Projeto Colméia - Zumbido

Leandro Inácio de Oliveira Bororo
Rodrigo Di Lorenzo Lopes
Orientador: Prof. Dr. Fabio Kon

26 de Novembro de 2007

Sumário

1 Introdução

O projeto Colméia surgiu com o intuito de informatizar e integrar as atividades realizadas em uma biblioteca universitária. Utilizando-se de padrões e ferramentas de software livres, o sistema pretende atender às necessidades dos estudantes, funcionários, professores e membros da comunidade externa.

O projeto vem sendo desenvolvido com a linguagem de programação Java e poderá ser acessado através da rede WEB, visando a atender usuários de diferentes plataformas e localidades.

A coordenação do Colméia está a cargo dos professores Eduardo Colli (Coordenador da Comissão de Biblioteca), Fabio Kon e João Eduardo Ferreira (ambos do Departamento de Ciência da Computação). Desde 2002 alunos das disciplinas de Programação Extrema, ministrada pelo professor Fabio Kon, e Laboratório de Programação, ministrada pelo professor João Eduardo Ferreira, trabalham no desenvolvimento do software, que possibilita aos alunos a aplicação de conhecimentos adquiridos nestas disciplinas.

O principal requisito do Colméia é atender as necessidades da biblioteca do IME[1], como Por exemplo, o cadastro de obras existentes no acervo da biblioteca, o cadastro de usuários e o controle de aquisições e empréstimos.

Dentro deste contexto, no início de Março de 2007 durante a disciplina de Programação Extrema, foi oferecido aos alunos, como opção de projeto para aplicação dos métodos da disciplina, o desenvolvimento de um módulo que atendesse ao requisito de exportação de dados bibliográficos existentes no sistema atual da biblioteca do IME.

O projeto foi então iniciado e recebeu o nome de Zumbido. O nome do projeto é uma metáfora que remete ao meio de comunicação entre abelhas, de modo que, assim como as abelhas se comunicam através do som utilizando seu zumbido, o módulo Colméia-Zumbido deve prover um meio e uma linguagem padrão para a comunicação com outros sistemas bibliográficos automatizados.

Um arquivo contendo informações à respeito das obras e exemplares que têm cadastro no sistema atual foi entregue aos alunos que deveriam, a partir de então, desenvolver um módulo que implementasse as seguintes funcionalidades:

- Cadastro de registros de obras, no banco de dados do sistema, descritos no formato MARC, contidos em um arquivo fornecido através da interface WEB do sistema.

- Cadastro de registros de obras, no banco de dados do sistema, descritos

[1] Instituto de Matemática e Estatística da USP

no formato MARC, fornecidos através de uma área de texto da interface WEB do sistema.

- Exportação de dados referentes à obras ou exemplares de obras, selecionadas a partir de algum tipo de filtro, para um arquivo de registros no formato MARC.

- Implementação do protocolo Z39.50

Tendo em vista a implementação das funcionalidades acima, fez-se necessário um estudo do formato MARC e do protocolo Z39.50. Adotou-se como fonte de pesquisa a Internet, uma vez que, esta contém referências para as instituições que mantêm o MARC e o Z39.50 e fornecem material contendo suas especificações.

Pode-se dizer que se denomina MARC o conjunto de formatos padrão para a representação bibliográfica e informações relacionadas que podem ser compreendidos por sistemas computadorizados. Já o protocolo Z39.50 é uma especificação para troca de mensagens entre um cliente e um servidor que habilita o cliente a realizar buscas por registros armazenados em um servidor segundo um determinado critério.

2 O formato MARC

2.1 O que é MARC?

O MARC surgiu em 1968 como resultado de uma série de conferências e experiências lideradas pela LoC[2], que buscava, desde meados de 1950, a automatização de suas operações. MARC é o acrônimo para *MAchine Readable Record*, ou seja, é um formato ou especificação para registros que podem ser interpretados por sistemas computadorizados. A partir de sua formulação inicial, muitas extensões foram criadas para atender requisitos específicos, algumas dessas extensões são o USMARC, utilizado nos Estados Unidos, o CANMARC, utilizado no Canadá, e o MARC21, que surgiu da tentativa de união entre o USMARC e o CANMARC.

O MARC fornece um mecanismo através do qual computadores podem trocar, utilizar e interpretar dados bibliográficos, geralmente é distribuído no formato binário e já é adotado na maioria dos sistemas de catálogo.

[2]Library of Congress - Biblioteca do Congresso Americano

2.2 Estrutura de um registro MARC

Um registro MARC é formado por três elementos básicos:

- Lider (*Leader*): Contém informações que podem ser usadas para o processamento do registro. Consiste de 24 caracteres que determinam individualmente ou em subconjuntos *meta dados* do registro.
- Diretório (*Directory*): Contém o comprimento e a posição de início de todos os campos de dados do material, indexados por MARCadores que os identificam.
- Campos de Variáveis (*VariableFields*): Campos que contém os dados do material, são indexados por marcadores que são armazenados no Diretório e devem, obrigatoriamente, terminar com o caractere ASCII[3] *1E* hexadecimal. Subdividem-se em:
  - Campos de Variáveis de Controle (*VariableControlFields*):
    São campos indexados por marcadores do tipo 00X, não podem ser modificados por indicadores ou conter sub-campos.
  - Campos de Variáveis de Dados (*VariableDataFields*):
    São campos indexados por marcadores do tipo 01X-9XX. Cada um dos índices (marcadores) corresponde a um tipo de informação a respeito do registro, como por exemplo, o autor da obra ou o seu título, e alguns desses campos são passíveis de repetição.

As seções subseqüentes trazem exemplos ilustrativos do Líder e dos Campos de Variáveis, além de explicar sua semântica de maneira mais detalhada.

2.2.1 Líder (*Leader*)

De maneira geral o líder traz meta dados do registro que está sendo processado. Consiste de 24 caracteres que trazem as seguintes informações:

- 00-04 Uma cadeia de caracteres numéricos que especificam o comprimento do registro.
- 05 Indica através de uma letra minúscula, a relação entre o registro e o arquivo. Por exemplo, se o registro é uma correção de um registro pertencente a um arquivo previamente gerado.

[3] American Standard Code for Information Interchange

- 06 Indica através de uma letra minúscula, características e/ou conteúdo do registro. Por exemplo, o caractere f indica que o registro refere-se a um material cartográfico manuscrito.

- 07 Indica através de uma letra minúscula, o nível bibliográfico do material. Por exemplo, se o material é parte de uma monografia ou se o material é parte de uma série de materiais.

- 08 Indica se o registro está arquivado através do caractere 'a' ou não através do caractere ' '.

- 09 Indica se a codificação do registro é do tipo MARC8, através do caractere ' ', ou UCS/Unicode, através do caractere 'a'.

- 10 Indica o número de posições reservadas para indicadores em cada campo de variável.

- 11 Indica o número de posições reservadas para a especificação de um sub-campo.

- 12-16 Uma cadeia de 5 caracteres numéricos que especificam a posição do primeiro caractere do primeiro campo de variáveis.

- 17 Indica através de uma letra minúscula, a forma de descrever o catálogo do material. Por exemplo, o caractere 'i' indica que o catálogo do material foi feito segundo especificações da ISBD[4].

- 19 Indica se é possível gerar, a partir dos campos de variáveis indexados pelo conjunto de marcadores 76X-78X, uma nota contendo informações básicas sobre o material, como Por exemplo, a entrada principal do título.

- 20-23 São quatro caracteres numéricos que indicam a estrutura de cada entrada no diretório. Estes campos correspondem a seguinte cadeia de caracteres '4500'.

Um exemplo em formato de texto de um campo Líder:

$$\underset{\text{marcador}}{LEADER}\ 00733nam\ \ \underset{\text{cadeia de 24 caracteres}}{2200241Ia}\ 4500$$

[4]International Standard Bibliographic Description

Assim, as informações contidas no líder permitem que um programa de computador interprete ou crie um arquivo binário no formato MARC.

2.2.2 Campos de Variáveis (*VariableFields*)

- Variáveis de Controle (*VariableControlFields*)

Campos de Variáveis de Controle contêm, de forma geral, informações fornecidas no momento em que o registro foi gerado. Essas informações podem ser utilizadas em processamentos posteriores do registro. Estes campos contêm números e informações codificadas de controle do registro, fornecidas pela instituição que gerou o arquivo MARC. Os marcadores utilizados variam de 001 a 008 e não pode haver mais de um campo com o mesmo marcador.

Tabela 1: Campos de variável de controle

marcador	Informação do campo
001	Número de controle fornecido pela instituição que gerou o registro
002	Não é mais utilizado
003	Código que identifica a instituição que gerou o registro
004	Não é mais utilizado
005	Contém a data da última transação envolvendo o registro
006	Características adicionais do material
007	Descrição Física
008	Aspectos especiais do material
009	Não é mais utilizado

- Variáveis de Dados (*VariableDataFields*)

Campos de Variáveis de Dados contêm informações à respeito do material registrado. Assim como os campos de variáveis de controle, são identificados por marcadores. Os marcadores para este tipo de campo variam de 01X a 9XX e cada um deles determina um tipo de informação diferente para a obra cujo registro foi feito.

Tabela 1: Exemplos de campos de variável de dados

marcador	Informação do campo
010	LCCN[5]
020	ISBN[6]
100	Entrada principal de nome pessoal (autor)
245	Informação de título (inclui título e sub títulos)
250	Informações a respeito da edição do material
260	Informações a respeito da publicação do material
300	Descrição física do material
440	Série ou coleção a qual o material pertence
520	Anotação ou notas de sumário
650	Assuntos associados
700	Entrada adicional para nome pessoal (autor secundário, tradutor etc)

Os marcadores para este tipo de campo seguem uma regra geral estabelecida em termos de intervalos de marcador, ou seja, determinados intervalos de marcadores são utilizados para armazenamento de informações relacionadas, como pode ser visto na tabela abaixo.

Tabela 3: Regras gerais para marcadores

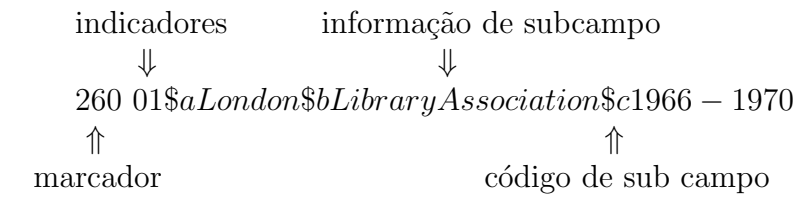
marcadores	Tipo de informação
0XX	Informação, números e códigos de controle
1XX	Entrada principal
2XX	Título, declaração de responsabilidade, edição, e publicação
3XX	Descrição física
4XX	Informação a respeito da série ou coleção
5XX	Notas
6XX	Entradas adicionais de assunto
7XX	Entradas adicionais diferentes de assuntos ou séries
8XX	Entradas adicionais de séries
9XX	Entradas definidas localmente

Os campos de variáveis de dados podem conter *indicadores* ou *códigos de subcampo* que adicionam informação para a interpretação de um campo.

Os indicadores são dois caracteres que ocupam alguma posição dentro do campo de acordo com o que foi especificado no Líder, devem ser compostos por letras minúsculas ou números e seus caracteres devem ser interpretados de maneira independente. Caso um dos indicadores não seja necessário em um determinado campo, sua posição deve conter o caractere ' '. Por motivos didáticos os indicadores também são referenciados como primeiro indicador e segundo indicador. Para cada tipo de campo, indexado por um dos marcadores, os indicadores adicionam um tipo de informação diferente.

Os códigos de subcampo são dois caracteres que servem para dividir a informação contida em um campo em elementos de dados. Cada código de subcampo é definido por dois caracteres, um delimitador de caractere (ASCII *1F*) seguido de uma letra minúscula ou de um número. Um mesmo código de campo pode aparecer em diferentes campos dividindo os campos de diferentes maneiras.

Um exemplo em formato de texto de um campo de variável de dado:



Marcadores, indicadores e códigos de subcampo são chamados, nos materiais de referência disponíveis, de *designadores de conteúdo*. Ao contrário dos marcadores que devem ser determinados no Líder do registro, indicadores e códigos de subcampos dependem do material cujo registro foi gerado e, portanto sua ocorrência é determinada em tempo de processamento.

2.3 Conclusões

O formato MARC estabelece um padrão para armazenamento de informações bibliográficas que possibilita processamento de um registro por sistemas computadorizados. A partir da informação existente no Líder e no Diretório do registro, um programa é capaz de processar um registro, uma vez que, estes campos fornecem dados estruturais importantes para a leitura do arquivo. Além disso, a interpretação dos dados existentes em um registro pode ser realizada através de seus *designadores de conteúdo* e das especificações de campo fornecidas pelas instituições que mantém o MARC.

3 O padrão Z39.30

3.1 O que é Z39.50?

Z39.50 é uma especificação que estabelece um protocolo do tipo cliente-servidor que possibilita consulta e recuperação de informação armazenada em um banco de dados remoto. O Z39.50 foi proposto em 1984 para suprir a necessidade de intercâmbio de informações bibliográficas que existia na época.

Em 1988 a primeira versão foi lançada e em 1990 um grupo chamado ZIG [7] foi criado para cuidar de sua implementação e especificação, o que resultou no lançamento, em 1992, da segunda versão do Z39.50, sendo a esta adicionadas novas características que culminaram no lançamento de uma nova versão em 1995. A versão de 1988 tornou-se obsoleta e as versões de 1992 e 1995 têm diferenças significativas.

O protocolo estabelecido a partir do Z39.50 especifica formatos e procedimentos para a troca de dados entre um cliente e um servidor, permitindo que um cliente realize buscas e consultas, que atendam a um critério por ele estabelecido, em um banco de dados localizado em um servidor remoto. O cliente pode enviar requisitos no papel de usuário, comunicando-se via protocolo Z39.50 com uma aplicação de um servidor.

Além de simplificar o trabalho de busca, o protocolo Z39.50 permite-nos trabalhar com dados volumosos ou seja, com muita informação e a comunicação com múltiplos sistemas de comunicação através de interface única.

De modo geral, o Z39.50 permite que um cliente envie uma requisição de consulta, em um ou mais bancos de dados remotos, e pode ou não receber registros como resposta, de acordo com a parametrização da consulta. O Z39.50 permite ao cliente:

- Conhecer previamente o tipo de arquivo recuperado, o tamanho e o custo, por exemplo, antes de fazer sua solicitação.
- Especificar a quantidade de registros que deseja receber como resposta por requisição.
- Especificar a sintaxe dos registros, como por exemplo, o MARC21 ou o USMARC.
- Especificar um rótulo para o conjunto de registros retornados.

Além disso, do lado do servidor o Z39.50 permite:

- Controlar o acesso de determinados usuários, que podem ser realizadas por meio de autenticação.
- Controlar o acesso a determinados recursos reportando ao usuário o status de sua consulta.
- Suspender processos de consulta correntes.

[7] Z39.50 Implementors Group

3.2 Utilização do protocolo

Como dito anteriormente o protocolo Z39.50 especifica um formato para os dados que devem ser enviados e recebidos, portanto fica a cargo das aplicações, do servidor e do cliente, a tradução e organização dos dados que devem trocados. As mensagens trocadas entre as aplicações são transmitidas em unidades de dados chamadas de PDU[8] que contêm a informação necessária para que a transmissão das mensagens seja realizada.

O processo de troca de mensagens ocorre da seguinte maneira:

- Estabelecimento de uma sessão de comunicação entre as duas aplicações envolvidas no processo, que pode ser feita, por exemplo, através da Internet.
- Envio de uma PDU contendo os parâmetros do serviço isto é, um conjunto de regras que deverão ser respeitadas por ambas as aplicações durante o processo de requisição e resposta. O resultado deste passo é a criação de uma associação Z39.50 (*Z-association*). Neste contexto, o cliente é chamado de origem da associação e o servidor de alvo. Dentro de uma mesma associação não é permitida uma troca de papéis entre as aplicações, ou seja, origem será sempre a aplicação do cliente e o alvo será sempre a aplicação do servidor. Entretanto uma aplicação pode estabelecer inúmeras associações com outra aplicação atuando como alvo ou origem em cada uma delas.
- Envio, a partir da origem, de PDUs contendo consultas para o alvo.
- Envio, a partir do alvo, de uma PDU notificando os resultados obtidos pela consulta para a origem.
- Envio, a partir da origem, de uma PDU contendo uma requisição de resposta ou de exclusão para o alvo.
- Caso a origem tenha enviado uma requisição de resposta ao alvo o resultado da consulta é enviado à origem, caso contrário os resultados da busca são desprezados pelo alvo.

3.2.1 Atributos de busca

Os atributos referentes a uma busca devem pertencer a um dos conjuntos de atributos que fazem parte da especificação do protocolo e, portanto é de responsabilidade da origem da associação atender à especificação. A

[8]Protocol Data Units

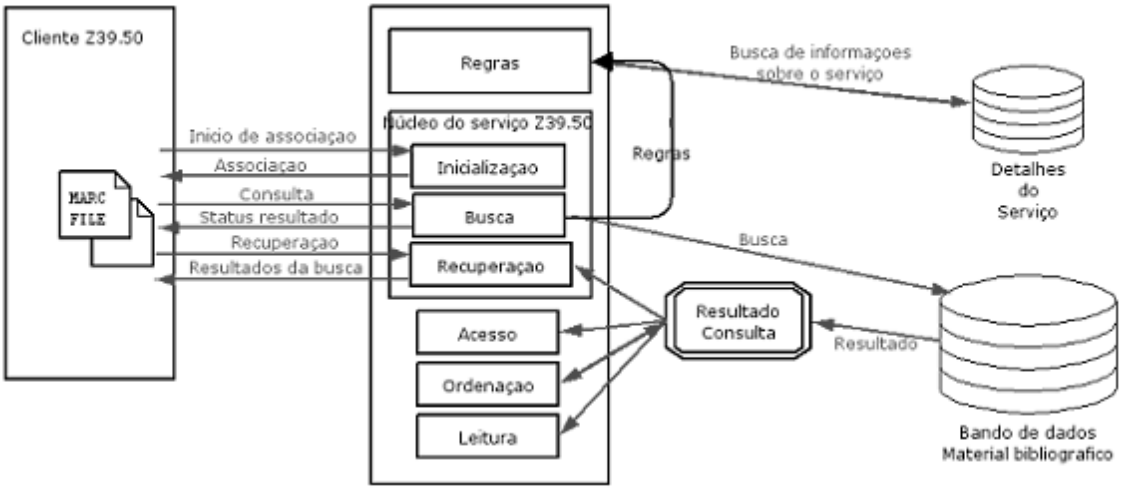


Figura 1: Uma visão geral do processo de troca de mensagens entre origem e alvo

versão 3 do Z39.50 possibilita à origem a utilização de diferentes conjuntos de atributos dentro de uma mesma associação, porém as versões anteriores estabelecem que apenas um conjunto de atributos, escolhido na inicialização da associação, pode ser utilizado. Além disso, as consultas podem ser de dois tipos:

Tipo 0: Requer um acordo privado entre os dois sistemas, que estabelece a forma da consulta e o conteúdo dos resultados que podem ser obtidos.

Tipo 1: Estabelece que as consultas devem ser realizadas em notação polonesa reversa.

3.2.2 Resultados da busca

Os resultados provenientes de uma busca atendem ao formato MARC. O protocolo Z39.50 permite que a origem especifique que extensão do formato MARC deseja-se obter como resposta de uma consulta, algumas das extensões disponíveis são USMARC, UKMARC e o CANMARC. O protocolo prevê também, a possibilidade de cadastro de novas extensões.

Os resultados obtidos em uma busca podem, dependendo das regras de associação estabelecidas entre origem e alvo, conter códigos de diagnóstico que fazem parte da especificação do protocolo, alternativamente também é

possível o cadastro de novos códigos, para que isso seja possível deve haver um acordo entre as aplicações envolvidas.

3.3 Conclusões

O protocolo Z39.50 possibilita a busca por dados existentes em bancos de dados remotos, independentemente da plataforma sob a qual está a aplicação do cliente, no caso quem realiza a consulta, e sob qual está a aplicação do servidor, no caso quem mantém o banco de dados. O processo de consulta ocorre de modo transparente para os usuários das aplicações, por meio de uma interface única.

4 Tecnologias e bibliotecas utilizadas

Esta seção é uma breve introdução às principais tecnologias utilizadas durante a implementação.

4.1 Linguagem Java

Java é uma linguagem de programação desenvolvida inicialmente pela Sun Microsystems, ela é baseada no paradigma de Programação Orientada a Objetos e tem sintaxe similar à sintaxe utilizada em linguagens procedurais. Além da sintaxe, a linguagem Java contém um conjunto de bibliotecas para solução de problemas mais comuns como o tratamento de dispositivos de entrada e saída, estruturas de dados, programação paralela e concorrente, tratamento de erros etc.

Além da linguagem a Sun Microsystems criou uma plataforma composta de um compilador e uma máquina virtual (JVM) para qual a linguagem foi projetada e o Java SE (edição padrão do Java). Essa plataforma permite o isolamento entre o programa e o sistema operacional sobre qual o programa é executado, ao passo que é mais eficiente do que seria com o uso de interpretadores.

4.2 Hibernate

O Hibernate é um projeto aberto desenvolvido sob a plataforma Java, ele provê um serviço de persistência de dados e suporte a consultas em um banco de dados qualquer. O serviço permite que os desenvolvedores implementem suas classes dentro do paradigma da orientação a objetos e realiza a persistência dos objetos utilizando-se de um mapeamento entre as classes da

aplicação e as entidades que as representam no banco de dados. É portanto, um arcabouço para o mapeamento objeto-relacional.

O Hibernate também oferece ao desenvolvedor uma linguagem de consultas chamada HQL que permite ao desenvolvedor lidar apenas com as classes que ele definiu, sem nunca fazer referência a tabelas do banco, não necessita de servidor de aplicação e não exige que o objeto que será persistido possua nenhuma característica especial. Para ser persistido um objeto deve fornecer *getters* e *setters* para seus atributos mapeados e um construtor que aceite como parâmetros estes atributos.

É parte do trabalho do desenvolvedor ao utilizar o arcabouço:

- Definir o mapeamento entre classes e tabelas do banco.
- Definir o mapeamento entre atributos de cada classe e as colunas existentes em suas tabelas correspondentes.
- Mapear as relações existentes entre as classes.

Existem duas maneiras de realizar o mapeamento entre classes e objetos para possibilitar sua persistência: mapeamento com a utilização de arquivos XML e o mapeamento utilizando a tecnologia *Annotations*, disponível a partir da especificação Java 5, que permite a introdução de meta dados nas classes das quais deseja-se realizar persistência.

4.3 Tomcat

O Apache Tomcat é um contêiner para servlets, desenvolvido e mantido pela fundação Apache e é compatível com as tecnologias Java Servlet[9] e Java Server Pages[10], O Tomcat é licenciado sobre os termos de licença Apache e muito utilizado no mercado. Entre os casos de uso do Tomcat temos o projeto JBoss[11] (que vem com o Tomcat como seu web contêiner padrão).

4.4 Biblioteca MARC4j

A biblioteca MARC4j é um projeto aberto que visa a oferecer aos desenvolvedores de diferentes domínios de aplicações uma interface de programação para trabalhar com arquivos binários no formato MARC e arquivos XML[12]

[9] Aplicação que pode ser executada em um servidor WEB
[10] Tecnologia para desenvolvimento de aplicações WEB
[11] Servidor de aplicações WEB
[12] EXtensible Markup Language

no formato MARCXML dentro da plataforma Java. A biblioteca foi implementada para trabalhar com arquivos que atendam ao formato de algumas extensões do MARC, como Por exemplo, o MARC21 e o UNIMARC.

A biblioteca MARC4j inclui:

- Um interface que possibilita o manuseio de um grande número de registros MARC e MARCXML.
- Objetos que possibilitam a leitura e a escrita de arquivos MARC e MARCXML.
- Suporte para conversão entre arquivos MARC e arquivos MARCXML.

4.5 Arcabouço Struts

O Struts é um arcabouço para construção de aplicações para a *web*. O Struts é constituído por uma camada flexível formada por diversos componentes como Java Servlets, Java Beans e XML que juntos fornecem toda a estrutura inicial necessária para a implementação da arquitetura MVC. A arquitetura MVC por sua vez, permite que a aplicação seja dividida em três camadas: a camada do modelo, contendo a lógica de negócio, a camada de visão, que nada mais é do que uma interface de interação usuário-sistema e uma camada de controle responsável pelo controle da aplicação e de seus estados.

4.6 Velocity

O Velocity é um arcabouço utilizado para interpretar e renderizar templates. Ele permite a geração de páginas HTML a partir de templates constituídos de linguagem html, controladores de fluxo e macros.

4.7 JUnit

O JUnit é um arcabouço para a criação de testes automatizados desenvolvido sob a plataforma Java.

5 Modelagem do Banco de Dados do Colméia

Como parte do processo de implementação, foi necessário a realização de um estudo da modelagem do banco de dados do sistema Colméia uma vez que, o banco de dados seria o destino dos dados contidos nos registros

MARC fornecidos. Quando o projeto foi iniciado, durante a disciplina de Programação Extrema, o banco já havia sido modelado e sofreu mudanças durante todo o semestre. Segue uma descrição da estrutura utilizada pelo módulo Zumbido neste trabalho, as tabelas do banco são referenciadas dentro de parênteses.

Um item do acervo (ITEM_ACERVO) é a base para a inserção de quatro tipos de entidades: livros (LIVRO), teses (TESE), coleções de livros, exemplar de periódico (EXEMPLAR_DE_PERIODICO) e exemplar de não periódico (EXEMPLAR_DE_NAO_PERIODICO). Um item do acervo deve possuir como atributos uma editora (EDITORA), que o publicou, uma língua (LINGUA), na qual este foi editado, um departamento (DEPARTAMENTO), onde ele está localizado, e um assunto principal (ASSUNTO). Além disso ele pode possuir assuntos secundários, subtítulos (SUBTITULO) e um ou mais identificadores (IDENTIFICADOR), que podem ser Por exemplo, o ISBN[13] ou o ISSN[14].Eventualmente itens do acervo podem estar relacionados com ocorrências de eventos (OCORRENCIA_EVENTO), com algum tipo de observação (OBSERVACAO) ou com algum tipo de nota (NOTA).

Além de itens do acervo, existem no banco mais dois tipos de entidades principais artigos (ARTIGO) e materiais multimídia (MULTIMIDIA). Livros, teses, artigos e materiais multimídia devem obrigatoriamente ter um autor principal e podem ter eventuais autores secundários.

Existem no banco também entidades que servem para representar sinônimos de outras entidades do banco, como Por exemplo, o sinônimo de uma editora (SINONIMO_EDITORA). Este recurso foi utilizado para que fosse evitada a inserção de duas tuplas numa tabela que representassem a mesmo objeto do mundo real.

A inserção de exemplares depende da periodicidade no qual estes são publicados e do tipo de material ao qual se referem. Exemplares de teses são sempre inseridos na tabela EXEMPLAR_DE_PERIODICO, enquanto que exemplares de livros dependem de seu atributo periodicidade.

Os identificadores utilizados para inserção de tuplas nas tabelas (ids) podem ser gerados de duas maneiras, dependendo da tabela: ou são gerados a partir de uma tabela de identificadores (TABELAID), que contém uma tupla referenciando cada tabela, ou por seqüências geradas pelo próprio banco.

[13]International Standard Book Number
[14]International Standard Serial Number

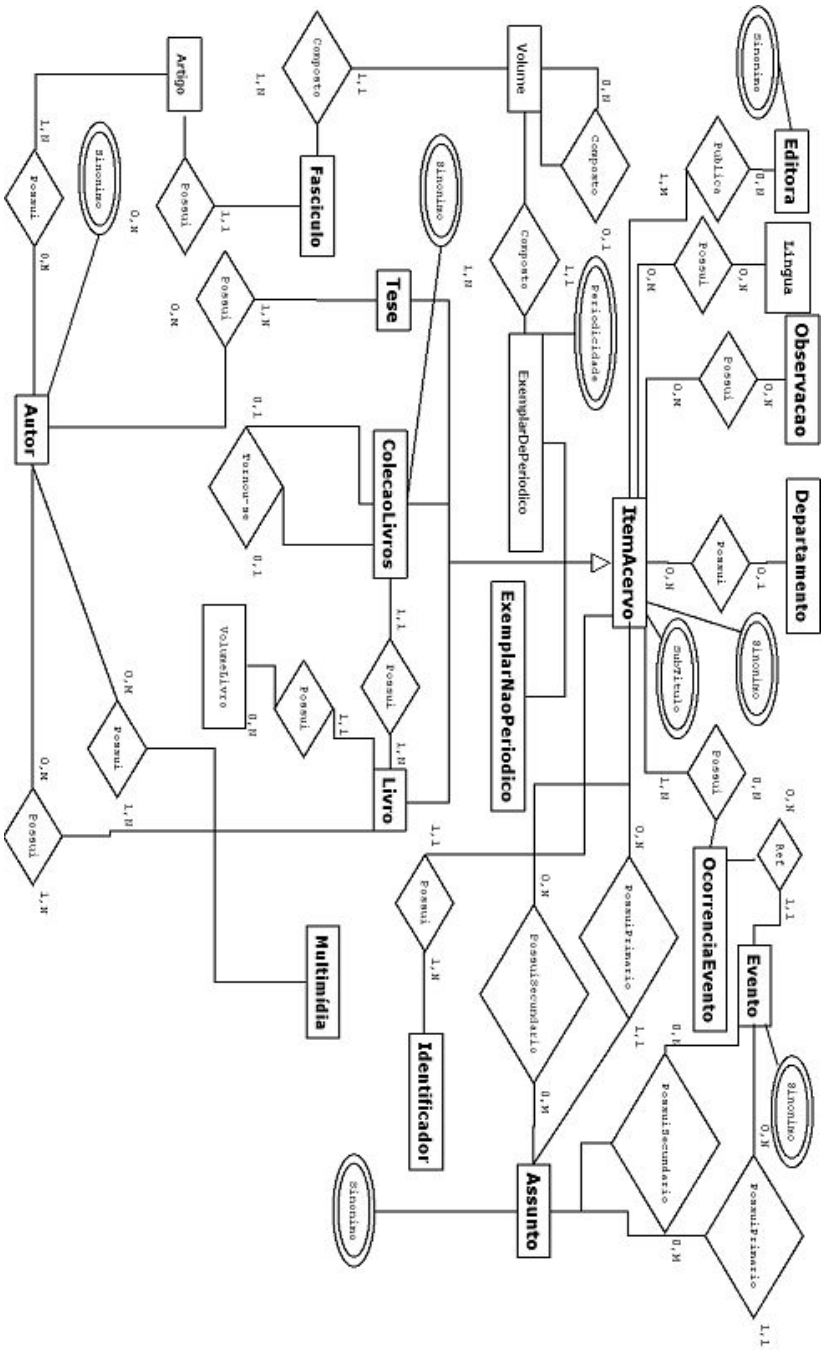


Figura 2: Modelo conceitual do banco do sistema Colméia

6 O que foi implementado?

O principal objetivo desta implementação é fornecer funcionalidades ao sistema web Comeia que permitam a seus usuários a importação de dados existentes em arquivos MARC e exportação de dados bibliográficos para arquivos MARC.

Para atingir este objetivo foi necessário conhecer o arcabouço Struts para implementar a arquitetura MVC [15]. Pode-se assim dividir o trabalho realizado em:

- Criação do componente responsável pelas rotinas envolvidas nos processos de exportação e importação de dados.
- Criação do componente gráfico, página web, e de servlets de controle.
- Criação de um mecanismo de acompanhamento do processo de importação e log.
- Criação de testes unitários.

Nas próximas seções, vamos explicar o que foi implementado para atender a essas exigências.

6.1 Importação e Exportação de dados

O pacote java br.usp.ime.colmeia.zumbido reune a lógica envolvida por trás dos processos de importação e exportação, e implementa essa lógica utilizando como subcomponentes a biblioteca MARC4j e as bibliotecas que permitam a utilização do arcabouço de persistência Hibernate. A maneira como esses subcomponentes se relacionam pode ser visualizada na figura 3.

6.1.1 O pacote br.usp.ime.colmeia.zumbido e o mar4j

Como foi dito anteriormente, o código fonte do módulo começou a ser produzido em março de 2007. Foi nessa época que se optou pela utilização da biblioteca MARC4j para facilitar a interpretação do arquivo binário no formato MARC, com o qual deveria ser realizada a população do banco de dados do Colméia. Porém, um primeiro empecilho surgiu: a biblioteca MARC4j era capaz de interpretar de maneira correta apenas arquivos que seguissem a especificação de um tipo específico de extensão do MARC. Esse tipo de extensão deveria ter uma característica existente no formato MARC21: os

[15] Model-View-Controller

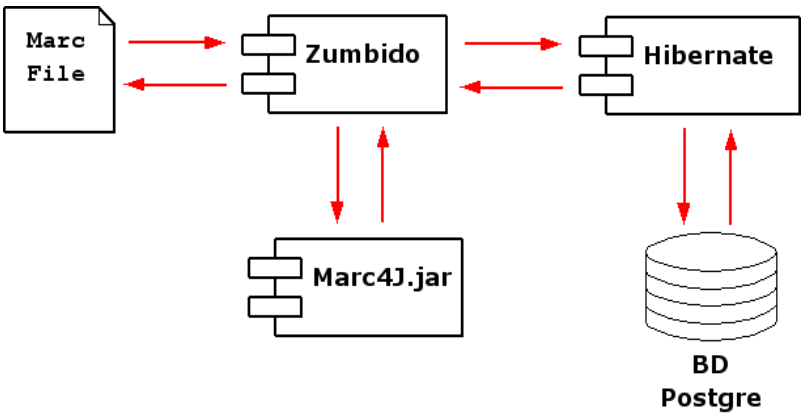


Figura 3: Relacionamento entre sub componentes

indicadores existentes nos campos de variável de dados deveriam sempre ocupar a primeira e segunda posição do campo, o que não ocorria com o arquivo fornecido para a população do banco, que tinha, assim como a primeira versão do MARC, a informação da posição dos indicadores no Líder do registro.

Tornou-se necessária então, a criação de uma classe que realizasse essa leitura e essa classe foi chamada de LenientMARCStreamReader. A função dessa classe é gerar, a partir de um objeto do tipo InputStream[16], exatamente os mesmos objetos que seriam gerados caso a interpretação do arquivo fosse feita inteiramente pela biblioteca MARC4j. Assim, o papel da biblioteca MARC4j na aplicação é fornecer objetos que encapsulem os dados contidos em um arquivo MARC e forneçam uma interface pública de manipulação dos dados. Quatro tipos de objetos existentes na biblioteca assumiram papel importante dentro do módulo, os objetos Record, DataField, ControlField e o objeto Subfield.

A classe LenientMARCStreamReader pode ser encontrada no subdiretório *util*.

6.1.2 O pacote br.usp.ime.colmeia.zumbido e o Hibernate

Para que fosse possível a utilização do Hibernate como provedor de persistência de dados, foi necessária a criação das classes que representariam as entidades alvo do banco de dados e um mapeamento entre elas e suas respectivas tabelas. Durante a disciplina de Programação Extrema a solução escolhida foi o uso de mapeamentos via arquivos XML. Sendo assim, para cada tabela do banco foi criada uma classe, que é parte do modelo da aplicação,

[16]pacote java.io.InputStream

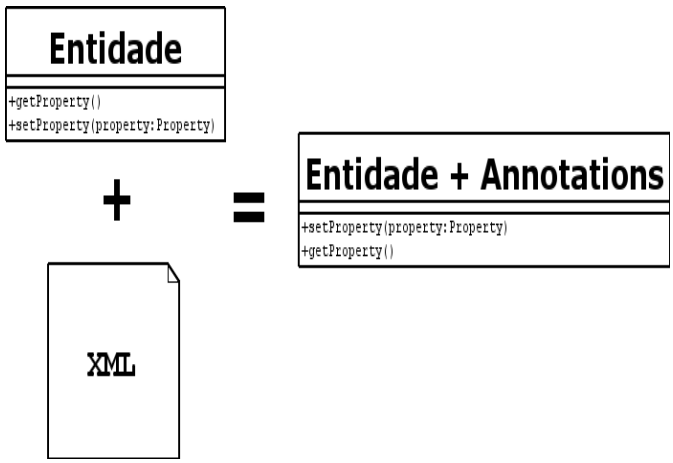


Figura 4: Equivalência entre os tipos de mapeamentos

um arquivo XML com o mapeamento objeto-relacional e uma classe DAO[17] nomeada com o sufixo *Home*. As classes DAO fornecem uma interface de persistência para a classe modelo correspondente, contendo métodos de acesso ao banco de dados. Esta arquitetura sofreu modificação no segundo semestre. Os arquivos XML foram eliminados e foram adicionados às classes metadados na forma de anotações[18], o que facilitou o processo de manutenção do código. As classes do modelo juntamente com as classes de acesso ao banco de dados estão no pacote *br.usp.ime.colmeia.zumbido.entidades*.

Além das classes e dos mapeamentos também é utilizado um arquivo de configurações do hibernate, hibernate.cfg.xml, que contém dados à respeito da conexão com o banco de dados e da localização das classes e mapeamentos relacionados com a aplicação. Esse arquivo está localizado na raíz do projeto e pode ser substituido, uma vez que API do Hibernate é compatível com uma ampla variedade de banco de dados.

6.1.3 Importação de dados

Para realizar a importação foi necessário encontrar as informações pertencentes ao modelo do banco de dados nos registros do padrão MARC. Nesse momento, não era claro quais campos do arquivo MARC continham os dados necessários para realizar a importação, e além do mais, os dados não haviam sido armazenados de maneira idêntica para todos os registros. Por exemplo, os MARCadores 490 e 440 são destinados às informações a respeito da série ou coleção a qual um item de acervo pertence. Em alguns registros essa in-

[17] Data Access Object

[18] annotations - Disponível a partir do Java 5.0

formação aparecia no campo indexado pelo MARCador 440 e em outros no campo indexado pelo MARCador 490.

A partir desse ponto, um estudo mais detalhado da estrutura do MARC começou a ser feito e foram necessárias pesquisas dentro da biblioteca do IME, com auxílio dos funcionários e utilização do sistema Dedalus[19], bem como a criação de algumas classes que permitissem o estudo do conteúdo dos registros MARC (fornecidos pela biblioteca do IME). Essas classes podem ser encontradas no subdiretório *realDataStudy*.

Havia ainda uma grande diferença entre o modelo de representação de dados utilizado nos registros MARC e o modelo de representação de dados utilizado no banco de dados do sistema. Para permitir a comunicação entre estes dois modelos criou-se uma classe *proxy* chamada ExemplarMARC[20]. Esta classe tem como objetivo receber dados contidos nas diversas variáveis de dados do registro MARC e prover uma interface de acesso aos dados que seja conivente com a modelagem do banco de dados, ou seja, por meio de métodos, ela devolve valores resultantes da interpretação dos dados fornecidos pelo registro MARC que fazem sentido para as entidades do banco. A criação desta classe é feita por meio de uma instância da classe MARCParser[21] que recebe um registro MARC e cria um objeto ExemplarMARC contendo todas as informações encontradas no registro.

Solucionado o problema de disparidade entre as representações de informações utilizadas, foram criadas as classes responsáveis pelo fluxo de controle da importação, das quais destaca-se:

MARCManager: Classe é responsável pelo controle de fluxo da importação. A ela deve ser fornecido um arquivo MARC que deverá ser analisado sintaticamente pelo MARCParser, resultando em uma instância da classe ExemplarMARC. Essa, por sua vez, será utilizada pelos diversos métodos da classe para, juntamente com as entidades e *data access objects*, para popular um grafo de objetos no banco. A inserção de tuplas no banco segue um conjunto de regras e determinações impostas pelo modelo usado na biblioteca. A classe pode ser utilizada de duas maneiras: através da interface web do sistema ou por linha de comando, onde deve ser fornecido como parâmetro o nome do arquivo MARC.

ImportExemplar: Classe que possibilita a importação de dados a partir de registros MARC no formato texto, gerado com o uso da biblioteca MARC4j ou através de serviços existentes na rede na rede *web*.

A fim de gerar um arquivo binário MARC a partir da entrada em for-

[19]Banco de Dados Bibliográficos da USP - Catálogo Online Global

[20]br.usp.ime.colmeia.zumbido.util

[21]br.usp.ime.colmeia.zumbido.importacao

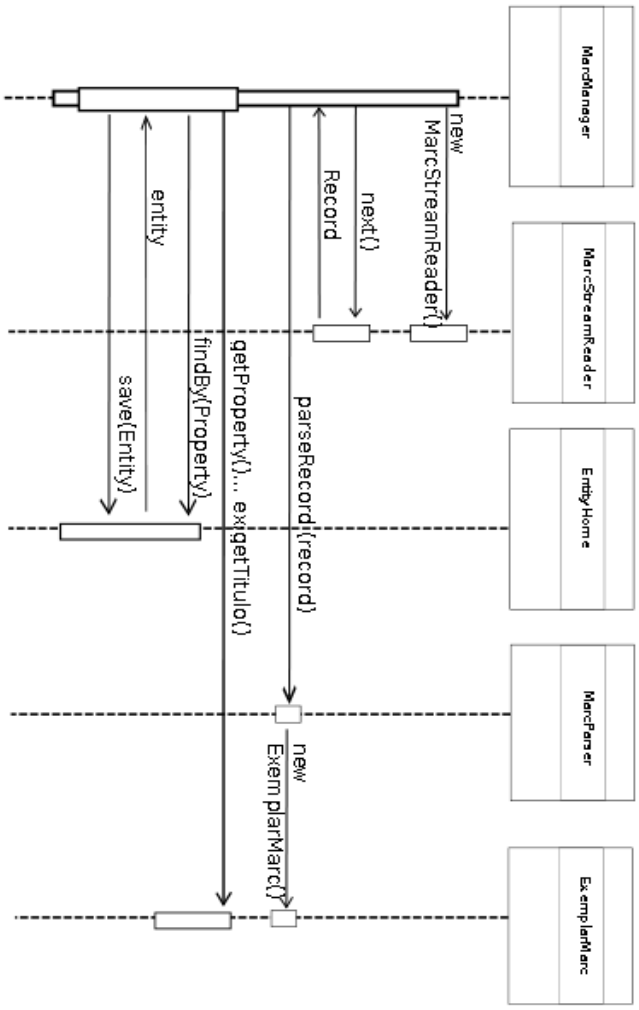


Figura 5: Diagrama de seqüência: importação de dados

mato texto, essa classe funciona como um autômato finito, onde os estados estão relacionados com as classes de identificadores contidos em um arquivo MARC. O autômato possui 4 estados:

- Leitura do líder (estado inicial).
- Leitura dos campos de controle.
- Leitura dos campos de dados.
- Estado terminal.

As regras de transição do autômato são:

- De qualquer estado é possível ir para o estado terminal, se a entrada for 'EOF'.
- Da leitura do líder, o autômato segue para o campo de controle (se entrada for diferente de 'EOF').
- No "campo de controle", enquanto a entrada possuir uma tag do tipo 00X, o autômato continua no estado "Leitura dos campos de controle". Do contrário ele segue para a "Leitura do campos de dados".
- O autômato continua enquanto um campo de dados for encontrado. Do contrário, o autômato ele retorna ao estado inicial (leitura do líder).

Essas classes estão localizadas no subdiretório *importacao* e juntamente com as outras classes já descritas constituem a importação de dados do sistema.

6.1.4 Exportação de dados

A exportação de dados da base para arquivos MARC é um processo similar ao processo de importação. Também foi necessário um controlador de fluxo de criação do arquivo e uma classe que funcionasse como objeto *proxy* entre os objetos do modelo de entidades e os campos de valor do padrão MARC.

A classe EntityWrapper[22] foi criada para ser utilizada como proxy de um arquivo MARC, a partir do encapsulamento de entidades do banco. Ela fornece métodos semanticamente coerentes com os valores armazenados por registros MARC.

[22] br.usp.ime.colmeia.zumbido.util

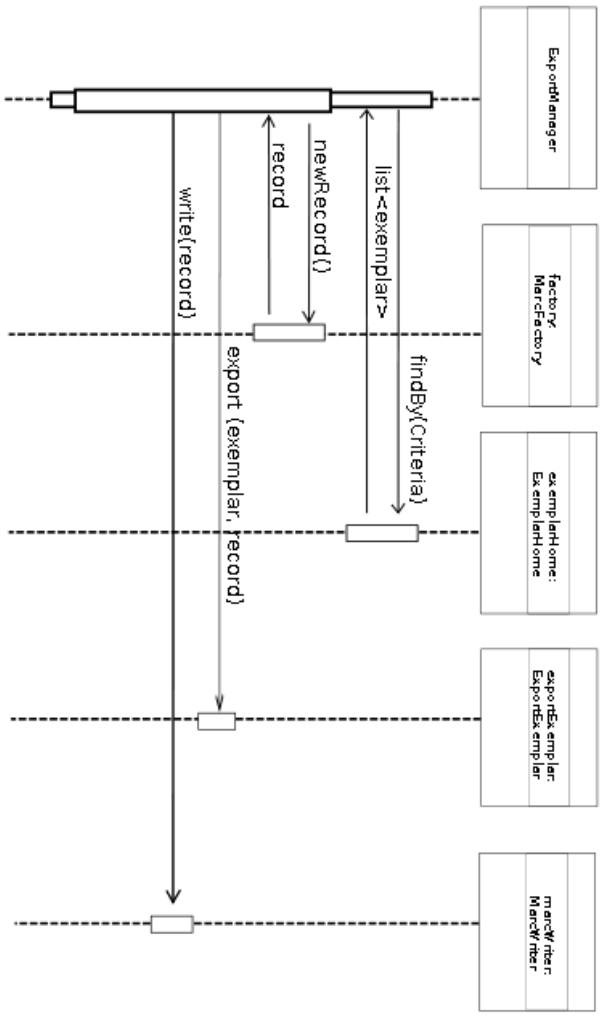


Figura 6: Diagrama de seqüência: exportação de dados

O fluxo de controle da criação do arquivo MARC é feito em duas camadas, a primeira camada é constituida pela classe MARCExport[23], que controla a lista de exemplares que devem ser transcritos, e a segunda pelas classes ExportExemplar, ExportLeaderExemplar e ExportControlFieldsExemplar[24], que cooperam para a criação de um registro MARC, ou seja, constroem, identificador a identificador, o registro.

6.2 Interface para o usuário dentro do Colméia

A integração do módulo zumbido com o Colméia foi feita através da criação de páginas html que viabilizam o uso das funcionalidades implementadas. Para tal foi necessário adequar-se à arquitetura do sistema tomando conhecimento dos arcabouços Struts e Velocity para entender o processo de desenvolvimento da aplicação.

Essa integração foi constituida da seguinte maneira:

- Criação de templates que são processados pela engine de templates do Velocity, visando a criação de páginas dinâmicas para o sistema. Essa etapa resultou na criação dos templates exportaMARC.vm, importaMARC.vm e listaImportacao.vm[25].

- Criação dos actions (classes especialidas do arcabouço Struts responsáveis pelo tratamento de requisições especificadas) responsáveis pelo controle do lado do servidor das funcionalidades implementadas. Foram criadas as classes ExportAction e ImportaMARCAction[26].

- Criação das classes ExportaForm e ImportaMARCForm[27] responsáveis por manter o estado da página dinâmica gerada.

As páginas dinâmicas resultantes permitem que o usuário faça uma busca por itens do acervo e os inclua numa lista de exportação, podendo então requisitar a exportação e salvar o arquivo gerado localmente. Já a importação permite que o usuário selecione um arquivo MARC local para *upload* ou entre com o conteúdo do arquivo em formato texto em uma área de texto e então requisite a importação dos dados, podendo acompanhar todos os processos de importação ativos em uma outra página.

[23] br.usp.ime.colmeia.zumbido.export
[24] br.usp.ime.colmeia.zumbido.export
[25] Todos no diretório: web/naoperiodicos
[26] br.usp.ime.colmeia.naoperiodicos.actions
[27] br.usp.ime.colmeia.naoperiodicos.forms

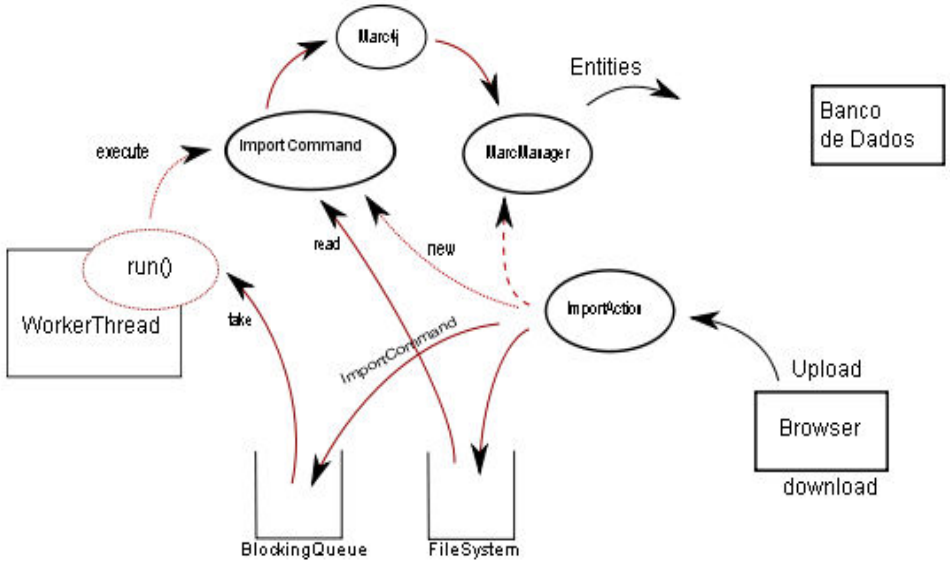


Figura 7: Fluxo de controle e de dados do processo de importação

6.3 Acompanhamento em tempo real da importação

Para que o usuário fosse realmente capaz de acompanhar o processo de importação, fez-se necessária a criação de um método de acompanhamento *on-line* da carga de dados. O sistema deveria ser capaz de dizer quantos registros foram inseridos com sucesso no banco, reportar os erros ocorridos e qualquer outra informação que fosse conveniente.

No entanto, da natureza de uma aplicação WEB emerge o problema da comunicação síncrona entre cliente e servidor quando a resposta demora mais do que alguns poucos minutos.

A análise de um arquivo MARC pode demorar bastante. O suficiente para a conexão HTTP (cliente-servidor) finalizar por timeout. Além disso, não seria a implementação mais responsiva (todos as informações direcionadas para a saída padrão serão exibidas com sorte em algum arquivo de registro do servidor, não visível para o cliente). Para resolver isso, o tratamento da informação deveria ser assíncrono.

Inicialmente, poderíamos pensar em enviar as mensagens diretamente para o navegador, o que seria inverter os papéis do navegador e do contêiner. De fato, é o navegador que faz as solicitações (request) de um certo recurso (disponível numa localização específica - URL) e é o servidor que provê tal re-

curso. Uma alterantiva seria o uso de Ajax[28] que faria o navegador aparentemente "receber notificações do servidor". Isso é possível porque com o Ajax, o navegador que faz chamadas remotas de tempos em tempos (sem que o usuário tome alguma ação específica). Abaixo, segue o funcionamento da comunicação síncrona entre o navegador e o servidor.

```
public Response processar (Request request) {
        File file = pegaArquivo(request);  // pega arquivo do request
        Resposta resposta = processa(arquivo); // processa o arquivo
        return transforma(resposta); // transforma resposta em response
}
```

A conversa do navegador com o servidor é feita por ciclos de requisições e respostas. Caso seja necessário que em uma requisição o servidor processe um dado arquivo, é preciso esperar todo o processamento para se obter a resposta do servidor. Pode-se separar o processamento da submissão do arquivo, delegando a responsabilidade para outro processo. Exemplo:

```
public Response processar (Request request) {
        File file = pegaArquivo(request);  // pega arquivo do request
        grava(arquivo); // grava o arquivo
        return paginaSucesso();
}
```

Se o processamento do arquivo é feito na mesma linha de execução da submissão, o cliente só terá uma resposta após o processamento do arquivo.

Para o servidor processar o arquivo de forma assíncrona é necessário que primeiro um "Servlet"processe e grave o arquivo num repositório temporário (seja ele uma fila JMS, um caixa de email, um diretório, etc) e um outro processo, de tempos em tempos, verifique o conteúdo desse repositório.

Esse outro processo pode atualizar um arquivo de log a medida que for necessário (por exemplo: a medida que os registros MARCs forem processados). Dessa forma, o cliente poderá "requisitar"o conteúdo do arquivo quantas vezes quiser.

Partindo desta implementação, a interface do usuário permite-lhe acompanhar o processo de carga de dados, mostrando na tela, de tempos em

[28] Asynchronous Javascript And XML

tempos, um pequeno relatório contendo o número de registros cadastrados com sucesso, o número de erros e uma descrição dos problemas encontrados na importação até o instante atual.

```
public Resposta processar (Request) {
         descobreIdArquivo(Request);
         String arquivo = leArquivo(idArquivo); // conteúdo atual do arquivo
         return converteParaResponse(arquivo);
}
```

A síntese da solução é a seguinte: Num primeiro momento o usuário faria o "upload"do arquivo. Depois o usuário deveria listar o arquivo em importação e consultar os detalhes da importação. O mesmo pode ser feito para a exportação. Uma fila de mensagens assíncronas e um processo em execução paralela devem ser implementados.

Implementamos o processo da seguinte maneira:

O servidor recebe um arquivo, via FormFile (multipart/form-data), grava o arquivo num diretório específico e envia uma objeto comando como mensagem para uma fila com o nome do arquivo em seu contexto. Por outro lado, o sistema mantém uma reserva (pool) de trabalhadores que consomem os "trabalhos"enviados para a fila. Cada processamento é registrado por um registrador (logger) num mapa cuja chave é o nome do arquivo. Uma Servlet, controla o ciclo de vida desses trabalhadores. Os trabalhadores devem por isso, realizar blocos de operação e verificar se pode executar o próximo bloco, para evitar bloquear o servidor caso seja necessário reiniciá-lo. Veja:

```
@Override
public void run() {
  try {
     while (condicao) {
        Command command = works.take();

        if (NO_MORE_WORK == command) {
           break;
        }
        while (command.hasWork() && condicao){
           command.process();
        }
        command.finish();
     }
```

```
  }
  catch (InterruptedException e) {
  }
}
```

works é a fila de trabalhos (BlockingQueue). Command é a classe que implementa o padrão Command[29]. Através de uma variável condicao verifica-se se a execução do processo deve continuar. Aqui a implementação do comando para a Importaçao do MARC:

```
private class ImportMARCCommand extends Command {

   private LenientMARCStreamReader reader;
   private MARCManager parser;
   private FileInputStream inputStream;
   private String arquivo;

   @Override
   public boolean hasWork() {
      return reader.hasNext();
   }

   @Override
   public void process() {
      parser.parse(reader.next());
   }

    public ImportMARCCommand(String nomeDoArquivo) throws NamingException,
      FileNotFoundException {
      init();
      arquivo = nomeDoArquivo;
      inputStream = new FileInputStream(nomeDoArquivo);
      reader = new LenientMARCStreamReader(inputStream);
      parser = new MARCManager(false, false);
      Logger.getInstance().getLogs().add(parser.getLog());
    }

    @Override
    public void finish() {
    parser.getLog().setFinalDate(new Date());
```

[29]Padrão Command referência Gof

```
    if (inputStream != null){
       try {
           inputStream.close();
       } catch (IOException e) {
           e.printStackTrace();
       }
    }
            delete(arquivo);
        }
    }
```

Por fim, o segredo do controle do ciclo de vida desses servelts está na classe ThreadMonitorServlet:

```
public class ThreadMonitorServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;
    static final int capacity = 10;
    final int numWorkers = 3;
    public static BlockingQueue<Command> queue = new ArrayBlockingQueue<Command>(ca
    ImportWorker[] workers = new ImportWorker[numWorkers];

    @Override
    public void destroy() {
        for (int i=0; i<workers.length; i++) {
            try {
                queue.put(ImportWorker.NO_MORE_WORK);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        super.destroy();
    }

    @Override
    public void init(ServletConfig servletConfig) throws ServletException {
        super.init(servletConfig);
        for (int i=0; i<workers.length; i++) {
            workers[i] = new ImportWorker(queue);
            workers[i].start();
        }
```

```
    }
}
```

Declarado no web.xml da seguinte maneira:

```
<servlet>
    <servlet-name>monitor</servlet-name>
    <servlet-class>
        br.usp.ime.colmeia.async.ThreadMonitorServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Uma ação listará a situação do processamento de cada arquivo. No final do processo, o trabalhador tenta apagar o arquivo processado. A cada novo arquivo, o registrador apaga todos registros mais antigos que uma data específica mantendo a lista de logs tão grande o número de solicitações no último instante.

6.4 Testes unitários

Foram criadas diversas rotinas de testes para o módulo Zumbido. Estas rotinas estão no pacote br.usp.ime.colmeia.zumbido dentro do diretório src/test. Para a criação dos testes foi utilizado o arcabouço de criação de testes automatizados JUnit. As rotinas de testes desempenham papel fundamental no desenvolvimento do módulo, pois existem mais pessoas trabalhando e modificando o banco de dados do sistema e os testes permitem uma detecção quase que imediata destas mudanças, dando segurança para os desenvolvedores.

7 O que não foi implementado?

Apesar de ter tido seu funcionamento estudado, a implementação do protocolo Z39.50 não pode ser realizada.

8 Método de desenvolvimento

Uma vez que o projeto passou a ser desenvolvido durante a disciplina de programação extrema, no primeiro semestre o desenvolvimento foi baseado nas técnicas aprendidas em aula.

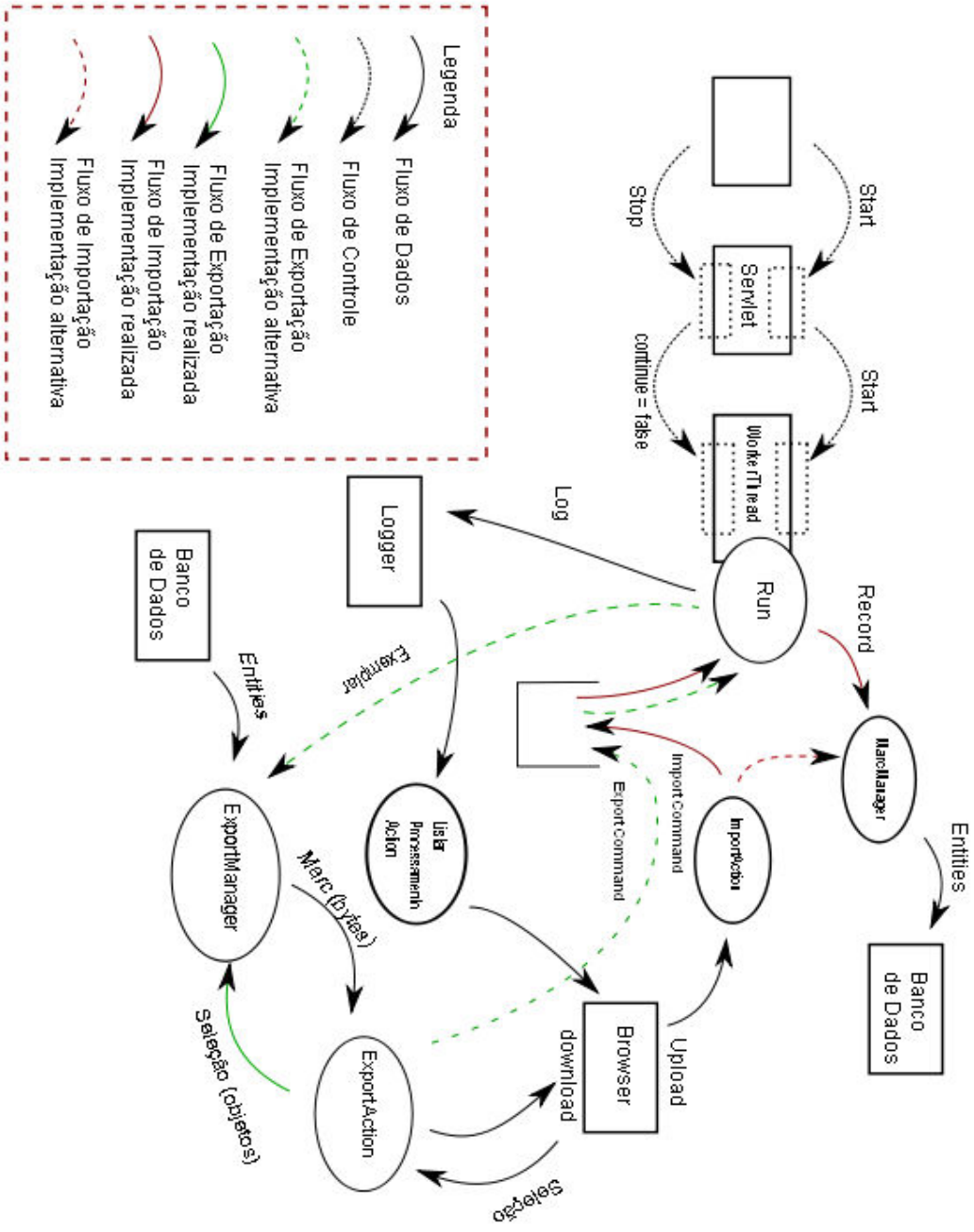


Figura 8: Fluxos de controle e dados (geral)

Programação Extrema é uma metodologia de desenvolvimento ágil para equipes de pequeno ou médio porte. A metodologia inclui 12 técnicas principais de desenvolvimento, das quais pode-se aplicar subconjuntos, dependendo do contexto de trabalho e necessidades da equipe de desenvolvimento. As 12 técnicas são:

- Jogo do Planejamento: São realizadas interações entre os desenvolvedores e os clientes. Em cada interação os clientes expõem novas funcionalidades que gostariam que fossem acrescentadas à aplicação, o desenvolvedor analisa a possibilidade de implementação na nova funcionalidade naquele momento e caso esta seja possível, a nova funcionalidade é descrita em cartões contendo sua descrição na forma de tarefas que o usuário pode realizar. Tarefas podem ser divididas em subtarefas. Os desenvolvedores devem estimar o tempo necessário para a implementação das novas funcionalidades.

- Fases Pequenas: As interações entre clientes e desenvolvedores devem ser constantes, de modo que, a cada interação, novas funcionalidades são apresentadas aos clientes que especificam novas funcionalidades que serão apresentadas na próxima interação. Para que seja possível interação constante, as tarefas especificadas não devem ser muito grandes.

- Metáfora: Metáforas são importantes para aumentar a comunicação entre clientes e desenvolvedores e muitas vezes ajudam o desenvolvedor no planejamento da arquitetura do software.

- Design Simples: A modelagem e arquitetura do sistema devem ser simples. Não é necessária a utilização de uma arquitetura sofisticada se ela não se faz necessária.

- Testes: A implementação de uma funcionalidade só está completa depois que o desenvolvedor preparou uma rotina de testes para ela. A criação de testes deve fazer parte do cotidiano do programador XP.

- Refatoração: A refatoração do código deve ser constante e deve ocorrer sempre que necessária. Essa prática permite que o código mantenha-se organizado e que no final do projeto o código da aplicação seja reutilizável e de fácil entendimento.

- Programação Pareada: Os programadores devem trabalhar em pares utilizando um mesmo micro. Esta técnica diminui a chance de criação de código ruim ou errado, além de possibilitar o surgimento de idéias mais elaboradas.

- Propriedade Coletiva: Não existe código de apenas um programador, uma classe escrita por um programador pode ser modificada por outro. O código deve estar sempre disponível em um repositório e é dever de todos os programadores o envio do código produzido todos os dias.

- Integração Contínua: As mudanças devem ser integradas continuamente, ou seja, não deve haver muita diferença entre a versão da aplicação apresentada ao cliente e a versão sob a qual os programadores trabalham.

- Semana de 40 Horas: Os programadores não devem trabalhar mais do que 40 horas semanais, pois sua produtividade e rendimento podem ser influenciados pelo cansaço e indisposição.

- Cliente junto aos Desenvolvedores: Os clientes devem estar sempre que possível junto aos desenvolvedores para possibilitar as fases pequenas de desenvolvimento.

- Padronização do Código: Uma vez que o código é de propriedade coletiva, é necessário que este siga um padrão mantendo-se organizado e compreensível para todos os membros da equipe.

Após o término da disciplina optou-se pela manutenção das seguintes técnicas aprendidas: design simples, testes, refatoração, integração contínua e padronização de código.

O código fonte foi todo editado dentro do ambiente de desenvolvimento integrado Eclipse, o repositório de código fonte escolhido foi o Subversion e como ferramenta de acesso ao banco de dados utilizamos o PgAdmin e o plugin do eclipse SQLExplorer. A ferramenta de acesso ao banco de dados PgAdmin foi muito importante, pois durante o desenvolvimento foi necessário o estudo também das StoredProcedures que realizavam inserção de itens dos quais o cadastro é feito pela interface *WEB* do sistema.

9 Conclusões

O módulo Colméia-Zumbido é parte fundamental do projeto Colméia. Alguns dos objetivos esperados foram atingidos, no que diz respeito à importação e exportação de dados o sistema já atende às exigências, além disso fornece uma funcionalidade adicional que permite o acompanhamento do processo de carga de dados. Espera-se que no futuro o módulo seja capaz de gerar arquivos binários no formato MARC com registros mais completos e homogêneos, para que isso seja possível o banco de dados do Colméia deve

ser extendido para armazenar mais informações a respeito dos materiais do acervo da biblioteca. O estudo realizado sobre o formato MARC e o protocolo Z39.50 deixou claro que o sistema Colméia deve, além de automatizar as tarefas realizadas em uma biblioteca universitária, integrar-se com outros sistemas bibliográficos o que possibilitará uma maior completude no que diz respeito à informações associadas a itens de seu acervo. O ideal seria que o módulo não oferecesse apenas uma interface de consulta para outras bases bibliográficas, mas que também permitisse que outros sistemas realizassem consultas remotas em seu banco de dados.

Referências

[1] http://colmeia.incubadora.fapesp.br/portal.

[2] http://eclipsesql.sourceforge.net.

[3] http://itsmarc.com/crs/bib1468.htm.

[4] http://marc4j.tigris.org.

[5] http://subversion.tigris.org.

[6] http://www.cbr.washington.edu/ camel/z/z.html.

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

[8] http://www.hibernate.org.

[9] http://www.ibict.br/cionline/viewarticle.php?id=430.

[10] http://www.libraryhq.com/z3950qa.html.