# Trabalho Colaborativo Design Patterns

Cada membro da turma, escolhe um padrão de projeto e contribui neste doc aqui respondendo qual padrão, o seu propósito, o problema e a solução como resolver este problema.

## Builder

### Nome: Marcus Vinícius Barcelos

#### O Builder é um padrão de projeto criacional que permite a você construir objetos complexos passo a passo. O padrão permite que você produza diferentes tipos e representações de um objeto usando o mesmo código de construção.

#### Imagine um objeto complexo que necessite de uma inicialização passo a passo trabalhosa de muitos campos e objetos agrupados. Tal código de inicialização fica geralmente enterrado dentro de um construtor monstruoso com vários parâmetros. Ou pior: espalhado por todo o código cliente.

#### Por exemplo, vamos pensar sobre como criar um objeto Casa. Para construir uma casa simples, você precisa construir quatro paredes e um piso, instalar uma porta, encaixar um par de janelas, e construir um teto. Mas e se você quiser uma casa maior e mais iluminada, com um jardim e outras miudezas (como um sistema de aquecimento, encanamento, e fiação elétrica)?

#### A solução mais simples é estender a classe base Casa e criar um conjunto de subclasses para cobrir todas as combinações de parâmetros. Mas eventualmente você acabará com um número considerável de subclasses. Qualquer novo parâmetro, tal como o estilo do pórtico, irá forçá-lo a aumentar essa hierarquia cada vez mais.

#### Há outra abordagem que não envolve a propagação de subclasses. Você pode criar um construtor gigante diretamente na classe Casa base com todos os possíveis parâmetros que controlam o objeto casa. Embora essa abordagem realmente elimine a necessidade de subclasses, ela cria outro problema.

Na maioria dos casos, a maioria dos parâmetros não será usada, tornando [**as chamadas do construtor em algo feio de se ver**](https://refactoring.guru/pt-br/smells/long-parameter-list). Por exemplo, apenas algumas casas têm piscinas, então os parâmetros relacionados a piscinas serão inúteis nove em cada dez vezes.

O padrão Builder sugere que você extraia o código de construção do objeto para fora de sua própria classe e mova ele para objetos separados chamados *builders*. “Builder” significa “construtor”, mas não usaremos essa palavra para evitar confusão com os construtores de classe.

O padrão organiza a construção de objetos em uma série de etapas (construirParedes, construirPorta, etc.). Para criar um objeto você executa uma série de etapas em um objeto builder. A parte importante é que você não precisa chamar todas as etapas. Você chama apenas aquelas etapas que são necessárias para a produção de uma configuração específica de um objeto.

Algumas das etapas de construção podem necessitar de implementações diferentes quando você precisa construir várias representações do produto. Por exemplo, paredes de uma cabana podem ser construídas com madeira, mas paredes de um castelo devem ser construídas com pedra.

Nesse caso, você pode criar diferentes classes construturas que implementam as mesmas etapas de construção, mas de maneira diferente. Então você pode usar esses builders no processo de construção (i.e, um pedido ordenado de chamadas para as etapas de construção) para produzir diferentes tipos de objetos.

Por exemplo, imagine um builder que constrói tudo de madeira e vidro, um segundo builder que constrói tudo com pedra e ferro, e um terceiro que usa ouro e diamantes. Ao chamar o mesmo conjunto de etapas, você obtém uma casa normal do primeiro builder, um pequeno castelo do segundo, e um palácio do terceiro. Contudo, isso só vai funcionar se o código cliente que chama as etapas de construção é capaz de interagir com os builders usando uma interface comum.

**Guard Clause/Bouncer Pattern**

**Nome:** Diogo Vinícius dos Santos Martins

Guard clause, ou bouncer pattern, é um pattern muito comum em qualquer linguagem. Seu principal benefício é diminuir o uso de ifs e elses aninhados no código para facilitar a manutenção e escalabilidade do código a longo prazo. Esse pattern funciona da seguinte forma: sempre que você precisar validar algum dado no início de algum fluxo, ou verificar alguma condição que seja essencial para o fluxo do código, você pode utilizar um guard clause, e caso essa condição não seja atingida, sair do método ou jogar um exception imediatamente, já que sem ela não faz sentido continuar a execução daquele bloco de código. O Guard Clause é um pattern muito fácil de implementar, e pode ser bem útil para evitar null pointers e controlar melhor o fluxo da aplicação como um todo em alguns casos específicos.

**Exemplo:**

inserir um usuário no BD, não faria sentido inserir um usuário com todos os campos nulos, então normalmente essa situação é tratada pela aplicação de antemão, e acaba sendo comum criar vários ifs e elses para validar esses campos.

public void CreateUser(String userName, String password)

{

if (userName == null)

{

// Do something...

}

else

{

// Do something else...

if (password == null)

{

// Do something...

}

else

{

// Do something else...

}

}

}

No código acima é necessário verificar se os valores são nulos antes de executar o comportamento, já que eles são a pré condição para que esse código possa ser executado. No entanto ele não está tão fácil de ler, e existem muitos ifs e elses aninhados com muitas chaves, e com o tempo, se isso não for refatorado, a tendência é que provavelmente fique pior, fazendo um código menos legível ainda e mais difícil de dar manutenção. Esse é um dos casos em que o Guard Clause diminuiria bastante a complexidade e aumentaria a legibilidade do código com muita facilidade.

**Exemplo:**

public void CreateUser(String userName, String password)

{

if (userName == null)

throw new NullArgumentException(nameof(userName));

if (password == null)

throw new NullArgumentException(nameof(password));

// Do something...

}

Nesse exemplo, o Guard Clause é utilizado para garantir que o código não vai ser executado caso não receba os parâmetros, garantindo que não ocorrerão possíveis null pointers, caso a aplicação não esteja preparada para tratar isso, e garantindo a regra de negócio. Esse pattern de jeito algum substitui a necessidade de tratar exceções, mas pode ser útil caso em algum momento você precise executar um método que necessariamente depende de algo pra que ele seja executado, já que não faz sentido continuar a execução caso essa condição não seja atendida. Além disso, fica fácil de ler e diminui a necessidade de criar vários ifs e elses, tornando o código mais legível e de fácil manutenção. Porém, ainda é possivel melhorar esse codigo. Caso esse método precise validar mais parâmetros ainda, criando uma classe que irá providenciar Guard Clauses para cenários diferentes, já que normalmente o menor pedaço de código que existirá em uma aplicação é um método, e a invocação dele tornará o código ainda mais legível.

Exemplo:

public class GuardProvider{

public void guardForString(String argument){

if(argument==nulls || argument.equals(""))

throw new CustomException(msg);

}

//guard clauses para cenários diferentes

}

**Código anterior com a adição de um método utilitário contra nulos**

public void CreateUser(String userName, String password){

Guard.guardForString(userName);

Guard.guardForString(password);

//executar código

}

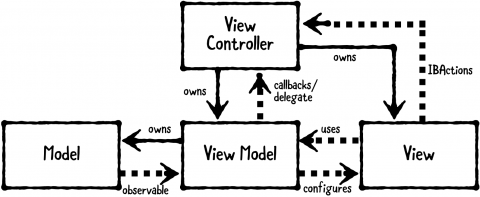
O código fica mais legível ainda e escalável, já que a classe GuardProvider pode ser usada para validar diferentes cenários sem ter que ficar criando diversos ifs e elses nas classes que precisam validar esses dados, ou mesmo métodos que fariam essa mesma validação, mas que ficariam espalhadas por toda a aplicação nas classes em que elas seriam necessárias. No entanto, é preciso ter cuidado para não fazer com que a classe de Guard valide literalmente tudo, mas sim aproveitá-la caso existam cenários que serão validados diversas vezes, como um objeto que é muito utilizado no código e não pode ter algum valor específico nulo ou algo do gênero.

Resumindo, o Guard Clause deve ser usado quando o código precisar validar muitos dados e acabar deixando o código pouco legível, ou caso alguma parte do código dependa explicitamente de algum valor para ser executado.

Nome: Natan Valim Cardoso

Model-View-ViewModel (MVVM) é um padrão de design estrutural que separa objetos em três grupos distintos:

* **Os modelos** armazenam os dados do aplicativo. Geralmente são structs ou classes simples.
* **As visualizações** exibem elementos visuais e controles na tela. Eles são tipicamente subclasses de UIView.
* **Os modelos de visualização** transformam as informações do modelo em valores que podem ser exibidos em uma visualização. Geralmente são classes, então podem ser passadas como referências.



Esse padrão soa familiar? Sim, é muito semelhante ao Model-View-Controller (MVC). Observe que o diagrama de classes na parte superior desta página inclui um controlador de exibição; controladores de visualização existem no MVVM, mas sua função é minimizada.

## **Quando você deve usá-lo?**

Use este padrão quando precisar transformar modelos em outra representação para uma vista. Por exemplo, você pode usar um modelo de exibição para transformar um Date em um formato de data String, um Decimal em um formato de moeda String ou muitas outras transformações úteis.

Este padrão complementa o MVC especialmente bem. Sem modelos de exibição, você provavelmente colocaria o código de transformação de modelo para exibição em seu controlador de exibição. No entanto, os controladores de exibição já estão fazendo um pouco: manipulando viewDidLoad e outros eventos de ciclo de vida de exibição, manipulando retornos de chamada de exibição IBActions e várias outras tarefas também.

**Nome:** Gustavo Padilha

**Assunto:** Singleton

O **Design Pattern Singleton** especifica que apenas uma única instância de uma classe deve existir com um ponto de acesso global. Isso funciona bem ao modelar objetos do mundo real tendo apenas uma instância.

O construtor é privado evitando que essa classe seja instanciada fora dela. Assim, para podermos instanciar ou acessar uma instância da classe criou-se um atributo público e estático (da classe) que retorna através de um método estático uma única instância dessa classe.

Altamente recomendado para ser utilizado em classes que nunca mudam de estado ou que sempre precisam de somente uma instância em execução, também melhora a leitura do código, tendo em mente que o **Singleton**, apesar de ser um padrão (aplicada a uma linguagem universal), é o mais conhecido dentre os padrões.

public class SingletonClass {

private static SingletonClass sSoleInstance;

private SingletonClass(){} //Construtor privado.

public static SingletonClass getInstance(){

if (sSoleInstance == null){ // Se não tiver uma instância criada, cria uma nova.

sSoleInstance = new SingletonClass();

}

return sSoleInstance;

}

}

Adicionando um construtor privado, apenas a própria classe pode criar um objeto de si mesma. Então utilizamos um método estático para acessar essa instância única.

Quando você precisa acessar métodos do objeto Singleton, basta fazer uma

chamada da seguinte forma:

public class SingletonTester {

public static void main(String[] args) {

//Instância

SingletonClass instance1 = SingletonClass.getInstance();

System.out.println("Instance 1 hash:" + instance1.hashCode());

}

}

O Singleton é provavelmente o Design Pattern mais fácil de entender inicialmente, mas pode ser perigosamente fácil de usar em excesso. Como é acessível a partir de vários objetos, o Singleton pode sofrer efeitos colaterais inesperados que são difíceis de rastrear.