ORM (Object-Relational Mapping)

Desenvolvimento de Sistemas Software
José Creissac Campos / António Nestor Ribeiro

Sobre o trabalho

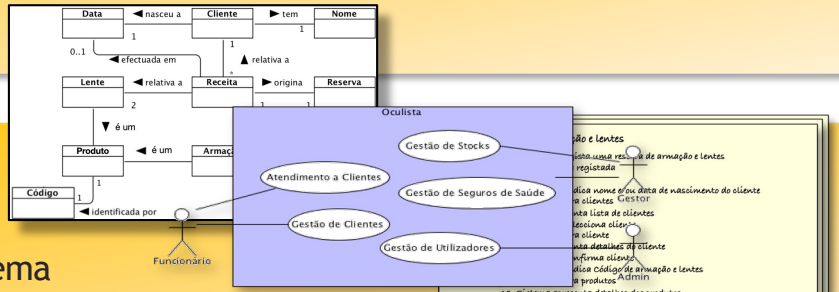
"Problema"

Planeamento

- Decisão de avançar com o projecto
- Gestão do projecto

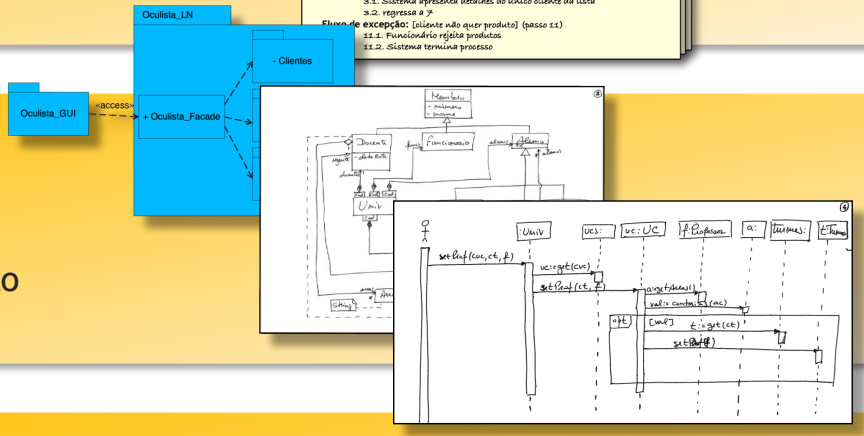
Análise

- Análise do domínio do problema
- Análise de requisitos



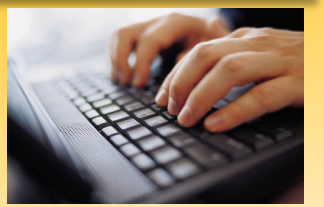
Concepção

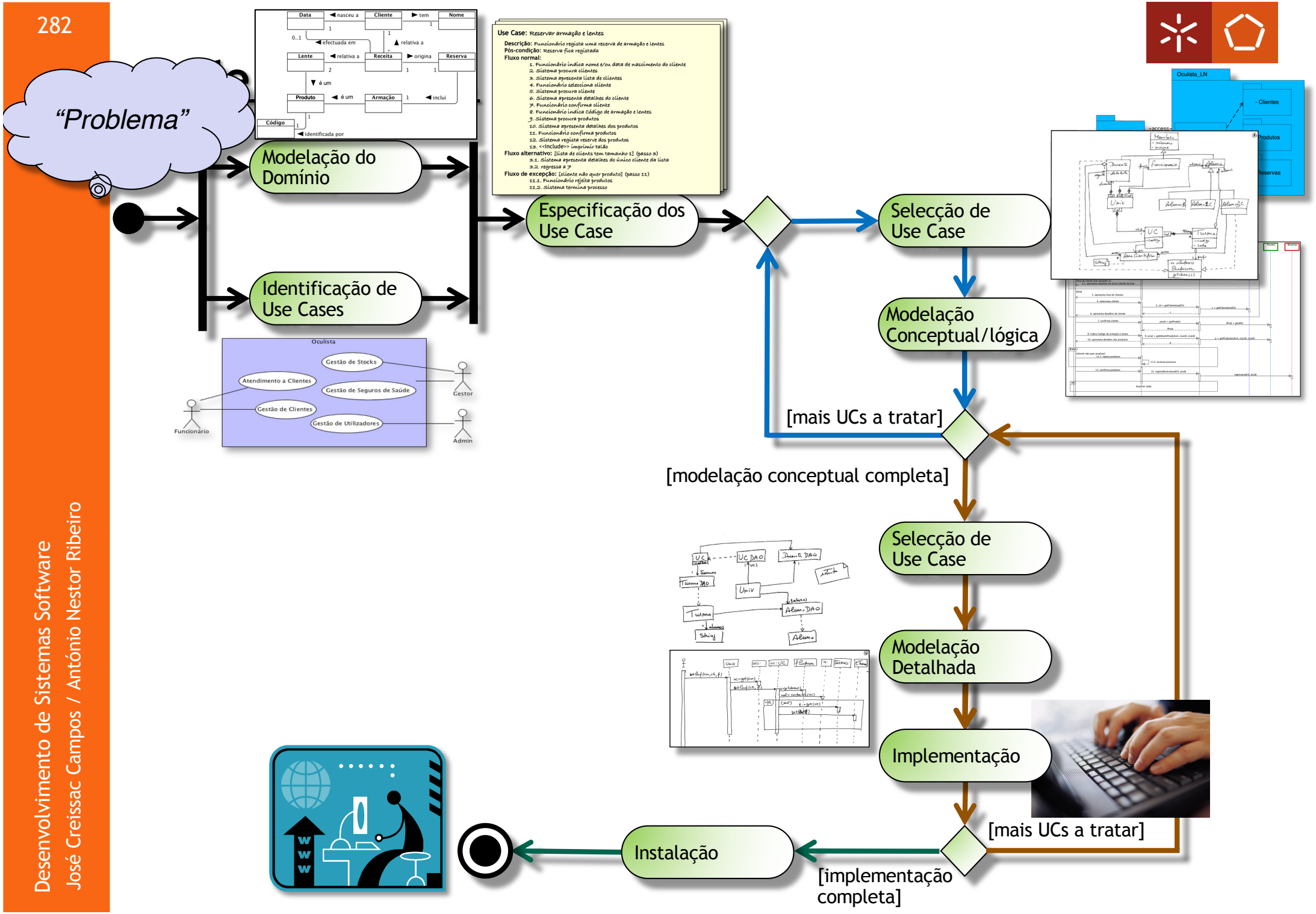
- Concepção da Arquitectura
- Concepção do Comportamento



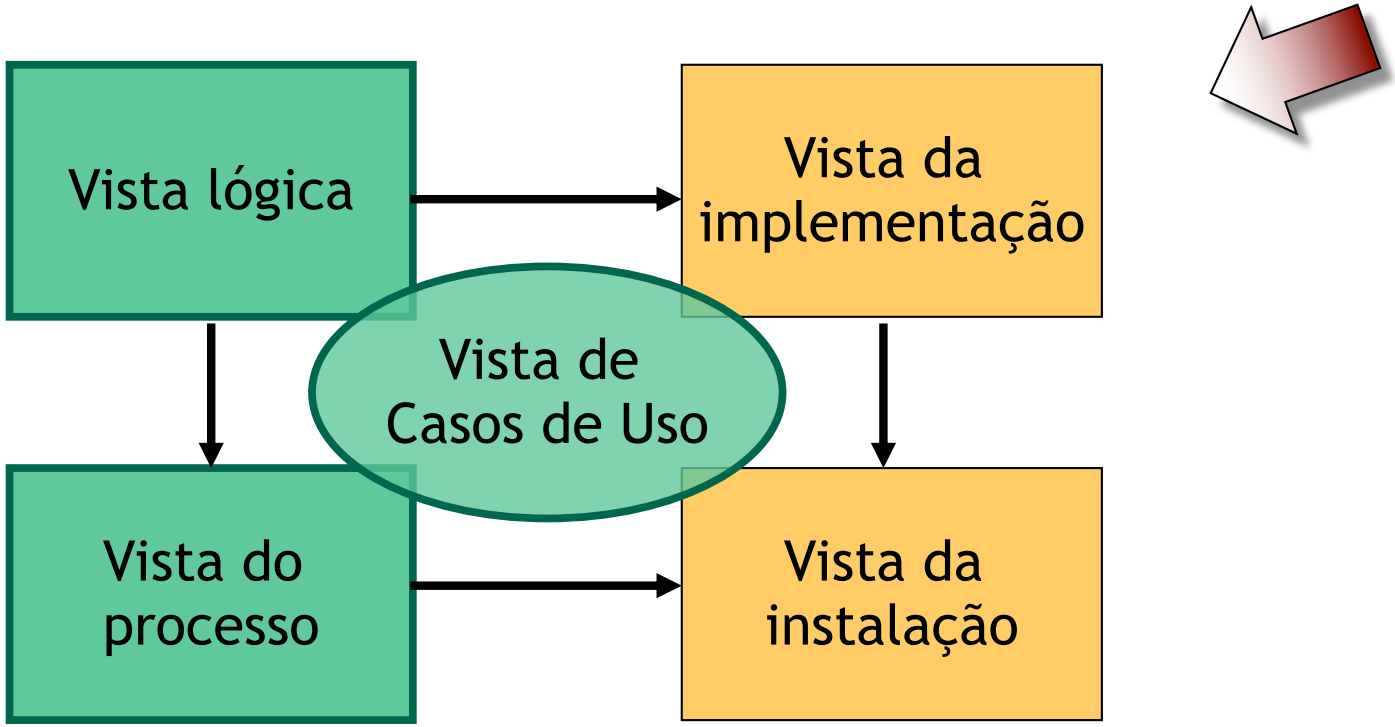
Implementação

- Construção
- Teste
- Instalação
- Manutenção



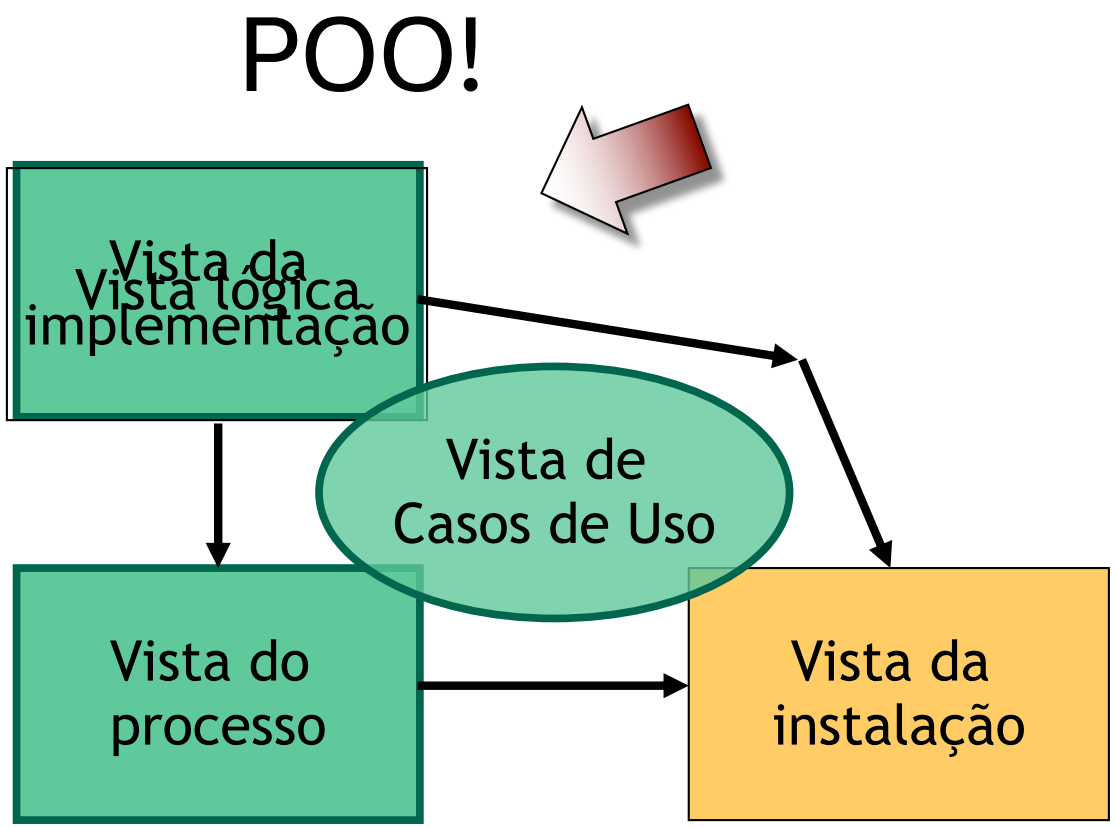

"Problema"
Modelação do Domínio
Identificação de Use Cases
Especificação dos Use Case
Selecção de Use Case
Modelação Conceptual/lógica
[mais UCs a tratar]
[modelação conceptual completa]
Selecção de Use Case
Modelação Detalhada
Implementação
[mais UCs a tratar]
[implementação completa]
Instalação
Data
Cliente
Nome
nasceu a
tem
efectuada em
relativa a
Lente
Receita
Reserva
origina
é um
Produto
Armação
inclui
Código
identificada por
Use Case: Reservar armação e lentes
Descrição: Funcionário regista uma reserva de armação e lentes
Pós-condição: Reserva fica registada
Fluxo normal:
1. Funcionário indica nome e/ou data de nascimento do cliente
2. Sistema procura clientes
3. Sistema apresenta lista de clientes
4. Funcionário selecciona cliente
5. Sistema procura cliente
6. Sistema apresenta detalhes do cliente
7. Funcionário confirma cliente
8. Funcionário indica Código de armação e lentes
9. Sistema procura produtos
10. Sistema apresenta detalhes dos produtos
11. Funcionário confirma produtos
12. Sistema regista reserva dos produtos
13. <<include>> imprimir talão
Fluxo alternativo: [lista de clients tem tamanho 1] (passo 3)
3.1. Sistema apresenta detalhes do único cliente da lista
3.2. regressa a 7
Fluxo de excepção: [cliente não quer produto] (passo 11)
11.1. Funcionário rejeita produtos
11.2. Sistema termina processo
Oculista
Gestão de Stocks
Atendimento a Clientes
Gestão de Seguros de Saúde
Gestão de Clientes
Gestão de Utilizadores
Gestor
Funcionário
Admin
Oculista_LN
- Clientes
Produtos
Reservas

Onde estamos…



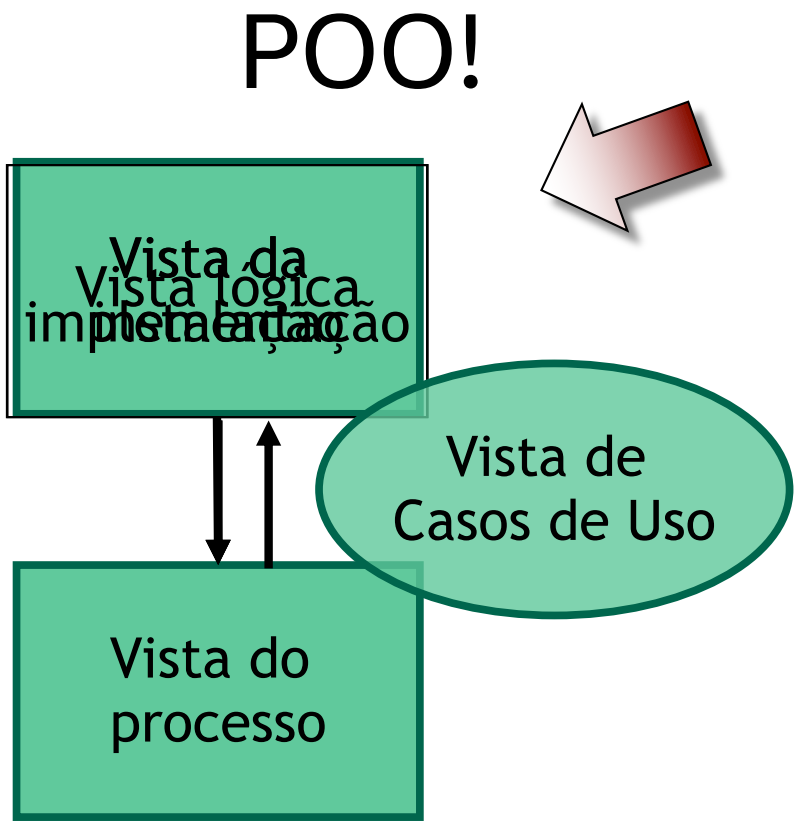
(Kruchten, 1995)

Onde estamos…



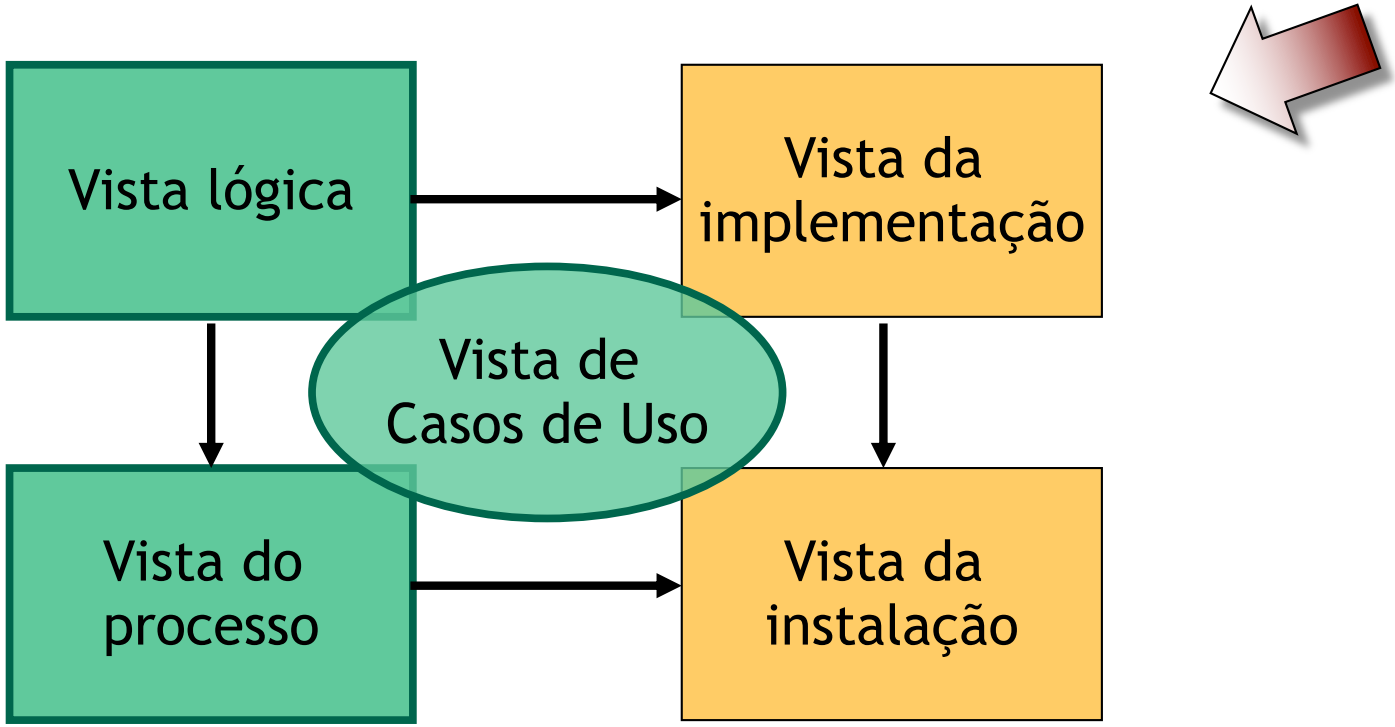
(Kruchten, 1995)

Onde estamos…



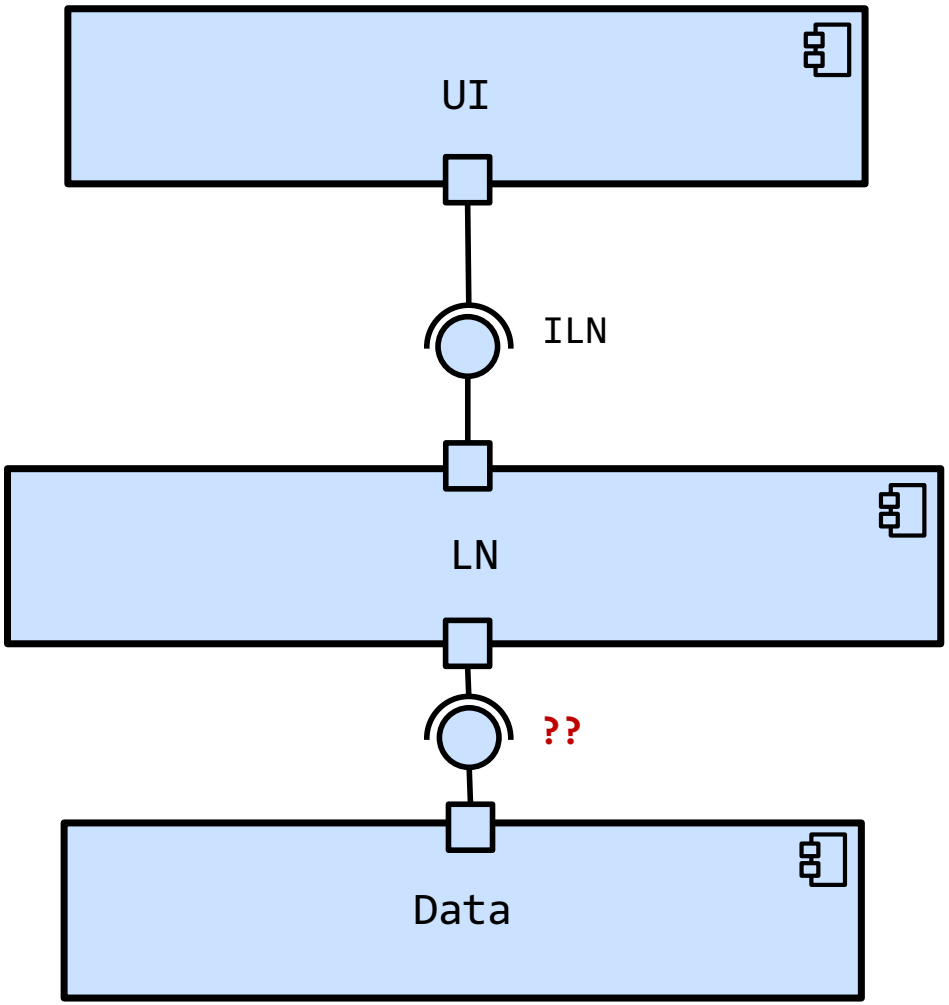
(Kruchten, 1995)

Onde estamos…



(Kruchten, 1995)

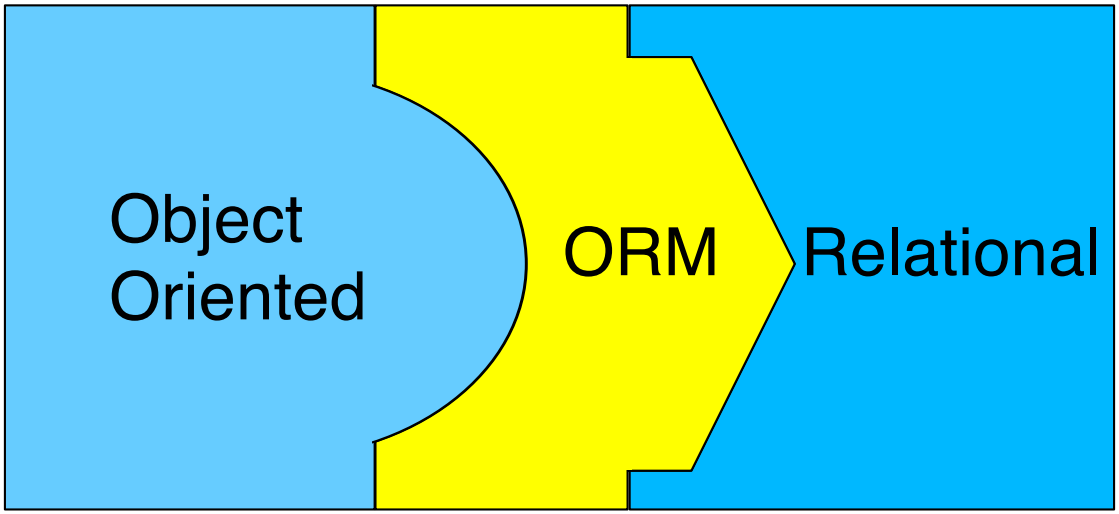
Arquitectura de 3 camadas para o exemplo



Object Relational Mapping - ORM

Camada de Negócio

Persistência de dados



- Tecnologias OO
  - Independente da camada de persistência

- Bases de dados relacionais
  - ou NoSQL
  - ou orientada a objectos
  - ou…

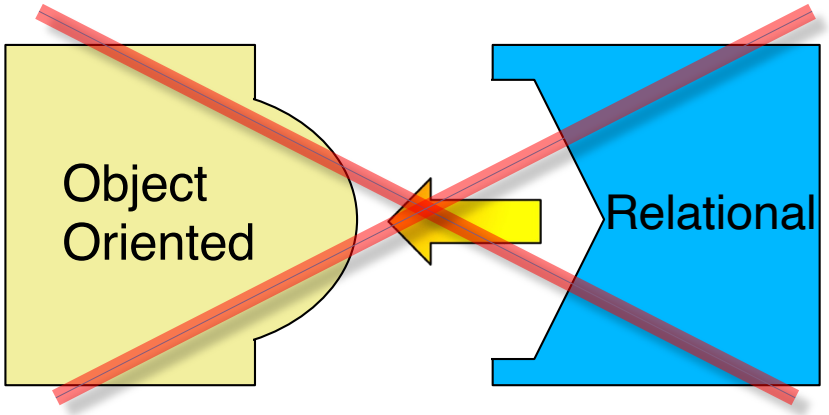
Paradigma OO vs Relacional

- Paradigma OO prevalente na programação da lógica de negócios
- Paradigma relacional prevalente nas bases de dados (persistência)

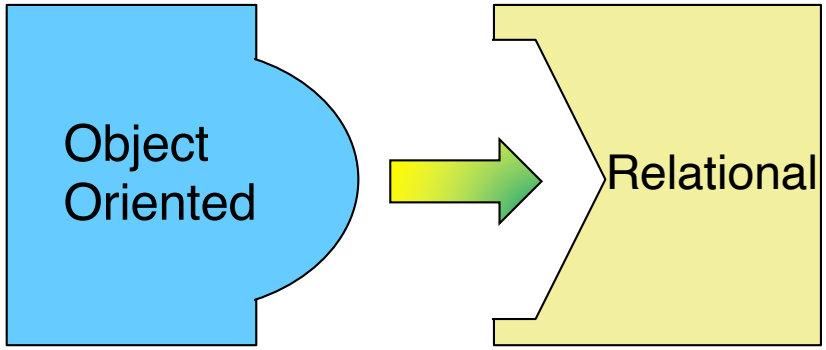
- Paradigma orientado a objetos e paradigma relacional não são diretamente compatíveis
  - Diagramas de classe não são esquemas de base de dados!
  - Modelo relacional não possui caraterísticas presentes no modelo OO como poliformismo ou identidade.
  - Objectos possuem identidade.
  - Linhas numa tabelas são identificadas por chaves.

- Torna-se necessário estabelecer um mapeamento entre os dois paradigmas - **O**bject **R**elational **M**aping (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 paradiagma 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 paradiagma 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…)

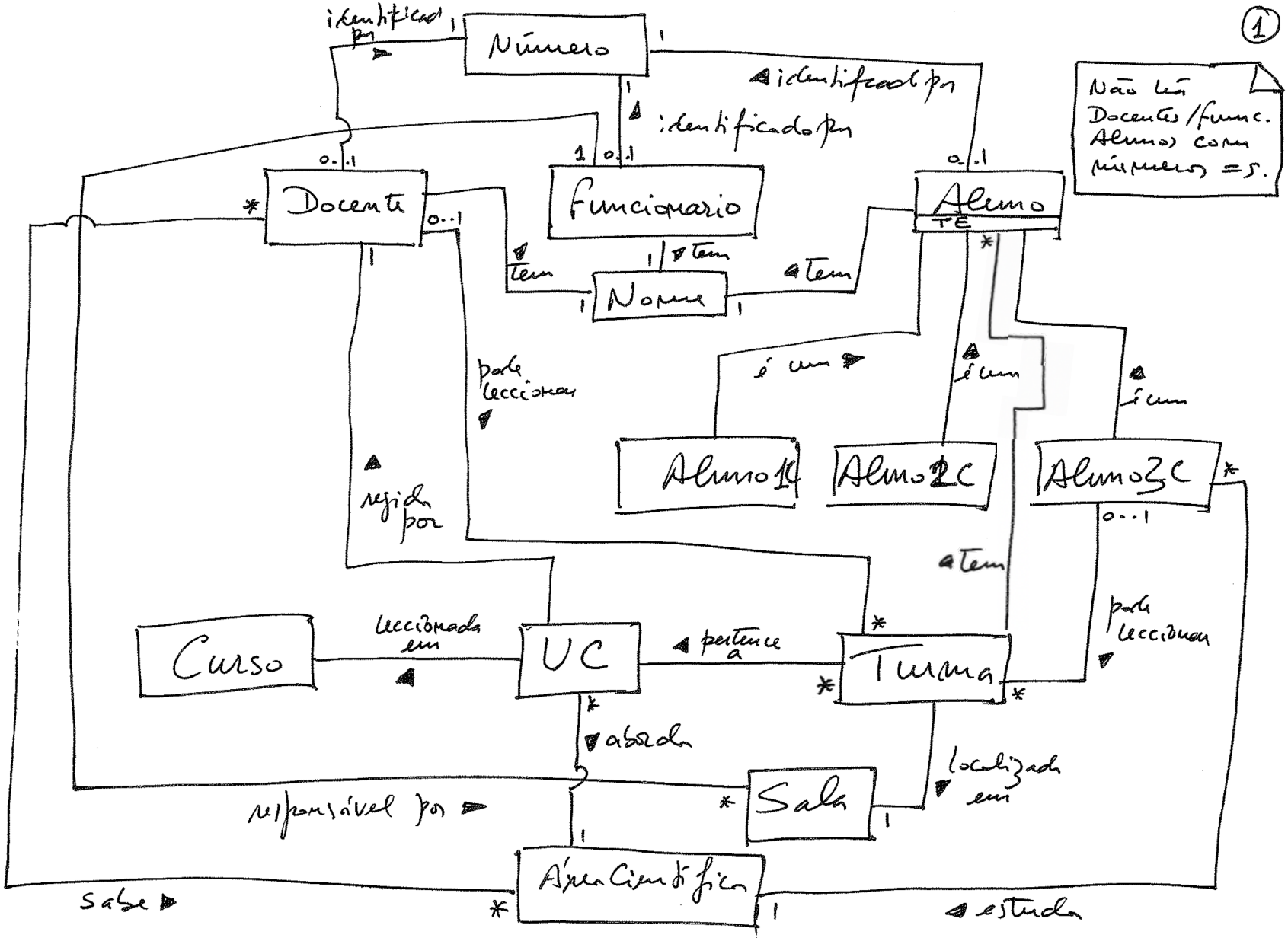
Persistência???

- 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.

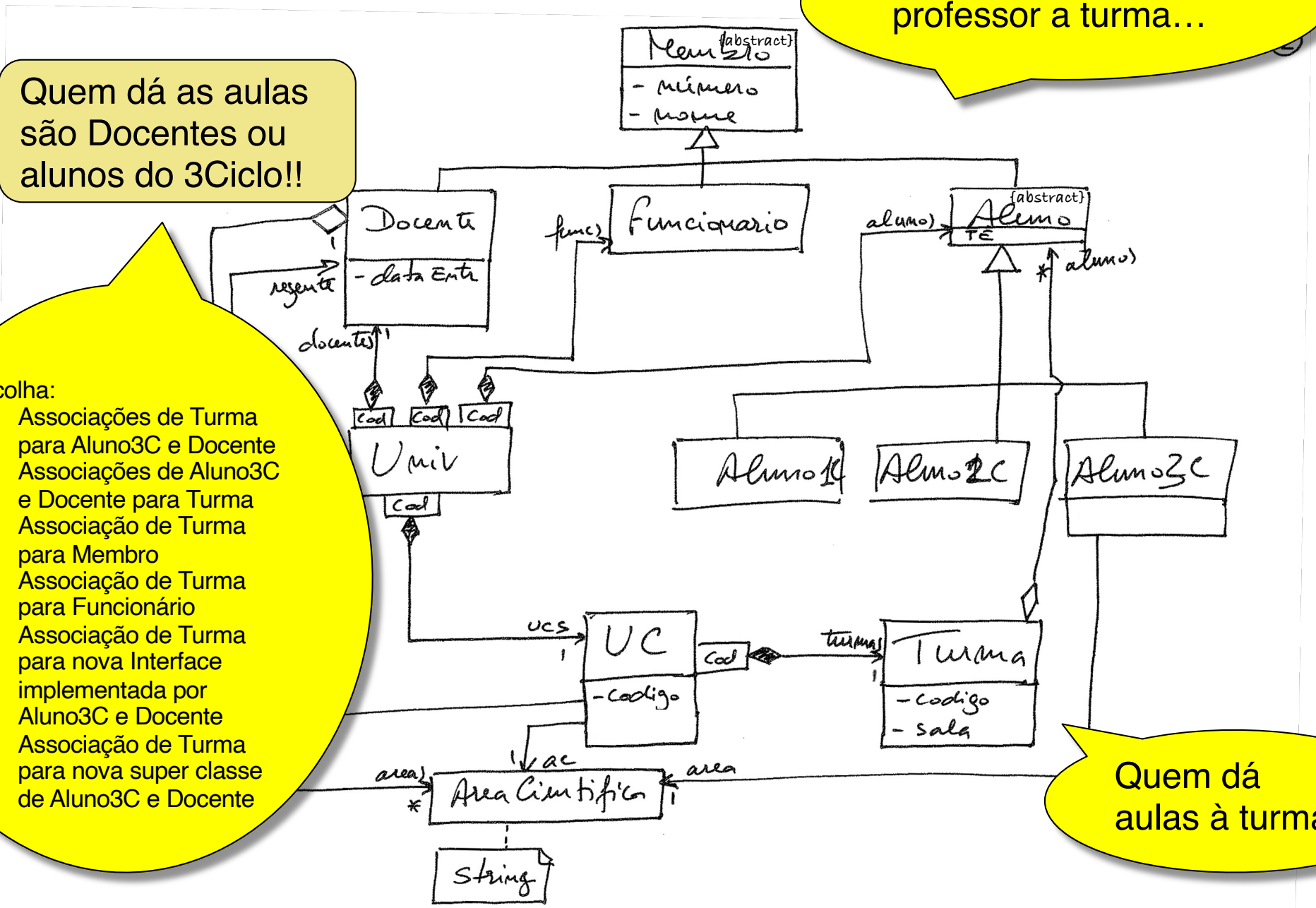
Exemplo - Um Modelo de Domínio...



Arquitectura parcial…

Use Case atribuir professor a turma…

Quem dá as aulas são Docentes ou alunos do 3Ciclo!!

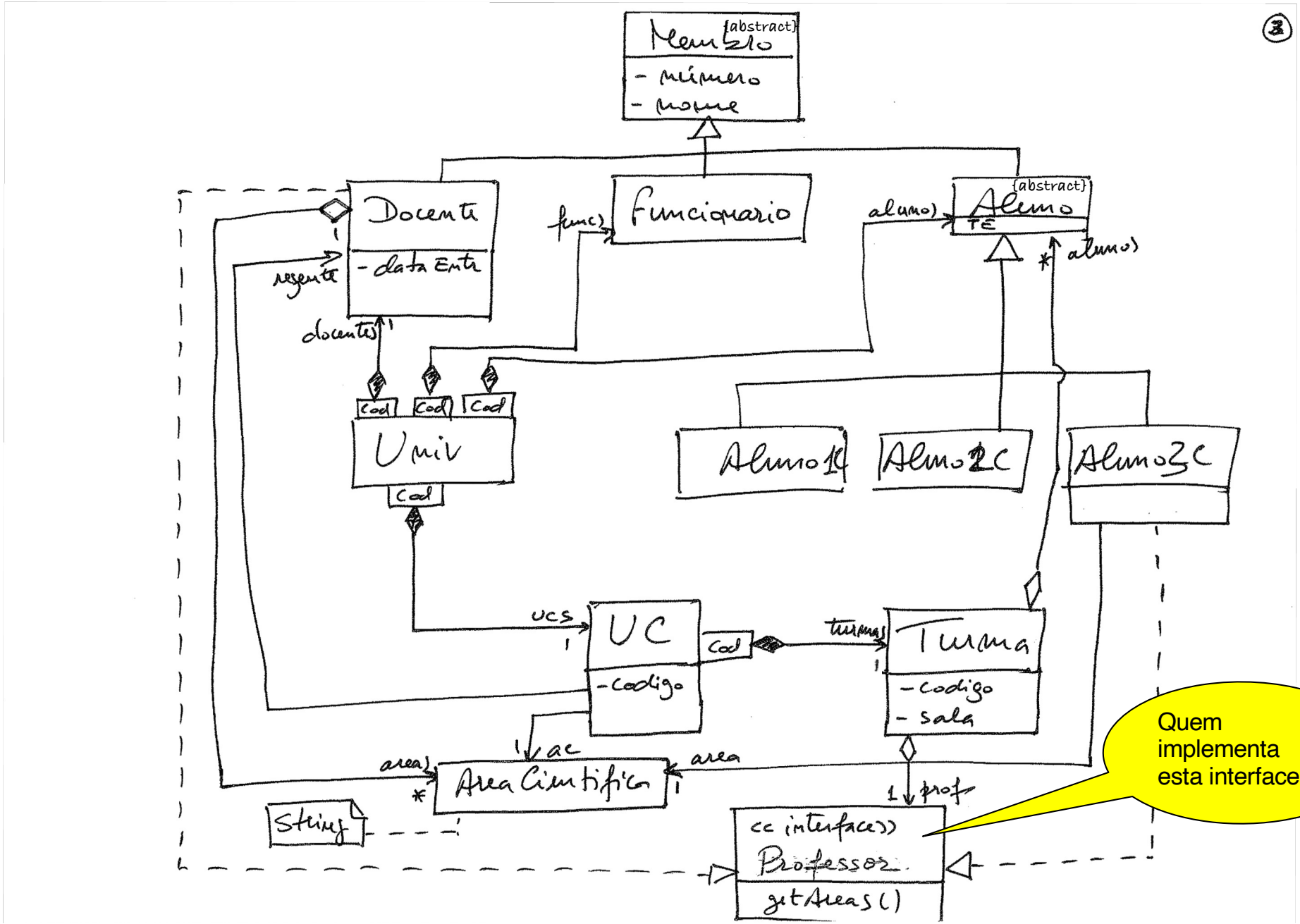


Escolha:

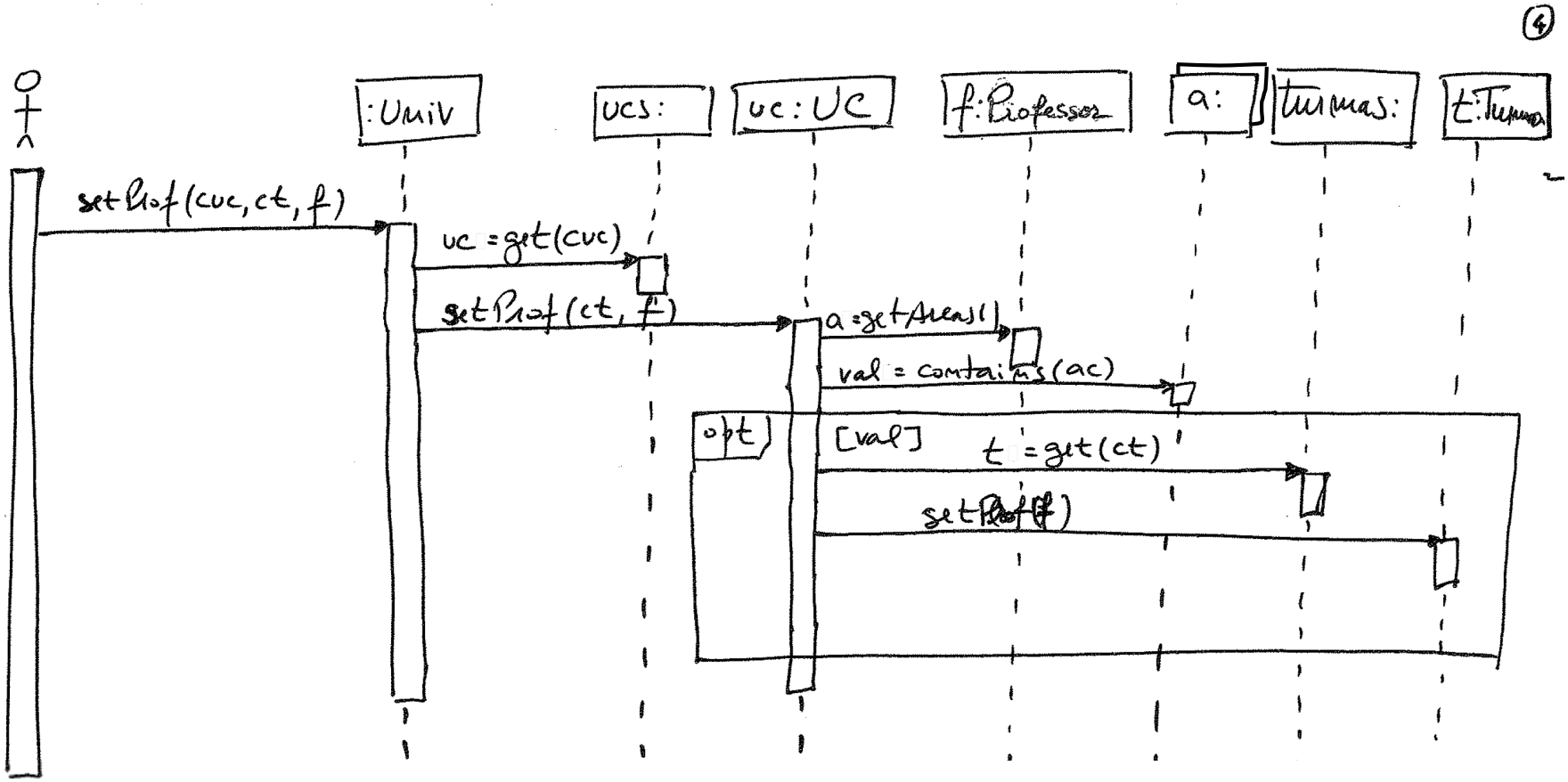
a) Associações de Turma para Aluno3C e Docente
b) Associações de Aluno3C e Docente para Turma
c) Associação de Turma para Membro
d) Associação de Turma para Funcionário
e) Associação de Turma para nova Interface implementada por Aluno3C e Docente
f) Associação de Turma para nova super classe de Aluno3C e Docente

Quem dá aulas à turma?

Arquitectura parcial...

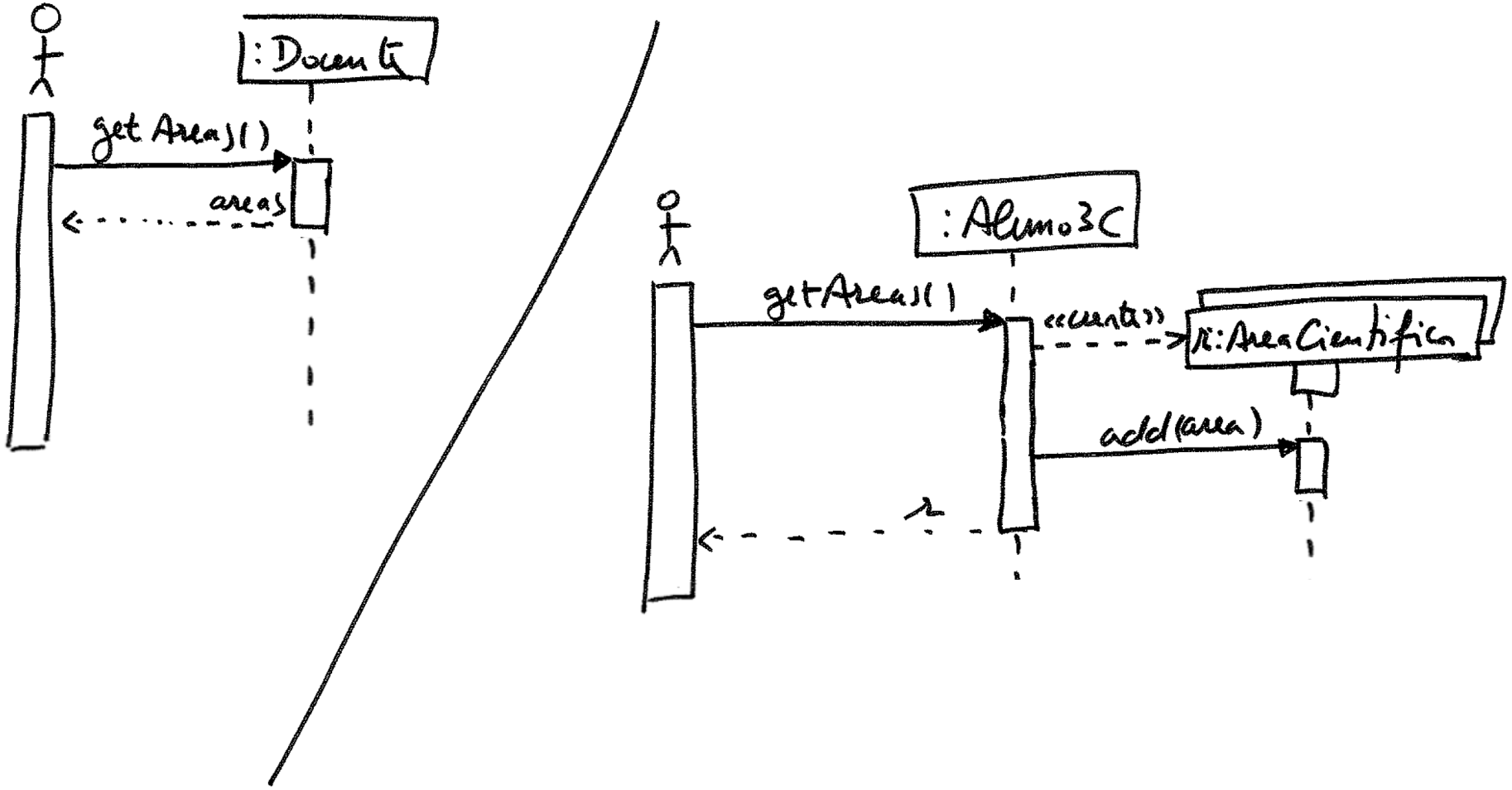


O comportamento...

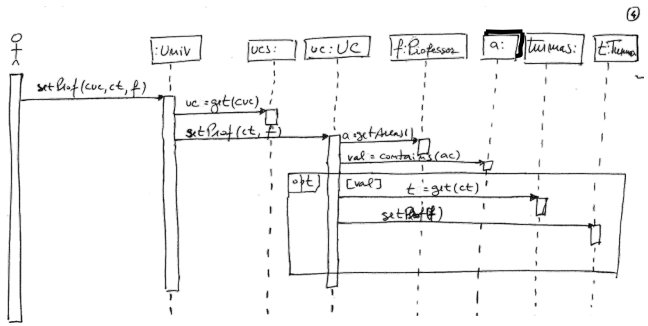


O comportamento…

getAreas() depende
da classe concreta…



O código...




```java
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);
    }

}
```

```java
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);
    }
}
```

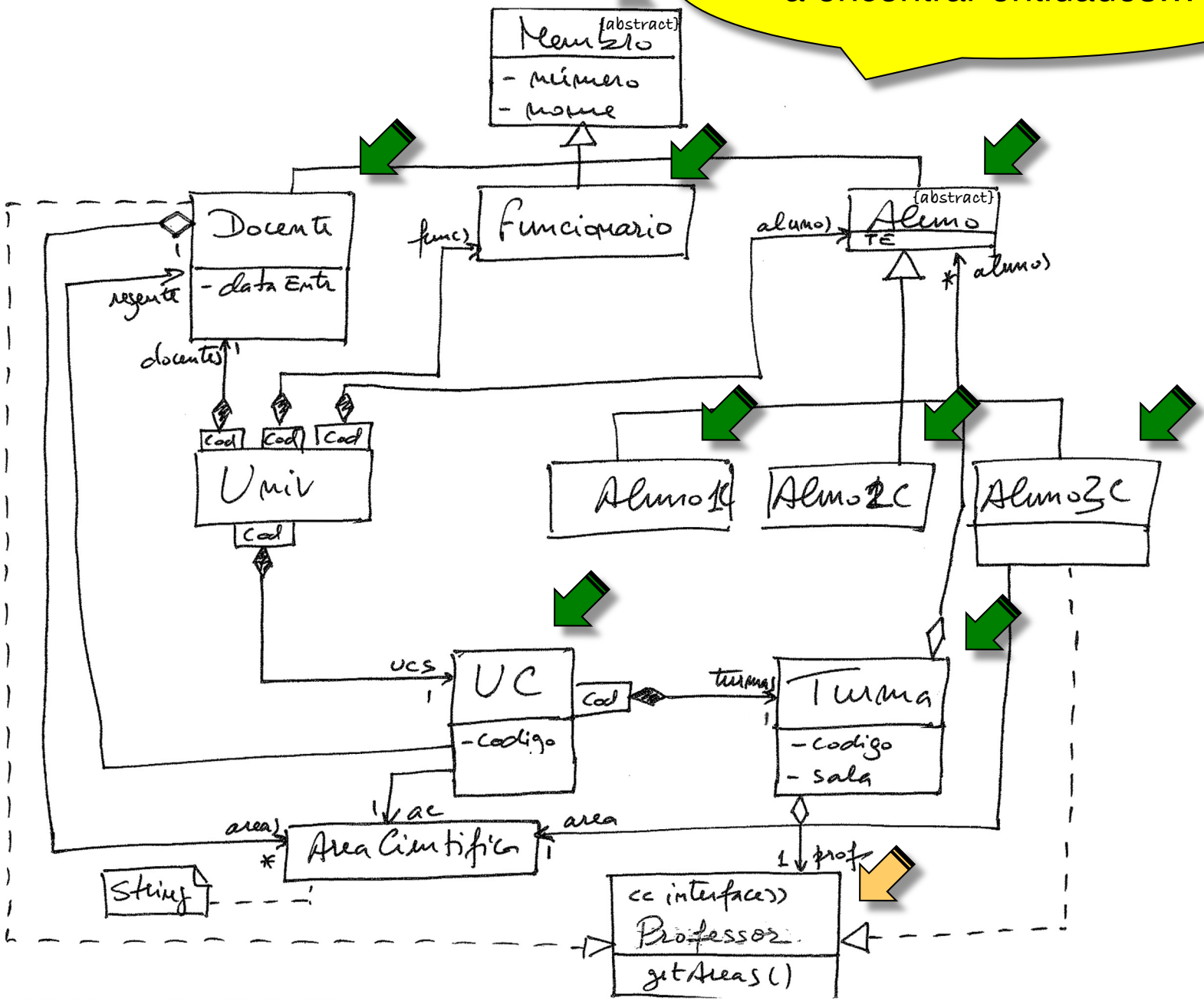
O código…


```java
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);
    }

}
```

```java
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);
    }
}
```

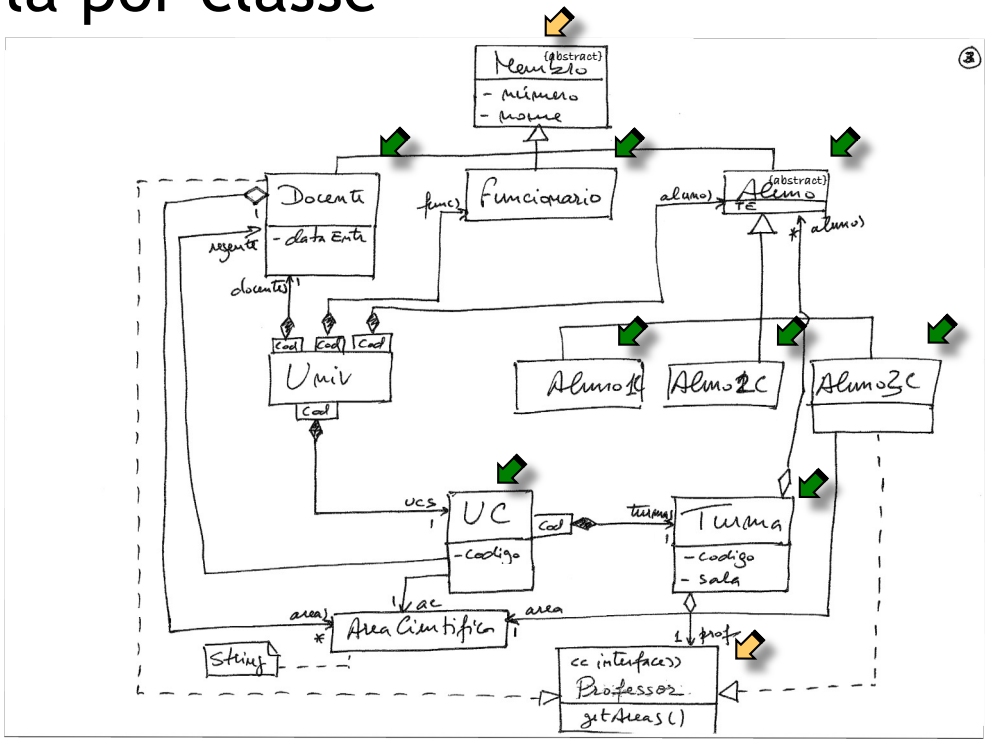
Que entidades persistir?

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"	'3'	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)
Funcionario(nº, nome)
Aluno1C(nº, nome, te)
Aluno2C(nº, nome, te)
Aluno3C(nº, nome, te, area)
```

- 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

Estratégia mais comum - a que vamos adoptar!

Mapeamento de uma tabela por classe

- uma tabela para cada classe da hierarquia – 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)`

- 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)

Aluno2C(nº)

Aluno3C(nº)

UC(codigo)

Turma(codigo, sala)

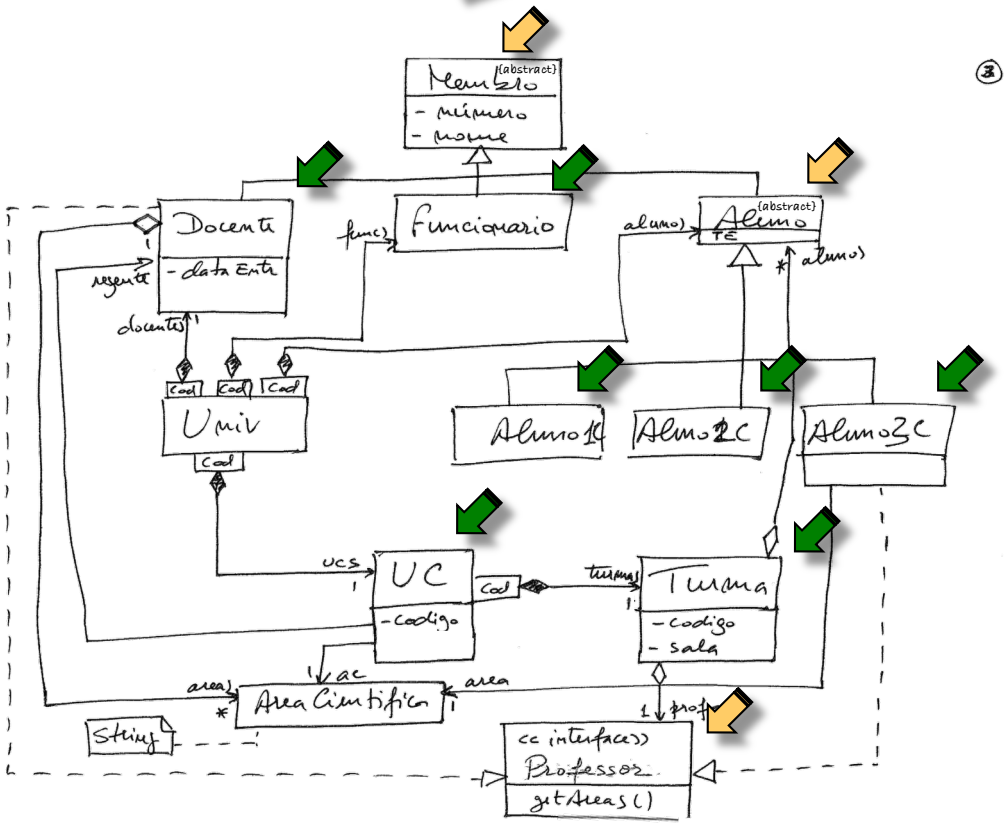
Professor(obj_id, tipo, nº)

AreaCientifica(obj_id, descr)

Como representar as associações

?

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



Mapeamento de associações

- Associações um-para-um
- Associações um-para-muitos,
- Associações muitos-para-muitos

Mapeamento de associações

- 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 colocação da chave estrangeira (navegabilidade é sempre da tabela que possui a chave estrangeira para a tabela referenciada).
- Associações um-para-muitos,
- Associações muitos-para-muitos

Que tabelas?

Membro (nº, nome, tipo)

Docente(nº, dataEntr)

Funcionario(nº)

Aluno(nº, te)

Aluno1C(nº)

Aluno2C(nº)

Aluno3C(nº)

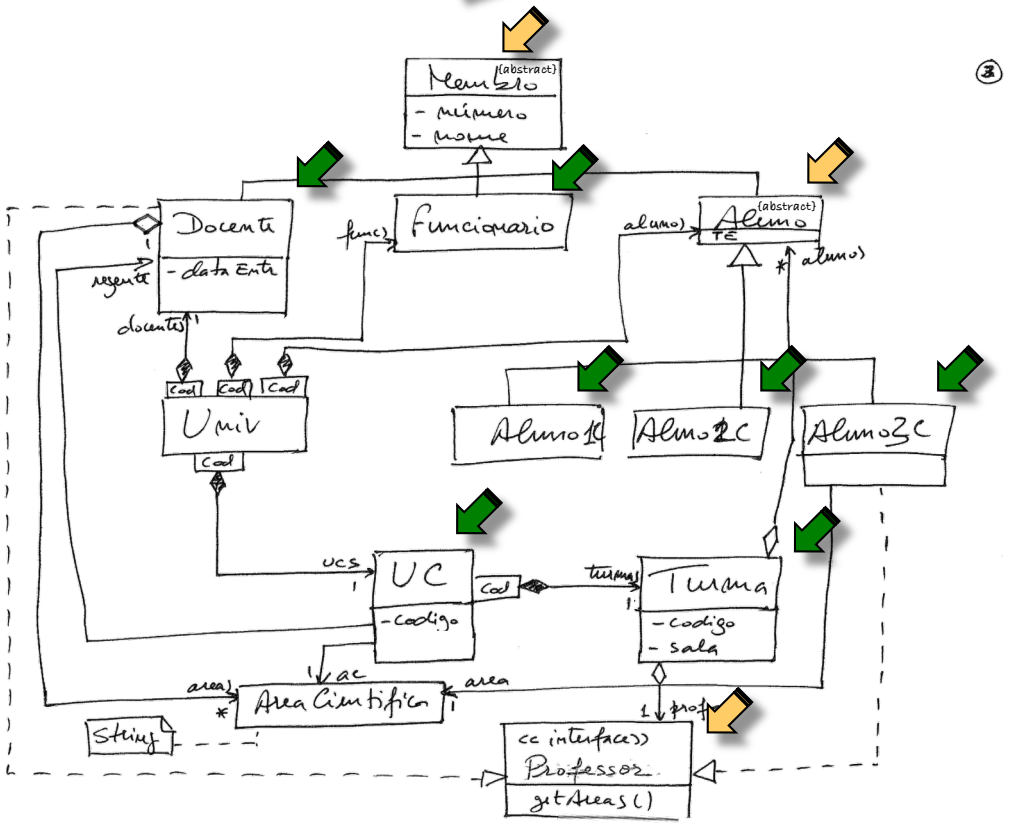
UC(codigo)

Turma(codigo, sala)

Professor(obj_id, tipo, nº)

AreaCientifica(obj_id, descr)

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



Que tabelas?

Membro (nº, nome, tipo)

Docente(nº, dataEntr)

Funcionario(nº)

Aluno(nº, te)

Aluno1C(nº)

Aluno2C(nº)

Aluno3C(nº, ***codac***)

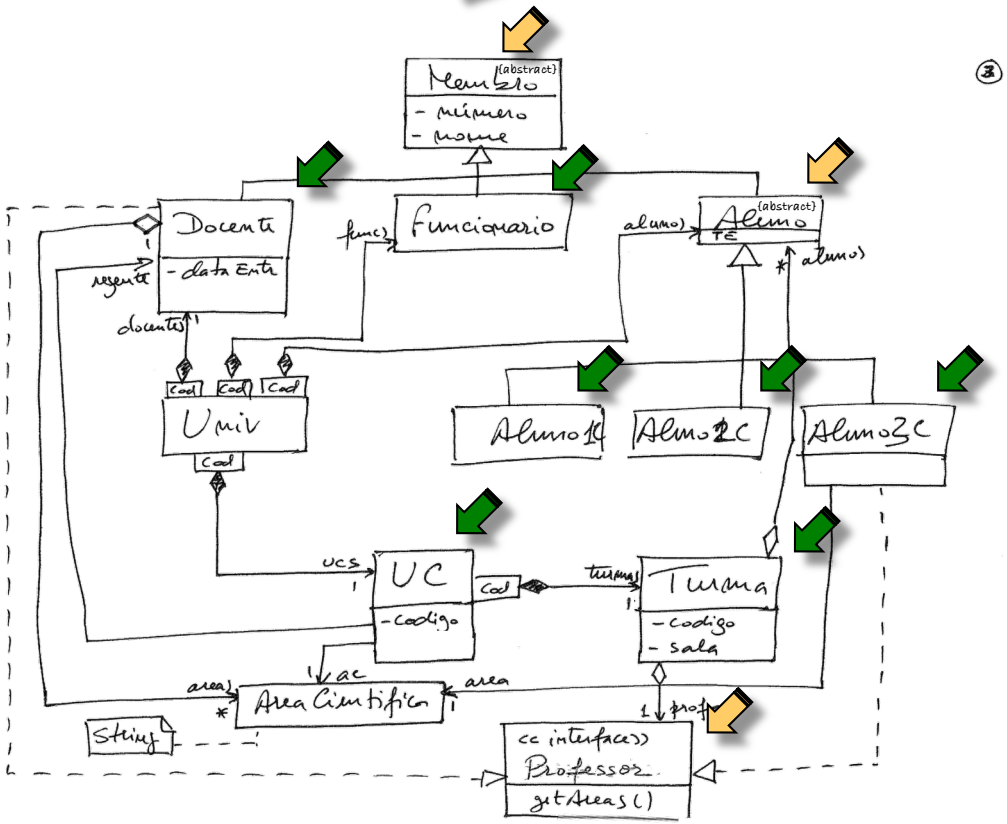
UC(codigo, ***codac***, ***regente***)

Turma(codigo, sala, ***prof***)

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 colocação da chave estrangeira (navegabilidade é sempre da tabela que possui a chave estrangeira para a tabela referenciada).
- Associações um-para-muitos,
- Associações muitos-para-muitos

Mapeamento de relacionamentos

- Associações um-para-um
- Associações um-para-muitos,
- Associações muitos-para-muitos

Mapeamento de relacionamentos

- Associações um-para-um
- 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

Que tabelas?

Membro (nº, nome, tipo)

Docente(nº, dataEntr)

Funcionario(nº)

Aluno(nº, te)

Aluno1C(nº)

Aluno2C(nº)

Aluno3C(nº, codac)

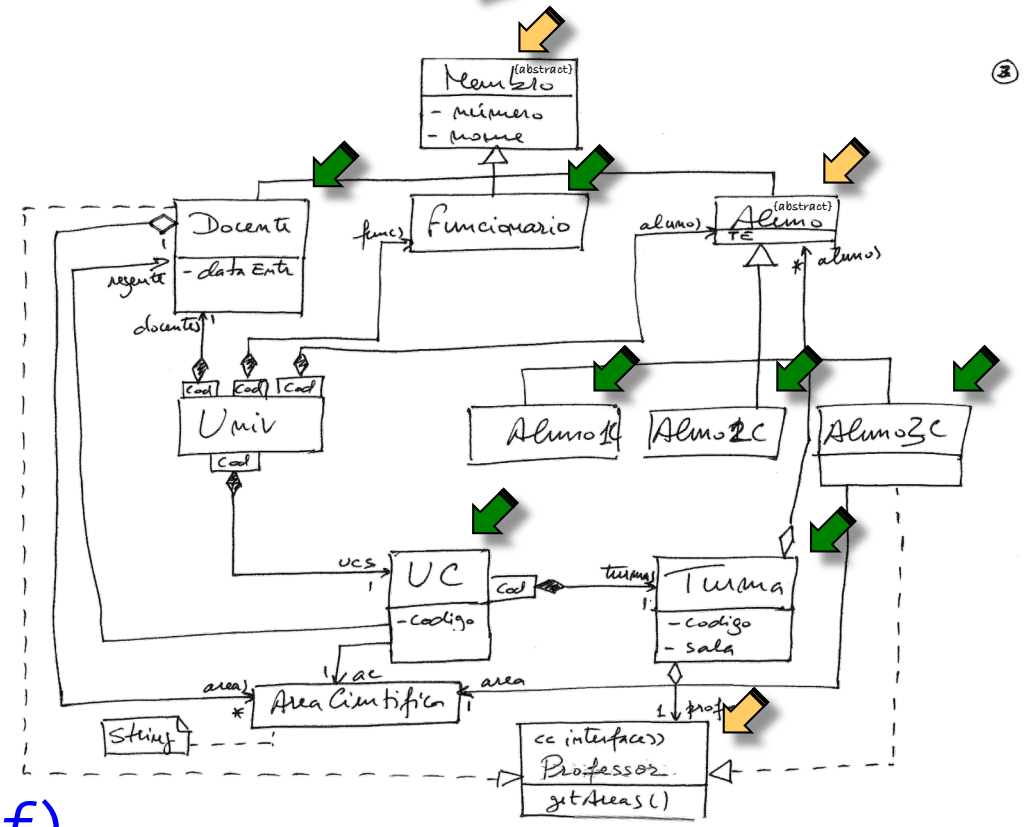
UC(codigo, codac, regente)

Turma(codigo, **uc**, sala, prof)

Professor(obj_id, tipo, nº)

AreaCientifica(obj_id, descr)

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



Mapeamento de relacionamentos

- Associações um-para-um
- Associações um-para-muitos,
- Associações muitos-para-muitos

Mapeamento de relacionamentos

- Associações um-para-um
- Associações um-para-muitos,
- 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)

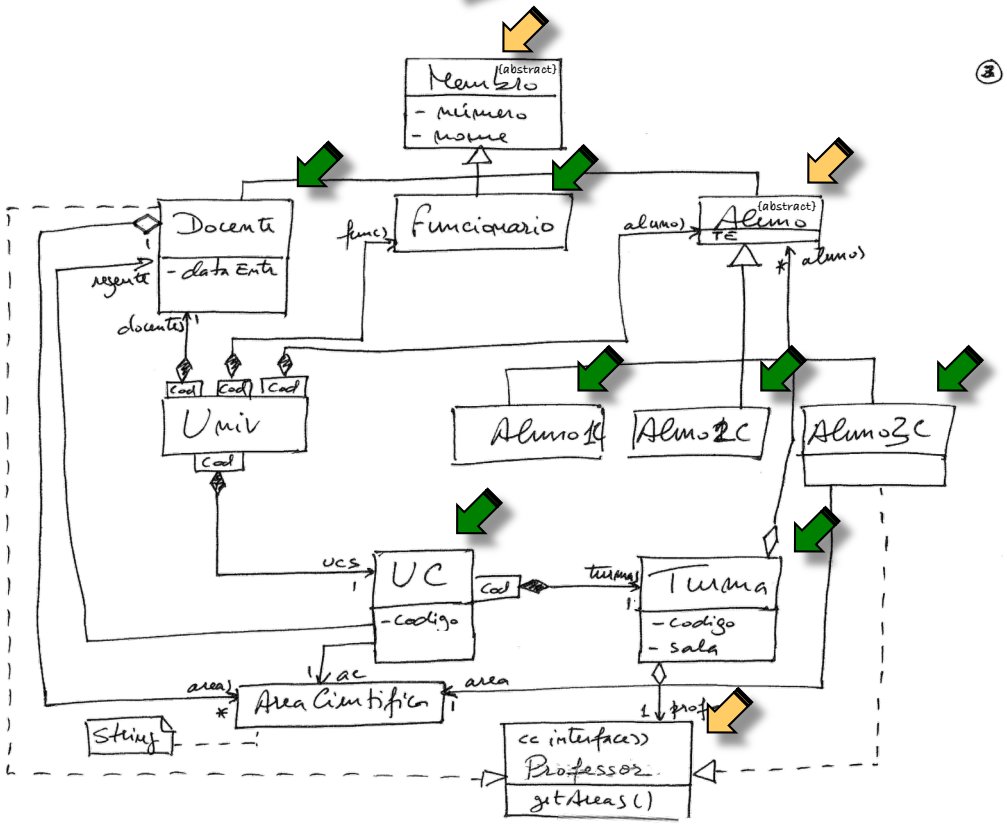
Professor(obj_id, tipo, nº)

AreaCientifica(obj_id, descr)

DocenteAreas(nº, codac)

TurmaAlunos(codigo, nº)

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



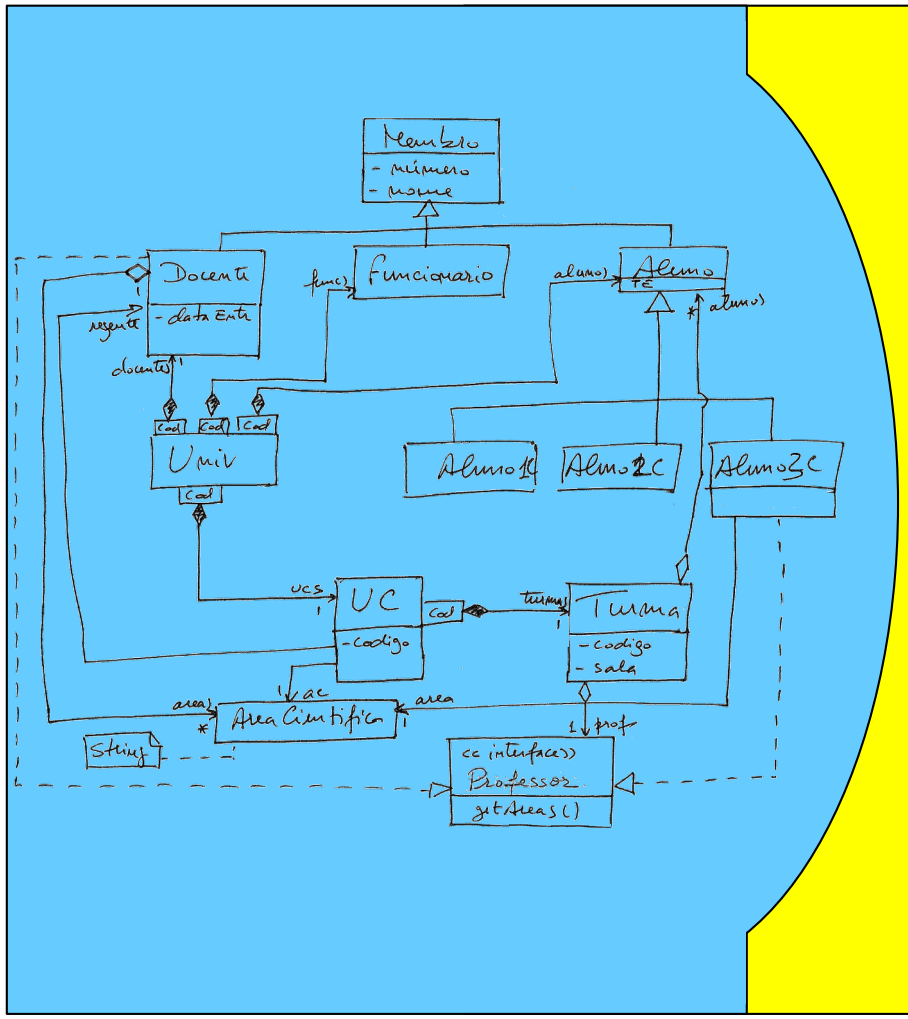
Regras de mapeamento

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” (“n 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



ORM?

Membro (<u>nº</u>, nome, tipo)

Docente(<u>nº</u>, dataEntr)

Funcionario(<u>nº</u>)

Aluno(<u>nº</u>, te)

Aluno1C(<u>nº</u>)

Aluno2C(<u>nº</u>)

Aluno3C(<u>nº</u>, <u>codac</u>)

UC(<u>codigo</u>, ac, <u>regente</u>)

Professor(<u>obj_id</u>, tipo, <u>nº</u>)

Turma(<u>codigo</u>, <u>uc</u>, sala, <u>prof</u>)

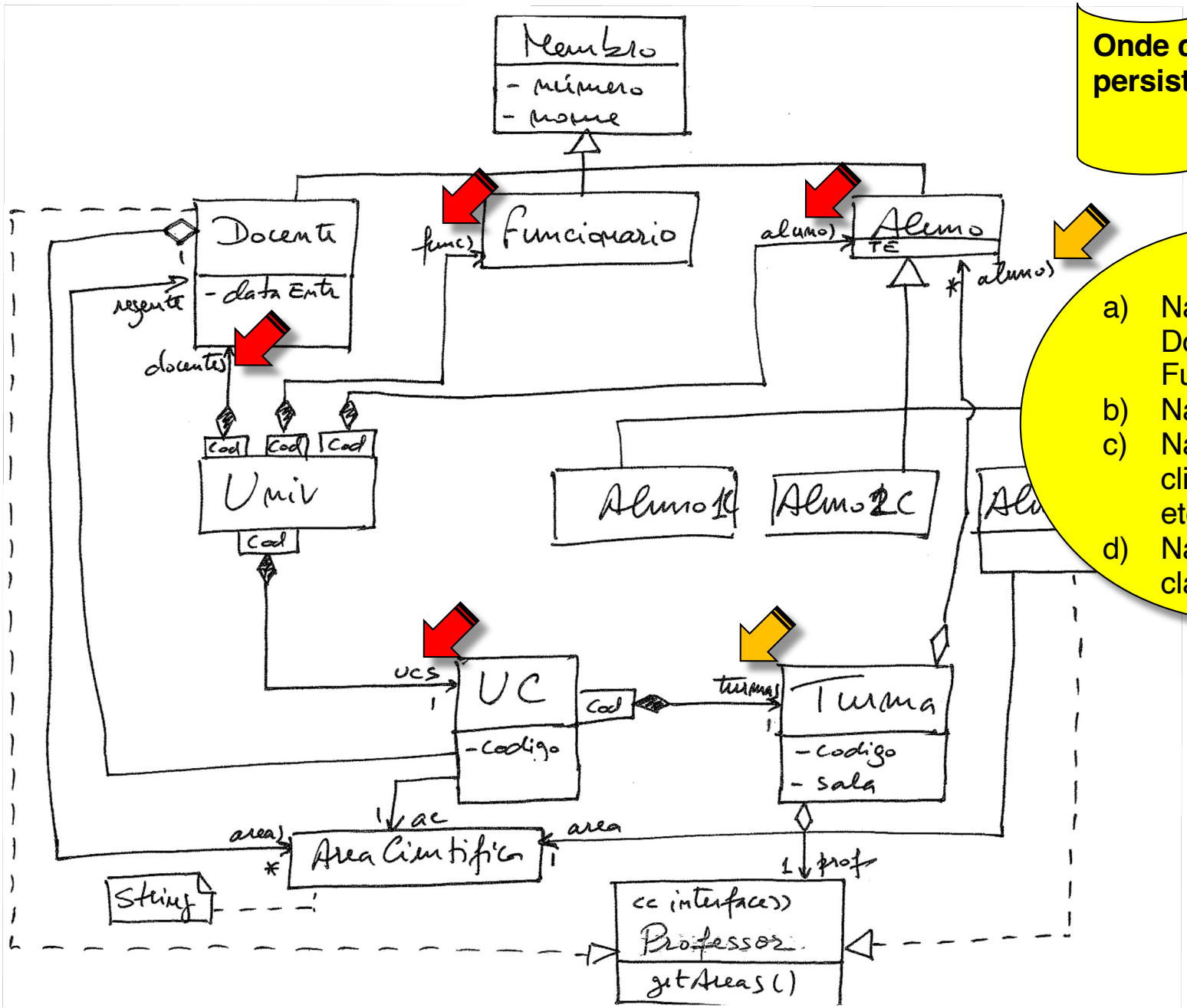
TurmaAlunos(<u>turma</u>, <u>aluno</u>)

AreasCientificas(<u>codigo</u>, descrição)

DocenteAreas(<u>nº</u>,<u>codac</u>)

TurmaAlunos(<u>codigo</u>, <u>nº</u>)

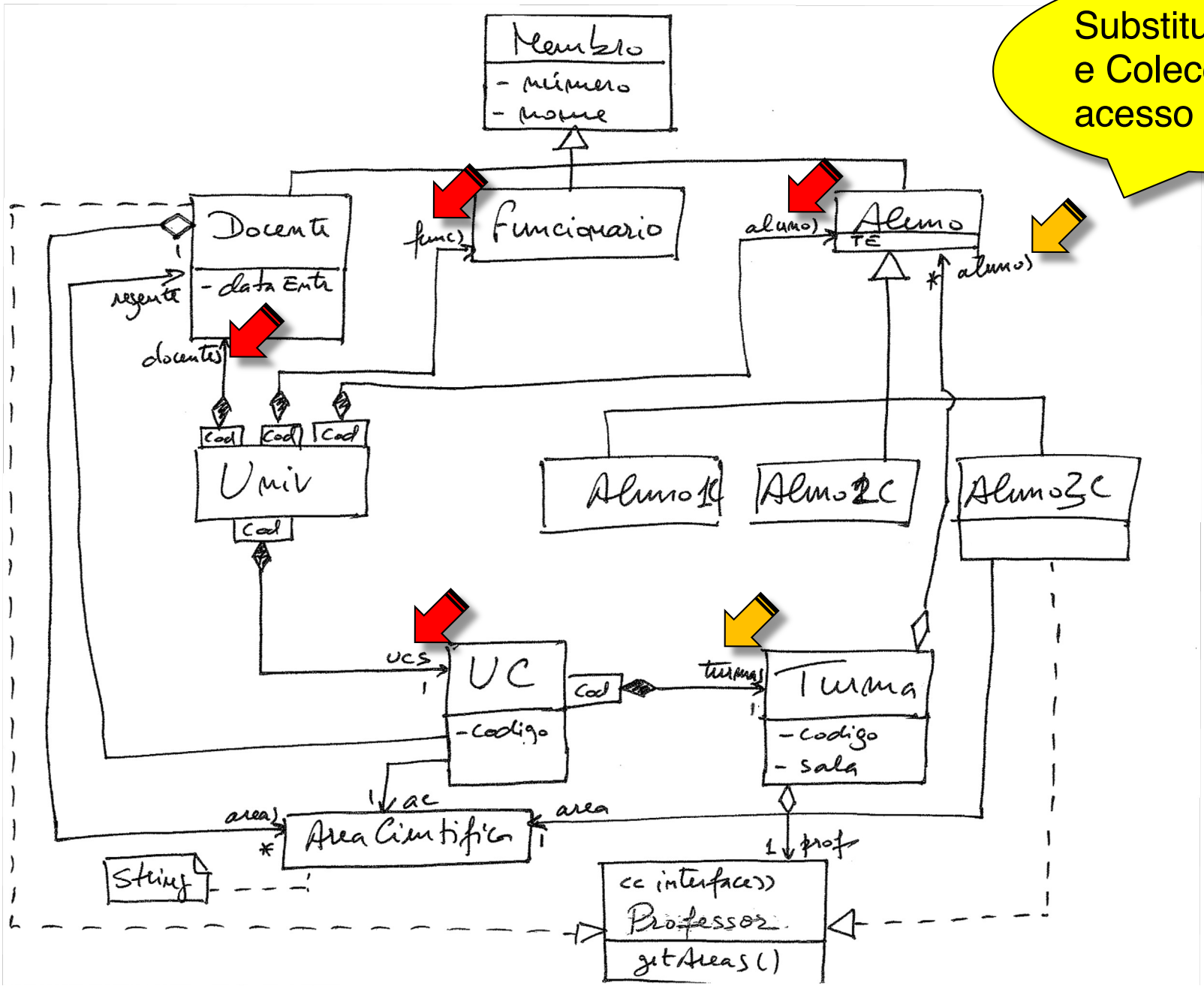
Arquitectura - versão sem persistência



Onde colocar a persistência?

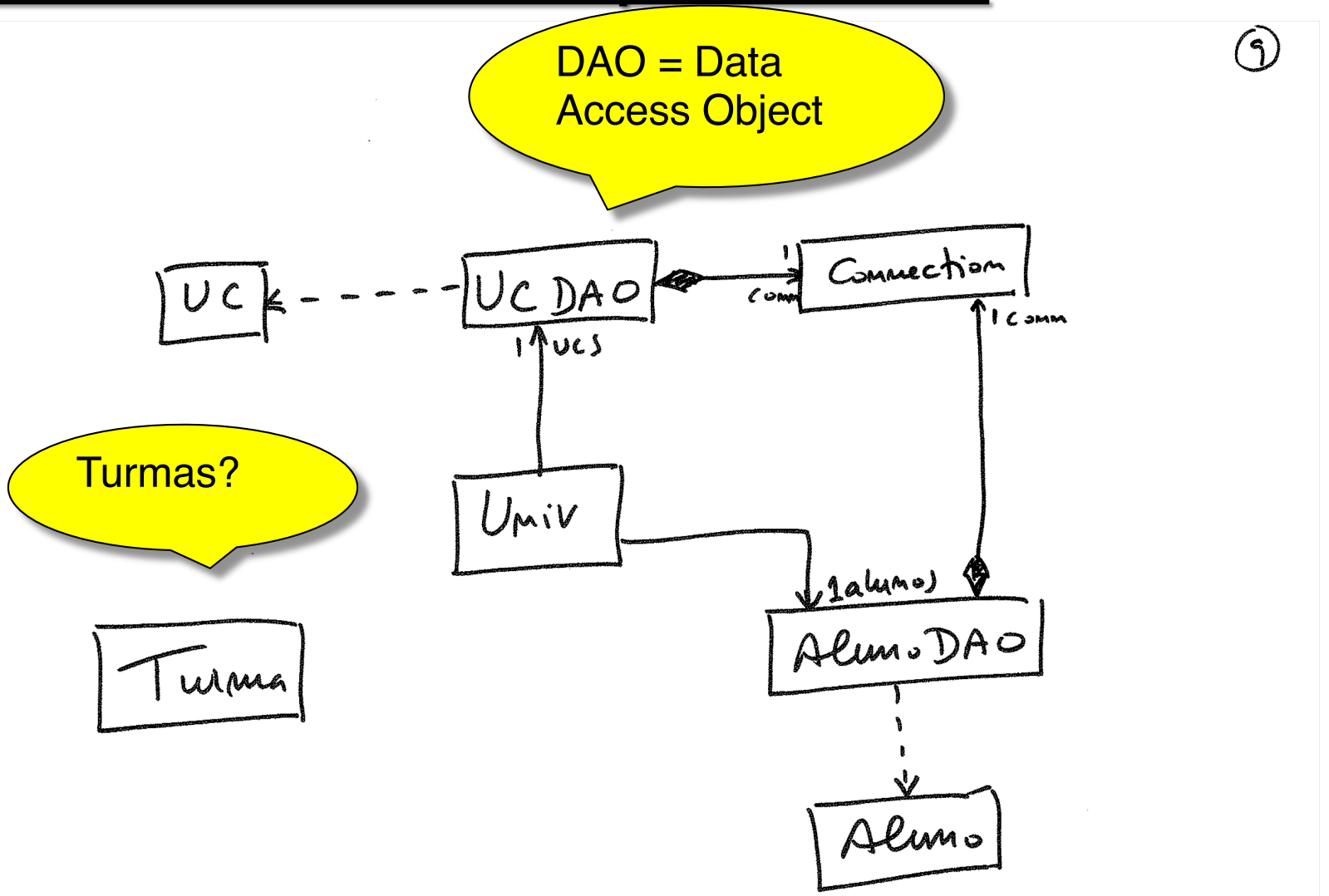
a) Nas classes Docente, Funcionario, etc.
b) Na classe Univ
c) Nas associações clientes, funcs, etc.
d) Na hierarquia de classes

Arquitectura - versão sem persistência

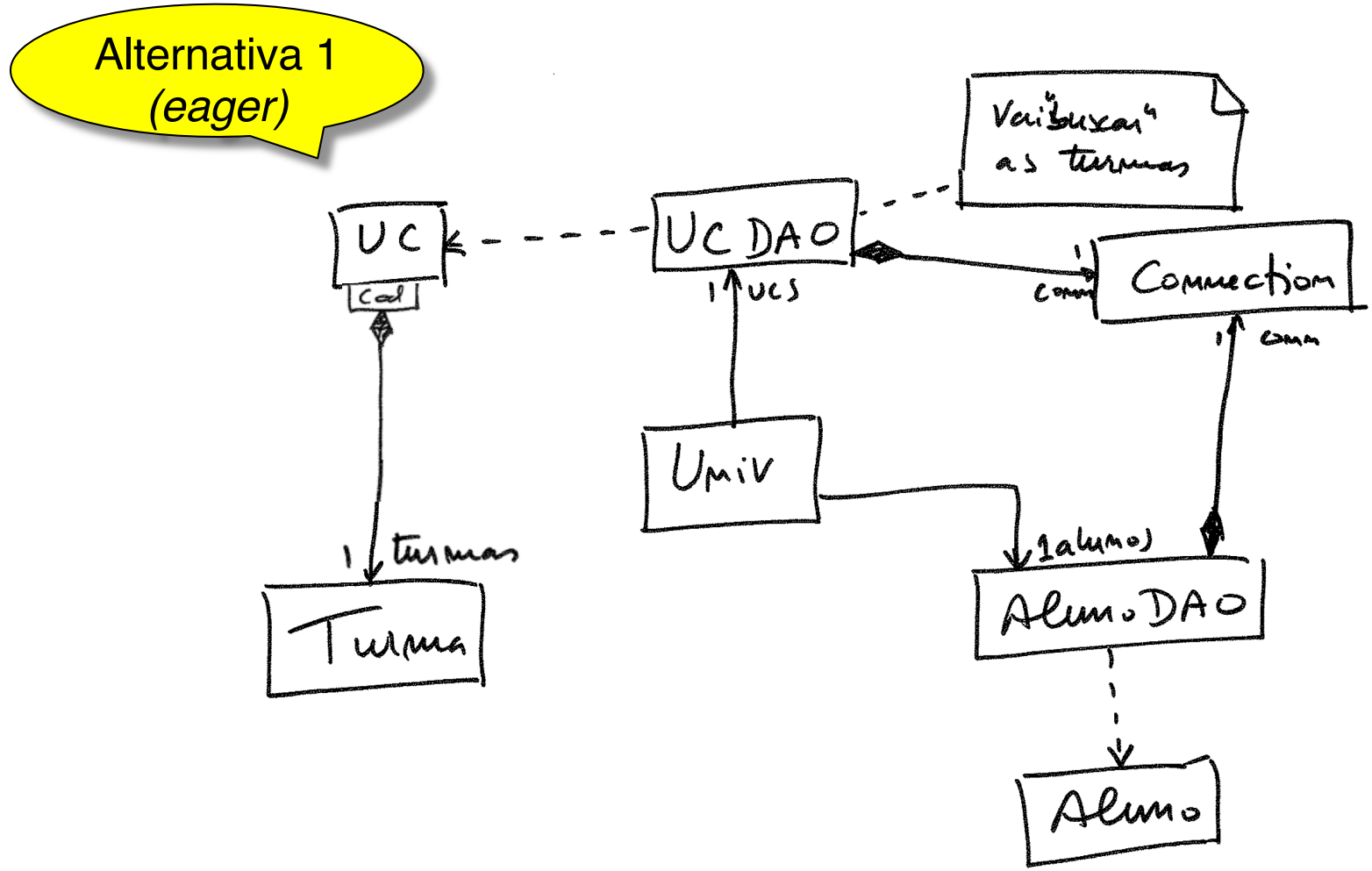


Substituir Maps e Colecções por acesso a BD.

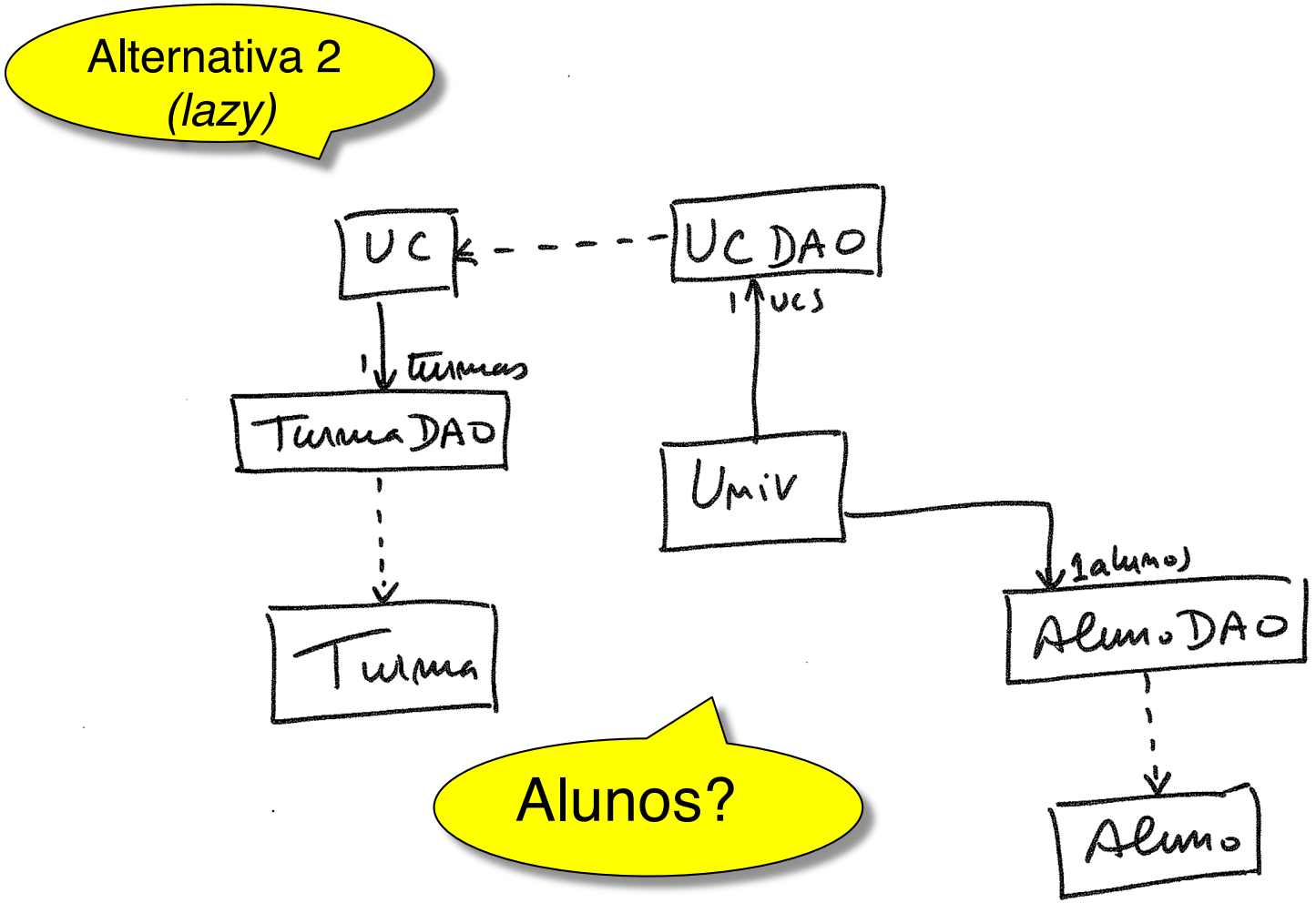
Arquitectura - versão com persistência



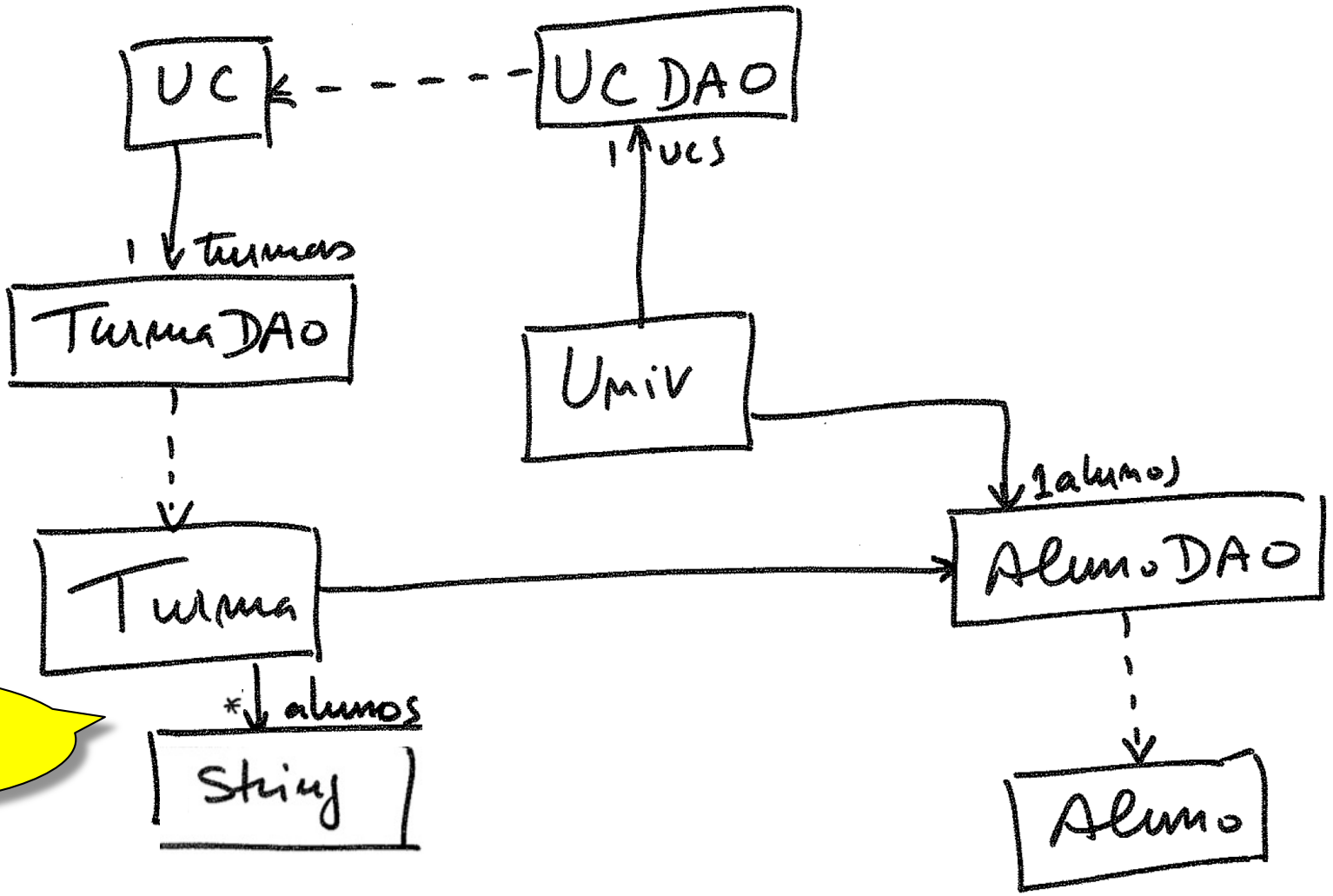
Arquitectura - versão com persistência



Arquitectura - versão com persistência

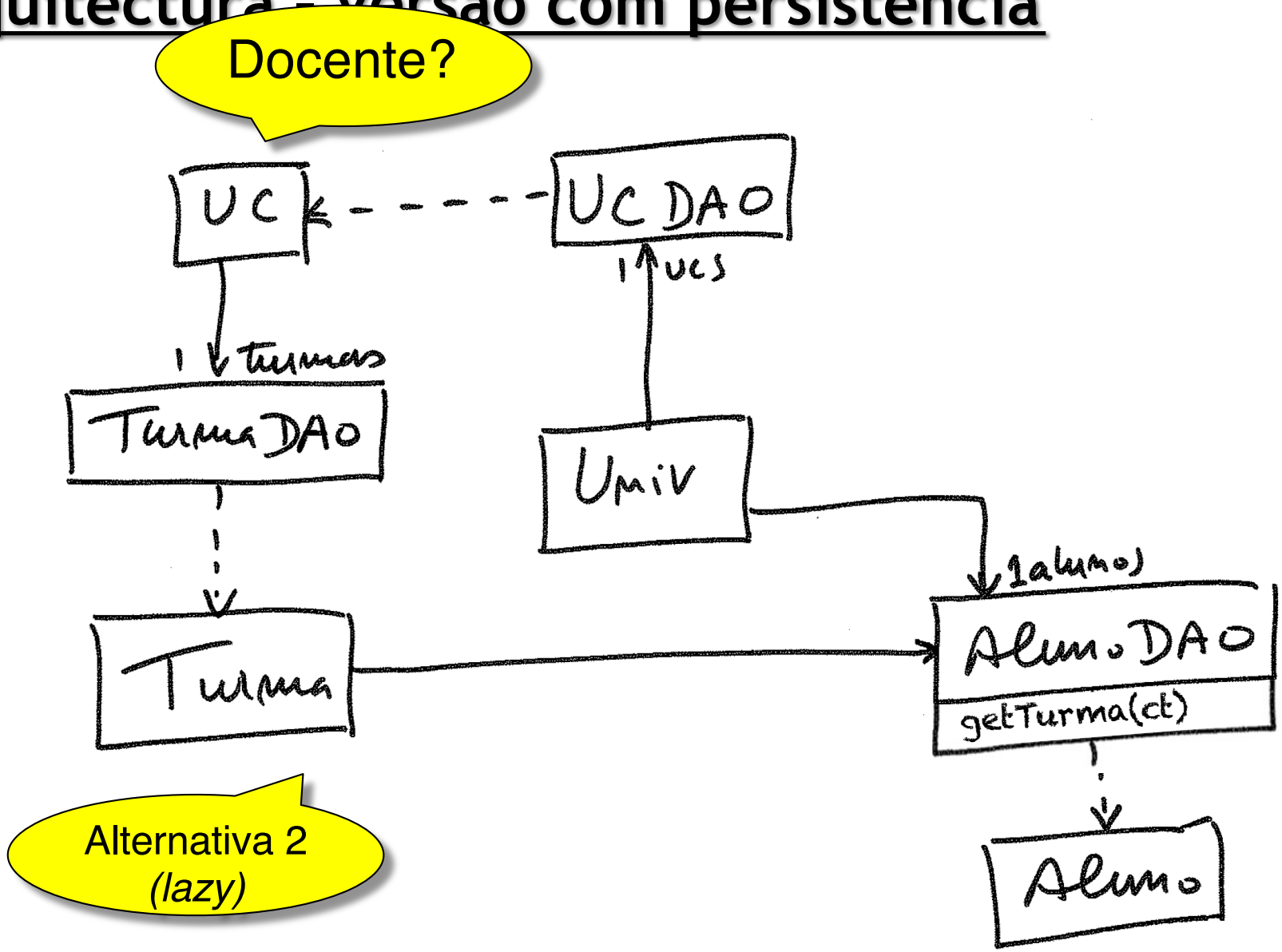


Arquitectura - versão com persistência



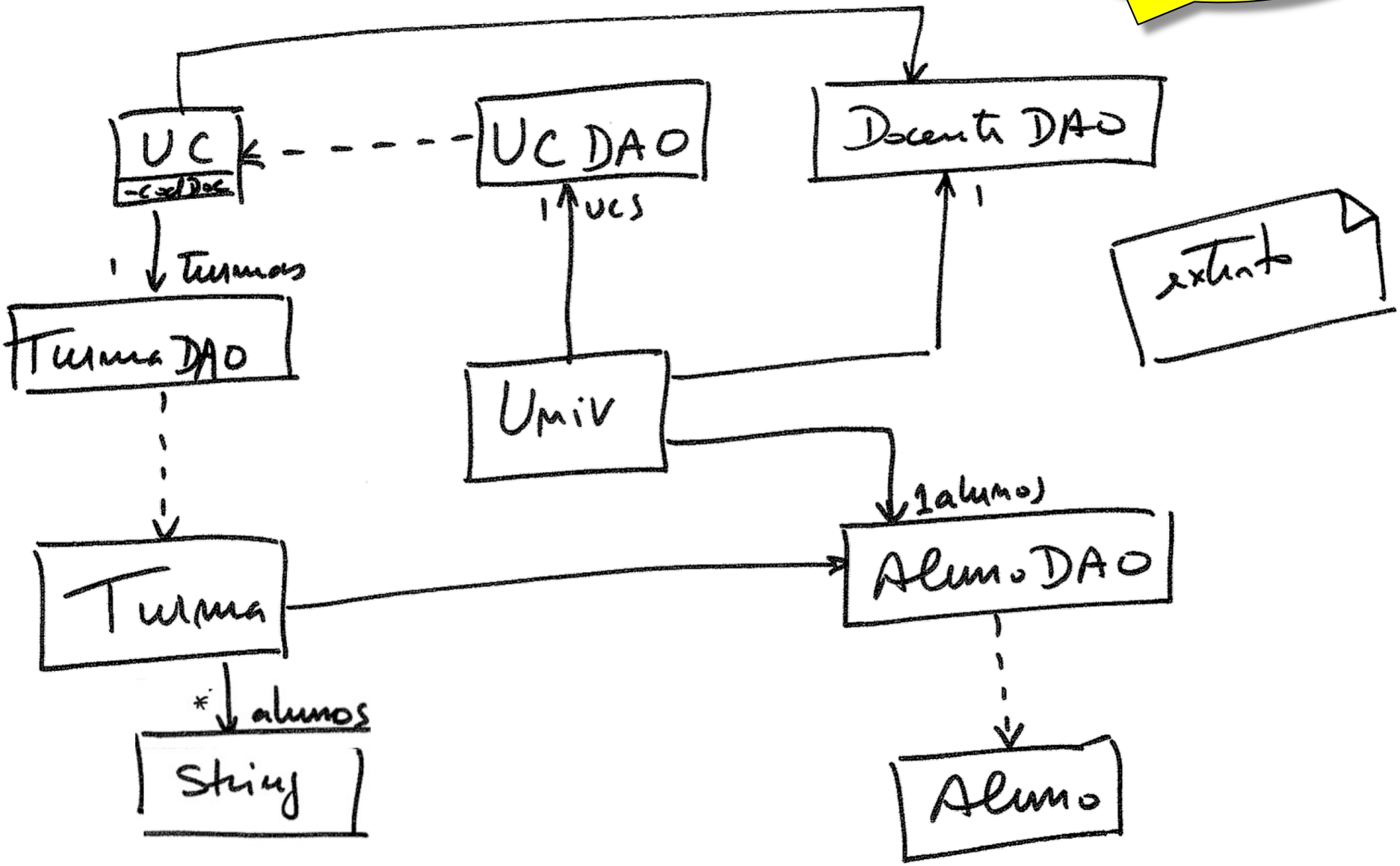
Alternativa 1
(lazy)

Arquitectura - versão com persistência

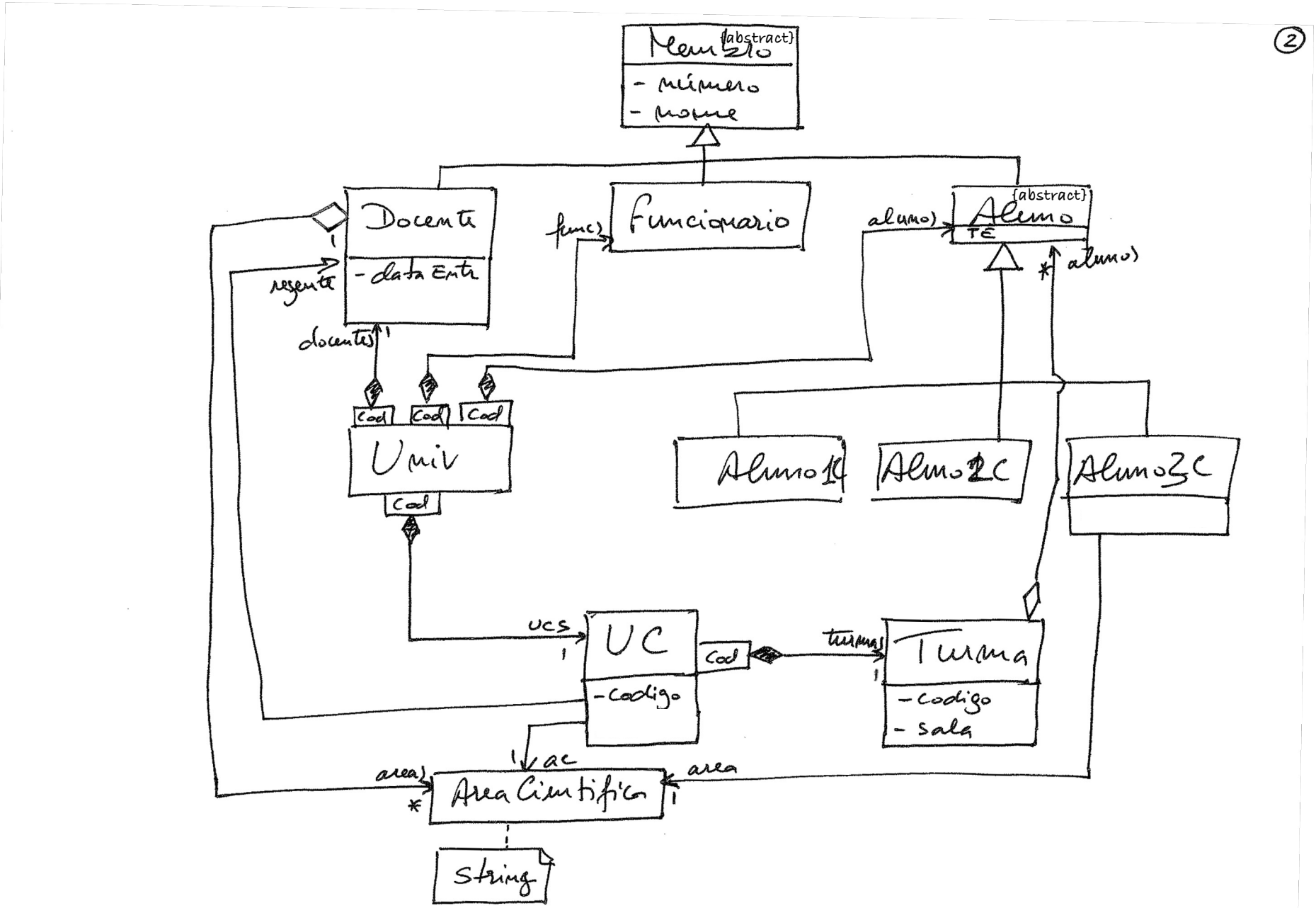


Arquitectura - versão com persistência

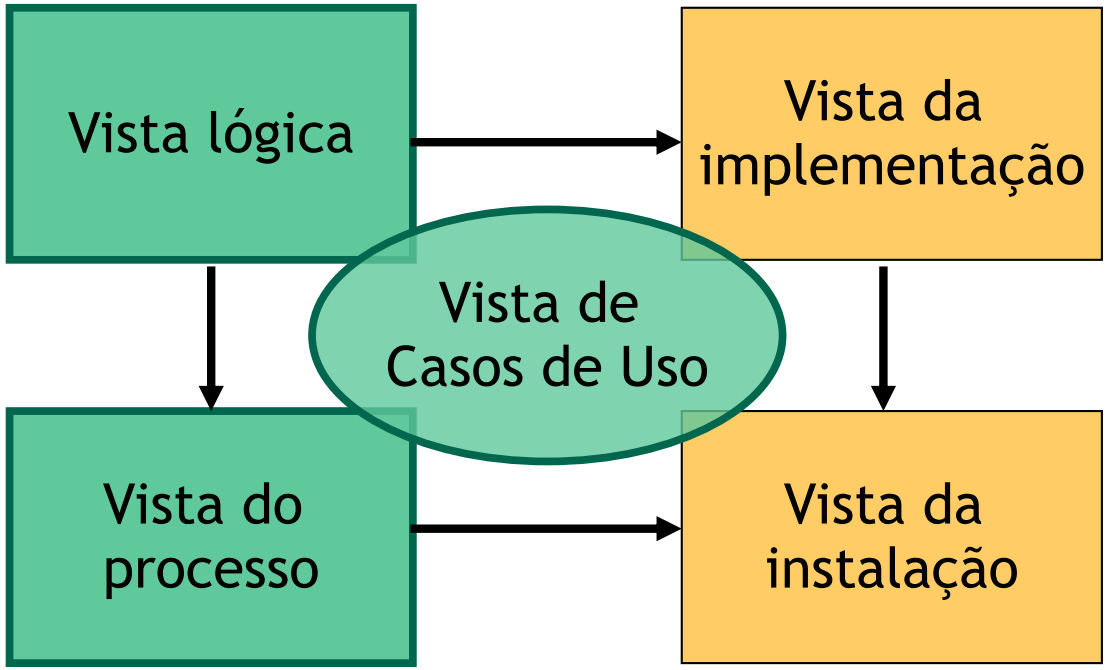
Falta completar com operações, etc.



Arquitectura parcial - nível lógico

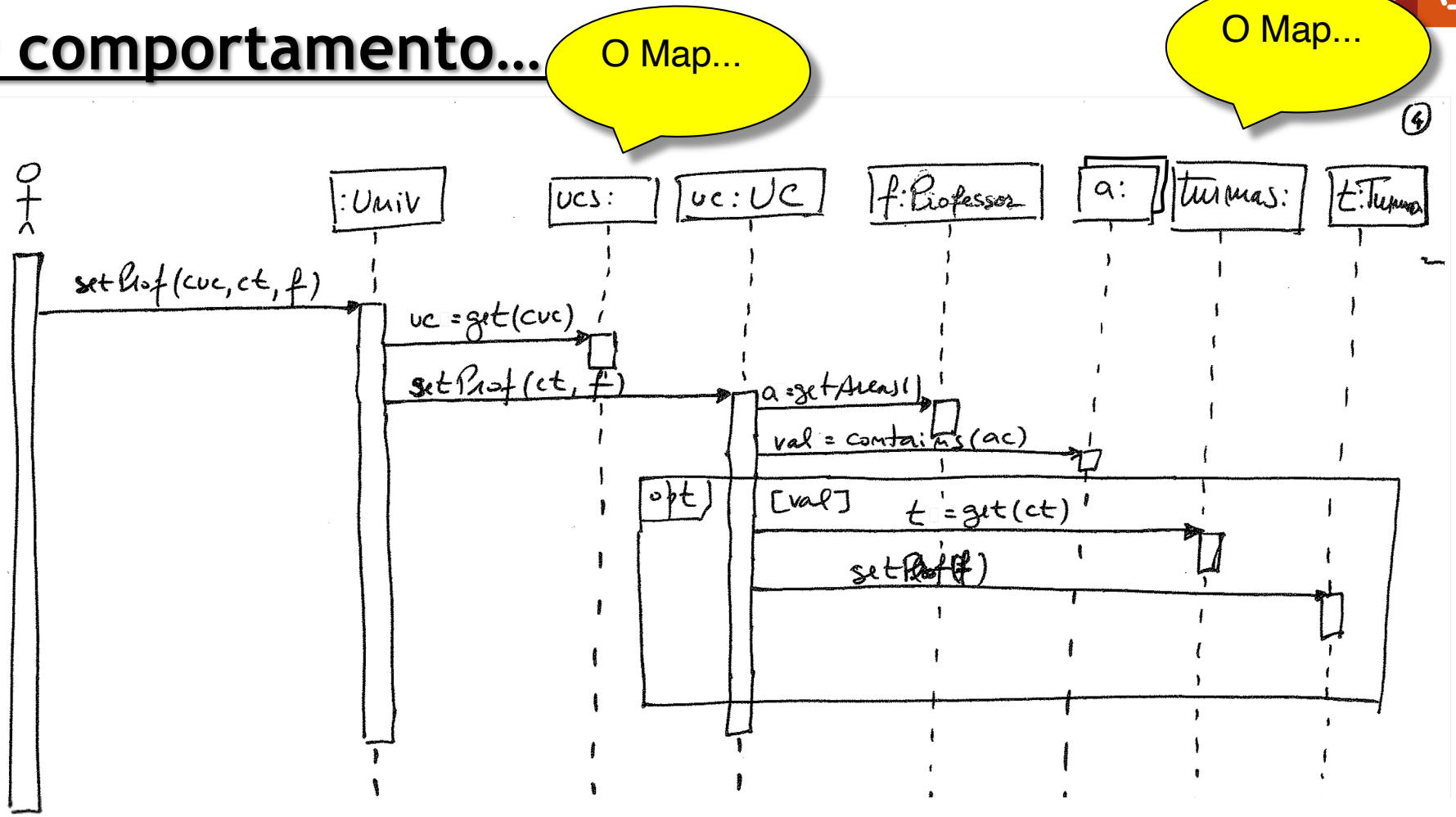


Veista lógica vs. Vista da implementação

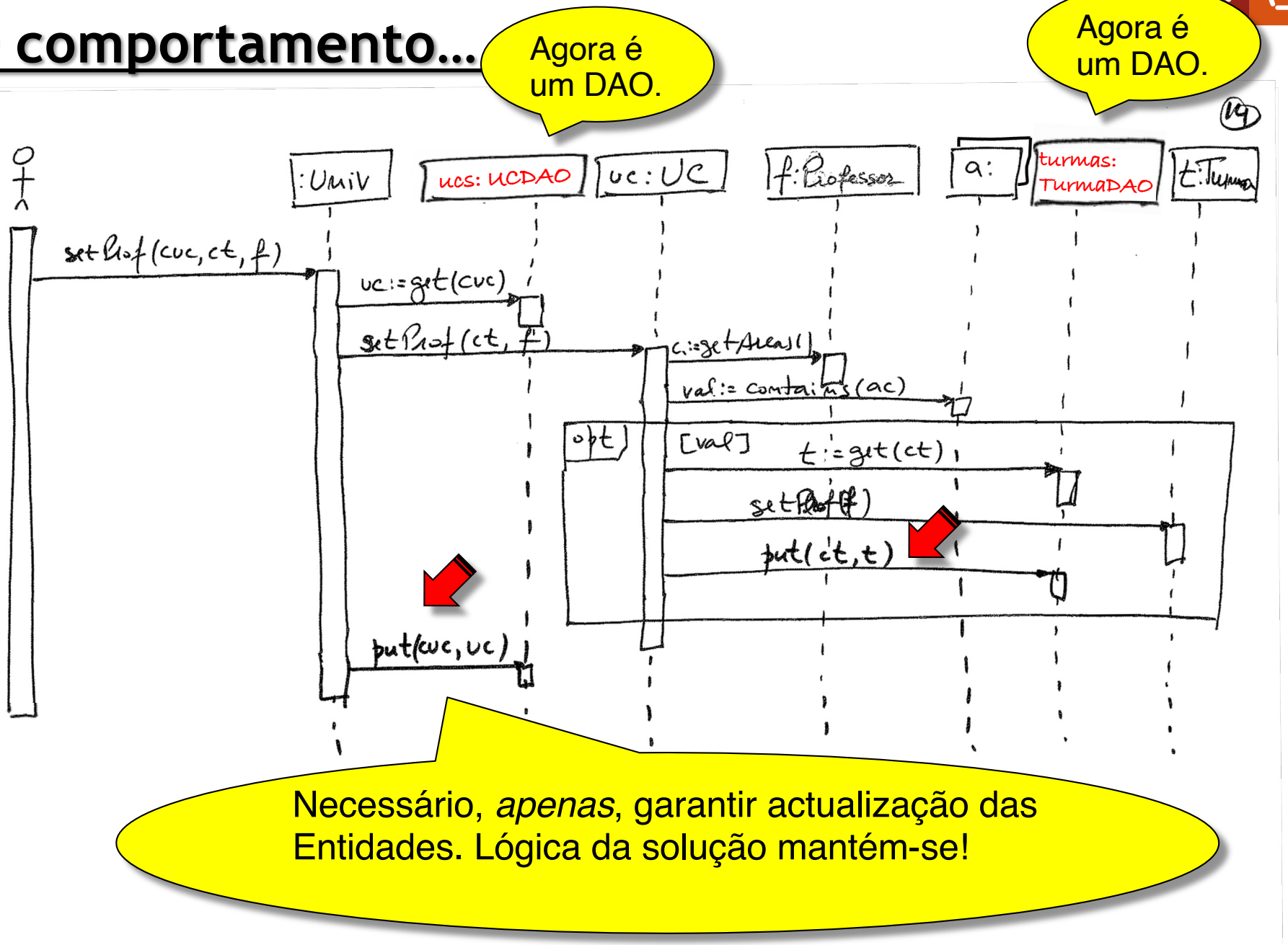


(Kruchten, 1995)

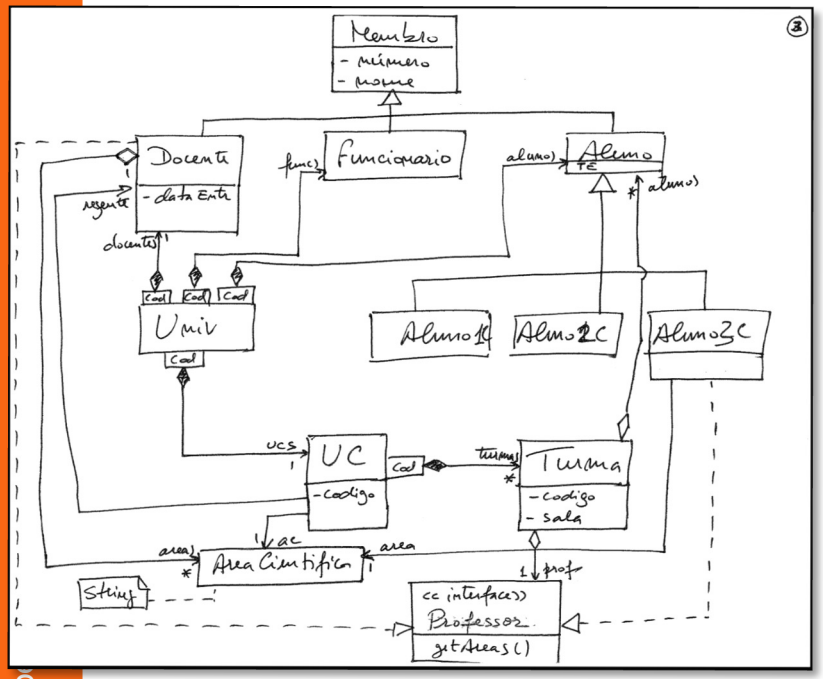
O comportamento...

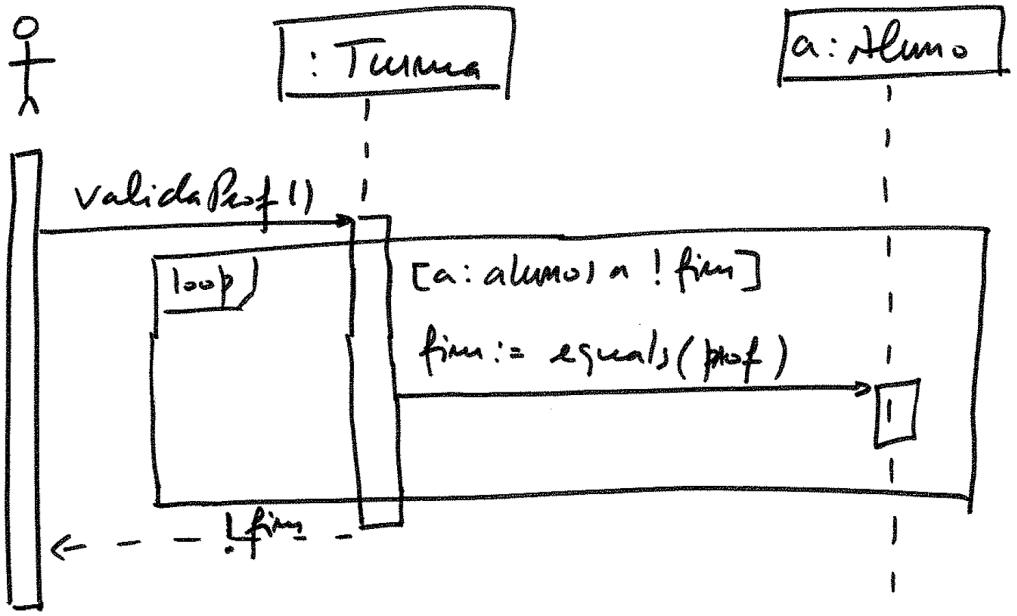


O comportamento...

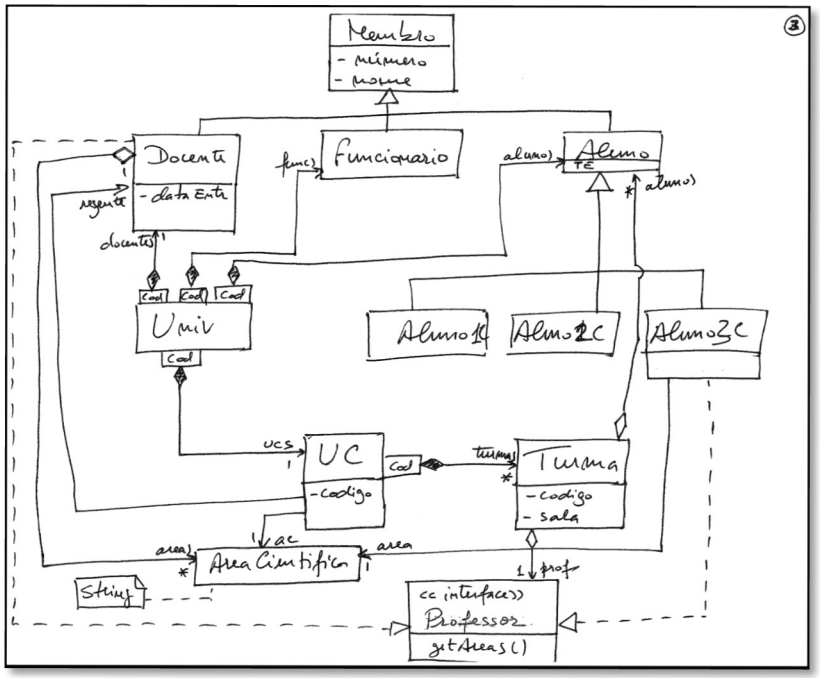


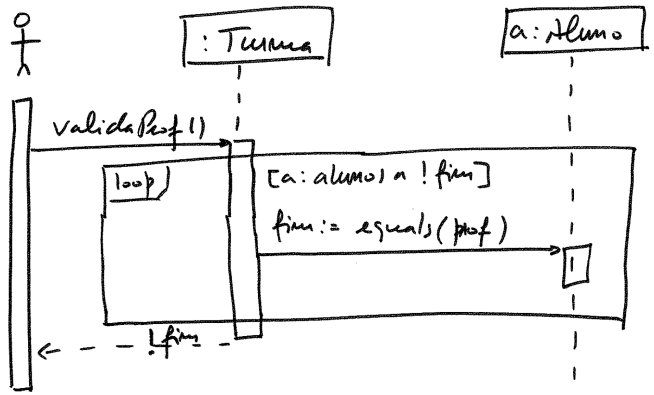
Outro Exemplo...



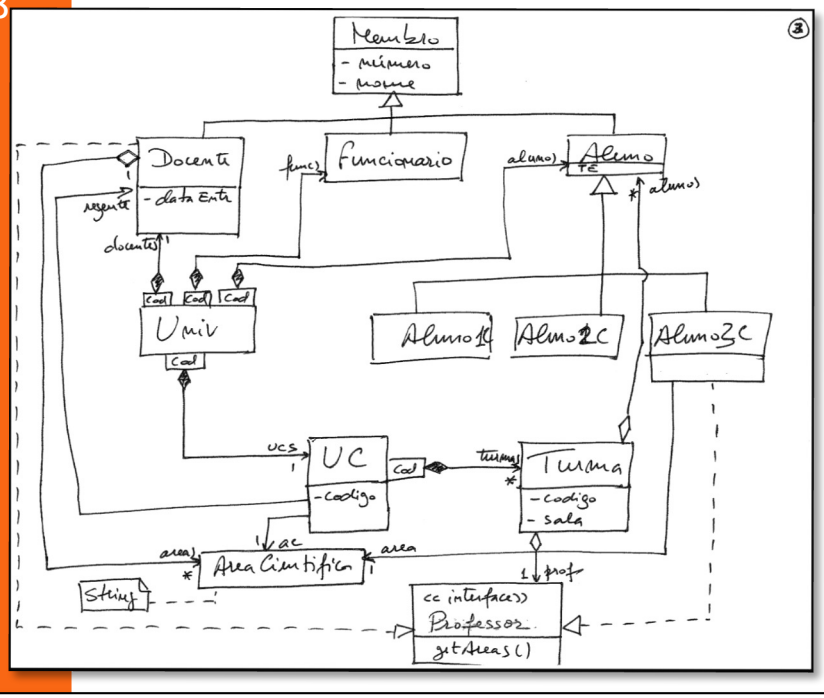


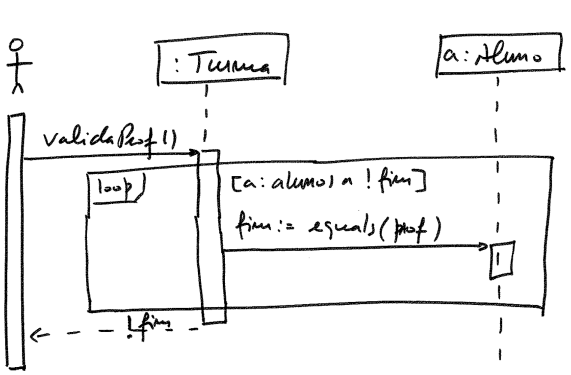
Código...

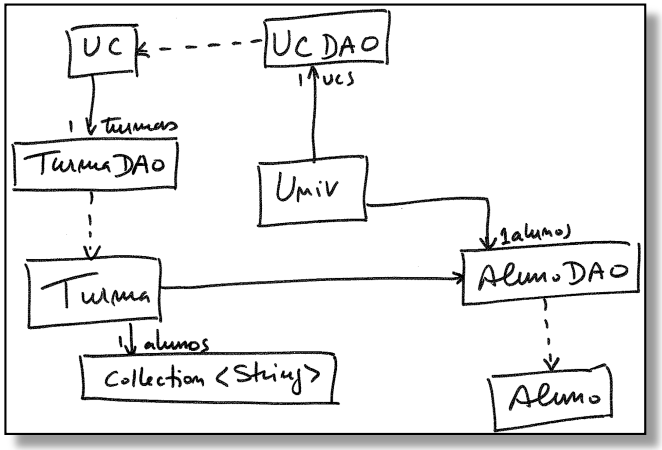





```
class Turma {
    private String codigo, sala;
    private Professor prof;
    private Collection<Aluno> alunos;

    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;
    }

}
```

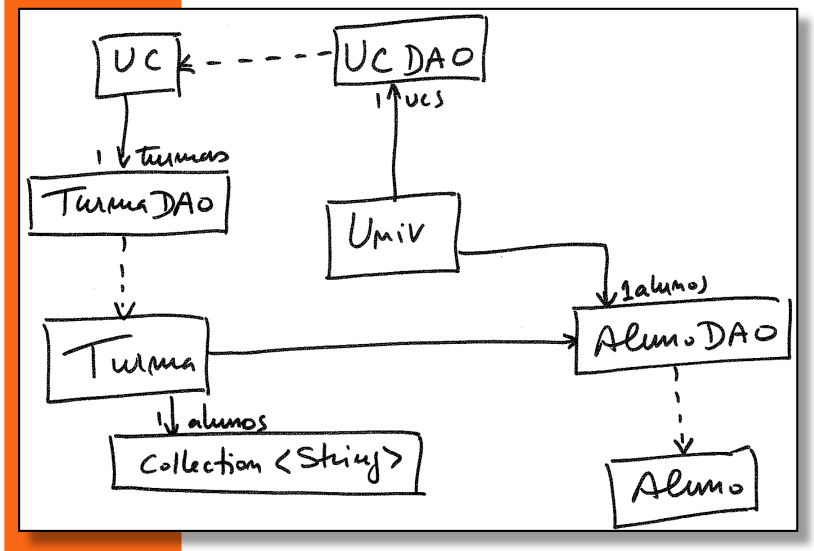


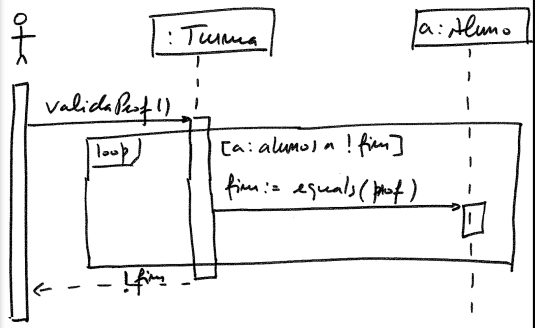


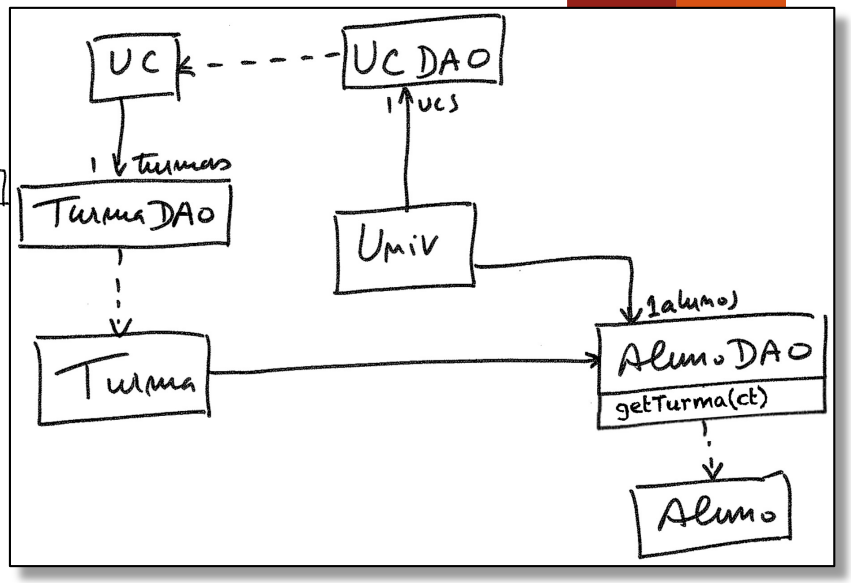



```java
class Turma {
    private String codigo, sala;
    private Professor prof;
    private Collection<Aluno> alunos;

    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;
    }

}
```

```java
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;
    }

}
```



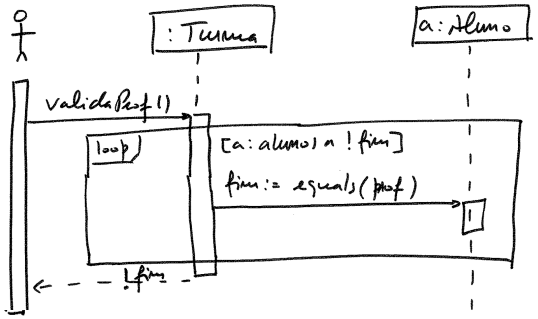


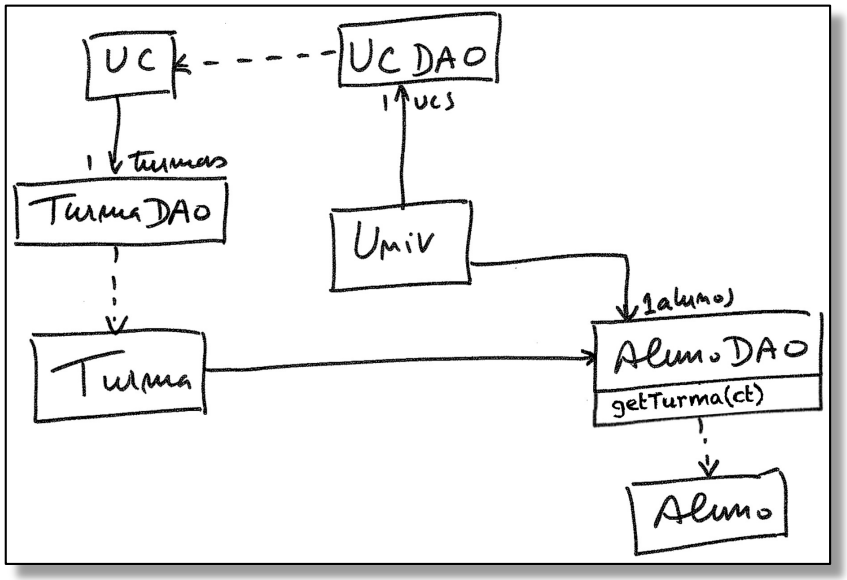



```java
class TurmaPersist {
    private String codigo, sala;
    private Professor prof;
    private Collection<String> alunos;
    private AlunoDAO alDAO;

    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;
    }

}
```

```java
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;
      }
}
```

Código...

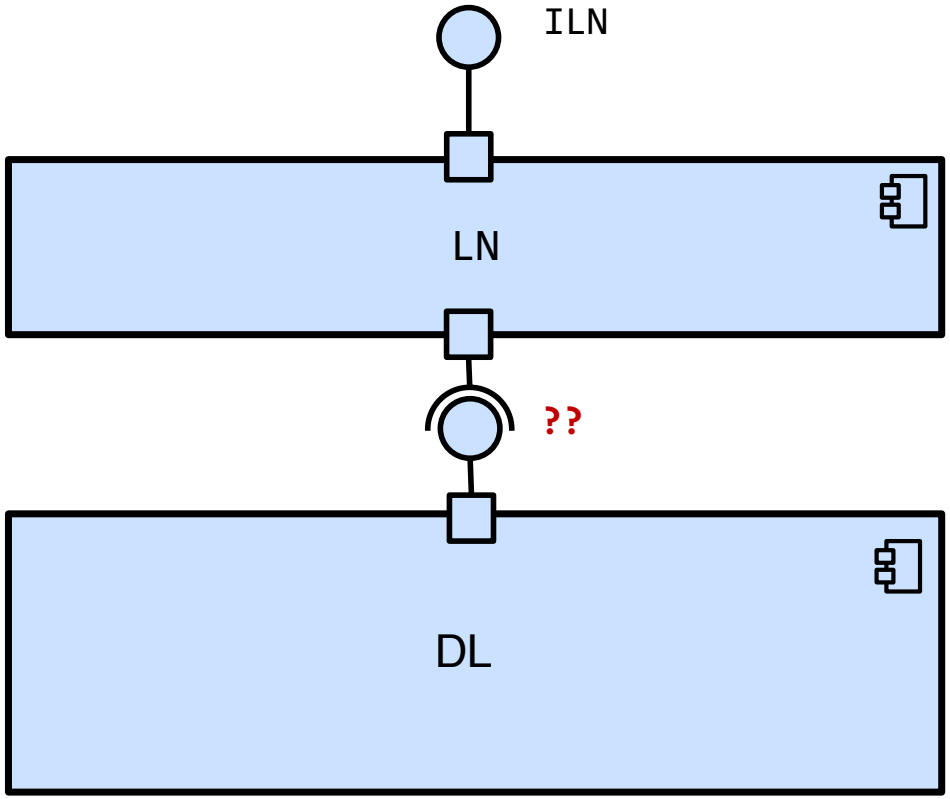




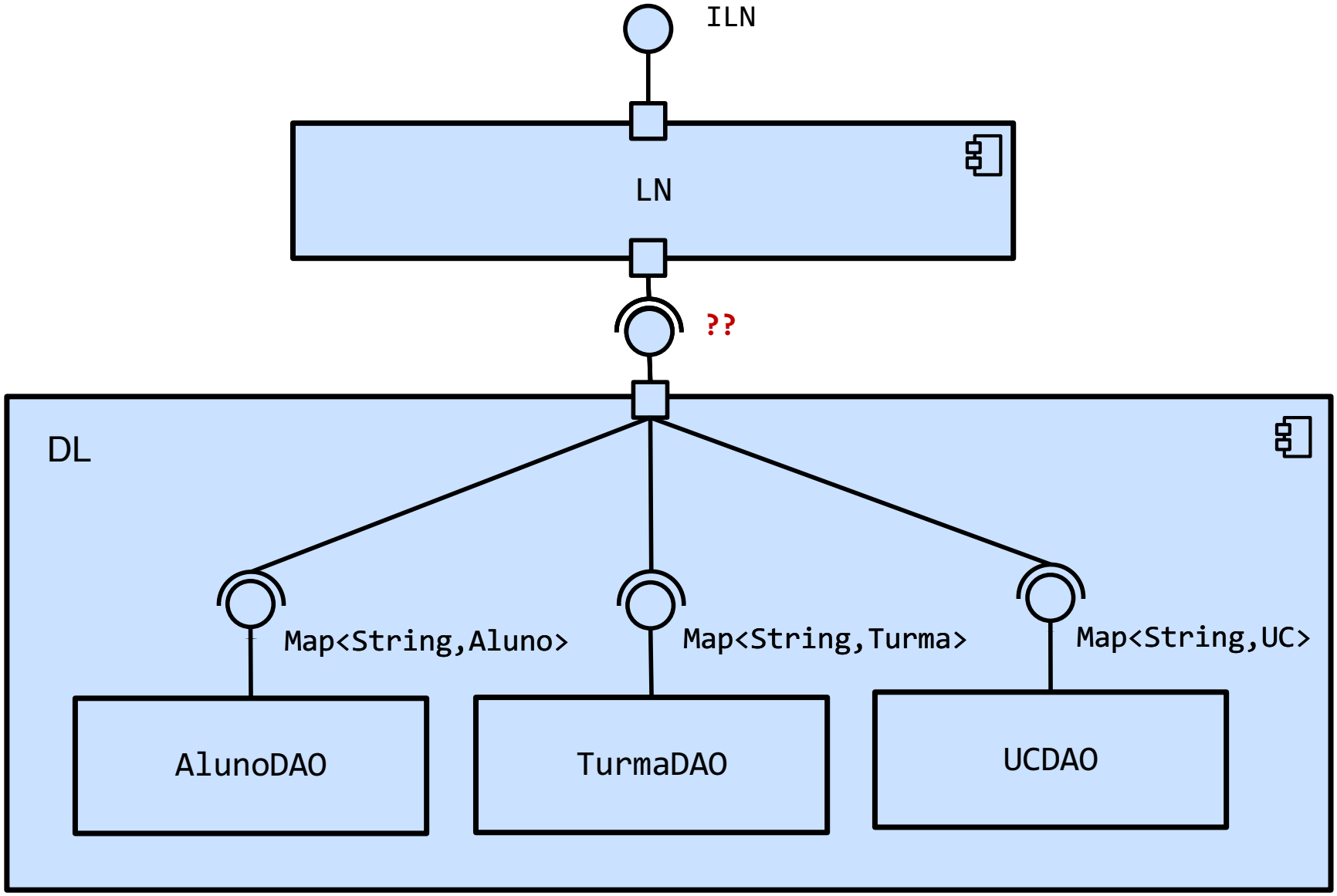

```
class TurmaPersist2 {
      private String codigo, sala;
      private Professor prof;
      private AlunoDAO alDAO;

      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;
      }
}
```

```
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;
            Iterator<Aluno> alunosIt = alDAO.getTurma(codigo);
            while (!fim & alunosIt.hasNext()) {
                  Aluno a = alunosIt.next();
                  fim = a.equals(prof);
            }
            return !fim;
      }
}
```

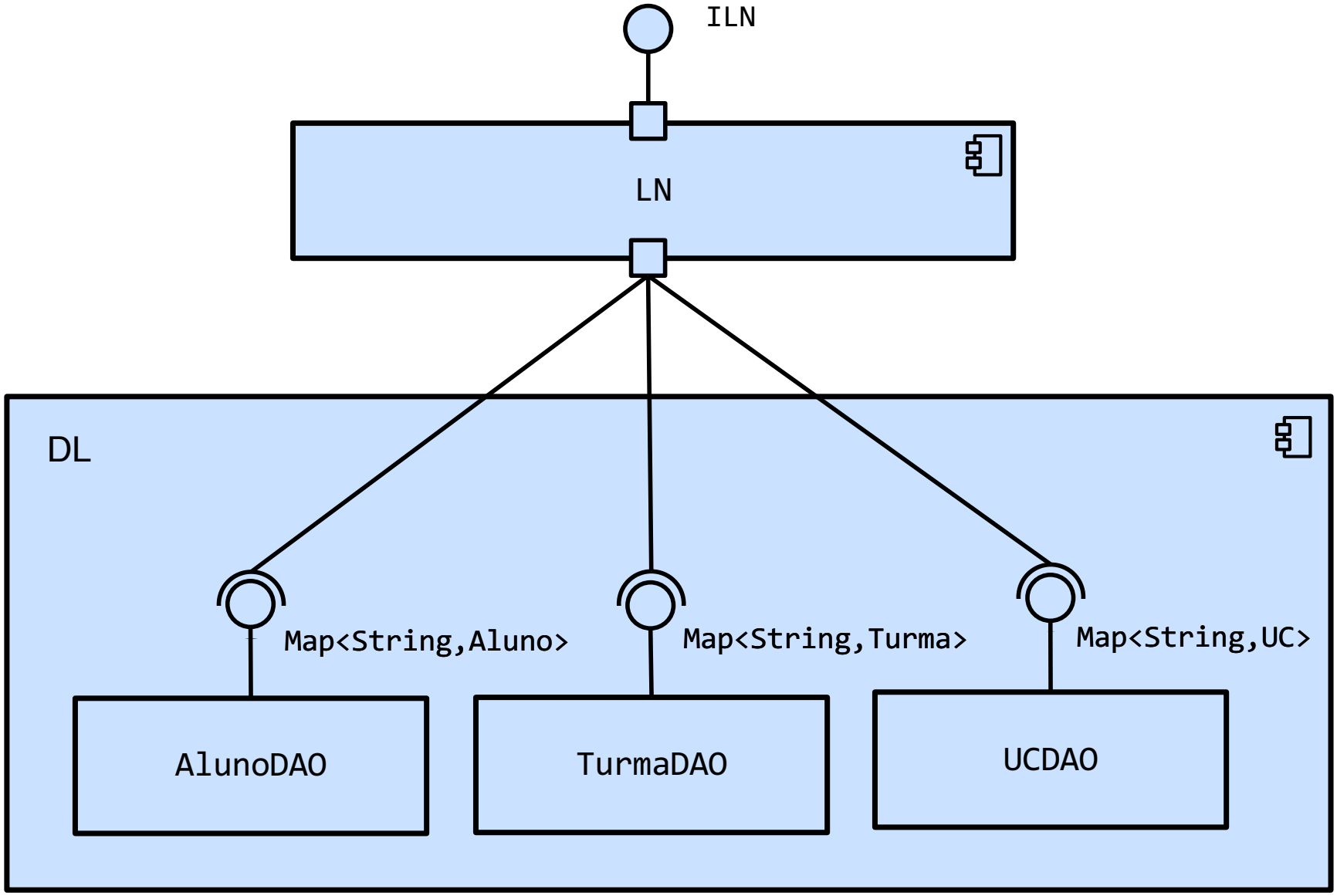
Arquitectura de 3 camadas para o exemplo



Camada de Dados para o exemplo



Camada de Dados para o exemplo



```java
public Turma put(String key, Turma t) {
    Turma res = null;
    Sala s = t.getSala();
    try (Connection conn = DriverManager.getConnection(DAOconfig.URL, DAOconfig.USERNAME, DAOconfig.PASSWORD);
         Statement stm = conn.createStatement()) {

        // Actualizar a Sala
        stm.executeUpdate(
                "INSERT INTO salas " +
                        "VALUES ('"+ s.getNumero()+ "', '"+
                                s.getEdificio()+"', "+
                                s.getCapacidade()+") " +
                        "ON DUPLICATE KEY UPDATE Edificio=Values(Edificio), " +
                                                "Capacidade=Values(Capacidade)");

        // Actualizar a turma
        stm.executeUpdate(
                "INSERT INTO turmas VALUES ('"+t.getId()+"', '"+s.getNumero()+"') " +
                        "ON DUPLICATE KEY UPDATE Sala=VALUES(Sala)");

        // Actualizar os alunos da turma
        Collection<String> oldAl = getAlunosTurma(key, stm);
        Collection<String> newAl = t.getAlunos().stream().map(Aluno::getNumero).collect(toList());
        newAl.removeAll(oldAl);         // Alunos que entram na turma, em relação ao que está na BD
        oldAl.removeAll(t.getAlunos().stream().map(Aluno::getNumero).collect(toList())); // Alunos que saem na turma, em relação ao que está na BD
        try (PreparedStatement pstm = conn.prepareStatement("UPDATE alunos SET Turma=? WHERE Num=?")) {
            // Remover os que saem da turma (colocar a NULL a coluna que diz qual a turma dos alunos)
            pstm.setNull(1, Types.VARCHAR);
            for (String a: oldAl) {
                pstm.setString(2, a);
                pstm.executeUpdate();
            }
            // Adicionar os que entram na turma (colocar o Id da turma na coluna Turma da tabela alunos)
            pstm.setString(1, t.getId());
            for (String a: newAl) {
                pstm.setString(2, a);
                pstm.executeUpdate();
            }
        }

    } catch (SQLException e) {
        // Database error!
        e.printStackTrace();
        throw new NullPointerException(e.getMessage());
    }
    return res;
}
```

Sobre ORM…

- Tem em consideração o que foi, até agora, dito sobre DAOs, assinale as afirmações verdadeiras:
  a) Nenhuma das frases abaixo é verdadeira
  b) Manter todos os dados em Base de Dados e em memória, em simultâneo, é uma boa solução para garantir persistência
  c) Um DAO contém em memória os objetos que é responsável por persistir (p.e. o AlunoDAO vai ter os alunos num Map em memória)
  d) Um DAO não contém em memória os objetos que é responsável por persistir
  e) A relação entre um DAO e a classe que ele deve persistir é de associação (p.e. o AlunoDAO tem uma associação para Aluno)
  f) A relação entre um DAO e a classe que ele deve persistir é de dependência (p.e. o AlunoDAO tem uma dependência para Aluno)
  g) A relação entre um DAO e a classe que ele deve persistir é de generalização (p.e. o AlunoDAO é uam superclasse de Aluno)

Código…

- Exemplo da Turma (3 camadas)
  - DAOs ficam num package que faz acesso aos dados
  - TurmaDAO
    - (fichei

a) Não estou a fazer o trabalho 😐
b) Congelei a nota prática 🥶
c) Já tenho o projecto da primeira aula a funcionar 😎
d) Não tenho o projecto a funcionar, mas consigo usar o JDBC para ligar a uma BD 🙂
e) Já tentei usar o JDBC, mas ainda não consegui fazer uma ligação a uma BD 😟
f) Ainda não tentei usar o JDBC 😱

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

Sobre o trabalho

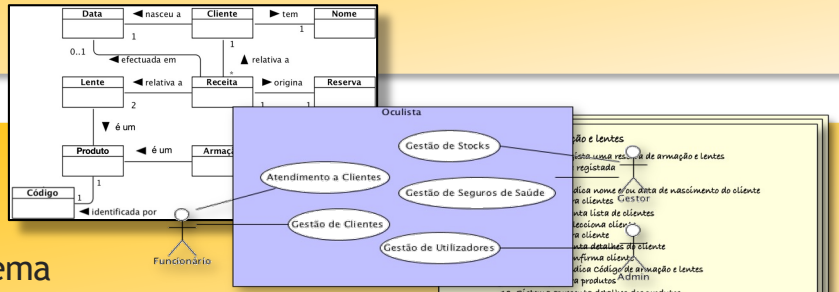
"Problema"

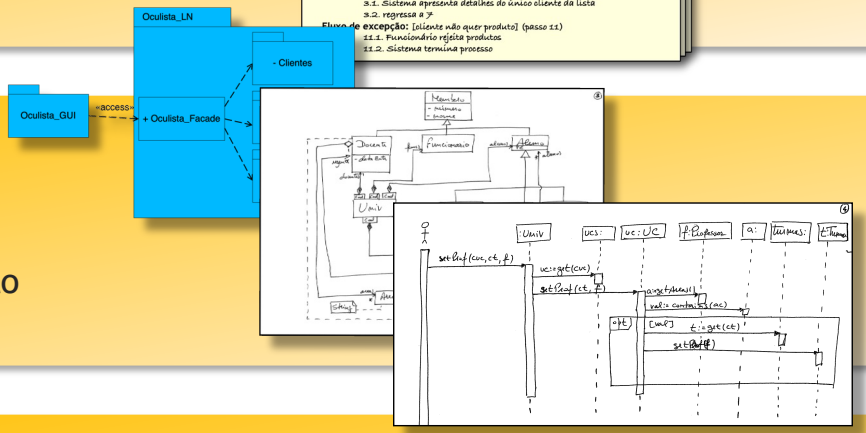
Planeamento

- 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