temática ISSN 1807-8931

Implementação de testes unitários na Engine Unity: estudo de caso do jogo Fantasy Stars

Unit test implementation at Engine Unity: a case study with Fantasy Stars game

Carlos CADORI[1]
Ewerton Eyre de Morais ALONSO[2]

Resumo

O objetivo deste artigo é, a partir de um estudo de caso do desenvolvimento do jogo *Fantasy Stars*, documentar e discutir o processo de implementação de testes unitários no ambiente do motor de jogos Unity. A discussão se dá por meio da análise das particularidades do ambiente de desenvolvimento de testes da *engine*, assim como os pontos positivos e negativos do uso de diferentes técnicas e ferramentas como *Mocking* e *PlayMode*. Como conclusão, o estudo de caso válida a utilização de ambas as técnicas no contexto da *engine*, de forma a maximizar a cobertura dos testes, destacando também que existem pontos negativos, como o aumento no tempo de desenvolvimento, que devem ser levados em consideração e avaliados no contexto de cada projeto individualmente.

Palavras-chave: *Fantasy Stars*. Testes Unitários. Testes Automatizados. Unity.

Abstract

The purpose of this article is, from a case study of Fantasy Stars game development, to document and discuss the process of implementing unit testing in the Unity game engine environment. The discussion takes place by analyzing the particularity of the engine's test development environment, as well as the strengths and weaknesses of using different techniques and tools such as Mocking and PlayMode. In conclusion, the case study validates the use of both techniques in the context of the engine in order to maximize test coverage, also highlighting that there are negative points, such as increased development time, that must be taken into account and evaluated in the context of each project individually.

Keywords: Fantasy Stars. Unit Tests. Automated Tests. Unity.

[1] Graduado em Design de Jogos e Entretenimento Digital pela Universidade do Vale do Itajaí – UNIVALI, Balneário Camboriú/SC. E-mail: carloscadori.dev@gmail.com

[2] Mestre em Ciências da Computação pela Universidade Federal de Santa Catarina – UFSC. Professor e pesquisador do Curso de Design de Jogos e Entretenimento Digital pela Universidade do Vale do Itajaí – UNIVALI, Balneário Camboriú/SC. E-mail: ewertonalonso@univali.br

Introdução

Testes são uma importante ferramenta no desenvolvimento de *software*, possibilitando ao desenvolvedor encontrar falhas antes que o produto chegue ao cliente final (PRESSMAN, 2005). Entretanto, com o crescimento de um projeto, os testes manuais tendem a se tornar trabalhosos e propensos ao erro, o que leva muitos desenvolvedores a recorrerem aos testes automatizados (NETO, 2007).

Os testes de *software* são divididos em diferentes níveis de interação. Neste artigo será abordada a implementação de testes unitários, que tem como objetivo avaliar a lógica de módulos ou unidades individualmente (WHITTAKER, 2002).

A proposta do presente artigo é abordar as diferentes formas de implementação de testes unitários, utilizando a técnica de caixa-branca no ambiente da *engine* Unity, além de discutir os resultados obtidos a partir do experimento proposto.

A abordagem será desenvolvida a partir do estudo de caso do jogo *Fantasy Stars*, desenvolvido pelos autores deste artigo.

Metodologia

A metodologia empregada no presente artigo é o estudo de caso, de natureza exploratória, referente ao processo de implementação de testes unitários no jogo *Fantasy Stars*, desenvolvido pelos autores.

O estudo de caso é uma modalidade de pesquisa que consiste no estudo aprofundado de um ou poucos objetos, a fim de proporcionar uma visão global a respeito do tema, assim como possíveis fatores que o influenciam ou são influenciados por ele (GIL, 2002).

No caso do presente artigo, o estudo acontece a partir da análise do processo de implementação de testes unitários na produção do jogo *Fantasy Stars*, analisando as características e limitações do motor de jogo utilizado, implementando diferentes abordagens e realizando um comparativo dos resultados.

1 Testes unitários

Testes unitários correspondem ao primeiro nível dos testes de *software*. Possuem a finalidade de avaliar isoladamente pequenos módulos ou unidades em busca de erros de lógica ou implementação (WHITTAKER, 2002).

São usualmente desenvolvidos simultaneamente com a lógica que será testada, ou mesmo antes, como é o caso do *test-driven development* (TDD) (ZHU et al, 1997). Entretanto, os testes são executados localmente pelo desenvolvedor e o código de teste não deve estar presente no *build* final do produto (ZHU et al, 1997).

1.1 Ferramentas de teste na engine Unity

A Unity, motor de jogos utilizado no desenvolvimento do jogo *Fantasy Stars*, apresentado no presente artigo, possui um conjunto de ferramentas voltadas a diferentes níveis de testes automatizados.

O *Test Runner* é a principal ferramenta para o gerenciamento dos testes. A partir dela é possível configurar o ambiente, criar novos arquivos de teste, visualizar e executá-los em conjunto ou individualmente. Além disso, é possível visualizar os *logs* e resultados de falhas (UNITY, 2019).

A fim de auxiliar o desenvolvimento de testes unitários, o *framework* NUnit (PROUSE et al, 2019) vem previamente instalado junto do motor, disponibilizando diferentes atributos e formas de comparação, também conhecidos como *Asserts*.

A *engine* oferece, também, suporte a testes executados em modo de jogo (*play mode*), facilitando testes onde a lógica possua dependências atreladas a variáveis disponíveis apenas no decorrer do jogo (UNITY, 2019).

1.2 Unity e componentes

Atualmente, a *engine* Unity disponibiliza duas formas de desenvolvimento de jogos: por meio de uma estrutura baseada em componentes e de outra baseada em entidades e sistemas. Entretanto, atualmente apenas o padrão de componentes se encontra disponível a nível de produção (UNITY, 2019).

Os jogos criados são separados em cenas, cada cena possuindo uma lista de objetos chamados internamente de *GameObjects*. Os objetos por sua vez podem possuir um número indefinido de componentes, chamados de comportamentos.

Componentes são criados com *scripts* que derivam da classe interna *MonoBehaviour* da Unity, não podendo ser instanciados normalmente a partir de um construtor, mas sim utilizando o método *AddComponent* de um *GameObject* (NYSTROM, 2014).

1.3 Teste de caixa-branca

Teste de caixa-branca é uma técnica de modelagem de testes a partir da perspectiva interna da aplicação. Nesta abordagem, os casos de teste devem ser escritos analisando o código fonte, a fim de cobrir todas as condições do modulo que está sendo validado (NETO, 2007).

2 Discussão

Fantasy Stars é um jogo desenvolvido pelos autores do presente artigo, do gênero ação, *multiplayer* e publicado inicialmente na loja de aplicativos *GooglePlay*.

O jogo possui suas principais mecânicas inspiradas em futebol, como: chute, posse de bola, marcar gols, entre outras. Os jogadores são inseridos em um ambiente virtual, divididos em dois times de dois a três jogadores, em partidas de 1 minuto. A equipe que terminar a partida com mais pontos, ou seja, tenha feito mais gols, ganha.

Figura 1: *Fantasy Stars.*

Fonte: O Autor.

Durante o seu desenvolvimento, fez-se necessário a construção de um sistema de inteligência artificial que suportasse os requisitos do projeto.

A primeira abordagem utilizada foi a de escrever um componente responsável pelo gerenciamento do sistema e separar os estados em corotinas, checando as condições

a cada novo *frame*. As corotinas perduravam apenas enquanto a inteligência artificial permanecia no seu respectivo estado.

Neste ponto do desenvolvimento, nenhuma técnica ou metodologia de testes automatizados era utilizada, os testes eram realizados manualmente durante o desenvolvimento e ao fim de cada nova versão.

A abordagem escolhida resultou na rápida implementação dos estados mais básicos da inteligência artificial. Entretanto, no decorrer do desenvolvimento, o processo de depuração do código se tornava cada vez mais custoso, acumulando erros que ocupavam cada vez mais tempo nos ciclos de produção.

A fim de trazer mais estabilidade para o projeto, foi optado pela implementação de testes unitários automatizados utilizando a técnica de testes de caixa-branca, testando pequenos trechos de código e descobrindo erros de lógica que estivessem passando aos testes manuais.

Ao iniciar o processo de criação dos testes automatizados para as classes derivadas de componentes da *engine* Unity, percebeu-se que havia um alto grau de acoplamento entre as diversas funcionalidades utilizadas, como o tempo de execução da aplicação, manipulação de componentes, movimentação de objetos em cena e instanciamento de novos objetos.

A principal dificuldade encontrada estava relacionada a alta dependência que os componentes têm para com os *GameObjects*. Componentes Unity possuem uma forma própria de inicialização, os construtores não podem ser utilizados e novos componentes precisam ser adicionados e removidos por meio de métodos públicos, como *AddComponent* e *Destroy*. Portanto, dificultando a lógica dos componentes de serem testadas isoladamente por meio dos testes unitários.

Além disso, o acesso a outros componentes internos à engine, como *Transform* e *Rigidbody*, extremamente necessário no caso do sistema de inteligência artificial do jogo *Fantasy Stars*, se tornou dificultada por serem altamente dependentes do *loop* de jogo.

Diante das dificuldades em adaptar os componentes (*MonoBehaviour*) aos testes unitários, foram analisadas diferentes abordagens como *Mocking*, *PlayMode*, *EditMode* e disponibilizadas pela própria *engine*, em postagens em seu blog e documentação oficial, com a finalidade de selecionar a que melhor se adequasse aos requisitos do projeto.

2.1 Mocking

A técnica de *Mocking* consiste na criação de classes que simulam uma ou mais dependências do componente, que não podem ou são inviáveis de serem utilizadas no ambiente de teste. Neste caso, a lógica deve ser desacoplada do componente em outra classe e as ações que necessitam do uso de funcionalidades da *engine* devem ser acessadas através de uma *interface*, que será estendida por duas novas classes, uma utilizada apenas no ambiente de teste, outra no ambiente real da aplicação.

O principal problema desta técnica é a necessidade de criação de código adicional, o que pode acabar por aumentar o tempo de desenvolvimento, além de ser necessário mudar a forma como os componentes foram idealizados.

Entre os pontos positivos, é possível destacar o desacoplamento de código, padrão de projeto que a longo prazo facilita os processos de refatoração e adição de novas funcionalidades (NYSTROM, 2014).

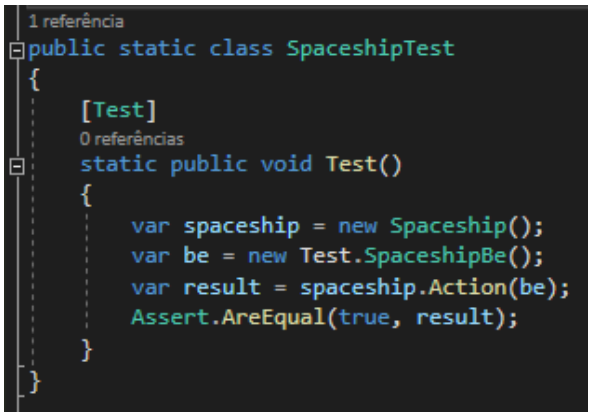
Na figura abaixo é possível visualizar um exemplo de *Mocking*, onde a classe *Spaceship* contém a lógica do sistema e acessa as funções da *engine* a partir da interface *ISpaceshipBe*, que pode ser tanto o componente *MonoBehaviour* quanto a classe *Mocking* utilizada apenas nos testes.

Figura 2: Exemplo de Mocking.

Fonte: O Autor.

Por fim, o código de teste deverá fornecer uma referência da classe *Mocking*, *SpaceshipBe*, resultando em um teste apenas da lógica da classe *SpaceShip*, como no exemplo abaixo.

Figura 3: Exemplo de teste de uma classe utilizando Mocking.



```
1 referência
public static class SpaceshipTest
{
    [Test]
    0 referências
    static public void Test()
    {
        var spaceship = new Spaceship();
        var be = new Test.SpaceshipBe();
        var result = spaceship.Action(be);
        Assert.AreEqual(true, result);
    }
}
```

Fonte: O Autor.

2.2 PlayMode

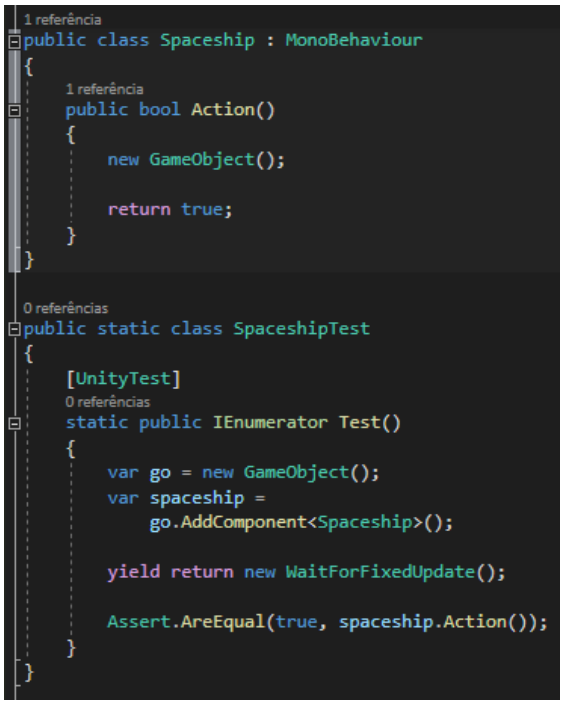
Outra opção de abordagem oferecida pela Unity é a utilização da função *PlayMode*. O *PlayMode* pode ser utilizado por meio da ferramenta *TaskRunner*, inicializando em *background* uma aplicação vazia a partir da qual os testes são executados. Desta forma, não é necessário fazer *Mocking* das dependências, os parâmetros estarão acessíveis aos componentes como se o jogo estivesse sendo executado.

A principal vantagem desta abordagem é não ser necessário alterar o código original, ou a forma na qual os componentes são construídos. Entretanto, como este modo necessita de uma aplicação vazia sendo executada, o tempo de inicialização e finalização dos testes se tornam ligeiramente mais longo.

Outro problema nesta abordagem, é a utilização de parâmetros externos que estão fora do controle do teste. Logo, se um deles falhar, o teste falhará como um todo, mesmo que a lógica testada não seja a mesma que causou a falha. O que leva ao questionamento se testes *PlayMode* ainda devem ser considerados testes unitários, já que acabam testando a integração entre diferentes módulos.

Segue, na figura abaixo, um exemplo de um teste em *PlayMode*. Neste caso, o teste deve ser escrito na forma de uma rotina que irá simular o *loop* de jogo a partir dos retornos *WaitForFixedUpdate*.

Figura 4: Exemplo de implementação *PlayMode*.



Fonte: O Autor.

2.3 EditMode

Também é possível utilizar o modo de edição, ou *EditMode*. A partir desta abordagem, é possível testar os componentes em modo de edição, assim como no *PlayMode*, sem alterar a forma como a lógica foi escrita.

A principal vantagem dessa abordagem é poder utilizar funções da engine como instanciamento de objetos e manipulação de componentes. Entretanto, como o próprio nome indica, o teste é executado em modo de edição, ou seja, alguns parâmetros acessíveis apenas em modo de jogo não estão disponíveis como, por exemplo, os eventos *Update* e *Start* que não são executados pela *engine*.

Ao contrário do *PlayMode*, o *EditMode* não ocasiona aumento no tempo de execução dos testes por não precisar iniciar a aplicação. Entretanto, compartilha outros problemas como a dependência de parâmetros externos, além de não ter o benefício de ter disponível todos os parâmetros do ambiente real.

2.4 Resultados dos testes

A fim de analisar qual seria a melhor abordagem para o caso do *Fantasy Stars*, foi construída uma pequena parte do sistema utilizando as diferentes técnicas apresentadas (*Mocking*, *PlayMode* e *EditMode*).

A opção que ofereceu maior praticidade de implementação, foi escrever os comportamentos normalmente através de componentes e utilizar o modo *PlayMode* para executá-los, justamente pela possibilidade de reaproveitar os códigos já existentes, realizando pequenas alterações e criando testes unitários para eles. Entretanto, o modo *PlayMode* precisa iniciar e finalizar uma instancia da aplicação a cada nova sessão de testes, o que reduz a agilidade do processo de desenvolvimento, principalmente quando se está avaliando um teste em específico.

Além disso, o fator acoplamento de código, já destacado no artigo, prejudica a confiabilidade dos testes, adicionando variáveis da aplicação que estão fora do controle do componente.

A opção de *Mocking* de dependências a partir do desacoplamento da lógica dos componentes Unity também trouxe bons resultados, gerando códigos contidos e fáceis de serem testados. Entretanto, o tempo gasto no desenvolvimento dos componentes foi maior, principalmente pela necessidade de criar interações adicionais entre as diferentes classes e *interfaces* que compõem o mesmo comportamento.

Por fim, optou-se por utilizar ambas as técnicas. A parte crítica do código foi externalizada, utilizando *Mocking* para as dependências e, para que o código integrado à *engine* não ficasse fora da cobertura dos testes, utilizou-se o modo *PlayMode*.

Ao fim dos ciclos de desenvolvimento, foram realizadas sessões de teste com membros da equipe e o público alvo a fim de analisar o produto a partir da interação com os jogadores, buscando erros de sistema e falhas de conceito.

As sessões de teste foram realizadas sob a supervisão do desenvolvedor, sendo o sistema de inteligência artificial analisado a partir da experiência do usuário utilizando os seguintes fatores:

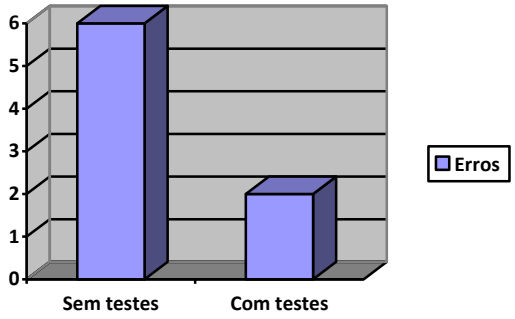
- Reação dos agentes condizentes aos *inputs* do jogador.
- Reação dos agentes em relação aos eventos do jogo (inicio de partida, gol, fim de partida).

- Reação dos agentes interagindo entre si no ambiente do jogo.

Comparando os resultados obtidos nas sessões de testes antes e depois da implementação do sistema de inteligência artificial, foi observado uma redução na quantidade de erros encontrados.

Durante os testes realizados na primeira versão (sem testes unitários), foram relatados 6 erros de mecânica, tais como: mudanças inesperadas de estado, condições não testadas e outros problemas relacionados ao funcionamento da máquina de estados. Durante os testes da versão já com testes implementados, apenas 2 problemas foram encontrados, neste caso, de condições que não haviam sido previstas nos testes unitários.

Figura 5: Quantidade de erros encontrados antes e depois da implementação dos testes.



Fonte: O Autor.

Os erros encontrados nas versões testadas antes da implementação dos testes unitários foram:

- Agente não voltou ao estado inicial após um gol.
- Agente não voltou ao estado inicial ao fim da partida.
- Enquanto no estado fazer gol, agentes nunca executam a ação chute.
- Agentes se movimentando em direções inesperadas.
- Agente ficou preso na parede.
- No modo em que existem dois jogadores por equipe, os agentes não receberam os papeis esperados e permaneceram parados.

Já após a implementação dos testes, foram identificados apenas os seguintes erros:

- Agente não voltou ao estado inicial após um gol.
- Agente não voltou ao estado inicial ao fim da partida.

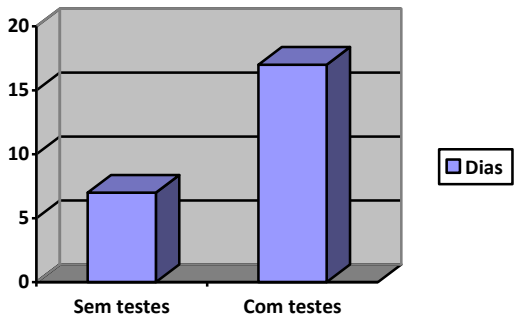
Entretanto, alguns destes erros já haviam sido corrigidos na versão anterior, logo, e apesar do sistema ter sido inteiramente reescrito, é possível que esta redução não tenha ocorrido em decorrência da inclusão de testes e sim do conhecimento prévio dos problemas.

Além disso, a equipe relatou maior facilidade no processo de depuração de código e desenvolvimento de novas funcionalidades que necessitassem de integração com o sistema de inteligência artificial.

A metodologia de testes foi utilizada primeiramente apenas no sistema de inteligência artificial. Dados os resultados positivos obtidos, foi replicado em outros sistemas que também possuem certo grau de complexidade, como o controle de partidas e histórico de movimentações.

Contudo, apesar dos aspectos positivos, é importante destacar que foi observado um aumento no tempo de produção das funcionalidades em comparação ao tempo levado sem a implementação dos testes unitários. O sistema de inteligência artificial havia sido implementado inicialmente (sem testes) em uma semana de desenvolvimento, enquanto a versão final, utilizando técnica de *Mocking* e *PlayMode*, incluindo os testes, levou por volta de duas semanas e meia, com a mesma equipe.

Figura 6: Comparação dos dias utilizados para a implementação do sistema de Inteligência artificial com e sem testes.



Fonte: O Autor.

Esta diferença pode ser atribuída também a inexperiência dos desenvolvedores com a metodologia e técnicas utilizadas, assim como o tempo gasto em pesquisa. Além disso, é possível que esta diferença seja reduzida após o time estar devidamente habituado aos processos, todavia, não é possível afirmar por meio deste estudo.

Conclusão

A partir do estudo de caso do jogo *Fantasy Stars*, foi possível observar que existem características no ambiente de desenvolvimento da *engine* Unity que dificultam o processo de implementação de testes unitários automatizados. Entretanto, como destacado no artigo, existem técnicas e ferramentas disponibilizadas pela própria *engine* que oferecem alternativas viáveis, contornando os principais problemas.

Por meio da utilização da técnica de *Mocking* e da ferramenta de execução de testes em modo de jogo (*PlayMode*), foi possível implementar de forma satisfatória os testes unitários, assim como obter cobertura de testes em todo o sistema alvo, sendo possível observar maior estabilidade de código e significativa redução na quantidade de erros encontrados nas novas versões do projeto.

O presente artigo utilizou como base a atual arquitetura de componentes utilizada pela *engine*, por ser a única solução disponibilizada a nível de produção. Entretanto, é importante destacar que há um novo sistema em desenvolvimento, conhecido pela sigla ECS, que quando lançado pode vir a solucionar as dificuldades relatadas.

Por fim, alguns pontos relevantes não puderam ser abordados neste artigo e ficam como sugestão para futuros trabalhos. O primeiro deles, em relação aos possíveis impactos em performance causados pela utilização da técnica de *Mocking*, sendo necessário um estudo mais aprofundado comparando com outras implementações. E por último, um comparativo entre o tempo necessário para o desenvolvimento de um jogo com e sem testes unitários, assim como a variação dessa diferença ao longo do projeto.

Referências

GIL, Antonio Carlos. **Como elaborar projetos de pesquisa**. São Paulo, v. 5, n. 61, 2002.

NETO, Arilo. **Introdução a teste de software**. Engenharia de Software Magazine, v. 1, 2007.

NYSTROM, Robert. **Game programming patterns.** Genever Benning, 2014.

PRESSMAN, Roger S. **Software engineering:** a practitioner's approach. Palgrave Macmillan, 2005.

WHITTAKER, James A. **What is software testing?** and why is it so hard?. IEEE software, v. 17, n. 1, 2000.Gill, Antonio Carlos, "Como elaborar projetos de pesquisa", vol. 5. São paulo, 2002.

ZHU, Hong; HALL, Patrick AV; MAY, John HR. **Software unit test coverage and adequacy.** Acm computing surveys (csur), v. 29, n. 4 , 1997.

Fontes adicionais

Unity Technologies, "Unity Test Runner".

Unity Technologies, "Creating and Using Scripts".

PROUSE, R.; POOLE, C.; SANDSTROM, T.; MADDOCK, C.; MUSSER, J.; BUNDGAARD, M. N. NUnit. Disponível em <https://nunit.org>. Acesso em: 26 nov 2019.

UNITY. Unity Technologies. Disponível em <https://unity.com/pt>. Acesso em: 26 nov 2019.