Interactive TV* e *Internet Access* têm que implementar os protocolos IP, TCP, UDP, HTTP e o DNS sobre o canal de interatividade. Esses protocolos são acessados usando classes do pacote `java.io`.

O canal de interatividade de um STB pode ser um recurso escasso especialmente com conexões discadas. Consequentemente o canal de retorno tem que ser reservado e liberado pela aplicação em pedidos realizados ao STB. Isto é realizado pela *Return Channel Connection Management API* (`org.dvb.net.rc`).

6.3.4 Introdução a API JavaTV

Até o desenvolvimento do MHP a linguagem Java era usada principalmente como uma linguagem *script* para a extensão de outras plataformas de TV tais como MHEG [MHEG 2007]. Os padrões tais como DAVIC tinham definido já muitas APIs Java para TVDI, mas sempre com Java como uma extensão de outras tecnologias.

Em março de 1998, a SUN anunciou a API JavaTV, e a partir daí se iniciou o trabalho de definir uma plataforma Java pura para aplicações de TVDI. Parte deste trabalho envolvia a padronização do uso das APIs existentes nas plataformas de TVDI, tais como o uso de JMF e DAVIC. Ao mesmo tempo, JavaTV necessitou definir muitos novos componentes, incluindo um novo modelo de aplicação, APIs para o acesso de funcionalidades específicas de TVDI e uma arquitetura coerente que permitisse todo o trabalho destes elementos juntos.

JavaTV não era o único padrão em desenvolvimento neste período. As exigências comerciais para o MHP já tinham sido definidas no período em que JavaTV foi anunciado, e alguns meses após o anúncio de JavaTV, o DVB selecionou Java como a base para a plataforma MHP. Muitas das companhias envolvidas na iniciativa de JavaTV estavam trabalhando também no MHP, e em muitos casos as mesmas pessoas destas companhias estavam envolvidas no desenvolvimento de JavaTV.

Esse relacionamento próximo entre os dois padrões significou que esses foram projetados desde o início quase que de forma complementar, e muitas das sobreposições nos dois padrões foram o resultado da preocupação de JavaTV em realizar uma abordagem mais neutra com relação a plataformas de TVDI do que o MHP.

Ao contrário dos padrões tais como MHP e DAVIC, JavaTV fornece um conjunto básico de características para aplicações de TVDI. Essa API não define conceitos para uma plataforma completa de TVDI que ofereça o acesso completo a cada característica do STB, pois os padrões, tais como MHP, foram projetados para realizar essa tarefa. Contudo, o DVB concordou com os termos para o uso de Java no MHP. Por conta disso, a inclusão de JavaTV no MHP foi assegurada desde a sua versão 1.0.

Pacotes da API JavaTV

A Tabela 6.1 apresenta os principais pacotes da API JavaTV.

Tabela 6.1. Pacotes da API JavaTV.

Pacote	Descrição
javax.tv.locator	Este pacote fornece meios para referenciar dados e recursos acessíveis através das APIs de JavaTV. Ele define a interface <code>Locator</code> , que é usada como a entrada de todos os métodos usados construir os objetos a serem povoados por dados de transporte.
javax.tv.service	Este pacote fornece mecanismos para acessar a base de dados de informações de serviços (<i>SI Database</i>) e APIs que representam os elementos de SI. Os componentes dentro deste pacote incluem <code>SIRequestor</code> , usado para iniciar pedidos de dados, e <code>SIRetrievable</code> , usado para entregar os dados recuperados.
javax.tv.media	Este pacote permite o gerenciamento dos componentes dentro do fluxo de transporte que correspondem às mídias do conteúdo recebido. Como um exemplo, ele permite a extração, a seleção, e a apresentação do vídeo individual e de trilhas de áudio, recebidas pelo demultiplexador.
javax.tv.Xlet	Este pacote fornece mecanismos padrão para a comunicação entre os Xlets e seu ambiente de execução. Os Xlets JavaTV compreendem uma coleção de Xlets, cada um com seu próprio ciclo de vida, controlados através da interface <code>xlet</code> , e seus contextos da execução, acessíveis através de um <code>XletContext</code> .
javax.tv.Graphics	Este pacote fornece mecanismos padrão para que os Xlets controlem sua área de exposição. Permite que os Xlets descubram seus <i>containers</i> raiz e fornece um mecanismo uniforme para controlar cores.
javax.tv.Carousel	Este pacote permite o acesso aos arquivos transmitidos em um sistema de arquivos em um fluxo de transporte. Este pacote trabalha em sintonia com o pacote <code>javax.io</code> .
javax.tv.Util	Este pacote fornece mecanismos para controlar eventos temporais. É projetado para o uso genérico pelo Xlets, por exemplo, marcar o <i>time-out</i> de pedidos ou para sincronizar a apresentação de dados.
javax.tv.Net	Este pacote fornece métodos de acesso a datagramas IP transmitidos em um fluxo de transmissão.

Os Xlets

Um Xlet é um tipo de aplicação em conformidade com a API JavaTV que pode ser executada em um STB. Um Xlet precisa ser controlado através de um gerenciador de aplicações, visto que esse não possui uma função principal de entrada (*main*). Para tanto, um Xlet possui uma interface que pode ser usada pelo gerenciador para controlar seus estados de execução.

A Figura 6.10 e a Tabela 6.2 descrevem os estados do ciclo de vida de um Xlet. Um cenário típico de execução seria o seguinte: o gerente da aplicação recebe o código do Xlet e carrega o arquivo da classe a partir do seu ponto de entrada, criando uma nova instância do Xlet. O método construtor do Xlet é invocado e o Xlet vai para o estado *Loaded*. Em seguida, o gerente da aplicação cria um objeto de serviço de contexto necessário para que o Xlet funcione, inicializa o Xlet, e realiza a transição para o estado *Paused*. O objeto do contexto é

usado para permitir que o Xlet acesse as informações sobre o ambiente no qual ele está executando. Em seguida, o gerente da aplicação sinaliza para que o Xlet entre no estado *Active*. Neste ponto, o Xlet pode adquirir todos os recursos (por exemplo, outros arquivos, imagens) que necessita, e começa a executar o seu serviço. No final do seu ciclo de vida, o gerente da aplicação sinaliza para que o Xlet pare sua operação. O Xlet para então de executar o seu serviço. É importante que o Xlet libere todos os recursos que usou. Assim, se um recurso não for liberado pelo Xlet, ele deve ser liberado automaticamente pelo gerente da aplicação.

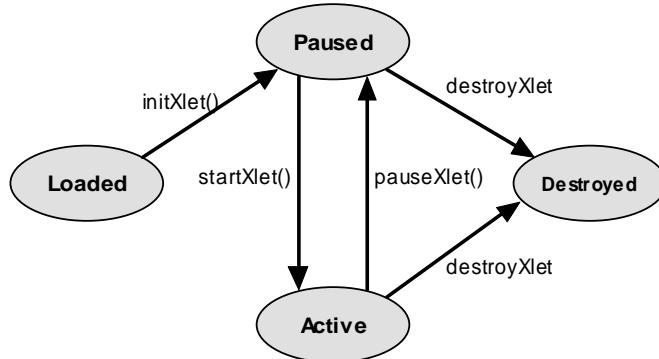


Figura 6.10. Ciclo de Vida de um Xlet.

Tabela 6.2 – Descrição dos Estados de um Xlet.

Estado	Descrição
<i>Loaded</i>	O Xlet foi carregado e não inicializado. Este estado é alcançado depois do Xlet ter sido criado (comando <i>new</i>). O Xlet realiza tipicamente pouca ou nenhuma atividade nesta etapa. Se uma exceção ocorrer, o Xlet imediatamente entra no estado <i>Destroyed</i> .
<i>Paused</i>	O Xlet é inicializado. Este estado é atingido a partir do estado <i>Loaded</i> após o método <i>Xlet.initXlet()</i> retornar sua chamada com sucesso, ou a partir do estado <i>Active</i> , após o método <i>Xlet.pauseXlet()</i> retornar sua chamada com sucesso, ou também a partir do estado <i>Active</i> , antes do método <i>XletContext.notifyPaused()</i> , que será detalhado mais adiante, retornar sua invocação com sucesso ao Xlet.
<i>Active</i>	O Xlet está funcionando normalmente e fornecendo seus serviços. Esse estado é alcançado a partir de estado <i>Paused</i> , após o método <i>Xlet.startXlet()</i> retornar sua chamada com sucesso.
<i>Destroyed</i>	O Xlet liberou todos os seus recursos e terminou sua execução. Esse estado é alcançado após o método <i>Xlet.destroyXlet()</i> retornar com sucesso, ou quando o método <i>XletContext.notifyDestroyed()</i> , que será detalhado mais adiante, retorna com sucesso ao Xlet.

O ponto de entrada de toda aplicação JavaTV é uma classe que implementa a interface *Xlet*. Esta interface fornece um gerente de aplicação com quatro métodos para sinalizar as mudanças de estado no ciclo de vida da aplicação para o Xlet. A API Xlet consiste de duas interfaces, *Xlet* e *XletContext*, que expressam a comunicação entre um Xlet e seu ambiente. O estado de um Xlet pode ser mudado pelo gerente da aplicação, através da invocação de um dos métodos da interface *Xlet* ou pelo próprio Xlet que notifica o gerente da aplicação de uma transição interna do seu estado através do objeto *XletContext*. A interface *Xlet*, cujos métodos são detalhados na Tabela 6.3, possui a seguinte estrutura:

```

public interface Xlet{
    public void initXlet(XletContext ctx) throws
        XletstateChangeException;

    public void startXlet() throws
        XletstateChangeException;

    public void pauseXlet();

    public void destroyXlet(Boolean uncondicional) throws
        XletstateChangeException;
}

```

Tabela 6.3. Métodos da Interface Xlet.

Método	Descrição
<i>initXlet</i>	Método com função parecida a de um construtor de classe, toda instância de objetos ou variáveis podem ser inicializados dentro do escopo deste método, um <i>container</i> pode ser criado para exibir interfaces gráficas ou conexões de rede necessárias podem ser abertas, em outras palavras, prepara a aplicação para ser executada. <i>InitXlet</i> é invocado somente uma vez no ciclo de vida do Xlet. É neste método que a aplicação recebe como parâmetro um objeto <i>Context</i> . Este objeto pode ser usado pelo Xlet para acessar as propriedades associadas ao seu ambiente, assim como para sinalizar ao gerente da aplicação uma mudança de estado. Se por alguma razão o Xlet não conseguir ser inicializado com sucesso, ele pode sinalizar esse fato ao gerente da aplicação lançando um <i>XletStateChangeException</i> . Caso contrário, o Xlet retorna o estado <i>Paused</i> .
<i>startXlet</i>	Método invocado para fazer a aplicação executar sua função principal. Este método deve implementar todos procedimentos necessários para que a aplicação entre no estado <i>Active</i> . É durante este estado que o Xlet gerencia seus componentes internos, manipula os objetos previamente criados e está apto a escutar todos os eventos internos e externos à aplicação. Se o Xlet não conseguir entrar no estado <i>Active</i> , ele pode lançar um <i>XletStateChangeException</i> , que notifica o gerente da aplicação que a mudança do estado falhou.
<i>pauseXlet</i>	Sinaliza a aplicação para que esta pare de fornecer seu serviço e entrar para o estado <i>Paused</i> . Se o Xlet não conseguir entrar no estado <i>Paused</i> , ele pode lançar um <i>XletStateChangeException</i> , que notifica o gerente da aplicação que a mudança do estado falhou.
<i>destroyXlet</i>	Neste método devem ser implementados todos os procedimentos necessários para que o Xlet termine sua execução de forma correta. Geralmente é usado para remover componentes gráficos da tela, liberar recursos como E/S de dados ou salvar dados.

Contexto de Xlets

Como já mencionado anteriormente, cada Xlet tem um contexto associado, fornecido por uma instância da classe de `javax.tv.xlet.XletContext`. Isto é similar à classe `AppletContext` que é associada com um applet. Em ambos os casos o contexto é usado para fornecer um mecanismo para a aplicação obter mais informação sobre seu ambiente e comunicar todas as mudanças em seu estado a esse ambiente. A interface `XletContext` possui a seguinte estrutura:

```
public interface XletContext {
```

```

        public static final String ARGS = "javax.tv.xlet.args"
        public void notifyDestroyed();
        public void notifyPaused();
        public void resumeRequest();
        public Object getXletProperty(String key);
    }
}

```

Os métodos `notifyDestroyed()` e `notifyPaused()` permitem que um Xlet notifique o receptor que está a ponto de terminar ou pausar a si próprio. O Xlet pode usar estes métodos para certificar-se de que o receptor está ciente sobre o estado de cada aplicação e pode realizar as ações apropriadas para cada estado. Estes métodos devem ser chamados imediatamente antes do Xlet entrar nos estados *Paused* ou *Destroyed*, para que o receptor possa realizar as ações necessárias.

Os passos seguintes mostram exatamente o que acontece quando um Xlet solicita uma mudança de estado através de seu contexto. Neste caso, o Xlet irá pausar a si mesmo e em seguida solicitar um reinício.

1. Primeiramente, o Xlet notifica seu contexto que pausou, invocando o método `XletContext.notifyPaused()`;
2. O contexto do Xlet passa então esta informação para o gerente da aplicação no *middleware*;
3. O gerente da aplicação atualiza seu estado interno para refletir a pausa da aplicação, e o Xlet vai “dormir”;
4. Quando a aplicação quer recomeçar a operação (o usuário pressionou uma tecla, por exemplo), ela invocando o método de `XletContext.requestResume()`;
5. Como ocorreu anteriormente, o contexto do Xlet passa este pedido para o gerente da aplicação;
6. O gerente da aplicação irá decidir se reinicia a aplicação ou não. Se for reiniciar, então ele irá atualizar seu estado interno para refletir esta mudança e invoca o método `startXlet()` do Xlet. Da mesma forma que todas as operações restantes que controlam o ciclo de vida do Xlet, este método é invocado diretamente do gerente da aplicação e não através do contexto do Xlet;
7. O Xlet recomeçará então a operação.

6.4 Testando Aplicações de TVDI

Uma das tarefas mais complicadas com relação ao desenvolvimento de aplicações MHP é se obter um ambiente de desenvolvimento que permita a construção e a execução de Xlets. Existem, basicamente, duas possibilidades para realizar esse desenvolvimento:

- Adquirir um sistema de desenvolvimento comercial. Estes sistemas são caros, contudo é fornecido um suporte técnico para o desenvolvedor.
- Utilizar um software emulador para executar os Xlets. Nessa abordagem é utilizado um software que permite a realização de teste de execução de Xlets no próprio PC.

Um emulador que pode ser usado para executar Xlets em um PC é o desenvolvido pela *Espial* [Espial 2007]. Esse emulador implementa parcialmente a especificação MHP. Além disso, implementa a API *JavaTV* e apresenta também uma implementação da API *HAVi*.

Uma outra opção é o emulador *mhp4free* [mhp4free 2007]. Criado para ser executado em Linux, é totalmente em alemão e possui também uma implementação da API *JavaTV* além das outras utilizadas pelo padrão MHP.

O OpenMHP [OpenMHP 2007], apesar de possuir boa quantidade de documentação e proporcionar um amplo debug do código testado, possui um baixo número de classes MHP

implementadas, causando assim problemas na execução devido a não implementação de métodos de diversas classes MHP, como widgets Java da classe HAVi.

O emulador mais popular que pode ser usado para executar Xlets em um PC é o *XleTView* [XleTView 2007]. Possui o código aberto sob a licença GPL, e além de uma implementação de referência da API *JavaTV*, traz consigo implementações de outras APIs especificadas no padrão MHP, como a HAVi, DAVIC e implementações especificadas do próprio DVB. Como é programado totalmente em Java, pode ser executado tanto em uma plataforma Linux como em Windows, bastando para isso utilizar o *Java 2 Standard Development Kit* para compilar Xlets e executar o *XleTView*.

6.4.1 Instalação do XleTView

Antes de utilizar o XleTView o desenvolvedor deve ter instalado a versão 1.4 ou superior da plataforma J2SE [SUN 2007]. Para desenvolver aplicações será necessário o SDK completo. Não há nenhum outro pré-requisito para utilizar o XleTView. Particularmente, não é necessário instalar a implementação de referência de JavaTV da Sun, ou qualquer outro software de TDVI. Esse emulador pode ser baixado da página do projeto XleTView no Sourceforge [XletTView 2007].

A instalação do XleTView é extremamente simples. É necessário apenas descompactar o arquivo contendo os binário no diretório onde ele deve ser utilizado. Após essa operação estarão presentes os seguintes arquivos e diretórios:

```
[config]
[jars]
[JMF2.1.1]
[license]
icon.ico
readme.txt
xletview.jar
```

O arquivo xletview.jar contém a aplicação XleTView. Nesse arquivo estão incluídas todas as classes da API MHP que são necessárias para o funcionamento das aplicações dentro do XleTView. O Arquivo readme.txt fornece algumas instruções de como iniciar a utilização do XleTView, e o arquivo icon.ico é o ícone usado pelo XleTView.

O diretório jars contém arquivos do tipo JAR de outros projetos que são usados pelo XleTView. Estes são customizados para o XleTView e não devem ser usados em outras aplicações. As licenças para estes arquivos jar, e para o próprio XleTView, estão contidas no diretório licenses. Uma exceção é o JMF, que está contido no diretório JMF2.1.1.

O diretório config contém todos os arquivos de configuração usados pelo XleTView, usados tanto para ajustar o próprio XleTView, como para configurar todas as aplicações a serão utilizadas nesse emulador. Uma descrição detalhada dessas configurações pode ser encontrada na página do XleTView.

6.4.2 Utilização do XleTView

Para executar o XleTView podem ser utilizadas duas abordagens. A primeira é clicar duas vezes no arquivo xletview.jar que se encontra no diretório raiz do arquivo gerado pelo zip. Com isso o emulador abrirá automaticamente (Figura 6.11). Nesse caso, não podem ser vistas as mensagens de debug (`System.out.println()`) que eventualmente são inseridas nos códigos dos Xlets. Para se ter acesso a essas mensagens o XleTView pode ser inicializado na linha de comando através do seguinte comando:

```
> java -jar xletview.jar
```

Nesse caso a janela de execução mostra todas as mensagens geradas pelo XleTView.

Para executar uma aplicação, o desenvolvedor deve ir para o menu “Applications” e escolha a aplicação que se quer executar. Este menu incluirá todas as aplicações que estão configuradas para executar no XleTView. No exemplo da Figura 6.11, temos dois grupos de aplicações, chamados “NASH” e “TVDI Exemplos”, sendo que o segundo grupo contém duas aplicações, denominadas “Menu Xlet” e “Xlet Simples”.

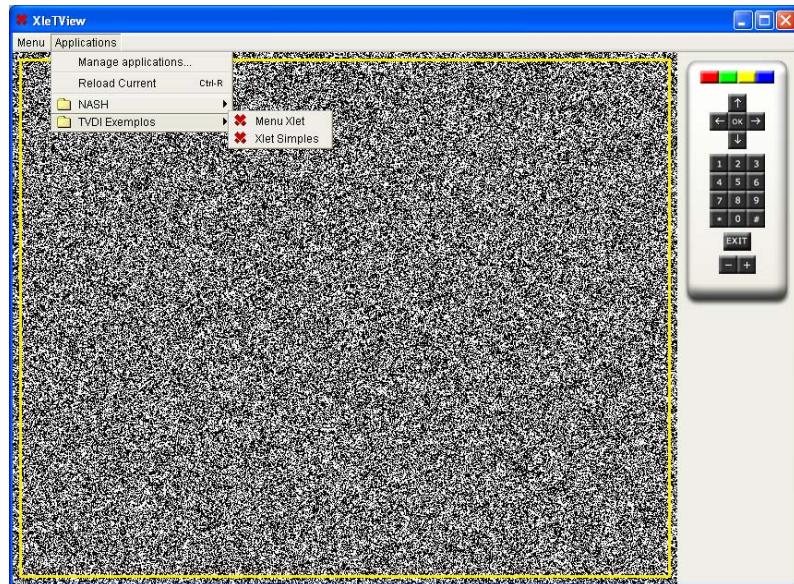


Figura 6.11. Tela do XleTView.

Quando uma aplicação é selecionada no menu, o XleTView inicializa automaticamente essa aplicação. Toda a saída gráfica da aplicação é mostrada na tela do simulador, e qualquer saída em System.out ou em System.err aparecerá no console (caso o XleTView tenha sido inicializado pela linha de comando). Escolhendo “Reload Current” no menu “Applications” (ou pressionando CTRL-R) a aplicação será reiniciada mas suas classes não serão recarregadas. Contudo, antes de executar uma aplicação, é necessário informar os dados das mesmas ao XleTView, de modo que esse saiba quais aplicações estão disponíveis e como carregá-las.

Em um receptor normal MHP as aplicações são sinalizadas através da tabela AIT. Isto não é possível com o XleTView, assim é necessária uma outra forma de colocar as aplicações disponíveis para execução. Para tanto, o desenvolvedor deve ir ao menu “Applications” e escolher a opção “Manage Applications...”. Essa ação mostrará o diálogo da Figura 6.12.

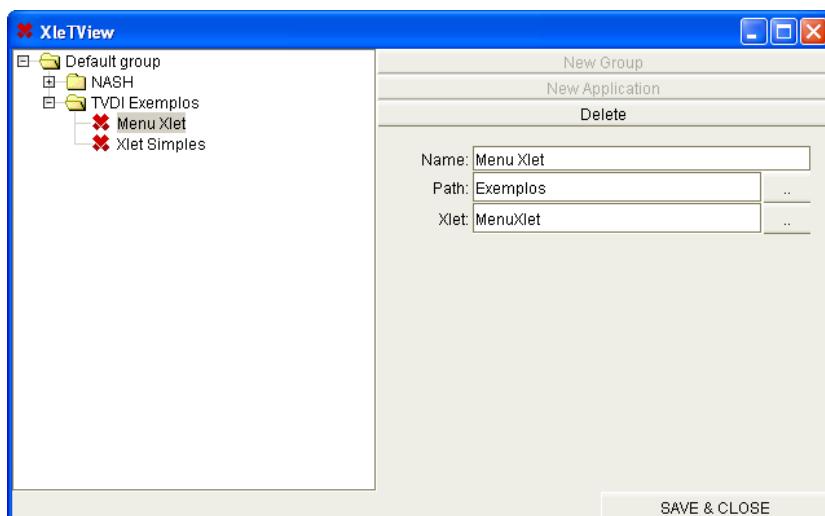


Figura 6.12. Gerenciando Aplicações no XleTView.

A partir da janela apresentada na Figura 6.12, podem ser definidas as aplicações e grupos de aplicações que serão executados pelo XleTView. Para definir um novo grupo, deve ser selecionado o grupo no painel da esquerda e pressionado o botão “New Group”. Para mudar o nome do novo grupo, deve ser selecionado o grupo e deve ser entrado o novo nome na caixa de diálogo.

Para adicionar uma aplicação deve se seguir a mesma estratégia. No painel da esquerda, deve ser selecionado o grupo que conterá a nova aplicação. Então, clicar no botão “New Application”. Isto criará uma entrada da nova aplicação. Selecionando esse caminho serão mostradas as propriedades dessa aplicação e permitirá que essas possam ser editadas. Nesse momento três propriedades estão disponíveis para cada aplicação:

- Nome: O nome que vai ser dado ao Xlet.
- Path: O caminho para o diretório raiz do Xlet
- Xlet: O nome da classe principal do Xlet

Estes são um subconjunto das propriedades que seriam sinalizadas na AIT de um serviço de transmissão. Atualmente não é possível enviar argumentos aos Xlets usando o XleTView. Uma vez que foram adicionadas as aplicações, o desenvolvedor deve clicar no botão “SAVE & CLOSE” para salvar as mudanças e para sair do diálogo. Uma maneira alternativa de configurar as aplicações no XleTView é usando o arquivo de configuração applications.xml, armazenado no diretório config. Mais informações sobre a utilização desse arquivo podem ser encontradas no site do XleTView.

Como nos STBs MHP, o usuário deve utilizar o controle remoto para interagir com as aplicações. Além disso, muitas das teclas de controle remoto também são mapeadas no teclado do PC, o que é útil se o desenvolvedor resolver controlar os Xlets usando uma unidade de controle remoto real conectada ao PC. As teclas numéricas do teclado estão mapeadas nos botões 0-9 no controle remoto, enquanto que as teclas de função de F1 a F4 correspondem aos botões coloridos vermelho, verde, amarelo e azul, respectivamente. As teclas de cursor correspondem aos quatro botões direcionais do controle remoto.

6.5 Exemplo de Aplicação

De forma a facilitar o entendimento do processo de desenvolvimento de aplicações em TVDI, será apresentada uma aplicação simples que permitirá a compreensão de alguns aspectos práticos desse tipo de aplicação. Nessa aplicação, é desenvolvido um Xlet simples no qual é apresentada uma caixa de texto cuja cor de fundo muda de acordo com a ativação dos botões de cores do controle remoto.

Inicialmente devem ser realizadas as importações de classes e pacotes Java a serem utilizados na implementação. Para esse Xlet são as seguintes:

```
import java.awt.Color;
import java.awt.Font;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

import xjavax.tv.xlet.Xlet;
import xjavax.tv.xlet.XletContext;
import xjavax.tv.xlet.XletStateException;

import org.havi.ui.HDefaultTextLayoutManager;
import org.havi.ui.HScene;
import org.havi.ui.HSceneFactory;
import org.havi.ui.HScreen;
import org.havi.ui.HStaticText;
```

```
import org.havi.ui.event.HRcEvent;
```

Inicialmente são importadas as classes da API AWT de Java para auxiliar o desenvolvimento da interface da aplicação, além da estrutura de eventos associados aos controles Java. Em seguida são importadas as classes da API JavaTV. Deve ser observada que, como estamos utilizando o XleTVView para compilar e executar as aplicações, os pacotes devem iniciar com x (xjavax). Em seguida são apresentadas as classes da API HAVi, que dão suporte aos componentes de interface a ao sistema de eventos relacionados à entrada de informações pelo usuário através do controle remoto.

```
public class CaixaTexto implements Xlet, KeyListener {  
    private XletContext context;  
    private HScene cena;  
    private HStaticText label;  
    public CaixaTexto() {  
    }  
    public void initXlet(XletContext xletContext) throws  
        XletStateChangeException {  
        System.out.println("Executa o metodo initXlet()");  
        context = xletContext;  
    }  
}
```

A classe que implementa o Xlet deve implementar os métodos das interfaces Xlet e KeyListener. A implementação da interface Xlet sinaliza que essa classe é um Xlet e deve ser gerenciado por um gerenciador externo. Já a implementação da interface KeyListener, permite a captura dos eventos de interface.

Em seguida são definidos os atributos da classe, que são do tipo XletContext, que permite o gerenciamento de contexto do Xlet; HScene, que é um controle HAVi similar a uma janela, mas com atributos mais restritos; e HStaticText, que também é um componente HAVi para exibição de texto

O construtor da classe não possui nenhum comando. Na verdade esses comandos são transferidos para o método initXlet. A classe inicia sua execução a partir desse método e nele apenas o contexto da aplicação é capturado.

```
public void startXlet() throws XletStateChangeException {  
    System.out.println("Executa o metodo startXlet()");  
    HSceneFactory hsceneFactory = HSceneFactory.getInstance();  
  
    cena = hsceneFactory.getFullScreenScene(  
        HScreen.getDefaultHScreen().getDefaulthGraphicsDevice());  
    cena.setSize(720, 576);  
    cena.setLayout(null);  
    cena.addKeyListener(this);  
  
    label = new HStaticText("Meu Primeiro Xlet !", 150, 150,  
        300, 200, new Font("Tiresias", Font.BOLD, 22),  
        Color.black, Color.red, new HDefaultTextLayoutManager());  
    cena.add(label);  
  
    cena.setVisible(true);  
    cena.requestFocus();  
}
```

No método startXlet, são instanciados os elementos a serem utilizados na aplicação. Inicialmente uma instância de HScene é criada através dos métodos de HSceneFactory. Em

seguida, o tamanho da cena e seu layout são definidos. Só então um *listener* é associado à cena para capturar os eventos de teclas que essa possa gerar.

Logo em seguida é criado um elemento de texto, denominado *label*, que é uma instância do componente HAVi `HStaticText`, cujo seguintes características podem ser definidas: texto a ser apresentado, posição, tipo de fonte, cor de frente, cor de fundo e layout de apresentação. Esse componente é inserido na cena, que o torna visível e apto a receber interações. É importante ressaltar que é recomendado que a fonte utilizada em aplicações de TV seja a “Tiresias” e com um tamanho mínimo de 21.

```
public void pauseXlet() {  
}  
  
public void destroyXlet(boolean flag) throws  
    XletStateChangeException {  
    System.out.println(" Executa o metodo destroyXlet()");  
    if (cena != null) {  
        cena.setVisible(false);  
        cena.removeAll();  
        cena = null;  
    }  
    context.notifyDestroyed();  
}
```

Os métodos `pauseXlet` e `destroyXlet` são implementados em seguida, sendo que o primeiro método não possui nenhum código associado. Já no método `destroyXlet` há a liberação dos recursos utilizados pela aplicação, no caso a cena. Em seguida o contexto do Xlet deve ser avisado sobre essa mudança de estado.

```
public void keyTyped(KeyEvent e) {  
}  
  
public void keyReleased(KeyEvent e) {  
}  
  
public void keyPressed(KeyEvent event) {  
    switch (event.getKeyCode()){  
        case HRcEvent.VK_COLORED_KEY_0:  
            label.setBackground(Color.red);  
            label.repaint();  
            break;  
  
        case HRcEvent.VK_COLORED_KEY_1:  
            label.setBackground(Color.green);  
            label.repaint();  
            break;  
  
        case HRcEvent.VK_COLORED_KEY_2:  
            label.setBackground(Color.yellow);  
            label.repaint();  
            break;  
  
        case HRcEvent.VK_COLORED_KEY_3:  
            label.setBackground(Color.blue);  
            label.repaint();  
    }  
}
```

```

        break;
    }
}
}

```

O tratamento dos eventos do controle remoto é realizado pela implementação dos métodos na interface KeyListener, que são keyTyped, keyReleased e keyPressed. Nesse exemplo será utilizado o método keyPressed para controlar as interações no usuário. Inicialmente a variável do event, que é do tipo KeyEvent, recebe as informações de eventos. O comando switch verifica que tipo de evento ocorreu, no caso dessa aplicação serão tratados apenas os eventos relacionados aos botões coloridos do controle remoto. Os códigos desses botões são descritos nas variáveis da classe HRCEvent, que permitem a identificação de todos os eventos relacionados ao controle remoto. Dessa forma, quando um botão de cor é acionado o fundo do label é pintado novamente com a cor do botão.

Depois de escrever todo o código, o desenvolvedor deve compilar a aplicação. Nesse caso é necessário, além do Java SDK instalado, apenas o XleTView, visto que esse emulador possui as classes necessárias para executar essa aplicação. Para tanto o desenvolvedor deve ou incluir o arquivo xletview.jar na variável de ambiente CLASSPATH, ou incluir esse arquivo diretamente na linha de comando para a execução do compilador Java. Assim, o comando para a compilação pode ser o seguinte:

```
> javac -classpath xletview.jar CaixaTexto.java
```

Caso não haja nenhum erro de compilação, a aplicação já pode ser instalada e executada no XleTView, como apresentado na seção 6.4.2. Como resultado será apresentada a tela inicial da Figura 6.13.

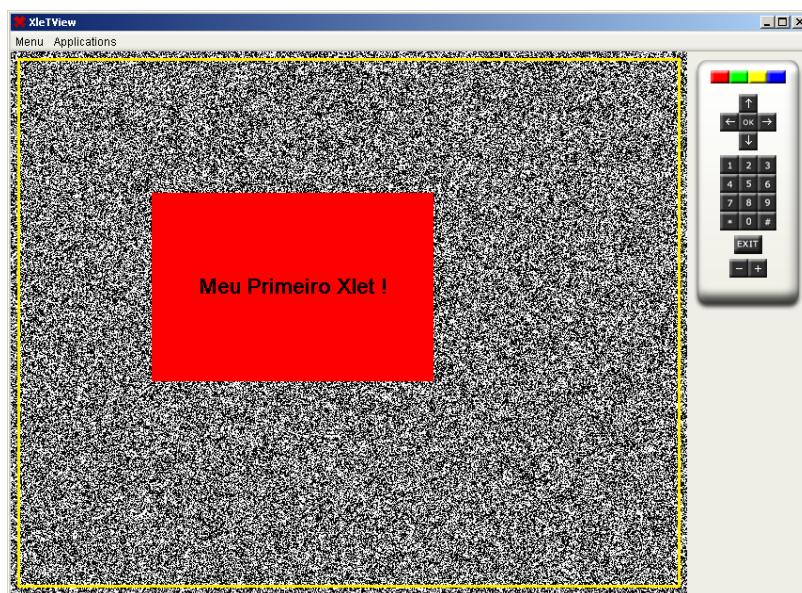


Figura 6.13. Aplicação Exemplo em Execução no XleTView.

Caso seja pressionado um botão colorido do controle remoto o fundo do texto será modificado de acordo com a cor pressionada. Também podem ser utilizadas as teclas de F1 a F4 para simular o controle remoto.

Existem várias tipos de controles diferentes que podem ser incluídos em aplicações, além de diversas formas de controlar eventos, como por exemplo à utilização de repositório de eventos. Contudo, de um modo geral, a estrutura das aplicações em TVDI seguem a estrutura da aplicação especificada nessa seção.

6.6 Conclusão

O sistema brasileiro de televisão aberta é um dos maiores do mundo. Uma de suas características mais importantes, sobretudo considerando a realidade sócio-econômica brasileira, é seu acesso totalmente gratuito para os telespectadores, sendo que cerca de 94% dos domicílios brasileiros possuem receptores de televisão, onde aproximadamente 80% recebem exclusivamente sinais de televisão aberta.

A televisão, além de ser a única fonte de lazer e informação da grande maioria da população brasileira, é o único serviço de comunicação realmente universal e gratuito no Brasil. Fato que mostra a inegável responsabilidade dessa mídia no que tange à cultura e à própria cidadania.

A TV Digital Interativa (TVDI) aberta, que está em discussão atualmente no Brasil, se tornará uma realidade brevemente. Com essa tecnologia serão disponibilizadas imagens com alta definição, som digital, interatividade, múltiplos programas simultaneamente, venda de produtos via televisão, jogos, entre outras funcionalidades. Nesse sentido a disseminação da televisão aberta digital é de importância estratégica para o Brasil, uma vez que beneficiará quase toda a população nos próximos anos, proporcionando crescente democratização do acesso à informação.

Contudo, um aspecto essencial dessa visão de futuro é que a questão da TVDI não se resume à definição de um padrão, instalação de infra-estrutura física, ou especificação de circuitos ou equipamento. Um sistema de TVDI depende fortemente do desenvolvimento de novas soluções de software. Assim, os reais benefícios da TVDI só serão observados com o aumento da produção de conteúdo de alta definição e com o desenvolvimento de aplicações de interatividade.

Entretanto, desenvolver software para TVDI é completamente diferente de desenvolver software para PCs. Nesse sentido, esse texto apresentou uma visão geral do processo de desenvolvimento de aplicações interativas para TVDI, no qual são tratados os principais aspectos relacionados a essa nova mídia.

Referências Bibliográficas

Becker, V. e Montez, C. *TV Digital Interativa: Conceitos, desafios e perspectivas para o Brasil*. 1. ed. Florianópolis: I2TV, 2004.

ATSC. *Advanced Television Systems Committee*. Disponível em: <http://www.atsc.org>. Acesso em: Junho de 2007.

Davenport, D. *Net4TV Voice: First Look at AOLTV*. 2000.

DAVIC. *Digital Audio-Video Council*. Disponível em: <http://www.davic.org>. Acesso em: Junho de 2007.

DIBEG. *Integrated Services Digital Broadcasting*. Disponível em: <http://www.dibeg.org>. Acesso em: Junho de 2007.

DVB. *Digital Video Broadcasting Project*. Disponível em: <http://www.dvb.org>. Acesso em: Junho de 2007.

Fernandes, J.; Lemos, G. e Silveira, G. Introdução à televisão digital interativa: Arquitetura, Protocolos, Padrões e Práticas. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 24., 2004.

ESPIAL. Espial Suíte. Disponível em: <http://www.espijal.com>. Acesso em: Junho de 2007.

FREEVO. Disponível em: <http://freevo.sourceforge.net>. Acesso em: Junho de 2007.

HAVi. Home Áudio-Video Council. Disponível em: <http://www.havi.com>. Acesso em: Junho de 2007.

JAVATV. The Java TV Application Programming Interface. Disponível em: <http://java.sun.com/products/javatv>. Acesso em: Jun. de 2007..

SUN. Java 2 Platform, Standard Edition (J2SE). Disponível em: <http://java.sun.com/j2se/1.4.2/> Acesso em: Junho de 2007.

Jucá, P., Lucena, U. Experiências no desenvolvimento de aplicações para televisão digital interativa. In: FORUM DE OPORTUNIDADES EM TELEVISÃO DIGITAL INTERATIVA, 3., 2005, Poços de Caldas.

MC. O que é o Sistema de TV Digital Terrestre. Disponível em: <http://sbtdv.cpqd.com.br/?obj=historico&mtd=texto&item=2>. Acesso em: Junho de 2007.

MHEG. Multimedia, Hypermedia Expert Group. Disponível em: <http://www.mheg.org/>. Acesso em: Junho de 2007.

MHP, Multimedia home platform. Disponível em: <http://www.mhp.org>. Acesso em: Junho de 2007.

MHP4FREE. Disponível em: <http://www.mhp4free.de>. Acesso em: Junho de 2007.

OpenMHP. OpenMHP Propject. Disponível em: <http://www.openmhp.org>. Acesso em: Junho de 2007.

Peng, C.; Cesar, P.; Vuorimaa, P. Integration of Applications into Digital Television Environment, In: INTERNATIONAL CONFERENCE ON DISTRIBUTED MULTIMEDIA SYSTEMS, 7., 2001, Taipei.

XleTView. Disponível em: <http://sourceforge.net/projects/xletview/>. Acesso em: Junho de 2007.

Zhang, Y. A Java 3D framework for digital television set-top box. 2003. Tese (Mestrado em Telecomunicações, Software e Multimídia) Helsinki University of Technology, Finlândia.

Chapter

7

Processamento de Consultas em Redes de Sensores Sem Fio

Angelo Brayner e Aretusa Lopes

Resumo

Uma Rede de Sensores Sem Fio (RSSF) consiste de um ou mais grupos de dispositivos de sensoriamento sem fio ou de aquisição contínua de dados interligados em rede. Esses tipos de dispositivos sensores normalmente possuem arquitetura simples e altamente especializada, sendo comumente utilizados em aplicações relacionadas à área ambiental, médica, industrial e militar. Uma RSSF pode ser genericamente caracterizada por: (i) possuir uma grande quantidade de nós densamente organizados; (ii) adotar uma comunicação broadcast; (iii) ter uma topologia freqüentemente alterada e; (iv) ser composta por sensores de arquiteturas simples, propensos a falhas, com restrições de energia, memória e capacidade de processamento. Aplicações executadas sobre redes de sensores normalmente trabalham com fluxos contínuos de dados. Um sensor utilizado para coleta de temperaturas, por exemplo, pode ser configurado para captar informações do ambiente continuamente. Neste caso, o volume de dados recepcionado é potencialmente infinito, inviabilizando o seu completo armazenamento. A alternativa mais viável consiste em injetar consultas na RSSF, para que estas sejam executadas de forma distribuída por nós da RSSF. Este capítulo apresenta aspectos do processamento de consultas em redes de sensores sem fio, bem como abordagens utilizadas para o processamento de tais consultas.

7.1. Introdução

Uma Rede de Sensores Sem Fio (RSSF) consiste de um ou mais grupos de dispositivos de sensoriamento sem fio ou de aquisição contínua de dados interligados em rede [1]. Esses tipos de dispositivos sensores normalmente possuem arquitetura simples e altamente especializada, sendo comumente utilizados em aplicações relacionadas à área ambiental, médica, industrial e militar. Uma RSSF pode ser genericamente caracterizada por: (i) possuir uma grande quantidade de nós densamente organizados; (ii) adotar uma comunicação broadcast; (iii) ter uma topologia freqüentemente alterada e; (iv) ser composta por sensores de arquiteturas simples, propensos a falhas, com restrições de energia, memória e capacidade de processamento [1].

Basicamente, uma RSSF é responsável pela disseminação dos dados coletados pelos sensores, embutidos nos nós da rede (nós-sensores), fazendo com que estes dados cheguem ao usuário requisitante da informação. Algumas características comuns a estas redes incluem [27]:

1. capacidade de auto-organização;
2. utilização de nós-sensores com limitações de energia, poder de comunicação, memória e capacidade de processamento;
3. comunicação broadcast, utilizando freqüências de rádio e protocolos de roteamento do tipo múltiplos saltos;
4. existência de um grande número de nós-sensores densamente organizados em uma área geográfica;
5. realização de trabalho colaborativo entre os nós-sensores; e
6. alteração dinâmica da topologia da rede, devido a falhas e/ou mudanças na localização dos nós.

Neste capítulo, são investigadas as características físicas dos nós-sensores (Seção 7.2), aspectos a serem avaliados no projeto de RSSFs (Seção 7.0). As seções 7.4, 7.5, 7.6 e 7.7 abordam aspectos relativos ao processamento de consultas em RSSFs. Finalmente, a seção 7.8 tece alguns comentários sobre este capítulo à título de conclusão.

7.2. Nós-sensores

Um nó sensor é um dispositivo capaz de coletar dados através da detecção de sinais ou de eventos ocorridos no meio ambiente, atividades conhecidas como sensoriamento. Outras tarefas destes dispositivos em uma rede de sensores incluem a recepção e o envio de pacotes através da rede, com o objetivo de atender a requisições do usuário. Adicionalmente, nós-sensores inteligentes são dotados com processador e memória, sendo capacitados a avaliar e transformar os dados coletados. Além da restrita capacidade computacional e de armazenamento, estes equipamentos têm o seu tempo de vida limitado à disponibilidade de energia da sua bateria. Por este motivo, uma das principais preocupações dos algoritmos projetados para os nós-sensores é reduzir o consumo de energia da bateria, como forma de aumentar a sobrevida do nó.

Sensores mais simples são capazes apenas de captar informações do ambiente e enviá-las a outros nós da rede. Um nó-sensor é composto, basicamente, de um dispositivo de sensoriamento, para captar informações do ambiente; um transceptor, que possibilita a transmissão e recepção de dados; e uma fonte de energia, que alimenta os demais componentes do nó-sensor. Neste tipo de arquitetura, todos os dados coletados do ambiente pelo nó-sensor são enviados através da rede. Assim, se for considerada uma RSSF com um grande número de nós-sensores, o tráfego de pacotes na rede tende a ser muito grande.

Uma evolução dos dispositivos-sensores mais simples é o nó-sensor inteligente, que conta com um microprocessador embutido em sua estrutura. Estes nós são capacitados a processar os dados coletados, antes de enviá-los a outros nós [1].

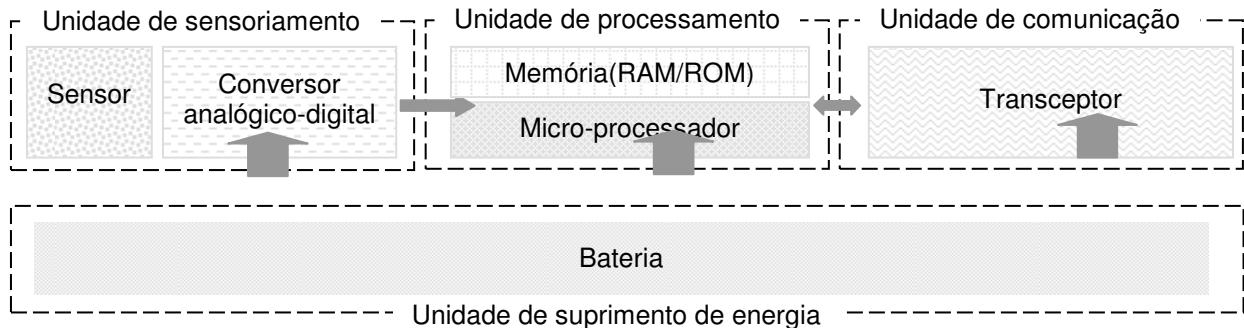


Figura 7.1. Componentes de um nó-sensor inteligente

O processamento de dados pode incluir, por exemplo, a aplicação de técnicas de sumarização dos dados, como forma de reduzir o tamanho e o número dos pacotes enviados através da rede. A Figura 7.1 mostra os componentes físicos básicos de um nó-sensor inteligente [1]. Observa-se que a energia da bateria alimenta as unidades de sensoriamento, processamento e comunicação.

A unidade de sensoriamento é responsável por coletar as informações do ambiente, sendo constituída por um dispositivo-sensor e um conversor de sinal analógico-digital. Quando um evento específico é detectado, o conversor analógico-digital converte o sinal analógico, produzido pelo dispositivo-sensor, em um sinal digital, o qual alimenta a unidade de processamento [1].

A “inteligência” do nó sensor é proveniente da unidade de processamento. Esta unidade possui um microprocessador com memória RAM (volátil), que garante ao nó-sensor a capacidade de processar e armazenar dados temporariamente.

A unidade de comunicação conecta o nó-sensor à RSSF, permitindo a transmissão e recepção de pacotes de dados. O dispositivo que constitui esta unidade, denominado transceptor, pode realizar a comunicação sem fio do nó-sensor através de um meio ótico (emissão de raios infravermelho) ou utilizar freqüências de rádio. O infravermelho é um meio de comunicação barato e fácil de construir, porém, requer um sinal dedicado entre transmissor e receptor [59]. Transceptores baseados em freqüências de rádio requerem modulação, banda passante, filtragem, demodulação e multiplexação de circuitos, o que coloca este meio de comunicação entre os mais caros e complexos. Ainda assim, a freqüência de rádio é o meio mais utilizado em RSSFs, devido a algumas características comuns a estas redes que favorecem o uso da freqüência de rádio. Estas características incluem o uso de pacotes pequenos, baixas taxas de transmissão e curtas distâncias entre nós-sensores que se comunicam, o que permite o reuso da freqüência do sinal na transmissão e recepção de pacotes [1].

A unidade de suprimento de energia é um dos componentes mais importantes do nó-sensor. Algumas baterias que compõem estas unidades de suprimento são recarregáveis (por exemplo, via uso de células solares), porém, a maior parte das arquiteturas de nós-sensores utiliza baterias de tempo de vida limitado.

Além dos componentes mostrados na Figura 7.1, também podem fazer parte da estrutura física dos nós-sensores outras subunidades como, por exemplo, subunidades responsáveis por fornecer a localização ou permitir a sincronização na recepção e envio de pacotes entre os nós-sensores. Muitas aplicações e técnicas de roteamento requerem o conhecimento preciso da localização do nó-sensor. Neste caso, o nó-sensor deve contar com um sistema de localização como o GPS (Global Position System). Todavia, equipar cada nó-sensor com um GPS pode não ser uma alternativa viável para RSSFs, devido aos custos de se implantar o GPS em um grande número de nós. Uma alternativa é utilizar o GPS apenas em alguns nós-sensores da rede, os quais ajudariam os seus nós-vizinhos a calcularem suas próprias localizações [1]. Um outro componente possível de um nó-sensor é a subunidade de sincronização, que sincroniza a recepção e envio de pacotes através da rede, o que requer o uso de relógios internos pelos nós da RSSF. Aspectos relacionados à sincronização em redes de sensores são explorados em [62].

7.3. Redes de Sensores Sem Fio (RSSFs)

Devido às características particulares das RSSFs, principalmente as restrições de recursos físicos, os algoritmos e protocolos projetados para redes convencionais não são aplicáveis a estas redes. O projeto de uma RSSF envolve a análise de aspectos relacionados aos nós-sensores, tais como consumo de energia, tolerância a falhas, sensoriamento e processamento de dados, e outros relacionados à estrutura da rede, tais como topologia, escalabilidade, comunicação e roteamento. Estes aspectos são descritos a seguir.

7.3.1. Consumo de energia

O aspecto mais importante a ser considerado no projeto de uma RSSF refere-se ao consumo de energia. Cada nó-sensor tem seu tempo de vida ditado pela disponibilidade de energia da sua bateria. Este fato leva a uma consequente limitação do tempo de vida da rede como um todo, caso não seja adotada uma estratégia de reposição dos nós da rede ou de recarga da bateria dos nós-sensores. Vale salientar que a recarga ou substituição da bateria não é uma opção fácil para estas redes, visto que elas geralmente são compostas por centenas ou até milhares de nós [1]. Desta forma, algoritmos, protocolos e componentes físicos projetados para RSSFs devem ter como principal objetivo a minimização do consumo de energia da bateria dos nós-sensores.

A partir da Figura 7.1, é possível identificar três fatores a serem considerados no consumo de energia em nós-sensores: sensoriamento, que é o objetivo primário de um sensor; processamento, que torna os nós-sensores dispositivos inteligentes, capazes de aplicar operações como, por exemplo, agregações e filtragens, sobre os dados coletados do ambiente; e comunicação, responsável pelas operações de transmissão e recepção, essenciais para formar uma RSSF [1]. Embora todas estas atividades consumam energia, é importante notar que o custo de energia para a transmissão ou recepção de um único dado sobre um meio sem fio, a pequenas distâncias, é muito maior do que o requerido para processar este dado ou coletá-lo do ambiente (por exemplo, o processamento de 3000 instruções pode ser executado, aproximadamente, com o mesmo gasto advindo do envio de 1 bit a uma distância de 100m) [62]. Assim, o caminho mais eficiente para diminuir o consumo de energia em uma RSSF é através da redução dos custos com comunicação [39].

Em geral, o transceptor (rádio-transceptor) de um nó-sensor pode operar em quatro modos distintos: Transmissão, Recepção, Ocioso, e Inativo. Cada um destes modos é caracterizado por diferentes taxas de consumo de energia. O modo Ocioso consome quase a mesma quantidade de energia que o modo de Recepção. Logo, o transceptor deve ser completamente desligado, colocado em modo Inativo, quando nenhum dado estiver sendo transmitido ou recebido, ou quando houver longos intervalos de tempo entre coletas sucessivas de dados; pois, é apenas no modo Inativo que a energia da bateria pode ser temporariamente conservada. Todavia, é importante observar que, no modo de operação Inativo, o nó-sensor não pode ser contactado por outros nós, o que muda a topologia ativa da rede [45].

Algumas alternativas, como a apresentada em [49], defende o uso de nós-sensores com múltiplos rádio-transceptores. Neste caso, um rádio com baixo consumo de energia é usado, exclusivamente, para ativar o rádio de mais alta potência, quando for necessário realizar a transmissão e recepção de dados. Desta forma, o nó-sensor não fica inacessível, pois o rádio-transceptor de mais baixa potência permanece ligado continuamente.

Outro aspecto a ser considerado é que, na maior parte das arquiteturas de rádio-transceptores, a simples mudança entre modos de operação causa uma dissipação significativa de energia, por exemplo, devido à mudança do modo Inativo para o modo de Transmissão. Portanto, é benéfico operar com pacotes tão grandes quanto possível, objetivando amortizar tanto o número de mudanças no modo de operação do transceptor quanto o overhead fixo sobre a manipulação de uma quantidade maior de bits (por exemplo, o cabeçalho do pacote) [45]. Por outro lado, um longo intervalo de tempo sem transmissão aumenta a latência total na troca das informações, o que pode não ser aceitável para uma dada aplicação.

Alternativas para obter uma maior economia de energia em RSSFs requerem o uso de algoritmos “conscientes” da disponibilidade de energia, projetados para ajustar seu comportamento a eventos específicos, como, por exemplo, restrições de recursos. Estes algoritmos deveriam ser capacitados a fazer uma análise dinâmica entre o consumo de energia, desempenho do sistema e fidelidade dos dados produzidos para o usuário. Estratégias de otimização no uso de energia podem ser aplicadas aos nós, ao canal de comunicação ou à rede como um todo, tendo por objetivo o aumento do tempo de vida da RSSF [45].

7.3.2. Tolerância a falhas

Considerando as restrições de energia da bateria dos nós-sensores e as condições extremas a que, muitas vezes, estes nós são expostos (por exemplo, implantados no fundo de oceanos, no interior de vulcões ou presos a corpos de animais), observa-se que os nós-sensores são altamente susceptíveis a falhas. Porém, a falha de um ou alguns nós-sensores não deve ser suficiente para comprometer o funcionamento da rede [1] ou distorcer a precisão dos resultados apresentados ao usuário.

A falha de um sensor que faz parte da rota de um pacote não deve impedir que o pacote chegue ao seu destino, visto que a topologia da rede deve dar caminhos alternativos para que os pacotes alcancem os seus destinos, seja através da adoção de mecanismos de retransmissão de pacotes ou apenas pela duplicação destes pacotes na rede. Para garantir a disponibilidade de rotas alternativas para os pacotes, os protocolos de roteamento precisam

reorganizar a rede periodicamente, traçando novas rotas em substituição àquelas interrompidas pela falha de nós-sensores, buscando, assim, uma maior tolerância a falhas.

7.3.3. Sensoriamento

A função primária de uma aplicação de RSSF é a atividade de sensoriamento, provida pelos sensores integrantes dos nós da rede. O tipo de informação a ser coletada depende do dispositivo físico de sensoriamento utilizado, que, por sua vez, é escolhido com base no objetivo da aplicação, como, por exemplo, coletar medições de temperatura. No contexto das aplicações de RSSFs, a forma de obtenção dos dados pode ser classificada como [47]:

1. Contínua, quando os dados são coletados continuamente;
2. Reativa, quando os dados são fornecidos em resposta a uma consulta do usuário ou a um evento específico do ambiente; e
3. Periódica, quando os dados são coletados segundo condições previamente configuradas na aplicação.

Algumas propostas, como a elaborada no Projeto TinyDB [39], da Universidade da Califórnia, suportam modelos híbridos, que consistem na coexistência de diferentes tipos de coleta de dados.

7.3.4. Processamento

O processamento de dados em nós-sensores é uma importante propriedade dos nós-sensores inteligentes. Conforme já mencionado, o consumo de energia com comunicação deve ser prioritariamente minimizado. Tendo em vista que este custo é proporcional ao volume total de dados transmitido através da rede, processamentos aplicados aos dados podem levar a reduções significativas do volume de dados enviado através da rede.

O processamento de dados em nós-sensores pode envolver operações de agregação, filtragem e outras técnicas de sumarização ou análise de dados, como as apresentadas na Subseção 3.2.2. Convém salientar que o processamento pode envolver não apenas operações aplicadas aos dados coletados do ambiente, mas também a fusão destes dados com aqueles recebidos de outros nós-sensores, originando novos pacotes de dados [47].

A agregação em rede [15][41][50] vem sendo a estratégia mais explorada na redução do volume de dados transmitido dos nós-sensores para a estação-base. Esta estratégia consiste na aplicação progressiva de operações de agregação aos dados, à medida que os pacotes vão sendo passados de um nó para outro na rede. Desta forma, a agregação de dados é efetuada ao longo de cada rota, por onde passam os pacotes.

A Figura 7.2 ilustra um exemplo de como funciona a agregação em rede. Neste exemplo hipotético é mostrada a planta baixa de um shopping center. Considera-se que cada entrada de veículo no estacionamento do shopping center é monitorada por um dispositivo-sensor, que faz parte de uma RSSF; e que um usuário, operando a estação-base, requisita o número de veículos que estão no estacionamento do shopping center em um determinado instante. Em resposta, os valores da contagem de veículos, computados em cada nó-sensor, são progressivamente somados, à medida que vão sendo passados através da rede, até chegarem à estação-base, onde o resultado final para a consulta é produzido.

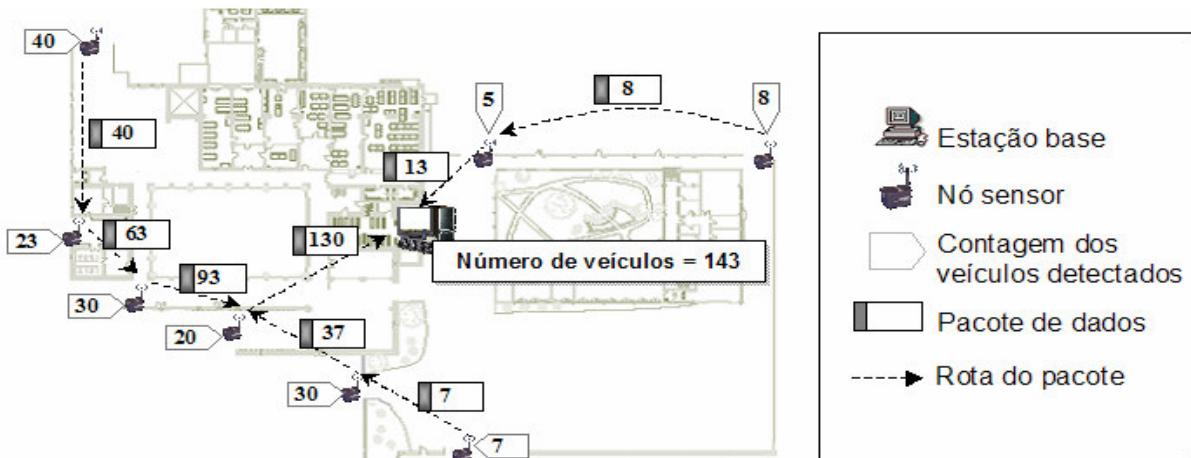


Figura 2. Exemplo de agregação em

7.3.5. Topologia

O número de nós-sensores em uma RSSF pode vir a ser muito grande e a densidade destes nós, na região geográfica considerada, pode ser alta (por exemplo, superior a 20 nós/m³). A distribuição dos nós-sensores no meio ambiente será mais ou menos aleatória, de acordo com a estratégia utilizada na implantação dos nós, por exemplo, de forma manual ou despejados a partir de aviões [1].

Após a primeira etapa de implantação dos nós de uma RSSF, alguns nós-sensores podem vir a mudar de localização, devido às condições do ambiente, por exemplo, devido a ventos ou deslizamentos, o que pode alterar a disposição física destes nós na rede. Além disso, seja por necessidade de reposição dos nós-sensores falhos ou por necessidade de crescimento da rede, novos nós poderão vir a ser implantados. Estas alterações de localização dos nós-sensores, bem como as manutenções realizadas na RSSF, requerem uma reorganização da estrutura da rede e, consequentemente, a atualização das rotas de transmissão dos pacotes por parte dos protocolos de roteamento.

7.3.6. Escalabilidade

Em uma RSSF, os protocolos da rede devem ser capazes de trabalhar não apenas com o número inicial de nós-sensores implantados no meio ambiente, mas também garantir a escalabilidade da rede; o que implica que o desempenho da aplicação não deverá ser fortemente afetado pelo aumento do tamanho da rede.

Adicionalmente, protocolos de roteamento podem ser escaláveis o suficiente para responder a eventos específicos do ambiente. Nesta estratégia, proposta em [2], a maior parte dos nós-sensores permanece no modo Inativo, sendo que apenas alguns nós-sensores da RSSF permanecem ativos para processar dados de pacotes remanescentes. Durante este período, os resultados fornecidos ao usuário serão menos precisos, visto que têm por base dados coletados apenas por alguns poucos nós-sensores ativos. Quando um determinado evento, configurado na aplicação, ocorre, todos os nós-sensores são ativados e a aplicação passa a dispor de resultados mais precisos. Esta estratégia favorece a economia de recursos

do nó-sensor em detrimento da precisão dos resultados, porém, a produção de resultados mais precisos é garantida nos períodos estabelecidos pelos eventos configurados na aplicação.

7.3.7. Comunicação

A comunicação não apenas permite o processamento colaborativo entre os nós-sensores, mas também a interação entre os usuários da aplicação e a rede. Em Ruiz et al. [47] são apontados dois tipos de comunicação: de infra-estrutura e de aplicação. A comunicação requerida para manutenção da infra-estrutura da rede objetiva manter atualizadas as rotas de comunicação, independente das falhas dos nós-sensores, da mobilidade destes nós ou do crescimento da rede (inclusão de novos nós). Já a comunicação requerida pela aplicação refere-se à disseminação dos dados pela rede. Durante a disseminação dos dados, a energia gasta na recepção e transmissão de um pacote tem um custo fixo, relacionado ao hardware. Para a operação de transmissão, também está associado um custo variável, que depende da distância entre o nó-origem da informação e o nó-destino. Desta forma, quanto mais próximos estiverem dois nós-sensores, menor será o consumo de energia necessário para realizar a comunicação entre eles, pois a potência dissipada será menor.

Além da distância entre os nós da rede, um outro fator crítico, relacionado à comunicação, é volume de dados que converge para o ponto no qual as informações são entregues à estação-base. Embora a estação-base seja robusta quanto à memória, disponibilidade de energia e capacidade de processamento, o tráfego de pacotes gerado por um número grande de nós-sensores para esta estação pode ser pesado. Portanto, a estação-base pode representar um gargalo que penaliza o throughput da rede. A perda de pacotes, seja por falha de alguns nós da rede ou pela incapacidade da estação-base em receber todos os pacotes a ela enviados, pode vir a gerar diferenças nos resultados apresentados ao usuário a cerca do fenômeno que está sendo considerado. Este aspecto é analisado em alguns trabalhos [27][47][58] através da garantia da qualidade de serviço (QoS – Quality of Service), referindo-se à capacidade da rede de produzir resultados corretos e em tempo aceitável.

A capacidade de esforço coletivo dos nós-sensores representou um grande avanço para as RSSFs. Os protocolos de comunicação provêem a infra-estrutura necessária para que o processamento colaborativo seja possível. Em Akyildiz et al. [1] é proposto que, além das camadas convencionais de redes (física, enlace, rede, transporte e aplicação), também sejam considerados componentes destinados ao:

- (i) **Gerenciamento de energia:** monitora o consumo de energia dos nós-sensores. A informação fornecida por este componente poderia, por exemplo, sinalizar para um nó-sensor enviar um aviso (por *broadcast*) aos seus vizinhos, informando que seu nível de energia esteja baixo e, portanto, não seja capaz de participar do roteamento de pacotes. Assim, a energia remanescente neste nó-sensor apenas passaria a ser utilizada em atividades de sensoriamento, processamento e envio de seus próprios dados;
- (ii) **Gerenciamento de mobilidade:** monitora a movimentação dos nós-sensores. Este monitoramento consistiria em detectar e registrar o movimento de um dado nó-sensor. A partir desta informação, o nó-sensor poderia ser capacitado a

identificar os seus novos nós-vizinhos e manter atualizadas as suas rotas de acesso à estação-base; e

- (iii) **Gerenciamento de tarefas:** monitora a distribuição de tarefas entre os nós-sensores. Basicamente, este componente faria o balanceamento das tarefas de sensoriamento em uma dada região geográfica. Por exemplo, nem todos os nós-sensores de uma região precisam ser utilizados para responder a uma solicitação do usuário, o que pode ocorrer quando dois ou mais nós-sensores compartilham a mesma área de sensoriamento. Neste caso, os nós-sensores com maior disponibilidade de energia podem ser selecionados para coletar os dados do ambiente, poupando recursos dos demais nós-sensores [1].

Esses componentes de gerenciamento contribuem para que cada nó-sensor seja capaz de controlar sua atividade de sensoriamento, baixar seu consumo de energia, permitir o compartilhamento de seus recursos e o trabalho colaborativo no roteamento dos pacotes.

7.3.7.1. Roteamento

Em Al-karaki et al. [2] é sugerida uma divisão para os protocolos de roteamento de RSSFs em três classes:

- (i) plana, na qual todos os nós-sensores da rede desempenham o mesmo papel e possuem as mesmas características físicas;
- (ii) hierárquica, na qual grupos de nós da rede têm diferentes responsabilidades e disponibilidades de recursos e;
- (iii) adaptativa, que permite ao protocolo adaptar-se às condições da rede e à disponibilidade dos níveis de energia dos nós-sensores, através da configuração de parâmetros que mapeiam características físicas dos nós da RSSF.

A classe de protocolos hierárquica vem sendo largamente explorada como estratégia capaz de poupar recursos dos nós-sensores. Estes protocolos definem que os dados sejam passados de um nó para outro, através de roteamento do tipo múltiplos saltos, até alcançarem a estação-base. Estações-base são a chave para uma comunicação do tipo backbone, servindo como gateway para outras redes, o que permite a integração entre diferentes redes [53]. O problema em utilizar uma estrutura hierárquica em RSSFs é que o tráfego torna-se cada vez mais intenso, à medida que os pacotes aproximam-se da estação-base. Assim, enquanto os nós-sensores mais distantes da estação-base apenas coletam dados do meio ambiente, os nós mais próximos têm seus recursos penalizados, devido ao fato de precisarem tanto coletar dados do ambiente como receptionar pacotes de outros nós da rede.

Estratégias de endereçamento convencionais, como o uso de IPs (Internet Protocols), não são aplicáveis a RSSFs porque demandariam um alto custo para atribuição e manutenção de endereços, tendo em vista o grande número de nós destas redes. O roteamento de pacotes em RSSFs também deve oferecer suporte à mobilidade dos nós. Tarefa difícil caso o endereço do nó não forneça nenhuma informação sobre a direção em que o pacote deve ser roteado. Uma alternativa é manter um servidor central, capaz de manter atualizadas informações sobre a posição de todos os nós da rede. Outra alternativa é utilizar um agente de endereçamento, associado a cada nó, capaz de manipular e

redirecionar todas as requisições destinadas a um dado nó para a sua localização corrente [26]. Na proposta do PicoRadio [44], os nós podem ser endereçados com base em suas posições geográficas. A localização dos nós é uma informação muito útil, pois permite que os protocolos de roteamento enviem os pacotes na direção correta.

As subseções a seguir detalham algoritmos que implementam protocolos de roteamento para RSSFs.

- **Flooding**

Esta é uma estratégia clássica de roteamento que consiste em um nó-sensor enviar pacotes para todos os seus vizinhos, independente do fato destes vizinhos já terem ou não recebido os mesmos pacotes. Neste caso, são feitas cópias de cada novo pacote para todos os vizinhos do nó, exceto aquele do qual ele acaba de receber o pacote. A maior vantagem deste algoritmo está na simplicidade quanto ao gerenciamento das rotas e endereços. Todavia, três deficiências não tratadas pelo flooding e enumeradas em Heinzelman et al. [31] desencorajam o uso deste protocolo em RSSFs:

- (i) Imploração: situação na qual pacotes duplicados são enviados para o mesmo nó, desperdiçando os recursos físicos da rede;
- (ii) Sobreposição: fato que ocorre quando dois ou mais nós compartilham a mesma área geográfica e, portanto, coletam a mesma informação, gerando dados duplicados para a aplicação; e
- (iii) Desconhecimento dos recursos físicos: neste caso o protocolo não é capacitado a adaptar seu comportamento à disponibilidade de recursos físicos do nó-sensor, como, por exemplo, energia.

- **Gossiping**

Na proposta do gossiping [28], cada pacote p_1 de um nó n_1 é enviado apenas para um dos vizinhos de n_1 , o qual é selecionado aleatoriamente. Desta forma, p_1 é propagado através da rede até alcançar a estação-base.

O uso do gossiping consome menos energia do que o flooding, pois permite reduzir o número de pacotes enviados por nó. Outra vantagem desta estratégia é a resolução do problema da implosão de pacotes, o que é conseguido graças ao fato de que apenas uma cópia de p_1 é mantida na rede, a cada intervalo de tempo. Todavia, o gossiping não trata alguns aspectos como:

- (i) caso o objetivo seja disseminar o pacote para todos os nós da rede, o intervalo de tempo necessário para atender a este objetivo será muito maior do que em estratégias baseadas em *broadcast*;
- (ii) não resolve o problema da duplicação de dados, resultante do compartilhamento de áreas geográficas entre nós-sensores vizinhos [31];
- (iii) é possível que um nó n_1 envie o pacote p_1 para um vizinho n_2 , que é o mesmo nó que havia enviado p_1 para n_1 , anteriormente. Este fato pode gerar retardos ou, no pior caso, um ciclo que impede os demais nós da rede de receberem p_1 ; e
- (iv) como os nós-sensores são suscetíveis a falhas, as chances de perda de um pacote devido à falha de um nó-sensor, localizado na rota deste pacote,

aumentam muito, visto que apenas uma cópia do pacote é mantida na rede a cada intervalo de tempo.

- **SPIN (Sensor Protocols for Information via Negotiation)**

Consiste em uma família de protocolos de roteamento, aplicáveis a RSSFs, que apenas disseminam pacotes na rede após uma prévia negociação com seus nós vizinhos. Esta negociação se dá através de trocas de mensagens, as quais evidenciam o interesse de cada nó em receber ou não o pacote que está sendo disseminado. As mensagens trocadas no SPIN são de três tipos: ADV, enviada por um nó n1 aos seus vizinhos, advertindo sobre a existência de um novo pacote; REQ, enviada a n1 por seus vizinhos, indicando o interesse de cada um em receber o novo pacote; e DATA, que é o próprio pacote de dados enviado por n1 aos nós que retornaram uma mensagem do tipo REQ [31].

Para que haja sucesso na negociação, os nós devem ser capazes de descrever, sucintamente, os dados que eles coletam do ambiente; esta informação é representada por metadados, específicos de cada aplicação. Com base nas informações contidas nos metadados, é realizada uma negociação entre os nós. Esta negociação garante que somente os pacotes úteis para um dado nó serão enviados a ele. Assim, consegue-se uma maior conservação de energia pela redução do número de pacotes enviados.

O SPIN é projetado para evitar os três problemas do flooding clássico. O problema de implosão é solucionado através da negociação que precede o envio dos pacotes, evitando a transmissão de dados redundantes; a sobreposição de área geográfica é tratada através da avaliação dos metadados; e o problema de desconhecimento de recursos físicos é resolvido pela verificação da disponibilidade de energia do sensor. Porém, a adoção do SPIN leva a um maior overhead, provocado pela troca extra de mensagens, o que aumenta os tempos de disseminação dos pacotes. Além disso, cada nó deve, periodicamente, atualizar sua lista de nós-vizinhos, o que requer consumo de memória e energia [60].

- **SMECN (Small Minimum Energy Communication Network)**

Este protocolo constrói sub-redes, a partir de uma RSSF, que possuem caminhos otimizados entre quaisquer pares de nós considerados. O caminho otimizado garante que a rota tomada por um pacote, a partir do nó-origem até o nó-destino, seja o que resulta em um menor consumo de energia. Vale salientar que o SMECN não encontra a rota que resulta no menor consumo de energia, mas apenas uma sub-rede que contém esta rota [2].

- **SAR (Sequential Assignment Routing)**

Este protocolo [2] cria múltiplas árvores, sendo que o nó-raiz de cada árvore é um nó-vizinho da estação-base. O SAR é utilizado por cada nó-sensor da rede para escolher a rota de envio de um dado pacote para a estação-base. Esta escolha é feita com base em três critérios: recursos de energia, o nível de prioridade de cada pacote e o nível de QoS (Quality of Service) calculado para cada caminho. As métricas de QoS adotadas incluem valores de throughput e tempos de retardo.

Para cada pacote roteado através da rede, uma métrica de QoS é computada. Esta métrica é calculada como sendo o produto entre a adição das métricas de QoS e um coeficiente associado ao nível de prioridade do pacote. Este cálculo é usado para avaliação de desempenho da rede.

O objetivo do algoritmo SAR é minimizar a média da métrica de QoS calculada durante todo o tempo de vida da rede. Pelo uso do SAR, cada nó da rede possui múltiplas rotas de envio possíveis, o que é vantajoso, pois caso uma rota seja interrompida pela falha de um nó sensor, outras rotas podem ser utilizadas. Como forma de contemplar as mudanças na topologia da rede; devido a falhas em nós sensores, alterações da disponibilidade de energia destes nós ou mesmo alterações dos valores de QoS; o SAR, periodicamente, requer que a estação base dispare uma atualização (recálculo) das rotas da rede. Este reprocessamento das rotas da rede demanda um custo de energia e processamento, particularmente caro para a estação base.

- **LEACH (Low-Energy Adaptive Clustering Hierarchy)**

Em Heinzelman et al. [30] é proposto o LEACH como um protocolo de comunicação auto-organizável e adaptativo que adota uma estrutura hierárquica para os nós de uma RSSF. A hierarquia é baseada na segmentação dos nós da rede em grupos, sendo que cada grupo tem um de seus nós eleito como coordenador. O nó-coordenador recebe os pacotes dos demais nós do grupo, realiza processamentos sobre os dados destes pacotes (agrupamento de dados), e os envia à estação-base.

Devido às tarefas extras realizadas pelo coordenador, observa-se que o consumo de energia neste nó será maior do que nos demais nós do seu grupo. LEACH objetiva distribuir a carga de consumo de energia uniformemente entre os nós, o que é conseguido através do revezamento, entre os nós-sensores, do papel de coordenador de grupo.

- **Directed diffusion**

A proposta Directed diffusion [34] consiste em um protocolo de comunicação orientado aos dados, o que significa que o roteamento dos pacotes é definido com base nas propriedades dos dados coletados, como, por exemplo, a região geográfica. Este protocolo especifica que a estação-base deve, inicialmente, disseminar uma notificação sobre os dados que tem interesse em obter, o que é realizado através do envio de mensagens, em broadcast, para todos os nós-vizinhos. Esta notificação de interesse descreve uma tarefa requerida aos nós-sensores, a qual consiste em pares de atributo e valor, por exemplo, a especificação de um intervalo entre coletas de dados = 20 segundos.

À medida que as notificações de interesse vão sendo passadas através da rede, gradientes vão sendo formados, indicando a direção em que os pacotes de dados deverão fluir de volta dos nós-origem de dados para a estação-base. Cada nova notificação de interesse recebida pelos nós sensores é armazenada localmente em sua memória cachê e interpretada para servir como parâmetro na coleta de dados. Finalmente, em cada nó-sensor, os dados coletados que atendem às especificações das notificações de interesse são enviados à estação-base.

Como benefícios desta proposta, tem-se uma adaptação mais rápida da rede às mudanças de topologia, provocadas pela falha ou mobilidade dos nós; e uma diminuição do consumo de energia dos nós-sensores, visto que menos pacotes são transmitidos. Além disso, é prevista a aplicação da técnica de agregação em rede, o que contribui para também reduzir o volume de dados transmitido.

7.3.8. RSSFs com topologia Scale-free

Uma arquitetura comum proposta para RSSFs é baseada na distribuição de nós-sensores em áreas geográficas, de maneira que os nós-sensores utilizem protocolos de roteamento do tipo múltiplos saltos para enviar os dados coletados a uma estação-base. Geralmente, estas propostas organizam os nós-sensores em árvores de roteamento [33][41][51].

O cenário de rede considerado neste trabalho consiste em uma RSSF de topologia scale-free [6]. Redes scale-free tendem a conter um subconjunto de seus nós, denominados hubs, conectados a um grande número de outros nós da rede. Observe que estes hubs são nós críticos, por influenciarem fortemente na forma como a rede opera. Por exemplo, a falha de um hub poderá prejudicar a comunicação de todos os nós conectados a ele. Em Barabasi et al. [6] é mapeada a conectividade da Web, classificando-a como sendo um exemplo de topologia scale-free ao considerar cada página como um nó (hub) que leva a várias outras páginas, através de hyper links. A própria organização da sociedade humana ou mesmo a interação de proteínas em um organismo vivo são também considerados exemplos de uma configuração scale-free. Todos estes exemplos podem ser vistos como redes de nós que interagem uns com os outros através de alguns poucos hubs que conectam um vasto número de nós, os quais, por sua vez, possuem poucas conexões.

No caso de uma RSSF de configuração scale-free, alguns nós-sensores agem como nós-pais de um conjunto de outros nós-sensores. Desta forma, dados coletados são passados de um nó-sensor para o outro até eles alcançarem a estação-base, onde um processamento final é aplicado aos dados, e os resultados da consulta são produzidos para o usuário.

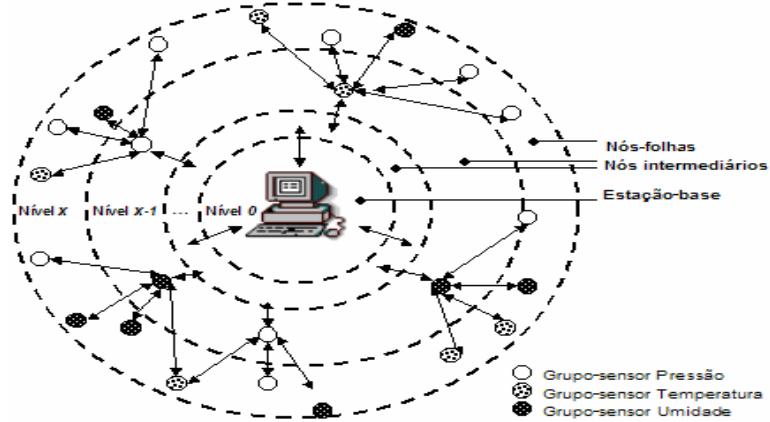


Figura 7.3. Arquitetura simplificada de uma RSSF de configuração scale-free

A partir da rede scale-free considerada neste trabalho é formulada uma visão lógica da rede, a qual categoriza os nós-sensores por sua atividade de sensoriamento, por exemplo, coleta de valores de temperatura. Assim, um conjunto de nós-sensores com a mesma atividade de sensoriamento é denominado como sendo um grupo-sensor, por exemplo, grupo-sensor Temperatura [9]. Desta forma, cada grupo-sensor é responsável por prover informações sobre algum fenômeno físico, tendo por base os dados coletados em determinadas regiões geográficas. Vale salientar que existem nós-sensores capazes de realizar j diferentes atividades de sensoriamento [32][49]. Neste caso, estes nós estarão categorizados em j diferentes grupos-sensores. Por simplicidade, a RSSF admitida neste

trabalho considera uma única estação-base que se comunica com um conjunto de nós-sensores (Figura 7.3).

Observe que a RSSF apresentada na figura 3 pode conter x níveis, dentro dos quais os nós da rede estão distribuídos: o nível 0, onde se encontra a estação-base; os níveis de 1 a $x-1$, onde se encontram os nós intermediários; e o nível x , onde estão os nós-folhas. A funcionalidade de cada um destes três tipos de nós é descrita a seguir [8]:

- (i) **Estação-base:** Nó robusto quanto à capacidade de processamento, armazenamento e disponibilidade de memória. Situado no nível 0, este nó é responsável por organizar a hierarquia entre os nós da rede, distribuir consultas para os nós-sensores, receber pacotes de dados enviados pelos nós-sensores e retornar resultados para os usuários;
- (ii) **Nós intermediários:** Distribuídos em uma área geográfica em níveis intermediários da rede (níveis 1 a $x-1$). Estes nós coletam dados do ambiente, recebem pacotes advindos de outros nós-sensores, além de processar, empacotar e enviar os dados para outros nós da rede; e
- (iii) **Nós-folhas:** São os nós situados no último nível da rede (nível x). Por esta razão, suas funcionalidades incluem apenas a coleta de dados do ambiente, o processamento e o empacotamento destes dados para posterior envio aos seus nós mais próximos (seus nós-pais).

A escalabilidade da rede apresentada na figura 3 pode ser conseguida através da comunicação entre diversas estações-base ou entre nós ligados a estações-base distintas, utilizando o meio de comunicação (Figura 7.4). Assim, uma RSSF pode ter vários pontos da rede (estações-base) através dos quais os usuários podem requisitar informações da rede. Convém salientar que quando um usuário escolhe um ponto da rede para submeter uma consulta q_1 ao sistema (aplicação de RSSF), este será o mesmo ponto que irá produzir o resultado final para q_1 .

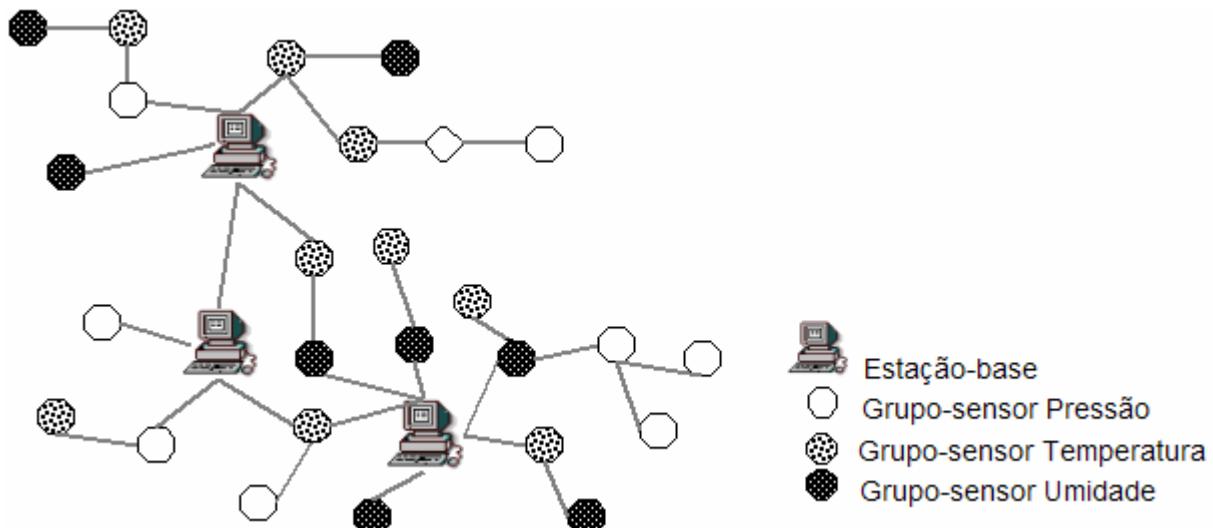


Figura 7.4. Generalização de uma RSSF com configuração scale-free.

Considera-se que as estações-base são conscientes da hierarquia da rede, o que significa que as informações sobre esta hierarquia devem ser regularmente atualizadas, objetivando refletir as possíveis mudanças na localização dos nós. Nesta abordagem, cada nó-sensor é capacitado a processar agregação em rede. Os dados processados pelos nós-sensores podem ser provenientes da coleta local, via atividade de sensoriamento, ou de pacotes de dados enviados por outros nós-sensores da rede. Considera-se, ainda, que dados locais e pacotes de dados que contenham informações a cerca de um mesmo fenômeno físico podem ser agregados, para compor um único pacote a ser enviado através da rede.

Observe que os nós intermediários de uma RSSF serão os nós-sensores mais penalizados quanto ao consumo de energia, pois estes nós não apenas coletam dados do ambiente como também recebem dados advindos de outros nós-sensores. Assim, o volume de dados a ser processado e enviado por estes nós tende a ser maior do que aquele manipulado pelos nós-folhas da rede. Como consequência, a bateria destes nós tem um desgaste mais rápido, o que influencia diretamente no tempo de vida da rede como um todo, visto que, no caso de falhas em nós intermediários, os nós-folhas provavelmente não terão um sinal suficientemente forte para alcançar a estação-base. Por esta razão, é muito importante que os algoritmos projetados para o processamento de dados em RSSFs sejam otimizados para reduzir o consumo de energia, particularmente, o custo com comunicação [39].

7.4. Modelo de dados proposto

Nesta seção é apresentado um modelo de dados genérico para aplicações de RSSFs. O objetivo, com o uso deste modelo, é permitir uma visão lógica sobre os fluxos de dados manipulados pelo sistema, de forma que os dados que fluem através da rede sejam “vistos” como tuplas de relações (virtuais). A vantagem no uso do modelo de dados proposto está em abstrair o usuário de detalhes físicos como [56]:

- (i) identificar quais nós-sensores são relevantes para responder a uma dada consulta (por exemplo, uma consulta que envolve apenas os dados coletados em uma região geográfica específica);
- (ii) determinar quais operadores de consulta podem ser aplicados aos dados nos nós-sensores ou apenas na estação-base; e
- (iii) aplicar estratégias que visem reduzir o volume de dados transmitido através da rede (por exemplo, agregação em rede).

Além disso, usuários podem definir consultas declarativas baseadas no modelo de dados, da mesma forma como ocorre em aplicações de bancos de dados convencionais.

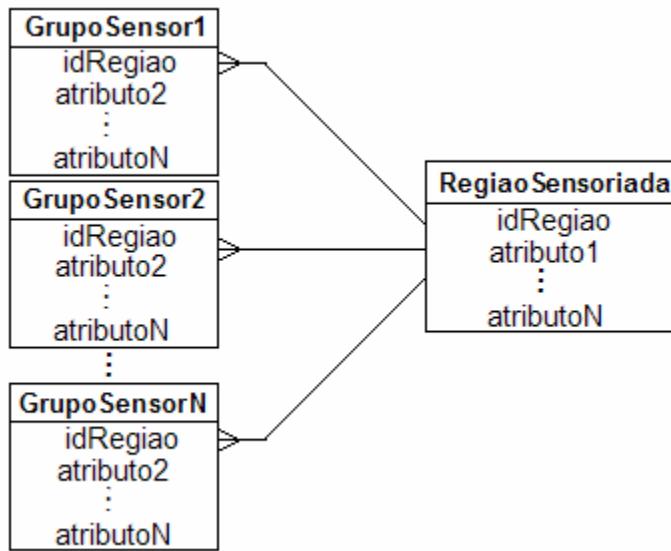


Figura 7.5. Modelo de dados para aplicações executadas sobre RSSFs.

O modelo de dados proposto captura as características de uma RSSF, de forma que cada fenômeno monitorado, considerado como sendo um grupo-sensor (*GrupoSensor1*, *GrupoSensor2*, ..., *GrupoSensorN*), é representado por uma entidade específica, que tem atributos relacionados ao fenômeno monitorado. A Figura 7.5 ilustra o modelo de dados proposto.

As entidades representantes dos grupos-sensores são relacionadas através de uma entidade, denominada *RegiaoSensoriada*, a qual representa áreas geográficas consideradas para a coleta de dados. A entidade *RegiaoSensoriada* possui um atributo denominado *idRegiao*, que identifica uma área geográfica onde um subconjunto dos nós-sensores da rede coleta dados, bem como atributos que dão as coordenadas destas regiões. O atributo *idRegiao* é comum a todas as entidades, assim, ele permite que dados gerados por diferentes grupos-sensores sejam relacionados através da área geográfica (região) onde foram coletados. Vale salientar que cada região pode conter nós-sensores pertencentes a diferentes grupos-sensores.

7.5. Aplicação-exemplo

A aplicação considerada neste trabalho consiste no mapeamento da biocomplexidade ambiental de uma dada região geográfica. O objetivo desta aplicação é relacionar informações sobre temperatura, pressão e umidade, advindas de nós-sensores espacialmente distribuídos no ambiente. Estas informações são relacionadas através da região geográfica onde cada nó-sensor se encontra.

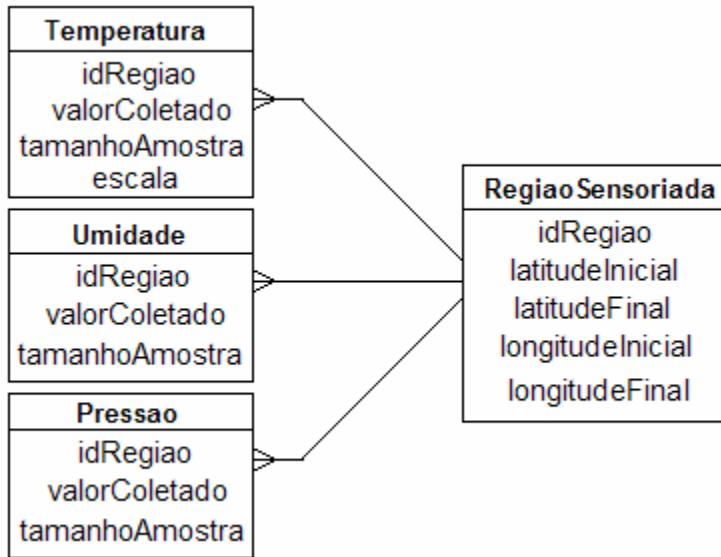


Figura 7.6. Modelo de dados - aplicação em biocomplexidade ambiental.

A Figura 7.6 ilustra a instanciação do modelo de dados proposto para a aplicação de mapeamento da biocomplexidade ambiental. Vale salientar que Temperatura, Umidade e Pressão são, de fato, tabelas virtuais, as quais representam os dados que fluem através da RSSF. Observe que cada entidade possui informações relacionadas ao fenômeno físico representado. Por exemplo, a relação Temperatura tem, como atributos, a região geográfica onde os dados são coletados (atributo idRegiao), o valor da temperatura mensurada (atributo valorColetado), o número de coletas realizadas (atributo tamanhoAmostra) e a escala de temperatura (Celsius ou Fahreneit) considerada (atributo escala). De forma similar, a relação RegiaoSensoriada tem atributos para representar as propriedades específicas de cada região.

Um exemplo clássico da necessidade de relacionar informações em uma aplicação de mapeamento da biocomplexidade ambiental é o índice de calor (índice de Temperatura-Umidade), que fornece a temperatura aparente ou sensação térmica do ambiente. O índice de calor relaciona valores de temperatura e umidade, com o propósito de estudar os riscos desta combinação de valores para a saúde, visto que, quanto maior este índice, maior será o risco. Por exemplo, se a temperatura registrada pelos sensores for superior a 30°C e, ao mesmo tempo, um alto valor de umidade é medido, o valor do índice de calor poderá ser muito superior à temperatura registrada pelos nós-sensores de temperatura.

Para esta aplicação-exemplo, admite-se que todos os nós-sensores da rede têm disponibilidade inicial de recursos semelhante (capacidade de processamento, quantidade de memória e energia); e que a RSSF está organizada como uma rede de topologia scale-free, como mostrado na Figura 7.3.

7.6. SNQL – Uma linguagem de consultas para RSSFs

O uso de nós-sensores inteligentes representou uma grande evolução para as RSSFs. Dois fatores contribuíram particularmente para esta evolução. O primeiro foi a distribuição do processamento de dados entre os nós da rede. Este fator fez com que a rede deixasse de ser passiva, por utilizar os nós-sensores apenas para coleta e envio dos dados, passando a

dividir o processamento de dados entre todos os nós da rede, inclusive os nós-sensores. O segundo fator foi a possibilidade de alterar, dinamicamente, configurações físicas do nó-sensor. Este último fator permitiu a parametrização de variáveis capazes de informar ao nó-sensor, por exemplo, quando realizar a coleta de dados do ambiente. Esta técnica de passagem de parâmetros para os nós-sensores evoluiu para o uso das linguagens de consultas, especialmente projetadas para as aplicações de RSSFs.

Neste contexto, a linguagem de consultas SNQL (Sensor Network Query Language) é proposta neste trabalho como uma alternativa para a especificação de consultas declarativas em aplicações de RSSFs. As propriedades da linguagem SNQL são descritas a seguir [9]:

- (i) Permite expressar consultas declarativas: partindo de um modelo de dados, que utiliza a noção de uma RSSF segmentada em grupos de nós-sensores, SNQL permite aos usuários a especificação de consultas declarativas (*ad-hoc*), construídas com base nas informações disponibilizadas na rede pelos nós-sensores;
- (ii) Controle da precisão dos resultados e do volume de dados a ser coletado: SNQL possui cláusulas que permitem ao usuário controlar o tamanho da amostra a ser coletada em resposta a uma consulta, bem como o quão preciso será o resultado obtido a partir desta amostra;
- (iii) Suporte a consultas contínuas: as consultas contínuas são especialmente aplicáveis a ambientes que trabalham com fluxos de dados. Quando uma consulta contínua é submetida à RSSF, a consulta é executada por um período de tempo indeterminado, sendo o resultado continuamente atualizado na estação-base e incrementalmente disponibilizado. Desta forma, a cada instante de tempo, uma nova posição sobre o resultado da consulta é fornecida ao usuário;
- (iv) Suporte a consultas pré-definidas: consiste em re-submeter uma mesma consulta ao sistema por um número indeterminado de vezes. Considera-se que cada submissão resulta em uma execução, de tempo limitado, da consulta; o que é considerado como uma nova instância desta. Ao final do tempo-limite de execução de cada uma destas instâncias, os dados, até então materializados na estação-base, são descartados; e uma nova materialização de dados é iniciada, a partir da execução de uma nova instância da consulta. Observa-se que uma consulta contínua resulta em uma única instância da consulta executada por tempo indeterminado, enquanto uma consulta pré-definida resulta em um número indeterminado de instâncias da consulta, cada uma sendo executada por um tempo limitado; e
- (v) Trabalha com fragmentos de consultas: SNQL permite especificar fragmentos de código destinados a ajustar os valores das cláusulas de uma consulta, correntemente em execução, de acordo com a necessidade do usuário. Por exemplo, o usuário pode decidir que o resultado da consulta, até então disponibilizado, já é suficiente para se chegar a alguma conclusão. Desta forma, a consulta pode ser finalizada através do envio de um fragmento de consulta, que informa aos nós-sensores que a consulta, correntemente em execução, deve

ser interrompida.

7.6.1. Especificação

A tabela 1 apresenta a descrição das funcionalidades das cláusulas da SNQL.

Tabela 1. Especificação das cláusulas admitidas em SNQL.

Cláusulas ¹	Especificação
<i>SELECT {<expr>}</i>	<expr>: Especifica um subconjunto de atributos das relações consideradas.
<i>FROM {<sensor group>}</i>	<sensor group>: Define os grupos-sensores considerados.
<i>[WHERE {<pred>}]</i>	<pred>: Especifica um conjunto de predicados que filtram tuplas processadas.
<i>[GROUP BY {<exprgroup>} [HAVING {<predhaving>}]]</i>	<exprgroup>: Define um subconjunto de atributos nos quais funções de agrupamento devem ser aplicadas. <predhaving>: Especifica predicados, baseados nas funções de agrupamento, para filtrar resultados agregados.
<i>TIME WINDOW <twseconds> CONTINUOUS</i>	<twseconds>: Define o intervalo de tempo durante o qual a consulta é válida para o sistema (por exemplo, 3600 s). O valor pré-definido <i>CONTINUOUS</i> especifica que a consulta tem um tempo de validade indeterminado.
<i>[DATA WINDOW <dwnumrows>]</i>	<dwnumrows>: Especifica a quantidade máxima de dados a ser coletada por cada nó-sensor (por exemplo, 50.000 coletas).
<i>SEND INTERVAL <sndseconds></i>	<sndseconds>: Define o intervalo de tempo entre dois envios sucessivos de pacotes de dados (por exemplo, 120 s).
<i>SENSE INTERVAL <snsseconds></i>	<snsseconds>: Especifica o intervalo de tempo entre duas coletas consecutivas de dados do ambiente pelos nós-sensores.
<i>[SCHEDULE <numexecutions> [{datetime}] CONTINUOUS]</i>	<numexecutions>: Define o número de vezes em que uma consulta deve ser submetida à rede e quando cada submissão deve ocorrer, por exemplo, <2 '10-oct-05 14:00:00', '15-oct-2005 14:00:00'>. O valor <i>CONTINUOUS</i> especifica que a consulta tem que ser submetida à rede por um número indeterminado de vezes.

Observe que as cláusulas SELECT, FROM, WHERE, GROUP BY e HAVING são definidas da mesma forma como no padrão SQL. Estas cláusulas, juntamente com as demais especificadas na SNQL, permitem que os usuários definam consultas declarativas. As cláusulas de uso específico em consultas para aplicações de RSSFs serão discutidas a seguir.

A cláusula TIME WINDOW especifica o período de tempo de validade da consulta. Em outras palavras, o intervalo de tempo em que os nós-sensores permanecerão coletando dados para responder à consulta, e também o intervalo de tempo em que a estação-base permanecerá recebendo pacotes de dados enviados pelos nós-sensores. Note que o valor pré-definido CONTINUOUS permite que o usuário defina uma consulta como sendo contínua. Já a cláusula DATA WINDOW especifica o número de coletas do ambiente a ser realizada por cada nó-sensor, para responder à consulta. Tanto a cláusula TIME WINDOW como a cláusula DATA WINDOW são úteis para definir o tamanho da amostra que será produzida em resposta a uma determinada consulta, porém, caso ambas sejam informadas na consulta, a cláusula TIME WINDOW terá prioridade.

¹ Considera-se “{}” para denotar conjunto, “[]” para denotar cláusulas opcionais, “<>” para uma expressão, e “|” para denotar que um ou o outro *token* deve aparecer, mas não ambos.

A cláusula SEND INTERVAL define o intervalo de tempo entre envios sucessivos de pacotes de dados para outros nós da rede. O valor desta cláusula deve ser cuidadosamente definido pois, se o seu valor for muito alto, mais dados serão acumulados nos nós-sensores, o que aumenta as chances de situações de “estouro” de memória; porém, valores baixos podem produzir maiores quantidades de pequenos pacotes de dados, o que contribui para aumentar o tráfego da rede e também leva a um maior overhead, visto que mais cabeçalhos de pacotes serão necessários.

O intervalo de tempo entre coletas de dados sucessivas do ambiente é especificado na cláusula SENSE INTERVAL. Esta cláusula também deve ser cuidadosamente definida, visto que ela influencia diretamente na precisão dos resultados da consulta. Assim, se o seu valor é alto, menos dados serão coletados e os resultados serão menos precisos, pois diversos valores diferentes de medições podem ter ocorrido entre duas coletas de dados consecutivas. Por outro lado, se o seu valor for muito baixo, mais coletas de dados serão realizadas e, dependendo da função de agregação especificada na consulta, pode-se ter um maior acúmulo de dados na memória do nó-sensor. Como consequência, tem-se que mais dados terão que ser processados e enviados através da rede, gerando um maior tráfego na rede. Desta forma, observa-se que os valores das cláusulas SEND INTERVAL e SENSE INTERVAL influenciam diretamente no desempenho da rede e no tempo de vida dos nós-sensores. Uma proposta, para realizar uma “calibração” dos valores destas cláusulas, é explorada pelo algoritmo ADAGA [8].

A cláusula SCHEDULE permite ao usuário definir o número de vezes e a periodicidade em que a consulta deve ser submetida ao sistema. Por exemplo, <2 ‘10-jan-06 14:00:00’, ‘15-jan-2006 14:00:00’> significa que a consulta será executada duas vezes, na data e hora especificadas, o que resulta em duas instâncias da consulta. Caso o valor pré-definido CONTINUOUS seja associado a esta cláusula, assume-se que a consulta é do tipo pré-definida.

SNQL suporta a especificação de fragmentos de consultas para redefinir os valores das seguintes cláusulas (em tempo de execução): TIME WINDOW, DATA WINDOW, SENSE INTERVAL, SEND INTERVAL e SCHEDULE. Assim, o usuário pode submeter um fragmento de consulta que redefina, por exemplo, um intervalo de tempo menor entre captações sucessivas de dados do ambiente, objetivando aumentar a precisão dos resultados fornecidos para uma consulta correntemente em execução.

Considere o exemplo de consulta descrito a seguir: verificar o maior valor de temperatura, associado ao menor valor de umidade, na área geográfica delimitada pelas coordenadas 30° 43' 08" S - 38° 31' 51" W e 30° 43' 16" S - 38° 31' 14" W. Além disso, deve-se considerar apenas valores de temperatura superiores a 20°C e valores de umidade inferiores a 70%. A consulta deve ser submetida ao sistema 2 vezes (em 01-jan-06 às 14:00:00 e em 15-jan-06 às 14:00:00), sendo que cada instância deve permanecer em execução por 3600 segundos. Os dados devem ser coletados do ambiente a cada 10 segundos e enviados através da rede a cada 120 segundos. A Figura 7.7 ilustra a representação desta consulta utilizando a notação da linguagem SNQL.

```

SELECT          r.IdRegiao AS RegiaoGeografica,
                MAX(t.valorColetado), MIN(h.valorColetado)
FROM            Umidade h, Temperatura t, RegiaoSensoriada r
WHERE           r.IdRegiao = t.IdRegiao AND r.IdRegiao = h.IdRegiao
AND             r.latitudeInicial > 034308 AND r.latitudeFinal < 034316
AND             r.longitudeInicial > 383151 AND r.longitudeFinal < 383114
AND             t.valorColetado > 20
AND             h.valorColetado < 0.7
GROUP BY        r.IdRegiao
TIME WINDOW    3600
SEND INTERVAL 120
SENSE INTERVAL 10
SCHEDULE       2 '01-jan-06 14:00:00', '15-jan-06 14:00:00'

```

Figura 7.7. Exemplo de consulta escrita em SNQL.

7.6.2. Suporte à agregação de dados

É muito importante que as linguagens de consulta para aplicações de RSSFs ofereçam suporte às funções de agregação. Quanto mais funções de agregação forem consideradas na especificação da linguagem, maiores serão as possibilidades de que a agregação em rede seja aplicável à resolução de uma consulta do usuário. A tabela 2 mostra as funções de agregação previstas na especificação da SNQL. Observe que a contrapartida em se considerar uma maior diversidade destas funções é que a camada de aplicação da RSSF precisa tornar-se mais robusta, pois o suporte a estas funções envolve a implementação de cada uma delas.

Tabela 2. Funções de agregação admitidas na especificação da linguagem SNQL.

Funções	Especificação
<i>COUNT()</i>	Número de ocorrências dos valores de entrada não-nulos.
<i>SUM()</i>	Somatório de todos os valores de entrada.
<i>MAX()</i>	Maior valor entre todos os valores de entrada.
<i>MIN()</i>	Menor valor entre todos os valores de entrada.
<i>AVERAGE()</i>	Média aritmética de todos os valores de entrada.
<i>VARIANCE()</i>	Medida da dispersão estatística dos valores de entrada, indica o quanto longe estes valores encontram-se do valor esperado.
<i>STDDEV()</i>	Corresponde à raiz quadrada da variância.
<i>MEDIAN()</i> ²	É a medida de localização do centro da distribuição dos dados de entrada.
<i>MODE()</i>	Valor de entrada que ocorre com maior freqüência.

Para mostrar a relação entre o tamanho dos resultados parciais (produzidos em nós-sensores) e resultados finais (produzidos na estação-base) para diferentes funções de agregação, considere o cenário a seguir. Dois nós-sensores, n1 e n2, pertencem ao grupo-sensor Temperatura. Os nós n1 e n2 são capacitados a processar a agregação em rede, de acordo com a função de agregação especificada na consulta. Suponha que n1 colete os

² Para determinar a mediana deve-se ordenar os y elementos de entrada (amostra). Se y é ímpar, a mediana é o elemento localizado na posição central. Se y é par, a mediana é a semi-soma dos dois elementos centrais.

valores de temperatura 20°C – 21°C – 22°C e gere o pacote p1 com o resultado da função de agregação aplicada a estes valores, e que n2 colete os valores de temperatura 21°C – 21°C – 22°C e gere o pacote p2. Em seguida, os dados de p1 e p2 são também agregados na estação-base, onde o resultado será, finalmente, disponibilizado ao usuário. A tabela 3 mostra os resultados das funções de agregação nos nós-sensores e na estação-base, conforme classificação sugerida em [24][41]. Observa-se que o tamanho dos pacotes gerados em n1 e n2 dependem da função de agregação utilizada.

Tabela 3. Resultados parciais obtidos para diferentes funções de agregação.

		Nó n_1	Nó n_2	Estação Base
Categoría ↓	Dados coletados →	20°C - 21°C - 21°C	21°C - 21°C - 22°C	<i>dados em p1 e p2</i>
Agregação distributiva	COUNT()	3	3	6
	SUM()	62	64	125
	MAX()	21°C	22°	22°C
	MIN()	20°C	21°	20°C
Agregação Algébrica	AVERAGE()	62 – 3	64 – 3	20,83 °C
	VARIANCE()	62 – 3	64 – 3	1
	STDDEV()	62 – 3	64 – 3	1
Agregação holística	MEDIAN()	20°C - 21°C - 21°C	21°C - 21°C - 22°C	21°C
	MODE()	20°C - 21°C - 21°C	21°C - 21°C - 22°C	21°C

funções de agregação
resultados parciais
resultado final

No contexto das aplicações de RSSFs, para as funções de agregação distributiva, o tamanho do resultado parcial produzido em cada nó-sensor será o mesmo do resultado final produzido na estação-base. Por exemplo, a função MAX() irá produzir um resultado de mesmo tamanho, independente do local da rede onde for aplicada. Já as funções de agregação algébricas requerem que cada nó-sensor envie dois valores por pacote, correspondentes aos resultados parciais produzidos pelas funções SUM() e COUNT(), sendo que a função de agregação especificada na consulta (por exemplo, AVERAGE()) apenas será computada na estação-base. Assim, o volume de dados transmitido através da rede, na agregação algébrica, será duas vezes maior que aquele gerado por agregações distributivas. As funções de agregação holística, por sua vez, não viabilizam a utilização da agregação em rede, visto que todos os dados coletados pelos nós-sensores devem ser enviados à estação-base, onde só então o resultado final poderá ser computado (por exemplo, a função MEDIAN()).

As estratégias de aproximação de resultados Histogramas, Wavelets, Sliding Windows, Punctuations, Sketches e Sampling (abordadas na Subseção 3.2.2) são alternativas para a sumarização de fluxos de dados, assim como as funções de agregação. SNQL já explora o conceito de Sliding Windows baseadas em tempo, através da cláusula TIME WINDOW.

7.7. Processamento de consultas

Geralmente, o processamento de consultas em RSSFs consiste em utilizar nós-sensores simples, que apenas coletam dados do ambiente e os enviam a uma estação-base, onde um

banco de dados convencional processa os dados e os entrega ao usuário. Esta proposta traz como desvantagens o alto tráfego na rede e o alto consumo de energia requerido dos nós-sensores. A distribuição do processamento da consulta em RSSFs é uma alternativa para superar estes problemas.

O processamento de uma consulta pode ser visto como o conjunto de atividades envolvidas na extração de dados de um banco de dados. Neste trabalho, admite-se que o processamento de uma consulta ocorre de maneira distribuída na RSSF. Desta forma, cada nó-sensor é considerado tanto uma fonte remota de dados, como também um dispositivo capacitado a pré-processar dados. Considera-se que as atividades envolvidas no processamento de uma consulta estão distribuídas em quatro etapas distintas, análise, decomposição, processamento de fragmentos e pós-processamento, que têm suas execuções divididas entre nós-sensores e estação-base.

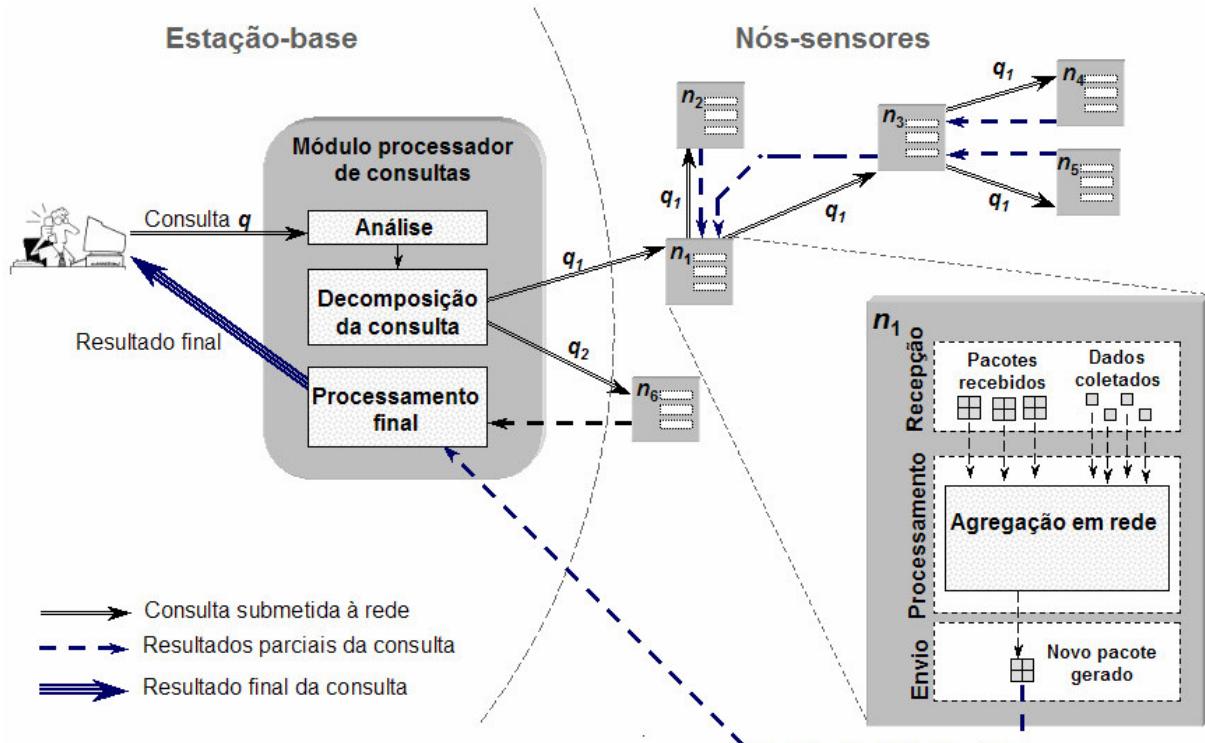


Figura 7.8. Estratégia admitida para o processamento de consultas em RSSFs.

A Figura 7.8 mostra um exemplo de como se dá o processamento de uma consulta no cenário de RSSF considerado. Quando o usuário submete uma consulta ao sistema (q), a consulta é analisada e decomposta nas sub-consultas q_1 e q_2 , uma para cada grupo-sensor referenciado na cláusula FROM. Em seguida, as sub-consultas são enviadas aos nós-sensores n_1, n_2, n_3, n_4, n_5 e n_6 . Após a recepção de uma consulta, as seguintes tarefas são executadas no nó-sensor: recebimento de pacotes de outros nós-sensores, coleta e processamento de dados, e envio de pacotes. A execução da consulta é finalizada no nó-sensor quando seu tempo de validade é alcançado. Na estação-base, a finalização da consulta envolve a suspensão do recebimento de pacotes de dados, a aplicação de algumas operações finais aos dados (por exemplo, filtragens e agregações de dados) e o fornecimento do resultado final ao usuário.

7.7.1. Análise

Nesta etapa, a consulta escrita em linguagem declarativa de alto nível, como SNQL, é analisada, objetivando verificar se a sintaxe utilizada está de acordo com as regras gramaticais da linguagem de consulta utilizada. Também é verificado se os nomes de atributos e relações, especificados na consulta, são válidos para o esquema do banco de dados, construído com base no modelo de dados. Finalmente, a consulta é convertida em uma árvore de análise, a qual é geralmente traduzida, no caso de sistemas de bancos de dados relacionais, para expressões da álgebra relacional. Como resultado desta etapa, um plano de execução da consulta é gerado. Observe que esta etapa é somente aplicada à estação-base, onde a consulta foi submetida pelo usuário.

7.7.2. Decomposição

Nesta etapa, o plano de execução, gerado na análise, é decomposto na estação-base, de acordo com os grupos-sensores especificados na consulta. Note que será gerada uma sub-consulta para cada grupo-sensor referenciado. Assim, uma consulta q , a qual relaciona dois grupos-sensores distintos g_1 e g_2 , será decomposta em dois fragmentos, sub-consultas q_1 e q_2 , correspondentes, respectivamente, aos grupos de sensores g_1 e g_2 . A decomposição da consulta simplifica o trabalho dos nós-sensores, visto que um dado nó-sensor n_1 apenas processará o fragmento da consulta relacionado às informações que ele está capacitado a fornecer, poupando o trabalho de interpretar cláusulas da consulta que não lhes dizem respeito. Para que isto seja possível, é necessário que o nó-sensor detenha o conhecimento sobre a parte do esquema da aplicação que modela os fenômenos tratados neste nó-sensor.

A decomposição é aplicada de forma diferenciada para os operadores de projeção, junção, seleção e agregação. Os atributos definidos na projeção serão decompostos em diferentes sub-consultas, de acordo com o grupo-sensor a que os atributos pertencem. Operações de junção não são previstas em uma sub-consulta, visto que esta apenas refere-se a um grupo-sensor. Operações de seleção também podem influenciar na decomposição da consulta. Por exemplo, uma consulta que envolva apenas um subconjunto das regiões geográficas cobertas pela RSSF pode ser dividida em sub-consultas direcionadas apenas a estas regiões. Tanto as operações de seleção quanto agregação são aplicáveis nos nós-sensores da rede, o que é conseguido com o uso de algoritmos baseados na técnica de agregação em rede.

Após a consulta inicial ser decomposta, cada fragmento é encapsulado em um pacote, cujo cabeçalho possui a informação necessária sobre o critério de decomposição, por exemplo, grupo-sensor ou localização do nó-sensor. Assim, quando um nó-sensor recebe um pacote, ele avalia o conteúdo do cabeçalho, como forma de identificar se deve aceitar ou simplesmente repassar a outros nós da rede a consulta ou o fragmento de consulta recebido.

7.7.3. Processamento de fragmentos

A execução desta etapa é distribuída entre os diversos nós-sensores da RSSF (veja figura 8). Considerando uma consulta definida em SNQL, quando um nó-sensor recebe e aceita um fragmento de consulta, um temporizador t é iniciado e, em seguida, a coleta de dados tem início, obedecendo aos intervalos definidos na cláusula SENSE INTERVAL. O valor da cláusula SEND INTERVAL especifica quando as atividades do nó-sensor devem ser

suspensas para que o envio e recepção de pacotes ocorram. Quando o nó-sensor não está coletando dados, ou recebendo e enviando pacotes, sua atividade consiste em processar os dados localmente armazenados, o que é feito através da agregação em rede. Finalmente, quando o temporizador t alcança o valor definido na cláusula TIME WINDOW, a coleta de dados e a recepção de pacotes são interrompidas, e é feito o empacotamento e envio dos dados correntemente armazenados no nó-sensor; o que encerra a execução da consulta neste nó. Uma estratégia para a execução dos passos da etapa de processamento de fragmentos é proposta no algoritmo ADAGA [8].

7.7.4. Processamento final

Esta etapa é executada com a finalidade de preparar os dados para serem apresentados ao usuário. O processamento final consiste em executar:

- (i) Operações de projeção, para reunir os atributos, especificados na consulta submetida pelo usuário, provenientes de diferentes grupos-sensores;
- (ii) Operações de junção e/ou união, para relacionar os dados enviados por nós de diferentes grupos-sensores;
- (iii) Operações de agregação, para reunir os dados advindos de diferentes nós-sensores; e
- (iv) Outras operações, não processadas nos nós-sensores por apenas terem sentido no contexto da estação-base. Um exemplo clássico é a aplicação da cláusula *HAVING*, que especifica predicados aplicáveis ao resultado final, obtido após a agregação de todos os resultados parciais gerados para a consulta. Neste caso, apenas a estação-base é capaz de processar a operação definida na cláusula *HAVING*.

O último passo desta etapa é disponibilizar o resultado final da consulta para o usuário. Uma alternativa proposta para a junção de dados provenientes de diferentes grupos-sensores é executar o algoritmo ADAPT [11].

7.8. Conclusão

Neste capítulo foram abordados alguns dos aspectos mais relevantes a serem considerados no projeto de uma rede de sensores sem fio (RSSF). Também foi realizado um estudo das principais pesquisas relacionadas aos sistemas de fluxos de dados, bem como extensões destes sistemas propostas para atender às necessidades específicas das aplicações de RSSFs. Este embasamento teórico foi utilizado como base para introduzir conceitos explorados nos demais capítulos deste trabalho.

Foi apresentada ainda uma máquina de consulta que apresenta suporte ao processamento adaptativo de consultas em aplicações de RSSFs. O objetivo é o de realizar o processamento de consultas visando à disponibilização incremental de resultados de consultas e também otimizar o uso dos recursos físicos da rede, particularmente, dos nós-

sensores. A metodologia proposta consiste em dois algoritmos, a serem executados nos nós de uma rede de sensores sem fio: ADAGA, para processamento de dados nos nós-sensores, e ADAPT, destinado ao processamento de junções nas estações-base destas redes.

Referências

- [1] Akyildiz, I., Su, W., Sankarasubramaniam, Y., Cayirci, E. *A survey on sensor networks*. IEEE Communications, pg. 102-114, Agosto de 2002.
- [2] Al-Karaki, J., Kamal, A. *A taxonomy of routing techniques in wireless sensor networks*. Handbook of sensor networks: compact wireless and wired sensing systems. Editado por Mohammad Ilyas e Imad Mahgoub, CRC Press, pg. 117 – 140, Julho de 2004.
- [3] Arasu, A., Babu S., Widom, J. *The CQL Continuous Query Language: semantic foundations and query execution*. Technical report, Stanford University, Outubro de 2003.
- [4] Babu, S., Bizarro, P. *Adaptive query processing in the looking glass*. Conference on Innovative Data Systems Research - CIDR, Janeiro de 2005.
- [5] Babu, S., Motwani, R., Babcock, B., Datar, M. *Models and issues in data stream systems*. ACM SIGMOD/PODS, Junho de 2002.
- [6] Barabasi, A., Albert, R. *Emergence of scaling in random networks*. Science 286, pg. 509-512, Outubro de 1999.
- [7] Bonnet, P., Gehrke, J., Seshadri, P. *Querying the physical world*. IEEE Personal Communications, Vol. 7, Nº 5, pg. 10-15, Outubro de 2000.
- [8] Brayner, A., Lopes, A., Menezes, R., Vasconcelos, R. *Balancing energy consumption and memory usage in sensor data processing*. ACM Symposium on Applied Computing – SAC. Março de 2007.
- [9] Brayner, A., Lopes, A., Meira, D., Vasconcelos, R., Menezes, R. *ADAGA: ADaptive AGgregation Algorithm for Sensor Networks*. XXI Simpósio Brasileiro de Banco de Dados – SBBD, pg. 191-205, Outubro de 2006.
- [10] Brayner, A., Paulino, T. *Pré-Cálculo de junções para o processamento de consultas em ambientes com recursos computacionais limitados*. Dissertação de mestrado da Universidade Federal do Ceará - UFC, Agosto de 2004.
- [11] Brayner, A., Vasconcelos, E. *MobiJoin: a join operator for mobile databases*. VI Workshop de Comunicação Sem Fio e Computação Móvel - WCSF, pg. 153-160, Outubro de 2004.
- [12] Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S. B. *Monitoring streams - a new class of data management applications*. Very Large Data Bases – VLDB, pg. 215-226, Agosto de 2002.
- [13] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M. A. *TelegraphCQ: continuous dataflow processing for an uncertain world*.

Conference on Innovative Data Systems Research – CIDR, pg. 269-280, Janeiro de 2003.

- [14] Chen, J., DeWitt, D. J., Tian, F., Wang, Y. *NiagaraCQ: a scalable continuous query system for internet databases*. ACM SIGMOD International Conference on Management of Data - COMAD, pg. 379–390, Maio de 2000.
- [15] Considine, J., Li, F., Kollios, G., Byers, J. *Approximate aggregation techniques for sensor databases*. International Conference on Data Engineering – ICDE, pg. 449-460, Março de 2004.
- [16] Cranor, C. D., Johnson, T., Spatscheck, O., Shkpenyuk, V. *Gigascope: a stream database for network applications*. ACM SIGMOD Conference, pg. 647-651, Junho de 2003.
- [17] Deshpande, A., Guestrin, C., Madden, S. R., Hellerstein, J. M., Hong, W. *Model-driven data acquisition in sensor networks*. Very Large Data Bases – VLDB, pg. 588-599, Setembro de 2004.
- [18] Elnahrawy, E. *Research directions in sensor data streams: solutions and challenges*. Technical report, Rutgers University, Maio de 2003.
- [19] Ganesan D., Estrin D. *DIMENSIONS: why do we need a new data handling architecture for sensor networks?* Workshop on Hot Topics in Networks (Hotnets-I), pg. 143-148, Outubro de 2002.
- [20] Garofalakis, M., Gibbons, P. B. *Wavelet synopses with error guarantees*. ACM SIGMOD International Conference on Management of Data – COMAD, pg. 476–487, Maio de 2002.
- [21] Goel, S., Imielinski, T. *Prediction-based monitoring in sensor networks: taking lessons from MPEG*. ACM SIGCOMM Computer Communication - CCR, pg. 82-98, Outubro de 2002.
- [22] Golab, L. *Querying sliding windows over on-line data streams*. International Conference on Data Engineering ICDE/EDBT Ph.D. Workshop, pg. 1-10, Março de 2004.
- [23] Govindan, R., Hellerstein, J., Hong, W., Madden, S., Franklin, M., Shenker, S. *The sensor network as a database*. Technical Report, University of Southern California, Setembro de 2002.
- [24] Gray, J., Bosworth, A., Layman, A., Pirahesh, H. *Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-total*. International Conference on Data Engineering - ICDE, pages 152-159, Fevereiro de 1996.
- [25] Gurgen, L., Labb  , C., Olive, V., Roncancio, C. *A scalable architecture for heterogeneous sensor management*. IEEE International Workshop on Database and Expert Systems Applications - DEXA, pg. 1108-1112, Agosto de 2005.
- [26] Hac, A. *Wireless sensor network designs*. John Wiley & Sons press. University of Hawaii at Manoa, Honolulu, EUA, Dezembro de 2003.

- [27] Haenggi, M. *Opportunities and challenges in wireless sensor networks*. Handbook of sensor networks: compact wireless and wired sensing systems. Editado por Mohammad Ilyas e Imad Mahgoub, CRC Press, pg. 22 – 35, Julho de 2004.
- [28] Hedetniemi, S., Hedetniemi, S., Liestman, A. *A survey of gossiping and broadcasting in communication networks*. *Networks*, 18, pg. 129–134, 1988.
- [29] Hellerstein, J. M., Haas, P. J. *Ripple Joins for Online Aggregation*. ACM SIGMOD Conference, pg. 287-298, Junho de 1999.
- [30] Heinzelman, W., Chandrakasan, R., Balakrishnan, A. *Energy-efficient communication in wireless sensor networks*. Hawaii International Conference on Systems Sciences - HICSS, pg. 1-10, Janeiro de 2000.
- [31] Heinzelman, W., Kulik , J., Balakrishnan, H. *Adaptive protocols for information dissemination in wireless sensor networks*. ACM/IEEE International Conference in Mobile Computing and Network – MobiCom, pg. 174–185, Agosto de 1999.
- [32] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K. *System architecture directions for networked sensors*. International Conference on Architectural Support Programming Languages Operating Systems - ASPLOS, pg. 93–104, Novembro de 2000.
- [33] Intanagonwiwat, C., Estrin, D., Govindan, R., Heidemann, J. *Impact of network density on data aggregation in wireless sensor networks*. International Conference on Distributed Computing Systems - ICDCS, pg. 457, Julho de 2002.
- [34] Intanagonwiwat, C., Govindan, R., Estrin D. *Directed diffusion: a scalable and robust communication paradigm for sensor networks*. ACM International Conference in Mobile Computing and Network - MobiCom, pg. 56-67, Agosto de 2000.
- [35] Ives, Z., Florescu, D., Friedman, M., Levy, A., Weld, D. *An adaptive query execution system for data integration*. ACM SIGMOD International Conference on Management of Data - COMAD, pg. 299–310, Junho de 1999.
- [36] Kalpakis, K., Dasgupta, K., Namjoshi, P. *Maximum lifetime data gathering and aggregation in wireless sensor networks*. IEEE International Conference on Networking - ICN, pg. 685-696, Agosto de 2002.
- [37] Koudas, N., Srivastava, D. *Data stream query processing: a tutorial*. XXI Simpósio Brasileiro de Banco de Dados – SBBD, Outubro de 2006.
- [38] Lindsey, S., Raghavendra, C. *PEGASIS: power-efficient gathering in sensor information systems*. IEEE Aerospace Conference, pg. 1–6, Março de 2002.
- [39] Madden, S., Franklin, M., Hellerstein, J. *TinyDB: an acquisitional query processing system for sensor networks*. ACM Transactions on Database Systems - TODS, Vol. 30, N^o. 1, pg. 122-173, Março de 2005.
- [40] Madden, S., Franklin, M., Hellerstein, J., Hong, W. *The design of an acquisitional query processor for sensor networks*. ACM SIGMOD International Conference on Management of Data - COMAD, pg. 491-502, Junho de 2003.

- [41] Madden, S., Franklin, M. J., Hellerstein, J., Hong, W. *TAG: a Tiny AGgregation service for ad-hoc sensor networks*. Symposium on Operating System Design and Implementation – OSDI, pg. 171–182, Dezembro de 2002.
- [42] Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G. S., Olston, C., Rosenstein, J., Varma, R. *Query processing, approximation, and resource management in a data stream management system*. Conference on Innovative Data Systems Research – CIDR, pg. 245-256, Janeiro de 2003.
- [43] Pottie, G.J., Kaiser, W.J. *Wireless integrated network sensors*. Communications of ACM, 43(5), pg. 51–58, Maio de 2000.
- [44] Rabaey, J.M., Ammer, M. J., Silva, J.L., Roundy, D. P. *PicoRadio supports ad hoc ultra-low power wireless networking*. IEEE Computer Magazine, 33(7), pg. 42–48, Julho de 2000.
- [45] Raghunathan, V., Schurgers, C., Park, S., and Srivasta, M. *Energy aware wireless microsensor networks*. IEEE Signal Processing Magazine, vol.19, n°2 , pg. 40-50, Março de 2002.
- [46] Ratnasamy, S., Estrin, D., Govindan, R., Karp, B., Shenker, S., Yin, L., Yu, F. *Data-centric storage in sensornets*. Workshop on Wireless Sensor Networks and Applications - WSNA, pg. 78-87, Setembro de 2002.
- [47] Ruiz, L. B., Nogueira, J. M., Loureiro, A. *Sensor network management*. Handbook of sensor networks: compact wireless and wired sensing systems. Editado por Mohammad Ilyas e Imad Mahgoub, CRC Press, pg. 57 – 84, Julho de 2004.
- [48] Schneider, Donavan A. *A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment*. ACM SIGMOD-International Conference on Management of Data - COMAD, pg. 110-121, Junho de 1989.
- [49] Shih, E., Cho, S., Ickes, N., Min, R. et al., *Physical layer driven protocol and algorithm design for energy-efficient wireless sensor networks*. ACM/IEEE International Conference in Mobile Computing and Network – MOBICOM, pg. 272–287, Julho de 2001.
- [50] Silberschatz, A., Korth,H. F. and Sudarshan, S. Database Systems Concepts. McGraw-Hill Higher Education , 4th edition, 2001.
- [51] Solis, I., Obraczka, K. *In-Network aggregation trade-offs for data collection in wireless in sensor networks*. Technical Report, University of California, Agosto de 2003.
- [52] Srisathapornphat, C., Jaikaeo, C., Shen, C. *Sensor information networking architecture and applications*. IEEE Personal Communication, 8(4), pg. 52–59, Agosto de 2001.
- [53] Su, W., Cayirci, E., Akan, O. B. *Overview of communication protocols for sensor networks*. Handbook of sensor networks: compact wireless and wired sensing systems. Editado por Mohammad Ilyas e Imad Mahgoub, CRC Press, pg. 314 – 329, Julho de 2004.

- [54] Tucker, P., Maier, D., Sheard, T., Fegaras, L. *Exploiting punctuation semantics in continuous data streams*. IEEE Transactions on Knowledge and Data Engineering, 15(3), pg. 555-568, Maio de 2003.
- [55] Urhan, T., Franklin, M. *Xjoin: a reactively scheduled pipelined join operator*. IEEE Data Engineering Bulletin, 23(2), pg. 27-33, Junho de 2000.
- [56] Yao, Y., Gehrke, J. *Query processing for sensor networks*. Conference on Innovative Data Systems Research – CIDR, pg. 233-244, Janeiro de 2003.
- [57] Yao, Y., Gehrke, J. *The Cougar approach to in-network query processing in sensor networks*. SIGMOD Record, Vol. 31, Nº 3, pg. 9-18, Setembro de 2002.
- [58] Younis, M., Akkaya, K., Eltoweissy, M., Wadaa, A. *On handling QoS traffic in wireless sensor networks*. IEEE Hawaii International Conference on Systems Sciences - HICSS, Vol. 9, pg. 90292.a, Janeiro de 2004.
- [59] Waneke, B., Lebowitz, B., Pister, K. S. J. *Smart dust: communicating with a cubic-millimeter computer*. IEEE Computer, pg. 2-9, Janeiro de 2001.
- [60] Wang, Q., Hassaneim, H. *A comparative study of energy-efficient (E2) protocols for wireless sensor networks*. Handbook of sensor networks: compact wireless and wired sensing systems. Editado por Mohammad Ilyas e Imad Mahgoub, CRC Press, pg. 239 – 360, Julho de 2004.
- [61] Wilschut, Annita N., Apers, Peer M.G. *Dataflow query execution in a parallel main-memory environment*. International Conference on Parallel and Distributed Information Systems - PDIS, pg. 68-77, Dezembro de 1991.
- [62] Zhao, F., Guibas, L. *Wireless sensor networks – An information processing approach*. Morgan Kaufmann press, Julho de 2004.

Capítulo

8

Programação Funcional em Haskell

Francisco Vieira de Souza e Pedro de Alcântara dos Santos Neto

Abstract

For being so used a programming language should support interactive work, to have extensive libraries, to be portable, to have a stable implementation and to be easily installed, to have debuggers and profiles, to have training courses and to be sucessfully used in some projects.

In this chapter, it will be showed all these characteristics are already present in Haskell programming language.

Resumo

Para ser muito utilizada, uma linguagem de programação deve suportar trabalho interativo, possuir bibliotecas extensas, ser portável, ter uma implementação estável e fácil de ser instalada, ter depuradores e profiles, ser acompanhada de cursos de treinamento e já ter sido utilizada, com sucesso, em uma boa quantidade de projetos.

Neste capítulo, será mostrado que todas estas características já estão presentes na linguagem de programação funcional Haskell.

8.1. Introdução

Em 1958, John McCarthy investigava o uso de operações sobre listas ligadas para implementar um programa de diferenciação simbólica. Como a diferenciação é um processo recursivo, McCarthy sentiu-se atraído a usar funções recursivas e, além disso, ele também achou conveniente passar funções como argumentos para outras funções. Ainda em 1958, no MIT, foi iniciado um projeto com o objetivo de construir uma linguagem de programação que incorporasse estas idéias. O resultado ficou conhecido como LISP 1, que foi descrita por McCarthy, em 1960, em seu artigo “*Recursive Functions of Symbolic Expressions and Their Computation by Machine*” [17]. Este fato é caracterizado, por alguns pesquisadores, como o marco inicial da programação funcional.

No final da década de 1960 e início da década de 1970, um grande número de cientistas da Computação começaram a investigar a programação com funções puras, chamada de “*programação aplicativa*”, uma vez que a operação central consistia na aplicação de uma função a seu argumento [20]. Em particular, Peter Landin (1964, 1965 e 1966) desenvolveu muitas das idéias centrais para o uso, notação e implementação das linguagens de programação aplicativas [13, 12].

8.2. Programação funcional

A programação aplicativa, ou funcional, é freqüentemente distinguida da programação imperativa que adota um estilo que faz uso de imperativos ou ordens. O mundo das atribuições é caracterizado também por ordens, por exemplo, “troque isto!”, “vá para tal lugar!”, “substitua isto!” e assim por diante. Ao contrário, o mundo das expressões envolve a descrição de valores, por isto, o termo “*programação orientada por valores*”.

A programação aplicativa tem um único construtor sintático que é a aplicação de uma função a seu argumento. Na realidade, este construtor é tão importante que, normalmente, é representado de forma implícita, por justaposição, em vez de explicitamente, através de algum símbolo.

8.2.1. Transparência referencial

Podemos verificar que se uma pessoa fosse avaliar manualmente a expressão $(3ax + b)(3ax + c)$ jamais iria avaliar a sub-expressão $3ax$ duas vezes. Uma vez avaliada esta sub-expressão em um contexto em que $a = 2$ e $x = 3$ (18), o avaliador humano substituiria a sub-expressão $3ax$ por 18, em todos os casos onde ela aparecesse.

Isto acontece porque a avaliação de uma mesma expressão, em um contexto fixo sempre dará como resultado o mesmo valor. Para os valores de $a = 2$ e $x = 3$, $3ax$ será sempre igual a 18. Esta técnica de avaliação humana também é utilizada na avaliação de expressões puras.

Esta propriedade é chamada de “*transparência referencial*”, e significa que, em um contexto fixo, a substituição de sub-expressões por seus valores é completamente independente da expressão envolvente. Portanto, uma vez que uma expressão tenha sido avaliada em um dado contexto, não é necessário avaliá-la novamente porque seu valor jamais será alterado.

8.3. A linguagem Haskell

A história de Haskell está bem documentada em [7]. Haskell é uma linguagem funcional pura, não estrita, fortemente tipada, cujo nome é uma homenagem a Haskell Brooks Curry, um estudioso da lógica combinatorial e um dos mais proeminentes pesquisadores sobre λ -cálculo [2, 6]. O site oficial na WEB sobre Haskell é <http://www.haskell.org>, onde muitas informações podem ser encontradas, além de vários *links* para compiladores e interpretadores para ela.

A primeira edição do *Haskell Report* foi publicada em 1 de abril de 1990 por Paul Hudak e Philip Wadler. A partir de então, Haskell98 passou a ser considerada como a versão a ser utilizada até a definição de *Standard Haskell*. No entanto, a linguagem continua

sendo pesquisada e muitas extensões estão sendo incorporadas, como Haskell concorrente, **IDLs** (*Interface Description Language*), por exemplo **HaskellDirect** e interfaces para C e C++, permitindo integração com estas e outras linguagens. Em particular, tem sido desenvolvida **AspectH**, uma extensão de Haskell para suportar orientação a aspectos [1], além de uma extensão para **OpenGL**, entre outras.

Para produzir código executável de máquina, foram desenvolvidos vários compiladores, entre eles, **GHC** (Glasgow Haskell Compiler) disponível para ambientes UNIX (Linux, Solaris, *BSD e MacOS-X) e Windows [21]. GHC está disponível em <http://www.dcs.gla.ac.uk/fp/software/ghc/>

Lenhart Augustsson resolveu implementar **hbc**, uma referência a Haskell Brooks Curry, o pesquisador a quem a linguagem deve o nome, disponível em <http://www.cs.chalmers.se/~augustss/hbc.html>.

Também **nhc**, considerado fácil de ser instalado, muito menor que os outros compiladores, disponível para todos os padrões UNIX e escrito em Haskell 98. Foi originalmente desenvolvido por Niklas Rögemo da Universidade de Chalmers [24].

Muito importante foi o projeto do *Dataflow* do MIT, conduzido por Arvind, cuja linguagem de programação era **Id**. Em 1993, Arvind e seus colegas resolveram adotar a sintaxe de Haskell com avaliação *eager* e paralela. A linguagem resultante foi chamada de **pH** (*parallel Haskell*), o tema do livro de Nikhil e Arvind [19].

Nos últimos 5 anos, têm surgido diversas implementações de Haskell. Entre elas, podem ser citadas: **Helium**, **UHC**, **EHC** e **jhc**.

8.4. Primeiros passos

A identação em Haskell é importante. Ele usa um esquema chamado *layout* para estruturar seu código [25], a exemplo de Python. Este esquema permite que o código seja escrito sem a necessidade de sinais de ponto e vírgula explícitos.

Neste caso, a identação do texto tem um significado bem preciso, descrito em três regras fundamentais:

- se uma linha começa em uma coluna à frente (mais à direita) do início da linha anterior, é considerada a continuação da linha anterior,
- se uma linha começa na mesma coluna que a do início da linha anterior, elas são consideradas definições independentes entre si e
- se uma linha começa em uma coluna anterior (mais à esquerda) da coluna de início da linha anterior, essa linha não pertence à mesma lista de definições.

O programador pode utilizar o interpretador Hugs ou um compilador que disponibilize a biblioteca de funções pré-definidas que compõem o arquivo “Prelude.hs”. Para executar qualquer função é necessário apenas chamar esta função na linha de comandos (*prompt*) seguida de seus argumentos e apertar a tecla “Enter”. O resultado desta execução é exibido imediatamente na linha seguinte ao “prompt”. O interpretador também pode ser

usado como uma calculadora para avaliar expressões aritméticas, booleanas, logarítmicas, trigonométricas, etc.

Tabela 8.1. Alguns operadores de Haskell com suas prioridades.

Prioridade	Assoc. à esquerda	Não associativa	Assoc. à direita
9	!, !!, //, > . >	>>=	.
8			**, ^, ^^
7		% , /, ‘div’, ‘mod’, ‘rem’, ‘quot’	
6	+, -	:+	
5		\ \	: , ++, > + >
4		/=, <, <=, ==, >, >=, ‘elem’, ‘notElem’	
3			&&
2			
1		:=	
0			\$

Os operadores podem ser infixos ou pré-fixos. Normalmente os operadores aritméticos são declarados como infixos, já as funções são normalmente declaradas como pré-fixas. No entanto, os operadores infixos podem ser utilizados como pré-fixos, apenas colocando o operador entre parênteses. Por exemplo, o operador ‘+’ (infixo) pode ser aplicado como (+) 2 3 (pré-fixo). Os operadores pré-fixos também podem ser aplicados como infixos, apenas colocando o operador entre aspas simples.

8.4.1. Identificadores em Haskell

Os identificadores em Haskell são sensíveis a caracteres, ou seja, as letras maiúsculas são distintas das letras minúsculas. Os identificadores, devem ser iniciados, *sempre*, por uma letra maiúscula, se for um tipo, ou minúscula, se for um outro identificador como uma variável, uma constante ou uma função. A esta primeira letra do identificador podem ser seguidos outros caracteres, que podem ser letras maiúsculas ou minúsculas, dígitos, sublinhado ou acentos agudos.

Haskell é uma linguagem funcional pura e, como tal, não permite atribuições des-trutivas, ou seja, não é possível fazer atualizações de variáveis. Desta forma, em Haskell, as variáveis são consideradas constantes, uma vez que elas não podem ser atualizadas.

8.5. Funções em Haskell

As formas de definição de funções em Haskell têm a ver com as formas de definição de funções utilizadas na Matemática. Elas podem receber um ou mais parâmetros como en-trada (argumentos), processa-os e constrói um resultado único que é exibido como saída.

8.5.1. Construindo funções

Um tipo de dado é uma coleção de valores com as mesmas características. Por exemplo, os números inteiros, os caracteres, os strings de caracteres, etc. Os tipos das funções, em

Haskell, são declaradas por um nome identificador, seguido de ::, vindo em seguida os tipos de seus argumentos, um a um, com uma flecha (->) entre eles e, finalmente mais uma flecha seguida do tipo do resultado que a aplicação da função produz.

O tipo da função é declarado em sua forma “*currificada*”, onde uma função de *n* argumentos é considerada como *n* funções de *um* único argumento [3].

Além dos tipos, a declaração de uma função exibe explicitamente como o processamento de seus argumentos deve ser feito. Por exemplo, verifiquemos a definição da função **somaAmbos**, a seguir. A forma como ela processa seus argumentos, x e y, é somando-os, x + y.

```
somaAmbos :: Int -> Int -> Int
somaAmbos x y = x + y
```

Em Haskell, a definição é feita por justaposição dos argumentos, x e y, ao nome da função.

Outra forma de definição de funções, em Haskell, é declará-las usando definições de outras funções. Por exemplo, a função *mmc* (mínimo múltiplo comum) entre dois valores inteiros, pode ser definida em função das funções *mdc* (máximo divisor comum) e *div* (divisão inteira) entre eles. Vejamos as definições, em Haskell:

```
div :: Int -> Int -> Int
div x y
  | x == y      = 1
  | x > y       = 1 + div x-y y
  | otherwise    = 0
mdc :: Int -> Int -> Int
mdc x y
  | x == y      = x
  | x > y       = mdc x-y y
  | otherwise    = mdc y x
mmc :: Int -> Int -> Int
mmc m n = div m*n (mdc m n)
```

Em Haskell as declarações explícitas dos tipos das funções é opcional. Mesmo assim, encoraja-se que ela seja feita, como forma de disciplina de programação. Isto permite ao programador um entendimento melhor a cerca do problema e da solução adotada.

8.5.2. Casamento de padrões (patterns matching)

O casamento de padrões é outra forma de codificação de funções em Haskell, baseada na definição por enumeração utilizada na Matemática. Neste tipo de definição, são exibidos todos os valores que os argumentos da função podem ter e, para cada um deles, declara-se o valor do resultado correspondente. Por exemplo, vejamos as declarações das funções **eZero** e **fat**, a seguir:

```
eZero :: Int -> Bool           fat :: Int -> Int
```

```

eZero 0 = True           e   fat 0 = 1
eZero _ = False          fat n = n * fat (n - 1)

```

Na execução de aplicações destas funções, os padrões são testados sequencialmente, de cima para baixo. O primeiro padrão que casa com o valor da entrada terá o valor correspondente como resultado da aplicação da função. Se não ocorrer qualquer casamento entre o valor de entrada e um padrão de entrada, o resultado será um erro.

Esta forma de análise seqüencial deve sempre ser levada em consideração para que erros grosseiros sejam evitados. Como exemplo, se, na declaração da função **eZero**, a definição da função para o caso de **n** ser 0 for trocada pela segunda definição, o resultado da aplicação da função será sempre igual a **False**, mesmo que o argumento seja 0, uma vez que o *_* (*underscore*) significa “qualquer caso”.

8.5.3. Definições locais

Haskell permite definições locais através da palavra reservada **where**. Por exemplo,

```

somaQuadrados :: Int -> Int -> Int
somaQuadrados n m = quadN + quadM
    where
        quadN = n * n
        quadM = m * m

```

As definições locais podem incluir outras definições de funções, além de poder usar definições locais a uma expressão, usando a palavra reservada **let**.

```
let x = 3 + 2; y = 5 - 1 in x^2 + 2*x*y - y
```

8.5.4. Expressões condicionais

Haskell admite também expressões condicionais como todas as outras linguagens. Neste caso, ao usar a construção **if** <**expB**> **then** <**exp1**> **else** <**exp2**>, a expressão booleana <**expB**> é avaliada e, se o resultado for verdadeiro, a expressão <**exp1**> é escolhida para ser avaliada e seu resultado será o valor da expressão como um todo. Se, ao contrário, a avaliação da expressão booleana tiver valor falso, a expressão <**exp2**> é escolhida para ser avaliada e seu resultado será o da expressão.

8.6. Tipos primitivos de dados em Haskell

Haskell, como qualquer linguagem de programação, tem tipos primitivos e estruturados. Nesta seção serão analisados os tipos primitivos e os estruturados serão deixados para a próxima.

8.6.1. O tipo inteiro (Int ou Integer)

O domínio dos valores inteiros é o mesmo das outras linguagens. Os valores do tipo **Integer** são representados com o dobro da quantidade de bits necessários para representar os valores do tipo **Int**. Seus operadores aritméticos e relacionais são os mesmos admitidos na maioria das outras linguagens.

8.6.2. O tipo booleano (Bool)

Os únicos valores booleanos são **True** e **False** e sobre eles podem ser utilizadas funções pré-definidas ou funções construídas pelo usuário. As funções pré-definidas são as mesmas definidas em outras linguagens.

8.6.3. O tipo caractere (Char)

Os caracteres em Haskell são literais escritos entre aspas simples.

8.6.4. O tipo ponto flutuante (Float ou Double)

Os valores do tipo ponto flutuante (números reais) pertencem aos tipos **Float** ou **Double**, da mesma forma que um número inteiro pertence aos tipos **Int** ou **Integer**. Isto significa que as únicas diferenças entre valores destes tipos se verificam na quantidade de bits usados para representá-los. A Tabela 8.2 mostra as principais funções aritméticas pré-definidas na linguagem. As funções aritméticas recebem a denominação especial de operadores.

Tabela 8.2. Operadores aritméticos de ponto flutuante em Haskell.

+,-,*	Float -> Float -> Float
/	Float -> Float -> Float
^	Float -> Int -> Float
**	Float -> Float -> Float
==,/=,<,>,<=,>=	Float -> Float -> Bool
abs	Float -> Float
acos, asin, atan	Float -> Float
ceiling, floor, round	Float -> Float
cos, sin, tan	Float -> Float
exp	Float -> Float
fromInt	Int -> Float
log	Float -> Float
logBase	Float -> Float -> Float
negate	Float -> Float
read	String -> Float
pi	Float
show	* -> String
signum	Float -> Int
sqrt	Float -> Float

8.6.5. O tipo cadeia de caracteres (String)

Os projetistas de Haskell admitiram duas formas para este tipo. Ele é um tipo pré-definido como **String**, mas também pode ser considerado como uma lista de caracteres. Para satisfazer as duas formas, as strings podem ser escritas entre aspas duplas ou usando a notação de lista de caracteres (entre aspas simples). Por exemplo, "Constantino" tem o mesmo significado que ['C', 'o', 'n', 's', 't', 'a', 'n', 't', 'i', 'n', 'o'].

Toda aplicação de função, em Haskell, produz um resultado. Ocorre, no entanto, que para mostrar um resultado, no monitor ou em outro dispositivo de saída, é necessário definir uma função para esta tarefa. E qual deve ser o resultado desta operação?

Haskell foi projetada como uma linguagem funcional pura e, para resolver este tipo de comunicação, adota uma semântica de ações baseada em Mônadas. Uma ação é um tipo de função que retorna um valor do tipo **IO ()**. Mostraremos aqui algumas funções usadas na comunicação com o mundo exterior e formas de aplicação para facilitar o entendimento pelo leitor.

A primeira destas funções pré-definidas em Haskell é **putStr** que é utilizada para mostrar strings no monitor. Assim,

```
putStr "Dunga\teh o bicho" = Dunga      eh o bicho  
putStr "jeri"++ "qua" ++ "quara" = jeriquaquara
```

E se um valor não for uma string? Neste caso, a solução é transformar o valor em uma String e usar a função **putStr**. Para esta missão, foi definida a função **show** que transforma um valor, de qualquer tipo, em uma String.

```
show (5+7) = "12"  
show (True && False) = "False"
```

8.7. O tipo lista

Lista é o tipo de dado mais importante nas linguagens funcionais. Todas as linguagens funcionais a implementam como um tipo primitivo, juntamente com uma gama imensa de funções para a sua manipulação. A notação utilizada para as listas é colocar seus elementos entre colchetes. Por exemplo, [1,2,3,4,1,3] é uma lista de inteiros e [True,False] é uma lista de booleanos. Haskell só admite listas homogêneas, em compensação, podemos ter a lista [**totalVendas**, **totalVendas**], que tem o tipo **[Int -> Int]** e a lista [[12,1], [3,4], [4,4,4,4], []] que tem o tipo **[[Int]]**.

Existem duas formas como as listas podem ser apresentadas:

- a **lista vazia**, simbolizada por [], que pode ser de qualquer tipo. Por exemplo, ela pode ser de inteiros, de booleanos, etc. significando que a lista vazia está na interseção de todas as listas, sendo o único elemento deste conjunto.
- a **lista não vazia**, simbolizada por (a : x), onde a representa um elemento da lista, portanto tem um tipo, e x representa uma lista composta de elementos do mesmo tipo de a. O elemento a é chamado de **cabeça** e x é a **cauda** da lista.

Algumas características importantes das listas em Haskell, são:

- A ordem em uma lista é importante, ou seja, [1,3] /= [3,1] e [False] /= [False, False].

- A lista $[m .. n]$ é igual à lista $[m, m+1, ..., n]$. Por exemplo, $[1 .. 5] = [1, 2, 3, 4, 5]$. A lista $[3.1 .. 7.0] = [3.1, 4.1, 5.1, 6.1]$.
- A lista $[m, p .. n]$ é igual à lista de m até n em passos de $p-m$. Por exemplo, $[7, 6 .. 3] = [7, 6, 5, 4, 3]$ e $[0.0, 0.3 .. 1.0] = [0.0, 0.3, 0.6, 0.9]$.
- A lista $[m..n]$, para $m > n$, é vazia. Por exemplo, $[7 .. 3] = []$.
- A lista não vazia tem uma cabeça (o primeiro elemento da lista) e uma cauda, que também é uma lista, vazia, ou não.
- A lista vazia não tem cabeça e nem cauda.

8.7.1. Funções sobre listas

As funções para a manipulação de listas são declaradas usando casamento de padrões. Neste caso, os padrões são apenas dois: a lista vazia e a lista não vazia. Por exemplo, para mostrar a soma dos elementos de uma lista de inteiros, podemos definir:

```
somaLista :: [Int] -> Int
somaLista [] = 0
somaLista (a:x) = a + somaLista x
```

Se a lista for vazia ($[]$), a soma será 0. Se a lista não for vazia ($a : x$), o resultado será a soma de sua cabeça (a) com o resultado da aplicação da mesma função **somaLista** à cauda da lista (x). Esta definição é recursiva, uma característica muito utilizada, por ser a forma usada para fazer iteração em programas funcionais. Devemos observar que a ordem em que os padrões são colocados tem importância fundamental.

O construtor de listas, chamado de **cons** e sinalizado por : (dois pontos), é um operador que toma como argumentos um elemento, de um tipo, e uma lista de elementos deste mesmo tipo e insere este elemento como a cabeça da nova lista. Por exemplo,

```
2 : 1 : 3 : [] = 2 : 1 : [3] = 2 : [1, 3] = [2, 1, 3]
```

8.7.2. Polimorfismo

As definições em Haskell podem ser polimórficas, aumentando a produtividade de software escritos nesta linguagem. Por exemplo, a função **length** é pré-definida em Haskell, da seguinte maneira:

```
length :: [t] -> Int
length []      = 0
length (a : x) = 1 + length x
```

Esta função tem um **tipo polimórfico** porque pode ser aplicada a qualquer tipo de lista homogênea. Em sua definição não existe qualquer operação que exija que a lista parâmetro seja de algum tipo particular. A única operação que esta função faz é contar os elementos de uma lista, seja ela de que tipo for.

Tabela 8.3. Algumas funções sobre listas do Prelude.hs.

Função	Tipo	Exemplo
:	$a -> [a] -> [a]$	$3:[2,5]=[3,2,5]$
++	$[a] -> [a] -> [a]$	$[3,2]\text{++}[4,5]=[3,2,4,5]$
$!!$	$[a] -> \text{Int} -> a$	$[3,2,1]!!0=3$
concat	$[[a]] -> [a]$	$[[2],[3,5]]=[2,3,5]$
length	$[a] -> \text{Int}$	$\text{length } [3,2,1]=3$
head	$[a] -> a$	$\text{head } [3,2,5]=3$
last	$[a] -> a$	$\text{last } [3,2,1]=1$
tail	$[a] -> [a]$	$\text{tail } [3,2,1]=[2,1]$
init	$[a] -> [a]$	$\text{init } [3,2,1]=[3,2]$
replicate	$\text{Int} -> a -> [a]$	$\text{replicate } 3 \text{ 'a'}=[\text{'a'},\text{'a'},\text{'a'}]$
take	$\text{Int} -> [a] -> [a]$	$\text{take } 2 [3,2,1]=[3,2]$
drop	$\text{Int} -> [a] -> [a]$	$\text{drop } 2 [3,2,1]=[1]$
splitAt	$\text{Int} -> [a] -> ([a], [a])$	$\text{splitAt } 2 [3,2,1]=([3,2],[1])$
reverse	$[a] -> [a]$	$\text{reverse } [3,2,1]=[1,2,3]$
zip	$[a] -> [b] -> [(a, b)]$	$\text{zip } [3,2,1][5,6]=[(3,5),(2,6)]$
unzip	$[(a, b)] -> ([a], [b])$	$\text{unzip } [(3,5),(2,6)]=([3,2],[5,6])$
and	$\text{Bool} -> \text{Bool}$	$\text{and } [\text{True},\text{False}]=\text{False}$
or	$\text{Bool} -> \text{Bool}$	$\text{or } [\text{True},\text{False}]=\text{True}$
sum	$\text{Int} -> \text{Int}$ $\text{Float} -> \text{Float}$	$\text{sum } [2,5,7]=14$ $\text{sum } [3.0,4.0,1.0]=8.0$
product	$\text{Int} -> \text{Int}$ $\text{Float} -> \text{Float}$	$\text{product } [1,2,3]=6$ $\text{product } [1.0,2.0,3.0]=6.0$

Em Haskell, já existe um grande número de funções pré-definidas para a manipulação de listas. Estas funções fazem parte do arquivo **Prelude.hs**, carregado no momento em que o sistema é chamado. Um resumo destas funções, com seus tipos e exemplos de utilização, pode ser visto na Tabela 8.3.

Além das funções pré-definidas, o usuário também pode construir funções para manipular listas. Aqui a criatividade é o limite. Vamos mostrar isto através de um exemplo.

Exemplo. Uma forma de ordenar uma lista não vazia é inserir a cabeça da lista no local correto, que pode ser na cabeça da lista ou pode ser na cauda já ordenada. Por exemplo, para ordenar a lista de inteiros $[3,4,1]$, devemos inserir 3 na cauda da lista, já ordenada, ou seja em $[1,4]$. Para que esta cauda já esteja ordenada é necessário apenas chamar a mesma função de ordenação, recursivamente, para ela. Vamos definir uma função de ordenação, **ordena**, que utiliza uma função auxiliar **insere**, cuja tarefa é inserir cada elemento da lista no lugar correto.

```
ordena :: [Int] -> [Int]
ordena [] = []
```

```
ordena (a:x) = insere a (ordena x)
```

A lista vazia é considerada ordenada por vacuidade. Por isto a primeira definição. Para o caso da lista não vazia, devemos inserir a cabeça (*a*) na cauda já ordenada (**ordena x**). A função **insere** é auto-explicativa.

```
insere :: Int -> [Int] -> [Int]
insere a [] = [a]
insere a (b:y)
| a <= b = a : (b : y)           -- a sera a cabeca
| otherwise = b : insere a y --poe no local correto
```

Este método de ordenação é conhecido como *inserção direta* e deve ser observada a simplicidade como ele é implementado em Haskell quando comparado com uma implementação em uma linguagem convencional.

Exemplo (baseado em [32]). Seja um banco de dados definido para contabilizar as retiradas de livros de uma Biblioteca, por várias pessoas. Para simular esta situação, vamos construir uma lista de tuplas compostas pelo nome da pessoa que tomou emprestado um livro e do título do livro. Para isto, teremos:

```
type Pessoa = String
type Livro = String
type BancodeDados = [(Pessoa, Livro)]
```

Podemos definir funções para realizar as seguintes tarefas:

1. Operações de consulta:

- Uma função que informe os livros que uma determinada pessoa tomou emprestado.
- Uma função que informe todas as pessoas que tomaram emprestado um determinado livro.
- Uma função que informe se um determinado livro está ou não emprestado.
- Uma função que informe a quantidade de livros que uma determinada pessoa tomou emprestado.

2. Operações de atualização:

- Uma função que atualiza o banco, quando um livro é emprestado a alguém.
- Uma função que atualiza o banco quando um livro é devolvido.

Vamos definir a função **livrosEmprestados** que pode ser utilizada como roteiro para a definição das outras funções de consulta.

```

livrosEmprestados :: BancodeDados -> Pessoa -> [Livro]
livrosEmprestados [ ] _ = [ ]
livrosEmprestados ((inquilino, titulo) : resto) fulano
| inquilino == fulano =
  titulo : livrosEmprestados resto fulano
| otherwise = livrosEmprestados resto fulano

```

As funções de atualização podem ser definidas da seguinte forma:

```

tomaEmprestado :: 
  BancodeDados -> Pessoa -> Livro -> BancodeDados
tomaEmprestado dBase pessoa titulo =
  (pessoa, titulo) : dBase

devolveLivro :: 
  BancodeDados -> Pessoa -> Livro -> BancodeDados
devolveLivro ((p, t) : r) f l
| p == f && t == l = r
| otherwise = (p,t) : devolveLivro r f l
devolveLivro [ ] ful tit =
  error "Nao ha livro emprestado"

```

8.7.3. Compreensões e expressões ZF (Zermelo-Fraenkel)

As comprehensões, também conhecidas como expressões ZF, são devidas a Zermelo e Fraenkel e representam uma forma muito rica de construção de listas. O domínio desta técnica permite ao programador resolver muitos problemas.

Vamos supor que **ex** = [2,4,7]. Usando **ex**, podemos construir **ex1**, a lista cujos elementos sejam o dobro dos elementos de **ex**, da seguinte forma:

```
ex1 = [2*a | a<-ex]
```

Desta forma **ex1** = [4,8,14]. Se quizermos encontrar a lista **ex2** composta dos elementos de **ex** que sejam pares, podemos declarar

```
ex2 = [a | a<-ex, a 'mod' 2 == 0]
```

Neste caso, **ex2** = [2,4].

A partir destes exemplos, podemos verificar que a sintaxe das expressões ZF é realmente simples. Formalmente ela é dada da seguinte forma:

[**e** | **q**₁, ..., **q**_k] onde cada **q**_i é um *qualificador*, que pode ter umas das seguintes formas:

1. pode ser um *gerador* do tipo **p** < -**IExp**, onde **p** é um padrão e **IExp** é uma expressão do tipo lista, ou

2. pode ser um teste do tipo **bExp**, uma expressão booleana.

Os geradores podem ser combinados com nenhuma, uma ou mais expressões booleanas e, além disso, pode-se usar qualquer padrão à esquerda de `< -` e adicionar qualquer quantidade de testes.

```
novaSomaPares :: [(Int, Int)] -> [Int]
novaSomaPares listadePares =
    [a + b | (a, b) <- listadePares, a < b]
novaSomaPares [(2,3), (5,4), (7,6)] = [5]
```

8.7.3.1. O algoritmo quicksort

O algoritmo de ordenação conhecido como **quicksort** pode ser implementado, em Haskell, destacando-se a simplicidade como isto é feito. O algoritmo utiliza o método de divisão e conquista em seu desenvolvimento. Em sua implementação, escolhe-se um elemento, o *pivot*, e a lista a ser ordenada é dividida em duas sub-listas: uma contendo os elementos menores ou iguais ao *pivot* e a outra contendo os elementos da lista que sejam maiores que o *pivot*. Neste ponto, o algoritmo é aplicado recursivamente à primeira e à segunda sub-lista, concatenando seus resultados, com o *pivot* entre eles. A escolha do *pivot*, normalmente, é feita pelo elemento do meio da lista, no entanto, em nossa implementação, ele será a cabeça da lista, por ser o elemento mais fácil de ser obtido.

```
quicksort :: [t] -> [t]
quicksort [] = []
quicksort (a : x) =
    quicksort [y | y <- x, y <= a] ++ [a] ++
    quicksort [y | y <- x, y > a]
```

Sugiro ao leitor comparar esta implementação com as equivalentes em outras linguagens.

8.8. Funções de alta ordem

Possivelmente não seja normal para alguns programadores a idéia de listas de funções ou de funções que retornam outras funções com resultados. Esta é uma característica importante das linguagens funcionais. A idéia central é a de que as funções são consideradas com os mesmos direitos que qualquer outro tipo de dado, dizendo-se, corriqueiramente, que “elas são cidadãs de primeira categoria” [18].

As funções de alta ordem são usadas de forma intensa em programas funcionais, permitindo que computações complexas sejam expressas de forma simples. Vejamos algumas destas funções, muito utilizadas na prática da programação funcional.

8.8.1. A função map

Um padrão de computação que é explorado como função de alta ordem envolve a criação de uma lista (**listaNova**) a partir de uma outra lista (**listaVelha**), onde cada elemento de **listaNova** tem o seu valor determinado através da aplicação de uma função a cada elemento de **listaVelha**.

Suponhamos que se deseja transformar uma lista de nomes em uma nova lista de tuplas, em que cada nome da primeira lista é transformado em uma tupla, onde o primeiro elemento seja o próprio nome e o segundo seja a quantidade de caracteres do nome. Vamos criar a função auxiliar **tuplaNum** que, aplicada a um nome, retorna a tupla formada pelo nome e a quantidade de caracteres do nome.

```
tuplaNum :: String -> (String, Int)
tuplaNum s = (s, length s)
```

Pode-se construir uma função que aplica esta função a cada elemento da lista de nomes. A função a ser aplicada é passada como parâmetro para a função de alta ordem. Neste caso, a função de alta ordem é chamada de “*mapeamento*”, simbolizado pela função pré-definida **map**. A função **listaTupla** pode ser definida da seguinte forma:

```
listaTupla :: [String] -> [(String, Int)]
listaTupla xs = map tuplaNum xs
```

8.8.2. A função fold

Uma outra função de alta ordem, também de grande utilização em aplicações funcionais, é a função **fold**, usada na redução de ítems. Ela toma como argumentos uma função de dois argumentos e a aplica aos elementos de uma lista. O resultado é um elemento do tipo dos elementos da lista. Vamos ver sua definição formal e exemplos de sua aplicação.

Exemplos:

fold () [False, True, False]	= () False (b old () [True, False])
	= () False (() True (b old () [False]))
	= () False (() True False)
	= () False True
	= True
fold (++) ["Chico", "Afonso", , "!"]	= "Chico Afonso!"
fold (*) [1..6]	= 720 (Verifique!)

8.8.3. A função filter

A função **filter** é uma outra função de alta ordem, pré-definida em todas as linguagens funcionais. Resumidamente, ela escolhe dentre os elementos de uma lista, aqueles que têm uma determinada propriedade. Vejamos o exemplo a seguir.

Exemplo. Um número natural é perfeito se a soma de seus divisores, incluindo o número 1, for o próprio número. Por exemplo, 6 é o primeiro número natural perfeito, porque $6 = 1+2+3$. Vamos definir uma função que mostra os números perfeitos entre 0 e m.

```

divide :: Int -> Int -> Bool
divide n a = n `mod` a == 0

fatores :: Int -> [Int]
fatores n = filter (divide n) [1..(n `div` 2)]

perfeito :: Int -> Bool
perfeito n = sum (fatores n) == n

perfeitos m = filter perfeito [0..m]

```

8.9. Composição de funções

Uma forma simples de estruturar um programa é construí-lo em etapas, uma após outra, onde cada uma delas pode ser definida separadamente. Em programação funcional, isto é feito através da composição de funções, uma propriedade matemática só implementada nestas linguagens.

A expressão $f(g(x))$, normalmente, é escrita pelos matemáticos como $(f \circ g)(x)$, onde \circ (ponto) é o operador de composição de funções. Esta notação é importante porque separa a parte das funções da parte dos argumentos.

Como é sabido, nem todo par de funções pode ser composto. O tipo da saída da função **g** tem de ser o mesmo tipo da entrada da função **f**. O tipo de \circ é: $(\cdot) :: (\mathbf{u} \rightarrow \mathbf{v}) \rightarrow (\mathbf{t} \rightarrow \mathbf{u}) \rightarrow (\mathbf{t} \rightarrow \mathbf{v})$.

8.10. Aplicação parcial

Uma característica importante de Haskell e que proporciona uma forma elegante e poderosa de construção de funções é a avaliação parcial que consiste na aplicação de uma função a menos argumentos que ela realmente precisa. Por exemplo, seja a função **multiplica** que retorna o produto de seus argumentos:

```

multiplica :: Int -> Int -> Int
multiplica a b = a * b

```

Esta função foi declarada para ser usada com dois argumentos. No entanto, ela pode ser chamada como **multiplica 2**. Esta aplicação retorna uma outra função que, aplicada a um argumento **b**, retorna o valor $2 \cdot b$. Esta característica é o resultado do seguinte princípio em Haskell: “*uma função com n argumentos pode ser aplicada a r argumentos, onde $r \leq n$* ”. Como exemplo, a função **dobraLista** pode ser definida da seguinte forma:

```

dobraLista :: [Int] -> [Int]
dobraLista = map (multiplica 2)

```

8.10.1. Seção de operadores

Como decorrência direta das aplicações parciais, Haskell implementa as seções de operadores. As seções são operações parciais, normalmente relacionadas com as operações

aritméticas. Nas aplicações, elas são colocadas entre parênteses. Por exemplo,

- $(+2)$ é a função que adiciona algum argumento a 2,
- $(2+)$ a função que adiciona 2 a algum argumento,

Uma seção de um operador **op** coloca o argumento no lado que completa a aplicação. Por exemplo, $(\mathbf{op} \mathbf{a}) \mathbf{b} = \mathbf{b} \mathbf{op} \mathbf{a}$ e $(\mathbf{a} \mathbf{op}) \mathbf{b} = \mathbf{a} \mathbf{op} \mathbf{b}$.

8.11. Classes de tipos

Já foram vistas funções que atuam sobre valores de mais de um tipo. Por exemplo, a função **length** pode ser empregada para determinar o tamanho de listas de qualquer tipo. Assim, **length** é uma função *polimórfica*, ou seja, com apenas uma única definição, ela pode ser aplicada a uma variedade de tipos. Por outro lado, algumas funções podem ser aplicadas a apenas alguns tipos de dados, mas não a todos, e têm de apresentar várias definições, uma para cada tipo. São os casos das funções $+$, $-$, $/$, etc. Estas funções são *sobrecarregadas*.

Os tipos sobre os quais uma função sobrecarregada pode atuar formam uma coleção de tipos chamada *classe de tipos* (ou **type class**) em Haskell. Um tipo que pertence a uma classe é dito ser uma *instância* desta classe.

8.11.1. Fundamentação das classes

A função **elem**, aplicada a um elemento e uma lista de valores do tipo deste mesmo elemento, verifica se este elemento pertence, ou não, à lista, retornando um valor booleano. Sua definição é a seguinte:

```
elem :: t -> [t] -> Bool
elem x [] = False
elem x (a : y) = x == a || elem x y
```

Analizando a definição da função aplicada à lista não vazia, verificamos que é feita uma comparação entre o elemento **x** e a cabeça da lista (**x==a**). Para este teste é utilizada a função de igualdade ($==$). Isto implica que a função **elem** só pode ser aplicada a tipos cujos valores possam ser comparados pela função $==$. Os tipos que têm esta propriedade formam uma classe que, em Haskell, é denotada por **Eq**.

O objetivo principal de se introduzir um teste de igualdade é ser capaz de usá-lo em uma variedade de tipos distintos, não apenas no tipo **Bool**. Em outras palavras, $==$ e $/=$ são operadores sobrecarregados. Estas operações serão definidas de forma diferente para cada tipo e a forma adequada de introduzí-las é declarar uma classe de todos os tipos para os quais $==$ e $/=$ vão ser definidas.

A forma de declarar **Eq** como a classe dos tipos que têm os operadores $==$ e $/=$ é a seguinte:

```
class Eq t where
  (==), (/=) :: t -> t -> Bool
```

Esta declaração estabelece que a classe de tipos **Eq** contém duas funções membros, ou métodos: `==` e `/=`. A definição inclui o nome da classe (**Eq**) e uma *assinatura*, que são as funções que compõem a classe juntamente com seus tipos. Um tipo **t** para pertencer à classe **Eq** tem de implementar as duas funções da assinatura.

Para que o tipo **Bool** seja uma instância da classe **Eq** devemos fazer:

```
instance Eq Bool where
    True == True = True
    False == False = True
    _ == _ = False
    x /= y = not (x == y)
```

Haskell contém algumas classes pré-definidas e elas podem ser encontradas em qualquer referência bibliográfica indicada, como [32, 4, 2, 25], entre outras.

8.12. Tipos algébricos

Haskell oferece uma forma especial para a construção de novos tipos de dados visando modelar situações peculiares, por exemplo, um tipo árvore.

Para construir um novo tipo de dados, usamos a declaração **data** que descreve como os elementos deste novo tipo de dados são construídos. Cada elemento é nomeado por uma expressão formulada em função dos construtores do tipo. Os tipos assim descritos, não as operações, são chamados “tipos concretos” [32, 2] e são modelados em Haskell através dos *tipos algébricos*.

A sintaxe de uma declaração de um tipo algébrico de dados é

```
data <Nome_do_tipo> =
    <Const1> | <Const2> | ... | <Constn>
```

onde **Const1**, **Const2**, ... **Constn** são os construtores do tipo.

Os tipos enumerados são tipos algébricos utilizados para modelar a união disjunta de conjuntos. Como exemplo destes tipos podemos citar:

```
data Tempo = Frio | Quente
data Estacao = Primavera | Verao | Outono | Inverno
```

Neste caso, **Frio** e **Quente** são os construtores do tipo **Tempo** e **Primavera**, **Verao**, **Outono** e **Inverno** são os construtores do tipo **Estacao**.

As funções sobre estes tipos são declaradas usando *pattern matching*. Para descrever o tempo das estações podemos usar:

```
clima :: Estacao -> Tempo
clima Verao = Quente
clima _ = Frio
```

Uma forma geométrica pode ser um círculo ou um retângulo. Então podemos modelá-la da seguinte maneira:

```
data Forma = Circulo Float | Retangulo Float Float
```

Agora podemos declarar funções que utilizam este tipo de dados, por exemplo, a função que calcula a área de uma forma geométrica:

```
area :: Forma -> Float
area (Circulo r) = pi * r * r
area (Retangulo h w) = h * w
```

Ao se introduzir um tipo algébrico, como **Estacao**, podemos desejar que ele também tenha igualdade. Isto pode ser feito informando que o tipo tem as mesmas funções que a classe **Eq**:

```
data Estacao = Primavera | Verao | Outono | Inverno
              deriving Eq
```

8.13. Árvores

As listas, aqui já analisadas, são consideradas estruturas de dados lineares porque seus ítems aparecem em uma seqüência pré-determinada, ou seja, cada ítem tem, no máximo, um sucessor imediato. As árvores, são estruturas *não lineares* porque seus ítems podem ter mais de um sucessor imediato.

As árvores podem ser utilizadas como representações naturais para quaisquer formas de dados organizados hierarquicamente, por exemplo, as estruturas sintáticas das expressões aritméticas. Na realidade, as árvores provêem uma generalização eficiente das listas como formas de organização e recuperação de informações.

8.13.1. Árvores de buscas binárias

Podemos definir árvores de buscas binárias levando em consideração a existência de uma árvore nula, da seguinte forma:

```
data (Ord t) =>
    ArvBusBin t = Nil
                | No (ArvBusBin t) t (ArvBusBin t)
```

Este é o exemplo de uma declaração de tipo de dado que utiliza um contexto, ou seja, a árvore do tipo **ArvBusBin t** é definida para tipos **t** que sejam instâncias da classe **Ord**. Isto significa que todos os valores dos nós internos desta árvore podem ser comparados pelas relações *maior*, *maior ou igual*, *menor*, *menor ou igual* e *igual*. Esta definição não apresenta diferença entre nós internos e externos e as informações armazenadas nos nós aparecem entre as subárvores componentes. Este tipo de árvore também é conhecida como “árvore binária rotulada”. A função **arvBBtolista** que transforma uma árvore de busca binária em uma lista pode ser definida da seguinte forma:

```

arvBBtolista :: (Ord t) => ArvBusBin t -> [t]
arvBBtolista Nil = []
arvBBtolista (No xt n yt) =
    arvBBtolista xt ++ [n] ++ arvBBtolista yt

```

Existe uma condição sobre os elementos do tipo **ArvBusBin t** que faz com que ela seja uma árvore de busca binária. Esta condição é que uma aplicação da função **arvBBtolista** deve retornar uma lista ordenada, sendo este o motivo da exigência de que os elementos do tipo **t** estejam na classe **Ord**. Formalmente, **xt** é uma árvore de busca binária se estiver ordenada na forma *in-order*.

As árvores de busca binárias podem ser utilizadas para realizar buscas de forma eficiente. A função **membroABB** que determina se um dado valor aparece, ou não, como um rótulo de algum nó pode ser definida da seguinte forma:

```

membroABB :: t -> ArvBusBin t -> Bool
membroABB x Nil = False
membroABB x (No xt n yt)
| (x < n)      = membroABB x xt
| (x == n)     = True
| (x > n)      = membroABB x yt

```

Por outro lado, uma árvore de busca binária pode ser construída a partir de uma lista de valores. Para isto, vamos definir a função **fazABB**:

```

fazABB :: (Ord t) => [t] -> ArvBusBin t
fazABB [] = Nil
fazABB (x:xs) = No (fazABB ys) x (fazABB zs)
                where (ys, zs) = particao (<= x) xs

particao :: (t -> Bool) -> [t] -> ([t], [t])
particao p xs = (filter p xs, filter (not . p) xs)

```

8.13.1.1. Inserção e remoção

As árvores de busca binária podem ser utilizadas em várias situações. Por exemplo, vamos definir funções de *inserção* e *remoção* de ítems nos nós destas árvores. A função de inserção será **insere**:

```

insere :: (Ord t) => t -> ArvBusBin t -> ArvBusBin t
insere x Nil = No Nil x Nil
insere x (No xt n yt)
| (x < n)  = No (insere x xt) n yt
| (x == n) = No xt n yt
| (x > n)  = No xt n (insere x yt)

```

A função de remoção é um pouco mais complexa e será definida como **remove**:

```
remove :: (Ord t) => t -> ArvBusBin t -> ArvBusBin t
remove x Nil = Nil
remove x (No xt n yt)
| (x < n) = No (remove x xt) n yt
| (x > n) = No xt n (remove x yt)
| (x == n) = junta xt yt
```

Uma implementação possível para a função **junta xt yt** consiste em selecionar o menor elemento da subárvore direita, **yt**, e colocá-lo como a nova raiz da árvore resultante desta junção das duas subárvores. Isto significa que esta nova raiz será removida da subárvore da direita, **yt**. Esta implementação pode ser feita da seguinte forma:

```
junta :: (Ord t) =>
          ArvBusBin t -> ArvBusBin t -> ArvBusBin t
junta Nil yt = yt
junta xt Nil = xt
junta (No uxt x vxt) (No uyt y vyt)
= (No uxt x vxt) k (remove k (No uyt y vyt))
  where k = minArv (No uyt y vyt)

minArv :: (Ord t) => ArvBusBin t -> t
minArv (No Nil x _) = x
minArv (No xt _ _) = minArv xt
```

Esta seção mostrou uma forma de construção do tipo árvore de busca binária. Deve ser notado que os outros tipos de árvores, por exemplo, as árvore AVL, árvores B e B+, estruturas TRIE, além de outros, também são possíveis e de forma muito mais fácil que em qualquer linguagem convencional.

8.14. Módulos em Haskell

Para John Hughes [8], a modularidade é a característica que uma linguagem de programação deve apresentar para que seja utilizada com sucesso. Ele afirma que esta é a característica principal das linguagens funcionais, proporcionada através das funções de alta ordem e do mecanismo de avaliação *lazy*.

A modularidade é importante porque:

- as partes de um sistema podem ser construídas separadamente,
- as partes de um sistema podem ser compiladas e testadas separadamente e
- permite a criação de grandes bibliotecas.

Para definir o módulo **Forma** deve-se fazer a declaração a seguir, que deve constituir o arquivo Forma.hs ou Forma.lhs. Cada módulo em Haskell constitui um arquivo.

```
module Forma where
  data Formas = . . .
  fazForma x = . . .
```

Em todo sistema deve existir um módulo chamado **Main** que deve conter a definição da função **main**. Em um sistema interpretado, como Hugs, ele tem pouca importância porque um módulo sem um nome explícito é tratado como **Main**.

8.14.1. Importação de módulos

Os módulos, em Haskell, podem importar dados e funções de outros módulos da seguinte forma:

```
module Retangulo where
  import Forma
  fazRetangulo = . . .
```

As definições *visíveis* no módulo **Forma** podem ser utilizadas no módulo **Retangulo**.

```
module Circulo where
  import Retangulo
```

Neste caso, as definições do tipo **Formas** e da função **fazForma** não são visíveis em **Circulo**. Elas podem se tornar visíveis pela importação explícita de **Forma** ou usando os controles de exportação para modificar o que é exportado a partir de **Retangulo**.

8.14.2. Controles de exportação e importação

É necessário poder controlar o que deve ou não ser exportado. Em Haskell, isto é declarado da seguinte forma: **module Retangulo (fazRetangulo, Formas(. . .), fazForma) where . . .** ou equivalentemente **module Retangulo (module Retangulo, module Forma) where . . .**

A palavra **module** dentro dos parênteses significa que tudo dentro do módulo é exportado, ou seja, **module Retangulo where** é equivalente a **module Retangulo (module Retangulo) where**.

Além dos controles para exportação, Haskell também apresenta controles para a importação de definições.

```
module Esfera where
  import Forma (Formas (. . .))
```

Neste caso, a intenção é importar apenas o tipo **Formas** do módulo **Forma**. Haskell também provê uma forma explícita de esconder alguma entidade. Por exemplo,

```
module Esfera where
  import Forma hiding (fazForma)
```

8.15. Tipos de dados abstratos

Um tipo abstrato não é definido pela nomeação de seus valores, mas pela nomeação de suas operações. Isto significa que a representação dos valores dos tipos abstratos não é conhecida. O que é realmente de conhecimento público é o conjunto de funções para manipular o tipo. Em geral, o programador que usa um tipo abstrato não sabe como seus elementos são representados. Tais tipos de abstração são usuais quando mais de um programador estiverem trabalhando em um mesmo projeto ou mesmo quando um mesmo programador estiver trabalhando em um projeto não trivial. Isto permite que a representação seja trocada sem afetar a validade dos *scripts* que usam o tipo abstrato. Vamos mostrar estes fundamentos através de um exemplo.

8.15.1. O tipo abstrato Pilha

As pilhas são estruturas de dados homogêneas onde os valores são colocados e/ou retirados utilizando uma estratégia **LIFO (Last In First Out)**.

As operações necessárias para o funcionamento de um TAD juntamente com seus tipos, formam a *assinatura* do tipo abstrato. A implementação de um tipo abstrato de dados em Haskell é feita através da criação de um módulo, que contém a interface e a implementação. Vejamos uma implementação baseada em tipos algébricos.

```
module Stack(Stack, push, pop, top, stackEmpty,
            newStack) where
push      :: t -> Stack t -> Stack t
            --coloca um item no topo da pilha
pop       :: Stack t -> Stack t
            --retira um item do topo da pilha
top       :: Stack t -> t
            --pega o item do topo da pilha
stackEmpty :: Stack t -> Bool
            --verifica se a pilha estah vazia
newStack   :: Stack t
            --cria uma pilha vazia

data Stack t = EmptyStk
              | Stk t (Stack t)

push x s = Stk x s

pop EmptyStk  = error "retirada em uma pilha vazia"
pop (Stk _ s) = s

top EmptyStk  = error "topo de uma pilha vazia"
top (Stk x _) = x

newStack = EmptyStk
```

```

stackEmpty EmptyStk = True
stackEmpty _         = False

instance (Show t) => Show (Stack t) where
    show (EmptyStk) = "#"
    show (Stk x s) = (show x) ++ " | " ++ (show s)

```

A utilização do tipo abstrato **Stack** deve ser feita em outros módulos cujas funções necessitem deste tipo de dado. Para isso, o módulo **Stack** deve constar na relação dos módulos importados pelo módulo usuário. Por exemplo,

```

module Main where import Stack

listaParaPilha :: [t]    -> Stack t
listaParaPilha []        = newStack
listaParaPilha (x : xs) = push x (listaParaPilha xs)

pilhaParaLista :: Stack t -> [t]
pilhaParaLista s
| stackEmpty s = []
| otherwise     = (top s) : (pilhaParaLista (pop s))

```

Este *script* deve ser salvo no arquivo **Main.hs** e deve ser carregado para ser utilizado.

A implementação mostrada foi baseada no tipo algébrico **Stack t**. No entanto, o programador pode desenvolver uma outra implementação, mais eficiente que esta, e fazer a mudança da implementação anterior para esta sem que os usuários precisem saber disto. Neste caso, as funções do módulo **Main** não precisam ser refeitas para serem utilizadas. O que não pode ser modificada é a *interface* com o usuário.

8.16. Entrada e Saída em Haskell

Um programa funcional consiste em uma expressão que é avaliada para encontrar um valor que deve ser retornado e ligado a um identificador. Quando as operações não se tratarem de expressões, por exemplo, no caso de uma operação de **IO**, uma operação de escrita de um valor na tela do monitor, que valor deve ser retornado? Este retorno é necessário para que o paradigma seja obedecido. Caso contrário, ele é corrompido.

A solução adotada pelos idealizadores de Haskell foi introduzir um tipo especial chamado “ação”. Quando o sistema detecta uma operação deste tipo, ele sabe que uma ação deve ser executada. Existem ações primitivas, por exemplo escrever um caractere em um arquivo ou receber um caractere do teclado, mas também ações compostas como imprimir uma string inteira em um arquivo.

As operações, em Haskell, cujos resultados de suas avaliações sejam ações, são chamadas de “comandos”, porque elas comandam o sistema para realizar alguma ação. Todos os comandos realizam ações e retornam um valor de um determinado tipo **T**, que pode ser usado, futuramente, pelo programa.

Haskell provê o tipo **IO a** para permitir que um programa faça alguma operação de I/O e retorne um valor do tipo **a**. Haskell também provê um tipo **IO ()** que contém um único elemento, representado por **()**. Uma função do tipo **IO ()** representa uma operação de I/O (ação) que retorna o valor **()**. Semanticamente, este é o mesmo resultado de uma operação de I/O que não retorna qualquer valor. Por exemplo, a operação de escrever a string “Olha eu aqui!” pode ser entendida desta forma, ou seja, um objeto do tipo **IO ()**.

Existem funções pré-definidas em Haskell para realizar ações, além de um mecanismo para sequençializá-las, permitindo que alguma ação do modelo imperativo seja realizada, sem ferir o modelo funcional.

8.16.1. Operações de entrada

Uma operação de leitura de um caractere (**Char**), a partir do dispositivo padrão de entrada, é descrita em Haskell pela função pré-definida **getChar** do tipo:

```
getChar :: IO Char
```

De forma similar, para ler uma string, a partir do dispositivo padrão de entrada, usamos a função pré-definida **getLine** do tipo:

```
getLine :: IO String
```

As aplicações destas funções devem ser interpretadas como operações de leitura seguidas de retornos; no primeiro caso, de um caractere e, no segundo, de uma string.

8.16.2. Operações de saída

A operação de impressão de um texto, é feita por uma função que toma a string a ser escrita, escreve esta string no dispositivo padrão de saída e retorna um valor do tipo **()**. Esta função é **putStr**, pré-definida em Haskell, com o seguinte tipo:

```
putStr :: String -> IO ()
```

Agora podemos escrever “Olha eu aqui！”, da seguinte forma:

```
aloGalvao :: IO ()  
aloGalvao = putStr "Olha eu aqui!"
```

Usando **putStr** podemos definir uma função que escreva uma linha de saída cujo efeito é adicionar o caractere de nova linha ao fim da entrada passada para **putStr**.

```
putStrLn :: String -> IO ()  
putStrLn = putStr . (++ "\n")
```

Para escrever valores em geral, Haskell provê a classe **Show** com a função

```
show :: Show a => a -> String
```

que é usada para transformar valores, de vários tipos, em strings para que possam ser mostradas através da função **putStr**.

8.16.3. A notação do

A notação **do** é um mecanismo flexível, construído para suportar duas coisas em Haskell:

1. a seqüencialização de ações de I/O e
2. a captura de valores retornados por ações de I/O, para repassá-los futuramente para outras ações do programa.

Por exemplo, a função **putStrLn str**, descrita anteriormente, é pré-definida em Haskell e faz parte do Prelude padrão de Hugs. Ela realiza duas ações. a primeira é escrever a string **str** no dispositivo padrão de saída e a segunda é fazer com que o prompt salte para a próxima linha. Esta mesma operação pode ser definida utilizando-se a notação **do**, da seguinte forma:

```
putStrLn :: String -> IO ()  
putStrLn str = do putStr str  
                  putStr "\n"
```

Neste caso, o efeito da notação **do** é a seqüencialização das ações de I/O, em uma única ação. A sintaxe da notação **do** é regida pela regra do *offside* e pode-se tomar qualquer número de argumentos (ações).

Como outro exemplo, pode-se querer escrever alguma coisa *n* vezes. Uma forma mais elegante de descrevê-la é transformar a quantidade de vezes que se deseja que a ação seja repetida em um parâmetro.

```
fazNvezes :: Int -> String -> IO ()  
fazNvezes n str =  
    if n <= 1 then putStrLn str  
    else do putStrLn str  
            fazNvezes (n-1) str
```

Deve ser observada a forma de recursão na cauda utilizada na definição da função **fazNvezes**, simulando a instrução de controle **while**, tão comum nas linguagens imperativas.

Apesar de terem sido mostrados apenas exemplos de saída, as entradas também podem ser parte de um conjunto de ações seqüencializadas. Por exemplo, pode-se querer ler duas linhas do dispositivo de entrada padrão e escrever a frase “duas linhas lidas”, ao final. Isto pode ser feito da seguinte forma:

```
leia2linhas :: IO ()  
leia2linhas = do getLine  
                  getLine  
                  putStrLn "duas linhas lidas"
```

8.16.4. Arquivos, canais e descritores

Em toda programação, é necessária uma forma de comunicação do usuário com os arquivos. Já foi vista uma forma, através do comando **do**. A outra, que será vista a seguir, é através de descritores.

Para obter um descritor (do tipo **Handle**) de um arquivo é necessária a operação de abertura deste arquivo para que futuras operações de leitura e/ou escrita possam acontecer. Além disso, é necessária uma operação de fechamento deste arquivo, após suas operações terem sido realizadas, para que os dados que ele deve conter, não sejam perdidos quando o programa de usuário terminar sua execução. Estas operações são descritas em Haskell, da seguinte forma:

```
data IOMode = ReadMode | WriteMode  
            | AppendMode | ReadWriteMode  
openFile :: FilePath -> IOMode -> IO Handle  
hClose :: Handle -> IO ()
```

Por convenção, todas as funções usadas para tratar com descritores de arquivos são iniciadas com a letra *h*. Por exemplo, as funções

```
hPutChar :: Handle -> Char -> IO ()  
hPutStr :: Handle -> String -> IO ()  
hPutStrLn :: Handle -> String -> IO ()  
hPrint :: Show a => Handle -> a -> IO ()
```

são utilizadas para escrever alguma coisa em um arquivo. Já os comandos

```
hGetChar :: Handle -> IO ()  
hGetLine :: Handle -> IO ()
```

são utilizados nas operações de leituras em arquivos. As funções **hPutStrLn** e **hPrint** incluem um caractere '\n' para a mudança de linha ao final da string.

8.17. Conclusões

Este artigo teve como objetivo mostrar que Haskell pode ser utilizada em vários campos de aplicação, com muitas vantagens em relação a outras linguagens de outros paradigmas. Estas vantagens podem ser sumarizadas nos seguintes ítems:

- As linguagens funcionais produzem um código de máquina com uma melhoria de uma ordem de magnitude e, nem sempre, estes resultados são mostrados. Normalmente, se mostram fatores de 4.
- Atualmente, os sistemas não são mais construídos monoliticamente, como eram no passado. Os programas se tornaram grandes (*programming in the large*) e agora eles devem ser escritos de forma modular por programadores em tempos e locais

possivelmente distintos. A Indústria da Computação está começando a distribuir padrões como CORBA e COM para suportar a construção de *software* a partir de componentes reutilizáveis. Atualmente, os programas em Haskell já podem ser empacotados como um componente COM e qualquer componente COM pode ser chamado a partir de Haskell.

- Muito pouco da atratividade de Java tem a ver com a linguagem em si, e sim com as bibliotecas associadas. Haskell tem *Fudgets*, *Gadgets*, *Haggis*, *HOpenGL*. Haskell tem um poderoso sistema de módulos que torna suas bibliotecas fáceis de serem construídas, já tendo a biblioteca Edison com estruturas de dados eficientes. Haskell ainda tem HSQL com interfaces para uma variedade de Bancos de Dados, incluindo MySQL, Postgres, ODBC, SQLite e Oracle. Haskell ainda tem *Happy*, um gerador de *parsers* LALR, similar ao *yacc*, atualmente extendido para produzir *parsers* LR para gramáticas ambíguas.
- Inegavelmente C e C++ têm sido preferidas em muitos projetos. No entanto, muito desta preferência não se deve ao fato de C gerar um código mais rápido que o código gerado pelas linguagens funcionais. Na realidade, esta preferência se deve mais à portabilidade inegável de C. Por outro lado, as técnicas de implementação de linguagens utilizando máquinas abstratas têm se tornado muito atrativas para linguagens funcionais [15] e também para Java. Isto se deve muito ao fato de que escrever a máquina abstrata em C a torna muito mais fácil de ser portada para uma grande variedade de arquiteturas.
- Em 1998, foi adotado o padrão Haskell98 que permanece inalterado até o momento, apesar de continuar a serem incorporadas novas extensões à sua biblioteca. Atualmente, a instalação dos compiladores Haskell é uma tarefa possível de ser feita sem qualquer trauma estando disponível para várias plataformas.
- Uma linguagem para ser utilizável necessita de ferramentas para depuração e *profiler*. Estas ferramentas são fáceis de serem construídas para linguagens estritas, no entanto, são muito difíceis de serem construídas para linguagens *lazy*, onde a ordem de avaliação não é conhecida *a priori*. Verifica-se uma exceção em Haskell, onde muitas ferramentas de *profiler* já estão disponíveis e muitas outras estão em pleno desenvolvimento.
- Uma solução imperativa é mais fácil de ser entendida e de ser encontrada em livros ou artigos. Uma solução funcional demora mais tempo para ser criada, apesar de muito mais elegante. Por este motivo, muitas linguagens funcionais atuais provêem um escape para o estilo imperativo. Haskell é uma linguagem funcional pura, mas consegue imitar as atribuições das linguagens imperativas utilizando uma teoria funcional complexa que é a semântica de ações, implementadas através de mônadas.
- Se um gerente escolher uma linguagem funcional para ser utilizada em um projeto e este falhar, provavelmente ele será crucificado. No entanto, se ele escolher C ou C++ e não tiver sucesso, tem a seu favor o argumento de que o sucesso de C++ já foi verificado em inúmeros casos e em vários locais. Este quadro também vem se

modificando em relação às linguagens funcionais, principalmente Haskell, onde ela tem se popularizado muito nos últimos anos.

- Há uma década atrás, os desempenhos dos programas funcionais eram bem menores que os dos programas imperativos, mas isto tem mudado muito ultimamente. Hoje, os desempenhos de muitos programas funcionais são melhores ou pelo menos estão em “pé de igualdade” com seus correspondentes em C. Isto depende da aplicação. Java tem uma boa aceitação e, no entanto, seu desempenho é muito inferior a C, na grande maioria das aplicações. Na realidade, existem linguagens com alto desempenho que não são muito utilizadas e existem linguagens com desempenho mediano com alta taxa de utilização. Desempenho é um fator importante, mas não tem se caracterizado como um fator decisivo na escolha de uma linguagem.

Referências

- [1] ANDRADE, Carlos Anreazza Rego. *AspectH: Uma Extensão Orientada a Aspectos de Haskell*. Dissertação de Mestrado. Centro de Informática. UFPE. Recife, Fevereiro 2005.
- [2] BIRD, Richard. *Introduction to Functional Programming Using Haskell*. 2nd. Edition. Prentice Hall Series in Computer Science - Series Editors C. A. Hoare and Richard Bird. 1998.
- [3] CURRY, H. B. et FEYS, R. and CRAIG, W. *Combinatory Logic*. Volume I; North Holland, 1958.
- [4] DAVIE, Antony J. T. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge Computer Science Texts. Cambridge University Press. 1999.
- [5] HINDLAY, J. Roger. *Basic Simple Type Theory*. Cambridge Tracts in Theoretical Computer Science, 42. Cambridge University Press. 1997.
- [6] HUDAK, Paul. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. Cambridge University Press, 2000.
- [7] HUDAK, Paul; HUGHES, John; PEYTON JONES, Simon; WADLER, Philip. *A History of Haskell: being lazy with class*. January 2007.
- [8] HUGHES, John. *Why Functional Programming Matters*. In Turner D. T. Ed. Research Topics in Functional Programming. Addison-Wesley, 1990.
- [9] JOHNSSON, T. *Efficient Computation of Lazy Evaluation*. Proc. SIGPLAN'84. Symposium on Compiler Construction ACM. Montreal, 1984.
- [10] JOHNSSON, T. *Lambda Lifting: Transforming Programs to Recursive Equations*. Aspenäs Workshop on Implementation of Functional Languages. Göteborg, 1985.
- [11] JOHNSSON, T. *Target Code Generation from G-Machine Code*. Proc. Workshop on Graph Reduction, Santa Fé Lecture Notes on Computer science, Vol: 279 pp. 119-159. Springer-Verlag, 1986.

- [12] JOHNSSON, T. *Compiling Lazy Functional Languages*. Ph.D. Thesis. Chalmers University of Technology, 1987.
- [13] LANDIN, P. J. *The Mechanical Evaluation of Expressions*. Computer Journal, Vol. 6, 4. 1964.
- [14] LAUNCHBURY, J and PEYTON JONES, S. L. *State in Haskell*. Lisp and Symbolic Computation, 8(4):293-342. 1995.
- [15] LINS, Rafael Dueire et LIRA, Bruno O. *ΓCMC: A Novel Way of Compiling Functional Languages*. J. Programming Languages 1:19-40; Chapman & Hall. 1993.
- [16] LINS, Rafael Dueire et all. *Research Interests in Functional Programming*. I Workshop on Formal Methods. UFRGS. Outubro, 1998.
- [17] MACLENNAN, Bruce J. *Functional Programming Practice*. Addison-Wesley Publishing Company, Inc. 1990.
- [18] MEIRA, Sílvio Romero de Lemos. *Introdução à Programação Funcional*. VI Escola de Computação. Campinas, 1988.
- [19] NIKHIL, R. S. and ARVIND, A. *Implicit Parallel Programming in pH*. Morgan Kaufman. 2001.
- [20] OKASAKI, Chris. *Purely Functional Data Structures*. Cambridge University Press. 2003.
- [21] PEYTON JONES, S. L. *The Implementation of Functional Programming Languages*. C. A. R. Hoare Series Editor. Prentice/Hall International. 1987.
- [22] PEYTON JONES, S. L.; GORDON, A. and FINNE, S. *Concurrent Haskell*. In 23rd ACM Symposium on Principles of Programming Languages (POPL'96), pages 295-308. St Petersburg Beach, Florida. ACMPress. 1996.
- [23] RABHI, Fethi et LAPALME, Guy. *Algorithms: A Functional Programming Approach*. 2nd. edition. Addison-Wesley. 1999.
- [24] ROJEMO, Niklaus. *Garbage Collection and Memory Efficiency in Lazy Functional Languages*. Ph.D Thesis. Department of Computing Science, Chalmers University. 1995.
- [25] SÁ, Claudio Cesar. *Haskell: uma abordagem prática*. Novatec Editora Ltda. São Paulo, 2006.
- [26] SCOTT, D. *Data Types as Lattices*. SIAM Journal of Computing. Vol. 5,3. 1976.
- [27] SKIENA, Steven S. et REVILLA, Miguel A. *Programming Challenges: The Programming Context Training Manual*. Texts in Computer Science. Springer Science+Business Media, Inc. 2003.
- [28] SOUZA, Francisco Vieira. *Teoria das Categorias: A Linguagem da Computação*. Exame de Qualificação. Centro de Informática. UFPE. 1996.

- [29] SOUZA, Francisco Vieira et LINS, Rafael Dueire. *Aspectos do Comportamento Espaço-temporal de Programas Funcionais em Uni e Multiprocessadores*. X Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho. Búzios-RJ. Setembro. 1998.
- [30] SOUZA, Francisco Vieira et LINS, Rafael Dueire. *Analysing Space Behaviour of Functional Programs*. Conferência Latino-americana de Programação Funcional. Recife-Pe. Março. 1999.
- [31] SOUZA, Francisco Vieira. *Aspectos de Eficiência em Algoritmos para o Gerenciamento Automático Dinâmico de Memória*. Tese de Doutorado. Centro de Informática-UFPE. Recife. Novembro. 2000.
- [32] THOMPSON, Simon. *Haskell: The Craft of Functional Programming*. 2nd. Edition. Addison Wesley. 1999.
- [33] TURNER, David A. *A New Implementation Technique for Applicative Languages*. Software Practice and Experience. Vol. 9. 1979.
- [34] WADLER, Philip. *Comprehending Monads*. Mathematical Structures in Computer Science; 2:461-493, 1992.
- [35] WADLER, Philip. *The Essence of Functional Programming*. In 20th ACM Symposium on Principles of Programming Languages (POPL'92), pages 1-14. ACM. Albuquerque. 1992.
- [36] WADLER, Philip. *Why no ones Uses Functional Languages*. Functional Programming. ACM SIGPLAN. 2004.

Capítulo

9

Programação Paralela em Clusters de Multiprocessadores com MPI e Open MP

Francisco Heron de Carvalho Junior

Resumo

A crescente importância de aplicações de computação de alto desempenho (CAD), a disseminação das arquiteturas de clusters, e a consolidação da tecnologia de múltiplos núcleos no projeto de novos processadores têm reforçado a necessidade da disseminação das práticas de programação paralela entre programadores em geral. Este trabalho, além de fornecer uma visão geral da evolução da área de CAD, aborda como o OpenMP e o MPI, padrões de fato e de direito usados no contexto do desenvolvimento de aplicações de CAD, podem ser usados em conjunto para explorar eficientemente o desempenho de clusters de multiprocessadores.

9.1. Introdução

Nos últimos anos, tecnologias oriundas da área de computação de alto desempenho (CAD) têm se tornado tendências dominantes, fato este motivado por movimentos independentes da indústria, respectivamente do *software* e do *hardware*.

Do ponto de vista da indústria do *software*, tem emergido o interesse pelo mercado de aplicações de CAD. Tradicionalmente, estas são oriundas de diversas áreas das ciências computacionais e engenharia, possuindo impacto significativo no potencial na produção de conhecimento e tecnologia por aqueles que detém tecnologias capazes de viabilizá-las, notadamente oriundos de instituições acadêmico-científicas e indústrias em diversas áreas de potencial emergente de mercado. Vale ainda ressaltar que, em tempos recentes, as técnicas de computação de alto desempenho têm sido aplicada com sucesso na área financeira e comercial [Grant 2007].

Do ponto de vista da indústria do *hardware*, já a partir de meados da década de 1990, observa-se: (1) a proliferação dos *clusters*, computadores paralelos de atraente custo/benefício constituídos com *hardware* de prateleira e *software* livre e aberto, permitindo uma maior base de usuários potenciais para arquiteturas de computação paralela devido a redução do custo de implantação destas em relação às tradicionais arquiteturas proprietárias de supercomputação [Baker et al 1999]; (2) o surgimento do conceito de *grade computacional* e a constante evolução das tecnologias para o seu suporte, oferecendo a possibilidade de virtualização de recursos computacionais

geograficamente distribuídos e conectados por meio da *internet* [Foster e Kesselman 2004]; e (3) a adoção de tecnologias de paralelismo vetorial, como MMX, SSE, SSE-2, SSE-3 e 3DNow!, e, principalmente, paralelismo em nível de *threads*, como *simultaneous multi-threading* e arquiteturas de *múltiplos núcleos*, em processadores de computadores de uso pessoal. Este contexto evidencia a importância do conhecimento de tecnologias de computação paralela por programadores e usuários em geral.

Apesar da notória evolução do desempenho das arquiteturas de computação de alto desempenho, ainda obedecendo a Lei de Moore¹, o mesmo não tem ocorrido com a evolução das tecnologias de *software* capaz de explorar o desempenho máximo destas arquiteturas [Dongarra *et al* 2003, Carvalho-Junior 2005]. Este fato tem sido observado e discutido pela comunidade científica desde meados da década de 1980, motivando a implantação de vários programas de pesquisa² voltados ao desenvolvimento de novos artefatos de programação paralela e distribuída capazes de conciliar requisitos de eficiência e portabilidade com o requisitos de generalidade, abstração e modularidade necessários para integrá-los aos artefatos de desenvolvimento de *software* disseminados no desenvolvimento de aplicações comerciais [Bernholdt 2004]. Assim, durante a década de 1990, surgiram ferramentas portáveis para programação em arquiteturas de memória distribuída, como o PVM (*Parallel Virtual Machine*) [Geist *et al* 1994] e MPI (*Message Passing Interface*) [MPI-Forum 1994], e memória compartilhada, como OpenMP [OpenMP 1997], além da importante evolução de bibliotecas científicas de propósito específico [Dongarra *et al* 2003], como ScaLAPACK, PETSc, NWChem, dentre outras. Até então, cada arquitetura fornecia seu conjunto próprio de ferramentas proprietárias, tornando ainda mais custoso sua implantação e manutenção.

A introdução, em computadores de uso pessoal e corporativo, de processadores super-escalares, de múltiplos núcleos e com extensões vetoriais e a disseminação de *clusters* de multiprocessadores, possivelmente de múltiplos núcleos, como servidores de alto desempenho voltados a aplicações de interesse científico, comercial e industrial, têm motivado a necessidade de disseminar as práticas de programação paralela capazes de explorar eficientemente o desempenho destas arquiteturas. Este trabalho apresenta detalhes sobre artefatos de programação paralela baseados em dois padrões, de fato e de direito, cujo uso em conjunto tem se mostrado adequado a exploração eficiente do paralelismo nos níveis distribuído e de memória compartilhada:

- **MPI**, para sincronização de processos supostos em memória distribuída, e
- **OpenMP**, para sincronização de processos em memória compartilhada.

Na Seção 2, caracterizaremos aplicações de computação de alto desempenho, de modo a identificar os requisitos que as diferenciam de aplicações convencionais. Na Seção 3, as tecnologias existentes para viabilizar esta classe de aplicações são introduzidas, enfatizando aspectos gerais sobre o conceito de programa paralelo e as técnicas de programação usadas no seu desenvolvimento. Na Seção 4, introduzimos o MPI, bi-

¹Ainda em 1965, Gordon Moore, fundador da Intel, em 1965, previu que o desempenho dos processadores duplicaria a cada dezoito meses.

² Por exemplo, o CRPC (*Center for Research on Parallel Computation*)² foi criado em 1989 nos EUA, financiado pela NSF. Este fato é considerado um marco [Dongarra *et al* 2003] na orquestração de esforços de importantes grupos acadêmicos e industriais para pesquisa e desenvolvimento de novas tecnologias de *software* para computação de alto desempenho, notadamente visando as áreas de algoritmos paralelos e programação paralela, tornando estas tecnologias mais próximas dos cientistas.

blioteca de sub-rotinas para criação e sincronização de processos em arquiteturas de memória distribuída, por meio de passagem de mensagens. Na Seção 5, abordaremos o OpenMP, padrão para criação e sincronização de *threads*, voltado às necessidades de aplicações de CAD sobre arquiteturas de memória compartilhada. Na Seção 6, discutimos as vantagens e restrições relativas ao uso conjunto do MPI e OpenMP para explorar eficientemente as múltiplas hierarquias de paralelismo presentes em *clusters* de multiprocessadores. Uma aplicação de ordenação paralela de chaves inteiras, cujo código fonte é apresentado no apêndice deste capítulo, é apresentada como exemplo do uso conjunto de MPI e OpenMP. Apresentamos nossas considerações finais na Seção 7.

9.2. Aplicações de Computação de Alto Desempenho

A existência e a evolução de uma tecnologia é primeiramente motivada pela existência de aplicações que desta demandam. Dessa forma, é útil discutir as características gerais de aplicações de área de computação de alto desempenho, de forma a identificar seus requisitos mais importantes. Tradicionalmente, estas aplicações emergem nas áreas de ciências computacionais e engenharia, de interesse tanto de instituições acadêmicas interessadas em pesquisa científica e tecnológica como de indústrias de grande porte. Como exemplo, vale enumerar várias delas:

- **Previsão climática**, envolvendo a solução computacional de modelos físicos acoplados. O uso de recursos de computação de alto desempenho por centros de previsão climática é bastante disseminado. Como ponto de partida para investigar estas aplicações, recomendamos: o site do NCAR (*National Center for Atmospheric Research*), nos EUA, em <http://www.ncar.ucar.edu/>; o site o Centro de Previsão de Tempo e Estudos Climáticos do Instituto de Pesquisa Espaciais (CPTEC-INPE), no Brasil, em <http://www.cptec.inpe.br/>; e o site do UK-Japan Climate Collaboration (UJCC), em <http://www.earthsimulator.org.uk/>;
- **Previsão e simulação dos efeitos de catástrofes**, incluindo erupções vulcânicas, terremotos, tsunamis, tornados e furacões. Exemplos de centros e projetos que desenvolvem e utilizam este tipo de tecnologia são o *Southern California Earthquake Center* (<http://www.scec.org/>), o *APEC Cooperation for Earthquake Simulation* (<http://www.quakes.uq.edu.au/ACES/>), financiado por Austrália, China, Japão, e EUA, e o NEES (*Network for Earthquake Engineering Simulation*) Grid [Spencer *et al* 2004];
- **Fisiologia dos seres vivos**, com a finalidade de investigar a dinâmica de sistemas orgânicos de seres vivos frente ao ataque de doenças, exposição à radiação, medicamentos, etc, por meio de simulação, além de buscar *insights* sobre o entendimento de seu funcionamento [Carson *et al* 2006];
- **Modelagem de reservatórios de petróleo**, com o objetivo de otimizar o processo de extração, reduzindo custos e aumentando a segurança. A indústria petrolífera é uma tradicional usuária de tecnologias de CAD. Recomendamos o site da SPE (*Society of Petroleum Engineers*) em <http://www.spe.org/>. No Brasil, destacam-se os esforços do CENPES (Centro de Pesquisa Leopoldo A. Miguez de Mello), da PETROBRÁS, associado em rede com várias instituições de pesquisa nacionais;
- **Dinâmica dos fluidos**, aplicada à avaliação e otimização aerodinâmica, notadamente de interesse das indústrias automobilística e aerospacial. Recomendamos o site <http://www.cfd-online.com/> como fonte de informações;

- **Problemas de otimização combinatória**, os quais envolvem algoritmos de natureza árdua usados em várias aplicações, exigindo grande capacidade de processamento para cobertura de instâncias realísticas;
- **Descoberta de novos fármacos**, através da modelagem molecular para composição de compostos químicos artificiais e simulação dos seus efeitos sobre organismos simulados através de experimentos *in silico* [Robson 2007];
- **Genômica e bioinformática**, aplicadas ao sequenciamento de sequências de DNA e a compreensão de seu significado fisiológico. O uso de *clusters* tem tido um papel central em projetos da área de genômica. No Brasil, a bioinformática e a genômica são áreas de interesse interdisciplinar do Laboratório Nacional de Computação Científica (LNCC), participante de várias de redes de pesquisa patrocinadas por órgãos de fomento brasileiros, como CNPq e FAPESP;
- **Engenharia financeira, econofísica e finanças quantitativas** [Grant 2007], as quais envolvem a implementação de complexos modelos matemáticos, muitos dos quais oriundos de analogia com teorias físicas e envolvendo modelos estatísticos, para solução de problemas típicos da área financeira, como, por exemplo, o movimento de preços no mercado, comumente associado ao modelo *Black-Scholes* [Black e Scholes 1973], e a previsão de colapso de sistemas de segurança;
- **Mineração de dados**, visando à análise de extensas bases de dados não estruturadas para extrair informações relevantes em domínios específicos. Técnicas de inteligência artificial são em geral usadas, com alta demanda computacional.

A lista cima, embora não exaustiva, ajuda-nos a identificar os requisitos comuns entre estas aplicações. Em primeiro lugar, aplicações relacionadas à simulação de fenômenos físicos geralmente recorrem a sofisticados modelos matemáticos baseados em equações diferenciais parciais cuja solução analítica não é conhecida, as quais são numericamente resolvidas por métodos de discretização como *diferenças finitas*, *elementos finitos* e *volumes finitos*. Estas resultam em sistemas de equações lineares cuja solução pode ser obtida aplicando-se técnicas que exigem alta velocidade para realizar cálculos em ponto-flutuante, razão pela qual uma das principais métricas usada para avaliar o desempenho de arquiteturas de CAD é a *quantidade de operações de ponto flutuante realizadas por segundo*, ou FLOPs.

Merce ainda destaque à necessidade de alto desempenho para operações de acesso a dados, devido à crescente demanda pela utilização de grandes bases de dados, especialmente em simulações, provenientes de medições reais e/ou outras simulações acopladas. Tais bases de dados têm crescido exponencialmente nos últimos anos, em taxas superiores à Lei de Moore, a qual tem governado a evolução da capacidade do *hardware*. Este fato tem sido observado em muitas aplicações oriundas, sobretudo, da genômica, física de alta energia, astronomia, previsão climática e sismologia, dificultando substancialmente a capacidade de análise de dados por meios automáticos, tendo em vista que a maioria dos algoritmos mais eficientes envolvidos são polinomiais de ordem superior a dois, além de tornar inviável que o cientista ou engenheiro copie e mantenha bases de dados em suas máquinas locais, exigindo ferramentas que ofereçam novos meios destes interagirem diretamente com os dados em seus locais de armazenamento. Ainda, tendo em vista que grandes bases de dados científicos são armazenadas em meios não-voláteis, outro fator preocupante é a taxa do crescimento da capacidade

de armazenamento em meios físicos não-voláteis nos últimos dez anos, uma ordem de magnitude mais rápido que a largura de banda de entrada e saída destes mecanismos (100X contra 10X). A semântica dos dados também tem emergido como um fator relevante a ser considerado. Para isso, são necessárias tecnologias que permitam a visualização e interpretação de grandes volumes de dados por cientistas, de forma que estes possam extrair as informações efetivamente necessárias para uma simulação ou mesmo para geração e verificação de hipóteses científicas. Técnicas de mineração de dados, notadamente associadas a técnicas de inteligência computacional, são comumente usadas, exigindo grande esforço computacional.

Muitas aplicações de CAD exigem ainda a integração de vários modelos acoplados e heterogêneos. Aplicações de simulação multi-física são casos típicos, dentre as quais as aplicações de simulação climática são os representantes mais populares. Nestas, cada modelo envolvido aborda um dentre os sistemas atmosfera, oceano, terra, e gelo, os quais são orquestrados por meio de um modelo acoplador, responsável por computar as contribuições de cada modelo no modelo global de clima. De fato, os diferentes modelos envolvidos podem estar codificados em linguagens de programação diferentes, motivando dificuldades de mapeamentos entre representações de estruturas de dados entre estas. Além disso, no caso de modelos implementados com técnicas de discretização, diferentes modelos podem assumir diferentes formatos de elementos e/ou resolução de malha, sendo necessário o mapeamento do espaço entre um domínio e outro, o qual pode se tornar uma operação bastante custosa. Com a introdução de computação distribuída na *internet*, especialmente com a implantação de grades computacionais, o compartilhamento *on-line* de modelos entre instituições e laboratórios de pesquisa tem sido explorado, especialmente através de portais científicos ou mesmo de serviços de computação espalhados na rede usando grades computacionais ou tecnologias de acesso a serviços distribuídos, como *web services*.

Um conceito comum em aplicações de alto desempenho é a “corretude por reputação”, estágio alcançado por um *software* científico o qual tem sido executado com sucesso durante um longo tempo, em múltiplos contextos arquiteturais. Há vários exemplos de código legado, em geral encapsulado em bibliotecas científicas nos mais diversos domínios, usados há mais trinta anos com sucesso em aplicações de computação de alto desempenho. Portanto, é natural para cientistas e engenheiros a divulgação de *software* para validação e testes, visando à futura promoção deste ao *status* de *software* finalizado, presumivelmente confiável e robusto, pela própria comunidade científica. Portanto, mecanismos de controle de versão de código são necessários.

Os requisitos enunciados nos dois parágrafos anteriores têm motivado, nos últimos anos, a adoção do paradigma de desenvolvimento de *software* baseado em componentes para aplicações de CAD [Steen 2006]. Com esta finalidade, vários modelos e infra-estruturas de componentes orientados às especificidades desta classe de aplicações têm sido propostos [Armstrong 1999, Baude 2003, Carvalho-Junior 2007]. Atualmente, a integração do paralelismo a modelos de componentes tem sido um dos problemas especialmente abordados por trabalhos de pesquisa nesta área, tendo em vista que os modelos de componentes tradicionais não foram desenvolvidos tendo em vista o requisito de paralelismo. De fato, a dificuldade reside na transversalidade inerente do paralelismo em relação à arquitetura de componentes de uma aplicação [Carvalho-Junior 2005], que torna a mera extensão dos modelos de componentes já existentes incapaz de conciliar a capacidade de expressar padrões de paralelismo não-triviais [Carvalho-Junior 2007].

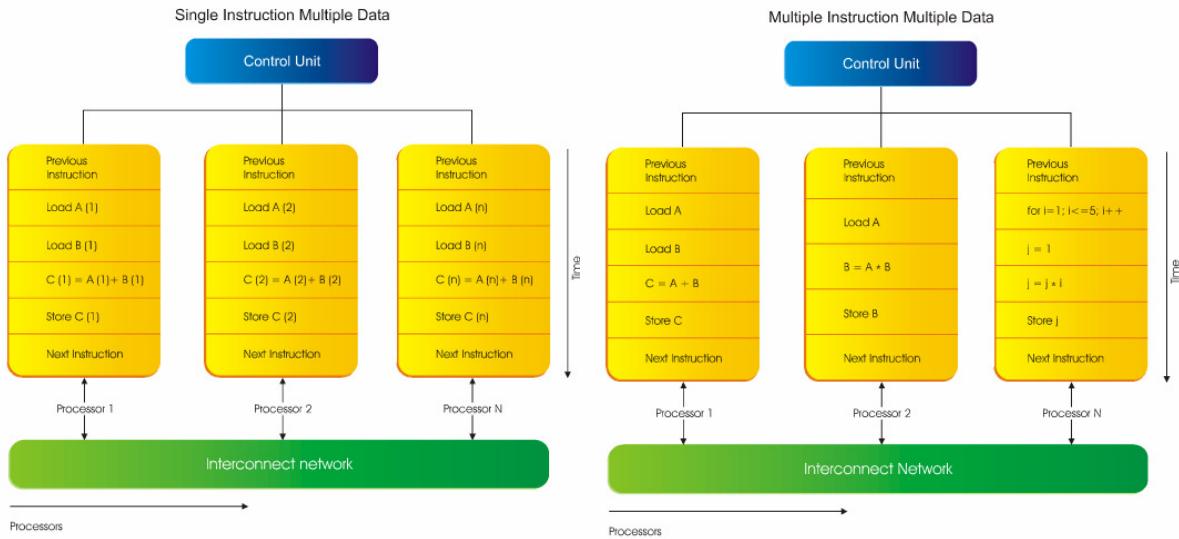


Figura 2. Arquitetura SIMD

Figura 2. Arquitetura MIMD

9.3. Tecnologias de Suporte à Computação de Alto Desempenho

Aplicações de CAD possuem requisitos particulares que demandam por tecnologias específicas, as quais provêm de desenvolvimentos em três áreas distintas: algoritmos abstratos, arquiteturas de computadores, e ambientes de *software* [Dongarra *et al* 2003]. A primeira diz respeito às investigações de algoritmos eficientes, do ponto de vista de complexidade abstrata, para solução dos diversos problemas que ocorrem em aplicações em ciências computacionais e engenharia. Por exemplo, algoritmos iterativos para solução de sistemas de equações lineares têm sido desenvolvidos e aprimorados ao longo dos anos, como *Gauss-Seidel*, *Red-Black*, *Successive Over Relaxation*, métodos *multigrid*, fatoração LU, etc. Algoritmos para geração de malhas, também necessários à implementação do método de elementos finitos, são também exemplos importantes. Em muitas situações, a escalabilidade de um algoritmo, ou o grau de paralelismo, é prioritária em relação a sua complexidade.

9.3.1. Arquiteturas de Computadores de Alto Desempenho

Arquiteturas de computadores para CAD remontam ao ano de 1972 quando Seymour Cray fundou a *Cray Research*, responsável pela implantação do primeiro supercomputador no Los Alamos National Laboratory (LANL), EUA, em 1976, ao custo de 8,8 milhões de dólares. Esta máquina era capaz de executar 160 megaflops (milhões de operações de ponto flutuante por segundo), com 8MB de memória principal. Tal desempenho superava em ordens de magnitude o desempenho do mais rápido dos computadores existentes do mercado. Arquiteturas vetoriais dominaram a supercomputação nas décadas de 1970 e 1980. Possuíam arquitetura SIMD (*Single Instruction Multiple Data*), sendo capazes de executar uma operação simultaneamente sobre cada elemento de um conjunto de dados (Figura 2). Demandam por linguagens específicas e compiladores “paralelizantes”, motivando intensa a investigação sobre paralelização automática de programas, em especial a vetorização de *loops*. Entretanto, arquiteturas vetoriais mostram-se pobemente escaláveis, apesar de seu alto custo.

Arquiteturas de memória distribuída emergiram a partir de meados da década de 80. Estas são de arquitetura MIMD, onde a cada processador corresponde uma linha de instrução independente (Figura 2). Tornaram-se viáveis devido ao surgimento de interfaces de comunicação de alto desempenho, em especial proprietárias, levando ao surgimento das arquiteturas ditas *massivamente paralelas*, as chamadas MPP (*Massive Parallel Processors*). Exemplos clássicos destas arquiteturas são: Intel Paragon, Cray T3D e T3E, Thinking Machine CM5, IBM SP2, ASCI Red e Sun HPC. Possuindo maior escalabilidade, ainda suplantavam máquinas vetoriais em desempenho bruto. Porém, exigiam modelos explícitos de programação paralela, em geral baseados em troca de mensagens, a fim de aproveitar ao máximo o seu desempenho.

Durante este período, difundiram-se no mercado de supercomputação as arquiteturas SMP (*Shared Memory Processors*), admitindo um conjunto de processadores de tecnologia vetorial compartilhando memória. Estas arquiteturas, ilustradas na Figura 4, também são de arquitetura MIMD. SMPs tornaram-se projeto padrão de supercomputadores Cray, como os da família Cray X-MP, e de outros fabricantes. Motivaram ainda arquiteturas SMP com processadores escalares, bastante utilizadas em laboratórios de pesquisa devido ao custo menor. A escalabilidade de SMPs é limitada pela capacidade do barramento de suportar a contenção de memória causada por muitos processadores.

No início da década de 90, iniciaram-se tentativas de viabilizar o paralelismo em redes de estações de trabalho (NOWs) [Anderson *et al* 1994]. De fato, para baratear os custos na fabricação de MPPs, cuja escala de mercado é restrita, os processadores usados em estações de trabalho e MPPs eram os mesmos. Devido ao tempo necessário para que fabricantes de MPPs desenvolvessem interconexões de comunicação otimizadas para uma nova classe de processadores, a MPP baseada nesta demorava certo tempo (*lag time*) para ser lançada no mercado. Na época

do lançamento, uma nova geração de processadores já estava sendo lançada para as estações de trabalho do fabricante, causando uma defasagem na tecnologia de processadores usados nas MPPs capaz de compensar a perda de eficiência causada pelo uso de redes de comunicação convencionais em NOWs, permitindo a estas abordarem nichos de aplicações antes exclusivas de MPPs.

A partir da primeira metade da década de 90, os investimentos possibilitados devido ao crescimento vertiginoso do mercado de computadores

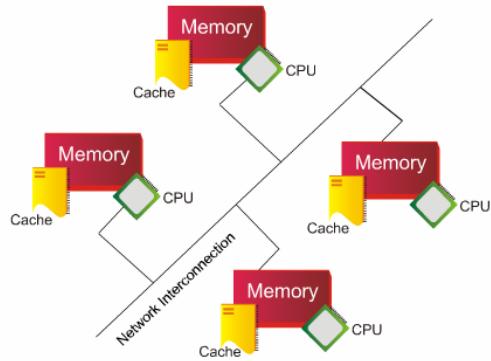


Figura 3. Massive Parallel Processors

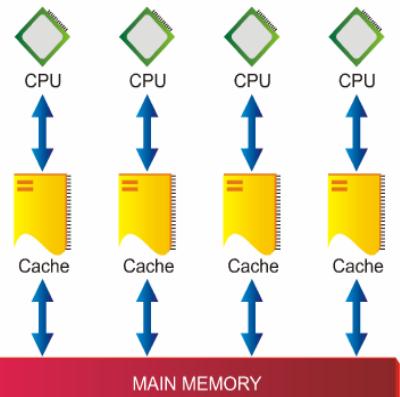


Figura 4. Multiprocessadores

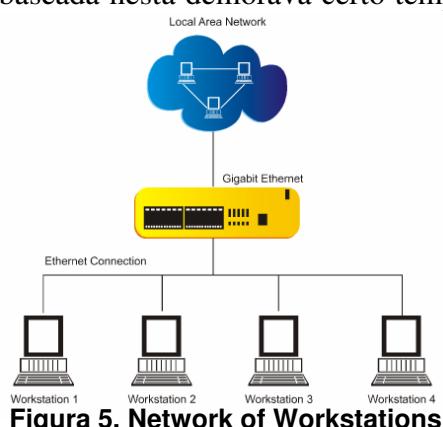


Figura 5. Network of Workstations

pessoais (PCs) levaram os processadores desenvolvidos para esta classe de computadores a um desempenho comparável às estações de trabalho, porém a um custo muito inferior, conduzindo a quase extinção destas. Então, logo surgiram as primeiras tentativas de explorar redes de PCs, do padrão *Ethernet*, para aplicação em CAD. Destaca-se como marco o projeto *Beowulf* [Becker *et al* 1995], conduzido por Donald Becker, em 1994, na NASA, o qual possuindo escassos recursos para aquisição de máquinas paralelas às quais demandava, resolveu construí-las a partir de *hardware* de prateleira, reusando equipamentos fora de uso. Estas arquiteturas ficaram conhecidas como *clusters* e significaram a proliferação das tecnologias de CAD em vários laboratórios de pesquisa espalhados pelo mundo, para os quais a tecnologia vigente de supercomputação era muito custosa. *Clusters* são fenômenos fundamentalmente influenciados pelo *software* que foi desenvolvido para o seu suporte. De fato, Becker e sua equipe desenvolveram interfaces de comunicação capazes de explorar ao máximo o desempenho de sua arquitetura. Para isso, foi fundamental o uso de código livre disponibilizado pelas emergentes distribuições do sistema operacional *Linux*. Segundo o site “Top 500” (Figura 6), onde são enumeradas as quinhentas computadoras de maior desempenho atualmente implantados segundo *benchmarks* padrões, de 1998, quando os *clusters* começaram a aparecer nesta lista, até meados do ano de 2003, a participação destes cresceu exponencialmente, atingindo a maior parte das posições da lista (atualmente 72,2% destas).

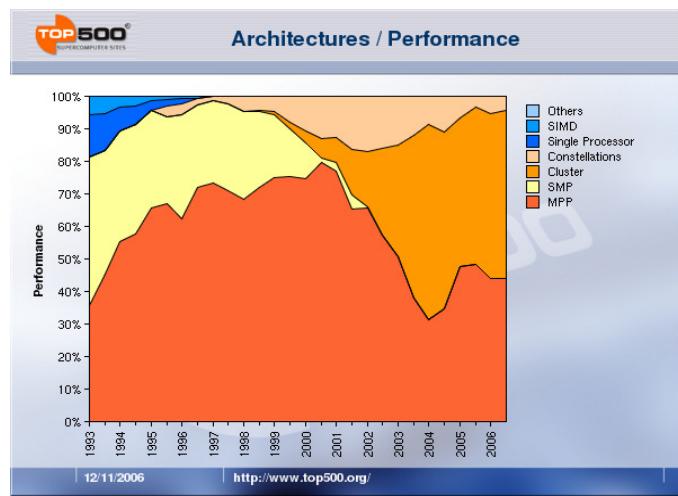


Figura 6. Participação das principais arquiteturas de supercomputação na lista Top 500 (<http://www.top500.org/>)

No final da década de 1990, o aumento da escala e do desempenho da infraestrutura de comunicação da *internet* motivou a idéia de sua extensão a uma infraestrutura de computação, a *grade computacional*, capaz de virtualizar os recursos computacionais disponíveis na internet para torná-las disponíveis a aplicações de CAD [Foster e Kesselman 2004]. O termo *grade* deve-se à analogia com a infraestrutura de distribuição de energia elétrica, a qual apresenta os recursos de maneira transparente aos usuários. Desde então, a pesquisa para viabilização da tecnologia de grades disseminou-se, motivando vários projetos. Espera-se mover-se das grades de propósito específico para as grades de propósito geral, onde diversas tecnologias possam coexistir de maneira transparente ao usuário. Atualmente, a alta latência da *internet* ainda não torna viável a sincronização de processos fortemente acoplados, característicos de *clusters* e, especialmente, MPPs. O paralelismo em grades tem sido explorado por meio de padrões de paralelismo que supõem a independência total entre as tarefas, como *bag-of-tasks*. Entretanto, espera-se que esta barreira tecnológica seja vencida nas próximas décadas, com a evolução da infraestrutura de comunicação sobre as quais se assentam as grades.

Nos últimos anos, o projeto de novos processadores para servidores e computadores de uso doméstico tem abordado tecnologias de suporte ao paralelismo em nível de

threads, especialmente com a introdução de múltiplos núcleos de processamento em um mesmo processador, o que resulta em melhor desempenho e menor custo energético. Estando fortemente acoplados em um mesmo *chip*, os núcleos de processamento têm potencial de paralelismo superior aos tradicionais multiprocessadores, compartilhando inclusive níveis de *cache* dependendo da arquitetura do processador. No estágio atual, encontram-se disponíveis processadores com até quatro núcleos. Aplicações atuais podem especialmente explorar o desempenho deste tipo de processadores, tendo em vista que o uso de *threads* tem sido particularmente comum no desenvolvimento de aplicações e que usuários normais de sistemas operacionais modernos tendem a utilizar diversas aplicações simultaneamente (editor de texto, mensagens instantâneas, antivírus, *players*, etc.). Neste contexto, técnicas gerais de programação concorrente são suficientes. Entretanto, com a tendência anunciada pela indústria de aumentar-se o número de núcleos em um mesmo processador nos próximos anos, surgirá a demanda pela exploração de técnicas escaláveis de programação paralela em aplicações convencionais, onde o objetivo tornar-se-á a divisão do processamento de uma aplicação em unidades menores de forma a reduzir o tempo de processamento pela execução destas partes em núcleos separados. A demanda por programação paralela tende a ser ampliada pelo crescente uso de *clusters* de multiprocessadores (de múltiplos núcleos) como servidores de alto desempenho em aplicações comerciais.

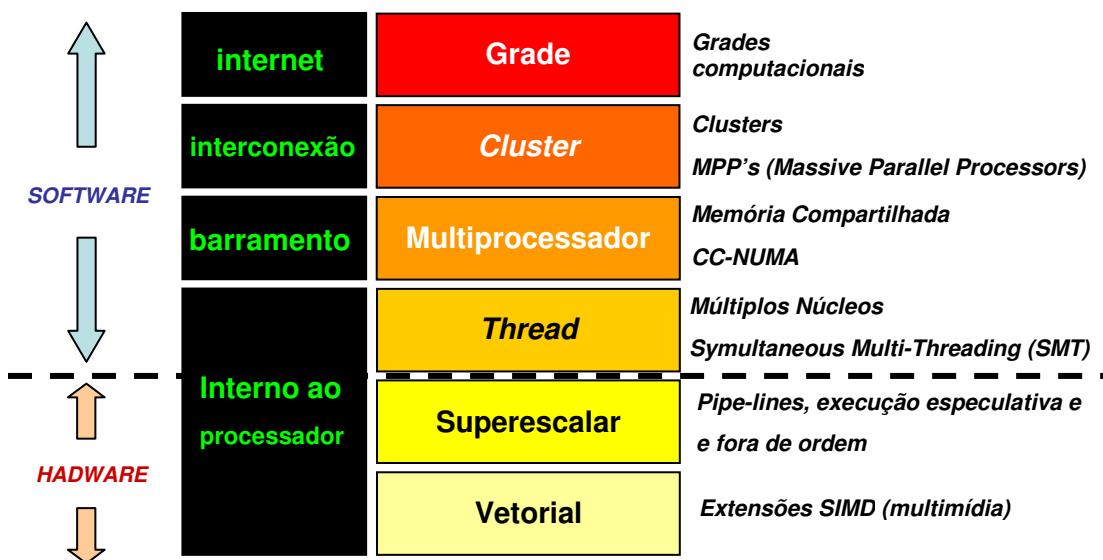


Figura 7. Hierarquias de Paralelismo

A Figura 7 ilustra os múltiplos níveis de paralelismo potencialmente presentes em arquiteturas contemporâneas de computação. Sobre o controle do programador estão aqueles níveis situados a partir do nível de *threads*. Com exceção dos níveis de *threads* e multiprocessamento, os paradigmas de programação adequados a exploração eficiente do paralelismo em cada um destes níveis são diferentes. Em grades computacionais, são necessários *middlewares* capazes de virtualizar um conjunto de recursos computacionais potencialmente infinitos. Em *clusters* e MPPs predominam as bibliotecas de passagem de mensagens, como MPI, capazes de expressar padrões de paralelismo de forte acoplamento entre os processos. A eficiência de interfaces de passagem de mensagens sobre estas arquiteturas é justificada devido a serem estas abstrações das interfaces de redes de comunicação e interconexões de mais baixo nível. Porém, são de difícil imple-

mentação eficiente em arquiteturas de multiprocessamento ou de múltiplos núcleos quando comparadas a alternativas como OpenMP, orientado a paralelização de laços nesta classe de arquiteturas. Compiladores OpenMP são capazes ainda de explorar extensões vetoriais. Bibliotecas de programação concorrente de propósito geral, como *PThreads*, ou mesmo funcionalidades presentes em linguagens de alto nível como Java e C#, são também comuns para explorar o paralelismo nestas arquiteturas. Porém, como OpenMP foi desenvolvida especialmente para programação paralela, este consegue explorar melhor o desempenho de aplicações de CAD. A exploração eficiente do potencial de desempenho de arquiteturas que envolvem múltiplos níveis de paralelismo envolve a utilização de vários artefatos de programação, exigindo conhecimento do programador a respeito das técnicas gerais de programação paralela.

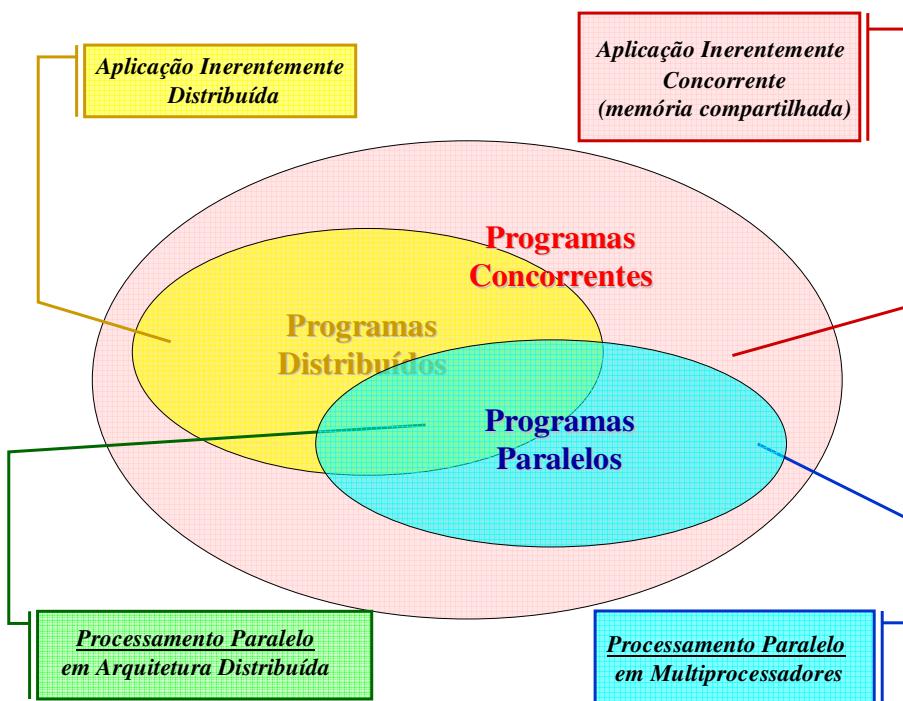


Figura 8. Classes de Programas Concorrentes

9.3.2. Programação Paralela: Conceitos, Técnicas e Métricas de Desempenho

Esta seção discute aspectos gerais relacionados à construção de programas paralelos e a avaliação de seu desempenho.

Concorrência e Paralelismo

Programas concorrentes são quaisquer programas compostos por várias linhas de instrução sobre o controle explícito ou semi-explícito do programador. Linhas de instrução concorrentes podem executar em processadores distintos ou no mesmo processador. Neste último caso, técnicas de multiprogramação são necessárias para intercalar e escalar instruções das múltiplas linhas no mesmo processador. É importante distinguir duas classes de programas concorrentes, distinguindo aqueles cuja concorrência é inherente à aplicação, como sistemas operacionais e outros sistemas reativos, daqueles que podem ser implementados sequencialmente, mas são forçados à execução concorrente com o intuito de redução do tempo de processamento, como

multiplicação de matrizes. A estes, denominamos *programas paralelos*, objeto deste trabalho, para os quais a execução em múltiplas unidades de processamento é essencial. Entre programas concorrentes, é ainda necessário distinguir entre aqueles cujas linhas de instruções acessam variáveis em um mesmo espaço de endereçamento (*memória compartilhada*) aqueles cujas linhas de instrução acessam variáveis em espaços de endereçamento disjuntos (*memória distribuída*). A estes, denominamos *programas distribuídos*. As classes de programas concorrentes enunciadas neste parágrafo estão ilustradas na Figura 8. Portanto, dentre os *programas paralelos* temos aqueles onde os processos compartilham memória e aqueles em que os processos são ditos distribuídos. Em teoria de programação concorrente, as técnicas de sincronização de processos nestes casos demandam abordagens distintas, com diferentes características de desempenho.

Paralelização de Programas

Independente do tipo de arquitetura alvo (memória compartilhada ou distribuída), a construção de programas paralelos exige o conhecimento sobre técnicas de decomposição da carga de computação de uma aplicação entre um conjunto de processos ou *threads*. Estas podem ser classificadas em *técnicas de decomposição funcional (paralelismo de controle)*, onde funcionalidades do programa são executadas em linhas de instrução concorrentes (processos ou *threads*), e *técnicas de decomposição de domínio (paralelismo de dados)*, onde os dados a serem processados são divididos em subconjuntos, cada qual potencialmente habilitado a ser processado por uma linha de instrução independente das demais. A sincronização entre estas pode ser ou não necessária durante o processamento. O paralelismo de dados é dito *escalável*, tendo em vista que, em muitas aplicações, a acurácia da solução é proporcional ao tamanho da instância coberta do problema. Assim, provendo-se um conjunto maior de processadores a um usuário, este pode optar por dividir os dados em subconjuntos ainda menores para diminuir o tempo de alcance da solução ou aumentar a instância do problema e manter a quantidade de trabalho executada por cada linha de instrução. Por outro lado, o paralelismo de controle é limitado pelas funcionalidades presentes na aplicação, as quais são fixas. O uso em conjunto de paralelismo de controle e de dados é desejável sempre que possível.

Usualmente distinguem-se quatro fases no processo de construção de um programa paralelo: *particionamento*, *sincronização*, *agregação*, e *mapeamento*. No particionamento, empregam-se as técnicas de decomposição funcional e de domínio com vistas a identificar os grãos mínimos de paralelismo presentes na aplicação. Na fase de sincronização, é identificado como as unidades mínimas de paralelismo devem interagir corretamente para produzir a solução do problema. Nas demais fases, surgem preocupações com a arquitetura alvo, onde o programa paralelo deverá executar. Na fase de aglomeração, unidades de paralelismo são agrupadas com a finalidade de obter um tamanho desejável de grão de paralelismo, de forma que o tempo de computação sobreponha-se ao tempo de comunicação. Este é um requisito essencial para se obter o melhor nível de desempenho e escalabilidade na arquitetura alvo. Em geral, o tamanho do grão é menor à medida que a velocidade da interconexão entre as unidades de processamento, ou grau de *acoplamento*, é maior. Na Figura 7, observa-se que o grau de acoplamento varia crescentemente das arquiteturas de processadores de múltiplos núcleos em direção às grades. Na fase de mapeamento, as unidades de paralelismo agregadas, as quais formam processos ou *threads*, são mapeadas aos recursos computacionais da arquitetura, ou nós de computação. O mapeamento pode ser feito estaticamente, em tempo de programação ou compilação, ou dinamicamente, permitindo a migração de

processos ou *threads* entre processadores de acordo com a carga de trabalho destes. O mapeamento pode ainda ser classificado como *implícito*, quando realizado por ferramentas automáticas ou pelo compilador, ou *explícito* quando sob total controle do programador. As fases de agregação e mapeamento são em geral feitas em conjunto, tendo em vista que a agregação em geral deve ser feita observando-se a capacidade computacional dos nós de processamentos da arquitetura, de forma a aproximar-se ao máximo do balanceamento de carga ótimo.

Tendo em vista a intratabilidade computacional dos problemas gerais relacionados à paralelização automática de programas, em todas as suas fases anteriormente citadas, o uso de métodos automáticos de paralelização é restrito a casos especiais, em geral assumindo-se suposições nem sempre realísticas sobre aplicações e arquiteturas. Em programas escritos em linguagens imperativas, o problema de particionamento é o mais difícil de ser tratado, tendo em vista a necessidade de analisar a dependência de dados entre as variáveis acessadas por trechos de código. Por outro lado, em programas escritos com linguagens funcionais puras, qualquer expressão pode ser avaliada em paralelo, o que sugeriu nos primórdios da programação funcional que estas linguagens dominariam o cenário da computação paralela [Lins 1996]. Porém, o problema de aglomeração, ou seja, como agrupar um conjunto de expressões a serem avaliadas em um mesmo processo, configurou-se como o gargalo para viabilizar a exploração eficiente e transparente do paralelismo em linguagens funcionais. Por estes motivos, as técnicas mais disseminadas de paralelização são aquelas ditas explícitas, onde o programador tem total controle sobre todas as etapas da paralelização do programa, exigindo destes, além do conhecimento sobre técnicas de paralelização, o conhecimento profundo sobre as características das aplicações e das arquiteturas sobre os quais estas executarão.

Métricas de Desempenho

Uma vez que o desempenho é o requisito dominante no projeto de programas paralelos, métricas específicas são necessárias para aferir o grau de qualidade de uma implementação em relação a uma determinada arquitetura paralela. Em computação de alto desempenho, as principais são o *speedup* e a *eficiência*.

O *speedup* é definido como a relação entre o tempo de execução da versão sequencial de um programa e de sua versão paralela, em N processadores:

$$S(N) = \frac{T_{seq}}{T_N}, \text{ onde: } \begin{cases} T_{seq} \text{ é o tempo de execução da versão sequencial} \\ T_N \text{ é o tempo de execução da versão em N processadores} \end{cases}$$

Costuma-se analisar a escalabilidade de um programa paralelo analisando-se sua curva de *speedup*, obtida ao construir-se o gráfico $N \times S$. O *speedup* de um programa paralelo em N processadores é dito ideal quando seu valor é N. Em geral, o valor do speedup assume um valor no intervalo $[0, N]$, tendo em vista que o tempo de execução da versão paralela de um programa contabiliza o tempo de sincronização e comunicação entre os processos, além do tempo de computação útil. Entretanto, não é raro se observar o efeito de *superspeedup*, quando o valor do *speedup* assume um valor maior que N. Vários fatores podem motivar o *superspeedup*. Em geral, isso ocorre no paralelismo de dados sobre arquiteturas com múltiplas hierarquias de memória devido a maior localidade dos dados na versão paralela. Dessa forma, o tempo menor desperdiçado em operações de movimentação de dados na memória compensa o tempo desperdiçado com comunicação e sincronização. Muitas vezes, há controvérsia sobre a forma como é ob-

tida a versão seqüencial do programa. É comum executar-se a versão paralela parametrizada em N processadores supondo-se $N=1$ e chamá-la de versão seqüencial, muitas vezes com o objetivo de fraudar resultados que atestem a escalabilidade de um programa paralelo. Por este motivo, distinguem-se o speedup absoluto, como definido anteriormente, e o speedup relativo, como definido abaixo:

$$S_A(N) = \frac{T_1}{T_N}$$

Em 1967, *Gene Amdahl* propôs que, em programas paralelos, o *speedup* estaria limitado pela existência de porções de código não-paralelizáveis, ou inherentemente seqüenciais, na versão seqüencial do programa [Amdahl 1967]. Assim, seja P um programa paralelo. Podemos definir s e p como as frações inherentemente seqüencial e paralelizável de P , respectivamente. Obviamente $s + p = 1$. Segundo Amdahl, o *speedup* máximo nessas circunstâncias seria definido pela seguinte fórmula:

$$S^{MAX}(N) = \frac{1}{s + \frac{p}{N}}$$

Note que quando o número de processadores tende ao infinito, o *speedup* máximo tende ao valor $1/s$, considerado o limite teórico do *speedup*, segundo Amhdal.

$$\lim_{n \rightarrow \infty} S^{MAX}(N) = \frac{1}{s}$$

No final da década de 80, Gustafson propôs reformular a Lei de Amhdal, baseado na idéia de que a proporção entre as porções seqüencial e paralelizável de um programa varia de acordo com o tamanho do problema a ser resolvido [Gustafson 1988]. Segundo Gustafson, o programador não fixa o tamanho do problema para execução em quantidades diferentes de processadores, o que seria uma suposição válida em um experimento. Na prática, o tamanho do problema aumenta de acordo com os recursos disponíveis no sistema (número de processadores), visando obter, por exemplo, resultados mais acurados em uma simulação. Segundo Gustafson, uma formulação mais realista para o *speedup*, assumiria fixo o tempo de execução. Surgiram então duas definições alternativas de *speedup*: *scaled speedup* e *fixed-time speedup*, os quais não impõem limites teóricos ao *speedup*.

A *eficiência* é definida como a parcela de tempo desperdiçada por um programa paralelo realizando computação útil, podendo ser aproximada pela seguinte fórmula:

$$E(N) = \frac{S(N)}{N}$$

Costuma-se ainda usar a medida *vazão (throughput)* em casos onde é possível determinar o número de operações de computação útil, ou dominantes, realizadas pelo programa em certa unidade de tempo. Por exemplo, em aplicações numéricas, é comum utilizar-se a unidade *Flops*, ou número de operações de ponto flutuante por segundo. Portanto, a caracterização do tipo de operação dominante a qual se deseja mensurar é dependente da aplicação.

9.4. O MPI: Explorando o Paralelismo em Memória Distribuída

O MPI (*Message Passing Interface*) foi desenvolvido por um comitê científico constituído no ano de 1993 com a finalidade de oferecer uma interface padrão, eficiente e portável para programação distribuída baseada em passagem de mensagens [MPI Fórum 1994]. A formação deste comitê foi motivada pelas dificuldades acarretadas pela existência de diferentes interfaces de passagem de mensagens disponíveis no mercado, em geral cada qual associada a uma plataforma específica, à disseminação da prática de programação paralela. O MPI é um dos principais produtos dos esforços empreendidos pela comunidade científica na década de 90 para melhorar os artefatos de programação para o desenvolvimento de programas paralelos, sobretudo em arquiteturas distribuídas. Desde que foi proposto, o MPI tornou-se um padrão de fato, sendo hoje virtualmente suportado por toda plataforma de computação distribuída de alto desempenho.

O MPI possui atualmente dois padrões válidos, conhecidos como MPI-1 e MPI-2, respectivamente descritos pelos documentos de versão 1.2 e 2.1 disponíveis no site do MPI Fórum (<http://www mpi-forum.org>). O padrão MPI-1 foi inicialmente proposto no ano de 1995, enquanto o MPI-2 surgiu no ano de 2003 adicionando várias funcionalidades ao padrão MPI-1. *Bindings* oficiais são especificados para as linguagens C, C++ e Fortran, por serem estas as linguagens de uso mais comum em ciências computacionais e engenharia, onde as técnicas de CAD são disseminadas. Há várias implementações do padrão MPI, algumas destas proprietárias e outras de código aberto e/ou disponíveis publicamente. As mais conhecidas são o LAM-MPI, MPICH, e Open MPI. Há ainda as implementações proprietárias, como o MSMPI, da Microsoft, desenvolvido em conjunto com o ANL (*Argonne National Laboratory*) visando o *Windows Compute Cluster Server (WCCS)*; o HP-MPI, desenvolvido para servidores e estações de trabalho da HP e disponível para Linux, UNIX e WCCS; o Sun MPI, incluído no pacote *Sun HPC Cluster Tools* notadamente visando o sistema Solaris; o IBM MPI, implementado como uma interface para o MPE (*Message Passing Environment*) da IBM, dentre outros. Há ainda implementações de MPI para grades computacionais, como MPICH-G2, muito embora não seja ainda um paradigma adequado para desenvolver aplicações paralelas neste tipo de arquitetura. Todas as implementações de MPI conformam com o padrão MPI-1, sendo que somente algumas delas são compatíveis com o padrão MPI-2. É preciso consultar a documentação da implementação específica para ter certeza a qual padrão esta adere.

Nas seções que se seguem, será fornecida uma visão geral do conjunto de funcionalidades presentes nos padrões MPI-1 e adicionadas ao MPI-2, procurando fornecer, em curto espaço, ao leitor uma noção da amplitude real das potencialidades desta ferramenta, dificilmente abordada nos muitos tutoriais de MPI disponíveis publicamente. Portanto, os detalhes completos sobre cada uma das sub-rotinas apresentadas devem ser obtidos nos respectivos documentos que especificam os padrões MPI-1 e MPI-2. Exemplos práticos podem também ser facilmente obtidos nos próprios documentos e através do vasto conjunto de tutoriais e cursos disponíveis publicamente em *sites* da internet, muitos destes acessíveis através do site do Fórum MPI.

Um programa MPI é composto por um conjunto de processos que se comunicam por meio de sub-rotinas de trocas de mensagens de diversos tipos suportados (ponto-a-ponto, coletiva, ou unilateral). A maioria das implementações de MPI suporta o estilo

SPMD (*Single Program, Multiple Data*), onde um mesmo código fonte é disparado para cada processo, sendo possível diferenciar o comportamento de cada processo por meio de sua identificação, representada por um número inteiro chamado *rank*. Assim, se há N processos, estes são identificados com *ranks* que variam de 0 a N-1. Em geral, a implementação MPI permite o controle sobre o processador no qual deve ser disparado o processo de *rank* i , $0 \leq i \leq N$. Algumas implementações MPI permitem também o disparo de vários programas, cada um associado a cada processo MPI, estilo que é conhecido como MPMD (*Multiple Program, Multiple Data*).

Tabela 1. MPI Básico

MPI_INIT	Inicializa o ambiente MPI para o processo.
MPI_COMMRANK	Retorna a identificação (<i>rank</i>) de um processo relativo ao grupo envolvido em um comunicador especificado, representado por um número inteiro. Os <i>ranks</i> variam de 0 a N-1, onde N é o número total de processos do grupo.
MPI_COMMSIZE	Retorna o número total de processos em um grupo envolvido em um comunicador especificado.
MPI_SEND	Envia uma mensagem, representada por um conjunto de elementos de um tipo especificado dispostos em um <i>buffer</i> contíguo, para um outro processo, identificado pelo seu <i>rank</i> .
MPI_RECV	Recebe uma mensagem de um processo identificado pelo seu <i>rank</i> , copiando-a em um <i>buffer</i> local.

9.4.1. O MPI Básico

A Tabela 1 apresenta as rotinas que constituem o subconjunto básico do MPI, com o qual é virtualmente possível a construção de qualquer programa paralelo. Isso é possível por este incluir primitivas essenciais e necessárias para o envio de mensagens entre dois processos e para a recuperação, por parte de um processo arbitrário, sobre informações a cerca da rede de processos, supostamente de topologia totalmente conectada, como a quantidade de processos envolvidos e a identificação do processo (*rank*). A execução de um par correspondente de MPI_SEND, pelo processo emissor, e um MPI_RECV, pelo processo receptor, corresponde a cópia de um *buffer* de dados mantido pelo emissor em um *buffer* correspondente no receptor. Acompanhando os dados transmitidos, uma mensagem é ainda caracterizada pelo seu *envelope*, composto por: (1) *rank* dos processos fonte e destino; (2) *tag*, um número inteiro que pode ser usado para caracterizar o assunto da mensagem; e (3) *comunicador*, representando um escopo de comunicação. O escopo de comunicação *default*, usado no MPI básico, é o global, envolvendo todos os processos, identificado por MPI_COMM_WORLD. O envelope de mensagens de pares MPI_SEND/MPI_RECV correspondentes devem ser iguais.

A implementação de programas paralelos que incluem padrões complexos de interação pode ser muito complicada com uso somente de sub-rotinas do MPI básico. Por este motivo, o MPI inclui sub-rotinas de mais alto nível para lidar com padrões recurrentes de paralelismo, descritos nas seções que se seguem.

Tabela 2. Comunicação Ponto-a-Ponto

Direção	Modo	Blocante	Não-Blocante
Envio	Básico	MPI_SEND	MPI_ISEND
	Síncrono	MPI_SSEND	MPI_ISSEND
	Bufferizado	MPI_BSEND	MPI_IBSEND
	Pronto	MPI_RSEND	MPI_IRSEND
Recebimento	Pronto	MPI_RECV	MPI_IRecv

9.4.2. Comunicação Ponto-a-Ponto Extendida (MPI-1)

O MPI inclui sub-rotinas para diversos modos de comunicação comuns em programação paralela sobre arquiteturas distribuídas, apresentadas na Tabela 2, divididas em dois subconjuntos: *blocante* e *não-blocante*. Sub-rotinas *blocantes* completam-se

somente após a efetivação completa da operação, enquanto aquelas *não-blocantes* podem ser iniciadas porém completadas sob a intervenção direta do programador, permitindo: (1) a sobreposição de comunicação e computação; e (2) a efetivação de operações sem uma ordem pré-determinada. Para isso, a execução de uma operação não-blocante retorna um *manipulador de requisição* (valor do tipo MPI_REQUEST). A Tabela 3 apresenta o conjunto de rotinas para manipular requisições, permitindo testar se estas se completaram ou aguardar até que se completem.

Tabela 3. Sub-Rotinas para Manipulação de Requisições

Operação	Requisição Única	Múltiplas Requisição	
		Uma entre várias	Todas
Testar	MPI_TEST	MPI_TEST_ANY	MPI_WAIT_ANY
Aguardar	MPI_WAIT	MPI_WAIT_ANY	MPI_WAIT_ALL
Cancelar	MPI_REQUEST_FREE		

Sub-rotinas blocantes e não-blocantes são de três tipos, as quais se diferenciam por determinar diferentes condições para operações de envio completarem-se: (1) síncronas, nas quais a rotina de envio completa-se somente quando a rotina de recepção correspondente completa-se; bufferizadas, nas quais a rotina de envio completa-se tão logo é executada, sendo a mensagem copiada em um *buffer* explicitamente alocado pelo usuário e informado ao ambiente MPI através da rotina MPI_BUFFER_ATTACH; e prontas, as quais exigem que a operação de recepção tenha sido realizada anteriormente a operação de envio correspondente, tornando possível a cópia direta da mensagem do *buffer* da aplicação do emissor para o *buffer* de aplicação do receptor.

Tabela 4. Sub-Rotinas de Comunicação Coletiva

Um para Todos	MPI_BCAST	O processo <i>raiz</i> envia o <i>buffer</i> <i>B</i> para todos os processos.
	MPI_SCATTER	O processo <i>raiz</i> envia o <i>buffer</i> $B_1B_2\dots B_N$, tal que B_i tem tamanho S , para todos os processos, de tal forma que o processo de <i>rank i</i> armazena o pedaço B_i do <i>buffer</i> .
	MPI_SCATTERV	O processo <i>raiz</i> envia o <i>buffer</i> $B_1B_2\dots B_N$, tal que B_i tem tamanho S_i , para todos os processos, de tal forma que o processo de <i>rank i</i> armazena o pedaço B_i do <i>buffer</i> .
Todos para Um	MPI_GATHER	Cada processo <i>i</i> envia <i>buffer</i> local B_i de tamanho S ao processo <i>raiz</i> , o qual concatena-os no <i>buffer</i> $B_1B_2\dots B_N$, de tamanho $N \times S$.
	MPI_GATHERV	Cada processo <i>i</i> envia <i>buffer</i> local B_i de tamanho S_i ao processo <i>raiz</i> , o qual concatena-os no <i>buffer</i> $B_1B_2\dots B_N$, de tamanho $S_1+S_2+\dots+S_N$.
	MPI_REDUCE	Cada processo <i>i</i> envia <i>buffer</i> local B_i de tamanho S ao processo <i>raiz</i> , o qual concatena-os no <i>buffer</i> $B_1 \oplus B_2 \oplus \dots \oplus B_N$, de tamanho S . A operação \oplus , do tipo MPI_OP, é aplicada entre os elementos correspondentes dos <i>buffers</i> .
Todos para Todos	MPI_ALLGATHER	O mesmo de MPI_GATHER, porém todos os processos atuam como raízes.
	MPI_ALLGATHERV	O mesmo de MPI_GATHERV, porém todos os processos atuam como raízes.
	MPI_ALLTOALL	Cada processo <i>i</i> envia o <i>buffer</i> $A_{i,1}A_{i,2}\dots A_{i,N}$, onde cada $A_{i,k}$ tem tamanho N , para todos os processos. Ao completar-se, cada processo <i>j</i> armazena o <i>buffer</i> $A_{1,j}A_{2,j}\dots A_{N,j}$. Esta operação corresponde a uma <i>transposição</i> de dados entre os processadores envolvidos.
	MPI_ALLTOALLV	Forma generalizada de MPI_ALLTOALL. Cada processo <i>i</i> ainda envia o <i>buffer</i> $A_{i,j}$, $j=1,\dots,N$, para o processo <i>j</i> . O <i>buffer</i> $A_{i,j}$ deve ser obtido de um <i>buffer</i> original B_s a um deslocamento $sdispl_i$ de seu endereço base e com tamanho $scount_{i,j}$. O processo <i>j</i> deve armazenar $A_{i,j}$ em um <i>buffer</i> B_r , a um deslocamento de $rdispl_i$ de seu endereço base, com tamanho $rcount_{j,i}$. Evidentemente, $scount_{i,j} = rcount_{j,i}$.
	MPI_ALLTOALLW	Forma generalizada de MPI_ALLTOALLV, proposta por MPI-2. Os elementos do <i>buffer</i> $A_{i,j}$ são do tipo $systype_{i,j}$, do lado do emissor <i>i</i> , e $stype_{j,i}$ do lado do receptor <i>j</i> .
	MPI_ALLREDUCE	O mesmo que MPI_ALLTOALL, com a diferença de que cada processo <i>j</i> armazena $A_{1,j} \oplus A_{2,j} \oplus \dots \oplus A_{N,j}$, aplicando a operação \oplus sobre os elementos correspondentes
	MPI_REDUCE_SCATTER	Corresponde a uma aplicação de MPI_REDUCE seguida de MPI_SCATTER do resultado.
Prefixo	MPI_SCAN	Cada processo <i>i</i> envia um <i>buffer</i> B_i . Ao final, o processo <i>j</i> armazena $B_1 \oplus \dots \oplus B_j$.
Barreira	MPI_BARRIER	Todos os processos bloqueiam até que todos tenham atingido a barreira.

9.4.3. Comunicação Coletiva (MPI-1, extendido no MPI-2)

De fato, na maioria dos programas paralelos em aplicações de alto desempenho, a interação entre processos é realizada por uma intercalação de fases de computação e

de comunicação, estas últimas envolvendo a movimentação de dados entre os processos, realizada por meio de uma seqüência coordenada de operações de comunicação ponto-a-ponto. Neste contexto, processos atuam como *pares*, sem uma diferenciação entre os papéis atribuídos a cada um deles ou qualquer organização hierárquica. Em geral, estas seqüências de operações ponto-a-ponto em fases de comunicação descrevem padrões de movimentação de dados entre uma coleção de processos que são frequentemente usados aplicações de CAD. Por este motivo, o padrão MPI incorpora o conceito de sub-rotina de comunicação coletiva, capturando um padrão típico e reusável de troca de dados entre um conjunto de processos. A Tabela 4 classifica e descreve, em alto nível, as operações de comunicação coletiva disponibilizadas pelos padrões MPI-1 e MPI-2. Em [Gorlatch 2004], é discutida a generalidade do estilo de programação paralela totalmente baseada em operações coletivas, atribuindo a este o conceito de *programação paralela estruturada*, fazendo uma analogia do uso de primitivas *send/receive* com comandos *goto* em linguagens de programação não-estruturada. O uso de sub-rotinas de comunicação coletiva garante a implementação eficiente dos padrões de movimentação de dados por estas implementadas, retirando essa preocupação do programador. De fato, a otimização de operações de comunicação coletiva, especialmente em arquiteturas com várias hierarquias de paralelismo, é assunto de vários projetos de pesquisa, com resultados novos surgindo eventualmente.

Tabela 5. Sub-Rotinas p/ Gerenciamento e Criação de Grupos e Comunicadores

	Grupo	Comunicador	
		Intra	Inter
Gerenciamento	MPI_GROUP_SIZE MPI_GROUP_RANK MPI_TRANSLATE_RANKS MPI_GROUP_COMPARE	MPI_COMM_SIZE MPI_COMM_RANK MPI_COMM_COMPARE	MPI_COMM_TEST_INTER MPI_COMM_REMOTE_SIZE MPI_COMM_REMOTE_GROUP
Construtores	MPI_COMM_GROUP MPI_GROUP_UNION MPI_GROUP_INTERSECTION MPI_GROUP_DIFFERENCE MPI_GROUP_INCL MPI_GROUP_EXCL MPI_GROUP_RANGE_INCL MPI_GROUP_RANGE_EXCL	MPI_COMM_DUP MPI_COMM_CREATE MPI_COMM_SPLIT	MPI_COMM_DUP MPI_INTERCOMM_CREATE MPI_INTERCOMM_MERGE
Destruidores	MPI_GROUP_FREE	MPI_COMM_FREE	

9.4.4. Escopo de Comunicação (Grupos e Comunicadores)

Um dos requisitos principais do projeto de MPI é oferecer o suporte necessário a construção de bibliotecas científicas de propósitos específicos com sub-rotinas pré-paralelizadas. Uma vez que várias bibliotecas podem estar sendo utilizadas em uma mesma aplicação, é necessária a introdução de um mecanismo de escopo de comunicação para evitar confusão entre mensagens em tráfego. Além disso, os mecanismos propostos por MPI para este fim são também úteis para execução de operações coletivas restritas a um certo subconjunto do conjunto de processos da aplicação, notadamente processos situados em alguma região de uma certa topologia. Por exemplo, em uma malha retangular de processos, pode ser necessário realizar operações coletivas envolvendo somente os processos situados na mesma linha, na mesma coluna, ou nas diagonais.

Para o suporte a escopos de comunicação, O MPI define os conceitos de grupo de processos e comunicador, este último para comunicação entre processos pertencentes a um grupo (*intra-comunicadores*) e para pares de processos pertencentes a grupos disjuntos (*inter-comunicadores*). Os respectivos manipuladores para grupos e

comunicadores são dos tipos MPI_GROUP e MPI_COMM. Para o suporte a grupos e comunicadores, MPI oferece sub-rotinas para criação e gerenciamento destes, as quais se encontram resumidas na Tabela 5. Vale ressaltar ainda que o *rank* de um processo está definido em relação a um grupo. Portanto, em cada grupo ao qual o processo pertence, este pode ter um *rank* diferente, relativo aos outros processos no grupo.

Intra-comunicadores surgem nos parâmetros de operações ponto-a-ponto e coletivas, estabelecendo o escopo em que as operações devem ser realizadas. Por exemplo, usando o exemplo citado anteriormente, comunicadores podem ser criados para os grupos de processos pertencentes a uma mesma linha e/ou mesma coluna em uma malha retangular de processos. Inter-comunicadores só podem ser usados em operações ponto-a-ponto. De fato, estes têm aplicação especialmente para o estabelecimento de relações cliente/servidor entre dois processos pertencentes a dois grupos, respectivamente chamados *grupo local*, onde se encontra o processo dito *cliente*, e o outro *grupo remoto*, onde se encontra o processo dito *servidor*, os quais devem ser disjuntos para evitar situações de impasse (*deadlock*).

9.4.5. Topologias

Formalmente, a programação por passagem de mensagens pressupõe a existência de *canais de comunicação* que interligam pares de processos. Por este motivo, este estilo é também conhecido como programação distribuída baseada em canais. Tomando-se processos por *vértices* e canais por *arestas*, o grafo resultante é conhecido como *topologia de processos*. A maioria dos algoritmos paralelos e distribuídos pressupõe a interconectividade total da topologia, muitas vezes como forma de simplificá-los. Esta é uma suposição razoável tendo em vista que se espera que numa rede cada par de processos em uma arquitetura paralela sejam capazes de trocar mensagens. Entretanto, a interconectividade total nem sempre é uma suposição realista do ponto de vista das interfaces de comunicação, especialmente entre as interconexões de alto desempenho, como Infiniband, Myrinet, Quadrics, dentre outras. Por exemplo, redes do padrão Quadrics assumem uma topologia *fat tree* entre os nós. O conhecimento sobre a topologia da rede física tem um impacto importante sobre o desempenho de operações de comunicação coletiva, uma vez que algoritmos de disseminação especiais para a topologia podem ser aplicados. Por este motivo, o MPI permite associar a um intra-comunicador uma topologia na qual os processos do grupo estão *virtualmente* organizados. Esta informação pode ser usada por sistemas em tempo de execução para mapeamento dos processos aos processadores, buscando otimizar as operações de comunicação.

O MPI oferece sub-rotinas para especificação e criação de topologias associadas a comunicadores, como MPI_GRAPH_CREATE (grafos arbitrários) e MPI_CART_CREATE (estruturas cartesianas e hipercubos); recuperação de informações sobre a topologia de um comunicador, como MPI_GRAPHDIMS_GET, MPI_GRAPH_GET, MPI_CARTDIM_GET, e MPI_CART_GET; mapeamento de coordenadas cartesianas em *ranks* de grupos, como MPI_CART_RANK e MPI_CART_COORDS; extração de sub-espacos cartesianos, como MPI_CART_SUB; informação necessária para comunicação com vizinhos em uma topologia cartesiana, como MPI_CART_SHIFT, ou em um grafo, como MPI_GRAPH_NEIGHBORS_COUNT e MPI_GRAPH_NEIGHBORS.

Tabela 6. Criação de Tipos de Dados Derivados

MPI_TYPE_STRUCT	Registro formado por <i>nfield</i> campos. O <i>i</i> -ésimo campo tem <i>counts_i</i> elementos do tipo <i>types_i</i> .
MPI_TYPE_CONTIGUOUS	Um tipo composto por <i>count</i> ocorrências contínuas do tipo <i>type</i> .
MPI_TYPE_VECTOR	<i>N</i> blocos, cada um com <i>bcount</i> elementos e o início de cada bloco separado por <i>stride</i> elementos
MPI_TYPE_INDEXED	Sequência de <i>N</i> blocos. O <i>i</i> -ésimo bloco com <i>length_i</i> elementos e com início separado do anterior por <i>displ_i</i> elementos.
MPI_TYPE_HVECTOR	O mesmo que MPI_TYPE_VECTOR , porém com <i>stride</i> em <i>bytes</i> .
MPI_TYPE_HINDEXED	O mesmo que MPI_TYPE_INDEXED , porém com <i>stride</i> em <i>bytes</i> .

9.4.6. Tipos Derivados

Nas sub-rotinas de comunicação de MPI, os dados a serem transmitidos devem estar dispostos em um *buffer*, no qual os elementos estão contiguamente dispostos. Em linguagens como C e Fortran, onde aplicações científicas em geral fazem uso de matrizes e vetores em armazenamento seqüencial, tal suposição é razoável, permitindo uma substancial redução do tempo de preparação de dados para envio em mensagens e recuperação de dados em mensagens recebidas. Entretanto, tal suposição não é válida quando as estruturas a serem transmitidas são complexas e dispostas não-contiguamente na memória, cujo exemplo mais simples que podemos citar é uma lista encadeada. Para estas o MPI oferece rotinas de empacotamento (**MPI_PACK**) e desempacotamento (**MPI_UNPACK**) que permitem ao programador dispor os elementos da estrutura em um *buffer* de envio e remontá-la a partir do *buffer* de entrada. De fato, o MPI usa internamente as sub-rotinas **MPI_PACK** e **MPI_UNPACK** para empacotar dados de tipos complexos nativos do MPI. Além disso, MPI oferece o suporte à criação de tipos de dados derivados pelos usuários, utilizando o conjunto de sub-rotinas descritas na Tabela 6. Atravessando estas estruturas por meio das informações providas pelo usuário, o MPI automaticamente é capaz de realizar as chamadas apropriadas de **MPI_PACK** e **MPI_UNPACK** antes de cada operação de comunicação, retirando essa responsabilidade de usuários do *tipo derivado*. Isso pode ser usado, por exemplo, para oferecer maior nível de abstração no projeto de bibliotecas científicas.

9.4.7. Criação e Gerenciamento Dinâmico de Processos

A possibilidade da criação e gerenciamento dinâmico de processos é uma das principais novidades do padrão MPI-2 em relação ao padrão MPI-1, tendo em vista as demandas de muitas aplicações, especialmente aquelas que desejam migrar do PVM (*Parallel Virtual Machine*) [Geist 1994], biblioteca de troca de mensagens para arquiteturas distribuídas heterogêneas bastante difundida antes do MPI. Entretanto, o padrão MPI, por questões de seu compromisso com portabilidade e eficiência, possui algumas limitações em relação a sistemas baseados em máquinas virtuais, as quais assumem responsabilidades inerentes aos sistemas operacionais.

O MPI-2 oferece basicamente duas sub-rotinas para criação dinâmica de processos: **MPI_COMM_SPAWN** e **MPI_COMM_SPAWN_MULTIPLE**. Estas permitem o disparo de um único ou vários programas MPI, respectivamente, retornando um inter-comunicador para comunicação com cada um dos grupos de processos associados a cada programa disparado. A sub-rotina **MPI_COMM_GET_PARENT** permite recuperar o inter-comunicador “pai” de um programa MPI disparado por meio destas sub-rotinas.

Além de permitir o disparo de programas MPI e o estabelecimento de comunicação com estes, o MPI oferece ainda a possibilidade da integração de programas MPI disparados independentemente, através das sub-rotinas **MPI_OPEN_PORT**, a ser chamada

pelo programa MPI dito *servidor* para estabelecer uma porta através da qual pode ser contatada por programas MPI ditos *clientes*; MPI_COMM_CONNECT, a ser chamada pelo cliente para conectar a um servidor; MPI_COMM_ACCEPT, a ser chamada pelo servidor para aceitar um pedido de conexão de um cliente. A sub-rotina MPI_CLOSE_PORT pode ser chamada pelo servidor para fechar uma porta previamente aberta. São ainda disponibilizadas rotinas para publicação de nomes de portas por programas MPI servidores, de forma a que estejam acessíveis a programas MPI clientes, como MPI_PUBLISH_NAME, MPI_UNPUBLISH_NAME, MPI_LOOKUP_NAME.

Tabela 7. Manipulação de Arquivos (E/S Paralela)

MPI_FILE_OPEN *	Abre um arquivo em todos os processos no contexto de um comunicador. Os modos de acesso possíveis são MPI_MODE_XXX, onde XXX pode ser RDONLY, RDWR, WRONLY, CREATE, EXCL, DELETE_ON_CLOSE, UNIQUE_OPEN, SEQUENTIAL, APPEND.
MPI_FILE_CLOSE	Fecha um arquivo previamente aberto por MPI_FILE_OPEN.
MPI_FILE_DELETE	Exclui um arquivo do sistema de arquivos.
MPI_FILE_SET_SIZE	Altera o tamanho de um arquivo para um novo tamanho.
MPI_FILE_PREALLOCATE	Garante que espaço de armazenamento é alocado para os primeiros <i>b</i> bytes de um arquivo.
MPI_FILE_GET_SIZE	Retorna o tamanho de um arquivo.
MPI_FILE_GET_GROUP	Retorna uma cópia do grupo do comunicador que está acessando um certo arquivo.
MPI_FILE_GET_AMODE	Retorna o modo de acesso de um certo arquivo aberto.
MPI_FILE_SET_INFO	Atribui novos valores às informações do arquivo especificado.
MPI_FILE_GET_INFO	Retorna um novo objeto MPI_INFO com as informações sobre o arquivo especificado.
MPI_FILE_SET_VIEW	Modifica a visão dos processos sobre os dados armazenados pelo arquivo especificado.
MPI_FILE_GET_VIEW	Retorna a visão dos processos sobre os dados armazenados pelo arquivo especificado.

* Operações coletivas entre os todos os processos envolvidos em um comunicador.

Tabela 8. Acesso a Dados (E/S Paralela)

Posicionamento	Modo de Sincronização	Coordenação	
		Simples	Coletiva
deslocamento explícito	Blocante	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	Não-blocante e divisão coletiva	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
Pointeiros de arquivo individuais	Blocante	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	Não-blocante e divisão coletiva	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
Ponteiros de arquivo compartilhados	Blocante	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	Não-blocante e divisão coletiva	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

9.4.8. Entrada/Saída

O acesso a dados brutos armazenados em arquivos compartilhados entre o conjunto de processos é uma necessidade recorrente em aplicações paralelas em computação de alto desempenho. Supõe-se que o conjunto de processos sejam capazes de ler e escrever sobre estes arquivos simultaneamente e de maneira eficiente, naquilo que convencionou-se definir como E/S paralela. Este tipo de funcionalidade não pode ser obtida, de maneira eficiente, em sistemas como POSIX, por exemplo, a despeito de sua portabilidade, exigindo mecanismos específicos que incorporam as diversas oportunidades de otimização em E/S paralela peculiares a aplicações de computação científica. Assim, o MPI-2 acrescentou ao MPI um conjunto de sub-rotinas para E/S

paralela, as quais permitem a descrição do particionamento de dados utilizando *tipos de dados derivados*, descritos anteriormente.

A importância da funcionalidade de E/S paralela eficiente tem crescido nos últimos anos, devido ao crescimento de extensas bases de dados usadas em muitas aplicações, especialmente simulações, incluindo dados obtidos de medição direta e gerados por outras simulações, e aplicações em genômica, onde as bases de dados de informações genéticas crescem exponencialmente. Entretanto, um foco cada vez maior tem sido dado à criação de formatos de dados científicos, especialmente usando o formato XML, permitindo as aplicações uma maior facilidade de acessar somente a informação relevante para um contexto específico, através da computação de resumos sobre os dados brutos. Esta possibilidade é especialmente importante nos casos onde as bases de dados são remotas e suficientemente grandes para que sua transmissão através de uma rede de comunicação seja impraticável.

9.4.9. Comunicação Unilateral

O modo de comunicação unilateral, mais precisamente conhecido como *acesso remoto à memória* (RMA), foi introduzido no padrão MPI-2 como uma alternativa ao modo de comunicação ponto-a-ponto característico do MPI-1, sobre o qual muitas aplicações podem se beneficiar, em especial aquelas onde os padrões de acesso aos dados modificam-se dinamicamente mas a distribuição dos dados é fixa ou modifica-se muito pouco no decorrer do seu ciclo de vida. Dessa forma, um processo que deseja acessar dados mantidos por outro processo pode usar o padrão de acesso que o convier, ao passo que o processo cujos dados serão acessados não precisam preocupar-se com que dados serão acessados ou atualizados em sua memória por outros processos, cuja identidade também pode ser ignorada. Assim, no modo de comunicação unilateral de MPI, a sub-rotina MPI_WIN_CREATE permite que grupos de processos possam criar janelas de comunicação em seus espaços locais de endereçamento, os quais podem ser diretamente manipulados por outros processos. Tais janelas podem ser liberadas utilizando a sub-rotina MPI_WIN_FREE ou ter seus atributos inspecionados por meio da rotina MPI_WIN_GET_GROUP. No que diz respeito ao acesso a janelas de comunicação por outros processos, enquanto a comunicação por passagem de mensagens tradicional combina os efeitos de *comunicação* e *sincronização* entre os processos comunicantes, o mecanismo de comunicação unilateral separa os dois efeitos, fornecendo conjuntos separados de sub-rotinas para *comunicação*, como MPI_PUT e MPI_GET, e *sincronização*, como MPI_WIN_FENCE, MPI_WIN_START, MPI_WIN_COMPLETE, MPI_WIN_WAIT, MPI_WIN_TEST, MPI_WIN_LOCK, MPI_WIN_UNLOCK. Portanto, é tarefa do programador garantir o acesso consistente de um conjunto de processos a uma mesma janela.

9.4.10. Interfaces Externas

O MPI-2 incorpora um conjunto de sub-rotinas necessárias definição de operações não-blocantes com interface semelhantes aquelas pré-existentes no MPI. Por tratar-se de um conjunto de funcionalidades geralmente útil somente para desenvolvedores de bibliotecas, porém raramente essencial a programadores de aplicações, para quem se destina este documento, sugerimos ao leitor buscar mais informações sobre as sub-rotinas envolvidas no capítulo 8 da referência padrão.

9.5. O OpenMP: Explorando Paralelismo em Memória Compartilhada

Durante a década de oitenta e início da década de noventa, a predominância das SMPs (*Shared Memory Multiprocessors*) no mercado de supercomputação levou à disponibilização de compiladores, especialmente aqueles desenvolvidos para extensões proprietárias da linguagem Fortran, que tentavam explorar o paralelismo destas arquiteturas por meio da paralelização implícita de laços no processamento de matrizes e vetores [Bell e Gray 2002]. Tais compiladores introduziam um conjunto de diretivas de compilação que permitiam aos programadores explicitamente informar quais os laços de programas que poderiam ser executados em paralelo. A primeira tentativa de padronizar este tipo de interface levou ao padrão ANSI X3H5, o qual não obteve o sucesso esperado. Anos mais tarde, a partir do ano de 1996, um consórcio iniciado na indústria propôs o OpenMP, especificação que inclui um conjunto de diretivas de compilação e sub-rotinas para suporte ao programação paralela de memória compartilhada ao nível de *threads* [Throop 1999]. O OpenMP, assim como seus precursores proprietários, está focalizado no paralelismo de dados, especialmente no paralelismo de laços onde estruturas de dados são percorridas, permitindo ainda a criação de *threads* para seções funcionalmente independentes de código e introduzindo o *paralelismo aninhado*. Para isso, inclui um sub-conjunto de diretivas e sub-rotinas para sincronização e controle de interferência entre *threads*.

A programação com OpenMP pode ser entendida incrementalmente nas próximas quatro seções, assim resumidas. Na Seção 5.1 é apresentado o conceito de regiões paralelas, trechos de código, possivelmente aninhados, a serem executados por um conjunto *threads* em paralelo. Na Seção 5.2 será mostrado como é possível dividir os trechos de uma região paralela entre *threads* diferentes. Na Seção 5.3, são apresentados os mecanismos de controle de concorrência suportados por OpenMP, permitindo o controle de interferência entre *threads*. Na Seção 5.4, será discutido como controlar o escopo de variáveis entre *threads*, as quais são globais por *default*, tornando-as locais aos *threads* e permitindo a redução de valores computados. Na Seção 5.5, são apresentadas as rotinas de controle do sistema de execução.

```
#pragma omp parallel [cláusula [ [, ] cláusula] ...] nova-linha
                    bloco-estruturado

cláusula → if(expressão-escalar) | private(lista-de-variáveis) | firstprivate(lista-de-
            variáveis) | default (shared | none) | shared(lista-de-variáveis) |
            copyin(lista-de-variáveis) | reduction(operator:lista-de-variáveis) |
            num_threads(expressão-inteira)
```

Diretiva 1. Região Paralela

9.5.1. Regiões Paralelas

Em Diretiva 1, é apresentada a estrutura sintática de uma região paralela. Quando um *thread* encontra uma região paralela, o trecho de código por esta delimitado é executado por um conjunto de *threads*, no qual o *thread* original é destacado como mestre. Isso somente não ocorre se houver alguma cláusula **if** cujo valor seja *zero*. Neste caso, o trecho de código é serializado. O número de *threads* a serem criados depende das seguintes condições, nesta ordem de prioridade: (1) presença da cláusula **num_threads**; (2) chamada a sub-rotina **omp_set_num_threads** mais recente; (3) valor da variável de ambiente **OMP_NUM_THREADS**; ou (4) valor *default* da implementação. O

número de *threads* pode também ser ajustado dinamicamente, usando-se chamada a sub-rotina `omp_set_dynamic` ou controlando-se o valor da variável de ambiente `OMP_DYNAMIC`. Neste caso, é escolhido o maior número de *threads* possíveis, independente do valor estaticamente estipulado, o qual depende da implementação. Ao final de uma região paralela, há uma barreira implícita, de forma que esta só termina após o término de todos os *threads*, quando então somente o *thread* mestre prossegue. Os significados das demais cláusulas serão explicados nas próximas seções, especialmente na seção 5.3, por estas tratarem em sua maioria do compartilhamento de dados entre *threads*.

```
#pragma omp for [cláusula [ [, ] cláusula] ...] nova-linha
    laço-for
    cláusula → private(lista-de-variáveis) | firstprivate(lista-de-variáveis) |
        reduction(operador:lista-de-variáveis) | ordered | schedule kind | nowait
    kind → (static, chunck_size) | (dynamic, chunck_size) | (guided, chunck_size) |
        runtime
```

Diretiva 2. Distribuição de Laços em uma Região Paralela

```
#pragma omp sections [cláusula [ [, ] cláusula] ...] nova-linha
{
    [#pragma omp section nova-linha]
    bloco-estruturado
    [#pragma omp section nova-linha]
    bloco-estruturado
    ...
}
    cláusula → private(lista-de-variáveis) | firstprivate(lista-de-variáveis) |
        reduction(operador:lista-de-variáveis) | nowait
```

Diretiva 3. Distribuição de Seções de Código em uma Região Paralela

```
#pragma omp single [cláusula [ [, ] cláusula] ...] nova-linha
    bloco-estruturado
    cláusula → private(lista-de-variáveis) | firstprivate(lista-de-variáveis) |
        copyprivate(lista-de-variáveis) | nowait
```

Diretiva 4. Serialização de um Trecho de Código em uma Região Paralela

9.5.2. Divisão de trabalho

As diretivas de divisão de trabalho apresentadas em Diretiva 2 e Diretiva 3 permitem distribuir a execução de um trecho de código associado entre os *threads* de uma região paralela. A primeira, conhecida como construto for, permite a distribuição de iterações de um laço `for` entre o conjunto de *threads*. A última permite a divisão de seções funcionalmente independentes de código, representado por blocos estruturados da linguagem, entre os *threads* da região paralela. Há ainda a diretiva `single`, especificada em Diretiva 4, a qual serializa um trecho de código, executado pelo *thread* mestre.

Para parallelizar laços, é necessária a independência de dados entre iterações executadas concorrentemente. Esta possibilidade é particularmente importante quando é necessário percorrer uma grande estrutura de dados, o que é comum em aplicações científicas e de engenharia, bem como em processamento de imagens. O laço `for` deve

obedecer uma forma canônica. A distribuição dos laços entre as *threads* obedece a política de escalonamento estabelecida por meio da cláusula **schedule** no parâmetro *kind*, o qual pode ser estático, onde o conjunto de iterações é dividido em blocos de tamanho *chunk_size*, atribuídos estaticamente a cada *thread*; dinâmico, onde cada bloco é atribuído dinamicamente a um *thread* que está ocioso, esperando por trabalho; guiado, onde as interações são associadas aos *threads* em blocos de tamanho descendente; ou por intermédio do sistema de tempo de execução (*runtime*), onde pode ainda ser controlado pelo usuário por meio da variável de ambiente OMP_SCHEDULE. Existe uma barreira ao final de uma região **for**, a menos que a cláusula **nowait** seja especificada.

```
#pragma omp master nova-linha
    bloco-estruturado

#pragma omp critical nova-linha
    bloco-estruturado

#pragma omp barrier nova-linha

#pragma omp atomic nova-linha
    expressão-de-atribuição

#pragma omp flush [(lista-de-variáveis)] nova-linha

#pragma omp ordered nova-linha
    bloco-estruturado
```

Diretiva 5. Controle de Concorrência entre Threads

9.5.3. Controle de Concorrência

A possibilidade de executar trechos funcionalmente independentes de código, através da diretiva **sections** torna virtualmente possível a construção de qualquer programa concorrente, oferecendo maior flexibilidade para suportar padrões de paralelismo mais gerais que o paralelismo de dados, porém exigindo a introdução de mecanismos explícitos de controle de concorrência.

Em Diretiva 5, encontram-se as principais diretivas do OpenMP para controle de concorrência entre *threads*. A diretiva **master** estabelece um trecho de código que deve ser executado somente pelo *thread* mestre. A diretiva **critical** define uma região crítica, na qual somente uma *thread* de cada vez pode executá-la. A diretiva **barrier** define uma barreira entre as *threads*, a qual deve ser atingida por todos os *threads* da região paralela para que todas prossigam a execução. A diretiva **atomic** garante que uma operação de atualização de uma variável inteira associada ocorra atomicamente. A diretiva **flush** garante a visão consistente de uma lista de variáveis por parte dos *threads*. A diretiva **ordered** garante que o trecho de código associado seja executado na mesma ordem que iterações seriam executadas em um laço seqüencial.

Tabela 9. Gerenciamento de Locks

OMP_INIT_LOCK	Inicializa um <i>lock</i> simples.
OMP_DESTROY_LOCK	Remove um <i>lock</i> previamente criado.
OMP_SET_LOCK	Espera até que um <i>lock</i> esteja disponível.
OMP_UNSET_LOCK	Libera um <i>lock</i> simples.
OMP_TEST_LOCK	Testa um <i>lock</i> .

É importante ainda ressaltar a existência de sub-rotinas para controle de concorrência explícita por meio de *locks*, as quais são apresentadas na Tabela 9. Discute-se atualmente a futura inclusão de rotinas para sincronização através de semáforos, por ser este um mecanismo largamente utilizado em programação concorrente.

```
#pragma omp thread-private (lista-de-variáveis) nova-linha
```

Diretiva 6. Declarando Variáveis Locais aos Threads

Tabela 10. Cláusulas de Gerenciamento do Ambiente de Dados

cláusula	Função	Comportamento
private	Variáveis na lista são privadas às <i>threads</i> .	Uma cópia local da variável é criada para cada <i>thread</i> , a qual mantém-se não inicializada.
firstprivate	Inicializa a cópia com o valor atual da <i>thread</i> mestre.	Inicializada para o valor da variável original anterior à entrada na região paralela ou diretiva de divisão de trabalho.
lastprivate	A variável global é atualizada com a cópia da última iteração.	O membro do time que fará a iteração final no laço ou a última seção de sections atualiza o valor na variável global.
shared	A variável é compartilhada entre todas as <i>threads</i> .	A variável existe em apenas uma locação de memória para a qual todas as <i>threads</i> têm acesso para leitura e escrita. Controle de interferência pode ser necessário.
default (shared none)	Especifica todas as variáveis no escopo de uma região paralela como shared ou none	Cláusulas private , shared , firstprivate , lastprivate , e reduction sobrepõem-se a default . Uma diretiva parallel pode ter somente uma cláusula default . Apenas a API Fortran suporta default (private)
copyin	Atribui o mesmo valor a todas as variáveis threadprivate para todas as <i>threads</i>	<i>List</i> contém os nomes das variáveis a copiar. O valor é copiado da variável da <i>thread</i> mestre.
reudction	Realiza uma redução na variável listada	Cria uma variável global, copia seu valor para todas as suas cópias privadas usadas pelas <i>threads</i> . Ao final, a variável global é atualizada combinando-se o valor original de cada uma das cópias privadas usando o operador especificado.

9.5.4. Ambiente de Dados

Por **default**, as variáveis no escopo de uma região paralela são consideradas globais aos *threads* que executam em seu contexto. Este comportamento pode ser alterado utilizando-se a diretiva **threadprivate**, apresentado em Diretiva 6, ou através de um conjunto de cláusulas que aparecem nas diretivas **parallel**, **sections**, **for**, e **single**, as quais encontram-se descritas na Tabela 10. A diretiva **threadprivate** torna um conjunto de variáveis privadas aos *threads* de regiões paralelas que a seguem. Para cada *thread*, é criada uma cópia independente da variável, cujo valor pode persistir entre regiões paralelas seguidas caso o número de *threads* mantenha-se entre estas. Caso este seja diferente, novas cópias são criadas em cada região paralela. Portanto, para que a persistência seja possível o mecanismo de controle dinâmico do número de *threads* nunca poderia estar desabilitado. Obviamente, em regiões seriais (**single**) ou em trechos que devem ser executadas somente pelo *thread* mestre (**master**), a cópia mantida pela *thread* mestre é a utilizada. Além disso, vale ressaltar a impossibilidade de um *thread* acessar uma cópia de uma variável privada mantida por um outro *thread*.

Tabela 11. Sub-Rotinas de Controle do Sistema de Tempo de Execução

OMP_SET_NUM_THREADS	Especifica o número default de <i>threads</i> a serem usadas em regiões paralelas subsequentes que não especificam uma cláusula num_threads .
OMP_GET_NUM_THREADS	Retorna o número de <i>threads</i> correntes que estão executando a região paralela de onde é chamada.
OMP_GET_MAX_THREADS	Retorna um inteiro que é garantido ser pelo menos tão grande quanto o número de <i>threads</i> que seriam usadas se uma região paralela sem a cláusula num_threads fosse encontrada naquele ponto.
OMP_GET_THREAD_NUM	Retorna o número da <i>thread</i> que executa a função.
OMP_GET_NUM_PROCS	O número de processadores disponíveis ao programa no momento que a função é chamada.
OMP_IN_PARALLEL	Retorna um valor não-zero se é chamado dentro de uma extensão dinâmica de uma região paralela executando em paralelo; caso contrário, retorna 0.
OMP_SET_DYNAMIC	Habilita ou desabilita o ajuste dinâmico do número de <i>threads</i> para executar as regiões paralelas.
OMP_GET_DYNAMIC	Retorna um valor não-zero se o ajuste dinâmico de <i>threads</i> está habilitado. Caso contrário, zero.
OMP_SET_NESTED	Habilita ou desabilita o paralelismo aninhado.
OMP_GET_NESTED	Retorna um valor não-zero se o paralelismo aninhado está habilitado. Caso desabilitado, zero.
OMP_GET_WTIME	Retorna o tempo decorrido.
OMP_GET_WTICK	Retorna os segundos entre ticks sucessivos.

9.5.5. Controle do Ambiente de Execução

O OpenMP oferece um conjunto de sub-rotinas e variáveis de ambiente que permitem o controle do sistema em tempo execução, cujo significado está respectivamente descrito na Tabela 11 e na Tabela 12.

Tabela 12. Variáveis de Ambiente

OMP_SCHEDULE	Especifica o tipo do escalonamento em tempo de execução e o tamanho do <i>chunk</i> . Valor default: é STATIC, sem especificação do tamanho do <i>chunk</i> .
OMP_NUM_THREADS	Especifica o número de <i>threads</i> a usar na execução. Valor default: Número de processadores.
OMP_DYNAMIC	Habilita/desabilita o ajuste dinâmico do número de <i>threads</i> . Valor default: falso
OMP_NESTED	Habilita/desabilita o paralelismo aninhado. Valor default: falso

9.6. Uso Conjunto do MPI e OpenMP

Em *clusters* compostos por nós que suportam multiprocessamento, arquitetura que tende a se tornar dominante no cenário da computação de alto desempenho, o uso conjunto de MPI e OpenMP é considerada a forma mais atraente para explorar o pico de desempenho destas arquiteturas, a despeito das tentativas, ainda incipientes, de tornar o MPI mais eficiente em memória compartilhada e das tentativas de oferecer versões distribuídas do OpenMP. Em um ambiente com tal nível de heterogeneidade e no qual os usuários demandam pela utilização eficiente dos recursos computacionais disponíveis, o uso em conjunto de MPI e OpenMP ainda é a melhor alternativa. Enquanto o MPI é responsável pela distribuição do processamento nos nós do *cluster*, o OpenMP é responsável pelo processamento paralelo do trabalho realizado em cada nó individual. Entretanto, é importante ressaltar que o uso em conjunto do MPI e OpenMP requer certos cuidados práticos, uma vez que poucas implementações do MPI são seguras para execução concorrentes de sub-rotinas de comunicação. De fato, a implementação *thread-safe* (segura para uso com *threads*) do padrão MPI, algo que é discutido no documento que contém a sua especificação oficial, em geral é acompanhada de uma maior sobrecarga de execução, considerada relevante em muitas aplicações, especialmente quando estas executam em arquiteturas de acoplamento forte entre os nós de processamento, como MPPs de interconexão proprietária. Uma vez que poucas aplicações, especialmente em CAD, fazem uso efetivo de operações de comunicação concorrentes, é comum optar-se por implementações MPI inseguras para o uso com *threads*, ou então oferecer-se a possibilidade de habilitar o desabilitar esta funcionalidade.

Para ilustrar o uso conjunto do OpenMP e MPI, utilizamos um programa paralelo para ordenação de 2^K chaves inteiras no intervalo $[0, 2^N]$ utilizando o algoritmo *bucketsort*, cujo código fonte está apresentado no Apêndice deste capítulo e baseia-se nas versões MPI e OpenMP do *benchmark IS* do *NAS Parallel Benchmarks* (NPB) [Bailey 1991]. O paralelismo desta aplicação é explorado em dois níveis: distribuído, onde faz-se uso do MPI, e memória compartilhada, onde faz-se uso do OpenMP.

A Figura 9 ilustra a estratégia de paralelização usada no nível distribuído, em cada nó do *cluster*, para $K=5$ e $N = 5$. É importante porém ressaltar que este algoritmo é eficiente para casos onde $N \gg K$. Em cada nó, na transição de (1) para (2), o conjunto de 16 chaves locais, armazenado no vetor *key_array*, é dividido em 4 *buckets*, cada qual cobrindo um sub-intervalo de chaves de mesma amplitude (linhas 93 a 121). De fato, os *buckets* são representados por fatias contíguas do vetor *key_buff1*. Esta pré-ordenação é

realizada por um conjunto de *threads* paralelas, como explicado em seguida. Cada processo divide o conjunto de seus *buckets* em sub-conjuntos de mesmo tamanho a serem enviados para cada processo, portanto entre si. No exemplo, como há 4 buckets, cada nó enviará os dois primeiros buckets para o processo 0 e os dois últimos para o processo 1. Isto corresponde a uma operação de comunicação coletiva do tipo *todos para todos*. De fato, é implementada por uma sequência de três chamadas a operações coletivas (`MPI_ALLREDUCE`, `MPI_ALLTOALL` e `MPI_ALLTOALLV`), com pode ser observado nas linhas 125 a 153 do código fonte, de forma a que todos os processos conheçam as chaves mantidas por cada outro processo. As duas primeiras chamadas servem para construir os vetores `send_count`, `recv_count`, `send_displ`, e `recv_displ` necessários na última chamada, nos os conjuntos de chaves são trocadas. Após esta operação, cada processo contém os *buckets* correspondentes as faixas de chaves que deverá manter no final. É então realizada, na transição de (3) para (4), um ordenação direta em paralelo com OpenMP (linhas 176 a 197), utilizando os próprios valores das chaves para indexá-las.

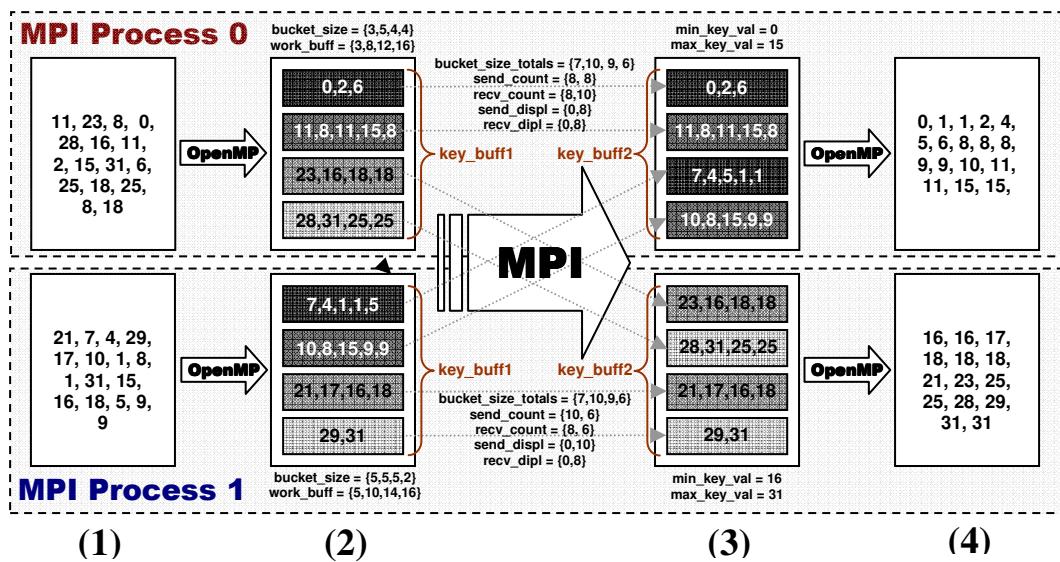


Figura 9. Ordenação de Inteiros Paralela (Nível MPI)

A Figura 10 ilustra o uso de OpenMP para separar as chaves armazenadas localmente por um nó nos *buckets*. Para tal, o vetor `key_array` é particionado entre as *threads*, de modo que cada uma computa a população de cada *bucket* relativa a sua fatia do vetor, a qual é armazenada na variável `bucket_size_local`, privada a cada *thread* (linhas 90 a 94). A população total de cada *bucket* é facilmente calculada acumulando-se as populações parciais de cada *bucket* no vetor global `bucket_size` (linhas 99 a 104). Além disso, usando o vetor `work_buf` computa-se as posições parciais das chaves em cada *bucket*, através de uma operação de *scan* sobre o vetor `bucket_size_local` (linhas 99 a 104) seguido da acumulação dos vetores no vetor `key_buff_ptr` (linhas 108 a 112). O buffer onde serão armazenada as chaves classificadas segundo os *buckets* é construído nas linhas 116 a 121. O uso do OpenMP para a ordenação final é semelhante. De fato, pode-se considerar que, neste caso, temos um *bucket* para cada chave, onde estão as chaves de mesmo valor. Por este motivo, esse algoritmo funciona bem quando o número de chaves possíveis é algumas ordens de magnitude menor que o número de chaves a serem ordenadas. No código fonte, está ilustrado o tamanho de problema B, do NPB-IS, onde K=21 e N=25. O número de *buckets* para esse caso é 2^{10} .

É importante enfatizar que as chamadas a sub-rotinas de comunicação do MPI neste programa são sempre realizadas por uma única *thread* (diretiva *single* na linha 123), tornando não essencial o uso de uma implementação MPI *thread-safe*. Embora a paralelização do trecho de comunicação possivelmente ofereça algum ganho de concorrência, isso poderia não compensar a sobrecarga associada ao uso de uma implementação MPI *thread-safe*, especialmente em arquiteturas de acoplamento muito forte de paralelismo e aplicações com paralelismo de granularidade fina.

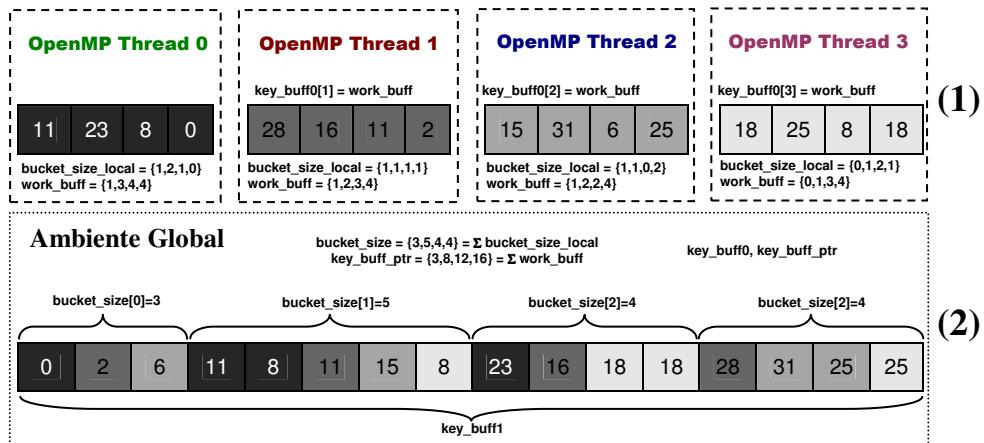


Figura 10. Ordenação de Inteiros Paralela (Nível OpenMP)

9.7. Considerações Finais

A previsão da consolidação da programação paralela como tecnologia dominante no cenário da computação tem sido largamente disseminada pela indústria de *hardware* e *software*, tendo em vista o crescimento em importância industrial das aplicações tradicionais de computação de alto desempenho (CAD), notadamente ciências computacionais e engenharia; o uso de técnicas de CAD em aplicações de interesse comercial, especialmente na área financeira; e, principalmente, a consolidação e disseminação das arquiteturas de processadores de múltiplos núcleos, especialmente quando tornar-se comum a existência de processadores com número igual ou maior de oito núcleos de processamento, quando será necessário às aplicações individuais de uso cotidiano fazerem uso efetivo das técnicas de processamento paralelo para dispôr, de maneira escalável, do desempenho máximo do processador. Devido a este contexto, faz-se necessário disseminar as práticas de programação paralela orientadas à computação de alto desempenho entre desenvolvedores de *software* em geral.

O MPI (*Message Passing Interface*) e o OpenMP são hoje padrões de fato e de direito, de uso disseminado tanto na indústria quanto na academia, para aproveitamento do desempenho de arquiteturas de CAD com memória distribuída e compartilhada, respectivamente. Embora esforço venha sido empreendido para tornar o MPI eficiente em memória compartilhada e o OpenMP eficiente sobre arquiteturas distribuídas, o uso em conjunto dessas ferramentas ainda é a melhor forma de explorar o desempenho de clusters de multiprocessadores, arquitetura onde o conjunto de nós, interligados por meio de uma interconexão ou rede de alto desempenho, são compostos por vários núcleos de processamento.

9.8. Bibliografia

- Amdahl, G. M. (1967) Validity of single-processor approach to achieving large-scale computing capability, *Proceedings of AFIPS Conference*. 1967. pp. 483-485
- Armstrong, R. et al (1999), Towards a Common Component Architecture for High-Performance Scientific Computing, Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing, Redondo Beach - USA.
- Bailey, D. H. et al (1991). The NAS Parallel Benchmarks. International Journal of Supercomputing Applications, vol. 5, n. 3, pp. 63-73.
- Baker, M., Buyya, R. e Hyde, D. (1999). Cluster Computing: A High Performance Contender. Communications of the ACM, vol.42, n.7, 79-83.
- Baude, F., Caromel, D. e Morel, M. (2003), From Distributed Objects to Hierarchical Grid Components, Proceedings of the International Symposium on Distributed Objects and Applications (DOA2003) .
- Bell, G. e Gray, J. (2002) What's the Next in High Performance Computing. Communications of the ACM, Vol. 45, n. 2, pp. 91-95.
- Bernholdt D. E. (2004) Raising Level of Programming Abstraction in Scalable Programming Models. Proceedings of IEEE International Conference on High Performance Computer Architecture (HPCA), Workshop on Productivity and Performance in High-End Computing (P-PHEC), pp. 76-84, Madrid, Spain.
- Black, S. e Scholes, M. (1973) The Price of Options and Corporate Liabilities. *Journal of Political Economy*, 81(3): pp. 657-654.
- Carson, E., Cobelli, C. e BronzinO, J. (editors) (2006) Modelling Methodology for Phisiology and Medicine. Academic Press Biomedical Engineering Series.
- Carvalho Junior, F. H. e Lins, R. D. (2007) Towards an Architecture for Component-Oriented Parallel Programming. *Concurrency and Computation: Practice and Experience*, v. 19 n. 6, pp. 697-719. DOI: 10.1002/cpe.1121.
- Carvalho-Junior, F. H. e Lins, R. D. (2005) Separation of Concerns for Improving Practice of Parallel Programming. *Journal of Information and Computation*, International Information Institute, v. 8, n. 5, Tokio, Japan.
- Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K. e Torczon, L. (2003), A. *Sourcebook of Parallel Computing*. Morgan Kauffman Publishers.
- Foster, I. e Kesselman, C. (2004) *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kauffman Publishers.
- Geist, G. A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. e Sunderam, V. S. (1994) PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge.
- Gorlatch, S. (2004), Send-Recv Considered Harmful? Myths and Truths about Parallel Programming, *ACM Transactions in Programming Languages and Systems*, vol. 26, n. 1, pp. 47-56, ACM Press.
- Grant, F. (2007) Getting Fisical with Phynance. *Scientific Computing World*, issue 91, pp. 14-26, Warren Clark (editor). <http://www.scientific-computing.com>.

- Gustafson, J. (1988) Reevaluating Amdahl's Law. Communications of the ACM, vol. 31, n. 5, ACM Press.
- Lins, R. D. (1996) Functional Programming and Parallel Processing. Lecture Notes in Computer Science, vol. 1215, pp. 429-457 (Proceedings of the 2nd International Conference on Vector and Parallel Processing – VECPAR'96)
- MPI-Forum. (1994) MPI: A Message-Passing Interface Standard. International Journal of Supercomputer Applications and High Performance Computing.
- OpenMP Architecture Review Board. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. Relatório Técnico, 1997.
- Post, D. E. e Votta, L. G. (2005), Computational Science Demands a New Paradigm, Physics Today, vol.58, n. 1, 35-41.
- Vogels, W. (2003) HPC.NET – are CLI-Based Virtual Machines Suitable for High Performance Computing. Conference on High Performance Networkin and Computing, Proceedings of the 2003 ACM/IEEE Conference on Supercomputing.
- Robson, D. (2007) Maths Maps Tomorrow's Drugs. Scientific Computing World, issue 92, pp. 24-25, Warren Clark (editor). <http://www.scientific-computing.com>.
- Spencer Jr, B. et al (2004) NEESGrid: A Distributed Collaboration for Advanced Earthquake Engineering Experiment and Simulation. 13th World Conference on Earthquake Engineering, Vancouver, Canadá, Aug. 1-6, paper #1674.
- van der Steen, A. J. (2006) Issues in Computational Frameworks. Concurrency and Computation: Practice and Experience, vol. 18, n. 2, pp. 141-150. Wiley.
- Shaya, E. e Thomas, B. (2001) Specifics on a XML Data Format for Scientific Data. Astronomical Data Analysis Software and Systems X, ASP Conference Series, v. 238, pp. 217-220.
- Skillicorn, D. B. e Talia, D. (1998) Models and Languages for Parallel Computation. ACM Computing Surveys, 30(2), pp. 123-169, ACM Press.
- Throop, J. (1999) OpenMP: Shared-Memory Paralellism From the Ashes. Computer, vol 32, n. 5, pp. 108-109, IEEE Computer Society.
- Becker, D. J., Sterling, T., Savarese, D., Dorban, J. E. e Ranawak, U. A.. (1995) Beowulf: A Parallel Workstation for Scientific Computation. Proceedings of the 1995 International Conference on Parallel Processing.
- Anderson, T. E., Culler, D. E. e Patterson, D. A. (1994) A Case for Networks of Workstations. IEEE Euromicro, 15(1), pp. 54-64.

APÊNDICE A. Código Fonte – Ordenação de Inteiros com MPI e OpenMP

O programa abaixo implementa uma ordenação de inteiros utilizando uma versão paralela do algoritmo *bucketsort*, como explicado na Seção 6 deste capítulo. Foi adaptado a partir das versões MPI e OpenMP do benchmark IS (*Integer Sorting*) do NAS Parallel Benchmarks [Bailey 1991], o qual é composto por um conjunto de benchmarks e aplicações simuladas desenvolvidos pelo Centro de Pesquisa de Ames da NASA para avaliar o desempenho de arquiteturas paralelas para aplicações de dinâmica dos fluidos (<http://www.nas.nasa.gov/Software/NPB/>).

```

1. #include <omp.h>
2. #include <mpi.h>
3. #include <stdio.h>
4.
5. #include "create_seq.h"
6.
7. #define NUM_PROCS 2
8.
9. /*****
10. /* CLASS B */
11. *****/
12. #define TOTAL_KEYS_LOG_2 25
13. #define MAX_KEY_LOG_2 21
14. #define NUM_BUCKETS_LOG_2 10
15.
16. #define TOTAL_KEYS (1 << TOTAL_KEYS_LOG_2)
17. #define MAX_KEY (1 << MAX_KEY_LOG_2)
18. #define NUM_BUCKETS (1 << NUM_BUCKETS_LOG_2)
19. #define NUM_KEYS (TOTAL_KEYS/NUM_PROCS)
20.
21. #if NUM_PROCS < 256
22. #define SIZE_OF_BUFFERS 3*NUM_KEYS/2
23. #else
24. #define SIZE_OF_BUFFERS 6*NUM_KEYS
25. #endif
26.
27. #define MAX_PROCS 512
28.
29. typedef int INT_TYPE;
30.
31. INT_TYPE key_array[SIZE_OF_BUFFERS],
32.     key_buff1[SIZE_OF_BUFFERS],
33.     key_buff2[SIZE_OF_BUFFERS];
34. bucket_size_t[NUM_BUCKETS],
35. bucket_size_totals[NUM_BUCKETS],
36. bucket_ptr[NUM_BUCKETS];
37. process_bucket_distrib_ptr1[NUM_BUCKETS],
38. process_bucket_distrib_ptr2[NUM_BUCKETS],
39. send_count[MAX_PROCS], recv_count[MAX_PROCS],
40. send_displ[MAX_PROCS], recv_displ[MAX_PROCS],
41. **key_buff0;
42.
43. INT_TYPE total_local_keys=0,
44. total_lesser_keys=0;
45.
46. int my_rank;
47. int comm_size;

```



```

48. void main(int* argc, char*** argv) {
49.
50.     MPI_Init(argc, argv);
51.
52.     MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
53.     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
54.
55.     create_seq(find_my_seed(my_rank, comm_size,
56.                             4*TOTAL_KEYS, 1220703125.00 ),
57.                             1220703125.00, MAX_KEY, NUM_KEYS, key_array
58. );
59.
60.     alloc_key_buff();
61.
62.     MPI_Finalize();
63.
64. }

```



```

65. void rank()
66. {
67.     INT_TYPE i, j, k;
68.     INT_TYPE shift = MAX_KEY_LOG_2 - NUM_BUCKETS_LOG_2;
69.     INT_TYPE key, key_bucket, min_key_val, max_key_val;
70.     INT_TYPE bucket_sum_accumul, local_bucket_sum_accumulator;
71.     INT_TYPE *key_buff_ptr;
72.
73.     key_buff_ptr = key_buff0[0];
74.
75.     #pragma omp parallel private(i, k)
76.     {
77.         INT_TYPE bucket_size_local[NUM_BUCKETS], *work_buff;
78.         int myid = 0, num_procs = 1;
79.
80.         myid = omp_get_thread_num();
81.         num_procs = omp_get_num_threads();
82.
83.         work_buff = key_buff0[myid];
84.
85.         for( i=0; i<NUM_BUCKETS; i++ ) {
86.             work_buff[i] = 0;
87.             bucket_size_local[i] = 0;
88.         }
89.
90.         #pragma omp for nowait
91.         for( i=0; i<NUM_BUCKETS; i++ ) {
92.             process_bucket_distrib_ptr1[i] = 0;
93.             process_bucket_distrib_ptr2[i] = 0;
94.         }
95.
96.         #pragma omp atomic
97.         bucket_size_local[key_array[i] >> shift]++;
98.
99.         work_buff[0] = 0;
100.        for( i=0; i<NUM_BUCKETS-1; i++ ) {
101.            work_buff[i+1] = work_buff[i] + bucket_size_local[i];
102.            bucket_size[i] += bucket_size_local[i];
103.        }

```



```

104.        bucket_size[NUM_BUCKETS-1] += bucket_size_local[NUM_BUCKETS-1];
105.
106.        #pragma omp barrier
107.
108.        for(k=1; k<num_procs; k++ ) {
109.            #pragma omp for nowait
110.            for( i=0; i<NUM_BUCKETS; i++ )
111.                key_buff_ptr[i] += key_buff0[k][i];
112.        }
113.
114.        #pragma omp barrier
115.
116.        #pragma omp single
117.        for (i=0; i<NUM_KEYS; i++) {
118.            key = key_array[i];
119.            key_bucket = key_buff_ptr[key >> shift]++;
120.            key_buff1[key_bucket] = key;
121.        }
122.
123.        #pragma omp single // communication section
124.        {
125.            MPI_Allreduce( bucket_size, bucket_size_totals,
126.                           NUM_BUCKETS, MPI_INT, MPI_SUM, MPI_COMM_WORLD );
127.            bucket_sum_accumulator = 0;
128.            local_bucket_sum_accumulator = 0;
129.            send_displ[0] = 0;
130.            process_bucket_distrib_ptr1[0] = 0;
131.            for( i=0, j=0; i<NUM_BUCKETS; i++ )
132.            {
133.                bucket_sum_accumulator += bucket_size_totals[i];
134.                local_bucket_sum_accumulator += bucket_size[i];
135.
136.                if( bucket_sum_accumulator >= (j+1)*NUM_KEYS ) {
137.                    send_count[j] = local_bucket_sum_accumulator;
138.                    if( j != 0 ) {
139.                        send_displ[j] = send_displ[j-1] + send_count[j-1];
140.                        process_bucket_distrib_ptr1[j] =
141.                            process_bucket_distrib_ptr2[j-1]+1;
142.                    }
143.                    process_bucket_distrib_ptr2[j++] = i;
144.                    local_bucket_sum_accumulator = 0;
145.                }
146.
147.                MPI_Alltoall( send_count, 1, MPI_INT,
148.                               recv_count, 1, MPI_INT, MPI_COMM_WORLD );
149.
150.                recv_displ[0] = 0;
151.                for(i=1; i<comm_size; i++)
152.                    recv_displ[i] = recv_displ[i-1] + recv_count[i-1];
153.
154.                MPI_Alltoallv(key_buff1, send_count, send_displ, MPI_INT,
155.                               key_buff2, recv_count, recv_displ, MPI_INT,
156.                               MPI_COMM_WORLD );
157.                min_key_val = process_bucket_distrib_ptr1[my_rank] << shift;
158.                max_key_val = ((process_bucket_distrib_ptr2[my_rank] + 1)
159.                               << shift)-1;
160.
161.            #pragma omp for reduction(:total_lesser_keys)
162.            for( k=0; k<my_rank; k++ )
163.                for( i = process_bucket_distrib_ptr1[k];
164.                     i<process_bucket_distrib_ptr2[k]; i++ )
165.                    total_lesser_keys += bucket_size_totals[i];
166.
167.            #pragma omp for reduction(:total_local_keys)
168.            for( i = process_bucket_distrib_ptr1[my_rank];
169.                 i<process_bucket_distrib_ptr2[my_rank]; i++ )
170.                 total_local_keys += bucket_size_totals[i];
171.
172.            work_buff -= min_key_val;
173.            key_buff0[myid] -= min_key_val;
174.
175.            #pragma omp single
176.            key_buff_ptr -= min_key_val;
177.
178.            #pragma omp for
179.            for( i=0; i<total_local_keys; i++ ) {
180.                #pragma omp atomic
181.                work_buff[key_buff2[i]]++;
182.
183.                for(i=min_key_val; i<max_key_val-1; i++)
184.                    work_buff[i+1] += work_buff[i];
185.
186.            #pragma omp barrier
187.
188.            for( k=1; k<num_procs; k++ ) {
189.                #pragma omp for nowait
190.                for( i=min_key_val; i<max_key_val; i++ )
191.                    key_buff_ptr[i] += key_buff0[k][i];
192.
193.                #pragma omp for
194.                for( k=min_key_val; k<max_key_val; k++ ) {
195.                    i = (k==0)? 0 : key_buff_ptr[k-1];
196.                    while (i<key_buff_ptr[k]) key_array[i++] = k;
197.                }
198.
199.            } // END OMP PARALLEL
200.
201.        } // END rank()

```


Capítulo

10

XML: Conceitos e Aplicações

Bernadette Farias Lóscio e Fernando Cordeiro Lemos

Abstract

The Extensible Markup Language (XML) is a general-purpose markup language that allows its users to define their own tags. Its primary purpose is to facilitate the sharing of data across different information systems, particularly via the Web. XML is a simplified subset of the Standard Generalized Markup Language (SGML), and is designed to be relatively human-legible. XML is recommended by the World Wide Web Consortium (W3C) and is an open standard.

Resumo

XML (Extensible Markup Language) é uma linguagem de propósito geral que permite aos seus usuários definir tags próprias. Seu principal objetivo é facilitar a troca de dados entre diferentes sistemas de informação, particularmente através da Web. XML é um subconjunto simplificado do SGML (Standard Generalized Markup Language), tendo sido projetada para ser relativamente de fácil leitura. XML é um padrão aberto recomendado pelo W3C (World Wide Web Consortium).

10.1. Introdução

A Web é um dos meios mais utilizados para a disseminação de informações e abrange diversas aplicações, como: bibliotecas digitais, museus virtuais, catálogos de serviços e produtos, informações governamentais, entre outros. A Web facilita o acesso aos dados provenientes de diversas áreas, que incluem não apenas centros acadêmicos, mas também empresas públicas e privadas, instituições militares e governamentais. A facilidade de acesso aos dados e o grande volume de informações disponível na Web revolucionaram o desenvolvimento dos sistemas de informação e motivaram o crescimento de aplicações que freqüentemente necessitam integrar dados heterogêneos e distribuídos em diferentes fontes de dados.

A Web pode ser vista como um enorme Banco de Dados (BD) onde documentos são disponibilizados para leitura, sendo alguns destes documentos gerados a partir de

consultas a um BD. Comparando as tecnologias Web e BD, constata-se que a Web proporciona uma infra-estrutura global e um conjunto de padrões para dar suporte à troca de documentos, um formato de apresentação para hipertexto (HTML), interfaces com o usuário bem construídas para recuperação de documentos e um formato (XML) para troca de dados.

A linguagem XML (*Extensible Markup Language*) foi proposta pelo W3C (*World Wide Web Consortium*) como um padrão para representação e troca de dados na Web [Bray 1998]. Uma das aplicações beneficiadas com a utilização de XML é a integração de dados, pois XML fornece uma plataforma para compartilhamento de dados através de uma sintaxe comum. Devido à flexibilidade de XML para representar dados semi-estruturados e dados estruturados, e à facilidade para converter dados para o formato de XML, existe um grande interesse em usar XML como modelo de dados comum para representação e troca de dados.

XML foi proposta inicialmente como um formato comum para representação e troca de dados na Web. Entretanto, atualmente, XML tem sido amplamente utilizada em diversos contextos diferentes. Por ser uma linguagem de marcação extensível, i.e., que permite a definição de novos marcadores (tags) de acordo com o domínio que está sendo modelado, XML deu origem diversas linguagens, como: XML Schema, RDF e RSS.

Neste capítulo apresentamos a linguagem XML destacando seus conceitos básicos e aplicações. Na seção 10.2, uma visão geral sobre XML é apresentada. Na seção 10.3 é apresentada a sintaxe básica para construção de documentos XML. Na seção 10.4 são apresentadas as linguagens de definição de esquemas para XML, com destaque para XML Schema. Na seção 10.5 são discutidos alguns modelos de dados para XML. Na seção 10.6 algumas linguagens de consulta para XML são apresentadas. Na seção 10.7 é apresentada a linguagem SQL/XML, que tem sido bastante utilizada para a geração de visões XML a partir de bancos de dados relacionais. Na seção 10.8 apresentamos algumas considerações finais.

10.2. Visão Geral

XML é uma linguagem de marcadores (*markup language*) para descrever informações. Um marcador é uma forma de tornar explícita a interpretação de um texto e uma linguagem de marcadores é um conjunto de convenções de marcadores utilizados para codificação de textos.

Ao contrário de HTML, que foi projetada exclusivamente para descrever a apresentação dos dados, XML foi projetada para descrever o conteúdo dos dados. A linguagem XML é mais poderosa do que HTML nos seguintes aspectos:

- XML é uma linguagem extensível, ou seja, permite que os usuários definam novos marcadores de acordo com o domínio que está sendo modelado;
- A estrutura dos documentos pode ser aninhada em qualquer nível;
- Um documento XML pode conter uma descrição opcional da sua gramática, a qual pode ser usada pelas aplicações para validações na estrutura dos documentos;
- XML é uma linguagem flexível, pois permite a apresentação de um mesmo conteúdo em diferentes formatos.

Por ser uma linguagem extensível, XML também pode ser vista como uma metalinguagem, ou seja, uma linguagem que pode ser usada para criação de outras linguagens de marcadores. Estas linguagens de marcadores podem ser criadas para domínios específicos (ex: matemática, química, comércio eletrônico, entre outros) através da definição de marcadores próprios de cada domínio. Por exemplo, quando o domínio de uma livraria é considerado, os seguintes marcadores podem ser definidos: <livro>, <autor> e <editora>. A idéia é que os marcadores possam agregar algum significado ao texto associado a eles e, assim, capturar mais semântica sobre os dados que estão sendo modelados. Isto não ocorre em um documento HTML, pois os marcadores definidos em HTML têm função apenas de formatar o texto para apresentação. Considere, por exemplo, o documento HTML e o documento XML apresentados na Figura 10.1. Qual destes documentos captura mais semântica? Claramente, o documento XML captura mais semântica, pois os marcadores <livro>, <título> e <autor> têm significado para o domínio que está sendo representado através destes documentos, ao contrário dos marcadores <TR>, <TH> e <TD> que não têm significado algum para o domínio.

```
<----HTML ---->
<TABLE BORDER=1>
  <TR>
    <TH>Livro ID</TH>
    <TH>Titulo</TH>
    <TH>Autor</TH>
  </TR>
  <TR>
    <TD>L01</TD>
    <TD> Foundations of
        Data Bases </TD>
    <TD> Abiteboul </TD>
  </TR>
</TABLE>
```

```
<----XML ---->
<livro>
  <id>L01</id>
  <título> Foundations of
          Data Bases </título>
  <autor> Abiteboul </autor>
</livro>
```

Fig. 10.1. Exemplo de um documento HTML e um documento XML com dados sobre uma livraria

XML é um subconjunto de SGML (*Standard Generalized Markup Language*), um padrão internacional para definição de formatos de representação de textos em meio eletrônico. SGML é um padrão muito poderoso e bastante geral, o que torna complicada a sua implementação e restringe sua utilização a grandes empresas. Como mostrado na Figura 10.2 [McGrath 1998], SGML e XML são metalinguagens utilizadas para definição de outras linguagens de marcadores aplicadas a domínios bastante distintos.

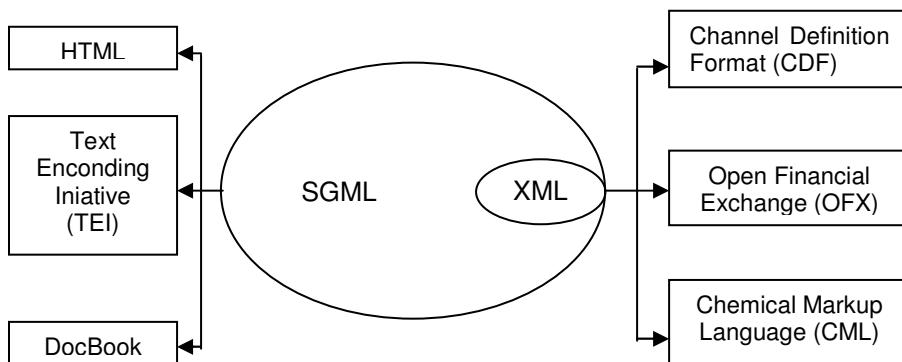


Figura 10.2. Relacionamento entre XML e outras linguagens de marcadores

Uma das idéias principais de XML é tornar explícita a separação entre os componentes de um documento eletrônico: apresentação, conteúdo e estrutura. Um documento XML descreve as informações de acordo com o domínio que está sendo modelado, sem levar em consideração como estas informações serão apresentadas em um *browser*. Sendo assim, documentos XML podem ser usados e reusados de diferentes formas e em diferentes formatos. Como o padrão XML trata apenas do conteúdo dos documentos, devem existir outros padrões para descrever o formato de apresentação dos dados. Atualmente, existem duas propostas do W3C para criação de estilos de apresentação para documentos XML: *Cascading Style Sheets* - CSS e *Extensible Style Language* - XSL.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE livraria SYSTEM "livraria.dtd">
<livraria>
    <livro id="L01" ano="1997">
        <autor>
            <nome>Marie</nome>
            <sobrenome>Buretta</sobrenome>
        </autor>
        <titulo>Data Replication</titulo>
        <editora>John Wiley & Sons </editora>
    </livro>
    <livro id="L02" ano="1996" bib="L01">
        <autor>
            <nome>Serge </nome>
            <sobrenome>Abiteboul</sobrenome>
        </autor>
        <autor>
            <nome>Richard</nome>
            <sobrenome>Hull</sobrenome>
        </autor>
        <autor>
            <nome>Victor</nome>
            <sobrenome>Vianu</sobrenome>
        </autor>
        <titulo>Foundations of Data Bases</titulo>
        <editora>Addison Wesley</editora>
    </livro>
</livraria>
```

Figura 10.3. Exemplo de documento XML (livraria.xml)

Os dados XML são auto-descritíveis. A Figura 10.3 apresenta um exemplo de documento XML que descreve informações sobre livros de uma livraria. Um

documento XML consiste de vários elementos. Um elemento pode conter apenas texto, outros elementos (subelementos) ou então pode ter um conteúdo misto, ou seja, pode conter texto e outros elementos. Por exemplo, `livro` é um subelemento do elemento `livraria`, enquanto que `autor` e `titulo` são subelementos do elemento `livro`. Um elemento também pode ter atributos, os quais sempre são especificados no marcador que define o início do elemento. Por exemplo, `ano` é um atributo do elemento `livro` e define o ano de publicação de um determinado livro.

10.2.1. Categorias de Documentos XML

Os documentos XML são classificados como bem-formados quando estão de acordo com a sintaxe definida na especificação de XML [Bray 1998]. Os documentos XML também podem ser classificados como documentos válidos. Isto ocorre quando o documento XML é bem-formado e está de acordo com a gramática que define a sua estrutura. Esta gramática é definida em um esquema, o qual descreve: a estrutura de um documento XML, os elementos que podem participar do documento e os atributos que podem estar associados a estes elementos. Atualmente, existem várias propostas de linguagens de descrição de esquemas para documentos XML, entre elas destacam-se: DTD [Bray 1998] e XML Schema [Biron 2000], [Thompson 2000].

A verificação sintática de um documento XML, a fim de detectar se este é bem-formado ou não, é feita por um *parser*, que analisa o documento e envia mensagens de erro quando encontra erros de sintaxe. Os *parsers* podem ser classificados em duas categorias: i) *parser* de validação: além de detectar se um documento XML é bem-formado, também é capaz de verificar se o documento está de acordo com o esquema XML ao qual o documento está associado e ii) *parser* de não validação: não é capaz de verificar se um documento bem-formado está de acordo com o esquema ao qual o documento está associado.

10.2.2. APIs para XML

DOM (*Document Object Model*) e SAX (*The Simple API for XML*) são APIs (*Application Programming Interface*) para XML que oferecem meios para acessar e manipular o conteúdo de um documento XML. As aplicações podem utilizar as operações disponíveis na API para acessar o conteúdo de um documento XML. Estas duas APIs oferecem diferentes visões de um mesmo documento: DOM oferece uma visão baseada em árvore e SAX oferece uma visão baseada em eventos. É importante observar que DOM é a API recomendada oficialmente pelo W3C.

Tanto DOM quanto SAX são APIs independentes de linguagem e plataforma que permitem programas e *scripts* acessarem e atualizarem o conteúdo e a estrutura de um documento dinamicamente.

DOM provê um conjunto de interfaces e objetos para representar a estrutura e o conteúdo de documentos XML e HTML. Esta API representa um documento como uma coleção de objetos (*nodes*), criando uma árvore de nós baseada na estrutura e na informação do documento XML.

SAX também é uma interface que permite a interação com documentos XML e apresenta um documento XML como uma seqüência de eventos. Esta API não permite acessos randômicos na manipulação do documento, sendo, portanto, mais adequada quando o processamento do documento é seqüencial.

10.2.3. Bancos de Dados e XML

De acordo com a comunidade XML:DB¹, três tipos diferentes de bancos de dados para XML podem ser definidos:

- **Banco de Dados XML Nativo** (*Native XML Database - NXD*): é um banco de dados projetado especificamente para armazenar e manipular dados XML. O acesso aos dados é através de XML e padrões relacionados, como XSLT, DOM ou SAX. Esta categoria inclui todos os bancos de dados onde a representação dos dados mantém a estrutura do documentos XML, bem como os metadados associados. A unidade fundamental de armazenamento é um documento XML. Tamino, dbXML e X-Hive são classificados nesta categoria.
- **Banco de Dados compatível com XML** (*XML Enabled Database - XEDB*): é um banco de dados que tem como funcionalidade adicional um nível de mapeamento para XML, que pode ser fornecido pelo próprio fabricante ou por terceiros. Este nível de mapeamento gerencia o armazenamento e a recuperação de dados XML. Os dados que são mapeados para o banco de dados são mapeados para formatos específicos da aplicação e, neste caso, a estrutura original do documento XML e os seus metadados podem ser perdidos. Além disso, os dados recuperados como XML não têm garantia de terem sido gerados no formato original de XML. A manipulação de dados pode ocorrer através de tecnologia específica para XML (i.e. XSLT, DOM ou SAX) ou outras tecnologias de banco de dados. (i.e. SQL). As soluções para XML propostas pela Oracle e Microsoft fazem parte desta categoria.
- **Banco de Dados XML Híbrido** (*Hybrid XML Database - HXD*): é um banco de dados que pode ser tratado ou como banco de dados nativo ou como um banco de dados compatível com XML, dependendo dos requisitos da aplicação. Exemplos de bancos de dados desta categoria são Excelon e Ozone.

10. 3. Construindo Documentos XML

Os elementos são os blocos principais da estrutura hierárquica de XML. Como mencionado anteriormente, o conteúdo de um elemento pode ser composto de: outros elementos, caracteres, ou outros elementos e caracteres. Os elementos são delimitados por marcadores tais como `<autor>` e `</autor>` apresentados no exemplo da Figura 10.3. Cada marcador de início (`<livro>`) deve ter um marcador de final correspondente (`</livro>`). Um documento XML também pode ter elementos com conteúdo vazio. Neste caso, o elemento deve ser representado de acordo com um dos seguintes formatos: `<referencia></referencia>` ou `<referencia/>`. Elementos vazios são úteis quando se deseja descrever um elemento apenas através de atributos ou para referenciar outros elementos.

Um elemento pode possuir um ou mais atributos que definem as propriedades dos elementos e adicionam novas características a eles. Os atributos são especificados no marcador de início do elemento ou no marcador que define um elemento vazio. Por exemplo, no documento “livraria.xml”, `ano` é um atributo do elemento `livro`. É importante lembrar que os valores de atributos devem ser delimitados por aspas (“ou ‘”).

¹ <http://www.xmldb.org/>

De forma geral, os elementos definem a estrutura de entidades do mundo real enquanto que os atributos definem propriedades sobre estas entidades. Como esta distinção nem sempre é clara, uma das dúvidas mais freqüentes quando se constrói um documento XML consiste em determinar se uma informação deve ser modelada como um elemento ou como um atributo. Entretanto, existem, algumas diretrizes que podem ser úteis no momento desta escolha. Quando a informação possui alguma estrutura hierárquica então esta informação deve ser modelada como um elemento, pois não é possível estabelecer uma hierarquia entre atributos. A ordem em que os atributos aparecem na definição de um elemento é irrelevante (`ano` e `id` em `livro` na 10.3.). Por outro lado, a ordem em que os elementos aparecem na definição de um elemento sempre deve ser obedecida (`nome` e `sobrenome` em `autor` na 10.3.). Outra diferença importante é que um atributo só pode ser definido uma única vez para cada elemento, enquanto que um subelemento pode se repetir várias vezes na definição de um elemento. Por exemplo, um livro pode ter vários autores, mas só pode ter um valor para `ano` de publicação.

A primeira informação de um documento XML é chamada de prólogo (*prolog*), o qual inclui a declaração XML e a declaração de tipo de documento. A declaração XML determina a versão de XML utilizada para a definição do documento. A declaração de tipo de documento especifica a DTD a qual o documento está associado. Após o prólogo, inicia-se o elemento raiz o qual contém todos os outros elementos do documento. Uma das regras básicas para criar um documento XML bem-formado é que deve existir apenas um único elemento raiz em um documento XML. O elemento raiz do documento “`livraria.xml`” é o elemento `livraria`. Além de outros elementos, o elemento raiz também pode conter:

- **Seções CDATA:** devem ser usadas em documentos XML que contêm grande número de caracteres especiais (ex: “<” e “&”). As seções CDATA são úteis porque quando um caractere especial se encontra dentro dos limites de uma seção CDATA ele passa a ser interpretado como um simples caractere. Considere, por exemplo a seção CDATA definida a seguir. Neste caso, o caractere “<” pode ser usado livremente sem ocasionar erros de sintaxe, pois este caractere está inserido dentro de uma seção CDATA.

```
<Documento>
<! [CDATA [
se a<b e b<c então a<c ]]>
</Documento>
```

- **Comentários:** a definição de comentários é bastante útil e faz parte da maioria das linguagens. Em XML, os comentários são definidos como mostrado a seguir:

```
<!-- Exemplo de comentário -->
```

- **Instruções de processamento:** são utilizadas para enviar comandos e informações à aplicação que está processando o documento XML. Um exemplo de instrução de processamento é a declaração de XML (`<?xml version="1.0" encoding="utf-8"?>`).

Todo o conteúdo que vier após o elemento raiz é denominado de epílogo, o qual pode conter apenas comentários e instruções de processamento.

Para facilitar o desenvolvimento de aplicações em XML, existem várias ferramentas disponíveis, incluindo: editores para criação e modificação de documentos XML, ferramentas para criação e modificação de esquemas e folhas de estilo, e ferramentas que oferecem suporte para o gerenciamento e armazenamento de documentos XML.

10.4. Esquemas XML

Um documento XML pode, opcionalmente, estar associado a um esquema. Um esquema define os elementos que podem aparecer em um documento, o conteúdo destes elementos e os atributos que podem estar associados a eles. Um esquema também define a estrutura de um documento, por exemplo: quais os subelementos de um determinado elemento e a seqüência na qual estes subelementos podem aparecer.

Um esquema é útil para validar o conteúdo de um documento, ou seja, para determinar se um documento é válido de acordo com a gramática definida pelo esquema. Além disso, a gramática definida pelo esquema poderá ser reutilizada para a definição de outros documentos.

A habilidade de testar a validade de documentos XML é muito importante para aplicações Web que trocam informações entre fontes diversas. Com a definição de esquemas XML, é possível verificar se um determinado documento está de acordo com a estrutura esperada, facilitando o processamento de dados pelas aplicações. Além disso, diferentes fontes de dados, relacionadas a um mesmo domínio (ex: comércio eletrônico), podem definir uma gramática comum para ser utilizada como base para criação de documentos. Assim, a integração de dados pode tirar vantagem da gramática comum que foi utilizada para criar os documentos.

A primeira linguagem proposta para definição de esquemas XML foi a DTD, que apesar de suas limitações, atualmente é a forma mais utilizada para especificação de esquemas de documentos XML. Para sobrepor as limitações das DTDs, recentemente, várias linguagens para definição de esquemas XML foram propostas [Lee 2000], dentre elas se destacam: XML Schema [Biron 2000], [Thompson 2000], XDR [Frankston 1998], SOX [Davidson 1999], Schematron [Jellife 2000] e DSD [Klarlund 2000]. Estas linguagens são mais ricas em semântica e oferecem alguns recursos adicionais para definição de esquemas, como por exemplo: maior variedade de tipos primitivos, definição de novos tipos e hierarquia. Estas linguagens também oferecem a vantagem de seguir a sintaxe de XML, ou seja são linguagens baseadas em XML. Desta forma, é possível tirar proveito de toda a tecnologia que já foi desenvolvida para XML. Para ilustrar as principais características dos esquemas XML, nas próximas seções serão apresentadas as linguagens DTD e XML Schema.

10.4.1. DTD - Document Type Definition

Um método para descrever esquemas para documentos XML corresponde à definição de uma DTD (*Document Type Definition*), que define a seqüência dos elementos no documento, o conteúdo dos elementos e os atributos associados a cada elemento.

A declaração de tipo de documento que aparece no prólogo de um documento XML especifica a DTD a qual o documento está associado. Por exemplo, a declaração de tipo de documento `<!DOCTYPE livraria SYSTEM "livraria.dtd">` (Figura 10.3.) define que o documento “livraria.xml” está associado a uma DTD armazenada

em um arquivo externo chamado “livraria.dtd”. É importante notar que a declaração de tipo de documento sempre deve referenciar o arquivo raiz, ou seja, sempre após a expressão !DOCTYPE deve-se colocar o nome do elemento raiz do documento.

As DTDs podem ser internas ou externas a um documento XML, ou seja, inserida ou não dentro do próprio documento XML. Considere, por exemplo, a definição da Figura 10.4 (a), onde as declarações de elemento fazem parte da cláusula !DOCTYPE do documento XML. A Figura 10.4 (b) apresenta um exemplo onde é feita uma referência a uma DTD externa que está armazenada no arquivo “livraria.dtd”.

Uma DTD é composta de vários tipos de declaração, entre elas destacam-se: declarações de elemento, declarações de atributo e declarações de entidade, as quais serão explicadas a seguir.

```
<-- DTD Interna -->
<!DOCTYPE livraria [
<!ELEMENT livro (titulo, autor)>
<!ELEMENT titulo (#PCDATA)>
<!ELEMENT autor (#PCDATA)>]>
```

(a)

```
<-- DTD Externa -->
<!DOCTYPE livraria SYSTEM
"livraria.dtd">
```

(b)

Figura 10.4. Exemplos de declaração de tipo de documento

- **Declaração de tipo de elemento:** permite a um documento XML restringir os elementos que ocorrem no documento. Todos os elementos utilizados em um documento XML devem ter uma declaração de elemento correspondente. Esta declaração especifica o conteúdo de um elemento, o qual pode ser dos seguintes tipos:

- *Seqüência de elementos:* especifica que um elemento consiste de outros elementos exatamente na ordem em que for especificada a seqüência.

Exemplo: <!ELEMENT artigo (autor, titulo)>.

- *Seleção a partir de uma lista de elementos:* especifica que apenas um dos elementos definidos na declaração pode fazer parte do conteúdo do elemento que está sendo declarado.

Exemplo: <!ELEMENT publicacao (livro|artigo)>.

- *A ocorrência de um elemento é opcional (?)*: especifica que a participação de um elemento é opcional e caso exista deve ser única.

Exemplo: <!ELEMENT livro (titulo, editora?)>.

- *Um elemento ocorre zero ou mais vezes (*)*: especifica que um elemento pode participar zero ou mais vezes do elemento que está sendo declarado.

Exemplo: <!ELEMENT livros (livro*)>.

- *Um elemento ocorre uma ou mais vezes (+)*: especifica que um elemento deve participar pelo menos uma vez do elemento que está sendo declarado, podendo também se repetir.

Exemplo: <!ELEMENT livro (autor+, titulo, editora)>.

- *Um elemento contém qualquer outro elemento em qualquer ordem:* especifica que um elemento consiste de qualquer combinação de elementos em qualquer ordem. O elemento também pode conter caracteres ou o elemento pode conter outros elementos e caracteres em qualquer ordem.

Exemplo: `<!ELEMENT livro ANY>`.

- *Um elemento também pode conter uma string de caracteres e pode ter conteúdo misto (caracteres e elementos).*

Exemplos:

```
<!ELEMENT titulo (#PCDATA)>.  
<!ELEMENT livro (#PCDATA, autor*)>.
```

- **Declaração de atributo:** especifica o nome, o tipo e, opcionalmente, o valor padrão dos atributos associados a um elemento. Os atributos podem ser dos seguintes tipos:

- *String:* o valor de um atributo do tipo *string* é uma cadeia de caracteres de qualquer tamanho.

Exemplo: `<!ATTLIST livro ano CDATA>`.

- *Enumerado:* cada um dos valores possíveis que o atributo pode assumir está explicitamente enumerado na declaração. O atributo pode assumir apenas um dos valores especificados na sua declaração.

Exemplo: `<!ATTLIST livro categoria (Banco de Dados|Web)>`.

- *ID:* os atributos do tipo ID identificam unicamente elementos individuais em um documento. Todos os valores usados para os atributos do tipo ID em um documento devem ser diferentes. Cada elemento pode ter um único atributo ID.

Exemplo: `<!ATTLIST livro id ID>`.

- *IDREF/ IDREFS:* o valor de um atributo do tipo IDREF deve ser o valor de um único atributo ID definido para algum elemento do documento. O atributo IDREFS é uma variação do tipo IDREF. O valor de um atributo IDREFS pode conter valores IDREF múltiplos separados por espaços em branco.

Exemplo: `<!ATTLIST livro bib IDREFS>`.

- *ENTITY/ ENTITIES:* o valor de um atributo ENTITY deve ser o nome de uma única entidade (o conceito de entidade será explicado logo a seguir). O valor de um atributo ENTITIES pode conter valores de entidades múltiplos separados por espaços em branco.

Exemplo: `<!ATTLIST livro resumo ENTITY>`.

O valor padrão de um atributo associado a um elemento pode ser:

- *Obrigatório (#REQUIRED):* o atributo deve ter um valor explicitamente especificado em cada ocorrência do elemento no documento.

Exemplo: `<!ATTLIST livro preco CDATA #REQUIRED>`.

- *Opcional (#IMPLIED):* o valor do atributo não é requerido e nenhum valor padrão é fornecido.

Exemplo: <!ATTLIST livro ano CDATA #IMPLIED>.

- *Fixo (#FIXED)*: um valor é fornecido na declaração. Sendo assim, nenhum valor precisa ser fornecido no documento. Entretanto, caso um valor seja fornecido no documento então ele deve corresponder ao valor fornecido na declaração.

Exemplo: <!ATTLIST livro qtdMinima CDATA #FIXED "15">.

A Figura 10.5 apresenta a DTD que define a gramática para o documento XML “livraria.xml” (Figura 10.3). Esta DTD especifica que um elemento `livraria` consiste em vários elementos `livro`. Um elemento `livro`, por sua vez, contém um ou mais elementos `autor`, seguidos de um elemento `titulo` e de um elemento `editora`. Além desses elementos, `livro` também possui um atributo opcional `ano`, um atributo `id` e um atributo `bib`, que armazena referências a outros `livros`. Um elemento `autor` consiste em um elemento `nome` e um elemento `sobrenome`, os quais contêm `strings`, assim como os elementos `titulo` e `editora` também contêm apenas `strings`.

```
<!ELEMENT livraria (livro*)>
<!ELEMENT livro (autor+, titulo, editora)>
<!ATTLIST livro ano CDATA #IMPLIED
           id ID
           bib IDREF>
<!ELEMENT autor (nome, sobrenome)>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT sobrenome (#PCDATA)>
<!ELEMENT titulo (#PCDATA)>
<!ELEMENT editora (#PCDATA)>
```

Figura 10.5. Exemplo de DTD (livraria.dtd)

10.4.2. XML Schema

XML Schema é uma proposta da W3C para descrever esquemas de documentos XML. Esta linguagem tem a vantagem de utilizar a mesma sintaxe de XML, além de oferecer um conjunto maior de tipos de dados, bem como suportar a criação de novos tipos de dados. Nesta seção, algumas características necessárias para definição de esquemas XML são apresentadas. Maiores detalhes sobre a linguagem XML Schema podem ser encontrados em [Biron 2000], [Thompson 2000].

• Namespaces

Uma das principais diferenças entre uma DTD e um esquema definido em XML Schema está relacionada ao uso de *namespaces* XML. *Namespaces* [Bray 1999] também é um padrão definido pelo W3C e pode ser visto como uma maneira simples e direta de distinguir nomes usados em documentos XML. Um *namespace* é identificado por um nome único (uma URL) e consiste em uma coleção de tipos de elementos e nomes de atributos. Qualquer tipo de elemento ou nome de atributo pertencente a um *namespace* pode ser identificado unicamente através de um nome composto por duas partes: i) o nome do *namespace* e ii) o nome do tipo de elemento ou atributo.

Namespaces são declarados em um documento XML através do atributo `xmlns`, o qual associa um prefixo a um *namespace*. O escopo desta declaração inclui o elemento que contém o atributo `xmlns` e todos os seus descendentes. Quando um

namespace contém um prefixo, então os tipos de elementos e nomes de atributos pertencentes a ele devem ser referenciados com o prefixo correspondente. Quando uma declaração de *namespace* não contém um prefixo, então este *namespace* será considerado *default* e os elementos pertencentes a ele são referenciados sem um prefixo. É importante lembrar que a única função de um *namespace* é prover uma forma de distinção entre nomes. Além disso, as URLs usadas como nomes de *namespaces* não apontam para esquemas, nem para informações sobre *namespaces*, ou seja, estas URLs são apenas identificadores.

Todo esquema XML está associado ao *namespace* <http://www.w3.org/2000/10/XMLSchema>. Isto quer dizer que todos os elementos utilizados para a definição de esquemas XML pertencem a este *namespace* (por exemplo: *complexType*, *sequence*). Os elementos que são declarados em um esquema XML também podem estar associados a um *namespace*, o qual é denominado de *targetNamespace*. Por exemplo, no esquema “livraria.xsd”, apresentado na Figura 10.6, foi definido que o *targetNamespace* corresponde ao *namespace* <http://www.livraria.org>. Os *namespaces* utilizados na definição de um esquema XML devem ser especificados no elemento raiz do esquema. Todo esquema XML tem como elemento raiz o elemento <*schema*>.

• Sintaxe Básica

Assim como uma DTD, um esquema XML é um conjunto de declarações de elementos e atributos. Entretanto, ao contrário da DTD, a linguagem XML Schema oferece meios para definir tipos de dados, que podem ser tipos simples como PCDATA ou podem ser tipos mais complexos e estruturados. XML Schema dispõe de dois tipos de elementos que podem ser usados para a definição de tipos: o *simpleType* e o *complexType*. O *complexType* deve ser usado quando se deseja definir um elemento que possui subelementos ou atributos. Por exemplo, no esquema “livraria.xsd” foi utilizado o *complexType* para definir um elemento *livro*, composto pelos elementos *autor*, *titulo* e *editora*.

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
              targetNamespace = "http://www.livraria.org"
              xmlns: "http://www.livraria.org">
  <xsd:element name="livro">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="autor" type="xsd:string"/>
        <xsd:element name="titulo" type="xsd:string"/>
        <xsd:element name="editora" type="xsd:string"/>
      </xsd:sequence>
      <xsd:attribute name="ano" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figura 10.6. Exemplo de esquema definido na linguagem XML Schema (livraria.xsd)

Por outro lado, deve-se utilizar o elemento `simpleType` para criar tipos de dados que correspondem a refinamentos de tipos de dados primitivos (por exemplo: `string`, `integer`). Um novo tipo de dados pode ser definido a partir de um tipo de dados existente pela atribuição de valores às “facetas” dos tipos de dados (*facets*). Como mostrado a seguir, o tipo de dados `meuInteiro` foi criado a partir do tipo base `integer` através da atribuição de limites (`minInclusive` e `maxInclusive`) para os valores que o tipo pode receber.

Exemplo:

```
<simpleType name="meuInteiro">
    <restriction base="integer">
        <minInclusive value = "1">
        <maxInclusive value = "10">
    </restriction>
</simpleType>
```

Os tipos de dados definidos em um esquema podem ser usados para a definição de elementos e atributos. Definir e usar tipos de dados é comparável a definir uma classe e usá-la para criar objetos. Os tipos complexos (`complexType`) podem ser usados apenas para definição de elementos, enquanto que os tipos simples (`simpleType`) podem ser usados para definição tanto de elementos como de atributos. Ao contrário de outras linguagens para definição de esquemas, XML Schema permite a definição de cardinalidade para um elemento (i.e. o número possível de ocorrências do elemento). Para isto podem ser utilizados o atributo `minOccurs` e o atributo `maxOccurs`, que determinam respectivamente o número mínimo e máximo de ocorrências de um elemento.

Basicamente, existem três formas diferentes de declarar elementos:

- **A declaração de um elemento tem como subelemento a definição de um tipo complexo.**

Exemplo:

```
<xsd:element name="livro">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="titulo" type="xsd:string"/>
            <xsd:element name="editora" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

- **A declaração de um elemento tem como subelemento a definição de um tipo simples.**

Exemplo:

```
<xsd:element name="meuInteiro">
    <simpleType>
        <restriction base="integer">
            <minInclusive value = "1">
            <maxInclusive value = "10">
        </restriction>
    </simpleType>
```

```
</xsd:element>
```

- **A declaração de um elemento faz referência a um tipo complexo já definido.**

Exemplo:

```
<xsd:element name="livro" type="Tlivro"/>
<xsd:complexType name="Tlivro">
    <xsd:sequence>
        <xsd:element name="titulo" type="xsd:string"/>
        <xsd:element name="editora" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

Declarações de elementos e tipos são ditas globais quando são filhas imediatas do elemento `<schema>`. Por outro lado, declarações de elemento e definições de tipos são consideradas locais quando estão aninhadas dentro de outros elementos ou tipos. Esta diferença é importante porque apenas elementos e tipos globais podem ser reusados.

XML Schema também permite que tipos complexos sejam derivados a partir de outros tipos. Esta derivação pode ser de duas formas:

- **Por extensão:** adicionando novos elementos a um tipo (semelhante a herança). Por exemplo, o tipo complexo `autorEstendido` é criado a partir de um elemento `autor` previamente definido através da adição do elemento `endereco`.

```
<complexType name="autorEstendido" base="autor">
    <element name="endereco" type="string"
        minOccurs='0' maxOccurs='*' />
</complexType >
```

- **Por restrição:** aplicando restrições aos elementos de um tipo. Por exemplo, o tipo `publicacaoAutorUnico` é criado a partir de um tipo `publicacao` restringindo-se o número de elementos `autor` para um único elemento.

```
<xsd:complexType name="publicacaoAutorUnico">
    <xsd:restriction base ="publicacao">
        <xsd:sequence>
            <xsd:element name="autor" type="xsd:string"
                maxOccurs="1"/>
            <xsd:element name="titulo" type="xsd:string"/>
            <xsd:element name="editora" type="xsd:string"/>
        </xsd:sequence>
    </xsd:restriction>
</xsd:complexType>
```

XML Schema também permite a definição de grupos que especificam restrições sobre um conjunto fixo de subelementos. Os grupos podem ser de três tipos:

- **sequence:** todos os elementos pertencentes a este grupo devem aparecer na ordem em que foram definidos e nenhum elemento pode ser omitido.

Exemplo:

```
<xsd:complexType>
    <xsd:sequence>
        <xsd:element name="titulo" type="xsd:string"/>
        <xsd:element name="autor" type="xsd:string"/>
```

```

    </xsd:sequence>
</xsd:complexType>
```

- **choice:** apenas um dos elementos pertencentes ao grupo deve aparecer em uma instância de documento XML.

Exemplo:

```

<xsd:complexType>
  <xsd:choice>
    <xsd:element name="livro" type="xsd:string"/>
    <xsd:element name="artigo" type="xsd:string"/>
  </xsd:choice>
</xsd:complexType>
```

- **all:** os elementos podem aparecer em qualquer ordem e podem ser repetidos ou omitidos

Exemplo:

```

<xsd:complexType>
  <xsd:all>
    <xsd:element name="titulo" type="xsd:string"/>
    <xsd:element name="autor" type="xsd:string"/>
  </xsd:all>
</xsd:complexType>
```

Com relação aos atributos, é importante notar que as declarações de atributo sempre aparecem após as declarações de elemento. Os atributos sempre dizem respeito ao elemento dentro do qual estão sendo definidos. Por exemplo, o esquema “livraria.xsd” tem uma declaração de atributo para o atributo *ano* do elemento *livro*. (<xsd:attribute name="ano" use="required" type="xsd:string"/>).

Assim como nas DTDs, também é possível declarar o valor padrão de um atributo. Isto é feito através da cláusula *use*, que pode assumir um dos seguintes valores: *required* (obrigatório), *optional* (opcional) e *fixed* (fixo). Neste último caso, deve-se dizer o valor padrão do atributo utilizando a cláusula *value*. Por exemplo, no esquema “livraria.xsd” a declaração do atributo *ano* especifica que este atributo é obrigatório, ou seja, deve fazer parte de todos os elementos *livro*.

10.5. Modelos de Dados para XML

Um documento XML pode ser visto como uma “linearização” de uma estrutura em árvore, onde os nós da árvore podem representar elementos, atributos, instruções de processamento, entre outros. A Figura 10.7 apresenta a representação em árvore para o documento “livraria.xml”, de acordo com o modelo de dados XML Query Data Model [Fernandez 2001]. Este modelo de dados é resultado de um refinamento do modelo de dados descrito na especificação XPath [Clark 1999a], no qual um documento é modelado como uma árvore de nós. No modelo de dados XML Query Data Model, os elementos são representados como nós, os subelementos são representados como arcos para os nós e os atributos são representados como campos que podem ser acessados a partir de seus elementos.

Até o momento, quatro especificações de modelos de dados para XML foram proposta pelo W3C: o modelo de dados Infoset [Cowan 2001], o modelo de dados XPath [Clark 1999a], o modelo DOM [Apparao 1998], [Le Hors 2000] e o XML Query Data Model

[Fernandez 2001]. Estes quatro modelos de dados descrevem um documento XML como uma estrutura em árvore, mas existem diferenças na estrutura das árvores e nas informações que estas disponibilizam.

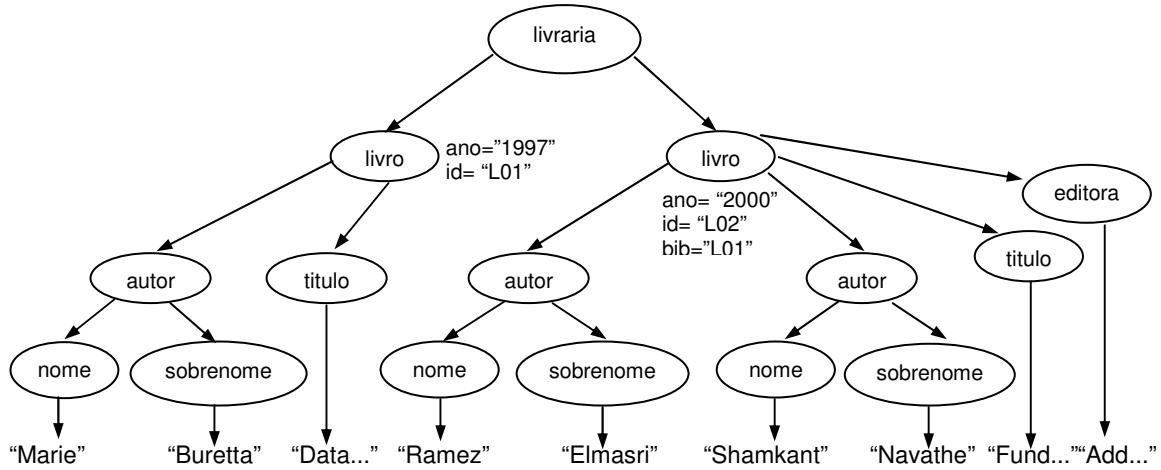


Figura 10.7. Modelo de dados para o documento livraria.xml

10.6. Linguagens de Consulta para XML

Dados XML são diferentes dos dados de bancos de dados relacionais ou orientados a objetos e, por isso, as linguagens de consulta convencionais como SQL e OQL não são adequadas para especificar consultas em documentos XML. A principal distinção entre os dados convencionais e os dados XML está no fato de que os dados XML não seguem uma estrutura rígida, como os esquemas relacionais. Em modelos relacionais ou orientados a objetos, existe a noção de um esquema fixo, pré-definido e independente dos dados, a partir do qual as instâncias do banco de dados são definidas. Por outro lado, em XML o esquema existe juntamente com os dados, ou seja, XML é autodescritível e pode, naturalmente, modelar irregularidades que não podem ser modeladas quando modelos de dados relacionais ou orientados a objetos são usados. Por exemplo, com o uso de XML é possível representar: i) itens de dados que podem ter elementos ausentes ou podem ter múltiplas ocorrências de um mesmo elemento, ii) elementos que podem ter valores atômicos em alguns itens de dados e valores estruturados em outros itens, e iii) coleções de elementos com estruturas heterogêneas.

Muitas linguagens para extração e reestruturação de dados XML têm sido propostas. Algumas dessas linguagens são baseadas nos conceitos de linguagens de consultas tradicionais, como SQL e OQL. Dentre as linguagens de consulta para XML destacam-se: LOREL [Abiteboul 1997b], XML-QL [Deutsch 1999], XML-GL [Ceri 1999], XSL [Clarck 1999b], XQL [Robie 1998a], [Robie 1998b].

10.6.1. XQuery

Para ilustrar a definição de consultas sobre dados XML, foi escolhida a linguagem de consultas XQuery [Chamberlin 2001]. Esta linguagem também é uma proposta da W3C, que agrupa características de várias outras linguagens de consultas.

Assim como as linguagens de consulta para dados semi-estruturados, as linguagens de consulta para XML também usam o conceito de expressão de caminho para navegar em árvores. As expressões de caminho XQuery usam a sintaxe abreviada de XPath. Em XQuery, o resultado de uma expressão de caminho é uma lista ordenada de nós (cada nó inclui seus descendentes, logo o resultado pode ser visto como uma floresta ordenada). Os nós de nível mais alto da expressão de caminho são ordenados de acordo com sua posição na hierarquia original seguindo a ordem *top-down* e *left-right*.

Como visto anteriormente, uma expressão de caminho consiste de uma série de passos. Um passo representa um movimento ao longo de um documento em uma determinada direção. Cada passo pode aplicar um ou mais predicados para eliminar nós que não satisfazem uma determinada condição. O resultado de um passo é uma lista de nós que pode ser vista como um ponto de início para o próximo passo.

Uma expressão de caminho pode começar com uma expressão que identifica um nó específico, tal como a função `document (string)`, que retorna o nó raiz de um dado documento. Uma consulta também pode conter uma expressão de caminho começando com “/” ou “//”, que representa um nó raiz implícito, determinado pelo ambiente no qual a consulta está sendo executada. Uma discussão completa sobre a sintaxe abreviada de XPath pode ser encontrada em [Clark 1999a]. Além das expressões de caminho, existem outras expressões de XQuery que podem ser destacadas.

▪ Expressões para construção de elementos

Uma expressão para construção de elementos consiste de um marcador de início e um marcador de final que delimitam o elemento a ser criado. Estes marcadores agrupam uma série de outras expressões que provêem o conteúdo do elemento.

Exemplo: Gere um elemento `<livros>` contendo todos os títulos dos livros cujo autor tem como último nome “Abiteboul”.

```
<livros>
  FOR $1 IN ("livraria.xml")//livro
    WHERE $1/autor/sobrenome= "Abiteboul"
    RETURN $1/titulo
</livros>
```

▪ Expressões FLWR

Uma expressão FLWR (pronunciada “*flower*”) é construída a partir de cláusulas FOR, LET, WHERE e RETURN. Assim como em uma consulta SQL, estas cláusulas devem aparecer em uma ordem específica. Uma expressão FLWR associa valores a uma ou mais variáveis e utiliza esses valores para construir um resultado. A primeira parte de uma expressão FLWR consiste de cláusulas FOR e/ou cláusulas LET que associam valores a uma ou mais variáveis. Os valores a serem associados às variáveis são representados por expressões (por exemplo: expressões de caminho).

Exemplo: Liste os títulos dos livros publicados pela editora “Addison Wesley”. No ano de 1996.

```
FOR $1 IN ("livraria.xml")//livro
WHERE $1/editora = "Addison Wesley"
AND $1/ano = "1996"
RETURN $1/titulo
```

▪ Expressões condicionais

Expressões condicionais são úteis quando a estrutura da informação a ser retornada depende de alguma condição. Assim como todas as outras expressões XQuery, as expressões condicionais podem ser aninhadas e podem ser usadas em qualquer lugar onde um valor é esperado.

Exemplo: Faça uma lista de todas as publicações, ordenadas por título. Para os livros, inclua a editora e para os artigos inclua a conferência onde eles foram publicados.

```
FOR $p IN //publicacoes
RETURN
<publicacoes>
  $p/titulo,
  IF $p/@tipo = "livro"
  THEN $p/editora
  ELSE $p/conferencia
</publicacoes> SORTBY (titulo)
```

▪ Expressões com quantificadores

Em algumas situações pode ser necessário testar a existência de algum elemento que satisfaz uma determinada condição, ou para determinar se todos os elementos em alguma coleção satisfazem uma condição. Para isto XQuery provê quantificadores existenciais e universais, os quais são exemplificados a seguir.

Exemplo: Encontre todos os títulos de livro no qual a palavra “web” é mencionada em todos os parágrafos.

```
FOR $l IN //livro
WHERE EVERY $p IN $l//paragrafo SATISFIES
      contains ($p, "web")
RETURN $l/titulo
```

10.7. SQL/XML

O problema de publicar dados relacionais no formato de XML tem especial importância, uma vez que XML emergiu como o formato padrão da troca de dados entre aplicações *Web* enquanto a maioria dos dados de negócio continua armazenada em SGBDs relacionais. Deste modo, grande parte dos SGBDs mais importantes desenvolveu extensões proprietárias para a geração de XML a partir de dados relacionais. Tais extensões usam diferentes abordagens e não existe interoperabilidade entre elas, obrigando usuários que trabalham com diversos bancos a escreverem códigos diferentes, um para cada SBD.

SQL/XML é um padrão ANSI/ISO desenvolvido pelo INCITS H2.3 que estende o SQL. Com SQL/XML é possível construir consultas declarativas que retornam dados XML a partir de bases relacionais. O XML resultante da consulta pode ter a estrutura que o projetista desejar, e as consultas podem ser arbitrariamente complexas. Para um programador SQL, SQL/XML é fácil de aprender, pois envolve um grupo pequeno de adições à já existente linguagem SQL. Uma vez que SQL é uma linguagem madura, existe um grande número de ferramentas e infra-estrutura disponíveis que podem ser facilmente adaptadas para suportar o novo padrão.

No padrão SQL/XML, as *funções de publicação* constituem a parte que provê a publicação de dados relacionais no formato XML. As funções de publicação são usadas diretamente na consulta SQL e permitem criar qualquer estrutura XML que o usuário deseje a partir do esquema relacional. O resultado de uma função de publicação é sempre uma instância do tipo de dados XML, o qual também é definido pelo padrão. Em uma consulta SQL/XML, as funções de publicação podem ser processadas pelo SGBD juntamente com as cláusulas SQL, o que representa um ganho de performance.

As principais funções de publicação do SQL/XML são: *XMLElement()*, *XMLAttributes()*, *XMLForest()*, *XMLAgg()*, descritas a seguir:

XMLElement e **XMLAttributes** – A função XMLElement constrói um novo elemento XML com atributos e conteúdo. A função recebe como parâmetro o nome do elemento, um conjunto opcional de atributos para o elemento, e zero ou mais argumentos adicionais que compõem o conteúdo do elemento. O resultado da função é uma instância do tipo XMLType. Caso todos os argumentos da função retornem valor nulo, então nenhum conteúdo é gerado para o elemento.

A função XMLAttributes especifica os atributos do elemento raiz. A função recebe como argumento um ou mais expressões escalares que podem apresentar *aliases* ou não. Se o valor de alguma expressão for nulo, o atributo correspondente não é gerado. A função XMLAttributes só pode ser chamada como primeiro argumento da função XMLElement.

Como exemplo, considere a seguinte consulta sobre a relação **CLIENTES_REL**:

```
SELECT XMLElement("cliente",
    XMLAttributes(C.CCODIGO AS "codigo"),
    XMLElement("nome", C.CNOME),
    XMLElement("fone", C.CFONE1),
    XMLElement("fone", C.CFONE2),
    XMLElement("fone", C.CFONE3) )
FROM CLIENTES_REL C;
```

O resultado da consulta é dado por:

```
<cliente codigo="193">
<nome>Bryan Huston</nome>
<fone>+91 11 012 4813</fone>
<fone>+91 11 083 4813</fone>
<fone>+91 11 012 4827</fone>
</cliente>
<cliente codigo="195">
<nome>Cary Stockwell</nome>
<fone>+91 11 012 4835</fone>
<fone></fone>
<fone></fone>
</cliente>
```

O identificador cliente dá nome ao elemento raiz. A expressão 'C.CCODIGO AS "codigo"' dentro da função XMLAttributes gera o atributo *codigo* do elemento a partir do valor da expressão escalar 'C.CCODIGO'. As expressões 'XMLElement("nome", C.CNOME)' e 'XMLElement("fone", C.CFONE1)' geram o conteúdo do elemento <cliente>. Em 'XMLElement("nome", C.CNOME)', o valor escalar da expressão

'C.CNOME' é mapeado no valor XML equivalente **Erro! Fonte de referência não encontrada.**, gerando assim o conteúdo do elemento <nome>. Note que, no segundo elemento <cliente> do resultado, como os valores de 'C.CFONE2' e 'C.CFONE3' são nulos, então nenhum conteúdo é gerado para o elemento correspondente.

XMLForest – A função XMLForest gera uma floresta de elementos XML a partir de seus argumentos, os quais são expressões escalares com *aliases* opcionais. Cada expressão é convertida no formato XML correspondente e, caso um *alias* tenha sido atribuído, este será o identificador do elemento gerado. Expressões que resultam em nulo não geram elementos.

Como exemplo, considere a seguinte consulta sobre a relação **CLIENTES_REL**:

```
SELECT XMLElement("cliente",
    XMLAttributes(C.CCODIGO AS "codigo"),
    XMLElement("nome", C.CNOME),
    XMLForest(C.CFONE1 AS "fone",
        C.CFONE2 AS "fone",
        C.CFONE2 AS "fone" ) )
FROM CLIENTES_REL C;
```

O resultado da consulta é dado por:

```
<cliente codigo="193">
    <nome>Bryan Huston</nome>
    <fone>+91 11 012 4813</fone>
    <fone>+91 11 083 4813</fone>
    <fone>+91 11 012 4827</fone>
</cliente>
<cliente codigo="195">
    <nome>Cary Stockwell</nome>
    <fone>+91 11 012 4835</fone>
</cliente>
```

O identificador *fone* dá nome aos elementos gerados das expressões 'C.CFONE1 AS "fone"', 'C.CFONE2 AS "fone"' e 'C.CFONE3 AS "fone"'. Note que, no segundo elemento <cliente> do resultado, como os valores de 'C.CFONE2' e 'C.CFONE3' são nulos, então nenhum elemento foi gerado.

XMLEgg – É uma função de agregação que gera uma agregado XML (uma lista de elementos XML) a partir de cada agrupamento SQL. Se nenhum agrupamento for especificado, é retornado um agregado XML para todas as cláusulas da consulta. Argumentos nulos são removidos do resultado. A função recebe como parâmetro uma única expressão que gera uma instância do tipo XMLType.

Como exemplo, considere as seguintes consultas sobre a relação **CLIENTES_REL**:

```
SELECT XMLElement("cliente",
    XMLAttributes(C.CCODIGO AS "codigo"),
    XMLElement("nome", C.CNOME),
    XMLForest(C.CFONE1 AS "fone",
        C.CFONE2 AS "fone",
        C.CFONE2 AS "fone"),
    (SELECT XMLAgg( XMLElement("pedido",
        XMLElement("codigo", P.PCODIGO),
        XMLElement("data", P.PDATA_PEDIDO) ) )
```

```

        FROM PEDIDOS_REL P
        WHERE P.PCLIENTE = C.CCODIGO ) )
FROM CLIENTES_REL C;

e
SELECT XMLElement("cliente",
    XMLAttributes(C.CCODIGO AS "codigo"),
    XMLElement("nome", C.CNOME),
    XMLForest(C.CFONE1 AS "fone",
               C.CFONE2 AS "fone",
               C.CFONE3 AS "fone"),
    XMLAgg(
        XMLElement("pedido",
            XMLElement("codigo", P.PCODIGO),
            XMLElement("data", P.PDATA_PEDIDO) ) ) )
FROM CLIENTES_REL C, PEDIDOS_REL P
WHERE P.PCLIENTE = C.CCODIGO
GROUP BY C.CCODIGO, C.CNOME, C.CFONE1, C.CFONE2, C.CFONE3,
P.PCODIGO, P.PDATA_PEDIDO;

```

O resultado de ambas é o mesmo, dado por:

```

<cliente codigo="193">
    <nome>Bryan Huston</nome>
    <fone>+91 11 012 4813</fone>
    <fone>+91 11 083 4813</fone>
    <fone>+91 11 012 4827</fone>
    <pedido>
        <codigo>405</codigo>
        <data>01/07/05</data>
    </pedido>
</cliente>
<cliente codigo="195">
    <nome>Cary Stockwell</nome>
    <fone>+91 11 012 4835</fone>
    <pedido>
        <codigo>407</codigo>
        <data>29/06/05</data>
    </pedido>
    <pedido>
        <codigo>408</codigo>
        <data>01/07/05</data>
    </pedido>
</cliente>

```

Na primeira consulta, é retornado um agregado XML para todas as cláusulas da subconsulta. Na segunda consulta, é retornado um agregado XML para cada agrupamento definido pela cláusula GROUP BY. Mais detalhes do padrão podem ser encontrados em [Eisenberg 2001] [Eisenberg 2004].

10.8. Conclusões

A linguagem XML consolidou-se como um padrão para representação e troca de dados tanto na Web como em aplicações de uma maneira geral. A simplicidade de XML aliada à sua capacidade de definição de novos marcadores fez desta linguagem um grande sucesso no contexto de definição de padrões para troca de dados. Os documentos XML possuem uma estrutura hierárquica que permitem tanto a representação de dados

estruturados como de dados semi-estruturados. Além disso, em um documento XML é possível representar tanto o conteúdo quanto a estrutura dos dados, aliando mais semântica à informação que está sendo representada. Outra importante característica de XML está relacionada à possibilidade de definição de esquemas para validação dos documentos XML. Assim como XML, o W3C também propôs uma linguagem padrão para definição de esquemas XML, chamada XML Schema. Dessa forma, é possível definir esquemas que serão usados para garantir que os documentos XML estão de acordo com o padrão esperado.

Neste capítulo, apresentamos a linguagem XML enfatizando seus conceitos básicos e aplicações. Também foram discutidas linguagens para definição de esquemas XML e linguagens para consultas em documentos XML. É importante salientar a discussão sobre a linguagem SQL/XML, a qual vem se consolidando como linguagem padrão para geração de visões XML a partir de dados armazenados em bancos de dados relacionais. Tendo em vista a grande quantidade de dados disponíveis no formato relacional e a crescente necessidade de publicação destes dados em formato XML, torna-se essencial que sejam definidos mecanismos que facilitem esta tarefa.

Além dos conceitos e das linguagens que foram apresentados neste capítulo, existem diversos outros padrões associados à XML, os quais foram propostos pelo W3C com objetivos diferentes, incluindo definição de links entre documentos XML e definição de folhas de estilo para documentos XML. Maiores informações sobre estes padrões podem ser encontrados no site do W3C (www.w3c.org).

Referências

- Abiteboul, S., J.Mchugh, D., Widom, J. and Wiener, J. (1997b) “The loren query language for semistructured data”, International Journal on Digital Libraries, vol. 1, no. 1, p.68-88.
- Apparo, V., Byrne, S., Champion, M., Isaacs, S., Jacobs, I., Le Hors, A., Nicol, G., Robie, J., Sutor, R., Wilson, C. and Wood, L. (1998) “Document Object Model (DOM) Level 1 Specification (Second Edition) Version 1.0”, World Wide Web Consortium, <http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/>.
- Biron, P. V. and Malhotra, A. (2000) “XML Schema Part 2: Datatypes”, World Wide Web Consortium, <http://www.w3.org/TR/xmlschema-2>.
- Bray, T., Hollander, D. and Layman, A. (1999) “Namespaces in XML”, World Wide Web Consortium, <http://www.w3.org/TR/REC-xml-names/>.
- Bray, T., Paoli, J. and Sperberg-McQueen, C. M. (1998) “Extensible Markup Language (XML) 1.0”, World Wide Web Consortium, <http://www.w3.org/TR/REC-xml>.
- Ceri, S., Comai, S., Damiani, E., Frernali, P., Paraboschi S. and Tanca, L. (1999) “Xml-gl: a graphical language for querying and restructuring www data”, In Proc. of 8th Int. World Wide Web Conference, WWW8, Toronto, Canada.
- Chamberlin, D., Florescu, D., Robie, J., Siméon, J. and Stefanescu, M. (2001) “XQuery: A Query Language for XML”, World Wide Web Consortium, <http://www.w3.org/TR/xquery/>.

- Clark, J. (1999a) “Xml Path Language (XPATH)”, World Wide Web Consortium, <http://www.w3.org/TR/xpath>.
- Clark, J. (1999b) “XSL Transformations (XSLT specification)”, World Wide Web Consortium, <http://www.w3.org/TR/WD-xslt>.
- Cowan J. and Tobin R. (2001) “XML Information Set”, World Wide Web Consortium, <http://www.w3.org/TR/xml-infoset/>
- Davidson, A., Fuchs, M., Hedin, M. et al. (1999) “Schema for Object-Oriented XML 2.0”, World Wide Web Consortium, <http://www.w3.org/TR/NOTE-SOX>.
- Deutsch, A., Fernandez, M., Florescu, D., Levy, A. and Suciu, D. (1999) “A query language for xml”, In International World Wide Web Conference.
- Eisenberg, A., Melton, J., Kulkarni, K., Michels, J.E. and Zemke, F. (2004) “SQL:2003 has been published”, In ACM SIGMOD Record, v. 33, n. 1 (Março 2004), COLUMN: Standards, pp. 119–126.
- Eisenberg, A., Melton, J. (2001) “SQL/XML and the SQLX Informal Group of Companies”, In ACM SIGMOD Record, v. 30, n. 3 (Setembro 2001), COLUMN: Standards.
- Fernandez, M. and Robie, J. (2001) “Query Datamodel”, World Wide Web Consortium, <http://www.w3.org/TR/query-datamodel/>.
- Frankston C. and Thompson, H. S. (1998) “XML-Data Reduced”, Internet Document, <http://www.ltg.ed.ac.uk/~ht/XMLData-Reduced.htm>.
- Jellife, R. (2000) “Schematron”, Internet Document, <http://www.ascc.net/ xml/resource/schematron/>.
- Karllund, N., Moller, A. and Schwatzbach, M. I. (2000) “DSD: a schema language for xml”, In Proc. of 3rd ACM Workshop on Formal Methods in Software Practice.
- Le Hors, A., Le Hégaret, P., Wood, L., Nicol, G., Robie, J., Champion, M. and Byrne, S. (2000) “Document Object Model (DOM) Level 2 Core Specification Version 1.0”, World Wide Web Consortium, <http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/>.
- Lee D. and Chu, W.W. (2000) “Comparative analysis of six xml schema languages”, SIGMOD Record, vol. 29, no. 3, p. 76-87.
- McGrath, S., Xml by Example: Building E-Commerce Applications, Prentice Hall Computer Books, 1998.
- Robie, J. (1998a) “The design of XQL”, World Wide Web Consortium, <http://www.w3.org/ Style/XSL/Group/1998/09/XQL-design.html>.
- Robie, J., Lapp, J. and Schach, D. (1998b) “Xml query language (xql)”, In Proc. of the Query Languages workshop, Cambridge, Mass.
- Schach, D., Lapp, J. and Robie, J. (1998) “Querying and transforming xml”, In Proc. of the Query Languages workshop, Cambridge, Mass.
- Thompson, H. S., Beech, D., Maloney, M. and Mendelsohn, N. (2000) “XML Schema Part 1: Structures”, World Wide Web Consortium, <http://www.w3.org/TR/xmlschema-1>.