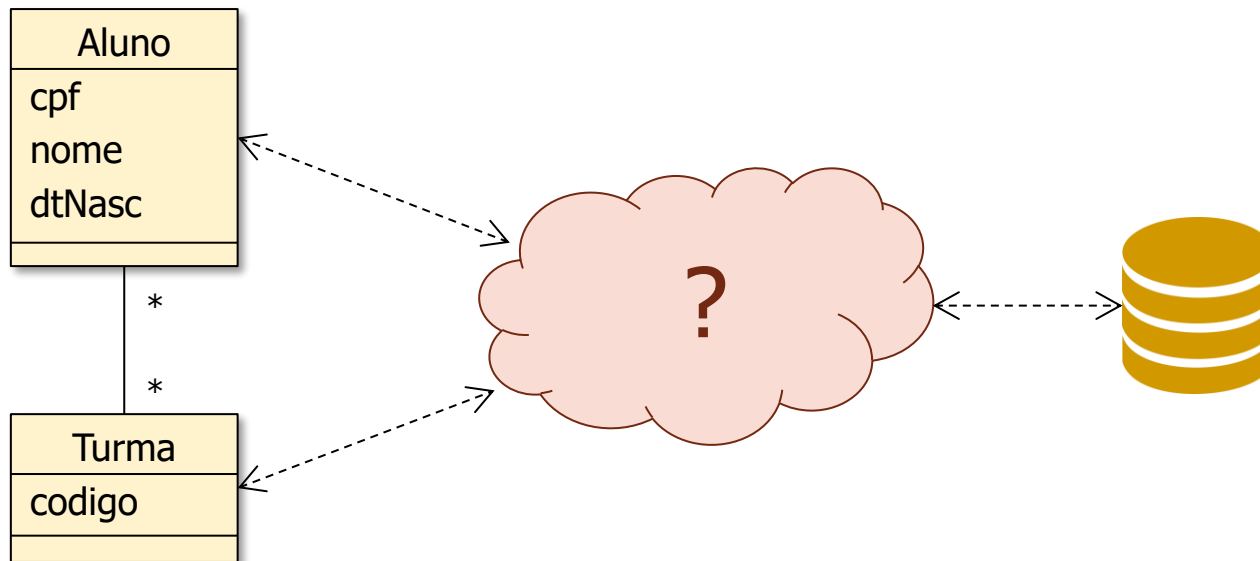

Projeto de Sistemas

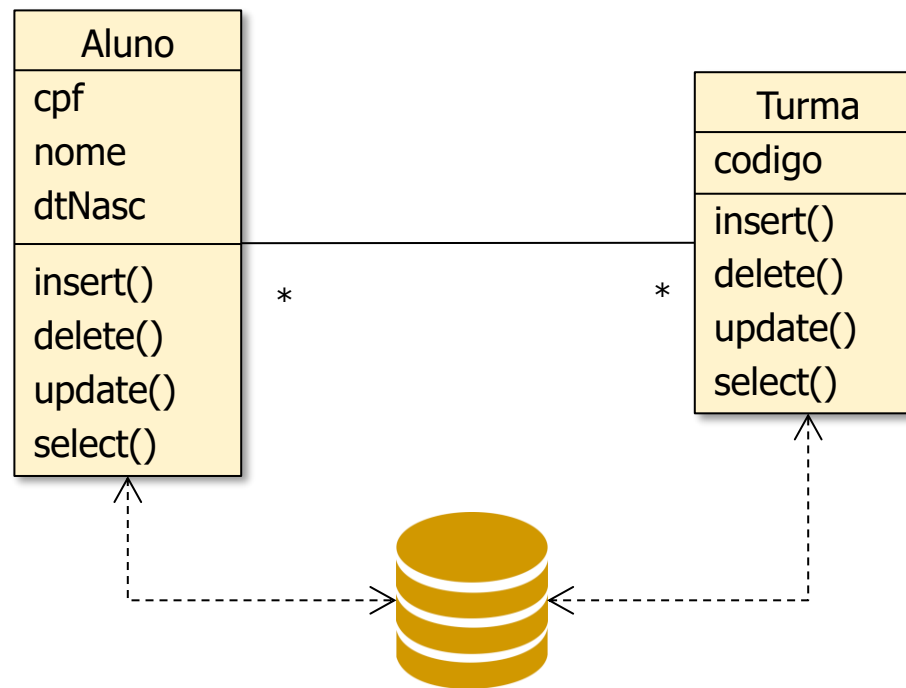
Persistência de Objetos + Sequelize

Prof. Jobson Massollar

Como **mapear** os objetos de um sistema em um banco de dados (SQL ou NoSQL)?



A implementação dos comandos de acesso ao BD nas classes do domínio **não** é uma solução adequada (contraria a separação de responsabilidades e vários outros princípios).

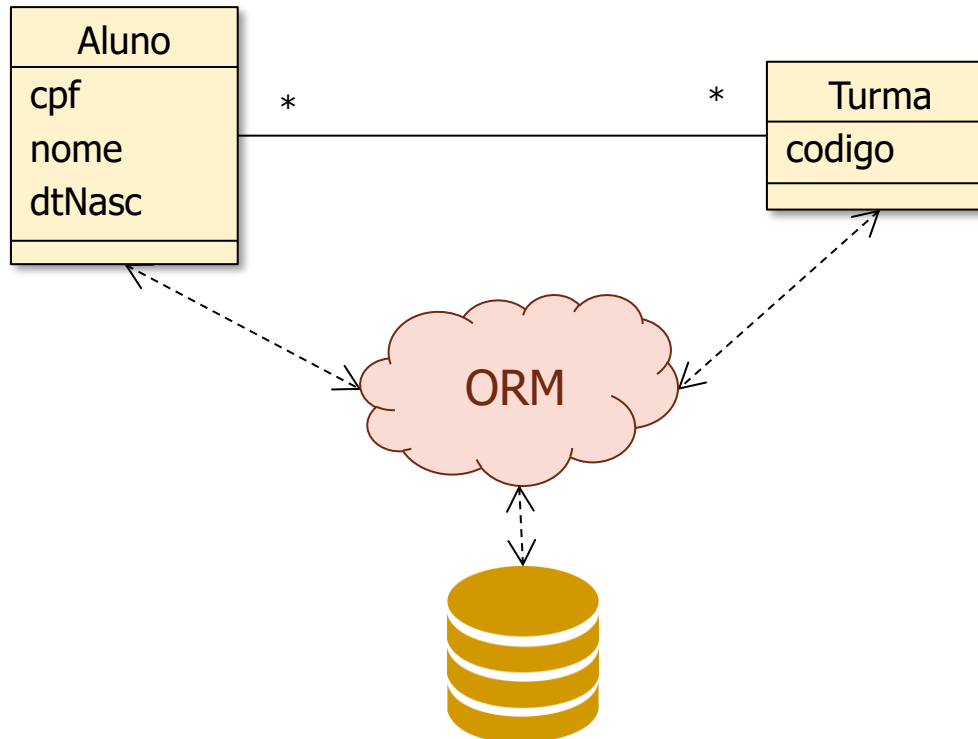


Alguns frameworks, chamados de **ORM** (Object-Relational Mapping), foram criados para realizar essa tarefa de forma (semi) automática:

- ✓ Hibernate (Java/Relacional)
- ✓ Entity Framework (.NET/Relacional)
- ✓ Sequelize (Javascript/Typescript/Relacional)
- ✓ TypeORM (Javascript/Typescript/Relacional)
- ✓ Mongoose (Javascript/Typescript/MongoDB)

dentre outros.

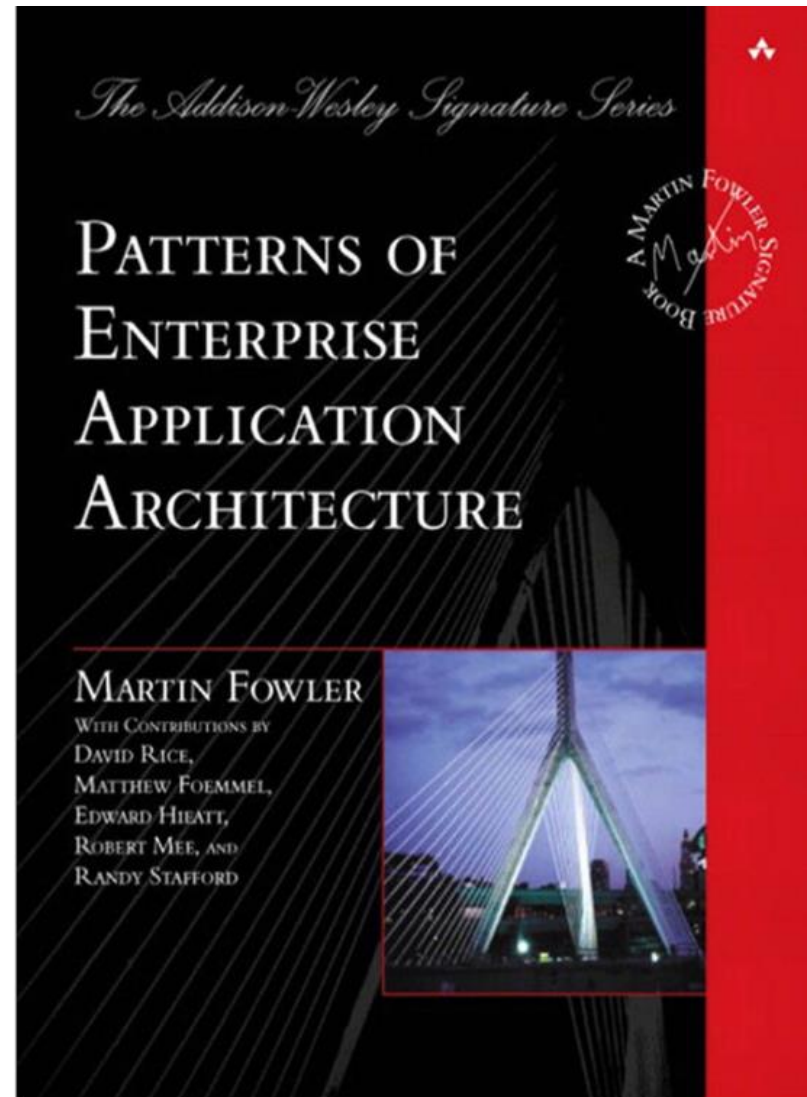
Entretanto, o acesso direto ao ORM a partir das classes do domínio também **não** é adequado, pois os comandos e estruturas do ORM seriam levados para essas classes.



Padrões de Acesso a Dados

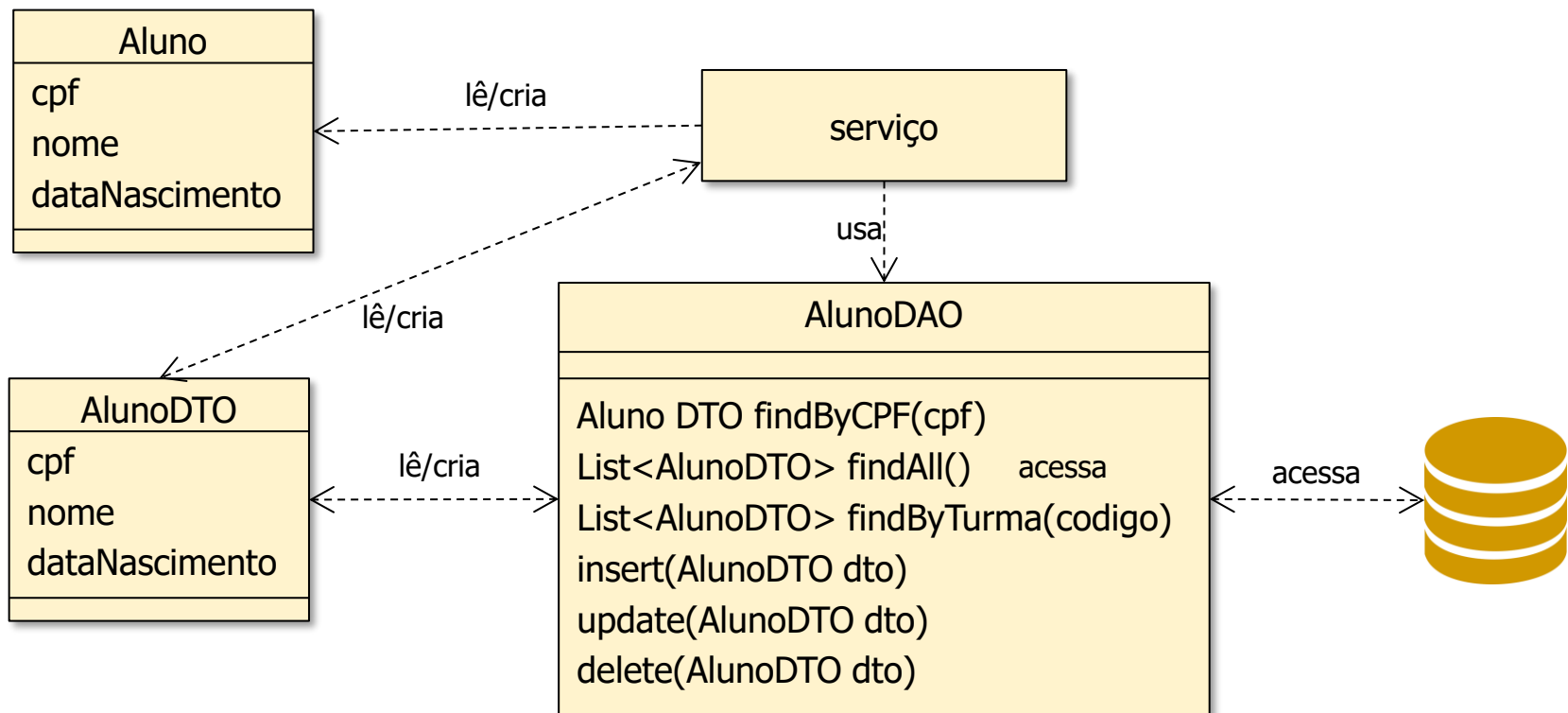
Os padrões de acesso a dados discutem um conjunto de estratégias e padrões para mapear os dados dos objetos nas entidades de um banco de dados e vice-versa.

Essas estratégias e padrões foram pensadas para bancos de dados relacionais, mas também podem ser aplicadas a bancos de dados NoSQL.



Data Access Object

Objeto que abstrai e encapsula as operações realizadas no banco de dados. Gerencia a conexão com o banco de dados para salvar e obter dados de um DTO.



Retornando um objeto:

```
AlunoDAO dao = new AlunoDAO();  
  
AlunoDTO dto = dao.findByCPF(85436753877);  
  
Aluno aluno = new Aluno(dto.cpf, dto.nome, dto.dataNascimento);
```

```
public class Aluno {  
    private CPF cpf;  
    private String nome;  
    private LocalDate dtNascimento;  
  
    public Aluno(CPF cpf, String nome, LocalDate dtNascimento)  
        // Inicializa os atributos do objeto  
    }  
}
```

```
public AlunoDAO {  
  
    public AlunoDTO findByCpf(CPF cpf) {  
  
        // SELECT CPF, NOME, DATA FROM ALUNO WHERE CPF = cpf  
  
        // Inicializa o objeto DTO com os dados do SELECT  
        return new AlunoDTO(CPF, NOME, DATA);  
    }  
}
```


Retornando uma coleção de objetos:

```
AlunoDAO dao = new AlunoDAO();  
  
List<AlunoDTO> dtos = dao.findByTurma(246);  
  
List<Aluno> alunos = new ArrayList<>();  
  
for(AlunoDTO dto : dtos)  
    alunos.add(new Aluno(dto.cpf, dto.nome, dto.dataNascimento));
```

```
public AlunoDAO {  
  
    public List<AlunoDTO> findByTurma(int codigo) {  
  
        // SELECT * FROM ALUNO WHERE...  
  
        List<AlunoDTO> dtos = new List<>();  
  
        // Para cada linha do SELECT  
        dtos.add(new AlunoDTO(CPF, NOME, DATA));  
  
        return dtos;  
    }  
}
```

Salvando um objeto:

```
Aluno aluno = new Aluno(82376525342, "Joao", LocalDate.of(1,1,2020));

AlunoDTO dto = new AlunoDTO(aluno.getCpf(), aluno.getNome(), aluno.getDtnascimento());

AlunoDAO dao = new AlunoDAO();

// Aluno já está no BD?

dao.insert(dto); // Não

dao.update(dto); // Sim
```

```
public AlunoDAO {

    public void insert(AlunoDTO dto) {

        // INSERT INTO ALUNO VALUES...
    }

    public void update(AlunoDTO dto) {

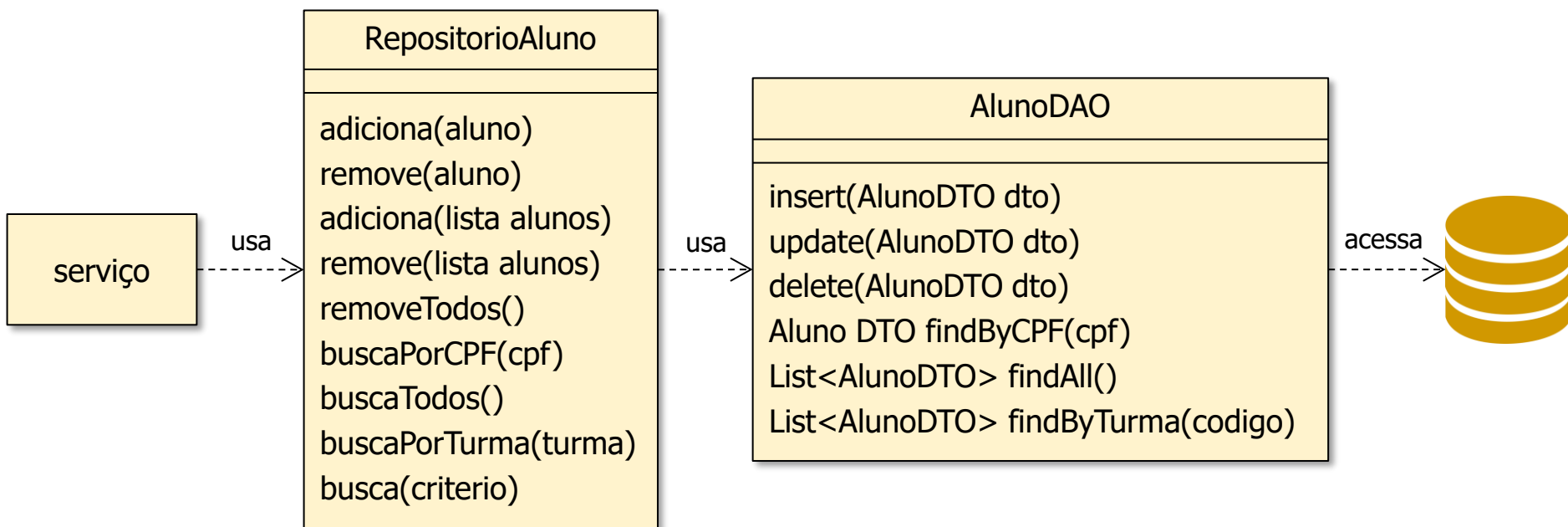
        // UPDATE ALUNO SET...
    }
}
```

Excluindo um objeto:

```
AlunoDTO dto = new AlunoDTO(aluno.getCpf(), aluno.getNome(), aluno.getDtnascimento());  
  
AlunoDAO dao = new AlunoDAO();  
  
dao.delete(dto);
```

```
public AlunoDAO {  
  
    public void delete(AlunoDTO dto) {  
  
        // DELETE FROM ALUNO WHERE ...  
    }  
}
```

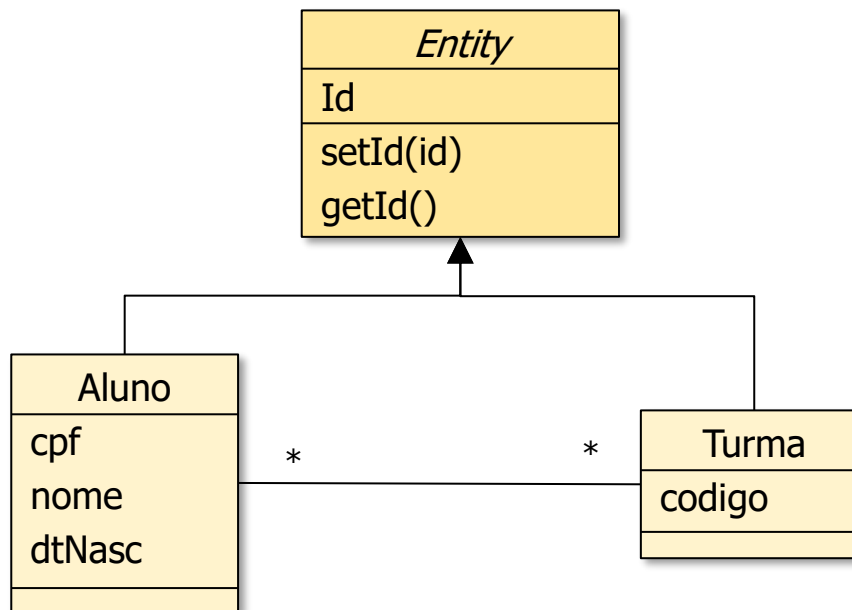
Cria uma camada de abstração e faz a mediação entre o objeto do domínio e o DAO, encapsulando toda a lógica de tratamento da persistência.



O Repositório:

- ✓ Adota uma interface semelhante a uma **coleção** para acessar objetos de domínio, evitando métodos com os termos *insert*, *delete* e *update*.
- ✓ Funciona como uma **fábrica para os objetos** que são recuperados do banco de dados.
- ✓ Não implementa o acesso ao banco de dados propriamente dito. Essas operações são realizadas pelos DAOs.
- ✓ Pode criar os objetos através de métodos estáticos, *factories* ou *builders*.
- ✓ Permite a especificação de critérios de seleção ou exclusão independente do banco de dados.

Essa estratégia permite associar um ID a cada objeto, de forma que ele possa ser identificado como único no BD. Nos BDs, esse ID vai ser usado como *primary key*.



Os tipos de dados usados nos BDs não são os mesmos usados nas linguagens de programação. Os Data Mappers realizam a tarefa de mapear (converter) os dados que vem do BD para os objetos.

AlunoMapper
map(bd, dto)

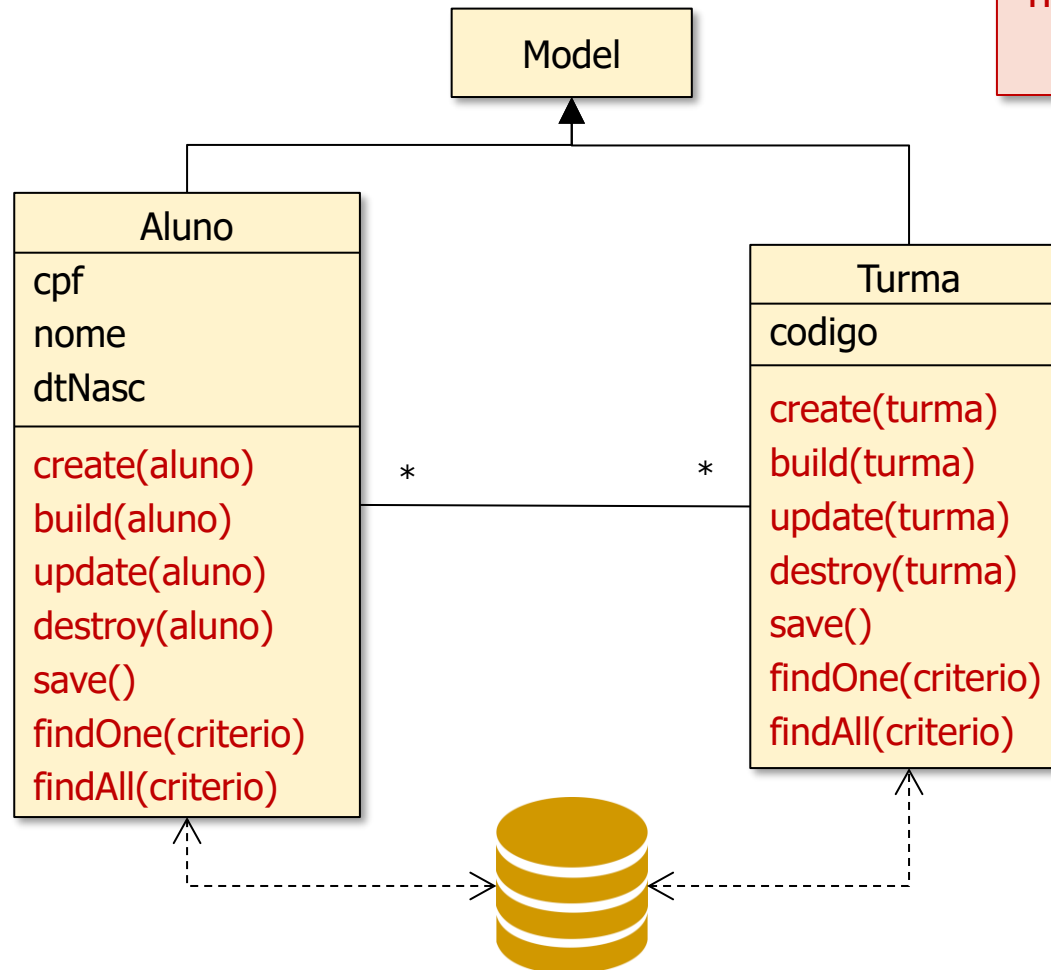
TurmaMapper
map(bd, dto)

O Sequelize provê duas estratégias para fazer o mapeamento das classes do domínio para as tabelas do BD:

- 1) Usando o método `Sequelize.define`
- 2) Estendendo a classe do domínio a partir da classe `Model` do Sequelize. Nesse caso, devemos implementar o método estático `init` para definir as propriedades da classe que serão mapeadas para o BD.

Nesse material, vamos adotar a segunda estratégia, pois permite uma separação mais clara entre a classe do domínio e as definições relacionadas ao BD.

Exemplo:



As classes que estendem Model herdam uma série de métodos para acessar o BD.

Exemplo: classe Aluno

```
class Aluno extends Model {  
  
  static of(cpf, nome, dtNascimento) {  
    // Implementação da validação aqui!  
  
    return Aluno.build({ cpf, nome, dtNascimento });  
  }  
  
  inscrever(turma) {  
    // Implementação aqui!  
  }  
}
```

A classe pode conter métodos e propriedades não mapeadas para o BD. Por outro lado, as propriedades que representam relacionamentos não são definidas na classe.

Exemplo: classe Aluno

```
class Aluno extends Model {  
  
  static of(cpf, nome, dtNascimento) {  
    // Implementação da validação aqui!  
  
    return Aluno.build({ cpf, nome, dtNascimento });  
  }  
  
  inscrever(turma) {  
    // Implementação aqui!  
  }  
}
```

Atenção!
Objetos mapeados para o BD **não**
podem ser criados com new. Devem ser
criados com o método estático build!

Exemplo: classe Aluno

```
const createModelAluno = (Aluno, sequelize, DataTypes) => {  
  Aluno.init(  
    {  
      id: {  
        type: DataTypes.UUID,  
        primaryKey: true,  
        defaultValue: DataTypes.UUIDV4,  
      },  
      cpf: { type: DataTypes.STRING(11), allowNull: false },  
      nome: { type: DataTypes.STRING(100), allowNull: false },  
      dtNascimento: { type: DataTypes.DATEONLY, allowNull: false },  
    },  
    { sequelize }  
  );  
};
```

As propriedades da classe mapeadas para o BD não são definidas na própria classe e sim no método init

```
class Db {
  #sequelize;

  async init() {
    this.#sequelize = new Sequelize(
      dbConfig.database, dbConfig.username, dbConfig.password,
      { host: dbConfig.host, dialect: dbConfig.dialect, logging: false, }
    );

    try {
      await this.#sequelize.authenticate();
    } catch (error) {
      return false;
    }

    // Cria os modelos (tabelas)
    createModelAluno(Aluno, this.#sequelize, Sequelize.DataTypes);
    createModelTurma(Turma, this.#sequelize, Sequelize.DataTypes);

    // Cria os relacionamentos
    Aluno.belongsToMany(Turma, { through: "AlunoTurma", as: "turmas" });
    Turma.belongsToMany(Aluno, { through: "AlunoTurma", as: "alunos" });

    return true;
  }

  get sequelize() { return this.#sequelize; }
}
```

Nesse exemplo, definimos uma classe para configuração do acesso ao BD e para criação dos modelos e relacionamentos.

Exemplo: criação de Aluno e Turma nas classes de serviço

```
// Cria o objeto em memória apenas
var result = Aluno.of("12345678901", "Fernanda de Abreu",
                     new Date(2000, 0, 1));

if (result.isSuccess) {
  // Em caso de sucesso o objeto está em result.value
  const aluno = result.value;
  console.log(aluno.cpf, aluno.nome, aluno.dtNascimento);
  await aluno.save();
} else
  console.log(`Erro na criação do aluno ${result.errors}`);

// Cria o objeto em memória apenas
result = Turma.of("T101");

if (result.isSuccess) {
  // Em caso de sucesso o objeto está em result.value
  const turma = result.value;
  console.log(turma.codigo);
  await turma.save();
} else
  console.log(`Erro na criação da turma ${result.errors}`);
```

Exemplo: recuperação de objetos nas classes de serviço

```
const aluno = await Aluno.findOne({ where: { cpf: "12345678901" } });

if (aluno)
  console.log(aluno.cpf, aluno.nome, aluno.dtNascimento);

const alunos = await Aluno.findAll();

for (let a of alunos)
  console.log(a.cpf, a.nome, a.dtNascimento);

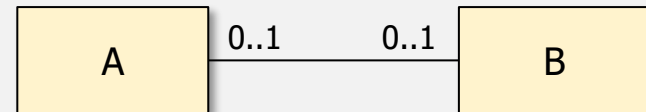
const turma = await Turma.findOne({ where: { codigo: "T101" } });

if (turma)
  console.log(turma.codigo);
```

Para definir os relacionamentos, usamos os seguintes métodos (por default, os relacionamentos são opcionais):

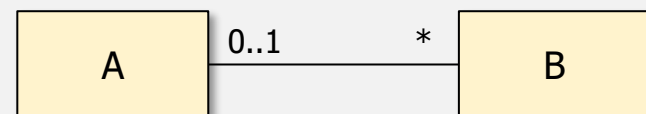
1) 1:1

```
A.hasOne(B);  
B.belongsTo(A);
```



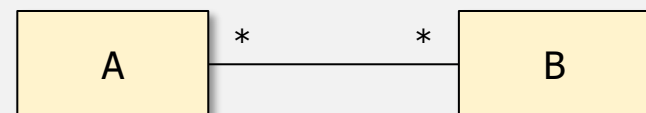
2) 1:N

```
A.hasMany(B);  
B.belongsTo(A);
```



3) N:M

```
A.belongsToMany(B);  
B.belongsToMany(A);
```



Exemplos: 1:1

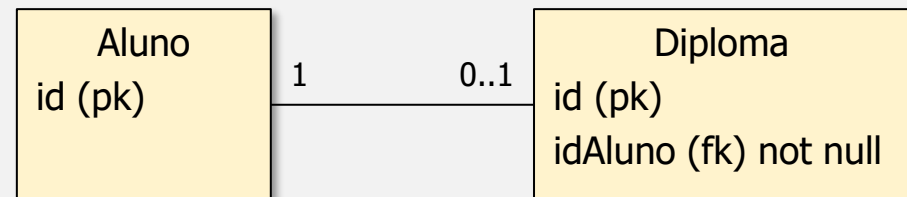
a) Opcional

```
Pessoa.hasOne(Casa);  
Casa.belongsTo(Pessoa);
```



b) Obrigatório

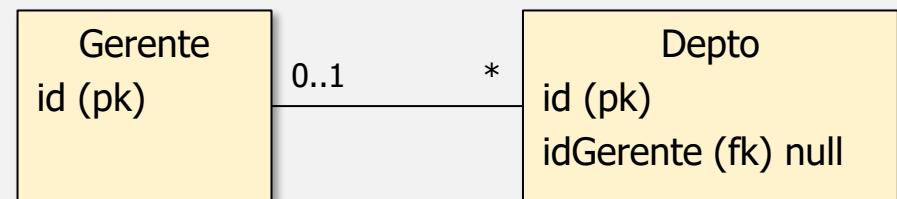
```
Aluno.hasOne(Diploma,  
  { foreignKey: {  
    allowNull: false,  
  }  
});  
Diploma.belongsTo(Aluno);
```



Exemplos: 1:N

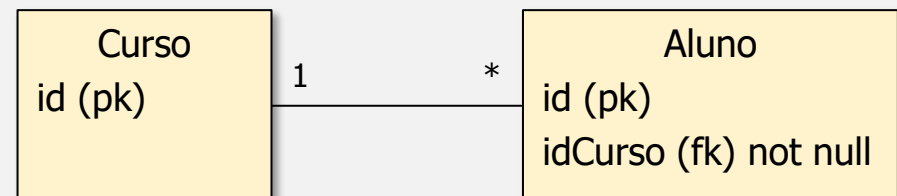
a) Opcional

```
Gerente.hasMany (Depto);  
Depto.belongsTo (Gerente);
```



b) Obrigatório

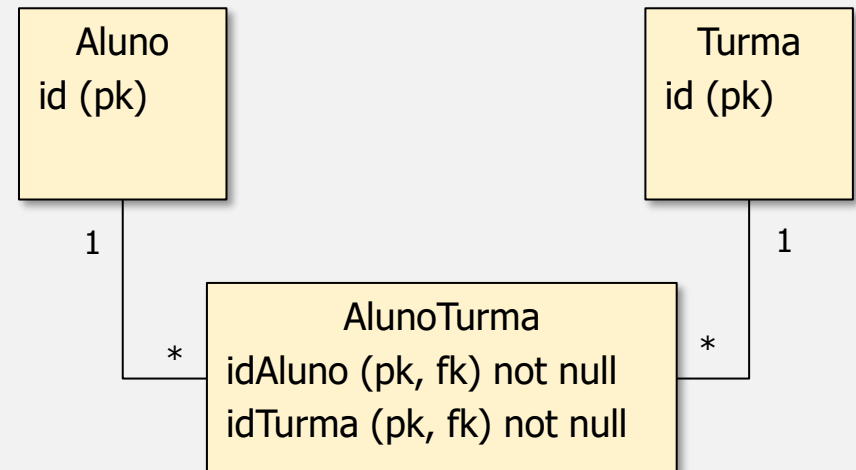
```
Curso.hasMany (Aluno,  
  { foreignKey: {  
    allowNull: false,  
  }  
});  
Aluno.belongsTo (Curso);
```



Exemplos: N:M

a) Opcional

```
Aluno.belongsToMany(Turma,  
  { through: "AlunoTurma",  
    as: "turmas"  
  }  
);  
Turma.belongsToMany(Aluno,  
  { through: "AlunoTurma",  
    as: "alunos"  
  }  
);
```



Quando um relacionamento é definido, o Sequelize gera vários métodos **mágicos** nas classes para gerenciar esse relacionamento.

Método	Descrição
add	Adiciona um ou mais objetos ao relacionamento
create	Cria um objeto e adiciona ao relacionamento
get	Recupera os objetos do relacionamento
count	Retorna a quantidade de objetos do relacionamento
has	Verifica se um ou mais objetos pertencem ao relacionamento
remove	Remove um ou mais objetos do relacionamento
set	Muda um ou mais objetos do relacionamento

Os métodos a serem gerados depende da cardinalidade (1:1, 1:N ou N:M)

1) 1:1

```
Pessoa.hasOne(Casa);
Casa.belongsTo(Pessoa);
```

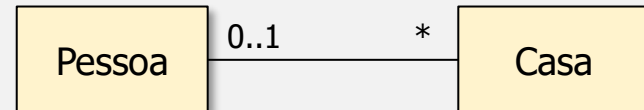


Método	Parâmetro	Descrição
peessoa.createCasa()	casa nova	Cria uma casa no BD e associa à pessoa. Retorna a casa criada
peessoa.getCasa()	nenhum	Retorna a casa associada à pessoa, ou null se não existir
peessoa.setCasa()	casa ou id da casa	Associa a casa à pessoa

Método	Parâmetro	Descrição
casa.createPessoa()	pessoa	Cria uma pessoa no BD e associa à casa. Retorna a pessoa criada
casa.getPessoa()	nenhum	Retorna a pessoa associada à casa, ou null se não existir
casa.setPessoa()	pessoa ou id da pessoa	Associa a pessoa à casa

1) 1:N (nesse exemplo, os métodos de Casa são os mesmos do 1:1)

```
Pessoa.hasMany(Casa);
Casa.belongsTo(Pessoa);
```



Método	Parâmetro	Descrição
peessoa.addCasa()	casa ou id da casa	Associa a casa à pessoa
peessoa.addCasas()	array de casas ou ids	Associa as casas à pessoa
peessoa.countCasas()	nenhum	Retorna o número de casas da pessoa
peessoa.createCasa()	casa	Cria uma casa no BD e associa à pessoa. Retorna a casa criada
peessoa.getCasas()	nenhum	Retorna as casas da pessoa
peessoa.hasCasa()	casa ou id da casa	true, se pessoa está associada com a casa
peessoa.hasCasas()	array de casas ou ids	true, se pessoa está associada com todas as casas
peessoa.removeCasa()	casa ou id da casa	Remove a casa da pessoa
peessoa.removeCasas()	array de casas ou ids	Remove as casas da pessoa
peessoa.setCasas()	array de casas ou ids	Remove todas as casas atuais da pessoa e associa as casas do parâmetro à pessoa

1) N:M (os métodos de Pessoa também vão existir em Casa)

```
Pessoa.belongsToMany(Casa);
Casa.belongsToMany(Pessoa);
```



Método	Parâmetro	Descrição
peessoa.addCasa()	casa ou id da casa	Associa a casa à pessoa
peessoa.addCasas()	array de casas ou ids	Associa as casas à pessoa
peessoa.countCasas()	nenhum	Retorna o número de casas da pessoa
peessoa.createCasa()	casa	Cria uma casa no BD e associa à pessoa. Retorna a casa criada
peessoa.getCasas()	nenhum	Retorna as casas da pessoa
peessoa.hasCasa()	casa ou id da casa	true, se pessoa está associada com a casa
peessoa.hasCasas()	array de casas ou ids	true, se pessoa está associada com todas as casas
peessoa.removeCasa()	casa ou id da casa	Remove a casa da pessoa
peessoa.removeCasas()	array de casas ou ids	Remove as casas da pessoa
peessoa.setCasas()	array de casas ou ids	Remove todas as casas atuais da pessoa e associa as casas do parâmetro à pessoa


Exemplo: associação de Aluno com Turma na classe de serviço

```
const aluno = await Aluno.findOne({ where: { cpf: "12345678901" } });
const turma = await Turma.findOne({ where: { codigo: "T101" } });

// Adiciona o aluno à turma
if (aluno && turma)
  await aluno.addTurma(turma);

// Lista os alunos da turma
for (let t of await aluno.getTurmas())
  console.log(t.codigo);

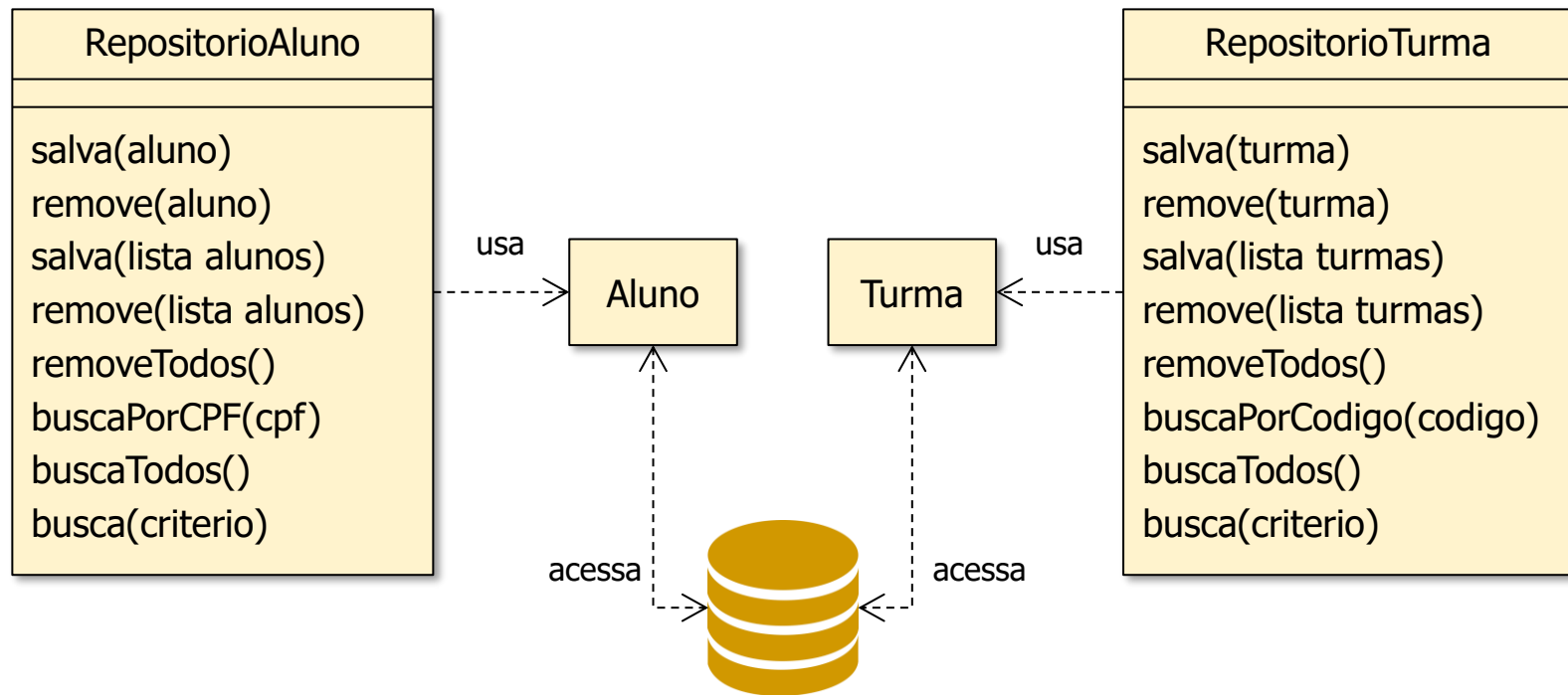
// Lista as turmas do aluno
for (let a of await turma.getAlunos())
  console.log(a.cpf, a.nome, a.dtNascimento);
```



Duas questões com essa implementação:

- a) O serviço precisa implementar as consultas com os comandos do Sequelize.
- b) Onde valida se o aluno pode ou não se inscrever na turma?

O padrão **Repository** também pode ser usado para abstrair algumas operações que estão nas classes de domínio e serviço.



Exemplo: criação de Aluno e Turma nas classes de serviço

```
// Cria o objeto em memória apenas
var result = Aluno.of("23456789012", "Jose Carlos",
                    new Date(2001, 0, 1));

if (result.isSuccess) {
  // Em caso de sucesso o objeto está em result.value
  const aluno = result.value;
  console.log(aluno.cpf, aluno.nome, aluno.dtNascimento);
  await repositorioAluno.salva(aluno);
} else
  console.log(`Erro na criação do aluno ${result.errors}`);

// Cria o objeto em memória apenas
result = Turma.of("T102");

if (result.isSuccess) {
  // Em caso de sucesso o objeto está em result.value
  const turma = result.value;
  console.log(turma.codigo);
  await repositorioTurma.salva(turma);
} else
  console.log(`Erro na criação da turma ${result.errors}`);
```

Exemplo: associação de Aluno com Turma nas classes de serviço

```
const aluno = await repositorioAluno.buscaPorCPF("23456789012");
const turma = await repositorioTurma.buscaPorCodigo("T102");

if (aluno && turma)
  await aluno.inscreve(turma);

// Lista as turmas do aluno
for (let t of await aluno.getTurmas())
  console.log(t.codigo);

// Lista os alunos da turma
for (let a of await turma.getAlunos())
  console.log(a.cpf, a.nome, a.dtNascimento);
```

```
class Aluno {
  async inscreve(turma) {
    // Valida a inscrição
    if (await turma.estaCheia())
      return false;

    // Associa o aluno com turma
    await this.addTurma(turma);

    return true;
  }
}
```

```
class Turma {
  async estaCheia() {
    return (await this.countAlunos()) >= 30;
  }
}
```