**Padrões GoF**

* **Decorator** - Atribui responsabilidade adicionais a um objeto dinamicamente. O Decorator fornece uma alternativa flexível a subclasses para a extensão da funcionalidade.
* **Flyweight** - Permite usar compartilhamento para suportar de forma eficiente grandes quantidades de objetos. A motivação é reduzir o consumo de memória e o custo em tempo de execução. Usa compartilhamento para dar suporte a vários objetos de forma eficiente.
* **Memento** - sem violar o encapsulamento, captura e externaliza o estado interno de um objeto, de forma que ele possa ser recuperado depois. Deve ser usado quando uma fotografia (parte) do objeto precisar ser salva, de forma que ela possa ser recuperada depois. Torna possível salvar o estado de um objeto de modo que o mesmo possa ser restaurado.
* **Factory Method**: define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe instanciar.
* **Adapter**: converter a interface de uma classe em outra interface que normalmente não poderiam trabalhar juntas.
* **Abstract Factory**: interface para criar famílias de objetos relacionados.
* **Singleton**: garante que uma classe tenha apenas uma instância e provê um ponto de acesso global a ela.
* **Builder**: criação de objeto complexo com representações diferentes; constrói partes de objetos passo a passo. Palavras-chave do builder: objeto complexo, separação da construção.
* **Chain of Responsability** - Evita o acoplamento do remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar a solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate.
* **Observer**: define uma dependência um-para-muitos entre objetos para que quando um objeto mudar de estado, os seus dependentes sejam notificados e atualizados automaticamente.
* **Bridge** é um padrão de projeto de software, utilizado quando é desejável que uma interface (abstração) possa variar independentemente das suas implementações. “**Desacoplar uma abstração da sua implementação, de modo que as duas possam variar independentemente.**” Ou seja, o Bridge fornece um nível de abstração maior que o Adapter, pois são separadas as implementações e as abstrações, permitindo que cada uma varie independentemente.
* **Strategy** - Usado quando uma classe é usuária de uma certa hierarquia/ família de algoritmos, como um algoritmo de busca ou ordenação. Esse padrão define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. Como existem diversos algoritmos em uma mesma família, não é desejável que se codifique todos eles em uma mesma classe. Permite que um algoritmo varie independentemente dos clientes que o utilizam. Fornece uma maneira de configurar uma classe com um dentre muitos comportamentos. Pode ser usado para evitar a exposição das estruturas de dados específicas de um algoritmo, que se deseja ocultar.
* ***Proxy:****provê um****substituto****ou ponto através do qual um objeto pode controlar o acesso a outro objeto. Pessoal, esse padrão de projeto deve ser utilizado quando houver uma necessidade de uma referência mais versátil ou sofisticada para um objeto do que um simples ponteiro.****Por exemplo, proxies virtuais criam objetos caros por demanda e proxies de proteção controlam o acesso ao objeto original****. Considerem a hipótese de um sistema que acesse um banco de dados por meio de uma classe de conexão.*
* **Template Method:** Definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para as subclasses. Template Method permite que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo. (Comportamental)
* **Interpreter** - Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador para as sentenças dessa linguagem. Usado quando houver uma linguagem para interpretar e se possível representar as sentenças da linguagem como árvores sintáticas abstratas. Interpreta a mensagem recebida e a direciona para um alvo.
* **Command** - Encapsula uma requisição como um objeto, deixando-o, dessa forma, parametrizar clientes com diferentes requisições.
* **Mediator**: Define um objeto que encapsula a forma como um conjunto de objetos interage, promovendo um fraco acoplamento ao evitar que objetos se refiram uns aos outros explicitamente.
* **Façade -** implementa um padrão que forneça uma interface unificada para um conjunto de interfaces em um subsistema. Esse padrão estrutural define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado.
* **State** - Permite que um objeto mude o seu comportamento quando o seu estado interno mudar. O objeto parecerá ter mudado de classe.

Em Engenharia de software, um **anti-padrão** é um padrão de projeto de software que pode ser comumente usado, mas é ineficiente e/ou contra produtivo em prática. O termo foi cunhado em 1995 por Andrew Koenig, inspirado pelo livro do Gang of Four *Design Patterns*, que desenvolveu o conceito de padrão de desenho de software. O termo foi largamente popularizado três anos depois pelo livro *AntiPatterns*, que estendeu o uso do termo além do campo de desenho de software para interações pessoais em geral.

**O Naked Objects Pattern (NOP)** é um padrão arquitetural que dá ênfase na criação dos objetos de negócio que modelam o domínio do sistema (chamados de naked objects) e na completude comportamental desses objetos. Diferente da arquitetura em camadas que promove a separação das operações dos seus objetos de domínio, deixando-os apenas com os atributos NOP, a lógica de negócio deve estar encapsulada nos objetos de domínio de forma a ter todo o comportamento esperado para o objeto.

O desenvolvedor do software cria apenas as classes de domínio do problema e estabelece seus relacionamentos, atributos e comportamentos, formando o modelo de domínio da aplicação. Nesta filosofia, a interface de usuário deve ser gerada automaticamente a partir desses objetos.

A **técnica de wrapping** decompõe a funcionalidade do sistema em pacotes/componentes de software. Esses pacotes são envolvidos por uma “casca de software” que atua como interpretadora entre o legado e a nova aplicação, que pode ser orientada a objetos ou não, enquanto que a aplicação legada, normalmente, não é orientada a objetos. Isso possibilita que o ambiente legado seja preservado e parte da aplicação legada seja reutilizada, reduzindo assim o esforço de migração - reengenharia.

**Padrões GRASP**

**General responsibility assignment software patterns** (ou **principles**), abreviado **GRASP**, consiste em diretrizes para atribuir responsabilidade a classes e objetos em projeto orientado a objetos.

O padrão **controlador(controller)** atribui a responsabilidade de manipular eventos do sistema para uma classe que não seja de interface do usuário (UI) que representa o cenário global ou cenário de caso de uso. Um objeto controlador é um objeto de interface não-usuário, responsável por receber ou manipular um evento do sistema.

O padrão **Creator** atribui a responsabilidade de criação de objetos entre duas classes A e B. Para uma classe *B* ter a responsabilidade de criar instâncias de outra classe *A*, ele deve obedecer a mais de um dos requisitos abaixo:

* B agrega objetos da classe A.
* B contém objetos da classe A.
* B registra instâncias da classe A.
* B usa muitos objetos da classe A.
* B possui os dados usados para inicializar A.

**Especialista na Informação** (ou **Especialista da Informação**, ou apenas **Especialista**, ou ainda seus equivalentes em inglês ***Information Expert*** e ***Expert***) é um [padrão de projeto de software](https://pt.wikipedia.org/wiki/Padr%C3%A3o_de_projeto_de_software), parte do conjunto de princípios conhecido como [GRASP](https://pt.wikipedia.org/wiki/GRASP) (*General Responsibility Assignment Software Patterns*). É uma abordagem genérica que visa atribuir a responsabilidade de fazer ou conhecer algo ao "especialista na informação" — a [classe](https://pt.wikipedia.org/wiki/Classe_(programa%C3%A7%C3%A3o)) que possui a informação necessária para cumprir tal responsabilidade.[[1]](https://pt.wikipedia.org/wiki/Expert_(padr%C3%A3o_de_projeto_de_software)" \l "cite_note-1)

Funcionamento

**Obrigações de fazer algo**

* Fazer algo a si mesmo
* Iniciar ações em outros objetos
* Controlar ou coordenar atividades em outros objetos

**Obrigações de conhecer algo**

* Conhecer dados encapsulados
* Conhecer [objetos](https://pt.wikipedia.org/wiki/Objeto) relacionados
* Conhecer coisas que se pode calcular

**Low Coupling:***Esse padrão é responsável por ditar como atribuir responsabilidades para apoiar baixa dependência entre classes, como suportar mudanças em uma classe que tenham baixo impacto em outras classes, e maior potencial de reúso.****O acoplamento está sempre associado à coesão. Eu sempre decorei assim: “Acoplamento é a dependência entre as partes”****.*

**Padrões Java - Java Enterprise Edition (JEE)**

O padrão utilizado para interceptar e manipular uma requisição e uma resposta antes e depois de a requisição ser processada é o Intercepting Filter. Já aquele utilizado para encapsular e gerenciar todo acesso a fontes de dados e suas persistências é o Data Access Object( DAO).

Padrões da Camada de Apresentação (Presentation Tier Patterns):

* Intercepting Filter
* Front Controller
* Context Object
* Application Controller
* View Helper
* Composite View
* Service to Worker
* Dispatcher View

Padrões da Camada de Negócio (Business Tier Patterns)

* Business Delegate
* Service Locator
* Session Façade
* Application Service
* Business Object
* Composite Entity
* Transfer Object
* Transfer Object Assembler
* Value List Handler

Padrões da Camada de Integração (Integration Tier Patterns)

* Data Access Object
* Service Activator
* Domain Store
* Web Service Broker

A principal ideia do **DDD - Domain-Driven Design** - é a de que o mais importante em um software não é o seu código, nem sua arquitetura, nem a tecnologia sobre a qual foi desenvolvido, mas sim o problema que o mesmo se propõe a resolver, ou em outras palavras, a regra de negócio. Ela é a razão do software existir, por isso deve receber o máximo de tempo e atenção possíveis. Em praticamente todos os projetos de software, a complexidade não está localizada nos aspectos técnicos, mas sim no negócio, na atividade que é exercida pelo cliente ou problema que o mesmo possui. Como já diz o título do livro de Eric Evans, esse é o “coração”, o ponto central de qualquer aplicação, portanto todo o resto deve ser trabalhado de forma que este “coração” seja entendido e concebido da melhor forma possível.

SOLID

O S.O.L.I.D é um acrônimo que representa cinco princípios da programação orientada a objetos e design de código

* **[S]ingle Responsibility Principle (Princípio da Responsabilidade Única)**

Esse primeiro princípio diz que *“uma classe deve ter apenas um motivo para mudar”,*ou seja*,*deve ter uma única responsabilidade. Basicamente, esse princípio trata especificamente a **coesão**.

* **[O]pen/Closed Principle (Princípio do Aberto/Fechado)**

Diz que *“as entidades de software (classes, módulos, funções etc.) devem ser abertas para ampliação, mas fechadas para modificação”.*De forma mais detalhada,diz que podemos estender o comportamento de uma classe, quando for necessário,por meio de **herança**, **interface**e **composição**, mas não podemos permitir a abertura dessa classe para fazer pequenas modificações.

* **[L]iskov Substitution Principle (Princípio da Substituição de Liskov)**

Diz que *“Os subtipos devem ser substituíveis pelos seus tipos base”,*e que as classes/tipos base podem ser substituídas por qualquer uma das suas subclasses, ponderando sobre os cuidados para usar a herança no seu projeto de software.

* **[I]nterface Segregation Principle (Princípio da Segregação de Interfaces)**

Diz que *“muitas interfaces específicas são melhores do que uma interface geral”.*

* **[D]ependency Inversion Principle (Princípio da Inversão de Dependências)**

Diz que devemos *“depender de abstrações e não de classes concretas”*. Uncle Bob quebra a definição desse princípio em dois sub-itens:

E isso se dá porque abstrações mudam menos e facilitam a mudança de comportamento e as futuras evoluções do código.