**Modelagem de Hipermídia**
**Prof. Giangiacomo Ponzo Neto**
**Aula7**

# Implementando o Diagrama de Classes

Podíamos chamar esse tópico de "persistência", porque vamos passar pelo assunto. No entanto, como alguém poderia dizer que nossa implementação não é exatamente persistência, vamos ser mais discretos e apenas nos referir ao Diagrama de Classes que vocês já conhecem. Qual a ligação entre essas duas coisas? Veremos mais adiante. Quanto à persistência, podemos dizer que se refere ao armazenamento duradouro de dados, ou seja: mesmo que um programa ou ambiente termine ou a máquina seja desligada, o dado continua existindo – daí o termo "persiste". Assim sendo, pode-se definir persistência, de maneira geral na área de computação, como a capacidade de alguma coisa continuar existindo enquanto seu suporte físico (um arquivo no HD, por exemplo) estiver íntegro.

## OO

Especificamente dentro da idéia de OO, os "objetos persistentes" são aqueles que permanecem existindo mesmo após o término da execução do programa. Eles estão normalmente relacionados a bancos de dados. Se o banco de dados for puramente OO (OODBMS-*Object Oriented DataBase Management System*), então a ligação é "direta": um objeto de uma classe acessado no programa corresponde a um objeto armazenado no banco e está garantida sua persistência de forma natural[1].

O problema é que os bancos de dados puramente OO não deram muito certo (pelo menos por enquanto), seja por incompetência dos fabricantes, seja por se acreditar que bancos OO têm uma curva de aprendizado mais longa ou são mais lentos, seja pelo fato de algumas modificações nas *queries* serem consideradas difíceis ou não suportadas, seja pelo enorme sucesso dos bancos relacionais tradicionais. Enfim, seja por que motivo for, a verdade é que, apesar dos bancos de dados puramente OO já existirem há tempos e de quase tudo no desenvolvimento de sistemas ter se rendido aos objetos, a maioria das aplicações comerciais (mesmo as orientadas a objeto) fazem uso de bancos relacionais, através do bom e velho SQL padrão.

Mas, modernamente, costumamos fazer a modelagem de todo o sistema usando OO (UML). Ah! Chegamos ao Diagrama de Classes... Onde estarão essas classes, se as telas (de cadastro, consultas, etc.) acessarem diretamente os dados no banco, via SQL? Provavelmente, lugar nenhum, pois seus métodos de inserir, alterar, etc. serão implementados diretamente em SQL pelo programa *front-end*. Então, qual o sentido de se fazer um Diagrama de Classes, se as classes não vão existir[2]? Aprender a fazer diagramas? Parece um pouco esquisito... A boa notícia é que existem soluções.

## Mapeamento Objeto-Relacional

Esse fato (ligar programas OO a bancos relacionais não-OO) é tão comum, que uma técnica foi criada para esse fim: a ORM (*Object-Relational Mapping*) ou Mapeamento Objeto-Relacional. Essa técnica garante a persistência dos objetos dentro de um programa OO, porque ajuda a fazer com que o programa enxergue as classes propostas no diagrama de classes e não as tabelas relacionais do banco de dados. Quem faz a tradução entre registros de tabelas e objetos é uma camada de *software* que pode ser chamada de "camada de mapeamento objeto-relacional", que é um tipo de "camada de persistência". O resultado é como se fosse um banco de dados orientado a objeto virtual, que pode ser acessado pela aplicação. Veja o esquema abaixo:


[1] Uma maneira de persistir objetos sem ter que usar necessariamente um banco de dados seria usar serialização. A serialização objetiva salvar o estado atual de um objeto, seqüenciando-o em bytes, para posterior utilização (deserializando-o). Utilizando essa técnica se podem gravar objetos, fazer sua transmissão remota, armazená-los em arquivos, etc. Não vamos cobrir este assunto aqui.
[2] Isso tem acontecido frequentemente nos sistemas desenvolvidos aqui como Trabalho de Conclusão de Curso. Cria-se o Diagrama de Classes, mas nem o programa (feito em Delphi, Java, PHP, etc.), nem o banco, contemplam essas classes.

Existem muitas ferramentas e tecnologias (Apache Cayenne, Hibernate, NHibernate, JPA, etc.) que auxiliam nesse mapeamento. Na nossa programação, no entanto, vamos usar um processo mais simples e didático, criando uma "camada" para implementar o Diagrama de Classes, usando apenas a própria linguagem. Essa classe deverá converter dados lidos de uma tabela do banco em valores de atributos de objetos e vice-versa ao armazenar esses dados.

Apesar de muito usadas, as técnicas de persistência em geral tem críticos que vêm de dois pontos de vista extremos:

1. Os que acham desnecessário seguir a modelagem UML (total ou parcialmente), desprezando o diagrama de classes, modelando os dados apenas a partir de diagramas como DER e/ou DTR e conversando diretamente com o banco via SQL, sem camadas intermediárias que, acreditam eles, podem inclusive piorar o desempenho do sistema.
2. Os que acham que a OO deveria ser seguida à risca, eliminando de vez os bancos relacionais (com o uso de bancos puramente OO), o que tornaria a camada de persistência desnecessária, pois os objetos persistiriam diretamente no banco.

A verdade é que cada uma dessas visões acima tem suas razões. Cabe a nós entendermos as diferenças para tomarmos uma melhor decisão. Se o sistema for algo pequeno, com poucas tabelas, talvez não seja necessário se "perder" tempo fazendo uma implementação de persistência (seja de que tipo for). E, se os bancos OO se popularizarem, talvez seja bom usá-los de vez, eliminando a camada de tradução e conversando direto com o banco em termos de classes e objetos, não mais tabelas. Mas, enquanto a maioria dos sistemas continuar lidando com os dois mundos e os desenvolvedores olharem a persistência como algo útil, vamos sempre ouvir falar sobre ela.

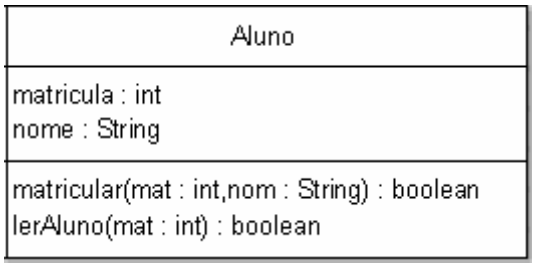
**Uma Classe do Diagrama de Classes em Java**

Como se cria uma camada de persistência? Como ela se parece? Existem várias formas de se implementar a persistência, com ou sem o uso de ferramentas externas, com uma ou mais camadas, seguindo a metodologia "x", "y", ou "z", etc. Nós vamos fazer da forma mais simples e didática possível: vamos criar uma classe que reflita exatamente o que o Diagrama de Classes propuser[3].

Vamos supor uma tabelinha de alunos em um banco de dados relacional, com a seguinte estrutura:

| **Campo** | **Tipo(Tamanho)** |
|---|---|
| Matricula | Int |
| Nome | Varchar(40) |

Vamos supor agora que o programa que deveria tratar essa tabela foi modelado em UML e deveria enxergar uma classe de alunos, com a seguinte estrutura:




O principal papel da nossa classe será traduzir de um modelo para o outro de modo que o programa consiga trabalhar 100% OO e o banco 100% relacional. Vamos programar a classe Aluno mostrada acima. Em Java, seria algo assim:

```
package dao;
```

[3] Por isso preferi não chamar o tópico de "persistência", pois alguns podem dizer que é apenas uma classe do diagrama e não estritamente persistência.

```
import java.sql.*;

public class Aluno {

public Aluno() {
  try {
    Class.forName(*** driver JDBC desejado ***);
    con=DriverManager.getConnection(*** URL de conexão, login, senha ***);
    stmt = con.createStatement();
  }
  catch (SQLException ex) {
    ex.printStackTrace();
  }
  matricula=-1;
  nome="";
}

  // atributos:
  private int matricula;
  private String nome;

  // objeto Connection
  private Connection con;
  // statement para usar o SQL no banco
  private Statement stmt;


  // métodos get/set

  public int getMatricula() {
    return matricula;
  }

  public String getNome() {
    return nome;
  }

  public void setMatricula(int mat) throws Exception {
    if (matricula>-1) {
      String m=Integer.toString(mat);
      String mAnterior= Integer.toString(matricula);
      try {
        stmt.execute("update aluno set matricula="+m+" where matricula="+mAnterior);
        matricula = mat;
      }
      catch (Exception e) {
        throw new Exception("erro: "+e.toString());
      }
    }
    else throw new Exception("Aluno não foi matriculado ainda");
  }

  public void setNome(String nom) throws Exception {
    if (matricula>-1) {
      String m= Integer.toString(matricula);
      try {
        stmt.execute("update aluno set nome="+nom+" where matricula="+m);
        nome = nom;
      }
      catch (Exception e) {
        throw new Exception("erro: "+e.toString());
      }
    }
    else throw new Exception("Aluno não foi matriculado ainda");
  }

  // métodos adicionais definidos no projeto da classe Aluno

  public boolean matricular(int mat,String nom) {
    String m=Integer.toString(mat);
    try {
      stmt.execute("insert into aluno values ("+m+",'"+nom+"')");
      matricula = mat;
      nome = nom;
      return true;
    }
    catch (Exception e) {
      return false;
    }
```

```
  }

  public boolean lerAluno(int mat) {
    String m=Integer.toString(mat);
    try {
      ResultSet rs=stmt.executeQuery("select * from aluno where matricula="+m);
      if (rs.next()) {
        matricula=rs.getInt("matricula");
        nome=rs.getString("nome");
        return true;
      }
      else return false;
    }
    catch (Exception e) {
      return false;
    }
  }
}
```

Essa classe fica sendo responsável pela ligação entre o programa (*front-end* e algumas regras de entrada) e o banco de dados, cumprindo dois papéis: define as características de um objeto Aluno e faz a ligação com a tabela Aluno do banco. Um nome encontrado na literatura para esse tipo de solução seria "camada DAO" (*Data Access Objects*), por isso chamei o pacote de "dao"[4]. Vamos a algumas considerações importantes:

- A classe Aluno acima é uma classe Java como tantas outras que já vimos. Quase um POJO[5], acrescido de alguns métodos. Repare que os *getters* e *setters* foram criados, embora nem figurem no Diagrama de Classes.
- Neste exemplo, o construtor padrão Aluno( ) foi responsável por garantir que essa classe enxergasse o banco de dados. Se existirem várias classes semelhantes para outros objetos persistentes, podemos usar alguma técnica de conexão pública ou reaproveitamento de conexões (*connection pooling*), dependendo do sistema ser *desktop* ou *web*.
- No construtor, iniciamos a matrícula com -1. Um pequeno "truque" para saber se o objeto já foi usado ou não (já que um aluno não pode ter uma matrícula negativa). O uso disso fica evidente nos métodos setMatricula( ) e setNome( ) – verifique.
- Os métodos setMatricula( ) e setNome( ) não só atualizam os atributos correspondentes, como fazem *update* no banco, mas só funcionam se o aluno já tiver sido matriculado, senão emitem um erro com o "throw Exception" (optou-se por fazer assim nesse exemplo, não que seja obrigatório agir dessa maneira).
- Os métodos getMatricula( ) e getNome( ) só têm sentido após ler um aluno do banco (senão retornam os valores iniciais: -1 e vazio).
- O método matricular( ) serve para inserir um aluno no banco. Ele retorna *true* em caso de sucesso ou *false* se falhar.
- O método lerAluno( ) serve para recuperar um aluno já gravado no banco. Ele retorna *true* se achar a matrícula ou *false* se não encontrar ou algo der errado.
- Repare que, depois de matricular ou ler um aluno, o objeto tem seus atributos atualizados com os valores de matrícula e nome, de modo que o programa possa fazer uso deles.
- Existem ferramentas que criam estas camadas DAO automaticamente a partir de tabelas já existentes em um banco (cito o FireStorm/DAO como exemplo). Como já aprendemos como elas se parecem, podemos usar as ferramentas com mais propriedade.

Para usar esta classe, bastaria a um programa de cadastro de Aluno acessar os métodos disponíveis, sem incluir uma única linha SQL. Veja um exemplo básico em JSP abaixo:

```
<%@ page import="dao.Aluno" %>
<html>
<body>
  <%
     int matricula=1001;
     String nome="Mario";
     Aluno alu1 = new Aluno();
     if (alu1.matricular(matricula,nome)==true) {
  %>
       Aluno criado e matriculado!
  <% }
```

[4] Implementações de DAO em Java normalmente conduzem à separação entre uma classe contendo um atributo para cada campo de uma tabela e outra classe com sufixo "DAO" contendo métodos para *insert*, *update*, *delete* e *select* de linhas da tabela, mas aqui simplificamos mais um pouco, colocando tudo em uma só classe.

[5] POJO (*Plain Old Java Object*) é um tipo de classe simples de objetos com seus atributos e métodos *get*/*set*.

```
      else {
  %>
        Problema na gravação!
  <% } %>
</body>
</html>
```

Repare a simplicidade da página acima, que não se refere em nenhum momento ao banco de dados, nem usa SQL. Apenas instancia um objeto aluno (alu1), que é persistente. Procedendo dessa forma, pelo menos a camada de apresentação (a "tela" para o usuário) está desacoplada do acesso direto ao banco de dados. O "problema" é que a classe de Alunos mesclou métodos e atributos de alunos com o acesso ao banco em SQL, o que, para alguns, denota a necessidade de mais camadas e um desacoplamento ainda mais acentuado.

Dá trabalho no início, é verdade, mas além de ficar elegante, daqui para frente, para qualquer coisa que se queira com o aluno – incluindo as operações com o banco de dados – basta se referir à sua classe.

Exercícios

1. Suponha que a tabela receba mais um campo (CPF). Faça as alterações necessárias na classe para que o programa enxergue esse novo atributo do aluno.
2. Crie o método eliminaAluno( ) dada uma matrícula.
3. Crie o método encontraAluno( ) dado um nome.

Há que se considerar um detalhe: o mapeamento de uma classe para uma tabela pode não ser sempre direto, um para um, como fizemos nesse pequeno exemplo (embora o mapeamento objeto-relacional mais estrito assim o faça). Se usarmos um esquema DAO mais simples, como o nosso, podemos fazer classes se referirem a mais de uma tabela e vice-versa, já que os modelos de tabelas de dados vindos de um DTR (Diagrama de Tabelas Relacionais) normalmente não coincidem totalmente com as classes de um Diagrama de Classes UML.

**Uma Classe do Diagrama de Classes em Delphi**

Como sabemos, muitos alunos optam pelo Delphi (no padrão *desktop win32*) na hora de implementar seus projetos de TCC. Mesmo um pouco já fora dos *spots*, como o Delphi é um IDE RAD (*Rapid Application Development*), essa escolha é plenamente compreensível. Embora seja uma excelente ferramenta, o "problema" com o Delphi é que, justamente pela sua RAD*icalidade* (perdão pelo trocadilho infame), acaba levando naturalmente ao uso direto do SQL de dentro dos *forms* de tela, ainda mais tendo-se em conta que os projetos costumam ser relativamente pequenos. De qualquer forma, se os alunos quiserem, podem usar o que vimos no Java e aplicar no Delphi (garanto que muitos sequer perceberam as classes no Delphi, mas elas estão lá).

Antes de tudo, vamos criar um projetinho simples com uma tabela idêntica a que usamos acima. Pode usar o banco que quiser, mas para não ficar na dependência de ferramentas externas, eu sigo aqui com o Paradox velho de guerra. Veja a seqüência abaixo:

1. Abra o Delphi e crie a tabela aluno.db, conforme figura 1 abaixo no Database Desktop (Tools-Database Desktop-File-New-Table-Paradox7). Salve como aluno.db em um diretório (sugiro c:\Delphi). Esse diretório será usado para todo o projeto.
2. Crie uma tela como indicado na figura 2, troque o *name* do *form* para frmMatAluno e salve a *unit* como untMatAluno.pas no mesmo diretório criado para a tabela.
3. Salve o projeto como Aluno.dpr.
4. Crie um novo *form* (File-New-Form) vazio (figura 3), troque o *name* para frmAlunoDAO e salve a *unit* como untAlunoDAO.pas.
5. Programe a nova *unit* untAlunoDAO conforme está na listagem 1.
6. Volte à primeira *unit* e use a segunda (`uses untAlunoDAO;`) após o "implementation".
7. Após o *uses* acima, crie um objeto aluno: `var alu1:Aluno;`
8. No evento OnCreate de frmMatAluno, programe `alu1:=Aluno.create;`
9. No botão matricular, programe:

```
if alu1.matricular(StrToInt(edtMat.text),edtNome.text)=true then
  showmessage('Aluno criado e matriculado!')
else
  showmessage('Problema na gravação!');
```

Field roster:

| | Field Name | Type | Size | Key |
|---|---|---|---|---|
| 1 | Matricula | I | | * |
| 2 | Nome | A | 40 | |

Figura 1 - Estrutura da tabela

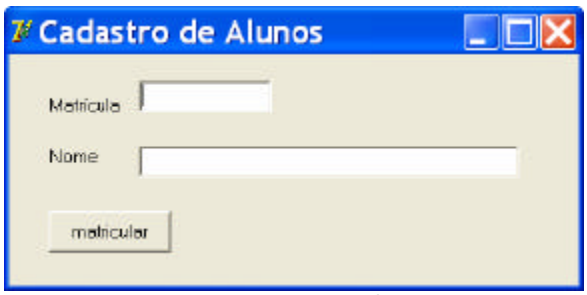



Figura 2 – Tela de cadastro (frmMatAluno)


Figura 3 – *Form* da classe (frmAlunoDAO) vazio

```
unit untAlunoDAO;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DB, DBTables;

type
  TfrmAlunoDAO = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

  // classe Aluno

  Aluno = class
  private
    matricula:integer;
    nome:string;
    qryAluno: TQuery;
  public
    constructor create;
    function getMatricula:integer;
    function getNome:string;
    procedure setMatricula(mat:integer);
    procedure setNome(nom:string);
    function matricular(mat:integer; nom:String):boolean;
    function lerAluno(mat:integer):boolean;
  end;

var
  frmAlunoDAO: TfrmAlunoDAO;

implementation

{$R *.dfm}
```

```
// construtor da classe Aluno

constructor Aluno.create;
begin
  matricula:=-1;
  nome:='';
  qryAluno:=TQuery.Create(frmAlunoDAO);
end;

// métodos get/set

function Aluno.getMatricula:integer;
begin
  result:=matricula;
end;

function Aluno.getNome:string;
begin
  result:=nome;
end;

procedure Aluno.setMatricula(mat:integer);
var m,mAnterior:string;
begin
  if (matricula>-1) then
  begin
    m:=IntToStr(mat);
    mAnterior:=IntToStr(matricula);
    try
      qryAluno.close;
      qryAluno.sql.clear;
      qryAluno.sql.Add('update aluno set matricula='+m+' where matricula='+mAnterior);
      qryAluno.execSQL;
      matricula:=mat;
    except
      on e:exception do
        showmessage('erro: '+e.Message);
    end;
  end
  else showmessage('Aluno não foi matriculado ainda');
end;

procedure Aluno.setNome(nom:String);
var m:string;
begin
  if (matricula>-1) then
  begin
    m:=IntToStr(matricula);
    try
      qryAluno.close;
      qryAluno.sql.clear;
      qryAluno.sql.Add('update aluno set nome='+nom+' where matricula='+m);
      qryAluno.execSQL;
      nome:=nom;
    except
      on e:exception do
        showmessage('erro: '+e.Message);
    end;
  end
  else showmessage('Aluno não foi matriculado ainda');
end;

// métodos adicionais definidos no projeto da classe Aluno

function Aluno.matricular(mat:integer; nom:String):boolean;
var m:string;
begin
  m:=IntToStr(mat);
  try
    qryAluno.close;
    qryAluno.sql.clear;
    qryAluno.sql.Add('insert into aluno values ('+m+','''+nom+''')');
    qryAluno.execSQL;
    matricula:=mat;
    nome:=nom;
    result:=true;
  except
    result:=false;
  end;
end;
```

```
function Aluno.lerAluno(mat:integer):boolean;
var m:string;
begin
  m:=IntToStr(mat);
  try
    qryAluno.close;
    qryAluno.sql.clear;
    qryAluno.sql.Add('select * from aluno where matricula='+m);
    qryAluno.open;
    if not qryAluno.EOF then
    begin
      matricula:=qryAluno.fieldByName('matricula').asInteger;
      nome:=qryAluno.fieldByName('nome').asString;
      result:=true;
    end
    else
      result:=false;
  except
    result:=false;
  end;
end;

end.
```

Listagem 1 – A *unit* untAlunoDAO.pas, relativa ao *form* frmAlunoDAO

Vamos analisar o que foi feito. Antes de qualquer coisa, veja que, sempre que uma tarefa é um pouco mais "profunda", há mais trabalho braçal, mesmo usando uma ferramenta como o Delphi. É verdade que, se usássemos um *framework* de persistência, poderíamos programar menos "na mão". O Delphi 2006 Architect, por exemplo, vem com o ECO III, que tem essa funcionalidade. Outro produto semelhante é o DePO (*Delphi Persistent Objects*). Só que aí existiria o tempo de aprendizado dessas ferramentas, que deve ser levado em conta. Mas vamos ao nosso caso...

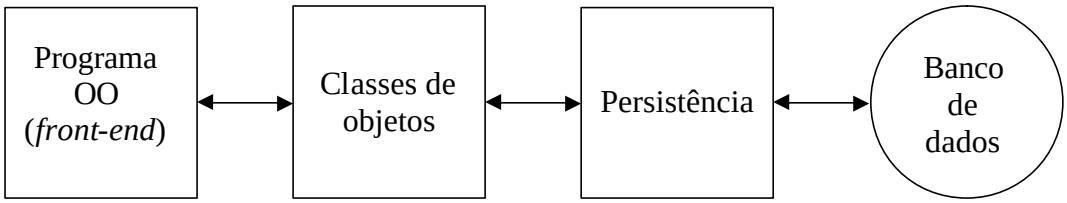
- Na listagem 1, em "type... Aluno=class" estamos definindo uma classe chamada Aluno que é subclasse de TObject (a classe mais genérica do Delphi) e herda funcionalidades mínimas, como o construtor "create" que usamos para criar o objeto Aluno na outra *unit*. Você já tinha visto o Delphi tão "OO" assim?
- Repare que um dos atributos de Aluno é outro objeto. Especificamente, um objeto TQuery, que permite interagir com o banco de dados, via SQL. Se não fosse por esse componente, que é da VCL (*Visual Component Library*) e precisa ser criado com um parâmetro que indica outro componente como "*owner*", nem precisaríamos ter um *form*, bastaria uma *unit*.
- Os demais atributos e métodos de Aluno são iguais aos homônimos que criamos na versão Java, resguardadas as diferenças entre as linguagens, obviamente.
- Outra pequena diferença é que, em Java, eu levantei exceções em alguns casos, enquanto no Delphi só mandei mensagens de erro. Isso porque, no Java, eu supus que a classe pudesse ser usada tanto em versão *desktop* quanto *web*, mas no Delphi *win32* só temos a versão de janela e eu posso enviar a mensagem de erro diretamente ao usuário.

Mais discussão sobre o assunto em sala. Sugiro fazer os mesmos exercícios propostos para o Java e implementar um programa que use todos os métodos criados.

**DAO, ORM, Ferramentas Externas...**

Como já adiantei, este tipo de persistência que vimos (que alguns poderiam dizer que nem o é) ainda poderia ser mais "refinado". A camada que chamamos de DAO poderia ser "quebrada" em duas, pois muitos consideram que uma camada dessas não deveria agregar os atributos e métodos do objeto; mas apenas a sua ligação com o banco de dados. Se procurarmos na *web*, vamos encontrar várias diferenças entre as formas de usar DAO ou ORM e, especificamente no mundo Java: JDO, EJB3, JPA, etc. Se usarmos ferramentas externas, podemos até optar por uma dessas soluções para implementar persistência. Mas, entre tantas "firulas" tecnológicas, não devemos esquecer o motivo principal dessa camada: fazer o programa enxergar objetos e não registros em tabelas, se quisermos que o programa seja realmente OO e o banco não.

Usando um tipo de persistência mais "refinado" poderíamos acabar criando mais divisões, chegando, por exemplo, ao seguinte esquema:




Lembrando que as divisões acima poderiam ser divididas também, gerando mais e mais camadas. Podemos refinar um sistema o quanto quisermos e chegar a várias dessas camadas, mas até onde devemos ir? Ficar separando muito as funcionalidades pode levar a dúzias de pequenas classes que não fazem quase nada, sendo especializadas demais. Isso realmente pode chegar a um ponto que atrapalhe o desenvolvimento. Nosso próximo assunto será justamente essa possibilidade de dividir um sistema em camadas e como isso se desenvolveu historicamente. Com isso, começamos lentamente a entrar na parte mais "teórica" da disciplina.