Princípios GRASP em Design de Software Orientado a Objetos

Introdução

Os **princípios GRASP** (General Responsibility Assignment Software Patterns) são diretrizes fundamentais para atribuir responsabilidades a classes e objetos em projetos de software orientado a objetos. Eles ajudam a criar sistemas com **baixo acoplamento**, **alta coesão** e **flexibilidade**, seguindo boas práticas de design. Abaixo está uma explicação resumida dos principais princípios GRASP, com exemplos práticos:

1 Princípios GRASP

1.1 Creator (Criador)

- **Problema**: Quem deve criar uma instância de uma classe?
- **Solução**: Atribua a responsabilidade de criação à classe que:
  - Contém ou agrega a classe a ser criada (ex.: `Sale` cria `SalesLineItem`)
  - Possui dados necessários para inicializar o objeto
  - Registra ou usa intensamente o objeto
- **Exemplo**: No sistema de venda (POS), a classe `Sale` cria `SalesLineItem`, pois agrega itens de venda.

1.2 Information Expert (Especialista na Informação)

- **Problema**: Qual classe deve ser responsável por uma tarefa?
- **Solução**: Atribua a responsabilidade à classe que possui **as informações necessárias** para executá-la.
- **Exemplo**:
  - O cálculo do total de uma venda (`getTotal()`) é atribuído à classe `Sale`, pois ela conhece todos os `SalesLineItem` e pode somar seus subtotais
  - Cada `SalesLineItem` calcula seu próprio subtotal, pois conhece a quantidade e o preço do produto (via `ProductSpecification`)

1.3 Low Coupling (Baixo Acoplamento)

- **Problema**: Como reduzir dependências entre classes?
- **Solução**: Minimize conexões diretas entre classes.
- **Exemplo**:
  - Em vez de `Register` criar `Payment`, a classe `Sale` assume essa responsabilidade, pois já está acoplada a `Payment` para calcular o saldo
  - Evitar que `Dog` acesse diretamente `Square` no jogo Monopoly; delegar a responsabilidade a `Board`

1.4 High Cohesion (Alta Coesão)

- **Problema**: Como manter classes focadas e gerenciáveis?
- **Solução**: Atribua responsabilidades relacionadas à mesma finalidade.
- **Exemplo**:
  - Uma classe `Matrix` com métodos para decomposição LU, Cholesky e SVD tem baixa coesão. Melhor dividir em classes especializadas (`CholeskyDecomposition`, `Inverse`)

1.5 Controller (Controlador)

- **Problema**: Quem deve lidar com eventos de sistema (ex.: interações do usuário)?
- **Solução**: Use uma classe intermediária (como `ReceiptController` no POS) para separar a interface do usuário da lógica de negócio.
- **Exemplo**: No jogo Monopoly, um controlador (`MonopolyGameController`) gerencia a sequência de jogadas, isolando a interface gráfica.

1.6 Polymorphism (Polimorfismo)

- **Problema**: Como tratar variações de comportamento sem usar condicionais?
- **Solução**: Use métodos polimórficos em interfaces ou classes abstratas.
- **Exemplo**:
  - Classes `Triangle`, `Circle` e `Square` implementam a interface `Shape2D` com métodos `area()` e `perimeter()`, evitando verificações de tipo com `if`

1.7 Pure Fabrication (Fabricação Pura)

- **Problema**: Como atribuir responsabilidades que não pertencem ao domínio do problema?
- **Solução**: Crie classes artificiais para tarefas técnicas (ex.: persistência).
- **Exemplo**:
  - `PersistentStorageBroker` salva objetos `Sale` em um banco de dados, evitando acoplar `Sale` a detalhes de persistência

1.8 Indirection (Indireção)

- **Problema**: Como evitar acoplamento direto entre componentes?
- **Solução**: Introduza um intermediário.
- **Exemplo**:
  - Um `CreditAuthorizationService` atua como intermediário entre o sistema de pagamento e APIs externas de comunicação

1.9 Protected Variations (Variações Protegidas)

- **Problema**: Como proteger o sistema contra mudanças em componentes instáveis?
- **Solução**: Isole pontos de variação com interfaces estáveis.
- **Exemplo**:
  - Usar a interface `ImageSaver` para salvar imagens em formatos diferentes (JPEG, PNG), permitindo adicionar novos formatos sem modificar a classe `Image`

2 Relação com Outros Princípios

- **SOLID**: GRASP complementa princípios como **Single Responsibility** (High Cohesion) e **Dependency Inversion** (Low Coupling)
- **Liskov Substitution Principle (LSP)**: Garante que subtipos sejam substituíveis por seus tipos base (ex.: evitar que `Square` herde de `Rectangle`)
- **Law of Demeter**: Evita acoplamento excessivo (ex.: `Company` não deve acessar `Employee` diretamente, mas via `Department`)

3 Aplicação Prática

3.1 Exemplo POS

- `Sale` calcula o total usando `SalesLineItem` (Information Expert)
- `PersistentStorageBroker` gerencia persistência (Pure Fabrication)
- `ReceiptController` coordena interações (Controller)

3.2 Exemplo POS

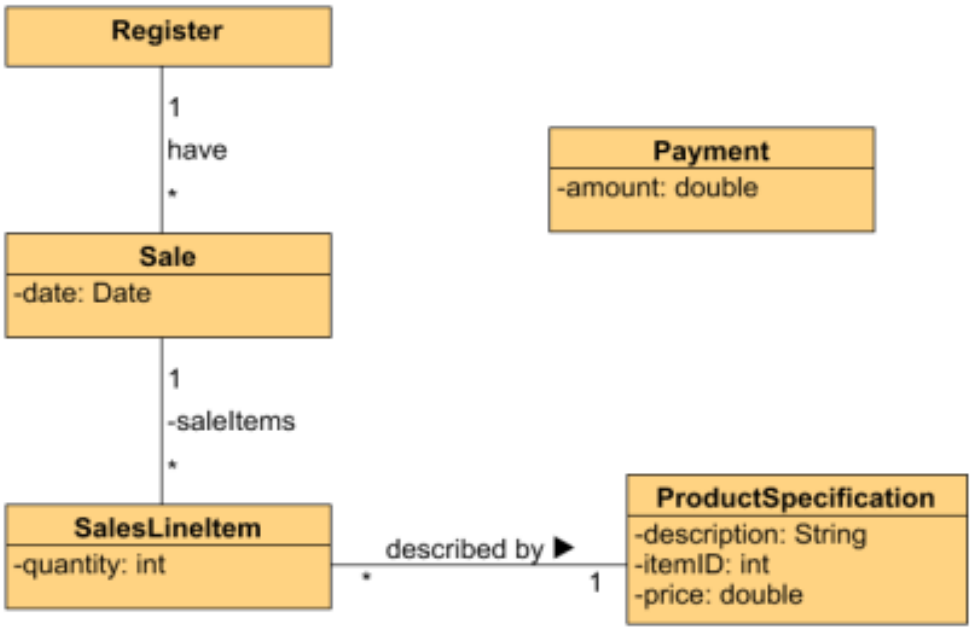

Figura 1: Diagrama de classes do sistema de venda (POS)

- `Sale` calcula o total usando `SalesLineItem` (Information Expert)
- `PersistentStorageBroker` gerencia persistência (Pure Fabrication)
- `ReceiptController` coordena interações (Controller)

3.3 Exemplo Monopoly

- `Board` cria `Square` (Creator)
- `Board` gerencia a posição das peças (Low Coupling)

3.4 Exemplo Monopoly

- `Board` cria `Square` (Creator)
- `Board` gerencia a posição das peças (Low Coupling)

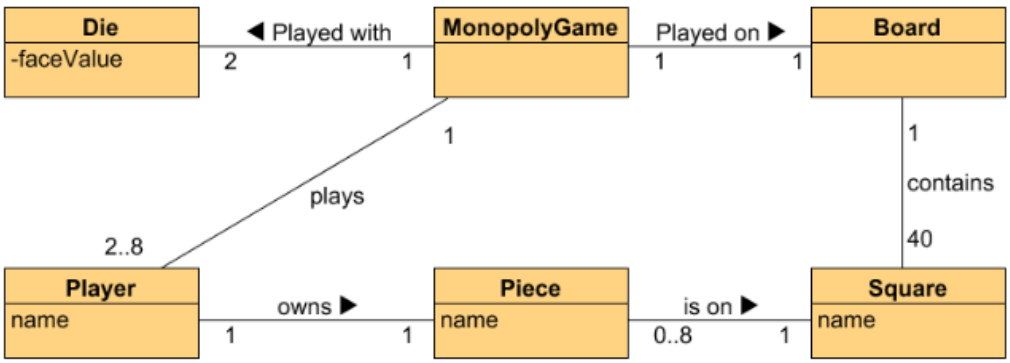

Figura 2: Diagrama de classes do jogo Monopoly

4 Conclusão

Os princípios GRASP são **ferramentas essenciais** para tomar decisões de design orientadas a objetos. Eles promovem sistemas **flexíveis**, **mantíveis** e **reutilizáveis**, alinhando-se com práticas modernas como SOLID e padrões de arquitetura. A chave está em **equilibrar** os princípios conforme o contexto, evitando sobreengenharia e focando em soluções simples e eficazes.