

AceLeraDev C#

SOLID

Módulo 4

Tópicos desta aula:

Tópico 1: O que é SOLID

Tópico 2: Single Responsibility Principle

Tópico 3: Open Closed Principle

Tópico 4: Liskov Substitution Principle

Tópico 5: Interface Segregation Principle

Tópico 6: Dependency Inversion Principle

O que é SOLID?

É o acrônimo de 5 princípios da orientação a objetos observados por Michael Feathers, baseado no trabalho do Robert C. Martin (Uncle Bob).

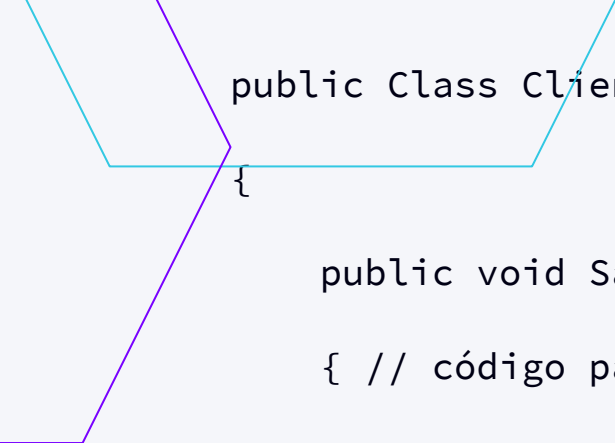
- Single Responsibility Principle
- Open Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Princípio de Responsabilidade Única – Single Responsibility Principle

A Classe deve ter somente uma razão para mudar.

Cada responsabilidade é um eixo de mudança, quando os requisitos mudam, essa mudança será manifestada através de uma mudança de responsabilidades entre classes. Se uma classe assumir mais de uma responsabilidade, terá mais que uma razão para mudar.

Esse tipo de acoplamento leva a projetos frágeis que quebram de maneira inesperada quando mudou.



```
public Class Cliente
```

```
{
```

```
    public void Salvar()
```

```
    { // código para acessar o banco de dados e salvar }
```

```
    private bool Validar()
```

```
    { // código para validar cliente }
```

```
    private void EnviarEmail()
```

```
    { // código para envio de email }
```

```
}
```



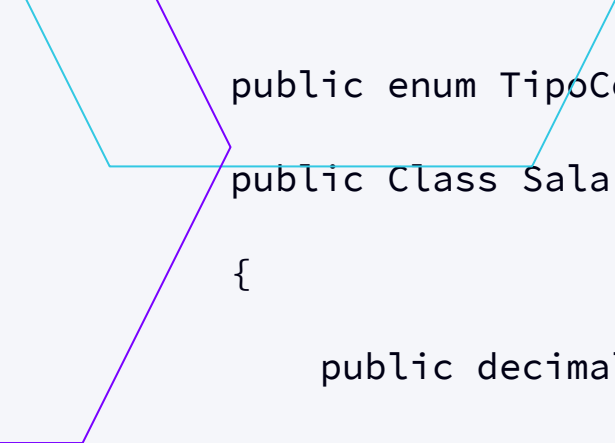
Princípio Aberto/Fechado

– Open Closed Principle

Este princípio estabelece que “entidades de softwares (classes, módulos, funções e etc ..)” devem ser abertas para extensão e fechadas para modificação.

Devemos projetar módulos que nunca mudam, quando os requisitos mudam, você estende o comportamento de tais módulos adicionando novo código, não alterando o antigo que já funciona.

Os principais mecanismos por trás do princípio de aberto/fechado são a abstração e o polimorfismo.



```
public enum TipoContratacao {CTL, PJ}
```

```
public Class Salarios
```

```
{
```

```
    public decimal Calcular(TipoContratacao tipo)
```

```
    {
```

```
        if(tipo == TipoContratacao.CLT)
```

```
        { // calcular clt }
```

```
        else if (tipo == TipoContratacao.PJ)
```

```
        { // calcular pj}
```

```
    }
```

```
}
```





```
public abstract Class Salarios
```

```
{
```

```
    public abstract decimal Calcular();
```

```
}
```

```
Public Class SalarioCLT : Salarios
```

```
{
```

```
    public override decimal Calcular() { // calculo clt }
```

```
}
```

```
Public Class SalarioPJ : Salarios
```

```
{
```

```
    public override decimal Calcular() { // calculo pj }
```

```
}
```



Princípio da substituição de Liskov

– Liskov Substitution Principle

Objetos em um programa devem ser substituíveis por instâncias de seus subtipos, sem alterar a funcionalidade do programa.

Se para cada objeto o_1 do tipo S existe um objeto o_2 do tipo T tal que, para todos os programas P definidos em T , o comportamento de P permanece inalterado quando o_1 é substituído por o_2 , então S é um subtipo de T .

Princípio da substituição de Liskov

– Liskov Substitution Principle



Princípio de Segregação de Interfaces

- Interface Segregation Principle

Muitas interfaces para finalidades específicas, são melhores do que uma para todos propósitos.

O princípio afirma que nenhum cliente deve ser forçados a depender de métodos que não utiliza. ISP divide interfaces que são muito grandes em menores e mais específicas, para que os clientes só necessitem saber sobre os métodos que são de interesse para eles. Tais interfaces encolhidas são também chamados de *papel de interfaces*. ISP destina-se a manter um sistema desacoplado e, assim, mais fácil para reestruturar, modificar



```
public Interface IAves
```

```
{
```

```
    decimal CalcularAltitude();
```

```
    decimal CalcularLocalizacao();
```

```
}
```

```
public Class Papagaio : IAves
```

```
{
```

```
    decimal CalcularAltitude(){ // código calculo };
```

```
    decimal CalcularLocalizacao(){ // código calculo };
```

```
}
```

```
public Class Pinguim : IAves
```

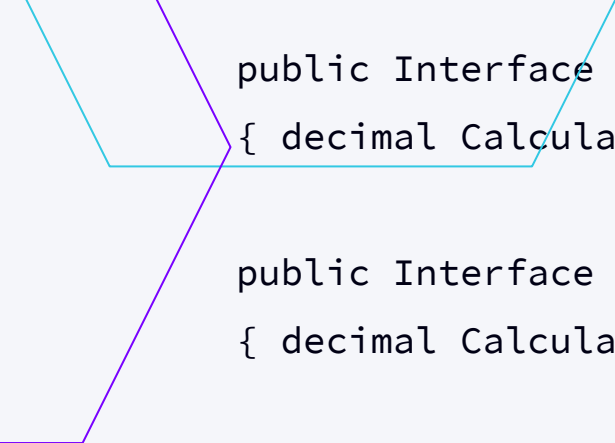
```
{
```

```
    decimal CalcularAltitude(){ // código calculo };
```

```
    decimal CalcularLocalizacao(){ // código calculo };
```

```
}
```



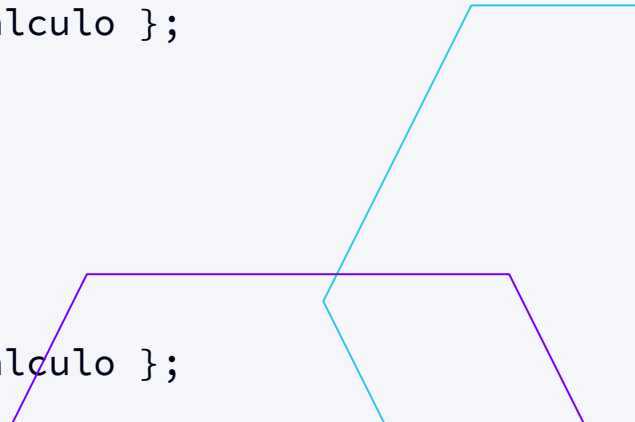


```
public Interface IAves
{ decimal CalcularLocalizacao(); }
```

```
public Interface IAvesQueVoam
{ decimal CalcularAltitude(); }
```

```
public Class Papagaio : IAves, IAvesQueVoam
{
    decimal CalcularAltitude(){ // código calculo };
    decimal CalcularLocalizacao(){ // código calculo };
}
```

```
public Class Pinguim : IAves
{
    decimal CalcularLocalizacao(){ // código calculo };
}
```



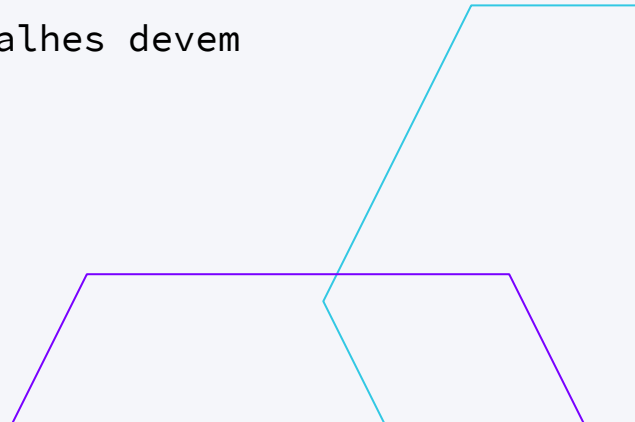


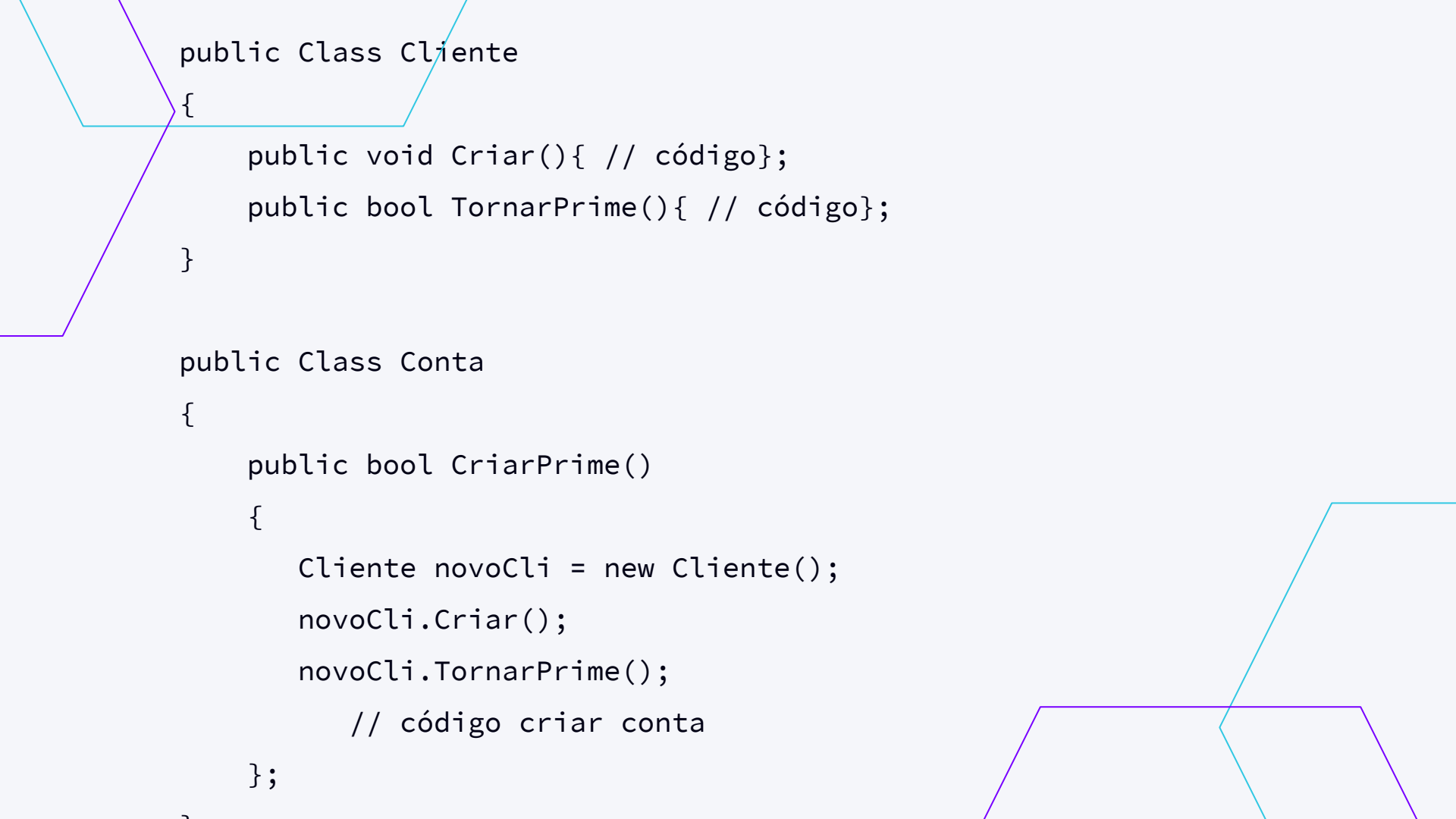
Princípio da Inversão de Dependência – Dependency Inversion Principle

Deve-se depender de abstrações, não de objetos concretos.

Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender da abstração.

Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.





```
public Class Cliente
```

```
{
```

```
    public void Criar(){ // código};
```

```
    public bool TornarPrime(){ // código};
```

```
}
```

```
public Class Conta
```

```
{
```

```
    public bool CriarPrime()
```

```
    {
```

```
        Cliente novoCli = new Cliente();
```

```
        novoCli.Criar();
```

```
        novoCli.TornarPrime();
```

```
        // código criar conta
```

```
    };
```

```
}
```

```
public Interface ICliente
{
    void Criar();
    bool TornarPrime();
}
```

```
public Class Cliente : ICliente
{
    public void Criar(){ // código};
    public bool TornarPrime(){ // código};
}
```

```
public Class Conta
{
    private readonly Icliente _cliente;

    public Conta (Icliente cliente)
    {
        _cliente = cliente
    }

    public bool CriarPrime()
    {
        _cliente.Criar();
        _cliente.TornarPrime();
        // código criar conta
    };
}
```


Revisão do que vimos hoje:

Tópico 1: O que é SOLID

Tópico 2: Single Responsibility Principle

Tópico 3: Open Closed Principle

Tópico 4: Liskov Substitution Principle

Tópico 5: Interface Segregation Principle

Tópico 6: Dependency Inversion Principle



Design Patterns

Módulo 4



A series of overlapping geometric shapes, primarily triangles and quadrilaterals, in shades of teal and purple, located in the top-left corner of the slide.

AceLeraDev C#

DESIGN PATTERNS

Módulo 4

A series of overlapping geometric shapes, primarily triangles and quadrilaterals, in shades of teal and purple, located in the bottom-right corner of the slide.



Tópicos desta aula:

Tópico 1: O que são Design Patterns

Tópico 2: Padrões de design Criacional

Tópico 3: Padrões de design Estruturais

Tópico 4: Padrões de design Comportamentais



O que são Design Patterns?

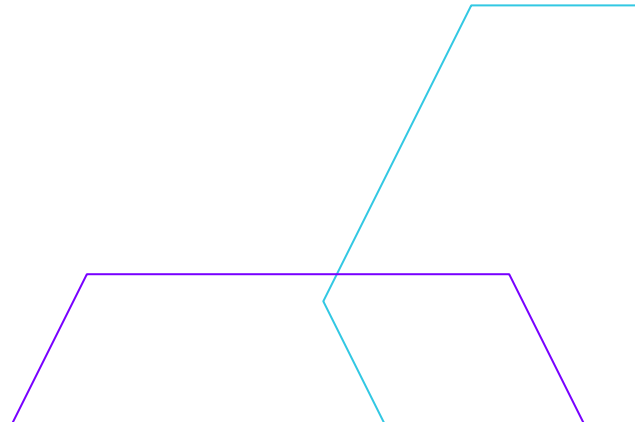
Na engenharia de software, um **padrão de design** é uma solução repetível geral para um problema comum no design de software. Um padrão de design não é um design final que pode ser transformado diretamente em código. É uma descrição ou modelo de como resolver um problema que pode ser usado em muitas situações diferentes.

A reutilização de padrões de design ajuda a evitar problemas sutis que podem causar grandes problemas e melhorar a legibilidade do código para codificadores e arquitetos familiarizados com os padrões



O que são Design Patterns?

Os padrões de design são classificados em três sub-classificações, com base no tipo de problema que eles resolvem:

- Padrões de design Criacional
 - Padrões de design Estruturais
 - Padrões de design Comportamentais
- 

Padrões de Design Criacional

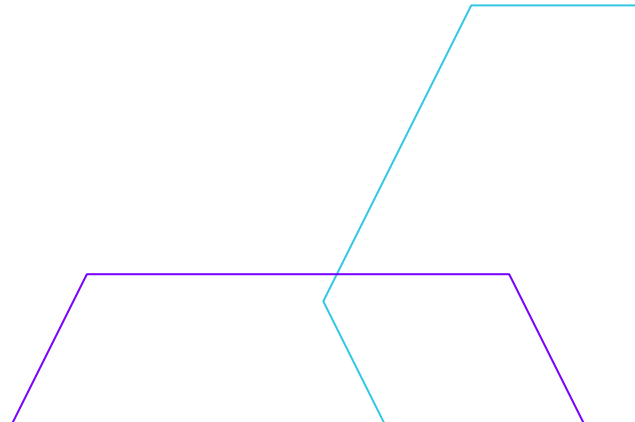
Os padrões criacionais fornecem vários mecanismos de criação de objetos, que aumentam a flexibilidade e a reutilização do código existente.

Esses padrões de design são sobre instanciação de classe. Esse padrão pode ser dividido em padrões de criação de classe e padrões de criação de objetos. Enquanto os padrões de criação de classe usam a herança efetivamente no processo de instanciação, os padrões de criação de objeto usam a delegação efetivamente para realizar o trabalho.



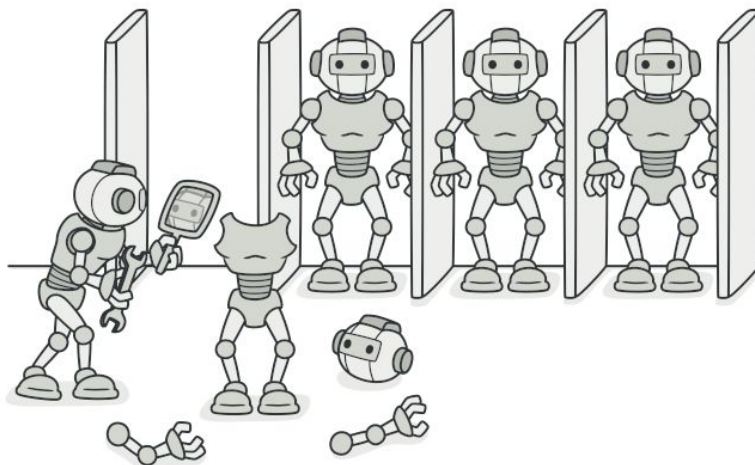
Padrões de Design Criacional

Entre vários temos:

- Factory Method
 - Abstract Factory
 - Builder
 - Prototype
 - Singleton
- 

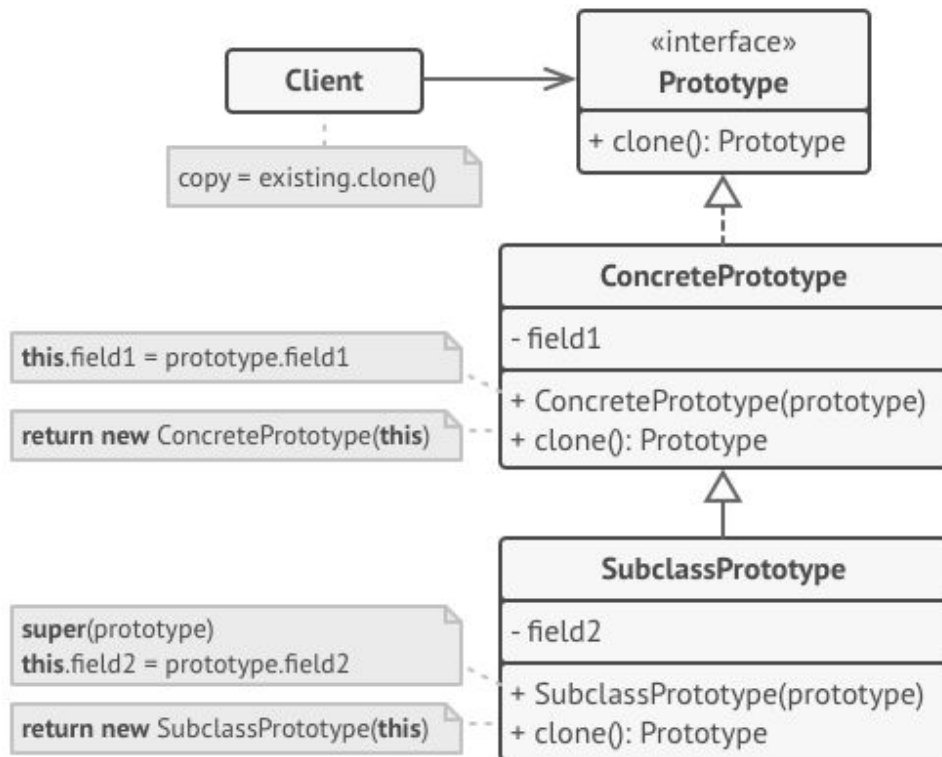
Prototype

É um padrão de design criacional que permite copiar objetos existentes sem tornar seu código dependente de suas classes.



3 O **cliente** pode produzir uma cópia de qualquer objeto que segue a interface do protótipo.

1 A interface **Prototype** declara os métodos de clonagem. Na maioria dos casos, é um `clone` método único.



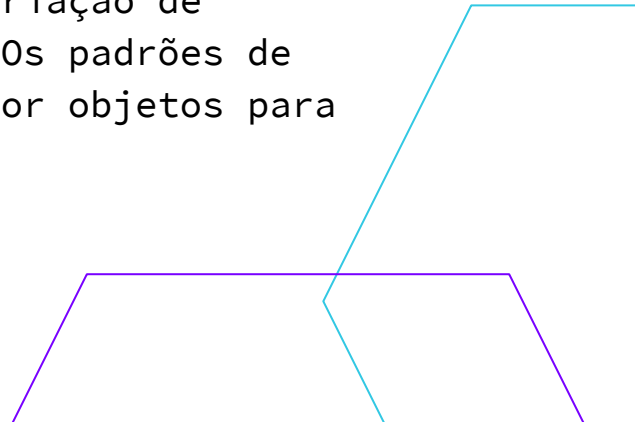
2 A classe **Concrete Prototype** implementa o método de clonagem. Além de copiar os dados do objeto original para o clone, esse método também pode lidar com alguns casos extremos do processo de clonagem relacionados à clonagem de objetos vinculados, desembaraço de dependências recursivas etc.



Padrões de Design Estruturais

Os padrões estruturais explicam como montar objetos e classes em estruturas maiores, mantendo essas estruturas flexíveis e eficientes.

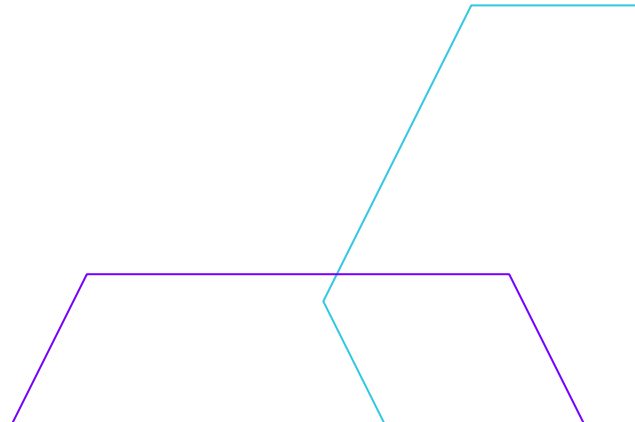
Esses padrões de design são todos sobre composição de classe e objeto. Os padrões estruturais de criação de classe usam herança para compor interfaces. Os padrões de objetos estruturais definem maneiras de compor objetos para obter novas funcionalidades.





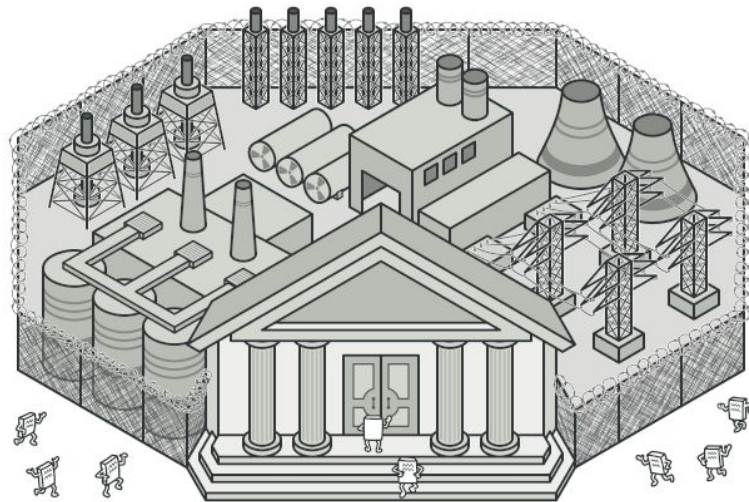
Padrões de Design Estruturais

Entre vários temos:

- Adapter
 - Bridge
 - Composite
 - Decorator
 - Facade
 - Flyweight
- 

Facade

É um padrão de design estrutural que fornece uma interface simplificada para uma biblioteca, uma estrutura ou qualquer outro conjunto complexo de classes.

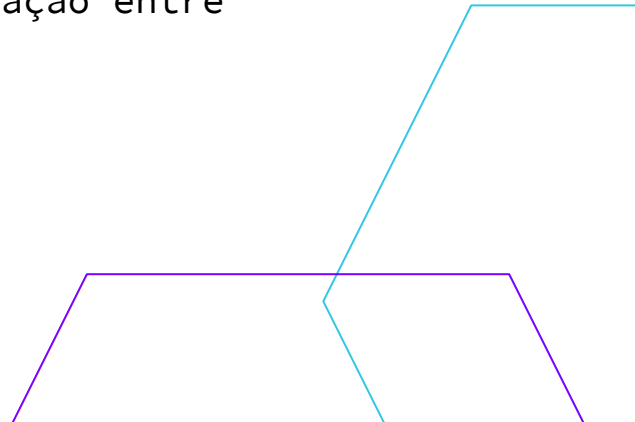




Padrões de Design Comportamentais

Os padrões de design comportamental preocupam-se com algoritmos e a atribuição de responsabilidades entre objetos.

Esses padrões de design são todos sobre comunicação de objetos da Classe. Padrões comportamentais são aqueles que se preocupam mais especificamente com a comunicação entre objetos.



The slide features decorative geometric lines in purple and teal. In the top-left corner, a purple line forms a large 'Z' shape, and a teal line forms a smaller 'Z' shape. In the bottom-right corner, a teal line forms a large 'Z' shape, and a purple line forms a smaller 'Z' shape.

Padrões de Design Comportamentais

Entre vários temos:

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer

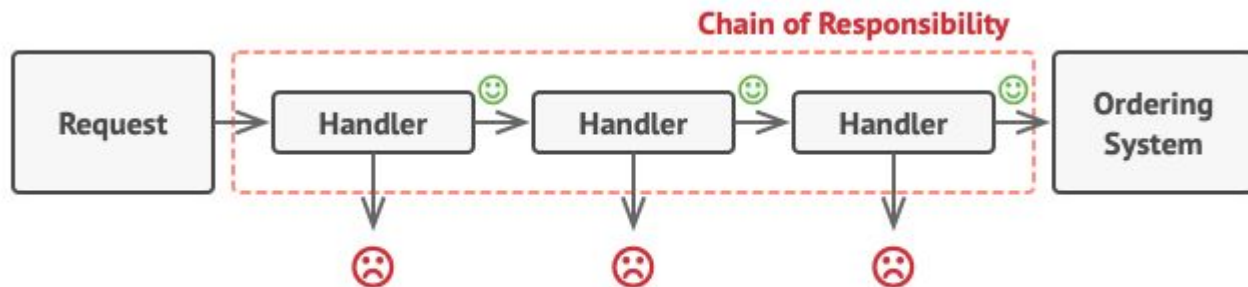
Chain of Responsibility

É um padrão de design comportamental que permite passar solicitações ao longo de uma cadeia de manipuladores. Ao receber uma solicitação, cada manipulador decide processar a solicitação ou passá-la para o próximo manipulador na cadeia.

- Conta com a transformação de comportamentos específicos em objetos independentes chamados *manipuladores*
- Cada manipulador vinculado possui um campo para armazenar uma referência ao próximo manipulador na cadeia

Chain of Responsibility

- Um manipulador pode decidir não passar a solicitação mais adiante na cadeia e interromper efetivamente qualquer processamento adicional

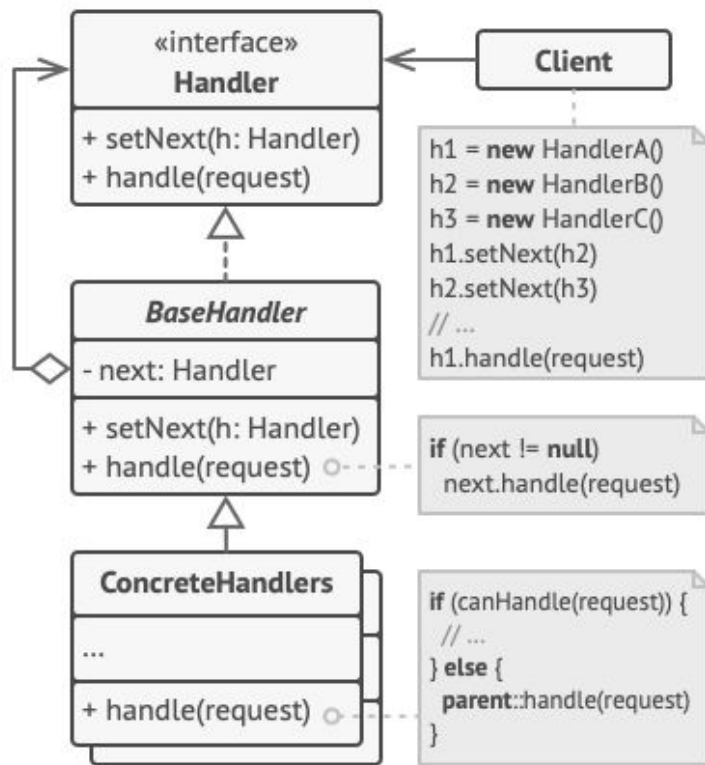


Os manipuladores são alinhados um a um, formando uma corrente.

1 O **manipulador** declara a interface, comum a todos os manipuladores de concreto. Geralmente, ele contém apenas um método para lidar com solicitações, mas às vezes também pode ter outro método para configurar o próximo manipulador na cadeia.

2 O **manipulador de base** é uma classe opcional na qual você pode colocar o código padrão comum a todas as classes de manipulador.

Normalmente, essa classe define um campo para armazenar uma referência ao próximo manipulador. Os clientes podem construir uma cadeia passando um manipulador ao construtor ou setter do manipulador anterior. A classe também pode implementar o comportamento de manipulação padrão: pode passar a execução para o próximo manipulador após verificar sua existência.



4 O **cliente** pode compor cadeias apenas uma vez ou compor dinamicamente, dependendo da lógica do aplicativo. Observe que uma solicitação pode ser enviada para qualquer manipulador da cadeia - não precisa ser a primeira.

3 **Manipuladores de concreto** contêm o código real para processar solicitações. Ao receber uma solicitação, cada manipulador deve decidir se deve processá-la e, adicionalmente, se deve transmiti-la pela cadeia.

Os manipuladores geralmente são independentes e imutáveis, aceitando todos os dados necessários apenas uma vez por meio do construtor.



Revisão do que vimos hoje:

Tópico 1: O que são Design Patterns

Tópico 2: Padrões de design Criacional

Tópico 3: Padrões de design Estruturais

Tópico 4: Padrões de design Comportamentais



AceLeraDev C#

TDD

Módulo 4



Tópicos desta aula:

Tópico 1: Teste de unidade

Tópico 2: Gerenciador de testes

Tópico 3: xUnit

Tópico 4: TDD



Teste de Unidade

Verifique se seu código está funcionando conforme o esperado criando e executando testes de unidade. É chamado de teste unitário porque você divide a funcionalidade do seu programa em comportamentos discretos e testáveis que você pode testar como *unidades* individuais.

O objetivo dos testes unitário é testar a menor funcionalidade existente do software, ou seja, isolar parte dos códigos e métodos, e analisar se essas funcionalidades tenham o retorno esperado mediando a um valor informado.

Teste de Unidade

O teste de unidade tem o maior efeito sobre a qualidade do código quando é parte integrante do fluxo de trabalho de desenvolvimento de software.

Mesmo com toda a indústria gritando as vantagens para quem queira ouvir, ainda existem mitos em torno da prática. O desenvolvedor agora vai gastar mais tempo escrevendo testes do que programando? Escrever testes dá trabalho. Testes manuais não são mais produtivos?



Teste de Unidade

O padrão AAA (Arrange, Act, Assert) é uma maneira comum de escrever testes de unidade para um método em teste.

- A seção **Organizar** (Arrange) de um método de teste de unidade inicializa os objetos e define o valor dos dados que são passados para o método sendo testado.
- A seção **Agir** (Act) invoca o método sendo testado com os parâmetros organizados.
- A seção **Declarar** (Assert) verifica se a ação do método em teste se comporta conforme o esperado.


```
[TestMethod]
public void Withdraw_ValidAmount_ChangesBalance()
{
    // arrange
    double currentBalance = 10.0;
    double withdrawal = 1.0;
    double expected = 9.0;
    var account = new CheckingAccount("JohnDoe", currentBalance);

    // act
    account.Withdraw(withdrawal);

    // assert
    Assert.AreEqual(expected, account.Balance);
}
```

```
[TestMethod]
public void Withdraw_AmountMoreThanBalance_Throws()
{
    // arrange
    var account = new CheckingAccount("John Doe", 10.0);

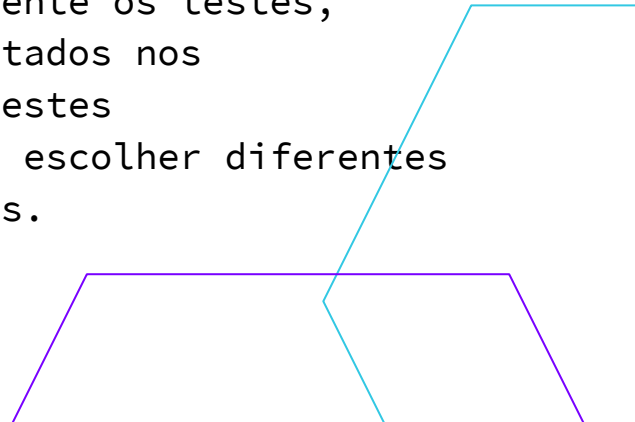
    // act and assert
    Assert.ThrowsException<System.ArgumentException>(() => account.Withdraw(20.0));
}
```



Gerenciador de testes

Quando você cria o projeto de teste, os testes são exibidos no Gerenciador de Testes. Se o Gerenciador de Testes não estiver visível, escolha Teste no menu do Visual Studio, Windows e, em seguida, Gerenciador de Testes.

Conforme você executa, grava e executa novamente os testes, o Gerenciador de Testes pode exibir os resultados nos grupos Testes com Falha, Testes Aprovados, Testes Ignorados e Testes Não Executados. Você pode escolher diferentes opções de agrupamento na barra de ferramentas.



Gerenciador de testes

The screenshot displays the Visual Studio Test Explorer interface. The top toolbar shows icons for running tests, with a summary of 6 tests, 3 passed, 1 failed, and 2 not run. The main pane on the left lists the test results:

Test	Duration
Failed (1)	65 ms
Withdraw_AmountMoreThanBalance_Throws	65 ms
Passed (3)	31 ms
Debit_WhenAmountIsLessThanBalance_Throws	29 ms
Debit_WhenAmountIsMoreThanBalance_Throws	1 ms
Debit_WithValidAmount_Throws	1 ms
Not Run (2)	
AccountInfo_CreateAccount_Throws	
WithdrawTest	

The right pane shows the 'Test Detail Summary' for the failed test 'Withdraw_AmountMoreThanBalance_Throws'.

Test Detail Summary

- Source: [BankAccountTests.cs line: 72](#)
- Duration: 65 ms

Message:

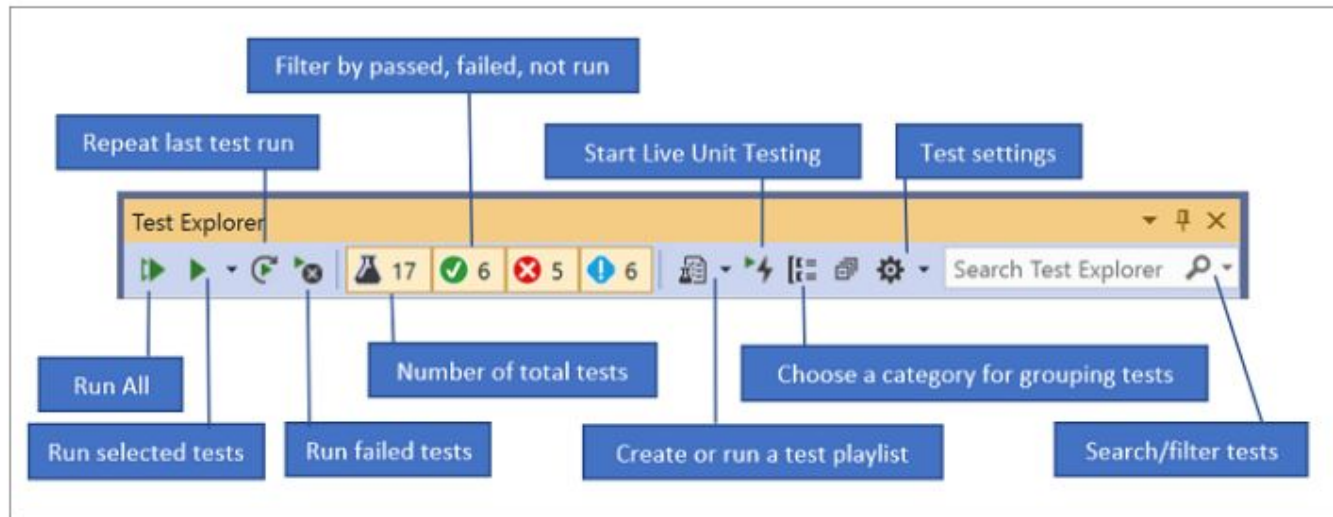
Assert.ThrowsException failed. Threw exception ArgumentOutOfRangeException, but expected exception type was ArgumentException. Exception Message: Debit amount exceeds balance. Parameter name: amount. Actual value was 20.

Stack Trace:

```
at Bank.BankAccount.Debit(Double amount) in C:\Code\VS docs\Bank\Bank\BankAccount.cs:line 10
at BankTests.BankAccountTests.<>c__DisplayClass3_0.<Withdraw_AmountMoreThanBalance_Throws>b__0() in BankAccountTests.cs:line 72
at Microsoft.VisualStudio.TestTools.UnitTesting.Assert.ThrowsException[T](Action action, String message, String source, String filePath, Int32 lineNumber) in Microsoft.VisualStudio.TestTools.UnitTesting.Assert.cs:line 1000
```

Gerenciador de testes

A barra de ferramentas do **Gerenciador de Testes** ajuda você a descobrir, organizar e executar os testes desejados.



xUnit

Nasceu a partir do NUnit. Os antigos criadores do NUnit se uniram com o objetivo de criar um framework de testes mais flexível e abrangente. Com isso encontramos muitas semelhanças entre ambos, apesar disso também existem diversas diferenças, como por exemplo, os nomes dos atributos; no XUnit não existe a necessidade de tornar uma classe testável como no `MSTest[TestClass]` e `NUnit[TestFixture]`, neste basta decorar os métodos com os atributos `[Facts]` e `[Theory]`, para testar métodos sem e com parâmetros respectivamente;

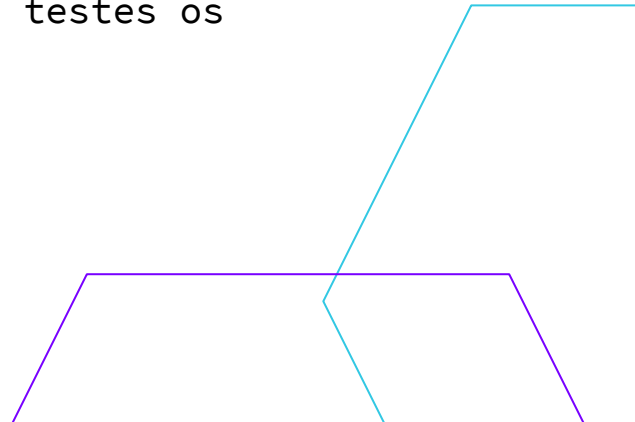


**TALK IS CHEAP
SHOW ME
THE CODE**



Test-Driven Development

A ideia é bem simples: escreva seus testes antes mesmo de escrever o código de produção. Mas por quê a ideia parece tão boa? Ao escrever os testes antes, o desenvolvedor garante que boa parte (ou talvez todo) do seu sistema tem um teste que garante o seu funcionamento. Além disso, muitos desenvolvedores também afirmam que os testes os guiam no projeto de classes do sistema.

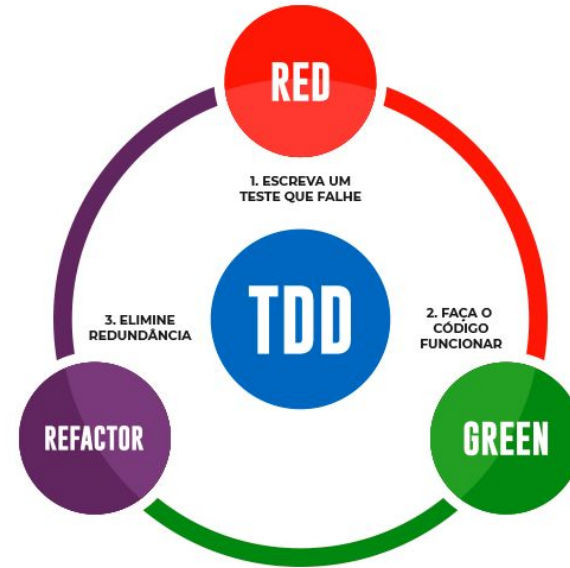


Test-Driven Development

o TDD se baseia em pequenos ciclos de repetições, onde para cada funcionalidade do sistema um teste é criado antes. Este novo teste criado inicialmente falha, já que ainda não temos a implementação da funcionalidade em questão e, em seguida, implementamos a funcionalidade para fazer o teste passar! Simples assim!

Quando o teste estiver ok, é preciso que esta funcionalidade que acabamos de escrever seja refatorada, ou seja, ela precisa passar por um pequeno banho de "boas práticas" de Desenvolvimento de Software. Estas boas práticas que garantirão um software com código mais limpo, coeso e menos acoplado.

Test-Driven Development





Revisão do que vimos hoje:

Tópico 1: Teste de unidade

Tópico 2: Gerenciador de testes

Tópico 3: xUnit

Tópico 4: TDD





AceLeraDev C#

CLEAN CODE E GIT VS

Módulo 4

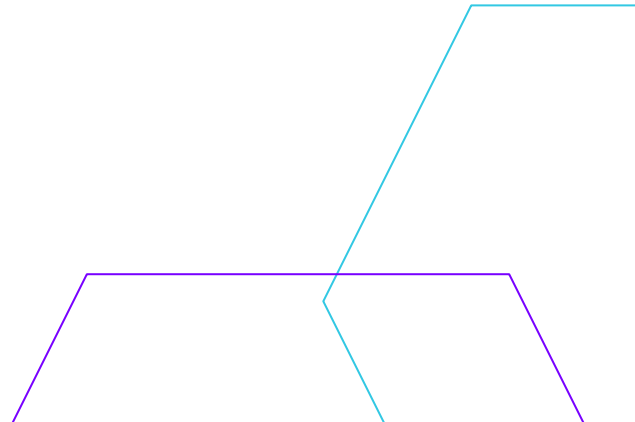




Tópicos desta aula:

Tópico 1: Clean Code

Tópico 2: Git no Visual Studio



Clean Code

Aprender a criar códigos limpos é uma tarefa árdua e requer mais do que o simples conhecimento dos princípios e padrões. Você deve suar a camisa; praticar sozinho e ver que cometeu erros; assistir os outros praticarem e errarem; vê-los tropeçar e refazer seus passos; Vê-los agonizar para tomar decisões e o preço que pagarão por as terem tomado da maneira errada.

Clean Code é uma filosofia de desenvolvimento cuja o principal objetivo é aplicar técnicas simples que visam facilitar a escrita e leitura de um código, tornando-o de fácil compreensão e revelando a sua real intenção.

Clean Code

- Converter métodos grandes em métodos menores
- Renomear variáveis
- Diminuir a complexidade do código
- Reduzir condicionais
- Eliminar códigos duplicados

“Qualquer um consegue escrever código que um computador entende. Bons programadores escrevem código que humanos entendem” – Martin Fowler



Revisão do que vimos hoje:

Tópico 1: Clean Code

Tópico 2: Git no Visual Studio





Tópicos da próxima aula:

Tópico 1: Banco de dados

