

Anotações Eapli:

Padrões

Grasp

List!

- Information Expert
- Low Coupling
- High Cohesion
- Creator
- Controller
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

definição mais aprofundada

- Information Expert

Problema:

qual é o princípio geral para a atribuição de responsabilidades aos objetos?

Solução:

Atribuir a responsabilidade ao information expert - a classe que contém a informação necessária para desempenhar essa responsabilidade

Information expert (also expert or the expert principle) is a principle used to determine where to delegate responsibilities. These responsibilities include methods, computed fields, and so on.

Using the principle of information expert, a general approach to assigning responsibilities is to look at a given responsibility, determine the information needed to fulfill it, and then determine where that information is stored.

Information expert will lead to placing the responsibility on the class with the most information required to fulfill it.[4]

- Low Coupling

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. Low coupling is an evaluative pattern that dictates how to assign responsibilities to support:

lower dependency between the classes,
change in one class having lower impact on other classes,
higher reuse potential.

- High Cohesion

High cohesion is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of low coupling. High cohesion means that the responsibilities of a given element are strongly related and highly focused. Breaking programs into classes and subsystems is an example of activities that increase the cohesive properties of a system. Alternatively, low cohesion is a situation in which a given element has too many unrelated responsibilities. Elements with low cohesion often suffer from being hard to comprehend, hard to reuse, hard to maintain and averse to change.[3]

- Creator

Creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes.

In general, a class B should be responsible for creating instances of class A if one, or preferably more, of the following apply:

Instances of B contain or compositely aggregate instances of A

Instances of B record instances of A

Instances of B closely use instances of A

Instances of B have the initializing information for instances of A and pass it on creation.

- Controller

The controller pattern assigns the responsibility of dealing with system events to a non-UI class that represents the overall system or a use case scenario. A controller object is a non-user interface object responsible for receiving or handling a system event.

A use case controller should be used to deal with all system events of a use case, and may be used for more than one use case (for instance, for use cases Create User and Delete User, one can have a single UserController, instead of two separate use case controllers).

It is defined as the first object beyond the UI layer that receives and coordinates ("controls") a system operation. The controller should delegate the work that needs to be done to other objects; it coordinates or controls the activity. It should not do much work itself. The GRASP Controller can be thought of as being a part of the application/service layer [2] (assuming that the application has made an explicit distinction between the application/service layer and the domain layer) in an object-oriented system with common layers in an information system logical architecture.

- Polymorphism

According to polymorphism principle, responsibility of defining the variation of behaviors based on type is assigned to the type for which this variation happens. This is achieved using polymorphic operations. The user of the type should use polymorphic operations instead of explicit branching based on type.

- Indirection

The indirection pattern supports low coupling (and reuse potential) between two elements by assigning the responsibility of mediation between them to an intermediate object. An example of this is the introduction of a controller component for mediation between data (model) and its representation (view) in the model-view-controller pattern.

- Pure Fabrication

A pure fabrication is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived (when a solution presented by the information expert pattern does not). This kind of class is called a “service” in domain-driven design.

- Protected Variations

The protected variations pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.

SOLID

List

- Single Responsibility Principle
- Open/Close
- Liskov Substitution Principle
- Interface Segregation
- Dependency Inversion

definição mais aprofundada

FONTE: <http://www.oodesign.com/design-principles.html> (<http://www.oodesign.com/design-principles.html>)
tava no moodle

- Single Responsibility Principle

A class should have only one reason to change.

In this context a responsibility is considered to be one reason to change. This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handle it. When we need to make a change in a class having more responsibilities the change might affect the other functionality of the classes.

Single Responsibility Principle was introduced Tom DeMarco in his book Structured Analysis and Systems Specification, 1979. Robert Martin reinterpreted the concept and defined the responsibility as a reason to change.

Ex: Class pessoa tem um Address.

Qual o responsavel de receber um endereço em String validar e subdividir para a class address?
o Address

- Open/Close

Software entities like classes, modules and functions should be open for extension but closed for modifications.

OPC is a generic principle. You can consider it when writing your classes to make sure that when you need to extend their behavior you don't have to change the class but to extend it. The same principle can be applied for modules, packages, libraries. If you have a library containing a set of classes there are many reasons for which you'll prefer to extend it without changing the code that was already written (backward compatibility, regression testing, etc). This is why we have to make sure our modules follow Open Closed Principle.

When referring to the classes Open Close Principle can be ensured by use of Abstract Classes and concrete classes for implementing their behavior. This will enforce having Concrete Classes extending Abstract Classes instead of changing them. Some particular cases of this are Template Pattern and Strategy Pattern.

- Liskov Substitution Principle

Derived types must be completely substitutable for their base types.

This principle is just an extension of the Open Close Principle in terms of behavior meaning that we must make sure that new derived classes are extending the base classes without changing their behavior. The new derived classes should be able to replace the base classes without any change in the code.

Liskov's Substitution Principle was introduced by Barbara Liskov in a 1987 Conference on Object Oriented Programming Systems Languages and Applications, in Data abstraction and hierarchy

- Interface Segregation

Many client specific interfaces are better than one general purpose interface.

Ex: em vez de uma interface impressora universal para os 80 clientes diferentes (scanner, publica, privada, etc), deveria haver algo:

Scanner, printer, printerGroup, etc

Fonte:

Clients should not be forced to depend upon interfaces that they don't use.

This principle teaches us to take care how we write our interfaces. When we write our interfaces we should take care to add only methods that should be there. If we add methods that should not be there the classes implementing the interface will have to implement those methods as well. For example if we create an interface called Worker and add a method lunch break, all the workers will have to implement it. What if the worker is a robot?



As a conclusion Interfaces containing methods that are not specific to it are called polluted or fat interfaces. We should avoid them.

- Dependency Inversion

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

Dependency Inversion Principle states that we should decouple high level modules from low level modules, introducing an abstraction layer between the high level classes and low level classes. Further more it inverts the dependency: instead of writing our abstractions based on details, the we should write the details based on abstractions.

Dependency Inversion or Inversion of Control are better know terms referring to the way in which the dependencies are realized. In the classical way when a software module(class, framework, ) need some other module, it initializes and holds a direct reference to it. This will make the 2 modules tight coupled. In order to decouple them the first module will provide a hook(a property, parameter, ) and an external module controlling the dependencies will inject the reference to the second one.

By applying the Dependency Inversion the modules can be easily changed by other modules just changing the dependency module. Factories and Abstract Factories can be used as dependency frameworks, but there are specialized frameworks for that, known as Inversion of Control Container.

Conceitos (tabelas para ligar dos exames):

- Decorator – Possível implementação do conceito de protected variation
- Singleton – Variável global do sistema que faz aumentar o acoplamento
- Layer – Segmentação lógica de responsabilidades.
- Alta Coesão – Medida que indica que os métodos de uma classe estão relacionados uns com os outros
- Adapter – Tradutor de interfaces incompatíveis
- Simple Factory – Classe cuja única responsabilidade é providenciar novos objetos.

- Repository – Permite diminuir o acoplamento entre a camada de domínio e as tecnologias de persistência
- Event – Representação de algo importante que sucedeu no sistema.
- Decorator - Associa dinamicamente responsabilidades adicionais a objetos
- Singleton - Garante acesso a uma única instância duma classe
- Version Control System - Sistema que tem como objetivo guardar a história do estado de uma coleção de ficheiros incluindo a funcionalidade de reverter a um qualquer desses estados.
- Iterator - Agrega e acede sequencialmente a elementos
- Strategy - Abstração para a seleção de um ou mais algoritmos
- Method Factory - Define uma interface para a criação dum objeto, mas permite que a subclasse decida que classe instanciar
- Refactoring - c. Processo de alteração do código de um sistema de software sem alterar o seu comportamento externo mas melhorando a sua estrutura.
- Abstract Factory - Fornece uma interface para criar famílias de objetos sem especificar as classes concretas.
- Entity - Objeto do mundo real identificado por um objeto identidade | Mantém a mesma identidade durante todo o seu tempo de vida
- Domain Service - Criado para realizar operações de negócio que não são naturalmente associadas a um objeto do domínio
- Value Object - Usa-se para quantificar ou descrever um conceito e deve ser implementado por um objeto imutável. | A sua identidade é baseada no seu estado e não em um objeto identidade.
- Domain Event - Acontecimento relevante que ocorre no domínio
- Aggregate Mantém relacionados um conjunto de objetos e controla o acesso externo a objetos desse conjunto via um objeto "root" | Grupo de objetos do domínio que podem ser tratados como uma unidade única.
- Domain Driven Design - Abordagem ao desenvolvimento de software caracterizada pela forte ligação da implementação a um modelo evolutivo dos conceitos fundamentais do negócio
- Domain Model - Serve de inspiração para as classes do domínio a criar | Promove uma colaboração criativa entre especialistas do negócio e técnicos de software no refinamento de conceitos
- Observer - Permitir a notificação de um objeto pela ocorrência de determinado evento.

Padrões e definição:

- Observer:

O padrão observer diminui o acoplamento entre observador e observado pois a ligação entre eles é estabelecida dinamicamente, sendo que o observador não “sabe” quem (se alguém) o está a observar. Como exemplo podemos considerar os use case de alertas do trabalho pratico 1 onde os objetos de domínio geravam eventos de domínio, ex., DespesaRegistada, que eram observados pelo “watchdog” de limites. Por sua vez esse watchdog gerava eventos que eram observados pela UI. Em nenhuma destas situações existia uma dependência (em tempo de compilação) entre as classes pelo que qualquer classe que implementa o padrão Observer pode observar qualquer outra classe que implemente o a responsabilidade Observable no padrão.

Example of JPA : atributos, lazy,etc

```
@Entity
@Table(Name=GRUPOAUTOMOVEL)
public class GrupoAutomovel {
    @Id //define a chave primária
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="IDGRUPOAUTOMOVEL")
    int idGrupoAutomovel;
    String nome;

    @Temporal(javax.persistence.TemporalType.DATE)
    private Date dataContrato;
    @OneToMany(mappedBy = "contrato", cascade = CascadeType.ALL)
    private List<CondutorAutorizado> listCondutorAutorizado=
        new ArrayList<CondutorAutorizado>();

    @ManyToOne(fetch=FetchType.LAZY)                /* LAZY - Contrato só será ca
    @JoinColumn(name="contrato_id")
    private ContratoAluguer contrato;

    @ElementCollection
    private Set<Telefone> telefones = new HashSet<>()

    ...
    public GrupoAutomovel(){} //obrigatório
```

Herança

SUPER class

```
@Entity
@Table (name="CLIENTE")
@Inheritance(strategy= InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISC")
public abstract class Cliente {
```

Classes que herdaram: NESTE CASO A BASE DE DADOS JUNTA TODOS OS PARAMETROS NA MESMA TABLE. TANTO DO CLIENTE TANTO DO ClienteParticular

```
@Entity
@DiscriminatorValue("PAR")
public class ClienteParticular extends Cliente{
    private String NIF;
    public ClienteParticular()
```

SUPER class para join: apesar de dar herança, na base de dados gera tables diferentes

```
@Entity
@Table(name="PAGAMENTO")
@Inheritance(strategy=InheritanceType.JOINED)
public class Pagamento {
```

herança

```
@Entity
@Table(name="PAGAMENTOCHIQUE")
public class PagamentoCheque extends Pagamento {
```

Locking:

There are two strategies for preventing concurrent modification of the same object/row.

- Optimistic

Optimistic locking assumes that the data will not be modified between when you read the data until you write the data.

This is the most common style of locking used and recommended in today's persistence solutions. The strategy involves checking that one or more values from the original object read, are still the same when updating it. This verifies that the object has not changed by another user in between the read and the write.

Optimistic Locking - JPA

- The `@Version` annotation or `<version>` element is used to define the optimistic lock version field.
- The annotation is defined on the version field or property for the object, similar to an Id mapping. The object must contain an attribute to store the version field.

```
@Entity
public abstract class Employee{
    @Id
    private long id;

    @Version
    private long version; ← version field
    ...
}
```

Prevents conflicts between concurrent business transactions by detecting a conflict and rolling back the transaction

```
@Entity
public abstract class Employee{
    @Id
    private long id;
    @Version
    private long version;    //version field used to Optimistic locking
    ...
}
```

Assume que os dados não são modificados entre o momento de leitura e escrita;

É o tipo de locking mais comum e recomendado nas soluções de persistências atuais;

Envolve a verificação se um ou mais valores do objeto original lido são iguais no momento de atualização;

Verifica se o objeto não foi alterado por outro utilizador entre a leitura e escrita;

Evita conflitos entre transações a decorrer detetando um conflito e fazendo o “rol back” da transação.

- Pessimistic locking.

Pessimistic locking means acquiring a lock on the object before you begin to edit the object, to ensure that no other users are editing the object.

Pessimistic locking is typically implemented through using database row locks, such as through the SELECT ... FOR UPDATE SQL syntax.

The data is read and locked, the changes are made and the transaction is committed, releasing the locks.

The main issues with pessimistic locking is they use database resources, so require a database

transaction and connection to be held open for the duration of the edit.

This is typically not desirable for interactive web applications.

The main advantages of pessimistic locking is that once the lock is obtained, it is fairly certain that the edit will be successful. This can be desirable in highly concurrent applications, where optimistic locking may cause too many optimistic locking errors.

Prevents conflicts between concurrent business transactions by allowing only one business transaction at a time to access data

locking modes:

- READ (JPA1)
- WRITE (JPA1)
- OPTIMISTIC (Synonymous to READ)
- OPTIMISTIC_FORCE_INCREMENT (Synonymous to WRITE)
- PESSIMISTIC_READ (JPA2)
- PESSIMISTIC_WRITE (JPA2)
- PESSIMISTIC_FORCE_INCREMENT (JPA2)
- NONE

Ex de código:

```
// Transaction t1 begin
t1.begin();
// Employee entity is read from the DB
Employee e = manager.find(Employee.class, 1);
// Lock is performed after read
manager.lock(e, LockModeType.PESSIMISTIC_READ);
// Transaction is committed
t1.commit();
```

em portugues pesismist locking

“Tranca” o objeto no momento de editar para ter a certeza que mais nenhum utilizador edita o mesmo objeto;

É tipicamente usado em bases de dados que utilizam o “row lock” como na sintaxe “SELECT ... FOR UPDATE SQL”;

Os dados são lidos e “trancados”, são realizadas alterações, faz-se commit e “destranca-se” os dados;

Os maiores obstáculos são a utilização de recursos da base de dados, necessitam de uma transação e conexão continua à base de dados durante a edição dos dados;

A principal vantagem é que uma vez trancado, temos a certeza que a edição é bem sucedida; Evita conflitos entre transações a acontecer, uma vez que apenas permite uma transação de cada vez.

Lazy vs Eager loading

Default é Eager, logo nao vou meter um exemplo

Ex codigo lazy

```
@Entity
public class CondutorAutorizado {
    @Id
    @GeneratedValue
    private int idCondutor;
    private String nome;
    private String numeroCartaConducao;
    @ManyToOne(fetch=FetchType.LAZY)    // Contrato só será carregado quando for
    @JoinColumn(name="contrato_id")
    private ContratoAluguer contrato;
```

Ciclo de vida das entidades:

Estados:

- Estado Transient ou New

Quando um objeto entidade é criado o seu estado é Transient ou New. Neste estado o objeto existe em memória mas não está associado com um EntityManager nem tem representação na base de dados.

- Estado Managed (persist)

Um objeto entidade passa para o estado Managed quando é persistido na base de dados (através do método persist de um EntityManager) ou quando é carregado da base de dados (através do método find de um EntityManager, ou da execução de uma query).

- Estado Managed (find)

No estado managed as entidades têm uma identidade persistente, uma chave que identifica univocamente cada instância. Se uma entidade managed é modificada dentro de uma transação, é marcada pelo EntityManager como dirty, e as modificações são atualizadas na base de dados no commit da transação.

- Estado Detached (detach)

Quando efetuamos várias alterações, não seguidas, ao estado de um objeto, para evitar a execução de vários updates, podemos passar a entidade para o estado detached (não gerido). No fim das alterações devemos passar a entidade para o estado managed usando o método merge.

- Estado Managed (merge) – funcionamento incorreto

O método merge leva uma entidade como parâmetro, cria uma nova entidade com os valores da base de dados, coloca essa entidade no estado managed, atribui-lhe os atributos diferentes da entidade parâmetro e retorna a entidade criada

- Estado Managed (merge) – funcionamento correto

A entidade criada pelo método merge é colocada no estado managed, em seguida o seu estado é alterado sendo marcada como dirty, e as modificações são atualizadas na base de dados no commit da transação.

Entity

É um objecto do domínio, uma classe normal em Java, com metadata para descrever como o seu estado é mapeado em tabelas relacionais.

Normalmente uma Entity representa um registo de uma tabela na base de dados relacional. Tem uma “persistence entity”

persistence entity

que corresponde à chave primária da tabela na BD

Entity manager

é a classe principal da API

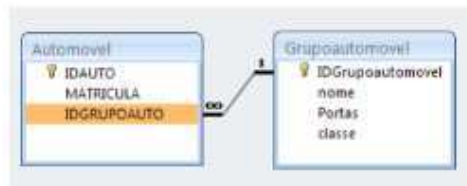
É com este objecto que se criam novas entidades, se criam queries para retornar conjuntos de entidades existentes, é através deste objecto que se realiza o CRUD na BD.

Associação many-to-one - JPA

```
@Entity
public class Automovel {
    @Id
    @GeneratedValue
    private Long id;
    private String matricula;

    private GrupoAutomovel grupo;

    public Automovel() {
    }
    ....
}
```



GrupoAutomovel
existe independente
dos Automoveis

Associação One-to-many

- Anotações
- ContratoAluguer > Lista de CondutoresAutorizados (bidireccional)

```
@Entity
public class ContratoAluguer {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int idContrato;

    @Temporal(javax.persistence.TemporalType.DATE)
    private Date dataContrato;
```

CondutorAutorizado é
parte integrante de
um Contracto

```
@OneToMany(mappedBy = "contrato", cascade = CascadeType.ALL)
private List<CondutorAutorizado> listCondutorAutorizado=
    new ArrayList<CondutorAutorizado>();
```

Associação One-to-many

@Entity

```
public class CondutorAutorizado {
```

@Id

@GeneratedValue

```
private int idCondutor;
```

```
private String nome;
```

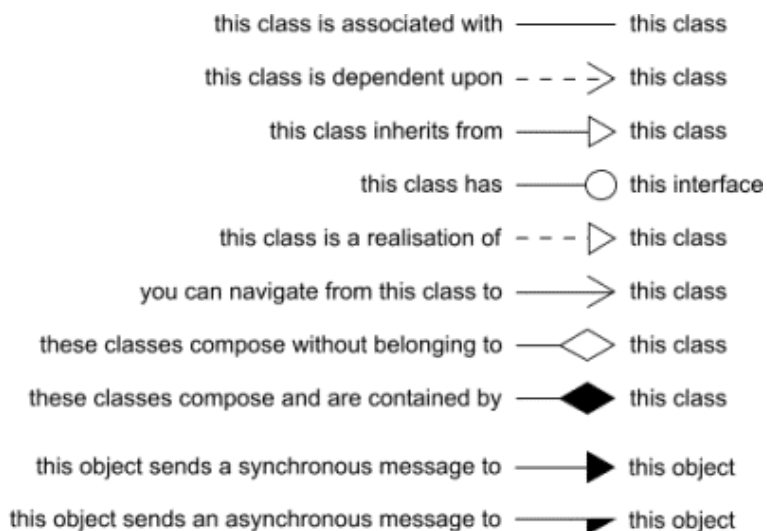
```
private String numeroCartaConducao;
```

@ManyToOne(fetch=FetchType.LAZY) *

@JoinColumn(name="contrato_id")

```
private ContratoAluguer contrato;
```

*** LAZY - Contrato só será carregado quando for pedido.**



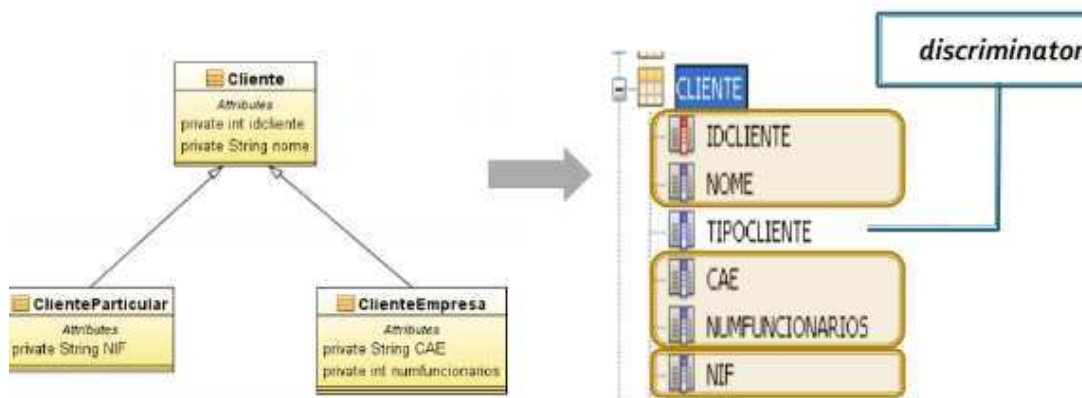
Mapeamento de herança entre classes

3 Modos:

- **Single Table** – Uma tabela para toda a hierarquia de classes (polimorfismo é obtido pela “desnormalização” do modelo relacional e introdução de um campo (“discriminator”) para definir o tipo de classe.)

Herança – JPA – Estratégia SINGLE_TABLE

- **Uma tabela para toda a hierarquia de classes: SINGLE_TABLE**
 - Todas as classes são mapeadas numa única tabela.
 - A tabela contém todas as propriedades de todas as classes.
 - Acrescenta-se na tabela um “discriminator” – TIPOCLIENTE - que indica qual a classe concreta do registo



Herança – JPA – Estratégia SINGLE_TABLE

- **Uma tabela para toda a hierarquia de classes**
- **Superclass Cliente**

```
@Entity
@Table (name="CLIENTE")
@Inheritance(strategy= InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISC")
public abstract class Cliente {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    int id;

    String nome;
    ...
    public Cliente () {}
    ...
}
```

Herança – JPA – Estratégia SINGLE_TABLE

Classes Derivadas

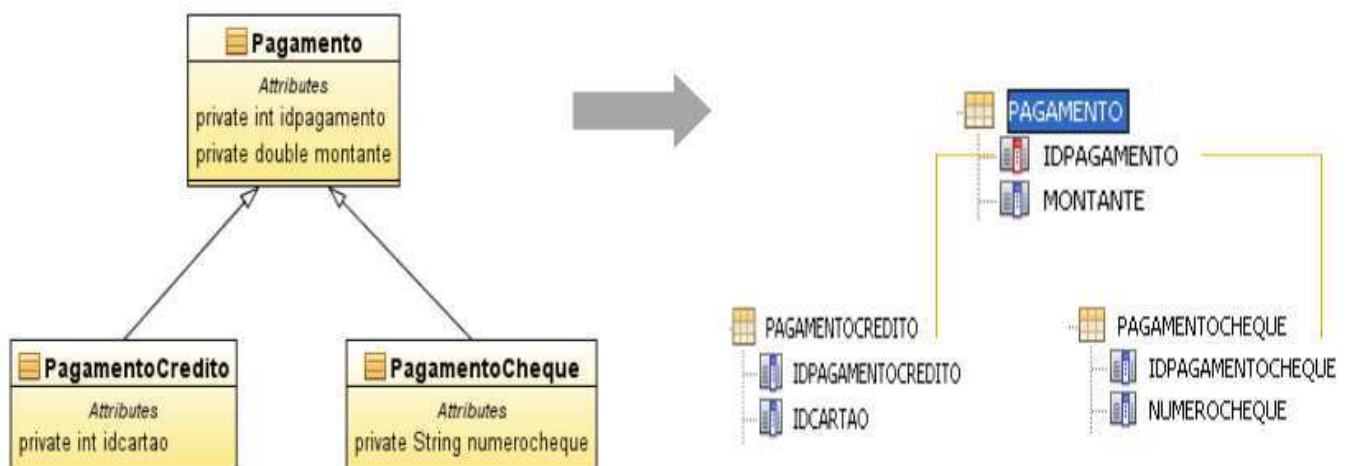
```
@Entity
@DiscriminatorValue("EMP")
public class ClienteEmpresa extends Cliente{
    private String CAE;
    public ClienteEmpresa()
    ....
}
```

```
@Entity
@DiscriminatorValue("PAR")
public class ClienteParticular extends Cliente{
    private String NIF;
    public ClienteParticular()
    ...
}
```

- **Joined** - Uma tabela por cada classe (representa a relação “is-a” por uma relação “has-a” (com chaves estrangeiras)

Herança – JPA – Estratégia JOINED

- **Uma tabela para cada classe da hierarquia: JOINED**
 - Cada classe é mapeada numa tabela
 - A tabela mapeamento de cada classe contém:
 - colunas apenas para as propriedades declaradas na classe (não herdadas)
 - Uma chave primária que é também uma chave estrangeira para a tabela da super classe



Herança – JPA – Estratégia JOINED

- Uma tabela para cada classe
- Super class Pagamento

```
@Entity
@Table(name="PAGAMENTO")
@Inheritance(strategy=InheritanceType.JOINED)
public class Pagamento {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="IDPAGAMENTO")
    protected int idPagamento;
    protected double montante;

    public Pagamento() {}
    void setMontante(double montante) {
        this.montante=montante;
    }
}
```

Herança – JPA – Estratégia JOINED

```
@Entity
@Table(name="PAGAMENTOCHIQUE")
public class PagamentoCheque extends Pagamento {
    private String numeroCheque;
    public PagamentoCheque( ){ }
    public PagamentoCheque(String numeroCheque,double montante) {
        super.setMontante(montante);
        this.numeroCheque=numeroCheque;
    } ....
```

```
@Entity
@Table(name="PAGAMENTOCREDITO")
public class PagamentoCredito extends Pagamento {
    private String idCartao;

    public PagamentoCredito( ){ }
    ....
```

Exemplo de código

```
EntityManager manager=PersistenceInit.getEntityManager();
PagamentoCheque pagCheque = new PagamentoCheque("12345678",200);
PagamentoCredito pagCredito = new PagamentoCredito("xpto12345",300);
manager.getTransaction().begin();
manager.persist(pagCheque); ...
```

- **Table per class** - Uma tabela para cada classe concreta (herança e polimorfismo não são considerados, nem representados no modelo relacional.)

Herança – JPA – Estratégia TABLE_PER_CLASS

■ Uma tabela por classe concreta: TABLE_PER_CLASS

Super-class Cliente:

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Cliente {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int idCliente;
    private String nome;
    . . .
}
```

Sub-class ClienteEmpresa:

```
@Entity
public class ClienteEmpresa extends Cliente {
    private String CAE;
    private int numFuncionarios;
    . . .
}
```

Sub-class ClienteParticular:

```
@Entity
public class ClienteParticular extends Cliente {
    private String NIF;
    . . .
}
```

Fetching:

- Lazy-> Fetch acontece quando ocorre um pedido;
- Eager-> Fetch acontece quando a Entity é carregada (por defeito)
-

Propriedades podem ser Carregadas pelo Lazy

Objectos Embedded

Úteis para representar classes não-entidade

Os campos de dados são embebidos dentro da tabela da entidade a que pertencem

Supportam o conceito de Value Object de Domain Driven Design

Um objeto Embedded pode conter outros objetos Embedded

Objetos Embedded

Exemplo: Uma Pessoa tem vários objetos Telefone

```
@Entity
public class Pessoa {
    @Id @GeneratedValue
    private Long id;
    private String nome;
    @ElementCollection
    private Set<Telefone> telefones = new HashSet<>();
    ...
}
```

```
@Embeddable
public class Telefone {
    private String numero;
    private String indicativoPaís;
    ...
}
```

```
public static void main(String[] args) {
    ...
    Pessoa p1 = new Pessoa("Manuel");
    Telefone t1 = new Telefone("123456789", "+351");
    Telefone t2 = new Telefone("987654321", "+49");
    Telefone t3 = new Telefone("234234234", null);
    p1.addTelefone(t1); p1.addTelefone(t2);
    p1.addTelefone(t3);
    em.getTransaction().begin();
    em.persist(p1);
    em.getTransaction().commit();
    em.close();
    emf.close();
}
```

A anotação **ElementCollection** atributo `Set<Telefone> telefones` necessária porque é uma coleção.

Tabelas criadas:

PESSOA(ID, NOME)	
1, Manuel	
PESSOA_TELEFONES(INDICATIVOPAIÍS, NUM)	
+351, 123456789	1
<NULL>, 234234	1
+49, 9876543	1

DTO

Isolam o modelo de domínio da apresentação, tendo como resultado:

Um baixo acoplamento

Transferência de dados otimizada

Uma maior flexibilidade no design de toda a aplicação quando existem alterações aos requisitos.

Vantagens vs Desvantagens / Alternativa

Vantagens:

Permite a utilização de estruturas diferentes das entidades de domínio.

Impede a exposição de informação desnecessária.

Pode evitar problemas de lazy-load na apresentação.

Desvantagens:

Ter centenas de entidades no modelo de domínio é normalmente um bom indicador para se considerar alternativas.

Se tivermos uma solução DTO 100% pura irá conduzir uma maior complexidade de classes extra.

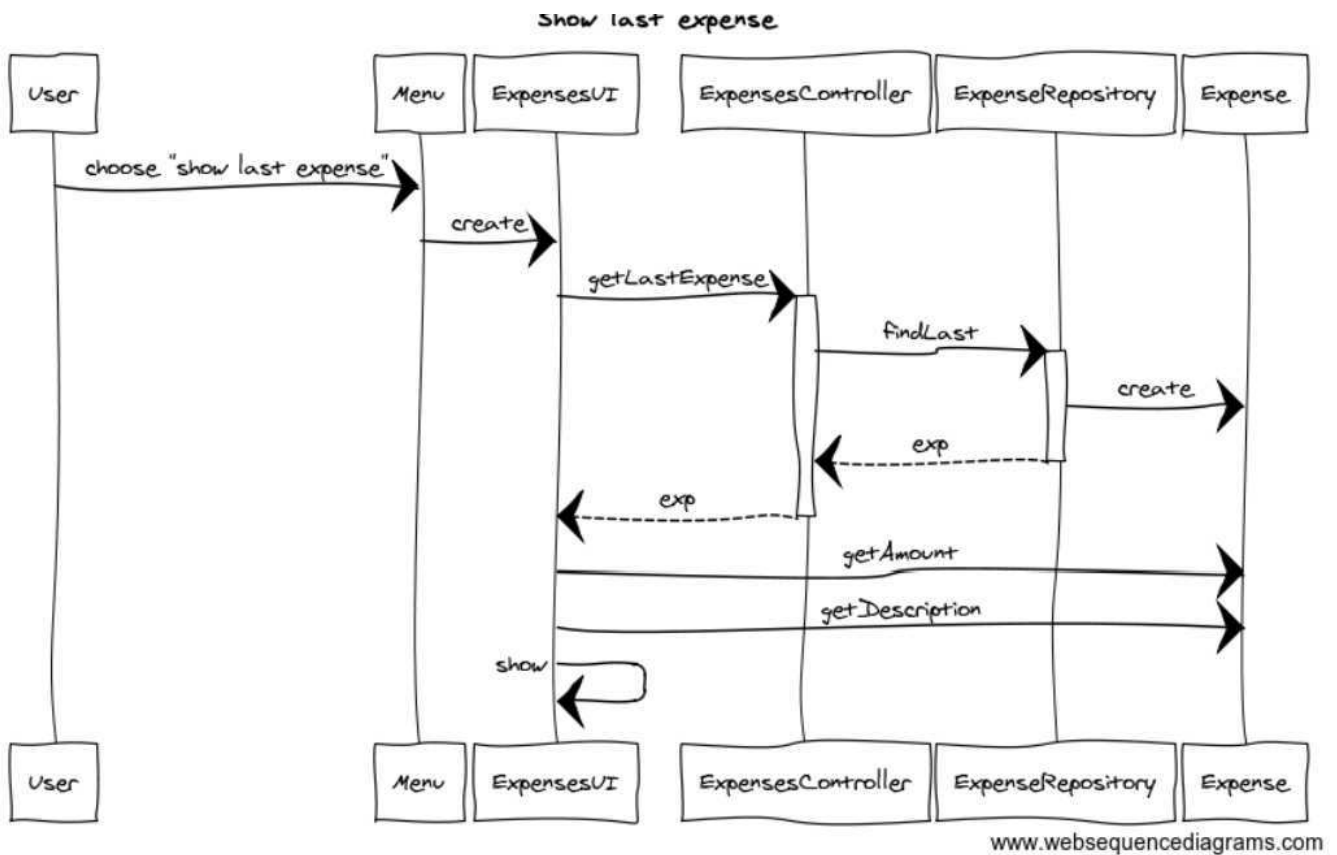
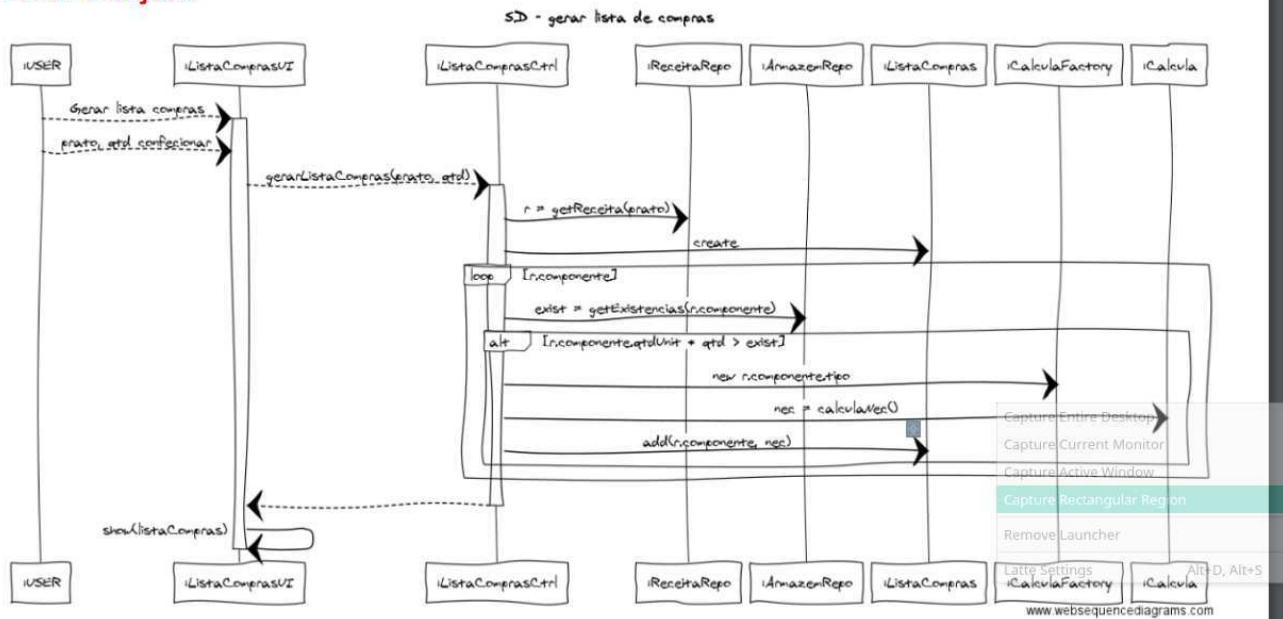
Uma solução DTO traz um trabalho adicional

Alternativa:

fazer referência ao modelo de domínio na camada de apresentação. O que vai provocar um maior acoplamento entre camadas, que poderá provocar um problema ainda maior.

DS Example:

Uma solução:



Register simple expense

