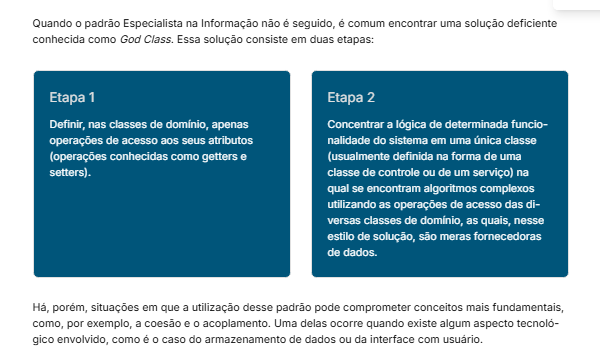
Módulo 1

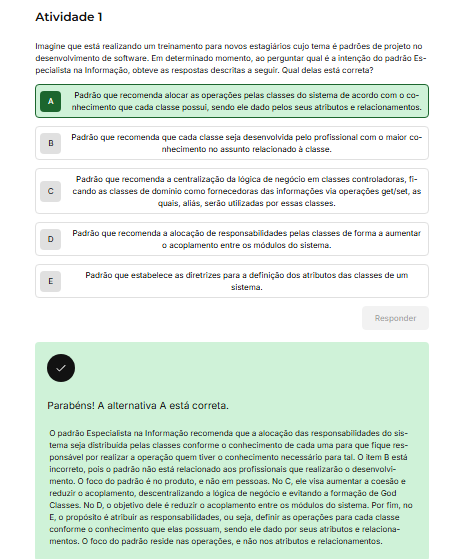
**O padrão Especialista na Informação**

Ele se concentra na atribuição de responsabilidades dentro de um sistema orientado a objetos. Ou seja, recomenda-se que a responsabilidade de realizar determinada tarefa seja atribuída à classe que possui a maior parte das informações necessárias para cumprir essa tarefa.

Atribua a **responsabilidade** ao especialista, isto é, ao módulo que possua o conhecimento necessário para realizá-la.

**Consequências do Especialista na Informação**

****



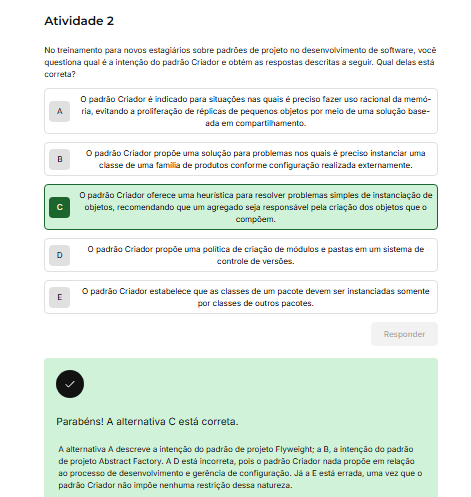
**O padrão Criador**

Atribui a responsabilidade de criar uma instância de uma classe à classe que contém, agrega, usa ou possui a informação necessária para inicializar essa nova instância. Isso promove o encapsulamento e reduz o acoplamento entre classes, pois a classe criadora tem um relacionamento estreito ou natural com aquela que será instanciada. O padrão Criador facilita a manutenção e a extensibilidade do sistema, evitando a dispersão da lógica de criação de objetos e promovendo uma organização mais clara e coesa no design orientado a objetos.

**Consequências do Criador**

O padrão Criador é especialmente indicado para a criação de instâncias que formam parte de um agregado, pois o elemento que controla o ciclo de vida das suas partes é o próprio agregado, o qual, aliás, naturalmente já está relacionado com as suas partes. Esse padrão não é apropriado em algumas situações especiais, como é o caso da criação condicional de uma instância de uma família de classes similares.

De forma geral, quando a instanciação de objetos envolver cenários mais complexos, como o compartilhamento de objetos para racionalizar o uso de memória ou a criação de uma instância de uma família de classes similares condicionada ao valor de alguma configuração externa, será mais adequado aplicar padrões de projeto específicos, como os padrões GoF **Abstract Factory**, **Builder**, **Prototype** ou **Factory Method**.



**O padrão Coesão Alta**

O que é Coesão?

Coesão é o grau em que os elementos de um módulo — seja um método, classe, pacote ou subsistema — estão relacionados e trabalham juntos para cumprir um único propósito.

Quanto maior a coesão, melhor: o módulo é mais simples, mais fácil de entender, manter, reutilizar e menos propenso a ser afetado por mudanças.

Identificando o Problema

Módulos com baixa coesão costumam ter muitas responsabilidades desconectadas. Isso dificulta o entendimento do sistema, prejudica o reuso e torna o código mais frágil a alterações.

A pergunta-chave aqui é: como agrupar responsabilidades de forma que a complexidade do sistema seja controlável?

A Solução: Módulos com Alta Coesão

A ideia é simples: organizar os módulos para que todos os seus elementos sirvam a um único objetivo. E isso vale para diferentes níveis:

* Métodos: devem ter um propósito claro.
* Classes: devem reunir atributos e métodos relacionados.
* Pacotes: devem agrupar classes que trabalham com o mesmo conceito.
* Subsistemas: devem reunir pacotes coesos em torno de uma funcionalidade maior.

Tipos de Coesão

A seguir, vamos explorar os tipos de coesão — do mais fraco para o mais forte:

1. Coesão Coincidente  
   Elementos são agrupados por conveniência, sem relação real. Exemplo: uma classe Utils com funções de formatação, envio de arquivos e conversão de medidas. Baixíssima coesão.
2. Coesão Lógica  
   Funções relacionadas logicamente, mas que fazem coisas diferentes. Exemplo: uma classe que lê dados de várias fontes (arquivo, FTP, banco de dados).
3. Coesão Temporal  
   Elementos agrupados por acontecerem na mesma fase, como na inicialização do sistema. Exemplo: log, interface e banco de dados iniciados juntos.
4. Coesão Procedural  
   Agrupamento baseado em sequência de execução, mas com diferentes conjuntos de dados. Exemplo: uma classe que controla todo o fluxo de um pedido.
5. Coesão de Comunicação  
   Funções que operam sobre os mesmos dados. Exemplo: uma classe Conta com operações diversas sobre o mesmo saldo.
6. Coesão Sequencial  
   Cada passo usa o resultado do anterior. Exemplo: processar uma string, transformá-la em objeto e salvar no banco.
7. Coesão Funcional  
   O ideal. Todos os elementos contribuem diretamente para um único propósito. Exemplo: classes do pacote java.io, como FileReader, FileWriter, FileInputStream, etc.

Consequências da Coesão Alta

Módulos com alta coesão geram sistemas:

* Mais fáceis de manter e entender
* Mais fáceis de reaproveitar
* Mais robustos contra mudanças
* Mais propensos a ter baixo acoplamento

Comparando os Extremos

* Em sistemas com coesão alta, a complexidade é distribuída em diversos módulos pequenos e focados.
* Em sistemas com coesão baixa, poucos módulos concentram grande complexidade, resultando em milhares de linhas difíceis de entender e modificar.

Conclusão

Adotar o padrão de Coesão Alta significa organizar o código de forma clara, eficaz e sustentável. É um passo fundamental para criar softwares com melhor qualidade, facilidade de manutenção e evolução a longo prazo.

Interface gráfica do usuário, Texto, Aplicativo

O conteúdo gerado por IA pode estar incorreto.

**O padrão Controlador**

O que é?  
O padrão Controlador define um objeto intermediário entre a interface de usuário (UI) e o modelo de domínio. Ele encapsula a lógica de controle e coordenação, promovendo baixo acoplamento e facilitando manutenção, modularidade e reutilização.

Problema  
Como lidar com eventos gerados por atores externos ao sistema (ex: clique em “fechar pedido”)? A quem atribuir a responsabilidade de processar esses eventos?

Solução  
Atribuir a responsabilidade a uma classe Controladora, com duas opções principais:

1. Controlador Fachada  
Classe que representa o sistema ou subsistema como um todo. Útil quando há poucos eventos.

2. Controlador por Caso de Uso  
Classe que representa um caso de uso específico (ex: ServicoTransferencia). Centraliza o tratamento de eventos de um mesmo contexto, promovendo alta coesão.

Funcionamento

* A interface com o usuário detecta ações (mouse, teclado etc.).
* Ela gera uma requisição lógica de sistema.
* O Controlador recebe essa requisição e coordena os objetos necessários para gerar a resposta.
* O Controlador não implementa regras de negócio diretamente.

Exemplo  
No internet banking, o botão “transferir” gera uma requisição ao controlador ServicoTransferencia, que interage com Conta e ContaRepository para executar a lógica.

Benefícios

* Baixo acoplamento entre UI e lógica de negócio.
* Melhoria na coesão e separação de responsabilidades.
* Facilita a reutilização da lógica de controle em diferentes interfaces (web, mobile, desktop).

Cuidados  
Evitar sobrecarregar o controlador com responsabilidades demais (ex: aplicar regras de negócio diretamente). Ele deve apenas orquestrar os componentes corretos para cada evento.

Analogia  
Como um maestro em uma orquestra: comanda os músicos (componentes), mas não toca instrumentos (não executa a lógica do negócio).

Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto

O conteúdo gerado por IA pode estar incorreto.

**O padrão Acoplamento Baixo**

O que é Acoplamento Baixo?  
O padrão Acoplamento Baixo visa minimizar as interdependências entre módulos ou classes de um sistema, permitindo que mudanças em uma parte do sistema afetem o mínimo possível as demais. Isso é alcançado por meio de:

* Interfaces bem definidas.
* Comunicação direta e simples.
* Redução de dependências entre implementações concretas.

Objetivo do padrão

* Reduzir impacto de mudanças.
* Facilitar manutenção e evolução do sistema.
* Promover reuso e modularidade.
* Estimular dependência de abstrações e não de implementações.

Identificando o problema  
Sistemas com alto acoplamento enfrentam:

* Propagação de mudanças em cascata.
* Dificuldade de compreender módulos isoladamente.
* Baixo reuso, exigindo presença de módulos dependentes.

Como o acoplamento é definido?  
O acoplamento representa o grau de dependência entre dois módulos. Quanto mais dependente, mais acoplado. Os tipos de acoplamento, do mais alto para o mais baixo, são:

1. Acoplamento de conteúdo:
   * Um módulo acessa diretamente os dados internos de outro.
   * Exemplo: acesso direto a atributos públicos.
2. Acoplamento global:
   * Módulos compartilham variáveis globais.
   * Exemplo: uso de atributos public static em classes Java.
3. Acoplamento externo:
   * Módulos compartilham recursos externos como banco de dados.
   * Exemplo: duas classes acessando e dependendo da mesma tabela.
4. Acoplamento de controle:
   * Um módulo influencia a lógica interna de outro via parâmetros de controle.
   * Exemplo: cliente indicando a fonte de dados que outro módulo deve usar.
5. Acoplamento de estrutura:
   * Um módulo fornece uma estrutura grande e o outro utiliza apenas uma parte.
   * Exemplo: passar objeto Pedido para função que só usa o endereço.
6. Acoplamento de dados (preferível):
   * Módulos se comunicam passando apenas os dados necessários.
   * Exemplo: passar endereço diretamente para cálculo de frete.

Heurísticas para reduzir acoplamento

1. Reduzir o número de dependências, mantendo a coesão.
2. Preferir níveis mais baixos de acoplamento, idealmente o de dados.
3. Fazer com que implementações dependam de abstrações (interfaces), especialmente nas relações com tecnologia (UI, banco de dados, etc).

Como acoplamentos são criados (Java como exemplo)

* Atributos de uma classe têm tipo de outra classe.
* Métodos invocam diretamente métodos de outras classes.
* Criação de instâncias dentro da classe (new).
* Herança direta ou indireta.
* Implementação de interfaces.
* Compartilhamento de recursos (variáveis globais, arquivos, bancos).

Consequências positivas do acoplamento baixo

* Facilidade de manutenção.
* Alta capacidade de reuso.
* Redução de efeitos colaterais em mudanças.
* Sistema mais adaptável e testável.

Arquiteturas recomendadas

* Arquitetura Hexagonal (Ports and Adapters)
* Arquitetura Limpa (Clean Architecture)

Essas abordagens reforçam a separação entre a lógica de domínio e elementos externos, como tecnologia, garantindo baixo acoplamento.

Interface gráfica do usuário, Texto, Aplicativo, Email

O conteúdo gerado por IA pode estar incorreto.

**O padrão Polimorfismo**

O Que é Polimorfismo?

Polimorfismo permite que diferentes classes respondam de forma distinta à mesma chamada de método, desde que implementem uma interface ou herdem de uma classe base comum. Isso proporciona maior flexibilidade e extensibilidade ao software, além de eliminar a necessidade de estruturas condicionais complexas baseadas no tipo de objeto.

Cenário: O Problema com Integração Direta com APIs

Imagine que você está desenvolvendo um sistema de loja online que precisa se comunicar com diferentes administradoras de cartão de crédito para processar pagamentos. Cada administradora exige um processo complexo de homologação e fornece sua própria API.

Hoje em dia, existem intermediários chamados brokers de pagamento que facilitam essa integração. No entanto:

* Cada broker tem sua própria API com métodos e argumentos diferentes;
* Podem surgir novos brokers com taxas mais vantajosas;
* Lojistas precisam escolher o broker que melhor atende às suas necessidades.

🎯 O Problema Técnico

Você é responsável por entregar um sistema flexível para vários lojistas, permitindo a escolha do broker ideal. Porém, cada broker tem uma implementação diferente. Alguém sem conhecimento de polimorfismo provavelmente resolveria isso com código assim:

Exemplo simplificado (Java):

public class FechamentoPedido {  
public void processarPagamento(Pedido pedido) {  
if (broker == "Broker1") {  
broker1.efetuarPagamentoBroker1(pedido.getValor());  
} else if (broker == "Broker2") {  
broker2.realizarPagamentoBroker2(pedido.getDados());  
} else if (broker == "Broker3") {  
broker3.executarPagamentoBroker3(pedido);  
}  
}  
}

Agora imagine esse código com 20 brokers! Além de confuso, o sistema fica altamente acoplado às implementações específicas, dificultando a manutenção e evolução.

💡 A Solução: Usando Polimorfismo com Interface Genérica

A solução elegante está no uso de uma interface genérica chamada BrokerPagamento. Cada broker passa a ser representado por uma classe adaptadora que implementa essa interface e sabe lidar com sua API específica.

Etapas da solução:

1. Criar a interface BrokerPagamento:

public interface BrokerPagamento {  
void efetuarPagamento(Pedido pedido);  
}

1. Criar adaptadores para cada broker:

public class Broker1Adapter implements BrokerPagamento {  
public void efetuarPagamento(Pedido pedido) {  
// Lógica da API do Broker1  
}  
}

public class Broker2Adapter implements BrokerPagamento {  
public void efetuarPagamento(Pedido pedido) {  
// Lógica da API do Broker2  
}  
}

1. Usar a interface no FechamentoPedido:

public class FechamentoPedido {  
private BrokerPagamento broker;

cpp

CopiarEditar

public FechamentoPedido(BrokerPagamento broker) {

this.broker = broker;

}

public void processarPagamento(Pedido pedido) {

broker.efetuarPagamento(pedido); // Chamada polimórfica

}

}

💬 O Que Ganhamos com Isso?

* O código da classe FechamentoPedido não precisa mais saber como cada broker funciona;
* Basta adicionar um novo adaptador se um novo broker surgir, sem modificar o sistema principal;
* Redução de acoplamento e aumento da coesão;
* Facilita testes, manutenção e evolução do software.

🔁 Consequências e Benefícios do Polimorfismo

* Elimina a necessidade de switch-case ou if-else baseados em tipo;
* Permite o uso de módulos plugáveis (adiciona-se um novo broker sem modificar os existentes);
* Flexibiliza a substituição de implementações;
* Está presente em diversos padrões de projeto, como:
  + Adapter
  + Command
  + Composite
  + Proxy
  + State
  + Strategy

⚠️ Atenção: Use com Sabedoria!

Embora o polimorfismo traga flexibilidade, ele só deve ser aplicado quando há uma real possibilidade de variação. Criar estruturas genéricas para algo que nunca muda pode tornar o código mais complexo sem necessidade.

📌 Conclusão

O polimorfismo é uma ferramenta poderosa que permite escrever código mais limpo, desacoplado e flexível. Ele nos ajuda a lidar com diferentes implementações de maneira elegante, como vimos com os brokers de pagamento.

Interface gráfica do usuário, Texto, Aplicativo

O conteúdo gerado por IA pode estar incorreto.

**O padrão Invenção Pura**

O padrão **Invenção Pura** propõe a criação de **classes artificiais**, que não representam conceitos do domínio, mas servem para **alcançar alta coesão e baixo acoplamento**. Essas classes agrupam comportamentos específicos que não pertencem claramente a nenhuma entidade do domínio.

**Problema:**  
No desenvolvimento de um sistema de vendas, classes como Produto, Cliente e Pedido representam bem o domínio. A lógica de negócio, como o cálculo do total do pedido, deve ficar na classe Pedido, seguindo o padrão **Especialista**. Porém, salvar o pedido no banco de dados não pertence diretamente à lógica do domínio.

**Solução:**  
Criar uma classe como PedidoRepository para isolar a responsabilidade de persistência. Essa é uma **"invenção pura"**, que separa a lógica de negócio da lógica de armazenamento, promovendo **modularidade** e **facilidade de manutenção**.

**Consequências e estratégias:**

* A **decomposição por representação** cria classes que refletem o mundo real (como Pedido ou Cliente).
* A **decomposição por comportamento** cria classes baseadas em funcionalidades (como PedidoRepository ou políticas de desconto).
* Ambas as estratégias devem ser equilibradas. Uso excessivo da representação pode gerar classes inchadas com pouca lógica útil.

**Exemplos práticos:**  
Separar algoritmos de desconto do Pedido em classes específicas (ex.: DescontoNatal, DescontoFrequente) torna o sistema mais flexível e fácil de evoluir.

**Padrões GoF como Adapter, Strategy e Command** são exemplos de Invenção Pura, pois organizam responsabilidades por comportamento e não por representação direta do domínio.

Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto

O conteúdo gerado por IA pode estar incorreto.

**O padrão Indireção**

O padrão **Indireção** introduz um **intermediário** para delegar responsabilidades entre componentes, **facilitando a comunicação** e **reduzindo o acoplamento direto** entre classes. Isso promove **flexibilidade, modularidade** e **facilidade de manutenção** no design orientado a objetos.

**Problemas comuns resolvidos pela Indireção:**

1. **Comunicação entre objetos remotos:**  
   Uso do padrão **Proxy** para abstrair detalhes técnicos da comunicação em rede (sockets, serialização, etc.).
2. **Uso de APIs de terceiros:**  
   Uso do padrão **Adapter** para traduzir interfaces diversas de fornecedores em uma interface comum para o cliente.
3. **Abstração de tecnologia:**  
   Padrões como **Service Locator** e **Business Delegate** escondem as tecnologias usadas (EJB, REST, etc.) dos clientes, permitindo mudanças sem afetar o código cliente.
4. **Redução de acoplamento:**  
   Padrões como **Facade** e **Mediator** simplificam interações entre objetos, centralizando a comunicação e reduzindo dependências diretas.

**Consequências da Indireção:**

* **Desacopla cliente e fornecedor**, promovendo reutilização e facilidade de evolução.
* **Isola variações tecnológicas** e diferenças entre implementações.
* **É amplamente usada** em padrões de projeto, como Proxy, Adapter, Facade, Mediator, Service Locator e Business Delegate.
* Também aplicada em áreas como **balanceamento de carga** e **plataformas como serviço (PaaS)**.

**Cuidado:**  
Embora poderosa, a Indireção pode **impactar a performance** em sistemas críticos (ex.: jogos e sistemas em tempo real), se usada em excesso.

Interface gráfica do usuário, Texto, Aplicativo, Email

O conteúdo gerado por IA pode estar incorreto.

**O padrão Variações Protegidas**

O padrão **Variações Protegidas** isola comportamentos que podem mudar por meio de **interfaces e abstrações**, protegendo o sistema contra alterações em componentes internos ou externos. Ele permite que implementações específicas sejam alteradas ou substituídas **sem afetar o restante do sistema**, promovendo **flexibilidade, manutenção facilitada** e **extensibilidade**.

**Problemas comuns resolvidos pelo padrão:**

1. **Fragilidade estrutural do sistema:**  
   Pequenas mudanças exigem alterações em muitos módulos, devido ao alto acoplamento.
2. **Variações na interação com o usuário:**  
   Interfaces como app, web e caixas eletrônicos podem mudar frequentemente, devendo ser isoladas.
3. **Mudanças nas regras de negócio:**  
   Regras de segurança ou políticas bancárias mudam com frequência e devem ser encapsuladas para evitar impactos amplos.
4. **Dependência de implementações específicas:**  
   Exemplo: uma classe que usa diretamente um banco de dados relacional não consegue se adaptar facilmente ao uso de um banco NoSQL.

**Solução com Variações Protegidas:**

* **Crie interfaces ou classes abstratas** que definem o comportamento esperado.
* **Implemente as variações** concretas por trás dessas abstrações.
* O **módulo cliente interage apenas com a abstração**, não com a implementação específica.

**Exemplo:**  
A classe ServicoFechamentoPedido usa a interface PedidoRepository, que pode ser implementada como:

* PedidoRDBMSRepository (relacional)
* PedidoMongoRepository (NoSQL)

Isso permite alternar ou expandir a forma de persistência **sem alterar** o código da classe cliente.

**Consequências das Variações Protegidas:**

✅**Facilita a evolução do sistema** frente a mudanças prováveis.  
✅**Reduz o impacto de alterações** e **melhora a testabilidade**.  
✅**Aproveita recursos da orientação a objetos**, como encapsulamento e polimorfismo.  
✅Aplicado em padrões como **Adapter**, **Strategy**, **Abstract Factory**, **Bridge**.

**Cuidado:**  
Isolar todas as variações possíveis pode ser **custoso e desnecessário**.  
Avalie **onde as variações são prováveis e críticas**, priorizando essas áreas.

Interface gráfica do usuário, Texto, Aplicativo, Email

O conteúdo gerado por IA pode estar incorreto.