

GRASP

1 O que é?

General Responsibility Assignment Software Patterns: é um conjunto de **nove princípios** para atribuição de responsabilidades em Design Orientado a Objetos (OO Design).

2 Por que é importante?

- ✓ **High Cohesion facilita manutenção e reutilização** – Uma classe bem coesa executa um conjunto de funções relacionadas e evita sobrecarga.
 - ✓ **Low Coupling reduz dependências** – Se uma classe depende pouco de outras, alterações num módulo não afetam fortemente o resto do sistema.
 - ✓ **Melhora a modularidade** – O sistema torna-se mais flexível, escalável e menos propenso a erros em mudanças futuras.
 - ✓ **Separa responsabilidades** – Organizar melhor a comunicação entre objetos.
-

3 Quais são os 9 padrões GRASP?

1 Information Expert

📌 **Quem deve ter uma determinada responsabilidade?**

✓ Atribuir a responsabilidade à **classe que já possui a informação necessária** para desempenhá-la.

◆ **Exemplo:** Quem deve calcular a média das notas? A classe `ALuno` porque já tem a informação necessária (o atributo notas).

2 Creator

📌 **Quem deve ser responsável por criar um objeto?**

✓ Atribuir a criação a uma classe que:

- Contém ou agrega instâncias do objeto.
- Regista instâncias do objeto.
- Usa frequentemente o objeto.

◆ **Exemplo:** `Turma` contém `Alunos`. A `Turma` deve criar os objetos `Aluno` porque é ela que contém esses alunos.

3 Controller

📌 Quem deve gerir eventos do utilizador e coordenar ações do sistema?

✓ Criar um **Controller** intermediário entre a UI e as classes de domínio.

4 Pure Fabrication

📌 E se nenhuma classe do domínio for adequada para assumir uma responsabilidade?

✓ Criar uma **classe auxiliar "artificial"** para evitar sobrecarregar classes do domínio.

◆ **Exemplo:** `AlunoRepository` é uma classe separada que lida com a persistência da `Aluno`.

5 Low Coupling

📌 Como reduzir dependências entre classes?

✓ As classes devem depender minimamente umas das outras para facilitar manutenção e reutilização.

◆ **Exemplo:** `OrderService` usa `OrderRepository` em vez de aceder diretamente à base de dados.

6 High Cohesion

📌 Como manter classes focadas numa única responsabilidade?

✓ Cada classe deve ter **funções bem relacionadas**, evitando misturar lógicas distintas.

◆ **Exemplo:** `Invoice` deve apenas gerar faturas, sem processar pagamentos.

7 Polymorphism

📌 Como evitar condicionais baseadas em tipos?

✓ O **polimorfismo em GRASP** permite que **diferentes classes tenham o mesmo método**, mas com implementações distintas. **Garante flexibilidade**,

pois o código trabalha com abstrações (superclasse ou interface) e não com classes concretas.

◆ **Exemplo:** `Payment` pode ter subclasses `CreditCardPayment` e `PayPalPayment`.

8 Indirection

📌 **Como reduzir dependências diretas entre classes?**

✓ Criar **uma classe intermediária** para gerir comunicações.

◆ **Exemplo:** `PaymentGateway` abstrai chamadas diretas para diferentes serviços de pagamento.

9 Protected Variations


📌 **Como proteger o sistema contra mudanças inesperadas?**

✓ Encapsular partes instáveis do código para reduzir impacto de alterações.

◆ **Exemplo:** Queremos permitir diferentes métodos de pagamento no futuro (PayPal, Stripe, MBWay): Criamos uma interface Pagamento, permitindo adicionar novos métodos sem afetar o código existente.

◆ Relação entre "Extends vs. Implements vs. Association" e o Polimorfismo (GRASP)

- **Extends** (Herança) → Classe B herda de A (`class B extends A`).
 - ✓ Herança permite polimorfismo porque uma subclasse pode **sobrescrever métodos** da superclasse e ser usada no lugar dela.
 - ✗ Alto acoplamento → Se A muda, B pode ser afetada.
- **Implements** (Interface) → Classe B implementa interface A (`class B implements A`).
 - ✓ Baixo acoplamento. Implementação de interfaces **é a melhor forma de aplicar polimorfismo** porque permite que classes diferentes sigam um mesmo contrato sem criar dependências diretas.
- **Association** → Classe B tem um objeto de A (`class B { private A obj; }`).

-  Relação fraca entre classes, favorecendo flexibilidade.

Melhor prática: Sempre que possível, usar **Interfaces** para reduzir acoplamento.