

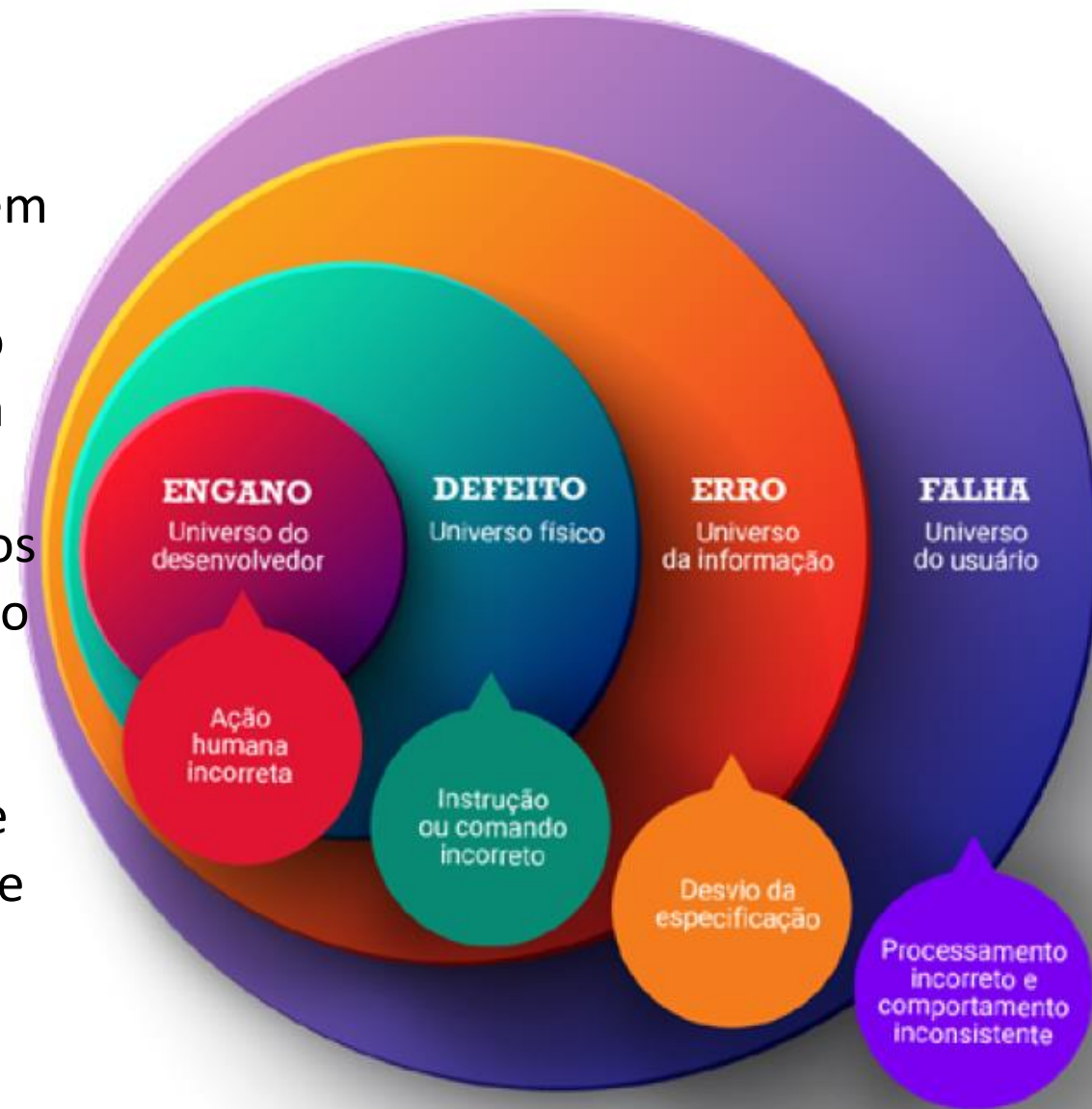
# Testes e Qualidade em Jogos

5º período

Professora: Michelle Hanne

# Terminologia de Testes

- **Engano:** é uma ação de alguém que produz um defeito.
- **Erro:** é uma manifestação concreta de um defeito em um artefato de software. Diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa constitui um erro.
- **Defeito:** é um passo, processo ou definição de dados incorretos ou inconsistentes. Por exemplo: utilização incorreta de uma instrução ou uso de comando incorreto para determinada operação.
- **Falha:** é o comportamento operacional do software diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos erros e alguns erros podem nunca causar uma falha.



# Tipos de Testes

Os tipos de teste vão diferir quanto às suas técnicas. As técnicas existentes são: **técnica funcional (caixa-preta) e estrutural (caixa-branca).**

# Tipos de Testes

- Com relação aos testes do tipo **caixa-preta**:

**Regressão:** toda vez que algo for mudado, deve ser testada toda a aplicação novamente.

**Requisitos:** verifica se o sistema é executado conforme o que foi especificado. São realizados através da criação de condições de testes e *cheklists* de funcionalidades.

**Controle:** assegura que o processamento seja realizado conforme sua intenção. Entre os controles estão a validação de dados, a integridade dos arquivos, as trilhas de auditoria, o backup e a recuperação, a documentação, entre outros.

**Usabilidade:** tem por objetivo verificar a facilidade que o software ou site possui de ser claramente compreendido e manipulado pelo usuário.

**Aceitação:** testa se a solução será bem avaliada pelo usuário. Ex.: caso exista um botão pequeno demais para executar alguma operação (aqui cabem itens fora da interface também).

# Tipos de Testes

Com relação aos testes do tipo **caixa-branca**:

**Desempenho:** verifica se o tempo de resposta é o desejado para o momento de utilização da aplicação.

**Carga:** verifica o funcionamento da aplicação com a utilização de uma quantidade grande de usuários simultâneos.

**Estresse:** testa a aplicação sem situações inesperadas. Testa caminhos, às vezes, antes não previstos no desenvolvimento/documentação.

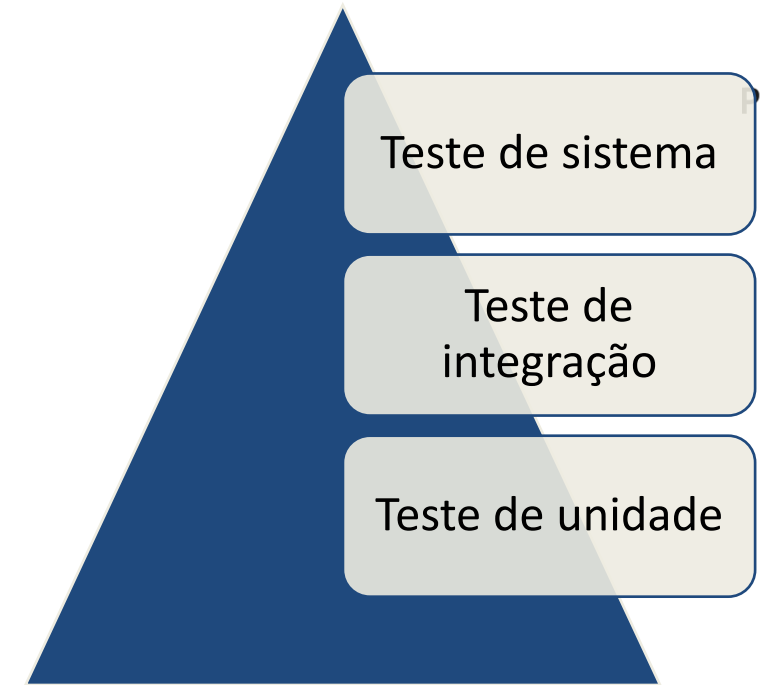
**Conformidade:** verifica se o software foi desenvolvido de acordo com padrões, normas, procedimentos e guias de TI.

**Contingência:** verifica se o sistema é capaz de retornar ao estado anterior antes da falha.

**Segurança:** avalia a adequação dos procedimentos de proteção e as contramedidas projetadas para garantir a confidencialidade das informações e a proteção dos dados contra o acesso não autorizado de terceiros.

# Fases ou níveis de Testes

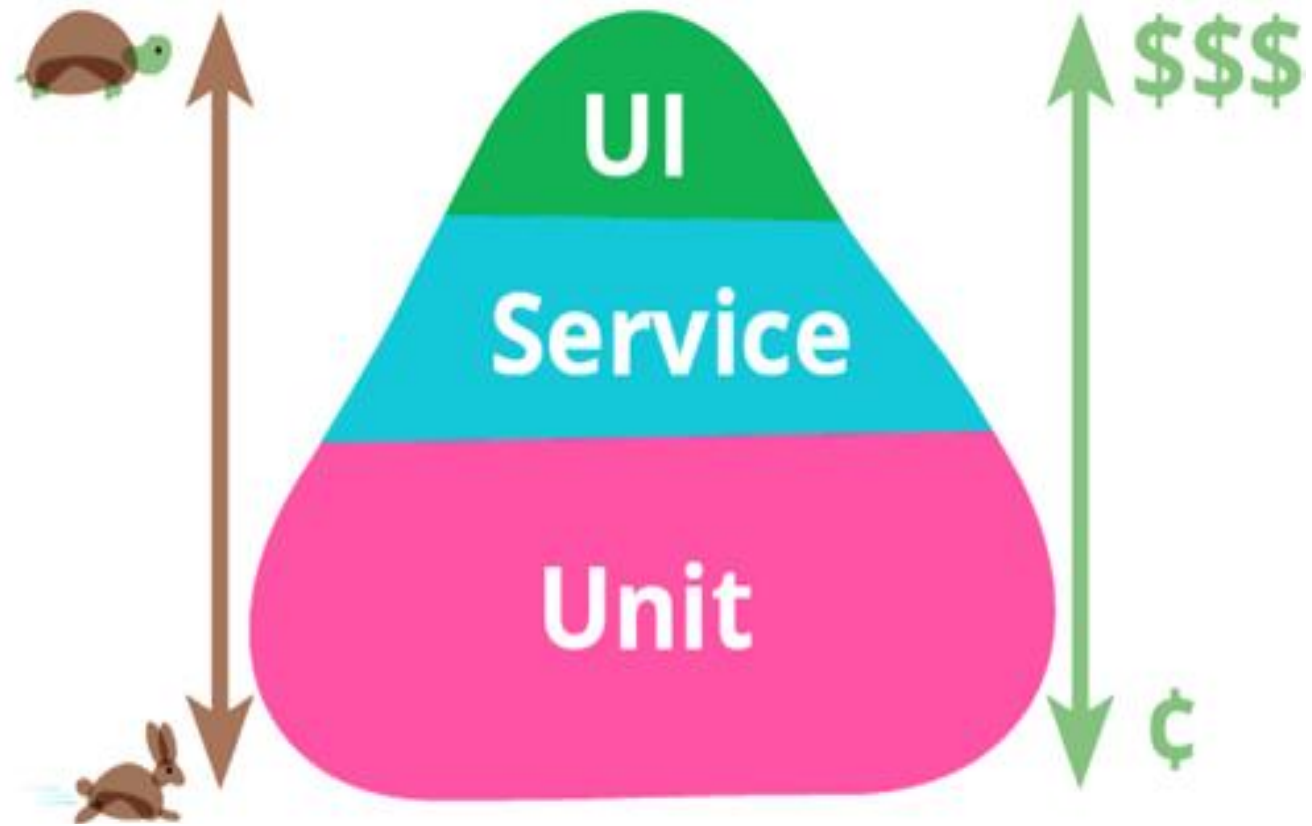
- O teste de unidade tem como **foco as menores unidades de um programa**, que podem ser **funções, procedimentos, métodos ou classes**.
- Nesse contexto, espera-se que sejam **identificados erros relacionados a algoritmos incorretos ou mal implementados, estruturas de dados incorretas ou simples erros de programação**.



Como cada unidade é testada separadamente, o teste de unidade pode ser aplicado à medida que ocorre a implementação das unidades e pelo próprio desenvolvedor, sem a necessidade de dispor-se do sistema totalmente finalizado (DELAMARO *et al.*, 2016).

# Fases ou níveis de Testes

Testes de unidade ou unitários são mais baratos e rápidos (como coelhos) que testes de usuários (UI) que são mais caros e lentos (como tartarugas).



# Testes Unitários na Unity

O Unity Test Framework é o plug-in oferecido pela Unity para criar e executar os testes unitários dentro do editor. Com ele é possível ver os testes criados, executá-los e checar o resultado.

- O Unity Test Framework (UTF) permite que os usuários do Unity testem seu código nos modos Edit e Play, e também em plataformas de destino, como Standalone, Android, iOS e outras.
- O UTF usa uma integração Unity da biblioteca NUnit, que é uma biblioteca de teste de unidade de código aberto para idiomas .Net. Para obter mais informações sobre o NUnit.



# Testes de Integração

Os testes de integração devem ser realizados após os testes unitários. A ênfase é dada na construção da estrutura do sistema. À medida que as diversas partes do software são colocadas para trabalhar juntas, é preciso verificar se a interação entre elas funciona de maneira adequada e não leva a erros.

Nesse caso é necessário um grande conhecimento das estruturas internas e das interações existentes entre as partes do sistema e, por isso, o teste de integração tende a ser executado pela própria equipe de desenvolvimento (DELAMARO *et al.*, 2016).

# Testes de Integração

O teste de integração possui duas estratégias para sua realização: as abordagens **bottom-up** e **top-down** (LOURENÇO, 2010).

Na abordagem **BOTTOM-UP**, o software é desenvolvido a partir de rotinas, funções e métodos básicos que fornecem serviços às demais rotinas, funções ou métodos de uma camada superior do software.

- Por exemplo, uma verificação de CPF pode ser chamada em vários pontos de um sistema de software.
- Provavelmente será uma das primeiras a serem implementadas.

Nesse exemplo, utiliza-se o recurso conhecido como **stubs**, que são rotinas vazias que retornam valores explícitos, definidos pelo programador.

# Testes de Integração

Na realização de testes para a abordagem de desenvolvimento **Bottom-up**, devem ser escritos códigos **que invoquem as rotinas de camadas inferiores, testando-as com diversas combinações de parâmetros.** As rotinas escritas para esses testes são conhecidas como **drivers**, pois sua função é acionar o código que deve ser testado.

# Testes de Integração

Já na abordagem **top-down**, faz-se o contrário. O desenvolvedor elabora seu código supondo que o código de uma camada inferior do sistema de software já esteja pronto.

- Utilizando-se o exemplo anterior de verificação de CPF, o desenvolvedor pode codificar chamadas (fazer o uso nominal da função) para a verificação de CPF, mesmo sabendo que ela ainda não existe.
- Nesse exemplo, utiliza-se o recurso conhecido como **stubs**, que são rotinas vazias que retornam valores explícitos, definidos pelo programador.

# Testes de Integração

**Drivers:** são responsáveis pelo controle do teste de uma unidade ou conjuntos de unidades de um sistema. É uma operação que exercita o módulo sob teste, envia valores (dados de entrada dos casos de teste), coleta e compara resultados.

**Stubs:** são implementações que simulam determinadas situações esperadas. Normalmente possuem a mesma assinatura dos métodos, funções ou rotinas que se deseja simular, porém retornam um comportamento fixo determinado pelo desenvolvedor.

# Testes de Regressão e Aceitação

Existem também os testes de aceitação e testes de regressão

## Teste de aceitação:

- também **conhecido por UAT (User Acceptance Tests)** tem por objetivo verificar a **conformidade com os requisitos de negócio** e usuário na última fase do ciclo de desenvolvimento, validando o produto para entrega.

## Teste de regressão:

- **acontece a cada modificação efetuada no sistema** após a liberação de nova versão. Nesses processos, corre-se o risco de que novos defeitos sejam introduzidos

De maneira geral, os testes de regressão tratam de uma nova execução do conjunto de casos de teste já executados (DELAMARO et al., 2016).

# Testes de Sistema

Depois que se tem o **sistema completo**, com todas as suas partes integradas, inicia-se o teste de sistema.

O **objetivo** é verificar se as **funcionalidades** especificadas nos **documentos de requisitos** estão todas corretamente implementadas.

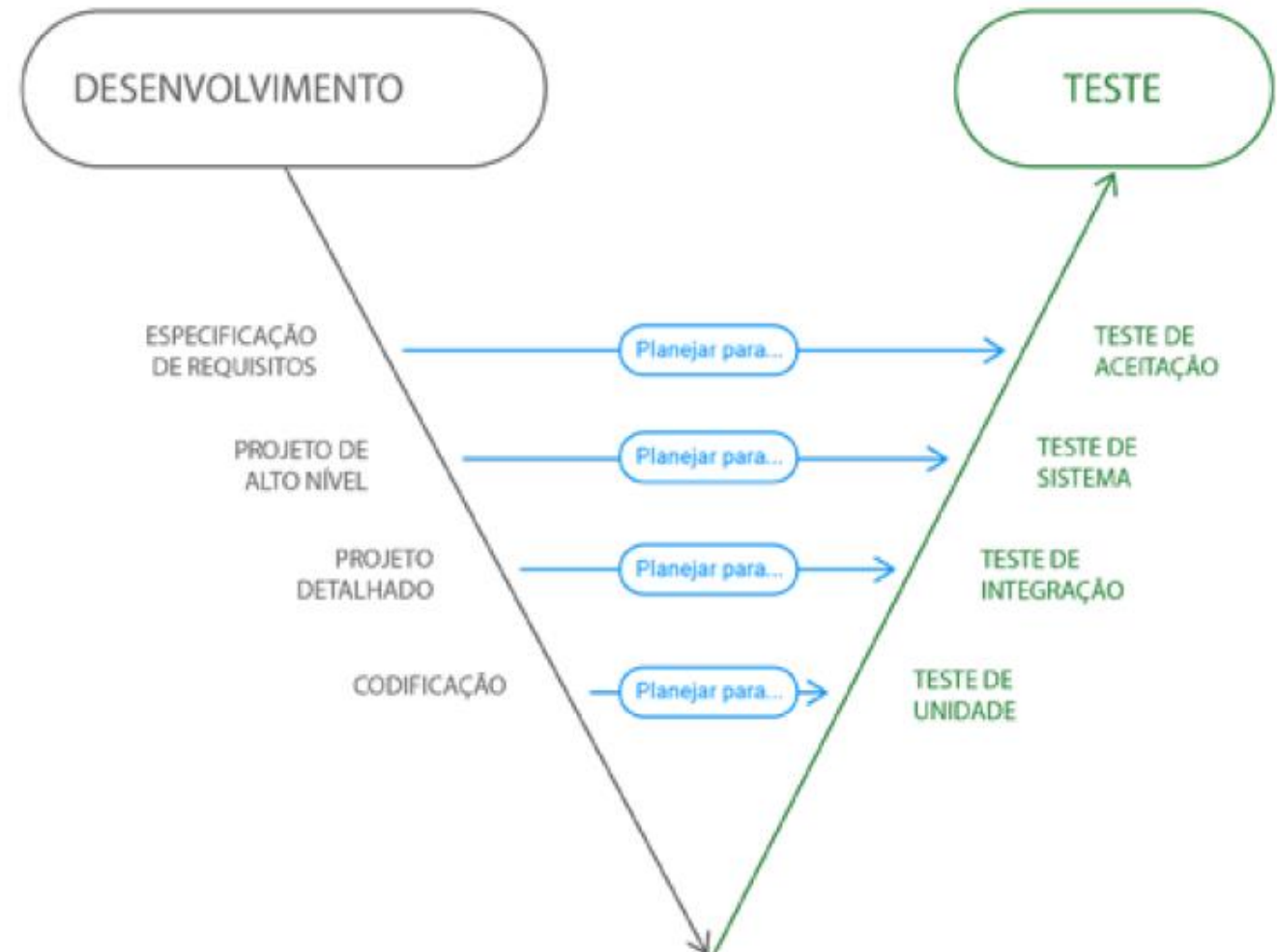
# Testes de Sistema

Os seguintes pontos  
sumarizam o teste de sistema  
(SOMMERVILLE, 2011):

- Se os componentes são compatíveis;
- Se eles interagem corretamente;
- Se transferem os dados certos no momento certo, por meio de suas interfaces

## NÍVEIS DE TESTE

Os níveis ou fases de teste possuem diferenças entre si, principalmente quanto ao seu escopo, time, origem dos dados, volume dos dados, interfaces e ambientes.



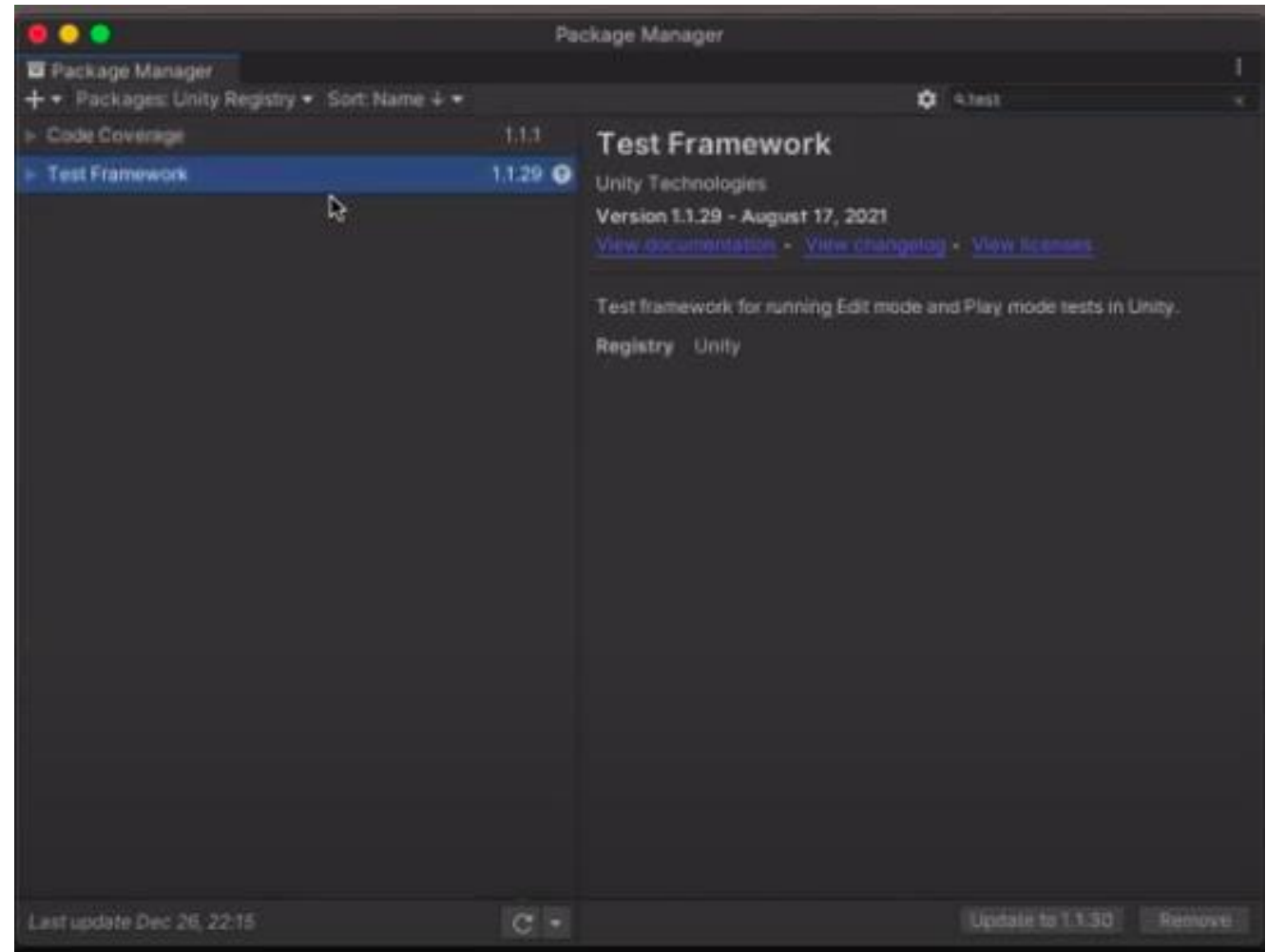


# Testes Unitários na Unit

Instalar ao seu projeto o pacote Test Framework, através do Gerenciador de pacotes.

**Acessar o menu Window → Package Manager**

Em seguida buscar o nome do pacote **Test Framework** e instalar



# Testes Unitários na Unit

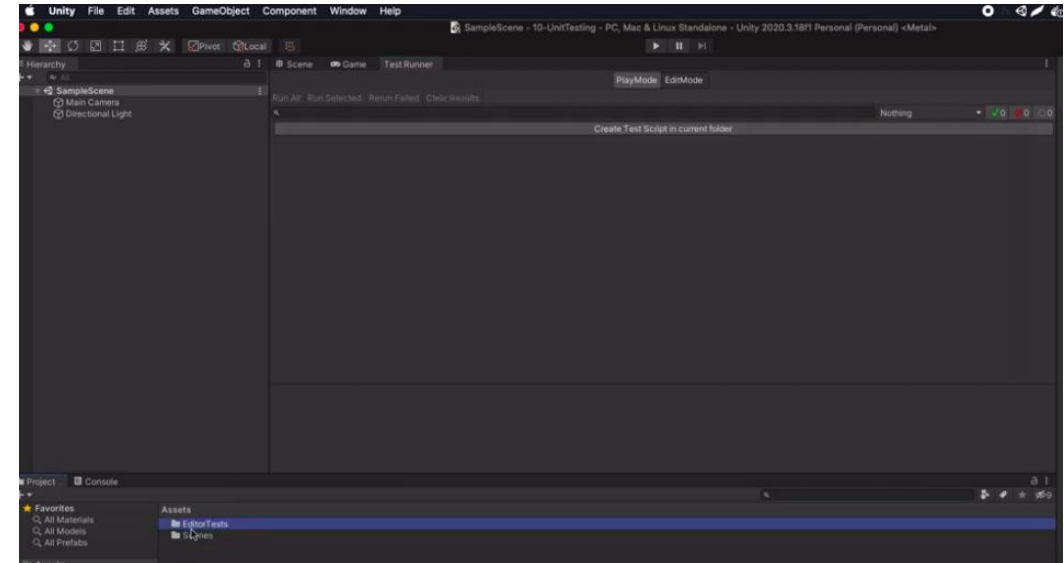
Assim que for adicionado ao seu projeto você visualizará uma nova opção em **Window**→**General**→**Test Runner**



# Testes Unitários na Unit

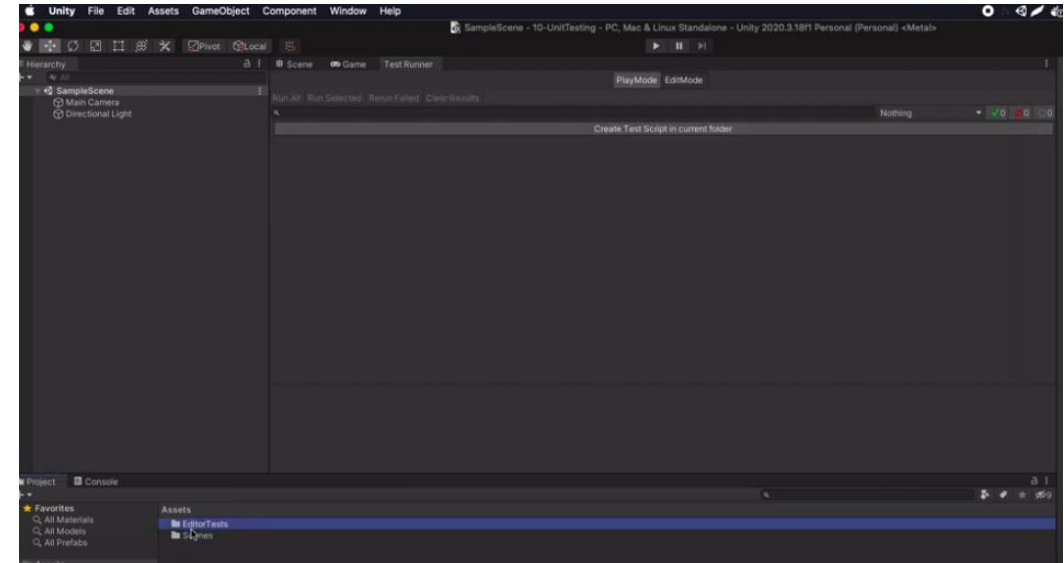
O Test Framework utiliza por padrão **Assembly Definitions Files**, arquivos criados para sobrepor a arquitetura padrão da Unity de utilizar apenas um Assembly para todos os scripts do projeto. Com os **Assembly Definitions Files**, os scripts são separados em assemblies diferentes, deixando a compilação mais rápida e o código separado.

Assemblies são basicamente uma forma de agrupar vários scripts e dizer ao compilador que são um único projeto, assim, ajuda a organizar o código e suas dependências, reduzindo o tempo de compilação.



# Testes Unitários na Unit

- Temos que configurar o **Assembly** para usar os **testes unitários**. Para isso clique no botão **“Create EditMode TestAssembly Folder”** para criar uma pasta para armazenar o Assets de teste – *Exemplo EditorTests*
- Os testes podem ser executados nos modos **PlayMode** e **EditMode**

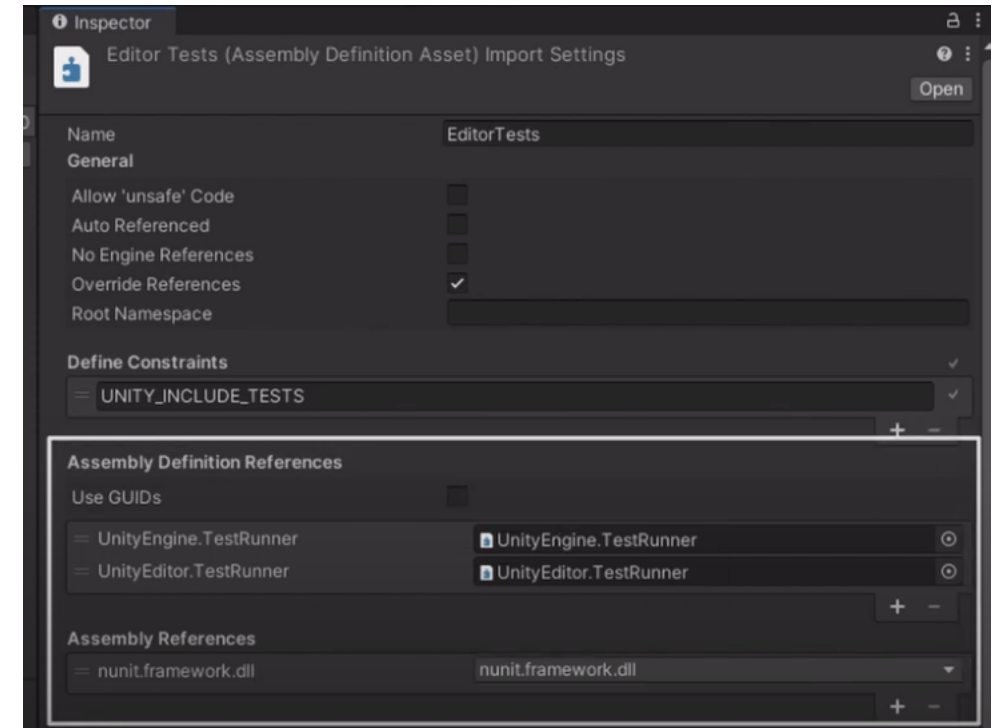


# Testes Unitários na Unit

É possível clicar e inspecionar as propriedades do Editor Tests.

**Exemplo de Dependência:**

**A dependência da biblioteca  
nunit.framework.dll C# e Unity  
Editor.TestRunner .NET**



# Criando o primeiro Teste

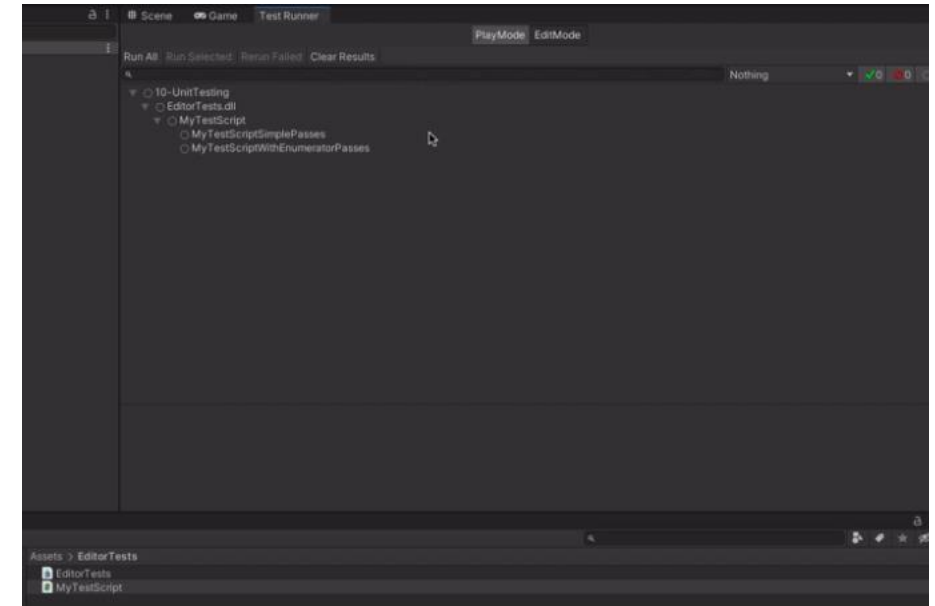
Estando em Test Runner clique em **Play Mode**. Será adicionado um novo Script C# na mesma pasta do seu Asset “**Editor Tests**”.

Assim que compilado você verá a lista dos testes listados na janela do executor. **O framework de teste cria uma estrutura hierarquica:**

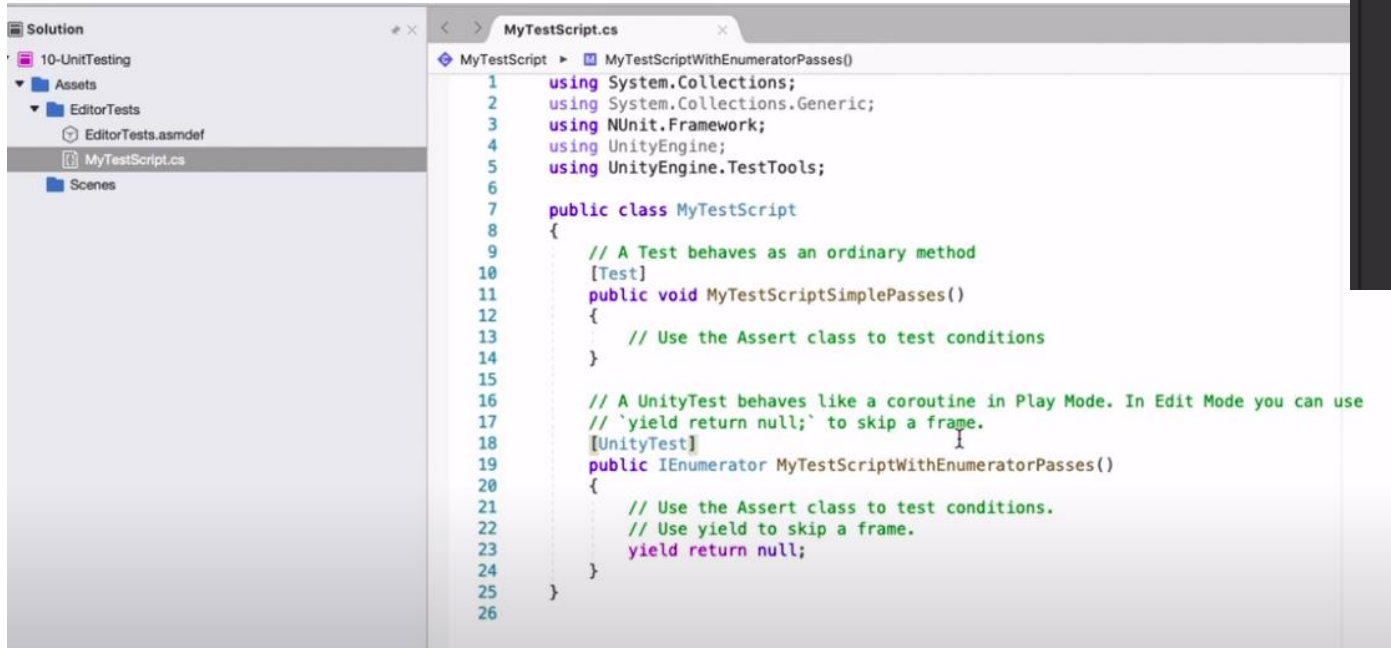
Editor Test.dll → conjunto de testes

MyTestScript → Classe de teste

**MyTestScriptSimplePasses** e **MyTestScriptWithEnumeratorPasses** - > casos de Testes – onde são escritos os códigos, testes unitários.



# Criando o primeiro Teste

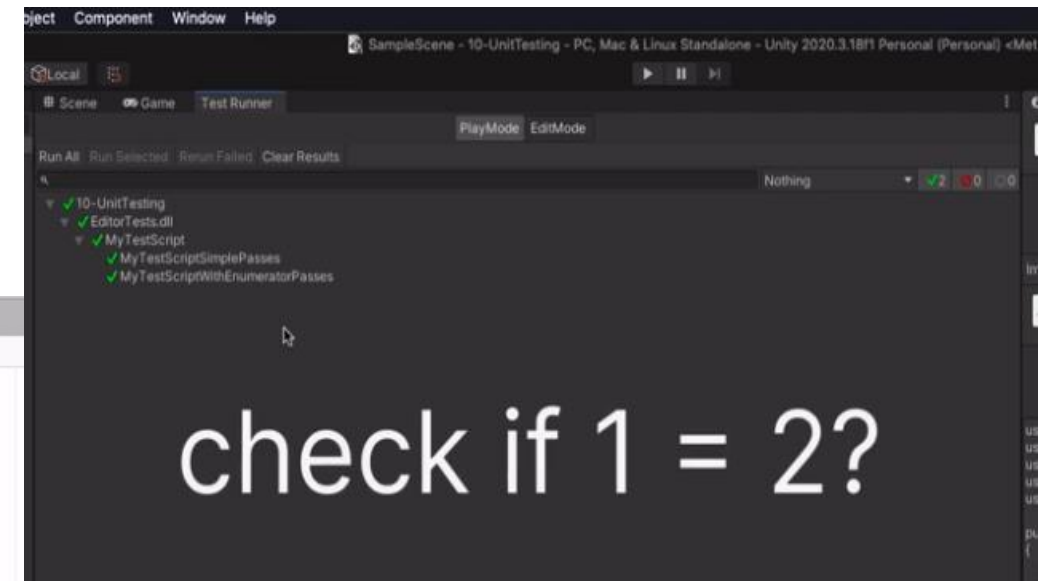
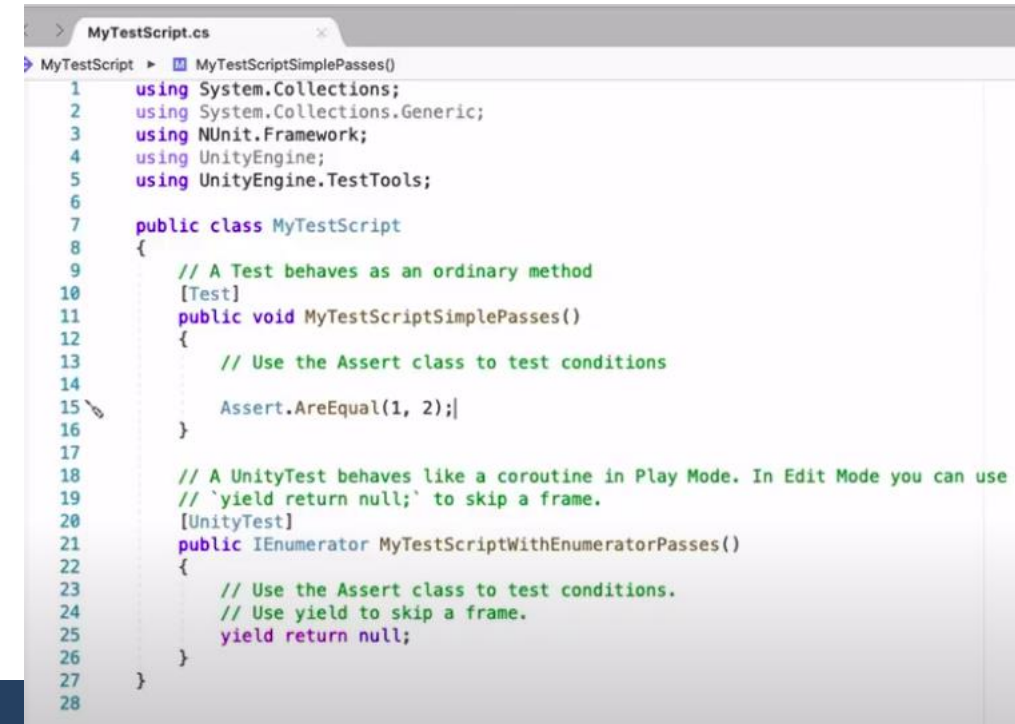


The Solution Explorer on the left shows the project structure: 10-UnitTesting > EditorTests > EditorTests.asmdef > MyTestScript.cs. The main editor shows the code for MyTestScript.cs:

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using NUnit.Framework;
4  using UnityEngine;
5  using UnityEngine.TestTools;
6
7  public class MyTestScript
8  {
9      // A Test behaves as an ordinary method
10     [Test]
11     public void MyTestScriptSimplePasses()
12     {
13         // Use the Assert class to test conditions
14     }
15
16     // A UnityTest behaves like a coroutine in Play Mode. In Edit Mode you can use
17     // 'yield return null;' to skip a frame.
18     [UnityTest]
19     public IEnumerator MyTestScriptWithEnumeratorPasses()
20     {
21         // Use the Assert class to test conditions.
22         // Use yield to skip a frame.
23         yield return null;
24     }
25 }
26

```

The code editor shows the MyTestScript.cs file with the MyTestScriptSimplePasses() method highlighted. The code is as follows:

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using NUnit.Framework;
4  using UnityEngine;
5  using UnityEngine.TestTools;
6
7  public class MyTestScript
8  {
9      // A Test behaves as an ordinary method
10     [Test]
11     public void MyTestScriptSimplePasses()
12     {
13         // Use the Assert class to test conditions
14
15         Assert.AreEqual(1, 2);
16     }
17
18     // A UnityTest behaves like a coroutine in Play Mode. In Edit Mode you can use
19     // 'yield return null;' to skip a frame.
20     [UnityTest]
21     public IEnumerator MyTestScriptWithEnumeratorPasses()
22     {
23         // Use the Assert class to test conditions.
24         // Use yield to skip a frame.
25         yield return null;
26     }
27 }
28

```

# Referências

Livro *Artificial Intelligence for Games, Second Edition* Capítulo 5: *Decision Making*

Link:

[https://spada.uns.ac.id/pluginfile.php/629724/mod\\_resource/content/1/gameng\\_AIFG.pdf](https://spada.uns.ac.id/pluginfile.php/629724/mod_resource/content/1/gameng_AIFG.pdf)