O *framework* de integração do sistema
DISCOVER

Ronaldo Cristiano Prati

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 20 de fevereiro de 2003

Assinatura:_____

O *framework* de integração do sistema DISCOVER¹

Ronaldo Cristiano Prati

Orientadora: *Prof^a Dr^a Maria Carolina Monard*

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação — ICMC/USP — como parte dos requisitos necessários à obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP - São Carlos
Fevereiro/2003

¹Trabalho realizado com auxílio financeiro da CAPES.

Este documento foi preparado utilizando-se o formatador de textos $\text{\LaTeX}$, para impressão frente e verso. Sua bibliografia é mantida com o auxílio da ferramenta BIBVIEW (Prati, Baranauskas, and Monard, 1999) e gerada automaticamente pelo BIB $\text{\TeX}$, utilizando o estilo Chicago. Todas ligações para sítios na internet presentes nesta dissertação foram acessados em 1 de agosto de 2003.

Dedicatória

aos meus pais

Agradecimentos

Seguindo a luz do sol, nós deixamos o velho mundo

– CRISTÓVÃO COLOMBO

As grandes jornadas somente são possíveis se bons referenciais estiverem disponíveis a quem as faz. Colombo, o descobridor das Américas, e os grandes navegadores da sua época, acreditaram em seus sonhos e seguiram as estrelas no céu.

Nesta minha jornada, não tão grandiosa quanto a de Colombo, também segui meus sonhos, e, tenho certeza absoluta, só estou conseguindo realizá-los graças a um grupo de pessoas que me mostraram, através de exemplos, o melhor caminho a seguir.

Que melhor lugar para iniciar uma jornada do que um bom ponto de partida? Meus queridos pais, João e Maria, que sempre me apoiaram em todas as minhas decisões e momentos difíceis. Vocês não me deram um berço de ouro, mas me mostraram que respeito, honestidade e cumplicidade valem muito mais que qualquer fortuna. Vocês, juntamente com o resto de minha família, formaram a constelação que sempre apontava o porto seguro de regresso, na eventualidade dos insucessos a que todos estamos sujeitos.

Não menos necessários são aqueles marcos fixos, os quais sempre sabemos qual a direção que apontam. Em muitas jornadas não há mapas, e, muitas vezes, tudo o que podemos contar para traçar nosso caminho são esses marcos, tal como a Estrela Polar marca o norte e o Cruzeiro aponta o Sul.

A senhora, Professora Carolina, que me orientou durante todos esses anos, sempre me mostrou o caminho da moralidade, justiça e perseverança, chamando a atenção para as nuances do percurso. Em poucas pessoas pode se observar tão nitidamente essas características. Aprendi muito com a senhora, nesses anos de convivência.

Também gostaria de ressaltar a garra e determinação da professora Solange e o dinamismo e criatividade do Professor André. Sábida é a nação japonesa, a qual até mesmo o imperador presta homenagem e se curva na presença do seu professor.

Meus amigos Augusto e Gustavo sempre representaram para mim aquelas estrelas mais brilhantes e que se destacam no céu. Augusto, com quem trabalhei na iniciação científica, ajudou a despertar em mim a paixão pela vida acadêmica. Gustavo, cujas conversas auxiliaram neste trabalho, ajudou a solidificá-la. Além disso, eles foram os idealizados do DISCOVER.

Algumas outras estrelas menores, meus amigos e colegas de trabalho, dentre eles Daniel, Valmir, Jaqueline, Cláudia Aparecida, Flávia, Patrícia, Alan, Huei, Rodrigo, Cláudia Regina, Marcos Roberto, Marcos Ferreira, Edson e outros que me faltam agora o nome, também embelezaram essa minha jornada. Não fiquem chateados

pensando que o adjetivo menor tem algum sentido pejorativo. São justamente essas estrelas que embelezam as constelações. Afinal, o majestoso Aquiles no céu não é formado apenas pelo seu brilhante cinturão.

Finalmente, as jornadas seriam insuportáveis sem alguém com quem compartilhar os triunfos e afugentar os medos. Minha querida namorada, Silvia, tem tudo que um homem deseja em uma boa companheira: compreensão, amizade, carinho, respeito e amor. Além de tudo, é linda e cativante. Minha jornada foi muito mais fácil desde o dia que estamos juntos.

O que mais uma pessoa tão afortunada pode querer da vida? Que ela lhes retribua em alegrias tudo o que fizeram por mim. Obrigado a todos.

Resumo

Para qualquer sentença que possa vir a ser lida mais de duas vezes, devemos ter certeza que ela foi pensada, no mínimo, duas vezes.

– HENRY DAVID THOREAU

Talvez uma das maiores capacidades do ser humano seja a sua habilidade de aprender a partir de observações e transmitir o que aprendeu para outros humanos. Durante séculos, a humanidade vem tentando compreender o mundo em que vive e, a partir desse novo conhecimento adquirido, melhorar o mundo em que vive.

O desenvolvimento da tecnologia colocou a descoberta de conhecimento em um momento ímpar na história da humanidade. Com os progressos da Ciência da Computação, e, em particular, da Inteligência Artificial — IA — e Aprendizado de Máquina —AM, hoje em dia é possível, a partir de métodos de inferência indutiva e utilizando um conjunto de exemplos, descobrir algum tipo de conhecimento implícito nesses exemplos.

Entretanto, por ser uma área de pesquisa relativamente nova, e por envolver um processo tanto iterativo quanto interativo, atualmente existem poucas ferramentas que suportam eficientemente a descoberta de conhecimento a partir dos dados. Essa falta de ferramentas se agrava ainda mais no que se refere ao seu uso por pesquisadores em Aprendizado de Máquina e Aquisição de Conhecimento.

Esses fatores, além do fato que algumas pesquisas em nosso Laboratório de Inteligência Computacional — LABIC — têm alguns componentes em comum, motivaram a elaboração do projeto DISCOVER, que consiste em uma estratégia de trabalho em conjunto, envolvendo um conjunto de ferramentas que se integram e interajam, e que supram as necessidades de pesquisa dos integrantes do nosso laboratório. O DISCOVER também pode ser utilizado como um campo de prova para desenvolver novas ferramentas e testar novas idéias.

Como o DISCOVER tem como principal finalidade o seu uso e extensão por pesquisadores, uma questão principal é que a arquitetura do projeto seja flexível o suficiente para permitir que novas pesquisas sejam englobadas e, simultaneamente, deve impor determinados padrões que permitam a integração eficiente de seus componentes.

Neste trabalho, é proposto um *framework* de integração de componentes que tem como principal objetivo possibilitar a criação de um sistema computacional a partir das ferramentas desenvolvidas para serem utilizadas no projeto DISCOVER.

Esse *framework* compreende um mecanismo de adaptação de interface que cria uma camada (interface horizontal) sobre essas ferramentas, um poderoso mecanismo de metadados, que é utilizado para descrever tanto os componentes que implementam as funcionalidades do sistema quanto as configurações de experimentos criadas pelos usuário, que serão executadas pelo *framework*, e um ambiente de execução para essas configurações de experimentos.

Abstract

What we call the beginning is often the end.
And to make an end is to make a beginning.
The end is where we start from...

– T. S. ELIOT, FOUR QUARTETS

One of human greatest capability is the ability to learn from observed instances of the world and to transmit what have been learnt to others. For thousands of years, we have tried to understand the world, and used the acquired knowledge to improve it.

Nowadays, due to the progress in digital data acquisition and storage technology as well as significant progress in the field of Artificial Intelligence — AI, particularly Machine Learning — ML, it is possible to use inductive inference in huge databases in order to find, or discover, new knowledge from these data. The discipline concerned with this task has become known as Knowledge Discovery from Databases — KDD. However, this relatively new research area offers few tools that can efficiently be used to acquire knowledge from data.

With these in mind, a group of researchers at the Computational Intelligence Laboratory — LABIC — is working on a system, called DISCOVER, in order to help our research activities in KDD and ML. The aim of the system is to integrate ML algorithms mostly used by the community with the data and knowledge processing tools developed as the results of our work. The system can also be used as a workbench for new tools and ideas.

As the main concern of the DISCOVER is related to its use and extension by researches, an important question is related to the flexibility of its architecture. Furthermore, the DISCOVER architecture should allow new tools be easily incorporated. Also, it should impose strong patterns to guarantee efficient component integration.

In this work, we propose a component integration framework that aims the development of an integrated computational environment using the tools already implemented in the DISCOVER project. The proposed component integration framework has been developed keeping in mind its future integration with new tools.

This framework offers an interface adapter mechanism that creates a layer (horizontal interface) over these tools, a powerful metadata mechanism, which is used to describe both components implementing systems' functionalities and experiment configurations created by the user, and an environment that enables these experiment execution.

Sumário

Lista de Figuras	xiii
Lista de Tabelas	xv
Lista de Códigos	xvii
1 Introdução	1
1.1 O Projeto DISCOVER	5
1.1.1 O Sistema DISCOVER	6
1.1.2 Trabalhos Realizados e em Desenvolvimento	8
1.2 Objetivos e Contribuições desta Dissertação	9
1.3 Notas sobre Tradução e Terminologia	10
1.4 Organização	11
2 Um Sistema para Descoberta de Conhecimento	13
2.1 O processo de Descoberta de Conhecimento	14
2.1.1 Interatividade no Processo de KDD	16
2.1.2 Iteratividade no Processo de KDD	16
2.1.3 KDD é um Processo Empírico	17
2.2 Transformando o Processo de KDD em Procedimentos	18
2.3 Um Sistema de Descoberta de Conhecimento	19
2.3.1 Controlando a Descoberta	21
2.3.2 Processando Dados	21
2.3.3 Procurando Padrões	22
2.3.4 Processando Padrões	23
2.3.5 Outras Características Desejáveis em um Sistema de KDD	24
2.4 Meta-Aprendizado	24
2.5 Considerações Finais	26
3 Arquitetura de Software	27
3.1 Arquitetura de <i>Software</i>	28
3.2 Estruturas de Arquitetura	30
3.3 Definição dos Atributos de Qualidade	31
3.4 Estilos de Arquitetura	33
3.4.1 Estilos heterogêneos	36
3.4.2 Operações Básicas	36
3.5 Desenvolvimento Baseado na Arquitetura	37
3.5.1 Criação do Esqueleto do Sistema	38
3.5.2 Explorando <i>Patterns</i> na Arquitetura de <i>Software</i>	39
3.6 Considerações Finais	39
4 Componentes, Frameworks e Patterns	41
4.1 Componentes	42
4.2 Frameworks e Modelos de Componente	45
4.2.1 Modelo de Componentes	45
4.2.2 Frameworks	46

4.3	<i>Software Patterns</i>	48
4.4	Arquitetura, Componentes, <i>Frameworks</i> , <i>Patterns</i> e todo o Resto	51
4.5	Considerações Finais	54
5	O Framework Proposto	57
5.1	Um <i>Framework</i> Baseado em <i>Patterns</i>	58
5.2	A Interface Horizontal	60
5.3	Uma Linguagem em XML para Especificar Componentes e Configurações de Experimentos	62
5.3.1	Descrição de uma Estrutura	63
5.3.2	A Descrição de um Componente	63
5.3.3	A Descrição de um Conector	64
5.3.4	A Descrição de uma Ligação	65
5.3.5	A Descrição de um Grupo	66
5.4	O Ambiente de Execução	66
5.5	Extensões e Melhorias	68
5.6	Considerações Finais	70
6	Exemplo da Utilização do Framework	71
6.1	A Interação com o Usuário	71
6.2	Um Primeiro Exemplo	73
6.3	Utilizando Conectores	77
6.4	Componentes Compostos	79
6.5	A Criação de Componentes	82
6.6	A Execução de um Experimento	82
6.7	Considerações Finais	85
7	Conclusões e Trabalhos Futuros	87
7.1	Resumo dos Capítulos e das Principais Contribuições desta Dissertação	87
7.2	Trabalhos Futuros	88
7.3	Considerações Finais	89
A	Descrição em XML da xDML	91
B	Notação	97
B.1	Diagramas Utilizados para Representar Esquemas XML	97
B.2	Diagrama de Interação	98
B.3	Diagrama de Classe	98
	Referências	107

Lista de Figuras

2.1	Etapas no Processo de KDD	15
2.2	KDD Visto Como um Processo Hierárquico	18
2.3	Modelo de uma Ferramenta Genérica para KDD	20
2.4	Compromisso entre Autonomia e Versatilidade em Sistemas de KDD	21
2.5	Meta-aprendizado Incorporado ao Modelo de Ferramenta Genérica de KDD	25
3.1	A Relação entre Modelos de Referência, Estilos de Arquitetura, Arquitetura de Referência e a Arquitetura de <i>Software</i> Implementada.	30
3.2	Catálogo de Estilos de Arquitetura	34
4.1	O <i>Design Pattern</i> Baseado em Componentes	53
4.2	Elementos de Reutilização de <i>Software</i>	54
5.1	O <i>Pattern Metadata</i>	60
5.2	O <i>Design Pattern Facade</i>	61
5.3	Utilizando o <i>Design Pattern Facade</i>	61
5.4	Representação do esquema de uma estrutura em xDML	64
5.5	Descrição de um componente em xDML	65
5.6	Representação do esquema de um conector em xDML	65
5.7	Representação do esquema de uma ligação em xDML	66
5.8	Representação do esquema de um grupo em xDML	66
5.9	O <i>pattern Interpreter</i>	67
5.10	O <i>pattern Visitor</i>	68
5.11	Diagrama de interação do <i>pattern Visitor</i>	69
6.1	Modelo da Interface Gráfica do DISCOVER	72
6.2	Configuração do Experimento	76
6.3	Configuração do Experimento Utilizando Conectores	78
6.4	Configuração do Experimento Utilizando Componente Composto	81
6.5	Diagrama de Classe de um Componente	83
6.6	Representação Gráfica em Árvore da Listagem 6.6	84
B.1	Diagrama Choice	97
B.2	Diagrama Sequence	97
B.3	Diagrama Elements	97
B.4	Notação de Diagrama de Interação	98
B.5	Notação de Diagrama de Classe	98

Lista de Tabelas

2.1	Fatores que Influenciam na Escolha de um Algoritmo na fase de DM . .	17
2.2	Exemplo de um Conjunto de Dados	22

Lista de Códigos

6.1	Descrição (compacta) em XML dos componentes	74
6.2	Descrição do Componente LOAD_DATA	75
6.3	Configuração de Experimento Utilizando xDML	76
6.4	LOAD_DATA descrito como um conector	78
6.5	Descrição (compacta) em XML dos componentes	80
6.6	Configuração de Experimentos Utilizando xDML	81
A.1	Descrição do Esquema da Linguagem xDML	91

Introdução

Cada problema que resolvi tornou-se uma regra que serviu mais tarde para resolver outros problemas.

– RENÉ DESCARTES

Desde seus primórdios, a humanidade vem tentando compreender o mundo em que vive. Relações mais simples eram explicadas a partir do senso comum, enquanto que lendas e mitos sobre deuses e demônios eram normalmente empregados para explicar o desconhecido. Milênios mais tarde, na idade antiga, com o aparecimento das primeiras cidades, grandes pensadores como Aristóteles, Sócrates e Platão iniciaram a busca pelo *arché*, o princípio de tudo, principalmente por meio do raciocínio lógico. Essa maneira de adquirir conhecimento estava baseada no pensamento de Aristóteles, que considerava que somente a lógica e a oratória eram suficientes para explicar todas as relações do mundo.

No final da idade média e início da idade moderna, o pensamento filosófico sobre a aquisição de conhecimento passa por uma profunda transformação. O espírito dessa transformação pode ser sentido principalmente no trabalho de Descartes e Galileu. Em *O Discurso do Método*, Descartes descreve seu método que, segundo ele mesmo, seria uma maneira mais satisfatória de adquirir conhecimento que a lógica aristotélica:

“Tenho tentado em minha *Dioptrique* e em meu *Météores* mostrar que o meu *Méthod* é melhor que o vulgar, e a minha *Géométrie* tem demonstrado isso.”

Apesar do pouco avanço quanto aos conhecimentos científicos por si mesmos (o trabalho de Descartes, quanto aos conhecimentos científicos, é tido por muitos como uma reescrita de idéias já estabelecidas), a importância de sua obra está na formalização de um método para adquirir conhecimento baseado na observação, anotação e análise de dados. Uma outra consequência importante do trabalho de Descartes é a possibilidade da cognição do mundo externo. Como os dados observados estão inseridos em um contexto, as relações existentes entre os fenômenos observados poderiam

estar refletidas nesses dados. Uma análise mais profunda poderia trazer à tona essas relações, em um processo de aquisição de conhecimento indutivo sobre os dados.

Já Galileu foi um dos primeiros a usar efetivamente a experimentação e a observação para comprovar ou refutar teorias, lançando as bases para a Física moderna. Galileu também foi um dos precursores da junção da técnica com a ciência, com a utilização, entre outras, do telescópio para observações astronômicas, um dos fatores importantes para a adoção do modelo copernicano¹ em relação ao modelo geocêntrico de gravitação, junção essa que resultaria em um novo tipo de ciência: a Tecnologia (etmologicamente, *tecno*: técnica, *logia*: estudo, ciência), que viria transformar radicalmente toda a sociedade a partir da revolução industrial.

A partir dessa revolução, a descoberta de conhecimento passou a ser vista de uma nova maneira. O *porque?* normalmente buscado pela lógica aristotélica passou a dar lugar ao *como?*. As observações deixaram de ser predominantemente qualitativas para serem também quantitativas. A história da ciência veio posteriormente comprovar a eficiência dessa abordagem em relação à antiga. Poucos anos mais tarde, por exemplo, Kepler travou a sua “Guerra contra Marte”² analisando dados e fazendo inferências sobre a órbita observada desse planeta. Pasteur, no século XIX, refutou a teoria da geração espontânea a partir de observações sobre bactérias e leveduras que se multiplicavam em determinadas culturas. Mendel descobriu os princípios da hereditariedade observando características das plantas.

Por outro lado, o desenvolvimento da tecnologia trouxe novas fronteiras para a aquisição de conhecimento. Desde as primeiras máquinas de somar e multiplicar inventadas por Pascal e Leibnitz, passando pela máquina de propósito geral sonhada por Babbage e Ada, a máquina de cartões de Hollerith, os primeiros computadores eletrônicos de válvulas construídas durante a 2ª guerra mundial, o UNIVAC de von Neumann, dos computadores baseados em transistors até os computadores construídos a partir de circuitos integrados, e os modernos computadores baseados em *chips* que colocam em uma única pastilha milhões de circuitos, a capacidade de processamento das máquinas vem crescendo assustadoramente.

Em conjunto com a capacidade de processamento, a capacidade de geração e armazenamento de dados também vêm crescendo em proporções semelhantes. Esta evolução pode ser vista pela maneira como o homem armazenou e repassou suas idéias através dos tempos, desde as pinturas nas paredes feitas por nossos ancestrais, passando pela invenção da escrita, do papel e da máquina de impressão de Gutenberg, até chegarmos às mídias atuais de armazenamento. Hoje em dia, estamos gradualmente nos acostumando ao fato que existem enormes volumes de dados em nossos computadores, nossas redes e nossas vidas. Agências do governo e comerciais têm dedicado enormes recursos para coletar e guardar informações. Sistemas de automação e informação vêm se tornando cada vez mais comuns na maioria das

¹O modelo copernicano é um modelo que admitia o Sol como centro do universo, tendo a Terra e os demais planetas orbitando em torno dele.

²Kepler passou anos analisando dados sobre a órbita de Marte até o enunciado de suas leis sobre gravitação. Esse episódio foi denominado por ele próprio como “guerra contra o planeta vermelho”.

empresas. A ciência tem se mostrado uma outra grande produtora de dados, como por exemplo no seqüenciamento de genomas, rastreamento do céu por potentes telescópios e radares, colisões de átomos e moléculas em aceleradores nucleares, dentre muitos outros. Novos dispositivos de armazenamento magnéticos e óticos são capazes de armazenar *gigabytes* e até *terabytes* de dados.

Mas, na verdade, apenas uma pequena quantidade desses dados serão amplamente explorados porque, na maioria das vezes, os volumes são simplesmente muito grandes para serem analisados ou as estruturas desses dados são demasiadamente complexas para serem acessadas eficientemente. Essa impossibilidade cria um impasse muito grande, uma vez que as grandes somas dispendidas na coleta e armazenamento de dados podem não trazer o retorno esperado.

A possibilidade de aquisição de conhecimento automático por meio de máquinas surgiu como alternativa a esse impasse. Máquinas pensantes têm sido um sonho que vem, há anos, povoando a mente de visionários, romancistas de ficção e cientistas.

Os primeiros passos para concretizar esse sonho foram dados em conjunto com o surgimento dos computadores eletrônicos, entre os anos de 1940 e 1950. Alan Turing, um dos principais teóricos da computação, propôs um teste para avaliar se uma máquina é inteligente. Herbert Simon e Allen Newell ([Newell and Simon, 1956](#)) assombraram o mundo quando, em 1955, anunciaram um programa capaz de demonstrar alguns teoremas matemáticos. Rosenblatt ([Rosenblatt, 1958](#)), em 1958, criou o Perceptron, um modelo de aprendizado baseado em neurônios. Esses foram alguns dos principais nomes responsáveis pelo surgimento da Inteligência Artificial.

As décadas de 1960 e 1970 ficaram marcadas pela transformação de grande euforia que se viu no início da década de 1960 em decepção no início da década de 1970. As limitações do Perceptron, apontadas por Minsky e Papert em 1969 ([Minsky and Papert, 1969](#)), a pouca versatilidade dos modelos simbólicos e os altos custos de construção de sistemas especialistas funcionaram como uma verdadeira “ducha de água fria” para os pesquisadores em Aprendizado de Máquina.

No final da década de 1970 e início da de 1980, o que se viu foi uma revitalização das pesquisas nessa área, o que colocou a questão do aprendizado como um dos principais focos em Inteligência Artificial. Logo nos primeiros anos da década de 80, o surgimento de sistemas de Aprendizado de Máquina simbólico como AQ11 ([Michalski and Stepp, 1983](#)) e ID3 ([Quinlan, 1986](#)), e a utilização de redes Perceptron de múltiplas camadas — MLP (*multi-layer Perceptron*) — com o algoritmo de aprendizado *backpropagation* ([Rumelhart, Hinton, and Williams, 1986](#)), foram os precursores de uma explosão nas pesquisas em Aprendizado de Máquina, com ramificações como aprendizado simbólico, conexionista, algoritmos genéticos, raciocínio baseado em casos, agrupamento, entre outros.

O aumento do interesse na utilização de algoritmos de Aprendizado de Máquina em bases de dados reais trouxe novas abordagens para as pesquisas nessa área. Em problemas de classificação, por exemplo, o foco principal dos primeiros sistemas de

aprendizado era obter a melhor precisão possível com a utilização de uma quantidade limitada de exemplos. Em bases de dados reais também é interessante a extração de algum tipo de conhecimento que esteja oculto nesses dados mas que a sua descoberta não seja trivial pela utilização de técnicas convencionais. Dessa maneira, além da precisão, o formato em que o conhecimento é apresentado também passou a ter grande importância, o que implica em “olhar dentro do classificador” para analisar e interpretar o conhecimento induzido.

Esses novos desafios motivaram a realização, em 1989, do primeiro *workshop* de Descoberta de Conhecimento em Bases de Dados, tendo como um dos principais trabalhos o artigo de Piatetsky-Shapiro ([Piatetsky-Shapiro, 1991](#)), que colocava essa área de pesquisa como uma área multidisciplinar, envolvendo outras áreas como Banco de Dados, Estatística e Inteligência Artificial. Esse *workshop* evoluiu posteriormente para uma conferência anual internacional, ocorrida em 1995: *The First International Conference on Knowledge Discovery and Data Mining*, na qual foi formalizado um processo de Aquisição de Conhecimento ([Fayyad, Platestsky-Shapiro, and Smyth, 1996](#)).

Apesar do grande número de pesquisas desde então, a Descoberta Automática de Conhecimento é uma área de pesquisa em aberto e cheia de desafios. Muitos desses desafios são pertinentes à própria aquisição de conhecimento a partir de dados. A aquisição de conhecimento depende, e muito, da qualidade e precisão dos dados que estão sendo utilizados (Kepler, por exemplo, só descobriu as leis da gravitação dos planetas pois tinha certeza que suas medições estavam corretas). Outro problema é que, muitas vezes, o potencial ou relevância do conhecimento induzido não é reconhecido a princípio (as descobertas de Mendel sobre a genética só foram reconhecidas quase um século após a sua publicação) ou confronto com um conhecimento já estabelecido (as teorias de Pasteur iam contra o senso comum estabelecido na época).

Outros desafios surgiram com a tentativa de automatização do processo de aquisição de conhecimento. Dentre alguns exemplos podemos citar a incorporação e utilização de conhecimento prévio, a dificuldade dos algoritmos de aprendizado trabalharem com dados incertos e as limitações das linguagens de descrição de hipóteses.

Existem também os problemas característicos da computabilidade. Enquanto o cérebro humano é capaz, muitas vezes, de resolver rapidamente problemas para os quais não se conhecem algoritmos de complexidade não-polinomial e até mesmo problemas para os quais não são conhecidos algoritmos, algoritmos de aprendizado necessitam de *biases*³ pré-definidos para induzir o conhecimento. Por outro lado, máquinas tem uma facilidade muito maior de trabalhar com grandes volumes de dados do que seres humanos.

Apesar dos desafios e dificuldades a serem vencidos, a descoberta automática de conhecimento vêm se tornando cada vez mais necessária. Ela é uma área de pesquisa altamente estimulante, e tem ajudado, cada vez mais, o homem em sua busca para compreender o mundo.

³*bias* são as decisões tomadas pelo algoritmo de aprendizado para induzir o conhecimento, além da conformidade com os dados

1.1 O Projeto DISCOVER

Nós podemos ver apenas uma pequena distância à frente, mas podemos ver lá a abundância do que precisa ser feito.

– ALAN TURING

A Descoberta de Conhecimento em Bases de Dados, tanto como uma aplicação efetiva ou como área de pesquisa, é um processo de engenharia. Ela não é algorítmica, ou seja, não existe nenhum tipo de receita ou roteiro que deve ser seguido. Ela requer experimentação, análise e comparação de diversos modelos na busca por um resultado satisfatório.

Na condução desses experimentos, atividades como transformações de formato, adaptações, execução de algoritmos, medições, entre outras, devem ser executadas um grande número de vezes.

Quando se trata de uma aplicação de descoberta de conhecimento, muitas dessas tarefas podem ser automatizadas pelo uso de sistemas integrados comerciais, tais como o Mineset™ ([Silicon Graphics, 2000](#)) e IntelliMiner™ ([IBM, 2000](#)). Geralmente, essas ferramentas tem um caráter mais exploratório e fazem uso de algoritmos e ferramentas proprietários, o que dificulta o seu uso por pesquisadores que pretendam analisar e desenvolver novos algoritmos e ferramentas.

Uma alternativa é o uso de ferramentas de domínio público, tais como *MCC++* — Machine Learning in C++ — ([Kohavi, John, Long, Manley, and Pfleger, 1994](#); [Kohavi, Sommerfield, and Dougherty, 1997](#)) e WEKA — Waikato Environment for Knowledge Analysis — ([Witten and Frank, 1999](#)). Entretanto, essas ferramentas também têm algumas características não apropriadas para o uso por pesquisadores, brevemente descritas a seguir.

MCC++ tem muitas facilidades, tais como interface para os principais algoritmos de aprendizado, formato padrão para os dados de entrada, obtenção de estatísticas de desempenho e visualização gráfica das estruturas simbólicas (classificadores) obtidas por alguns algoritmos. Entretanto, esses classificadores são tratados como uma “caixa preta”, não fornecendo uma visão única dos classificadores simbólicos que podem ser induzidos utilizando a biblioteca. A incorporação de novos aplicativos à *MCC++* também não é uma tarefa trivial, já que é necessário recompilar a biblioteca a cada novo aplicativo adicionado, além da dificuldade da adaptação do novo aplicativo aos padrões da *MCC++*. Além disso, a partir de 1995 a *MCC++* passou a ser de responsabilidade da Silicon Graphics e a última versão disponibilizada data de 1997.

WEKA, ao contrário da *MCC++*, reimplementa os algoritmos de aprendizado na linguagem Java. Essa abordagem padroniza as interfaces e produz código uniforme, facilitando a inclusão de novos aplicativos, mas as novas versões dos algoritmos originais podem não ser disponibilizadas no WEKA, pois exigem a sua conversão em código Java. Além disso, a recodificação de algoritmos sempre está sujeita a falhas,

as quais podem causar um comportamento anômalo do algoritmo reimplementado que não ocorre no código fonte original.

Esses fatores conduzem, muitas vezes, à necessidade de utilização dos algoritmos de aprendizado tal qual como foram implementados pelos seus idealizadores e, por conseguinte, todas as atividades necessárias para a execução de experimentos devem ser feitas para cada algoritmo. Essa necessidade implica, muitas vezes, no desenvolvimento de programas para automatizar essas tarefas.

Muitos pesquisadores em nosso laboratório de pesquisa (LABIC⁴ — Laboratório de Inteligência Computacional) utilizam esses algoritmos em suas pesquisas e, muitas vezes, reimplementavam algum tipo de código semelhante para a realização de seus experimentos. O projeto DISCOVER surgiu como uma tentativa de reutilizar esses códigos implementados, de maneira que novas pesquisas relacionadas pudessem ser conduzidas sem a necessidade de recriar esses programas.

Surgiu então, a partir de dois alunos de doutorado, a proposta de criar um projeto no qual todos os membros do laboratório estariam envolvidos, o projeto DISCOVER (Baranauskas and Batista, 2000).

1.1.1 O Sistema DISCOVER

Uma das primeiras tentativas do projeto DISCOVER de reutilizar alguns desses programas foi a implementação, na forma de *scripts* e na linguagem PERL (Wall, Christiansen, and Orwant, 2000) — Practical Extraction and Report Language — de uma série de ferramentas para facilitar a configuração e execução dos experimentos (Prati, Baranauskas, and Monard, 2001b,a; Batista, 2001; Kemp, Batista, and Monard, 2001).

A princípio, o projeto DISCOVER consistiria apenas de um repositório de *scripts*. De uma maneira geral, *scripts* são pequenos programas que realizam tarefas atômicas. Por meio da combinação desses *scripts* independentes seria possível a realização de tarefas mais complexas. Desta maneira, o projeto consistiria de uma série de ferramentas independentes que poderiam ser compostas conforme as necessidades dos usuários, e reunidas em repositório compartilhado entre esses usuários.

Posteriormente surgiu a proposta de se criar um ambiente integrado, o sistema DISCOVER, no qual os *scripts* seriam substituídos por bibliotecas de classes e essas classes empacotadas como componentes, com a composição dos componentes sendo feita por meio de uma interface gráfica.

Dessa forma, os objetivos do projeto DISCOVER passaram a ser, além de reutilizar os programas desenvolvidos por outros pesquisadores, construir um sistema computacional capaz de utilizar alguns desses programas, de tal maneira que pela utilização de uma ferramenta fosse possível construir experimentos pela composição desses componentes (Prati and Monard, 2002; Prati et al., 2003).

⁴<http://labic.icmc.sc.usp.br>

Essa nova abordagem, além de benefícios tais como uma maior facilidade de utilização e melhor performance (com a utilização de *scripts* não é possível o compartilhamento de memória, causando uma sobrecarga com a leitura/escrita), trouxe também novos desafios para a execução do projeto. Para se tornar um ambiente integrado, várias questões que dizem respeito ao gerenciamento do processo, comunicação entre os participantes e interação entre os diferentes componentes devem ser respondidas.

Além disso, um outro ponto importante diz respeito à arquitetura do sistema. Várias considerações devem ser feitas quanto à integração dos componentes, além de um conjunto de processos que suportam e criam um ambiente de execução apropriado para o sistema.

No DISCOVER serão utilizados os algoritmos de aprendizado implementados pela comunidade e ferramentas com finalidades específicas, desenvolvidas pelos pesquisadores relacionados ao projeto, tais como ferramentas de processamento de dados e de textos tanto para aprendizado supervisionado quanto para aprendizado não-supervisionado, amostragem e avaliação de erro, mesclagem de regras, cobertura de regras, entre outros.

O desenvolvimento de novas ferramentas, tanto de processamento de dados, descoberta e processamento de padrões envolvem a manipulação de alguns objetos em comum. Por exemplo, uma ferramenta de aquisição de dados a partir de sistemas Web ou uma ferramenta de amostragem de dados para reamostragem, apesar de pertencerem a tarefas diferentes (aquisição de dados e amostragem de dados), têm em comum a manipulação de dados.

Além disso, esses objetos podem estar representados em diferentes formatos. Por exemplo, uma data pode estar armazenada como 'dia/mês/ano' em uma determinada base de dados e como o número de dias contados a partir de uma determinada data em outra base. Um outro exemplo são os diversos formatos que os algoritmos de aprendizado simbólico usam para representar os classificadores, ou hipóteses, por eles induzidos.

De uma maneira geral, o sistema DISCOVER pode ser entendido como um conjunto de métodos que são aplicados sobre os dados ou sobre o conhecimento extraído a partir dos dados. Dessa maneira, é muito importante que o sistema DISCOVER ofereça uma base sólida para manipular dados e conhecimento. Essa base é composta por sintaxes padrões para a representação de dados e conhecimento, e por bibliotecas que oferecem um conjunto de funcionalidades básicas de manipulação de dados e conhecimento. Atualmente, existem definidas sintaxes padrões para a representação de dados e para a representação do conhecimento de diversos indutores simbólicos, e bibliotecas que oferecem funcionalidades sobre essas sintaxes padrões. Futuramente, novas sintaxes padrões devem ser especificadas, principalmente para a representação de regras de regressão (Pugliesi, 2001; Dosualdo, 2002), regras de associação (Melan-da, 2002) e agrupamentos (Martins, 2001).

Para os dados, o formato padrão adotado é similar ao formato de dados da *MCC++*, mas com extensões para aprendizado não-supervisionado e Text Mining, além da inclusão de novos tipos de dados e atributos derivados, ou seja, atributos calculados a partir de atributos originais (Batista, 2001).

A princípio, o sistema DISCOVER terá como principal objetivo fornecer um campo de prova para os pesquisadores de nosso laboratório. A vantagem do DISCOVER como ferramenta de apoio à pesquisa a descoberta de conhecimento, em relação a outros sistemas, é a visão unificada que os formatos padrão oferecem para quem está desenvolvendo novos componentes, além de um conjunto de ferramentas para a sua manipulação.

Uma outra vantagem que se espera é que o sistema DISCOVER ofereça uma série de vantagens em relação à composição das ferramentas desenvolvidas em nosso laboratório, evitando que essas implementações se percam e, além disso, reunindo-as em um ambiente integrado.

1.1.2 Trabalhos Realizados e em Desenvolvimento

Como já mencionado, diversas ferramentas e sistemas que compõem o DISCOVER já estão implementadas, utilizando a linguagem PERL (Wall, Christiansen, and Orwant, 2000). Uma biblioteca para a manipulação de conjuntos de regras (classificadores) no formato padrão do DISCOVER foi desenvolvida (Prati et al., 2002), sendo que já estão disponíveis ferramentas que fazem a conversão de sintaxe entre conjuntos de dados em diversos formatos para o formato padrão de dados e ferramentas que transformam os classificadores simbólicos para o formato padrão de regras (Prati, Baranauskas, and Monard, 2001b).

Uma vez convertido no formato padrão, um classificador simbólico pode ser avaliado por uma outra ferramenta também disponível (Prati, Baranauskas, and Monard, 2001a, 2002), com um conjunto de dados, obtendo um conjunto de medidas, também em um formato padrão. Essas medidas incluem os valores da matriz de contingência, calculada para cada regra, de maneira que métricas de qualidade de regras possam ser facilmente obtidas (Lavrač, Flach, and Zupan, 1999). Também estão disponíveis ferramentas para obtenção de estatísticas de desempenho de classificadores utilizando métodos de reamostragem (Kemp, Batista, and Monard, 2001).

Um conjunto de ferramentas para processamento de dados vem sendo desenvolvida, sendo que já estão disponíveis ferramentas para cálculo de estatísticas descritivas básicas (máximo, mínimo, média, desvio padrão, número de valores desconhecidos, etc.), remoção, troca de posição e criação de novos atributos a partir de expressões, embaralhamento, amostragem aleatória e complementar, e filtros de exemplos a partir de expressões, entre outros (Batista and Monard, 2003b).

Uma outra ferramenta implementada é o sistema XRULER (Baranauskas and Monard, 2000b, 2003; Baranauskas, 2001), no qual um conjunto de exemplos é dividido em amostras de treinamento e teste. As amostras de treinamento são submetidas

a indutores diferentes e os classificadores obtidos são convertidos para o formato padrão de regras. As regras no formato padrão são avaliadas e um algoritmo de cobertura é utilizado, gerando um subconjunto das regras que cobrem os exemplos obtendo, dessa maneira, um classificador simbólico final.

Algumas outras pesquisas realizadas e em curso em nosso laboratório também resultaram em diversos trabalhos e idéias sobre novas ferramentas para o DISCOVER, processamento de dados (Batista and Monard, 2003a; Batista et al., 2000; Batista and Monard, 2001; Batista, 2003) e texto (Martins, 2001; Imamura, 2001), Aprendizado de Máquina (Monard and Batista, 2002; Pila and Monard, 2002, 2001; Pila, 2001; Bernardini, 2002; Gomes, 2002) e processamento de padrões (Batista, 1997; Baranauskas and Monard, 2003; Lee, 2000; Milaré et al., 2003; Martins et al., 2002; Baranauskas and Monard, 2000b,a; Baranauskas, 2001; Milaré, 2000; Paula, 2003; Melanda, 2002; Pugliesi, 2001).

1.2 Objetivos e Contribuições desta Dissertação

Se os meus escritos valem alguma coisa, possam os que os tiverem (...) utilizá-los do melhor modo que entenderem.

– RENÉ DESCARTES

Embora a idéia do DISCOVER seja aparentemente simples, sua implementação não é trivial. Como o DISCOVER tem como principal finalidade o seu uso e extensão por pesquisadores, a questão principal é que a arquitetura do projeto deve ser flexível o suficiente para permitir que novas pesquisas sejam englobadas e, simultaneamente, deve impor determinados padrões que permitam a integração eficiente de seus componentes, além de criar um ambiente de execução propício para a execução desses componentes.

O objetivo principal deste trabalho é a construção de um *framework* de integração dos componentes no projeto DISCOVER, de tal maneira que seja possível criar o sistema DISCOVER. Entretanto, o *framework* é apenas o resultado final de um processo de análise e elaboração de uma arquitetura para o sistema, com a definição de suas estruturas e modelos de componentes que serão utilizados.

Essa definição da arquitetura passa pela confrontação das ferramentas e funcionalidades já implementadas, os propósitos do sistema e das restrições (tanto de natureza técnica quanto do ambiente de desenvolvimento) com os conceitos de qualidade de arquitetura de *software*, e, principalmente, estilos e *patterns* de arquitetura.

1.3 Notas sobre Tradução e Terminologia

Traduções (como as esposas) raramente são fiéis se elas são, no mínimo, atraentes.

– ROY CAMPBELL

As abordagens de *Software Pattern* e *Frameworks* têm sido desenvolvidas em inglês. Como esta dissertação tem um caráter introdutório e deve servir como referência para um projeto de um sistema na área de Descoberta de Conhecimento e, na maioria das vezes, por pesquisadores em Inteligência Artificial, optou-se em manter os termos *pattern* e *framework* em inglês, devido às considerações descritas a seguir.

Em português, a tradução de *pattern* deveria ser para o termo “padrão”. Porém, esse termo já está associado a uma série de significados, tais como padrões de comportamento, normas (como a ISO 9000 por exemplo), e reconhecimentos de padrões de sinais. Apesar do significado do termo nesta abordagem compartilhar elementos desses outros significados, mas em níveis de profundidade e ênfase diferentes, é melhor, principalmente em uma introdução, não associar nenhum outro significado que o termo possa induzir. Então, recomenda-se que o leitor entenda o termo “*pattern*” como um termo novo. A manutenção do termo em inglês ajuda nessa dissociação.

A tradução de *framework* deveria ser para o termo “arcabouço”. Entretanto, como esse termo não vem sendo utilizado pela comunidade de Engenharia de *Software*, optou-se pelo termo em inglês para manter a associação do termo à abordagem.

Alguns outros termos foram traduzidos para o português, mas os acrônimos utilizados vêm dos termos em inglês. Esses acrônimos foram mantidas nessa forma por serem amplamente utilizados pela comunidade.

Deve ser observado que a utilização desses termos e acrônimos não se trata, em nenhum momento, de desrespeito à língua portuguesa, e somente foram utilizados pelas considerações acima apresentadas.

Uma outra menção refere-se ao uso da locução algoritmo de aprendizado. O dicionário Aurélio define algoritmo como o “*processo de cálculo, ou resolução de um grupo de problemas semelhantes, em que se estipulam, com generalidade e sem restrições, regras formais para a obtenção de resultados, ou da solução do problema*”. Entretanto, disseminou-se na comunidade de Aprendizado de Máquina a utilização do termo algoritmo de aprendizado como um sistema computacional que implementa um algoritmo de aprendizado. Portanto, o termo algoritmo de aprendizado deve ser entendido como um sistema computacional que implementa um algoritmo para aprendizado de máquina.

1.4 Organização

Um lugar para tudo, e tudo em seu lugar.

– SAMUEL SMILES

O restante desta dissertação está organizado da seguinte forma: no Capítulo 2 é apresentado uma ferramenta genérica de KDD, com o objetivo de criar um modelo teórico para um sistema desta natureza. No Capítulo 3 é apresentada uma introdução à Arquitetura de *Software*. No Capítulo 4 são apresentados os conceitos de componentes, *frameworks* e *software patterns*. No Capítulo 5 é apresentado o *framework* proposto neste trabalho, enquanto que no Capítulo 6 é apresentado um exemplo da sua execução. Finalmente, no Capítulo 7 são apresentadas as conclusões desta dissertação.

Um Sistema para Descoberta de Conhecimento

O segredo do sucesso é conhecer algo que ninguém mais conhece.

– ARISTÓTELES ONASSIS

Em uma grande variedade de campos de atuação humana, dados são coletados e armazenados na expectativa de que informações úteis (conhecimento) possam ser extraídas a partir desses dados. Tradicionalmente, a transformação de dados em conhecimento baseia-se na análise e interpretação manual desses dados.

Em geral, esses métodos clássicos de extração de conhecimento consistem em um (ou mais) analista humano gerando e comparando relatórios, cruzando informações e usando o seu conhecimento prévio sobre a área de interesse na tentativa de extrair algo novo dos dados existentes.

Caso esses dados estejam em uma base de dados computadorizada, algumas tarefas tais como consultas e geração de relatórios e gráficos podem ser automatizadas pelo uso de planilhas eletrônicas, sistemas gerenciadores de bases de dados, sistemas de informação, entre outros. Um outro paradigma recente para o processamento de informações, mas que ainda não é capaz de extrair conhecimento utilizando processos indutivos automáticos, é o Processamento Analítico *On-line* — OLAP (*On-Line Analytical Processing*) — focado na análise de dados de uma maneira multi-dimensional, com os dados geralmente proveniente de *Data Warehouses* (Berson and Smith, 1997).

Mesmo com o apoio de ferramentas computacionais, a maneira manual de tratamento de dados pode ser lenta, cara e altamente subjetiva. Soma-se a isso o fato que o crescimento desses bancos de dados torna essa abordagem de extração de conhecimento cada vez mais impraticável. É muito improvável que um ser humano consiga analisar e inferir algo importante de uma base de dados que possua milhões de re-

gistros, com cada um desses registros contendo centenas ou até mesmo milhares de campos.

Esses fatores impõem a necessidade de uma nova geração de técnicas e ferramentas computacionais que auxiliem analistas na extração de conhecimento de uma maneira mais automática. Essas técnicas e ferramentas são o objeto de estudo da Extração de Conhecimento de Bases de Dados — KDD (*Knowledge Data Discovery*) — , uma área de pesquisa relativamente nova em Ciência da Computação, que combina elementos de áreas como Inteligência Artificial, Banco de Dados, Estatística, Visualização, entre outros.

O objetivo deste capítulo é estudar os requisitos de um sistema genérico para KDD. No contexto deste trabalho, não são considerados algoritmos e ferramentas para a mineração visual de dados, mas a utilização de técnicas de aquisição de conhecimento implícita ou automática, que visem adquirir conhecimento a partir dados e/ou informações referentes a um domínio por meio de, na maioria das vezes, sistemas de Aprendizado de Máquina.

Quanto à organização, este capítulo está dividido da seguinte forma: na Seção 2.1 é apresentado o processo de descoberta semi-automática de conhecimento em bases de dados, KDD. Na Seção 2.2 é mostrada a transformação do processo de KDD idealizado em procedimentos computacionalmente executáveis. Na Seção 2.3 é apresentado um modelo genérico para um sistema de KDD. Na Seção 2.4 é discutida uma possível melhoria do modelo, pelo uso de meta-aprendizado e, finalmente na Seção 2.5 é apresentado um resumo dos pontos principais deste capítulo.

2.1 O processo de Descoberta de Conhecimento

A essência do conhecimento é, tendo-o, aplicá-lo, não tendo-o, confessar sua ignorância.

– CONFÚCIO

Fayyad, Platestsky-Shapiro, and Smyth (1996) definem Descoberta de Conhecimento em Base de Dados como

“o processo não trivial de identificação de padrões válidos, novos, potencialmente úteis e compreensíveis em conjuntos de dados.”

no qual dados (também chamados de exemplos, na literatura de Aprendizado de Máquina) são um conjunto de fatos referentes ao domínio, e padrões são expressões, em alguma linguagem, que descrevem de uma maneira mais compacta um subconjunto desses dados, ou um modelo aplicável a esse subconjunto de dados.

O termo *processo* implica que KDD é composto por um conjunto de passos (Fayyad, Platestsky-Shapiro, and Smyth, 1996; Weiss and Indurkha, 1998) que podem ser agrupados em três fases (Baranauskas, 2001):

1. preparação e redução do conjunto de dados (processamento de dados);
2. descoberta de padrões (Mineração de Dados) e
3. avaliação, refinamento e disponibilização do conhecimento (processamento de padrões).

Uma representação esquemática do processo de KDD pode ser vista na Figura 2.1.

Figura 2.1: Etapas no Processo de KDD

O pré-processamento dos dados é a etapa mais crucial em todo o processo de KDD e pode facilmente consumir 80% de todo o tempo necessário para a aquisição de conhecimento (Mannila, 1996). Entender o domínio dos dados é um pré-requisito natural para a extração de qualquer informação útil, uma vez que bases de dados reais geralmente apresentam características que impedem a aplicação imediata das ferramentas de mineração de dados, tais como dados dinâmicos, ruído, incerteza, dados incompletos, informações redundantes, dados esparsos e em grande volume (Matheus, Chan, and Piatetsky-Shapiro, 1993). A preparação dos dados envolve tarefas como a definição dos atributos que serão utilizados, extração e integração dos dados provenientes de diferentes fontes, transformação dos dados para formatos mais apropriados para a extração de conhecimento, limpeza e composição de atributos. A redução do conjunto de dados justifica-se quando o uso de grandes bases de dados pode exceder a capacidade de processamento do algoritmo de aprendizagem, ou como uma tentativa de diminuição do tempo de resposta para se encontrar uma solução.

A Mineração de Dados (Rezende et al., 2003) —DM (*Data Mining*)— é a etapa onde os padrões interessantes e que ocorrem mais frequentemente são descobertos a partir dos dados. Esta etapa pode usar várias técnicas de Estatística, tais como modelos lineares e paramétricos, ou Aprendizado de Máquina (Monard and Baranauskas, 2003a) —AM—, tais como indução de regras, árvores de decisão (Monard and Baranauskas, 2003b), agrupamento (*clustering*), redes neurais, programação lógica indutiva e métodos baseados em distância. Neste trabalho, o termo Mineração

de Dados refere-se apenas à etapa de descoberta de padrões dentro do processo de KDD. Em outros lugares, especialmente na indústria, Mineração de Dados é frequentemente usado como sinônimo de KDD.

O processo de KDD não termina quando os padrões são encontrados. Os padrões descobertos devem ser entendidos, avaliados, confrontados com o conhecimento já existente, ordenados, filtrados e traduzidos em termos inteligíveis para os usuários. A seguir, tem-se a incorporação do conhecimento extraído, seja pela documentação e disponibilização desse conhecimento às partes interessadas ou pela tomada de decisões baseadas nesse conhecimento.

2.1.1 Interatividade no Processo de KDD

Não é possível extrair conhecimento útil simplesmente submetendo um conjunto de dados a um sistema automático, no qual os dados são apresentados e o conhecimento é obtido sem a intervenção do usuário. Além disso, o processo de KDD é orientado à meta, ou seja, cada aplicação tem um conjunto de requisitos que leva a uma especificação para o problema.

Isso torna o processo de KDD essencialmente interativo e dependente do usuário ([Brachman and Anand, 1996](#)). O usuário de um sistema de KDD deve usar o seu conhecimento do domínio para conduzir todo o processo de extração de conhecimento, selecionando as tarefas de processamento de dados, a função e o algoritmo que serão utilizados na mineração de dados e estabelecendo os critérios de qualidade e interesse dos padrões encontrados.

Dessa maneira, um sistema de KDD deve ser visto como um conjunto de ferramentas interativas e não como um sistema de análise automática ([Mannila, 1997](#)).

2.1.2 Iteratividade no Processo de KDD

Uma vez definidas as tarefas a serem realizadas, o processo de KDD se transforma em um conjunto de procedimentos iterativos, que vão transformar, limpar, selecionar e preparar os dados; aplicar os algoritmos de aprendizado; transformar e classificar os padrões obtidos em diferentes formatos ([Klemettinen, Mannila, and Toivonen, 1996](#)); etc.

O processo de KDD pode envolver também uma iteração significativa entre as fases, de tal maneira que os resultados obtidos em uma iteração podem ser usadas para o refinamento de uma próxima iteração ([Brodley and Smyth, 1995](#)).

Além disso, alguns métodos estatísticos computacionalmente intensivos de estimativa de parâmetros, tais como o método de validação cruzada (*cross-validation*), que utiliza reamostragem (*resampling*) de exemplos para, por exemplo, estimar a taxa de erro, requerem a execução do algoritmo de aprendizado diversas vezes com diferentes amostras de exemplos ([Cohen, 1995](#)).

Um suporte eficiente para essas iterações é um importante tópico para o desenvolvimento de sistemas de KDD ([Klemettinen, Mannila, and Toivonen, 1997](#)).

2.1.3 KDD é um Processo Empírico

Uma das tarefas mais críticas em todo o processo de KDD é a escolha do modelo/algoritmo que será utilizado na fase de DM. Para um dado domínio, não existem garantias de que um determinado algoritmo seja necessariamente melhor que outro, ou seja, não existe uma análise matemática capaz de determinar qual algoritmo de aprendizado terá um melhor desempenho em relação aos outros ([Kohavi, Sommerfield, and Dougherty, 1997](#)).

Na escolha do modelo/algoritmo, devem ser levados em consideração fatores que dizem respeito à natureza da aplicação, tais como os objetivos pretendidos, conhecimento prévio sobre o domínio e restrições operacionais como tempo, *software*, *hardware*, etc; fatores sobre os dados disponíveis, tais como sua linguagem de representação e representatividade dos exemplos; e fatores humanos, tais como quem serão os usuários finais, os especialistas no domínio e analistas responsáveis pelo KDD. Alguns desses fatores estão resumidos na Tabela 2.1.

Aplicação	Dados	Humanos
Objetivos	Representação	Analista
Conhecimento Prévio	Exemplos	Especialista
Operacionais		Usuário Final

Tabela 2.1: Fatores que Influenciam na Escolha de um Algoritmo na fase de DM

Entretanto, esses fatores apenas oferecem algumas heurísticas sobre a escolha do algoritmo. O uso dessas heurísticas muitas vezes fornece apenas a indicação de um grupo de algoritmos que podem ser utilizados. A verificação de quais algoritmos, dentro desse grupo, apresentam melhor desempenho, para esse conjunto de dados, em relação aos outros, requer experimentação e comparação entre os modelos/algoritmos desse grupo.

Nos modelos/algoritmos também é possível um ajuste de vários parâmetros, na tentativa de se encontrar a melhor configuração para aquele domínio específico. Aqui, mais uma vez, heurísticas podem ser utilizadas, mas quase sempre o melhor ajuste de parâmetros é conseguido por experimentação.

Os experimentos realizados na tentativa de se encontrar a melhor configuração para uma dada aplicação podem ser vistos como uma busca num espaço de estados, consistindo de combinação de parâmetros e opções. Os nós deste espaço, representando as possíveis combinações, podem ser tentadas em sequência e a variação que fornece os melhores resultados é selecionada ([Baranauskas, 2001](#)).

2.2 Transformando o Processo de KDD em Procedimentos

Se você não souber como fazer alguma coisa, você não vai saber como fazer essa mesma coisa com um computador.

– ANÔNIMO

O processo de KDD admite várias visões, dependendo da funcionalidade desejada. Na transformação do processo idealizado em procedimentos, KDD pode ser descrito em termos de um processo hierárquico (Chapman, Clinton, Kerber, Khabaza, Reintartz, Shearer, and Wirth, 2000), consistindo de um conjunto de tarefas descritas em quatro níveis de abstração (do geral ao específico): fases, tarefas genéricas, tarefas especializadas e procedimentos, como é mostrado na Figura 2.2.

Figura 2.2: KDD Visto Como um Processo Hierárquico

No primeiro nível, o processo de KDD é organizado em fases. Essas fases são as mesmas descritas na Seção 2.1 na página 14. Cada fase consiste em diversas tarefas genéricas no segundo nível. O segundo nível é chamado genérico pois precisa cobrir todas as possíveis situações do processo de KDD. Por exemplo, a fase de Mineração de Dados inclui tarefas genéricas como classificação, categorização, segmentação, entre outras.

No terceiro nível — tarefas especializadas — são descritas quais as ações, em determinadas situações específicas, devem ser tomadas para realizar as ações do nível genérico. Por exemplo, se no segundo nível houver uma tarefa genérica denominada *limpeza de dados*, no terceiro nível são descritas as ações, nas diferentes situações que essa tarefa pode ser realizada, tais como a limpeza de atributos numéricos *versus* a limpeza de atributos categóricos; se os valores serão aglomerados; ou se alguma técnica de predição será usada para atributos com valores desconhecidos.

O quarto nível, o de procedimento, é um registro das ações, decisões e resultados da configuração atual do processo de KDD. Os procedimentos são organizados de acordo com as tarefas definidas nos níveis precedentes, mas representam o que

ocorre em uma configuração particular. Nesse nível, é descrito todo o processo de KDD, incluindo as bases de dados utilizadas, técnicas e algoritmos utilizados (com os respectivos parâmetros utilizados), e os resultados e medições obtidos.

A transformação do processo de KDD em procedimentos que possam ser executados por um computador passa pelo mapeamento dos níveis de fase e tarefa genéricas para os níveis de tarefas especializadas e procedimentos. Esse mapeamento não é uma operação simples, uma vez que o processo de KDD envolve atividades tanto iterativas (que envolvam escolhas e necessitam de interferência do usuário) como iterativas (que envolvam a realização de uma determinada tarefa um número repetitivo de vezes).

Além disso, a descrição de fases e tarefas como um conjunto discreto de passos em uma ordem específica, por meio de uma sequência ordenada de eventos, é apenas uma visão idealizada do processo, uma vez que, na prática, algumas atividades exigem o retorno a uma etapa anterior e a repetição de algumas ações. Por exemplo, uma atividade de processamento de dados que envolva a predição de valores desconhecidos pelo uso de algum algoritmo de aprendizado envolve uma sequência de atividades distribuídas nas três fases do processo de KDD.

2.3 Um Sistema de Descoberta de Conhecimento

Descoberta é encontrar algo previamente desconhecido. Um sistema de descoberta de conhecimento é, então, um sistema que encontra conhecimento que anteriormente não possuía.

– MATHEUS, CHAN, AND PIATETSKY-SHAPIRO (1993)

Para [Matheus, Chan, and Piatetsky-Shapiro \(1993\)](#), um sistema de descoberta de conhecimento

“compreende uma coleção de componentes que juntos podem eficientemente identificar e extrair novos padrões interessantes e úteis a partir de dados armazenados em bases de dados reais.”

Nessa definição, um sistema é caracterizado como uma coleção de componentes. Uma vez que esses componentes podem variar entre sistemas diferentes, certas funções básicas podem, usualmente, ser identificadas. Na Figura 2.3, baseada na definição de ([Matheus et al., 1993](#)), é apresentado um modelo de um sistema genérico de KDD, em que quatro classes de componentes podem ser identificadas:

Controle: controla a invocação e parametrização dos outros componentes, de acordo com a entrada do usuário.

Processamento de Dados: conjunto de componentes que processam dados para serem usados pelos componentes de extração de padrões.

Extração de Padrões: conjunto de algoritmos de extração de padrões.

Processamento de Padrões: conjunto de componentes que processam e avaliam os padrões descobertos pelos componentes de extração de padrões.

Figura 2.3: Modelo de uma Ferramenta Genérica para KDD

O fluxo de informação (representado pelas setas na Figura 2.3) pelo sistema segue a seguinte ordem: o usuário seleciona os comandos, em alto nível, no controle. Os componentes de processamento de dados processam os dados, deixando-os no formato apropriado para as ferramentas de extração de padrões. Os padrões são extraídos a partir dos dados e processados para que possam ser apresentados ao usuário.

Esse modelo representa uma abstração do que acontece em um sistema de KDD. Em muitos sistemas reais, não é possível separar claramente os componentes individuais do modelo. Entretanto, essa abstração é útil na comparação de abordagens e serve como um guia para o projeto de novos sistemas. O modelo também é particularmente útil para fazer um contraste do compromisso entre autonomia (isto é, o quão dependente o sistema é das decisões do usuário) e versatilidade (isto é, a faixa de domínios e diversidade de algoritmos de aprendizado e componentes de processamento disponíveis). A linha no gráfico da Figura 2.4 (Matheus, Chan, and Piatetsky-Shapiro, 1993) descreve o compromisso entre autonomia e versatilidade. Um sistema muito autônomo tem a tendência de ser pouco versátil, pois geralmente tem poucos algoritmos de aprendizado e poucos componentes de processamento de dados e padrões, reduzindo a faixa de domínios em que pode ser utilizado. Por outro lado, um sistema muito versátil geralmente possui várias técnicas de extração de padrões e componentes de pré e pós-processamento, o que aumenta a faixa de domínios em que o sistema pode ser utilizado, mas requer muito mais do usuário no que diz respeito à condução do processo. A meta de um sistema de KDD aparece no ponto do gráfico onde ambos, autonomia e versatilidade, são maximizados em um sistema ideal. Os componentes nesse modelo de sistema são discutidos nas subseções seguintes.

Figura 2.4: Compromisso entre Autonomia e Versatilidade em Sistemas de KDD

2.3.1 Controlando a Descoberta

As informações de controle são baseadas nas informações passadas pelo usuário. O controle interpreta essas informações e as usa para direcionar o processamento dos dados, extração e processamento de padrões. Na maioria dos sistemas, a tarefa de descobrimento é bem definida e estática, e o controle simplesmente executa uma sequência pré-definida de operações. Em uma nova classe de sistemas que utilizam meta-aprendizado para auxiliar na direção do processo, que será discutida na Seção 2.4, o controle pode assumir maiores responsabilidades de decisão sobre o processo. O controle também é responsável por registrar as ações, os resultados e as medições de desempenho.

Um ponto importante no controle é a interface de comunicação com o usuário, pois por meio dessa interface é que o usuário terá toda a percepção do sistema. O estudo de interfaces com o usuário é tema de pesquisa da área Interação Usuário Computador — HCI (*Human Computer Interaction*) (Faulkner, 1998).

Da mesma maneira, a apresentação dos resultados também é um importante problema em KDD, pois descobertas que não podem ser eficientemente comunicadas não podem ser efetivamente utilizadas.

2.3.2 Processando Dados

Como já foi dito na Seção 2.1, o processamento de dados é a etapa mais crucial em todo o processo de KDD. Os algoritmos de aprendizado geralmente olham os dados de uma maneira uniforme, e trabalham com eles em um formato padrão. O formato padrão é composto por um conjunto de exemplos (ou casos) com valores de atributos, geralmente representado por uma tabela na forma atributo-valor. Quando esses casos já estão pré-classificados, existe um atributo especial que corresponde à classe associada ao exemplo. Na Tabela 2.2 é apresentado o formato padrão de um conjunto de exemplos T com n exemplos e m atributos pré-classificados. A última coluna, Y , corresponde ao atributo classe para os exemplos correspondentes.

	X_1	X_2	$\dots$	X_m	Y
T_1	x_{11}	x_{12}	$\dots$	x_{1m}	y_1
T_2	x_{21}	x_{22}	$\dots$	x_{2m}	y_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
T_n	x_{n1}	x_{n2}	$\dots$	x_{nm}	y_n

Tabela 2.2: Exemplo de um Conjunto de Dados

Um outro ponto importante no processamento de dados diz respeito à qualidade e representatividade dos dados. O processamento de dados pode então ser dividido em dois grupos ([Weiss and Indurkha, 1998](#)):

1. Organizar os dados no formato padrão para que esses dados estejam prontos para a aplicação dos algoritmos de aprendizado.
2. Transformar os dados para produzir atributos e exemplos com um bom poder de previsão.

O primeiro grupo engloba atividades de seleção, extração, integração e transcrição dos dados brutos em dados no formato padrão. No segundo grupo, estão atividades de transformação dos dados em formatos mais apropriados para a extração de conhecimento, tais como limpeza visando melhorar a qualidade dos dados, e composição de atributos visando melhorar o poder preditivo dos atributos.

Como foi dito na Seção 2.1, em alguns casos uma redução nos dados deve ser considerada para atender a requisitos de tempo e memória do algoritmo de aprendizado. Essa redução pode ser feita em cada uma das três dimensões do conjunto de dados: remoção de exemplos, remoção de atributos ou redução do número de valores de um atributo (pela suavização, discretização ou agrupamento de valores de um atributo).

As tarefas de processamento de dados são, em geral, muito específicas para cada aplicação de KDD, o que resulta, na prática, em um reconhecimento do domínio da aplicação, considerando o conhecimento prévio relevante e os objetivos da aplicação.

2.3.3 Procurando Padrões

Os componentes que fazem a descoberta de padrões podem ser divididos em dois grupos: predição, onde o sistema procura padrões com o propósito de prever uma característica futura de algo e descrição, onde o sistema procura padrões com o propósito de apresentá-los ao usuário em uma linguagem que ele possa interpretar.

Apesar da divisão entre predição e descrição não ser muito clara (muitos dos modelos de predição podem ser descritivos, no sentido de serem interpretáveis pelo usuário, e vice versa), a distinção é útil para entender os objetivos da tarefa de descoberta. A importância relativa entre predição e descrição para as aplicações de Mineração de Dados podem variar consideravelmente. Entretanto, no contexto de KDD, descrição tem a tendência de ser mais importante que predição.

Os objetivos de predição e classificação são conseguidos pelos seguintes métodos de mineração ([Fayyad, Platestsky-Shapiro, and Smyth, 1996](#)):

Classificação: Aprender uma função que mapeie (classifique) um dado item em uma classe, dentro de um número pré-definido de classes.

Regressão: Aprender uma função que mapeie um dado item em um valor real.

Agrupamento: Identificar um conjunto finito de categorias, ou grupos, que descrevem os dados.

Sumarização: Procurar uma descrição compacta para um subconjunto de dados.

Modelagem de dependência: Procurar um modelo que descreva dependências significativas entre variáveis.

Deteção de mudança e desvio: Descobrir as mudanças mais significativas nos dados a partir de valores previamente medidos ou normalizados.

2.3.4 Processando Padrões

Dados são repletos de padrões, mas apenas uma parte deles é interessante. Um padrão é interessante quando é preciso, novo e útil no que diz respeito aos conhecimentos e objetivos do usuário final. Os componentes de processamento de padrões determinam o grau relativo de interesse dos padrões extraídos, selecionam e ordenam esses padrões para serem apresentados ao usuário.

Significância estatística é usualmente um fator chave na determinação de interessabilidade. Se um certo padrão não pode ser mostrado como sendo válido com algum grau de certeza, ele não é muito útil, e também não é muito interessante. Para muitos padrões, a porcentagem de cobertura ou o grau de precisão podem ser medidas suficientes. Se os padrões forem extraídos a partir de amostras dos dados, uma medida de confiança também pode ser necessária para comparar como estatisticamente esses padrões são suportados na população inteira. Várias outras medidas sobre interessabilidade de padrões podem ser encontradas em ([Freitas, 1998a,b, 1999](#)).

Para outros padrões, entretanto, somente a significância estatística não é suficiente para determinar o seu grau de interesse. Por exemplo, um aumento de 5% nas vendas de um determinado produto em uma região pode ser mais interessante que um aumento de 50% em outra região, desde que o volume de vendas desse produto na primeira região seja muito maior do que na segunda. Nesse caso, o “impacto” do primeiro padrão aumenta o seu interesse relativo. Os fatores específicos que influenciam o impacto e a interessabilidade de um padrão podem variar para diferentes dados e tarefas, requerendo portanto um maior conhecimento do domínio ([Matheus, Chan, and Piatetsky-Shapiro, 1993](#)).

2.3.5 Outras Características Desejáveis em um Sistema de KDD

O modelo apresentado anteriormente ilustra as características funcionais de um sistema de KDD. Nesta subseção são apresentadas algumas outras características desejáveis em um sistema desse tipo, tais como:

Extensibilidade: como KDD é um processo que envolve a experimentação com vários algoritmos de aprendizado e ferramentas, e novos componentes desse tipo estão sempre sendo desenvolvidos, a arquitetura do sistema deve ser planejada de maneira que a inclusão de novos componentes seja feita de uma maneira fácil (Wrobel, Wettschereck, Sommer, and Emde, 1996). Para que a extensibilidade seja alcançada, não basta integrar novos métodos, tendo cada um a sua própria interface e diferentes maneiras de iniciar as tarefas para recuperar os resultados. O controle deve oferecer um mecanismo que possibilite manejar, de maneira uniforme, as tarefas e recuperar os resultados.

Integrabilidade e Reusabilidade: Incorporar um novo componente ao sistema não significa somente uniformizar as chamadas externas e a recuperação de resultados, mas também mapear as funcionalidades de cada um desses componentes de modo a integrá-los de tal maneira que garanta o funcionamento do sistema de uma maneira mais coesa. Essa integração proporciona a reutilização de alguns componentes no desenvolvimento de novos, uma vez que algumas tarefas dentro do processo de KDD são comuns a vários componentes.

Escalabilidade: A execução das tarefas de processamento de dados e padrões, bem como a aplicação de algoritmos de aprendizado, podem ser atividades computacionalmente caras. Uma alternativa para solucionar este problema é o uso de sistemas distribuídos para adicionar escalabilidade ao sistema (Botía, Garijo, Velasco, and Skarmeta, 1998). O uso de sistemas distribuídos pode explorar tanto o paralelismo de dados como o paralelismo de processos (Freitas and Lavington, 1998).

2.4 Meta-Aprendizado

A melhor maneira de se ter uma boa idéia é ter várias idéias.

– LINUS PAULING

Meta-aprendizado (*meta-learning*) é definido como o aprendizado de meta-conhecimento sobre o conhecimento aprendido (Chan and Stolfo, 1993). Em alguns casos, esse conhecimento pode ser utilizado para combinar modelos e conjuntos de padrões para tarefas de classificação e predição (Chan, 1996).

Uma outra possível utilização de meta-aprendizado é a classificação (Brazdil and Soares, 2000) ou a seleção (Flach, 2001) de algoritmos de aprendizado para um dado

domínio, deixando o processo de KDD menos dependente do usuário, em relação à escolha do algoritmo de aprendizado.

Essa característica pode ser incorporada ao modelo do sistema de KDD mostrado na Figura 2.3 na página 20, transladando um pouco mais a curva do gráfico mostrada na Figura 2.4 na página 21 em direção ao ponto que indica o sistema ideal. O novo modelo, com uma nova classe de componentes para processar o meta-conhecimento, é mostrado na Figura 2.5.

Figura 2.5: Meta-aprendizado Incorporado ao Modelo de Ferramenta Genérica de KDD

O fluxo de informação (representado pelas setas na Figura 2.5) pelo sistema segue a seguinte ordem: o usuário entra com os comandos, os parâmetros de qualidade de Serviço — QoS (*Quality of Service*) — e com os objetivos de tarefa de KDD. Os componentes de processamento de dados processam os dados, agora não só fornecendo dados para os componentes de mineração, mas também meta-dados (tais como número de classes, atributos e exemplos, informações estatísticas sobre os dados, etc) para os componentes que processam o meta-conhecimento. Dentre os componentes que processam o meta-conhecimento, existe um componente que irá aconselhar/ranquear (utilizando alguma forma de inferência sobre os meta-dados, meta-conhecimentos e os objetivos do usuário) os possíveis algoritmos de aprendizado que serão utilizados. O algoritmo selecionado é aplicado aos dados, gerando os padrões. Os componentes que processam padrões, além do processamento para a avaliação e apresentação ao usuário, também fornecem meta-conhecimento para as atividades KDD.

Um exemplo de sistema que usa Meta-aprendizado é o JAM (Stolfo, Prodromidis, Tselepis, Lee, Fan, and Chan, 1997)— *Java Agentes Meta-learning* —, no qual os componentes são agentes de *software* escritos na linguagem de programação JAVA. Um outro exemplo, o projeto ESPRIT Metal (Flach, 2001), tem como objetivo produzir um assistente que proporcione um suporte ao usuário na escolha do algoritmo de aprendizado.

Entretanto, um sistema desse tipo tem como grande problema a formação da meta-base de conhecimento. Essa meta-base deve conter tanto o meta-conhecimento (conhecimento adquirido sobre o conhecimento), quanto do conhecimento sobre a utili-

zação dos diferentes algoritmos dentro do domínio da aplicação, reduzindo a aplicação desse sistema a áreas específicas (que já possuam um bom conhecimento sobre a utilização de algoritmos naquela área), além da necessidade de que o processo de KDD seja uma atividade contínua, para a incorporação do meta-conhecimento.

2.5 Considerações Finais

Qualquer coisa que você pode fazer, ou acredita que pode, comece-a. A audácia tem em si mesma genialidade, poder e magia.

– GOETHE

Um sistema de KDD é uma coleção de componentes que juntos possibilitam a realização de todo o processo de descoberta de conhecimento, processando dados, extraindo e processando padrões. Um sistema ideal deve realizar todo esse processo de uma maneira autônoma, podendo ser utilizado em uma grande quantidade de domínios. O modelo apresentado neste capítulo está longe do ideal, sendo que o compromisso entre versatilidade/autonomia é mostrado no gráfico da Figura 2.4.

A utilização de meta-aprendizado em um sistema de KDD é uma tentativa de transitar a curva no gráfico em direção ao sistema ideal mas, na maioria das vezes, resultados significativos são conseguidos apenas com a incorporação de conhecimento do domínio. Dessa maneira, para a maioria dos sistemas atuais, autonomia requer conhecimento do domínio enquanto versatilidade implica em independência do domínio. Essas duas características são inconciliáveis atualmente, e muito ainda deve ser feito para promover essa situação.

Outros pontos importantes em um sistema de KDD são extensibilidade, integrabilidade, reusabilidade e escalabilidade. A extensibilidade é necessária para que novos componentes sejam adicionados ao sistema de uma maneira simples. Integrabilidade e reusabilidade implicam que os componentes do sistema funcionem de uma maneira mais coesa, aumentando o reuso desses componentes. A escalabilidade pode ser conseguida pelo uso de sistemas e algoritmos distribuídos de maneira que seja possível a exploração de paralelismo de dados e processos.

Ao se projetar a arquitetura de um sistema de KDD todos esses fatores devem ser levados em consideração, em conjunto com os objetivos do sistema.

Arquitetura de *Software*

Os homens de maior sucesso no mundo usaram a imaginação. Eles pensaram na frente e criaram suas imagens mentais, e então foram trabalhar materializando essa imagem em todos os seus detalhes, preenchendo aqui, adicionando um pouco ali, alterando um pouquinho aqui e ali, mas sempre construindo, sempre construindo...

– ROBERT COLLIER

Arquitetura de *Software* é um dos aspectos mais importantes no projeto de um sistema e tem um efeito de longo alcance no seu sucesso. Arquitetura é, em primeiro lugar, uma abstração da implementação de um sistema. Existem muitos modelos de arquitetura que ajudam a entender sistema. Esses modelos ajudam a analisar certas qualidades do sistema: qualidades de execução, tais como desempenho, segurança ou confiança; e qualidades de projeto, tais como facilidade de modificação e portabilidade. Essas qualidades são importantes para as diferentes pessoas envolvidas com o sistema: o usuário final, administrador, desenvolvedor, mantenedor, entre outros. Diferentes cenários de utilização, incluindo cenários de modificações e extensões do sistema, podem ajudar a avaliar a arquitetura segundo essas qualidades.

Este capítulo tem como principal objetivo discutir o desenvolvimento de sistemas baseado na arquitetura de *software*, uma prática moderna de planejamento e desenvolvimento de sistemas que pondera não somente as características técnicas do sistema mas também os atributos de qualidade nele desejáveis.

Quanto à organização, este capítulo está dividido da seguinte forma: na Seção 3.1 é apresentada uma definição da arquitetura de *software* e são introduzidos outros conceitos, como estilo de arquitetura, modelo e arquitetura de referência. Na Seção 3.2 são discutidas as estruturas que compõem a arquitetura. Os atributos de qualidade de um sistema são apresentados na Seção 3.3. Na Seção 3.4 é apresentada uma visão mais aprofundada de estilos de arquitetura. A transformação da arquitetura em um sistema computacional é abordada na Seção 3.5. Finalmente, na Seção 3.6 é apresentado um resumo dos pontos chave deste capítulo.

3.1 Arquitetura de Software

Recomendação a um homem que estava se queixando sobre a chuva que escorria pelo telhado em cima de sua mesa de jantar: "Mova a mesa".

– ARQUITETO FRANK LLOYD WRIGHT

Bass, Clements, and Kazman (1999) definem arquitetura de *software* como

“estrutura ou estruturas do sistema, que incluem os componentes de software, as propriedades externamente visíveis desses componentes e as relações entre eles.”

A propriedade *externamente visível* refere-se às aceções que os outros componentes podem fazer sob um dado componente, tais como os serviços por ele providos, características de performance, gerenciamento de exceções, recursos compartilhados, entre outros. O ponto principal desta definição é que a arquitetura de *software* deve representar uma abstração de somente algumas informações do sistema (de outra maneira, estaríamos olhando todo o sistema), e ainda providir informações suficientes para servir como base para análise, tomada de decisões e redução de riscos.

Algumas outras implicações podem ser tiradas dessa definição. Primeiramente, **arquitetura define componentes**. A arquitetura engloba informações sobre como os componentes interagem uns com os outros. Isso significa que as arquiteturas omitem certas informações sobre os componentes que não são pertinentes à sua interação. Dessa maneira, arquitetura é, primeiramente, uma abstração do sistema que suprime detalhes dos componentes desde que esses detalhes não afetam seu uso, como são usados, informam para ou interagem com outros componentes. Em sistemas que usam ocultação de informação como um critério de projeto, os componentes reagem uns com os outros por interfaces que dividem detalhes sobre um componente em partes públicas e privadas. A arquitetura está na parte pública dessa divisão. Os detalhes privados dos componentes (detalhes que dizem respeito apenas a implementação interna) não são arquiteturais.

Em segundo lugar, a definição deixa claro que **sistemas podem conter mais de uma estrutura** e que, nesse caso, nenhuma estrutura sozinha pode ser considerada como a arquitetura do sistema. Por exemplo, grandes projetos usualmente são divididos em módulos que são usados na divisão de trabalho entre os times de programadores. Este é um tipo de estrutura que freqüentemente é usado para descrever um sistema. Suponha agora que o sistema deva ser construído para ser executado em processos paralelos. O conjunto de processos que existirão em tempo de execução, os diversos programas, divididos entre os vários módulos, que formam seqüencialmente em conjunto cada processo e as relações de sincronização entre os processos, formam outros tipos de estruturas e nenhuma delas pode, sozinha, representar toda a arquitetura. Este exemplo também ilustra que podem existir mais de um tipo de

componente (no exemplo, módulos e processos), mais de um tipo de interação (divisão, sincronização) e até mesmo mais de um contexto (desenvolvimento e execução). Estruturas de arquitetura são tratadas em maiores detalhes na Seção 3.2 na próxima página.

Em terceiro lugar, a definição implica que **todo sistema de software tem uma arquitetura** porque todo sistema pode ser definido como composto por componentes e relações entre eles. No caso mais trivial, o sistema é ele mesmo um componente. Enquanto uma arquitetura composta por um único componente não é interessante e provavelmente não muito útil, ela não contradiz a definição de arquitetura de *software*. Dizer que todo o sistema de *software* tem uma arquitetura, no entanto, não é necessariamente dizer que sua arquitetura é conhecida. Este detalhe revela a diferença entre a arquitetura de um sistema e a sua descrição ou especificação.

Por último, **o comportamento de cada componente é parte da arquitetura**, tanto que esse comportamento pode ser observado ou discernido do ponto de vista de um outro componente. Este comportamento é o que permite que componentes interajam uns com os outros.

A seguir serão apresentadas algumas outras definições que auxiliam no processo de criação da arquitetura do sistema.

Estilo de arquitetura é uma descrição dos tipos de componentes e padrões de controle e/ou transferência de dados em tempo de execução. Estilos podem ser vistos como um conjunto de restrições em uma arquitetura (restrições quanto aos tipos de componentes e seus padrões de interação) e essas restrições definem um conjunto de famílias de arquiteturas que as satisfazem. Estilos de arquitetura são descritos com maiores detalhes na Seção 3.4 na página 33.

Modelo de referência é a divisão das funcionalidades, em conjunto com o fluxo de dados entre os componentes. Um modelo de referência é uma decomposição padronizada de um problema conhecido em partes que, em conjunto, resolvem o problema. Para este trabalho, o modelo apresentado na Seção 2.3 na página 19 será usado como modelo de referência.

Arquitetura de referência é o modelo de referência mapeado nos componentes de *software* (que irão implementar as funcionalidades definidas no modelo de referência) e o fluxo de dados entre os componentes. Enquanto o modelo de referência divide as funcionalidades, a arquitetura de referência é o mapeamento dessas funcionalidades na decomposição do sistema. O mapeamento pode ser (mas não necessariamente precisa ser) um para um. Um componente também pode implementar parte de uma função ou um conjunto delas.

O modelo de referência, estilo de arquitetura e arquitetura de referência não são arquiteturas, mas sim passos muito úteis na definição da arquitetura do sistema. Cada uma é o resultado da antecipação da tomada de decisões dentro do projeto, e funciona como um pano de fundo para os próximos passos, preparando as próximas

etapas por meio da identificação de necessidades. A relação entre eles está representado na Figura 3.1. A cada fase nesta progressão novos requisitos são incorporados, até que a arquitetura final do sistema esteja definida.

Figura 3.1: A Relação entre Modelos de Referência, Estilos de Arquitetura, Arquitetura de Referência e a Arquitetura de *Software* Implementada.

3.2 Estruturas de Arquitetura

A estrutura sem a vida está morta. Mas a vida sem a estrutura não é percebida

— JOHN CAGE, 'LECTURE ON NOTHING' SILENCE, 1961

Assim como em um prédio, onde as plantas baixa, elétrica e hidráulica, o projeto da fachada externa e dos ambientes interiores, entre outras estruturas, todas juntas, descrevem a sua arquitetura, a arquitetura de um sistema complexo é composta por diversas estruturas.

Cada uma dessas estruturas tem um significado diferente, dependentes do motivo e do ponto de vista para os quais foram descritas. A interpretação de cada uma dessas estruturas só é possível sabendo-se claramente o que ela descreve. Algumas das estruturas mais comuns são listadas a seguir.

Estrutura modular As unidades são as tarefas designadas em cada módulo, contendo os produtos associados (tais como a especificação da interface, código, plano de teste, etc.). Essas unidades são ligadas pela relação “*é um submódulo de*”.

Estrutura conceitual (ou lógica) As estruturas são abstrações dos requisitos funcionais do sistema. Essas abstrações estão ligadas pela relação “*compartilham dados com*”. O modelo de referência, definido anteriormente na Seção 3.1, é um exemplo deste tipo de estrutura. Esta estrutura é útil para o entendimento das interações entre as entidades no escopo do problema e suas variações.

Estrutura de processo (ou coordenação) Esta é uma visão ortogonal às estruturas modular e conceitual e trata dos aspectos dinâmicos de um sistema em execução. As unidades são processos e *threads*. As relações representadas pelas ligações incluem “*sincronizadas com*”, “*não pode executar com*”, “*interrupções*”

ou quaisquer outras relações negociáveis entre sincronização e concorrência de processos.

Estrutura física Esta visão mostra o mapeamento do *software* no *hardware*. Ela é particularmente interessante em sistemas paralelos distribuídos. Os componentes são entidades de *hardware* (processadores), e as ligações são caminhos de comunicação. Relações entre processadores são do tipo “*comunica-se com*”.

Estrutura de uso As unidades são procedimentos ou módulos, que são ligadas por relações do tipo “*assumem a presença correta de*”. A estrutura de uso é utilizada pelos engenheiros do sistema que, a partir dela, podem facilmente dividir ou estender módulos, utilizando uma abordagem incremental na integração.

Estrutura de chamadas As unidades são, usualmente, (sub)procedimentos que estão relacionadas por relações de “*chamada*” ou “*invocação*”. A estrutura de chamadas é utilizada para traçar o fluxo de execução de um programa.

Estrutura de fluxo de dados As unidades são programas ou módulos, e a relação é do tipo “*pode enviar dados para*” e as ligações são rotuladas com o nome dos dados transmitidos. A estrutura de fluxo de dados é muito útil no rastreamento de requisitos.

Estrutura de fluxo de controle Unidades são programas, módulos ou estados do sistema, e a relação entre eles é do tipo “*deve ser ativada antes de*”. Esta estrutura é útil na verificação das ações funcionais, bem como de características temporais. Se o único mecanismo de transferência de controle é chamada de programa, esta estrutura é idêntica à estrutura de chamada.

Estrutura de classe As unidades são objetos e as relações são do tipo “*é herdada de*” ou “*é uma instância de*”. Esta estrutura suporta uma visão sobre coleções de comportamentos similares (isto é, as classes das quais as outras classes são herdadas).

Cada estrutura é uma abstração que corresponde a um critério diferente. Cada abstração lança detalhes sobre o *software* que são dependentes dos interesses refletidos na abstração. Cada abstração pode ser considerada como uma “planta de *software*”. Conseqüentemente, cada estrutura pode ser usada com a sua própria notação, podendo refletir suas próprias escolhas de estilos de arquiteturas, e pode definir independentemente o que é significativo para componentes, inter-relacionamentos, raciocínios, princípios e guias.

3.3 Definição dos Atributos de Qualidade

A beleza não é uma qualidade das coisas por si mesmas. Ela simplesmente existe na mente de quem as contempla.

– DAVID HUME (1757)

Existem duas grandes categorias para os atributos de qualidade que podem ser medidas em um sistema (e, em particular, podem ser medidas na arquitetura):

1. *Observáveis na execução*, que procuram avaliar se o sistema, durante a sua execução satisfaz os requisitos, produz os resultados corretos ou executa as funções em um tempo razoável.
2. *Não observáveis na execução*, que procuram avaliar o sistema quanto a facilidade de integração, teste e modificabilidade do sistema, custos e tempo de desenvolvimento.

Saber alguma coisa a respeito do sistema em uma dessas categorias não diz nada a respeito do desempenho na outra categoria. Por exemplo, sistemas que satisfazem todos os requisitos de execução podem ser altamente custosos para o desenvolvimento e podem ou não apresentar facilidades para modificação.

A qualidade deve ser considerada em todas as fases de projeto, implementação e desenvolvimento, mas diferentes qualidades manifestam-se diferentemente durante essas fases. Por exemplo, muitos aspectos de usabilidade não são arquiteturais: decidir se uma determinada informação deve aparecer em uma janela separada ou em uma caixa de texto na própria janela ou se uma informação deve ser digitada ou selecionada a partir de uma caixa de seleção, tem uma grande importância para o usuário final no que diz respeito a usabilidade, mas não são arquiteturais pois, na maioria das vezes, encontram-se encapsulados em um único componente.

Modificabilidade, por outro lado, é amplamente arquitetural. A modificabilidade é determinada por como a funcionalidade é dividida. Um sistema é facilmente modificável se as mudanças não envolvem um grande número de componentes. Certamente um código obscuro será difícil de ser modificado, mas o que determina a modificabilidade do sistema é, primeiramente, a sua divisão de responsabilidades entre os componentes.

Performance é um exemplo de qualidade que tem dependências arquiteturais e não arquiteturais. A performance depende parcialmente da quantidade de comunicação necessária entre os componentes (arquitetural), quais funcionalidades foram alocadas para cada componente (arquitetural), da escolha dos algoritmos que implementam aquelas funcionalidades (não arquitetural) e de como esses algoritmos serão codificados (não arquitetural).

Dessa maneira, a arquitetura é crucial para a obtenção de determinados atributos de qualidade interessantes em um sistema, e esses atributos devem ser designados e avaliados neste nível. Outros atributos, entretanto, não são sensíveis à arquitetura, e tentar alcançar essas qualidades, ou analisá-las em função da arquitetura, não trará bons resultados.

Também é importante ressaltar que não é possível alcançar esses atributos isoladamente. A obtenção de um determinado atributo de qualidade afetará, positiva ou

negativamente, a obtenção dos outros atributos de qualidade. Por exemplo, uma das principais técnicas para se alcançar a portabilidade é o isolamento das dependências do sistema operacional. Este isolamento introduz uma sobrecarga na execução do sistema, tipicamente na forma de processos ou procedimentos na fronteira entre o sistema operacional e o *software*, o que causa uma diminuição na performance do sistema.

Uma listagem com as definições de atributos de qualidade pode ser encontrada em [ISO and IEC \(1991\)](#)

3.4 Estilos de Arquitetura

As notas que toco não são melhores que de muitos pianistas. Mas as pausas entre as notas — ah, aí é que a arte reside!

– ARTUR SCHNABEL

Um estilo de arquitetura consiste em alguns atributos chaves e de um conjunto de regras para combinação desses atributos, de tal maneira que a integridade da arquitetura seja preservada. Um estilo de arquitetura de *software* é determinado por:

- um conjunto de tipos de componentes (por exemplo, repositório de dados, um processo, um procedimento) que cumprem alguma função em tempo de execução;
- um desenho topológico desses componentes indicando seus relacionamentos em tempo de execução;
- um conjunto de restrições semânticas (por exemplo, um repositório de dados que não esteja autorizado a mudar os valores nele armazenados);
- um conjunto de conectores (por exemplo, uma chamada de subrotina, uma chamada remota, fluxo de dados, *sockets*) que fazem a intermediação da comunicação, coordenação ou cooperação entre os componentes.

Um estilo define uma classe de arquiteturas, ou seja, é uma abstração de um conjunto de arquiteturas que o compõe. Estilos são normalmente (intencionalmente) ambíguos sobre o número de componentes envolvidos. Por exemplo, um fluxo de dados do tipo “*tubo e filtro*” pode conter dois filtros conectados por um tubo ou vinte filtros conectados por dezenove tubos. Estilos podem ser ambíguos sobre os mecanismos de como os componentes interagem, mas muitos estilos, tal como o “*programa principal e subrotinas*” por exemplo, tem esse mecanismo explicitamente definido. Estilos são sempre ambíguos sobre a função de um sistema: um dos componentes pode ser um banco de dados, por exemplo, mas os tipos de dados podem variar.

Em [Shaw and Garlan \(1996\)](#) estão catalogados um conjunto de estilos de arquitetura, que foram por eles observados examinando uma coleção de sistemas existentes.

Estilos não ocorrem somente de uma maneira regular nos projetos de sistemas, mas principalmente com pequenas variações em seus formatos. Na Figura 3.2, são apresentados alguns estilos de arquitetura, que estão categorizados em grupos, de uma maneira hierárquica. Por exemplo, um *sistema de eventos* é um subestilo de *componentes independentes*. O próprio *sistema de eventos* tem dois subestilos: *invocação implícita* e *invocação explícita*. Uma breve descrição de cada um dos estilos de arquitetura mostrados nessa figura é apresentada a seguir.

Figura 3.2: Catálogo de Estilos de Arquitetura

Arquiteturas centrada em dados: Esse grupo de arquiteturas tem o objetivo de alcançar a qualidade de integrabilidade dos dados. O termo arquitetura centrada nos dados refere-se a sistemas em que o acesso e modificação dos dados é feito por um armazém de dados que pode ser compartilhado por vários componentes. Esse armazém pode ser tanto passivo (*repositório*) ou ativo (*quadro negro*). A diferença entre ambos está no fato que o *quadro negro* envia notificações aos componentes registrados quando dados de seus interesses são modificados.

Arquiteturas de fluxo de dados: Esse grupo de arquiteturas tem o objetivo de alcançar a qualidade de reuso e modularidade. O estilo de fluxo de dados é caracterizado por enxergar o sistema como uma série de transformações em passos sucessivos, a partir dos dados de entrada. Os dados entram no sistema e fluem através dos componentes, um por vez, até serem enviados para um destino final (saída ou armazém de dados). Este estilo tem dois subtipos, *batch sequencial* e *tubo e filtro*. No estilo *batch sequencial*, os componentes são programas independentes e supõe-se que cada passo é completamente executado antes do próximo passo começar. O estilo *tubo e filtro* enfatiza a transformação incremental dos dados por componentes sucessivos. Este estilo é típico na família de sistemas Operacionais UNIX.

Arquitetura de máquina virtual: Esse grupo de arquiteturas tem o objetivo de alcançar a qualidade da portabilidade. Máquinas virtuais são estilos de *softwares* que simulam alguma funcionalidade que não é nativa ao *hardware* ou *software* em que estão implementados. Alguns exemplos de arquiteturas de máquinas virtuais são *interpretadores* e *sistemas baseados em regras*.

Arquitetura de chamada e retorno: Esse grupo de arquiteturas tem o objetivo de alcançar as qualidades de modificabilidade e escalabilidade, e tem sido a arquitetura dominante em sistemas de *software* nos últimos 30 anos.

A *arquitetura programa principal e subrotinas* é o paradigma clássico de programação. O objetivo é decompor o programa em peças menores para facilitar a modificabilidade. Um programa é decomposto hierarquicamente e, tipicamente, existe uma única linha de controle. Cada componente na hierarquia toma o controle de seus antecessores e repassa o controle aos sucessores. *Invocação remota de procedimentos* são sistemas do tipo *programa principal e subrotinas* que são decompostas em partes que estão em computadores diferentes conectados por uma rede, com o objetivo de aumento de performance.

O paradigma *orientação a objeto* enfatiza o empacotamento, a maneira de manipulação e o acesso aos dados. O objetivo é alcançar a qualidade da modificabilidade. Este empacotamento é um encapsulamento que esconde os dados dentro de um ambiente (objeto). Este encapsulamento promove o reúso e modificabilidade, principalmente porque promove a separação entre funcionalidade e implementação: um usuário não precisa (e não deve) saber nada sobre a implementação da função desejada.

Sistemas de camadas são aqueles em que os componentes são divididos em camadas que controlam a interação entre os componentes. Na versão pura desse estilo, cada nível pode se comunicar apenas com os seus vizinhos imediatos. A camada mais baixa¹ provê funcionalidades centrais, como acesso ao *hardware* ou *kernel* de sistemas operacionais. Cada camada sucessiva é construída sobre a sua antecessora, escondendo as camadas anteriores e provendo alguns serviços para serem utilizadas pela camada superior.

Arquiteturas de componentes independentes: Esse grupo de arquiteturas consiste em um número de processos ou objetos independentes que se comunicam por trocas de mensagem e tem como objetivos alcançar a modificabilidade, decompondo a computação em várias partes (componentes). Esses componentes enviam dados uns aos outros mas, tipicamente, não controlam outro componente diretamente.

Sistemas de eventos são um subestilo em que o controle é parte do modelo. Os componentes individualmente anunciam (publicam) os dados que desejam compartilhar com o ambiente. Os outros componentes podem registrar interesse (subscrever) nessa classe de dados e quando o dado é gerado, os componentes registrados são invocados e recebem os dados.

¹Arquiteturas de camadas são, muitas vezes, desenhadas como círculos concêntricos. Neste caso, mais baixa significa mais interna. Entretanto, não existe nenhuma diferença conceitual.

Além de *sistemas de eventos*, o outro subestilo de *componentes independentes* é o estilo dos *processos comunicantes*, que são os sistemas multi-processados clássicos. Dentre eles, o modelo *cliente-servidor* é o subtipo mais conhecido. O objetivo é alcançar a qualidade da escalabilidade.

3.4.1 Estilos heterogêneos

Sistemas raramente são construídos a partir de um único estilo. Esses sistemas são chamados heterogêneos. Existe três tipos de heterogeneidade:

1. *Heterogeneidade localizacional*, na qual o desenho das estruturas do sistema em tempo de execução revelarão padrões de diferentes estilos em diferentes areas, como, por exemplo, alguns ramos de um sistema *programa principal e subrotinas* pode ter um *repositório de dados* compartilhado.
2. *Heterogeneidade hierárquica*, na qual o componente de um estilo, quando decomposto, está estruturado de acordo com as regras de um estilo diferente.
3. *Heterogeneidade simultânea*, na qual mais de um estilo pode ser apropriado para a descrição de um sistema.

A definição da arquitetura do sistema não é uma tarefa trivial, uma vez que a mesma deve ponderar os requisitos de qualidade, descritos na Seção 3.3 na página 31, na busca da melhor solução. A utilização de estilos pode facilitar essa tarefa, uma vez que eles podem servir como uma primeira descrição do sistema. A utilização de diferentes estilos pode ser aplicada e comparada, e as melhores alternativas refinadas e melhoradas, resultando ao final, na arquitetura do sistema. Uma classificação dos diferentes estilos segundo seus atributos pode ser encontrada em (Shaw and Clements, 1997). Este artigo contém muitos outros subestilos dos que os citados neste trabalho, bem como uma lista de autores que escreveram sobre eles.

3.4.2 Operações Básicas

Estilos de arquitetura são muito úteis para a abstração de um problema em uma arquitetura. Entretanto, o mapeamento e abstração do estilo para a arquitetura não é uma tarefa trivial. Para auxiliar esse mapeamento, um conjunto de operações básicas pode ser definido (Shaw, 1990). Essas operações são diferentes de estilos de arquitetura e *patterns* no sentido de que são mais primitivas. Estilos e *patterns* usam (ou são derivadas de) essas operações básicas. Algumas dessas operações básicas são listadas a seguir.

Separação coloca uma funcionalidade distinta em um componente distinto que tem uma interface bem definida com o resto do mundo. É a ferramenta mais comum e mais primitiva de um arquiteto de *software*. A separação isola uma parte da funcionalidade do sistema. A motivação para determinar quais partes das funcionalidades do sistema serão isoladas está fortemente ligada aos objetivos de

qualidade que se deseja alcançar. Pode-se, por exemplo, separar as funcionalidades por performance ou facilidade de criação. Separação permite distribuição e, a partir dela, paralelismo.

Separação também pode ser usada para assegurar que mudanças no ambiente externo não afetarão um componente, e mudança no componente não afetarão o ambiente, a não ser que a interface seja modificada.

Abstração é a operação de criação de componentes cuja função é esconder ou encapsular alguma funcionalidade externa do sistema. Esses componentes são de difícil criação, mas uma vez criados, podem ser adotados e reutilizados por outros componentes de *software*, simplificando a criação e manutenção desses componentes.

Separação e abstração estão relacionados mas são conceitos diferentes. Existem muitos exemplos de separação que tiveram como principal objetivo a criação de um conjunto de serviços abstratos, tais como balanceamento de carga, paralelização de operações, divisão de trabalhos entre os desenvolvedores, entre outros.

Compressão é a operação de remoção de camadas ou interfaces que separam funções do sistema, e por isso, pode ser vista como o oposto da separação. Quando um *software* é comprimido, algumas funções distintas são postas juntas. A compressão serve para três propósitos principais:

1. Aumentar a performance do sistema, eliminando a sobrecarga de atravessar camadas entre funções.
2. Contornar camadas quando elas não provêem serviços necessários.
3. Aumentar a velocidade de desenvolvimento, eliminando o requisito de que diferentes partes das funcionalidades do sistema devem ser alocadas em diferentes componentes de *software*.

Compartilhamento de recursos é a operação que encapsula dados ou serviços e os compartilha entre os múltiplos componentes independentes. Tipicamente existe um gerenciador de recursos que provê acesso ao recurso de uma maneira única. Recursos compartilhados, apesar de serem freqüentemente caros de serem construídos, aumentam a integrabilidade, portabilidade e modificabilidade de sistemas, principalmente porque reduzem a duplicação entre os componentes. Em muitos casos, recursos compartilhados também são abstrações.

No desenvolvimento do *software*, estamos interessados nos efeitos e interações dessas unidades de operação com respeito aos atributos de qualidade.

3.5 Desenvolvimento Baseado na Arquitetura

Nenhum homem é uma ilha, terminado em si mesmo; todo homem é uma peça do continente, uma parte do todo.

– JOHN DONNE

A definição de uma arquitetura também tem um grande reflexo no desenvolvimento do sistema. Um dos pontos mais importantes para a arquitetura é que se ela é difícil de implementar, ela é sem valor. Os principais benefícios que arquitetura traz para o desenvolvimento são a possibilidade de criação de um esqueleto do sistema e a exploração de *patterns*, que são brevemente discutidas nas próximas seções.

3.5.1 Criação do Esqueleto do Sistema

Uma das grandes vantagens do desenvolvimento de *software* baseado na arquitetura é a possibilidade da criação de um esqueleto para o sistema. A idéia é implementar as características principais da arquitetura nesse esqueleto e cada novo componente descrito na arquitetura é implementado para estendê-lo.

Ao contrário das práticas convencionais de Engenharia de *Software*, que primeiro recomendam a criação dos componentes mais simples e a simulação dos outros componentes de nível mais alto na hierarquia por meio de *stubs*, dificultando um entendimento global do sistema, utilizar a arquitetura como guia torna o desenvolvimento do sistema muito mais claro.

Em primeiro lugar, implementa-se o *software* que cuida da execução e interação dos componentes da arquitetura. Nesse ponto, a execução/interação do sistema já está definida, e os componentes de nível mais baixo na hierarquia são simulados por meio de *stubs*. A implementação desses componentes pode ser feita de uma maneira mais fiel para com a arquitetura, facilitando o desenvolvimento global do sistema.

Esta mesma abordagem pode ser empregada para cada um dos novos componentes que serão desenvolvidos, o que torna o desenvolvimento do sistema um processo iterativo e incremental de refinamento do sistema, em direção ao produto final. Algumas vantagens do desenvolvimento de *software* utilizando a arquitetura como guia são listadas a seguir.

- O desenvolvimento do sistema, pelo refinamento sucessivo em componentes, fica mais compreensível e fiel a arquitetura do sistema.
- Permite que os problemas mais complexos de implementar ou aqueles que têm um alto grau de incerteza sejam tratados antecipadamente.
- A integração do sistema é facilitada, pois esses aspectos são tratados no início do projeto.
- As tarefas de teste e revisão também é facilitada, pois um protótipo do sistema (o esqueleto) já existe.
- Dependências complexas são identificadas nas primeiras fases do processo, uma vez que um maior planejamento e desenvolvimento da integração ocorrem nessas fases.

3.5.2 Explorando *Patterns* na Arquitetura de *Software*

A possibilidade de análise e desenvolvimento de um sistema como um conjunto de “blocos regulares” pode trazer grandes benefícios, tal como uma maior compreensibilidade para os sistemas complexos, o que auxilia tanto o desenvolvimento como a manutenção (Shaw, 1996). A utilização de blocos regulares na construção de *software* implica em *patterns* comuns de uso.

Um *pattern* é uma pequena coleção de unidades atômicas e a descrição de suas relações. *Pattern* são ubíquos no desenvolvimento de sistemas e são freqüentemente usados de maneira inconsciente. Em *design patterns* orientados a objetos, por exemplo, um *patterns* é uma coleção de classes de objetos e a hierarquia e relações de uso entre eles. A interação por meio de uma estrutura de dados é um exemplos de *pattern* de código. A exploração de *patterns* é uma ferramenta poderosa como base para o desenvolvimento de bibliotecas de modelos (*templates*) que suportam o desenvolvimento de componentes, como será visto no próximo capítulo.

3.6 Considerações Finais

Aprender sem raciocinar é um trabalho perdido

– CONFÚCIO

Arquitetura de *software* compreende um conjunto de componentes e relações entre eles. A arquitetura de *software* é, geralmente, composta por várias estruturas que descrevem esses componentes e relações. Na definição de uma arquitetura para um sistema, além dos requisitos funcionais, também devem ser levados em consideração requisitos de qualidade do sistema.

Estilo de arquitetura, modelo e arquitetura de referência são passos intermediários na definição da arquitetura do sistema. A arquitetura de *software* funciona como um veículo de comunicação entre os interessados no sistema, bem como um mecanismo de antecipação de decisões e minimização de riscos.

O desenvolvimento baseado na arquitetura permite uma melhor visualização no desenvolvimento do sistema, pela criação de um esqueleto do mesmo, além de uma maior conformidade dos componentes com a arquitetura. Um outro ponto importante é a exploração de *patterns* no desenvolvimento.

Componentes, *Frameworks* e *Patterns*

Um ponto de vista vale mais que vinte pontos de QI

– ADELE GOLDBERG

*U*m dos mais importantes temas na Engenharia de *Software* têm sido o reúso de *software*, isto é, reutilizar de alguma forma e o mais possível o *software* existente para construir novas aplicações.

Na verdade, virtualmente toda a nova aplicação desenvolvida nos dias de hoje reutiliza *software* em algum grau. O *software* reutilizado pode ser desde uma pequena subrotina de uma biblioteca pessoal de um programador a uma aplicação grande e inteira.

Uma das primeiras práticas de como reutilizar *software* foi a subrotina. Ao invés da mesma tarefa, tal como ordenação, ser repetidamente codificada, o seu código inicial pode ser abstraído como uma subrotina e chamada de diferentes partes de um mesmo programa.

Algoritmos implementados como subrotinas também são úteis quando divididos entre programas. É relativamente simples copiar o código fonte, mas compilar esse código toda vez é caro computacionalmente. Desse fato surgiu a idéia da ligação, que permitia reutilizar subrotinas compiladas. Por questões de armazenagem, um conjunto de subrotinas são juntadas em uma biblioteca. Bibliotecas de subrotinas ainda são a maneira mais utilizada para reutilizar *software*.

A idéia da programação orientada a objetos, originada a partir do SmallTalk-80, ganhou uma grade popularidade nos anos de 1990, depois da introdução do C++, uma extensão do C. Neste tipo de programação, um objeto encapsula os detalhes de implementação separando-a claramente da sua interface. Além disso, uma classe pode ser reutilizada herdando a maioria de suas características e modificando outras

sobrecarregando-as. Um conjunto de classes orientadas a objeto é chamado biblioteca de classes. Essas classes foram envolvidas em *frameworks*, que possibilitam uma inversão de controle quanto ao processamento.

Um outro aspecto importante dessa evolução é a visão de sistemas como um conjunto de blocos de montar (componentes). Grosseiramente falando, o termo componente de *software* refere-se a peças reutilizáveis de *software* que podem ser manipuladas e combinadas utilizando uma ferramenta de construção.

Além disso, o conceito de *software pattern* tem sido explorado como uma nova prática de criação de *softwares*. *Software Pattern* enfatiza a identificação e comunicação entre pessoas por meio do compartilhamento dos melhores exemplos e práticas na construção de *software*.

Este capítulo tem como objetivo apresentar uma visão sobre esses três conceitos: componentes, *frameworks* e *patterns*. Quanto à organização, ele está dividido da seguinte forma: na Seção 4.1 são apresentados os conceitos de componentes e interfaces. Na Seção 4.2 são apresentados os conceitos de modelo e *framework* de componentes e na Seção 4.3 o conceito de *patterns*. Na Seção 4.4, é apresentada uma visão unificada desses conceitos. Finalmente na Seção 4.5 é apresentado um resumo dos pontos principais deste capítulo.

4.1 Componentes

O todo engloba suas partes, e as partes compõem o todo. Compor, do Latim **cum** “junto” e **ponere** “por”. As partes que compõem são, etimologicamente falando, componentes. Porque essa pedagogia? Por que, por definição, todos os sistemas de *software* incluem componentes.

– BACHMAN ET AL. (2000)

Componentes são resultado da decomposição de problemas, uma técnica padrão para a resolução de problemas. Em termos de *software*, diferentes concepções sobre como sistemas devem ser organizados resultam em diferentes tipos de componentes. Então, dois sistemas podem englobar componentes, mas os componentes desses sistemas podem não ter mais nada em comum do que o nome “componente”.

Não existe na literatura uma única definição de componente. Este fato não causa surpresa, uma vez que diferentes interpretações para problemas a serem resolvidos e diferentes abordagens na resolução desses problemas invariavelmente levam a diferentes interpretações das partes que constituem a solução (componentes). Na literatura sobre reúso, por exemplo, componente é visto como qualquer coisa que possa ser reutilizada; usuários de *softwares* “*commercial off-the-shelf*” — COTS — equiparam componentes a produtos COTS; para os metodologistas de *software*, componentes são unidades de projeto e de gerenciamento de configuração; e para arquitetos de *software*, abstrações de *design*. Outras definições análogas também são amplamente encontradas na literatura.

Para este trabalho usaremos a definição de componente do *Software Engineering Institute* — SEI — da Universidade de Carnegie Mellon. Essa definição é consistente com a maioria das outras definições apontadas na literatura, e em particular como a apontada em [Szyperski \(1998\)](#). Mas a definição da SEI reflete fortemente o argumento apresentado em [Wang and MacLean \(1998\)](#) de que componentes de *software* combinam duas perspectivas distintas: componentes como implementação e componentes como uma abstração da arquitetura.

O conceito de componentes como implementação é familiar ao encontrado no mercado e, como mencionado anteriormente, é mais freqüentemente utilizado para se referir a produtos COTS. Mas esses produtos podem implementar funcionalidade (o que um componente faz) e coordenação (como um componente interage com o mundo exterior) de uma maneira que é única para o produto. Em contraste, na definição do SEI, componentes são requeridos para implementar uma ou mais interfaces que descrevem como os componentes devem interagir com as outras restrições de arquitetura. Como será descrito na Seção 4.2.1 na página 45, essas interfaces compõem um modelo de componentes. Esse fato assegura que um componente respeita as restrições arquiteturais. Para o SEI, um componente é ([Bachman et al., 2000](#))

“uma implementação opaca de uma funcionalidade, sujeita a composição por terceiros e em conformidade com um modelo de componente.”

Existem duas motivações para o critério de que um componente é uma implementação opaca. Primeiro, essa abordagem prevê a utilização de componentes de *software* como produtos. Em segundo lugar, como diz um preceito bem estabelecido em ciência da computação, componentes de *software* não devem deixar visíveis detalhes de implementação que possam ser modificados. A opacidade têm para componentes o mesmo valor que abstração e ocultação de informação têm para a programação.

A motivação para composição por terceiros vêm do fato que o uso desses componentes não deve depender de ferramentas ou conhecimento sobre o componente que somente o desenvolvedor possui. Este critério implica que um sistema pode englobar componentes provenientes de múltiplas e independentes fontes.

O último critério, o de que os componentes devem estar de acordo com um modelo de componente é o que diferencia componentes dos produtos COTS convencionais. Modelos de componentes descrevem como componentes interagem uns com os outros, e por conseqüência expressam restrições globais ou arquiteturais. Ao contrário de sistemas montados sobre COTS, que são resultado de uma confusa integração de esquemas específicos de interação, os modelos de componentes são baseados em esquemas uniformes e padronizados de coordenação, transformado a implementação do *software* em uma implementação arquitetural.

A nossa habilidade de integrar componentes em uma montagem, raciocinar sobre essa montagem ou construir um repositório de componentes depende fundamentalmente da noção de interface de componente. Uma abstração de interface provê um

mecanismo de controle de dependências que surgem entre os módulos¹ em um programa ou um sistema. Uma *interface de programação de aplicação* — API (*Application Programming Interface*) — é uma especificação dessas propriedades de um módulo, em uma linguagem de programação, a qual clientes desse módulo dependem. A maioria das linguagens de programação modernas suportam alguma maneira de abstração de interface. Algumas outras linguagens independentes de especificação de interface estão sendo desenvolvidas, como a *Interface Definition Language* — IDL, do *Object Management Group's* ([Object Mangement Group, 2000](#)).

Decisões de projeto e implementação que provavelmente não sofrerão mudanças são especificadas na API, enquanto que as decisões em que existe a probabilidade de mudança são internamente “escondidas”. Na prática, o que se deseja é que a ocultação de informação torne os módulos substituíveis (como por exemplo, com uma nova versão de um componente) e conseqüentemente tornar os sistemas mais facilmente modificáveis. Entretanto, essa abordagem depende da possibilidade de ocultação das informações por meio de uma API. Uma vez que essas API's estão restritas as linguagens de programação (ou especificação de interface) e, conseqüentemente, as limitações dessas linguagens, nem todas as propriedades podem ser escondidas nas API's . Todas as outras propriedades podem “vazar” por meio de uma abstração da interface.

Além disso, as API's convencionais (aquelas cuja especificação da interface é escrita utilizando-se linguagens de programação) podem representar apenas propriedades funcionais. As propriedades funcionais incluem os serviços que um módulo prove e a sua assinatura (os tipos e ordem dos argumentos dos serviços e a maneira na qual cada resultado é retornado do serviço). As API's convencionais geralmente não provem recursos para a representação das propriedades de qualidade. Essas propriedades incluem performance, precisão, avaliabilidade, latência, segurança, entre outros. Como essas API's não podem descrever essas propriedades, não podem oculta-las. Como resultado, a probabilidade de substituir módulos que possam depender dessas propriedades diminui.

O ponto crucial na definição das interfaces é distinguir entre interface abstrata (aquela em que é descrita independentemente de qualquer implementação) e interface específica (aquela que é associada com uma implementação). Essa distinção é essencial para certificação, composição e análise de sistema. As propriedades da interface abstrata podem ser capturadas utilizando o modelo de componente, como descrito na Seção 4.2.1. As interfaces específicas podem ser estendidas com a utilização de contratos entre os componentes.

Interfaces são uma maneira de descrever dependências de uma maneira unidirecional. Contratos, por outro lado, expressam uma co-dependência entre os componentes. Um contrato deve contemplar, além dos padrões de interação com o componente, os padrões de interação e funções entre diferentes componentes.

¹O termo *módulo* é usado para se referir a implementações de *software* que tenham interface. O uso do termo componente está restrito à definição apresentada na Seção 4.1.

4.2 Frameworks e Modelos de Componente

Freqüentemente nós superestimamos o que uma nova tecnologia pode fazer em alguns meses e subestimamos o que ela pode fazer em alguns anos

– FEIGENBAUM, MCCORMACK & NII

Existe uma certa confusão quanto a terminologia na literatura sobre modelo e *framework* de componentes. Weck (1997), por exemplo, não faz distinção entre esses dois conceitos, definindo um *framework* de componente como padrões e convenções que podem ou não incluir suporte a serviços. D'Souza and Wills (1999) usa o termo *kit* de componente com o mesmo significado do *framework* de Weck, e usa o termo modelo de componente para se referir a interfaces, suposições e outros recursos associados a componentes individuais (isso é, cada componente tem o seu próprio modelo de componente). Essas diferenças se referem a como a categoria deve ser chamada e sobre as próprias categorias. Entretanto, existe um consenso que sistemas que se baseiam em componentes devem contar com convenções e padrões bem definidos, o que segundo a nomenclatura adotada pela SEI é chamada de modelo de componente, e uma infraestrutura de suporte, chamada *framework* de componente.

O modelo de componente especifica os padrões e convenções impostas aos desenvolvedores dos componentes. A conformidade com o modelo de componentes é o que distingue componentes das outras formas de encapsulamento de *software*. O *framework* de componentes é a implementação de serviços que suportam ou impõem o modelo de componente. Ambos serão examinados com maiores detalhes nas próximas seções.

4.2.1 Modelo de Componentes

Não existe um consenso entre a comunidade sobre o que deve ou não ser incluído em um modelo de componente. Bachman et al. (2000) apresenta alguns pontos sobre os propósitos de um modelo de componentes:

Composição uniforme: dois componentes podem interagir se e somente se eles compartilham suposições sobre o que cada um provê e o que cada um requer do outro. Mas também algumas outras suposições que devem ser padronizadas entre todos os componentes. Esses padrões devem incluir como os componentes são localizados, como o fluxo de controle é sincronizado, qual o protocolo de comunicação utilizado, como os dados são codificados e assim por diante.

Atributos de qualidade apropriados: A padronização dos tipos de componentes utilizados em um sistema e seus padrões de interação (o que é conhecido como estilo de arquitetura, descrito na Seção 3.4 na página 33) é uma maneira de assegurar que o sistema composto pelos componentes irá alcançar certas metas de qualidade (Bass, Clements, and Kazman, 1999).

Extensão dos componentes e aplicações: Os componentes necessitam de um mecanismo de extensão uniforme e independente da natureza dos mesmos, para facilitar a composição em tempo de construção e execução. Dessa maneira, o modelo de componentes deve impor padrões e convenções sobre os tipos de componentes, esquemas de interação e compartilhamento e alocação de recursos.

A partir desses pontos, o modelo de componentes impõe padrões e convenções dos seguintes tipos:

Tipos de componente: Um tipo de componente pode ser definido em termos das interfaces que ele implementa. Se um componente implementa três diferentes tipos de interface *X*, *Y* e *Z*, então esse componente é do tipo *X*, *Y* e *Z*. Um modelo de componente requer que um componente implemente uma ou mais interfaces e, dessa maneira, um modelo de componentes pode definir um ou mais tipos de componente. Diferentes tipos de componentes têm diferentes funções em um sistema, e participam de diferentes tipos de esquemas de interação.

Esquemas de interação: Modelos de componentes devem especificar como os componentes devem ser localizados, quais protocolos de comunicação serão utilizados e quais serviços de qualidade, tais como segurança e transações são alcançados. O modelo de componente também deve descrever como componentes interagem uns com os outros, e como eles interagem com o *framework* de componente. As classes de interação incluem restrições sobre quais tipos de componentes podem ser clientes de outros tipos, o número de clientes simultâneos permitidos e outras restrições quanto a topologia. Um segundo ponto relacionado as classes de interação inclui atividades relativas ao gerenciamento de recursos, tais como o ciclo de vida do componente (ativação, suspensão e desativação), gerenciamento da linha de execução, persistência e assim por diante. Esquemas de interação podem ser comuns para todos os tipos de componentes ou particular para um dado tipo de componente.

Alocação de Recursos: O processo de composição de componentes é uma maneira de alocar a um componente um ou mais recursos. Um recurso pode ser ou um serviço provido pelo *framework* ou um serviço provido por qualquer outro componente acoplado ao *framework*. Um modelo de componente descreve quais recursos estão disponíveis para os componentes e como e quais componentes são alocados para esses recursos. Da mesma maneira, o *framework* vê um componente como recursos que devem ser gerenciados.

4.2.2 Frameworks

Um *framework* é um *design* em larga escala que descreve como um programa é decomposto em um conjunto de objetos que se interagem (Johnson, 1997). É usualmente representado como um conjunto de classes abstratas e a maneira com que

instanciações dessas classes interagem (Johnson and Foote, 1988; Wirfs-Brock and Johnson, 1990). Como o *framework* inclui a maneira como os componentes interagem, ele é o modelo colaborativo ou o padrão de interação dos objetos. A parte mais importante de um *framework* é a maneira que o sistema é dividido em componentes (Deutsch, 1987, 1989). *Frameworks* também reutilizam implementação, mas isso é menos importante do que a reutilização das interfaces internas do sistema e a maneira que as funções são divididas em seus componentes. Esse *design* de alto nível é a parte intelectual mais importante do *software*, e *frameworks* são uma maneira de reutilizá-la.

Os principais objetivos da utilização de um *framework* derivam da modularidade, reusabilidade, extensibilidade e inversão de controle que ele provê aos desenvolvedores, como descrito a seguir (Fayad and Schmidt, 1997):

Modularidade: *frameworks* aumentam a modularidade encapsulando detalhes voláteis da implementação atrás de interfaces estáveis. A modularidade de um *framework* ajuda a promover a qualidade do *software* localizando o impacto das mudanças de projeto e implementação. Esta localização reduz o esforço necessário para entender e manter um sistema.

Reusabilidade: As interfaces estáveis providas pelo *framework* aumentam a reusabilidade definindo componentes genéricos que podem ser reaplicados para gerar novos componentes. A reutilização dos componentes do *framework* podem ainda aumentar substancialmente a produtividade dos desenvolvedores, bem como a qualidade, performance, confiança e interoperabilidade do *software*.

Extensibilidade: Um *framework* aumenta a extensibilidade provendo métodos de “gancho” explícitos (Pree, 1994) que permitem às aplicações estender suas interfaces.

Inversão de Controle: A arquitetura de tempo de execução de um *framework* é caracterizada por uma “inversão de controle”. Essa arquitetura possibilita um processamento padronizado do gerenciamento de eventos que são invocados pelo mecanismo de envio do *framework*. Quando eventos ocorrem, o emissário do *framework* reage invocando métodos de gancho previamente registrados no objeto, que fazem um processamento específico da aplicação para os eventos. A inversão de controle possibilita ao *framework* (assim como toda a aplicação) determinar quais os conjuntos de métodos específicos da aplicação serão invocados em resposta aos eventos externos.

Frameworks podem ser divididos segundo o seu escopo de utilização em:

- *frameworks* de aplicação,
- *frameworks* de *middleware* e
- *frameworks* de infra-estrutura.

O *framework* de aplicação é utilizado para domínios específico de aplicação, provendo um conjunto de classes para a construção de novas aplicações para esse domínio. *Frameworks* de *middleware*, tais como CORBA ([Object Management Group, 2001](#)) e COM ([Microsoft Corporation, 1998](#)) são normalmente utilizados para integrar aplicações de componentes distribuídos.

Uma boa maneira de pensar em um *framework* de infraestrutura de componentes é como um mini sistema operacional. Nesta analogia, componentes estão para *frameworks* como processos estão para o sistema operacional. O *framework* gerencia os recursos compartilhados pelos componentes e prove mecanismos de suporte que habilitam a comunicação (interação). Como no sistema operacional, o *framework* de componente é ativo e atua diretamente sobre os componentes para gerenciar o seu ciclo de vida e outros recursos, como por exemplo, iniciar, suspender, reiniciar ou terminar a execução do componente.

Entretanto, ao contrário de um sistema operacional de propósito geral que suporta um grande número de mecanismos de interação, *frameworks* de componentes são especializados para suportar somente um pequeno número de tipos de interação. Em troca a essa perda de flexibilidade, o *framework* de componente tem maiores facilidades para a composição de componentes.

4.3 Software Patterns

Todos nós conhecemos o valor da experiência de projeto. Quantas vezes você teve um *dejá-vu* — aquele sentimento que você já resolveu o problema mas não sabe exatamente onde ou como? Se você pudesse relembrar os detalhes do problema anterior e como você o resolveu, então você poderia reutilizar a experiência ao invés de redescobri-la.

— A GANGUE DOS QUATRO ([GAMMA ET AL., 1994](#))

Software Pattern está fortemente baseado nos trabalhos de Christopher Alexander, que propôs uma nova atitude para arquitetura, construção e planejamento de cidades, comunidades, prédios e casas ([Alexander, Ishikawa, Silverstein, Jacobson, Fiksdahl-King, and Angel, 1977](#)). Esta atitude é uma visão humanística do desenvolvimento, com a participação efetiva de todos os habitantes no processo de desenvolvimento. Desenvolvimento este no sentido mais amplo do termo, incluindo atividades como planejamento, construção e manutenção.

Segundo [Alexander \(1979\)](#), a natureza de cada lugar é determinada pelos padrões de eventos humanos que acontecem regularmente nesse lugar. Esses padrões de eventos estão interligados com padrões geométricos do espaço. Esses padrões de eventos no espaço são os únicos elementos que se repetem em diferentes lugares. Existe um pequeno número desses padrões que formam os “átomos” do conhecimento para a criação desses lugares, que são formados por combinações desses padrões. Cada um desses padrões podem ser descritos em uma forma literária, chamada “*pattern*”. Esses *patterns* formam uma linguagem de *patterns* (*pattern language*).

Os 253 *patterns* descritos em Alexander, Ishikawa, Silverstein, Jacobson, Fiksdahl-King, and Angel (1977) são um exemplo de uma linguagem de *pattern* para arquitetura. Esses *patterns* não estão descritos formalmente, mas sim como textos literários, de maneira a melhorar a comunicação de seus significados. A sua utilização na construção de lugares não pode ser automatizada. A utilização desses *patterns* é então uma atividade criativa, que auxilia a geração da qualidade, a partir das ações indiretas das pessoas envolvidas no processo.

Alexander (1979) enfatiza a importância da participação efetiva de todos os envolvidos na construção e utilização de, por exemplo, um prédio, no seu desenvolvimento e construção. Para guiar essa participação, uma linguagem de *patterns* é extraída das pessoas experientes na área. Essa linguagem de *patterns* permite o compartilhamento do conhecimento para a geração das soluções. Cada *pattern* descreve em uma forma literária um problema que ocorre várias vezes no nosso ambiente e então descreve o núcleo da solução desse problema, de tal maneira que podemos utilizar essa solução inúmeras vezes, gerando resultados diferentes a cada vez. Tal linguagem de *patterns* permite a qualidade, que não pode ser fabricada, mas apenas gerada, indiretamente, pelas ações das pessoas.

A utilização de *patterns* na construção de *software* ganhou um grande impulso com a publicação do livro *Design Patterns: Elements of Reusable Object Oriented Software* (Gamma, Helm, Johnson, and Vlissides, 1994), que descreve um catálogo com 23 *design patterns*. Tão importante quando o livro em si, foi a forma que o livro foi escrito, com revisões e discussões do seu conteúdo pela comunidade antes de sua publicação. Essa forma de trabalho é uma das características mais importante dessa abordagem.

A abordagem *Software Pattern* pode ser definida por quatro perspectivas complementares (Salviano, 1997):

Uma Atitude: Visão humanística de *software*, tanto no resultado quanto no desenvolvimento.

Uma Ênfase: Identificar (descobrir, e não inventando) e comunicar exemplos de boas soluções e das melhores práticas.

Um Objetivo: Criação de uma nova literatura (linguagens de *patterns*).

Uma Forma: Uma forma literária (*pattern*).

Apesar de todo o avanço em tecnologia para *software*, ou mesmo por causa destes avanços, um problema fundamental ainda existe para a qualidade de *software*: a comunicação das melhores práticas. Um dos aspectos mais importantes desta abordagem são as atividades relacionadas com a comunicação pessoal de conhecimento em *software*, por meio de *software patterns*. Comunicação pessoal no sentido mais amplo do termo: como transferência de significado entre pessoas. Para tanto essas atividades envolvem pelo lado do escritor de *patterns* reflexão, escrita e revisão e pelo lado do usuário o entendimento e aplicação destes *patterns* (Salviano, 1997).

A definição de *pattern* mais difundida tem sido

“*pattern* é a essência de uma solução de um problema recorrente em um contexto”.

Solução de um problema significa que cada *pattern* identifica e descreve uma solução e o problema que aquela solução resolve. **Essência de uma solução** significa que apenas os elementos essenciais são descritos, deixando os aspectos específicos para geração pela pessoa que utiliza o *pattern*. Estes aspectos específicos tendem a ser dependentes da situação específica e portanto não são reutilizáveis. **Problema recorrente** significa que o problema deve ter ocorrido várias vezes de modo a viabilizar o esforço a ser investido na descrição da solução. Em um contexto significa que aquela solução é reconhecida como válida em um contexto particular. Coplien (1996) descreveu as seguintes considerações sobre a definição de *pattern*, referindo-se várias vezes às definições originais de Alexander:

“Um pattern é uma peça de literatura que descreve um problema de projeto e uma solução geral para o problema em um contexto particular. Alexander define: cada pattern é uma regra com três partes, que expressam a relação entre um certo contexto, um problema e uma solução (Alexander, 1979). Um pattern, em resumo, é ao mesmo tempo algo que acontece no mundo, e a regra que nos diz como isso deve ser criado, e quando devemos criá-la. Ele é tanto o processo como o objeto; tanto a descrição de um objeto que está vivo, e a descrição do processo que gera este objeto (Alexander, 1979).”

Ainda segundo Coplien (1996), um bom *pattern* é aquele que faz o seguinte:

resolve um problema pois capturam soluções, não apenas princípios ou estratégias abstratas;

conceito provado pois capturam soluções com um caminho registrado, não teorias ou especulações;

a solução não é óbvia várias técnicas de resolução de problemas (como métodos e paradigmas para projeto de *software*) tentam derivar soluções a partir de primeiros princípios. Os melhores *patterns* geram a solução para um problema de forma indireta. Esta é uma abordagem necessária para problemas difíceis;

descreve um relacionamento *patterns* não descrevem módulos, mas sim estruturas e mecanismos profundos de um sistema;

tem um significativo componente humano: todo sistema de *software* serve ao conforto humano e a qualidade de vida. Os melhores *patterns* descrevem explicitamente esta característica.

Linguagem de *patterns* (*pattern language*) é a coleção de *patterns*, em todos os níveis de abstração, que trabalham juntos para resolver um problema complexo em uma solução ordenada de acordo com uma meta (Appleton, 1997). Meszaros and Doble (2001) organizaram o conhecimento atual sobre a escrita de *patterns* e linguagens de *patterns* em uma linguagem de *patterns* para escrita de *patterns*, ou seja, a abordagem foi utilizada para definir a própria abordagem.

Um outro exemplo interessante de *software pattern* é a utilização de *patterns* para descrever a organização de processos. Coplien (1995) descreve 42 *patterns* para orientar a estrutura de organizações de *software* e seus processos de desenvolvimento. Esses *patterns* formam uma linguagem e estão baseados em estudos de organizações de *software* de alta produtividade. Os *patterns* combinam observações empíricas com o raciocínio que os explicam.

4.4 Arquitetura, Componentes, Frameworks, Patterns e todo o Resto

Em algum lugar no passado remoto um pequeno grupo aleatório de átomos foi seriamente traumatizado e levado da esterilidade vazia do espaço a se juntaram no mais extraordinário e jamais visto padrão. Esses padrões rapidamente aprenderam a se copiarem (isto era parte do que era extraordinário sobre esses padrões) e causaram uma grande confusão em cada planeta que eles foram levados. Assim foi que a vida começou no universo...

– DOUGLAS ADAMS (THE HITCHHIKER'S GUIDE TO THE GALAXY)

Reutilização de *software* em geral tem sido um objetivo primordial em Engenharia de *Software*. Especialistas na reutilização sugerem que o sucesso nessa prática deve envolver mais do que o uso de componentes de código isolados. Esses especialistas sugerem que uma técnica robusta de reutilização deve incluir a reutilização sistemática de *designs*, arquiteturas, *frameworks*, entre outros.

A comunidade de *patterns* tem aplicado a mesma idéia de *patterns* sobre diferentes arquiteturas, para coletar soluções que ocorrem rotineiramente em problemas de arquitetura (conhecidos como *architectural patterns*) (Buschmann et al., 1996; Fowler, 1997; Mowbray and Malveau, 1997). Alguns desses *patterns* incluem *Layers*, *Pipe and Filters*, *Broker*, *Blackboard*, *Horizontal-Vertical Metadata* e *Model-View-Controller*. Outros *patterns* foram desenvolvidos para serem utilizados durante a fase de análise (conhecidos como *analysis patterns*) (Fowler, 1997), refinamento de sub-sistemas e componentes (conhecidos como *design patterns*) (Gamma et al., 1994) e implementação (conhecidos como idiomas) (Buschmann et al., 1996).

Design patterns podem ser utilizados para definir a interação entre componentes, mais especificamente como uma “interação padronizada”. *Architectural patterns* são similares, mas a sua utilização é focalizada em um nível mais alto de hierarquia, definindo a interação entre sub-sistemas e até mesmo sistemas. A utilização de interações padronizadas por meio de *patterns*, ao invés de especificar apenas as interações, torna as soluções mais genéricas. Muitos livros, como (Larman, 2002;

Jacobson, Griss, and Jonsson, 1997), exemplificam o desenvolvimento e reutilização de arquiteturas pela utilização de *software patterns*.

Frameworks, por outro lado, implementam estruturas chaves e mecanismos de *design* que tem como objetivo facilitar o desenvolvimento de componentes independentemente e utilizá-los por meio de mecanismos de suporte por ele providos, agregando esses componentes à interface do sistema. Não é uma tarefa fácil desenvolver novos componentes sem um entendimento claro de como eles interagem com o *framework* e os outros componentes. Na verdade, é importante documentar e impingir essa restrição, e isso não é fácil em muitos casos. Uma maneira de assegurar isso é descrever o *framework* e o modelo de componentes como *patterns*, ou mais especificamente, como os modelos de interação que os novos componentes devem cumprir.

Muitos *patterns*, como o *Broker* ou *Model-View-Controller* são, sozinhos, o centro de um *framework* que resolvem uma combinação de problemas arquiteturais e de projeto. Por exemplo, o *pattern Broker* (Pree, 1994) é um mecanismo genérico para o gerenciamento de objetos distribuídos. O *Broker* possibilita que objetos chamem outros objetos remotos por meio de um processo ou serviço que transfere a chamada para o computador apropriado que suporta o componente requisitado. Esse repasse é feito transparentemente (muitas vezes utilizando o *pattern proxy* o que significa que quem a fez não precisa saber nem mesmo se o objeto chamado é remoto ou não) (Mowbray and Malveau, 1997).

Existem outros *patterns* que ajudam a preencher a estrutura geral do sistema. Alguns exemplos são os *patterns Layers, Client-Server, Three-Tier* e *Peer-to-Peer*. Esses *patterns* definem a estrutura de interação entre os componentes, e também sugerem como os componentes devem ser alocados no sistema. Muitas vezes um *pattern* é claramente predominante, preenchendo todo o sistema, enquanto outros podem ser significantes mas com uma menor importância.

O próprio conceito de desenvolvimento baseado em componente pode ser descrito como um *pattern*, como é mostrado graficamente na Figura 4.1 (Bachman et al., 2000). Esse *pattern* é encontrado em tecnologias comerciais de componentes de *software*, tais como o Enterprise JavaBeans (SUN Microsystems, 1996), CORBA (Object Management Group, 2001) e COM+ (Microsoft Corporation, 1998) e também em sistemas customizados.

No *pattern* baseado em componentes (Figura 4.1), um componente(1) é uma implementação em *software* que pode ser executada em um dispositivo físico ou lógico. Um componente implementa uma ou mais interfaces que lhe são impostas(2). Esse fato implica que um componente deve satisfazer certas obrigações, denominadas de contrato(3). Essas obrigações contratuais implicam que componentes independentemente desenvolvidos obedecem certas regras de como esses componentes interagem (ou não podem interagir), de modo que possam ser utilizadas em ambientes padronizados e montados em tempo de construção e/ou execução(4). Um sistema baseado em componentes é baseado em um pequeno número de tipos de componentes distintos, cada um deles responsável por uma tarefa especializada no sistema(5) e que

Figura 4.1: O *Design Pattern* Baseado em Componentes

é descrita na interface(2). Um modelo de componente (6) é o conjunto de tipos de componentes, suas interfaces e, adicionalmente, uma especificação dos padrões de permissões de interação entre os componentes. Um *framework* de componente provê uma variedade de serviços que podem ser utilizados em tempo de execução(8) para suportar e obrigar o cumprimento do modelo de componente. Em muitos aspectos, o *framework* de componente é parecido com um sistema operacional de propósito especial, funcionando, entretanto, em um nível de abstração muito mais alto.

A formalização de componentes e conectores, descrevendo os conectores como entidades e desenvolvendo uma teoria, linguagens e ferramentas para a manipulação de arquiteturas teve como resultado os estilos de arquitetura (Griss, 1998), discutidos na Seção 3.4 na página 33. A partir da perspectiva de modelagem, um estilo de arquitetura define um conjunto de (bons) modelos de projeto que podem ser construídos utilizando *patterns* para maximizar o compromisso entre estrutura e restrições. O estilo, por meio dos *patterns* e restrições, provê um vocabulário de alto nível, permitindo que projetistas e arquitetos com um bom entendimento da estrutura e uma boa descrição dos benefícios e consequências se comuniquem.

De uma maneira geral, cada um desses conceitos pode ser aplicado a uma ou mais etapas do desenvolvimento de um sistema. Na Figura 4.2 estão representados os principais elementos de reutilização de *software* e a sua relação com o nível de abstração e a etapa de projeto ou implementação do sistema.

Figura 4.2: Elementos de Reutilização de *Software*

4.5 Considerações Finais

Todo grande movimento deve experimentar três estágios: ridicularização, discussão, adoção.

– JOHN STUART MILL

Um componente de *software* combina duas perspectivas distintas: componentes como implementação e componentes como uma abstração arquitetural. Componentes são então implementações arquiteturais. Vistos como implementação, componentes podem ser desenvolvidos e compilados em (sub)sistemas. Vistos como abstrações arquiteturais, componentes expressam regras de projeto que impingem um modelo de coordenação padronizado para todos os componentes. Essas regras de projeto assumem a forma de um modelo de componentes, ou um conjunto de padrões e convenções que cada componente deve obedecer.

Interfaces descrevem aquelas propriedades dos componentes que podem controlar as dependências entre os componentes. Entretanto, o atual estágio das linguagens de especificação de interface são limitadas em suas habilidades de descrever propriedades dos componentes, o que implica que algumas propriedades possam “vazar” pela interface. Contratos mudam o foco da especificação dos componentes isoladamente para padrões de interação, e as obrigações mútuas dos participantes nessa interação.

Modelos de componentes especificam as regras de projeto que devem ser obedecidas pelos componentes. Essas regras de projeto possibilitam a compatibilidade removendo uma variedade de fontes de não casamento de interfaces. Essas regras

asseguram que alguns atributos de qualidade do sistema são conseguidos, e componentes podem ser facilmente integrados em tempo de desenvolvimento ou execução. *Frameworks* provem os serviços de suporte, por meio de um ambiente integrado para a execução dos componente, e impõem o modelo de componentes.

Software Pattern está baseado em pressupostos diferentes daqueles assumidos por grande parte da comunidade de pesquisa em Engenharia de *Software*. Ao invés de valorizar os aspectos “meta” de *software*, tais como, modelos, teorias, formalismos e ferramentas para a produção (ou qualquer outro aspecto) de *software*, *Software Pattern* enfatiza a identificação e comunicação entre pessoas pelo compartilhamento de elementos dos melhores exemplos e práticas em *software*.

Sistemas baseados em componentes são resultados da adoção de uma estratégia de projeto baseada em componentes e de uma tecnologia de componentes de *software* que inclua produtos e conceitos que suportem essa estratégia de projeto. Por estratégia de projeto subentende-se algo muito próximo ao conceito de estilo de arquitetura — um *pattern* de alto nível que é descrito pelos tipos de componentes em um sistema e seus padrões de interação.

O *Framework* Proposto

"Por onde devo começar Majestade?", ele perguntou.
"Comece pelo começo", disse o rei em um tom grave, "e continue até chegar ao fim: então pare."

– LEWIS CARROLL (ALICE NO PAÍS DAS MARAVILHAS)

*D*ada as características de experimentação na pesquisa em descoberta de conhecimento, é extremamente útil ter um sistema que auxilie a configuração e a execução de experimentos.

Em um esforço para atender essa situação, neste trabalho propomos um *framework* de integração de componentes para o sistema DISCOVER, que provê uma infra-estrutura capaz de integrar seus componentes e executar diferentes configurações de experimentos. Esse *framework* provê:

- um mecanismo de adaptação de interface que permite criar componentes a partir de bibliotecas de classes;
- uma linguagem baseada em XML que permite a descrição, tanto dos componentes que podem ser usados no sistema bem como de configurações particulares de experimentos;
- um ambiente de execução.

Utilizando essa infra-estrutura e a interface gráfica, que vem sendo desenvolvida por [Geromini \(2002\)](#) paralelamente ao desenvolvimento deste projeto, pesquisadores poderão criar e testar efetivamente e eficientemente seus experimentos, o que pode resultar em uma significativa redução de esforço, além do reaproveitamento dos programas desenvolvidos.

Esse *framework* também pode servir como uma base para o desenvolvimento tanto de novas ferramentas que explorem outra subáreas de pesquisa, quanto proporcionar uma arquitetura modular que permita sua própria expansão.

Este capítulo tem como objetivo descrever a abordagem utilizada para criar o *framework* de integração do sistema DISCOVER. Detalhes técnicos e de implementação não serão aqui descritos e podem ser encontrados em [Prati and Monard \(2003\)](#).

O restante deste capítulo está organizado da seguinte maneira: Na Seção 5.1 é descrito o *pattern* arquitetural base para a criação do *framework*. Na Seção 5.2 é descrito o mecanismo de adaptação de interface para se criar componentes com as bibliotecas de classes do DISCOVER. Na Seção 5.3 é descrita a linguagem proposta, baseada em XML para descrever componentes e configurações de experimento. Na Seção 5.4 é descrito o ambiente de execução do *framework*. Na Seção 5.5 são discutidas algumas extensões e melhorias ao *framework* proposto e, finalmente, na Seção 5.6 são apresentadas as considerações finais deste capítulo.

5.1 Um *Framework* Baseado em *Patterns*

Se você construiu castelos no ar, não precisa perder o seu trabalho;
isso é o que precisa ser feito. Agora ponha fundações sob eles.

– HENRY DAVID THOREAU

No projeto DISCOVER, pesquisadores desenvolvem bibliotecas de classes e sub-rotinas. Essas bibliotecas podem ser utilizadas para programar novas bibliotecas e, também, por meio da instanciação de objetos dessas classes, programar a execução de experimentos.

Essa maneira de configurar e executar experimentos é extremamente trabalhosa e pouco intuitiva, pois implica que se conheça a documentação, entenda a interface e escreva código para utilizar essas bibliotecas. Ressalta-se o fato dessas bibliotecas conterem um grande número de classes e funções internas, ou seja, que são utilizadas por elas próprias, o que dificulta ainda mais sua utilização para configurar os experimentos.

A idéia para criar o sistema DISCOVER a partir desse conjunto de classes é criar mecanismos para encapsular essas bibliotecas de classes como componentes, permitindo uma fácil composição desses componentes pela utilização de uma interface gráfica. Isso possibilita ao DISCOVER ter um papel dual, servindo para duas categorias de pesquisadores¹:

desenvolvedor pesquisadores que utilizam a biblioteca de classes e seus próprios códigos para desenvolver novas bibliotecas de classes, das quais algumas delas se tornarão componentes do DISCOVER;

usuário pesquisadores que utilizam os componentes construídos a partir das classes de componentes para montar e executar configurações de experimentos, assistidos pela interface gráfica do DISCOVER.

¹Um mesmo pesquisador pode estar nas duas categorias

Para se construir um sistema a partir desse conjunto de classes, sem levar em consideração a interação com o usuário, que é tratada, como já mencionado, em um outro plano de pesquisa correlacionado (Geromini, 2002), três questões principais surgem no que diz respeito ao seu projeto:

1. definição das interfaces independentes da implementação (horizontais);
2. definição das interfaces dependentes da implementação (verticais);
3. definição de um ambiente de execução.

Quanto à primeira questão, um acesso eficiente às classes de objetos que já estão implementadas é facilitada pela utilização de um formato uniforme para a composição como componentes. Além disso, essa abordagem permite uma fácil extensão, fazendo com que os novos componentes também obedeçam a esse formato.

Entretanto, a forma usual de orientação à objeto é baseada quase que exclusivamente na análise do domínio da aplicação, o que produz interfaces especializadas e específicas àquele domínio, chamadas de interfaces verticais. Interfaces verticais também são o resultado do uso de subsistemas de *software*. No caso do DISCOVER, temos esses dois tipos de interfaces verticais: as bibliotecas de classes implementadas pelos pesquisadores de nosso laboratório e os algoritmos de aprendizado que serão utilizados como subsistemas.

Interfaces verticais têm a desvantagem de serem especializadas para uma necessidade única, e, dessa forma, muito específicas a uma implementação. Normalmente, a maioria das interfaces de *software* não são particularmente adequadas para atender necessidades de intercâmbio e interoperação.

Quando esses subsistemas são considerados como parte de uma grande classe de subsistemas, é possível definir novas interfaces que são aplicadas à toda a classe do subsistema, e não somente a uma instância. Interfaces que se aplicam a uma categoria inteira de subsistemas são chamadas de interfaces horizontais.

Se existirem vários subsistemas no mesmo sistema que podem ser categorizados em conjunto, então existe um benefício de intercâmbio e interoperação imediato. Uma vez que as interfaces horizontais não são dependentes das implementações internas, é possível fazer mudanças locais aos subsistemas sem um impacto no sistema como um todo.

As interfaces verticais e horizontais definem as estruturas estáticas do sistema. Uma maneira de definir a estrutura dinâmica do sistema é pela utilização de metadados. Com a utilização de metadados é possível postergar algumas das decisões do sistema para a execução, como a instanciação de objetos. A reunião das interfaces horizontal, vertical e metadados definem o *pattern* Horizontal-Vertical-Metadata (HVM) (Mowbray and Malveau, 1997). Uma representação esquemática desse *pattern* pode ser visto na Figura 5.1.

Figura 5.1: O *Pattern* Metadata

A eficácia dos metadados está correlacionada com a prevalência das interfaces horizontais. O uso das interfaces horizontais possibilita aos clientes se integrarem com uma grande faixa de objetos implementados, mas com menos codificação do que se a implementação suportasse apenas interfaces verticais. O uso de interfaces verticais faz a descrição dos metadados mais simples e consistentes. Se esses metadados fossem tomados com base na interface vertical, seria extremamente difícil compô-los e utilizá-los eficientemente.

Entretanto, interfaces verticais também têm o seu papel em uma implementação HVM. As interfaces verticais podem capturar funcionalidades específicas e necessidades de desempenho que não seriam alcançadas pela generalização provida pela interface horizontal e pelos metadados. Uma importante convenção é que os subsistemas deveriam ser escritos para interoperarem usando somente interfaces horizontais e metadados, e, só então, integradas às interfaces verticais para prover funcionalidades especializadas.

5.2 A Interface Horizontal

A ação para ser efetiva deve estar direcionada para conceber claramente um final.

– JAWAHARLAL NEHW

Como dito anteriormente, um formato uniforme para composição tem uma grande importância para o desenvolvimento de interfaces horizontais. No DISCOVER, decidiu-se pela utilização do *design pattern* Facade ([Gamma, Helm, Johnson, and Vlissides, 1994](#)) para a implementação dessas interfaces.

O Facade tem como objetivo prover uma interface unificada em um subsistema. O Facade define uma interface de alto nível que torna esses subsistemas mais fáceis de utilizar. Uma representação esquemática do *design pattern* Facade pode ser vista na Figura 5.2.

Figura 5.2: O *Design Pattern* Facade

Na Figura 5.3 é mostrada a utilização de um subsistema de classes sem (à esquerda) e com (à direita) a utilização do *pattern* Facade. Sem a sua utilização, cada classe cliente acessa diretamente as classes do subsistema. Isso obriga que cada uma das classes clientes conheça cada uma das interfaces das classes que acessa. Pela utilização do Facade, a interface para o subsistema passa a ser única, de modo que os clientes que acessam esse subsistema precisam apenas conhecer essa nova interface.

Figura 5.3: Utilizando o *Design Pattern* Facade

Essa abordagem permite que clientes fiquem protegidos dos componentes do subsistema, reduzindo dessa forma o número de objetos que o cliente utiliza, tornando o subsistema mais fácil de ser utilizado.

Além disso, essa abordagem promove um fraco acoplamento entre os subsistemas e seus clientes. Frequentemente, os componentes em um subsistema tem um forte acoplamento. Entretanto, um fraco acoplamento permite realizar modificações nos componentes do subsistema sem afetar seus clientes. O Facade ajuda a criar uma camada entre o sistema e as dependências entre os objetos. Essas camadas

podem eliminar dependências complexas ou circulares, permitindo que o cliente e o subsistema sejam implementados de maneira independente.

Um outro fator importante é que o Facade não impede que as aplicações utilizem as subclasses do sistema, se necessário, permitindo que se possa escolher entre facilidade de utilização e generalidade.

5.3 Uma Linguagem em XML para Especificar Componentes e Configurações de Experimentos

A linguagem exerce um poder oculto, como a lua na maré.

– RITA MAE BROWN

Para utilizar o HVM é necessário um mecanismo eficiente para armazenar e recuperar os metadados. Esses metadados devem fornecer informações sobre os componentes. Esses componentes, por sua vez, provêm serviços de processamento e mineração de dados e processamento de padrões.

Além disso, esses metadados devem fornecer informações sobre a dinâmica do sistema. No DISCOVER, essa dinâmica está relacionada à execução de uma configuração de experimento.

Uma configuração de experimento é uma instanciação de componentes de processamento e mineração de dados, processamento de padrões e ligações entre esses componentes as quais representam o fluxo na configuração. Dessa maneira, para se utilizar os metadados no DISCOVER, três fatores devem ser levados em consideração:

1. pode-se esperar uma boa quantidade de representações comuns entre a descrição de componentes e as configurações de experimentos no DISCOVER;
2. representações de experimentos podem ser divididas em conjuntos de componentes e ligações entre esses componentes, reunidos em uma estrutura hierárquica;
3. uma boa representação para os componentes e as configurações de experimentos deve ser modular e extensível.

A estrutura hierárquica, a modularidade e a necessidade de extensão das representações de componentes e configurações de experimentos podem ser descritas utilizando representações em XML — Linguagem de Marcação Extensível (eXtensible Markup Language). Nesta seção é descrita a linguagem proposta para a representação de componentes e configurações de experimentos, utilizando esquemas de XML.

XML é uma linguagem de marcação extensível baseada na Linguagem de Marcação Padrão Generalizada — SGML (Standard Generalized Markup Language) ([International Organization for Standard ISO, 1986](#)), projetada para descrever dados. A XML

também oferece uma grande quantidade de ferramentas de processamento disponíveis para esta linguagem. As duas principais características da XML são a ênfase em uma marcação descritiva ao invés da procedural e a o conceito de tipo de documento. Utilizando XML é possível definir um conjunto próprio de marcas. Além disso, pela utilização de esquemas de XML, é possível definir estruturas (tipos) de documentos.

Esquemas de XML ([Fallside, 2001](#)) tornaram-se recentemente uma recomendação do consórcio da World Wide Web — W3C², provendo uma rica e extensível meta-linguagem com a qual é possível criar novos formatos de representação.

A meta-linguagem original provida pela linguagem XML foi a Definição de Tipos de Documentos — DTD (Document Type Definition). A utilização de DTDs permite a autores especificarem quais elementos e atributos podem estar presentes em um documento em conformidade com a linguagem. Entretanto, esquemas de XML são mais expressivos que DTDs, primeiramente porque eles provêem um sistema muito mais rico que oferece um suporte significativamente melhor para o desenvolvimento de linguagens extensíveis e modulares. Autores de esquemas de XML podem definir tipos básicos em um esquema e estender esses tipos em outros esquemas, adicionando, modificando e, em muitos casos, removendo elementos ou atributos em tipos estendidos. Ao contrário de DTDs, essas operações não requerem modificações na base do esquema.

A linguagem aqui proposta, denominada de xDML— Linguagem de Marcação Extensível do Discover (eXtensible Discover's Markup Language) — está baseada na linguagem para descrever famílias de arquiteturas de *software* xADL ([Dashofy and van der Hoek, 2001](#)).

xDML consiste de um conjunto de esquemas de XML que capturam os elementos básicos de configurações de experimentos: estrutura, componentes, conectores e ligações. A descrição completa em XML da linguagem xDML pode ser encontrada no Apêndice A na página 91. Uma outra característica interessante da linguagem xDML é a possibilidade de definir agrupamentos desses componentes básicos. Cada um desses elementos são detalhados a seguir.

5.3.1 Descrição de uma Estrutura

Uma estrutura permite descrever no DISCOVER conjuntos de componentes e configurações de experimentos. A definição da estrutura é composta por uma descrição, componentes, conectores, ligações e grupos. Uma ilustração do esquema de uma estrutura pode ser vista na Figura 5.4.

5.3.2 A Descrição de um Componente

Componentes são os elementos principais da xDML. A linguagem xDML possibilita a descrição de dois tipos de componentes:

²www.w3c.org

Figura 5.4: Representação do esquema de uma estrutura em xDML

componentes simples componentes que contenham uma implementação codificada;

subestruturas componentes que contenham uma subestrutura interna, que é descrita de uma maneira recursiva como uma estrutura.

Uma representação gráfica do esquema de um componente pode ser visto na Figura 5.5. Um componente é composto por uma descrição, uma ou mais interfaces de entrada/saída e uma implementação ou subestrutura, descritas a seguir.

A descrição compreende uma entrada de texto, a qual tem como objetivos fornecer uma descrição a respeito daquele componente.

Cada uma das interfaces representa um ponto de entrada, saída ou entrada e saída do componente. Elas são compostas de uma descrição e da direção correspondente.

Componentes simples estão correlacionados a uma implementação. Essa implementação contém a localização das classes que implementam o componente. Cada implementação deve ter uma classe principal, mas também pode referenciar classes auxiliares necessárias para a sua execução. Também faz parte de um componente simples os parâmetros necessários para a sua execução bem como as possíveis exceções que podem ser por ele lançadas.

Uma subestrutura é composta por uma descrição, uma definição recursiva de uma estrutura, um mapeamento da interface do componente que tem uma subestrutura e as interfaces dentro dos componentes da subestrutura.

5.3.3 A Descrição de um Conector

Uma outra possibilidade da linguagem xDML é a descrição de conectores. Conectores são usados para representar ligações múltiplas. Eles são formados por uma descrição e por interfaces.

No caso de haver alguma ação relacionada ao conector, uma subestrutura pode ser associada ao conector. Nesse caso, as interfaces de conector são mapeadas nas interfaces dos componentes da subestrutura associada ao conector. A representação gráfica do esquema do conector pode ser vista na Figura 5.6.

Figura 5.5: Descrição de um componente em xDML

Figura 5.6: Representação do esquema de um conector em xDML

5.3.4 A Descrição de uma Ligação

Ligações representam uma conexão entre duas interfaces. Todas as ligações existem entre interfaces de componentes ou conectores. Uma ligação tem uma descrição e exatamente dois pontos finais de ligação. O esquema de uma ligação está representado graficamente na Figura 5.7.

Figura 5.7: Representação do esquema de uma ligação em xDML

5.3.5 A Descrição de um Grupo

Grupos descrevem um agrupamento lógico de outros elementos (componentes, conectores e ligações), de forma que seja possível agrupar esses elementos de maneiras arbitrárias, como por exemplo: por autor, categoria, máquina (em um ambiente distribuído), etc. Grupos também podem ser membros de outros grupos.

Uma representação gráfica do esquema de um grupo pode ser encontrada na Figura 5.8. Grupos são compostos por uma descrição e por um conjunto de elementos membros.

Figura 5.8: Representação do esquema de um grupo em xDML

5.4 O Ambiente de Execução

Comece fazendo o necessário, depois o possível, e logo você estará fazendo o impossível.

– SÃO FRANCISCO DE ASSIS

Como dito anteriormente, um outro ponto importante para se criar o *framework* de integração do DISCOVER é o ambiente de execução. Esse ambiente de execução deve receber as configurações de experimentos descritas em xDML e executá-las, devolvendo os resultados obtidos.

No DISCOVER, esse ambiente de execução é composto pela interação de dois *patterns*: o Interpretar e Visitor (Gamma, Helm, Johnson, and Vlissides, 1994).

O *pattern* Interpretar define uma representação de uma dada linguagem (no caso do DISCOVER, a linguagem xDML) em conjunto com um interpretador que usa a representação de um experimento para interpretar as sentenças nessa linguagem. Uma representação esquemática desse *pattern* pode ser vista na Figura 5.9. Esse *pattern* é composto por cinco componentes:

Elemento Abstrato declara um interpretador abstrato que é comum a todos os nós da árvore sintática.

Figura 5.9: O *pattern* Interpretador

Elemento Terminal implementa um interpretador associado a cada símbolo terminal da linguagem. No caso do DISCOVER, os símbolos terminais são os componentes simples.

Elemento não-Terminal implementa um interpretador associado aos símbolos não terminais. Nesse caso o interpretador chama-se recursivamente para interpretar os símbolos não terminais. No caso do DISCOVER, os símbolos não-terminais são as subestruturas, que estão associadas a componentes compostos e conectores.

Contexto contém informações que são globais ao interpretador.

Cliente fornece uma representação particular abstrata em forma de árvore, que a linguagem define, e invoca o interpretador.

O cliente do *pattern* Interpretador é o responsável por criar a representação da linguagem em uma estrutura de árvore. Como a linguagem xDML é representada em XML, pode-se utilizar um *parser* XML que cria uma árvore DOM — *Document Object Model* — e, a partir dessa árvore, invocar o interpretador.

O componente contexto é o responsável por guardar informações sobre o gerenciamento de exceções, objetos instanciados, arquivos abertos, etc.

As operações de interpretação são feitas utilizando o *pattern* Visitor. Esse *pattern* representa uma operação a ser executada para elementos de uma dada estrutura. Além disso, o Visitor permite definir novas operações de interpretação sem mudar a classe dos elementos sobre as quais opera. Uma representação esquemática do *pattern* Visitor pode ser vista na Figura 5.10. Esse *pattern* é composto por cinco componentes:

Visitante declara uma operação de visita para cada classe Elemento Concreto na estrutura do objeto. O nome e assinatura da operação identificam a classe que envia a solicitação de visita ao visitante. Isso permite ao visitante determinar

Figura 5.10: O *pattern* Visitor

a classe concreta do elemento que está sendo visitado. Então, o visitante pode acessar o elemento diretamente por meio da interface.

Visitante Concreto implementa cada operação declarada pelo Visitante. Cada operação implementa um fragmento do algoritmo definido para a correspondente classe de objeto na estrutura. O Visitante Concreto fornece o contexto para o algoritmo e armazena o seu estado local. Esse estado, freqüentemente, acumula resultados obtidos durante o percurso da estrutura.

Elemento define uma operação Aceita que aceita um visitante como um argumento.

Elemento Concreto implementa uma operação Aceita que aceita um visitante como um argumento.

Estrutura de Objetos contém a estrutura dos objetos a serem visitados. Ela também define uma enumeração de objetos que impinge uma ordem para a visitação.

O cliente do *pattern* Visitor (no caso, o *pattern* Interpretar), deve criar o objeto VisitanteConcreto e, então, passar essa estrutura para que ela seja percorrida pelo visitante. Quando um elemento é visitado, chama a operação do Visitor que corresponde ao seu tipo. O diagrama de interação é mostrado na Figura 5.11.

5.5 Extensões e Melhorias

Toda a extensão de conhecimento surge a partir de tornar consciente o inconsciente.

– FRIEDRICH NIETZSCHE

A arquitetura modular do *framework* proposto permite que várias extensões e melhorias sejam feitas. Pode-se, por exemplo, implementar um visitante utilizando o

Figura 5.11: Diagrama de interação do *pattern Visitor*

pattern Broker (Buschmann, Meunier, Rohnert, Sommerland, and Stal, 1996), para que o visitante possa ser executado em um ambiente distribuído.

O Broker é um *pattern* arquitetural que pode ser utilizado para estruturar sistemas de *software* distribuídos que interagem pela invocação de serviços remotos. Um componente *broker* é responsável por coordenar a comunicação nesse ambiente, tais como o repasse de mensagens, e transmitir resultados e exceções. Nesse caso, os componentes que prestam serviços também deveriam implementar um servidor *proxy* e os componentes que requisitam serviços deveriam implementar clientes *proxy*.

Uma outra alternativa é implementar um visitante utilizando o *pattern* DecompositionStrategy (Massingill et al., 1999a,b) para a execução concorrente.

Programas paralelos ajudam a diminuir o tempo de execução de programas, devido ao fato de diferentes problemas serem resolvidos em diferentes processos alocados em diferentes processadores. Isso só pode ser feito se os problemas a serem resolvidos contenham uma concorrência explorável, isto é, múltiplas atividades ou tarefas podem ser executadas ao mesmo tempo. Explorar essa concorrência requer uma decomposição do problema em duas diferentes dimensões:

decomposição de tarefas quebram o fluxo de execução de instruções em múltiplos blocos, chamados de tarefas, que podem ser executados simultaneamente. Para conseguir um razoável aumento de performance, essas tarefas precisam ser executadas com uma necessidade mínima de interação, isto é, a sobrecarga associada com o gerenciamento das dependências deve ser pequeno comparado com a execução total do programa;

decomposição de dados determina como os dados interagem com as tarefas. Muitos dados são modificados por somente uma tarefa, ou seja, são locais a cada tarefa. Para dados desse tipo, deve-se determinar como deve ser dividido o conjunto de dados e associá-los apropriadamente a cada tarefa. Outros dados são modificados por múltiplas tarefas, ou seja, os dados são compartilhados por elas.

As atividades relacionadas às atividades de mineração de dados, tanto distribuídas quanto paralelas, têm tido um considerável aumento no número de pesquisas nos últimos anos (Freitas and Lavington, 1998; Zaki and Ho, 2000) devido ao aumento da dimensionalidade dos conjuntos de dados tratados. Entretanto, uma descrição mais detalhada a respeito foge ao escopo desta dissertação. Deve ser ressaltado que a arquitetura modular do *framework* de integração proposto permite deixar uma porta em aberto para uma futura realização de pesquisas nessa área.

5.6 Considerações Finais

Não há erro mais grave do que a conclusão precipitada de que as opiniões são sem valor porque são mal discutidas.

– THOMAS H. HUXLEY

Três questões principais devem ser respondidas na criação de um sistema como o DISCOVER: como são definidas as interfaces verticais, horizontais e o ambiente de execução. Neste capítulo é apresentado um *framework* de integração de componentes para o sistema discover que tenta apresentar uma solução para essas questões.

Esse *framework* compreende um mecanismo de adaptação de interface que cria uma camada (interface horizontal) sobre as interfaces verticais; um poderoso mecanismo de metadados, que é utilizado para descrever tanto os componentes que implementam as funcionalidades do sistema quanto as configurações de experimentos criadas pelos usuário, que serão executadas pelo *framework*; e um ambiente de execução para essas configurações de experimentos.

No próximo capítulo é apresentado um exemplo da execução desse *framework* no sistema DISCOVER.

Exemplo da Utilização do Framework

Exemplo não é a coisa mais importante que influencia os outros. É a única coisa.

– ALBERT SCHWEITZER

Neste capítulo é apresentado um exemplo da utilização do *framework* de integração do DISCOVER. Este capítulo está organizado da seguinte forma: na Seção 6.1 é apresentada a interação do DISCOVER com o usuário. Na Seção 6.2 é mostrado um exemplo utilizando apenas componentes simples e ligações. Na Seção 6.3 esse exemplo é remodelado, utilizando-se conectores para substituir alguns componentes. Na Seção 6.4 é mostrada a definição de um componente composto a partir de dois componentes simples. Na Seção 6.5 é apresentado um exemplo da criação de um componente a partir das bibliotecas de classe. Na Seção 6.6 é apresentado um exemplo da execução de um experimento utilizando o ambiente de execução. Finalmente, na Seção 6.7 são apresentadas as considerações finais deste capítulo.

6.1 A Interação com o Usuário

Eu ouço e esqueço. Eu vejo e relembro. Eu faço e entendo.

– CONFÚCIO

Além do *framework* de integração, o sistema DISCOVER proverá uma outra ferramenta para auxiliar pesquisadores na atividade de configuração de experimentos: uma interface gráfica (Geromini, 2002). Essa interface gráfica possibilitará criar as configurações de experimentos com operações de “arrastar e soltar”.

O que se pretende é a utilização de ícones representando os diversos componentes simples e complexos (subestruturas). Esses componentes implementam algum serviço de processamento e mineração de dados ou processamento de padrões, e setas, representando ligações e conectores, que indiquem as relações entre esses componentes. O diagrama obtido por esse conjunto de ícones e setas compõem o fluxo de

processamento daquela configuração particular, formando estruturas que podem ser traduzidas como grafos. A utilização desse tipo de interação com o usuário, bem como a representação dos processos por meio de grafos, é extremamente amigável e natural para o usuário. Na Figura 6.1 é mostrado um modelo da interface gráfica do DISCOVER, no qual o usuário está desenhando uma configuração particular de experimento.

Figura 6.1: Modelo da Interface Gráfica do DISCOVER

A linguagem xDML, descrita na Seção 5.3, funciona como um elo de ligação entre a interface gráfica e o *framework* de integração. Essa linguagem tem um papel dual, servindo tanto para descrever os componentes utilizados no sistema, como para configurações particulares de experimentos.

Quanto ao primeiro papel, a interação *framework*-interface gráfica, é possível definir em xDML quais são os componentes e conectores que fazem parte do sistema. Quando iniciada, a interface gráfica lê, a partir de um arquivo em xDML, quais são os componentes e conectores disponíveis para montar uma paleta de ícones. Grupos podem ser usados para reunir esses elementos em classes, o que possibilita a criação de várias paletas para cada uma dessas classes (como, por exemplo, uma paleta para processamento de dados, algoritmos de aprendizado, etc.). Além disso, o código XML correspondente a esses elementos pode ser utilizado como um molde para a instanciação desses elementos.

No que se refere à interação interface gráfica-*framework*, a partir de um grafo representando uma configuração particular de experimento, a interface gráfica irá traduzir essa representação em instâncias de componentes (simples e complexos), conectores e ligações, pertencentes a uma estrutura. Essa estrutura descreve, de maneira única e uniforme, essa particular configuração de experimento.

Dessa forma, o experimento construído pelo usuário usando a interface gráfica é codificado em xDML, guardando todas as configurações particulares do usuário. Essa descrição pode também ser guardada em um arquivo, permitindo uma posterior modificação/utilização. Essa abordagem também possibilita que ambos *framework* e interface gráfica sejam desenvolvidos e mantidos separadamente.

6.2 Um Primeiro Exemplo

Deixe isto ser um exemplo para a aquisição de todo o conhecimento, virtude, e riqueza. Pela queda das gotas da água, aos poucos, um pote é enchido.

– A HITOPADESA (LIVRO DOS BONS CONSELHOS)

Nesta seção é apresentado um exemplo simples da utilização da linguagem xDML para descrever componentes e configurações de experimentos no DISCOVER. Nesse exemplo, utiliza-se um conjunto de dados arbitrário `exemplo.all`¹, e sua respectiva definição `exemplo.names`, ambos na sintaxe padrão do DISCOVER (Batista, 2001). Quer-se dividir esse conjunto de dados em dois, um deles, chamado `exemplo.data` contendo 2/3 dos exemplos para treinamento e o outro, chamado de `exemplo.test`, contendo 1/3 dos exemplos para avaliação.

O objetivo do experimento é utilizar o algoritmo de indução de árvore de decisão C4.5 (Quinlan, 1993) no conjunto de exemplos de treinamento para induzir uma árvore de decisão, transformar essa árvore de decisão no formato padrão de regras do DISCOVER (Prati, Baranauskas, and Monard, 2001b), e calcular as informações padrão para a avaliação dessas regras (Prati, Baranauskas, and Monard, 2001a, 2002), utilizando o conjunto de exemplos de teste.

Para realizar essa tarefa, necessitamos de 6 componentes:

1. um componente que leia o conjunto de dados e carregue-o na memória;
2. um componente que divida o conjunto de dados e gere os novos arquivos de treinamento e teste;
3. um componente que envolva e sirva de interface para o algoritmo de aprendizado de máquina C4.5;
4. um componente que leia a árvore de decisão induzida pelo C4.5 e transforme essa árvore para o formato padrão de regras;
5. um componente que calcule as informações adicionais dessas regras no formato padrão e
6. um componente que armazene em disco essas regras com as informações no formato padrão.

¹Observar que, na realidade, `exemplo.<ext>` refere-se ao nome do arquivo no qual encontram-se armazenados os dados/informações correspondentes.

Esses componentes foram construídos tendo-se como base as bibliotecas de processamento de dados (Batista, 2001) e regras simbólicas (Prati et al., 2001b) como descrito na Seção 5.2 na página 60. Essa transformação será exemplificada posteriormente na Seção 6.5. Como dito anteriormente, a linguagem xDML tem um papel dual, servindo para representar tanto configurações de experimentos quanto os componentes que podem ser utilizados no DISCOVER.

Na Listagem 6.1 é feita uma descrição (compacta) dos componentes usados nesse exemplo. Essa descrição é a passada para a interface gráfica, na interação *framework*-interface gráfica, e não contém ligações, pois refere-se apenas à descrição dos componentes, e não de uma configuração de experimento. Essas descrições serão utilizadas como moldes para a instanciação dos componentes presentes no experimento. Por motivos de espaço, somente os componentes utilizados neste exemplo são descritos.

Entre as linhas 8 a 14 da listagem são feitas as descrições dos componentes ² Entre as linhas 15 a 37 é feita a descrição dos grupos componentes. Um componente é membro de um grupo fazendo-se uma referência a esse componente como membro do grupo. Por exemplo, o componente LOAD_DATA é declarado como membro do grupo DATA_PROCESSING nas linhas 17 e 18. Além disso, como dito anteriormente, um componente pode fazer parte de mais de um grupo (como autor e função) e um grupo também pode ser membro de outro grupo, o que possibilita a criação de uma estrutura hierárquica de componentes em grupos e subgrupos.

Código 6.1: Descrição (compacta) em XML dos componentes

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <xDML xmlns:xlink="http://www.w3.org/1999/xlink"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:noNamespaceSchemaLocation="Z:\xDML.xsd">
5   <structure id="Discover">
6     <description>This file describes the components used in the
7     Discover System.</description>
8     + <component id="LOAD_DATA">
9     + <component id="SPLIT_DATA">
10    + <component id="SAVE_DATA">
11    + <component id="C45_INDUCER">
12    + <component id="STANDARD_RULES">
13    + <component id="STANDARD_INFO">
14    + <component id="SAVE_RULES">
15    <group id="DATA_PROCESSING">
16      <description>Data processing</description>
17      <member xlink:href="#LOAD_DATA" xsi:type="XMLLink"
18        xlink:type="simple" />
19      <member xlink:href="#SPLIT_DATA" xsi:type="XMLLink"
20        xlink:type="simple" />
21      <member xlink:href="#SAVE_DATA" xsi:type="XMLLink"
22        xlink:type="simple" />
23    </group>

```

²As listagens não estão completas por motivos de espaço, e para um melhor fluxo de leitura.


```

24     <group id="LEARNING_ALGORITHM">
25         <description>Learning Algorithm</description>
26         <member xlink:href="#C45_INDUCER" xsi:type="XMLLink"
27             xlink:type="simple" / >
28     </group>
29     <group id="PATTERN_PROCESSING">
30         <description>Pattern processing</description>
31         <member xlink:href="#STANDARD_RULES" xsi:type="XMLLink"
32             xlink:type="simple" />
33         <member xlink:href="#STANDARD_INFO" xsi:type="XMLLink"
34             xlink:type="simple" />
35         <member xlink:href="#SAVE_RULES" xsi:type="XMLLink"
36             xlink:type="simple" />
37     </group>
38 </structure>
39 </xDML>

```

A descrição do componente `LOAD_DATA` está na listagem 6.2. Os outros componentes têm uma descrição semelhante e são aqui omitidos por uma questão de espaço. A descrição da entrada e saída é feita entre definição das interfaces, entre as linhas 4 e 13. O componente `LOAD_DATA` tem como entrada um conjunto de dados no formato padrão do DISCOVER (Batista, 2001) e, como retorno, um apontador para o objeto em memória que manipula esse conjunto de dados. Segue-se a descrição da implementação (linhas 14 a 30), na qual são descritos a referência à classe principal (linhas 17 a 20), os parâmetros necessários para a execução (linhas 21 a 24) e as exceções que podem ocorrer durante a execução (linhas 26 a 29). Na descrição do componente, além da sua descrição sintática, existe uma conotação semântica associada. Na definição de exceção, por exemplo, se a exceção `er_OpenFileReading` ocorrer, a ação associada é a de parar o processamento.

Código 6.2: Descrição do Componente `LOAD_DATA`

```

1 <component id="LOAD_DATA">
2     <description>This component loads the dataset and
3     returns a dataset object</description>
4     <interface id = "LOAD_DATA(IN) ">
5         <description>Input Data</description>
6         <direction>in</direction>
7         <type>Dataset</type>
8     </interface>
9     <interface id = "LOAD_DATA(OUT) ">
10         <description>Dataset Object</description>
11         <direction>out</direction>
12         <type>Data Object</type>
13     </interface>
14     <implementation>
15         <description>This component was implemented over Batista
16         Core Dataset class</description>
17         <mainClass>

```

```

18     <classname>LoadData</classname>
19     <path>/Discover/LoadData.pm</path>
20 </mainClass>
21 <parameters>
22     <parameterName>Dataset</parameterName>
23     <value>Undefined</value>
24     <required>yes</required>
25 </parameters>
26 <exceptions>
27     <exceptionName>er_OpenFileReading</exceptionName>
28     <action>stop</action>
29 </exceptions>
30 </implementation>
31 </component>

```

Dado esse conjunto de componentes, deseja-se montar o experimento descrito anteriormente. Uma representação esquemática da configuração do experimento, que pode ser montada utilizando a interface gráfica (Geromini, 2002), pode ser vista na Figura 6.2.

Figura 6.2: Configuração do Experimento

A descrição dessa configuração em xDML está na Listagem 6.3. Nessa listagem são feitas instanciações dos componentes do DISCOVER, descritos na Listagem 6.1. Para cada uma dessas instâncias, devem ser definidos identificadores (**id**'s) únicos para componentes e interfaces. Também são definidas as ligações entre os componentes, como mostrado entre as linhas 16 e 32 dessa listagem. Nela, uma ligação (Link1) é detalhada. As ligações são feitas entre dois pontos de interface.

Código 6.3: Configuração de Experimento Utilizando xDML

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <xDML xmlns:xlink="http://www.w3.org/1999/xlink"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:noNamespaceSchemaLocation="Z:\xDML.xsd">
5     <structure id="Example1">
6         <description>This file describes the experiment configuration
7         of the example 1.</description>
8         + <component id="LOAD_DATA_INSTANCE">
9         + <component id="SPLIT_DATA_INSTANCE">
10        + <component id="SAVE_DATA_INSTANCE1">
11        + <component id="SAVE_DATA_INSTANCE2">

```

```

12 + <component id="C45_INDUCER_INSTANCE">
13 + <component id="STANDARD_RULES_INSTANCE">
14 + <component id="STANDARD_INFO_INSTANCE">
15 + <component id="SAVE_RULES_INSTANCE">
16   <link id="Link1">
17     <description>Connects the loaded dataset to the split
18       componente</description>
19     <point>
20       <anchorOnInterface xlink:href="#LOAD_DATA(OUT) "
21         xsi:type="XMLLink" xlink:type="simple"/>
22     </point>
23     <point>
24       <anchorOnInterface xlink:href="#SPLIT_DATA(IN) "
25         xsi:type="XMLLink" xlink:type="simple"/>
26     </point>
27   </link>
28 + <link id="Link3">
29 + <link id="Link4">
30 + <link id="Link5">
31 + <link id="Link6">
32 + <link id="Link7">
33 </structure>
34 </xDML>

```

6.3 Utilizando Conectores

Observações gerais extraídas de observações particulares são as jóias do conhecimento, compreendendo grandes depósitos em pequenas salas.

– JOHN LOCKE

No exemplo da Seção 6.2 são utilizados apenas componentes e ligações. Entretanto, alguns desses componentes são apenas operacionais, como por exemplo ler ou escrever um conjunto de dados ou um conjunto de regras no disco, não tendo nenhuma relação funcional no que diz respeito à lógica do processo.

Para uma melhor abstração do processo estudado, esses componentes podem ser ocultos por conectores, facilitando o entendimento da representação gráfica do processo. Na Figura 6.3 é mostrada a mesma configuração mostrada na Figura 6.2, mas os componentes operacionais foram substituídos por conectores (representados pelas setas largas).

Na Listagem 6.4 é mostrada a redefinição do componente LOAD_DATA, mostrado na Listagem 6.2, como um conector. Nessa nova definição como conector, é feita uma nova declaração da interface, como mostrado entre as linhas 3 e 12 da Listagem 6.4. Essa nova interface é a visível externamente. O componente é recursivamente definido como uma estrutura, como mostrado entre as linhas 13 e 48. É importante ressaltar que essa definição poderia conter uma descrição de estrutura mais comple-

Figura 6.3: Configuração do Experimento Utilizando Conectores

xa, contendo outros componentes, ligações e conectores. Também é feita a definição da ligação da estrutura interna com a interface externa, por meio do mapeamento de uma na outra, como descrito entre as linhas 49 e 60 da Listagem 6.4.

Código 6.4: LOAD_DATA descrito como um conector

```

1 <connector id="LOAD_DATA">
2   <description>This connector wrappers load data component</description>
3   <interface id="LOAD_DATA_IN">
4     <description>Input Data</description>
5     <direction>in</direction>
6     <type>Dataset</type>
7   </interface>
8   <interface id="LOAD_DATA_OUT">
9     <description>Dataset Object</description>
10    <direction>out</direction>
11    <type>Data Object</type>
12  </interface>
13  <subStructure>
14    <description>Load Data</description>
15    <structure>
16      <description>Load Data</description>
17      <component id="LOAD_DATA_COMPONENT">
18        <description>This component loads the dataset and returns a
19        dataset object</description>
20        <interface id="LOAD_DATA_COMPONENT_IN">
21          <description>Input Data</description>
22          <direction>in</direction>
23          <type>Dataset</type>
24        </interface>
25        <interface id="LOAD_DATA_COMPONENT_OUT">
26          <description>Dataset Object</description>
27          <direction>out</direction>
28          <type>Data Object</type>
29        </interface>
30        <implementation>
31          <description>This component was implemented over Batista
32          Core Dataset class</description>
33          <mainClass>
34            <classname>LoadData</classname>

```

```

35      <path>/Discover/LoadData.pm</path>
36    </mainClass>
37    <parameters>
38      <parameterName>Dataset</parameterName>
39      <value/>
40      <type>Data Object</type>
41    </parameters>
42    <exceptions>
43      <exceptionName>er_OpenFileReading</exceptionName>
44      <action>stop</action>
45    </exceptions>
46  </implementation>
47 </component>
48 </structure>
49 <interfaceInstanceMapping>
50   <outerInterfaceInstance xlink:href="#LOAD_DATA_IN"
51     xsi:type="XMLLink" xlink:type="simple"/>
52   <innerInterfaceInstance xlink:href="#LOAD_DATA_COMPONENT_IN"
53     xsi:type="XMLLink" xlink:type="simple"/>
54 </interfaceInstanceMapping>
55 <interfaceInstanceMapping>
56   <outerInterfaceInstance xlink:href="#LOAD_DATA_OUT"
57     xsi:type="XMLLink" xlink:type="simple"/>
58   <innerInterfaceInstance xlink:href="#LOAD_DATA_COMPONENT_OUT"
59     xsi:type="XMLLink" xlink:type="simple"/>
60 </interfaceInstanceMapping>
61 </subStructure>
62 </connector>

```

Os outros componentes operacionais SAVE_DATA e SAVE_RULES também foram definidos como conectores, de uma maneira semelhante. A descrição do experimento ilustrado na Figura 6.3, em xDML, é semelhante à Listagem 6.3, mas com os componentes operacionais substituídos por conectores. As ligações são feitas entre as interfaces dos componentes e/ou as interfaces dos conectores.

6.4 Componentes Compostos

É bom lembrar que todo o universo, com uma única exceção insignificante, é composto por outros.

– JOHN ANDREW HOLMES

Uma outra opção interessante da linguagem xDML é a possibilidade de criar componentes formados por outros componentes, ligações e conectores. Esses componentes são chamados componentes compostos. Essa opção é particularmente útil quando uma mesma tarefa que envolva vários componentes precisa ser repetida um mesmo número de vezes. Nesse caso, a parte do experimento que se repete é utilizada para formar um componente composto, formado por uma subestrutura.

Supondo que a tarefa de transformar a saída de um algoritmo de aprendizado simbólico para o formato padrão de regras, e, a partir dessas regras, calcular as informações padrão, seja uma tarefa rotineira. Assim deseja-se criar um componente composto para executar essas duas tarefas.

Na Listagem 6.5 é mostrada a definição desse componente composto. Entre as linhas 4 a 15 é feita a definição da interface externa do componente composto. Entre as linhas 22 a 40 é feita a descrição do componente proposto propriamente dito, como uma subestrutura.

Código 6.5: Descrição (compacta) em XML dos componentes

```
1 <component id="STANDARD_RULES_INFO">
2   <description>This component converts to standard rule
3   format and calculates standard info</description>
4   <interface id="DATA_IN">
5     <description>Object containing the instances to
6     calculate standard info</description>
7     <direction>in</direction>
8     <type>Dataset Object</type>
9   </interface>
10  <interface id="MODEL_IN">
11    <description>Output generated by a symbolic machine
12    learning algorithm</description>
13    <direction>out</direction>
14    <type>Classification Model</type>
15  </interface>
16  <interface id="RULES_OUT">
17    <description>Rules in the standard format with standard
18    info added</description>
19    <direction>out</direction>
20    <type>Ruleset</type>
21  </interface>
22  <substructure>
23    <structure id="STANDARD_RULES_INFO_STRUCTURE">
24      <description>This component has two subcomponents</description>
25      + <component id="STANDARD_RULES_INSTANCE">
26      + <component id="STANDARD_INFO_INSTANCE">
27        <link id="LinkInstance">
28          <description>Connects the standard rule component
29          to the standard info component</description>
30          <point>
31            <anchorOnInterface xlink:href="#STD_RULES_OUT"
32            xsi:type="XMLLink" xlink:type="simple"/>
33          </point>
34          <point>
35            <anchorOnInterface xlink:href="#STANDARD_INFO_IN"
36            xsi:type="XMLLink" xlink:type="simple"/>
37          </point>
38        </link>
```

```

39 </structure>
40 <interfaceInstanceMapping>
41   <outerInterfaceInstance xlink:href="#DATA_IN"
42     xsi:type="XMLLink" xlink:type="simple" />
43   <innerInterfaceInstance xlink:href="#STANDARD_INFO_DATA_IN"
44     xsi:type="XMLLink" xlink:type="simple" />
45 </interfaceInstanceMapping>
46 <interfaceInstanceMapping>
47   <outerInterfaceInstance xlink:href="#MODEL_IN"
48     xsi:type="XMLLink" xlink:type="simple" />
49   <innerInterfaceInstance xlink:href="#STANDARD_INFO_MODEL_IN"
50     xsi:type="XMLLink" xlink:type="simple" />
51 </interfaceInstanceMapping>
52 <interfaceInstanceMapping>
53   <outerInterfaceInstance xlink:href="#RULES_OUT"
54     xsi:type="XMLLink" xlink:type="simple" />
55   <innerInterfaceInstance xlink:href="#STANDARD_INFO_RULES_OUT"
56     xsi:type="XMLLink" xlink:type="simple" />
57 </interfaceInstanceMapping>
58 </substructure>
59 </component>

```

A representação gráfica do experimento descrito na Seção 6.2 na página 73, utilizando conectores e o componente composto descrito nesta seção, pode ser vista na Figura 6.4. Essa representação é bem mais simples e intuitiva que a mostrada na Figura 6.2.

Figura 6.4: Configuração do Experimento Utilizando Componente Composto

A Listagem 6.6 mostra a descrição da configuração do experimento representado na Figura 6.4. Essa listagem também será utilizada no exemplo da execução de um experimento, utilizando o ambiente de execução do framework, apresentado na Seção 6.6.

Código 6.6: Configuração de Experimentos Utilizando xDML

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <xDML xmlns:xlink="http://www.w3.org/1999/xlink"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:noNamespaceSchemaLocation="Z:\xDML.xsd">
5   <structure id="Example1">
6     <description>This file describes the experiment configuration

```



```
7 of the example 1.</description>
8 + <component id="SPLIT_DATA_INSTANCE">
9 + <component id="C45_INDUCER_INSTANCE">
10 + <component id="STANDARD_RULES_INFO_INSTANCE">
11 + <connector id="LOAD_DATA_INSTANCE1">
12 + <connector id="LOAD_DATA_INSTANCE2">
13 + <connector id="SAVE_RULES_INSTANCE">
14 + <link id="Link3">
15 + <link id="Link4">
16 + <link id="Link3">
17 + <link id="Link4">
18 + <link id="Link5">
19 </structure>
20 </xDML>
```

6.5 A Criação de Componentes

Imaginação é o início da criação. Você imagina o que deseja, você deseja o que você imagina, e depois você cria o que você quer.

– GEORGE BERNARD SHAW

Para que as bibliotecas de classes desenvolvidas possam ser transformadas em componentes no sistema DISCOVER, é preciso acrescentar à essas bibliotecas de classes uma nova camada, de maneira que seja possível uma composição uniforme. Como descrito na Seção 5.1 na página 58, as bibliotecas de classes formam as interfaces verticais do sistema. Essa nova camada corresponde à interface horizontal, e como descrito na Seção 5.2 na página 60, no DISCOVER é implementado utilizando o *pattern* Facade.

Na Figura 6.5 é apresentado o diagrama de uma classe abstrata Componente, que deve ser herdada para implementar componentes no DISCOVER. Cada componente a ser adicionado ao sistema deve implementar uma classe filha dessa classe base. Essa classe contém uma função `Acerta_Parâmetros(parâmetro, valor)` que é chamada para acertar cada um dos parâmetros. A função `Aceita(visitante)` é necessária para aceitar o visitante do *pattern* Visitor. Ela é responsável por verificar se todas as pré-condições foram satisfeitas. Finalmente, a função `Executa` é utilizada para executar o componente. Ela é uma função abstrata (cada componente reimplementa a sua) que corresponde à operação chamada pelo visitante. Nessa função são encapsuladas todas as dependências entre o componente e outras classes, provendo uma única interface.

6.6 A Execução de um Experimento

A qualidade nunca é um acidente; ela sempre é resultado de uma elevada intenção, esforço sincero, direção inteligente e hábil execução; ela representa uma sábia escolha entre muitas alternativas.

Figura 6.5: Diagrama de Classe de um Componente

– WILLIAM A. FOSTER

Uma vez configurado um experimento utilizando-se a interface gráfica, esse experimento é traduzido para a linguagem xDML. A interface gráfica utiliza as informações sobre os componentes, também descritas em xDML, para fazer a checagem e validação do experimento. Dessa forma, uma configuração de experimento criada com o auxílio da interface gráfica e passada ao *framework* é uma configuração válida, que pode ser executada.

Para executar um experimento, o *framework* recebe essa configuração e utiliza um parser XML DOM para criar uma representação abstrata em árvore para essa configuração. Além disso, as informações sobre exceções e apontadores de objetos são armazenadas como contexto no interpretador do ambiente de execução, como mencionado na Seção 5.4 na página 66.

Nesta seção é exemplificado a execução da Listagem 6.6 na página 81, correspondente à Figura 6.4. Na Figura 6.6, é mostrada uma representação gráfica em árvore da Listagem 6.6. Nessa figura, componentes são representados como retângulos, conectores como elipses e ligações são representadas por setas. Por simplicidade, uma seta dupla indica uma ligação de entrada e saída entre um componente e um conector, enquanto ligações feitas por meio de conectores são representadas por um conector (elipse) entre componentes e as setas.

Como descrito na Seção 5.4 na página 66, o processamento é feito utilizando o *pattern* Interpreter. Esse *pattern* define um interpretador para uma dada linguagem. Além disso, o *pattern* Visitor é utilizado para visitar os nós dessa árvore.

O processamento começa pelo nó indicado com a seta que se origina em Estrutura. O fluxo de processamento segue as ligações entre os componentes e conectores definida na configuração de experimento, e representada em sua definição pelas ligações (e ligações para subestruturas). Quando um componente simples (Elemento Terminal) é encontrado, ele deve ser executado. As informações a respeito de exceções são armazenadas em Contexto.

A função `Aceita(Visitante)` de cada um dos componentes checa se todos os pré-requisitos (todos os parâmetros obrigatórios foram passados, todos os objetos necessários estão instanciados, todos os componentes que deviam se executados ante-

Figura 6.6: Representação Gráfica em Árvore da Listagem 6.6

riamente já foram executados) necessários à sua execução são cumpridos. A função `Aceita(Visitante)` também passa os parâmetros necessários, obtidos da sua descrição em xDML, ao componente.

Se o visitante é aceito (os pré-requisitos são cumpridos), ele “visita” o elemento, isto é, executa o componente. As exceções por ventura lançadas são analisadas pelo Contexto, e, se nenhum problema grave ocorrer após a execução do componente, ele é marcado como executado no Contexto.

6.7 Considerações Finais

A conclusão é o lugar onde você está cansado de pensar.

– MARTIN H. FISCHER

Neste capítulo foi mostrado um exemplo da utilização do *framework* de integração do DISCOVER proposto neste trabalho. Foi também descrita a utilização/interação da interface gráfica em desenvolvimento com o *framework* de integração, e vice-e-versa.

Também foi exemplificado a utilização da linguagem XDML para representar tanto um conjunto de componentes que faz parte do sistema DISCOVER, quanto uma configuração particular de experimento. Além disso, essa configuração de experimento foi exemplificada utilizando-se apenas componentes simples; componentes simples e conectores; e componentes simples, conectores e componentes compostos.

A execução de um experimento, utilizando o ambiente de execução proposto no *framework*, também foi descrita neste capítulo. No próximo capítulo são apresentadas as considerações finais desta dissertação.

Conclusões e Trabalhos Futuros

Mire para a lua. Se você errar, ainda pode acertar uma estrela.

– W. CLEMENT STONE

O projeto DISCOVER tem como principal objetivo reaproveitar as implementações decorrentes de projetos de pesquisa realizados no LABIC. Dentro desse projeto, surgiu a idéia de criar um sistema que englobe essas implementações e uma interface gráfica que facilite a sua utilização. Nesta dissertação foi proposto um *framework* de integração para esse sistema.

O restante deste capítulo está organizado da seguinte forma: na Seção 7.1 é apresentado um resumo dos capítulos e as principais contribuições deste trabalho ao projeto DISCOVER. Na Seção 7.2 são apresentados os trabalhos futuros decorrentes desta dissertação, principalmente quanto à implementação do *framework* e, finalmente, na Seção 7.3 são apresentadas as considerações finais desta dissertação.

7.1 Resumo dos Capítulos e das Principais Contribuições desta Dissertação

A sua única contribuição à soma de tudo é você mesmo.

– FRANK CRANE

No Capítulo 1 são brevemente apresentados o projeto DISCOVER e o sistema DISCOVER. Além disso, um breve relato das ferramentas implementadas, bem como referências para os projetos relacionados ao DISCOVER desenvolvidos em nosso laboratório nos últimos anos são apresentados.

No Capítulo 2 são apresentadas as funcionalidades principais de um sistema de KDD. Essas funcionalidades têm como principal objetivo formular um modelo teórico, ainda que de maneira simplificada, para um sistema desse tipo.

No Capítulo 3 é apresentada uma introdução à área de Arquitetura de *Software* como uma disciplina fundamental para o projeto e desenvolvimento de sistemas de software.

No Capítulo 4 são apresentados os conceitos de componentes, *software patterns* e *frameworks*. Esses conceitos representam, cada um à sua maneira, e em diferentes graus, um ferramental poderoso para o reúso de sistemas de *software*.

No Capítulo 5 é apresentado o *framework* de integração proposto neste trabalho. Esse *framework* compreende um mecanismo de adaptação de interface que cria uma camada (interface horizontal) sobre essas ferramentas, um poderoso mecanismo de metadados, implementado como uma linguagem baseada em XML, que é utilizado para descrever tanto os componentes que implementam as funcionalidades do sistema quanto as configurações de experimentos criadas pelos usuário, que serão executadas pelo *framework*, e um ambiente de execução para essas configurações de experimentos.

Finalmente, no Capítulo 6 é apresentado um exemplo da utilização do *framework* proposto. Nesse exemplo são descritas a interação entre o *framework* e a interface gráfica; a utilização da linguagem xDML como um elemento de ligação entre o *framework* e a interface interface gráfica; e o funcionamento do ambiente de execução.

7.2 Trabalhos Futuros

Apesar de terminando, e ainda começando.

– COWPER (1731–1800)

Este trabalho têm como principal objetivo propiciar um catalisador para o desenvolvimento do sistema DISCOVER. Entretanto, muito ainda precisa ser feito para que o sistema DISCOVER deixe de ser apenas uma proposta para se tornar um sistema efetivo.

O protótipo implementado neste trabalho contempla as implementações já realizadas, tanto de processamento e mineração de dados, e processamento de padrões voltados principalmente às tarefas de classificação. A camada de interface horizontal, bem como as informações a respeito dos metadados dessas implementações, precisam ser estendidas à projetos em desenvolvimento relacionados à outras atividades de descoberta de conhecimento, tais como regras de associação, regressão e agrupamento, para também contemplá-las.

Além disso, essas implementações precisam ser reunidas em um repositório comum, detalhadas, documentadas, mantidas e gerenciadas. Um processo para realizar essas atividades foi definido em Rozante (2003), com a participação ativa do autor deste trabalho no grupo de estudos responsável por essa definição. A definição desse processo foi feita por meio de reuniões desse grupo de estudo¹, que também contou

¹Foram realizadas aproximadamente 20 reuniões, com uma duração média de 3 horas cada uma

com a participação de outros dois pesquisadores do LABIC. Esse processo necessita agora ser implementado e incorporado à rotina de trabalho do nosso laboratório.

Além disso, sistemas de *software* precisam evoluir de acordo com às necessidades do usuário. Quando a implementação da interface gráfica estiver concluída, e for possível a realização de testes com a participação dos usuários do sistema, o protótipo implementado necessitará ser ajustado e modificado para um melhor desempenho.

Uma outra possibilidade de trabalhos futuros é a implementação das extensões e melhorias propostas na Seção 5.5. Essas implementações seriam extremamente úteis à pesquisa de Mineração de Dados distribuída e paralela, uma área de pesquisa extremamente ativa nos dias de hoje.

Finalmente, os padrões sugeridos neste trabalho deveriam viesar o desenvolvimento e implementação de novos componentes, de tal forma a maximizar as potencialidades e minimizar as restrições por eles impostos.

7.3 Considerações Finais

O que é importante é continuar aprendendo, apreciar o desafio e tolerar a ambiguidade. No final não há respostas exatas.

– MARTINA HORNER

A área de Descoberta de Conhecimento à partir de dados é relativamente nova e ferramentas que suportam todo o processo são raras. Além disso, quanto à disponibilidade e facilidade de utilização dessas ferramentas disponíveis para pesquisadores, essa falta de ferramentas se agrava ainda mais.

O sistema DISCOVER, nascido dentro do projeto DISCOVER, no Laboratório de Inteligência Computacional (LABIC), da USP-São Carlos, tem como objetivos ser uma ferramenta que integre as implementações e algoritmos de aprendizado mais utilizados pela comunidade de Aprendizado de Máquina, servindo como um campo de prova para pesquisas tanto em processamento e mineração de dados, e processamento de padrões, que fazem parte da descoberta de conhecimento.

O objetivo deste trabalho é criar um *framework* de integração de componentes para o sistema DISCOVER. Esse *framework* foi projetado utilizando *software patterns*. Como resultado da aplicação desses *patterns*, foi necessário a criação de uma linguagem baseada em XML para descrever tanto componentes que implementam funcionalidades para o sistema DISCOVER quanto configurações particulares de experimentos que utilizam essas funcionalidades.

Esperara-se que a efetiva implementação do framework proposto neste trabalho traga uma importante contribuição para que o sistema DISCOVER possa se tornar uma realidade.

Descrição em XML da xDML

Código A.1: Descrição do Esquema da Linguagem xDML

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v5 rel. 2 U (http://www.xmlspy.com) by labic (usp) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xlink="http://www.w3.org/1999/xlink"
  <!-- I'm a new comment node. -->
  <xs:import namespace="http://www.w3.org/1999/xlink" schemaLocation="http://www.w3.org/1999/xlink" />
  <xs:annotation>
    <xs:documentation xml:lang="en">Put here a description of this schema.</xs:documentation>
  </xs:annotation>
  <!-- I'm a new comment node. -->
  <xs:element name="xDiscover">
    <xs:complexType>
      <xs:sequence>
        <xs:any namespace="##any" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- I'm a new comment node. -->
  <xs:element name="structure" type="StructureInstance" />
  <!-- I'm a new comment node. -->
  <xs:simpleType name="Identifier">
    <xs:restriction base="xs:ID">
      <xs:minLength value="1" />
    </xs:restriction>
  </xs:simpleType>
  <!-- I'm a new comment node. -->
  <xs:complexType name="Description">
    <xs:simpleContent>
      <xs:extension base="xs:string" />
    </xs:simpleContent>
  </xs:complexType>
  <!-- I'm a new comment node. -->
  <xs:simpleType name="DirectionSimpleType">
```

```

<xs:restriction base="xs:string">
  <xs:enumeration value="none" />
  <xs:enumeration value="in" />
  <xs:enumeration value="out" />
  <xs:enumeration value="inout" />
</xs:restriction>
</xs:simpleType>
<!--I'm a new comment node.-->
<xs:complexType name="Direction">
  <xs:simpleContent>
    <xs:extension base="DirectionSimpleType" />
  </xs:simpleContent>
</xs:complexType>
<xs:simpleType name="ObjectSimpleType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Data_Object" />
    <xs:enumeration value="Dataset" />
    <xs:enumeration value="Ruleset" />
    <xs:enumeration value="C45DecisionTree" />
    <xs:enumeration value="Classification_Model" />
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="ObjectType">
  <xs:simpleContent>
    <xs:extension base="ObjectSimpleType" />
  </xs:simpleContent>
</xs:complexType>
<!--I'm a new comment node.-->
<xs:complexType name="XMLLink">
  <xs:attribute ref="xlink:type" />
  <xs:attribute ref="xlink:href" />
</xs:complexType>
<!--I'm a new comment node.-->
<xs:complexType name="InterfaceInstance">
  <xs:sequence>
    <xs:element name="description" type="Description" />
    <xs:element name="direction" type="Direction" />
    <xs:element name="type" type="ObjectType" />
  </xs:sequence>
  <xs:attribute name="id" type="Identifier" />
</xs:complexType>
<!--I'm a new comment node.-->
<xs:complexType name="Implementation">
  <xs:sequence>
    <xs:element name="description" type="Description" />
    <xs:element name="mainClass" type="ClassFile" />
    <xs:element name="auxClass" type="ClassFile" minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="parameters" type="Parameter" minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="exceptions" type="Exception" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>

```

```

    <xs:attribute name="id" type="Identifier"/>
  </xs:complexType>
  <!-- I'm a new comment node.-->
  <xs:complexType name="ClassFile">
    <xs:sequence>
      <xs:element name="classname" type="ClassName"/>
      <xs:element name="path" type="Path"/>
    </xs:sequence>
  </xs:complexType>
  <!-- I'm a new comment node.-->
  <xs:complexType name="ClassName">
    <xs:simpleContent>
      <xs:extension base="xs:string"/>
    </xs:simpleContent>
  </xs:complexType>
  <!-- I'm a new comment node.-->
  <xs:complexType name="Path">
    <xs:simpleContent>
      <xs:extension base="xs:string"/>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="Parameter">
    <xs:sequence>
      <xs:element name="parameterName" type="xs:string"/>
      <xs:element name="value" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Exception">
    <xs:sequence>
      <xs:element name="exceptionName" type="xs:string"/>
      <xs:element name="action" type="Action"/>
    </xs:sequence>
  </xs:complexType>
  <!-- I'm a new comment node.-->
  <xs:simpleType name="Action">
    <xs:restriction base="xs:string">
      <xs:enumeration value="raise"/>
      <xs:enumeration value="stop"/>
      <xs:enumeration value="ignore"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="ComponentInstance">
    <xs:sequence>
      <xs:element name="description" type="Description"/>
      <xs:element name="interface" type="InterfaceInstance" minOccurs="0" maxOccurs="1"/>
      <xs:choice>
        <xs:element name="implementation" type="Implementation"/>
        <xs:element name="subStructure" type="SubStructure"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>

```

```

    <xs:attribute name="id" type="Identifier"/>
</xs:complexType>
<!--I'm a new comment node.-->
<xs:complexType name="ConnectorInstance">
    <xs:sequence>
        <xs:element name="description" type="Description"/>
        <xs:element name="interface" type="InterfaceInstance" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="subStructure" type="SubStructure"/>
    </xs:sequence>
    <xs:attribute name="id" type="Identifier"/>
</xs:complexType>
<!--The Point type describes an endpoint of a link.
All links at the instance level exist between interfaces on exactly one interfaceInstance -->
<xs:complexType name="Point">
    <xs:sequence>
        <xs:element name="anchorOnInterface" type="XMLLink"/>
    </xs:sequence>
</xs:complexType>
<!--The LinkInstance type describes a link between two interface instances.
All links at the instance level exist between interfaces on components and connectors.
A linkInstance has an ID, a description, and exactly two endpoints.-->
<xs:complexType name="LinkInstance">
    <xs:sequence>
        <xs:element name="description" type="Description"/>
        <xs:element name="point" type="Point" minOccurs="2" maxOccurs="2"/>
    </xs:sequence>
    <xs:attribute name="id" type="Identifier"/>
</xs:complexType>
<!--The Group type describes a logical grouping of elements (components, connectors, and li
Groups may also be members of other groups.
-->
<xs:complexType name="Group">
    <xs:sequence>
        <xs:element name="description" type="Description"/>
        <xs:element name="member" type="XMLLink" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="Identifier"/>
</xs:complexType>
<!--I'm a new comment node.-->
<xs:complexType name="InterfaceInstanceMapping">
    <xs:sequence>
        <xs:element name="outerInterfaceInstance" type="XMLLink"/>
        <xs:element name="innerInterfaceInstance" type="XMLLink"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="SubStructure">
    <xs:sequence>
        <xs:element name="description" type="Description"/>
        <xs:element name="structureInstance" type="StructureInstance"/>
        <xs:element name="interfaceInstanceMapping" type="InterfaceInstanceMapping" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

```

```

    </xs:sequence>
</xs:complexType>
<xs:complexType name="StructureInstance">
  <xs:sequence>
    <xs:element name="description" type="Description"/>
    <xs:element name="component" type="ComponentInstance" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="connector" type="ConnectorInstance" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="link" type="LinkInstance" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="group" type="Group" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="id" type="Identifier"/>
</xs:complexType>
</xs:schema>

```


Notação

B.1 Diagramas Utilizados para Representar Esquemas XML

Figura B.1: Diagrama Choice: Este diagrama indica que apenas um dos elementos à direita deve ser instanciado

Figura B.2: Diagrama Sequence: Este diagrama indica que todos elementos (se necessário) à direita deve ser instanciado

Figura B.3: Diagrama Elements: Este diagrama indica a representação elementos. O primeiro é opcional, o segundo, indica que devem haver exatamente dois elementos, o terceiro, de um a cinco elementos e o ultimo representa um atributo

B.2 Diagrama de Interação

Figura B.4: Notação de Diagrama de Interação

B.3 Diagrama de Classe

Figura B.5: Notação de Diagrama de Classe

Referências Bibliográficas e Bibliografia

- Alexander, C., S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel (1977). *A Pattern Language*. New York: Oxford University Press. 48, 49
- Alexander, C. (1979). *The Timeless Way of Building*. New York: Oxford University Press. 48, 49, 50
- Appleton, B. (1997, May). Patterns and software: Essential concepts and terminology. *Object Magazine Online* 3(5). Also available at <http://www.enteract.com/~bradapp/docs/patterns-intro.html>. 51
- Bachman, F., L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau (2000, May). Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, SEI — Software Engineering Institute, Pittsburg, PA. <http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr008.pdf>. 42, 43, 45, 52
- Baranauskas, J. A. and G. E. A. P. A. Batista (2000). O projeto DISCOVER: Idéias iniciais (comunicação pessoal). 6
- Baranauskas, J. A. and M. C. Monard (2000a). An approach for extracting symbolic ensembles from databases. In L. E. Vaz (Ed.), *Proceedings 21st Iberian Latin American Congress on Computational Methods in Engineering, CILAMCE 2000*, Rio de Janeiro, Brazil. publicado em CD. 9
- Baranauskas, J. A. and M. C. Monard (2000b, July). A computational environment for extracting rules from databases. In N. Ebecken and C. A. Brebbia (Eds.), *Proceedings Second International Conference in Data Mining*, Cambridge, UK, pp. 321–330. WIT Press. 8, 9
- Baranauskas, J. A. and M. C. Monard (2003). Combining symbolic classifiers from multiple inducers. *Knowledge Based Systems* 16(issue 3), 129–136. Elsevier Science. 8, 9
- Baranauskas, J. A. (2001, agosto). *Extração Automática de Conhecimento por Múltiplos Intutores*. Tese de Doutorado, ICMC-USP. <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-08102001-112806/restrito/tese.pdf>. 8, 9, 14, 17
- Bass, L., P. Clements, and R. Kazman (1999, may). *Software Architecture in Practice*. Massachusetts: Addison Wesley. 28, 45
- Batista, G. E. A. P. A., A. C. P. L. F. Carvalho, and M. C. Monard (2000, April). Applying one-sided selection to unbalanced datasets. In O. Cairó, L. E. Sucar, and F. J. Cantu (Eds.), *Mexican International Congress on Artificial Intelligence, MICAI-2000*, Volume 1793 of *Lecture Notes in Artificial Intelligence*, Acapulco, Mexico, pp. 315–325. Springer-Verlag. 9

- Batista, G. E. A. P. A. and M. C. Monard (2001). Uma proposta para o tratamento de valores desconhecidos utilizando o algoritmo k-vizinhos mais próximos. In *Simpósio Brasileiro de Automação Inteligente – SBAI’2001*. publicado em CD. 9
- Batista, G. E. A. P. A. and M. C. Monard (2003a). An analysis of four missing data treatment methods for supervised learning. *Applied Artificial Intelligence* 17(5). in print. 9
- Batista, G. E. A. P. A. and M. C. Monard (2003b). Descrição da arquitetura e do projeto do ambiente computacional discover learning environment - dle. Technical Report 186, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/RT_186.pdf. 8
- Batista, G. E. A. P. A. (1997, outubro). Um ambiente para avaliação de algoritmos de aprendizado de máquina simbólico utilizando exemplos. Dissertação de Mestrado, ICMC-USP. <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-19082002-234842/>. 9
- Batista, G. E. A. P. A. (2001, maio). Sintaxe padrão do arquivo de exemplos do projeto DISCOVER. <http://www.icmc.sc.usp.br/~gbatista/Discover/SintaxePadraoFinal.htm>. 6, 8, 73, 74, 75
- Batista, G. E. A. P. A. (2003). *Pré-processamento de Dados em Aprendizado de Máquina Supervisionado*. Tese de Doutorado, ICMC-USP. a ser defendida. 9
- Bernardini, F. C. (2002, abril). Combinação de classificadores para melhorar o poder-preditivo e descritivo de *Ensembles*. Dissertação de Mestrado, ICMC-USP. Dissertação de Mestrado, ICMC-USP. 9
- Berson, A. and S. J. Smith (1997). *Data warehousing, Data Mining, and OLAP*. New York: McGraw-Hill. 13
- Botía, J. A., M. Garijo, J. R. Velasco, and A. G. Skarmeta (1998). A generic datamining system. Basic design and implementation guidelines. In *Workshop on Distributed Data Mining at the Fourth Intl. Conf. on Data Mining and Knowledge Discovery (KDD-98)*, New York, USA. AAAI Press. 24
- Brachman, R. J. and T. Anand (1996). *Advances in Knowledge Discovery and Data Mining*, Chapter The Process of Knowledge Discovery in Databases: A Human Centered Approach, pp. 37–58. In Fayyad et al. [Fayyad et al. \(1996\)](#). 16
- Brazdil, P. and C. Soares (2000). A comparison of ranking methods for classification algorithm selection. In R. L. de M’antaras and E. Plaza (Eds.), *Machine Learning: ECML 2000, 11th European Conference on Machine Learning*, Volume 1810, Barcelona, Spain, pp. 63–74. Springer-Verlag. 24
- Brodley, C. E. and P. Smyth (1995). The process of applying machine learning algorithms. In D. W. Aha and P. J. Riddle (Eds.), *Working Notes for Applying Machine Learning in Practice: A Workshop at the Twelfth International Machine Learning Conference*, Washington, DC. Available at <http://www.aic.nrl.navy.mil/~aha/imlc95-workshop/notes.html>. 16
- Buschmann, F., R. Meunier, H. Rohnert, P. Sommerland, and M. Stal (1996). *Pattern-Oriented Software Architecture – A System of Patterns*. Chichester, England: Wiley. 51, 69
- Chan, P. K.-W. (1996). *An Extensible Meta-Learning Approach for Scalable and Accurate Inductive Learning*. Ph. D. thesis, Department of Computer Science, Columbia University, New York, NY. 24

- Chan, P. K. and S. J. Stolfo (1993). Experiments in multistrategy learning by meta-learning. In B. Bhargava (Ed.), *Proceedings of the second international conference on information and knowledge management*, Washington, DC, pp. 314–323. 24
- Chapman, P., J. Clinton, R. Kerber, T. Khabaza, T. Reinartz, C. Shearer, and R. Wirth (2000, august). *CRISP-DM 1.0*. Cross Industry Standard Process for Data Mining. <http://www.crisp-dm.org/CRISPWP-0800.pdf>. 18
- Cohen, P. R. (1995). *Empirical methods for artificial intelligence*. Cambridge, Mass.: MIT Press. 16
- Coplien, J. O. (1995). *Pattern Languages of Program Design*, Volume 1, Chapter A Generative Development Process Pattern Language, pp. 183–237. Addison Wesley. 51
- Coplien, J. O. (1996). *Software Patterns*. SIGS Books. 50
- Dashofy, E. M. and A. van der Hoek (2001). Representing product family architectures in an extensible architecture description language. In F. van der Linden (Ed.), *Proceedings of The International Workshop on Product Family Engineering (PFE-4)*, Volume 2290 of *Lecture Notes in Computer Science*, Bilbao, Spain, pp. 330–341. Springer-Verlag. 63
- Deutsch, L. P. (1987). Levels of reuse in the Smalltalk-80 programming. In P. Freeman (Ed.), *Tutorial: Software Reusability*. IEEE Computer Society Press. 47
- Deutsch, L. P. (1989). Design reuse and frameworks in the Smalltalk-80 programming system. In T. J. Biggerstaff and A. J. Perlis (Eds.), *Software Reusability*, Volume II, pp. 55–71. ACM Press. 47
- Dosualdo, D. G. (2002). Investigação de regressão para data mining. Dissertação de Mestrado, ICMC-USP. Monografia para o Exame de Qualificação. 7
- D'Souza, D. F. and A. C. Wills (1999). *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Boston, MA: Addison-Wesley. 45
- Fallside, D. C. (2001). Xml schema part 0: Primer. Technical report, W3C World Wide Web Consortium. <http://www.w3.org/TR/xmlschema-0/>. 63
- Faulkner, C. (1998). *The essence of human-computer interaction*. London: Prentice Hall. 21
- Fayad, M. and D. C. Schmidt (1997, October). Object-oriented application frameworks. *Communications of the ACM* 40(10), 32–38. Guest editorial. 47
- Fayyad, U. M., G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy (Eds.) (1996). *Advances in Knowledge Discovery and Data Mining*. Menlo Park, CA: American Association for Artificial Intelligence (AAAI Press). 100, 101
- Fayyad, U. M., G. Platestsky-Shapiro, and P. Smyth (1996). *Advances in Knowledge Discovery and Data Mining*, Chapter From Data Mining to Knowledge Discovery: An Overview, pp. 1–30. In Fayyad et al. [Fayyad et al. \(1996\)](#). 4, 14, 23
- Flach, P. A. (2001). Esprit metal project: A meta-learning assistant for providing user support in machine learning and data mining. <http://www.cs.bris.ac.uk/Research/MachineLearning/METAL/project-summary.html>. Executive Summary. 24, 25
- Fowler, M. (1997). *Analysis Patterns: Reusable Object Models*. Addison-Wesley. 51

- Freitas, A. A. and S. H. Lavington (1998). *Mining Very Large Databases with Parallel Processing*. Kluwer Academic Publishers. 24, 70
- Freitas, A. A. (1998a, September). A multi-criteria approach for the evaluation of rule interestingness. In *Proceedings of the International Conference on Data Mining*, Rio de Janeiro, RJ, pp. 7–20. 23
- Freitas, A. A. (1998b, September). On objective measures of rule surprisingness. In J. M. Żytkow and M. Quafafou (Eds.), *Principles of Data Mining & Knowledge Discovery: Proceedings of the Second European Symposium (PAKDD'98)*, Volume 1510 of *Lecture Notes in Artificial Intelligence*, Nantes, France, pp. 1–9. Springer-Verlag. 23
- Freitas, A. A. (1999, October). On rule interestingness measures. *Knowledge-Based Systems* 12(5–6), 309–315. 23
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. 48, 49, 51, 60, 66
- Geromini, M. R. (2002). Projeto e desenvolvimento da interface gráfica do sistema DISCOVER. Dissertação de Mestrado, ICMC-USP. Monografia para o Exame de Qualificação. 57, 59, 71, 76
- Gomes, A. K. (2002). Análise do conhecimento extraído de classificadores simbólicos utilizando medidas de avaliação e interessabilidade. Dissertação de Mestrado, ICMC-USP. <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-04072002-144610/>. 9
- Griss, M. L. (1998, November). Models and patterns of success. *Component Strategies Magazine*. Also available at <http://www.hpl.hp.com/reuse/papers/obml2.htm>. 53
- IBM (2000). *DB2 Intelligent Miner Brochure*. IBM. <http://www-4.ibm.com/software/data/iminer/fordata/>. 5
- Imamura, C. Y. (2001). Pré-processamento para extração de conhecimento de bases textuais. Dissertação de Mestrado, ICMC-USP. Dissertação de Mestrado. 9
- International Organization for Standard ISO (1986). *Information processing – Text and office systems – Standard Generalized Markup Language (SGML)*. International Organization for Standard ISO. <http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=16387>. 62
- ISO and IEC (1991). *Information Technology — Software Product Evaluation — Quality Characteristics and Guidelines for their Use*. ISO and IEC. ISO/IEC 9216. 33
- Jacobson, I., M. Griss, and P. Jonsson (1997). *Software reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley. 52
- Johnson, R. E. and B. Foote (1988). Designing reusable classes. *Journal of Object-Oriented Programming* 1(2), 22–35. 47
- Johnson, R. E. (1997). Components, frameworks, patterns. In *ACM SIGSOFT Symposium on Software Reusability*, pp. 10–17. 46
- Kemp, A. H., G. E. A. P. A. Batista, and M. C. Monard (2001, maio). Descrição da implementação dos métodos estatísticos de resampling do ambiente DISCOVER. Technical Report 143, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/Rt_143.ps.zip. 6, 8

- Klemettinen, M., H. Mannila, and H. Toivonen (1996). Interactive exploration of discovered knowledge: A methodology for interaction, and usability studies. Technical Report C-1996-3, University of Helsinki, Finland. citeseer.nj.nec.com/klemettinen96interactive.html. 16
- Klemettinen, M., H. Mannila, and H. Toivonen (1997). A data mining methodology and its application to semi-automatic knowledge acquisition. In *8th International Conference and Workshop on Database and Expert Systems Applications (DEXA'97 Workshop)*, Toulouse, France, pp. 670–677. IEEE Computer Society Press. 17
- Kohavi, R., G. John, R. Long, D. Manley, and K. Pfleger (1994). MLC++: A machine learning library in C++. In *6th International Conference on Tools with Artificial Intelligence*, pp. 740–743. IEEE Computer Society Press. 5
- Kohavi, R., D. Sommerfield, and J. Dougherty (1997). Data mining using MLC++: A machine learning library in C++. *International Journal on Artificial Intelligence Tools* 6(4), 537–566. 5, 17
- Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design* (2nd edition ed.). Prentice-Hall. 51
- Lavrač, N., P. A. Flach, and B. Zupan (1999, June). Rule evaluation measures: A unifying view. In S. Džeroski and P. A. Flach (Eds.), *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, Volume 1634 of *Lecture Notes in Artificial Intelligence*, Bled, Slovenia, pp. 174–185. Springer-Verlag. 8
- Lee, H. D. (2000). Seleção de features relevantes para aprendizado de máquina. Dissertação de mestrado, ICMC-USP. <http://www.teses.usp.br/disponiveis/55/55134/tde-15032002-113112/>. 9
- Mannila, H. (1996, June). Data mining: Machine learning, statistics, and databases. In J. C. F. Per Svensson (Ed.), *Eight International Conference on Scientific and Statistical Database Management*, Stockholm, Sweden, pp. 2–11. IEEE Computer Society Press. 15
- Mannila, H. (1997, January). Methods and problems in data mining. In F. N. Afrati and P. G. Kolaitis (Eds.), *Proceedings of International Conference on Database Theory*, Volume 1186 of *Lecture Notes in Computer Science*, Delphi, Greece, pp. 41–55. Springer-Verlag. 16
- Martins, C. A., M. C. Monard, and G. Halembeck (2002, June). Combining clustering and inductive learning to find and interpret patterns from dataset. In *Proceedings ACIS International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA-02)*, Foz do Iguaçu, Brazil, pp. 90–95. 9
- Martins, C. A. (2001). *Interpretação de Clusters em Aprendizado de Máquina*. Tese de Doutorado, ICMC-USP. Monografia para o Exame de Qualificação. 7, 9
- Massingill, B. L., T. G. Mattson, and B. A. Sanders (1999b). Patterns for parallel application programs. In D. Manolescu and B. Woolf (Eds.), *Proceedings of Pattern Languages of Programs'99 (PLoP'99)*. available at <http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/>. 69
- Massingill, B. L., T. G. Mattson, and B. A. Sanders (1999a). A pattern language for parallel application programming. Technical Report 99-022, Computer & Information Science & Engineering (Unv. Florida). 69

- Matheus, C. J., P. K. Chan, and G. Piatetsky-Shapiro (1993). Systems for knowledge discovery in databases. *IEEE Transactions On Knowledge And Data Engineering* 5, 903–913. 15, 19, 20, 23
- Melanda, E. (2002). *Pós-processamento de conhecimento de regras de associação*. Tese de Doutorado, ICMC-USP. Monografia para o Exame de Qualificação. 7, 9
- Meszaros, G. and J. Doble (2001). A pattern language for pattern writing. <http://hillside.net/patterns/writing/patternwritingpaper.htm>. 51
- Michalski, R. S. and R. E. Stepp (1983). Chapter Learning from Observation: Conceptual Clustering, pp. 331–363. Palo Alto, CA.: TIOGA Publishing Co. 3
- Microsoft Corporation (1998, August). *Microsoft Component Services, Server Operating System, A Technology Overview*. Microsoft Corporation. <http://www.microsoft.com/com/wpaper/compsvcs.asp>. 48, 52
- Milaré, C. R., A. C. P. L. F. Carvalho, and M. C. Monard (2003). An approach to explain neural networks using symbolic algorithms. *International Journal of Computational Intelligence and Applications*. (in print). 9
- Milaré, C. R. (2000, Março). *Extração de Conhecimento de Redes Neurais*. Tese de Doutorado, ICMC-USP. Monografia para o Exame de Qualificação. 9
- Minsky, M. S. and S. Papert (1969). *Perceptrons: An Introduction to Computational Geometry*. Cambridge, Mass: MIT Press. 3
- Monard, M. C. and J. A. Baranauskas (2003a). *Sistemas Inteligentes: fundamentos e aplicações*, Chapter Conceitos sobre Aprendizado de Máquina, pp. 89–114. In Rezende (2003). 15
- Monard, M. C. and J. A. Baranauskas (2003b). *Sistemas Inteligentes: fundamentos e aplicações*, Chapter Indução de Regras e Árvores de Decisão, pp. 115–140. In Rezende (2003). 15
- Monard, M. C. and G. E. A. P. A. Batista (2002). Learning with skewed class distribution. In J. M. Abe and J. I. da Silva Filho (Eds.), *Advances in Logic, Artificial Intelligence and Robotics*, Volume 85 of *Frontiers in Artificial Intelligence and Applications*, São Paulo, pp. 173–180. IOS Press. 9
- Mowbray, T. J. and R. C. Malveau (1997). *Corba Design Patterns*. Chichester, England: Wiley. 51, 52, 59
- Newell, A. and H. A. Simon (1956). The logic theory machine. *IRE Transactions on Information Theory* 2(3), 61–79. 3
- Object Management Group (2001, June). *CORBA overview*. Object Management Group. <http://www.omg.org/cgi-bin/doc?formal/01-02-06>. 48, 52
- Object Mangement Group (2000, October). *OMG IDL Syntax and Semantics*. Object Mangement Group. Common Object Request Broker Architecture (CORBA) v2.4. 44
- Paula, M. F. (2003). Ambiente para disponibilização de conhecimento. Dissertação de Mestrado, ICMC-USP. A ser defendida. 9
- Piatetsky-Shapiro, G. (1991). Knowledge discovery in real databases: A report on the UJCAI-89 workshop. *Artificial Intelligence Magazine* 11(5), 68–70. 4

- Pila, A. D. and M. C. Monard (2001, Outubro). An empirical comparison of rough sets reducts and other filter approaches for feature subset selection. In *Anais do VI Simpósio Iberoamericano de Reconhecimento de Padrões - SIARP2001*, Florianópolis, Brazil, pp. 41–49. 9
- Pila, A. D. and M. C. Monard (2002, September). Rules induced by symbolic learning algorithms using rough sets reducts for selecting features: An empirical comparison with other filters. In A. Zapico and J. M. Santos (Eds.), *Proceedings Argentine Symposium on Artificial Intelligence, ASAI'2002*, Santa Fe, Argentina, pp. 206–217. 9
- Pila, A. D. (2001, abril). Seleção de atributos relevantes para aprendizado de máquina utilizando a abordagem de Rough Sets. Dissertação de Mestrado, ICMC-USP. http://www.teses.usp.br/teses/disponiveis/55/55134/tde-13022002-153921/publico/dissertacao_ADp.pdf. 9
- Prati, R. C., J. A. Baranauskas, and M. C. Monard (1999, outubro). BIBVIEW: Um sistema para auxiliar a manutenção de registros para o BibTeX. Technical Report 95, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/Rt_95.ps.zip. i
- Prati, R. C., J. A. Baranauskas, and M. C. Monard (2001a, junho). Extração de informações padronizadas para a avaliação de regras induzidas por algoritmos de aprendizado de máquina simbólicos. Technical Report 145, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/RT_145.ps.zip. 6, 8, 73
- Prati, R. C., J. A. Baranauskas, and M. C. Monard (2001b, março). Uma proposta de unificação da linguagem de representação de conceitos de algoritmos de aprendizado de máquina simbólicos. Technical Report 137, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/RT_137.ps.zip. 6, 8, 73, 74
- Prati, R. C., J. A. Baranauskas, and M. C. Monard (2002, setembro). Padronização da sintaxe e informações sobre regras induzidas a partir de algoritmos de aprendizado de máquina simbólico. *Revista Eletrônica de Iniciação Científica* 2(3). <http://www.sbc.org.br/reic/edicoes/2002e3>. 8, 73
- Prati, R. C., M. R. Geromini, and M. C. Monard (2003). An integrated environment for data mining. Submitted to IJCAI-03. 6
- Prati, R. C. and M. C. Monard (2002). Projeto e implementação do *Framework* de integração do sistema DISCOVER. In R. S. M. da Graça Pimentel (Ed.), *VII Workshop de Teses e Dissertações em Andamento*. Publicado em CD-ROM. 6
- Prati, R. C. and M. C. Monard (2003). Projeto e implementação do *Framework* de integração do sistema DISCOVER. Technical report, ICMC-USP. em preparação. 58
- Pree, W. (1994). *Design Patterns for Object-Oriented Software Development*. Reading, Mass.: Addison-Wesley. 47, 52
- Pugliesi, J. B. (2001, março). *O Pós-Processamento em Extração de Conhecimento de Bases de Dados*. Tese de Doutorado, ICMC-USP. Monografia para o Exame de Qualificação. 7, 9
- Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning* (1), 81–106. 3
- Quinlan, J. R. (1993). *C4.5 Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann. 73

- Rezende, S. O., J. B. Pugliesi, E. A. Melanda, and M. F. Paula (2003). *Sistemas Inteligentes: fundamentos e aplicações*, Chapter Mineração de Dados, pp. 307–336. In [Rezende \(2003\)](#). 15
- Rezende, S. O. (2003). *Sistemas Inteligentes: fundamentos e aplicações*. Barueri, SP, Brasil: Editora Manole. 104, 106
- Rosenblatt, F. (1958). Perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review* (65), 368–408. 3
- Rozante, T. A. A. (2003). Implantação do reuso de componentes no processo de desenvolvimento de software. Dissertação de Mestrado, ICMC-USP. A ser defendida. 88
- Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). Learning representations by back-propagating errors. *Nature* (323), 533–536. 3
- Salviano, C. F. (1997, outubro). Introdução à software patterns. Notas para um tutorial apresentado no XI SBES. 49
- Shaw, M. and P. Clements (1997, August). A field guide to boxology: Preliminary classification of architectural styles for software systems. In A. L. Wolf (Ed.), *Second International Software Architecture Workshop (ISAW-2)*, pp. 6–13. 36
- Shaw, M. and D. Garlan (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice Hall. 33
- Shaw, M. (1990). Prospects for an engineering discipline of software. *IEEE Software* 7(6), 15–24. 36
- Shaw, M. (1996). Some patterns for software architectures. In J. M. Vlissides, J. O. Coplien, and N. L. Kerth (Eds.), *Proceedings of the Second Pattern Languages of Program Design Workshop*, Volume 2, Reading, MA, pp. 255–269. 39
- Silicon Graphics (2000). *MineSet™3.0 Enterprise Edition*. Silicon Graphics. http://www.sgi.com/software/mineset/mineset_data.html. 5
- Stolfo, S. J., A. L. Prodromidis, S. Tselepis, W. Lee, D. W. Fan, and P. K. Chan (1997, August). JAM: Java agents for meta-learning over distributed databases. In D. Heckerman, H. Mannila, D. Pregibon, and R. Uthurusamy (Eds.), *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, Newport Beach, CA, pp. 74–81. AAAI Press. 25
- SUN Microsystems (1996, November). *Introducing Java Beans*. SUN Microsystems. <http://developer.java.sun.com/developer/onlineTraining/Beans/Beans1/index.html>. 52
- Szyperski, C. (1998). *Component Software Beyond Object-Oriented Programming*. Boston, MA: Addison-Wesley and ACM Press. 43
- Wall, L., T. Christiansen, and J. Orwant (2000). *Programming Perl* (3 ed.). O'Reilly. 6, 8
- Wang, G. and H. A. MacLean (1998). Architectural components and object-oriented implementations. In *International Workshop on Component-Based Software Engineering*. available at <http://www.sei.cmu.edu/cbs/icse98/papers/p9.html>. 43

- Weck, W. (1997, June). Independently extensible component frameworks. In M. Muehlhaeuser (Ed.), *Proceedings of the 1st International Workshop on Component-Oriented Programming*, Heidelberg, pp. 177–183. dpunkt Verlag. In Special Issues in Object-Oriented Programming –Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP’96. 45
- Weiss, S. M. and N. Indurkha (1998). *Predictive Data Mining: A Practical Guide*. San Francisco, CA: Morgan Kaufmann. 14, 22
- Wirfs-Brock, R. J. and R. E. Johnson (1990). Surveying current research in object-oriented design. *Communications of the ACM* 33(9), 104–124. 47
- Witten, I. H. and E. Frank (1999, october). *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Volume 1. Morgan Kaufmann. <http://www.cs.waikato.ac.nz/ml/weka/index.html>. 5
- Wrobel, S., D. Wettschereck, E. Sommer, and W. Emde (1996). Extensibility in data mining systems. In E. Simoudis, J. W. Han, and U. Fayyad (Eds.), *Proceedings 2nd International Conference On Knowledge Discovery and Data Mining*, Portland, OR, USA, pp. 214–219. AAAI Press. 24
- Zaki, M. J. and C.-T. Ho (Eds.) (2000). *Large-Scale Parallel Data Mining*, Volume 1759 of *Lecture Notes in Computer Science*. Heidelberg, Germany: Springer. 70