UNIVERSIDADE DO EXTREMO SUL CATARINENSE – UNESC

CURSO DE CIÊNCIA DA COMPUTAÇÃO

DIEGO PEREIRA DO NASCIMENTO

GERADOR DE CÓDIGO PYTHON DA CAMADA DE MODELO DE APLICAÇÕES DJANGO A PARTIR DE DIAGRAMAS UML

CRICIÚMA, DEZEMBRO DE 2008

DIEGO PEREIRA DO NASCIMENTO

GERADOR DE CÓDIGO PYTHON DA CAMADA DE MODELO DE APLICAÇÕES DJANGO A PARTIR DE DIAGRAMAS UML

Trabalho de Conclusão de Curso apresentado para obtenção do Grau de Bacharel em Ciência da Computação da Universidade do Extremo Sul Catarinense.

Orientadora: Profa. MSc. Ana Cláudia Garcia Barbosa

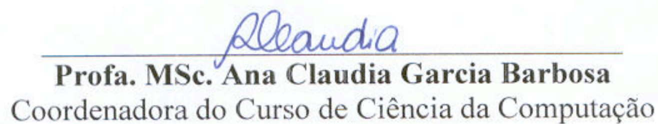
Co-orientador: Prof. MSc. Eduardo Menna da Silva

CRICIÚMA, DEZEMBRO DE 2008

DIEGO PEREIRA DO NASCIMENTO

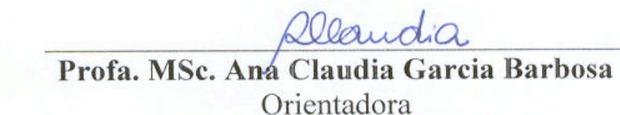
GERADOR DE CÓDIGO PYTHON DA CAMADA DE MODELO DE APLICAÇÕES DJANGO A PARTIR DE DIAGRAMAS UML

Submetido ao corpo docente do Curso de Ciência da Computação da Universidade do Extremo Sul Catarinense como um dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

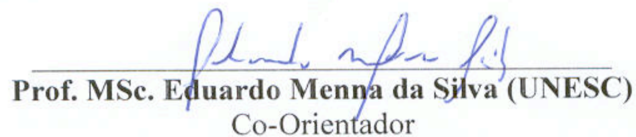


Profa. MSc. Ana Claudia Garcia Barbosa
Coordenadora do Curso de Ciência da Computação

Banca Examinadora:



Profa. MSc. Ana Claudia Garcia Barbosa
Orientadora



Prof. MSc. Eduardo Menna da Silva (UNESC)
Co-Orientador

Prof. MSc. Daniel Pezzi da Cunha (UNESC)

Prof. Fabrício Giordani (UNESC)

Aos meus pais, Jandir Pereira do Nascimento e Maurina Inácio do Nascimento, e ao meu amor, Paula P. Teixeira.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, que sempre me deram a estrutura necessária para que eu pudesse desenvolver todo o meu potencial intelectual e criativo e me apoiaram nas minhas escolhas. Agradeço a minha namorada Paula, tão importante para a conclusão desse trabalho, seja me ajudando com a revisão ou me dando incentivo e motivação para continuar adiante. Agradeço também aos meus amigos Cláudio, Dirceu, Fabiano, Rickifer e Rodrigo que em algum momento da minha vida acadêmica ou pessoal, contribuíram significativamente para que eu pudesse seguir adiante. A todas as pessoas que de alguma forma me ajudaram, direta ou indiretamente para a conclusão desse trabalho, meus mais sinceros agradecimentos.

″Ninguém é mais escravo do que aquele que falsamente se acredita livre.″
(Johann Wolfgang Von Goethe)

RESUMO

O presente trabalho demonstra uma ferramenta, denominada DjangoGen, capaz de gerar a estrutura de arquivos padrão e a camada de modelo de uma aplicação Django, ao mesmo tempo em que documenta o projeto, utilizando o diagrama de classes da *Unified Modeling Language* (UML), pré requisito para o funcionamento da ferramenta desenvolvida. O processo é realizado através do *parsing* do XML referente ao arquivo UML do diagrama de classes salvo pelo modelador UML *Dia Modeler* e posteriormente o código Python é gerado, baseado nos dados obtidos do *parsing.* Através do processo de *parsing* e geração de códigos, o desenvolvimento da camada de modelo e geração dos arquivos base de uma *aplicação Web* do Django é abstraída do desenvolvedor, bem como a padronização e a criação automática de tabelas e campos na base de dados baseados nas classes e atributos definidos no diagrama UML de classes.

Palavras chave: Engenharia de Software, Arquitetura Orientada por Modelos, Frameworks Web.

ABSTRACT

The following work describes a tool called DjangoGen, which can generate the default directory tree, base files and model layer of a Django application, and at the same time can document the project using class diagrams from Unified Modeling Language (UML), a dependency of the developed tool. The proccess works by parsing the XML structure from the UML class diagram created with UML Dia Modeler and creating Python code based on the data found in the parsing. Through parsing and code generation, the development of the model layer and the creation of the base files of a Django web application are abstracted from the developer, as well as the standardization and creation of tables and fields based on data found in the classes and attributes of the UML class diagram.

Keywords: Software Engineering, Model Driven Architecture, Web Frameworks.

LISTA DE ILUSTRAÇÕES

Figura 1. Exemplo de código Python. 21

Figura 2. Execução do código Python da Figura 1 em um Shell Unix. 21

Figura 3. Exemplo básico de código HTML. 22

Figura 4. Diagrama conceitual do fluxo entre as camadas do padrão MVC para a *Web*. 26

Figura 5. Exemplo de arquivo XML. 28

Figura 6. Fluxo de trabalho de um processo utilizando MDA. 29

Figura 7. Representação de herança de classes na UML. 30

Figura 8. Funcionamento de uma aplicação Web. 36

Figura 9. Exemplo de camada de modelo do Django. 41

Figura 10. Adicionando a classe *Vinho* na interface de administração do Django. 41

Figura 11. Listagem padrão de objetos cadastrados, na interface de administração do Django. 42

Figura 12. Customização da aparência da interface de administração de uma classe Django. 42

Figura 13. Listagem dos campos customizados da interface de administração do Django. 43

Figura 14. Visão padrão ao editar um objeto do Django. 43

Figura 15. Mensagem de erro gerada pela validação automática de campos do Django. 44

Figura 16. Exemplo de uso de variáveis no Django. 45

Figura 17. Exemplo de filtros do Django. 46

Figura 18. Fluxo de trabalho da ferramenta DjangoGen. 51

Figura 19. Diagrama de componentes da ferramenta DjangoGen. 53

Figura 20. Diagrama de atividades da ferramenta DjangoGen. 54

Figura 21. Diagrama UML de classes exemplo, que segue as recomendações de construção adequadas ao DjangoGen. 58

Figura 22. Exemplo de uso de *widgets* do Django. 61

Figura 23. Exemplo de uso de *widgets* do Django. 61

Figura 24. Uso de *widgets* no presente trabalho. 62

Figura 25. Estrutura de diretórios do DjangoGen. 63

Figura 26. XML de um diagrama de classes de exemplo salvo pelo *Dia Modeler*. 64

Figura 27. Função que lê o arquivo UML do *Dia Modeler* e converte em objeto Python. 65

Figura 28. Função que imprime o código Python final. 66

Figura 29. Interface da ferramenta DjangoGen. 69

Figura 30: Estrutura de arquivos de uma *app* que acabou de ser gerada. 70

Figura 31: Exemplo de classe contendo a função __*unicode*__ configurada de forma errada. 71

Figura 32: Exemplo de classe contendo a função __*unicode*__ configurada de forma correta. 72

LISTA DE TABELAS

Tabela 1. Tipos de diagramas oficiais da UML. 32

Tabela 2. Conversão de tipos do *Dia Modeler* em tipos do Django. 57

LISTA DE SIGLAS

APP	Application
CIM	Computer Independent Model
COSMOS	Component Structuring Model for Object-Oriented Systems
CSS	Cascading Style Sheets
DTD	Document Type Definition
FAQ	Frequently Asked Questions
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
JPEG	Joint Photographic Experts Group
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MTV	Model Template View
MVC	Model View Controller
OMG	Object Management Group
OO	Orientação a Objeto
ORM	Object Relational Mapper
PHP	PHP Hypertext Processor
PIM	Platform Independent Model
PNG	Portable Network Graphics
PSM	Platform Specific Model
SIG	Sistemas de Informação Geográfica
SQL	Structured Query Language

SVG	Scalable Vectorial Graphics
UML	Unified Modeling Language
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XHTML	eXtensible HyperText Markup Language
XIS	XML Information System Models and Architectures
XML	eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO 15
1.1 OBJETIVO GERAL 16
1.2 OBJETIVOS ESPECÍFICOS 16
1.3 JUSTIFICATIVA 17
1.4 ESTRUTURA DO TRABALHO 19
2 CONCEITOS GERAIS E PADRÕES WEB 20
2.1 LINGUAGEM DE PROGRAMAÇÃO PYTHON 20
2.2 LINGUAGEM HTML 21
2.3 PADRÃO CSS 23
2.4 SERVIDOR WEB INTEGRADO DO DJANGO 23
2.5 PADRÃO MVC 24
2.6 EXTENSIBLE MARKUP LANGUAGE (XML) 26
2.7 MODEL DRIVEN ARCHTETURE (MDA) 28
2.8 HERANÇA 29
3 UNIFIED MODELING LANGUAGE (UML) 31
3.1 TIPOS DE DIAGRAMAS UML 32
3.2 DIAGRAMA DE CLASSES 33
3.2.1 Atributos 33
3.2.2 Operação 34
4 FRAMEWORK WEB 36
4.1 FRAMEWORK WEB DJANGO 38
4.1.1 Camada de Modelo do Django 39
4.1.2 Interface de Administração do Django 40
4.1.3 Sistema de templating do Django 44
4.1.4 Variáveis 45
4.1.5 Filtros 46
5 TRABALHOS CORRELATOS 47
5.1 PROCESSO RAPDIS 47
5.2 ABORDAGEM XIS 47
5.3 PROCESSO DE DESENVOLVIMENTO BASEADO EM COMPONENTES ADAPTADOS AO MDA 48
5.4 DESENVOLVIMENTO DE SIG PARA WEB UTILIZANDO MDA 48
5.5 CRIAÇÃO DE COMPONENTES DINÂMICOS PARA JOOMLA POR MEIO DE UML 49
6 DJANGOGEN: MODELAGEM, ESTRUTURA E DESENVOLVIMENTO 50
6.1 METODOLOGIA DO TRABALHO DESENVOLVIDO 52
6.2 MODELAGEM DA FERRAMENTA DJANGOGEN 53
6.2.1 Diagrama de Componentes 53
6.2.2 Diagrama de Atividades 54
6.3 FERRAMENTA CASE UML DIA 55
6.4 REGRAS DE CRIAÇÃO DE DIAGRAMAS UML NO DIA 56
6.5 DESENVOLVIMENTO DE APPS PARA O FRAMEWORK WEB DJANGO 58
6.5.1 Estrutura de arquivos de uma aplicação Django 59

6.5.2 Desenvolvimento em Django utilizando widgets 61
6.6 A ESTRUTURA DA FERRAMENTA DJANGOGEN 62
6.7 EXTRAÇÃO DE INFORMAÇÕES DO ARQUIVO UML / XML DO DIA MODELER...64
6.8 GERAÇÃO DE APPS DJANGO A PARTIR DE DIAGRAMAS UML / XML DO DIA MODELER 65
6.9 INSTALANDO A FERRAMENTA DJANGOGEN 67
6.10 UTILIZAÇÃO DA FERRAMENTA DJANGOGEN 69
6.11 RESULTADOS OBTIDOS..72

CONCLUSÃO 74

REFERÊNCIAS 76

1 INTRODUÇÃO

A disseminação da internet provocou uma revolução na forma de comunicação e relacionamento das pessoas. O que fazia-se apenas presencialmente, pode ser feito hoje no conforto de casa, em alguns poucos cliques do *mouse*. Estas facilidades vão de compras e vendas *online* a pagamentos de títulos e outros tipos de transações financeiras, tão importantes para muitas pessoas e empresas.

A demanda por serviços *online* tem aumentado dia após dia, bem como a busca por padrões de qualidade, segurança e agilidade das informações disponibilizadas na *Web*. Para cumprir os prazos, muitas empresas de desenvolvimento acabam reduzindo o número de testes necessários em seus *softwares*, implementando funcionalidades importantes sem o domínio necessário do conhecimento e acabando por deixar de documentar o código fonte desenvolvido. Os problemas causados por estas escolhas são o aumento no número de erros na aplicação, insatisfação do cliente, dificuldade de manutenção e retrabalho (THOMAZ, 2007).

Como forma de resolver os problemas mencionados anteriormente, criou-se os *frameworks*[1] *Web*, que auxiliam significativamente na implementação e administração de portais e outros tipos de aplicações *Web* dinâmicas. Dentre os *frameworks Web* desenvolvidos tem-se o Django. O Django é escrito em Python e é baseado na metodologia *Model View Controller* (MVC) e pode ser utilizado no desenvolvimento e administração de aplicações *Web* dinâmicas, focando a agilidade e facilidade no desenvolvimento. O Django é utilizado ativamente desde 2005 em sites que necessitam de atualizações constantes no seu conteúdo, como é o caso do site do

---

[1] Um *framework* é um conjunto de componentes de software que provêm uma arquitetura e estrutura básica para o desenvolvimento de uma aplicação.

jornal *Washington Post* (DJANGO SITES, 2008).

Apesar dos *frameworks Web* proporcionarem uma diminuição no tempo de desenvolvimento, o desenvolvedor ainda tem de se preocupar com duas coisas: documentação do projeto e criação da camada de modelo da aplicação.

Para que o processo de desenvolvimento pudesse se tornar ágil, sem perder a qualidade exigida, esse trabalho se propôs a desenvolver uma ferramenta de conversão de diagramas de classes UML, para código Python do *framework Web* Django. O processo de conversão se resume em realizar a leitura dos dados provenientes do arquivo UML salvo pelo *Dia Modeler*, que internamente, organiza os dados utilizando o formato *eXtensible Markup Language* (XML).

1.1 OBJETIVO GERAL

Facilitar o desenvolvimento e documentação de aplicações Django por meio de uma ferramenta de geração da camada de modelo de aplicações do *framework Web* Django a partir de diagramas de classes UML.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos dessa pesquisa consistem em:

a) compreender os mecanismos do *framework* Django e a modelagem de diagramas de classes UML;
b) analisar as informações armazenadas nos diagramas de classes UML no formato XML;

c) desenvolver uma ferramenta que possibilite a geração da camada de modelo de aplicações do *framework Web* Django a partir de diagramas de classes UML;
d) facilitar ao desenvolvedor a geração da documentação e código do sistema, correspondente a camada de modelo da aplicação;
e) testar por meio do desenvolvimento de um diagrama de classes, a conversão para código Python.

1.3 JUSTIFICATIVA

Muitas empresas atualmente buscam o crescimento e o aumento de seu *market share*[2] por meio da disponibilização de seus serviços pela *Web*. Para suprir essa demanda são necessários profissionais capacitados com conhecimentos de vários tipos de linguagens, padrões e ferramentas de desenvolvimento *Web*.

Devido à complexidade crescente no desenvolvimento de aplicações para a *Web*, muitos profissionais se vêem pressionados por prazos cada vez menores, o que conseqüentemente causa uma redução na integridade do *software*, impedindo que se siga um padrão de qualidade no desenvolvimento, também se refletindo na elaboração da documentação do projeto que acaba ficando em segundo plano. A redução na qualidade do *software* é materializada em forma de *bugs*, funcionalidades diferentes das solicitadas pelo cliente, dificuldades no entendimento do código por outros desenvolvedores e retrabalho.

Com o intuito de resolver esses problemas, surgiram os *frameworks Web*,

[2] Medida da participação de uma marca ou produto no mercado.

que auxiliam no desenvolvimento de aplicações conforme os padrões de desenvolvimento. O objetivo é que o *framework* forneça um ponto de partida e um caminho a seguir, de forma que o desenvolvedor possa se concentrar no desenvolvimento de uma solução do problema de domínio específico (THOMAZ, 2007).

Dentre diversos *frameworks Web* escritos em Python, temos o Django que tem por objetivo tornar o desenvolvimento de aplicações *Web* mais ágil, além de garantir um padrão de qualidade e organização do código fonte. O Django estrutura a aplicação em camadas, utilizando uma variante da metodologia MVC, chamada *Model Template View* (MTV) e possui um recurso conhecido como Interface de Administração[3]. Por meio desta interface é possível visualizar, adicionar ou apagar objetos da base de dados da aplicação.

Segundo Holovaty e Moss (2007) o Django possui uma curva de aprendizado muito pequena, facilitando o desenvolvimento de novas aplicações, mesmo por pessoas com pouco conhecimento sobre o *framework*.

Além do desenvolvimento do código da aplicação, existem outros aspectos importantes em um projeto de desenvolvimento de *software*, como por exemplo a documentação. A área de engenharia de *software* explora essa questão já no desenvolvimento da especificação do projeto, por meio de diagramas UML, pois desta forma pode-se ter uma visão completa do projeto, facilitando o entendimento da aplicação e demonstrando melhor como os componentes do sistema se relacionam (FOWLER, 2000).

Considerando as vantagens do uso da UML, foram desenvolvidas algumas ferramentas capazes de converter diagramas UML em código Python, mas atualmente

---

[3] Mais informações no tópico 4.1.2 desse trabalho.

nenhuma gera código para o Django.

Com o propósito de pesquisar os problemas e vantagens do uso de diagramas UML para documentação e desenvolvimento de aplicações Django, este trabalho propõe a implementação de uma ferramenta capaz de gerar a estrutura base de aplicações do *framework* Django a partir de diagramas de classes da UML.

1.4 ESTRUTURA DO TRABALHO

Este trabalho é formado por 6 seções. Na seção 1 há uma introdução aos objetivos do projeto, os temas envolvidos e a justificativa para a realização do mesmo.

Nas seções 2, 3 e 4 são abordados os temas relacionados a conceitos e padrões *Web*, UML e *frameworks Web*.

A seção 5 apresenta trabalhos correlatos, onde o objetivo dos mesmos é semelhante ao deste trabalho.

A ferramenta desenvolvida, bem como detalhes sobre a implementação, instalação e uso são abordados na seção 6.

Por fim, há uma conclusão e as possibilidades para trabalhos futuros.

2 CONCEITOS GERAIS E PADRÕES WEB

Para o completo entendimento do presente trabalho, se faz necessária a compreensão de alguns conceitos básicos sobre padrões *Web* e de engenharia de *software*. Tais conceitos serão apresentados nos próximos tópicos e complementados no restante do trabalho.

2.1 LINGUAGEM DE PROGRAMAÇÃO PYTHON

Conforme Ferri (2002) Python é uma linguagem de programação interpretada, de alto nível, orientada à objetos e desenvolvida por Guido van Rossum em meados da década de 1990.

De acordo com Lutz e Ascher (2004) algumas das vantagens técnicas do Python são:

a) orientado a objetos;

b) gratuito;

c) possui portabilidade;

d) apresenta diferentes características como:

- tipagem dinâmica[4],
- gerenciamento de memória automático,
- bibliotecas,
- utilitários de outros fornecedores;

[4] O tipo ao qual a variável está associada pode variar durante a execução do programa.

e) pode ser misturado com outras linguagens;

f) fácil de usar e aprender.

Um exemplo simples de código Python pode ser visto na Figura 1. O código da Figura 1 solicita o nome da pessoa, imprime quantos caracteres possui o primeiro nome e em seguida imprime o nome da pessoa.

```
#!/usr/bin/python
# -*- coding: utf8 -*-
texto = raw_input("Digite o seu nome: ")
print "O seu nome possui %s caracteres." % len(texto)
if 'Diego' in texto:
    print "Olá Diego!"
else:
    texto = texto.split()[0]
    print "Ola %s" % texto
```

Figura 1. Exemplo de código Python.

O resultado da execução do código da Figura 1, executando em um *Shell Unix*, pode ser visto na Figura 2.

```
diego:Desktop diego$ python teste.py
Digite o seu nome: Diego
O seu nome possui 5 caracteres.
Olá Diego!
diego:Desktop diego$ python teste.py
Digite o seu nome: Roberto
O seu nome possui 7 caracteres.
Ola Roberto
diego:Desktop diego$
```

Figura 2. Execução do código Python da Figura 1 em um Shell Unix.

2.2 LINGUAGEM HTML

De acordo com Niedest (1998) HTML é uma linguagem usada para criar

documentos *Web,* que define o uso de instruções especiais, chamadas de *tags*, que não são exibidas ao usuário ao visualizar a página a partir do navegador de páginas *Web* (*browser*), mas informa ao mesmo, de que forma o conteúdo deve ser exibido. A HTML também é usada para criar links para outros documentos, tanto da rede local como uma rede externa como a Internet.

Um exemplo de *tag* HTML é a *tag* *<html>* que define o início de um documento HTML. Outro exemplo é a *tag* *<body>* que define o início do corpo do documento HTML. As *tags* iniciadas com </ definem o fim de um elemento, de forma que *</body>* define o fim do corpo do documento e *</html>* define o fim do documento HTML. Um exemplo de código HTML, pode ser visualizado na Figura 3, onde é definido o título para a página HTML e um pequeno para o corpo da página.

O padrão HTML e todos os outros padrões *Web* são desenvolvidos e mantidos sob a autoria da *World Wide Web Consortium* (W3C), um consórcio formado por várias empresas, que regulam diversos padrões utilizados na *Web* (NIEDEST, 1998).

```
<html>
<head>
    <title>Pagina de testes</title>
</head>
<body text="#FF6600">

    Pagina exemplo.

</body>
</html>
```

Figura 3. Exemplo básico de código HTML.

2.3 PADRÃO CSS

Conforme Schmitt (2006) a Folha de estilo em Cascata, do inglês *Cascading Style Sheet* (CSS) é uma sintaxe padrão que possibilita aos *Web designers* o controle sobre como é feita a apresentação das páginas *Web*. O uso do CSS, permite que os *Web designers* possam definir com precisão, as cores, fontes, posicionamento e outros atributos dos objetos relacionados ao estilo do documento *Web*, o que em outras palavras, significa que o CSS permite estender o *design* e controlar como os componentes da página *Web* são exibidos no *browser*. Utilizando-se o CSS é possível centralizar a definição dos estilos em um único arquivo e utilizá-lo em diversos documentos HTML diferentes. Desta forma, a alteração de uma propriedade no arquivo CSS fará que todos os documentos HTML que o utilizam, tenham o seu estilo modificado.

Outra vantagem do uso de CSS é a não necessidade de um *hardware* especial ou *software* para usá-lo. O mínimo requerido é um computador, um *browser* como o Firefox, Safari ou Chrome e um editor de HTML. Os *browsers* atuais possuem a capacidade de exibir o conteúdo conforme o CSS de cada página. Pode-se utilizar um editor de textos como editor HTML, ou algo para uso profissional, como por exemplo o *Adobe Dreamweaver* (SCHMITT, 2006).

2.4 SERVIDOR WEB INTEGRADO DO DJANGO

Para que seja possível disponibilizar um site na rede, seja ela pública ou

privada, é necessário que o site em questão seja disponibilizado para acesso via servidor *Web.* Existem vários tipos de servidores *Web* em uso atualmente, como o *lighthttp, nginx,* dentre outros, sendo o servidor *Apache* o mais conhecido e utilizado servidor *Web* (NETCRAFT, 2008).

Segundo Django Project (2008b) o Django possui um servidor *Web* integrado que permite editar e executar códigos sem a necessidade de reiniciar o servidor de aplicações, porém não deve ser usado em servidores de produção devido ao fato de ele atender uma única requisição por vez, o que inviabilizaria o acesso simultâneo do site a mais de uma pessoa ao mesmo tempo. Nesse trabalho, é utilizado o servidor integrado do Django, por suas características de desenvolvimento e *debug*[5] rápidos, porém o Django pode ser utilizado em conjunto com vários outros tipos de servidores *Web*, como os supracitados *nginx* e *Apache*.

2.5 PADRÃO MVC

Segundo Sun Microsystens (2008) Modelo, Visualização e Controle, do inglês *Model View Controller* (MVC) é um padrão arquitetural usado em engenharia do *software*, que determina que um programa deve ser dividido em três partes ou camadas distintas, separando o código de cada camada em arquivos diferentes. Essa separação deve ser realizada de tal forma que a alteração de um arquivo não afete significativamente os outros arquivos.

---

[5] Depuração ou *debug* é o processo de encontrar bugs ou problemas, num aplicativo de *software* ou mesmo em *hardware*.

A camada de modelo representa os dados da aplicação, como classes e atributos de classes. Nesta camada, é desenvolvido todo o código referente aos dados, e deve ser construída de forma a garantir a integridade dos dados manipulados pela aplicação.

A camada de visualização é responsável por definir quais dados serão fornecidos ao usuário e como o usuário irá interagir com a aplicação, permitindo que o usuário possa realizar a entrada de dados e a execução de operações. Nessa camada, deve ficar apenas o código referente a interface com o usuário.

A camada de controle traduz as interações com a camada de visualização em ações para serem realizadas no modelo. Em uma aplicação *Web* as interações com o usuário podem ser as operações de *GET* e *POST HTTP*[6]. As ações da camada de controle são baseadas na interação com o usuário e no resultado das ações na camada de modelo. O padrão MVC sugere que todo o fluxo de dados passe pela camada de controle, considerando que um objeto da camada de visualização não deve interagir com os objetos da camada de modelo diretamente.

Na Figura 4 pode-se observar como ocorre o fluxo de dados entre as camadas do padrão MVC para a *Web*.

[6] *HyperText Transfer Protocol* é um protocolo usado para transferir páginas *Web* entre um servidor e um *browser*.

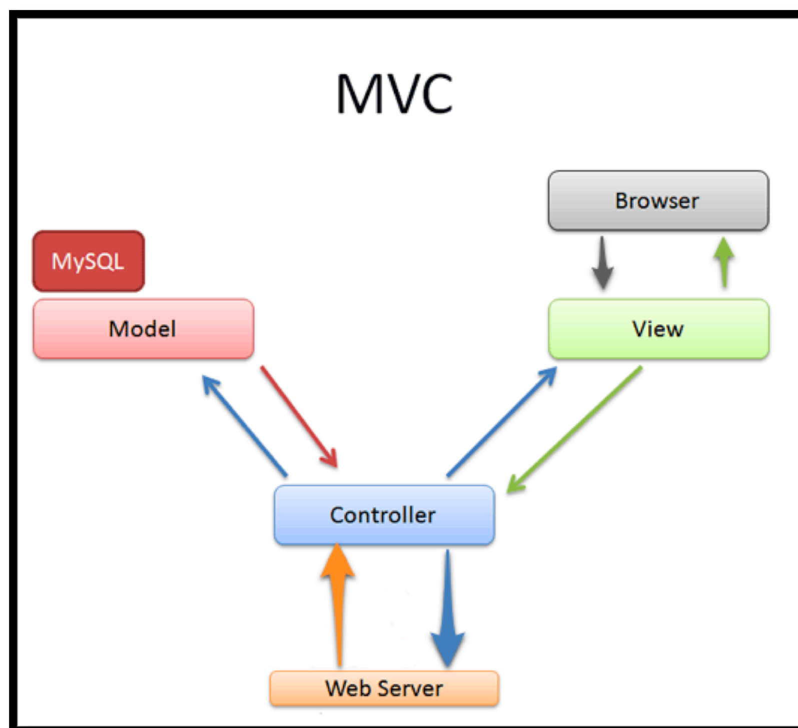


Figura 4. Diagrama conceitual do fluxo entre as camadas do padrão MVC para a *Web*.

2.6 EXTENSIBLE MARKUP LANGUAGE (XML)

Conforme Holzner (2001) o *eXtensible Markup Language* (XML) é uma recomendação da W3C[7] que permite a geração de linguagens de marcação para necessidades especiais. Seu propósito principal é a facilidade de compartilhamento de informações por meio da Internet.

De acordo com Mendes (2004), entre as linguagens baseadas em XML incluem-se o *eXtensible HyperText Markup Language* (XHTML), o *Scalable Vectorial Graphics* (SVG), dentre outros.

Algumas características do XML são:

a) separação do conteúdo da formatação;

---

[7] Órgão que regulamenta os padrões que são aplicados à internet.

b) simplicidade e legibilidade, tanto para humanos quanto para computadores;
c) possibilidade de criação de *tags* sem limitação;
d) criação de arquivos para validação de estrutura (DTD);
e) interligação de bancos de dados e aplicações distintas;
f) concentração na estrutura da informação, e não na sua aparência.

Conforme W3C (2008) o XML é um formato recomendado para a criação de documentos contendo dados organizados de forma hierárquica. Pode-se citar como exemplo de uso do XML os documentos de texto formatados, imagens vetoriais ou bancos de dados.

De acordo com Pitts-Moultis (2000) XML pode também ser usado para a comunicações entre aplicações distintas, desde que as aplicações em questão retornem XML como resposta de suas funções e interpretem o XML retornado pela outra aplicação. Este recurso é muito usado em bancos de dados e *Webservices*[8], que são transparentes à tecnologia usada na aplicação. Um exemplo de código XML pode ser visto na Figura 5, onde tem-se os dados de um currículo representado de forma estruturada.

Outro exemplo de aplicação prática de XML é o uso na codificação dos arquivos de saída originados pelas ferramentas case UML, o que teoricamente permite que outras ferramentas possam ler e processar os dados contidos nesses arquivos.

[8] Os *Webservices* são componentes que permitem às aplicações enviar e receber dados em formato XML.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<curriculo>
   <InformacaoPessoal>
      <DataNascimento>23-07-68</DataNascimento>
      <Nomecompleto>...</Nomecompleto>
      <Contatos>
         <Telefone>9999-9999</Telefone>
         <CorreioEletronico>email@email.com</CorreioEletronico>
      </Contatos>
      <Nacionalidade>brasileiro</Nacionalidade>
      <Sexo>M</Sexo>
   </InformacaoPessoal>
   <objetivo>Atuar na area de TI</objetivo>
   <Experiencia>
      <Cargo>Suporte tecnico</Cargo>
      <Empregador>Empresa, Cidade - Estado</Empregador>
   </Experiencia>
   <Formacao>Superior Completo</Formacao>
</curriculo>
```

Figura 5. Exemplo de arquivo XML.

2.7 MODEL DRIVEN ARCHTETURE (MDA)

Conforme Fowler (2000) a MDA é uma estratégia padrão para usar a UML como linguagem de programação. Este padrão é controlado pelo *Object Management Group* (OMG).

De acordo com Mellor et al. (2005) a MDA realiza a divisão do trabalho de desenvolvimento em duas áreas principais: o *Platform Independent Model* (PIM) e o *Platform Specific Model* (PSM).

O PIM é o componente que descreve todo o modelo de negócios do sistema, sem se preocupar em qual plataforma ou tecnologia será implementado, ou seja, é um modelo da UML independente de tecnologia específica. Algumas ferramentas podem realizar a transformação do PIM em PSM.

O PSM combina a especificação do modelo PIM com especificações de uma determinada plataforma, recebendo detalhes da construção do sistema baseado na tecnologia escolhida. Esse modelo é o de mais baixo nível de abstração e seus elementos estão prontos para a geração de código. Em outras palavras, o PSM é um modelo de sistema destinado a um ambiente de execução específico (MELLOR et al., 2005).

De acordo com Fowler (2000) para se construir um sistema utilizando MDA, primeiro deve-se começar criando um PIM do sistema que se quer desenvolver. Caso deseja-se que esse sistema seja executado em Ruby[9] ou em Python, deve-se utilizar as ferramentas apropriadas para criar dois PSMs: um para cada plataforma.

Pode-se observar na Figura 6 qual o fluxo de trabalho para a conversão de códigos utilizando a metodologia MDA.

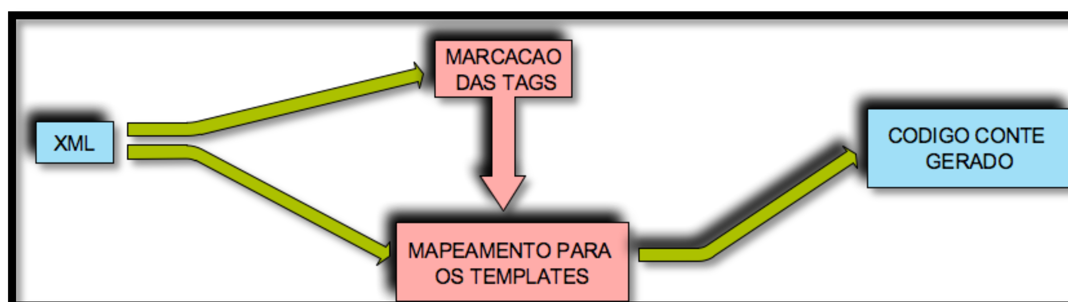


Figura 6. Fluxo de trabalho de um processo utilizando MDA.
Fonte: Mellor et al. (2005)

2.8 HERANÇA

Conforme Lutz e Ascher (2003) o objetivo da herança é criar um

---

[9] Ruby é uma linguagem de programação interpretada, com tipagem dinâmica e forte, orientada a objetos e semelhante ao Perl, SmallTalk e Python.

relacionamento entre classes, de forma a permitir que compartilhem atributos e operações. É usada geralmente para evitar que classes que possuem atributos ou métodos semelhantes sejam repetidamente criadas. Do ponto de vista de herança, existem dois tipos de classes: as *superclasses* que contém as características gerais e as *subclasses* que herdam as características definidas nas superclasses.

Pode-se citar como exemplo, uma superclasse chamada *Pessoas* definida pelos dados comuns de uma pessoa, como nome, idade, sexo e uma subclasse que herde essas características, como *Aluno*. Nesse caso pode-se definir que *Aluno* implementa todos os atributos da superclasse mais os atributos dele próprio. Exemplo: *nome, idade, sexo, codigoAluno, serie, turma, sala*.

A representação de herança na UML é exibida na Figura 7.

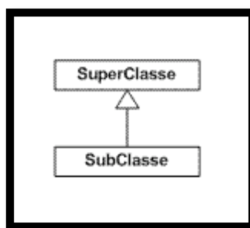


Figura 7. Representação de herança de classes na UML.

O próximo capítulo abordará um dos assuntos mais importantes para a compreensão e desenvolvimento deste trabalho: o padrão UML e outros padrões derivados que formam a base para o funcionamento da ferramenta desenvolvida.

3 UNIFIED MODELING LANGUAGE (UML)

Segundo Fowler (2000) a UML é um padrão formado por uma família de notações gráficas, que auxiliam os desenvolvedores na descrição e no projeto de sistemas de *software*, preferencialmente os sistemas construídos baseados na Orientação a Objetos (OO)[10].

As linguagens gráficas de modelagem já existem a muito tempo e o que fomenta o seu uso cada vez mais crescente é o fato de que as linguagens de programação não possuem um nível de abstração suficientemente alto para permitir as discussões sobre o projeto pelos engenheiros de *software* e desenvolvedores (FOWLER, 2000).

Conforme Pender (2003) a UML é um padrão controlado pelo OMG que é formado por um conjunto de empresas. A finalidade da criação do OMG foi a necessidade de estabelecer padrões que suportassem interoperabilidade, principalmente em sistemas orientados a objetos.

A UML foi criada a partir da unificação de várias linguagens gráficas de modelagem orientadas a objetos que surgiram no final dos anos 80. Desde que surgiu, em 1997, a UML estabeleceu um padrão de representação gráfica de modelagem da época (FOWLER, 2000).

Será abordado no tópico a seguir os vários tipos de diagramas UML e quais os objetivos de cada um.

---

[10] OO é um paradigma de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de objetos.

3.1 TIPOS DE DIAGRAMAS UML

Segundo Fowler (2000) a UML 2 descreve 13 tipos de diagramas oficiais, conforme listado na Tabela 1. Cada diagrama representa uma parte ou funcionalidade de um projeto de *software*. Dessa forma, quanto maior o número de diagramas distintos para o mesmo projeto, mais fácil será visualizar qual o resultado final desse projeto e como desenvolvê-lo.

Tabela 1. Tipos de diagramas oficiais da UML.

Diagrama	Objetivo	Linhagem
Atividades	Comportamento procedimental e paralelo	UML 1
Casos de uso	Como os usuários interagem com um sistema	UML 1
Classes	Classes, características e relacionamentos	UML 1
Componentes	Estrutura e conexão de componentes	UML 1
Comunicação	Interação entre objetos, ênfase nas ligações	UML 1
Distribuição	Distribuição de artefatos nos nós	UML 1
Estruturas compostas	Decomposição de uma classe em tempo de execução	UML 2
Máquinas de estado	Como os eventos alteram um objeto no decorrer de sua vida	UML 1
Objetos	Exemplo de configurações de instâncias	UML 1
Pacotes	Estrutura hierárquica em tempo de compilação	UML 1
Seqüência	Interação entre objetos, ênfase na seqüência	UML 1
Sincronismo	Interação entre objetos, ênfase no sincronismo	UML 2
Visão geral da interação	Mistura de diagramas de seqüência e de atividades	UML 2

Fonte: Fowler, M. (2000)

No próximo item serão abordados os diagramas de classes, usados na ferramenta desenvolvida para a geração de código Python.

3.2 DIAGRAMA DE CLASSES

Segundo Fowler (2000) os diagramas de classes realizam a descrição dos tipos de objetos presentes no sistema e os diversos tipos de relacionamentos existentes entre eles. Os diagramas de classes também determinam quais as propriedades, as operações e os relacionamentos de uma classe e descreve as restrições aplicadas à maneira com que os objetos se conectam.

Nos próximos tópicos serão apresentados os conceitos e outras características relacionadas a diagramas de classes.

3.2.1 Atributos

De acordo com Pender (2003) os atributos descrevem uma propriedade na forma de uma linha de texto dentro da caixa da representação da classe.

Um exemplo de atributo é descrito a seguir:

- *nome: String [1]*

No exemplo descrito acima, apenas o *nome* é obrigatório.

Outras considerações relevantes relacionadas a esse exemplo são:

a) o marcador de visibilidade indica se o atributo é considerado público (+), privado (-) ou protegido (#). Nesse caso ele é privado (-);

b) o nome de um atributo corresponde ao nome de um campo em uma linguagem de programação;

c) o tipo do atributo indica uma restrição sobre o tipo de objeto que pode ser colocado no atributo. Pode-se considerar isso como o tipo de um campo em uma linguagem de programação.

3.2.2 Operação

Conforme Pender (2003) Operações são as ações que uma classe pode realizar. As operações correspondem aos métodos presentes em uma classe.

Um exemplo de operação é descrito a seguir:

+ saldoEm (data: Date): Dinheiro

Outras considerações relevantes relacionadas a esse exemplo são:

a) o marcador de visibilidade é público (+), privado (-) ou protegido (#);
b) o nome é uma seqüência composta de caracteres;
c) a lista de parâmetros entre parêntesis, é a lista de parâmetros da operação;
d) o tipo de retorno (*Dinheiro*) é o tipo do valor retornado, se houver um.

Os diagramas de classes serão empregados nesse trabalho para a elaboração do projeto, por ser de fácil entendimento, ser um dos diagramas da UML mais utilizados e conhecido e por representar de forma geral o sistema modelado pelas classes do

digrama.

O capítulo a seguir descreve o conceito de *frameworks Web* e qual a importância deles para os desenvolvedores *Web*. Também é apresentado o *framework Web* Django, o qual se foca esse trabalho.

4 FRAMEWORK WEB

De acordo com Daly (2007) os *frameworks Web* surgiram como forma de auxiliar no desenvolvimento e administração de aplicações dinâmicas conforme os padrões de desenvolvimento para a *Web*. O objetivo é que o *framework* forneça um ponto de partida para o processo de desenvolvimento, de forma que o desenvolvedor se preocupe exclusivamente com o desenvolvimento da lógica de negócio da aplicação, sem precisar se preocupar com os aspectos de segurança e organização básicos da estrutura.

Conforme Thomaz (2007) durante o acesso a uma aplicação *Web*, o *browser* estabelece uma comunicação constante com a aplicação, de forma a estabelecer algum acesso de escrita ou leitura. A cada requisição a aplicação identifica a operação a ser efetuada e se o usuário possui privilégios suficientes sobre a aplicação para realizar determinada ação. Caso possua os privilégios necessários, a operação é realizada e a reposta da operação é enviada ao usuário solicitante, por meio de seu número IP[11], contido nos cabeçalhos das solicitações HTTP.

A Figura 8 mostra de forma mais clara como é realizada a comunicação entre o *browser* e uma aplicação *Web*.

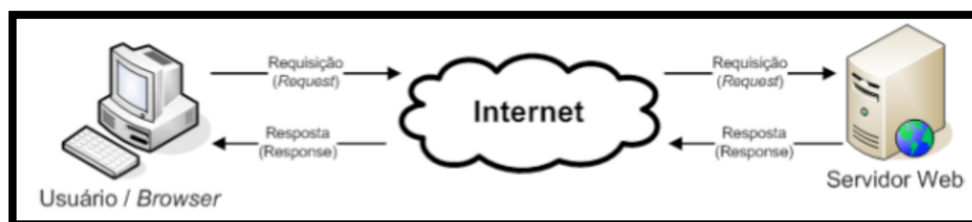


Figura 8. Funcionamento de uma aplicação Web.
Fonte: Thomaz (2007)

---

[11] Protocolo de comutação de pacotes de dados, utilizado para encaminhar e transportar informação na Internet.

Conforme Moore, Budd e Wright (2007) um *framework* de desenvolvimento de aplicações é uma estrutura de suporte com uma base definida de tal forma que se possa realizar o desenvolvimento de *software* muito mais rapidamente. Geralmente, um *framework* possui diversos tipos de ferramentas que auxiliam o desenvolvedor no processo de desenvolvimento, tais como um sistema integrado de permissões, autenticação, fórum, gerenciamento de usuários, dentre outros.

Um *framework* procura reunir funcionalidades comuns a várias aplicações, para que o desenvolvedor possa se concentrar no problema em questão, que pertence ao domínio específico da aplicação. Em uma aplicação *Web* baseada no padrão MVC, a comunicação entre os objetos do *Model,* da *View* e do *Controller* é semelhante, independente do domínio do problema da aplicação (DALY, 2007).

Conforme Thomaz (2007) a utilização de um *framework* no processo de desenvolvimento de aplicações traz várias vantagens, tais como:

a) maximização do re-uso de códigos;

b) diminuição do tempo de desenvolvimento;

c) redução de custos;

d) estabilização do código, devido ao uso em várias aplicações;

e) desenvolvimento voltado a adicionar valor à aplicação;

f) melhor consistência e compatibilidade entre aplicações;

g) diminuição e facilidade de manutenção.

Outras vantagens que podem ser citadas quando se faz uso de um *framework Web*:

a) constantes atualizações no conteúdo de forma fácil e rápida;

b) unificação e integração do conteúdo;

c) agilidade na criação de novos tipos de conteúdo;

d) não necessidade de recompilar todo o código a cada alteração (no

caso de linguagem interpretada / script);

e) resposta rápida para demandas de mudança;
f) sistema de *Template*;
g) mapeamento objeto relacional (não é necessário conhecimentos em SQL);
h) sistema de autenticação e permissões embutido, não sendo necessário construir um sistema de autenticação;
i) alguns *frameworks* permitem a tradução para várias línguas, das páginas contidas no site. Não é necessário manter dois sites diferentes para cada país, mas apenas um arquivo de tradução para cada língua. O design é o mesmo, mas os textos são renderizados de forma diferente, dependendo da configuração de idioma do *browser* que está fazendo o acesso ao site.

Dentre os *frameworks Web* desenvolvidos tem-se o Django, que é utilizado no desenvolvimento de aplicações *Web* dinâmicas.

Esse trabalho não se destina a explicar ou fazer comparações com outros tipos de *frameworks Web*, apenas apresenta o conceito geral de *frameworks Web*, com foco no *framework Web* Django.

4.1 FRAMEWORK WEB DJANGO

Conforme Daly (2007) o Django é um *framework Web open source* escrito em Python, que permite desenvolver aplicações *Web* de forma ágil. Foi desenvolvido originalmente pela *Word Online*, o departamento *Web* de um jornal em Lawrence,

Kansas, nos EUA, mas atualmente é mantido por um grupo internacional de voluntários. O Django possui uma documentação detalhada e uma interface de administração que facilita o gerenciamento das aplicações *Web* desenvolvidas. O Django inclui todos os componentes necessários para se construir uma aplicação *Web* básica, como o Mapeador Objeto Relacional (do inglês *Object-Relational Mapper*, ORM) que abstrai a camada de banco de dados, um sistema de *templates* para a inserção de conteúdos dinâmicos nas páginas *Web* e um local onde fica toda a lógica de negócios da aplicação.

Uma das características do Django é poder ser estendido de forma a agregar inúmeros outros tipos de funcionalidades, como enquetes, livro de visitas, fóruns, dentre outros, as quais são chamadas de *apps*.

No próximo tópico serão apresentados os conceitos gerais sobre o Django, como camada de modelo, interface de administração, sistema de *templating,* variáveis e filtros.

4.1.1 Camada de Modelo do Django

Segundo Holovaty e Moss (2007) todo o código referente à camada de modelo de uma aplicação *Web* Django, pode ser encontrado em um único lugar: o arquivo *models.py*. Toda a parte de definição dos campos, tabelas e tipos de dados, se encontra nesse arquivo. Esta é uma das características mais importantes do princípio *Don't Repeat Yourself* do Django, – que prega a não repetição de código em seus programas - pois o modelo é descrito em um único arquivo e os detalhes sobre como os dados serão persistidos em um banco de dados ficam escondidos, não sendo necessário

o desenvolvedor se preocupar com isso.

No ORM do Django, uma tabela é definida como uma classe que herda de *django.db.models.Model*, conforme mostra o exemplo da Figura 9, onde tem-se três classes que herdam de *models.Model.*

De acordo com Django Project (2008b) alguns tipos de campos de classe, como o *CharField*, possuem parâmetros requeridos, como *maxlength*. Muitos destes parâmetros informam não apenas o tipo dos atributos que serão criados no banco de dados, mas também a forma que o campo será exibido na interface de administração e conseqüentemente definem as rotinas de validação providas pelo sistema. O Django permite que os campos sejam opcionais (*blank* = *True*) e que valores nulos sejam diferenciados de campos vazios (*null* = *True*).

Uma vez que as tabelas e campos tenham sido criados no banco de dados, com a utilização do comando *manage.py syncdb*, os dados podem ser recuperados do banco de dados, usando os métodos providos pelo *models.Model* (DALY, 2007).

4.1.2 Interface de Administração do Django

Conforme Holovaty e Moss (2007) na maioria dos *frameworks Web* que seguem o padrão MVC, uma vez que a camada de modelo esteja definida, o desenvolvedor passa a construir a camada de visualização. Dependendo do projeto, esta camada pode ser uma interface de usuário pública ou mesmo uma interface de administração.

Para ativar a interface de administração, deve-se criar no diretório da aplicação um arquivo chamado *admin.py* com o código exibido na Figura 10.

```
# coding: utf8
from django.db import models
from django.contrib import admin

class Pais(models.Model):
    nome = models.CharField(max_length=500)
    class Meta:
        verbose_name_plural = "paises"
        ordering = ['nome']

    def __str__(self):
        return "%s" % (self.nome)


class Regiao(models.Model):
    nome = models.CharField(max_length=500)
    pais = models.ForeignKey(Pais)
    class Meta:
        verbose_name_plural = "regioes"

    def __str__(self):
        return "%s" % (self.nome)

class Vinho(models.Model):
    nome = models.CharField(max_length=500)
    preco = models.PositiveSmallIntegerField()
    ano = models.IntegerField('Ano da Safra', null=True)
    comentario = models.TextField(blank=True)
    regiao = models.ForeignKey(Regiao,null=False)

    def __str__(self):
        return "%s %d" % (self.nome, self.ano)
```

Figura 9. Exemplo de camada de modelo do Django.

Com a inserção do código da Figura 10, é possível visualizar os objetos que existem no banco de dados em um estilo de visualização padronizado, conforme mostra a Figura 11.

```
from exemplo.vinhos.models import *
from django.contrib import admin

admin.site.register(Vinho)
```

Figura 10. Adicionando a classe *Vinho* na interface de administração do Django.

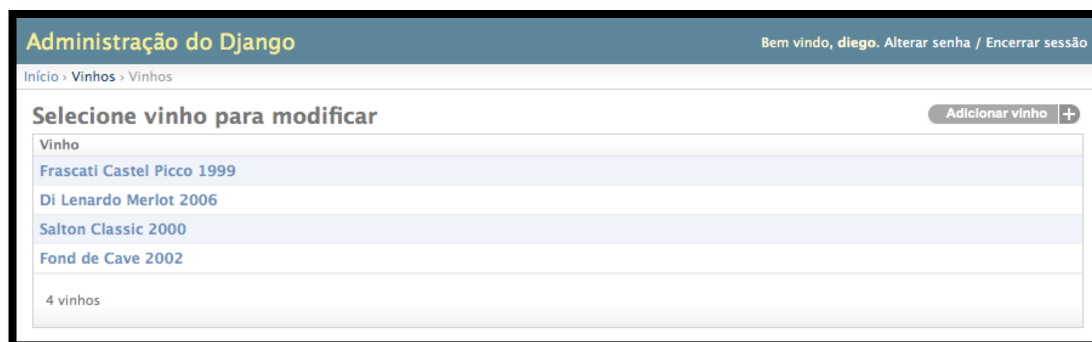


Figura 11. Listagem padrão de objetos cadastrados, na interface de administração do Django.
Fonte: Interface de administração do Django.

Um exemplo de customização da interface de administração, a partir da edição do arquivo *admin.py*, pode ser visto na Figura 12.

```
from exemplo.vinhos.models import *
from django.contrib import admin

class VinhoAdmin(admin.ModelAdmin):
    list_display = ('nome', 'preco', 'ano', 'regiao')
    list_filter = ['preco', 'ano']

admin.site.register(Vinho, VinhoAdmin)
admin.site.register(Regiao)
admin.site.register(Pais)
```

Figura 12. Customização da aparência da interface de administração de uma classe Django.

Após a modificação sugerida no arquivo *models.py*, conforme Figura 12, será exibido o resultado demonstrado na Figura 13, demonstrando a exibição dos objetos com organização por colunas e filtros pré definidos.

Ao selecionar um objeto previamente cadastrado, pela interface de administração do Django, é exibida uma página de edição com campos de formulário HTML gerados automaticamente para cada campo editável da base de dados, conforme Figura 14.

Os campos opcionais são exibidos em cinza, e os campos de *foreign-key* possuem um botão para adição perto da caixa de seleção, que possibilita aos editores de

conteúdo ou desenvolvedores, expandir facilmente o número de opções do menu.

Figura 13. Listagem dos campos customizados da interface de administração do Django.
Fonte: Interface de administração do Django.

Os modelos de classes do Django possuem um conjunto de validadores associados ao seu tipo de dado. Caso haja uma tentativa de entrada de um valor não inteiro na coluna *preco*, irá ocorrer um erro de validação, conforme é exibido na Figura 15.

Figura 14. Visão padrão ao editar um objeto do Django.
Fonte: Interface de administração do Django.

Algumas aplicações requerem customizações na interface de administração, mas para pequenos projetos e acesso a dados básico, talvez não seja necessário customizá-la (DALY, 2007).

Figura 15. Mensagem de erro gerada pela validação automática de campos do Django.
Fonte: Interface de administração do Django.

No próximo tópico será demonstrado como funciona o sistema de *templates* do Django, que são os responsáveis em definir como os dados serão apresentados ao usuário da aplicação.

4.1.3 Sistema de templating do Django

De acordo com Holovaty e Moss (2007) o Django possui um sistema de *templating* que tem como objetivo proporcionar uma maior rapidez e flexibilidade no desenvolvimento. O formato de saída dos *templates* do Django não se restringe apenas a HTML, como geralmente ocorre em outros sistemas de *templating*, mas também pode gerar XML, texto puro, dentre outros.

Assim como outras linguagens de *templating*, o Django determina os códigos funcionais com o uso de caracteres especiais, conforme mostra a Figura 16.

```
{% if erro %}
    <div class="erro">
        {{ erro }}
    </div>
{% endif %}
```

Figura 16. Exemplo de uso de variáveis no Django.

Outra característica do Django é que ele não permite a execução de códigos Python dentro do *template*, separando totalmente a camada de controle da camada de visualização.

4.1.4 Variáveis

Conforme Daly (2007) o sistema de variáveis do Django é mais expansível que o da linguagem Python. Quando o Django encontra *variável.atributo,* tenta as seguintes operações, nessa ordem:

a) busca em dicionário;

b) busca por atributo;

c) chamada de método;

d) busca por índice de lista (*nome_da_lista.3* == *nome_da_lista[3]*).

Se nenhuma dessas operações forem resolvidas, ou se a variável não existir, o Django retorna uma *string* vazia.

4.1.5 Filtros

De acordo com Holovaty e Moss (2007) os filtros estendem o poder do sistema de variáveis permitindo que as variáveis sejam processadas por uma série de funções pré-definidas. Os filtros podem ser encadeados, utilizando a notação "|" (*pipe*), bastante familiar aos desenvolvedores e administradores Unix. Um exemplo pode ser visto na Figura 17.

```
<h1> {{ vinho.nome | lower | }} </h1>
```

Figura 17. Exemplo de filtros do Django.

O filtro encadeado da Figura 17, obtém o nome do vinho e o converte para caixa baixa.

Existe um número variado de filtros prontos para serem utilizados no Django e até mesmo pode-se criar filtros customizados de acordo com a necessidade do desenvolvedor. Demais tipos de filtros podem ser encontrados em Django Project (2008a).

No próximo capítulo serão apresentados os trabalhos correlatos, similares ao presente trabalho.

5 TRABALHOS CORRELATOS

Nos próximos tópicos serão demonstrados alguns trabalhos realizados na área de MDA, bem como as características de desenvolvimento de cada um.

5.1 PROCESSO RAPDIS

Na tese de mestrado defendida por Morgado (2007) é demonstrando o processo RAPDIS que auxilia o desenvolvimento de Sistemas de Informação, oferecendo um suporte completo ao MDA, ou seja, da produção dos modelos específicos dessa abordagem, que são CIM (Modelo Independente de Computação), Modelo Independente de Plataforma (PIM) e o Modelo Específico de Plataforma (PSM), até a geração do código fonte, que é realizada por meio de uma ferramenta específica.

5.2 ABORDAGEM XIS

Trabalho realizado por Silva (2003), tem como objetivo principal o desenvolvimento de *software* baseado em modelos com o apoio de MDA e centrado em arquiteturas de *software* e baseado em técnicas de geração automática de templates disponíveis. Essa abordagem preocupa-se apenas com a elaboração dos modelos PIMs, seguindo o padrão MVC. As transformações desses modelos para uma determinada plataforma são definidas pelos arquitetos quando utilizam os *templates* correspondentes

a uma determinada arquitetura de *software* e, finalmente, pelos desenvolvedores quando selecionam os modelos e os *templates* necessários para efetuar as transformações em componentes de *software* correspondentes.

5.3 PROCESSO DE DESENVOLVIMENTO BASEADO EM COMPONENTES ADAPTADOS AO MDA

Tese de mestrado defendida por Sousa (2004) que inclui o tratamento explícito dos requisitos não-funcionais por meio do refinamento da arquitetura de *software* e do uso de um modelo de estruturação de componentes independentes de plataforma, mais especificamente o *Component Structuring Model for Object-Oriented Systems* (COSMOS). Esse modelo permite um mapeamento entre as abstrações de um diagrama UML para a construção de códigos em plataformas de componentes específicos.

5.4 DESENVOLVIMENTO DE SIG PARA WEB UTILIZANDO MDA

O artigo foi publicado por Mello, Zimbrão e Souza (2007) e o objetivo deste trabalho é aumentar a produtividade no desenvolvimento de Sistemas de Informação Geográfica (SIG) para *Web* através do uso do padrão MDA. No trabalho, foi definida uma extensão da UML que agrega ao modelo de classes informações geográficas utilizadas nos SIG. A partir desta extensão, foi proposto a implementação de um plug-in para a ferramenta *AndroMDA* que permite a geração automática dos arquivos de configuração do *MapServer*.

5.5 CRIAÇÃO DE COMPONENTES DINÂMICOS PARA JOOMLA POR MEIO DE UML

Trabalho de Conclusão de Curso defendido por Smania (2008), teve como objetivo o desenvolvimento de uma ferramenta de extração de informações da modelagem visual dos diagramas de classes UML e por meio desses, proporcionar a geração automática da estrutura de um componente do *framework Web* Joomla. A ferramenta foi desenvolvida utilizando a linguagem PHP 5.

O processo de geração do código é constituído dos seguintes passos:

a) o programador ou o engenheiro de software constrói o diagrama de classes do componente que pretende ser criado na ferramenta UML de sua preferência (desde que permita a exportação dos diagramas no formato XMI 1.3) e o exporta para o formato XMI;
b) em seguida, importa-se o arquivo de saída correspondente ao diagrama na ferramenta desenvolvida;
c) a ferramenta desenvolvida extrai os dados do arquivo XMI e transforma-os em código fonte adaptado ao *framework* do Joomla, disponibilizando-o por meio de um arquivo compactado.

As conclusões realizadas pelo autor mostraram a viabilidade de sua ferramenta, que provê redução de tempo de programação, abstração da complexidade do código do componente, reutilização de códigos, entre outros. O autor também destaca a capacidade do uso de funções pré-definidas fazendo com que aumente a reutilização de códigos.

6 DJANGOGEN: MODELAGEM, ESTRUTURA E DESENVOLVIMENTO

O DjangoGen é uma ferramenta para geração de código Python da camada de modelo do Django, a partir de diagramas de classes UML da ferramenta case *Dia Modeler*. É possível também, criar *apps* customizadas com classes, métodos, dentre outros e disponibilizar na *Web* para que qualquer pessoa interessada na funcionalidade de sua *app* Django possa obtê-la e instalá-la pelos métodos convencionais.

Desenvolver uma *app* Django envolve o desenvolvimento de códigos em diversos arquivos de uma forma padronizada, para que seja possível utilizar os recursos do Django. Esses códigos, muitas vezes se repetem, o que pode gerar um gasto de tempo desnecessário para o desenvolvedor.

No sentido de automatizar o processo de geração, foi criada a ferramenta DjangoGen - fusão das palavras Django + *Generator* – que tem por finalidade extrair as informações pertinentes do arquivo XML gerado pelo modelador UML *Dia Modeler* a partir de um modelo UML de classes e realizar a conversão para código Python adaptado ao Django.

O sistema operacional utilizado no desenvolvimento e testes foi o MacOS X 10.5.5, um sistema Unix proprietário da empresa *Apple Inc.*[12], porém a ferramenta desenvolvida funciona em qualquer sistema operacional onde seja possível instalar e executar o Python.

O funcionamento da ferramenta DjangoGen segue o seguinte fluxo:

---

[12] http://www.apple.com/

a) o diagrama de classes contendo as classes básicas e seus componentes é projetado no modelador UML Dia Modeler e salvo no formato .dia compactado;

b) executa-se o DjangoGen e por meio da interface, importa-se o arquivo *.dia*, gerado no modelador UML *Dia Modeler*;

c) na interface do DjangoGen clica-se em *Gerar Código!* e em seguida o DjangoGen lê o arquivo XML gerado no *Dia Modeler*, extraindo as informações pertinentes e convertendo essas informações para código Python, adaptado ao *framework Web* Django, em forma de uma *app* Django;

d) após a geração do código Python, a *app* gerada é instalada.

O processo descrito nos itens acima, é representando pelo diagrama da Figura 18.

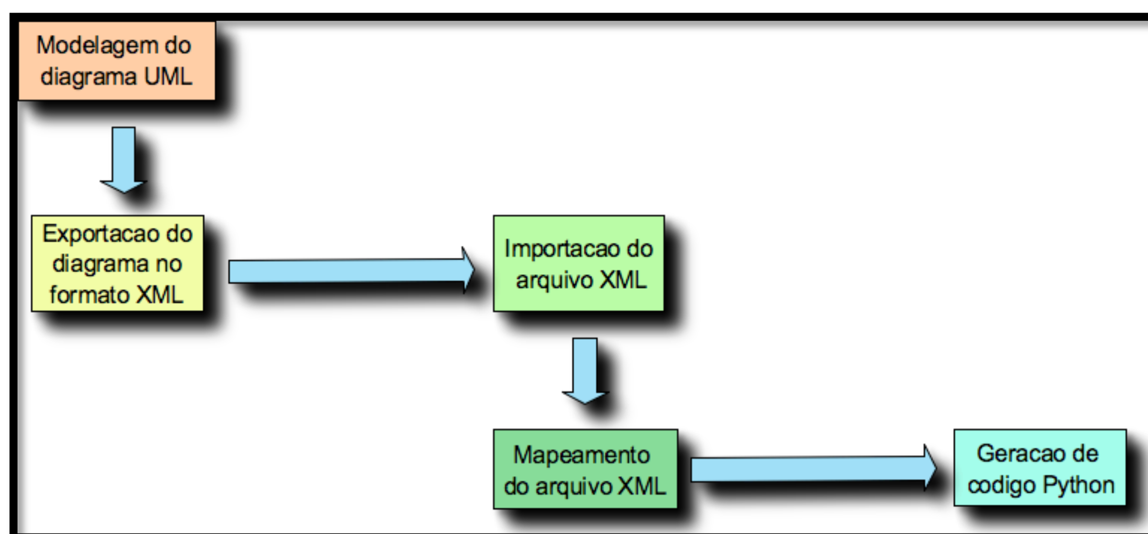


Figura 18. Fluxo de trabalho da ferramenta DjangoGen.

Como pode ser observado, seguindo esse fluxo de trabalho, o tempo despendido criando arquivos padrões e escrevendo a camada de modelo da *app* é ganho com o DjangoGen, além de criar-se parte da documentação da *app* gerada, na forma de

diagramas de classes UML.

6.1 METODOLOGIA DO TRABALHO DESENVOLVIDO

A metodologia para a realização do projeto foi composta das seguintes etapas:

a) definição do tema e escopo do trabalho;
b) realização do levantamento bibliográfico;
c) estudo dos conceitos de engenharia de *software*, dentre eles UML, XML, MVC e MDA;
d) compreensão da forma que a ferramenta MDA especifica a conversão de diagramas UML para código fonte;
e) escolha de um modelador UML baseando-se em dois pré-requisitos: característica multi plataforma e código XML organizado e de fácil entendimento. Nesse caso, após vários testes, foi escolhido o modelador UML *Dia Modeler*;
f) testes utilizando a biblioteca *xml.dom.minidom* do Python e os diagramas UML exportados pelo modelador *Dia Modeler;*
g) aplicação dos conhecimentos obtidos para a criação de uma ferramenta de conversão de diagramas UML do *Dia Modeler* para *apps* do Django;
h) aplicação de layout e estilo CSS à interface da ferramenta DjangoGen;
i) realização de diversos testes de geração de código e correção de bugs;
j) documentação da ferramenta desenvolvida, comentários no código e conclusão do trabalho teórico.

6.2 MODELAGEM DA FERRAMENTA DJANGOGEN

A modelagem de um projeto é de fundamental importância para o seu sucesso. Além de mostrar uma visão geral para a aplicação a ser desenvolvida, também é útil como complemento da documentação, de fundamental importância, seja qual for o porte do projeto.

Nos tópicos a seguir será possível visualizar a modelagem UML do projeto desenvolvido e entender melhor suas peculiaridades.

6.2.1 Diagrama de Componentes

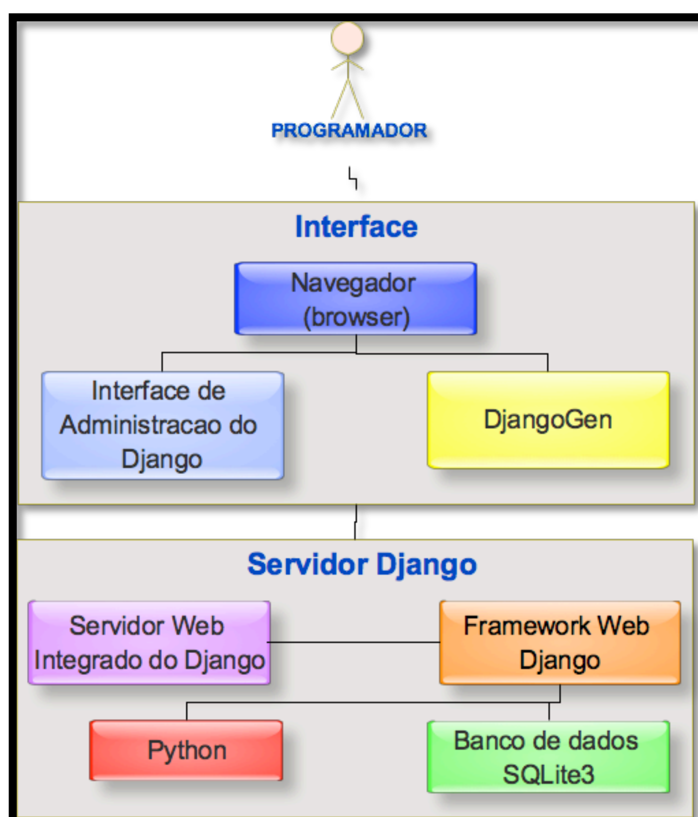


Figura 19. Diagrama de componentes da ferramenta DjangoGen.

O diagrama de componentes tem por finalidade mostrar os componentes físicos do sistema, como usuários, servidores, linguagens, entre outros.

O diagrama de componentes do sistema DjangoGen pode ser visto na Figura 19.

6.2.2 Diagrama de Atividades

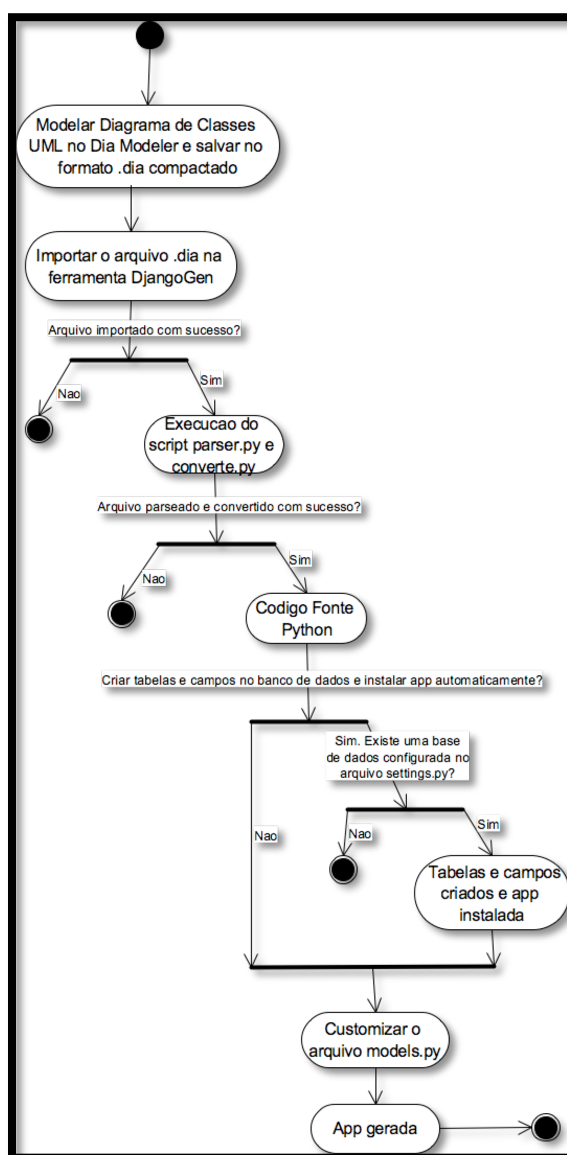


Figura 20. Diagrama de atividades da ferramenta DjangoGen.

O diagrama de atividades mostra o *workflow* de um sistema. Normalmente, demonstra um processo seqüencial de uma atividade para a outra (BOOCH; RUMBAUGH; JACOBSON, 2000).

O diagrama de atividades do sistema DjangoGen pode ser visto na Figura 20.

6.3 FERRAMENTA CASE UML DIA

A ferramenta case utilizada no desenvolvimento do trabalho foi o modelador UML *Dia Modeler*, pelos seguintes motivos:

a) ser uma ferramenta *free software / open source*, ou seja, licença de utilização gratuita e código fonte aberto;
b) facilidade na criação de diagramas;
c) facilidade no entendimento do XML gerado;
d) pode ser estendido em Python, a partir da criação de *plug-ins*;
e) ser multi plataforma;

Segundo Gnome (2008) o modelador UML *Dia Modeler* é capaz de exportar os diagramas UML em uma variedade de tipos de formatos diferentes, tais quais:

SVG, PNG, JPEG, dentre outros.

O *Dia Modeler* grava e carrega os arquivos de diagramas UML em um formato XML próprio, que é compactado com o *gzip*[13] por padrão (GNOME, 2008).

---

[4] gzip é a abreviação de GNU zip e é um Software Livre de compressão de arquivos, sem perda de dados.

6.4 REGRAS DE CRIAÇÃO DE DIAGRAMAS UML NO DIA

Para que seja possível o uso da ferramenta DjangoGen a partir de diagramas do *Dia Modeler*, é necessário que se observe alguns pontos ao projetar um diagrama UML, tais quais:

a) usar diagramas UML de classes;
b) o nome da classe informado no diagrama UML do *Dia Modeler* será mapeado para o nome da classe Django quando o código for gerado;
c) quando há herança entre duas classes, a classe pai é usada como classe base na classe descendente. Quando não há herança, será usado *models.Model* como superclasse;
d) os atributos no diagrama UML do *Dia Modeler* são mapeados como campos nas classes Django;
e) tipos de atributos de um diagrama UML do *Dia Modeler* são mapeados em um tipo de campo (*FieldType*) no Django;
f) valores de atributos de um diagrama UML do *Dia Modeler* são mapeados como parâmetros passados para o tipo de campo (*FieldType*) no Django;
g) é possível importar classes externas ao diagrama, apenas adicionando um elemento do *Dia Modeler* chamado *SmallPackage* e inserindo nele o pacote que se quer importar no código Python gerado posteriormente. Feito isso deve-se marcar a chave estrangeira (*ForeignKey*) como protegida (#).

Para que os tipos dos diagramas UML sejam mapeados corretamente em tipos do Django, o diagrama UML deve se limitar a conter os tipos listados na Tabela 2.

Tabela 2. Conversão de tipos do *Dia Modeler* em tipos do Django.

Dia Modeler	Django
Text	TextField
Date	DateField
Varchar	CharField
Int	IntegerField
Float	FloatField
Serial	AutoField
Boolean	BooleanField
Numeric	FloatField
timestamp	DateTimeField
Bigint	IntegerField
Datetime	DateTimeField
Time	TimeField
Bool	BooleanField

A Figura 21 mostra um diagrama UML do *Dia Modeler* que exemplifica todas as recomendações consideradas nos itens anteriores.

Após desenhar o diagrama UML seguindo as recomendações dos itens anteriores, o diagrama deve ser gravado no formato *.dia* compactado de forma que as informações do XML gravado possam ser processadas.

No item a seguir será exibida a estrutura de diretórios e arquivos que formam a ferramenta DjangoGen, explicado qual a importância de cada arquivo para a ferramenta.

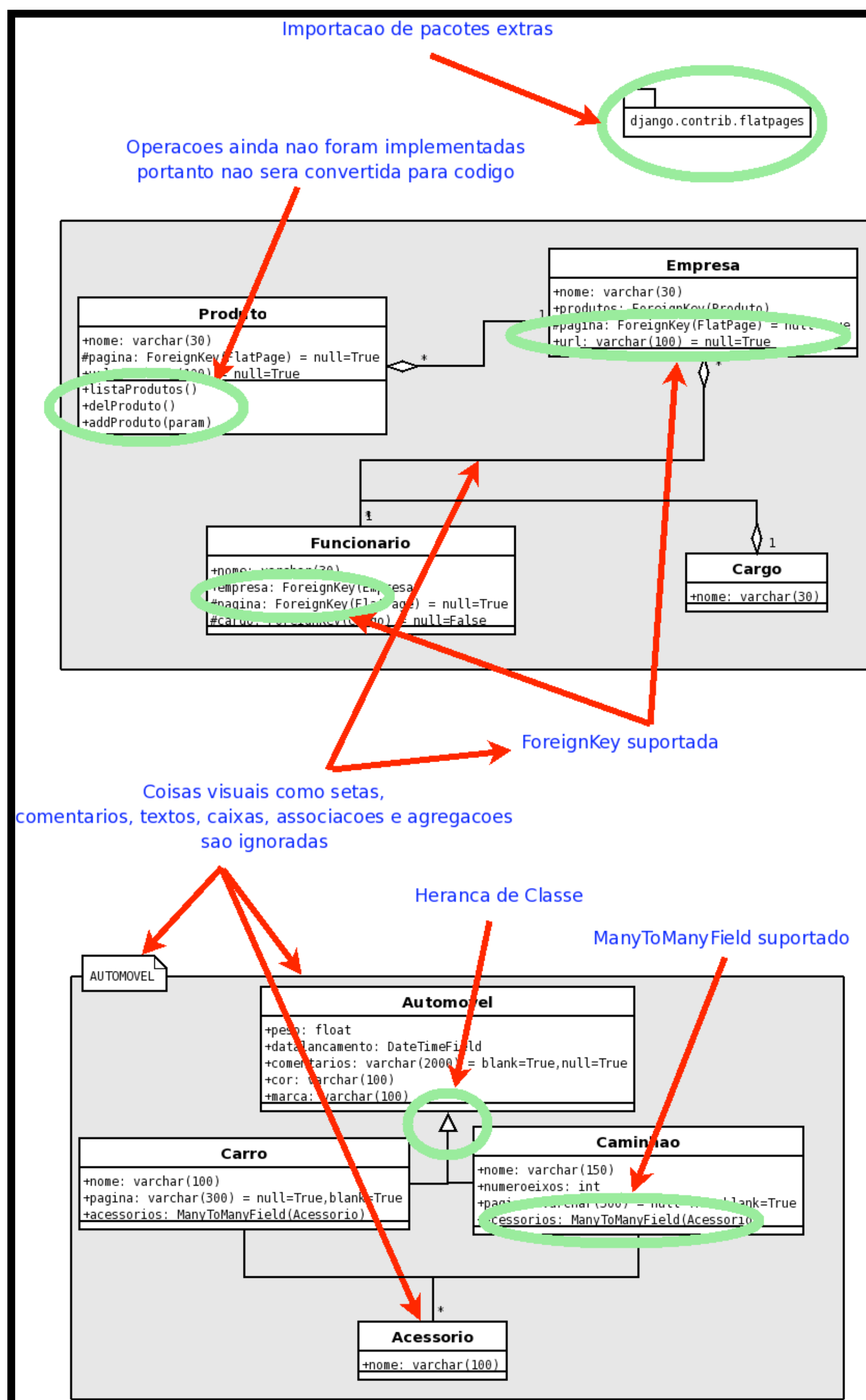


Figura 21. Diagrama UML de classes exemplo, que segue as recomendações de construção adequadas ao DjangoGen.

6.5 DESENVOLVIMENTO DE APPS PARA O FRAMEWORK WEB DJANGO

As *apps* são extensões que agregam funcionalidades específicas para um

site Django e possuem uma estrutura básica padrão que possibilita o seu uso como um módulo dentro de um site Django. O desenvolvimento de *apps* em Django requer o conhecimento das seguintes áreas:

a) python;
b) django;
c) orientação a objetos;
d) html;
e) bancos de dados (não obrigatório);
f) configuração de servidor *Web* (em caso de ambientes de produção).

Nos próximos tópicos serão abordadas as particularidades do desenvolvimento em Django, de modo que se possa melhor entender os arquivos e a importância de cada um no funcionamento da ferramenta.

6.5.1 Estrutura de arquivos de uma aplicação Django

Conforme Django Project (2008b) o Django segue um padrão de organização de código e esse padrão inclui a forma que os arquivos são organizados e o nome de cada um, de modo que um desenvolvedor Django entenda uma aplicação desenvolvida por outra pessoa, ou mesmo onde modificar algo em sua própria aplicação.

O Django, como comentado em capítulos anteriores, utiliza uma variante da metodologia MVC, chamada MTV onde cada camada fica separada em um arquivo diferente.

De acordo com Django Project (2008b) a construção de uma aplicação

Django é baseada nos seguintes arquivos:

a) *__init__.py*: necessário para informar ao Python que a pasta corrente contém um módulo Python executável;
b) *manage.py*: usado no gerenciamento das aplicações Django, permitindo criar novas *apps* dentro da aplicação principal, sincronizar a base de dados de acordo com a camada de modelo, criar usuários de acesso autenticado na aplicação, dentre outros;
c) *models.py:* contém toda a camada de modelo da aplicação, como as classes e os tipos de dados usados na mesma. Uma vez que se tenha esse arquivo configurado, o Django é capaz de gerar automaticamente as tabelas de banco de dados para uso na aplicação. Nesse arquivo também é definida a classe *Meta* que personaliza as opções e a forma que cada classe será exibida na interface de administração do Django;
d) *settings.py*: contém todas as configurações de ambiente da aplicação, como diretório dos arquivos de mídia, *templates*, *apps* instaladas, dentre outras configurações;
e) *urls.py*: contém o mapeamento de todas as URLs utilizadas na aplicação, podendo-se mapear diretórios e até mesmo fazer uso de expressões regulares no mapeamento de urls;
f) *views.py*: este arquivo agrega a lógica de negócios da aplicação, contendo todas as definições de funções utilizadas na aplicação e é utilizado pelo arquivo *urls.py* no redirecionamento de *URLs*. Em outras palavras, este arquivo define quais os dados que serão exibidos, enquanto os *templates* definem de que forma eles serão exibidos;

g) *foo.html*: geralmente toda aplicação possui algum tipo de apresentação gráfica. Nesses casos é necessário escrever um *template* destinado a exibição das informações manipuladas pelo arquivo *views.py*. Esse *template* poderá ser usado tanto em um fim específico quanto para um fim genérico, tudo dependerá de como ele será implementado. No Django os *templates* possuem uma sintaxe própria - conforme citado no item 4.1.3 - e não apenas código HTML puro.

6.5.2 Desenvolvimento em Django utilizando widgets

No sentido de minimizar a repetição e tornar o desenvolvimento mais ágil o Django permite a inclusão de funções prontas – *widgets* - para fins específicos. Exemplo: é possível criar um objeto *select box* e integrá-lo em qualquer *template* facilmente, escrevendo apenas no *views.py* o código exibido na Figura 22.

```
escolhas = [(1,'A'),(2,'B')]
escolhas = forms.CharField(widget=forms.Select(choices=escolhas))
```

Figura 22. Exemplo de uso de *widgets* do Django.

Outro exemplo seria a criação de um campo destinado ao *upload* de arquivos e uma caixa de textos, realizados pelo código da Figura 23.

```
uploadFile = forms.Field(widget=forms.FileInput)
nomeApp = forms.CharField(widget=forms.TextInput(attrs={'size':'40'}))
```

Figura 23. Exemplo de uso de *widgets* do Django.

Os *widgets* Django foram usados no presente trabalho no *template* *upload.html*, conforme o código da Figura 24, onde é demonstrado o código para

renderizar um formulário de upload de arquivos.

```
<form action="/upload/" method="POST" enctype="multipart/form-data">
    Localize o arquivo do Dia: {{ form.uploadFile }}
    <input type=submit name=upload value= ' Fazer Upload '> <br><br>
</form>

<form action="/gera/" method="POST" enctype="multipart/form-data">
    Informe um nome para a APP Django a ser gerada: {{ form.nomeApp }}
    <br><br><input type=submit name=submit value= ' Gerar Código! '> <br><br>
```

Figura 24. Uso de *widgets* no presente trabalho.

Esses códigos são responsáveis por inserir o campo de *upload* de arquivo e um campo de caixa de texto na interface do DjangoGen.

Esses e outros *widgets* podem ser encontrados na página de documentação do Django[14].

6.6 A ESTRUTURA DA FERRAMENTA DJANGOGEN

A estrutura de arquivos da ferramenta DjangoGen é composta pelos arquivos base descritos no item 6.5.1, mais alguns outros diretórios que serão descritos a seguir:

a) *apps*: diretório destinado a armazenar as *apps* geradas pelo DjangoGen;
b) *mda:* pasta que contém o módulo que realiza o *parsing* do XML e a conversão para código Python;
c) *model:* pasta reservada ao armazenamento do arquivo de diagrama importado e processado pelo DjangoGen e pelo banco de dados da

---

[14] Disponível em: http://docs.djangoproject.com/en/dev/ref/forms/widgets/#module-django.forms.widgets

aplicação, que nesse caso é *sqlite3*[15]. Também pode ser usada no armazenamento do modelo salvo diretamente a partir do *Dia Modeler*;

d) *templates:* pasta onde se encontram os arquivos que definem como os dados serão exibidos, ou seja, a interface da aplicação.

Toda a estrutura de diretórios da ferramenta DjangoGen pode ser visualizada na Figura 25.

Como pôde-se observar até então, a ferramenta segue o padrão MVC, separando a parte lógica, de controle e de apresentação em camadas distintas. Na verdade, conforme a FAQ do Django[16] o padrão de separação de camadas adotado pelo Django é chamado MTV. Esse padrão foi definido pelos criadores do Django que o consideram mais adequado que o padrão MVC.

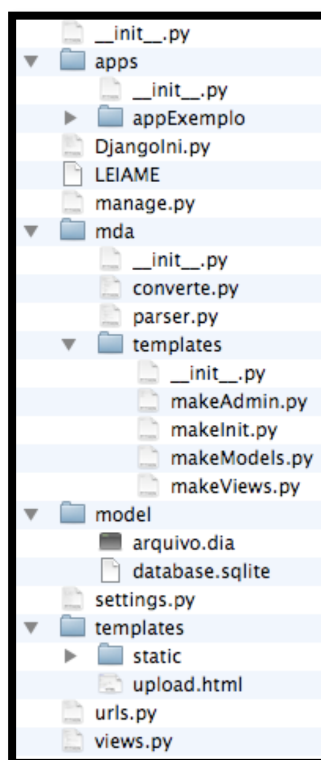


Figura 25. Estrutura de diretórios do DjangoGen.
Fonte: Finder (visualizador de pastas e arquivos do MacOS X).

[15] SQLite é uma biblioteca C que implementa um banco de dados SQL embutido, onde a biblioteca SQLite lê e escreve diretamente para o arquivo de banco de dados no disco.
[16] http://docs.djangoproject.com/en/dev/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names

6.7 EXTRAÇÃO DE INFORMAÇÕES DO ARQUIVO UML / XML DO DIA MODELER

Ao observar o arquivo XML gerado pelo *Dia Modeler* na Figura 26, pode-se verificar que existe um padrão para as *tags* XML. Por meio desse padrão é possível extrair as informações necessárias do XML e realizar a conversão para código Python.

A partir do XML de exemplo demonstrado na Figura 26 é possível, com o auxílio da biblioteca Python *xml.dom.minidom*, extrair os dados necessários para a geração do código Python. O código Python responsável por abrir o arquivo XML salvo pelo *Dia Modeler* e transformar o fluxo de dados em um objeto Python para manipulação pela ferramenta DjangoGen é demonstrado na Figura 27.

```
<?xml version="1.0" encoding="UTF-8"?>
<dia:diagram xmlns:dia="http://www.lysator.liu.se/~alla/dia/">
  <dia:diagramdata>
    <dia:attribute name="background">
      <dia:color val="#ffffff"/>
    </dia:attribute>
    <dia:attribute name="pagebreak">
      <dia:color val="#000099"/>
    </dia:attribute>
    <dia:attribute name="paper">
      <dia:composite type="paper">
        <dia:attribute name="name">
          <dia:string>#A4#</dia:string>
        </dia:attribute>
        <dia:attribute name="tmargin">
          <dia:real val="2.8222000598907471"/>
        </dia:attribute>
        <dia:attribute name="bmargin">
          <dia:real val="2.8222000598907471"/>
        </dia:attribute>
        <dia:attribute name="lmargin">
          <dia:real val="2.8222000598907471"/>
        </dia:attribute>
        <dia:attribute name="rmargin">
          <dia:real val="2.8222000598907471"/>
        </dia:attribute>
        <dia:attribute name="is_portrait">
          <dia:boolean val="true"/>
        </dia:attribute>
```

Figura 26. XML de um diagrama de classes de exemplo salvo pelo *Dia Modeler*.
Fonte: Arquivo UML do *Dia Modeler*, descompactado.

Toda a tarefa de mapeamento das informações contidas nas *tags* XML do arquivo do *Dia Modeler* se encontra no *script* Python */mda/parser.py* e é realizado pela função *parser(pim)* onde o parâmetro *pim* é o arquivo UML do *Dia Modeler*, mapeado em um objeto Python. A função *parser(pim)* não é executada diretamente, mas invocada pela função *imprimePSM(pim, model_file)* do script de geração de códigos encontrado em */mda/converte.py*.

```
nomeApp = request.POST['nomeApp']
arquivo_dia = "/".join(TEMPLATE_DIRS[0].split('/')[:-1]) + '/model/arquivo.dia'

try:
    f = codecs.open(arquivo_dia,"rb")
    # Os arquivos do Dia UML, são compactados com o gzip
    data = gzip.GzipFile(fileobj = f).read()
    pim = parseString(data)
    imprimePSM(pim, nomeApp)
    if (request.POST['gera'] == 'Gerar Codigo e instalar app'):
        instalaApp(nomeApp)
        call_command('syncdb')

    return HttpResponseRedirect('/upload?arq=%s' % unicode(u'Arquivo gerado com sucesso!'))
except:
    return HttpResponseRedirect('/upload?err=%s' % unicode(u'Erro na geração do arquivo!')
```

Figura 27. Função que lê o arquivo UML do *Dia Modeler* e converte em objeto Python.
Fonte: Trecho do arquivo *views.py* da ferramenta DjangoGen.

6.8 GERAÇÃO DE APPS DJANGO A PARTIR DE DIAGRAMAS UML / XML DO DIA MODELER

A partir da extração das informações dos arquivos XML, explicado no item 6.7 é possível realizar a geração dos arquivos de código Python.

O módulo responsável por gerar o código Python a partir do XML já importado pelo módulo *parser.py* é o script */mda/converte.py*, que conseqüentemente é usado pela função *geraEstruturaApp* do módulo *views.py.* A função *geraEstruturaApp*

é exibida na Figura 27 onde é utilizada a função *imprimePSM(pim, model_file)* do módulo *converte.py*.

A função que imprime a estrutura final do código Python gerado, pode ser vista na Figura 28.

Uma função fundamental do módulo *converte.py* é a função *converteTipos* que converte os tipos definidos no diagrama de classes do *Dia Modeler* para os tipos do Django, conforme a Tabela 2.

```
def imprimePSM(pim, nomeApp):
    """Ordenana a aparência do código fonte Python (PSM) e o imprime"""
    ordenado = []
    cont = 0
    psm = []
    pre_psm, imports = parser(pim)
    for indice,valor in pre_psm.iteritems():
        valor[2] = valor[2] + "\n    def __unicode__(self):\n        return u\"\"\n"
        for nome_classe in valor[0]:
            if nome_classe not in outras_classes:
                pre_psm[nome_classe][3] += 1
        ordenado.append([indice] + valor)
    while cont < len(ordenado):
        marca = cont
        cont2 = cont + 1
        while cont2 < len(ordenado):
            if ordenado[cont][0] in ordenado[cont2][1]:
                marca = cont2
            cont2 += 1
        if marca == cont:
            cont += 1
        else:
            a = ordenado[cont]
            ordenado[cont] = ordenado[marca]
            ordenado[marca] = a
        if cont == len(ordenado) - 1:
            break
    ordenado.reverse()
    if imports:
        psm.append(imports)
    for linha in ordenado:
        psm.append(linha[3])
        psm.append("\n")
    itens = [linha[3].split()[1] for linha in ordenado if "class" in linha[3]]
    classes = []
    for classe in itens:
        limite = classe.find('(')
        classes.append(classe[:limite])
    makeAdmin(nomeApp, classes)
    makeModels(nomeApp, psm)
    makeInit(nomeApp)
    makeViews(nomeApp)
```

Figura 28. Função que imprime o código Python final.
Fonte: Trecho do arquivo *converte.py* da ferramenta DjangoGen.

6.9 INSTALANDO A FERRAMENTA DJANGOGEN

A ferramenta DjangoGen possui alguns pré-requisitos de *software* para que funcione de forma correta. Os *softwares* recomendados e que foram utilizados no desenvolvimento e testes dessa ferramenta, são descritos abaixo.

Softwares obrigatórios:

a) python 2.5.2;

b) django 1.0;

c) *dia modeler* 0.96.1.

Softwares recomendados:

a) *browser* mozilla firefox 3.0.3.

A instalação dos itens descritos acima, não será coberta por esse trabalho, pois exigem diversos passos e variam para cada Sistema Operacional. Portanto o recomendando é que se sigam as instruções de instalação padrão, contidas em cada um dos pacotes de *software* mencionados.

A ferramenta DjangoGen é capaz de executar em qualquer Sistema Operacional onde seja possível instalar os *softwares* obrigatórios descritos anteriormente, seja Windows, Unix, Linux, ou FreeBSD. No desenvolvimento e testes desse trabalho, foi utilizado o sistema operacional MacOS X 10.5.5.

A instalação da ferramenta DjangoGen pode ser efetuada a partir dos seguintes passos:

a) descompactar o DjangoGen em uma pasta de fácil acesso, onde a ferramenta será instalada;

b) no diretório do DjangoGen descompactado, executar em um *prompt* de comando ou terminal o comando *python manage.py syncdb* dentro da pasta da ferramenta DjangoGen descompactada. Este comando irá criar de forma automática todas as tabelas e campos contidos no arquivo *models.py* da aplicação, diretamente no banco de dados configurado no arquivo *settings.py*. Também será possível configurar a senha de administrador por meio desse comando;

c) executar em um *prompt* de comando ou terminal o comando *python manage.py runserver* dentro da pasta da ferramenta DjangoGen descompactada. Esse comando irá iniciar o servidor *Web* do Django;

d) acessar via *browser* o endereço *http://127.0.0.1:8000/admin/* afim de visualizar a interface de administração do Django. Nesse endereço são listados todos os objetos disponíveis por meio da interface do Django (conforme definido em cada classe no arquivo *models.py*);

e) por fim, para que seja possível visualizar a interface da ferramenta DjangoGen, deve-se acessar o endereço *http://127.0.0.1:8000/upload/*.

Como já foi descrito no tópico 6.6 todo o código gerado pela ferramenta DjangoGen é armazenado no diretório *apps*, na pasta que leva o nome definido pelo usuário pela interface do DjangoGen.

No próximo tópico serão abordados detalhes sobre a geração de *apps* utilizando a ferramenta DjangoGen.

6.10 UTILIZAÇÃO DA FERRAMENTA DJANGOGEN

Foi utilizado o diagrama da Figura 21 na demonstração de conversão de diagramas UML do *Dia Modeler* em código Django.

O primeiro passo após se ter em mãos o arquivo UML gravado pelo *Dia Modeler* é executar o servidor do Django que contém a aplicação DjangoGen e acessar a *URL http://127.0.0.1:8000/upload/* onde é possível visualizar a interface da ferramenta DjangoGen.

Na Figura 29 pode-se verificar que a interface da ferramenta DjangoGen é bastante simples, contendo um campo para o *upload* do diagrama UML do *Dia Modeler*, um campo para informar o nome da *app* - e conseqüentemente o nome da pasta onde o código será gerado – e um botão para a geração de código.

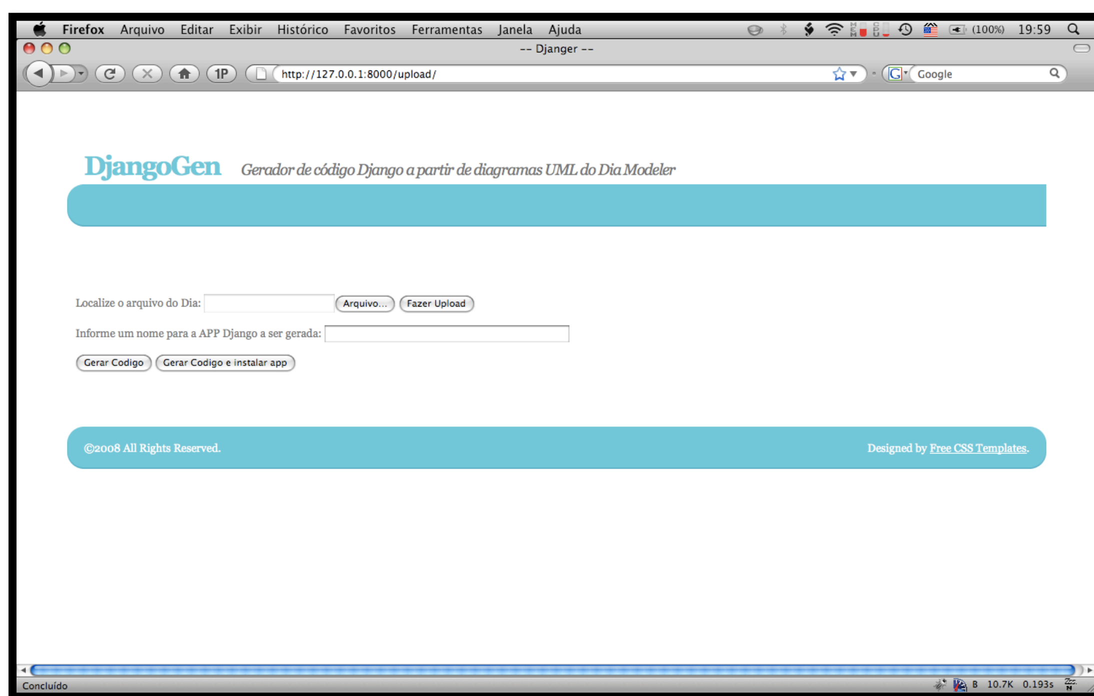


Figura 29. Interface da ferramenta DjangoGen.

Após clicar no botão *Gerar Código e instalar app* o código será gerado na pasta *apps* da aplicação e será instalado automaticamente, ficando disponível na interface de administração do Django imediatamente. De qualquer forma, também é possível copiar a *app* gerada, possibilitando a instalação em uma aplicação Django de outro servidor.

A ferramenta DjangoGen cria os seguintes arquivos no momento da geração de código, conforme a Figura 30:

a) *__init__.py:* conforme item *a* do tópico 6.5.1;
b) *admin.py*: arquivo responsável por registrar as classes que serão manipuláveis e customizar a forma de interagir e exibir as mesmas via interface de administração do Django;
c) *models.py*: conforme item *c* do tópico 6.5.1;
d) *views.py*: conforme item *f* do tópico 6.5.1.

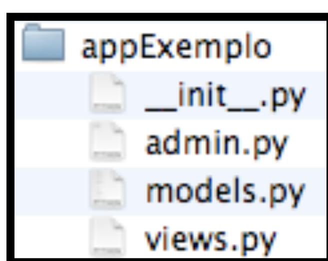


Figura 30: Estrutura de arquivos de uma *app* que acabou de ser gerada.
Fonte: Finder

Por padrão a ferramenta DjangoGen, define a função *__unicode__* conforme é exibido na Figura 31, onde pode-se visualizar a classe *Carro* e os atributos *nome*, *pagina* e *acessórios.* Porém para que os objetos sejam exibidos corretamente na interface de administração do Django, o arquivo *models.py* deve conter a definição de quais dados serão exibidos como padrão para cada classe, definido pela função *__unicode__*. Como essa é uma característica que muda de aplicação para aplicação,

não é possível prever na ferramenta o que deve ou não ser exibido para cada objeto, sendo necessária a intervenção do desenvolvedor na edição do código do arquivo *models.py,* conforme cada caso.

Como exemplo, pode-se supor que no arquivo *models.py* exista o código exibido na Figura 31. Nesse caso como a função __*unicode*__ não retorna nenhum dado (apenas *u""*), quando for criado um objeto *Carro* via interface de administração do Django, este objeto não terá um dado visível que represente-o, ficando uma linha em branco na listagem de objetos adicionados, o que não é desejável.

Para resolver o problema descrito acima, é preciso editar o arquivo *models.py* informando um dado de retorno válido, que represente o objeto.

```
class Carro(Automovel) :
    nome = models.CharField(max_length=100)
    pagina = models.CharField(max_length=300, null=True,blank=True )
    acessorios = models.ManyToManyField(Acessorios)

    def __unicode__(self):
        return u""
```

Figura 31: Exemplo de classe contendo a função __*unicode*__ configurada de forma errada.

Nesse caso, o ideal seria editar o código e deixá-lo igual ao código exibido na Figura 32, pois desta forma cada objeto criado via interface de administração do Django terá uma representação correta na exibição. Evidentemente, a forma que a função __*unicode*__ deverá ser configurada, irá depender de cada caso, pois em determinadas situações pode ser necessário retornar mais de um tipo de dado diferente, ou retornar os dados formatados.

```
class Carro(Automovel) :
    nome = models.CharField(max_length=100)
    pagina = models.CharField(max_length=300, null=True,blank=True )
    acessorios = models.ManyToManyField(Acessorios)

    def __unicode__(self):
        return self.nome
```

Figura 32: Exemplo de classe contendo a função __*unicode*__ configurada de forma correta.

6.11 RESULTADOS OBTIDOS

Tendo em vista os resultados apresentados nas seções anteriores, constatou-se que o uso da ferramenta DjangoGen é viável e eficiente no que diz respeito ao desenvolvimento de aplicações utilizando o *framework Web* Django, observando-se as pequenas peculiaridades de construção dos diagramas UML. Os fatores que contribuíram para esta conclusão foram respectivamente:

a) desenvolvimento da camada de modelo da aplicação, de modo simplificado, por meio de diagramas UML de classes;

b) auto-documentação do projeto, por meio dos diagramas UML de classes, conforme item *a*;

c) ganho no tempo de desenvolvimento das classes e atributos e os arquivos base de uma aplicação Django que são gerados automaticamente a partir do item *a*;

d) a geração automática de código, sempre mantém o padrão e sintaxe corretas, evitando retrabalho e eventuais *bugs* na aplicação;

e) geração automática das tabelas e campos na base de dados, conforme as classes e atributos definidos no diagrama UML de classes, exigindo

apenas a configuração prévia de uma base de dados.

A única barreira para a total automatização da geração do código fonte, citada no final do item 6.10 e demonstrada na Figura 31, é o fato do desenvolvedor ter que editar o arquivo *models.py* após a geração do código fonte, afim de ajustar quais e de que forma os atributos irão representar os objetos na interface de administração do Django.

CONCLUSÃO

Este trabalho apresentou um estudo sobre conceitos de engenharia de *software*, aplicado na construção de uma ferramenta de conversão de diagramas UML do modelador UML *Dia Modeler* em código Python da camada de modelo do *framework Web* Django. A base do trabalho, como pode-se verificar na fundamentação teórica, foi engenharia de *software*, que aborda dentre outras coisas, os conceitos de UML, XML, MDA e padrões *Web*.

A partir do estudo de arquiteturas orientadas a modelo, sobre como as classes Django são definidas e de que forma o diagrama UML é representado pelo XML gerado pela ferramenta *case* UML *Dia Modeler*, foi possível desenvolver uma ferramenta que gera código Python para o Django a partir de diagramas UML do *Dia Modeler*, em outras palavras, é possível gerar a estrutura base de uma *app* Django, incluindo a camada de modelo, apenas com o diagrama de classes UML.

A ferramenta DjangoGen foi testada utilizando um diagrama UML de testes contendo chaves estrangeiras e importação de bibliotecas externas. Os testes foram realizados em um computador utilizando o Sistema Operacional MacOS X 10.5.5 utilizando como servidor o próprio servidor *Web* embutido do Django 1.0 e banco de dados *sqlite3*, porém não há problemas em se utilizar outros servidores *Web*, ou outros servidores de bancos de dados.

Após os testes, verificou-se que os benefícios na utilização dessa ferramenta, contra um desenvolvimento sem a utilização da mesma, são:

a) agilidade na geração do código fonte;

b) auto instalação das *apps*;

c) auto geração da documentação por meio de diagramas UML de classes;

d) auto criação das tabelas e campos no banco de dados;

e) padronização e reuso dos códigos.

Como sugestões de trabalhos futuros que podem dar continuidade a este projeto, bem como melhorias na ferramenta desenvolvida, pode-se listar:

a) suporte a outras ferramentas Case (modeladores UML);

b) escrever o DjangoGen todo em inglês e criar um arquivo de localização (tradução) para pt-br posteriormente. Isso facilitaria contribuições futuras;

c) implementação da leitura das operações/funções nos diagramas de classes e posterior conversão em código Python;

d) permitir que sejam feitas customizações no arquivo *models.py*, de forma que quando ele seja re-gerado após uma mudança no diagrama UML de classes e o código customizado não seja sobrescrito;

e) permitir a importação/leitura de diagramas no formato XMI.

REFERÊNCIAS

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML:** Guia do usuário. Rio de Janeiro: Campus, 2000.

DALY, Liza. **Next-generation web frameworks in Python.** Sebastopol, CA: O'Reilly, 2007.

DJANGO PROJECT. **BUILT-IN FILTER REFERENCE.** Disponível em: <http://www.djangoproject.com/documentation/templates/#built-in-filter-reference>. Acesso em: 14 jun. 2008a.

DJANGO PROJECT. **The Web framework for perfectionists with deadlines.** Disponível em: <http://www.djangoproject.com>. Acesso em: 17 jun. 2008b.

DJANGO SITES. **Latest Additions.** Disponível em: <http://www.djangosites.org/>. Acesso em: 19 out. 2008.

FERRI, Jean R. **Ambiente para Contrução de Textos (ACT):** Um Ambiente para a Construção Colaborativa e Publicação de Textos na Web. 2002. 74 f. Monografia (Bacharelado) - Universidade Regional do Noroeste do Estado do Rio Grande do Sul, Ijuí, 2002.

FOWLER, Martin. **UML Distilled:** A Brief Guide to the Standard Object Modeling Language. 3. ed. Boston: Addison-wesley, 2000.

GNOME. **Dia a drawing program.** Disponível em: <http://www.gnome.org/projects/dia/home.html>. Acesso em: 23 out. 2008.

HOLOVATY, Adrian; MOSS, Jacob Kaplan. **The Django Book.** Disponível em: <http://www.djangobook.com/en/1.0/>. Acesso em: 06 out. 2007.

HOLZNER, Steven. **Desvendando XML.** Rio de Janeiro: New Riders, 2001.

LUTZ, Mark; ASCHER, David. **Learning Python.** 2. ed. Beijing: O'Reilly, 2004.

MELLO, Carlos Eduardo R.; SILVA, Geraldo Zimbrão; SOUZA, Jano M. **Desenvolvimento de SIG para Web utilizando MDA.** In: BRAZILIAN SYMPOSIUM ON GEOINFORMATICS, 9., 2007, Rio de Janeiro. Campos do Jordão: INPE, 2007. p. 1 - 6.

MELLOR, Stephen J. et al. **MDA Destilada:** Princípios de arquitetura Orientada por Modelos. Rio de Janeiro: Ciência Moderna, 2005.

MENDES, Antonio. **Programando com XML.** Rio de Janeiro: Elsevier, 2004.

MOORE, Dana; BUDD, Raymond; WRIGHT, William. **Professional Python frameworks:** Web 2.0 programming with Django and Turbogears. Indianapolis: Wiley, 2007.

MORGADO, Gisele Pereira. **RAPDIS:** Um Processo e um Ambiente MDA para o Desenvolvimento de Sistema de Informação. 2007. 159 f. Dissertação (Mestrado) - UFRJ, Rio de Janeiro, 2007.

NETCRAFT. **October 2008 Web Server Survey.** Disponível em: <http://news.netcraft.com/archives/2008/10/29/october_2008_web_server_survey.html>. Acesso em: 18 nov. 2008.

NIEDEST, Jennifer. **Web Design in a Nutshell.** Sebastopol, CA: O'Reilly, 1998.

PENDER, Tom. **UML Bible.** Indianapolis: Wiley, 2003.

PITTS-MOULTIS, Natanya. **XML:** black book. São Paulo: Makron Books, 2000.

SCHMITT, Christopher. **CSS Cookbook.** Sebastopol, CA: O'Reilly, 2006.

SILVA, A. R. **Abordagem XIS ao Desenvolvimento de Sistemas de Informação.** In: CONFERÊNCIA DA ASSOCIAÇÃO PORTUGUESA DE SISTEMAS DE INFORMAÇÃO, 4., 2003, Porto. Anais… Porto: Universidade Portucalense, 2003.

SMANIA, Rodrigo Spillere. **Criação de Componentes Dinâmicos para o Joomla por meio de UML.** 2008. 88 f. Monografia (Bacharelado) - Unesc, Criciúma, 2008.

SOUSA, M. C. F. **Um Processo de Desenvolvimento Baseado em Componentes Adaptado ao Model Driven Architecture.** 2004. 136 f. Dissertação (Mestrado) - UNICAMP, Campinas, 2004. Disponível em: <http://libdigi.unicamp.br/document/?down=vtls000334841>. Acesso em: 19 nov. 2008.

SUN MICROSYSTENS. **Java BluePrints:** Model-View-Controller. Disponível em: <http://java.sun.com/blueprints/patterns/MVC-detailed.html>. Acesso em: 21 abr. 2008.

THOMAZ, Fabio Eduardo. **Estudo do framework Django e da sua utilização no desenvolvimento de uma aplicação Web para o controle de alocação dos professores do Instituto Superior TUPY.** 2007. 95 f. Monografia (Bacharelado) - Curso de Sistemas de Informação, Sociesc, Joinville, 2007.

W3C. **Extensible Markup Language (XML).** Disponível em: <http:// http://www.w3.org/XML>. Acesso em: 14 jun. 2008.