**Anexo 3**

**Padrões de Desenvolvimento de Software: Uma Visão Geral dos Conceitos Fundamentais**

O desenvolvimento de software é uma disciplina complexa que exige uma combinação de conhecimentos técnicos, boas práticas e padrões bem estabelecidos. Entre os conceitos mais influentes e amplamente adotados na indústria, destacam-se os padrões de projeto do Gang of Four (GoF), os princípios SOLID, as práticas de código limpo (Clean Code) e o padrão de arquitetura MVC.

### A3.1 Gang of Four (GoF)

Os padrões de projeto do Gang of Four [Gamma *et al.* 1994], são fundamentais para a programação orientada a objetos. Os autores identificam 23 padrões de projeto que resolvem problemas recorrentes em design de software, divididos em suas respectivas categorias: padrões criacionais, estruturais e comportamentais.

### Padrões Criacionais:

1. **Abstract Factory**: Fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas. Útil para garantir a compatibilidade entre produtos.
2. **Builder**: Separa a construção de um objeto complexo da sua representação, permitindo a criação passo a passo. É útil quando a criação do objeto envolve muitas etapas.
3. **Factory Method**: Define uma interface para criar um objeto, mas deixa para as subclasses a decisão de qual classe instanciar. Promove a reutilização de código ao encapsular a lógica de criação.
4. **Prototype**: Permite a criação de novos objetos copiando um objeto existente (protótipo). É útil quando a criação direta do objeto é complexa ou custosa.
5. **Singleton**: Garante que uma classe tenha apenas uma instância e fornece um ponto global de acesso a ela. Útil para recursos que precisam ser compartilhados, como conexões de banco de dados.

### Padrões Estruturais:

1. **Adapter**: Permite que interfaces incompatíveis trabalhem juntas, convertendo a interface de uma classe em outra esperada pelos clientes. É útil para integrar classes com interfaces incompatíveis.
2. **Bridge**: Separa a abstração da implementação, permitindo que ambas variem independentemente. Útil quando as abstrações e implementações devem ser extensíveis por subclasses.
3. **Composite**: Compõe objetos em estruturas de árvore para representar hierarquias parte-todo, permitindo que clientes tratem objetos individuais e composições de objetos de maneira uniforme.
4. **Decorator**: Adiciona responsabilidades a um objeto dinamicamente, sem modificar sua estrutura. Útil para adicionar funcionalidades a objetos de forma flexível e escalável.
5. **Facade**: Fornece uma interface simplificada para um conjunto complexo de interfaces de um subsistema, facilitando o uso do subsistema pelo cliente.
6. **Flyweight**: Reduz a sobrecarga de criação de um grande número de objetos semelhantes, compartilhando o máximo de dados possível entre eles. Útil para melhorar a eficiência em sistemas com muitos objetos semelhantes.
7. **Proxy**: Fornece um substituto ou marcador para outro objeto para controlar o acesso a ele. Útil para adicionar controle de acesso, carregar objetos sob demanda, entre outros.

### Padrões Comportamentais:

1. **Chain of Responsibility**: Passa uma solicitação ao longo de uma cadeia de handlers, onde cada handler decide processar a solicitação ou passá-la adiante. Útil para desacoplar o remetente do receptor da solicitação.
2. **Command**: Encapsula uma solicitação como um objeto, permitindo parametrizar clientes com diferentes solicitações, filas ou registros de solicitações. Útil para implementação de operações reversíveis e filas de operações.
3. **Interpreter**: Define uma gramática para uma linguagem e usa um intérprete para interpretar sentenças dessa linguagem. Útil para implementar linguagens de domínio específico.
4. **Iterator**: Fornece uma maneira de acessar sequencialmente os elementos de uma coleção sem expor sua representação subjacente. Útil para percorrer coleções de maneira uniforme.
5. **Mediator**: Define um objeto que encapsula a forma como um conjunto de objetos interage, promovendo um acoplamento fraco e centralizando a comunicação. Útil para reduzir interdependências complexas entre objetos.
6. **Memento**: Permite capturar e restaurar o estado interno de um objeto sem violar seu encapsulamento. Útil para implementar funcionalidades de desfazer operações.
7. **Observer**: Define uma dependência um-para-muitos entre objetos, de forma que, quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente. Útil para implementar notificações.
8. **State**: Permite que um objeto altere seu comportamento quando seu estado interno muda. Útil para implementar máquinas de estado.
9. **Strategy**: Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. Permite que o algoritmo varie independentemente dos clientes que o utilizam.
10. **Template Method**: Define o esqueleto de um algoritmo em uma operação, deixando alguns passos para subclasses implementarem. Útil para definir a estrutura de um algoritmo de forma extensível.
11. **Visitor**: Representa uma operação a ser executada nos elementos de uma estrutura de objeto, permitindo definir novas operações sem mudar as classes dos elementos nos quais opera. Útil para adicionar operações a estruturas de objetos complexas.

### A3.2 SOLID

Os princípios SOLID são um conjunto de cinco diretrizes destinadas a promover um design de software mais flexível e fácil de manter, também conhecido como *Uncle Bob*, esses princípios são descritos por [Martin 2002].

1. **Single Responsibility Principle (SRP)**: Uma classe deve ter apenas uma razão para mudar, ou seja, deve ter uma única responsabilidade.
2. **Open/Closed Principle (OCP)**: Entidades de software (classes, módulos, funções, etc.) devem estar abertas para extensão, mas fechadas para modificação.
3. **Liskov Substitution Principle (LSP)**: Objetos de uma classe derivada devem poder substituir objetos da classe base sem alterar o comportamento desejado.
4. **Interface Segregation Principle (ISP)**: Múltiplas interfaces específicas são melhores do que uma única interface geral.
5. **Dependency Inversion Principle (DIP)**: Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.

### A3.3 Clean Code

O conceito de código limpo, O autor [Martin 2008] enfatiza a importância de escrever código que seja fácil de ler, entender e manter. Ele propõe várias práticas e princípios para alcançar, seguem algumas:

**Nomes Significativos**:

* Escolher nomes claros e descritivos para variáveis, funções, classes e outros elementos do código é crucial para a legibilidade.
* Nomes devem comunicar a intenção. Por exemplo, calculaSalarioAnual é mais descritivo que calcSal.

**Funções Pequenas e Coesas**:

* Funções devem ser pequenas, realizando apenas uma tarefa específica e claramente definida.
* Seguir o princípio do *Single Responsibility Principle* (SRP), onde cada função tem uma única responsabilidade ou motivo para mudar.

**Evitar Comentários Desnecessários**:

* O código deve ser autoexplicativo, reduzindo a necessidade de comentários.
* Comentários podem se tornar obsoletos e enganosos se não forem mantidos adequadamente. Utilize-os para explicar por quê, não o quê.

**Formatação Consistente**:

* Consistência na formatação, como indentação e espaçamento, melhora a legibilidade.
* Seguir convenções de código e padrões da equipe ou projeto.

**Tratamento de Erros**:

* Erros devem ser tratados adequadamente, preferencialmente utilizando exceções em vez de códigos de erro.
* Evitar o uso de estruturas complexas de tratamento de erros, mantendo o código claro e legível.

### A3.4 MVC (Model-View-Controller)

O padrão de arquitetura MVC, inicialmente descrito na década de 1970 para a linguagem de programação *Smalltalk*. O livro *Design Patterns* do GoF também menciona o MVC como um exemplo de padrão arquitetural. O padrão de arquitetura MVC é uma abordagem amplamente adotada para estruturar aplicações de software, especialmente aquelas que possuem uma interface gráfica de usuário (GUI). O MVC divide uma aplicação em três componentes interconectados principais: Model, View e Controller. Essa separação de responsabilidades facilita a manutenção, a escalabilidade e a reutilização do código.

### Componentes do MVC

1. **Model (Modelo)**:
   * **Responsabilidade**: O Model representa a lógica de negócios e os dados da aplicação. Ele é responsável por gerenciar o estado da aplicação, acesso a dados e regras de negócios.
   * **Comunicação**: O Model notifica a View sobre as mudanças nos dados e fornece os dados solicitados pelo Controller.
   * **Exemplo**: Em uma aplicação de e-commerce, o Model pode incluir classes como Produto, Carrinho, Pedido, que contêm a lógica para manipular os dados desses elementos.
2. **View (Visão)**:
   * **Responsabilidade**: A View é responsável pela apresentação dos dados ao usuário. Ela define a estrutura visual e a interface de usuário.
   * **Comunicação**: A View obtém os dados do Model para exibi-los e envia as interações do usuário ao Controller.
   * **Exemplo**: Em uma aplicação web, a View pode ser representada por páginas HTML, CSS e JavaScript, ou templates em frameworks como Angular, React ou Vue.js.
3. **Controller (Controlador)**:
   * **Responsabilidade**: O Controller atua como um intermediário entre a View e o Model. Ele recebe as entradas do usuário através da View, processa essas entradas (interagindo com o Model) e retorna o resultado para a View.
   * **Comunicação**: O Controller invoca métodos do Model para alterar seu estado ou recuperar dados e seleciona a View apropriada para renderizar a resposta.
   * **Exemplo**: Em uma aplicação de e-commerce, um Controller pode gerenciar ações como adicionar um produto ao carrinho, finalizar uma compra ou atualizar o perfil do usuário.

### Benefícios do MVC

1. **Separação de Preocupações**: Cada componente tem uma responsabilidade claramente definida, o que facilita a manutenção e o desenvolvimento paralelo.
2. **Reutilização de Código**: A lógica de negócios (Model) é independente da interface de usuário (View), permitindo a reutilização em diferentes interfaces.
3. **Facilidade de Teste**: A separação facilita a escrita de testes unitários e de integração, especialmente para a lógica de negócios no Model e a lógica de controle no Controller.
4. **Manutenção e Extensibilidade**: Alterações na interface de usuário podem ser feitas independentemente da lógica de negócios, e vice-versa, tornando a aplicação mais fácil de manter e expandir.

### A3.5 DAO (Data Access Object)

O padrão DAO (Data Access Object) é uma prática de design fundamental no desenvolvimento de software, reconhecida por sua capacidade de separar a lógica de acesso aos dados da lógica de negócios em aplicações orientadas a objetos. O conceito de DAO foi formalmente apresentado [Gamma *et all.* 1994].

No contexto do padrão DAO, a principal preocupação é separar as operações de acesso aos dados das operações de manipulação de dados e lógica de negócios. Isso é alcançado encapsulando operações CRUD (*Create, Read, Update, Delete*) em objetos DAO dedicados, que oferecem uma interface abstrata para interação com diferentes fontes de dados, como bancos de dados relacionais.

O padrão DAO continua sendo uma peça fundamental na caixa de ferramentas de desenvolvedores de software, proporcionando uma abstração eficaz para o acesso a dados e promovendo boas práticas de arquitetura. Sua origem nos princípios estabelecidos pelos "Gang of Four" e sua evolução para aplicações modernas demonstram sua relevância contínua no desenvolvimento de software orientado a objetos.

Em suma, ao implementar o padrão DAO, os desenvolvedores não apenas melhoram a estrutura e a manutenibilidade de suas aplicações, mas também adotam uma abordagem robusta para lidar com a complexidade do acesso a dados em ambientes de software sofisticados e em constante evolução.

#### Benefícios do Padrão DAO

A adoção do padrão DAO traz uma série de benefícios significativos para o desenvolvimento de software:

1. **Separação de Responsabilidades:** Ao isolar o acesso aos dados em objetos DAO, a lógica de negócios torna-se mais limpa e focada em suas responsabilidades principais, promovendo uma arquitetura mais modular e de fácil manutenção.
2. **Reutilização de Código:** Os objetos DAO encapsulam operações comuns de acesso a dados, facilitando a reutilização do código em toda a aplicação. Isso reduz a duplicação de código e promove uma implementação consistente de operações de acesso aos dados.
3. **Facilidade de Testabilidade:** Por permitir a substituição fácil de implementações DAO (por exemplo, para testes unitários), o padrão DAO melhora a testabilidade do código. Testes podem ser realizados de forma mais eficaz e sem depender diretamente de infraestruturas externas, como bancos de dados.

#### Aplicações Modernas e Frameworks

Desde sua introdução, o padrão DAO tem sido amplamente adotado em frameworks modernos de desenvolvimento de software. Por exemplo, o framework Hibernate para Java utiliza o padrão DAO como parte de suas práticas recomendadas para interação com bancos de dados relacionais. Da mesma forma, o Spring Framework oferece suporte robusto para implementação de DAOs, promovendo a modularidade e a escalabilidade em aplicações empresariais.