

**LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO DO RIO GRANDE DO
NORTE**

CAMILO DE LÉLIS DIAS COSTA DE MEDEIROS

**DESENVOLVIMENTO DE SISTEMA DE APOIO À CONSULTORIA EM
OVINOCULTURA E CAPRINOCULTURA UTILIZANDO
JRUBY ON RAILS**

**DESENVOLVIMENTO DE SISTEMA DE APOIO À CONSULTORIA EM
OVINOCULTURA E CAPRINOCULTURA UTILIZANDO
JRUBY ON RAILS**

apresentado ao Curso de
Especialização em Desenvolvimento de
Sistemas Corporativos da Faculdade Natalense
para o Desenvolvimento do Rio Grande do
Norte, como parte das atividades necessárias
para a obtenção do título de Especialista em
Desenvolvimento de Sistemas Corporativos.

Orientador: David Ricardo do Vale Pereira, M.
Sc.

Natal

2008

FARN

CAMILO DE LÉLIS DIAS COSTA DE MEDEIROS

DESENVOLVIMENTO DE SISTEMA DE APOIO À CONSULTORIA EM
OVINOCULTURA E CAPRINOCULTURA UTILIZANDO
JRUBY ON RAILS

**DESENVOLVIMENTO DE SISTEMA DE APOIO À CONSULTORIA EM
OVINOCULTURA E CAPRINOCULTURA UTILIZANDO
JRUBY ON RAILS**

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Sistemas Corporativos da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte, como parte dos requisitos necessários para a obtenção do título de Especialista em Desenvolvimento de Sistemas Corporativos.

Orientador: David Ricardo do Vale Pereira, M. Sc.

**Natal
2008**

00143
FARN
BIBLIOTECA
16.10.08
10.09.08

Catálogo na Publicação – Biblioteca da FARN
Setor de Processos Técnicos

Medeiros, Camilo de Lélis Dias Costa de

Desenvolvimento de sistema de apoio à consultoria em ovinocultura e caprinocultura utilizando Jruby on Rails / Camilo de Lélis Dias Costa de Medeiros. – Natal: FARN, 2007.

58 f.

Orientador: David Ricardo do Vale Pereira (M.Sc.)

Monografia (Especialização em Desenvolvimento de Sistemas Corporativos) – Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte.

1. Linguagem de Programação – Monografia. 2. Plataforma Java – Monografia. 3. Sistema Web Orientados a Objetos – Monografia. 4. Desenvolvimento de Sistema – Monografia. I. Título.

RN/FARN/BC

CDU 004.43

RB- 30.708

FARN
BIBLIOTECA
Nº Reg. M000899
16 / 04 / 08

00443
M438d
2008
px.01

CAMILO DE LÉLIS DIAS COSTA DE MEDEIROS

Dedico este trabalho as pessoas que realmente fizeram parte dele:

A Deus, Pai de infinita bondade, que me abençoe todos os dias para que a benção diária seja ganha.

**DESENVOLVIMENTO DE SISTEMA DE APOIO À CONSULTORIA EM
OVINOCULTURA E CAPRINOCULTURA UTILIZANDO
JRUBY ON RAILS**

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Sistemas Corporativos da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte, aprovado em _____, pela banca examinadora composta pelos seguintes membros.

Orientador: David Ricardo do Vale Pereira, M. Sc.

Dias, Tios, Primos, Familiares, todos os que se preocupam por mim.

A Nizéida, Clóves e Cláudia, sogros e cunhada, a todo momento me dão incentivo.

Professora Val _____ David Ricardo do Vale Pereira

David Ricardo do Vale Pereira, M. Sc.
Orientador

Dedico Dantas, Marcos Alexandre e Edipo Elder, que juntamente com Marcelo Medeiros, fizeram um sonho de mostrar as necessidades dos criadores de bodes e carneiros.

_____ Gleydson de Azevedo F. Lima

Gleydson Azevedo Ferreira Lima, M. Sc.
Membro examinador

Ass. Amigos Gleydson de Azevedo F. Lima, Wandell, Juliano Sena.

Meus Filhos, amigos da SINFO que me ajudaram a escrever, ensinaram e ajudaram para o meu desenvolvimento nesta jornada.

Às Amigas, Americano, Profissionais, Casa Familiar, David Pereira. Sempre disposto a ajudar, não importava o dia ou a hora. Com direito a puxões de orelhas e tudo mais. O meu maior e sincero agradecimento.

AGRADECIMENTOS

Dedico este trabalho as pessoas que realmente fizeram parte dele:

A Deus, Pai de Infinita bondade, que me abençoa todos os dias para que a batalha diária seja ganha.

Ana Karina, esposa, amiga que me acompanhou em todos os momentos incentivando, apoiando e sendo sempre compreensiva.

Maria Zélia Dias da Costa, uma Mãe de verdade, uma Guerreira, Águia, Amiga certa das horas incertas.

Maria Dias da Costa, Vovó, com suas orações, em todas as horas, para que eu chegasse ao fim. Te Amo!!!

Tias, Tios, Primos, Familiares, todos os que torceram por mim.

A Nizeuda, Cloves e Cinthia, sogros e cunhada, a todo momento um novo incentivo.

Professora Valéria Cruz, Amiga, criadora de rebanho caprino e a grande idealizadora dos projetos tecnológicos para o sertão do cabugi.

Dalton Dantas, Marcos Alexandre e Édipo Elder, que juntamente comigo e Marcelo Mariano, tivemos um sonho de atender as necessidades dos criadores de bodes e carneiros.

Aos Amigos Gleydson Lima, Raphaela Galhardo, Ricardo Wendell, Juliano Sena, Itamir Filho, amigos da SINFO que sempre incentivaram, ensinaram e ajudaram para o meu crescimento nesta jornada.

Ao Amigo, Americano, Professor, Cara Fantástico, David Pereira. Sempre disposto a ensinar, não importava o dia ou a hora. Com direito a puxões de orelhas e tudo mais. O meu mais sincero agradecimento.

RESUMO

Este trabalho tem por finalidade realizar um estudo detalhado sobre linguagens dinâmicas, especificamente linguagens dinâmicas que possam ser executadas na plataforma Java. A linguagem escolhida foi JRuby e o framework foi o Ruby on Rails. O estudo de caso é baseado em um levantamento feito junto a criadores de rebanho ovino e caprino, onde foram relatadas dificuldades para que se tenha um maior controle dos animais e dos produtos que eles produzem como leite, lã e carne. Diante das informações que tivemos, surge a proposta de desenvolvimento de um produto usando a tecnologia para que, tanto os criadores possam ter este controle simplificado, quanto para consultores da área, que tem interesse em trabalhar de perto com os fazendeiros. Descrevem-se neste texto conceitos de desenvolvimento de sistemas Web orientados a objetos, onde incluem-se uma introdução aos sistemas Web, UML (Unified Modeling Language), orientado a objetos e desenvolvimento em camadas. Será abordado também a plataforma Java, Ruby, Ruby on Rails, o estudo de caso e o modelo do sistema proposto diante das informações levantadas.

Palavras chave: Ruby, Ruby on Rails, ovinocultura, caprinocultura, controle animal, desenvolvimento web

Ao meu Amigo Marcelo Mariano. Apesar do pouco tempo que passou conosco, muito ensinou a todos que o cercavam.

RESUMO

Este trabalho tem por finalidade realizar um estudo detalhado sobre linguagens dinâmicas, especificamente linguagens dinâmicas que possam ser executadas na plataforma Java. A linguagem escolhida foi JRuby e o framework foi o Ruby on Rails. O estudo de caso é baseado em um levantamento feito junto a criadores de rebanho ovino e caprino, onde foram relatadas dificuldades para que se tenha um maior controle dos animais e dos produtos que eles produzem como: leite, lã e carne. Diante das informações que tivemos, surge a proposta do desenvolvimento de um produto usando a tecnologia para que, tanto os criadores possam ter este controle almejado, quando para consultores da área, que tem interesse em trabalho de parceria com os fazendeiros. Descrevem-se neste texto conceitos de desenvolvimento de sistemas Web orientados a objetos, onde incluímos uma introdução aos sistemas Web, UML (Unified Modelling Language), orientação a objetos e desenvolvimento em camadas. Será abordado também a plataforma Java, Ruby, Ruby on Rails, o estudo de caso e o modelo do sistema proposto diante das informações levantadas.

Palavras chave: Ruby, Ruby on Rails, ovinocultura, caprinocultura, controle animal, desenvolvimento web

LISTA ABSTRACT

This work has the purpose of making a study of dynamic languages, more specifically, sinamic languages that can be executed in the Java Platform. The chosen language was JRuby and the framework was Ruby on Rails. The case study is based on a research made with breeders of goats and sheeps, who reported difficulties in having control of the animals and de products that they produce, like milk, wool and meat. Considering the gathered information, it was proposed the development of a product using the mentioned technology, so that it can be used by the breeders, to have the desired control, and the consultants in the area, who have interests in working with the farmers. In this text, the concepts of object oriented web development are described, with an introduction to web systems, UML (Unified Modelling Language), object orientation e layer development. Also, the Java platform, the Ruby language and the Ruby on Rails framework will be approached.

Key words: Ruby, Ruby on Rails, sheep, goat, animal control, web development

Figura 10 – Diagrama de Caso de Uso do Controle de Partidos..... 49

LISTA DE ILUSTRAÇÕES

Figura 1 - Desenvolvidos em Camadas.....	17
Figura 2 – Desenho do Modelo MVC.....	19
Figura 3 – Diagrama de classe do Controle Animal.....	42
Figura 4– Diagrama de Caso de Uso do Controle Animal.....	43
Figura 5 – Diagrama de Classe do Controle da Propriedade.....	44
Figura 6 – Diagrama de Caso de Uso do Controle da Propriedade.....	45
Figura 7 – Diagrama de Classe do Manejo Alimentar.....	46
Figura 8 – Diagrama de Caso de Uso do Manejo Alimentar.....	47
Figura 9 – Diagrama de Classe do Controle de Parição.....	48
Figura 10 – Diagrama de Caso de Uso do Controle de Parição.....	49
Código 11 – Exemplo de Duck Typing.....	29
Código 12 – Exemplo de Closures.....	30
Código 13 – Referência de variáveis.....	30
Código 14 – Exemplo de Closure.....	31
Código 15 – Função associada a variável.....	31
Código 16 – Variável salaries_alter.....	31
Código 17 – Variável highPaid fazendo teste.....	32
Código 18 – Método para escrever um número em português.....	33
Código 19 – Construção do módulo container.....	33
Código 20 – Definição da classe BigInteger com mixin.....	34
Código 21 – Criação do objeto BigInteger e chamada de método herdado.....	34
Código 22 – Exemplo de mixin através do extend.....	34
Código 24 – Código Mixin.....	35
Código 25 – Adicionando método em tempo de execução.....	35
Código 26 – Exemplo de prompt com o puts para testes.....	36
Código 27 – Forma de executar o comando puts.....	36
Código 28 – Exemplo de utilização da API Java em um programa Ruby.....	37
Código 29 – Exemplo de classe Propriedade.....	51
Código 30 – Exemplo de classe Funcionario.....	51
Código 31 – Exemplo de objeto responsável por sua persistência.....	52

LISTA DE CÓDIGOS

Código 1 – Criação de uma classe Ruby.....	24
Código 2 – Classe mostrando utilização de variáveis e chamada de métodos.....	25
Código 3 – Demonstração de criação de Arrays multi-dimensionais.....	26
Código 4 – Demonstração aninhamento de Array.....	26
Código 5 – Demonstração de indexação de Arrays.....	26
Código 6 – Criação de uma Hash.....	27
Código 7 – Demonstração de colocação de símbolos como objetos.....	27
Código 8 – Exemplo de Duck Typing.....	28
Código 9 – Classe Pato.....	29
Código 10 – Classe Ganso.....	29
Código 11 – Exemplo de Duck Typing.....	29
Código 12 – Exemplo de Closures.....	30
Código 13 – Referência de variáveis.....	30
Código 14 – Exemplo de Closure.....	31
Código 15 – Função associada a variável.....	31
Código 16 – Variável salarios_altos.....	31
Código 17 – Variável highPaid fazendo teste.....	32
Código 18 – Método para escrever um número em português.....	33
Código 19 – Construção do módulo container.....	33
Código 20 – Definição da classe BigInteger com mixin.....	34
Código 21 – Criação de objeto BigInteger e chamada de método herdado.....	34
Código 22 – Exemplo de mixin através do extend.....	34
Código 24 – Código Mixin.....	35
Código 25 – Adicionando método em tempo de execução.....	35
Código 26 – Exemplo de prompt com o puts para testes.....	36
Código 27 – Forma de executar o comando puts.....	36
Código 28 – Exemplo de utilização da API Java em um programa Ruby.....	37
Código 29 – Exemplo de classe Propriedade.....	51
Código 30 – Exemplo de classe Funcionário.....	51
Código 31 – Exemplo de objeto responsável por sua persistência.....	52

Código 32 – Migration para a tabela funcionários.....	52
Código 33 – Exemplo de criação de um controller no Scaffolding.....	53
Código 34 – Exemplo de instalação de plugin no Rails.....	54
Código 35 – Execução da ação discovery.....	54
Tabela 36 - Descrição das responsabilidades de cada camada no modelo de três camadas.....	18
Tabela 37- Efetivo de caprinos no nordeste.....	40

BROWSER - Navegador

BYTECODE - Resultado da compilação de um programa Java

CRUD - Create, Retrieve, Update e Delete

DESKTOP - Microcomputador utilizado para trabalhar com sistemas

DL - Dynamic Link Library

DRY - Don't Repeat Yourself

HTML - HyperText Markup Language

HTTP - HyperText Transfer Protocol

JAR - Java Archive

JDK - Java Development Kit

JIT - Just-In-Time

JSP - Java Server Pages

VM - Java Virtual Machine

MVC - Model View Controller

CC - Create Custom Control

CPD - Chat Per Drama

ERD - Entity-Relationship Diagram

PC - Personal Computer

PO - Plano de Origem

RAM - Random Access Memory

SQL - Structured Query Language

UML - Unified Modelling Language

WWW - Abreviatura para designar World-Wide-Web

XHTML - eXtensible Markup Language

LISTA DE ABREVIATURAS E SIGLAS

.CLASS – Classe Java	12
ACOSC – Associação dos Criadores de Ovinos e Caprinos do Sertão do Cabugi	14
API – Sigla de Application Programming Interfaces	15
BD – Banco de Dados	15
BROWSER - Navegador	16
BYTECODE – Resultado da compilação de um programa Java	17
CRUD - Create, Retrieve, Update e Delete	18
DESKTOP – Microcomputador utilizado para trabalhar com sistemas	19
DLL – Dynamic-Link Library	19
DRY - Don't Repeat Yourself	20
HTML - HyperText Markup Language	22
HTTP - HyperText Transfer Protocol	23
JAR – Java ARchive	24
JDK – Java Development Kit	26
JIT – Just-In-Time	27
JSP – Java Server Pages	28
JVM – Java Virtual Machine	30
MVC – Model View Controller	32
OCX – Ole Custom Control	35
OPG – Ovos Por Grama	38
ORM - Object Relational Mapper	41
PC – Power Computer	41
PO – Puro de Origem	44
RAM – Random Access Memory	45
SQL – Structured Query Language	47
UML - Unified Modelling Language	49
WEB – Abreviatura para designar World-Wide-Web	50
XML - eXtensible Markup Language	51

1 INTRODUÇÃO	12
1.1 ORGANIZAÇÃO DO TRABALHO	14
2 OBJETIVOS	15
2.1 GERAL	15
2.2 ESPECÍFICOS	15
3 REFERENCIAL TEÓRICO	16
3.1 INTRODUÇÃO AOS SISTEMAS WEB	16
3.2 DESENVOLVIMENTO DE SOFTWARE EM CAMADAS	17
3.2.1 Camada de Apresentação	18
3.2.1.1 Padrão MVC (Model-View-Controller)	18
3.2.2 Camada de Aplicação	19
3.2.3 Camada de Modelo (ou Domínio)	19
3.3.4 Camada de Persistência	19
3.3.4.1 Mapeamento Objeto-Relacional	20
3.3 UML (UNIFIED MODELLING LANGUAGE)	20
3.4 A PLATAFORMA JAVA	22
3.5 RUBY	23
3.5.1 Sintaxe e Funcionalidades	24
3.5.1.1 Arrays e Hashes	26
3.5.1.2 Símbolos	27
3.5.2 Duck Typing	28
3.5.3 Closures	30
3.6 MIXINS	32
3.7 JRUBY	35
3.7.1 Acessando API do Java	36
4 METODOLOGIA	38
4.1 MODELO DO SISTEMA PROPOSTO	41
4.1.1 Controle Animal	41
4.1.2 Propriedade	44
4.1.3 Manejo Alimentar	45
4.1.4 Controle de Parição	47
4.2 IMPLEMENTAÇÃO COM RUBY ON RAILS	49
4.2.1 Arquitetura	50
4.2.2 Active Record	51
4.2.3 Migrations	52
4.2.4 Scaffolding	53
4.2.5 Fixtures	53
4.2.6 Plugins	54
5 CONCLUSÕES	55
REFERÊNCIAS	57

o controle da parição envolve, além dos dados citados acima, informações sobre a puberdade do animal, a forma através da qual ele atua em gestação (cruzamento ou inseminação artificial), o controle do animal durante a gestação, controle com a alimentação, a criação do animal até o nascimento e o acompanhamento do animal até a fase adulta. Além disso, não

1 INTRODUÇÃO

Nos últimos anos, a caprino-ovinocultura vem assumindo um papel significativo no contexto do agronegócio do Brasil. A produção de ovinos e caprinos representa uma importante alternativa de oferta de carne, leite, pele e derivados, não somente para a população rural, mas também para o mercado emergente das grandes metrópoles, que vêm consumindo de forma crescente carnes, leite e derivados destas espécies.

Diante de pesquisas realizadas, foi constatada a existência de um problema que afeta os praticantes dessa atividade: a falta de controle informatizado de animais. A criação de caprinos é uma atividade que lida com grande número de variáveis tais como o controle da produção de leite e de carne, aspectos reprodutivos, avaliação da condição fisiológica do rebanho, controle da genética animal, entre outros, o que inevitavelmente gera uma grande quantidade de dados que dificultam a análise individual dos mesmos.

Através do controle desses dados por um sistema de informação, seria possível obter informações do rebanho, do nascimento e crescimento dos animais, realizar acompanhamento de vacinação, dentre outras possibilidades. Com os dados apresentados por tal ferramenta, seria possível decidir se um animal deve ser um ovino ou caprino de leite, de corte ou reprodutor; realizar o controle da propriedade, para que seja feita uma avaliação da terra e seja possível realizar melhorias; realizar o manejo alimentar, que tem por finalidade controlar a alimentação do animal, a mudança de ração e o tipo de ração a ser utilizada; e, por fim, o controle de parição, no qual os criadores e fazendeiros fazem um acompanhamento do animal que vai parir até o nascimento de sua cria.

Com esta demanda, foram identificadas algumas soluções que poderiam ser implementadas de imediato, para que fosse efetuada uma administração mais precisa da cultura. Entretanto, percebe-se que o mercado oferece produtos que tratam diretamente cada caso, como controle animal, manejo alimentar e controle de parição, no entanto não fazem gestão de uma forma detalhada. Os produtos para controle de parição, por exemplo, só abordam informações sobre o animal que vai dar a cria, a raça do animal, o tempo de gestação, a quantidade de animais que nasceram e o sexo do animal que nasceu. No entanto, o controle de parição envolve, além dos dados citados acima, informações sobre a puberdade do animal, a forma através da qual ele entrou em gestação (cruzamento ou inseminação artificial), o controle do animal durante a gestação, controle com a alimentação, a engorda do animal até o nascimento e o acompanhamento do animal até a fase adulta. Além disso, não

foram encontrados produtos no mercado que abordem, dentro de um mesmo sistema de controle de rebanho caprino e ovino, o controle integrado dos animais com a propriedade, para que seja feita toda a gestão das melhorias das fazendas. Isso demonstra que os produtos existentes no mercado não são completos o suficiente para que se possa ter um controle apropriado de um rebanho.

Com a chegada da internet no campo, surge a oportunidade de se trabalhar em cima de produtos informatizados, nos quais o proprietário dos rebanhos caprinos e ovinos possa ter, de forma mais rápida e precisa, informações sobre seus rebanhos. É importante ressaltar que as soluções conhecidas hoje para este mercado, não são voltadas para ambiente de internet.

Diante do exposto e portando as informações necessárias, adquiridas através de um levantamento feito junto a zootecnista, veterinário, criadores de rebanho e pesquisas desenvolvidas na área de criação, foi possível mergulhar neste campo a fim de desenvolver um estudo onde possa ser dada uma parcela de contribuição para o crescimento da criação e da comercialização dos animais e dos produtos que eles geram, através do desenvolvimento de aplicações Web para a plataforma Java utilizando linguagens dinâmicas como alternativas à linguagem Java.

Existem diversas linguagens dinâmicas que podem ser utilizadas na máquina virtual Java, dentre as quais podemos destacar o Groovy, o JRuby, o Javascript. Este trabalho será desenvolvido com JRuby e um framework MVC chamado Ruby on Rails.

Para mostrar a eficiência da metodologia utilizada, será apresentada a sua utilização em projeto de sistema para apoio a consultoria de ovinos e caprinos, evidenciando-se, primeiramente, uma análise com diagramas de classe, que tem no seu “enfoque principal a permissão da visualização das classes que comporão o sistema com seus respectivos atributos e métodos, bem como em demonstrar como as classes do diagrama se relacionam, complementam e transmitem informações entre si” (GUEDES, 2004, p.69) e de casos de uso, que “possibilita a compreensão do comportamento externo do sistema por qualquer pessoa, tentando apresentar o sistema através de uma perspectiva do usuário” (GUEDES, 2004, p. 45), facilitando o desenvolvimento da aplicação e operacionalização do sistema proposto, com vistas ao sucesso dos empreendedores.

1.1 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado da seguinte maneira: no capítulo 2 serão apresentados o objetivo geral e os objetivos específicos do trabalho. No capítulo 3 são abordados os referenciais teóricos, onde são relacionados os conceitos, tecnologias e ferramentas necessárias ao desenvolvimento da aplicação. No capítulo 4 contém uma descrição do problema que este trabalho pretende resolver e mostra a metodologia utilizada para resolvê-lo, focando nas ferramentas do Ruby on Rails que foram utilizadas. No capítulo 5 são relacionadas as conclusões e perspectivas futuras.

OBJETIVOS ESPECÍFICOS

- Modelagem do sistema utilizando UML;
- Desenvolver aplicação utilizando a linguagem JRuby;
- Utilização de Framework MVC Ruby on Rails;
- Implantar a aplicação no servidor web Tomcat utilizando a máquina virtual Java (JVM).

2 OBJETIVOS TEÓRICO

O objetivo deste trabalho é implementar uma solução para a informatização do controle de rebanhos de caprinos e ovinos.

2.1 GERAL

Desenvolver uma aplicação web para facilitar o controle de informações referentes aos rebanhos por parte de consultores e criadores. A ferramenta a ser desenvolvida deverá funcionar na plataforma Java e utilizar a linguagem Jruby e o framework MVC Ruby on Rails. A modelagem da ferramenta será feita utilizando UML. O foco principal do produto são os consultores e criadores de rebanho caprino e ovino do estado do Rio Grande do Norte.

2.2 ESPECÍFICOS

- Modelagem do sistema utilizando UML;
- Desenvolver aplicação utilizando a linguagem JRuby;
- Utilização de Framework MVC Ruby on Rails;
- Implantar a aplicação no servidor web Tomcat utilizando a máquina virtual Java (JVM).

3 REFERENCIAL TEÓRICO

Para desenvolver este trabalho é necessário recorrer a alguns conceitos e tecnologias com o objetivo de acelerar e simplificar o desenvolvimento de aplicações web. Neste capítulo serão apresentados alguns desses conceitos.

3.1 INTRODUÇÃO AOS SISTEMAS WEB

Sistemas Web são sistemas que rodam em um servidor e que os usuários acessam com um browser através do protocolo Hypertext Transfer Protocol (HTTP). A tecnologia cliente-servidor é utilizada nos sistemas Web, pois é uma arquitetura onde o processamento da informação é dividido em módulos distintos de processos, onde, um é responsável pela informação, que é o servidor, e o outro pela captação da informação, o cliente. Existe uma troca cooperativa na interação entre o cliente e o servidor, onde o cliente requisita uma operação e o servidor processa e responde ao cliente. Existem sistemas que o processo servidor não responde diretamente. É criado um processo escravo para cada pedido que o cliente faz.

É possível citar como vantagens da tecnologia cliente-servidor a independência de plataformas, melhor desempenho, facilidade de acesso aos dados e redução de custos operacionais, como também, a escalabilidade, na qual o sistema pode ser expandido de forma vertical, com a adição de mais recursos a máquina servidora, ou ainda horizontalmente, com a aquisição de mais servidores.

Os sistemas acessados através da web em um browser são chamados de sistemas com clientes magros, ou thin clients. O oposto dos clientes magros são os clientes gordos, ou fat clients, que podem executar o processamento de informações e passar para o servidor somente os dados necessários para comunicação e armazenamentos dos mesmos. Os clientes magros possuem vantagens sobre os clientes gordos, tais como: baixo custo do hardware; baixo custo de administração por parte de técnicos; utilização de uma menor largura de banda de rede. Esses fatores fazem com que a relação custo benefício se torne bastante positiva. Por outro lado, existe o problema de existir um único ponto de falha, que é o fato de o servidor ou um Hub que interliga a rede inteira poder “cair”, fazendo com que toda a rede fique fora do ar.

Por isso, se faz necessário investir numa boa infra-estrutura de rede para que o dados possam trafegar de forma rápida e confiável.

3.2 DESENVOLVIMENTO DE SOFTWARE EM CAMADAS

O desenvolvimento de software em camadas traz possibilidade da separação de componentes por responsabilidades em comum. Em um grande projeto de software, onde se tenha uma equipe envolvida, chegará um momento que o código ficará praticamente ilegível. Com o desenvolvimento em camadas, teremos a facilidade de implementar códigos onde não existirá dificuldade de manutenção, um servidor de páginas HTML onde as interfaces dos aplicativos serão criadas. Para integrar as páginas com os servidores de aplicação, onde estão os objetos responsáveis pela lógica de negócio e os dados do banco de dados, pode ser utilizado JSP, por exemplo. Com isso, o usuário, para ter acesso a qualquer página, basta ter um navegador instalado em sua máquina.

Com o crescimento da demanda do desenvolvimento de sistemas web, causado principalmente pelo crescimento da internet, surgiu-se o modelo em camadas. A idéia básica desse modelo consiste em subdividir o sistema em camadas, diminuindo sua complexidade. A Figura 3.1 apresenta as três principais camadas (apresentação, domínio e dados), definidas durante o desenvolvimento de um sistema web, onde cada uma dessas camadas possui responsabilidades bem definidas, conforme apresentado na Figura 1.

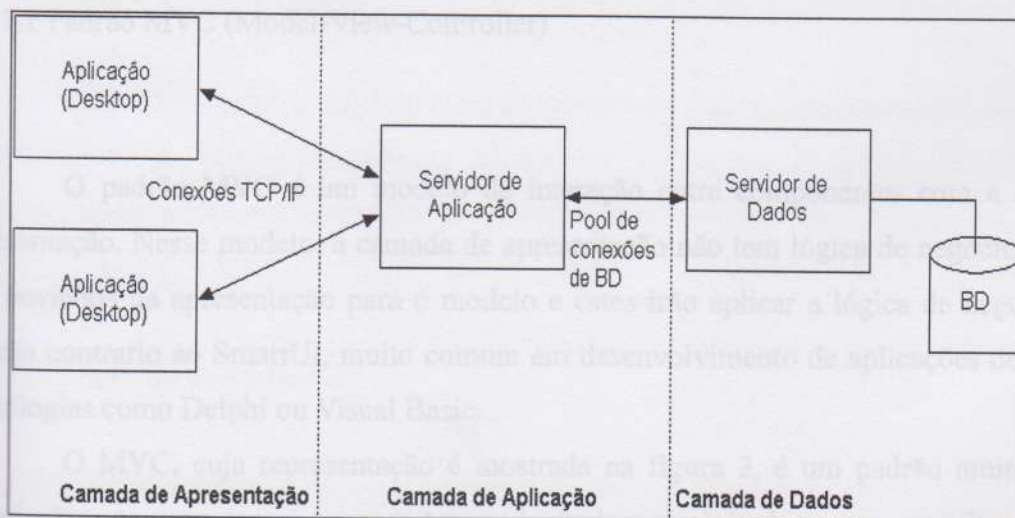


Figura 1: Desenvolvimento em camadas.

Fonte: Sauv , 2007.

Camada	Responsabilidade
Apresentação	Provisão de serviços, apresentar informações (por exemplo, em janelas ou páginas web)
Domínio	A lógica de negócio do sistema
Dados	Comunicação com o banco de dados, gerenciamento de transações.

Quadro1 - Descrição das responsabilidades de cada camada no modelo de três camadas.

Fonte: FOWLER, M., 2004.

No desenvolvimento em três camadas tem-se que cada camada possui uma responsabilidade bem definida. Apesar disso, uma das grandes dificuldades desse modelo de desenvolvimento consiste em subdividir o sistema e reconhecer quais partes do mesmo se encaixam em cada camada.

3.2.1 Camada de Apresentação

É a camada de visualização do usuário. O usuário visualiza dados e interage com componentes. Sua função é interpretar e manter a informação quando as aplicações estão sendo executadas, além de transformar e formatar os dados apresentados pelo usuário. É a peça do sistema que possibilita a interação do mesmo com as entidades externas.

3.2.1.1 Padrão MVC (Model-View-Controller)

O padrão MVC é um modelo de interação entre componentes com a camada de apresentação. Nesse modelo, a camada de apresentação não tem lógica de negócio. Os dados são enviados da apresentação para o modelo e estes irão aplicar a lógica de negócio. É um padrão contrario ao SmartUI, muito comum em desenvolvimento de aplicações desktop com tecnologias como Delphi ou Visual Basic.

O MVC, cuja representação é mostrada na figura 2, é um padrão muito utilizado principalmente por ter uma arquitetura legível e maleável e que permite uma fácil manutenção. Ele é utilizado para organizar a interação entre componentes da camada de

apresentação com a camada de modelo.

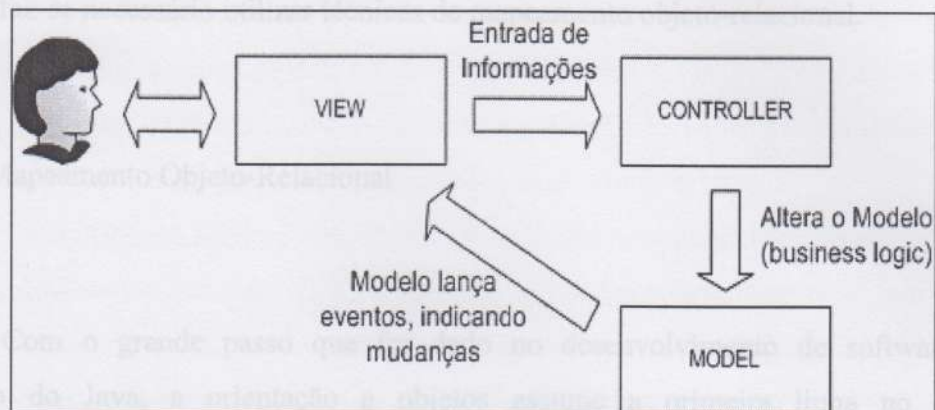


Figura 2 – Representação do Modelo MVC

Fonte: Almeida, 2007.

3.2.2 Camada de Aplicação

Faz a ligação da camada de apresentação com o modelo e normalmente é composto por objetos que seguem os padrões Facade ou Command. O propósito desta camada é servir de “janela” entre usuários no ambiente, pois é por ela que ocorre a grande troca de informações entre eles.

3.2.3 Camada de Modelo (ou Domínio)

Contém a lógica de negócio da aplicação e seus dados. Tem como uma das principais características a implementação da lógica da aplicação. É nessa camada que as funcionalidades referentes ao domínio do negócio são implementadas.

3.3.4 Camada de Persistência

A camada de persistência tem como objetivo tornar os objetos do domínio persistentes. Nesse trabalho será abordada a persistência em bancos de dados relacionais. Para

uma melhor abordagem na utilização de bancos de dados relacionais em sistemas orientados a objetos, faz-se necessário utilizar técnicas de mapeamento objeto-relacional.

3.3.4.1 Mapeamento Objeto-Relacional

Com o grande passo que foi dado no desenvolvimento de software devido a aceitação do Java, a orientação a objetos assume a primeira linha no cenário das implementações de aplicações. Diante de um modelo rico de dados e de conceitos que melhoram a produtividade, como encapsulamento, herança e polimorfismo, as tecnologias de objetos são as preferidas pelos atuais desenvolvedores.

Mesmo assim, grande parte dos dados residem em bancos de dados relacionais. Partindo daí, desenvolvedores de aplicações se vêem lutando com problema de diferença de impedância, ou seja, a diferença existente entre os modelos de dados relacionais e orientado a objetos.

Para um melhor entendimento, no modelo de dados relacional todos os dados são armazenados em linhas e colunas de uma tabela. Cada linha com dados representa um registro. Com a complexidade dos dados, outras tabelas podem ser criadas e armazenar informações relacionadas. Dessa maneira, cada tabela vai armazenar determinados dados, que juntos vão compor a informação completa.

Já, no modelo orientado a objeto não existe o armazenamento em linhas e colunas. O desenvolvedor cria um modelo onde ele cria uma determinada classe de informação, onde um registro é um objeto. Todos os objetos contém os itens do registro. Os métodos das classes atuam diretamente sobre os dados descritos pelas classes.

3.3 UML (UNIFIED MODELLING LANGUAGE)

Na fase de análise de requisitos, com a utilização de sistemas desenvolvidos orientados a objetos, não existia uma notação padronizada nem eficaz que tivesse uma abrangência de uma aplicação como todo. Desta forma surgia algumas simbologias, cada uma com seus conceitos pré-definidos, seus gráficos e terminologias.

Através do lançamento da UML houve um entusiasmo muito grande por parte dos desenvolvedores de software porque viram nesta nova forma de modelar uma padronização que foi o que eles sempre esperaram. Surgia, junto com a UML, novos conceitos que não eram tão utilizados, fazendo com que os profissionais não aprendessem somente a simbologia e o seu significado, mas aprendessem a modelar orientado a objetos.

Grady Booch, James Rumbaugh e Ivar Jacobson, conhecidos como “os três amigos”, foram os desenvolvedores da UML. Diante de um vasto conhecimento de modelagem, eles resolveram juntar o que havia de melhor nas metodologias utilizadas e assim fizeram nascer a Unified Modelling Language.

A UML possui uma quantidade grande de diagramas que tem por objetivo fornecer algumas visões do sistema a ser modelado, podendo o mesmo ser analisado e modelado sob diversos aspectos. A idéia é que cada diagrama complemente o outro, pois cada um tem a função de analisar o sistema a partir de uma determinada ótica. Os diagramas estão listados a seguir:

- Diagrama de Caso de Uso: É um diagrama mais geral, sempre utilizado em levantamento de informações e análise de requisitos. Mesmo assim, geralmente é consultado em todas as fases do projeto. Ele é escrito em termos de atores externos, casos de uso e o sistema modelado;
- Diagrama de Classes: O mais importante e utilizado da UML. Serve de apoio para todos os outros diagramas. Ele demonstra a estrutura estática das classes de um sistema;
- Diagrama de Objetos: Utiliza, praticamente, todas as notações do Diagrama de Classes. Funciona como um complemento do Diagrama de Classes, pois é bastante dependente dele;
- Diagrama de Estado: Este diagrama procura acompanhar todas as mudanças sofridas pelo objeto dentro de um determinado processo;
- Diagrama de Seqüência: Este acompanha a ordem temporal em que as mensagens são trocadas nos objetos. Ele mostra a interação entre os objetos, informando o que vai acontecer em um determinado ponto do sistema;
- Diagrama de Colaboração: Semelhante ao Diagrama de Seqüência, ou melhor, um complementa o outro. A diferença está no Diagrama de Colaboração que não se preocupa com a temporalidade do processo. Este diagrama se refere mais a troca de mensagens entre os objetos;

- Diagrama de Atividades: Este diagrama está mais preocupado em descrever os passos que serão percorridos para que uma atividade específica seja concluída. O foco é a execução da implementação de um método e suas atividades na instância de um objeto;
- Diagrama de Componentes: É o diagrama que está totalmente associado a linguagem que vai ser utilizada para a implementação do sistema modelado. Ele mostra o sistema pelo lado funcional;
- Diagrama de Implantação: Este determina qual o hardware que o sistema vai precisar para que seja executado de maneira eficiente;
- Diagrama de Pacotes: É o diagrama que representa todos os subsistemas, e as partes que eles compõem.

3.4 A PLATAFORMA JAVA

O sistema a ser construído neste trabalho deverá ser executado na plataforma Java. A plataforma Java foi criada por James Gosling e surgiu como um produto do projeto Green, financiado pela Sun Microsystems para o desenvolvimento de softwares para dispositivos inteligentes voltados ao usuário final. Inicialmente, era chamada de Oak, mas descobriram que já existia uma linguagem com esse nome e ele teve que ser mudado. O nome Java surgiu após uma conversa entre membros da equipe de desenvolvimento em uma cafeteria que servia cafés importados da ilha de Java.

O Java foi anunciado pela Sun no ano de 1995 durante uma conferência. Como existia um grande interesse pelo WWW, a linguagem Java passou a ser vista com bons olhos pelo fato de poder ser utilizada para criar páginas na web, utilizando conteúdo interativo e dinâmico, desenvolver aplicações corporativas de grande porte, aprimorar as funcionalidades de servidores World Wide Web, fornecer aplicativos para usuários finais, dentre outras funcionalidades.

Os programas em Java consistem em classe, que por sua vez consistem de métodos que realizam tarefas e retornam o resultado quando estas são completadas. A linguagem Java é muito rica nas suas coleções (classes que existem em bibliotecas), que também são conhecidas com Application Programming Interfaces (APIs).

Pode-se citar como uma das grandes vantagens da plataforma Java, o fato de os programas poderem ser compilados independente de plataforma. O fato dos arquivos de byte serem pequenos facilitam para que eles sejam carregados rapidamente na internet. Um outro ponto positivo do Java é que a linguagem tem uma biblioteca muito vasta com componentes e recursos, tratamento de eventos, funções matemáticas, entradas e saídas, dentre outras. A linguagem é orientada a objetos e isso facilita a reutilização de código. Possui herança simples e desalocação automática de memória (coletor de lixo).

Os programas em Java são executados em uma máquina virtual chamada a Java Virtual Machine (JVM). Para que um código seja executado por esta máquina ele precisa possuir a extensão .class e possuir código compatível com as instruções do Java.

O código é compilado e gera instruções chamadas bytecode, que por sua vez é aplicado a JVM, que se encarrega de interpretar os comandos para o Sistema Operacional que está rodando, ou seja, a JVM faz a tradução do das instruções do código Java.

Em Java, o código é compilado somente uma vez gerando o bytecode que poderá ser interpretado por qualquer Máquina Virtual Java, rodando em Windows, Linux, Mac OS, Solaris, dentre outros sistemas operacionais que possuam uma JVM implementada.

A JVM não permite que um programa Java acesse recursos do hardware diretamente, protegendo o computador de operações perigosas, como acesso a regiões protegidas da memória ou formatação física do disco rígido.

3.5 RUBY

Ruby é uma linguagem criada por Yukihiro Matsumoto, orientada a objetos e com tipagem dinâmica e forte. O Ruby tem uma ligação muito forte com o Smalltalk, e possui construções semelhantes a essa linguagem.

A linguagem Ruby é puramente orientada a objetos. Tudo que é manipulado em Ruby são objetos e os resultados dessas manipulações também são objetos. Um texto é um objeto da classe String, um número inteiro é um objeto da classe Fixnum e um vetor é um objeto da classe Array.

A linguagem oferece sintaxe e funcionalidades que facilitam o trabalho do programador na hora do desenvolvimento de aplicações. Iremos descrever sobre classes, métodos, escopo, arrays, hashes e os símbolos que fazem parte dessa tecnologia.

3.5.1 Sintaxe e Funcionalidades

O Ruby tem uma sintaxe bastante simples, com possibilidade de sobrecarga de operadores, utilização do conceito de módulos, permite adicionar métodos em uma classe em tempo de execução, possui blocos em sua sintaxe e um garbage collector que atua em todos os objetos.

Em Ruby as classes tem um método initialize como construtor. Para que seja possível entender melhor a criação de uma classe Ruby, pode-se ver o exemplo do Código 1, que é a construção da classe Carro. Nela, serão passadas as informações da marca, modelo e cor. Quando uma classe contém um método initialize, o método é automaticamente chamado quando um objeto é criado pelo método new.

```
class Carro
  def initialize(marca,modelo,cor)
    @marca = marca
    @modelo = modelo
  @cor = cor
  end
end
gol = Carro::new("Volkswagen", "Gol", :azul)
puts gol
```

Código 1 – Criação de uma classe Ruby

Fonte: Dados do Pesquisador (2007)

As variáveis @marca, @modelo e @cor são variáveis de instância da classe Carro, ou seja, atributos da classe. Qualquer variável definida através da utilização de um "@" na frente do nome da variável é um atributo da classe. No Código 2 é possível verificar a utilização de tais variáveis. Nessa listagem, além do construtor é definido um método to_s, que tem como função converter objetos para String.

```

class Carro
  def initialize(marca,modelo,cor)
    @marca = marca
    @modelo = modelo
    @cor = cor
  end

  def to_s
    "marca: " + @marca + " modelo: " + @modelo + " cor: " + @cor
  end
end

gol = Carro.new("Volkswagen", "Gol", :azul)
puts gol

```

Código 2 – Classe mostrando utilização de variáveis e chamada de métodos

Fonte: Dados do Pesquisador (2007)

Uma das coisas interessantes do Ruby é que as classes nunca são fechadas, por isso, se for necessário adicionar mais algum novo método num programa, dinamicamente, basta inserir o método utilizando a sintaxe: `def novo_método` e finalizando com `end`. Feito isso, as instâncias já podem acessar seu novo método. Com isso, pode-se alterar a classe sem ter que escrevê-la novamente.

Existem também as variáveis de classe que, em Java, seriam as variáveis estáticas. Tais variáveis estão no contexto da classe e não da instância, de modo que todas as instâncias da classe acessam uma mesma variável de classe. Elas começam com `@@`, e devem ser inicializadas antes de ser utilizadas.

Para se definir métodos, basta que os definam utilizando `def`, e finalizando com `end`. Em Ruby cada objeto é constituído de métodos. Para utilizar um método se faz necessário colocar um ponto após o objeto e adicionar o nome do método. Um exemplo prático é: `puts ("Hello World".upcase)`. O método `upcase` foi utilizado para deixar o "Hello World" todo em maiúsculo. Métodos como `puts` e `gets` estão disponíveis em todo lugar e não precisam ser associados a um objeto específico. Eles são fornecidos por um Kernel do Ruby e estão em todos os objetos do Ruby. Quando executamos uma aplicação Ruby, um método `main` é automaticamente criado e este objeto fornece acesso aos métodos do Kernel. Utilizar `"( )"` nos métodos é opcional, apesar do Ruby estar se movimentando para a utilização dos parênteses, pois eles ajudam a evitar ambigüidade nos códigos.

Código 3 – Demonstração de utilização de Arrays

Fonte: Dados do Pesquisador (2007)

3.5.1.1 Arrays e Hashes

Em Ruby também é possível declarar Arrays. Um único Array pode ter itens de tipos de dados diferentes, como: String, inteiros e números de ponto flutuante. Um Array é uma coleção seqüencial de itens, onde cada um pode ser indexado. Vejamos um exemplo de array: `a1 = [ 1, "dois", 5.3, array.length(a0)]`. No Array o primeiro índice é 0 (zero), o que significa que o último vai ser igual a quantidade de itens menos um. Para criar um Array, assim como em outras linguagens, o Ruby também utiliza `[ ]` para delimitar. Os Arrays também são objetos e são definidos por uma classe Array.

Para criar Arrays multi-dimensionais, basta criar um Array e adicionar um "slot" para dentro desse Array. Pode-se ver esse comportamento no Código 3.

```
a = Array.new(2)
a [0] = Array.new(2, 'hello')
a [1] = Array.new(2, 'world')
```

Código 3 – Demonstração de criação de Arrays multi-dimensionais

Fonte: Dados do Pesquisador (2007)

Ou ainda, pode-se aninhar um Array dentro de outro utilizando colchetes, conforme mostrado no Código 4.

```
a = [ [1,2,3,4 ],
      [5,6,7,8 ],
      [9,10,11,12 ],
      [13,14,15,16 ] ]
```

Código 4 – Demonstração aninhamento de Array

Fonte: Dados do Pesquisador (2007)

Pode ser utilizado indexação de Arrays utilizando o sinal (-) no fim de um Array. O valor do índice do último elemento será -1. Isso pode ser visto no Código 5.

```
arr = ['h','e','l','l','o',' ','w','o','r','l','d']
print( arr [0,5 ] )           #=> 'hello '
print( arr [5,5 ] )           #=> 'world '
print( arr [0..4 ] )           #=> 'hello '
print( arr [5..1 ] )           #=> 'world '
```

Código 5 – Demonstração de indexação de Arrays

Fonte: Dados do Pesquisador (2007)

As Hashes, assim como os Arrays, fornecem uma boa forma de indexar uma coleção de itens por número, no entanto, existem situações que utilizar Hashes é mais conveniente. Pode-se imaginar que gostaríamos de armazenar receitas de um restaurante de comida sertaneja, sendo mais interessante organizar pelo nome do prato, como: “pernil caprino”, “picanha de bode” e “costelinhas de cordeiro”. Seria muito mais indexar por números. Diante disso, o Ruby tem a classe Hash que permite o usuário fazer isso. O armazenamento se daria como num dicionário, que terão as entradas armazenadas com alguma chave única. Pode-se criar uma hash criando uma nova instância da classe Hash.

```
h1 = Hash.new
h2 = Hash.new("Some kind of ring")
```

Código 6 – Criação de uma Hash
Fonte: Dados do Pesquisador (2007)

Um objeto Hash sempre terá um valor padrão, ou seja, um valor que é retornado quando nenhum valor específico é encontrado em um dado índice. No exemplo acima, as duas Hashes foram criadas vazias, no entanto, temos a Hash h2 inicializada com “Some kind of ring”, enquanto que h1 terá o valor padrão nil.

3.5.1.2 Símbolos

Os símbolos representam nomes e algumas Strings dentro do interpretador Ruby. Eles são sempre o mesmo objeto durante a execução de um programa. Para criarmos um símbolo, basta colocar “:” na frente do seu nome. Por exemplo, :modelo, :combustível. O exemplo mostrado no Código 7 mostra o fato de dois símbolos serem sempre o mesmo objeto.

```
s1 = :carro
s1.id #=> 2514190

s2 = :carro
s2.id #=> 2514190

s3 = :modelo
s3.id #=> 2514132
```

Código 7 – Demonstração do fato de símbolos representarem sempre o mesmo objeto.
Fonte: Dados do Pesquisador (2007)

3.5.2 Duck Typing

Notamos que em Ruby não se declara tipos de variáveis ou métodos, tudo é um tipo de objeto. Os objetos Ruby podem ser modificados individualmente. Sempre podem ser adicionados métodos em uma base por objetos.

Em Ruby se confia mais na capacidade de um objeto do que no seu tipo ou classe. Duck Typing significa um tipo de objeto que é definido por aquilo que ele pode fazer, não pelo que ele é. Duck Typing se refere a uma tendência do Ruby de ser menos preocupado com a classe de um objeto e se detém mais aos com os métodos que podem ser chamados nele e com as operações que podem ser realizadas sobre eles. Em Ruby poderia ser usado `respond_to?` Ou podem simplesmente passar de um objeto para um método e saber que uma exceção será criada se for usada inadequadamente. Esse comportamento pode ser visto no Código 8.

```
puts ('A string'.respond_to? :to_str) # => true
puts (Exception.new.respond_to? :to_str) # => true
puts (4.respond_to? :to_str) # => false
```

Código 8 – Exemplo de Duck Typing
Fonte: Dados do Pesquisador (2007)

O código acima é mais um simples exemplo do Ruby utilizando a filosofia Duck Typing: Se um objeto age como um Pato, ou seja, possui os métodos de um Pato, o Ruby irá tratá-lo como um Pato. Sempre que possível um objeto deve ser tratado de acordo com o método que, em vez de definir a classe que herdaram ou os módulos que incluem.

Exceções, por exemplo, são essencialmente Strings que tem informação extra que são associadas. Mas não subclasses do nome “String”. O código usa `is_a`. String que checa o `stringness` domina essencialmente a exceção.

Um exemplo de Duck Typing pode ser visto nos Códigos 9, 10 e 11. Considerando as classes Pato e Ganso, pode-se perceber que ambas contém os métodos falar e nadar. Os métodos `fale` e `nade`, no Código 11 recebem um animal e chama os seus métodos falar e nadar. Com o Duck Typing é possível utilizar esses métodos tanto para Pato quanto para Ganso mesmo sem que eles sejam subclasses de uma classe comum, como é comum no polimorfismo.

```
class Pato
```

```
  def falar
    'Quack!'
  end
```

```
  def nadar
    'Splash, splash, splash...'
  end
```

```
end
```

Código 9 – Classe Pato

Fonte: Dados do Pesquisador (2007)

```
class Ganso
```

```
  def falar
    'Honk!'
  end
```

```
  def nadar
    'Splash splash splash...'
  end
```

```
end
```

Código 10 – Classe Ganso

Fonte: Dados do Pesquisador (2007)

```
def fale(animal)
  animal.falar
end
```

```
puts fale(Pato.new) # Quack!
puts fale(Ganso.new) # Honk!
```

```
def nade(animal)
  animal.nadar
end
```

```
puts nade(Pato.new) # Splash splash splash...
puts nade(Goose.new) # Splash splash splash...
```

Código 11 – Exemplo de Duck Typing

Fonte: Dados do Pesquisador (2007)

3.5.3 Closures

Existe um interesse muito grande em linguagens dinâmicas, com isso, tem crescido a quantidade de interessados em aprender o conceito de programação chamado Closures ou Blocks. Pessoas que programam em C / C++ / Java / C# como linguagem de background, não tem o conhecimento da força dessas linguagens.

Closures estão sendo utilizados há muito tempo. Uma das primeiras vezes que foi encontrado adequadamente foi no Smalltalk onde se denominavam Blocks. Eles estão presentes na linguagem Ruby e é um dos principais motivos que os desenvolvedores desta tecnologia gostam de utilizar.

Na sua essência, o closure é um bloco de códigos que pode ser passado como argumento para o chamado de uma função. Para ilustrar melhor, vejamos o seguinte exemplo: Eu tenho uma lista de objetos e quero uma relação de funcionários que são gestores, o que vou determinar, com uma propriedade chamada isManager. Em Ruby, que é uma linguagem que possui Closure, deve-se fazer da forma mostrada no Código 12:

```
def gerente(emps)
  return emps.select {|e| e.isGerente}
end
```

Código 12 – Exemplo de Closures
Fonte: Dados do Pesquisador (2007)

O método select é um método definido na collection do Ruby. Identifica um bloco do código, o Closure e o argumento. Em Ruby se escreve um bloco de código em curlies, mas não é a única forma. Se um bloco de código tem qualquer argumento, ele é declarado entre barras verticais "|". O iterate seleciona a matriz de entrada, executa o bloco com cada elemento, e retorna um array desses elementos para que o bloco seja avaliado.

Uma diferença do Closure em relação a outras linguagens como C e Java é que Closure pode se referir a variáveis visíveis no momento em que elas são definidas. O Código 13 mostra um exemplo disso.

```
def salarios_altos(emps)
  limiar = 5000
  return emps.select {|e| e.salario > limiar }
end
```

Código 13 – Referência de variáveis
Fonte: Dados do Pesquisador (2007)

Percebe-se que o código do bloco escrito se refere a uma variável local no método. Muitas das linguagens que utilizam Closures como alternativa não tem verdadeiramente como fazer isso. Closures permite fazer muita coisa interessante: o Código 14 mostra uma função que retorna um Closure. Ele tem o comportamento dependente do argumento enviado. Pode-se criar esta função e associá-la a uma variável.

```
def mais_pago(qtd)
  return Proc.new {|e| e.salario > qtd}
end
```

Código 14 – Exemplo de Closure
Fonte: Dados do Pesquisador (2007)

```
salarios_altos = mais_pago(5000)
```

Código 15 – Função associada a variável
Fonte: Dados do Pesquisador (2007)

A variável `salarios_altos`, do Código 15, contém um bloco de código que retornará um objeto testado se o salário for maior que 5000.

```
joao = Funcionario.new
joao.salario = 2000
print salarios_altos.call(joao)
```

Código 16 – Variável `salarios_altos`
Fonte: Dados do Pesquisador (2007)

A expressão `salarios_altos.call(joão)`, no Código 16, chama o código `e.salario > qtd` que foi definido anteriormente, com a variável `amount` o código vincula 2000 quando é criado o objeto `Proc`. Mesmo que o valor de 2000 saia do escopo quando for chamado o `print`, a ligação permanece.

Com isso, é possível concluir que o primeiro ponto importante é que Closures é um bloco de código. Isso formaliza e estabelece que Closures vai além de uma função de ponteiros e técnicas similares.

Existe uma segunda diferença formal, bastante importante, principalmente na prática. Linguagens que suportam Closures permitem que as definições sejam feitas com pouca sintaxe. Mesmo não aparentando tanta importância, é fundamental, é a chave para torná-lo natural e de uso freqüente. Olhando para Lisp, Smalltalk ou Ruby, é possível ver Closures em todo lugar, e com muito mais freqüência é utilizado em estruturas semelhantes de outras linguagens. A capacidade de se ligar a variáveis locais é muito grande, mas um dos principais motivos desta utilização é pela clareza e simplicidade.

Um exemplo disso acontece quando usuários de Smalltalk começaram a trabalhar com Java. Pessoas experientes fizeram testes para transformar algumas das coisas que haviam sido feitas em Blocks no Smalltalk. O resultado foi um código super confuso.

Como tudo em software existe uma grande quantidade de definições para Closures. Alguns falam que este termo só se aplica a um valor real, que inclui uma ligação a partir do seu ambiente, como um valor retornado por `highPaid`. Outros utilizam o termo Closures para se referir a uma programação de construção que tem a capacidade de se ligar ao ambiente.

O Ruby utiliza muito Closures, mas não tem a tendência de criar Procs e encaminhá-los. Grande parte da utilização de Closures é baseado na `CollectionClosureMethod`. Outro uso comum é o método `execute`, quando processa um arquivo.

```
File.open(filename) {|f| fazer_algo(f) }
```

Código 17 – Variável `highPaid` fazendo teste

Fonte: Dados do Pesquisador (2007)

No exemplo do Código 17, o método abre o arquivo, executa o bloco fornecido e fecha. Esta pode ser uma linguagem muito bastante acessível para se trabalhar com transações. O uso pode ser feito extensivamente no XML para rotinas de transformação.

A utilização do Closures é atualmente muito menor do que as pessoas do Lisp e programações funcionais fazem. É uma coisa pequena a primeira vista, no entanto, cresce muito quando se passa a conhecer melhor o Closures.

3.6 MIXINS

Na linguagem Ruby, quando um método é acrescentado em tempo de execução a uma classe, ele é chamado de Mixin. Este mecanismo permite ao programador estender um módulo ou uma classe e, junto com o Duck Typing, tras bastante flexibilidade ao programador. A utilização de Mixins pode ser utilizado para simular o uso de interfaces ou mesmo de Herança Múltipla.

Como exemplo, será utilizado o Código 18, que define um módulo que converte um número inteiro para português:

```

# Converte um valor inteiro para Português
module Stringify
  # Requer uma variável de instância @value
  def stringify
    if @value == 1
      "Um"
    elsif @value == 2
      "Dois"
    elsif @value == 3
      "Três"
    end
  end
end
end

```

Código 18 – Método para escrever um número em português
Fonte: Dados do Pesquisador (2007)

O módulo Stringify faz uso da variável `@value` no exemplo. A classe que for Mixin desse módulo precisará definir a variável `@value` na instância deste módulo para usá-la. Além de variáveis exemplo um módulo não podia invocar métodos definidos no módulo em si, mas na classe que vai ser mixin.

Deve-se agora construir um módulo container que não seja dependente da aplicação de qualquer classe e que pode ser mixin.

```

module Math
  def add(val_one, val_two)
    BigInteger.new(val_one + val_two)
  end
end
end

```

Código 19 – Construção do módulo container
Fonte: Dados do Pesquisador (2007)

Os métodos no módulo Math, Código 19, são invocados como métodos de classes, também conhecidos como métodos estáticos. O método `add` no módulo Math aceita dois valores inteiros e retorna uma instância `BigInteger`. Deve-se agora definir a classe `BigInteger`, conforme o Código 20.

```

class Number
  def intValue
    @value
  end
end
end

```

Código 20 – Exemplo de mixin através do extend
Fonte: Dados do Pesquisador (2007)

```
class BigInteger < Number
```

```
  include Stringify
```

```
  extend Math
```

```
  def initialize(value)
```

```
    @value = value
```

```
  end
```

```
end
```

Código 20 – Definição da classe BigInteger com mixin

Fonte: Dados do Pesquisador (2007)

A classe BigInteger definiu um construtor e herda diretamente um método da classe Number. Para misturar os métodos implementados como os módulos Math e Stringify e a classe BigInteger, usa-se os métodos include e extend, respectivamente. Um exemplo do uso do método intValue, herdado da classe Number, pode ser visto no Código 21.

```
bigint1 = BigInteger.new(10)
```

```
puts bigint1.intValue # --> 10
```

Código 21 – Criação de objeto BigInteger e chamada de método herdado

Fonte: Dados do Pesquisador (2007)

O método extend irá misturar os métodos de um módulo a uma classe. O extend, na listagem 3.20, definiu que o método add do módulo Math pode ser usado como método estático da classe BigInteger. No Código 22 pode ser visto o uso do método add.

```
bigint2 = BigInteger.add(-2, 4)
```

```
puts bigint2.intValue # --> 2
```

Código 22 – Exemplo de mixin através do extend

Fonte: Dados do Pesquisador (2007)

O método include vai misturar os métodos dos módulos à classe, como mostra o exemplo do Código 23, significando que o método se tornará uma instância do método. O método definido no módulo Stringify pode ser utilizado em uma instância da classe BigInteger.

```
puts bigint2.stringify # --> 'Dois'
```

Código 23 – Exemplo de mixin através do extend

Fonte: Dados do Pesquisador (2007)

Existe uma outra utilização do método `extend`. Pode-se melhorar o objeto instanciando pelo `mixing` com o módulo em tempo de execução. É um recurso muito poderoso. Permite criar um módulo que será utilizado no objeto para o `extend`. A mudança é no tempo de execução.

Para juntar uma instância de um objeto com um módulo, deve-se chamar o método `extend` do objeto passando como parâmetro o módulo, conforme mostram os Códigos 24 e 25.

```
module CurrencyFormatter
  def format
    "$#{@value}"
  end
end
```

Código 24 – Código Mixin
Fonte: Dados do Pesquisador (2007)

```
bigint2.extend CurrencyFormatter
puts bigint2.format # --> '$2'
```

Código 25 – Adicionando método em tempo de execução
Fonte: Dados do Pesquisador (2007)

Chamando o método `extend` em uma instância só extenderá um objeto, os objetos de uma mesma classe não serão estendidos com uma nova funcionalidade. Chamar o método `format` em um outro objeto da mesma classe irá provocar um erro.

3.7 JRUBY

O JRuby é a implementação da linguagem Ruby para a máquina virtual Java. Por isso é fortemente integrado ao Java para permitir a incorporação de código Ruby em qualquer aplicação Java, com acesso bidirecional entre Java e Ruby.

Scripts ficam cada vez mais populares com os desenvolvedores pela facilidade de escrevê-los, onde precisamos de menos tempo para implementar e exige pouca manutenção. Desenvolvedores continuam criando novas linguagens para o desenvolvimento de novos scripts, mesmo sabendo que já existem vários desenvolvidos e muitas outras, para determinados fins específicos, como por exemplo, JavaScript que é muito utilizada para desenvolvimento de interface web.

Ruby, que se centra na simplicidade e na produtividade, é uma linguagem de script

interpretada orientada a objetos, e o JRuby é uma linguagem de programação Java implementada em Ruby utilizando suas sintaxes, núcleos de bibliotecas e as bibliotecas padrões.

O JRuby tem todas as vantagens do Ruby e a interoperabilidade de aplicações da plataforma Java e a uma gama de funcionalidades da plataforma Java. O JRuby e o Java é uma combinação que pode ser aplicada em qualquer tipo de programação. Por exemplo, a partir do script JRuby, é possível chamar a biblioteca Math do Java para utilizarmos a capacidade computacional, ou ainda, chamar a biblioteca Swing para mostrar uma caixa de diálogo que precisa de uma entrada de um usuário antes do script permitir que prossiga a operação. Podem, também, ser utilizados o JSR 223 Scripting APIs ou o Bean Scripting Framework (BSF) APIs para chamar um script JRuby de dentro de uma aplicação Java.

O JRuby requer o JDK 1.4 ou versões mais atuais. Depois que esta tudo instalado, para que se possa iniciar o Ruby, basta digitar `jirb` na linha de comando. O prompt do Ruby aparece da seguinte forma: `irb(main):003:0`, onde se pode testar o `puts`. Nota-se que `=> nil` é retornado pelo `puts` quando o valor retorna nulo, conforme pode ser visto no Código 26.

```
server jirb
irb(main):003:0> puts "Olá, mundo!"
Olá, mundo!
=> nil
irb(main):004:0>
```

Código 26 – Exemplo de prompt com o puts para testes
Fonte: Dados do Pesquisador (2007)

Para executar, basta colocar o `puts` na linha de comando para o arquivo, tendo como exemplo o `myprog.rb`, no Código 27.

```
jruby myprog.rb
Hi There!
```

Código 27 – Forma de executar o comando puts
Fonte: Dados do Pesquisador (2007)

3.7.1 Acessando API do Java

Pode-se chamar a API da plataforma Java ou qualquer outra API escrita em Java a

partir de um programa escrito em JRuby. Para isso, basta usar o comando `require` com o argumento `java`, para informar ao JRuby que serão utilizadas APIs escritas em Java e, em seguida, utilizar o `require` passando como parâmetro o nome da classe ou o nome de um arquivo jar que se deseja importar para o programa Ruby. O exemplo do Código 28 mostra como utilizar o `require` para referenciar ou incluir as bibliotecas da API do Java.

```
#Definição das dependências
require 'java'
require 'xml-apis-1.3.02.jar'

#Utilização da API
...
org.w3c.dom.Element root = g.getRoot()
svgdocument = impl.createDocument(svgNS, "svg", null)
org.w3c.dom.Node node = svgdocument.importNode(root, true)
...
```

Código 28 – Exemplo de utilização da API Java em um programa Ruby
Fonte: Dados do Pesquisador (2007)

Para executar um código JRuby dentro de uma aplicação Java, a JSR 233 (“Scripting na plataforma Java”) fornece uma API para chamar um script de dentro de uma aplicação Java e realizar a troca de dados entre a aplicação e o script. Estas características tornam possíveis a combinação de scripts existentes com aplicações Java e para estender a aplicação Java.

4 METODOLOGIA

A caprinocultura tem tido um destaque especial no sertão do Nordeste, principalmente pelo clima árido que favorece a criação da maioria das raças destas espécies. Os primeiros animais chegaram ao Brasil na época da colonização, trazidos da Europa. A convivência com as adversidades do sertão nordestino fizeram com que os animais adquirissem uma característica de adaptação e resistência ao calor, que foram primordiais para sua sobrevivência.

Os animais foram isolados geograficamente em seus respectivos habitat, separados por suas características e deram origem as raças que hoje predominam no nordeste brasileiro. Raças como Moxotó e Canindé são mais encontradas em nosso estado. Essas raças têm como características a rusticidade da pele, a prolificidade, ou seja, o número de crias paridas por fêmeas paridas, e a qualidade do couro desses animais.

No Rio Grande do Norte a criação de cabras tem uma maior concentração na região central do estado, onde podemos destacar as cidades de Lajes, Angicos, Santana do Matos, Afonso Bezerra, Pedro Avelino e Pedra Preta.

O investimento feito por criadores desta cultura tem dado muito resultado, principalmente no que diz respeito ao animal de produção, pois, além do leite, os investimentos também rondam a produção de queijo, coalhada, manteiga e até sorvete com leite de cabra. A pele também tem sido de muita utilidade principalmente na confecção de produtos como bolsas, carteiras, porta cartões, cintos e muito mais.

A cada ano que passa o volume de negócios nesta área cresce. Grandes feiras e eventos tem sido constantes durante todo ano em várias partes do país, e no Rio Grande do Norte tem-se como referência a festa do Boi, realizada em Parnamirim, a Festa do Bode, em Mossoró e a Caprifeira de Lajes. Um dos fatores que também tem colaborado bastante é a abertura do mercado internacional, que vem em busca de otimizar cada vez mais as suas unidades produtivas, para que se tornem mais competitivas.

As associações tem tido um papel preponderante no desenvolvimento desta cultura, pois, tem levado ao criador a amostra da importância do trabalho, as várias frentes em que se pode trabalhar, tanto com o animal quanto com seus derivados, e ao consumidor ser oferecido um produto de boa qualidade, com credibilidade e com valores bem acessíveis.

O investimento tem sido alto e a tecnologia está chegando ao campo. Hoje a internet, é um elo entre todas as pessoas que se envolvem em qualquer ramo de atividade, e não

poderia ser diferente com os criadores de caprinos. A chegada da rede mundial no campo tem sido de grande valia. A quantidade de informações que podem ser adquiridas, os contatos, inclusive com comércio exterior, os fóruns de discussões existentes para a melhoria de rebanho, a troca de informação visando o melhoramento da caprinocultura, são alguns dos benefícios que se pode citar.

Mesmo com estas facilidades que a tecnologia trás, não se deve esconder que ainda tem muito que fazer pelo campo, em especial pela caprinocultura. As dificuldades são muito grandes, principalmente no tocante as dificuldades geo-climáticas, como os longos períodos de estiagem, o tipo de solo, a degradação do ecossistema "caatinga", disponibilidade de recursos para investimento e, até mesmo, o aspecto sócio-cultural.

Diante de pesquisas e conversas, foram identificados alguns aspectos que caracterizam a caprinocultura no nordeste brasileiro, que colaboram para que os criadores tenham dificuldades como as características geográficas da região, a forma que o sistema de produção é implementado na atividade, parâmetros produtivos e reprodutivos e a cadeia reprodutiva de caprinos na região.

No nordeste encontra-se o semi-árido brasileiro abrangendo cerca de 75% dessa região, isso representa 13% do Brasil. O interior caracteriza-se por apresentar clima quente e seco, com chuvas irregulares. Só existem duas estações: inverno, que dura de dois a quatro meses, e o verão que é totalmente seco. O semi-árido apresenta sub-regiões, como: agreste, cariri, seridó, curimataú e sertão. Nessas sub-regiões a vegetação que predomina é a caatinga, que tem por característica uma vegetação com plantas de baixo e médio porte. Nas áreas de melhor potencial, persiste a agricultura de subsistência.

Mesmos sabendo que os dados estatísticos da agropecuária foram resultados de censos anteriores, sempre ficou muito claro que a região nordeste tem uma vocação natural para a criação de caprinos. E um dos fatores que faz com que este dado seja positivo é a fácil adaptação de caprinos às condições ecológicas do semi-árido, por apresentar características peculiares em relação ao clima e a vegetação, que fazem com que o desenvolvimento da espécie se torne positivo.

As projeções para o crescimento do rebanho caprino no nordeste tem apontado para um crescimento de 2% ao ano, conforme mostra a Tabela 1:

Tabela 1 - Efetivo de caprinos no nordeste

Estado	Efetivo	%
1. Bahia.....	4.446.155	42,13
2. Piauí.....	2.233.609	21,16
3. Pernambuco.....	1.351.406	12,80
4. Ceará.....	1.149.543	10,89
5. Maranhão.....	511.499	4,86
6. Paraíba.....	481.298	4,56
7. Rio Grande do Norte.....		
8. Alagoas.....	285.428	2,70
9. Sergipe.....	72.235	0,68
	23.186	0,22
NORDESTE.....	10.554.359	100,0

Fonte: FNP – ANUALPEC (1998)

É de grande relevância o trabalho de melhoramento do rebanho caprino nordestino através de raças “exóticas”, mas é de fundamental importância que se mantenham os projetos de preservação dos tipos nativos, que apresentam características genética-produtivas com o da região. Para que isso se torne possível, é importante que se tenha um manejo alimentar bem planejado, atrelado a um programa de melhoramento genético destas raças, reduzir a idade de abate dos animais e, também, incrementar a taxa de desfrute do rebanho, gerando uma maior produção de carne para o mercado consumidor.

Com a abertura do mercado internacional, a cadeia produtiva começou a se especializar aos diversos segmentos produtivos do rebanho caprino, que vai desde os geradores de recursos, a fabricação de insumos, unidades produtivas, processamento de produto gerado, comercialização de produto e sub-produto e o consumidor final. Assim se caracteriza a caprinocultura de corte.

Os elementos da agroindústria são bem diversificados, com uma potencialidade enorme na produção de produtos e sub-produtos pouco explorados. Eles vão desde alimentos como: carne e derivados, leite e derivados, à produtos oriundos da pele, pêlo, sêmen (reprodutores e matrizes), e sub-produtos como: esterco, urina, osso e chifre.

Diante dessas informações, encontrou-se um bom motivo para propor o desenvolvimento de um produto que viesse a facilitar o controle de rebanho caprino e ovino, utilizando tecnologia web, fazendo com que os criadores possam ter mais informações, precisão nos dados, relatórios que possam ajudar a tomar decisões, o manejo alimentar dos animais, controle na parição, controle de vacinação do rebanho, um controle completo do rebanho. Também será necessário um registro de pessoas, que trabalham na propriedade onde está o rebanho, e da propriedade em si, onde se pode ter um controle das melhorias desenvolvidas.

4.1 MODELO DO SISTEMA PROPOSTO

Para que se tivesse a visão geral do mercado e dos problemas existentes e se chegasse a uma proposta concreta para desenvolvimento de uma aplicação para ajudar a evolução no rebanho caprino e ovino, foi necessária a ajuda de especialistas na área de caprinocultura. Com a ajuda de tais especialistas foi possível identificar os cinco módulos que a aplicação deveria envolver para realizar um controle total, tanto dos animais, quanto de propriedades, manejo alimentar e controle de parição. Para cada um desses módulos, foi necessário realizar modelos UML para poder iniciar o desenvolvimento do sistema. Tais modelos serão detalhados nas sessões a seguir.

4.1.1 Controle Animal

O controle de animal é feito a partir de uma série de informações que envolvem desde a data de nascimento do animal até o tipo de chifre que ele possui. Todas as informações são importantes para que se tenha um controle completo dos animais e para que seja possível obter informações para que a continuação da sua espécie possa acontecer. O diagrama de classes para o módulo de controle animal pode ser visto na Figura 3.

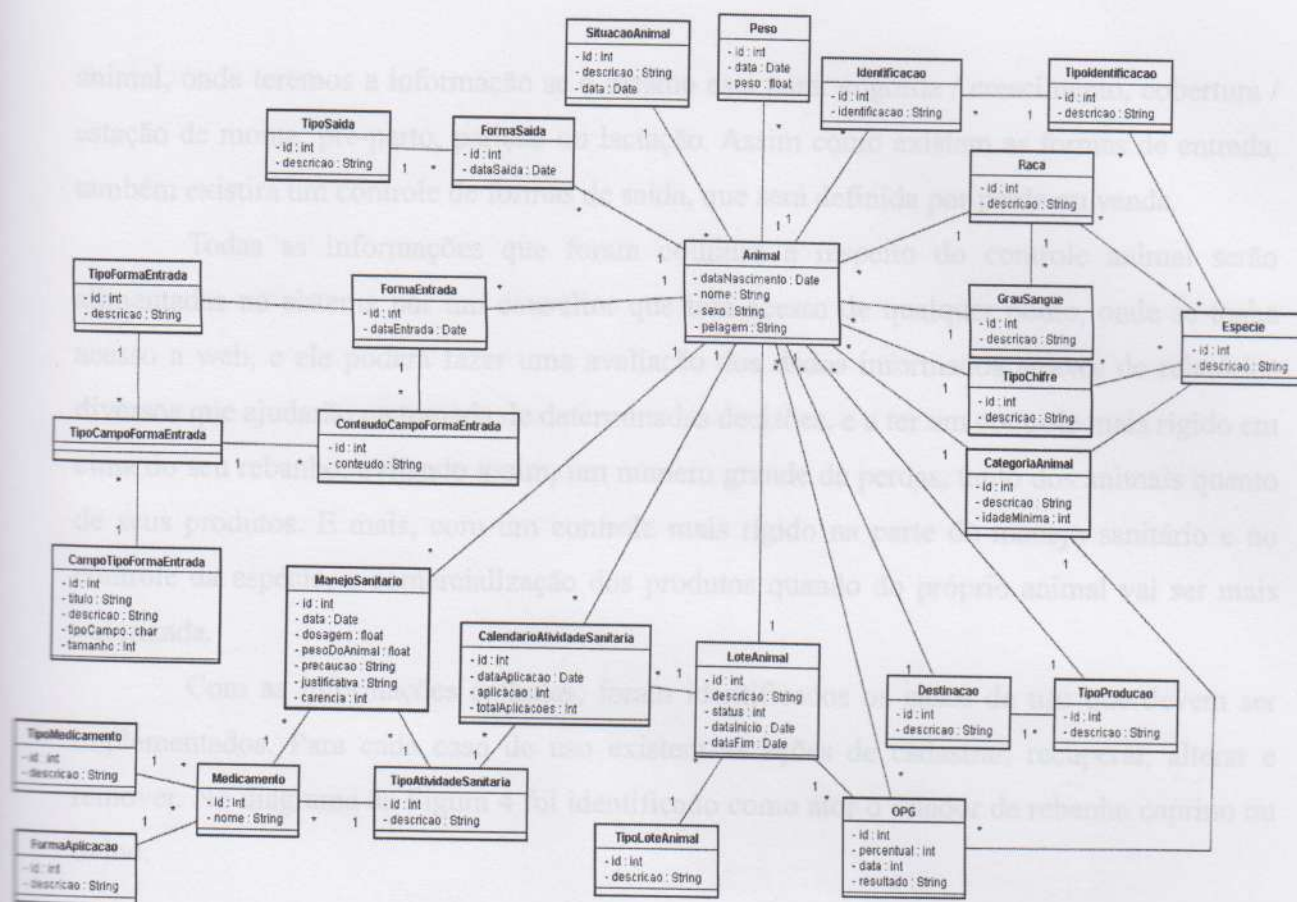


Figura 3 – Diagrama de classe do Controle Animal
Fonte: Dados do Pesquisador (2007)

O diagrama de controle de animais mostra informações referentes a data de nascimento, nome, sexo e a pelagem do animal. Os animais têm várias formas de entrada na propriedade, como: nascimento, doação ou compra, essa informação possui a data em que o animal entrou na propriedade. Como também, o manejo sanitário, que é desenvolvido a partir de medicamento ou por um tipo de atividade sanitária, como: vermifugação, vacinação ou por medicamentos, que por sua vez tem um tipo e uma forma de aplicação. Existe um calendário de atividades sanitárias onde está diretamente relacionado ao animal, ao tipo de atividade sanitária e ao lote animal, que é a separação de animais em apriscos (tipo de curral para o rebanho ser separado). Este lote animal possui um tipo de lote e OPG, que é um exame feito em cada animal. Os animais terão uma destinação, onde indica se o mesmo vai ser de corte ou de produção. E o tipo de produção informa se é de leite, carne ou pele. Os animais possuem uma categoria, um tipo de chifre, um grau de sangue, onde é definido se é PO (puro de origem) ou mestiço, e uma raça. Todas essas informações compõem a espécie. Cada animal possui uma identificação, que pode ser uma tatuagem, um brinco, um chip, e se for ovino, a identificação vai de acordo com um corte feito na orelha. Essa identificação também se relaciona com a espécie. O peso do animal também será controlado, assim como a situação do

animal, onde teremos a informação se o mesmo está para: engorda / crescimento, cobertura / estação de monta, pré-parto, parição ou lactação. Assim como existem as formas de entrada, também existirá um controle de formas de saída, que será definida por perda ou venda.

Todas as informações que foram colhidas a respeito do controle animal serão alimentadas no sistema por um consultor que terá acesso de qualquer ponto, onde se tenha acesso a web, e ele poderá fazer uma avaliação dos dados informados através de relatórios diversos que ajudarão na tomada de determinadas decisões, e a ter um controle mais rígido em cima do seu rebanho, evitando assim, um número grande de perdas, tanto dos animais quanto de seus produtos. E mais, com um controle mais rígido na parte do manejo sanitário e no controle da espécie, a comercialização dos produtos quando do próprio animal vai ser mais valorizada.

Com as informações colhidas, foram identificados os casos de uso que devem ser implementados. Para cada caso de uso existem as ações de cadastrar, recuperar, alterar e remover. No diagrama da Figura 4 foi identificado como ator o criador de rebanho caprino ou ovino.

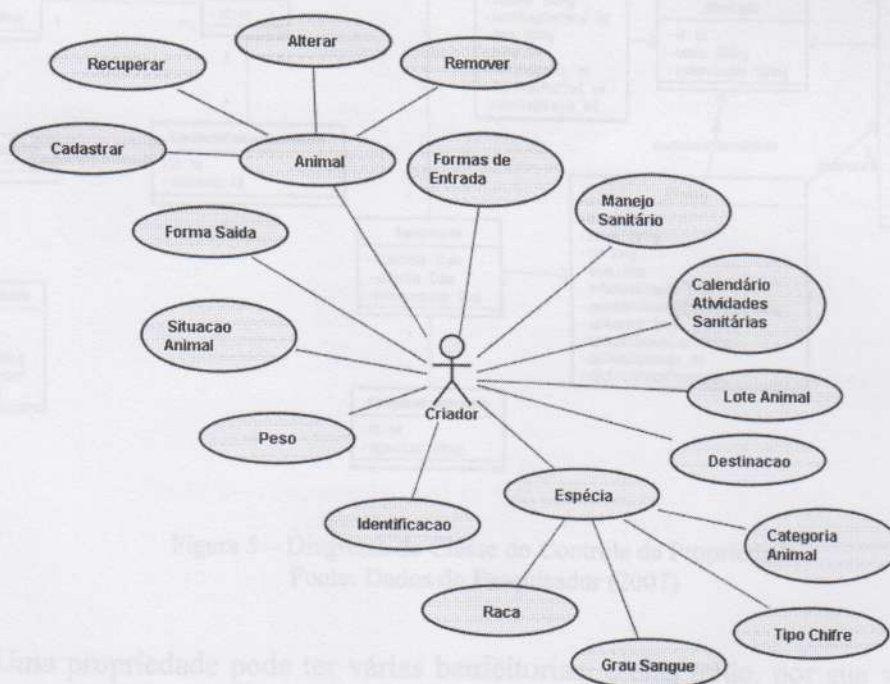


Figura 4– Diagrama de Caso de Uso do Controle Animal
Fonte: Dados do Pesquisador (2007)

4.1.2 Propriedade

O controle de propriedade é muito importante principalmente quando o produtor investe em benfeitorias, fazendo desta forma com que seu negócio possa vir a crescer mais e tenha uma infra-estrutura que suporte maiores investimentos na área animal. Uma das grandes reformas que se pode fazer em uma fazenda, na região do sertão nordestino, onde se cria caprinos e ovinos é a construção de açudes ou até barragens, já que a seca predomina na região. O diagrama da Figura 5 reflete a informação que é necessária para fazer o controle de propriedade.

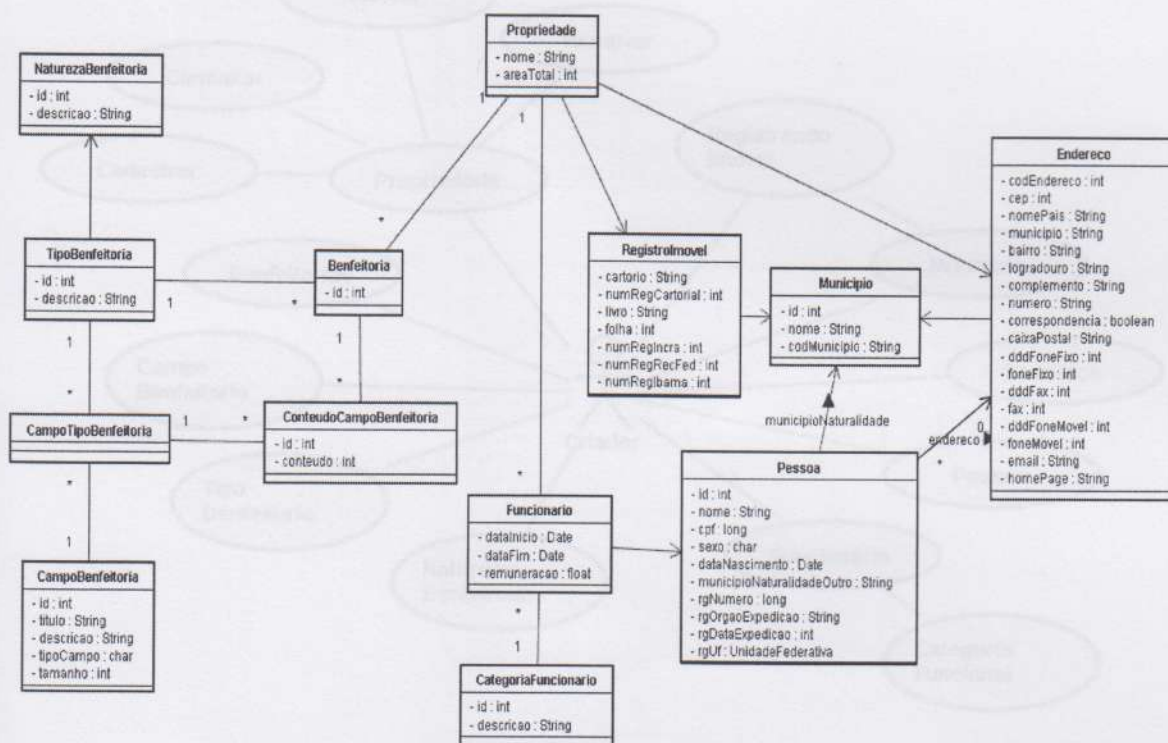


Figura 5 – Diagrama de Classe do Controle da Propriedade
Fonte: Dados do Pesquisador (2007)

Uma propriedade pode ter várias benfeitorias, e elas terão, por sua vez, um tipo de benfeitoria onde o usuário poderá descrever, detalhadamente, tudo o que fez para melhorar a terra onde cria os animais. Além disso, pode ser informado em qual campo foi feito o melhoramento, se na parte de infra-estrutura, parte de eletrificação, irrigação, melhoramento de aprisco, construção de açude, dentre outros. Informando o campo, informa-se, também, o tipo e a natureza da benfeitoria.

Uma propriedade possui vários funcionários, que tem suas divisões de tarefas

definidas, gerando, dessa forma, uma categoria de definição para cada um. Como qualquer imóvel, uma fazenda tem o seu registro em cartório, no município em que é sediado. Desta forma, terá um endereço.

Como é necessário utilizar a classe Pessoa para fazer o cadastro de todos os envolvidos no processo de utilização do software, foi necessário utilizar esta classe neste diagrama, pois será preciso cadastrar os funcionários da fazenda.

Deve-se identificar, agora, casos de uso que serão de grande importância para o controle de propriedades. Para cada caso de uso teremos, assim como no controle animal, as ações de cadastrar, alterar, consultar e remover, como pode ser visto na figura 6.

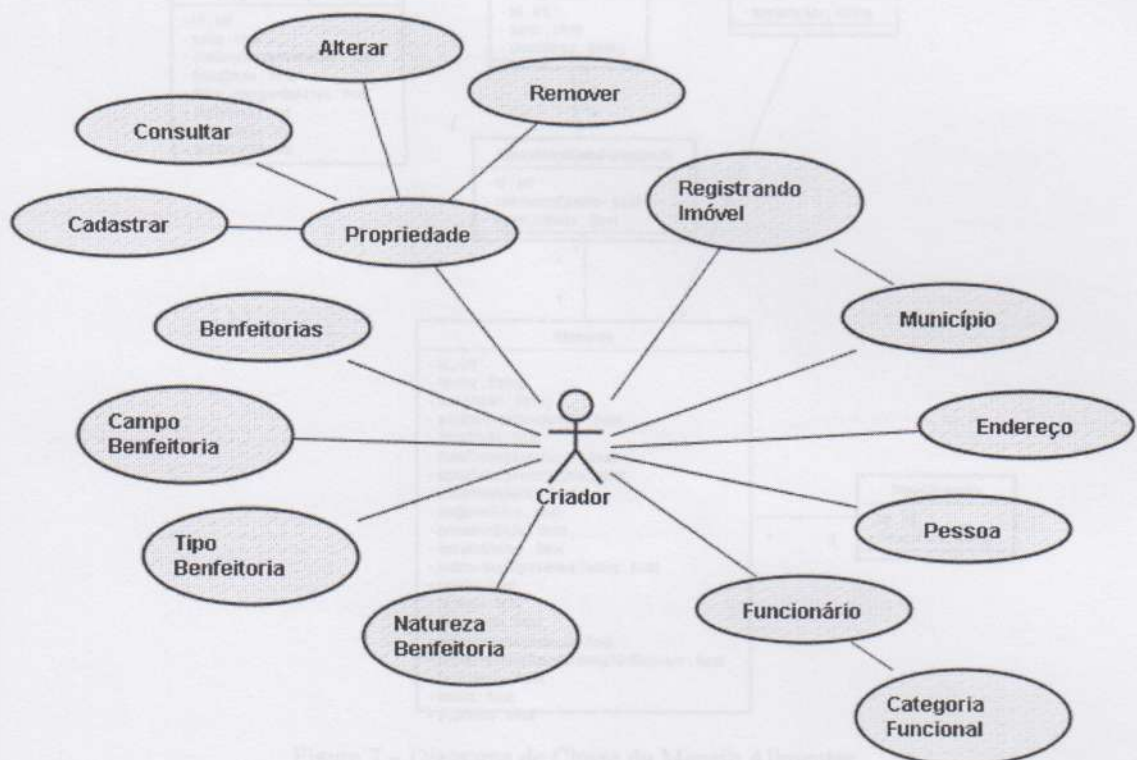


Figura 6 – Diagrama de Caso de Uso do Controle da Propriedade
Fonte: Dados do Pesquisador (2007)

4.1.3 Manejo Alimentar

O manejo alimentar no sertão nordestino tem características bem definidas. No período de chuva existe a abundância em plantas forrageiras, ou seja, plantas de pastagem, onde os animais podem se alimentar à vontade, fazendo, desta forma, uma dieta rica em nutrientes, onde os fazendeiros utilizam cocho somente para suplementação. No período de

seca, os animais necessitam de alimentação diferenciada, tanto as matrizes quanto as crias. Desta forma, utiliza-se culturas para uso da seca como: mandioca, palma forrageira e cana-de-açúcar, que são totalmente viáveis para o criador.

Diante das informações colhidas, foi possível chegar do diagrama exibido na Figura 7.

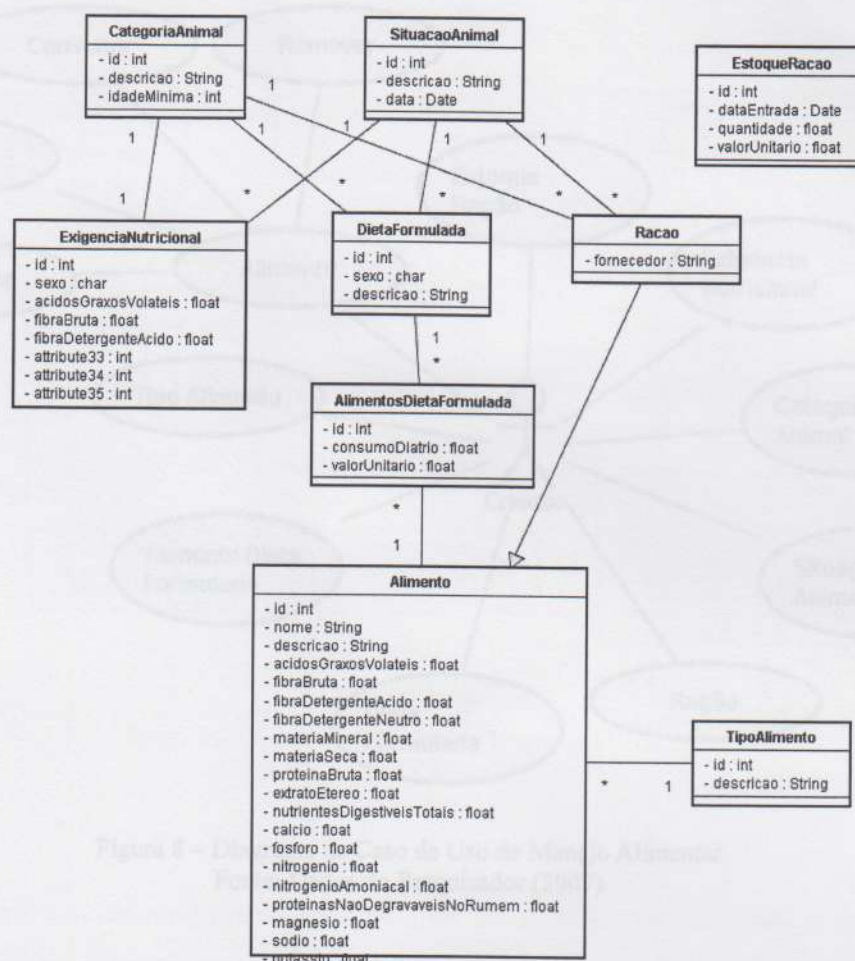


Figura 7 – Diagrama de Classe do Manejo Alimentar
Fonte: Dados do Pesquisador (2007)

Todo alimento vai possuir minerais, cálcio, fósforo, sódio, dentre outras substâncias que vão compor o mesmo. Eles possuem um tipo que será determinante para o consumo animal já que, conforme a situação do animal (amamentando, para corte, reprodutor) ele terá uma alimentação balanceada, com uma dieta formulada que constará do consumo diário e o valor unitário consumido. Isso faz com que possa existir algumas dietas formuladas a partir daquele alimento, que pode ser também uma ração. Esta ração terá um fornecedor específico que possa acompanhar, inclusive na indicação, na alimentação animal. Uma dieta formulada vai conter vários alimentos e vai ser conforme uma categoria de animal, ou seja, se é para burrego, carneiro, cabra, e conforme a situação do animal: se ele é de cobertura / estação de

monta, pré-parto, parição ou lactação. A exigência nutricional varia conforme a situação do animal e a categoria do mesmo. Pode-se ter, também, um estoque de ração que será controlado pela data de entrada, a quantidade em estoque e o valor unitário.

Diante das informações colhidas, foram identificados os casos de uso da Figura 8.

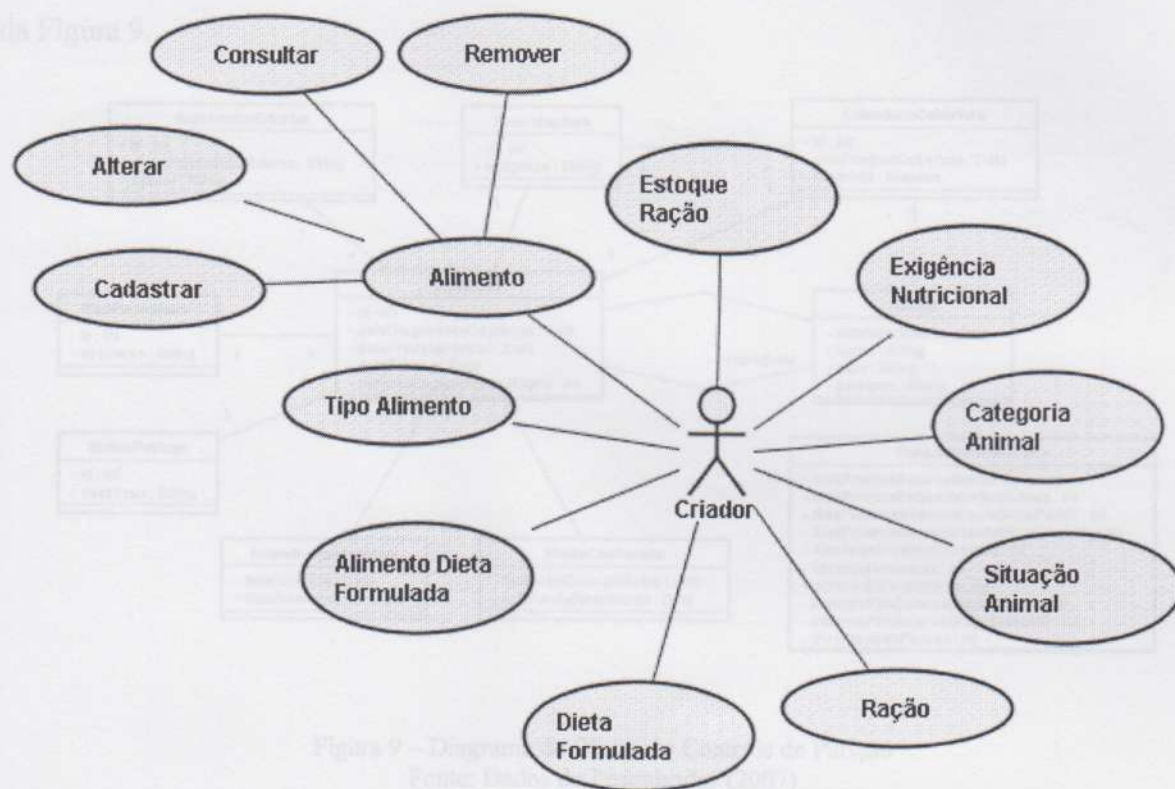


Figura 8 – Diagrama de Caso de Uso do Manejo Alimentar

Fonte: Dados do Pesquisador (2007)

4.1.4 Controle de Parição

Para que possa ser feito um bom controle de parição se faz necessário saber alguns detalhes sobre a puberdade dos caprinos. Eles se tornam aptos a reprodução quando chegam aproximadamente aos 4 meses, e acontece a primeira ovulação nas fêmeas e a presença de espermatozóide nos machos. Contudo, este não é um período bom para iniciar a vida produtiva, pois não tem desenvolvimento corporal suficiente. Nesse tempo eles podem chegar a, no máximo 50% do peso de um animal adulto. O ideal é que se tenha início quando o animal chega aos 7 ou 8 meses, atingindo assim cerca de 70% do peso de um animal adulto.

O controle de parição se torna bastante interessante e importante para um criador,

pois, é a partir destes dados que ele poderá já começar a selecionar os animais para reprodução, para corte, pele, leite. Como também, o produtor poderá escolher, através do tipo de cobertura, uma melhoria significativa para o seu rebanho, a começar pela raça que deseja criar.

Através de algumas informações colhidas, foi possível chegar ao diagrama de classes da Figura 9.

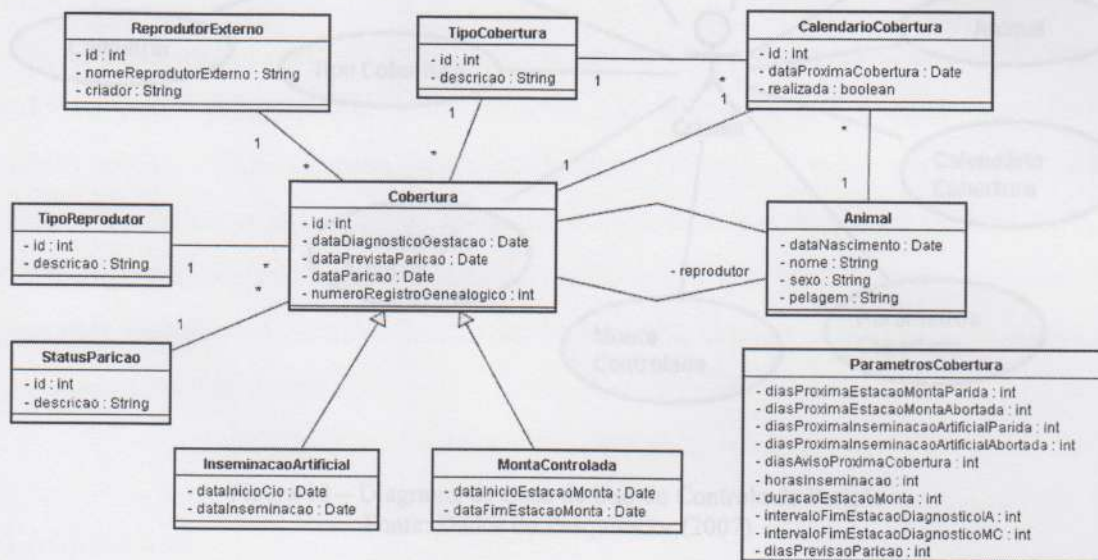


Figura 9 – Diagrama de Classe do Controle de Parição

Fonte: Dados do Pesquisador (2007)

Na cobertura do animal consta a data do diagnóstico da gestação, data prevista da parição, data da parição e o número do registro genealógico, e pode ser feita por inseminação artificial, constando a data início do cio e a data da inseminação, ou por monta controlada, informando a data início e a final da estação de monta, o que caracteriza o tipo de cobertura. Este tipo de cobertura poderá ter vários calendários, que por sua vez terá um animal vinculado a ele. Este animal tanto poderá ser o reprodutor, como também o que irá ser coberto, ou seja, a fêmea. A cobertura terá um status de parição onde será informado se o animal pariu ou abortou, terá o tipo de reprodutor, e um reprodutor externo. Se acontecer este último caso, será informado o nome do animal e o criador. Também vai existir uma tabela onde serão feitos os controles dos parâmetros de cobertura como: dias da próxima estação de monta parida e abortada, dias da próxima inseminação artificial, hora da inseminação, dentre outras informações.

Os casos de uso do controle de parição estão identificados na Figura 10.

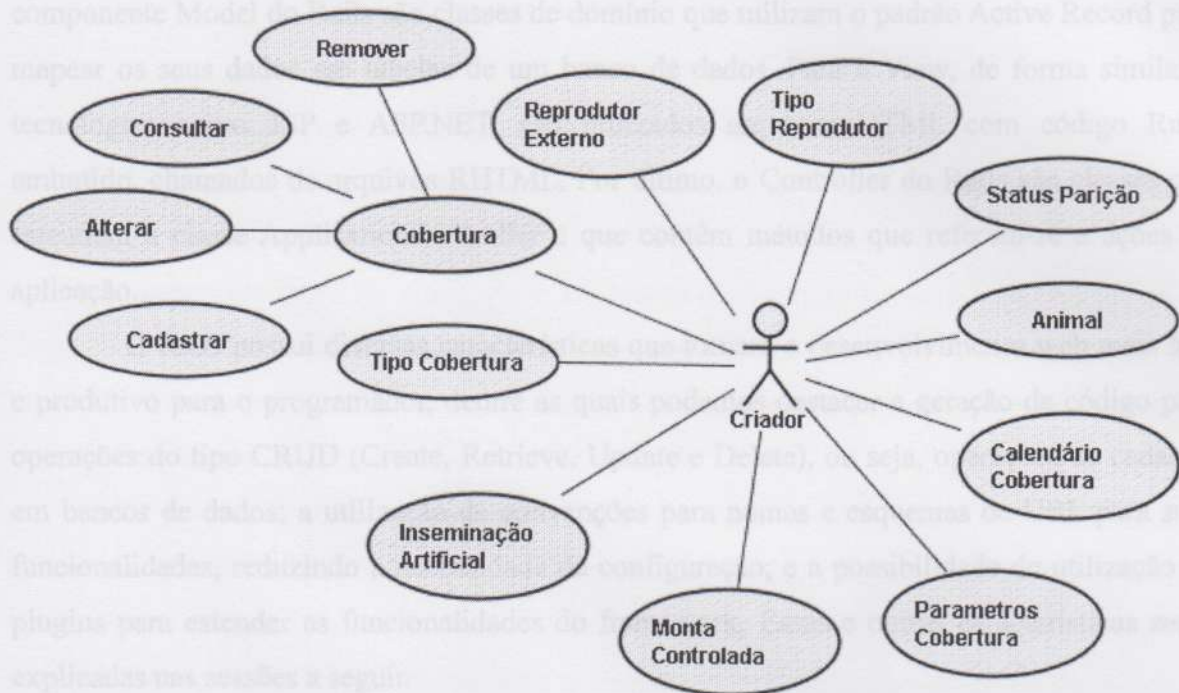


Figura 10 – Diagrama de Caso de Uso do Controle de Parição
Fonte: Dados do Pesquisador (2007)

4.2 IMPLEMENTAÇÃO COM RUBY ON RAILS

Ruby on Rails ou, simplesmente, Rails, é um framework open source, distribuído sob a licença MIT (1988), para desenvolvimento de aplicações web com a linguagem Ruby. Foi criado em 2004 por David Heinemeier Hansson, programador de uma empresa de consultoria e desenvolvimento de aplicações web chamada 37Signals. O Rails foi inicialmente utilizado para a construção do sistema Basecamp para gerenciamento de projetos. Segundo Hansson (2007):

No início, não havia Rails. existia apenas o Basecamp. Após trabalhar no Basecamp por um tempo, eu vi a possibilidade de dar a todas as peças genéricas uma vida própria. Mas mesmo assim, continuei trabalhando no Basecamp primeiro. Isso significa que todas as funcionalidades do Rails foram extraídas de uma aplicação real e não de uma fantasia do tipo 'alguma coisa que alguém pode precisar um dia', que é tão predominante no desenvolvimento de frameworks.

As aplicações web criadas com o Ruby on Rails utilizam o padrão MVC. O componente Model do Rails são classes de domínio que utilizam o padrão Active Record para mapear os seus dados em tabelas de um banco de dados. Para a View, de forma similar a tecnologias como JSP e ASP.NET, são utilizados arquivos HTML com código Ruby embutido, chamados de arquivos RHTML. Por último, o Controller do Rails são classes que estendem a classe ApplicationController e que contêm métodos que referem-se a ações da aplicação.

O Rails possui diversas características que tornam o desenvolvimento web mais ágil e produtivo para o programador, dentre as quais podemos destacar a geração de código para operações do tipo CRUD (Create, Retrieve, Update e Delete), ou seja, operações de cadastro em bancos de dados; a utilização de convenções para nomes e esquemas de URL para suas funcionalidades, reduzindo a necessidade de configuração; e a possibilidade de utilização de plugins para estender as funcionalidades do framework. Essas e outras características serão explicadas nas sessões a seguir.

4.2.1 Arquitetura

O Ruby on Rails é formado por outros cinco frameworks, que são: Active Record, Action Pack, Action Mailer, Active Support e o Active Web Services. Todos são instalados automaticamente quando se instala o gem do Rails. A configuração do Rails é praticamente baseada em convenções, o que facilita a configuração do banco de dados.

O Don't Repeat Yourself (DRY) é um dos pontos fortes do Rails porque é um conceito que existe na definição de nomes, propriedades e códigos somente em um lugar, e poder aproveitar em outras partes do código. As propriedades e os métodos que são necessários são introduzidos nas classes através de funcionalidades da linguagem. Quando se altera o banco de dados, tudo o que se baseia nessas informações são atualizadas automaticamente.

```
class Funcionario < ActiveRecord::Base
  belongs_to :departamento
end
```

Código 30 - Exemplo de classe Funcionario
Fonte: Dados do Pesquisador (2007)

4.2.2 Active Record

O framework para mapeamento objeto relacional (ORM, do inglês Object Relational Mapping) do Rails é chamado Active Record e é baseado no design pattern de mesmo nome. Segundo Martin Fowler (2004), O Active Record é “um objeto que envolve um registro numa tabela de banco de dados ou numa view, encapsula o acesso ao banco e adiciona lógica de domínio nos dados”. No Rails, cada objeto de domínio estende ActiveRecord::Base, que prevê as operações de CRUD. Neste trabalho, todas as classes existentes nos diagramas de classe apresentados anteriormente são Active Records.

O Active Record não requer arquivo de mapeamento, como o Hibernate. Um desenvolvedor trabalhando com Active Record não precisa dos métodos getters e setters, nem mesmo as propriedades de classe. Através de uma análise léxica, o Active Record é capaz de determinar que uma classe será mapeada em uma determinada tabela do banco de dados. Usando uma combinação de Ruby e metaprogramação, as colunas das tabelas tornam-se propriedades do objeto.

Os Códigos exibidos, 29 e 30, mostram um exemplo de mapeamento objeto-relacional com Rails. No Código 31, a classe “Propriedade” vai ser, por convenção, associada à tabela “propriedades”, enquanto que, no Código 30, a classe “Funcionario” será associada à tabela “funcionarios”. Existe ainda no código das classes as funções `has_many` e `belongs_to`, que indicam o relacionamento entre as classes. De acordo com tais funções, a classe Propriedade possui uma associação “um para muitos” com a classe Funcionario, enquanto que a classe Funcionario possui uma associação “muitos para um” com a classe Propriedade.

```
class Propriedade < ActiveRecord::Base
  has_many :funcionarios
end
```

Código 29 – Exemplo de classe Propriedade
Fonte: Dados do Pesquisador (2007)

```
class Funcionario < ActiveRecord::Base
  belongs_to :propriedade
end
```

Código 30 – Exemplo de classe Funcionário
Fonte: Dados do Pesquisador (2007)

Em Active Record, os próprios objetos são responsáveis por serem persistidos no banco de dados, por realizar atualizações, remoções e buscas. Na listagem 4.3 é possível ver um exemplo desse comportamento.

```
P = Propriedade.find(1) # Busca propriedade com id 1
p.nome = "Nome alterado" # Altera o valor do atributo nome
p.save # Persiste a atualização no banco de dados
```

Código 31 – Exemplo de objeto responsável por sua persistência

Fonte: Dados do Pesquisador (2007)

4.2.3 Migrations

Outra tecnologia utilizada no desenvolvimento deste trabalho foram as Migrations. Migrations é um mecanismo para auxiliar a criação das tabelas e a manutenção do esquema do banco de dados a partir de um conjunto de comandos Ruby.

É mais vantajoso utilizar Migrations que alterar o esquema diretamente no banco, pois além de a sintaxe ser mais enxuta, utilizando a mesma sintaxe podemos criar as tabelas em vários tipos de banco diferentes. Além disso, o Migrations suporta o desenvolvimento incremental do banco de dados e o controle de versões do esquema. A qualquer momento é possível mudar um esquema para uma versão anterior ou posterior, conforme as necessidades do desenvolvedor.

Um exemplo de utilização de Migrations pode ser visto no Código 32, na qual é mostrado o código Ruby para criar uma nova tabela no banco de dados.

```
class AdicionarTabelaFuncionario < ActiveRecord::Migration
  def self.up
    create_table :funcionarios do |t|
      t.column :id, :integer
      t.column :data_inicio, :date
      t.column :data_fim, :date
      t.column :pessoa_id, :integer
      t.column :propriedade_id, :integer
      t.column :categoria_funcionario_id, :integer
    end
  end

  def self.down
    drop_table :funcionarios
  end
end
```

Código 32 – Migration para a tabela funcionários

Fonte: Dados do Pesquisador (2007)

4.2.4 Scaffolding

Scaffolding é um mecanismo utilizado pelo Rails que ajuda na geração de CRUDs (Create, Retrieve, Update e Delete) de forma rápida. Todas as operações básicas de um sistema, criação de interfaces, manipulação de dados, e que pode apoiar no desenvolvimento, até que sejam inseridas regras.

Para criar um scaffold, deve-se utilizar o comando `script/generate scaffold nome_do_dominio`. Será criado, com isso, o controller, a classe de domínio, as views e os testes para a classe de domínio informada. Se desejado, pode-se criar apenas uma das entidades listadas anteriormente. Para criar apenas um controller, pode-se usar `script/generate controller nome_do_controller`. O Código 33 mostra um exemplo de criação de controller.

```
camilo@sagan:~/projetos/acosc$ script/generate controller animais
exists  app/controllers/
exists  app/helpers/
create  app/views/contexts
exists  test/functional/
create  app/controllers/animais_controller.rb
create  test/functional/animais_controller_test.rb
create  app/helpers/animais_helper.rb
```

Código 33 – Exemplo de criação de um controller no Scaffolding

Fonte: Dados do Pesquisador (2007)

Resumindo, scaffolding é como o Rails chama o mecanismo de geração de cadastros para as entidades de um sistema. Quando é criado um scaffold para uma entidade, o Rails gera todo o código para criar, remover, listar, editar e visualizar.

4.2.5 Fixtures

Fixtures é uma palavra para “amostra de dados”. Com fixtures é permitido que os testes de banco de dados sejam populados com base predefinida antes de executar os testes. Fixtures tem base de dados independentes e assume um dos dois formatos: YAML ou CSV.

4.2.6 Plugins

Os plugins são bibliotecas escritas pelo programador específicas para o Rails. É uma ótima forma para possibilitar a reutilização do pacote ou de um conjunto de funcionalidades. Os plugins estão instalados diretamente em um projeto feito em Rails. Isso faz com que eles sejam mais fáceis de serem implementados remotamente, juntamente com toda uma aplicação.

Existem diversos plugins publicados que podem ser utilizados para acrescentar funcionalidades a uma aplicação feita com Rails. Neste trabalho foram usados os plugins Brazilian Rails, que acrescenta funcionalidades específicas do Brasil, como validação de CPF, impressão de valores monetários por extenso, feriados nacionais, etc; Squirrel, que acrescenta ao Active Record novas formas de se realizar buscas de dados; e o Userstamp, que possibilita que o Active Record registre automaticamente os dados dos usuários que criaram ou alteraram um determinado registro em uma tabela.

Para instalar um plugin em um aplicativo Rails deve-se, utilizando um aplicativo de terminal, acessar o diretório do aplicativo e chamar `script/plugin install nome_do_plugin`, conforme pode ser visto no Código 34.

```
# Instalar um plugin pelo nome
./script/plugin install calendar_helper

# Instalar a partir de uma URL
./script/plugin install http://topfunky.net/svn/plugins/calendar_helper
```

Código 34 – Exemplo de instalação de plugin no Rails

Fonte: Dados do Pesquisador (2007)

Se o script não encontrar automaticamente o plugin, pode ser necessário executar a ação `discover`, que atualiza os repositórios de plugins do Rails, acrescentando novos repositórios e atualizando os conhecidos. A utilização dessa ação pode ser vista no Código 35:

```
# Importar novos repositórios de plugins
./script/plugin discover
```

Código 35 – Execução da ação `discover`

Fonte: Dados do Pesquisador (2007)

Através da utilização dos recursos apresentados ao longo deste capítulo, a ferramenta para controle de rebanhos de caprinos e ovinos está sendo construída de forma ágil e satisfatória. O capítulo a seguir mostra as conclusões que puderam ser tiradas a partir da utilização de Ruby e Rails no projeto, e as perspectivas para projetos futuros.

5 CONCLUSÕES

Estudar a pecuária do Brasil, em especial a do Rio Grande do Norte, foi um grande desafio para que este trabalho tivesse sucesso. Mergulhar no mundo dos criadores, veterinários, zootecnistas e consultores da área, abriu muito o leque para que fosse possível aplicar conceitos de uma tecnologia que tivesse por finalidade facilitar o trabalho e colaborar para o desenvolvimento da cultura de caprinos e ovinos.

Escolher a tecnologia correta para desenvolver aplicações não é fácil, chegando até a ser crítico pelo fato de ter no sistema proposto uma ferramenta que surge para facilitar o controle de rebanho caprino e ovino, principalmente quando o usuário final não tem um domínio de sistemas ou tecnologias. Determinar a arquitetura correta será a porta de entrada para um equilíbrio adequado entre o entendimento dos usuários, facilidades de desenvolvimento, testes e utilização do software a ser desenvolvido. Isto parte, também, dos usuários que se encontram mais exigentes, querendo que exista respaldo na parte que lhes cabe.

Desenvolver um software, para ser utilizado em ambiente web, é a solução mais prática para que os criadores tenham um maior controle do seu negócio. As informações chegando através de um sistema, atualizado diretamente por quem está trabalhando em uma propriedade, fazendo levantamento de informações, tanto de animais quando da própria terra, faz com que o criador se sinta bem e saiba onde pode ser o foco do seu investimento.

Por isso, foi identificado que a tecnologia a ser utilizada seria de uma linguagem dinâmica, que pode ser utilizada em uma máquina virtual Java, e ficou resolvido a utilização do JRuby pelo fato de ter a característica dos seus scripts rodarem na JVM. Ela apresenta conceitos de orientação a objetos, tem todas as vantagens que o Ruby oferece, inclusive na utilização do núcleo das bibliotecas, a sintaxe e as bibliotecas padrões. O framework a ser utilizado é o Ruby on Rails, por ser MVC e ter as diversas facilidades, tanto no tocante a desenvolvimento como na sua manutenção, justamente por ter camadas bem definidas em apresentação, aplicação e dados. É importante também saber utilizar cada um dos frameworks que o próprio Ruby on Rails oferece.

Fazer também a escolha correta para o banco de dados a ser utilizado é de grande importância, porque os dados são o que há de mais importante para os criadores e consultores. E o utilizado será um orientado a objetos pela facilidade que vai existir a criação do banco determinado através de uma classe e os dados armazenados, na realidade, serão objetos.

O propósito é de dar continuidade a este trabalho continuando a implementação do sistema utilizando as tecnologias que foram definidas para que o projeto saia totalmente do papel e chegue a quem vai trabalhar diretamente com ele, que são os consultores e criadores de rebanhos caprinos e ovinos.

Como perspectivas futuras, espera-se poder agrupar o código deste projeto que pode ser reutilizado em outros sistemas e criar um plugin para o Rails, de forma a agilizar o desenvolvimento de projetos futuros.

ALMEIDA, J. C. S. UML: guia do usuário. Rio de Janeiro: Elsevier, 2005.

BOND, M. et al. Aprenda J2EE com EJB, JSP, Servlets, JNDI, JDBC e XML. São Paulo: Makron Books, 2003.

BRASIL, Ministério da Educação. O Nordeste do Brasil. Recife, 2003. Disponível em: <http://www.fundaj.gov.br/noticia/servlet/newspage.presentation.NavigationServlet?publicationCode=16&pageCode=310&textCode=987&date=currentDate>. Acessado em: dez 2007.

BRASIL, Ministério do Planejamento, Orçamento e Gestão. Produto da pesquisa municipal - 2004. Disponível em: http://www.bge.com.br/home/presidencia/noticias/noticia_visualiza.php?id_noticia=459&id_pagina=1. Acessado em: dez 2007.

COSTA, R. O A produção de carne caprina na visão do agronegócio. Revista Conhecimento. João Pessoa V. 1 n. 8, p. 95-101 jan./jun. 2003.

EVANS, E. Domain Driven Design: Tackling Complexity into the Heart of the Software. Boston: Addison - Wesley, 2004.

FOWLER, M. Patterns of Enterprise Application Architecture. Boston: Addison - Wesley, 2004.

GAMMA, E. et al. Padrões de projeto: Soluções Reutilizáveis de Software Orientado a Objetos. Porto Alegre: Bookman, 2000.

GUEDES, G. T. A. UML: Uma abordagem prática. São Paulo: Novatec Editora, 2004.

HANSON, D. H. The genesis and benefits of Rails. <http://www.37signals.com/en/posts/660-ask-37signals-the-genesis-and-benefits-of-rails>. Acessado em: dez 2007.

REFERÊNCIAS

- ALMEIDA, R. R. Model View Controller (MVC). Disponível em :
<<http://jacques.dsc.ufcg.edu.br/cursos/map/html/arqu/mvc/mvc.htm>>. Acessdo em: dez 2007.
- BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. UML : guia do usuario. Rio de Janeiro: Elsevier, 2000.
- BOND, M. et al. Aprenda J2EE com EJB, JSP, Servlets, JNDI, JDBC e XML. São Paulo: Makron Books, 2003.
- BRASIL, Ministério da Educação. O Nordeste do Brasil. Recife, 2003. Disponível em:
<<http://www.fundaj.gov.br/notitia/servlet/newstorm.ns.presentation.NavigationServlet?publicationCode=16&pageCode=310&textCode=982&date=currentDate>>. Acessado em: dez 2007.
- BRASIL, Ministério do Planejamento, Orçamento e Gestão. Produção da pecuária municipal – 2004. Disponível em: <http://www.ibge.com.br/home/presidencia/noticias/noticia_visualiza.php?id_noticia=499&id_pagina=1>. Acessado em: dez 2007.
- COSTA, R. G. A produção de carne caprina na visão do agronegócio. Revista Conceitos. João Pessoa v. 1 n. 8, p. 95-101 jan./jun. 2003.
- EVANS, E. Domain Driven Design: Tackling Complexity into the Heart of the Software. Boston: Addison - Wesley, 2004.
- FOWLER, M. Patterns of Enterprise Application Architecture. Boston: Addison - Wesley, 2004.
- GAMMA, E. et al. Padrões de projetos : Soluções Reutilizáveis de Software Orientado a Objetos. Porto Alegre: Bookman, 2000.
- GUEDES, G. T. A. UML : Uma abordagem prática. São Paulo: Novatec Editora, 2004.
- HANSON, D. H. The genesis and benefits of Rails.
<<http://www.37signals.com/svn/posts/660-ask-37signals-the-genesis-and-benefits-of-rails>>. Acessado em: dez 2007.

HILL, D. Escolhendo a arquitetura da camada de apresentação correta. Disponível em: <<http://www.microsoft.com/brasil/msdn/Tecnologias/arquitetura/ChoosingPresentationLayer.msp>>. Acessado em: dez 2007.

LÔBO, R. N. B. Programas de seleção para caprinos e ovinos no Brasil. Disponível em: <http://www.sbpcnet.org.br/livro/57ra/programas/CONF_SIMP/textos/raimundobragalobo.htm>. Acessado em: dez 2007.

SAUVÉ, J. P. Introdução e motivação: arquitetura em n camadas. Disponível em: <<http://www.dsc.ufcg.edu.br/~jacques/cursos/j2ee/html/intro/intro.htm>>. Acessado em: dez 2007.

MEYER, B. Object-Oriented Software Construction. Santa Barbara: Bookstores, 1997.

THOMAS, D, HANSSON, D.H. Agile Web Development with Rails.USA: The Facets of Ruby Series. 2005.

OLIVEIRA, M. O. Como fazer : projetos, relatórios, monografias, dissertações e teses. Rio de Janeiro: Impetus, 2003.

Open Source Initiative - OSI. <<http://www.opensource.org/licenses/mit-license.php>>. MIT License>. Acessado em: dez 2007.

ROCHA, H. J930: GoF Design Patterns em Java. Disponível em: <<http://www.argonavis.com.br/cursos/java/j930/index.html>>. Acessado em: dez 2007.

RODRIGUES, M. T. et al. Capricornius – um sistema de informação para a gestão do agronegócio da caprinocultura. Revista Brasileira de Agroinformática, São Paulo v. 6, n. 2, p.55-66, jul/dez 2004.

VASCONCELOS, V. R.; VIEIRA, L. S. A evolução da caprino - ovinocultura brasileira. Disponível em: <<http://www.fmvz.unesp.br/Informativos/ovinos/utilid11.htm>>. Acessado em: dez 2007.