



# Desenvolvimento de Sistemas Software

## ORM (Object-Relational Mapping)

# Sobre o trabalho

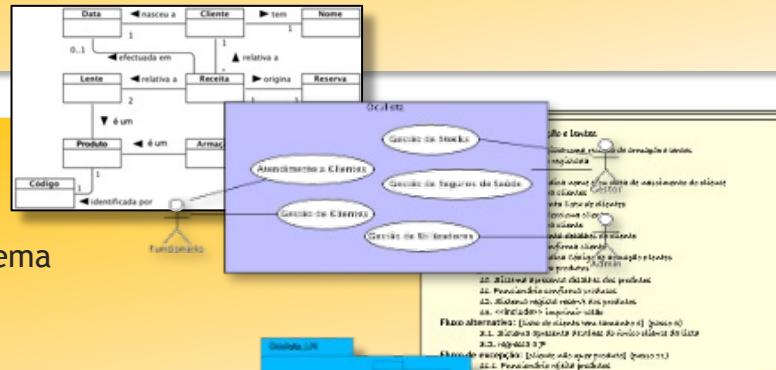
*"Problema"*

## Planeamento

- Decisão de avançar com o projecto
- Gestão do projecto

## Análise

- Análise do domínio do problema
- Análise de requisitos

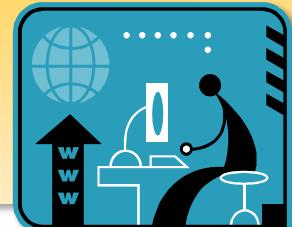


## Concepção

- Concepção da Arquitectura
- Concepção do Comportamento

## Implementação

- Construção
- Teste
- Instalação
- Manutenção



**"Problema"**

Modelação do Domínio



Use Case: Reservar Armação e Lente.  
Descrição: Possibilitar realização de um novo pedido de reservação de óculos.  
Pós-condição: Reserva é feita respeitando:  
Fluxo normal:  
1. Reservar óculos é iniciada com o novo pedido de reservação de óculos.  
2. Utilizadora preenche cliente.  
3. Utilizadora apresenta lente e armação.  
4. Utilizadora indica tipo de óculos.  
5. Utilizadora indica óculos.  
6. Utilizadora indica óculos de sol.  
7. Utilizadora indica óculos de sol.  
8. Utilizadora indica óculos de sol.  
9. Utilizadora indica óculos de sol.  
10. Utilizadora indica óculos de sol.  
11. Utilizadora indica óculos de sol.  
12. Utilizadora indica óculos de sol.  
13. Utilizadora indica óculos de sol.  
14. Utilizadora indica óculos de sol.  
15. Utilizadora indica óculos de sol.  
16. Utilizadora indica óculos de sol.  
17. Utilizadora indica óculos de sol.  
18. Utilizadora indica óculos de sol.  
19. Utilizadora indica óculos de sol.  
20. Utilizadora indica óculos de sol.  
21. Utilizadora indica óculos de sol.  
22. Utilizadora indica óculos de sol.  
23. Utilizadora indica óculos de sol.  
24. Utilizadora indica óculos de sol.  
Fluxo alternativo: [Lentes não existentes na base de dados]  
25. Utilizadora apresenta óculos que não existem na base de dados.  
26. Utilizadora indica óculos que não existem na base de dados.  
Fluxo de exceção: [Lentes não correspondem ao pedido] (ReservarArmaçãoLenteException)  
27. Utilizadora indica óculos que não correspondem ao pedido.  
28. Utilizadora indica óculos que não correspondem ao pedido.

Identificação de Use Cases



Especificação dos Use Case

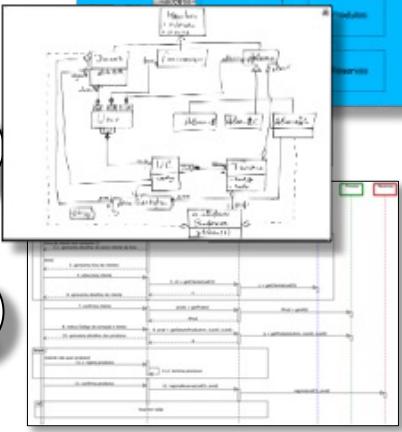


Seleção de Use Case

Modelação Conceptual/lógica

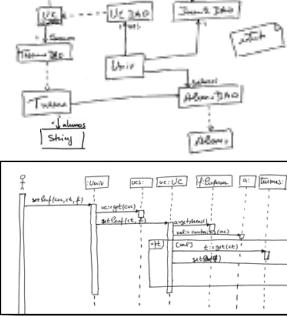
[mais UCs a tratar]

[modelação conceptual completa]



Seleção de Use Case

Modelação Detalhada



Implementação



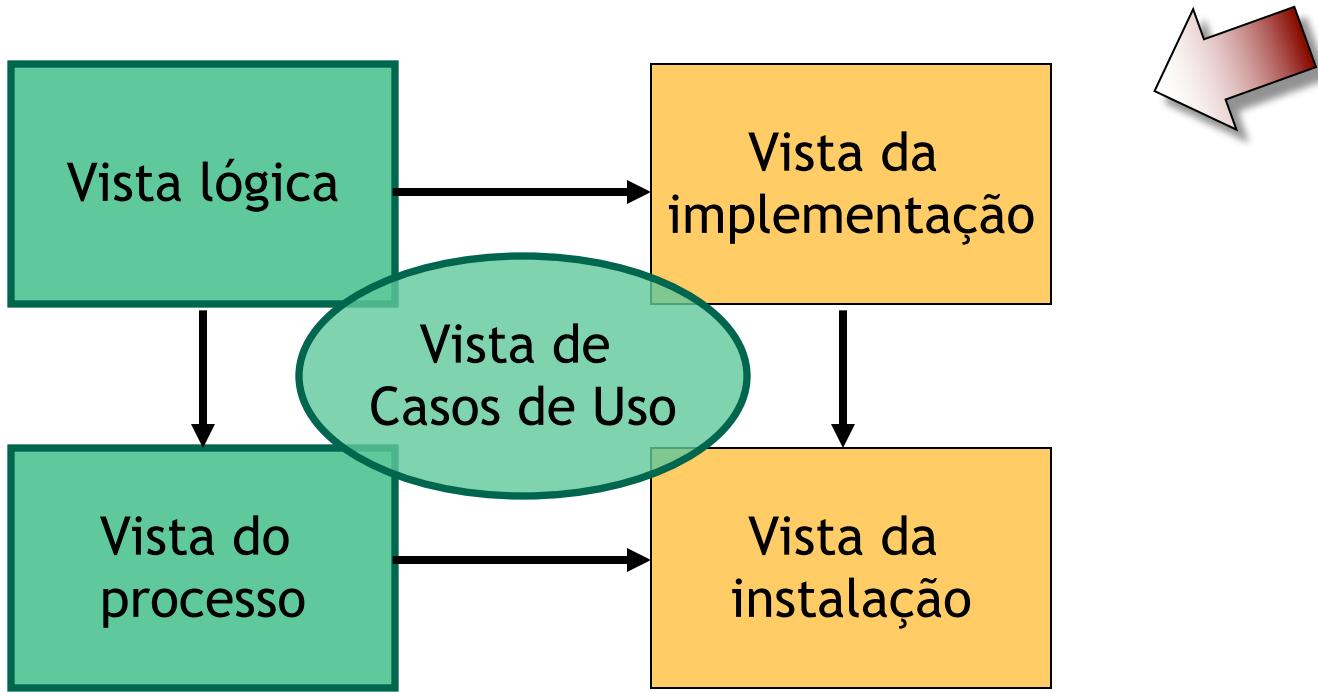
[mais UCs a tratar]

[implementação completa]

Instalação



# Onde estamos...

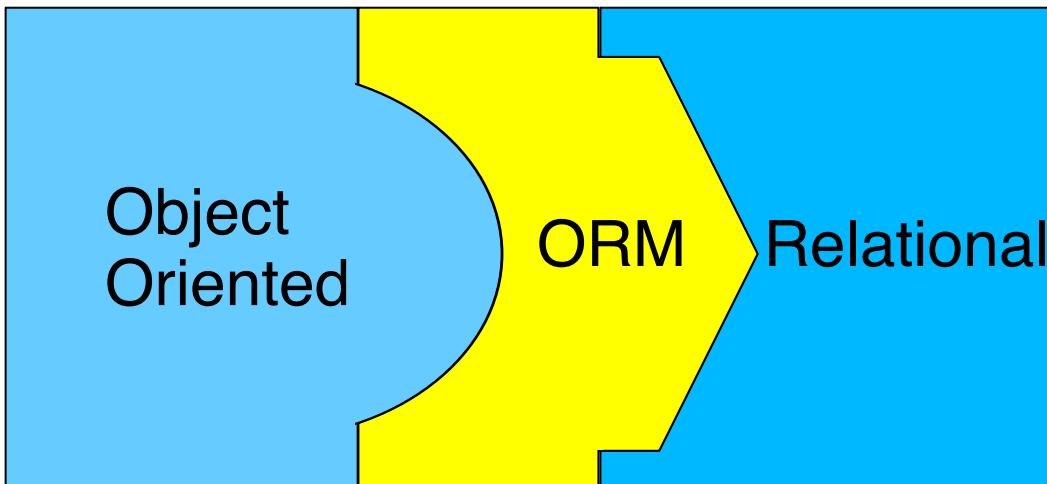


(Kruchten, 1995)

# Object Relational Mapping - ORM

Camada de Negócio

Persistência de dados

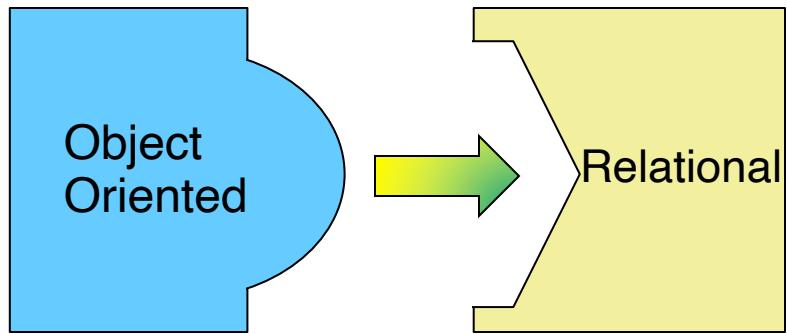
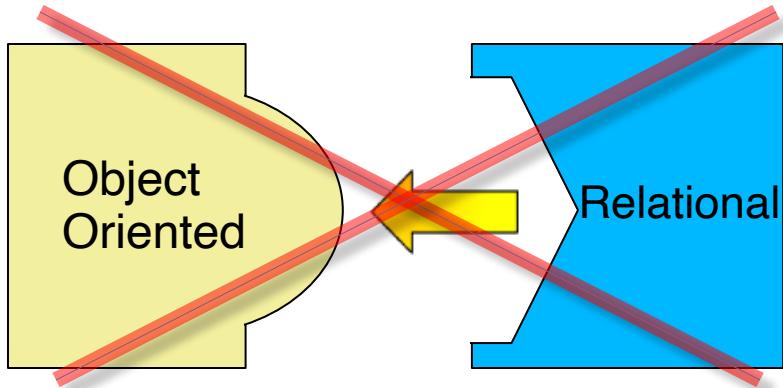


- Tecnologias OO
  - Independente da camada de persistência
- Bases de dados relacional
  - ou NoSQL
  - ou orientada a objectos
  - ou...

# Paradigma OO vs Relacional

- Paradigma OO prevalente na programação da lógica de negócios
- Paradigma relacional prevalente nas bases de dados (persistência)
- Paradigma orientado a objetos e paradigma relacional não são diretamente compatíveis
  - Diagramas de classe não são esquemas de base de dados!
  - Modelo relacional não possui características presentes no modelo OO como polimorfismo ou identidade.
  - Objectos possuem identidade.
  - Linhas numa tabelas são identificadas por chaves.
- Torna-se necessário estabelecer um mapeamento entre os dois paradigmas - **Object Relational Mapping (ORM)**

## Abordagens ao ORM



- **Derivar objectos a partir das tabelas**
- Objectos servem apenas para aceder às tabelas
- Tendência para se perder separação de camadas:
  - Lógica de negócio dependente de modelo relacional
  - Lógica de negócio no modelo relacional
- **Vantagens do paradigma OO?**
  - Não se está a fazer desenvolvimento OO
  - Não se tira partido do paradigma OO
- **Derivar tabelas a partir do objectos**
- Lógica de negócio desenvolvida independentemente da Base de Dados
- Base de dados utilizada como serviço de persistência de dados
- Possível explorar vantagens do paradigma OO
- Resulta melhor quando lógica de negócio é complexa

# Lógica de negócio na Base de dados?

- Um dos princípios base das arquitecturas em camadas é a existência de uma camada de lógica de negócio
- Vantagens de manter a lógica de negócio separada da Base de Dados
  - poder expressivo das linguagens OO
  - facilidade de compreensão (debug, manutenção, ...)
  - escalabilidade
- Justificável colocar lógica na Base de Dados
  - Operações *data intensive*
  - Lógica dos Dados
  - Segurança
  - Desempenho
  - (mas devemos manter a separação através de uma camada de acesso...)

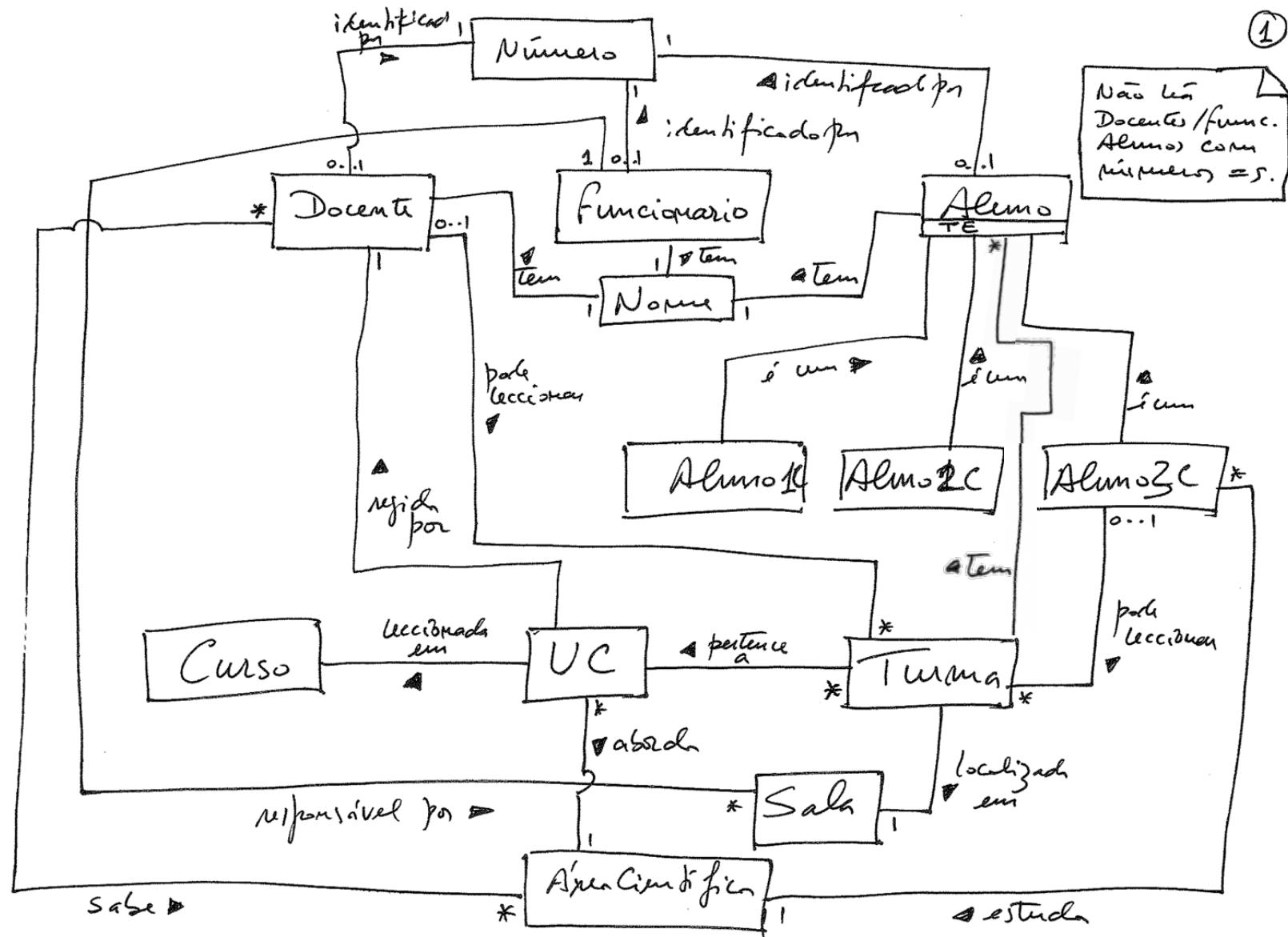
# Persistência???

- É necessário mapear as entidades/classes em tabelas
- É necessário mapear os atributos das entidades/classes em colunas das tabelas
  - Tipicamente mapeamento 1-para-1 mas...
    - alguns atributos podem ser mapeados para mais que uma coluna (ex.: nome ser dividido em 'nome próprio' e 'apelido')
    - dois ou mais atributos podem ser mapeados para uma mesma coluna (prefixo e número de telefone, por exemplo).
- **Mas, primeiro é necessário decidir que entidades/classes persistir**

# Persistência???

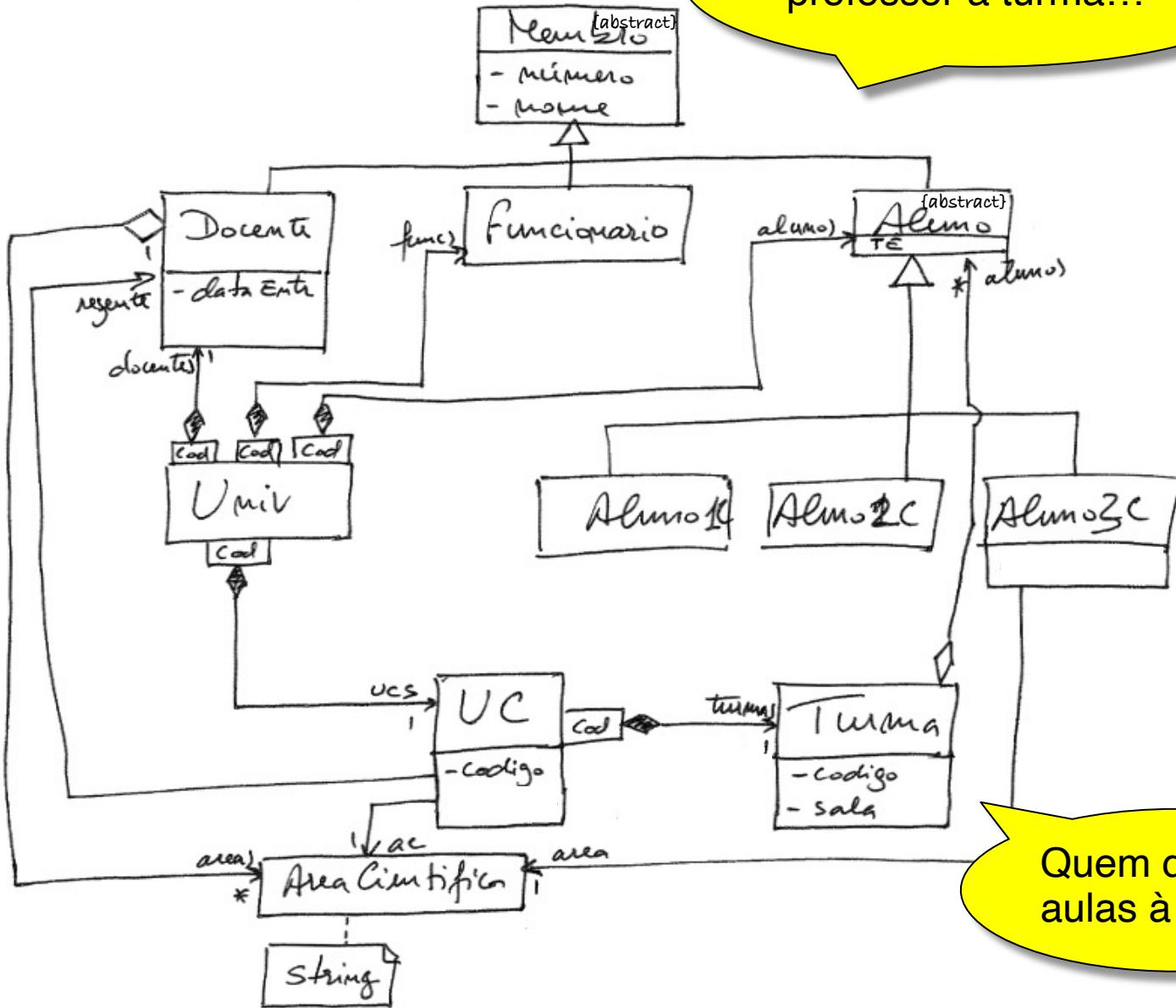
- Que entidades/classes persistir?
  - Que classes “armazenam” dados?
  - Essas serão as que devem ser persistidas
- Tipicamente as entidades também encapsulam algum controlo
  - Esse não é representado na camada de dados.
  - Existirão algumas classes que apenas existem para implementar o controlo da lógica de negócio
  - Logo, essas não é necessário persistir.

# Exemplo - Um Modelo de Domínio...



## Arquitectura parcial...

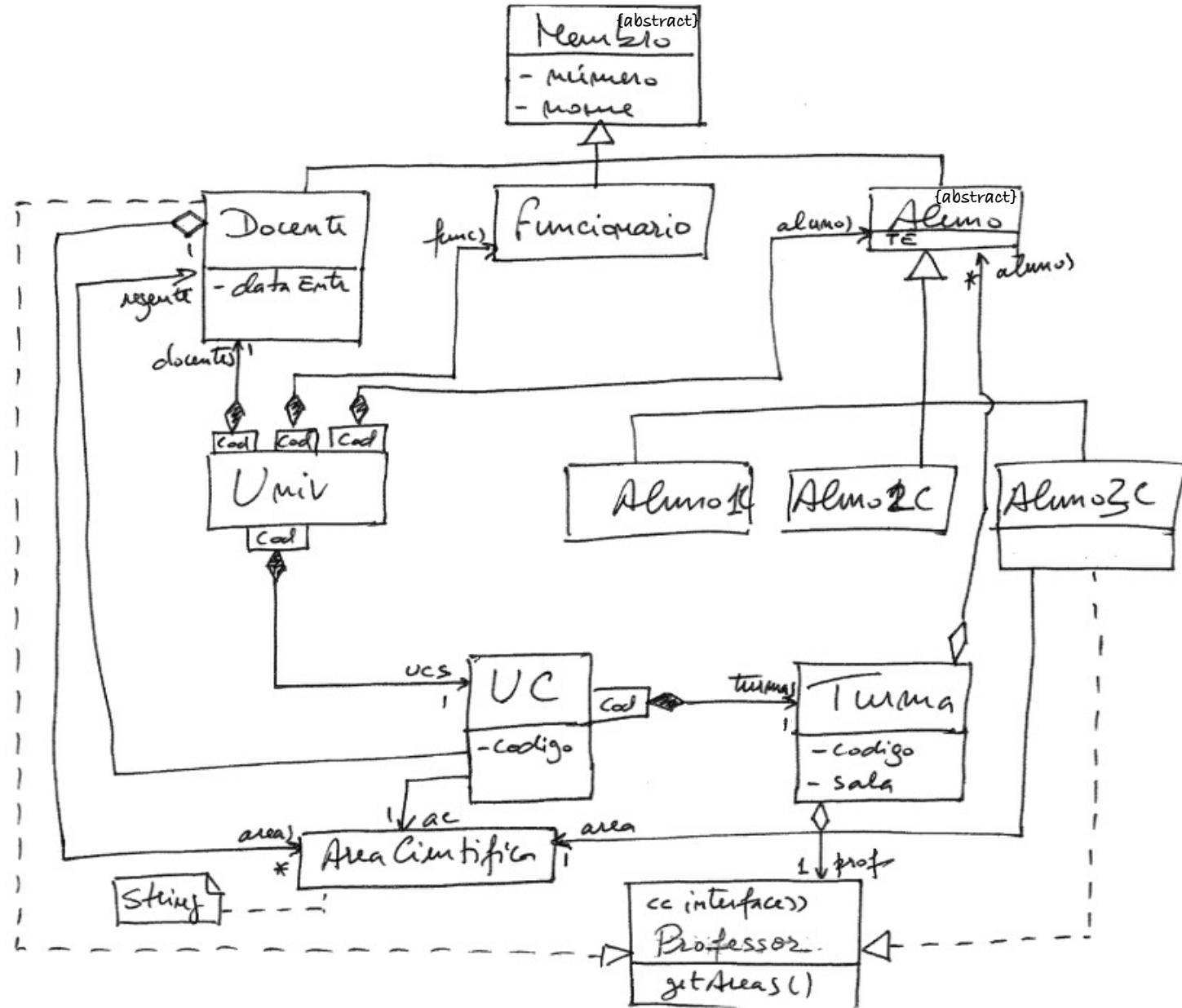
## Use Case atribuir professor a turma...



# Quem dá aulas à turma?

# Arquitectura parcial...

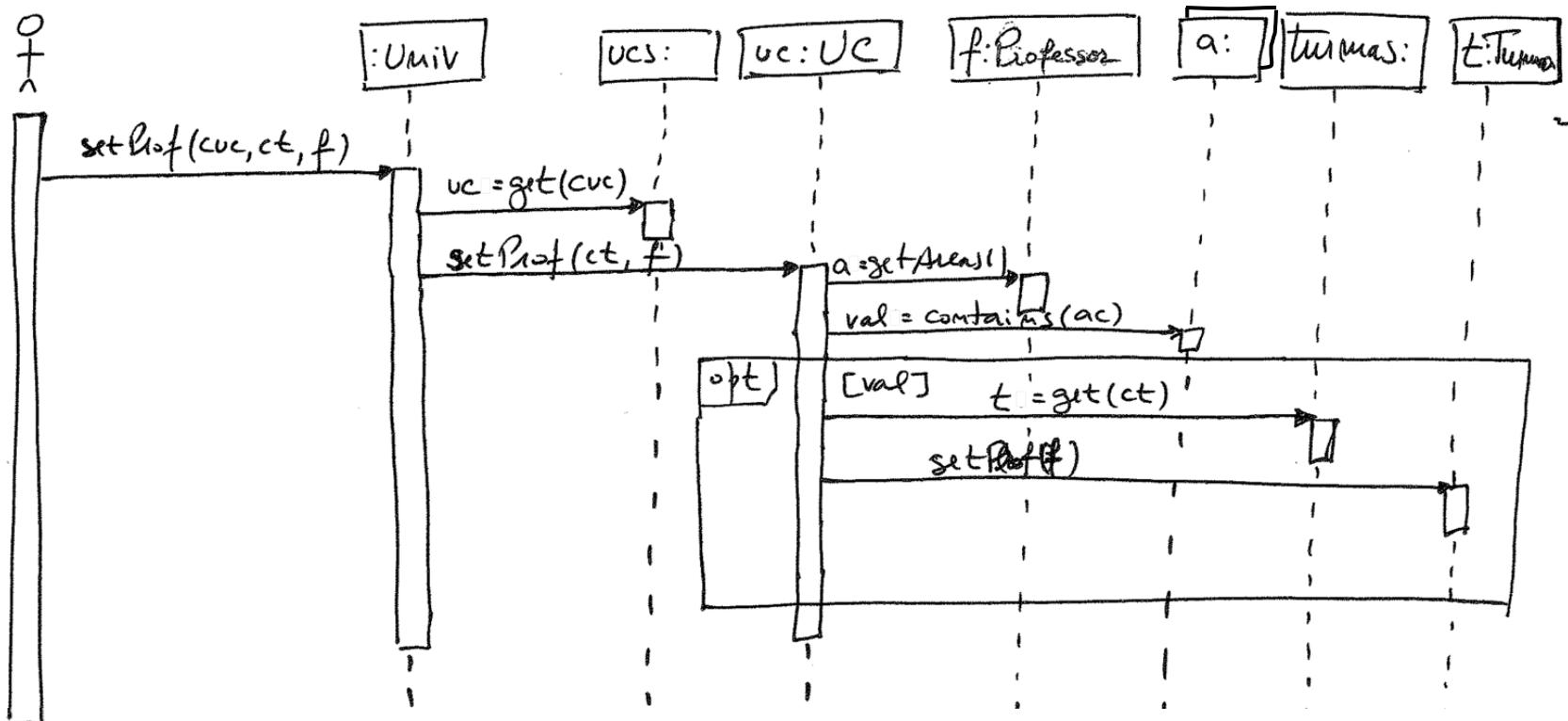
3





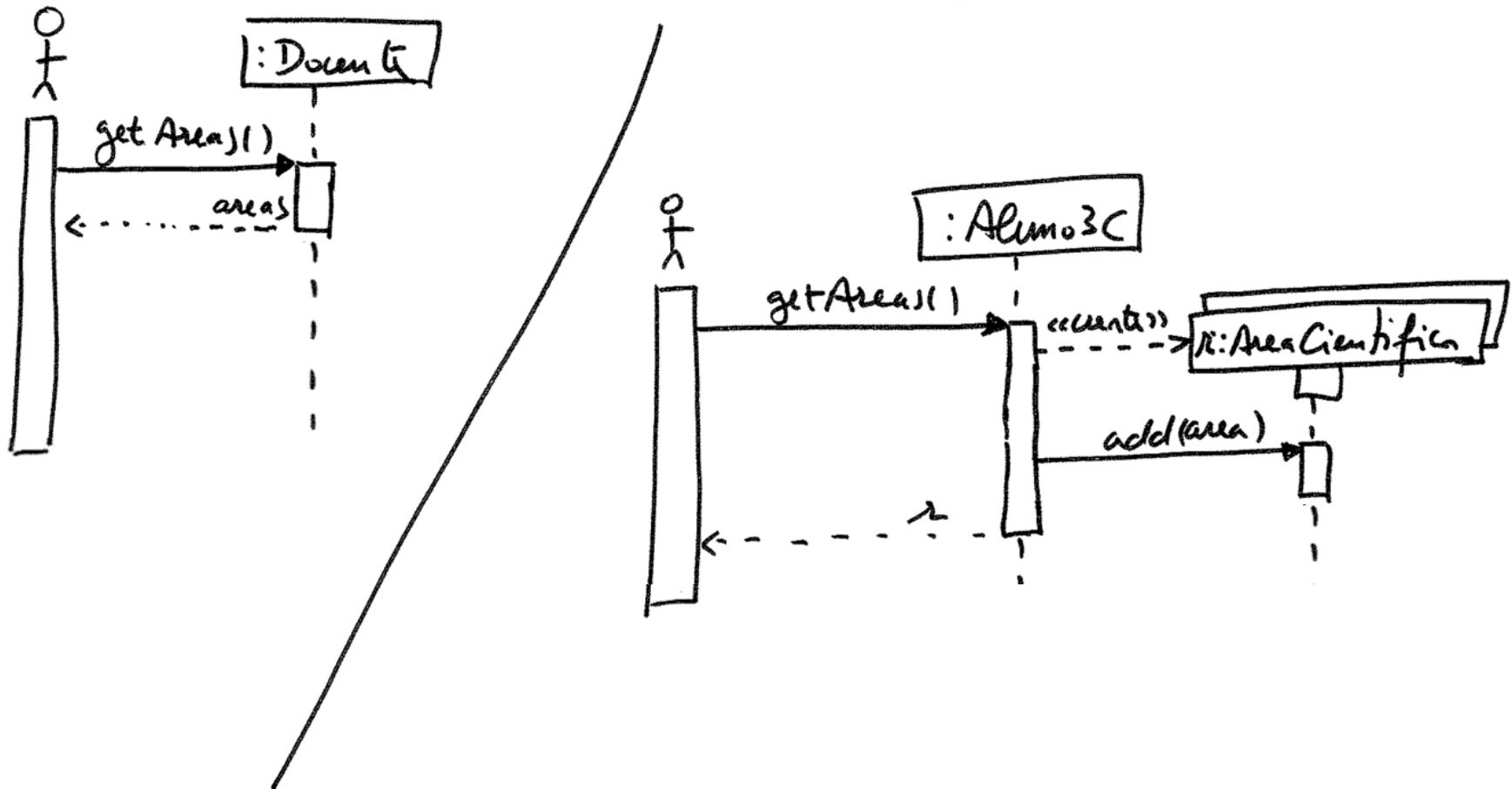
# O comportamento...

(4)



# O comportamento...

getAreas() depende  
da classe concreta...



# O código...



```

public class Univ {
    private Map<String,UC> ucs = new HashMap<>();
    // ...

    public void setProf(String cuc, String ct, Professor f) throws ProfessorInvalido {
        UC uc = ucs.get(cuc);
        uc.setProf(ct, f);
    }
}

public class UC {

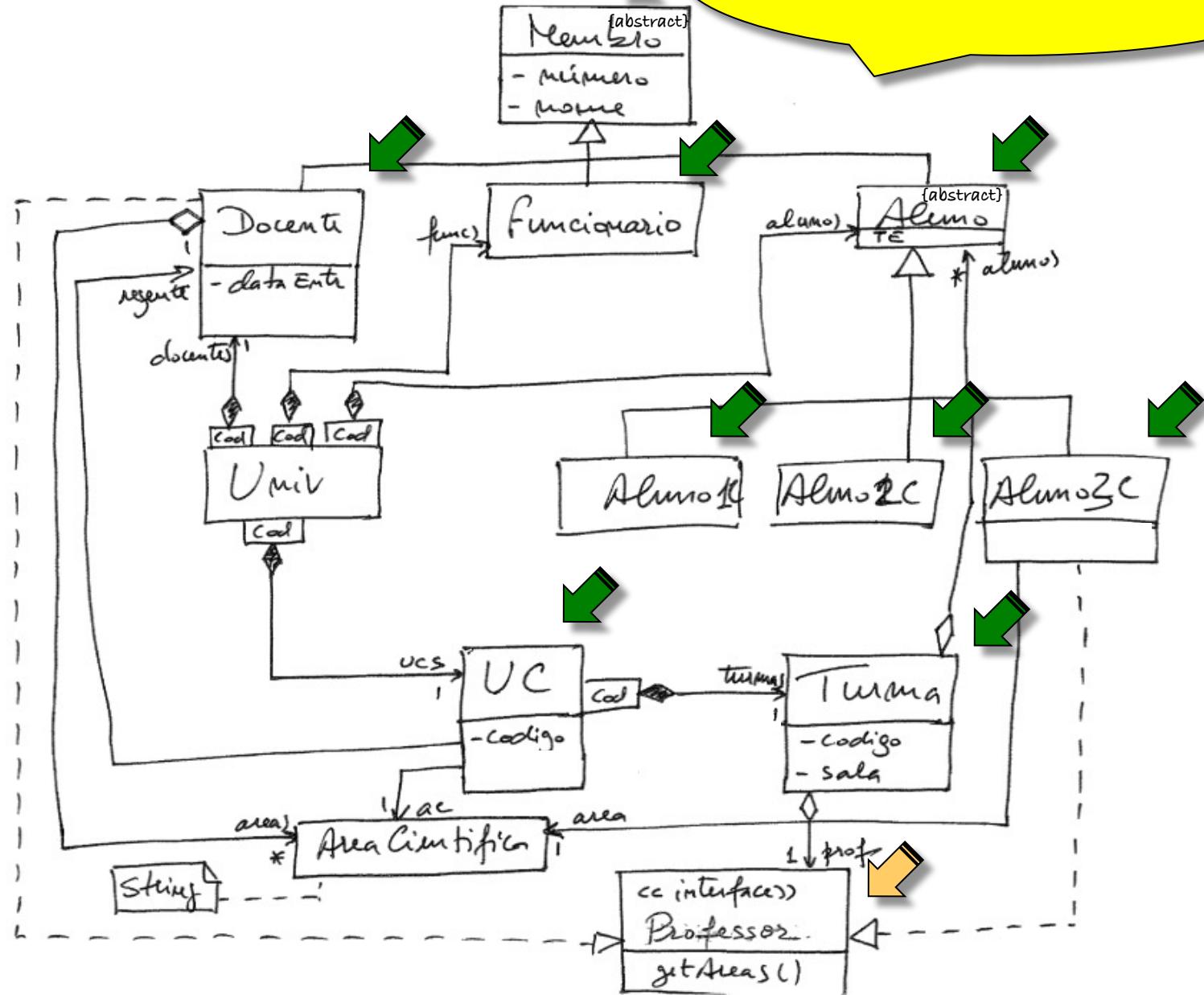
    private String codigo;
    private String ac;
    private Map<String,Turma> turmas = new HashMap<>();

    public void setProf(String ct, Professor f) throws ProfessorInvalido {
        Collection<String> a = f.getAreas();
        if(a.contains(ac)) {
            Turma t = turmas.get(ct);
            t.setProf(f);
        }
        else
            throw new ProfessorInvalido(f, this.codigo);
    }
}

```

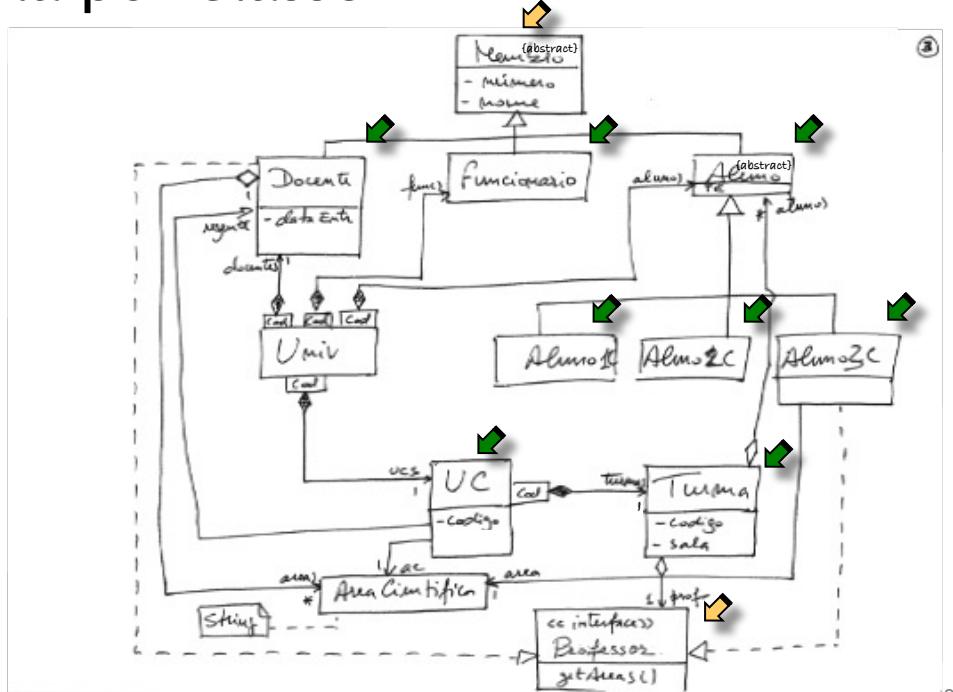
# Que entidades persistir?

Modelo de Domínio ajuda  
a encontrar entidades...



# Mapeamento de classes em tabelas

- Três alternativas
  - Mapeamento de uma tabela por hierarquia
  - Mapeamento de uma tabela por classe concreta
  - Mapeamento de uma tabela por classe



# Mapeamento de classes em tabelas

## Mapeamento de uma tabela por hierarquia (herança/generalização)

- toda a hierarquia de classes representada por uma mesma tabela
- adicionada uma coluna para identificar a classe do objeto representado por cada linha na tabela

chave

Membro(nº, nome, tipo, dataEnt, te, area)

- Problemas:
  - ausência de normalização dos dados
  - proliferação de campos com valores nulos (especialmente para hierarquias de classes com muitas especializações).

nº	nome	tipo	dataEnt	te	area
1234	“José Rosas”	‘D’	01/08/2000	null	null
5678	“António Dias”	‘F’	null	null	null
9876	“Rui Freitas”	‘I’	null	False	“I”

# Mapeamento de classes em tabelas

## Mapeamento de uma tabela por classe concreta

- uma tabela para cada classe concreta
- não é necessário o mecanismo de indicação de tipo adotado na estratégia anterior.

`Docente(nº, nome, dataEnt)`

`Aluno2C(nº, nome, te)`

`Funcionario(nº, nome)`

`Aluno3C(nº, nome, te, area)`

`Aluno1C(nº, nome, te)`

- Problemas:
  - redundância de dados: atributos definidos na superclasse são repetidos em todas as tabelas que representam subclasses da mesma.
  - fazer *cast* de um objeto torna-se um problema: é necessário transferir todos os seus dados de uma tabela para outra

# Mapeamento de classes em tabelas

## Mapeamento de uma tabela por classe

Estratégia mais comum -  
a que vamos adoptar!

- uma tabela para cada classe da hierarquia - estrutura final das tabelas mais próxima da hierarquia de classes

`Membro(nº, nome, tipo)`

`Aluno(nº, te)`

`Aluno3C(nº, area)`

`Docente(nº, dataEnt)`

`Aluno1C(nº)`

`Funcionario(nº)`

`Aluno2C(nº)`

- utilização de chaves estrangeiras para relacionar tabelas

chave estrangeira

- colocação de um identificador de tipo na superclasse

- permite identificar o tipo concreto dos objetos sem *joins*

- melhorias na performance

- Problemas:

- implementação potencialmente mais complexa

- alguma penalização no desempenho



# Que tabelas?

Membro (nº, nome, tipo)

Docente(nº, dataEntr)

Funcionario(nº)

Aluno(nº, te)

Aluno1C(nº)      Como representar  
as associações

Aluno2C(nº)

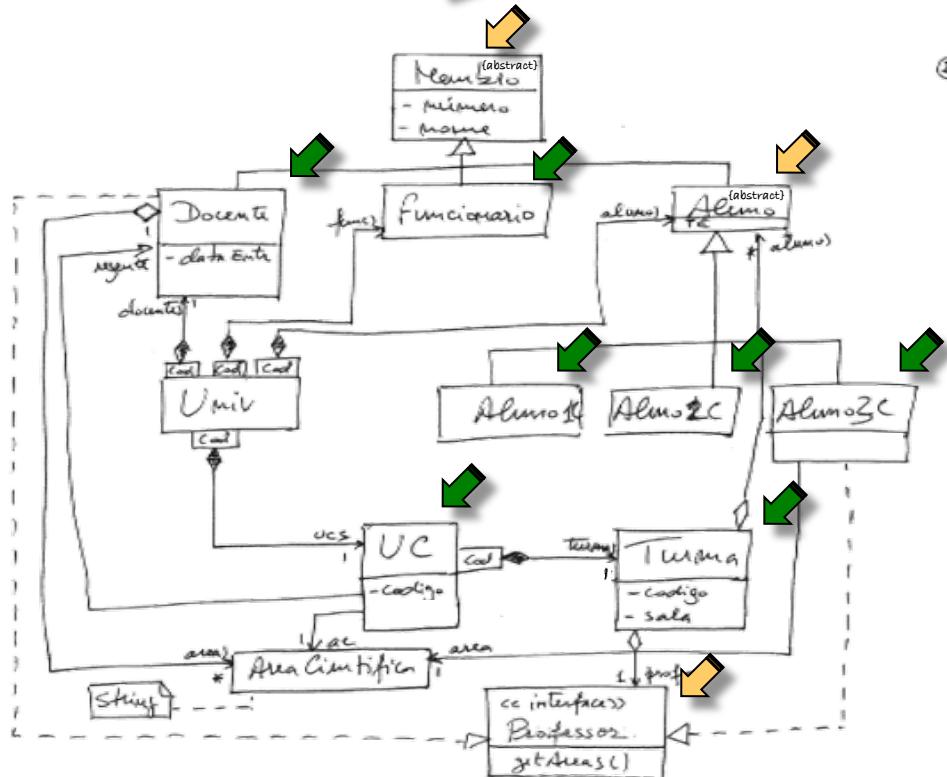
Aluno3C(nº)      ?

Turma(codigo, sala)

Professor(obj\_id, tipo, nº)

AreaCientifica(obj\_id, descr)

Atenção:  
Implementação de **um** Use Case. Faltam outros para  
completar modelo/tabelas!



# Mapeamento de relacionamentos

- Associações um-para-um
  - necessitam que uma chave estrangeira seja posta numa das duas tabelas, relacionando o elemento associado na outra tabela.
  - dependendo da navegabilidade da associação, assim será feita a disposição desta chave estrangeira (navegabilidade é sempre da tabela que possui a chave estrangeira para a tabela referenciada).
- Associações um-para-muitos,
  - adota-se a mesma técnica, mas...
  - a chave estrangeira deve ser posta na tabela que contém os objetos múltiplos (para onde se navega)
- Associações muitos-para-muitos
  - cria-se uma tabela intermédia de pares de chaves, identificando os dois lados do relacionamento.



# Que tabelas?

Membro (nº, nome, tipo)

Docente(nº, dataEntr)

Funcionario(nº)

Aluno(nº, te)

Aluno1C(nº)

Aluno2C(nº)

Aluno3C(nº, codac)

UC(codigo, codac, regente)

Turma(codigo, uc, sala, prof)

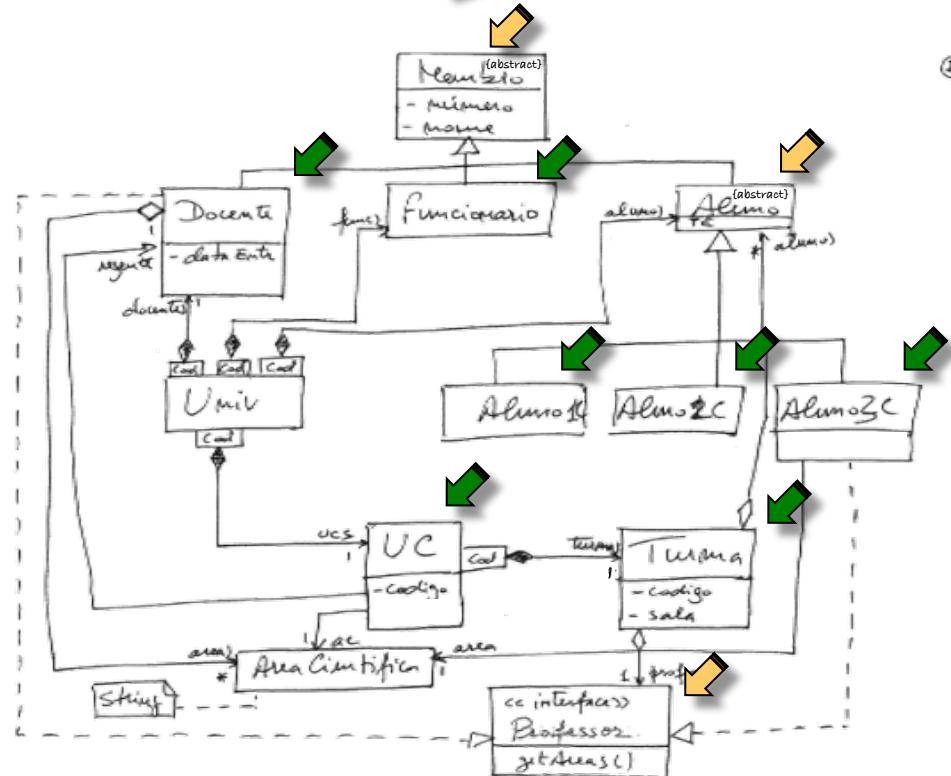
Professor(obj id, tipo, nº)

AreaCientifica(obj id, descr)

DocenteAreas(nº, codac)

TurmaAlunos(codigo, nº)

Atenção:  
Implementação de **um** Use Case. Faltam outros para completar modelo/tabelas!



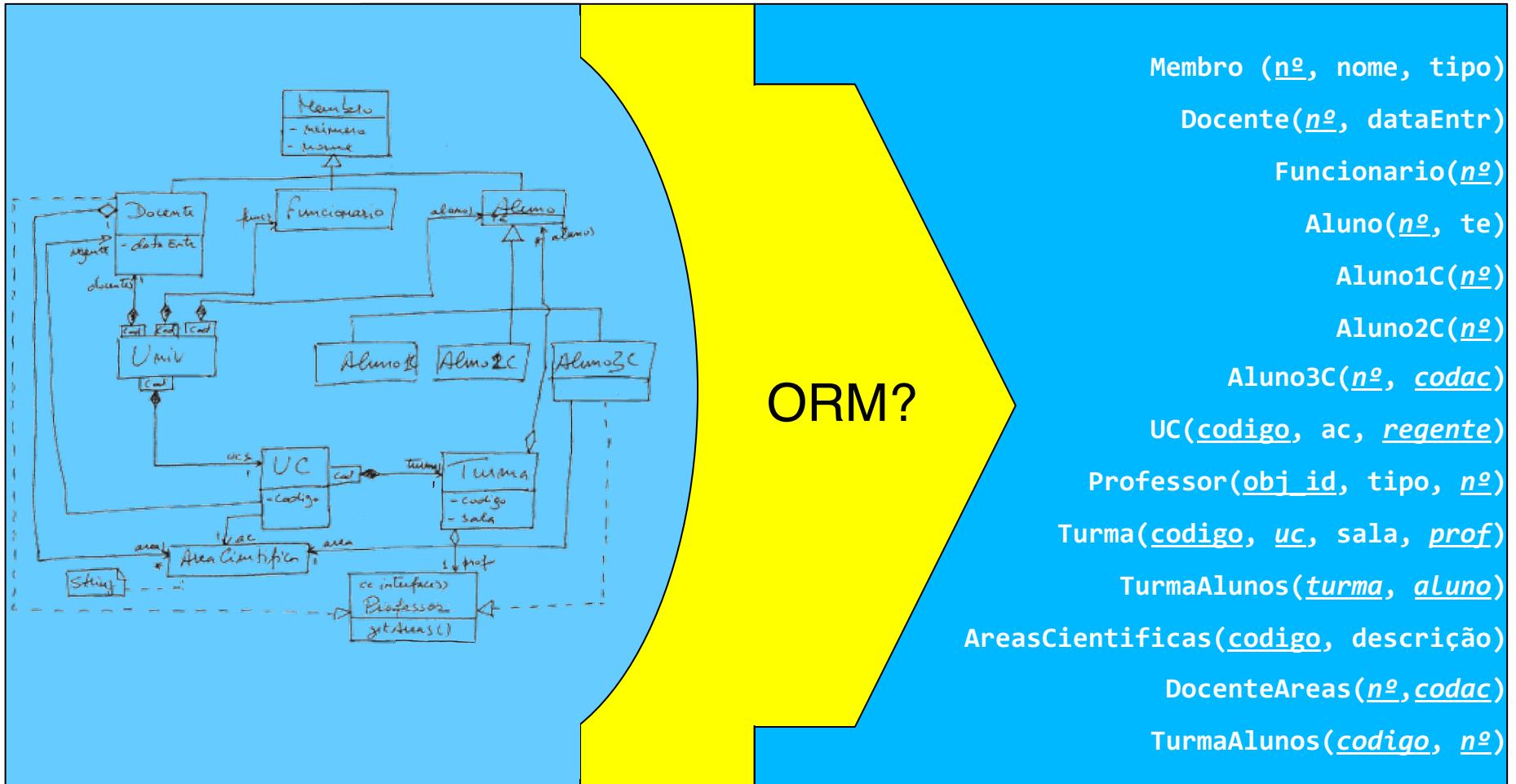
# Regras de mapeamento

1. As tabelas resultam exclusivamente das classes/interfaces do modelo e das associações de “muitos para muitos”
  - Nem todas as classes darão origem a tabelas
2. Todas as tabelas terão uma chave primária
  - Poderá ter que ser criado um identificador único (obj\_id)
3. Mapeamento de Associações “um para um”: a tabela origem inclui como chave estrangeira a chave primária da tabela destino
4. Mapeamento de Associações “um para *n*”: a tabela do lado *n* inclui como chave estrangeira a chave primária da tabela do lado um.
5. Mapeamento de Associações de “muitos para muitos”: dá origem a uma tabela onde a chave primária é composta pelas chaves primárias das tabelas associadas.

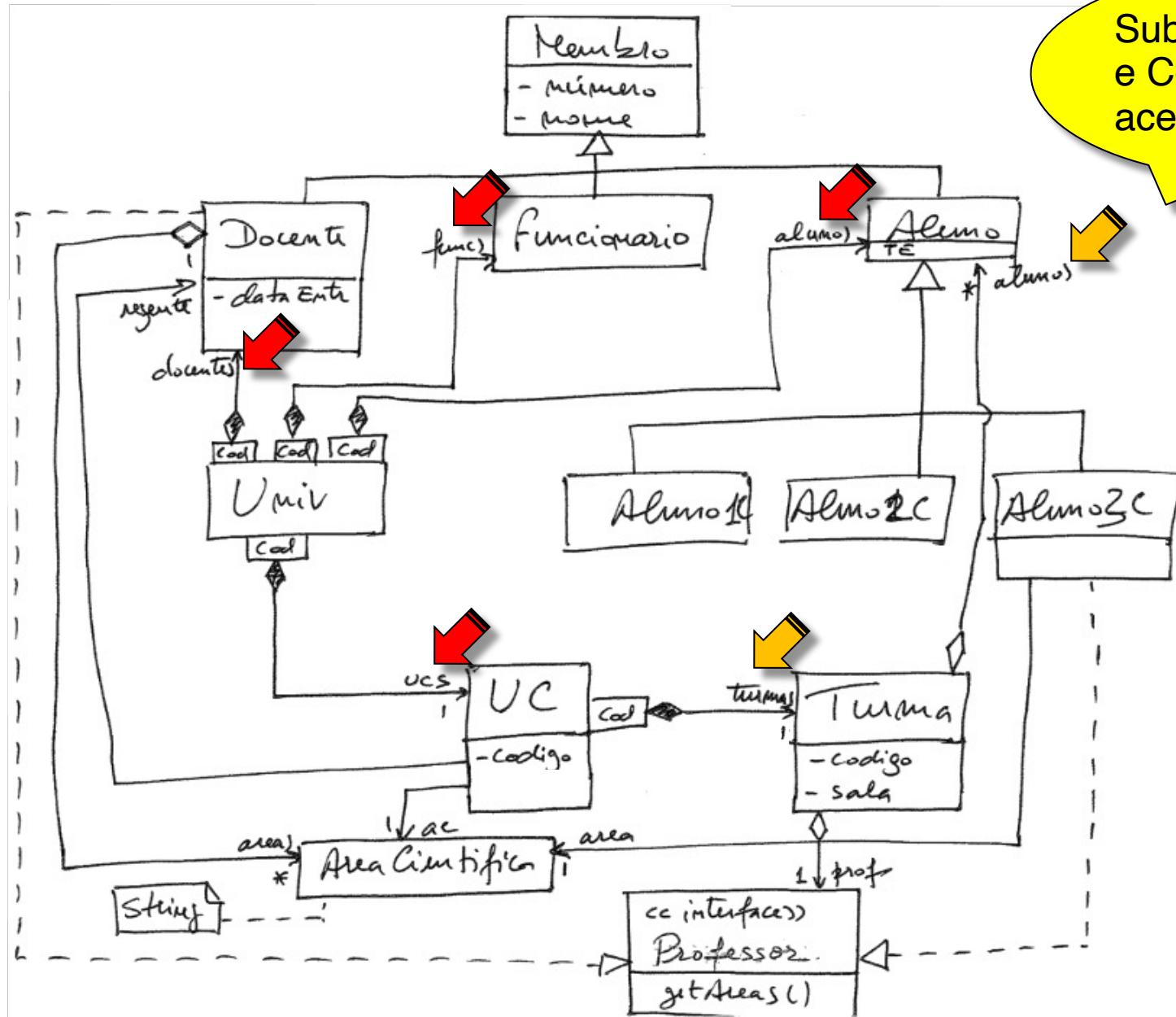
# Regras de mapeamento

6. Mapeamento de Agregações: ver associações
7. Mapeamento de Composições: tabela da classe composta recebe chave primária da tabela da classe que compõe (fica com chave composta se já tiver a sua propria chave)
8. Mapeamento de Generalizações: uma tabela por classe
  - a) As subclasses não têm identidade própria:
    - tabelas que mapeiam os subclasses utilizam chave primária da superclasse
  - b) As subclasses têm identidade própria:
    - tabelas que mapeiam as subclasses têm chave primária própria.
    - chave primária da tabela que implementa a superclasse incluída nas tabelas que implementam as subclasses como chave estrangeira
9. Interfaces
  - Tabelas que mapeiam as interfaces têm identificador de tipo e incluem como chave estrangeira as chaves primárias dos objectos concretos que implementam a interface

# Object Relational Mapping - ORM



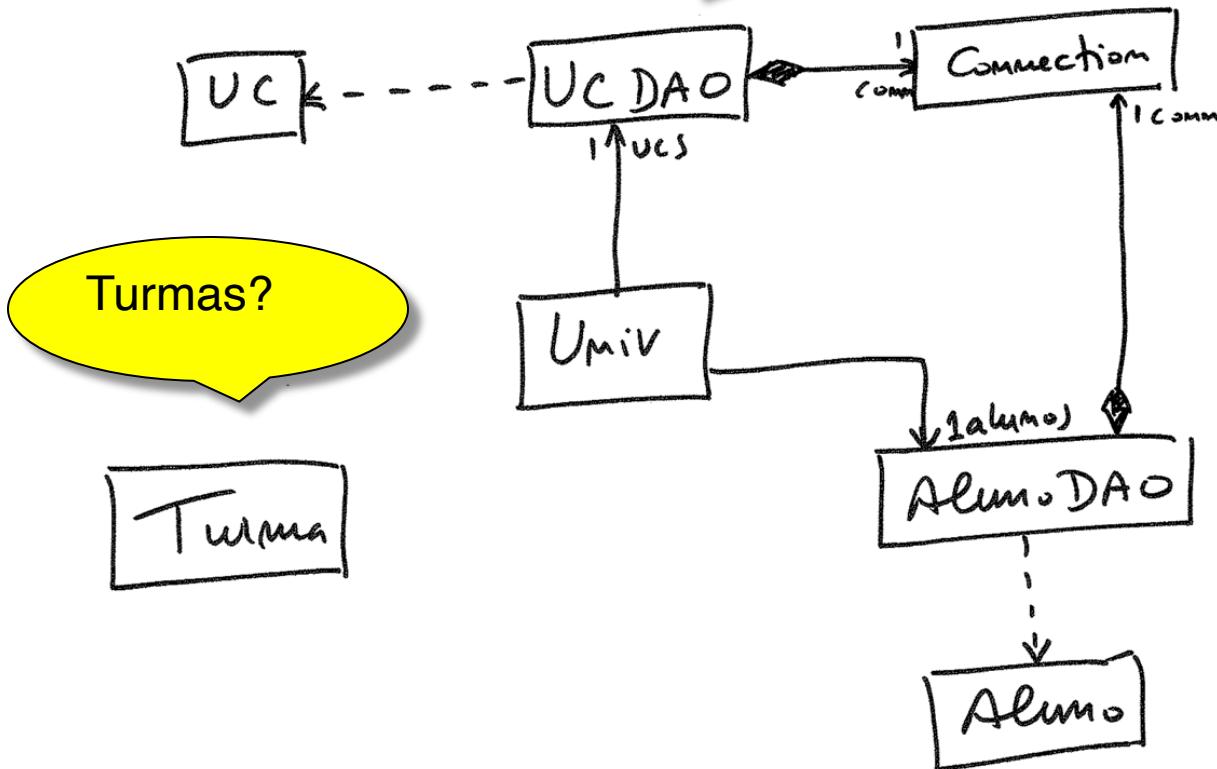
# Arquitectura - versão sem persistência



# Arquitectura - versão com persistência

⑨

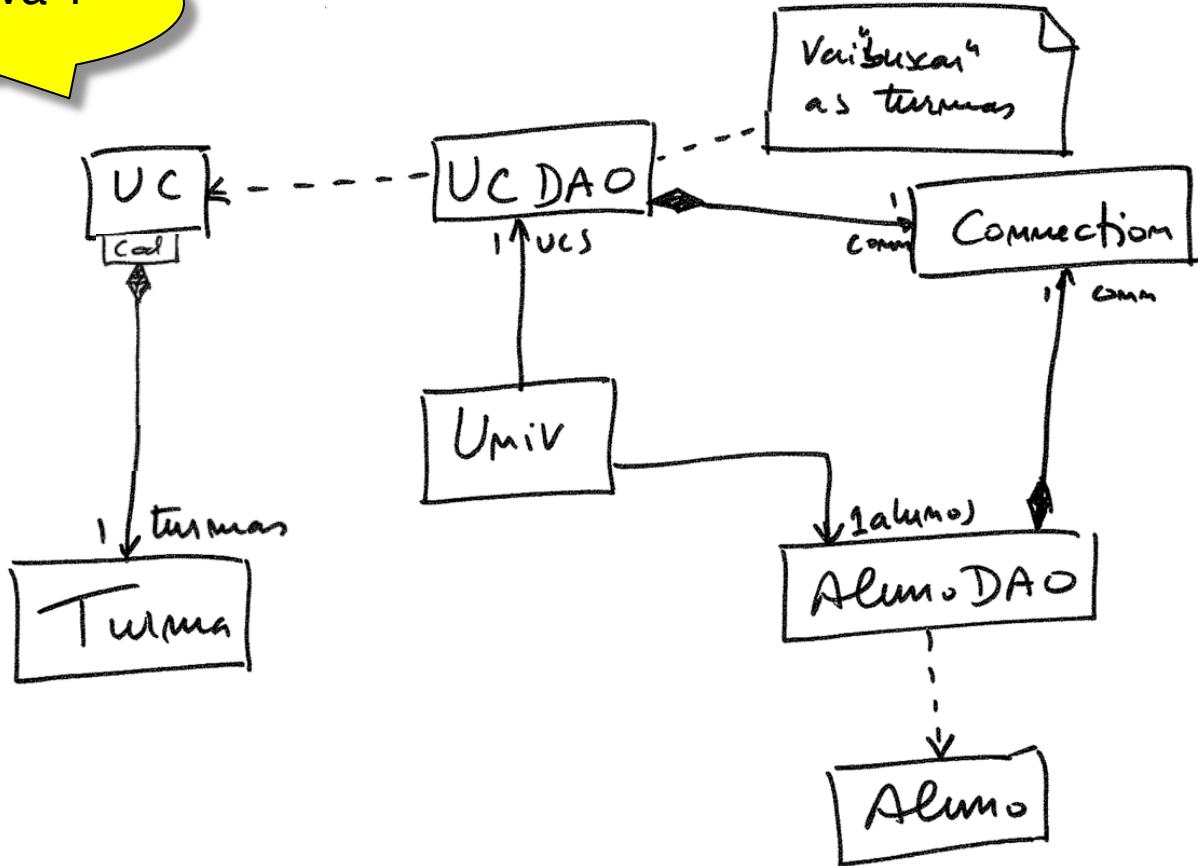
DAO = Data Access Object



# Arquitectura - versão com persistência

10

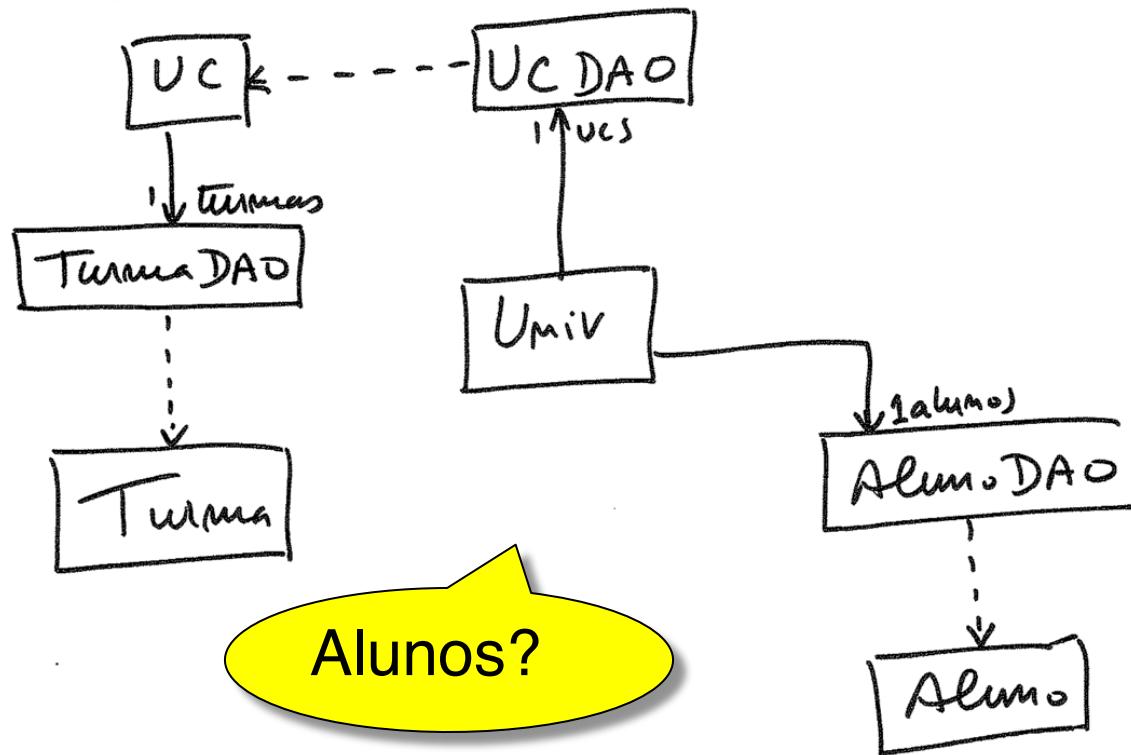
Alternativa 1



# Arquitectura - versão com persistência

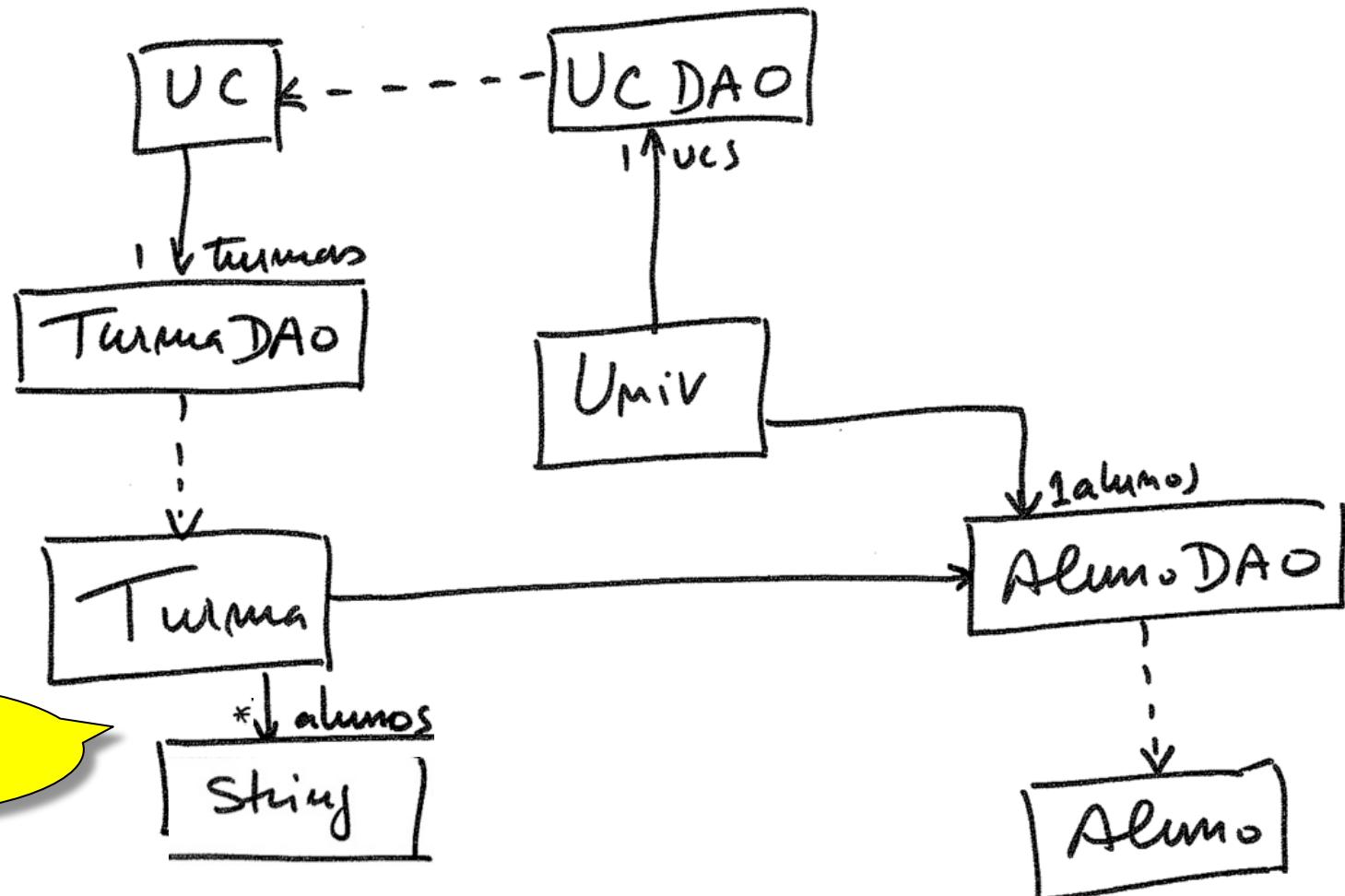
11

Alternativa 2



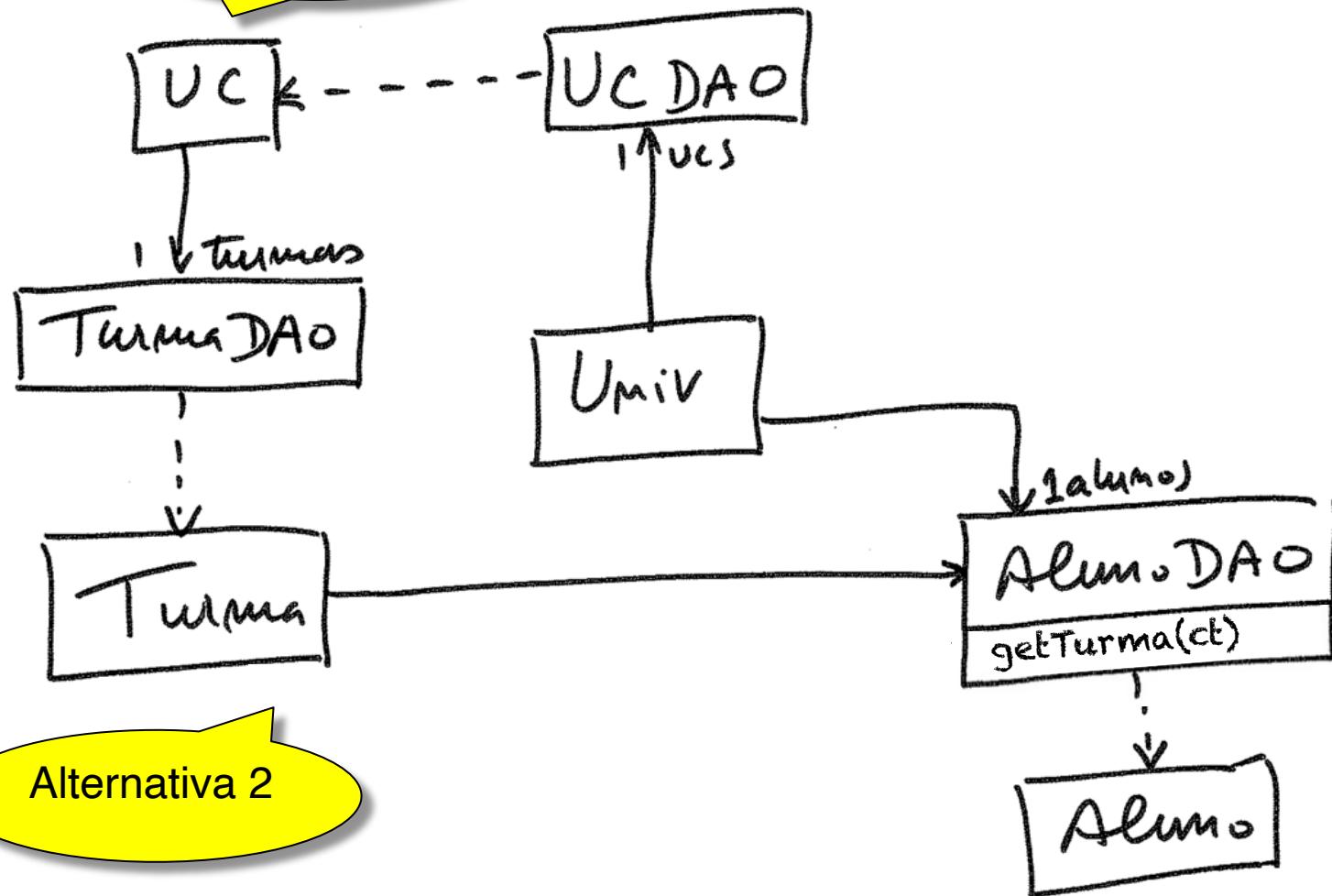
Alunos?

# Arquitectura - versão com persistência



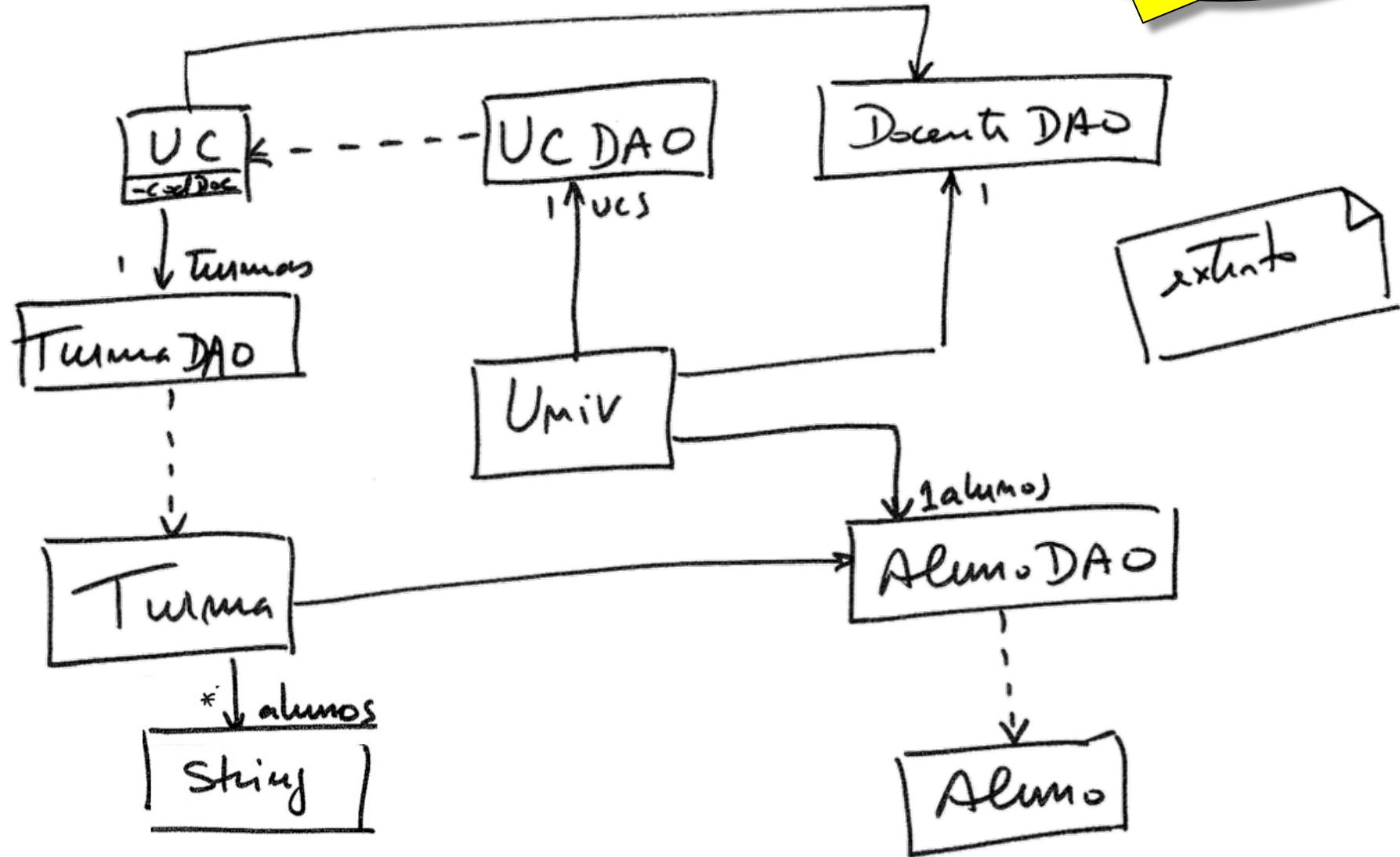
# Arquitectura - versão com persistência

Docente?



# Arquitectura - versão com persistência

Falta completar  
com operações,  
etc.

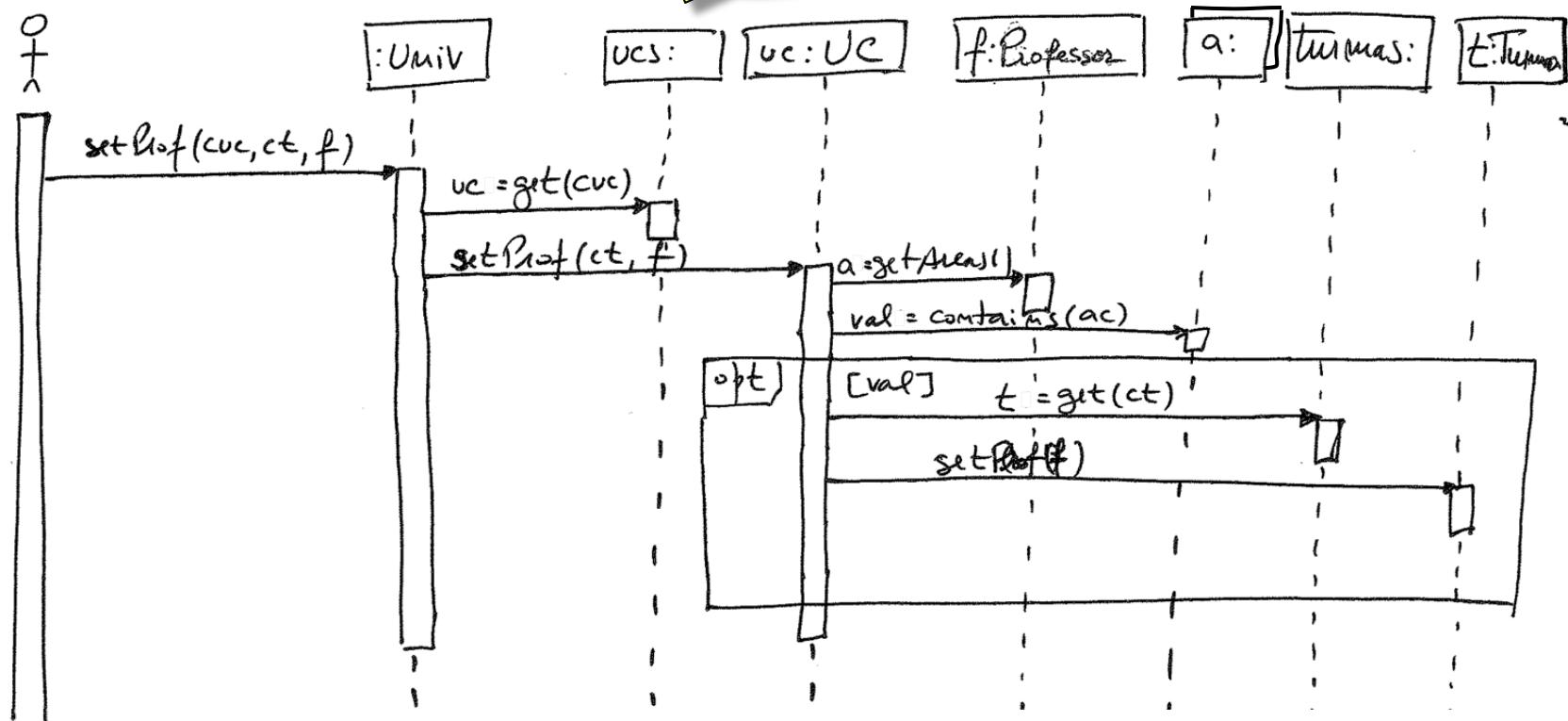




# O comportamento...

O Map...

O Map...

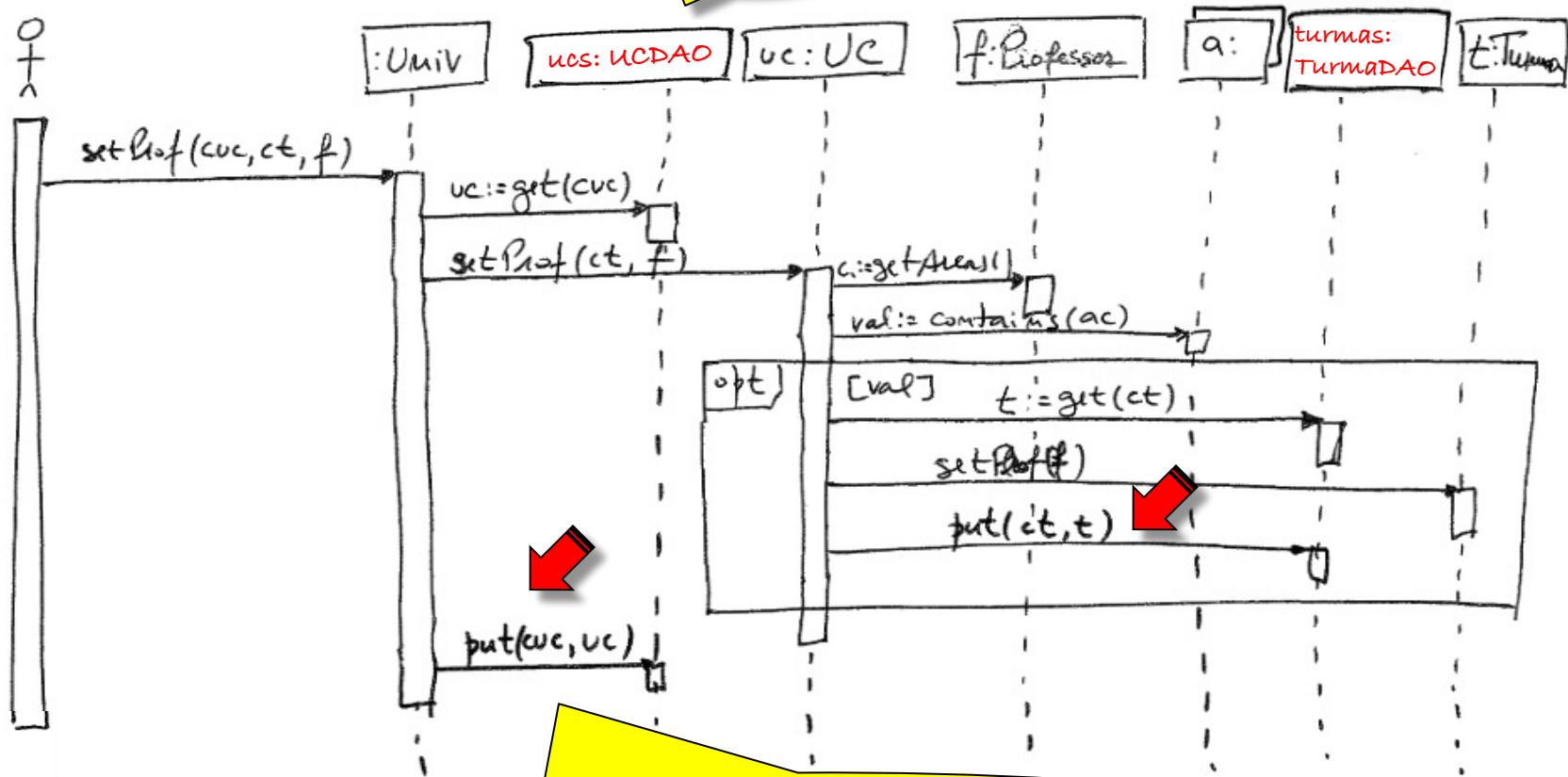


# O comportamento...

Agora é  
um DAO.

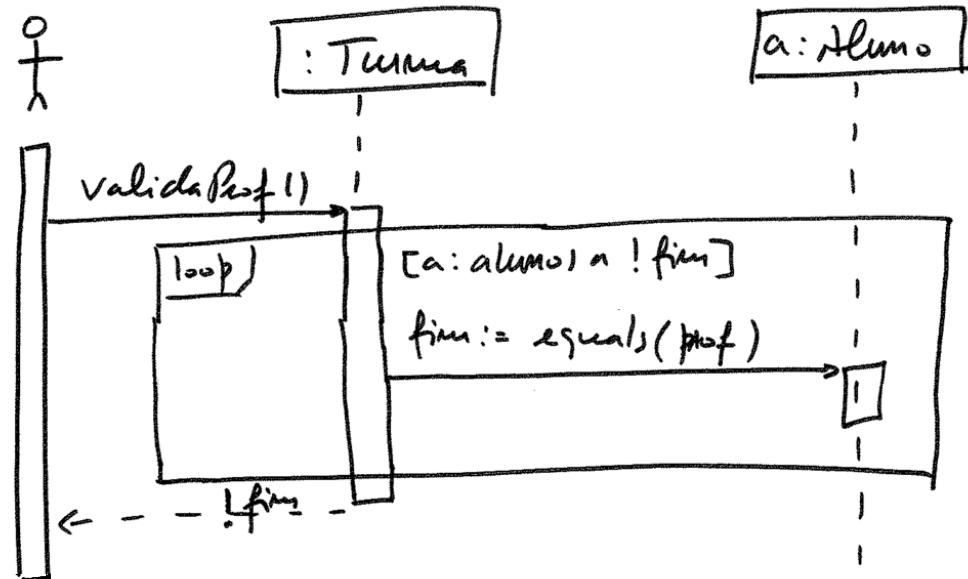
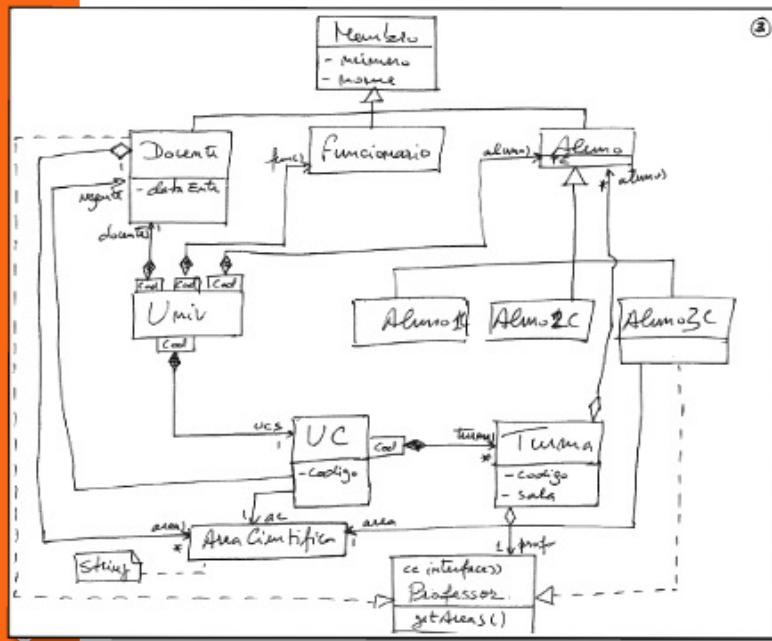
# Agora é um DAO.

19

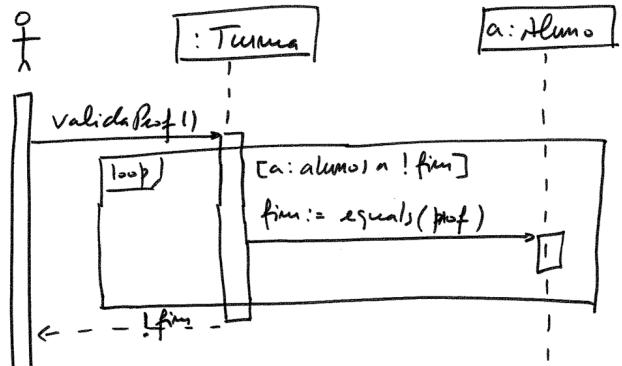
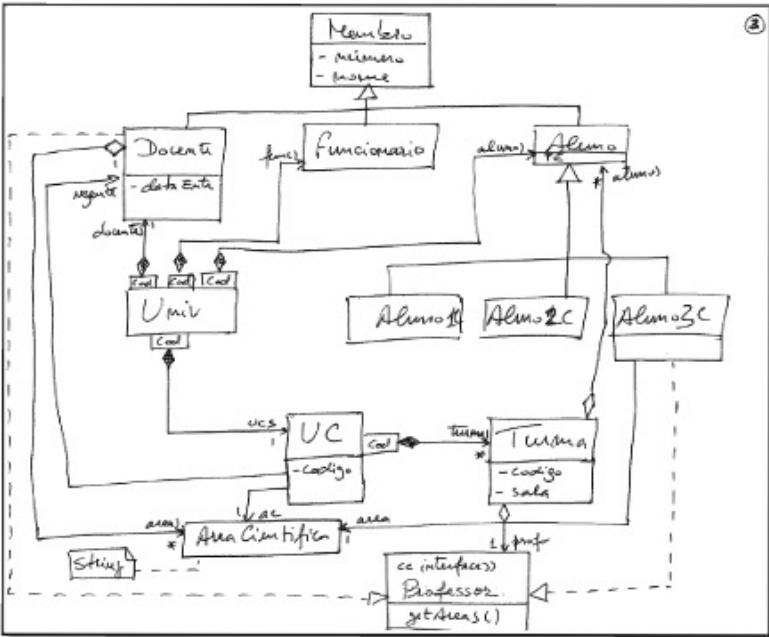


Necessário, apenas, garantir actualização das entidades. Lógica da solução mantém-se!

# Outro Exemplo...



# Código...



```

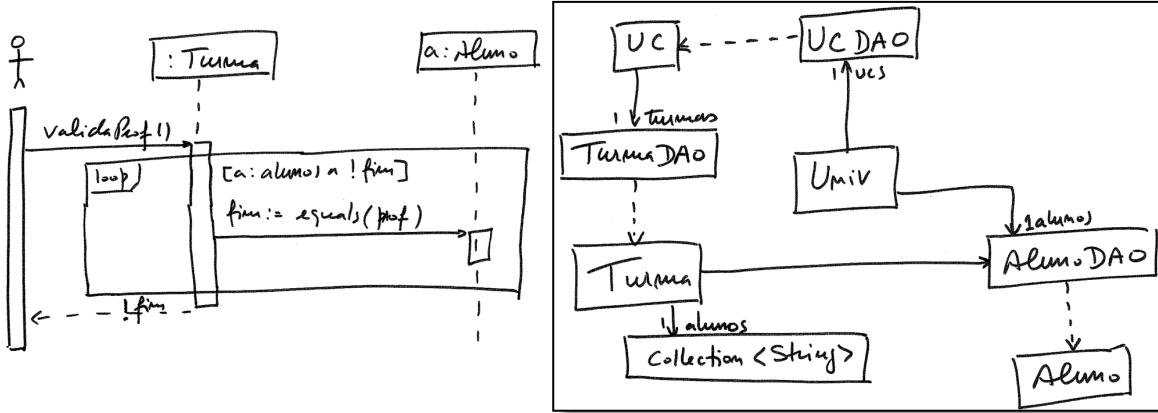
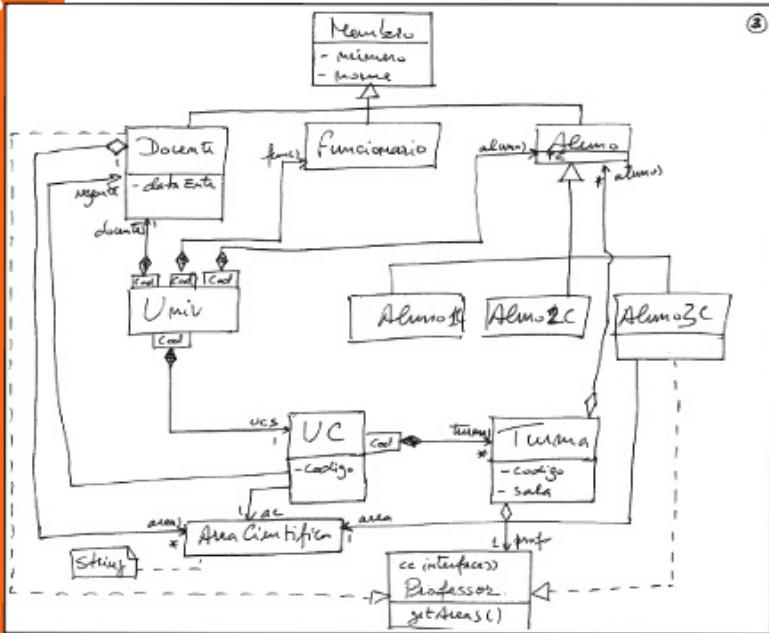
class Turma {
    private String codigo, sala;
    private Professor prof;
    private Collection<Aluno> alunos;

    void setProf(Professor f) {
        this.prof = f;
    }

    public boolean validaProf() {
        boolean fim = false;
        Iterator<Aluno> alunosIt = alunos.iterator();

        while (!fim && alunosIt.hasNext()) {
            Aluno a = alunosIt.next();
            fim = a.equals(prof);
        }
        return !fim;
    }
}

```



```

class Turma {
    private String codigo, sala;
    private Professor prof;
    private Collection<Aluno> alunos;

    void setProf(Professor f) {
        this.prof = f;
    }

    public boolean validaProf() {
        boolean fim = false;
        Iterator<Aluno> alunosIt = alunos.iterator();

        while (!fim && alunosIt.hasNext()) {
            Aluno a = alunosIt.next();
            fim = a.equals(prof);
        }
        return !fim;
    }
}

```

```

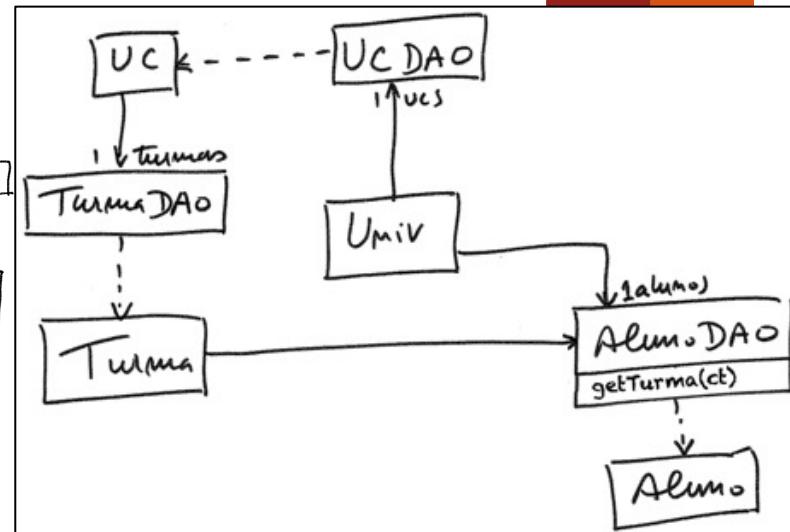
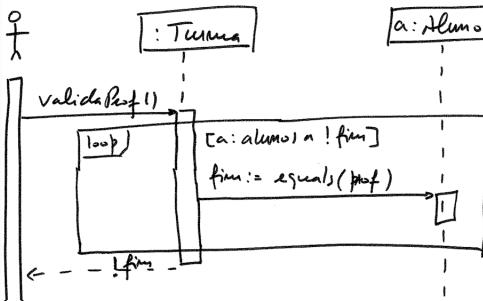
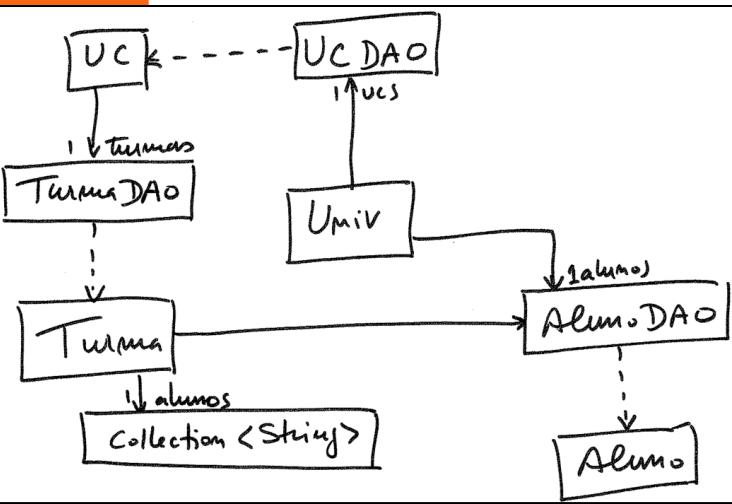
class TurmaPersist {
    private String codigo, sala;
    private Professor prof;
    private Collection<String> alunos;
    private AlunoDAO alDAO;

    void setProf(Professor f) {
        this.prof = f;
    }

    public boolean validaProf() {
        boolean fim = false;
        Iterator<String> alunosIt = alunos.iterator();

        while (!fim && alunosIt.hasNext()) {
            Aluno a = alDAO.get(alunosIt.next());
            fim = a.equals(prof);
        }
        return !fim;
    }
}

```



```

class TurmaPersist {
    private String codigo, sala;
    private Professor prof;
    private Collection<String> alunos;
    private AlunoDAO alDAO;

    void setProf(Professor f) {
        this.prof = f;
    }

    public boolean validaProf() {
        boolean fim = false;
        Iterator<String> alunosIt = alunos.iterator();

        while (!fim && alunosIt.hasNext()) {
            Aluno a = alDAO.get(alunosIt.next());
            fim = a.equals(prof);
        }
        return !fim;
    }
}
  
```

```

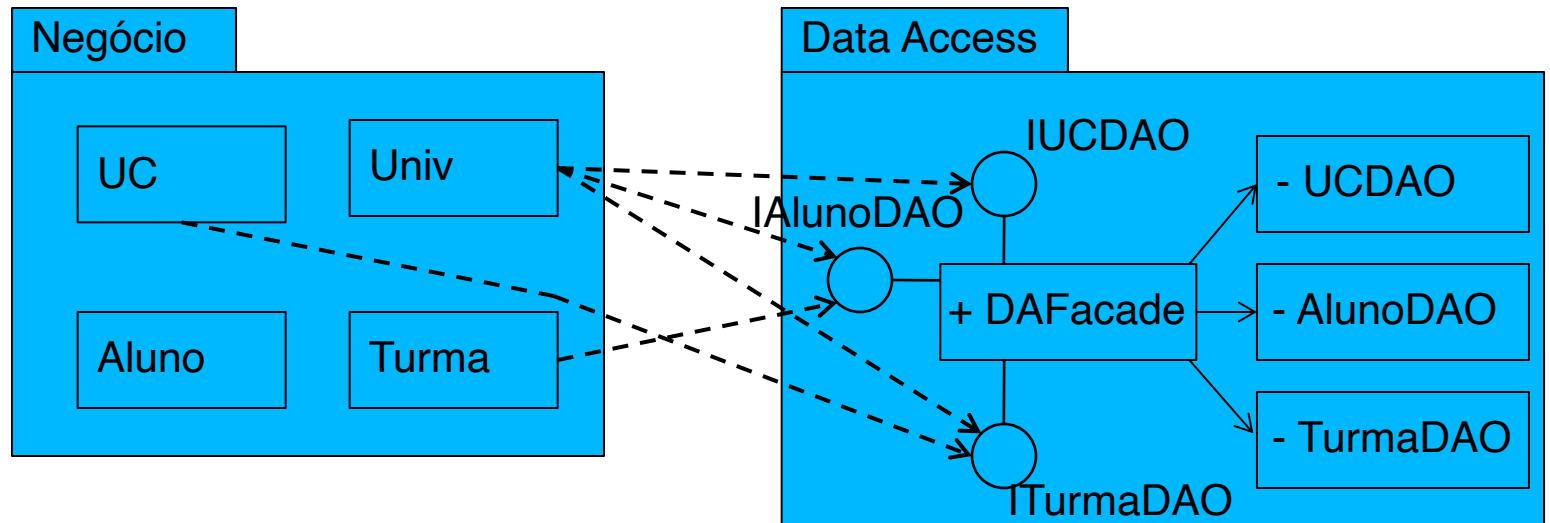
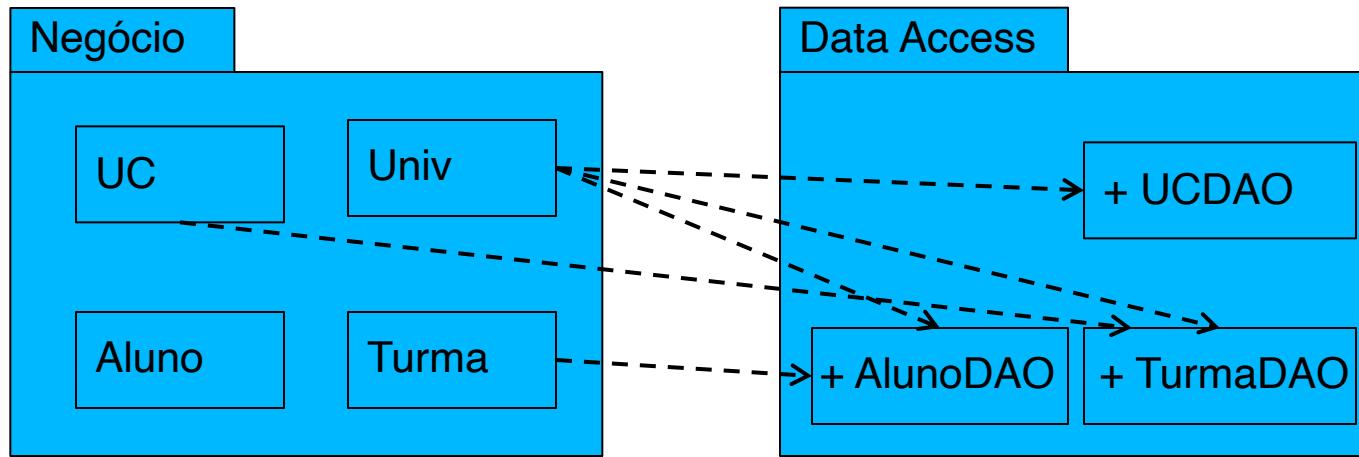
class TurmaPersist2 {
    private String codigo, sala;
    private Professor prof;
    private AlunoDAO alDAO;

    void setProf(Professor f) {
        this.prof = f;
    }

    public boolean validaProf() {
        boolean fim = false;
        Collection<Aluno> alunos = alDAO.getTurma(codigo);
        Iterator<Aluno> alunosIt = alunos.iterator();

        while (!fim & alunosIt.hasNext()) {
            Aluno a = alunosIt.next();
            fim = a.equals(prof);
        }
        return !fim;
    }
}
  
```

# Packages



# Impacto na lógica de negócio?

- Classes que representem entidades a persistir mapeadas na base de dados
- Estratégia *simples*:
  - focar processos nas associações, em especial
    - associações qualificadas - maps
    - associações um para muitos - coleções
  - aplicar regras dos slides anteriores
- Impacto nos métodos que acedem às associações
  - substituídos por acessos à camada de dados
  - reconstroem os objectos a partir da camada de dados (por simplicidade, com métodos com os mesmos nomes dos métodos das estruturas de dados Java)
- Problema:
  - não funcionam por referência
  - onde parar a reconstrução? - perigo de trazer toda a Base de Dados
  - caching? - evitar ter muitos objectos em memória