UNIVERSIDADE DO ESTADO DE SANTA CATARINA
DCC – Departamento de Ciência da Computação
Joinville – SC

CURSO: Bacharelado em Ciência da Computação
DISCIPLINA: POO0001 – Programação Orientada a Objetos
PROFESSOR: Diego Buchinger
AULA 10 – Padrões de Projeto

1. PADRÕES DE PROJETO

Desenvolvedores de software experientes costumam almejar programas ou subprogramas que sejam específicos o suficiente para o problema a ser resolvido, mas também genérico o suficiente para atender futuros problemas e requisitos similares. Projetistas experientes de software orientados a objetos dizem que um projeto reutilizável e flexível é difícil de obter logo nas primeiras tentativas e que é comum os "novatos" acabarem usando técnicas não orientadas a objetos. A dica destes desenvolvedores experientes é reutilização de soluções que funcionaram no passado – quando encontram uma solução eficiente e bem organizada, utilizam-na repetidamente, criando-se padrões.

Fazendo uma analogia, roteiristas raramente projetam suas tramas do zero. É comum que eles sigam padrões característicos como: "o herói problemático" ou "a heroína justa e de bom coração que é enganada". De forma similar, projetistas de software também utilizam padrões característicos para implementar software quando percebem que já viram ou resolveram um problema similar anteriormente, como: "represente estados como objetos" ou "decore objetos de maneira que seja possível acrescentar ou remover características facilmente".

Pode-se dizer que os padrões de projeto (*design patterns*) são soluções reutilizáveis de software orientado a objetos que foram registradas, documentadas e extensivamente testadas. Estes padrões são independentes de linguagem e possuem quatro elementos essenciais: (1) um nome que identifica o padrão de maneira fácil e singular, (2) qual o(s) problema(s) resolvido(s) com a utilização deste padrão, (3) qual a solução e (4) quais são as consequências do uso do padrão.

O uso de padrões de projeto em desenvolvimento de software teve forte influência da chamada "Gangue dos Quatro" (GoF) – Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides – que escreveram o livro "*Design Patterns: Elements of Reusable Object-Oriented Software*" (1995) de grande notoriedade nesta área. Foram definidos 23 padrões de projeto, divididos em três famílias: padrões criacionais (relacionados à criação de objetos), padrões estruturais (tratam das associações entre classes e objetos) e padrões comportamentais (tratam das interações e divisões de responsabilidade entre classes e objetos).

1.1. CATÁLOGO DE PADRÕES DE PROJETO

O catálogo de padrões de projeto apresentado a seguir, com nome e definição breve de cada padrão, foi retirado do livro "*Design Patterns*" mencionado anteriormente. A Tabela 1 mostra a classificação destes padrões conforme seu propósito e escopo.

- ***Abstract Factory***: fornece uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas;

- ***Adapter***: converte a interface de uma classe em outra interface esperada pelos clientes. O Adapter permite que certas classes trabalhem em conjunto, pois de outra forma seria impossível por causa de suas interfaces incompatíveis;
- ***Bridge***: separa uma abstração da sua implementação, de modo que as duas possam variar independentemente;
- ***Builder***: separa a construção de um objeto complexo da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações;
- ***Chain of Responsibility***: Evita o acoplamento do remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar a solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate.;
- ***Command***: encapsula uma solicitação como um objeto, desta forma permitindo a parametrização de clientes com diferentes solicitações, enfileiramento ou registro (log) de solicitações e suporte a operações que podem ser desfeitas;
- ***Composite***: compõe objetos em estrutura de árvore para representar hierarquias do tipo partes-todo. O Composite permite que os clientes tratem objetos individuais e composições de objetos de maneira uniforme;
- ***Decorator***: atribui responsabilidades adicionais a um objeto dinamicamente. Os *decorators* fornecem uma alternativa flexível a subclasses para extensão da funcionalidade;
- ***Façade***: fornece uma interface unificada para um conjunto de interfaces em um subsistema. O Façade define uma interface de nível mais alto que torna o subsistema mais fácil de usar;
- ***Factory Method***: define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe a ser instanciada, postergando a instanciação.
- ***Flyweight***: usa compartilhamento para suportar grandes quantidades de objetos de granularidade fina, de maneira eficiente;
- ***Interpreter***: dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nessa linguagem;
- ***Iterator***: fornece uma maneira de acessar sequencialmente os elementos de uma agregação de objetos sem expor sua representação subjacente;
- ***Mediator***: define um objeto que encapsula a forma como um conjunto de objetos interage. Promove acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros, permitindo que suas interações sejam variadas independentemente;
- ***Memento***: sem violar o encapsulamento, captura e externaliza um estado interno de um objeto, de modo que o mesmo possa posteriormente ser restaurado para este estado;
- ***Observer***: define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados;
- ***Prototype***: especifica os tipos de objetos a serem criados usando uma instância prototípica e criar novos objetos copiando esse protótipo;
- ***Proxy***: fornece um objeto representante (*surrogate*), ou um marcador de outro objeto, para controlar o acesso ao mesmo;
- ***Singleton***: garante que uma classe tenha somente uma instância e fornece um ponto global de acesso para ela;
- ***State***: permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe;

- ***Strategy***: define uma família de algoritmos, os encapsula e os torna intercambiáveis. Permite que o algoritmo varie independentemente dos clientes que o utilizam;
- ***Template Method***: define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses. O *Template Method* permite que as subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura;
- ***Visitor***: representa uma operação a ser executada sobre os elementos da estrutura de um objeto. O Visitor permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera.

TABELA 1 – Classificação dos padrões de projeto ("Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos", 1995, GAMMA, HELM, JOHNSON e VLISSIDES).

		PROPÓSITO		
		De criação	**Estrutural**	**Comportamental**
ESCOPO	**Classe**	*Factory Method*	*Adapter (class)*	*Interpreter* *Template Method*
	Objeto	*Abstract Factory* *Builder* *Prototype* *Singleton*	*Adapter (object)* *Bridge* *Composite* *Decorator* *Façade* *Flyweight* *Proxy*	*Chain of Responsibility* *Command* *Iterator* *Mediator* *Memento* *Observer* *State* *Strategy* *Visitor*

1.2. PADRÕES J2EE

Falando em padrões, existem diretrizes específicas para a tecnologia Java Enterprise Edition (JEE) que ficaram conhecidos como Padrões J2EE e que foram introduzidas no livro "*Core J2EE Patterns: Best Practices and Design Strategies*" (Deepak Alur, John Crupi e Dan Malks). Apesar de ser voltadas para aplicações robustas na web, alguns destes padrões são universalmente aplicáveis. A seguir é apresentada uma breve descrição desses padrões:

- ***Composite View***: forma uma camada de visão composta de múltiplas subcamadas de visão atômicas. Cada componente pode ser incluído dinamicamente e o layout geral pode ser gerenciado independentemente do conteúdo;
- ***Business Delegate***: esconde os detalhes da implementação, como detalhes de consulta e de acesso da arquitetura, dos serviços de negócio. Reduz o acoplamento entre clientes de apresentação e serviços de negócio;
- ***Transfer Object (TO)***: serve para encapsular dados de negócio, contidos em um *enterprise bean*, para que uma única chamada de método possa enviar ou buscar esses dados;
- ***Transfer Object Assembler***: usado para construir o modelo desejado, após ter usado *Transfer Objects* para recuperar dados das várias partes do sistema;
- ***Composite Entity***: representa um grafo de objetos, permitindo que se modele, represente e gerencie um conjunto de objetos persistentes não relacionados;

- ***Data Access Object (DAO)***: abstrai e encapsula todo o acesso à fonte de dados. Gerencia a conexão com a fonte de dados para obter e armazenar dados;
- ***Intercepting Filter***: cria filtros conectáveis para se processar serviços comuns de forma padrão, sem precisar mudar o código de processamento. Os filtros interceptam requisições e respostas, permitindo pré e pós-processamento;
- ***Front Controller***: usa um controlador como ponto inicial para tratamento de requisições, revogando serviços de segurança como autenticação, delegando processamento de negócio, gerenciando a escolha de uma *view* apropriada, tratando erros, e mais;
- ***View Helper***: classes auxiliares (normalmente JavaBeans), para as quais uma *view* delega suas responsabilidades de processamento. Também guardam o modelo de dados da *view*;
- ***Session Facade***: Usa um *bean* de sessão como *Facade* para encapsular a complexidade das interações entre os objetos de negócio participantes, providenciando uma interface uniforme de serviços aos clientes;
- ***Value List Handler***: controla a busca, gera cache de resultados e provê os resultados ao cliente em um "result set" cujo tamanho e método de iteração;
- ***Service Activator***: recebe mensagens assíncronas de clientes, localizando e invocando os métodos de negócio necessários para preencher a requisição assincronamente.

2. DETALHES DE IMPLEMENTAÇÃO DE PADRÕES

Entre os padrões listados na seção 1 existem alguns que são, ou podem ser, utilizados amplamente para uma vasta gama de aplicações. Para estes, será elaborado a seguir uma explicação mais detalhada de suas características e também de sua implementação.

2.1. PADRÃO SINGLETON

O objetivo do padrão singleton é garantir que uma classe seja instanciada apenas uma única vez. Isto é, garantir que exista apenas um objeto de uma determinada classe. Esse objetivo pode ser interessante para diversas aplicações, como por exemplo: garantir que determinadas configurações/opções sejam definidas apenas em um objeto (e.g. o volume, a dificuldade em um jogo, opção de legenda etc.), ou então garantir que uma classe que controla o acesso ao banco de dados tenha apenas uma instância (evitando múltiplas conexões). Mas como podemos fornecer tal garantia de existência de apenas um único objeto?

A ideia deste padrão é fazer com que a classe em si se torne responsável pela definição, criação e preservação de uma única instância. Para isto, privatizam-se os métodos construtores, impedindo a instanciação de novos objetos da classe através do operador *new* nas classes externas; cria-se um atributo privado e estático da própria classe, e um método público, estático e sincronizado (evitando problemas com acesso concorrente entre threads) responsável por retornar a única instância da classe. Um modelo de implementação é apresentado no quadro a seguir. Note que o método estático *getInstance()* provê o único acesso a um objeto do tipo Opcoes e que faz a instanciação de tal objeto somente quando este for requisitado pela primeira vez.

Arquivo: Opcoes.java

```
 1.  package padroes;
 2.
 3.  public class Opcoes {
 4.      private static Opcoes instance = null;
 5.
 6.      private int musicVolume;
 7.      private int effectVolume;
 8.
 9.      private Opcoes(){
10.          musicVolume = 70;
11.          effectVolume = 80;
12.      }
13.
14.      public static synchronized Opcoes getInstance(){
15.          if( instance == null )
16.              instance = new Opcoes();
17.          return instance;
18.      }
19.
20.      public void setMusicVolume(int volume){
21.          if( volume<0 ) volume = 0;
22.          else if( volume>100 ) volume = 100;
23.          musicVolume = volume;
24.      }
25.
26.      public void setEffectVolume(int volume){
27.          if( volume<0 ) volume = 0;
28.          else if( volume>125 ) volume = 125;
29.          effectVolume = volume;
30.      }
31.
32.      public int getMusicVolume(){ return musicVolume; }
33.      public int getEffectVolume(){ return effectVolume; }
34.  }
```

Arquivo: Main.java

```
 1.  package padroes;
 2.
 3.  public class Main {
 4.      public static void main( String args[] ){
 5.          Opcoes op = Opcoes.getInstance();
 6.          op.setMusicVolume( 150 );
 7.          int mv = Opcoes.getInstance().getMusicVolume();
 8.      }
 9.  }
```

2.2. PADRÃO FAÇADE

Muitas aplicações ou bibliotecas são constituídas de subsistemas complexos, que são compostos por diversas classes que modelam ou implementam um comportamento desejado. Esses subsistemas, no entanto, podem possuir uma estrutura interna difícil de compreender e utilizar, principalmente para programadores que não conhecem a estrutura ou detalhes deste subsistema. Nestes casos é interessante construir uma “fachada” de acesso simples, organizada e flexível para este subsistema complexo, o que é exatamente o objetivo do padrão Façade. A Figura 1 ilustra uma

comparação entre o acesso/uso de um subsistema sem fachada e outra versão usando este padrão. Note que a versão que utiliza o padrão de Fachada centraliza o uso do subsistema encapsulando o comportamento esperando e não exige o conhecimento das demais classes que estão dentro deste pacote do subsistema.

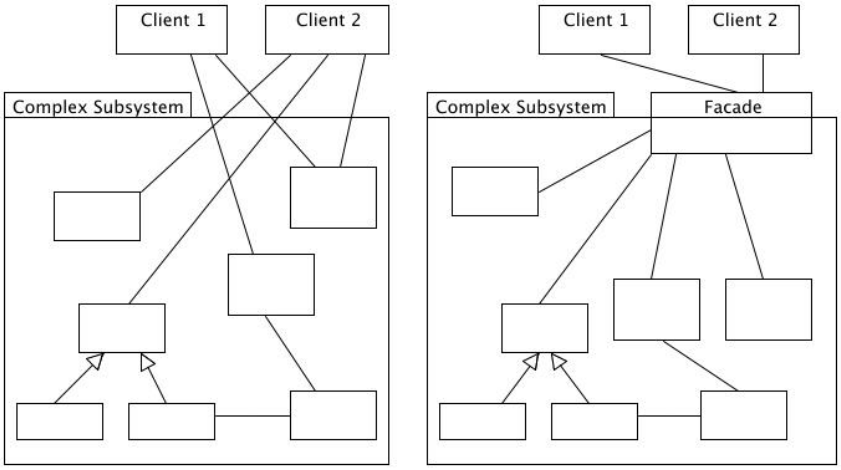


FIGURA 1 – Comparação entre um modelo de acesso e uso de um subsistema sem utilização do padrão Façade (esquerda) e outro modelo com uso do padrão Façade (direita).

Para tentar exemplificar este padrão, considere a construção de um software para uso de um *Home Theater*. Note que este equipamento é constituído por diversos componentes: a tela, o projetor, o amplificador, a unidade de leitura de DVD (*dvd player*) e o controlador da luz ambiente. Todos esses componentes são então modelados e codificados isoladamente em classes conforme apresentado na Figura 2, cada um com as definições de suas operações e funcionalidades típicas. A definição e implementação destas partes isoladas, no entanto, deixa este subsistema difícil de ser utilizado externamente, pois uma classe que deseja fazer uso deste *Home Theater* terá que coordenar os objetos em conjunto, e isso será uma tarefa muito difícil para um programador que não conhece bem a estrutura interna de cada classe. Por exemplo, para providenciar uma operação de assistir um filme, seria necessário: ligar o projetor, ajustar a tela, desligar a luz ambiente, definir o modo *wide screen* para o projetor, ligar o amplificador, ajustar seu volume inicial, ligar o dvd player, colocar o filme e iniciá-lo. De fato, cada classe externa que deseja utilizar o *Home Threater* teria que implementar essa sequencia de passos, o que pode não ser muito lógico ou claro para a maioria dos programadores.

Para este tipo de sistema apresentado, o padrão Façade se aplica muito bem: constrói-se uma classe de fachada que centraliza e modela o comportamento do subsistema, abstraindo o uso das partes do subsistema para classes e programadores externos. Logo, as operações relacionadas ao *Home Theater*, como assistir filme, por exemplo, seriam implementada na classe de fachada, utilizando as classes internas e provendo acesso simplificado para as classes externas que desejam utilizar este subsistema. Isto é, encapsulamos a lógica de negócio fazendo com que outras camadas não precisem se preocupar com isto. O quadro a seguir ilustra como seria a implementação da classe `HomeTheaterFacade`.

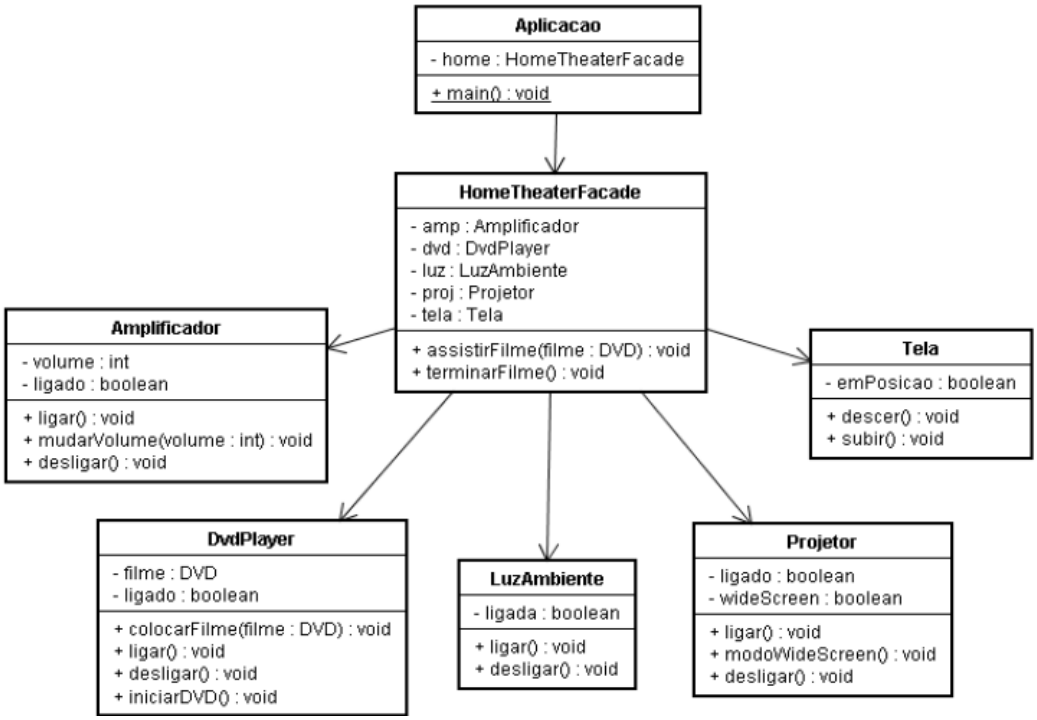


FIGURA 2 – Diagrama de classes que ilustra o uso do padrão Façade para o subsistema Home Theater

Arquivo: HomeTheaterFacade.java

```
 1.  package padroes.hometheater;
 2.
 3.  public class HomeTheaterFacade {
 4.      Amplificador amplificador;
 5.      DvdPlayer dvdPlayer;
 6.      LuzAmbiente luz;
 7.      Projetor projetor;
 8.      Tela tela;
 9.
10.      public HomeTheaterFacade(){
11.          //...
12.      }
13.
14.      public void assistirFilme(DVD filme){
15.          projetor.ligar();
16.          tela.descer();
17.          luz.desligar();
18.          projetor.modoWideScreen();
19.          amplificador.ligar();
20.          amplificador.mudarVolume( 5 );
21.          dvdPlayer.ligar();
22.          dvdPlayer.colocarFilme( filme );
23.          dvdPlayer.iniciarDVD();
24.      }
25.      //...
26.  }
```

2.3. PADRÃO DAO

As aplicações podem receber ou armazenar informações de diversas fontes de dados. Elas podem estar ou vir de um arquivo salvo em disco (XML, CSV ou outra extensão, por exemplo), de um banco de dados (relacional ou orientado a objetos, por exemplo), ou de um *web service*, para citar alguns. Os dados podem vir de uma fonte hoje, mas de outra amanhã. O padrão *Data Access Object* é uma solução simples que atende a essas questões. A ideia deste padrão é isolar a responsabilidade de receber ou resgatar informação em uma camada específica para este propósito, sendo que um objeto se torna unicamente responsável por estas atividades, abstraindo a visão dos dados para o resto da aplicação.

Uma aplicação que utiliza o padrão DAO pode produzir, por exemplo, três *Data Access Objects*, um para cada tipo de armazenamento: um para trabalhar com o banco de dados, outro para trabalhar com arquivos XML e outro para trabalhar apenas na memória RAM, mas todos eles possuem uma base de operações em comum. Isto é, todos inserem, alteram e removem registros.

Justamente por ter essa necessidade de possuir uma base de operações em comum, este modelo costuma estar atrelado ao uso de uma classe abstrata ou de uma interface que modela o comportamento a ser implementado destes objetos de acesso aos dados. O padrão DAO é, portanto, utilizado como uma maneira centralizada para se acessar e modificar dados, fornecendo maior praticidade e flexibilidade, além de encapsular os detalhes de acesso referente a cada tecnologia de armazenamento de dados utilizada. O quadro de código a seguir ilustra um exemplo de uso do padrão DAO. Note através deste exemplo que o padrão DAO geralmente utiliza o padrão Singleton, já mencionado, para garantir a existência de uma única instância dos *Data Access Objects*.

Arquivo: DAOInterface.java

```
 1.  package padroes.dao;
 2.  import interfaceGrafica.Person;
 3.
 4.  public interface DAOInterface {
 5.      boolean addContact( Person c );
 6.      boolean alterContact( Person c );
 7.      boolean removeContact( String cpf );
 8.      Person getContact( String cpf );
 9.      Person[] getContactList( );
10.  }
```

Arquivo: DAOMemoria.java

```
 1.  package padroes.dao;
 2.  import interfaceGrafica.Person;
 3.  import java.util.ArrayList;
 4.
 5.  public class DAOMemoria implements DAOInterface {
 6.      private DAOMemoria dao;
 7.      ArrayList<Person> contactList;
 8.
 9.      private DAOMemoria(){ }
10.      public DAOMemoria getInstance(){
11.          if( dao == null )
12.              dao = new DAOMemoria();
13.          return dao;
14.      }
15.
```

```
16.       @Override
17.       public boolean addContact(Person c) {
18.           // adiciona 'c' à 'contactList'
19.       }
20.
21.       @Override
22.       public boolean alterContact(Person c) {
23.           // procura a pessoa de mesmo cpf que 'c' em
24.           // 'contactList' e atualiza seus dados
25.       }
26.
27.       //...
28.
29.   }
```

Arquivo: DAOArquivo.java

```
 1.  package padroes.dao;
 2.  import interfaceGrafica.Person;
 3.
 4.  public class DAOArquivo implements DAOInterface {
 5.      private DAOArquivo dao;
 6.      private String contactFile = "meus-contatos.txt";
 7.
 8.      private DAOArquivo(){ }
 9.      public DAOArquivo getInstance(){
10.          if( dao == null )
11.              dao = new DAOArquivo();
12.          return dao;
13.      }
14.
15.      @Override
16.      public boolean removeContact(String cpf) {
17.          // procura um registro de pessoa com cpf = cpf
18.          // e remove tal registro do arquivo
19.      }
20.
21.      @Override
22.      public Person getContact(String cpf) {
23.          // procura por um registro de pessoa com cpf = cpf
24.          // e retorna seus dados
25.      }
26.
27.      @Override
28.      public Person[] getContactList() {
29.          // resgata os dados de todas as pessoas cadastradas
30.          // no arquivo e retorna como um array
31.      }
32.
33.      //...
34.  }
```

2.4. PADRÃO DTO

Chamadas de métodos remotos, que acessam dispositivos ou aplicações externas, podem degradar consideravelmente a velocidade de resposta de um sistema. Abordagens que procuram

reduzir este *overhead* investem na minimização do número destes tipos de acessos. O padrão *Data Transfer Object* encapsula uma entidade da aplicação em um determinado instante – uma cópia direta dos dados que é construída e enviada no lugar dos originais – de modo que os dados referentes a um DTO possam ser buscados externamente de uma única vez, ou com o menor número possível de chamadas externas. Ao invés de fazer chamadas separadas para receber nome, endereço, CPF e outros dados, o sistema requisita todas as informações de uma única vez e compõem um objeto de transferência de dados que será encaminhado para a classe que o requisitou.

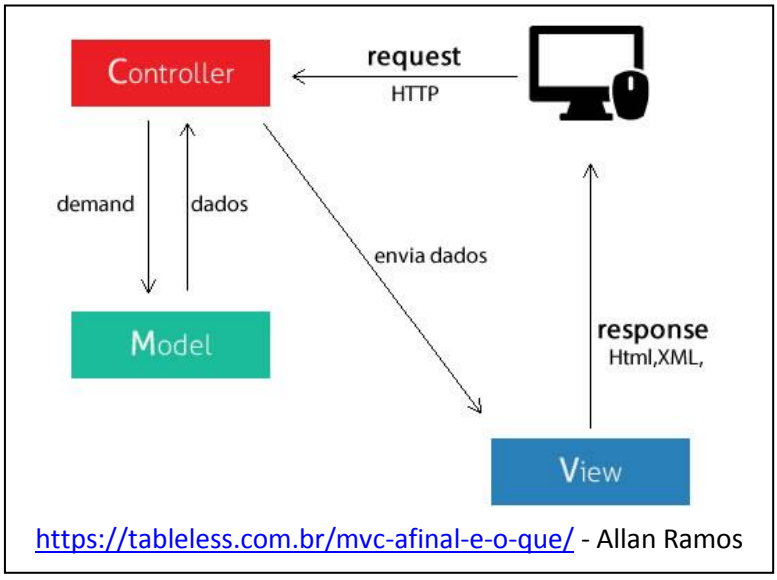
DTOs são muito semelhantes às classes que modelam as entidades de um sistema orientado a objetos. Talvez a diferença mais significativa seja que estes objetos não possuem qualquer lógica de negócio atrelada a eles. São basicamente objetos que são montados / construídos e utilizados. Em determinados situações, classes que representam entidades no sistema podem e são usadas como DTOs também, mas deve-se ter cuidado com possíveis verificações implementadas na classe que podem gerar *overhead* excessivo.

3. OUTROS PADRÕES DE PROJETO

Além dos padrões citados e apresentados, existem outros padrões que são apresentados na literatura e merecem atenção especial dos desenvolvedores. Dois padrões arquiteturais – ou seja, que trabalham sobre a forma / esboço de como um sistema é elaborado – merecem destaque especial por serem amplamente utilizados em aplicação de médio a grande porte.

3.1. PADRÃO MODEL VIEW CONTROLLER (MVC)

O padrão *Model View Controller* é um padrão arquitetural de alto nível, que visa à reusabilidade e a separação do código de um software em três partes interconectadas: modelo, visão e controle. Cada uma dessas partes possui suas responsabilidades, permitindo o desenvolvimento, teste e manutenção isolados das partes do software. As responsabilidades de cada camada são as seguintes:



https://tableless.com.br/mvc-afinal-e-o-que/ - Allan Ramos

- ***Model***: camada de representação e manipulação dos dados, isto é, uma camada que se responsabiliza pela leitura e escrita de dados em um repositório (memória, arquivo etc.), além das suas devidas validações. Também tem a responsabilidade de notificar suas visões e c controladores associados quando ocorre uma mudança em seu estado;

- ***View***: camada de apresentação e interação com o usuário. É responsável pela exibição dos dados e por receber as interações ou respostas dos usuários;

- ***Controller***: camada de controle da aplicação, responsável por processar as requisições do usuário, buscando e enviando dados para a camada de modelo e comandos para uma visão a fim de alterar a sua apresentação.

Para exemplificar a dinâmica que ocorre neste padrão arquitetural, considere um caso no qual um usuário inseriu seus dados de *login* e senha e está pedindo para se autenticar no aplicativo do sistema acadêmico da universidade. A classe da camada de visão associada a este formulário irá receber esta requisição e irá comunicar o controlador repassando os dados recebidos. O controlador, por sua vez, irá processar este pedido. Neste caso, o controlador deverá pedir à camada de modelo para verificar se a combinação de *login* e senha estão cadastradas, encaminhando estes dados. A camada de modelo realizará tal verificação consultando os dados armazenados e retornará uma resposta ao controlador. O controlador então analisará a resposta e tomará uma decisão baseada nela. Neste exemplo, se a resposta da camada de modelo for negativa (falsa) então o controlador enviará um comando para que a classe de visão associada ao formulário mostre uma mensagem de erro de autenticação. Caso contrário, se *login* e senha forem válidos, então o controlador requisitará à camada de modelo os dados do aluno com tais credenciais, retornando a resposta à camada de visão, a qual mostrará a próxima tela ao usuário.

3.2. PADRÃO 3-TIER

O padrão *3-Tier* ou Modelo em Três Camadas também é um padrão arquitetural de alto nível que visa à flexibilidade, manutenibilidade, reusabilidade, escalabilidade e confiabilidade empregando a ideia de separação do código de um software em partes interconectadas. Este modelo é derivado do modelo genérico de ‘n’ camadas e apresenta algumas semelhanças em relação ao modelo MVC, mas estes não são a mesma coisa. Este modelo arquitetura recebe este nome, pois o código do software é separação em três camadas principais:

- **Camada de apresentação**: é a camada responsável pela interação com o usuário, seja ela gráfica ou através do console ou outro dispositivo. É através desta camada que são mostrados os dados ao usuário e que se recebem as requisições de operações;
- **Camada de negócio**: é a camada responsável pela implementação das regras de negócio, ou seja, as restrições e o controle feito sobre as operações do sistema. Note que esta classe não deve ser responsável por mostrar nenhum formulário ou diálogo com o usuário e não deve persistir dados (armazenar dados durante a execução do programa). Em sistemas de médio a grande porte, nos quais existem diversas regras de negócio, é comum utilizar o padrão *façade*, a fim de centralizar as possíveis operações que podem ser realizadas no sistema;
- **Camada de dados**: é a camada responsável por gerenciar a persistência e recuperação de dados, seja ela local ou remota, volátil ou não volátil. Esta camada geralmente utiliza os *design patterns* singleton e *data access object* (DAO).

Além da separação lógica e bem delimitada de responsabilidades de cada camada, o modelo *3-Tier* emprega outra restrição de dependência: a camada de apresentação usa e depende da camada de negócio; a camada de negócio, por sua vez, usa e depende da camada de dados. A Figura 3 ilustra bem essa delimitação de responsabilidade e dependência entre camadas.

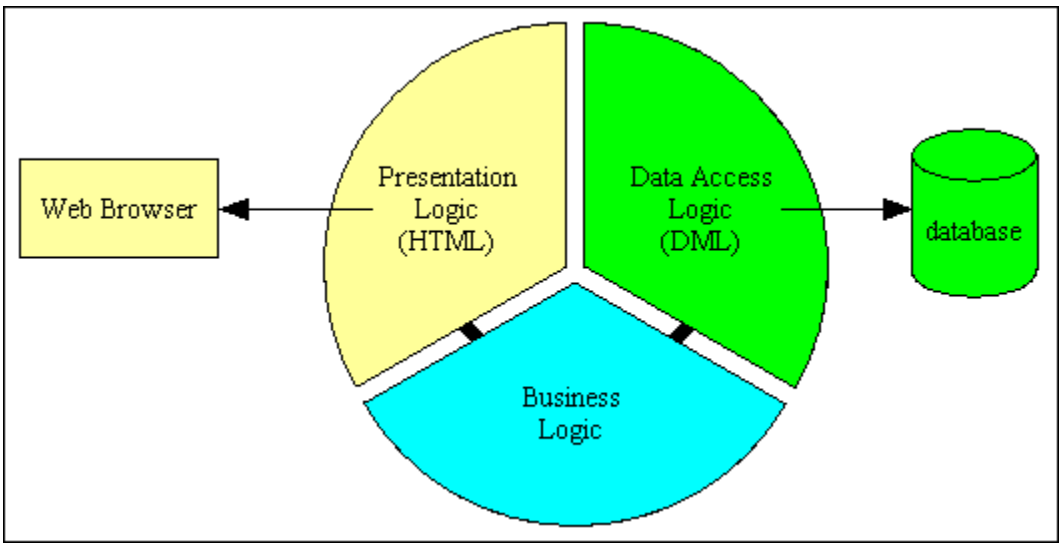


FIGURA 3 – Divisão de responsabilidade e dependência entre camadas considerando um caso de uma aplicação web que utiliza uma base de dados como repositório (Tony Marston, 2012)

Seguindo esta restrição de organização e dependências, pode-se perceber que para consultar um dado armazenado em um repositório de dados a requisição parte inicialmente do usuário para a camada de apresentação, que por sua vez encaminha o pedido para a camada de negócio. Esta camada pode eventualmente efetuar alguma validação referente as regras de negócio da aplicação e repassa a requisição a camada de dados, que enfim realiza o resgate da informação no repositório de dados. O resultado então é encaminhado para a camada de negócios que encaminha para a camada de apresentação, a qual apresenta de alguma forma este resultado ao usuário. A Figura 4 ilustra este processo de requisições e respostas.

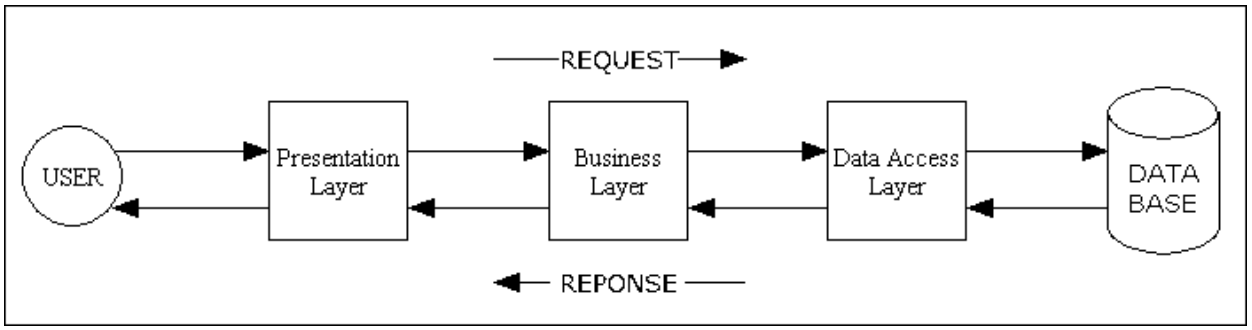


FIGURA 4 – Relação de requisição e resposta entre as camadas do sistema (Tony Marston, 2012)

Este processo de cadeia de requisições e respostas pode parecer custoso a primeira vista, mas como os dados são sempre replicados e retornados por referência, o custo destas requisições não é tão significativo para o sistema. Em contrapartida, a organização e a facilidade de manutenção posterior são significativamente aprimoradas. Um exemplo de código fonte para este modelo provavelmente ocuparia pelo menos duas páginas deste material. Para simplificar isto, analise o diagrama de classes apresentado na Figura 5, que apresenta o projeto de uma aplicação de pedidos de marmitas, no qual se destacam as classes da aplicação e suas responsabilidades em relação a algumas operações que o sistema realiza.

Tanto o modelo MVC quanto o modelo em três camadas podem ser mais custosos (trabalhoso e demorado) no início de seu desenvolvimento, mas os benefícios que trazem para alterações posteriores e a manutenção de código costumam compensar.

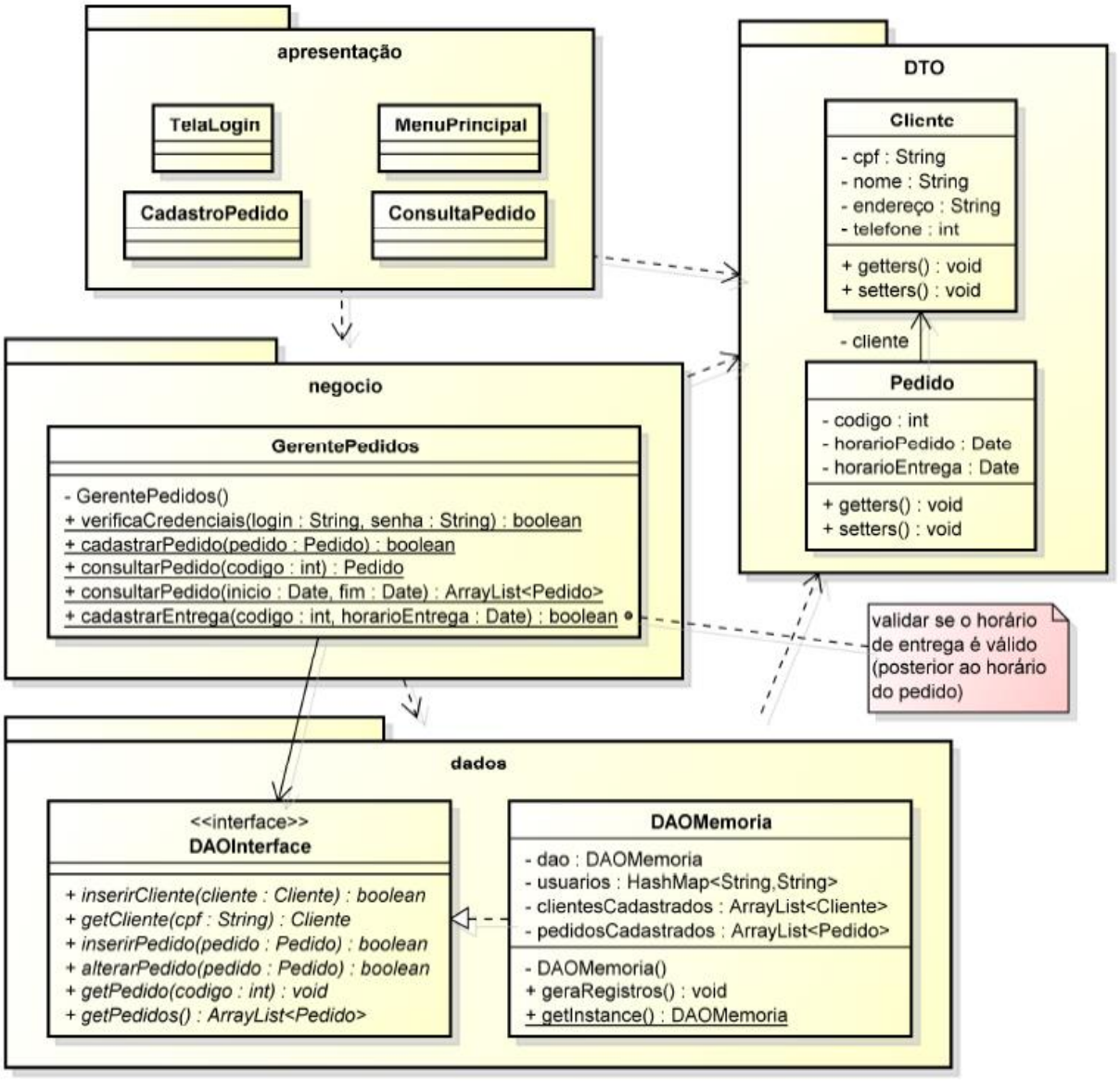


FIGURA 5 – Diagrama de classes de um sistema de pedido e entrega de marmitas usando o padrão arquitetural *3-Tier*s

Caso tenha mais interesse sobre o modelo *3-Tier*, uma boa fonte de dados (em inglês) que descreve com detalhes este modelo é o *website* http://www.tonymarston.net/php-mysql/3-tier-architecture.html, que fica como uma sugestão de leitura complementar.

Exercício

1) Escreva uma aplicação que permita a troca de mensagens assíncronas entre um grupo de cinco amigos (estilo *whatsapp* local). O sistema possui apenas três telas:

- **Tela de *login***: formulário no qual o usuário insere seu número de telefone e sua senha para se autenticar no sistema – os dados estão previamente cadastrados;
- **Tela de amigos**: formulário que mostra uma lista dos quatro outros usuários com uma prévia da última mensagem trocada entre o usuário *logado* e este amigo;
- **Tela de mensagens**: formulário que mostra todas as mensagens trocadas com um determinado amigo. Este formulário possui também um espaço para enviar uma nova mensagem que será salva no sistema.

O ponto inicial do sistema é a tela de *login*. Após entrar com as credenciais corretas, o usuário é levado ao formulário de amigos, que lista todos os quatro outros usuários, mostrando a última mensagem trocada e um botão para visualizar a conversa completa. Ao clicar sobre um destes botões, a tela de mensagens referente à troca de mensagens entre o usuário *logado* e o amigo selecionado é exibida com detalhes (todas as mensagens). Note que uma mensagem é composta pelo texto enviado e pela data de envio – além obviamente, dos dados de quem enviou e para quem a mensagem foi enviada.

Escreva dois *Data Access Objects* que são responsáveis por manter (salvar e resgatar) as mensagens trocadas entre os amigos em memória e em arquivo. A definição do cadastro dos amigos pode ser feita diretamente no código-fonte. Sabe-se que nenhum novo amigo acessará o sistema - ou seja, tamanho do grupo é fixo! Escolha apenas um DAO para ser usado pelo sistema por vez, sendo que ambos os DAOs deverão ter os seguintes métodos:

- `getMessages( senderPhone : int ) : ArrayList<Message>`
  `@description: seleciona e retorna todas as mensagens Enviadas pelo usuário que possui o telefone senderPhone`

- `sendMessage( msg : Message ) : boolean`
  `@description: adiciona uma nova mensagem na lista de mensagens salvas pelo sistema.`

- `login( phone : int , String password ) : boolean`
  `@description: verifica se o telefone passado está Cadastrado como usuário e valida a senha. Retorna true se o login for válido, caso contrário retorna false.`

OBS: modele uma classe usuário e mantenha um HashMap e um arquivo de cadastro de usuários para validar o login.