Módulo III
Arquitetura em Camadas

Prof. Ismael H F Santos

Ementa

- Módulo III – Camadas de Software

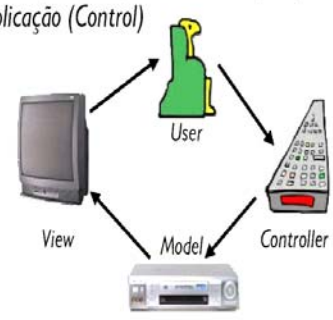
POO-Java

Arquitetura Em Camadas

Arquitetura MVC

- Surgiu nos anos 80 com a linguagem SmallTalk
- Divide a aplicação em tres partes fundamentais
  - Model – Representa os dados da aplicação e as regras de negócio (business logic)
  - View – Representa a informação recebida e enviada ao usuário
  - Controller – Recebe as informações da entrada e controla o fluxo da aplicação

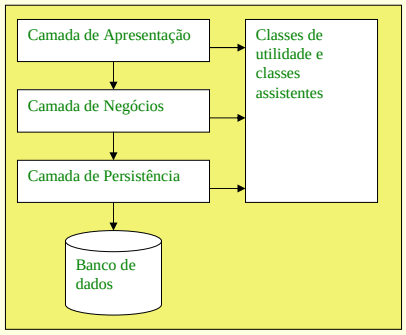
- *Técnica para separar dados ou lógica de negócios (Model) da interface do usuário (View) e do fluxo da aplicação (Control)*



Fonte: http://www.computer-programmer.org/articles/struts/

Arquitetura em camadas

- Arquitetura em camadas visa a criação de aplicativos modularisados, de forma que a camada mais alta se comunica com a camada mais baixa e assim por diante, fazendo com que uma camada seja dependente apenas da camada imediatamente abaixo.



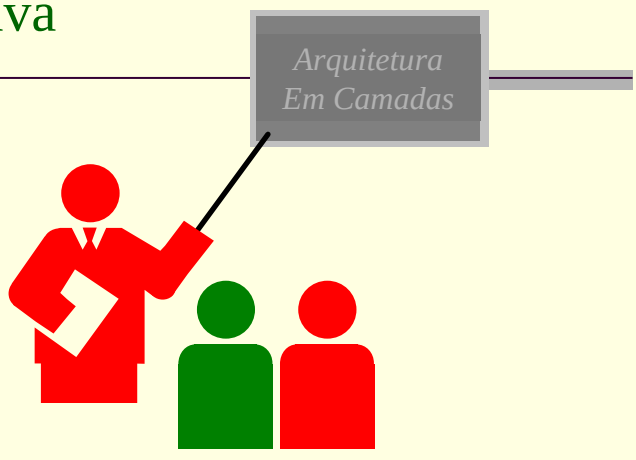
Arquitetura em camadas

- **Camada de Apresentação:** Lógica de interface do usuário (GUI). O código responsável pela apresentação e controle da página e tela de navegação forma a camada de apresentação;
- **Camada de Negócios:** Código referente a implementação de regras de negócio ou requisitos do sistema;
- **Camada de persistência:** Responsável por armazenamento e recuperação dos dados quando solicitado. Objetivo é o de garantir uma independencia da fonte de dados (arquivos, bancos de dados, etc)

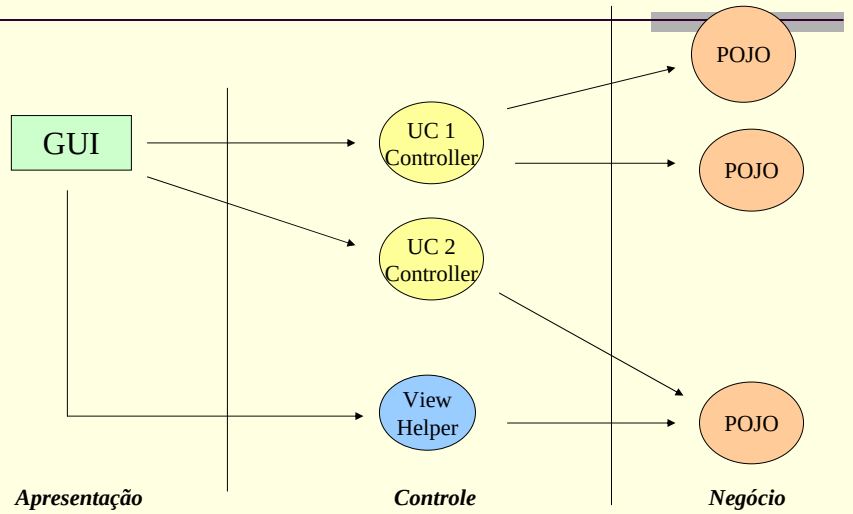
Arquitetura em camadas

- **Banco de dados**: O BD existe fora da aplicação Java, é a atual representação persistente do estado do sistema.
- **Assistentes e Classes de utilidade**: São classes necessária para o funcionamento ou mesmo o complemento de uma aplicação ou parte dela, como por exemplo o Exception para tratamento de erros.

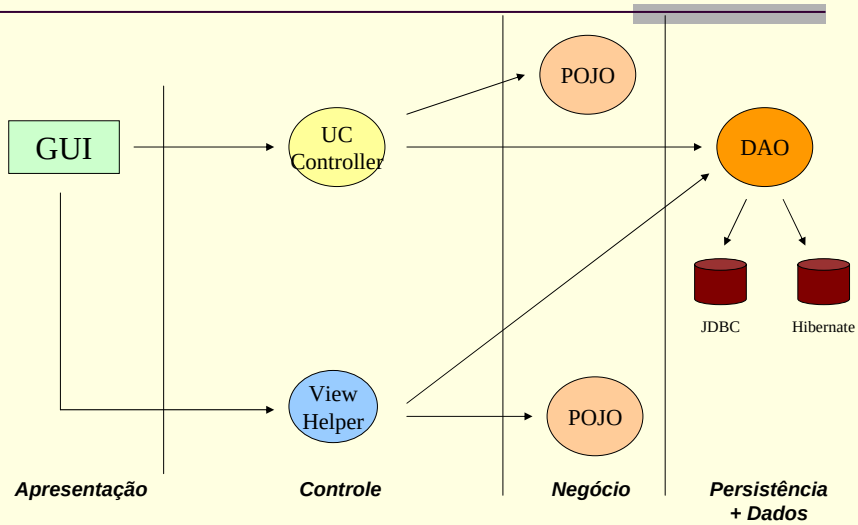
POO-Java



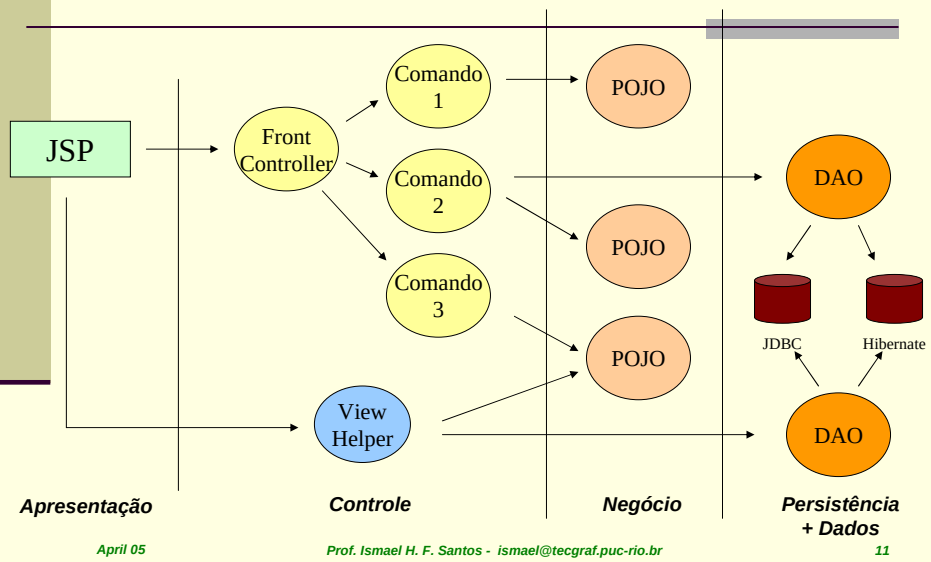
Modelo de Camadas – Apps Desktop



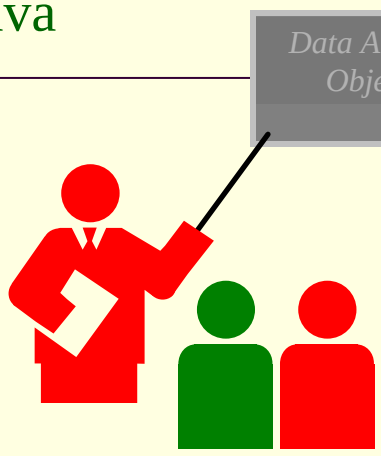
Modelo de Camadas – Apps Desktop



Modelo de Camadas – Apps Web



POO-Java



Patterns

- Não existe uma definição exata para o que são patterns. O autor Martin Fowler, define patterns como uma idéia que foi útil em um contexto prático e que provavelmente será útil em outros. Outros autores definem como uma regra que expressa uma relação entre um contexto, um problema e uma solução. Mas em geral, patterns tem sempre as seguintes características:
  - são notados através da experiência;
  - evitam que se reinvente a roda;
  - existem em diferentes níveis de abstração;
  - são artefatos reutilizáveis;
  - passam aos desenvolvedores designs corretos e
  - podem ser combinados para resolver um grande problema;
  - aceitam melhoramentos continuos.

Pattern Data Access Object (DAO)

- Objetivo
  - ***Abstrair e encapsular todo o acesso a uma fonte de dados. O DAO gerencia a conexão com a fonte de dados para obter e armazenar os dados.***

Pattern DAO

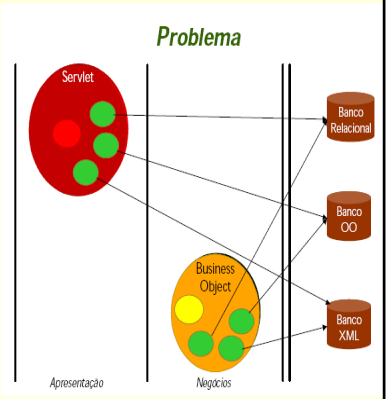
- O padrão Data Access Object , também conhecido como o padrão DAO, abstrai a recuperação dos dados tal com com uma base de dados. O conceito é "separar a relação do cliente de um recurso dos dados de seu mecanismo de acesso dos dados."
- O DAO é utilizado para encaplusar a lógica de acesso a dados. Assim, se for necessário a alteração de banco de dados, não é necessário alterar todo sistema, mas somente os DAOs.

Pattern DAO

- Dentro do DAO são realizadas as querys ou o acesso aos métodos do Hibernate. A intenção real de existência dos DAOs é que eles não possuam nenhuma lógica de negócio, apesar de algumas vezes ser necessário encaplusar algo dentro deles, especialmente quando outros patterns da camada de modelo não estão presentes.
- Quando utilizado junto com Hibernate, ambos realizam o trabalho de abstrair a base, pois o Hibernate já mascara o tipo do banco de dados, ficando para o DAO a parte de controlar as conexões, excessões, retornos para os níveis superiores, entre outros.

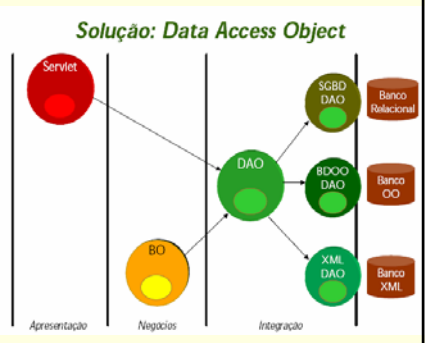
Problema

- ***Forma de acesso aos dados varia consideravelmente dependendo da fonte de dados usado***
  - ***Banco de dados relacional***
  - ***Arquivos (XML, CSV, texto, formatos proprietários)***
  - ***LDAP***
- ***Persistência de objetos depende de integração com fonte de dados (ex: business objects)***
  - ***Colocar código de persistência (ex: JDBC) diretamente no código do objeto que o utiliza ou do cliente amarra o código desnecessariamente à forma de implementação***
  - ***Ex: difícil passar a persistir objetos em XML, LDAP, etc.***



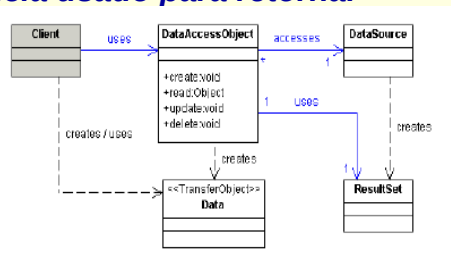
Solução

- *Data Access Object (DAO)* ***oferece uma interface comum de acesso a dados e esconde as características de uma implementação específica***
  - ***Uma API: métodos genéricos para ler e gravar informação***
  - ***Métodos genéricos para concentrar operações mais comuns (simplificar a interface de acesso)***
- *DAO* ***define uma interface que pode ser implementada para cada nova fonte de dados usada, viabilizando a substituição de uma implementação por outra***
- *DAOs* ***não mantêm estado nem cache de dados***

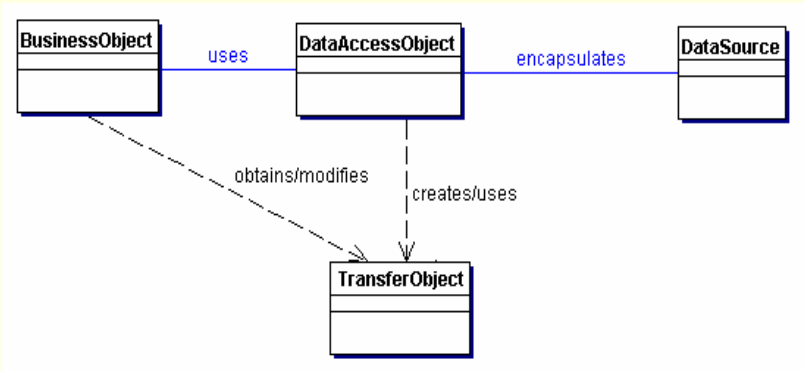


UML

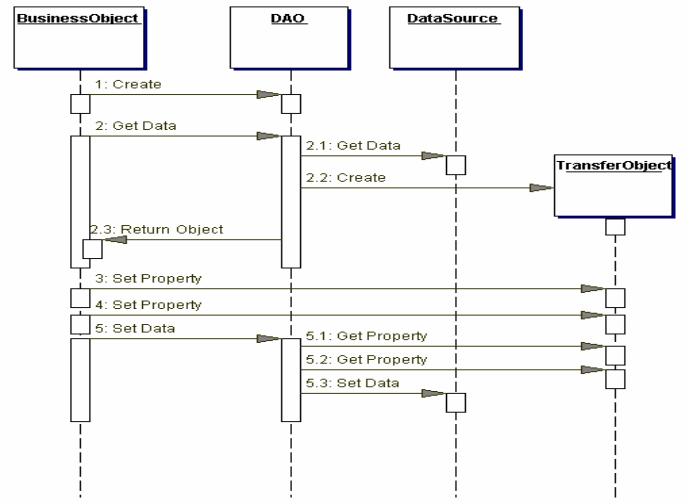
- *Client*: ***objeto que requer acesso a dados: Business Object, Session Façade, Application Service, Value List Handler, ...***
- *DAO*: ***esconde detalhes da fonte de dados***
- *DataSource*: ***implementação da fonte de dados***
- *Data*: ***objeto de transferência usado para retornar dados ao cliente. Poderia também ser usado para receber dados.***
- *ResultSet*: ***resultados de pesquisa no banco***



Exemplo



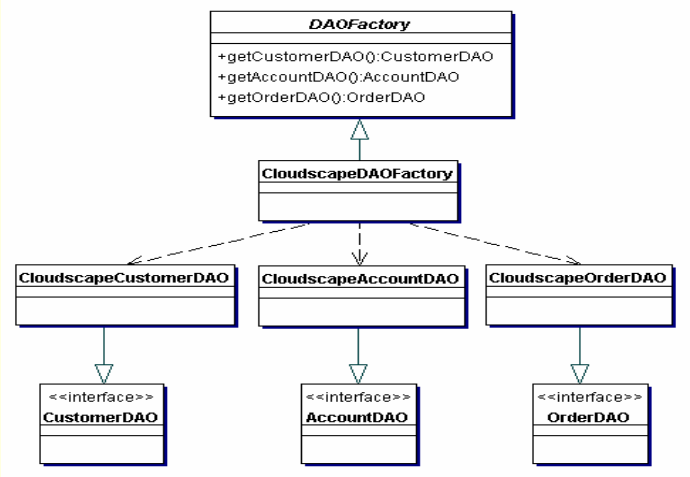
Exemplo - Diagrama de Sequência



Estratégias de Implementação

- ***Custom DAO Strategy***
  - ***Estratégia básica. Oferece métodos para criar, apagar, atualizar e pesquisar dados em um banco.***
  - ***Pode usar Transfer Object para trocar dados com clientes***
- ***DAO Factory Method Strategy***
  - ***Utiliza Factory Methods em uma classe para recuperar todos os DAOs da aplicação***
- ***DAO Abstract Factory Strategy***
  - ***Permite criar diversas implementações de fábricas diferentes que criam DAOs para diferentes fontes de dados***

Implementing Factory for DAOStrategy Using Factory Method Pattern



Abstract class DAO Factory

```
public abstract class DAOFactory {
  // List of DAO types supported by the factory
    public static final int CLOUDSCAPE = 1;
    public static final int ORACLE = 2;
    public static final int SYBASE = 3; ...
  /* There will be a method for each DAO that can be created. The concrete
      factories will have to implement these methods. */
    public abstract CustomerDAO getCustomerDAO();
    public abstract AccountDAO getAccountDAO();
    public abstract OrderDAO getOrderDAO(); ...
    public static DAOFactory getDAOFactory( int whichFactory) {
      switch (whichFactory) {
        case CLOUDSCAPE: return new CloudscapeDAOFactory();
        case ORACLE : return new OracleDAOFactory();
        case SYBASE : return new SybaseDAOFactory(); ...
        default : return null;
      }
    }
}
```

Cloudscape concrete DAO Factory implementation

```
public class CloudscapeDAOFactory extends DAOFactory {
  public static final String DRIVER= "COM.cloudscape.core.RmiJdbcDriver";
  public static final String DBURL="jdbc:cloudscape:rmi://localhost:1099/CoreJ2EEDB";
  // method to create Cloudscape connections
  public static Connection createConnection() { ..... // create a connection    }
  public CustomerDAO getCustomerDAO() {
    // CloudscapeCustomerDAO implements CustomerDAO
    return new CloudscapeCustomerDAO();
  }
  public AccountDAO getAccountDAO() {
    // CloudscapeAccountDAO implements AccountDAO
    return new CloudscapeAccountDAO();
  }
  public OrderDAO getOrderDAO() {
    // CloudscapeOrderDAO implements OrderDAO
    return new CloudscapeOrderDAO();
  }
  ...
}
```


Interface CustomerDAO

```
public interface CustomerDAO {
  public int insertCustomer(...);
  public boolean deleteCustomer(...);
  public Customer findCustomer(...);
  public boolean updateCustomer(...);
  public RowSet selectCustomersRS(...);
  public Collection selectCustomersTO(...);
  ...
}
```

CloudscapeCustomerDAO impl

```
/* This class can contain all Cloudscape specific code and SQL
    statements. The client is thus shielded from knowing these
    implementation details */
import java.sql.*;
public class CloudscapeCustomerDAO implements CustomerDAO {
   public CloudscapeCustomerDAO() { // initialization }
   /* The following methods can use CloudscapeDAOFactory.
      createConnection() to get a connection as required */
   public int insertCustomer(...) {
     /* Implement insert customer here. Return newly created customer
        number or a -1 on error */
   }
  public boolean deleteCustomer(...) {
    /* Implement delete customer here // Return true on success,
       false on failure */
   }
```


CloudscapeCustomerDAO impl

```
 public Customer findCustomer(...) {
  /* Implement find a customer here using supplied argument values as search
     criteria. Return a Transfer Object if found, return null on error or if not found */
 }
 public boolean updateCustomer(...) {
  /* implement update record here using data from the customerData
     Transfer Object Return true on success, false on failure or error */
 }
 public RowSet selectCustomersRS(...) {
    /* implement search customers here using the supplied criteria.
       Return a RowSet. */
 }
 public Collection selectCustomersTO(...) {
    /* implement search customers here using the supplied criteria.  Alternatively,
       implement to return a Collection of Transfer Objects. */
 }
 ...
}
```

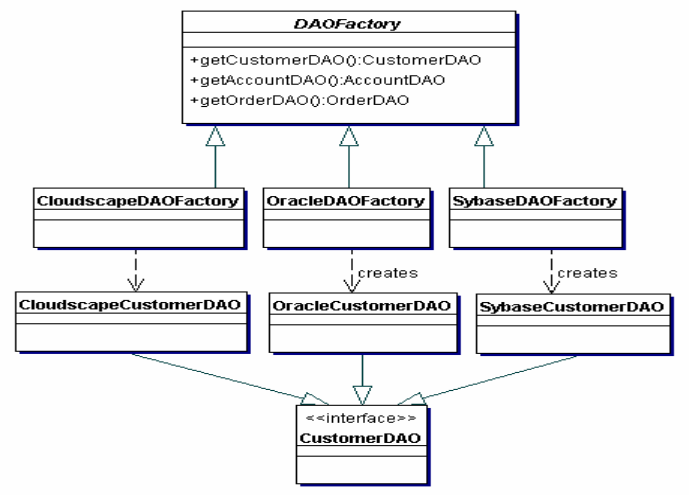
Client Code

```
// Create the required DAO Factory and the DAO Object
DAOFactory cloudscapeFactory =
      DAOFactory.getDAOFactory(DAOFactory.DAOCLOUDSCAPE);
CustomerDAO custDAO = cloudscapeFactory.getCustomerDAO();
 // Create a new customer. Find a customer object. Get Transfer Object.
 int newCustNo = custDAO.insertCustomer(...);
 Customer cust = custDAO.findCustomer(...);
 // modify the values in the Transfer Object. Update the customer object
 cust.setAddress(...); cust.setEmail(...); custDAO.updateCustomer(cust);
// delete a customer object
 custDAO.deleteCustomer(...);
// select all customers in the same city
Customer criteria=new Customer(); criteria.setCity("New York");
Collection customersList = custDAO.selectCustomersTO(criteria);
// returns collection of CustomerTOs. iterate through this collection to get
   values.
...
```


CustomerTransferObject

```
import java.io.*;
public class Customer implements Serializable {
   // member variables
   int CustomerNumber;
   String name;
   String streetAddress;
   String city;
    ...
    // getter and setter methods...
    ...
}
```

Using Abstract Factory Pattern



Conseqüências DAO

- *Transparência quanto à fonte de dados*
- *Facilita migração para outras implementações*
  - ***Basta implementar um DAO com mesma interface***
- *Reduz complexidade do código nos objetos de negócio (ex: Entity Beans BMP)*
- *Centraliza todo acesso aos dados em camada separada*
  - ***Qualquer componente pode usar os dados (servlets, EJBs)***
- *Camada adicional*
  - ***Pode ter pequeno impacto na performance***
- *Requer design de hierarquia de classes (Factory)*
- *Exemplos de DAO*
  - ***DAO para cada Business Object***
  - ***DAO para serviços arbitrários***