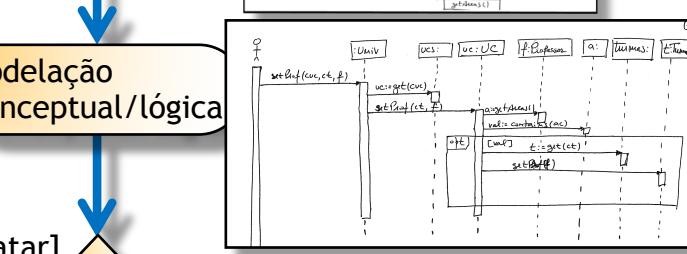
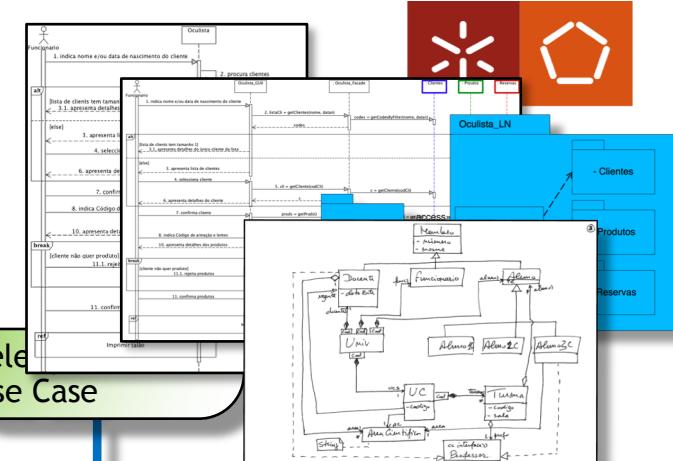
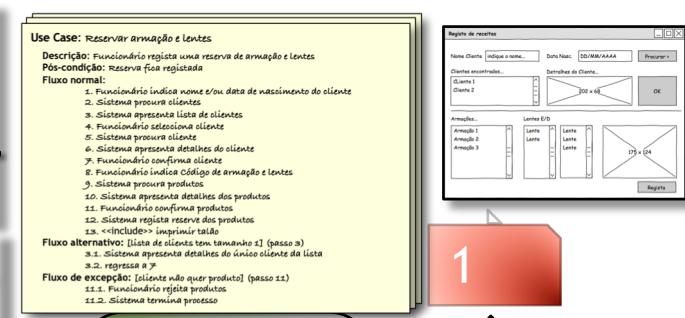
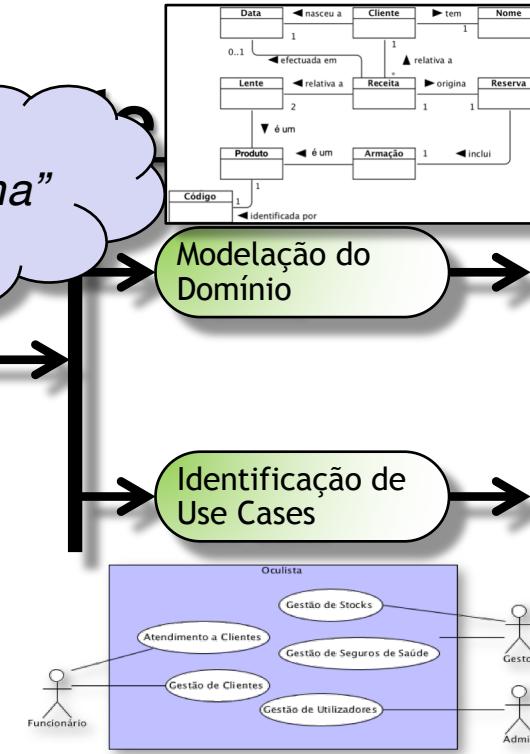
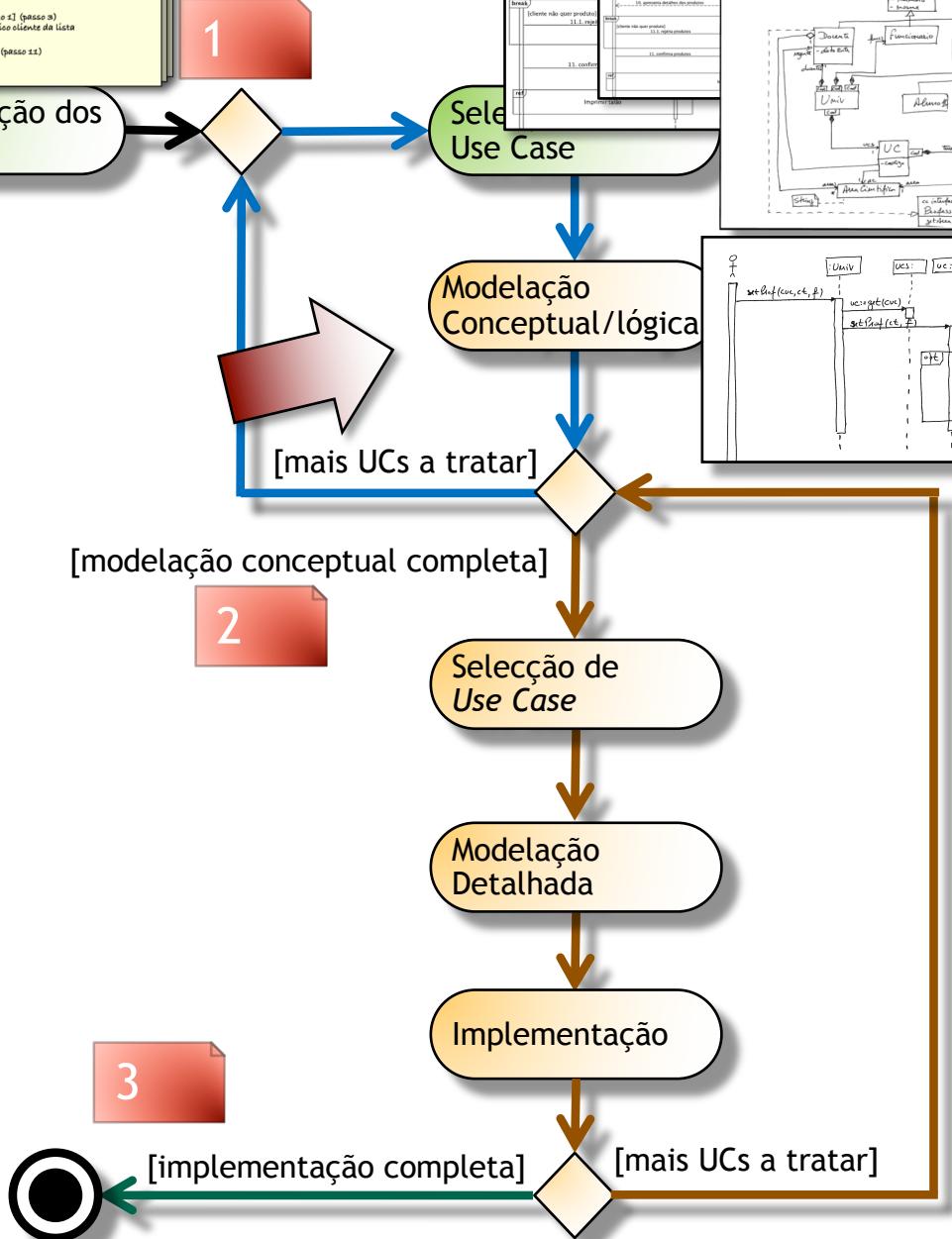


“Problema”

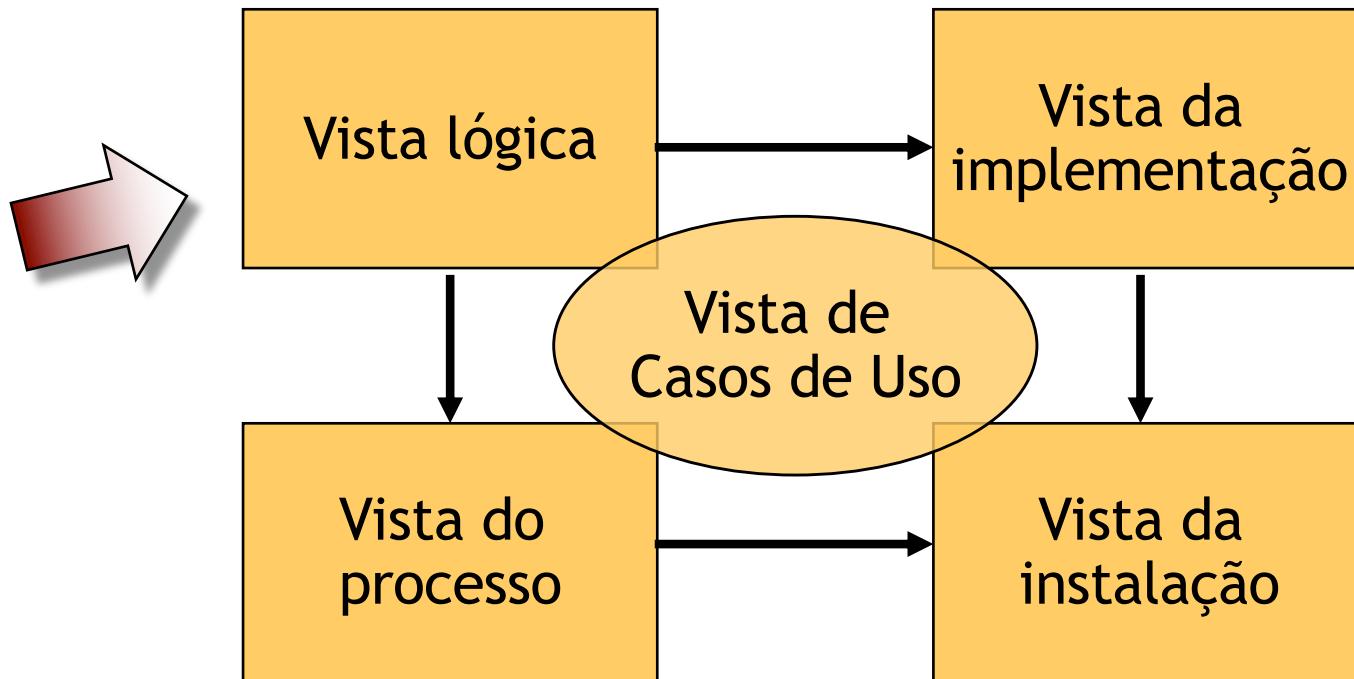


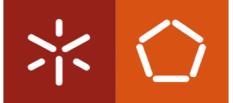
- Inception
- Elaboration
- Construction
- Transition

Fases do trabalho



Onde estamos...





Desenvolvimento de Sistemas Software

Aula Teórica 17/18

ORM (Object-Relational Mapping)

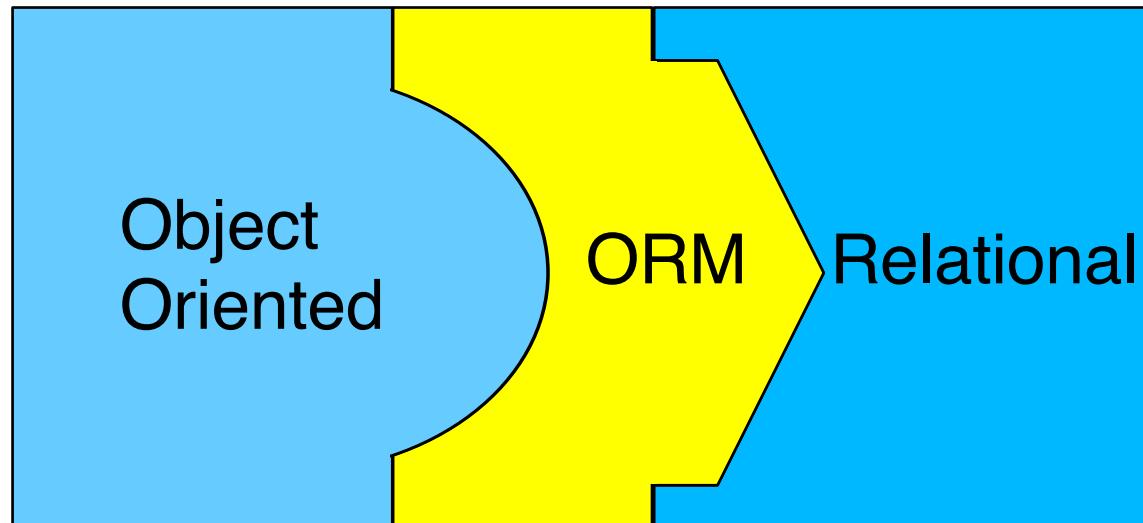
Lógica de negócio na Base de dados?

- 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
 - (mas muito disto pode ser implementado numa camada de acesso... ;)

Paradigma OO vs Relacional

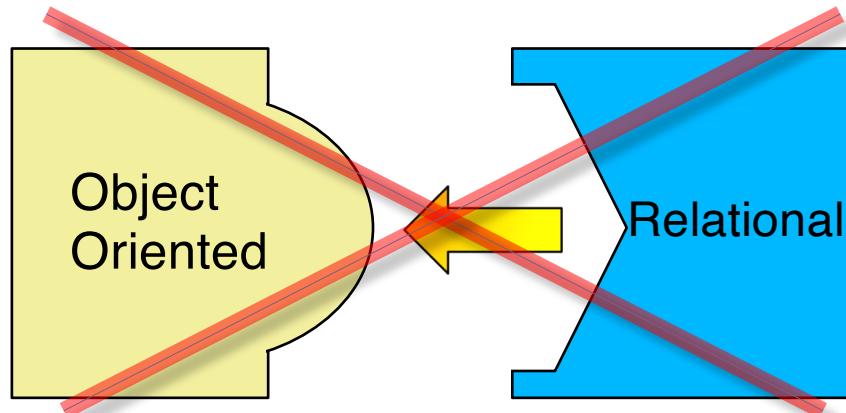
- Paradigma OO prevalente na programação da lógica de negócios
- Paradigma relacional prevalente nas bases de dados (persistência)
- Paradigma orientado a objetos e paradigma relacional não são diretamente compatíveis
 - Diagramas de classe não são esquemas de base de dados!
 - Modelo relacional não possui características presentes no modelo OO como poliformismo 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

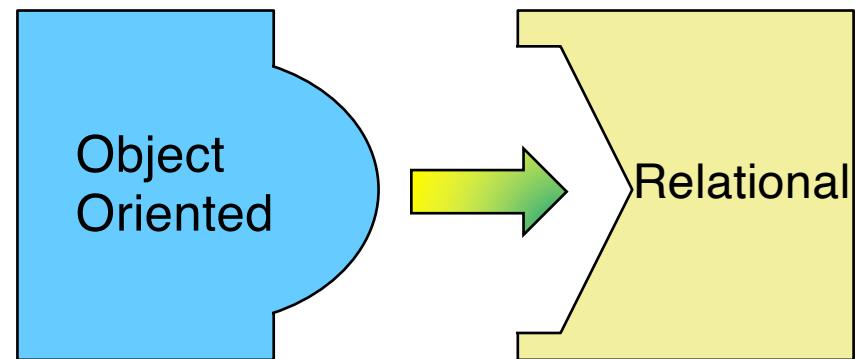


- Camada de Negócio desenvolvida em tecnologias OO
- Desenvolvimento deve ser, tanto quanto possível, independente da camada de persistência
- Persistencia de dados em:
 - Bases de dados relacional
 - ou NoSQL
 - ou orientada a objectos
 - ou...

Abordagens ao ORM

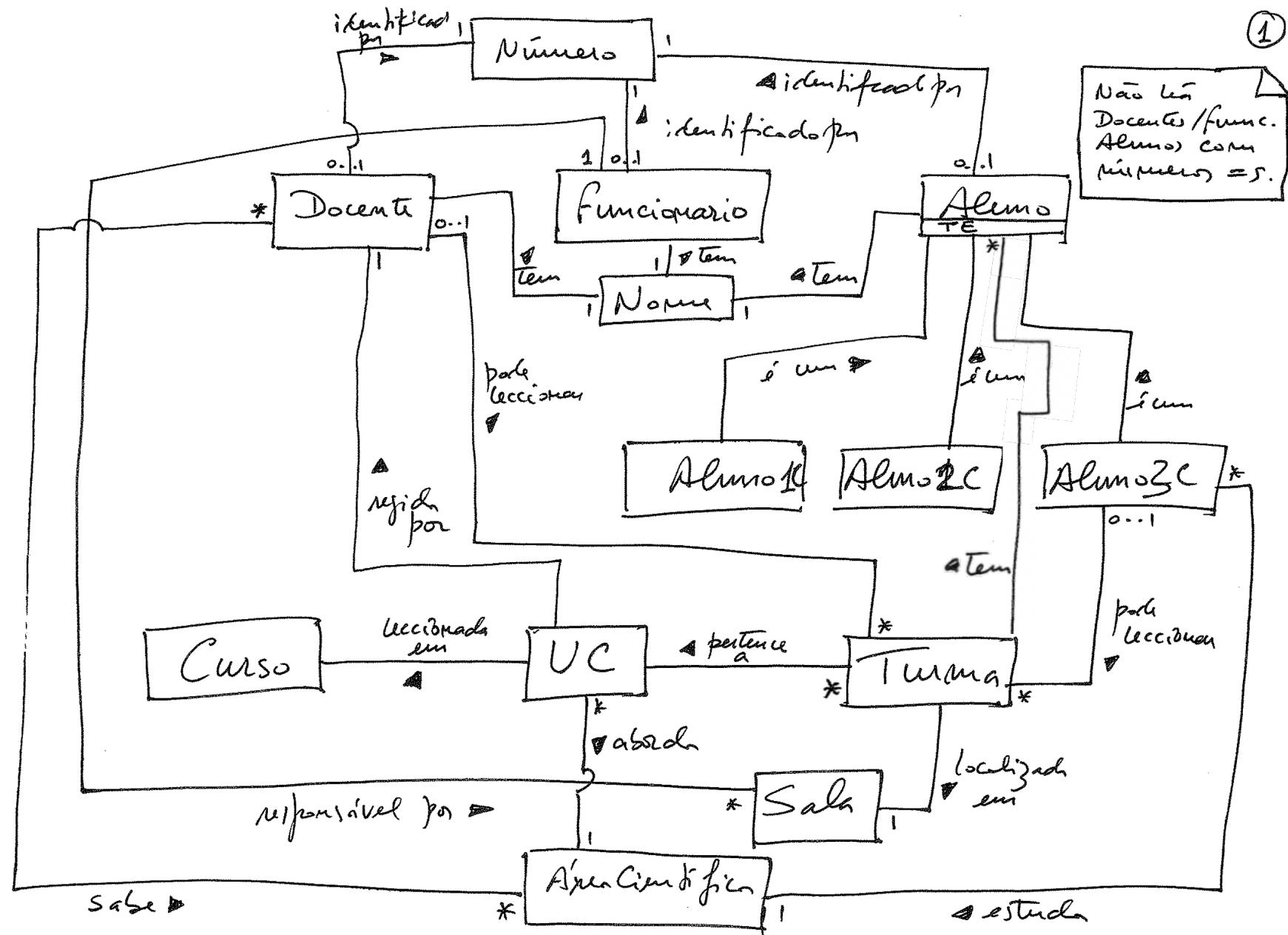


- Derivar objectos a partir das tabelas
- Objectos servem apenas para aceder às tabelas
- Tendência para se perder separação de camadas:
 - Lógica de negócio dependente de modelo relacional
 - Lógica de negócio no modelo relacional
- Vantagens do paradiagrama OO?
 - Não se está a fazer desenvolvimento OO
 - Não se tira partido do paradiagrama OO



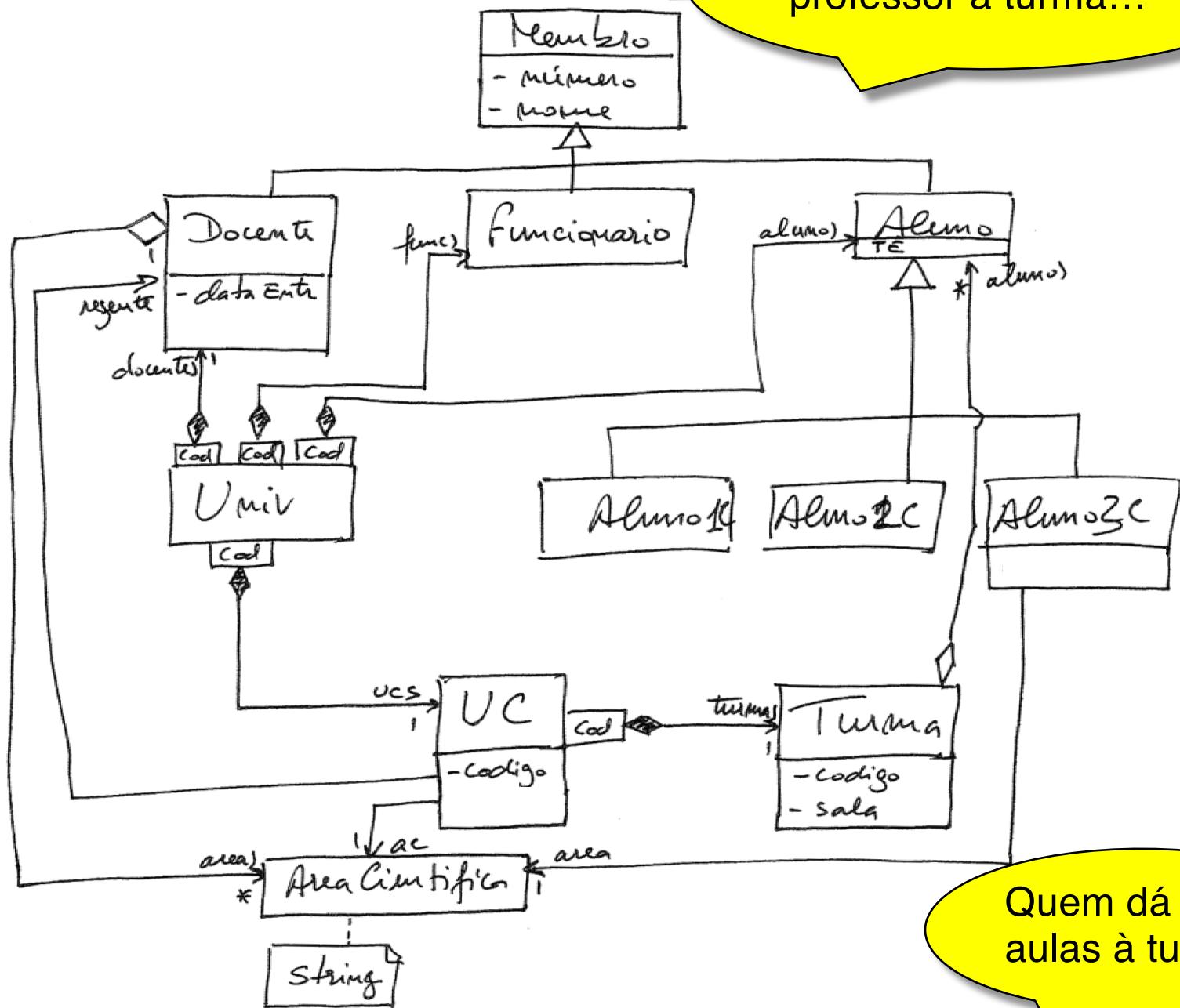
- Derivar tabelas a partir do objectos
- Lógica de negócio desenvolvida independentemente da Base de Dados
- Base de dados utilizada como serviço de persistência de dados
- Possível explorar vantagens do paradiagrama OO
- Resulta melhor quando lógica de negócio é complexa

Exemplo - Um Modelo de Domínio...



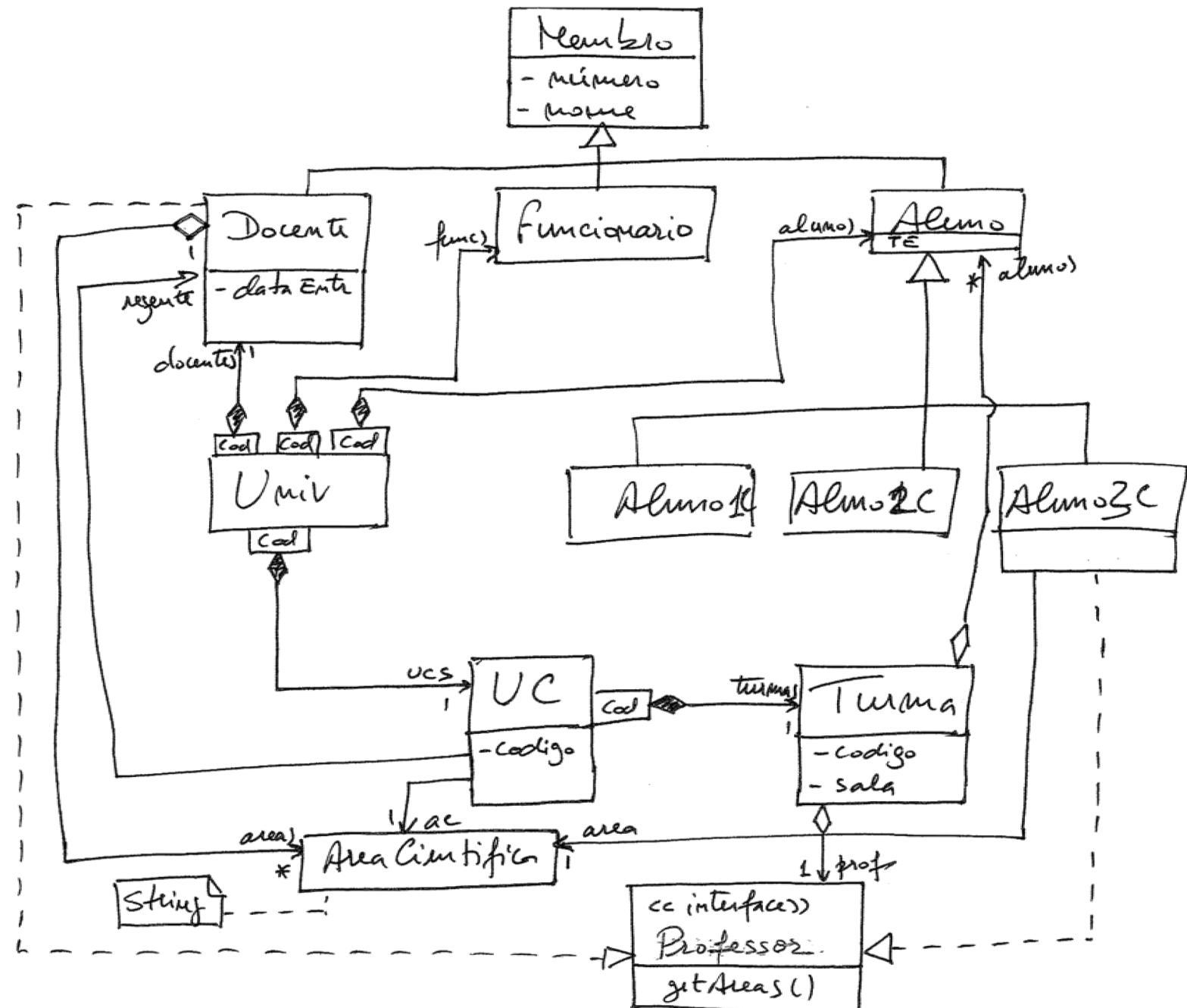
Arquitectura parcial...

Use Case atribuir professor a turma...



Quem dá aulas à turma?

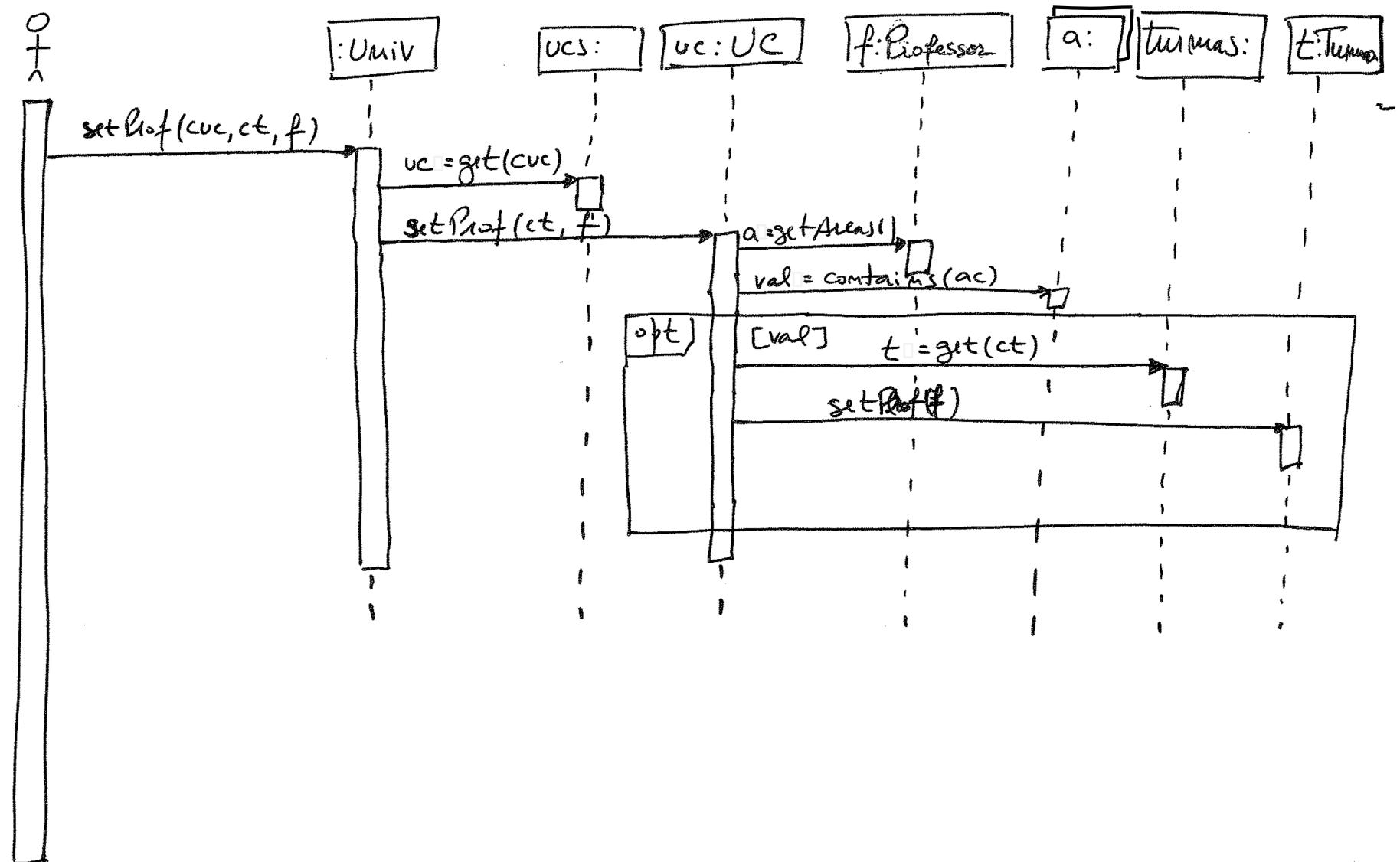
Arquitectura parcial...





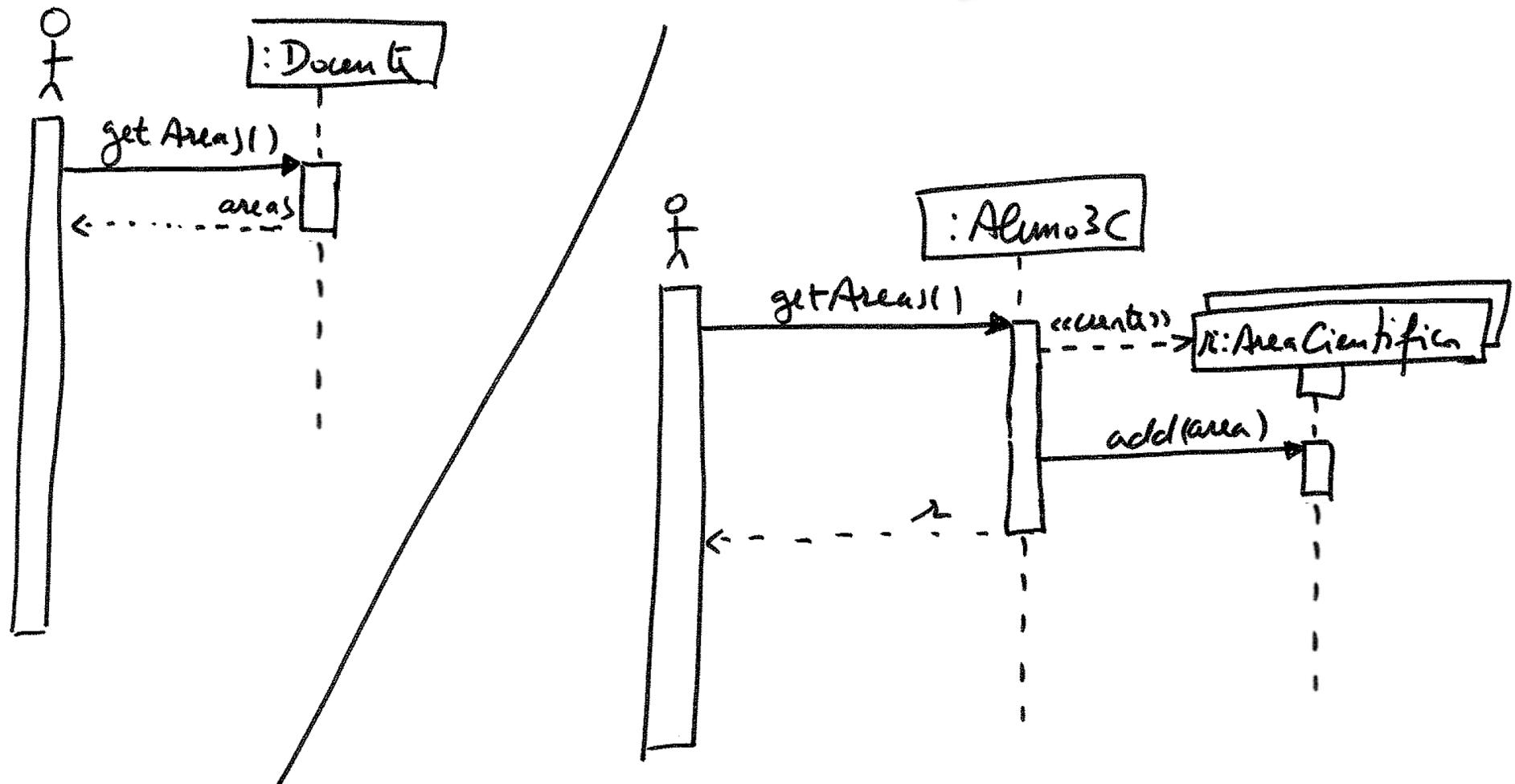
O comportamento...

(4)



O comportamento...

`getAreas()` depende
da classe concreta...





O código...

```
public class Univ {
    private Map<String,UC> ucs;
    // ...

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

}
```

Não há persistência de dados!

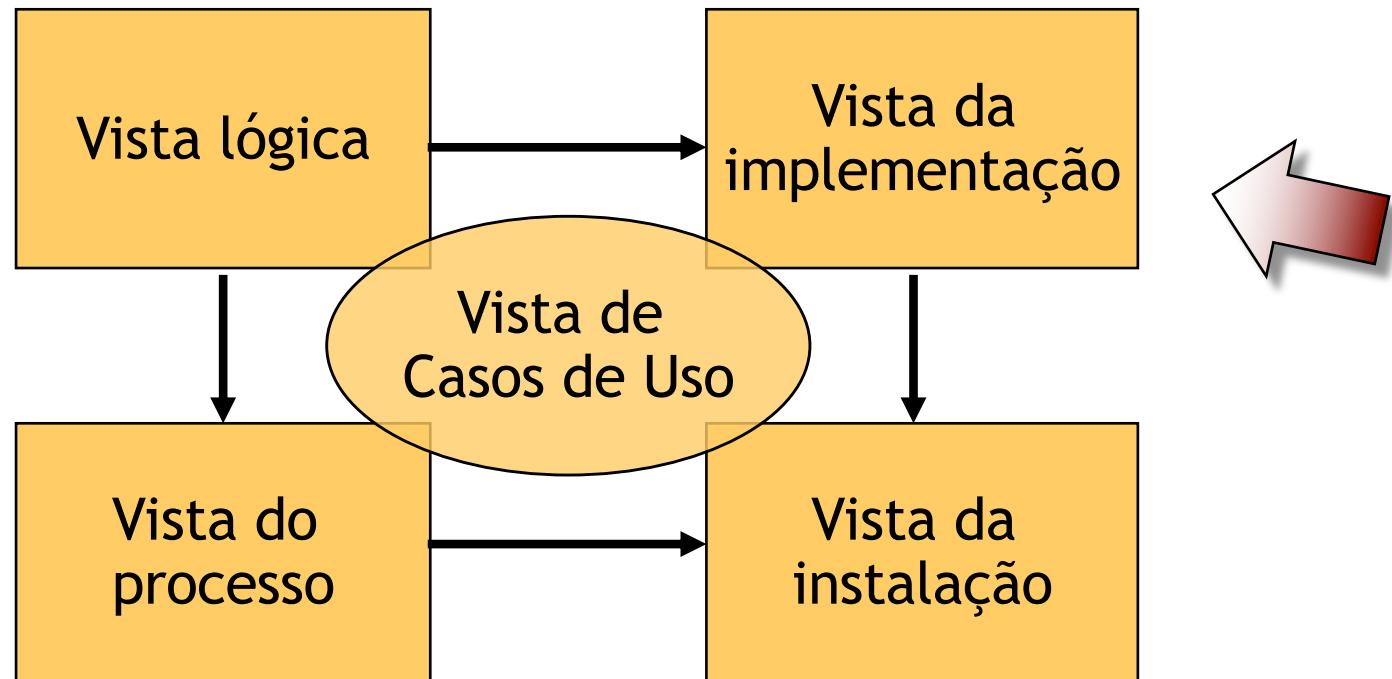
```
public class UC {

    private String codigo;
    private String ac;
    private Map<String,Turma> turmas;

    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);
    }

}
```

Onde estamos...



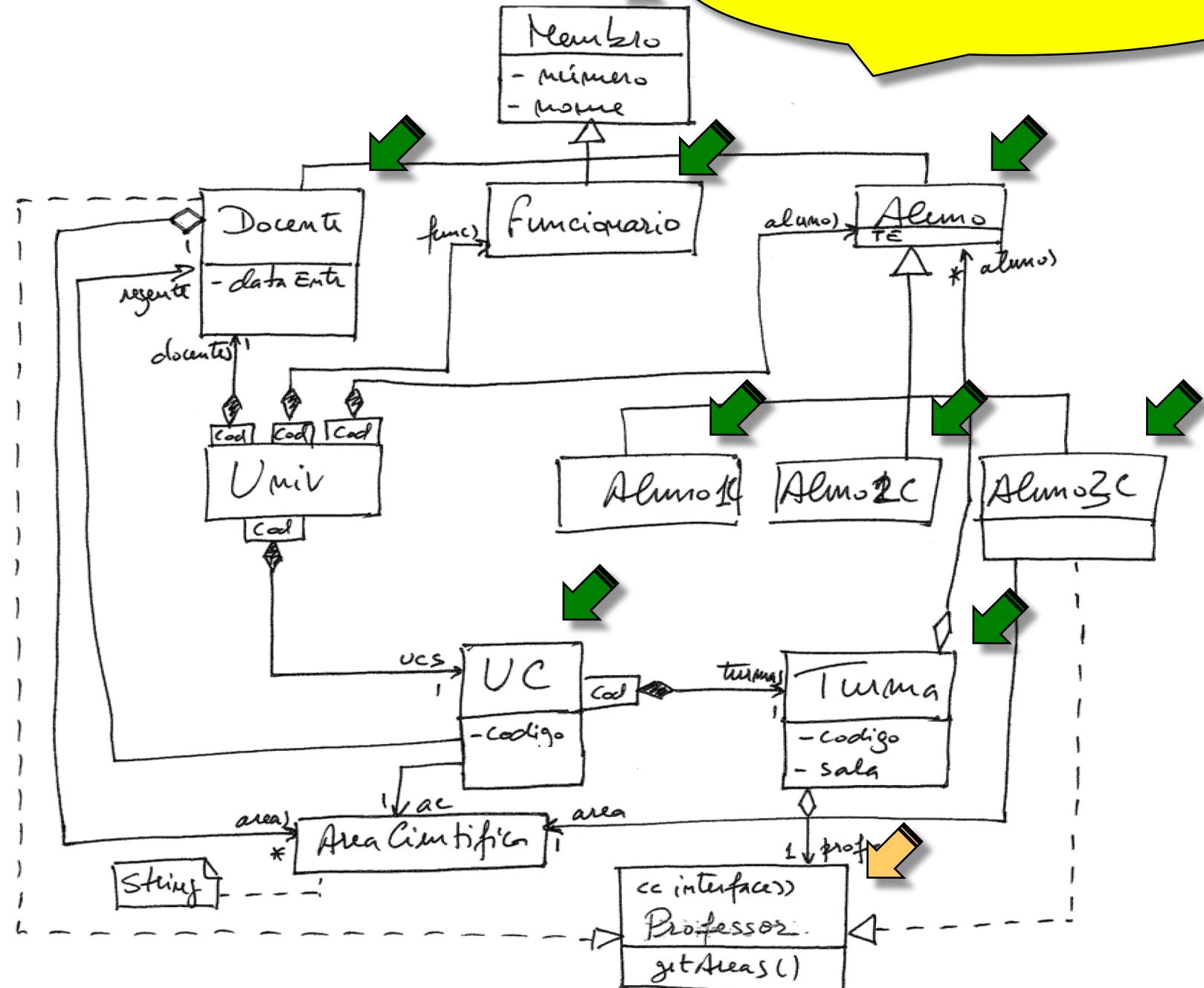


Persistência???

- Primeiro: Identificar as Entidades.
 - Que classes “armazenam” dados?
 - Essas serão as que devem ser persistidas
- Tipicamente as entidades também encapsulam algum controlo
- No entanto, existirão algumas classes que apenas existem para implementar o controlo da lógica de negócio
 - Logo, essas não é necessário persistir.

Que entidades persistir?

Modelo de Domínio ajuda
a encontrar entidades...





Persistência

- Estratégias para persistir as entidades?
 - (i.e. as classes que as representam)
- É necessário mapear as entidades/classes em tabelas
- É necessário mapear os atributos das entidades/classes em colunas das tabelas
 - Excepções:
 - alguns atributos podem não ser mapeados (ex.: atributos derivados).
 - alguns atributos podem ser mapeados para mais que uma coluna (ex.: nome ser dividido em 'nome próprio' e 'apelido')
 - vários atributos podem ser mapeados para uma mesma coluna (prefixo e número de telefone, por exemplo).

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

`Membro(nº, nome, tipo); Docente(nº, dataEnt); Funcionario(nº)`

`Aluno(nº, te); Aluno1C(nº); Aluno2C(nº); Aluno3C(nº, area)`

- 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

chave estrangeira

Que tabelas?

Membro (nº, nome, tipo)

Docente(nº, dataEntr)

Funcionario(nº)

Aluno(nº, te)

Aluno1C(nº)

Aluno2C(nº)

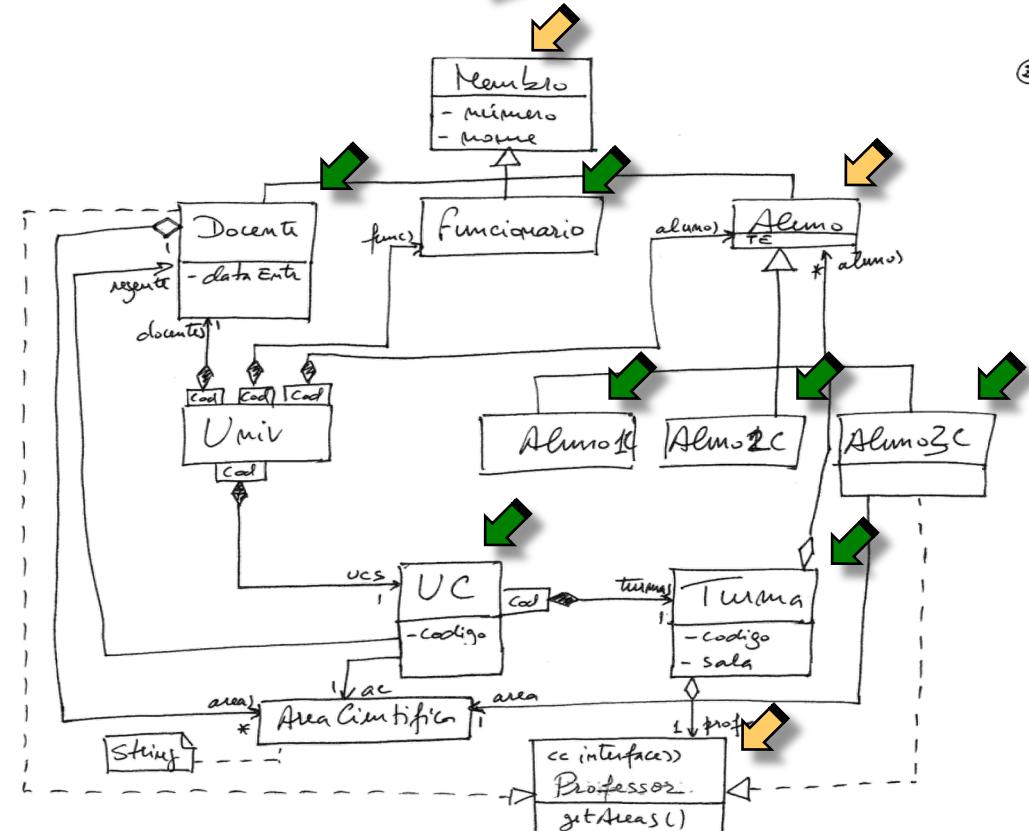
Aluno3C(nº, area)

UC(codigo, ac, regente)

Turma(codigo, codigoUC, sala, prof)

Professor(obj id, tipo, nº)

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 (que se dá 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
- Associações muitos-para-muitos
 - cria-se uma tabela intermédia de pares de chaves, identificando os dois lados do relacionamento.

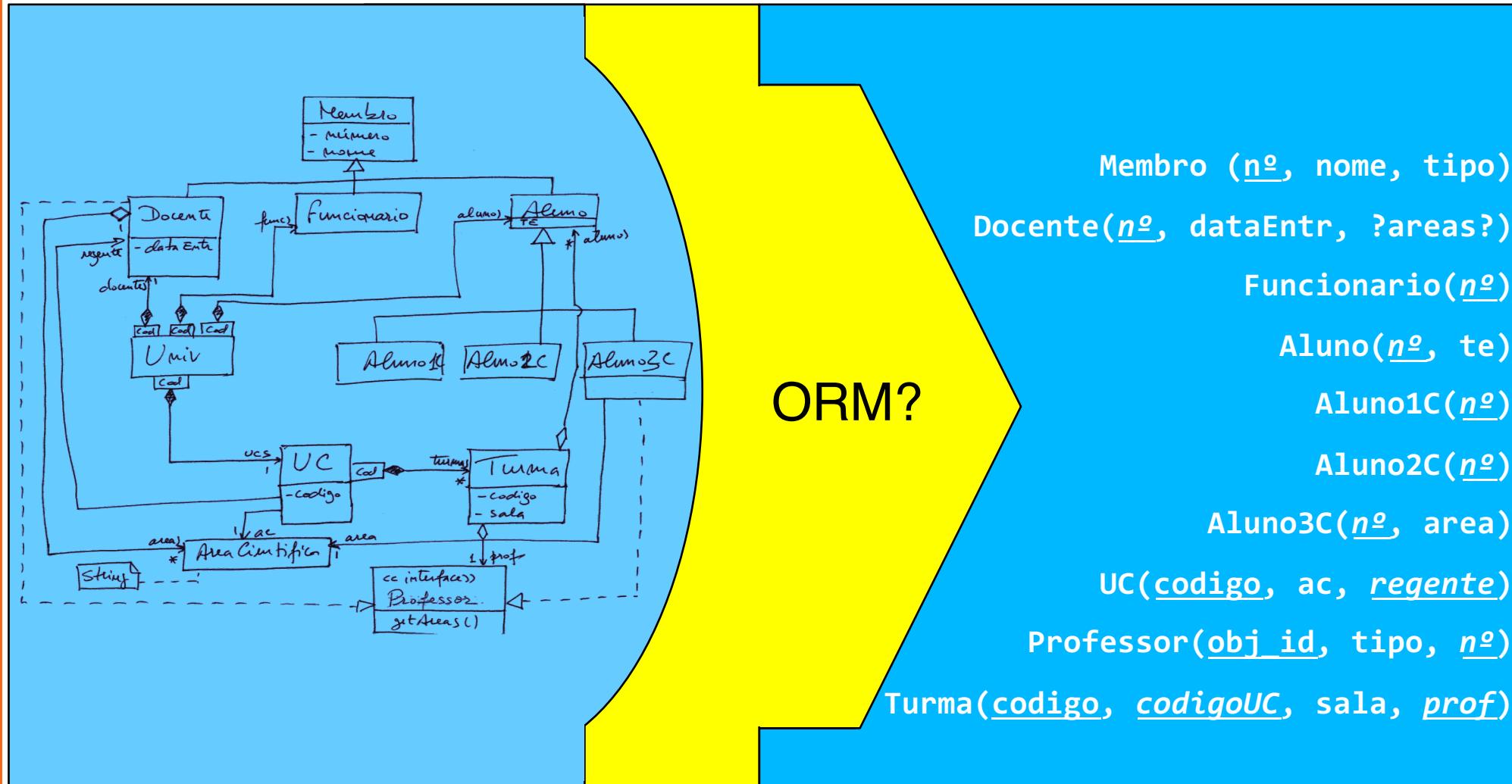
Regras de mapeamento

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.

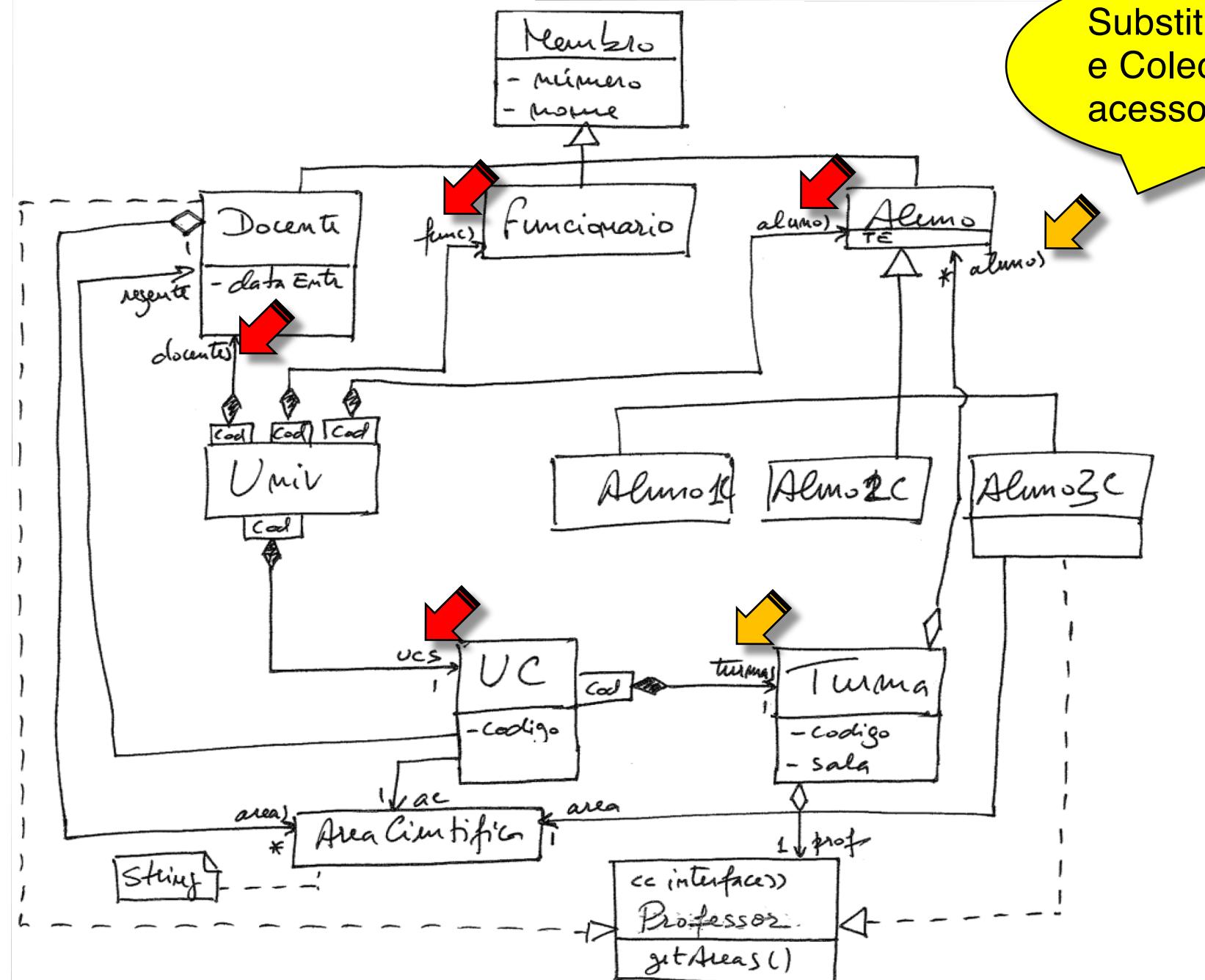
Regras de mapeamento

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 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
 - b) As subclasses não têm identidade própria:
 - tabelas que mapeiam os subclasses utilizam chave primária da superclasse
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



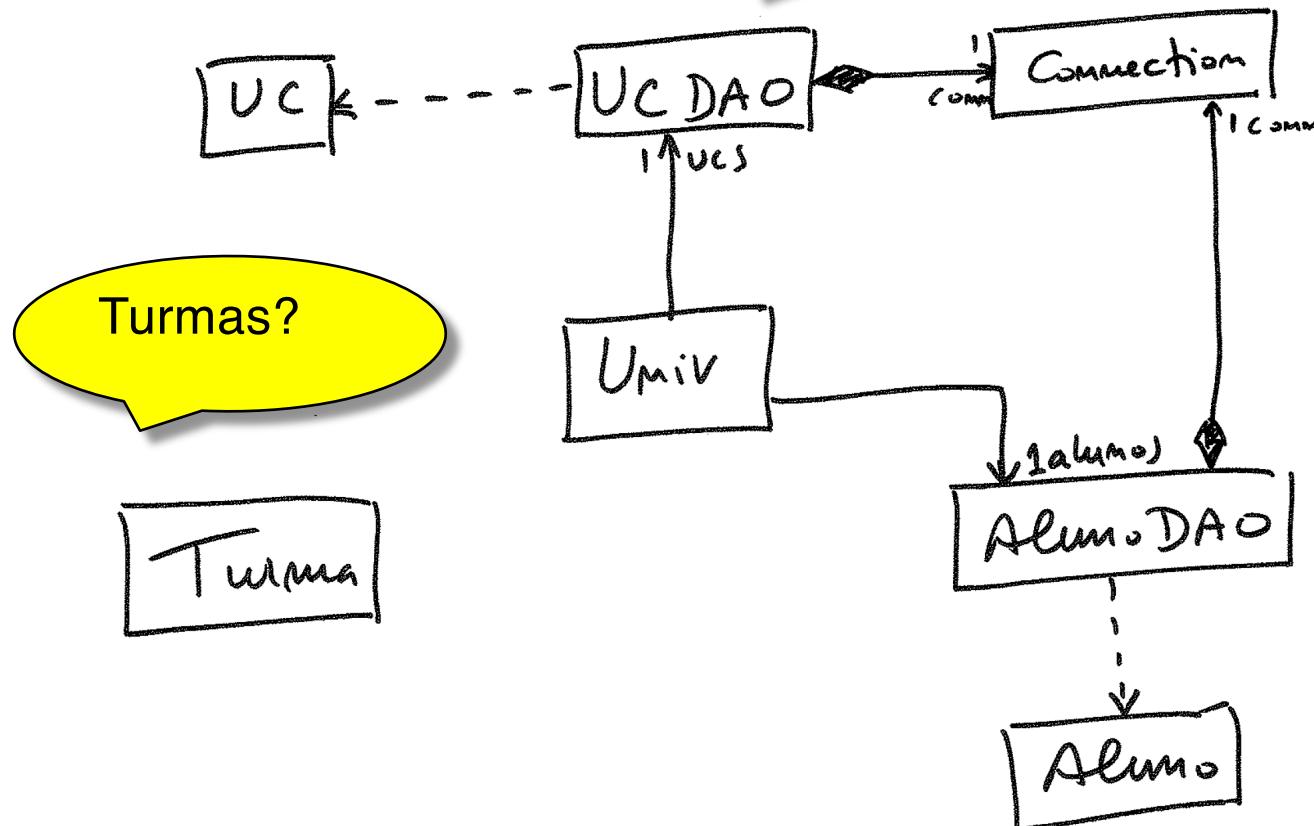
Arquitectura - versão sem persistência



Arquitectura - versão com persistência

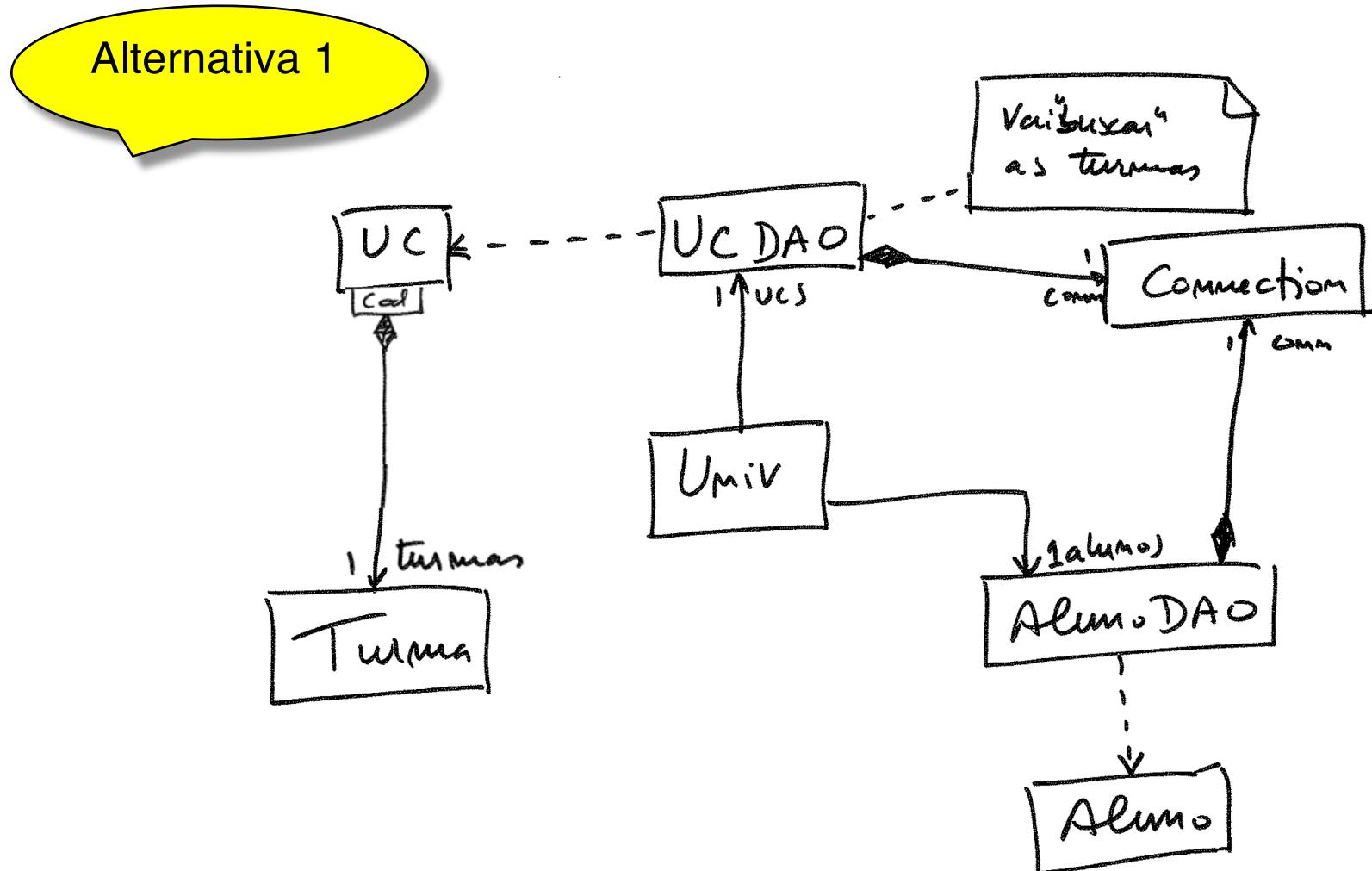
⑨

DAO = Data
Access Object



Arquitectura - versão com persistência

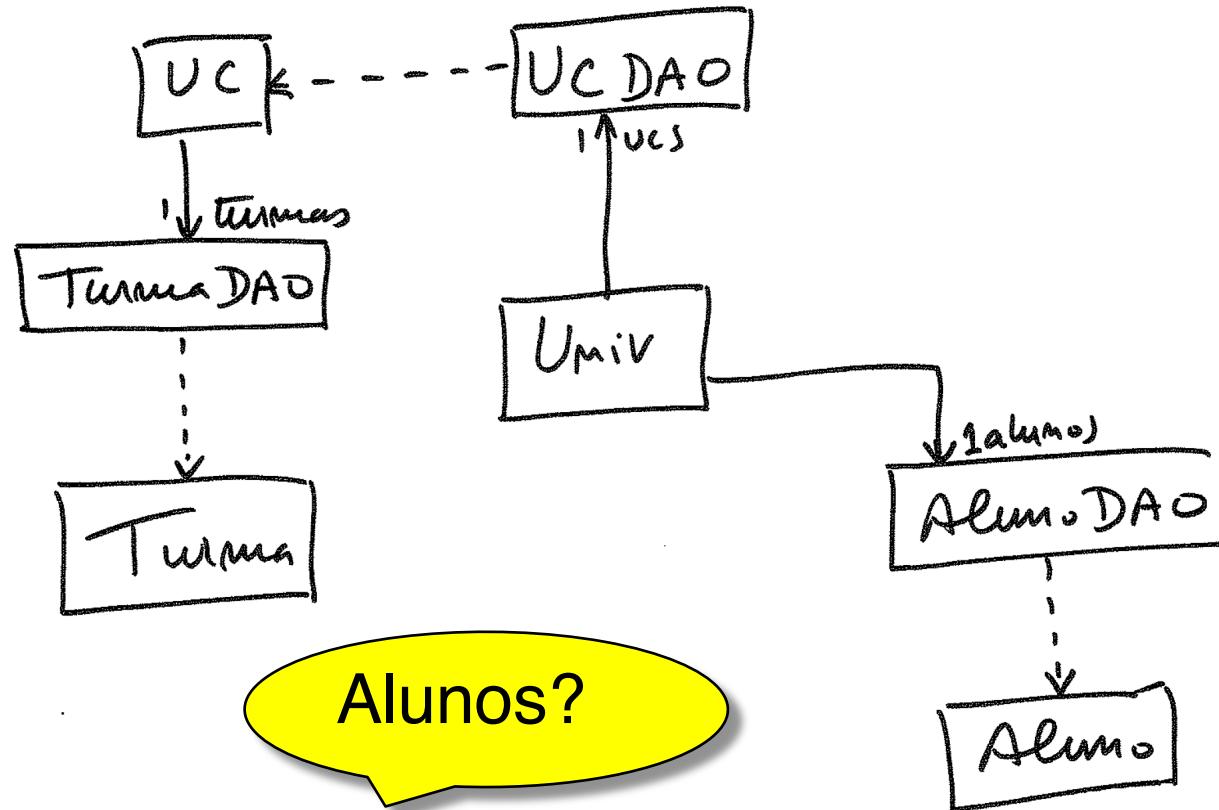
(10)



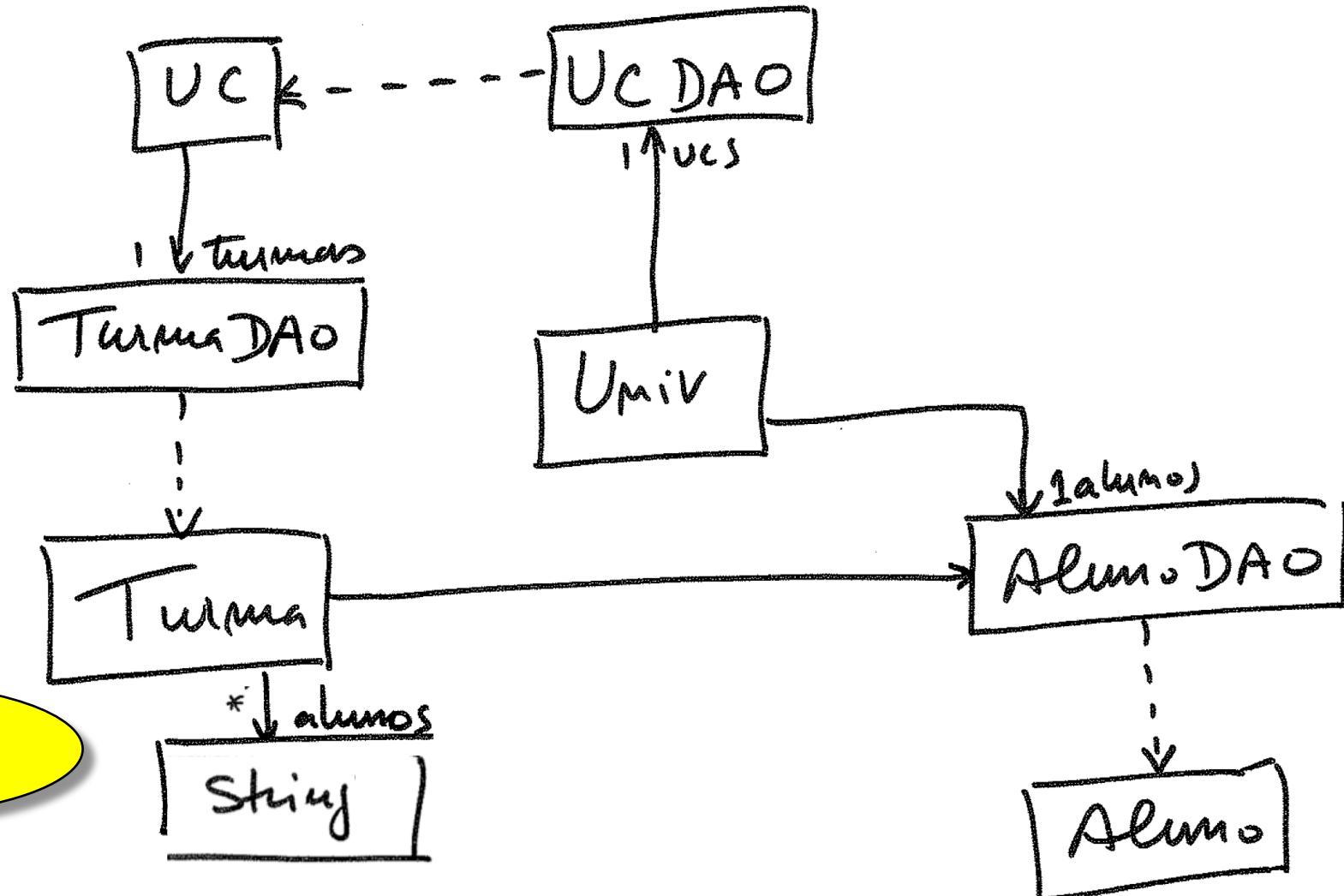
Arquitectura - versão com persistência

(11)

Alternativa 2

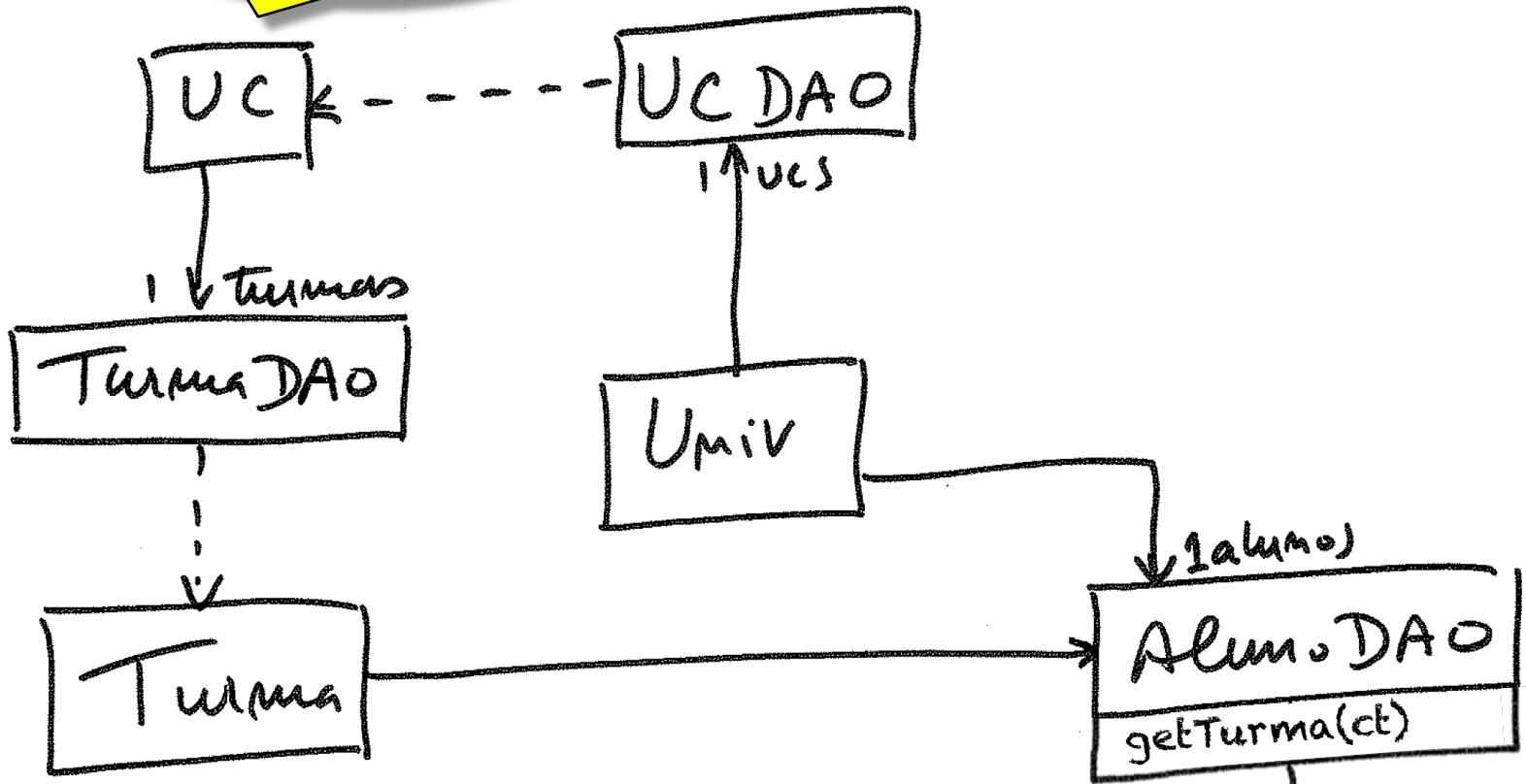


Arquitectura - versão com persistência



Arquitectura - versão com persistência

Docente?

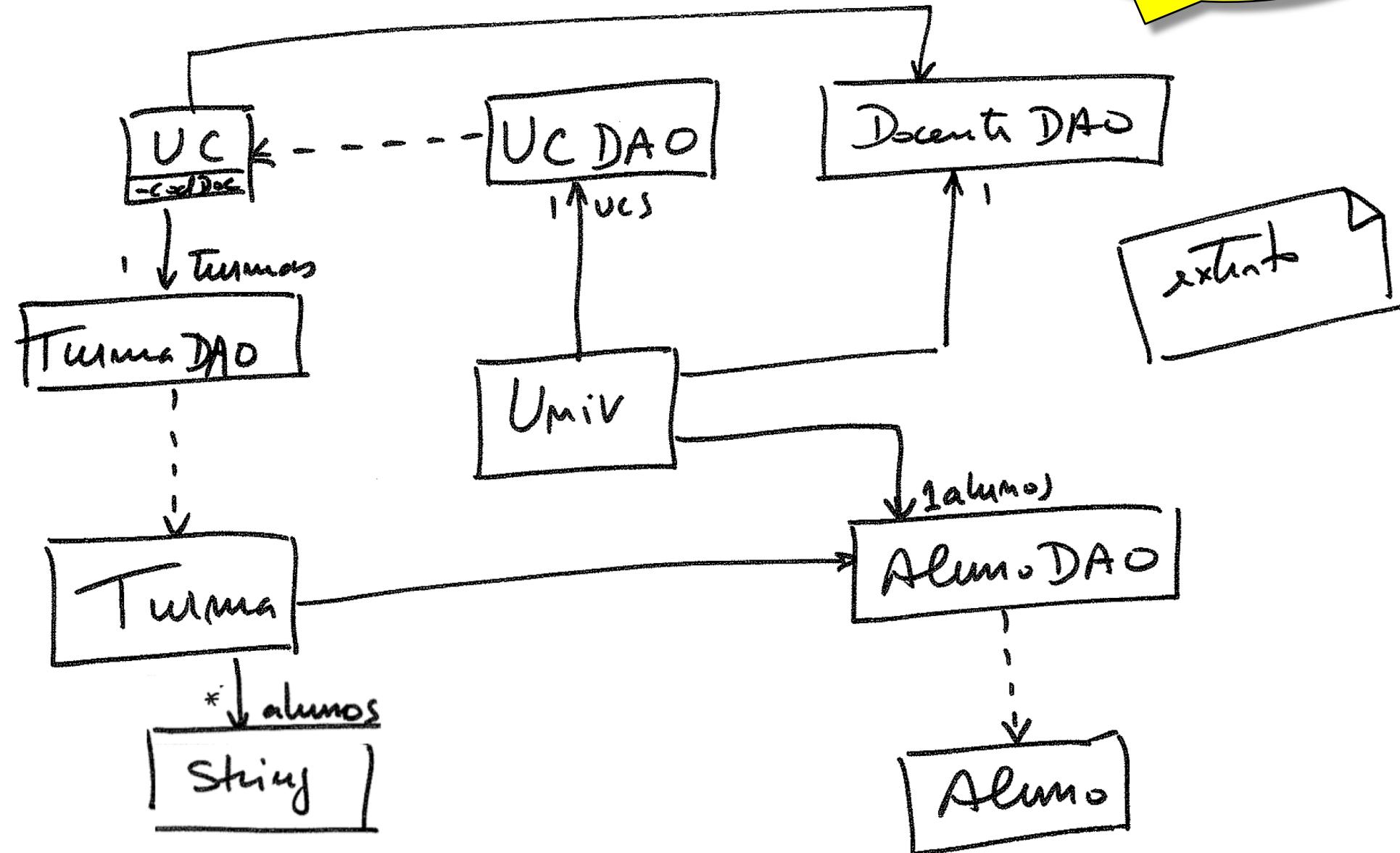


Alternativa 2

Necessária Tabela
Turma-Aluno

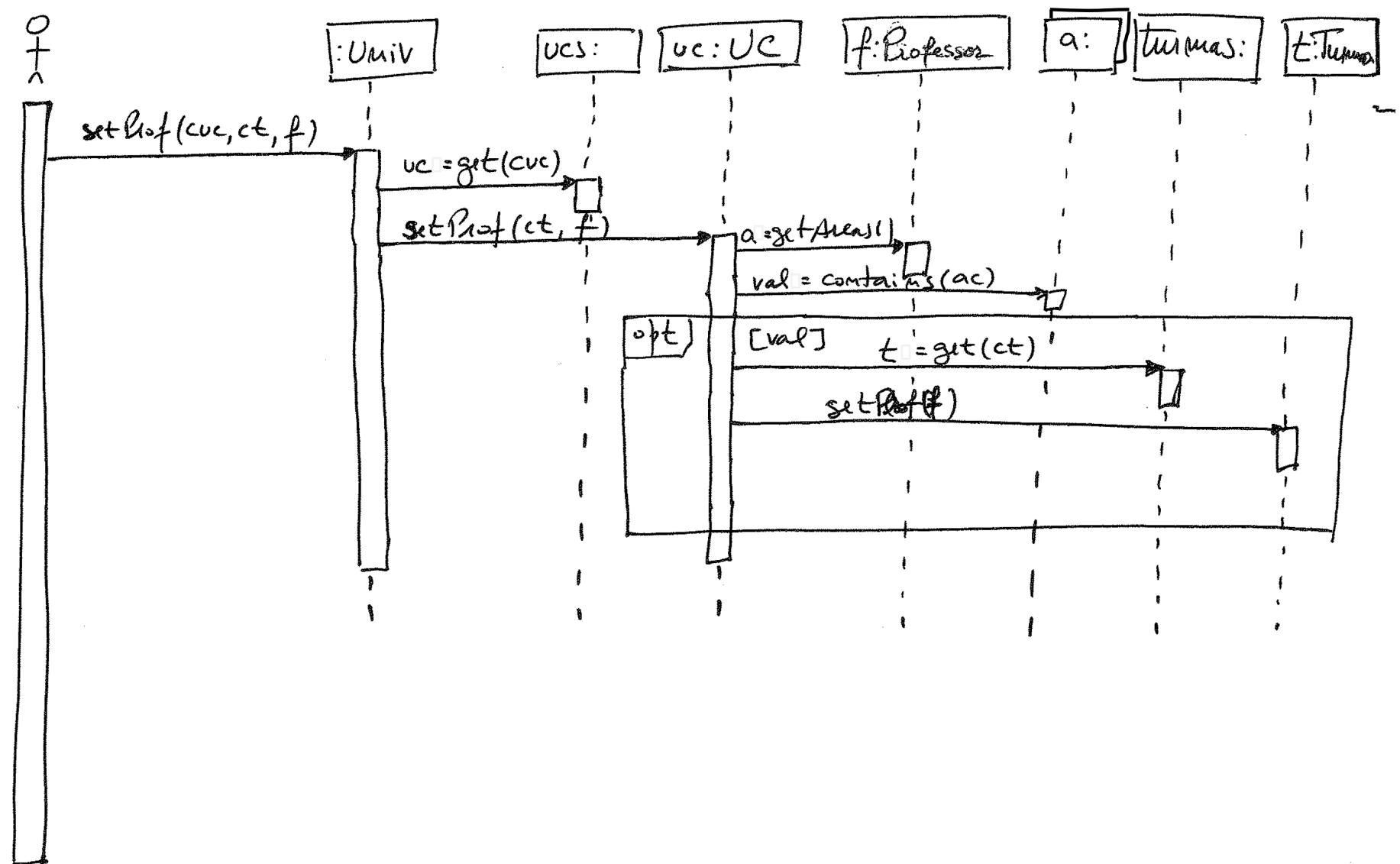
Arquitectura - versão com persistência

Falta completar
com operações,
etc.



O comportamento...

(4)

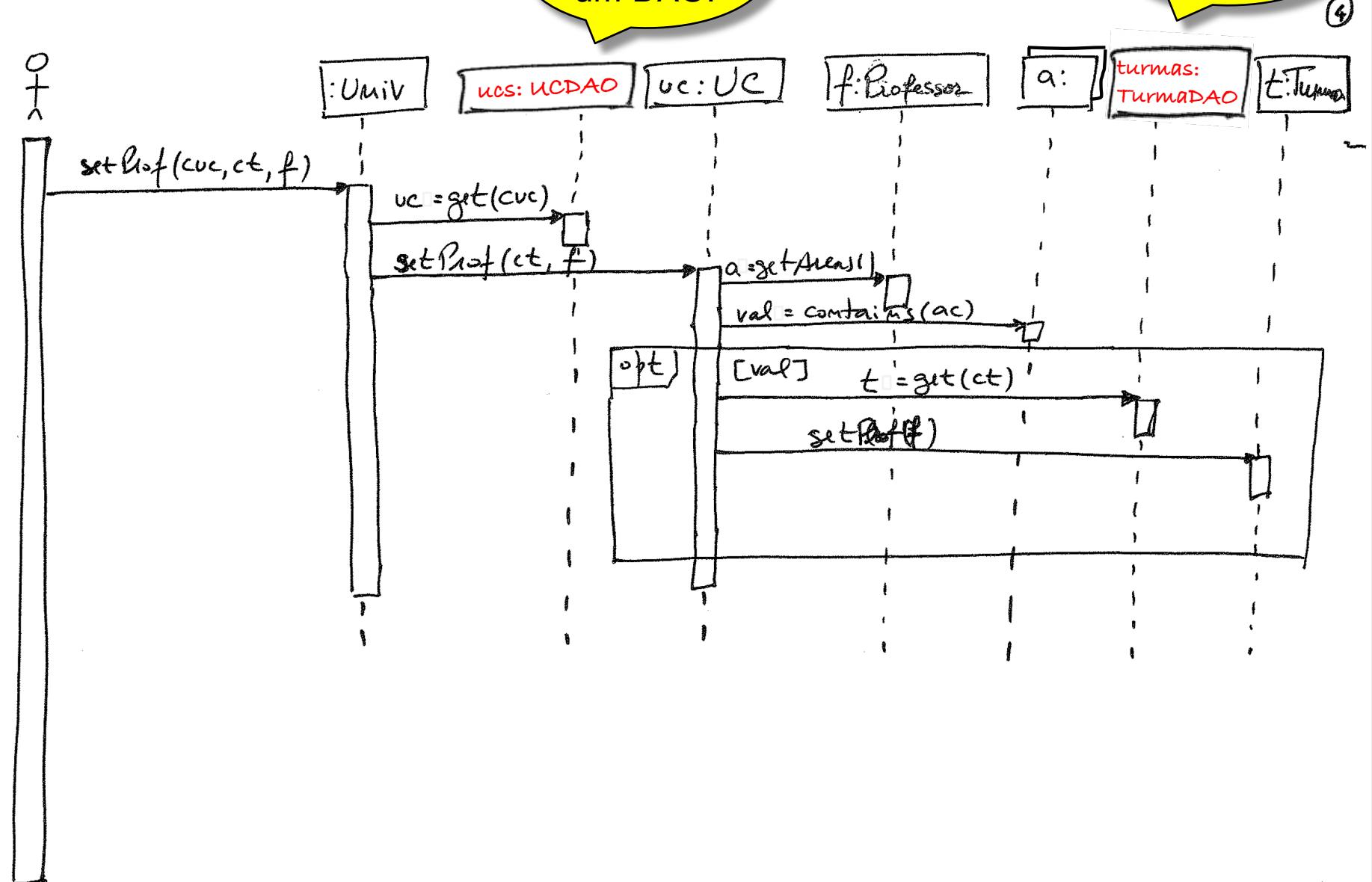




O comportamento...

Agora é um DAO.

Agora é um DAO.

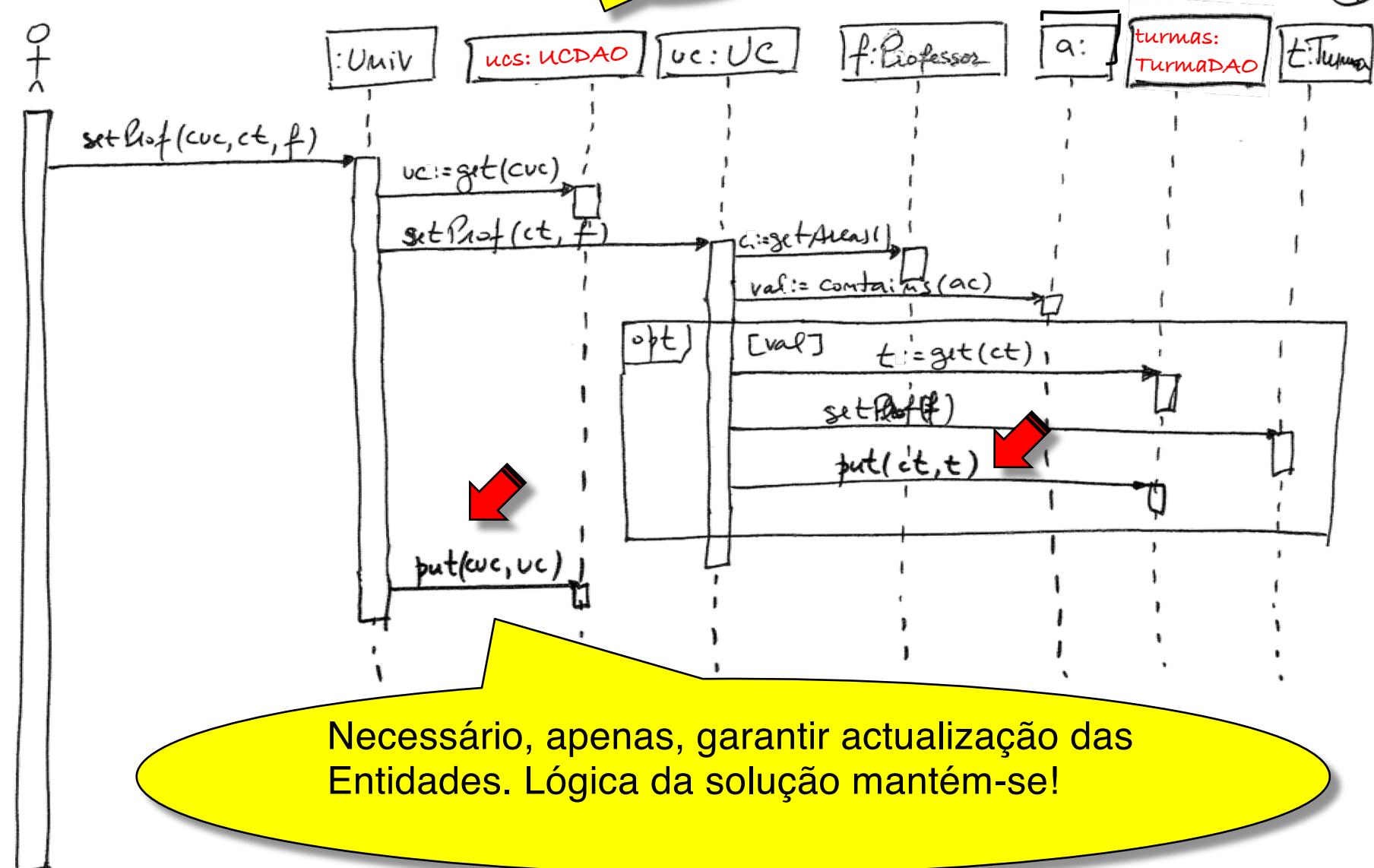




O comportamento...

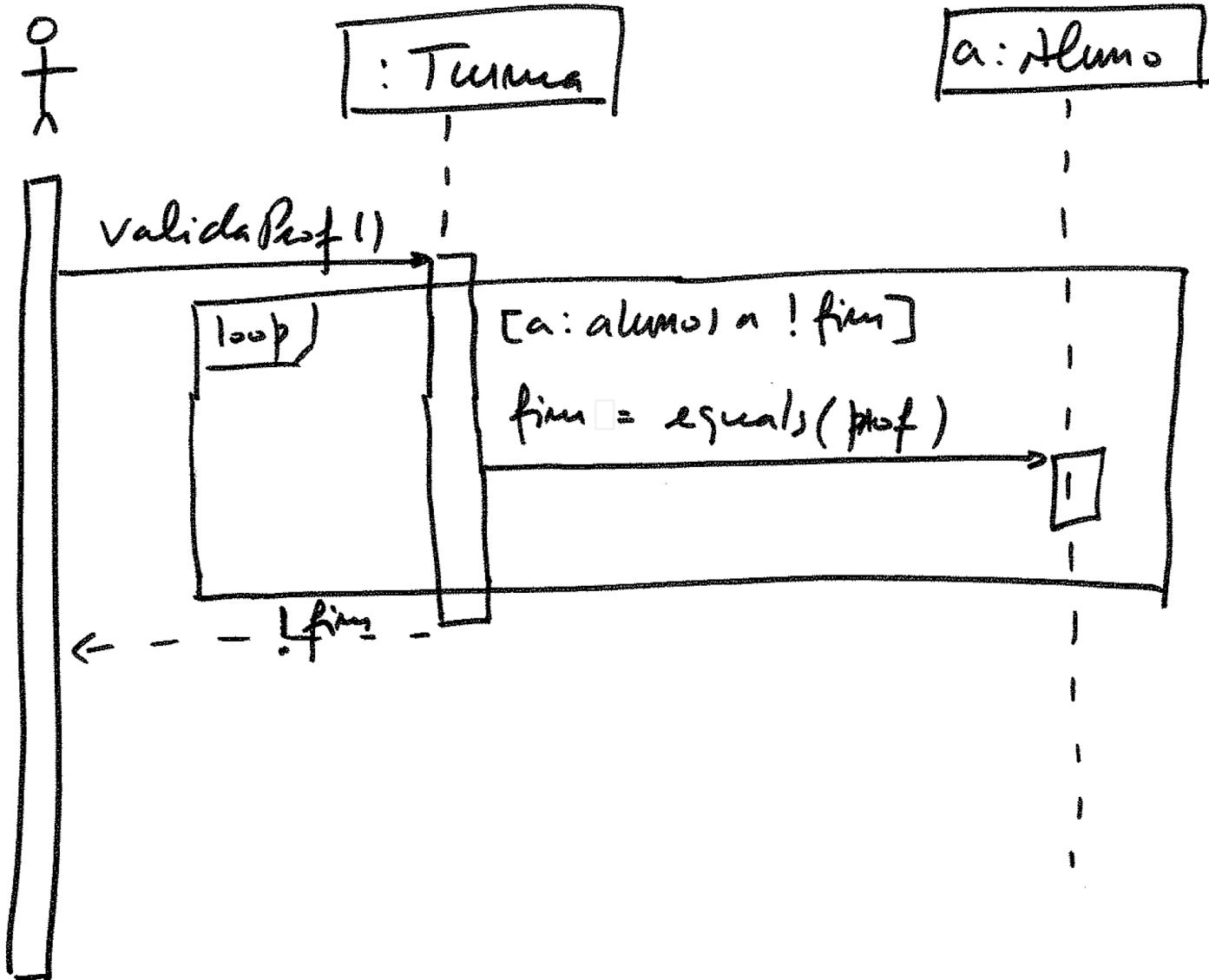
Agora é um DAO.

Agora é um DAO.



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

Outro Exemplo...



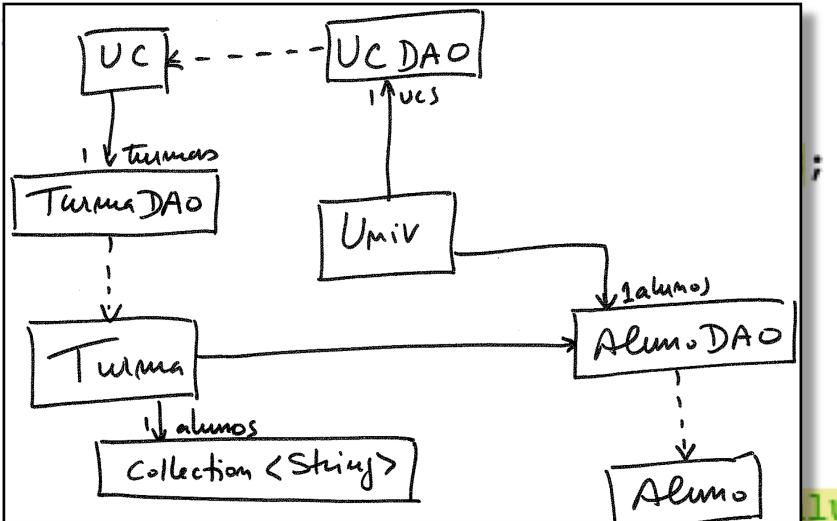


```
class Turma {  
    private String codigo, sala;  
    private Professor prof;  
    private Collection<Aluno> alunos;  
  
    void setProf(Professor f) {  
        this.prof = f;  
    }  
  
    public boolean validaProf() {  
        boolean fim = false;  
        Iterator<Aluno> alunosIt = alunos.iterator();  
  
        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;  
    }  
}
```



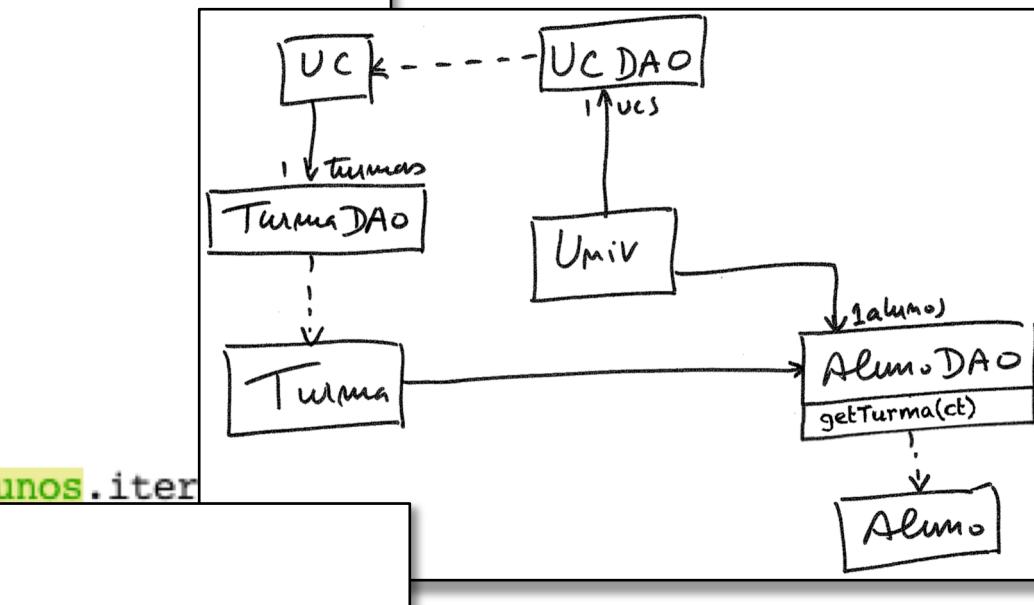
```

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();

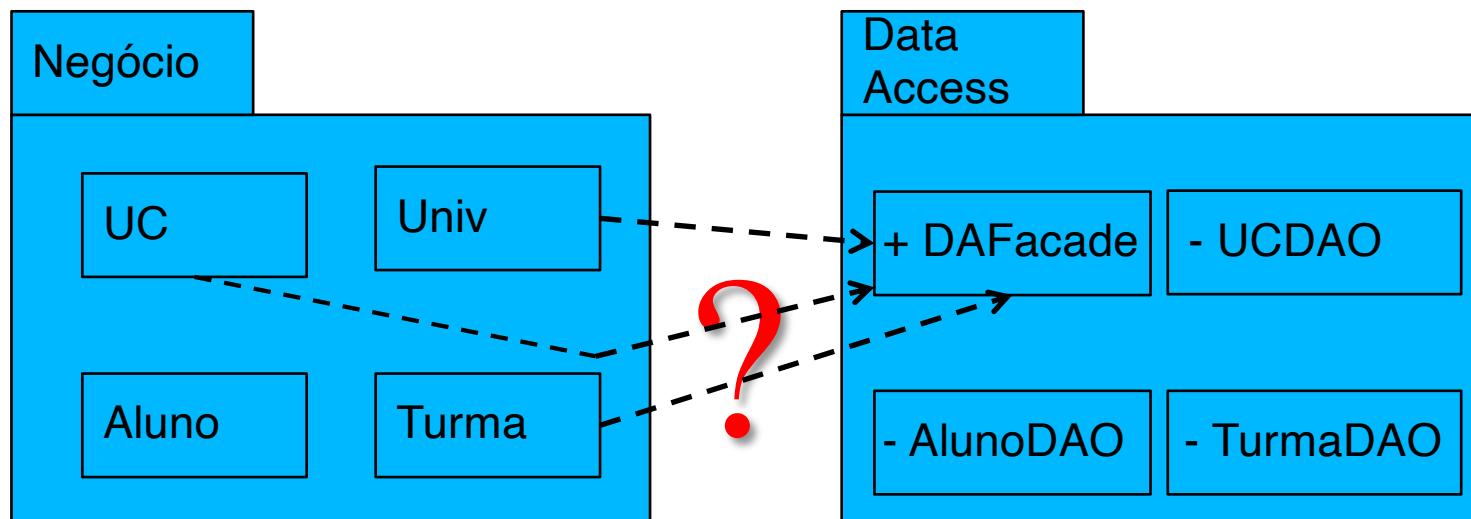
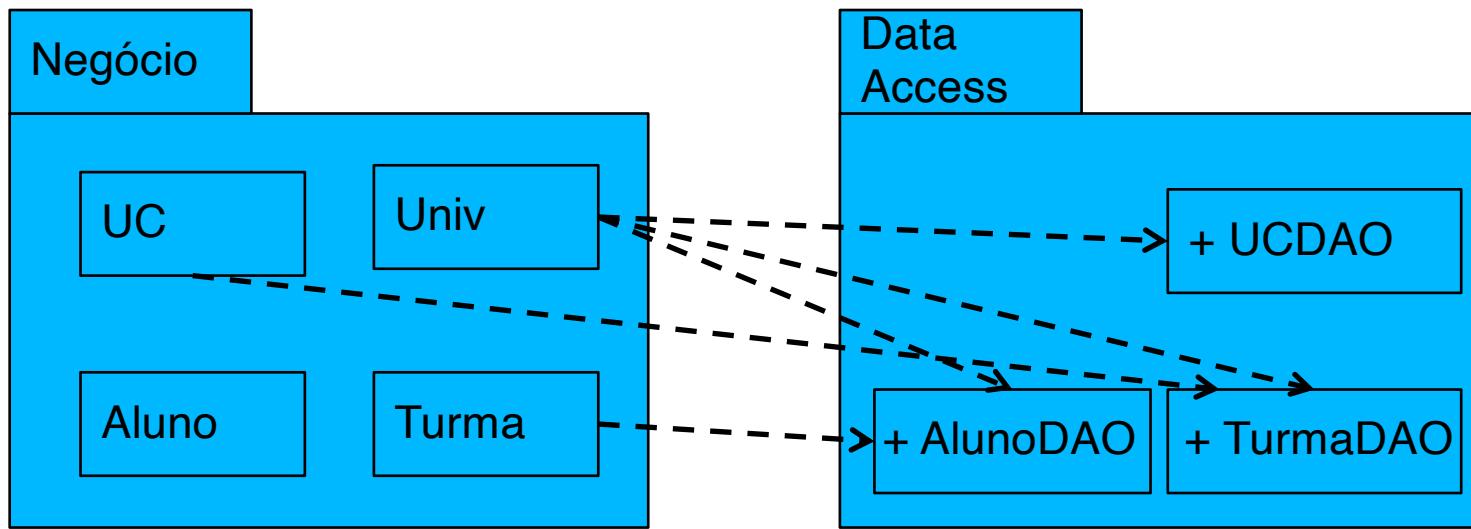
        while (!fim &amp; alunosIt.hasNext()) {
            Aluno a = alunosIt.next();
            fim = a.equals(prof);
        }
        return !fim;
    }
}
  
```



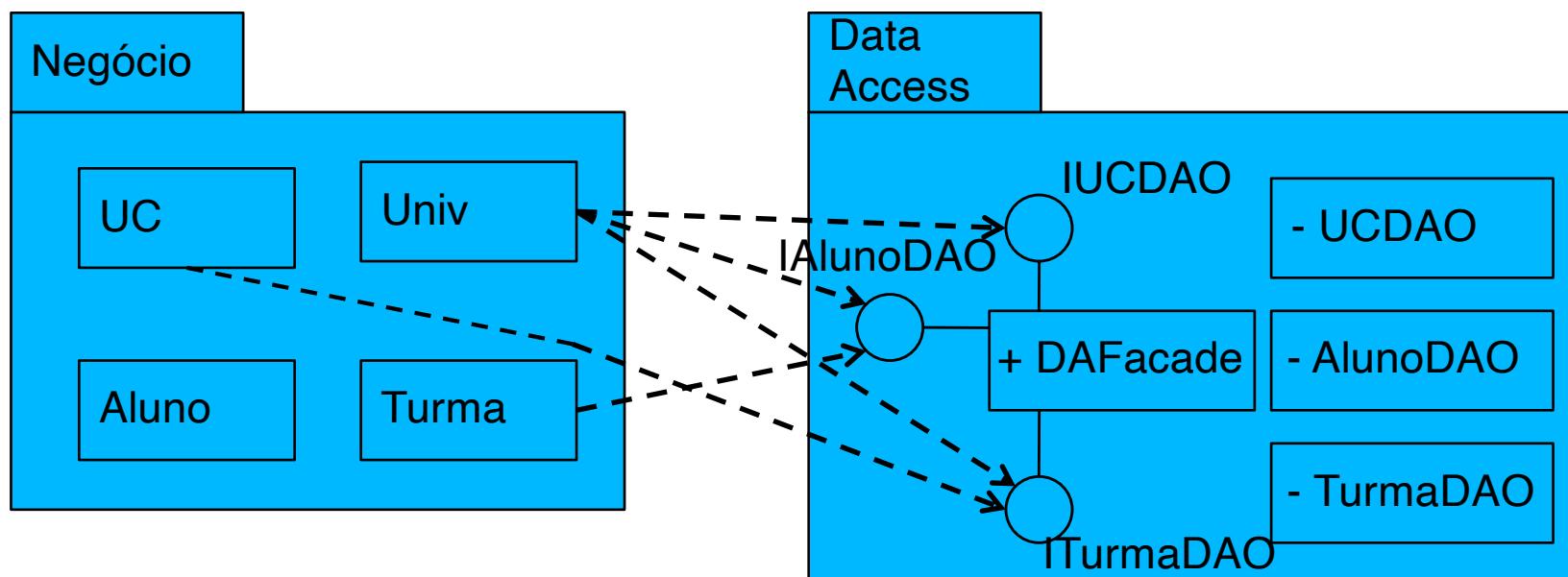
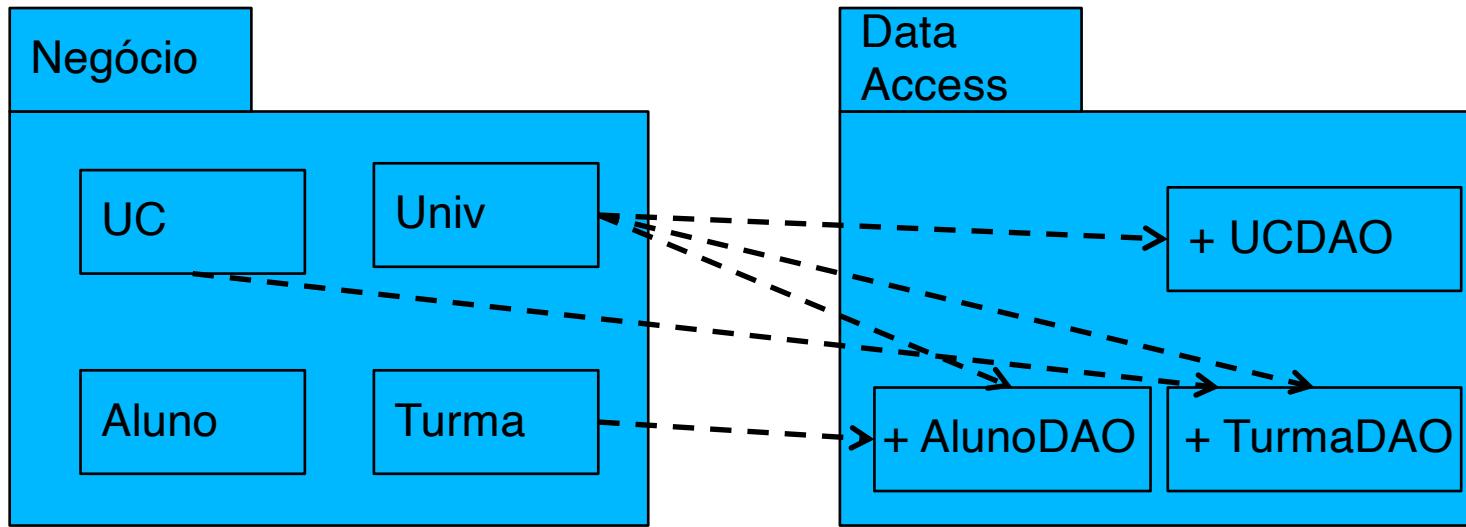
Código...

- AlunoDAO
 - (ficheiro AlunoDAO.java)
- Exemplo da Turma (3 camadas)
 - DAOs ficam num package que faz acesso aos dados
 - Podemos ou não colocar um *Facade*
 - *Facade* pode implementar multiplas interfaces

Packages



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
 - substituídos por acessos à camada de dados
 - reconstruem 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



ORM

Sumário

- Mapeamento Objetos <-> Relacional
- Regras de mapeamento
 - Atributos
 - Classes
- Exemplos