UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Henrique Andrade Mariz

Explorando Padrões de Projeto no Desenvolvimento de Jogos Digitais na *Unity* 3D

RECIFE

2023

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Henrique Andrade Mariz

Explorando Padrões de Projeto no Desenvolvimento de Jogos Digitais na *Unity* 3D

Monografia apresentada ao Centro de Informática (CIn) da Universidade Federal de Pernambuco (UFPE), como requisito parcial para conclusão do Curso de Ciência da Computação, orientada pelo professor Leopoldo Motta Teixeira.

RECIFE
2023

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Mariz, Henrique.
Explorando padrões de projeto no desenvolvimento de jogos digitais na Unity3D / Henrique Mariz. - Recife, 2023.
119 p. : il.

Orientador(a): Leopoldo Texeira
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2023.
Inclui referências, anexos.

1. Engenharia de Software. 2. Jogos Digitais. 3. Padrões de Projeto. 4. Boas práticas. 5. Unity. I. Texeira, Leopoldo. (Orientação). II. Título.

000 CDD (22.ed.)

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Henrique Andrade Mariz

Explorando Padrões de Projeto no Desenvolvimento de Jogos Digitais na *Unity* 3D

Monografia submetida ao corpo docente da Universidade Federal de Pernambuco, defendida e aprovada em 02 de outubro de 2023.

Banca Examinadora:

Orientador

Leopoldo Motta Teixeira

Doutor

Examinador(a)

Breno Miranda

Doutor(a)

AGRADECIMENTOS

Aos meus pais, Abílio e Ana Célia, agradeço por todo o suporte que foi essencial para a realização de tudo. Obrigado pelo exemplo que são, pelos sermões, incentivo, apoio e orações. Vocês foram essenciais ao longo de todo o percurso, não teria conquistado as mesmas coisas se não fosse por vocês.

Agradeço à minha irmã, Aline Mariz, que também sempre esteve presente, bem como a Mariana Siqueira por todo suporte emocional, paciência, compreensão, amor e carinho.
Em especial, gostaria de agradecer à minha avó, a qual não tive oportunidade de mostrar onde eu cheguei. Obrigado por sempre acreditar em mim e me guiar.

Gostaria de agradecer ao meu orientador, Prof. Dr. Leopoldo Motta Teixeira, pela disponibilidade e encorajamento que foram fundamentais para realizar e prosseguir com este estudo.

Agradeço, também, aos amigos que me ajudaram neste percurso da graduação, Guilherme Melo, Ivan Neves, Lucas Lin, Matheus Lima e Rodrigo Falcão, sinto que vocês tiveram um papel fundamental nesta jornada, tornando-a mais leve, seja nos estudos ou nos projetos de jogos dos quais realizamos juntos.

RESUMO

A *Unity* é uma das ferramentas mais populares no desenvolvimento de jogos digitais, pois oferece uma licença gratuita ou com ótimo custo-benefício a depender da proporção do jogo desenvolvido. Além de possuir suporte para múltiplas plataformas, apresenta também uma grande comunidade de utilizadores, permitindo o acesso a muita informação em fóruns que auxiliam no desenvolvimento, e também disponibilização de código e *assets*. Possui um ambiente de desenvolvimento com diversos recursos facilitadores para reduzir a complexidade de implementação, principalmente para amadores; entre outros benefícios. No entanto, para ter qualidade no desenvolvimento de jogos é importante estruturar o código de forma eficiente e organizada, visando ter um código manutenível, com boa performance, flexibilidade e escalabilidade. Dessa maneira, este trabalho visa explorar os padrões de projeto mais frequentemente recomendados para jogos, aplicados à *Unity*, bem como, as boas e más práticas neste ambiente de desenvolvimento, a fim de expor soluções comprovadas para os problemas recorrentes que permeiam esta área, disponibilizando implementações e orientações práticas para o uso em projetos reais.

Palavras-chave: padrões de projeto, boas práticas, más práticas, jogos digitais, *Unity*.

ABSTRACT

Unity is one of the most popular tools in the development of digital games, as it offers a free license or excellent cost-effectiveness depending on the scale of the game being developed. In addition to supporting multiple platforms, it also boasts a large user community, providing access to a wealth of information through forums that assist in development, as well as the sharing of code and assets. It features a development environment with various helpful resources to reduce implementation complexity, particularly for amateurs, among other benefits. However, to achieve quality in game development, it is important to structure the code efficiently and systematically, aiming for maintainable code with good performance, flexibility, and scalability. Thus, this work aims to explore the design patterns most frequently recommended for games when applied to Unity. It also delves into best practices and pitfalls in this development environment in order to present proven solutions to common issues in this field and provide practical implementations and guidance for use in real projects.

Keywords: design patterns, best practices, bad practices, digital games, Unity.

Sumário

Lista de Figuras

Tabela de Siglas

Sigla	Significado
3D	Tridimensional
CEO	Chief Executive Officer
CPU	Central Processing Unit
E-book	Eletronic Book
FSM	Finite State Machine
GoF	Gang of Four
IA	Inteligência Artificial
IDE	Integrated Development Environment
iOS	Iphone Operating System
KISS	Keep It Simple, Stupid!
LOD	Level of Detail
RLC	Revisão de Literatura Cinza
RQ	Review Questions
RR	Revisão Rápida
SR	Systematic Review
UI	User Interface
UML	Unified Modeling Language
WebGL	Web Graphics Library

1. Introdução

A priori, é importante mencionar que o desenvolvimento de jogos digitais é uma área que tem se tornado cada vez mais popular no Brasil. A pesquisa da Abragames, realizada em 2022, sobre a indústria brasileira de *games*, constatou que, entre 2018 e 2022, houve um aumento de cerca de 152% de empresas desenvolvedoras de jogos (Fortim, 2022), evidenciando um pouco da relevância atual da área.

Tendo isso em vista, vale também ressaltar uma das *game engines* mais populares de desenvolvimento de jogos digitais, a *Unity* 3D, visto que, nessa mesma pesquisa, a ferramenta é utilizada por cerca de 83% das empresas desenvolvedoras brasileiras (Fortim, 2022). Vários autores da área tentam explicar os diversos motivos da popularidade da *Unity* (Shah, 2017; Dealessandri, 2020; Schardon, 2023), a exemplo do artigo "*Unity Game Development Engine: A Technical Survey*" dos autores Hussain et al. (2020), o qual discorre acerca da definição da *Unity*, faz uma pesquisa técnica e lista as vantagens por ela oferecidas.

Em resumo os motivos da popularidade são: licença gratuita ou com ótimo custo-benefício a depender da proporção do jogo desenvolvido; suporte para múltiplas plataformas como Windows, Linux, Mac, Nintendo Switch, Android, iOS, webGL etc.; grande comunidade ativa, permitindo o acesso a muita informação em fóruns - a exemplo da *Unity Community*, *Stack Over Flow*, *Game Dev Stack Exchange* - como também disponibilização de código e assets pronto para uso na *Unity Asset Store*; Ambiente de desenvolvimento com diversos recursos facilitadores que diminuem a complexidade de implementação, principalmente para amadores; entre outras razões.

Como era de se esperar pela sua popularidade, a *game engine* foi utilizada no desenvolvimento de diversos jogos de sucesso pelo mundo. No blog The Gamer, o autor Jeff Drake, lista dezenove sucessos mundialmente jogados, como Pokémon Go, Cuphead, Ori and The Blind Forest, Hearthstone e outros (Drake, 2023). Além disso - segundo John Riccitiello, CEO da *Unity*, em 2018 no TechCrunch DisruptSF, evento anual tecnológico - a *Unity Engine* chegou a estar presente no desenvolvimento de mais da metade de todos os jogos de celular (Dillet, 2018).

A implementação de jogos consiste numa competência multidisciplinar que envolve diversos elementos interdependentes, como personagens, cenários, física, computação gráfica, inteligência artificial, entre outros elementos, por isso, se não tiver cuidado, o código pode ficar cada vez mais difícil de dar manutenção. O post de Artur Levchenko, no blog Visartech,

evidencia isso ao dizer que quanto mais complexa e diversificada for a mecânica do jogo, variedade de conteúdo e possíveis interações, mais difícil será executar corretamente e evitar o chamado “código espaguete”, ou seja, as conexões entre classes e módulos tornam-se extremamente estreitas, as interações das classes ficam fortemente entrelaçadas, assim, a alteração de qualquer um dos mecanismos existentes ou a adição de um novo comportamento torna-se incrivelmente difícil (Levchenko, 2023).

Padrões de projeto podem ajudar nesse contexto, visto que os autores Gamma et al. (1993) definem padrão de projeto como um problema que ocorre inúmeras vezes em determinado contexto, e descreve ainda a solução para esse problema, de modo que essa solução possa ser utilizada sistematicamente em distintas situações. Cada padrão tem uma característica diferente para ajudar em algum lugar onde se precisa de mais flexibilidade ou precisa encapsular uma abstração ou de se fazer um código menos casado (Gamma et al., 1993).

Portanto, este trabalho se propõe a explorar os padrões de projeto mais frequentemente recomendados pela literatura acadêmica e pela literatura cinza para jogos desenvolvidos em Unity, assim como, as boas e as más práticas neste ambiente de desenvolvimento, visto que é importante entender os contextos problemáticos que podem surgir durante o desenvolvimento de jogos, para se utilizar de soluções adequadas para tais problemas

Para alcançar esse propósito, foi conduzida uma revisão da literatura de forma sistemática, utilizando de Revisão Rápida (RR) e de Revisão de Literatura Cinza (GLR) - devido à natureza prática e concreta da pesquisa - visando oferecer implementações e orientações práticas para projetos na Unity no contexto de jogos, a fim de se obter um código mais flexível, escalável, manutenível e performático.

1.1. Objetivos

A *Unity* 3D é uma das ferramentas mais populares no desenvolvimento de jogos digitais, sua ampla comunidade auxilia bastante o programador, visto que há bastante conteúdo pronto para uso, como *assets*, *plugins*, bibliotecas, vídeos, até mesmo muitas dúvidas respondidas em fóruns. Tendo isso em vista, muitos dos problemas comuns já foram discutidos, resolvidos e disponibilizados na internet.

Portanto, o objetivo deste trabalho é realizar uma pesquisa exploratória sobre o uso de padrões de projeto no desenvolvimento de jogos na *Unity* 3D, assim como demonstrar sua aplicação prática em contexto real, visando apresentar aos desenvolvedores soluções comprovadas para problemas conhecidos e recorrentes na área, como também a identificação destes cenários.

2. Conceitos Básicos

No mundo da engenharia de *software*, a construção de sistemas eficientes, flexíveis e de fácil manutenção é uma busca constante. Nesse contexto, padrões de projeto visam prover soluções para problemas recorrentes de forma que estejam alinhadas com estes requisitos. Deste mesmo modo, este trabalho visa explorar padrões de projeto, bem como, boas e más práticas de código no contexto de desenvolvimento de jogos na *Unity*. Contudo, antes de adentrar neste tópico, é essencial trazer a definição de alguns conceitos prévios, com o intuito de trazer uma melhor compreensão ao analisar os benefícios de alguns dos padrões de projeto que serão apresentados.

2.1. Favorecer a composição em vez de herança

Segundo Gamma et al. (1994), em sistemas orientados a objetos, as duas técnicas mais frequentemente utilizadas para reutilizar funcionalidades são a de herança de classes e a composição de objetos. No entanto, é importante destacar que ambas as técnicas possuem vantagens e desvantagens, não havendo uma regra rígida que determine a superioridade de uma sobre a outra. Isto dependerá do contexto da situação.

De maneira geral, de acordo com Barbosa, Rêgo e Medeiros (2015), a abordagem via composição torna as classes encapsuladas e com potenciais custos de manutenção menores. Por outro lado, requer mais expertise para usar, demandando mais tempo de desenvolvimento da equipe. Já a herança não contempla essas vantagens, mas elimina duplicação de código e diminui o tempo de desenvolvimento da equipe, visto que demanda de menos proficiência para ser utilizada (Barbosa, Rêgo e Medeiros, 2015)

Contudo, o princípio abordado neste tópico, o de favorecer a composição em vez de herança, de acordo com Gamma et al, ajuda a manter as classes encapsuladas e focadas em uma única tarefa. Assim, estas classes tendem a permanecer pequenas, bem como, tendem a não se tornarem grandes monstros incontroláveis (Gamma et al., 1994).

2.2. Alta coesão e baixo acoplamento

Coesão e acoplamento são dois conceitos básicos na engenharia de *software*, os quais estão relacionados, visto que normalmente quando se tem um alto acoplamento se tem uma baixa coesão e vice e versa.

Richards e Ford (2020), referem-se a coesão como uma medida que representa o quão relacionadas estão as partes de um módulo entre si, de modo que, em um cenário ideal, um

módulo é considerado coeso quando todas as suas partes devem ser mantidas juntas, pois ao separá-las em pedaços menores exigiria acoplamento das partes por meio de chamadas entre eles.

De forma análoga, quão menos relacionadas estiverem as partes, menor coesão tem o módulo, isto quer dizer que, num cenário de mínima coesão, as partes poderiam estar separadas sem haver acoplamento entre elas, isto é, sem haver chamadas entre si, visto que não existe relação de uma com a outra.

Richards e Ford (2020) afirmam que coesão é uma métrica menos precisa que acoplamento, visto que para definir acoplamento basta ver as conexões de entrada e saída de um artefato de código (componente, classe, função e assim por diante) com outro artefato de código. De maneira mais prática, pode-se definir como a medida de conexão entre módulos diferentes de código, mostrando o quão dependentes ou interligadas estão os diferentes módulos.

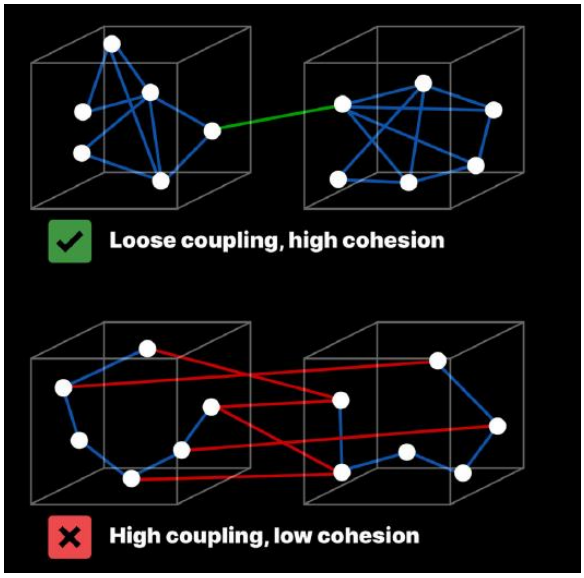
Portanto, é possível concluir que coesão se refere a conexão interna de um módulo, já acoplamento se refere a conexão externa de um módulo. Dessa forma, ao projetar software geralmente tenta-se maximizar a coesão e minimizar o acoplamento, visto que isto tornaria os módulos mais independentes de modo que mudanças em um determinado módulo não afetaria ou pouco impactaria outro módulo. Ou seja, é a partir daí que vem a expressão alta coesão, baixo acoplamento.

De maneira similar, em um projeto evita-se ter acoplamento forte e baixa coesão, visto que isto tornaria os módulos mais dependentes um dos outros, isto é, quando houver mudanças em um determinado módulo afetaria outros módulos de forma direta, tornando o código mais difícil de dar manutenção, visto que haveria mais locais para ajustar do que se o módulo fosse mais independente. Pode-se visualizar na figura 1, uma representação do cenário ideal, o qual se refere a alta coesão e baixo acoplamento

Lin (2021) reforça isso ao afirmar que, idealmente, deve-se buscar, de preferência, minimizar as dependências entre classes, de maneira que cada classe deve ser capaz de operar de forma harmoniosa com suas partes internas, em vez de depender fortemente de conexões externas.

Pode-se visualizar na figura 1, uma representação dos cenários de alta coesão e baixo acoplamento, bem como, o cenário de baixa coesão e alto acoplamento, com relação às conexões internas a um módulo (representado por uma caixa) e as conexões externas.

Figura 1 – Ilustração que representa a conexão entre módulos de forma interna e externa, ou seja, coesão e acoplamento.



Fonte: Lin (2021).

2.3. SOLID

Em harmonia com a definição estabelecida por Martin (2017), os princípios *SOLID* consistem em um arranjo das primeiras letras de cinco princípios que nos dizem como organizar funções e estruturas de dados em classes e como essas classes devem ser interconectadas de forma que sejam tolerantes a mudanças, fáceis de entender e bem como a base para sistemas de *software*, sendo os princípios: o princípio da responsabilidade única, princípio aberto-fechado, princípio da substituição de Liskov, princípio da segregação de interface e princípio da inversão de dependência.

De maneira similar, Lin (2021) define os princípios *SOLID* como diretrizes para ajudar a escrever um código mais limpo para que seja mais eficiente de manter e estender. Entretanto o livro ressalta que em alguns casos, ao aderir ao *SOLID*, de início, pode resultar em um trabalho adicional, o que talvez demande refatoração de algumas de suas funcionalidades em abstrações ou interfaces, no entanto, muitas vezes há uma recompensa a longo prazo (Lin, 2021).

i. Princípio da responsabilidade única

Conforme Martin (2017) explica, historicamente este princípio é descrito como: “um módulo deve ter um, e apenas um, motivo para mudar”. Em outras palavras, a classe deve ser responsável por um único ator, de modo que se a classe tiver com a responsabilidade de diferentes atores, o princípio está sendo violado. Ele ainda reforça que a chave para este

princípio é a palavra coesão, a qual é a força que une o código responsável por um único ator (Martin, 2017).

Este princípio está bastante ligado com a coesão, como afirmado por Martin (2017). Quando se tem uma alta coesão, o princípio da responsabilidade única está sendo respeitado, visto que as partes internas de um código não têm motivo para estarem separadas, uma vez que esta não assume responsabilidade de atores diferentes. De maneira similar, caso se tenha baixa coesão, significa que existem partes dos módulos as quais não se relacionam entre si, indicando que possivelmente a classe está com responsabilidade de diferentes atores.

ii. Princípio aberto-fechado

De acordo com Martin (2017), o princípio aberto-fechado diz que um artefato de *software* deve estar aberto para extensão, mas fechado para modificação. Ou seja, o comportamento de um artefato de *software* deve ser extensível, sem a necessidade de modificar esse artefato (Martin, 2017). Em outras palavras, isso significa que ao adicionar novos recursos ou funcionalidades a um sistema existente, deve-se realizar estendendo comportamento, sem gerar alterações no código original desse sistema.

Na opinião de Martin (2017), o princípio aberto-fechado é uma das forças motrizes da arquitetura de sistemas, visto que este tem como objetivo tornar o sistema fácil de estender sem implicar em um grande impacto de mudança. Segundo ele, este objetivo é alcançado particionando o sistema em componentes e organizando esses componentes em uma hierarquia de dependências que protege componentes de nível superior a partir de alterações em componentes de nível inferior (Martin, 2017).

iii. Princípio da substituição de Liskov

A herança na programação orientada a objetos permite adicionar funcionalidade por meio de subclasses, no entanto, isso pode levar a comportamentos inesperados se a herança for mal utilizada. Um exemplo disto é quando objeto derivado sobrescreve um método da classe base e não chama ou mantém o comportamento de sua classe base, podendo levar a erros ou comportamentos inesperados, visto que é esperado que um objeto derivado assuma o papel de uma classe base com comportamentos extras.

O princípio da substituição de Liskov tenta evitar isso e tornar subclasses mais robustas e flexíveis ao postular que classes derivadas devem ser substituíveis por sua classe base, isto é, uma classe derivada deve manter os comportamentos de sua classe base (Lin, 2021).

É válido ressaltar que uma das formas para resolver este problema é utilizando composição no lugar de herança, pois em vez de tentar transmitir funcionalidade por meio de herança, pode-se criar uma interface ou uma classe separada para encapsular um comportamento específico. Basta, em seguida, construir uma composição de funcionalidades diferentes misturando e combinando (Lin, 2021).

iv. Princípio da segregação de interface

Lin (2021) estabelece este princípio como a ideia de que uma classe não deve ser forçada a depender de métodos que não utiliza. Sendo assim, as interfaces tendem a serem menores, compactadas e com máxima flexibilidade (Lin, 2021).

Dito isto, este princípio busca dividir interfaces extensas em interfaces menores, a fim de evitar que classes sejam forçadas a implementar todo o conteúdo de uma interface quando, na verdade, apenas desejam implementar uma parte dela. Ou seja, pode-se concluir que este princípio tem relação com o conceito de coesão, porém aplicado a interfaces, visto que se uma classe está sendo obrigada a implementar algo, significa que ela tem interesse apenas em uma parte do módulo, o que leva a questionar se as partes são separáveis ou se fazem sentido estarem juntas, isto é, se estão coesas.

v. Princípio da inversão de dependência

O princípio da inversão de dependência diz que módulos de alto nível não devem depender diretamente de módulos de baixo nível, ambos devem depender de abstrações. Além disso, abstrações não devem depender de detalhes e detalhes devem depender de abstrações (Lin, 2021).

De maneira similar, Martin (2017) afirma que o princípio da inversão de dependência diz que os sistemas mais flexíveis são aqueles em que as dependências do código-fonte se referem apenas a abstrações, não a concreções. Isto significa que classes devem depender de abstrações e não de implementações, uma vez que se uma classe sabe muito sobre como outra classe funciona, modificar a primeira classe pode prejudicar a segunda ou vice-versa, isto em um alto grau de acoplamento pode causar um efeito bola neve, no qual um erro pode se transformar em vários. Em outras palavras, pode-se dizer que ao aplicar este princípio está enfraquecendo o acoplamento, visto que este princípio visa diminuir as consequências dos efeitos causados por conexões a outros módulos.

2.4. Princípio KISS

O princípio *KISS* diz respeito a uma expressão em inglês: "*Keep it simples, stupid!*", traduzindo para o português seria algo como "Matenha simples, estúpido!". Em outras palavras, este princípio prega que a simplicidade é a chave em um projeto e complexidade desnecessária deve ser evitada. De forma similar, Lin (2021) define este princípio como a ideia de que apenas se deve adicionar complexidade, caso seja necessário, caso contrário deve-se manter simples.

Um exemplo prático deste princípio ocorre quando se incorpora um padrão de projeto em um *software*. De início, isso pode resultar na inclusão de estruturas adicionais para dar manutenção e em uma configuração inicial mais elaborada. Isso, por sua vez, pode tornar o código mais complexo e difícil de compreender. No entanto, é importante analisar a situação para determinar se essa complexidade é justificável em relação aos benefícios que o padrão traz. Isto é, se os benefícios proporcionados pelo padrão não compensarem o aumento de complexidade para uma situação que não demanda uma solução elaborada, é mais sensato optar por manter a simplicidade.

3. Ambiente *Unity* de desenvolvimento

De antemão, é importante esclarecer que a *Unity* se trata de um motor de jogo (*game engine*) utilizado no desenvolvimento de jogos 2D e 3D, a qual é bastante popular no mundo inteiro. Inclusive, segundo John Riccitiello - CEO da *Unity* -, em 2018, no *TechCrunch DisruptSF*, evento anual tecnológico, a *Unity Engine* chegou a estar presente no desenvolvimento de mais da metade de todos os jogos de celular.

Conforme pesquisa realizada pela Abragames em 2022 sobre a indústria brasileira de games, foi constatado que esta é a ferramenta de trabalho utilizada por cerca de 83% das empresas do país que atuam neste ramo, de modo que resta evidente o seu considerável crescimento dentro do cenário brasileiro de games (Fortim, 2022).

Em resumo, os motivos de sua popularidade, de acordo com diversos autores que tentam explicar seu sucesso, são a licença gratuita ou com ótimo custo-benefício, a depender da proporção do jogo desenvolvidos; suporte para múltiplas plataformas como *Windows, Linux, Mac, Nintendo Switch, Android, iOS, webGL* etc.; grande comunidade ativa, permitindo o acesso a diversas informações em fóruns, a exemplo da *Unity Community*, *Stack Over Flow*, *Game Dev Stack Exchange*, como também disponibilização de código e *assets* prontos para uso na *Unity Asset Store*; ambiente de desenvolvimento com diversos recursos facilitadores que

diminuem a complexidade de implementação, principalmente para amadores; entre outras razões (Reddit, 2018; Ellis, 2019; Krogh-jacobsen, 2022; Unity, 2023)

3.1. Interface da *Unity*

O editor da *Unity* (figura 2) é uma das peças fundamentais do ambiente de desenvolvimento, visto que permite a visualização de vários recursos de forma facilitada e intuitiva para o usuário. Inclusive, as janelas do editor podem ser customizadas à preferência do usuário, bem como é possível criar extensões destas para implementar uma visualização própria de algo que não seja possível por padrão.

Existem diversos recursos no editor da *Unity*, entretanto, com o intuito de apenas trazer uma noção principal do uso da ferramenta, serão explicadas as principais janelas do editor, as quais são mais utilizadas num fluxo de desenvolvimento. São elas: a janela *Project*, *Hierarchy*, *Inspector*, *Scene* e *Game*, conforme destacadas na figura 2.

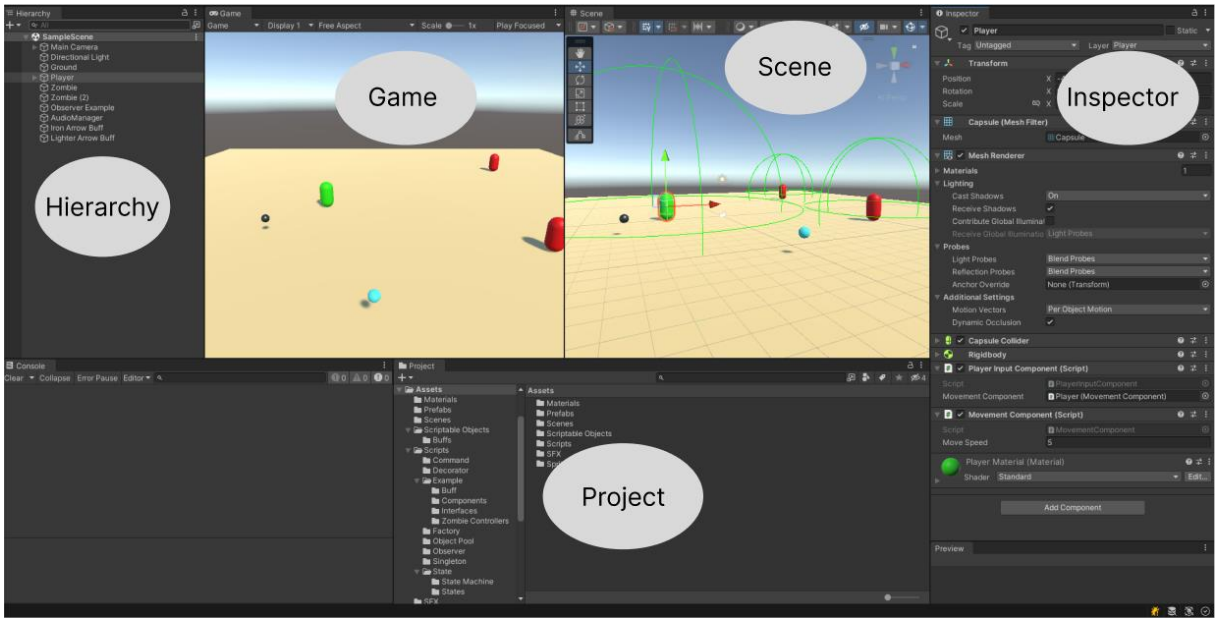
Inicialmente, a janela *Project* é a interface responsável pelo gerenciamento de arquivos na *Unity*, como *assets*, imagens, modelos 3D, *scripts*, áudios, entre outros arquivos que estão disponíveis para uso no projeto.

A janela *Hierarchy*, por sua vez, é a interface responsável por exibir os objetos que estão na cena atual do jogo, a qual permite que o desenvolvedor possa definir uma hierarquia entre os objetos de jogo numa estrutura de árvore, assim como pastas.

Na sequência, a janela *Inspector* é a interface responsável por exibir os detalhes de um elemento selecionado. Ao clicar num objeto que está na hierarquia ou em um arquivo que está na pasta do projeto, ela muda a visualização correspondendo ao objeto selecionado. É extremamente utilizada para configurar os detalhes de um objeto, a exemplo de adicionar componentes, adicionar referências, atribuir valores a propriedades, entre outros.

Por sua vez, a janela *Scene* permite a navegação, visualização e edição da cena numa perspectiva dentro do mundo virtual do jogo, como se o observador da tela estivesse nos "bastidores" do jogo. A janela *Game*, por fim, simula a renderização final do jogo, a qual vai utilizar da câmera principal, configurada na hierarquia, para permitir a visualização da imagem advinda da câmera.

Figura 2 – Visualização do editor da *Unity*.



Fonte: autoral.

3.2. Fluxo de trabalho na Unity

A *Unity* utiliza do conceito de cenas para facilitar a criação de jogos no editor, e tais cenas servem para compor o mundo virtual em tempo de edição. Um exemplo comum de uso destas é a construção de uma fase de um jogo, visto que permite adicionar objetos (*GameObjects*) no mundo, ajustar a posição, ter o feedback em tempo real de edição, além de permitir visualização de elementos para depuração etc.

Os objetos variados podem ter funcionalidades variadas e dependerão de seus componentes atrelados para definir o seu respectivo comportamento, como componente de luz, câmera, colisão, *sprite*, *input*, áudio, entre outros. Diversos componentes são disponibilizados pela *Unity* para facilitar o desenvolvimento, entretanto, para definir comportamentos específicos do seu jogo, provavelmente será necessário criar scripts próprios, isto é, componentes customizados.

Diante disso, cumpre mencionar que existem maneiras diversificadas de criar um *script*, sendo uma delas clicar com o botão direito do mouse e selecionar a opção "*Create*" e, em seguida, clicar em "C# *script*" - linguagem de programação padrão na *Unity*. Assim, um arquivo de script será criado com uma classe que deriva de *MonoBehaviour* - classe base para componentes na *Unity* (Figura 3). Na sequência, implementado o *script*, basta associá-lo a um *GameObject* (Figura 4) para adicionar o comportamento pretendido.

Adicionado o componente ao objeto, este estará vinculado a uma cena e será automaticamente instanciado ao executar a cena. Contudo, em tempo de execução, tais componentes só podem ser instanciados via *AddComponent()* ou *Instantiate()*, ou seja, não podem ser instanciados via palavra-chave *new*. Ante o exposto, as dependências de uma componente não podem ser transmitidas via construtor e, geralmente, fazem-no via *Inspector* ao utilizar variáveis públicas ou privadas com o atributo [*SerializeField*], permitindo que o editor crie um campo para associação de valores ou referências de objetos.

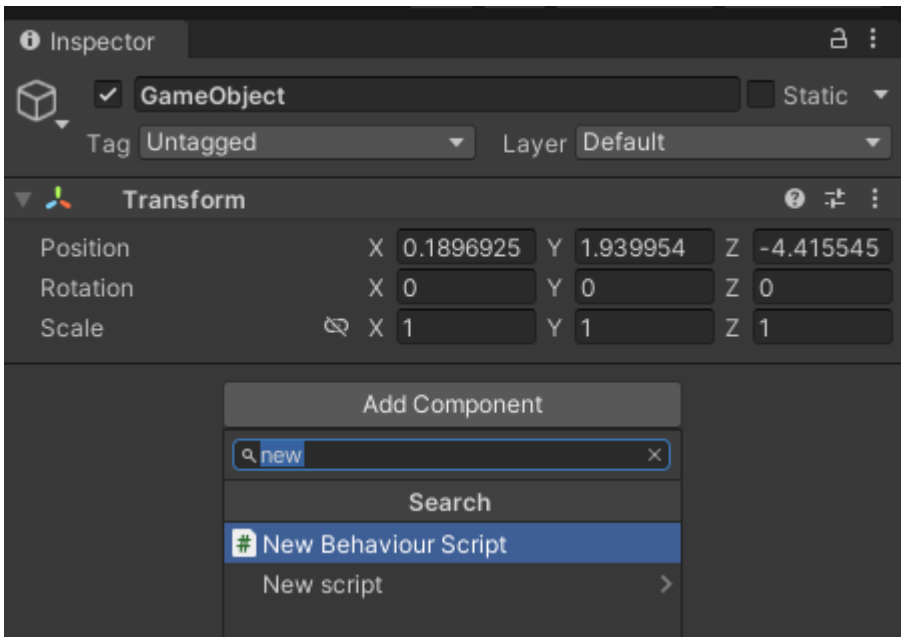
É importante, ainda, ressaltar que os *MonoBehaviours* fazem parte do ciclo de vida da *Unity*, de modo que esta classe oferece várias funções ou mensagens, que facilitam o desenvolvimento do código. As principais são: *Awake()*, *OnEnable()*, *Start()*, *Update()*, *FixedUpdate()*, *OnDisable()*.

Figura 3 – Script vazio recém-criado na *Unity*.

```
using UnityEngine;
using System.Collections;

public class NewBehaviourScript : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

Fonte: Unity Technologies (2023).

Figura 4 – Visualização do inspector ao adicionar um script em um *GameObject*.



Fonte: autoral.

O *Awake()* é um método executado apenas uma vez no ciclo de vida de um componente, no momento em que este é carregado. Sua função é de executar lógica ao carregar o objeto, como atualizar referências, inicializar classes etc., entretanto, neste momento é possível que outros componentes ainda não estejam completamente inicializados. De maneira parecida funciona o método *Start()*, contudo este é executado após o método *OnEnable()*, o qual será esclarecido a seguir.

O método *OnEnable()* é um método que é executado assim que o componente ou o objeto associado é ativado na cena. De forma análoga, funciona o método *OnDisable()*, cuja execução acontece quando o componente ou objeto vinculado é desabilitado.

O método *Update* é um método disparado a todo frame da aplicação, isto é, o *loop* de jogo. De forma similar funciona o método *FixedUpdate()*, que, por seu turno, é executado em tempo fixo. Este método é utilizado normalmente para executar operações exaustivas, as quais não devem ser desempenhadas a todo frame, porém precisam ser executadas com constância; um exemplo disto são rotinas de cálculos físicos, como movimentação e colisão.

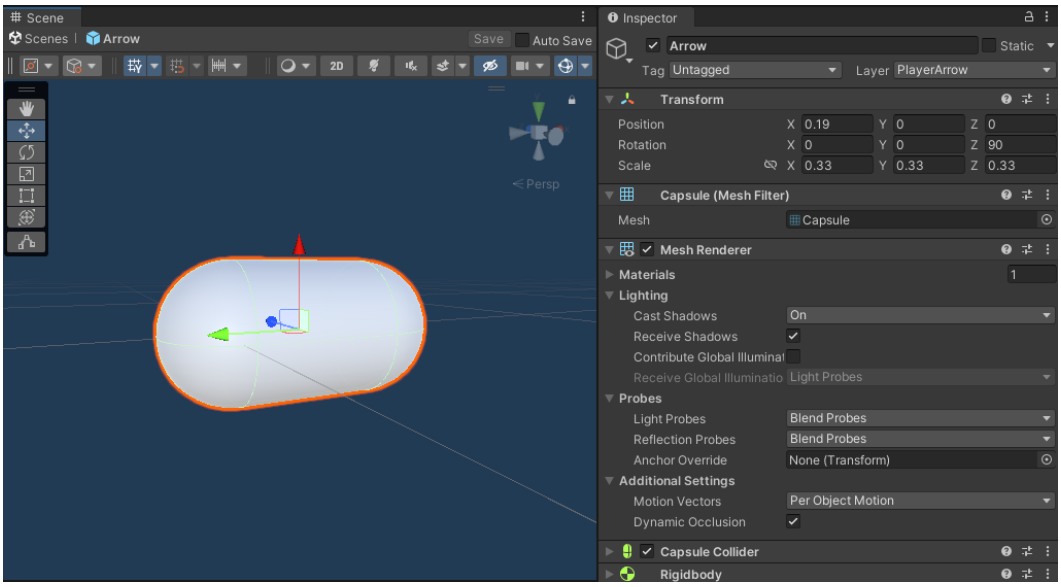
3.3. Prefabs

Há diversos recursos na *Unity* com a finalidade de facilitar o desenvolvimento, sendo o sistema de *prefabs* um deles, o qual permite criar, configurar e armazenar um *GameObject* completo com todos os seus componentes, valores de propriedade e *GameObjects* filhos,

funcionando como um *asset* reutilizável. Em outras palavras, o *prefab* funciona como uma forma de *template*, ou seja, como um modelo ou receita de qual maneira um objeto deve ser instanciado.

Quaisquer edições feitas em um *prefab asset* são refletidas automaticamente nas suas instâncias, permitindo que se façam alterações amplas de modo facilitado em todo o projeto, sem ser necessário repetir a mesma edição em cada cópia do *asset*. Estas edições são feitas numa cena isolada, de modo que apenas o *prefab* fica em evidência, como pode ser visto na figura 5.

Figura 5 – Visualização de prefabs na *Unity*.



Fonte: autoral.

4. Metodologia

Baseado na motivação discutida anteriormente, este estudo se propõe a explorar e elucidar padrões de projeto, como também boas e más práticas, no ambiente Unity de desenvolvimento, a fim de oferecer estratégias conhecidas para resolver problemas comuns no contexto de desenvolvimento de jogos, juntamente com implementações e orientações práticas de como aplicar estas propostas no motor de jogos Unity.

Deste modo, foi realizada uma revisão de literatura de forma sistemática, a qual foi baseada na combinação de uma Revisão Rápida (RR) e de uma Revisão da Literatura Cinza (RLC) a fim de unir e sintetizar o conhecimento científico, com o conhecimento prático de

desenvolvedores, difundido em diversas fontes, seja acadêmica ou experiencial do campo de atuação.

Como resultado, este trabalho traz uma compilação de padrões de projeto, bem como, boas e más práticas mais mencionadas no contexto da Unity que foram contempladas na pesquisa. Os detalhes da metodologia aplicada são demonstrados nas próximas seções.

4.1. Revisão Rápida (RR) e Revisão de Literatura Cinza (RLC)

Este estudo utiliza de RR, as quais são um estudo secundário baseado em adaptações no processo sistemático de revisão (SR) com o objetivo de dar suporte em decisões profissionais baseadas num contexto mais prático, conforme descrito por Cartaxo et al. (2020). Estas 49adaptações são feitas para facilmente transferir conhecimento científico para o conhecimento prático, reduzindo o custo e tempo, ao omitir ou simplificar algumas etapas do processo de SR (Cartaxo et al., 2020). Neste contexto, RRs se encaixaram bem devido ao contexto prático deste trabalho, visto que procura trazer implementações e orientações práticas na Unity Engine.

Desta mesma forma, devido à ampla comunidade da Unity, bem como, devido à natureza do campo de pesquisa, existe um vasto conhecimento informal que está em blogs, fóruns de desenvolvedores, documentações oficiais e tutoriais online, por isto, este trabalho também utiliza da literatura cinza para complementar possíveis lacunas não contempladas pela perspectiva acadêmica. Assim sendo, este estudo utiliza de RLC, consoante com Garousi et al. (2018), o qual define RLC como um tipo de revisão sistemática que permite a inclusão de materiais da literatura como recurso primário, como white papers, blogs, documentação e outras fontes não científicas.

4.2. Perguntas da pesquisa

Com base nos objetivos desta pesquisa, em consonância com o que foi inicialmente introduzido, foram definidas algumas perguntas (RQ), para nortear ambas as revisões (RLC e RR) de acordo com as diretrizes definidas por Cartaxo et al. (2020) para a RR e por Garousi et al. (2018) para RLC, embora sejam similares. São elas: (RQ1) “Quais são os padrões de projeto utilizados por desenvolvedores de jogos no ambiente de desenvolvimento *Unity*?”; (RQ2) “Quais boas práticas são sugeridas no ambiente de desenvolvimento *Unity*?”; (RQ3) “Quais são os anti-patterns no desenvolvimento de jogos na *Unity*?”.

Dito isto, a RQ1 almeja coletar os padrões de projeto aplicados no âmbito de desenvolvimento da *Unity*. De maneira similar, a RQ2 busca coletar quais são as orientações

sugeridas ao trabalhar no ambiente *Unity* de desenvolvimento, tal como, a RQ3 se preocupa em entender quais são as más práticas que permeiam este mesmo contexto.

4.3. Estratégia de busca

Para conduzir a busca na RR, foi utilizado o Google Acadêmico como ferramenta de busca visando englobar uma ampla variedade de artigos de pesquisa, uma vez que essa plataforma indexa documentos das principais bibliotecas digitais, limitando-o, conforme recomendado por Cartaxo et al. (2020).

Deste modo, diversos testes foram realizados com diferentes palavras-chaves até que fosse possível chegar em um resultado satisfatório via experimentação, de forma que fossem obtidos resultados relevantes para as perguntas anteriormente definidas, sendo elas: (“*game programming pattern*” *OR* “*design pattern*” *OR* “*game pattern*” *OR* “*best practices*” *OR* “*game architecture*” *OR* “*anti-pattern*” *OR smell OR* “*bad practices*”) *AND* (*Unity*) *AND* (“*game development*”).

A pesquisa da RLC foi realizada de maneira similar, com a alteração da ferramenta de pesquisa devido ao contexto da literatura cinza, sendo alterada para o uso do Google. Sendo assim, após diversas experimentações, o conjunto de palavras sofreu apenas uma pequena alteração em comparação as palavras definidas na RR, sendo estas: ("*game programming pattern*" *OR* "*design pattern*" *OR* "*game pattern*" *OR* "*best practices*" *OR* "*game architecture*" *OR* "anti-*pattern*" *OR smell OR* "*bad practices*") *AND* (*Unity*).

4.4. Procedimento de seleção

Dando sequência à metodologia aplicada, foi definido um conjunto de critérios de seleção para filtrar os resultados de modo que melhor atendesse às necessidades da pesquisa em termos de coerência, qualidade e disponibilidade, tanto para RR quanto para RLC. Sendo eles: (1) O conteúdo da fonte deve ter conexão com o contexto de engenharia de software e estar diretamente associado ao desenvolvimento na *Unity*; (2) O conteúdo da fonte deve responder pelo menos uma das perguntas de forma significativa; e (3) A fonte deve ser de acesso gratuito, sem custos associados.

A definição do item (1) tem como intuito restringir que os resultados não se distanciem para outros contextos, visto que esse trabalho tem como objetivo obter respostas práticas ao contexto de desenvolvimento na *Unity*. De modo similar, o item (2) procura rejeitar menções superficiais, as quais pelo menos contenham explicações plausíveis e coerentes mesmo que sejam breves. Por fim, o item (3) está associado à questão da disponibilidade de acesso devido

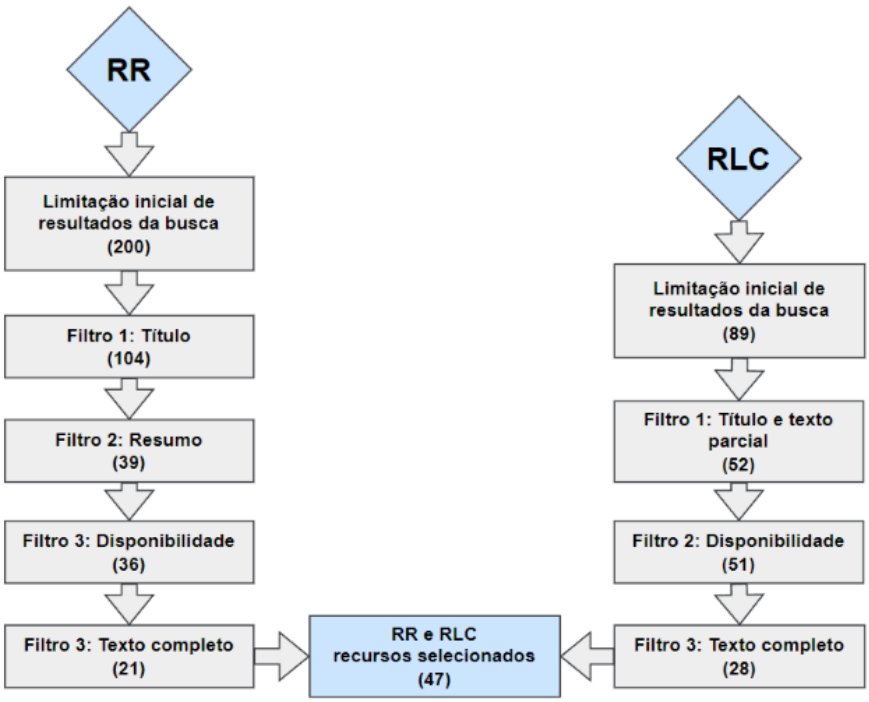
a alguns resultados apontarem para livros pagos, bem como, nem todo material foi possível ter acesso via pessoa física ou instituição acadêmica.

Durante a pesquisa, o Google acadêmico retornou milhares de resultados (cerca de 3.560), desta forma, foi necessário definir um limite de esforço para parar a pesquisa, assim como recomendado por Garousi et al. (2018). Para a RR definiu-se que apenas os primeiros 200 resultados mais relevantes pelo Google seriam considerados. De maneira semelhante, foi necessário fazer o mesmo procedimento para a RLC, visto que esta retornava cerca de 28 milhões de resultados em sua busca no Google, deste modo, foi definido que seriam considerados apenas os 100 primeiros resultados. No entanto, não foi necessário continuar até o final, pois ao atingir o resultado de número 89, o Google emitiu um aviso indicando que vários resultados haviam sido omitidos para destacar apenas os mais relevantes. Portanto, optou-se por encerrar a pesquisa neste ponto.

Na RR, dentre os primeiros duzentos resultados, foram excluídos aqueles cujos títulos claramente não se relacionavam com o escopo do estudo, deixando-nos com um total de 104 resultados. Em seguida, foram avaliados com base nos resumos, resultando em apenas 39 artigos restantes. Destes, foram descartados aqueles que não atendiam ao critério de disponibilidade de acesso (3), resultando em um total de 36 resultados. Por fim, os textos completos foram analisados e filtrados de acordo com todos os critérios mencionados, resultando em um total de 21 artigos selecionados.

De maneira análoga, a RLC passou pelo mesmo processo de filtragem. Inicialmente, os resultados foram avaliados com base nos títulos e no conteúdo parcialmente disponibilizado pela ferramenta de busca, resultando em 52 fontes. Apenas uma fonte foi excluída devido a problemas de acesso. Em seguida, os textos foram analisados por completo, totalizando em 28 fontes aprovadas. Este processo foi ilustrado na figura 6, a qual podemos ver as etapas de forma resumida, totalizando no final 47 recursos ao juntar a pesquisa RR com a RLC e remover as duplicatas.

Figura 6 – Diagrama representando o processo de seleção de recursos (RR e GLR).



Fonte: autoral.

5. Resultados e discussões

Conforme evidenciado no tópico anterior, após a conclusão do processo de seleção, permaneceram 47 recursos a serem incorporados ao material de pesquisa. Estes recursos abrangem tanto padrões de projeto quanto boas e más práticas no contexto do desenvolvimento *Unity* para jogos. Consequentemente, os padrões de projeto foram categorizados e consolidados em uma discussão. No entanto, com a intenção de trazer os padrões mais relevantes, bem como, traçar um limite de esforço cabível neste trabalho, foi definido trazer os padrões de projeto que foram mencionados pelo menos em quatro destes recursos finalistas. Isto resultou na identificação de nove padrões, os quais estão detalhados a seguir.

De maneira semelhante, as boas e más práticas também foram compiladas em um tópico de discussão. O procedimento abordado neste tópico está documentado em alguns arquivos armazenados em uma pasta na nuvem[1], que inclui as definições-chave utilizadas para conduzir a pesquisa.

[1] Disponível em: < http://bit.ly/3POB0vK>.

5.1. Padrões de Projeto na *Unity*

Conforme introduzido inicialmente, na engenharia de software, os padrões de projeto oferecem soluções abrangentes para problemas que surgem repetidamente em determinados contextos, possibilitando sua reutilização em diversas situações. Cada padrão possui características distintas que podem ser vantajosas quando se busca maior flexibilidade, encapsulamento, abstração, a redução do acoplamento, performance, entre outros aspectos.

Este trabalho aborda os padrões de projeto mais relevantes identificados durante a pesquisa, conforme detalhado na seção de seleção de procedimentos. Ele se concentra na aplicação desses padrões no desenvolvimento de jogos na Unity, e, por isso, os exemplos, bem como, as análises e situações apresentadas estão direcionadas especificamente para esse contexto.

5.1.1. Padrão *Singleton*

O *Singleton pattern* é um padrão bastante popular entre os desenvolvedores, provavelmente por ser muito fácil de ser implementado, como também, bastante poderoso. No entanto, por ser bastante poderoso, se utilizado de maneira inadequada, pode se tornar um problema.

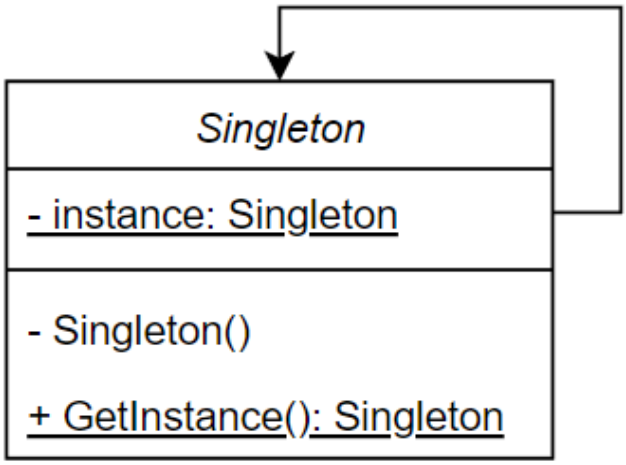
Charles Hache, desenvolvedor indie de jogos e professor de *game design*, afirma que esse padrão é especialmente útil no *Unity* para (1) gerenciar sistemas e serviços de jogos que exigem um único ponto de acesso, (2) estado persistente, e (3) quando você precisa garantir uma instância única na vida do jogo, como um *GameManage*r ou *AudioManager* (Hache, 2023)

Já Charles Amat, desenvolvedor sênior na empresa de desenvolvimento de software Force5 e dono do canal Infalible Code, fala que este padrão tem algumas desvantagens sérias que a maioria dos programadores não descobre até que seja tarde demais (Amat, 2020).

i. Definição

O *Gang of Four* define o padrão *Singleton* como uma classe que precisa garantir uma instância única e prover um acesso global para a mesma (Gamma et al., 1994).

Figura 7 – Representação UML do padrão *Singleton*.



Fonte: autoral.

Na figura 7 é possível ver uma representação do padrão no diagrama UML, no qual, de acordo com a definição, a classe contém uma variável estática com acesso privado que faz uma autorreferência, bem como, tem um construtor privado responsável por sua própria instanciação para garantir uma única instância e remover duplicatas. Por último, provê um acesso global para a instância.

ii. Implementação na *Unity*

Existem várias implementações de *Singleton*, em diferentes graus de robustez. A implementação ilustrada na figura 8 é a mais básica que pode ser feita na *Unity*. Como mencionado na seção sobre o ambiente *Unity* de desenvolvimento, para estar presente na hierarquia de objetos de cena e participar do *Lifecycle* da *Unity*, um *script* precisa herdar de *MonoBehaviour*. Por causa disto, não se pode instanciar diretamente com a palavra-chave *new*. Desta forma, a instância está sendo atribuída no *Awake( )*, método chamado ao carregar os objetos da cena.

Figura 8 – Código-exemplo de um *Singleton* simples na *Unity*.

```
using UnityEngine;

public class SimpleSingleton : MonoBehaviour
{
    public static SimpleSingleton Instance { get; private set; }

    private void Awake()
    {
        if(Instance == null)
        {
            Instance = this;
        }
        else
        {
            Destroy(gameObject);
        }
    }
}
```

Fonte: autoral.

Entretanto, esta implementação tem alguns problemas: caso um outro objeto acesse a instância *Singleton* no seu próprio *Awake( )*, é possível que a referência seja *null*, pois a ordem de execução deste método é não-determinística, podendo um objeto ser carregado anteriormente a outro que está acima na hierarquia de cena; Não há persistência dos dados entre cenas, pois ao trocar de cena todos os objetos são destruídos; Também é necessário que na cena corrente do jogo, um *GameObject* tenha um *script* da classe *SimpleSingleton* vinculado; E se houver mais de um S*ingleton* no jogo, é necessário repetir esse código em classes separadas.

Tendo isso em vista, a implementação da figura 9 corrige os problemas citados: caso um objeto acesse o *Singleton*, esta implementação utiliza *Lazy Instantiation*, no qual instancia o objeto só quando requisitado, e garante que haverá uma instância; Garante a persistência entre cenas, utilizando do método *DontDestroyOnLoad( )* no objeto *Singleton*; Se não houver um objeto na cena do tipo *Singleton*, quando for requisitado, cria um; e, também, permite a reutilização do código para que outras classes possam se tornar *Singleton*. Para isso, basta herdar de *Singleton*, como na figura 9.

Figura 9 – *Game Manager* e *Audio Manager* como *Singletons*.

```
using UnityEngine;

public class Singleton<T> : MonoBehaviour where T : Component
{
    private static T instance;
    public static T Instance
    {
        get
        {
            if (instance == null)
            {
                instance = (T)FindObjectOfType(typeof(T));
                if (instance == null)
                {
                    SetupInstance();
                }
            }
            return instance;
        }
    }
    public virtual void Awake()
    {
        RemoveDuplicates();
    }
    private static void SetupInstance()
    {
        instance = (T)FindObjectOfType(typeof(T));
        if (instance == null)
        {
            GameObject gameObj = new GameObject();
            gameObj.name = typeof(T).Name;
            instance = gameObj.AddComponent<T>();
            DontDestroyOnLoad(gameObj);
        }
    }
    private void RemoveDuplicates()
    {
        if (instance == null)
        {
            instance = this as T;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }
}
```

```
public class GameManager : Singleton<GameManager>
{
    // place your specific GameManager code here;
}

public class AudioManager : Singleton<AudioManager>
{
    // place your specific AudioManager code here;
}
```

Fonte: autoral.

iii. Prós e Contras

Segundo a Unity Technologies (2022), este padrão, diferentemente de outros, é relativamente fácil de aprender, assim como simples de utilizar, pois basta apenas referenciar a instância pública e terá sempre disponível o objeto *Singleton* em qualquer componente, inclusive, entre diferentes cenas. Além de ser performático, já que evita a busca por referências, como *GetComponent()* ou operações de *Find* (Unity Technologies, 2022).

Nystrom (2014) também menciona alguns de seus benefícios, como a possibilidade de salvar memória e ciclos de CPU na implementação com *Lazy Instantiation*; A utilidade de ter membros da classe disponíveis não-estáticos mas com acesso global; Capacidade de encapsular comportamento específico ao servir como uma interface abstrata para subclasses que desejam implementar, sem necessidade de alterar as chamadas da instância. (Nystrom, 2014).

Entretanto, a praticidade do *Singleton* a longo prazo pode ser bastante perigosa, principalmente, ao abusar do uso, visto que alguns autores são enfáticos ao afirmar que este padrão encoraja acoplamento, tornando o código mais difícil de refatorar, assim como, podem esconder dependências tornando mais difícil de solucionar bugs. Como também, impactam negativamente na testabilidade do código, pois segundo a Unity Technologies: "*Singletons* dificultam os testes: os testes de unidade devem ser independentes uns dos outros. Como o *Singleton* pode alterar o estado de muitos *GameObjects* na cena, eles podem interferir no seu teste."

Para se utilizar bem deste padrão, é necessário entender bem os contextos e as necessidades que o exige, pois de acordo com a Unity Technologies: "Desenvolvedores tendem a aplicar *Singletons* em situações inapropriadas, introduzindo estados ou dependências globais desnecessárias" (Unity Technologies, 2022). Por isso, Nystrom (2014) se preocupa na

utilização deste padrão, tanto que em seu livro, no capítulo sobre *Singletons*, comenta: “Este capítulo é uma anomalia. Todos os outros capítulos deste livro mostram como usar um padrão de projeto. Este capítulo mostra como não usar um.”

Contudo, a utilização deste padrão é um assunto que divide opiniões. Enquanto uns não encorajam o uso deste padrão, pois reiteram que este padrão costuma fazer mais mal do que bem (Nystrom, 2014), outros dizem que existem contextos dos quais podem ser interessante, como a Unity Technologies orienta a utilizar, desde que utilize em jogos pequenos dos quais não precise estender continuamente como jogos de nível empresarial fazem e se use ao mínimo em scripts que precisem de acesso global como Audio Manager, Game Flow Manager (Lin, 2021). Ou então, como Tulleken (2016) que em seu artigo 50 Tips and Best Practices for Unity recomenda utilizar Singletons por conveniência, contanto que evite utilizar para classes que não são únicas e não são gerenciadoras, como a classe Player. Deve-se utilizar para classes que sejam únicas e gerenciem sistemas, como UI Manager, Audio Manager, Game Manager. (Tulleken, 2016).

iv. Alternativas

O singleton pattern não é a única solução para acesso global ou que tenha acesso compartilhado de membros da classe com outros componentes. Em suas próprias palavras, Nystrom afirma: “Nunca usei a implementação completa [de singleton] do Gang of Four em um jogo. Para garantir a instanciação única, geralmente simplesmente uso uma classe estática. Se isso não funcionar, usarei um sinalizador estático para verificar em tempo de execução se apenas uma instância da classe foi construída” (Nystrom, 2014).

Nem sempre a melhor solução precisa ser algo robusto, muitas vezes, pode ser algo simples, como simplesmente passar a referência de um objeto para um método, explicitando a dependência (Nystrom, 2014).

Diante disso, vale ressaltar alguns padrões que podem substituir o singleton em alguns contextos, como o Service Locator, já que torna um objeto globalmente disponível, utilizando de métodos estáticos para prover serviços, porém precisa de uma pré-configuração para atribuir a implementação que será utilizada. Esta abordagem tem o benefício da flexibilidade, pois necessitando alterar o serviço, basta alterar a configuração e a implementação é trocada (Nystrom, 2014).

Por fim, o Subclass Sandbox pattern é outra opção a ser considerada, pois apesar de não dispor a instância globalmente, ele restringe o acesso apenas para as subclasses, o que pode tornar o acesso mais seguro e organizado em determinados cenários (Nystrom, 2014).

5.1.2. Padrão *State*

Oriundo do *Finite State Machine* (FSM), o qual consiste em um objeto guardar o status de um determinado momento e, a partir desse status, permitir alteração de seu comportamento baseado no status. Este padrão se mostra essencial no desenvolvimento de jogos, pois muitos elementos de jogos têm suas ações definidas para determinados estados. Esta afirmação pode ser corroborada por Galach (2019), que diz que a máquina de estados é um dos padrões de projeto mais utilizados no desenvolvimento de jogos. É útil para uma variedade de propósitos, como IA, animações, controladores de jogo, lógica de jogo, diálogos, cenas e muito, muito mais (Galach, 2019).

Como mencionado por Hache (2023), o *State Pattern* é uma forma de implementar o *Finite State Machine* (FSM), de um jeito mais limpo e manutenível, sendo útil para gerenciar comportamentos dependente de estados complexos. Ao encapsular os estados em objetos separados, o código fica mais modular e fácil de estender a novos estados.

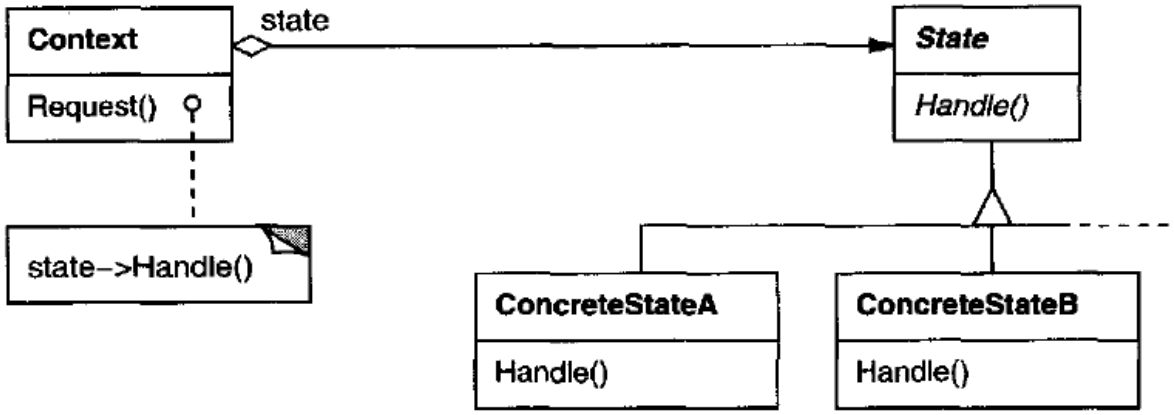
i. Definição

O *Gang of four* define o *State Pattern* como: “Permita que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe.” (Gamma et al., 1994). Ou seja, significa que ao mudar de estado, a classe deve mudar de comportamento.

Entretanto, para Nystrom (2014), apesar desta definição não estar equivocada, ela por si só não é tão clara, visto que é possível implementar esta definição ao utilizar apenas de um switch case para cada estado na classe, definindo uma ação específica para cada um no mesmo local. Esta implementação ainda teria problemas, já que para adicionar estados ou modificar comportamentos prévios, a mesma classe seria modificada (Nystrom, 2014).

Neste sentido, Nystrom (2014) esclarece que é importante encapsular os comportamentos de cada estado numa classe separada, dessa forma, ao adicionar novos estados, não é necessário alterar a classe que controla os estados. Ao modificar um comportamento de um estado existente, a mudança fica isolada na classe do estado, como também proporciona a possibilidade de reuso. Deixando, portanto, o código mais modular e flexível a mudanças (Nystrom, 2014).

Figura 10 – Diagrama UML do *State Pattern*.



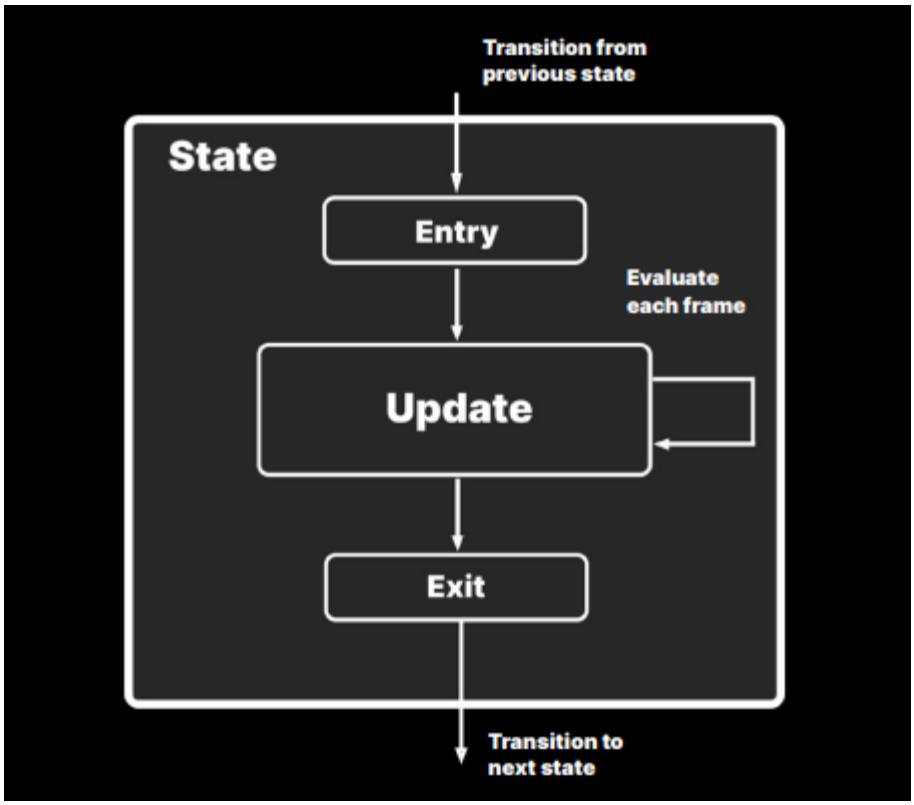
Fonte: Gamma et al. (1994).

Conforme a figura 10, podemos ver um diagrama UML, proposto por Gamma et al. (1994), que define o design *State Pattern* de forma geral e básica. Nela podemos observar que a classe *State* define uma interface, a qual estados concretos devem implementar seu comportamento específico e encapsulado. Como também, percebe-se que a classe *Context*, guarda uma referência para a classe *State* e apenas chama o método definido na abstração do *State*, o método *Handle().*

Isto é, o *Context* pode trocar de estado, e assim mudar de comportamento, sem depender de uma implementação concreta. Para adicionar novos estados, basta apenas criar uma nova classe e implementar a interface *State*, sem alterar código na classe *Context*. Além disso, a interface e seus métodos definidos são completamente contextuais, podendo variar entre as implementações, inclusive, é possível que os estados tenham referência para a classe *Context*, com a finalidade de terem acesso a uma forma de trocar de estado.

Entretanto, em jogos, o *State Pattern* tem outros elementos comuns em diversas implementações como: um método para a entrada do estado, um que seja executado a todo frame e um que seja executado na saída do estado, como mostra a figura 11, a qual representa um possível fluxo da classe *State* (*e-book Unity*). Vale ressaltar que nem todo estado implementa todos os métodos, muitas vezes estados simples têm alguns dos métodos vazios.

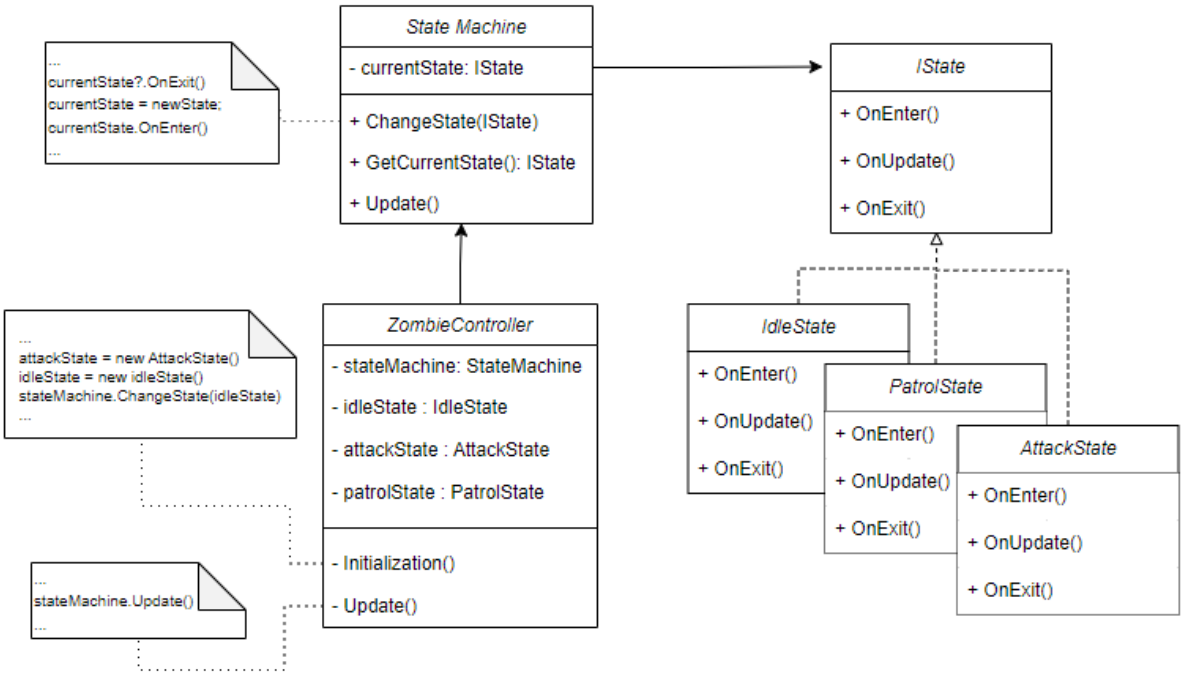
Figura 11 – Representação de um fluxo do *State* no contexto de jogos.



Fonte: Lin (2021).

Ainda neste contexto, este gerenciamento da transição de estados deve garantir a ordem de execução destes métodos da interface *State*, sendo de responsabilidade de uma classe especializada, normalmente conhecida como *State Machine*, a qual será melhor detalhada na próxima seção. Assim, uma outra classe tem a responsabilidade de estabelecer e iniciar os estados, como também a máquina de estados. Esta classe desempenha o papel de elo central, onde se define as implementações concretas, ao mesmo tempo em que fornece os parâmetros essenciais para os estados. Na Figura 12, é possível reconhecê-la como o "*ZombieController*", o qual utiliza do *State Pattern* para compor um zumbi no contexto deste exemplo.

Figura 12 – Exemplo comum do *State Pattern* em jogos em um diagrama UML.



Fonte: autoral.

ii. Implementação

Como mencionado antes, *design patterns* não são soluções prontas, elas dependem do contexto. Em geral, são apenas uma ideia para resolver um problema comum de determinadas situações. Cada padrão pode ter uma implementação com complexidade diferente, por isso, nem sempre encaixam em qualquer situação, visto que podem introduzir complexidade desnecessária a priori. Isto acontece com o padrão *State*.

Dito isto, esta seção abordará um exemplo de situação-problema, a qual o padrão *State* promete melhorar, como também, abordará duas implementações na *Unity* do *State Pattern*: uma solução mais básica que tem um problema de acoplamento, porém mais fácil de utilizar, sem preocupação com a reutilização à primeira vista. E, outra solução um pouco mais complexa proposta por Jason Weimann (2020), mas que reduz o acoplamento e aumenta a reusabilidade dos estados. Vale ressaltar que os exemplos a seguir, não estão completos, apenas demonstram um conceito. Além disso, esta seção tentará mostrar algum dos recursos dos quais este padrão possibilita com a finalidade de demonstrar sua utilidade.

Dando continuidade, vejamos a situação-problema anteriormente mencionada. O Anexos A e B mostra um exemplo de implementação simples de uma FSM, a qual define o

comportamento de um Zumbi que tem os estados de Patrulhar, Atacar e Parado. Toda a lógica está contida dentro da mesma classe, a qual troca de comportamento através de um *switch case* no método *Update( )*, que define uma ação para cada possível estado atual.

É possível visualizar que a implementação tem mais de uma responsabilidade, visto que o script tenta definir comportamentos diferentes numa mesma classe. Pode-se ver também que a implementação do estado de patrulha, precisa de uma *flag* para apenas executar uma vez uma certa rotina, assim como precisa depois limpar o que foi feito no momento de troca de estado, mexendo em diferentes regiões do código. Considerando isso, imagine se os outros estados estivessem completamente implementados, também precisando de novas *flags*, novos métodos, em breve a classe poderia ficar gigante, difícil de manter e com praticamente nenhuma reusabilidade, visto que haveria muitas dependências aninhadas na mesma classe, dificilmente uma outra classe poderia adicionar comportamento a esta implementação.

Apesar da implementação ser funcional, não é escalável, muitas vezes pode ser útil para testar conceitos ou para comportamentos pequenos dos quais não terão alterações. Contudo, a *Unity* enfatiza que este tipo de implementação pode rapidamente se tornar uma bagunça, pois para adicionar novos estados ou modificar estados existentes, será necessário revisitar a mesma classe várias vezes (Lin, 2021).

De maneira similar, Nystrom (2014) reafirma isto e adiciona que rapidamente a classe poderá inflar, visto que novos atributos específicos de cada estado seriam introduzidos na mesma classe para controlar as transições entre os estados, misturando e inflando a classe. Tornando mais difícil a manutenção e acoplada (Nystrom, 2014).

Dessa forma, surge o *State Pattern* com a finalidade de tornar os estados independentes, separando em classes diferentes. Os Anexos C, D, E e F compõem um exemplo de implementação do padrão, de forma simples, para controlar um zumbi. A fim de ter uma compreensão mais aprofundada dos benefícios desta implementação, o conteúdo será abordado de maneira segmentada.

Assim como vimos na última seção, esta implementação utiliza de uma interface *IState*, a qual tem os métodos *Enter*, *Update* e *Exit* definidos para serem executados assim que entra no estado, a cada execução de frame, e na saída do estado, respectivamente. A interface pode ser visualizada no anexo C. Além disso, o anexo D também define a classe *State Machine*, a qual gerencia a troca de estados, garantindo a ordem de execução dos métodos da interface *IState*. É importante perceber que essas implementações provavelmente não precisarão de alterações e podem ser reutilizadas em vários outros contextos, já que dependem apenas de uma abstração.

Analisando o anexo E, é possível ver uma melhoria clara na classe Zumbi *Controller*, a qual ficou bem mais enxuta, os membros da classe foram minimizados. Agora, a classe ficou com a responsabilidade de definir, inicializar e repassar os parâmetros necessários para os estados, como também, para a máquina de estados. Ou seja, a classe agora está apenas com a responsabilidade de unir os componentes necessários para definir o comportamento de um Zumbi. Dessa forma, ao adicionar novos estados, provavelmente as novas alterações seriam pontuais com pouca alteração de código e menos impactantes em outras áreas.

Por último, no anexo F, pode-se ver a classe *Patrol State*, o qual é o único estado implementado no exemplo, visto que é suficiente para representar uma implementação concreta de estado. É possível observar que a classe ficou mais legível e autocontida, ou seja, qualquer alteração lógica relacionada a como orquestrar o comportamento de patrulha de um zumbi, está concentrado nesta classe.

Contudo, o estado de patrulha ainda tem um grande problema, que é comum aos desenvolvedores na *Unity*, devido a algumas dificuldades em resolver dependências. A classe está fortemente acoplada à classe *Zombie Controller*, visto que em vários momentos acessa a referência da classe do zumbi para acessar os métodos de troca de estado, como também, as referências para as instâncias de outros estados com a finalidade de direcionar a transição do próximo estado. Isto é, se porventura um desenvolvedor quisesse reaproveitar o estado de patrulha para o comportamento de um esqueleto, por exemplo, não seria possível. Primeiro que não seria possível devido à amarração a classe do zumbi, como também, seria necessário que as transições do esqueleto também fossem iguais as transições do estado de patrulha.

Dessa forma, é interessante desacoplar estas dependências que não deveriam ser de responsabilidade do ato de patrulhar, mas da classe que une os comportamentos individuais, neste exemplo, ficaria mais coeso, a classe Zumbi *Controller* definir quais seriam as transições que zumbis deveriam ter a partir de determinados estados.

Assim, Weimann (2020), desenvolvedor *Unity* de jogos que tem um canal famoso de tutoriais na *Unity* com 193 mil seguidores, propõe uma forma interessante de desacoplar a lógica de transição de cada estado. A ideia consiste em mover a condição de troca de estado para a classe *Controller*, a qual pertencia ao estado, ao adicionar transições a *State Machine* definidas pelo trio: estado de origem, estado para transacionar e uma condição para a transição ocorrer. Assim, a *State Machine* teria uma nova responsabilidade, não apenas a responsabilidade de verificar se as condições de transições foram atingidas, mas também prover a troca de estado antes da execução do método *Update* (Weimann, 2020).

Para entender melhor este conceito, é necessário visualizar os anexos G, H, I e J, pois foi necessário alterar as classes *Zombie Controller, State Machine* e *Patrol State* as quais foram anteriormente definidas, assim como, definir uma nova classe chamada *Transitions* (anexo G), a qual é apenas uma definição para um par (estado, condição), sendo o estado para ir, se a condição for atingida.

Dando continuidade, podemos identificar no anexo H, a classe *State Machine* alterada, com uma nova lógica para considerar as transições. A classe tem um dicionário no qual guarda uma lista de transições baseada numa chave com o tipo da classe, ou seja, ela faz relação com o estado (tipo) e uma lista de possíveis transições do estado, suportando *n* transições. Adicionalmente, disponibiliza um método para adicionar estas transições, como também, verifica no método update se as transições do estado atual foram atingidas, caso seja positivo, troca de estado. Em caso negativo, apenas executa a atualização do estado.

Nesse contexto, a classe *Zombie Controller* também teve alterações (anexo I), o qual descartou as referências dos estados e adicionou transições à máquina de estados. A priori, a classe pode ter ganho um pouco mais de complexidade, mas o ganho foi considerável, visto que as transições não são mais ditas pelos estados. São ditas pela classe controladora e repassadas para a máquina de estados, tornando mais fácil criar novas classes de outros monstros, ou até mesmo, outras classes que nem tenham relação com monstros. Como por exemplo, um soldado, o qual poderia, enquanto patrulha, avistar um inimigo e tentar avisar a outros soldados a presença de inimigos, para isso, bastaria adicionar à máquina de estados suas próprias transições relacionadas ao estado de patrulha.

Por fim, é importante notar que a classe responsável pelo estado de Patrulha (anexo J), ficou um pouco menor. Entretanto, o fato de maior relevância é que ficou mais fácil de ser utilizada por outros controladores, dando mais flexibilidade ao controlador decidir para qual estado ir enquanto está realizando uma patrulha, visto que houve um desacoplamento da classe *Zombie Controller*, a classe apenas depende dos componentes essenciais para executar o comportamento de patrulha, como componente de movimentação, deteção de inimigo, etc.

Além disso, a classe ficou mais fácil de ser estendida, com poucas alterações, como alterar os modificadores de acesso dos membros da classe, é possível reaproveitar alguns comportamentos. Dessa forma, a Lin (2021) enfatiza que acaba sendo inevitável criar estados mais complexos que herdam de outros estados, visto que haverá alguns comportamentos em comum. Quando é utilizado de uma forma mais estruturada, criando níveis de herança, este tipo de estrutura pode ser conhecida como máquinas de estado hierárquicas (Lin, 2021).

Assim como, a classe poderia depender de abstrações ao invés de implementações concretas, como diz um dos princípios do *SOLID*, princípio da inversão de dependência, para dar ainda mais flexibilidade. Se a classe de patrulha dependesse de uma abstração para o componente de movimentação, por exemplo, daria ainda mais liberdade para os controladores escolherem diferentes implementações de movimentação para o estado de patrulha, dando ainda mais flexibilidade.

iii. Melhorias

Considerando ainda a implementação anterior do *State Pattern*, algumas melhorias ainda poderiam ser listadas, entretanto adicionariam um pouco mais de complexidade à arquitetura do padrão *State*.

Para começar, em outros cenários, pode ser útil ter outros métodos na interface *IState*. Um exemplo disto é o método *FixedUpdate*, o qual é similar ao método *Update*, mas não varia de acordo com o *frame rate* do dispositivo, ele executa uma vez a cada x milisegundos garantidamente. Podendo ser útil para calcular física, por exemplo. Ou de repente, pode-se adicionar um método chamado *HandleInput()* para lidar com a entrada de um controle, que difere para cada estado que o personagem controlado está disparando diferentes ações. Estes métodos da interface dependem de cada contexto.

Adicionalmente, pode-se criar estados base para facilitar implementações em comum entre os estados. Uma possível demonstração disso seria que se todo estado precisar de um Timer, assim como o estado de patrulha precisa nos exemplos citados, poderia ser o caso de criar um estado base do qual disponibilizaria esta implementação através de herança. Entretanto, é necessário ter cuidado pois herança se mal utilizada, pode levar a quebras de alguns dos princípios do *SOLID*, assim como outros problemas.

Outro caso, seria adicionar um novo tipo de transição à máquina de estados, com precedência as transições anteriores, a qual poderia ter um dicionário de transição similar ao apresentado, porém, com a diferença de que este novo dicionário definiria transições de qualquer estado para outro, ou seja, é um dicionário separado que dita transições gerais, das quais qualquer estado pode transicionar para. Um exemplo disto é o estado de morto no caso de um monstro, visto que se um monstro morrer, independe qual é o estado atual, deve-se transicionar para o estado de morto, já que ficaria impossibilitado de realizar ações. Este tipo de transição adiciona bastante complexidade e pode causar *bugs*, se mal utilizado.

Além disso, pode-se registrar os estados as transições da máquina de estados numa pilha, para caso algum estado dependa de ações anteriores, possa ter um histórico para resgatar o

último estado, Nystrom define este comportamento como *Pushdown Automata* (Nystrom, 2014).

Finalmente, um último cenário alternativo, poderia ser a utilização de mais de uma máquina de estado no mesmo controlador. Nystrom (2014) define este tipo de uso como máquinas de estado concorrentes, as quais são independentes, mas também, podem ter algumas relações, ao invés de criar um estado novo para cada possível combinação, pode-se apenas checar os estados atuais das máquinas. Um exemplo citado é quando um jogador tem uma máquina de estado para suas ações como andar, atirar, pular, nadar, entre outras, mas também tem outra máquina de estados para os equipamentos como armas etc. Não é necessário criar um estado novo para andar com uma arma, porém pode-se haver relações como no estado atirar do qual depende do equipamento (Nystrom, 2014).

iv. Prós e contras

Tal como dito anteriormente, o padrão State, em jogos, é uma necessidade bastante comum. Um exemplo disto é para definir comportamentos de inteligência artificial como inimigos. Se o jogo a ser desenvolvido for utilizar bastante de inteligência artificial, diferentes tipos, ações etc., usar este padrão pode ser bastante benéfico pois ajudará a seguir alguns princípios do padrão *SOLID*, tornando o código mais flexível e manutenível, a custo de complexidade.

Normalmente, o custo-benefício aumenta quando se tem muitas situações similares de uso e quando se espera que o projeto irá crescer e ganhar novos estados, novos comportamentos, como o padrão reforça o princípio aberto-fechado, é bastante útil nesses cenários.

O padrão *State* pode ajudá-lo a aderir aos princípios *SOLID* ao configurar a lógica interna de um objeto. Cada estado é relativamente pequeno e apenas monitora as condições de transição para outro estado. Mantendo o princípio aberto-fechado, você pode adicionar mais estados sem afetar os existentes e evitar trocas ou declarações complicadas. Por outro lado, se você tiver apenas alguns estados para monitorar, a estrutura extra pode ser um exagero. Esse padrão só pode fazer sentido se você espera que seus estados cresçam até uma certa complexidade.

5.1.3. Padrão *Command*

Akhtar (2020), desenvolvedor de jogos, traz uma analogia interessante sobre o padrão *Command*, comparando-o com o ato realizar pedidos na realidade, o qual uma pessoa realiza

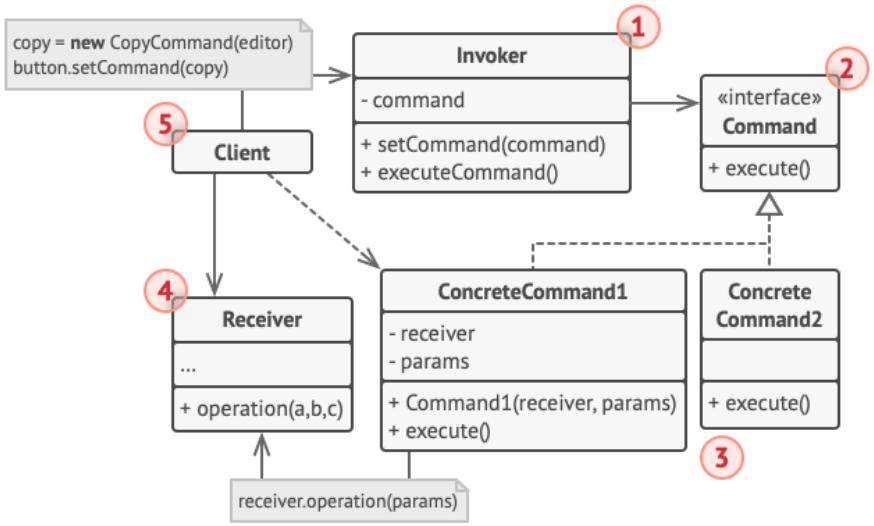
pedidos (ou ordens ou comandos) para outra pessoa, que pode (ou não) realizar o pedido designado. O padrão *Command* funciona de forma análoga, ele é responsável por transmitir de um componente para outro a execução de algum pedido (Akhtar, 2020).

Ele se encontra presente em muitos softwares e jogos, sua aplicação mais comum é prover uma forma de desfazer ações, como desfazer erros em um editor de texto, ou planejar ações em um jogo de estratégia como *Starcraft* (jogo famoso de estratégia em tempo real, no qual consiste em controlar unidades e conquistar bases inimigas). Mas muito mais simples que isso, muitas vezes, jogos disponibilizam uma forma do usuário escolher quais botões, ou atalhos, o usuário quer atribuir para determinadas ações, como fazer um personagem pular ao clicar na barra de espaço de um teclado. Normalmente por trás disto, está o padrão *Command*.

i. Definição

Este padrão também foi originado pelo grupo conhecido como *Gang of Four*, o qual definiram como um padrão que: “Encapsula uma solicitação como um objeto, permitindo assim parametrizar clientes com diferentes solicitações, enfileirar ou registrar solicitações e oferecer suporte a operações que podem ser revertidas” (Gamma et al., 1994).

Figura 13 – Representação UML do padrão *Command*.



Fonte: Dmitry Zhart (2023).

É possível observar na figura 13, uma das possíveis representações deste padrão em UML. Pode-se considerar, a parte indicada pelo número 2 na figura 13, como a parte mais

importante deste padrão, visto que define uma interface *Command* com um método *Execute()*, que servirá de base para desacoplar as chamadas de execução do comando.

As classes concretas de comando (número 3 na figura 13) implementam esta interface e definem uma ação dentro do método *Execute()*. Elas servem apenas como uma espécie de intermediação entre o invocador da ação e o receptor da ação. Encapsulam uma chamada de um objeto que tem a lógica de negócio, o qual é recebido como referência no momento da construção do comando, como também, outros parâmetros relacionados, se necessário.

Normalmente, a classe *Receiver* (número 4 figura 13) pode ser qualquer objeto que, de fato, realiza o trabalho por trás da ação. São chamadas dentro do comando e quem realmente sabe os detalhes da implementação da ação.

O Client (número 5 na figura 13) tem a responsabilidade de construir o comando e passar os parâmetros, incluindo o Receiver, para o comando. Mas também tem a responsabilidade de associar o comando desejado para a classe Invoker, delegando a execução do comando, de fato, para a classe Invoker.

Por fim, o *Invoker* (número 1 na figura 13) é responsável por inicializar as execuções do comando, visto que esta classe, é a que gerencia as chamadas dos comandos. Dito isto, esta classe pode executar instantaneamente o comando, como também pode não realizá-lo, pode atrasar a execução do comando, pode enfileirar para aguardar execução, pode guardar em estruturas como pilhas e outras formas de armazenamento com a finalidade de ter um histórico resgatável das execuções dos comandos. Tudo depende do contexto da aplicação. É importante notar que apesar da classe poder executar os comandos, ela apenas tem a referência para a interface *Command*, ou seja, não conhece a classe *Receiver* e nem as classes concretas de comando.

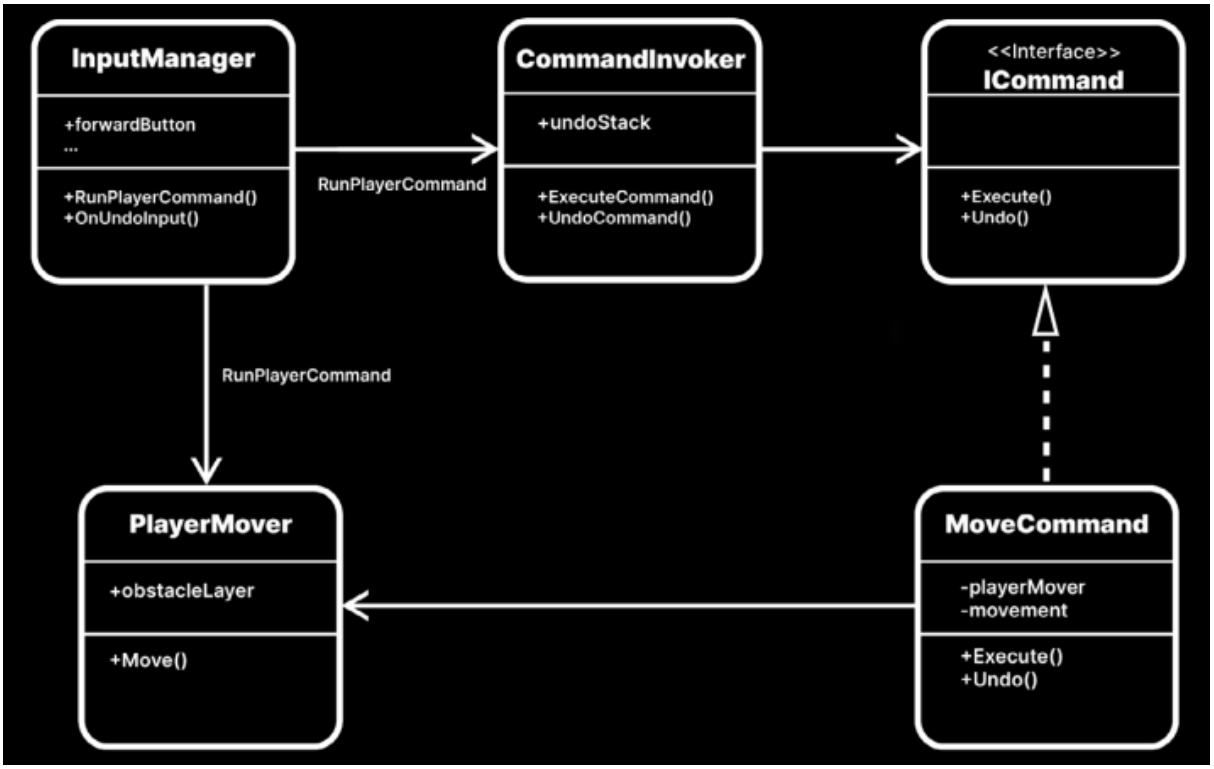
Assim, já se tem o padrão *Command*, entretanto, o ponto crucial é que basta alterar pouca coisa para permitir que o padrão desfaça execuções de comandos já feitos, algo bastante útil em muitas aplicações.

Basta adicionar um novo método na interface *Command*, um método responsável por desfazer o método *Execute()*, o método *ExecuteUndo()*. Adicionalmente, fazer os ajustes necessários, dado que as classes concretas de comando precisam implementar o novo método, e que o *Invoker* necessita gerenciar a ordem dos comandos apropriadamente, como guardando um histórico numa pilha. Pronto, com isso, agora é possível realizar implementações úteis em jogos, como simular ações e desfazê-las, ou até mesmo possibilitar refazê-las.

ii. Implementação

Existem implementações diversas do padrão *Command*, assim como qualquer padrão de projeto. Contudo, esta seção aborda uma implementação exemplo disponibilizada pela própria *Unity* no *e-book* "*Level up your code with game programming patterns*", a qual ilustra bem o uso deste padrão, que pode servir como base para outras implementações mais complexas deste padrão.

Figura 14 – Representação UML de exemplo do *Command Pattern*.



Fonte: Lin (2021).

Para melhor ilustrar, será demonstrado na prática, as classes definidas na seção anterior, com implementações reais no contexto de aplicar um comando para mover um jogador. Na figura 14, pode-se observar a estrutura geral do exemplo, numa representação em UML, a qual a classe *MoveCommand* é uma implementação concreta da interface *Command*; A classe *PlayerMover* representa a classe *Receiver*; A classe *InputManager* representa o *Client*; e a classe *CommandInvoker* representa o *Invoker*, como foram definidos na seção anterior.

Figura 15 – Interface *Command* implementada na *Unity*.

```
public interface ICommand
{
    void Execute();
    void Undo();
}
```

Fonte: Lin (2021).

Vejamos o código de cada classe separadamente, iniciando pelo código da interface *Command* (figura 15) que é a base do padrão, a qual define o método *Execute()*, responsável por executar o comando, como também, define o método *Undo()*, responsável por desfazer a execução de um comando realizado.

Figura 16 – Exemplo de código de um comando de movimentação de um jogador na *Unity*.

```
public class MoveCommand : ICommand
{
    PlayerMover playerMover;
    Vector3 movement;

    public MoveCommand(PlayerMover player, Vector3 moveVector)
    {
        this.playerMover = player;
        this.movement = moveVector;
    }

    public void Execute()
    {
        playerMover.Move(movement);
    }

    public void Undo()
    {
        playerMover.Move(-movement);
    }
}
```

Fonte: Lin (2021).

Dando continuidade, a classe *MoveCommand*, demonstrada na figura 16, implementa a interface *ICommand* anteriormente definida. Nela, é possível visualizar um exemplo de como mover um jogador através de um comando, o qual é bem simples. Apenas guarda a referência da classe *Receiver*, que no caso é a classe *PlayerMover*, a qual recebe através de parâmetros

pelo construtor, como também, um vetor que define a movimentação. Dessa forma, dentro do método *Execute()*, apenas monta uma chamada de método da classe *PlayerMover*, repassando o vetor de movimentação. E, de forma similar, realiza a mesma coisa no método *Undo()*, porém passa o vetor de movimentação na direção contrária como parâmetro com a finalidade de reverter a movimentação.

Dito isto, é importante mostrar a implementação da classe *PlayerMover*, pois ela é quem de fato dita como funciona a movimentação do jogador, ou seja, ela que contém a lógica de negócio.

Figura 17 – Exemplo de código de uma Classe *Receiver* do padrão *Command* na *Unity*.

```
public class PlayerMover : MonoBehaviour
{
    [SerializeField] private LayerMask obstacleLayer;
    private const float boardSpacing = 1f;

    public void Move(Vector3 movement)
    {
        transform.position = transform.position + movement;
    }

    public bool IsValidMove(Vector3 movement)
    {
        return !Physics.Raycast(transform.position, movement, board-
Spacing, obstacleLayer);
    }
}
```

Fonte: Lin (2021).

Como é possível observar na figura 17, a classe *PlayerMover*, é um componente da *Unity*, como qualquer outro, apenas com a especialidade de aplicar movimento ao *GameObject* do qual está vinculado, que no caso, define a movimentação de um jogador em um tabuleiro ou grid através do método *Move* que soma o vetor recebido à posição atual do jogador, de forma instantânea, fazendo-o movimentar.

Tendo isso em vista, vejamos como o comando é criado e, em seguida, executado. Para isso, deve-se olhar para a classe que representa o *Client*, que nesse contexto é o *InputManager*. Entretanto, esta classe tem outras lógicas as quais não são importantes para o exemplo, portanto será mostrado apenas a parte importante. Ao observar a figura 14, percebe-se que o método *RunPlayerCommand*, presente na classe *InputManager*, é o responsável por fazer a ligação entre o *PlayerMover* e o *CommandInvoker*.

Desta forma, basta apenas olhar o método *RunPlayerCommand*, na figura 18 abaixo. Neste trecho de código, é visível que a classe apenas cria um comando do tipo *MoveCommand*,

passando os parâmetros necessários, e passa o comando para ser executado pela classe *ComandInvoker* em seguida, através de um método estático disponibilizado pela classe *CommandInvoker*. É válido ressaltar que nem toda implementação da classe *Invoker* utiliza de métodos estáticos, neste caso, foi uma forma de centralizar a execução dos comandos de forma prática.

Figura 18 – Exemplo de código para criar comando na *Unity*.

```
private void RunPlayerCommand(PlayerMover playerMover, Vector3 move-
ment)
{
    if (playerMover == null)
    {
        return;
    }

    if (playerMover.IsValidMove(movement))
    {
        ICommand command = new MoveCommand(playerMover, movement);
        CommandInvoker.ExecuteCommand(command);
    }
}
```

Fonte: Lin (2021).

Por último, para finalizar, é importante ver como funciona a classe *CommandInvoker*, que é um exemplo de um *Invoker* (figura 19). Pode-se observar, no método *ExecuteCommand*, que além de executar o comando recebido, a classe guarda as referências para os comandos executados numa estrutura de pilha. Dessa forma, permite uma forma de resgatar o histórico e, caso seja necessário, realizar uma operação para desfazer o comando, apenas acessando o primeiro elemento da pilha e chamando o método *Undo()* presente na interface *ICommand*.

Figura 19 – Implementação exemplo de uma classe *Invoker* na *Unity* do padrão *Command*.

```
public class CommandInvoker
{
    private static Stack<ICommand> undoStack = new Stack<ICommand>();

    public static void ExecuteCommand(ICommand command)
    {
        command.Execute();
        undoStack.Push(command);
    }

    public static void UndoCommand()
    {
        if (undoStack.Count > 0)
        {
            ICommand activeCommand = undoStack.Pop();
            activeCommand.Undo();
        }
    }
}
```

Fonte: Lin (2021).

iii. Prós e contras

Tal como outros padrões abordados neste trabalho, este padrão introduz complexidade ao código, visto que adiciona uma camada entre o *Invoker* e o *Receiver*. Entretanto, este padrão não é tão complexo, o que pode fazer valer a pena para a grande maioria dos casos.

Curiosamente, Nystrom (2014) revela que este padrão é um de seus favoritos, pelo fato de que a maioria dos programas que escreveu, sejam jogos ou outros programas, ele sempre acaba utilizando-o em algum lugar. Quando bem usado, desembaraça códigos complicados.

Não é à toa a exaltação de Nystrom (2014) sobre este padrão, pois é simples de implementar e está bem alinhado com os princípios do *SOLID*, como o princípio da responsabilidade única e o princípio do aberto-fechado, já que desacopla as classes que invocam operações das classes que performam as operações e, também, pode introduzir novos comandos sem quebrar outros comandos feitos. Assim como pode criar um conjunto de comandos, compostos por outros comandos, para criar comandos complexos.

5.1.4. Padrão *Observer*

De acordo com Nystrom (2014), o padrão *Observer* é um dos mais amplamente utilizados padrões de projeto originais do grupo *Gang of Four*. Ele argumenta que este padrão

é tão difundido que Java o colocou em sua biblioteca principal (*java.util.Observer*), como também, C# o incorporou diretamente na linguagem (com a palavra-chave *event*).

Imagine um jogo, no qual o jogador recebe um dano de um monstro inimigo. Quando isso acontece, vários componentes diferentes do jogo podem querer reagir a esta situação. Entre os quais o componente de áudio pode tocar um áudio que representa dano; um componente de partículas pode liberar partículas de sangue; um componente de interface pode diminuir a barra que representa vida do jogador; entre várias outras possíveis reações. O padrão *Observer* é crucial neste tipo de cenário, o qual provê uma comunicação de "um-para-muitos" de forma desacoplada, deixando o código mais modular e flexível.

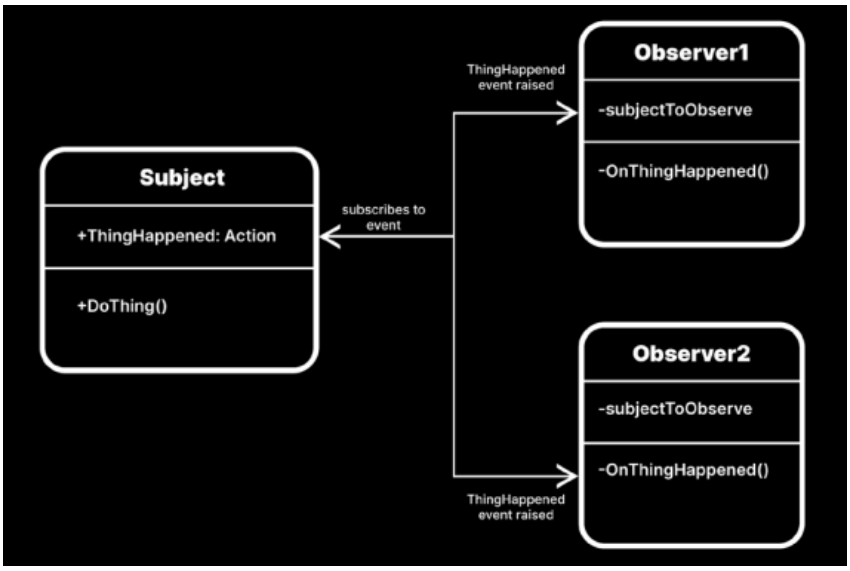
i. Definição

O grupo *Gang of Four* descreve o *Observer* como um padrão comportamental que: "Define uma dependência de um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente" (Gamma et al., 1994).

Em adição a isto, Nystrom (2014), em seu livro Game *Programing Patterns*, destrincha este padrão de forma mais prática. Trata-o como uma situação na qual há uma comunicação entre o *Subject*, objeto no qual é responsável por notificar outros objetos. Estes são chamados de *Observers*, eles escutam as notificações lançadas pelo *Subject* e reagem como bem entenderem.

Neste sentido, Nystrom (2014) ainda reforça que o *Subject* precisa ter uma lista dos Observadores dos quais deve notificar. Porém, não é de responsabilidade do *Subject* gerenciar esta lista. Ele apenas deve disponibilizar uma API pública para que os próprios *Observers* possam se adicionar como também se remover desta lista. Ou seja, os *Observers* que têm a responsabilidade de se inscrever e desinscrever da lista de notificação.

Figura 20 – Representação UML do padrão *Observer*.



Fonte: Lin (2021).

A *Unity Technologies* mostra a relação entre o *Subject* e os *Observers*, na figura 20, através de uma representação UML de uma possível implementação na *Unity Engine*. Nela, pode-se visualizar que foi escolhida a utilização de uma *Action* na classe *Subject* para referenciar as implementações dos *Observers*. Em termos conceituais, esta implementação poderia ser substituída por uma lista de interface *IObserver*, as quais os *Observers* deveriam implementar. Entretanto, assim como dito anteriormente, a linguagem C# já provê artifícios para facilitar o desenvolvimento deste padrão, que neste caso, a classe *Action* provê a API necessária para o registro de funções externas a *Subject*, as quais são delegadas para que sejam executadas no momento que a ação for invocada.

Tendo isso em vista, ainda na figura 20, é importante perceber que os *Observers* se inscrevem na ação *ThingHappened*, definida na classe *Subject*. Este último, apenas invoca a ação *Thing Happened* para aqueles que se registraram. Isto é, para a classe *Subject* não faz diferença se há apenas um observador ou vários. Inclusive, a classe *Subject* não tem noção do que os *Observers* fazem ao serem notificados, como também, os Observadores não sabem da existência de outros Observadores, eles agem de forma independente. Ou seja, pode-se adicionar novos *Observers* para reagir a *Action Thing Happened* e nada no código precisaria ser alterado, respeitando os conceitos o princípio da responsabilidade única e o princípio aberto-fechado do *SOLID*.

i. Implementação

Existem várias formas de implementar o padrão *Observer*, inclusive diferentes tipos de linguagem podem ter recursos diferentes para implementar este padrão. Em C# é muito comum se utilizar de eventos, palavra-chave *event* em C#, ou utilizar de ações, usando a classe *Action<T>*, ao invés de implementar utilizando uma interface *IObserver*. Além disso, algumas bibliotecas já utilizam este padrão por debaixo dos panos. Um exemplo disto é a classe *ObservableCollection<T>*, a qual consiste em uma coleção genérica de dados que implementa o padrão *Observer* e invoca eventos quando a coleção é alterada, permitindo que classes observadoras se inscrevam nesses eventos para executar rotinas.

Tendo isso em vista, este tópico abordará dois exemplos de uso do padrão *Observer*, importantes ao contexto aqui discutidos, na Unity. Inicialmente, será abordado uma implementação que diz respeito a uma situação hipotética de jogo, fazendo alusão às classes *Subjects* e *Observers* anteriormente discutidas na seção de definição. E, por fim, mostrar o uso de *UnityEvents*, um artifício da Unity que facilita o vínculo entre componentes utilizando apenas de um *drag and drop*.

A priori, na figura 21 abaixo, pode-se ver um exemplo prático de uma classe que está fazendo um papel de *Subject*. A classe *SingleEnemyDectector* representa um componente que detecta o inimigo mais próximo em um determinado alcance. Esta classe disponibiliza os eventos *OnNewEnemyDetected* e *OnStopDetectEnemy*, que são respectivamente disparados ao detectar um novo inimigo e ao perder a detecção de um inimigo previamente detectado. Ou seja, os *Observers* que estiverem inscritos nestes eventos serão notificados quando os eventos forem disparados.

Figura 21 – Exemplo de *Subject* na *Unity*.

```
public class SingleEnemyDetector : MonoBehaviour
{
    Private Variables

    public delegate void EnemyDetected(GameObject sender, Collider enemyDetected);
    public event EnemyDetected OnNewEnemyDetected;
    public event EnemyDetected OnStopDetectEnemy;

    public Collider DetectedEnemy
    {
        get { return enabled ? detectedEnemy : null;}
        set
        {
            if (detectedEnemy != value)
            {
                if(detectedEnemy != null)
                    OnStopDetectEnemy?.Invoke(self.gameObject, detectedEnemy);
                detectedEnemy = value;
                if(value != null)
                    OnNewEnemyDetected?.Invoke(self.gameObject, detectedEnemy);
            }
        }
    }

    private void FixedUpdate()
    {
        Detect();
    }
```

Fonte: autoral.

Isto posto, para implementar oficialmente o padrão discutido, basta apenas um objeto se inscrever nestes eventos para ser notificado, para assim, adicionar um comportamento em resposta ao evento. Deste modo, é válido observar as figuras 21 e 23, as quais associam uma rotina em resposta aos eventos disponibilizados na Figura 21, isto é, realizam o papel de *Observer*.

Figura 22 – Exemplo de *Observer* na *Unity*.

```
public class EnemyAimSpotter : MonoBehaviour
{
    [SerializeField]
    private SingleEnemyDetector enemyDetector;
    [SerializeField]
    private GameObject aimPrefab;

    private Dictionary<int, GameObject> aimDictionary = new Dictionary<int, GameObject>();

    private void OnEnable()
    {
        enemyDetector.OnNewEnemyDetected += AddAimToEnemy;
        enemyDetector.OnStopDetectEnemy += RemoveAimOfEnemy;
    }

    private void OnDisable()
    {
        enemyDetector.OnNewEnemyDetected -= AddAimToEnemy;
        enemyDetector.OnStopDetectEnemy -= RemoveAimOfEnemy;
    }
    private void AddAimToEnemy(GameObject sender, Collider enemyDetected)
    {
        GameObject aim = Instantiate(aimPrefab, enemyDetected.transform);
        aimDictionary.Add(enemyDetected.gameObject.GetInstanceID(), aim);
    }

    private void RemoveAimOfEnemy(GameObject sender, Collider enemyDetected)
    {
        var removedAim = aimDictionary[enemyDetected.gameObject.GetInstanceID()];
        aimDictionary.Remove(enemyDetected.gameObject.GetInstanceID());
        Destroy(removedAim);
    }
}
```

Fonte: autoral.

A classe *EnemyAimSpotter*, disponibilizada na figura 22, representa uma classe que adiciona uma mira para inimigos detectados, como também remove a mira adicionada quando perde a detecção do inimigo previamente detectado. Para isso, a classe inscreve o método *AddAimToEnemy*, o qual cria uma mira e guarda a referência para tal numa coleção de dados, para responder ao evento *OnNewEnemyDetected*, responsável por notificar quando um inimigo é detectado. Dessa maneira, sempre que um inimigo novo for detectado, uma mira será adicionada a este inimigo. O análogo também é feito para quando o inimigo sai de detecção.

Um ponto importante a ressaltar é que a classe remove o vínculo dos métodos discutidos com os eventos quando é desabilitada ou destruída, como podemos ver no método *OnDisable()* na figura 22. Este é um fator importante, pois se uma classe a qual não está mais disponível ainda estiver registrada em um evento, pode causar *bugs* quando o evento for lançado, já que este não sabe se a classe ainda está disponível e, dessa forma, falhará ao tentar notificar um

objeto indisponível. Segundo Nystrom (2014), este é um motivo comum de *bugs* ao utilizar o padrão *Observer*, por isso deve-se ser rigoroso quanto à limpeza da inscrição do *Observer* ao *Subject*.

Dito isto, é válido ressaltar que toda a lógica necessária para adicionar uma mira a um inimigo está auto-contida numa classe separada da classe que tem a lógica de detectar inimigos, ou seja, está alinhado com o princípio da responsabilidade única. Bem como, o princípio do aberto-fechado também está sendo respeitado, já que se for necessário adicionar novas lógicas, novas respostas, aos eventos discutidos, basta apenas criar novas classes para adicionar comportamento, sem necessariamente modificar o código pré-existente. Um exemplo disso é pode ser visto na figura 23, a qual mostra uma classe tocando sons ao detectar inimigos e ao perder a detecção.

Figura 23 – Segundo exemplo de *Observer* na *Unity*.

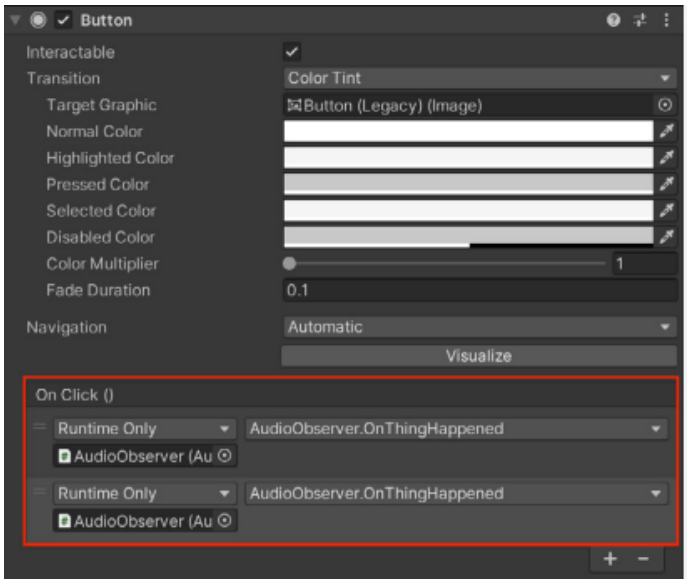
```
public class AudioPlayer : MonoBehaviour
{
    #region Target Acquired/Lost
    [SerializeField] private SingleEnemyDetector singleEnemyDetector;
    [SerializeField] private AudioClip targetAcquired;
    [SerializeField] private AudioClip targetLost;
    #endregion

    private void OnEnable()
    {
        singleEnemyDetector.OnNewEnemyDetected += PlayTargetAcquiredAudio;
        singleEnemyDetector.OnStopDetectEnemy += PlayTargetLostAudio;
    }

    private void OnDisable()
    {
        singleEnemyDetector.OnNewEnemyDetected -= PlayTargetAcquiredAudio;
        singleEnemyDetector.OnStopDetectEnemy -= PlayTargetLostAudio;
    }

    private void PlayTargetAcquiredAudio(GameObject sender, Collider enemyDetected)
    {
        AudioManager.Instance.PlayerAudio(targetAcquired);
    }

    private void PlayTargetLostAudio(GameObject sender, Collider enemyDetected)
    {
        AudioManager.Instance.PlayerAudio(targetLost);
    }
}
```

Fonte: autoral.

Na sequência, conforme mencionado no início deste tópico, o segundo exemplo trata-se da utilização de *UnityEvents*. Esta classe, a qual faz parte da biblioteca da *Unity*, funciona de forma similar a ações ou eventos, porém provê uma interface gráfica para o padrão *Observer*. O intuito é facilitar o vínculo entre o *Subject* e o *Observer*, como também, tornar mais prático o uso de eventos para pessoas que não são tão familiarizadas com programação.

Vários componentes de interface do usuário fornecidos pela *engine* utilizam de *UnityEvents*. Um exemplo disto é o componente de botão, o qual permite arrastar outros scripts da cena para vincular um método que será executado quando o evento *OnClick()* é lançado. Dessa forma, pode-se facilmente vincular algumas ações na Cena para adicionar comportamentos novos sem necessariamente programar. Pode-se visualizar o uso disso na figura 24 abaixo, a qual tem vinculado duas respostas ao evento *OnClick()*, o método *OnThingHappened* do script *AudioObserver* está vinculado duas vezes.

Figura 24 – Interface gráfica de *UnityEvents*, sendo utilizada em um botão na *Unity*.



Fonte: autoral.

Pode-se dizer que uma vantagem do uso desta classe é que nem o script do botão e nem o *AudioObserver* tem uma dependência explícita. No exemplo anterior, apesar do script responsável por detectar inimigos não saber quem escuta seus eventos, os observadores tinham uma dependência com a classe *SimpleEnemyDetector*. Já ao utilizar da interface para vincular os observadores, o vínculo fica registrado no arquivo de cena da *Unity*.

Entretanto, apesar de *UnityEvents* serem práticos em alguns momentos, sua performance é mais lenta do que se comparado aos eventos, ou ações, disponibilizados pela biblioteca base de C# e por isso, deve-se estar atento quanto a seu uso. Outro problema associado a este uso é que em projetos grandes pode não ser ideal arrastar cada script via editor. As referências podem ficar difíceis de arrastar, tanto em quantidade, quanto em complexidade, caso a hierarquia da cena esteja muito complexa. Este assunto será melhor explicado na seção sobre más práticas.

ii. Prós e contras

De acordo com a *Unity Technologies*, o padrão *Observer* ajuda a dissociar objetos, já que o publicador do evento não precisa saber nada sobre os próprios assinantes do evento. Em vez de criar uma dependência direta entre uma classe e outra, o sujeito e o observador comunicam-se mantendo um certo grau de separação (Lin, 2021). Em adição a isso, Nystrom (2014) fala que o padrão *Observer* é uma ótima maneira de permitir que pedaços de código não relacionados conversem entre si sem que se fundam em um grande pedaço.

Em outras palavras, este padrão permite que objetos sejam mais coesos e coerentes, devido ao fato de conseguir comunicar partes do código sem que haja uma dependência explícita entre as partes, como também, permite que cada observador implemente sua própria lógica para responder à notificação de forma independente, em conformidade com o princípio do aberto-fechado. De acordo com Lin (2021), isto simplifica a depuração de código, bem como, a realização de testes de unidade.

Além disso, outros fatores interessantes deste padrão é que existem diversas implementações prontas para uso, como mencionado anteriormente, e são extremamente úteis para a interface do usuário, já que permite separar a lógica de negócio da lógica da interface de usuário, fazendo com que a interface apenas mude quando notificada. Inclusive, esta é a base para outro padrão de projeto, o qual será melhor detalhado na próxima seção, o MVP, também conhecido como *Model-View-Presenter*.

Em contraponto, o *Observer* pode adicionar complexidade ao projeto, como também, pode impactar na performance a depender de seu uso. De acordo com Nystrom (2014), algumas pessoas podem considerar sistemas que utilizem de eventos como lentos, entretanto ele afirma que o custo associado ao padrão é em sua grande maioria insignificante, com exceção apenas de programas críticos de desempenho, pois existem artifícios dos quais podem mitigar o custo deste padrão, pois no final das contas, enviar notificações é simplesmente percorrer uma lista e chamar alguns métodos virtuais.

A primeira vista, o padrão ocorre de forma síncrona, com isso, ao inscrever diversos observadores com rotinas exaustivas, pode ocasionar travamentos. Entretanto, segundo a *Unity Technologies*, nestes casos, pode-se combinar o padrão *Observer* com o padrão *Command*, para disponibilizar uma espécie de fila de priorização dos eventos, esta solução é conhecida como *Event Queue* (Lin, 2021).

Outro problema deste padrão é que como o *Subject* não tem responsabilidade de gerenciar a lista de *Observers* dos quais estão registrados, é de responsabilidade do *Observer* se inscrever e se remover da lista. Desta forma é importante ter atenção para adicionar e remover observadores da lista de notificação. É necessário garantir que um objeto destruído, se remova da lista para não ocasionar erros. Conforme Nystrom (2014), este é um problema comum em sistemas de notificação, chamado de *lapsed listener problem*.

Por fim, com base em Lin (2021), os observadores ainda dependem da classe que está publicando o evento e isto, pode ser considerado um ponto negativo. Entretanto, existem implementações que ajudam a desacoplar ainda mais, como por exemplo, usar um gerenciador de eventos estático (ou *singleton*) que lida com todos os eventos e realiza a intermediação entre o *Observer* e o *Subject*.

5.1.5. Padrão MVP (*Model-View-Presenter*)

O padrão MVP (*Model-View-Presenter*) é um padrão arquitetural, uma variação do padrão MVC, o qual é comum de ser utilizado no desenvolvimento de interfaces de programas em geral, no intuito de reduzir dependências desnecessárias, separando-as em camadas, as quais têm suas responsabilidades bem estabelecidas.

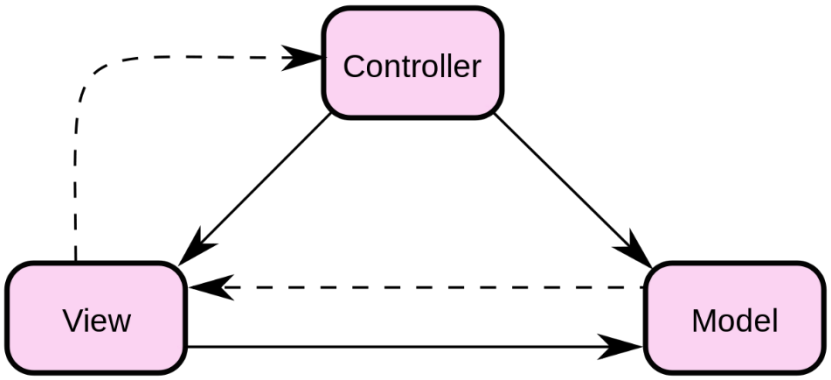
i. Definição

Primeiramente, antes de entrar nos detalhes da definição do padrão MVP, é importante entender o padrão, do qual o MVP foi originado, o MVC. Neste sentido, o padrão MVC é um acrônimo para *Model-View-Controller*, este nome representa as camadas das quais este padrão é dividido. Segundo Lin (2021), cada camada é bem definida, cada uma das partes do MVC, realizam apenas uma coisa e fazem isso bem, podendo se considerar, de forma superficial, como uma forma de extensão do princípio da responsabilidade única.

De acordo com Lin (2021). a camada *Model* é responsável por conter os dados e não performa lógicas de jogo ou algoritmos. Já a camada *View* é responsável por formatar e apresentar os dados de forma gráfica para o usuário. Por fim, na última camada, o *Controller* é

responsável por processar os dados de jogo, como realizar algoritmos e manipular dados em tempo de execução.

Figura 25 – Diagrama demonstrando as interações entre as camadas do MVC.



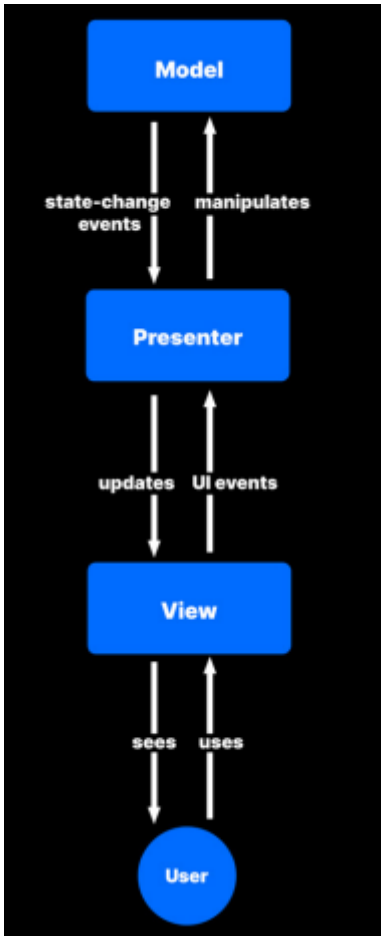
Fonte: Stannered (2010).

Existem diversas variações do próprio MVC a depender da linguagem e *framework* do qual se está inserido e utiliza de eventos (Padrão *Observer*) para se comunicar efetivamente com as camadas. Tendo isso em vista, uma possível visualização das interações do MVC está presente na figura 25. Num cenário hipotético, um usuário interage com a camada da *View* ao clicar num botão, disparando um evento de clique. Assim, o *Controller* escuta o evento e realiza alguma ação, como aplicar uma lógica e manipular os dados da *Model*, neste exemplo, poderia ser algo como realizar um pulo em um personagem do jogo, o qual poderia esbarrar num obstáculo e perder vida. Em seguida, a camada *Model* ao ser alterada, neste caso ter a vida alterada, dispara um evento visto que houve uma mudança de estado. Desta forma, a camada da *View* observa essa mudança e atualiza a representação gráfica.

É importante frisar que a camada *Controller* e a *View*, em conformidade com suas responsabilidades, não guardam dados para si e funcionam com base no padrão *Observer*. Além disso, no MVC é comum um *Controller* ter mais de uma *View* associada.

A partir disto, pode-se dar início à definição do padrão MVP, o qual substitui a camada *Controller* por uma camada chamada de *Presenter*, a qual, diferentemente do *Controller*, tem a função de ser um intermediário entre a *View* e o *Model*, deixando a *Model* mais distante da *View*, como pode-se observar na figura 26. Além disso, é menos comum a camada que substitui o *Controller*, ter várias *Views* associadas, é mais comum ser um para um. Segundo Lin (2021), apesar de serem parecidos e manterem a separação de conceitos, as camadas interagem de forma diferente, bem como, têm responsabilidades um pouco diferentes.

Figura 26 – Diagrama demonstrando as interações entre as camadas do MVP.



Fonte: Lin (2021).

Ao desenvolver no ambiente *Unity*, o *framework* de UI (*UI toolkit* ou *UnityUI*) provê diversos componentes que já agem como uma *View*, não necessitando desenvolver componentes individuais do zero, porém estes componentes são genéricos e não tem especialização.

Desta forma, os componentes que agem como *Presenters*, adicionam alguma funcionalidade de apresentação a estas *Views* disponibilizadas. No MVC, as V*iews* acabam sendo mais especializadas, pois como pode-se observar na figura 25, elas têm uma referência direta ao modelo. Lafritz (2022), corrobora com esta afirmação, ao dizer que o MVP é tipicamente utilizado na *Unity*, porque não se pode fazer muito com a *View* e a renderização, já que isto é feito de maneira interna ou pelos bastidores, e como a alteração destes elementos é integrada com eventos, faz necessário ter um intermediário para atualizar a camada da *View*, bem como, atribuir uma lógica para a *View*.

ii. Implementação

Para formalizar um exemplo de implementação deste padrão, será utilizado um código disponibilizado no *e-book Level your code with game programming patterns*, publicado pela *Unity Technologies*. Neste exemplo, será abordado um sistema de vida, o qual terá um modelo que representa a vida de um personagem, um apresentador que realiza uma lógica para fazer atualizações gráficas, como também, manipular os dados de vida; por fim, não será necessário implementar uma *View*, visto que a *Unity* disponibiliza um componente *Slider*, o qual se encaixa perfeitamente para representar uma barra de vida.

Para começar, pode-se visualizar no anexo K, o código relacionado ao modelo. Este consiste num modelo para representar uma vida de um personagem, disponibilizando APIs para modificação, como um método para incrementar, decrementar ou recuperar por completo a vida, assim como também disponibiliza um evento para quando a vida for alterada, outros componentes poderem ser notificados.

É importante notar que a classe *Health* não implementa lógica de jogo, não depende de nenhuma outra classe e, pode ser facilmente aproveitada para outras situações, como vida de itens etc. Esta classe não necessariamente precisaria herdar de *MonoBehaviour* visto que não utiliza do ciclo de vida de um componente da *Unity* e isto pode deixar desnecessariamente carregada a classe.

Em seguida, a classe *HealthPresenter* é demonstrada no anexo L, a qual tem uma referência para a classe *Health* e para um componente *Slider*, a qual atribui uma lógica simples para aplicar dano e cura, como também, realiza um vínculo entre o evento de alteração de vida com a atualização do valor da barra do *Slider*. Esta é a classe mais especializada dentre as outras classes, pois ela é um intermediário, mas dificilmente será reaproveitada para outras situações.

É importante notar que comportamentos adicionais poderiam ser implementados nesta classe, como por exemplo: não poder realizar uma cura se o componente de vida estiver zerado. Por fim, outros *GameObjects* irão interagir com o *Health Presenter* para aplicar dano, cura etc.

Como mencionado, a *View* já está implementada pela *Unity* e a lógica de como será o comportamento da barra de vida, está implementada na classe *HealthPresenter*, diferentemente de como seria no MVC à primeira vista. Outros fatores de comportamento poderiam ser feitos na classe *HealthPresenter*, como por exemplo alterar a direção da barra, entre outras lógicas relacionadas. Como o código da classe *Slider* não é disponibilizado, entretanto pode ser vista uma documentação sobre (Unity, 2023) Além disso, o relevante de se perceber aqui é que o componente *Slider* está separado de outros componentes, não tem dependência para outras

classes que não seja de renderização. Portanto, permite que facilmente seja reutilizado em outro contexto.

iii. Prós e contras

Tendo em vista que a abordagem do MVP reforça o princípio da responsabilidade única e separa bem os conceitos, é de se esperar que a longo prazo seja perceptível que o código fica mais fácil de manter e escalar.

De acordo com Lin (2021), neste tipo de abordagem, os desenvolvedores tendem a realizar classes menores e mais legíveis, com poucas dependências, o que provavelmente leva a ter menos lugares de código quebráveis ou escondendo *bugs*.

Em adição a isso, Lin (2021) afirma que promove uma divisão do trabalho, já que há uma modularidade maior de código, se você precisar de *Views* mais complexas, pode-se separar um desenvolvedor *frontend* para lidar apenas com a composição de *Views* para a interface, enquanto outros realizam a lógica de jogo.

Além disso, o uso de testes de unidade no código será facilitado, pois devido a separação bem estabelecida dos conceitos de lógica, interface e modelo, será mais fácil realizar técnicas de teste como o *mock*, o qual simula objetos para testar isoladamente alguns casos de teste, bem como, facilita o uso de testes da própria ferramenta de *Unity*, não necessariamente precisando executar o jogo para realizar testes (Lin, 2021).

Contudo, este padrão pode ser um pouco mais complexo que outros padrões de projeto, visto que utilizá-lo leva a criar mais classes e manter uma certa organização, projeto e componentes pequenos podem não beneficiar tanto deste padrão. Por isso, conforme a *Unity Technologies* afirma, é necessário planejar para averiguar se é o caso do projeto utilizar deste padrão, como também, nem todo contexto caberá facilmente nessas camadas, visto que nem todo componente da *Unity* é facilmente quebrado em dados, lógica e interface. Um exemplo disto, segundo Lin (2021), é o *MeshRenderer*, componente do qual renderiza a malha de triângulos que representa um modelo 3d.

5.1.6. Padrão *Factory*

Em jogos, frequentemente surge a necessidade de criar inimigos, obstáculos, itens e outros elementos dinamicamente durante a execução. Portanto, ter uma maneira de abstrair a lógica de criação de objetos em uma classe especializada pode se revelar altamente vantajoso a longo prazo, pois isso permite evitar modificações em classes que não necessitam de conhecimento sobre a classe exata que está sendo instanciada, dependendo apenas de uma

abstração comum, tornando o sistema mais flexível e fácil de manter, como também mais organizado.

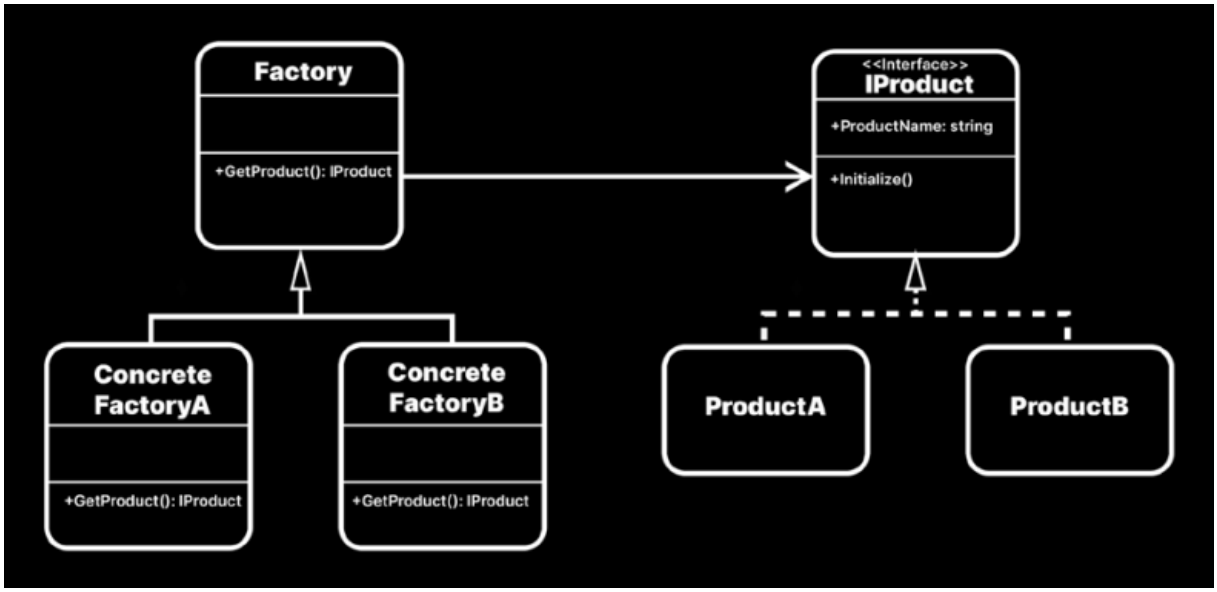
Isto posto, pode-se verificar que o padrão F*actory* se encaixa neste contexto, conforme esclarecido por Charles Hache (2023), em seu artigo "*Top 7 Design Patterns Every Unity Game Developer Should Know*". Vejamos: "O padrão *Factory* é particularmente útil quando você precisa criar vários tipos de objetos, como inimigos ou itens, que compartilham uma interface comum ou classe base. Ajuda a encapsular o processo de criação de objetos, tornando seu código mais sustentável e escalável" (Hache, 2023).

Além do mais, a *Unity Technologies* (2023) também afirma que às vezes é útil ter um objeto especial que crie outros objetos. Muitos jogos geram uma variedade de coisas ao longo do jogo, e muitas vezes você não sabe o que precisa em tempo de execução até que realmente precise.

i. Definição

O grupo *Gang of Four* estabelece o padrão *Factory* como um padrão criacional, o qual define uma interface para criar um objeto, mas permite que subclasses decidam qual classe instanciar. Este permite que uma classe adie a instanciação para suas subclasses (Gamma et al., 1994).

Figura 27 – Representação da estrutura do padrão *Factory*.



Fonte: Lin (2021).

Por conseguinte, pode-se visualizar a estrutura do padrão *Factory*, conforme ilustrado na figura 27. Primeiramente, é necessário definir uma interface comum entre os produtos, para assim, poder criar uma *Factory*, uma classe abstrata que tem um método para retornar uma instância de *IProduct*.

Os produtos concretos implementam a interface *IProduct* e definem seu comportamento específico. Bem como, as fábricas concretas implementam a classe abstrata *Factory* e, também, definem seu comportamento específico. Neste exemplo, os produtos concretos precisam definir o nome do produto, como também terem uma lógica de inicialização e as fábricas concretas sabem como instanciar um produto concreto, assim como, podem definir uma rotina a ser executada pós-criação.

Deste modo, quando um cliente pedir um produto para uma classe do tipo *Factory*, receberá uma instância de *IProduct*, ou seja, o cliente não tem conhecimento da classe especializada do produto, apenas da abstração. É importante notar que o cliente também pode depender da abstração de *Factory* ao invés de uma fábrica concreta, facilitando assim, a troca de implementação de criação de objeto, caso necessário.

É válido ressaltar que as fábricas podem precisar de alguma funcionalidade comum compartilhada. Por isso, nesta definição fala-se de classes abstratas. Contudo, de acordo com

Krogh-Jacobsen (2022), nestes cenários é importante atentar-se ao princípio da substituição de Liskov, um dos princípios *SOLID*, ao usar subclasses.

Além disso, não necessariamente precisam de fato criar uma instância, elas podem reaproveitar instâncias previamente alocadas, porém este cenário é mais conhecido como o padrão *Object Pool*, o qual tem uma seção destinada neste trabalho.

ii. Implementação

Esta seção abordará uma implementação exemplo do padrão *Factory* demonstrada no *e-book Level Up Your Code With Game Programming Patterns*, a qual ilustra uma fábrica que cria instâncias de um produto A numa certa posição do mundo, executa uma rotina de inicialização do produto e o retorna para o cliente que hipoteticamente solicitou. Além disto, o exemplo se aproveita do sistema de *prefabs* da *Unity*, conforme discutido anteriormente na seção deste trabalho sobre a *Unity*, o qual trata-se de um *GameObject* pré-configurado no editor que contém a receita de como instanciar o objeto, adicionando os componentes e parâmetros registrado no *prefab*.

Figura 28 – Interface *IProduct* e classe abstrata *Factory* do padrão *Factory* na *Unity*.

```
public interface IProduct
{
    public string ProductName { get; set; }

    public void Initialize();
}

public abstract class Factory : MonoBehaviour
{
    public abstract IProduct GetProduct(Vector3 position);

    // shared method with all factories
    …
}
```

Fonte: Lin (2021).

Dito isto, na figura 28 podemos ver uma definição da interface dos produtos deste exemplo, o qual apenas define um método para inicialização do produto, assim como, uma propriedade para o nome do produto. Adicionalmente, também é visível a definição da classe

abstrata *Factory*, a qual tem um método abstrato que precisa de um parâmetro de posição apenas para atribuir a posição do produto, bem como retorna um produto da interface *IProduct*.

É importante notar que esta interface bem como os métodos da classe abstrata depende do contexto o qual o produto e a fábrica estão inseridos, podendo ter mais elementos definidos em ambos, assim como, outros parâmetros. Um exemplo disto é que se o produto criado representasse um projétil, talvez fizesse sentido passar uma direção a qual o projétil foi disparado (além da posição inicial), bem como, repassar para a interface do produto. O interessante deste padrão é que se uma arma (classe hipotética) precisasse criar um projétil, ela não precisaria saber criar um projétil, esta responsabilidade estaria presente na fábrica de projéteis, a qual apenas solicitaria uma instância de projétil.

Continuando o exemplo, pode-se observar a implementação de um produto concreto e uma fábrica concreta no anexo L (*ProductA* e *ConcreteFactoryA*, respectivamente). Conforme o exemplo mencionado, é notório observar que a classe *ProductA* implementa a interface *IProduct*, ou seja, define o nome de seu produto, bem como, dita uma lógica de inicialização para ela mesma, procurando uma referência de um script de partícula, o qual manda parar a execução e recomeçar.

Nesse mesmo contexto, percebe-se que a classe *ConcreteFactoryA* tem uma referência para um *prefab*, ou seja, tem a receita para criar o objeto do tipo *ProductA*. Desta forma, no método *GetProduct*, instancia um novo *GameObject* a partir do *prefab* do tipo *ProductA*. Em seguida, executa o método de inicialização da interface *IProduct* e retorna o produto.

Dito isto, vale ressaltar que cada classe ficou com sua responsabilidade bem definida, já que o próprio produto sabe se inicializar e a fábrica sabe coordenar este processo de inicialização. Ademais, criar novos produtos não impacta um possível cliente que usufrui do serviço de uma fábrica visto que novos scripts seriam criados, mas nada no cliente precisaria ser alterado, já que este recebe apenas uma interface *IProduct*. Deste mesmo modo, trocar de fábrica (lógica de construção) seria algo trivial como apenas atribuir uma nova referência.Ou seja, ao criar um novo produto, ou modificar o atual para tocar um áudio ao invés de controlar partículas, nenhum código do cliente precisaria ser alterado.

Por fim, é válido salientar que a classe *Factory* não necessariamente precisa herdar de *MonoBehaviour*, porém no ambiente da Unity é comum e, muitas vezes, prático passar a referência de *prefabs* via *serialized field* no editor da *Unity*, assim como na demonstração de código de fábrica concreta ilustrada no anexo L. Neste sentido, outras abordagens poderiam ser utilizadas para a classe referenciar um produto, como utilizar de injeção de dependência, criar

objetos utilizando a palavra-chave new ou até mesmo procurar por objetos previamente criados na hierarquia de cena da *Unity*.

iii. Prós e Contras

O padrão *Factory* promove encapsulamento e abstração do comportamento de construção de um objeto, o que traz uma separação clara entre os componentes do sistema, bem como, permite que um cliente requisite um objeto a uma classe especializada na construção do mesmo, sem precisar conhecer detalhes de implementação. Por fim, permite adição de novos produtos sem quebrar código anterior no cliente, assim como, torna fácil a troca de algoritmo de criação. Ou seja, este padrão promove um código flexível, manutenível e escalável com acoplamento baixo.

Segundo Charles Hache (2023), em seu artigo "*Top 7 Design Patterns Every Unity Game Developer Should Know*", os benefícios deste padrão são o encapsulamento da criação de objetos, promoção de reusabilidade de código e escalabilidade, assim como, a promoção de baixo acoplamento.

Ademais, de acordo com o artigo *"Factory Method*", do *Refactoring Guru*, este padrão está alinhado com o princípio do aberto-fechado, pois é possível introduzir novos produtos no programa, sem quebrar código previamente existente no cliente. Como também, está alinhado com o princípio da responsabilidade única, visto que a parte do código relacionada a criação de objetos é movida para uma classe especializada em construir objetos, deixando o código mais fácil de manter. E por fim, evita o acoplamento entre o criador e as classes concretas dos produtos (Refactoring Guru, 2023).

Contudo, em alguns casos, pode não ser ideal implementar o padrão *Factory*, pois este pode introduzir um pouco de complexidade desnecessária. Krogh-Jacobsen (2022) corrobora com isto ao afirmar que o benefício máximo deste padrão é atingido quando se tem muitos produtos para configurar, para que quando novos tipos de produtos sejam definidos, não precise alterar um código anterior. Entretanto, para casos com poucos produtos ou poucas modificações, pode-se introduzir sobrecarga desnecessária.

Ademais, conforme o artigo "*Factory Method*", do *Refactoring Guru*, "O código pode se tornar mais complicado, pois é necessário introduzir muitas subclasses novas para implementar o padrão. O melhor cenário é quando você introduz o padrão em uma hierarquia existente de classes de criadores." (Refactoring Guru, 2023).

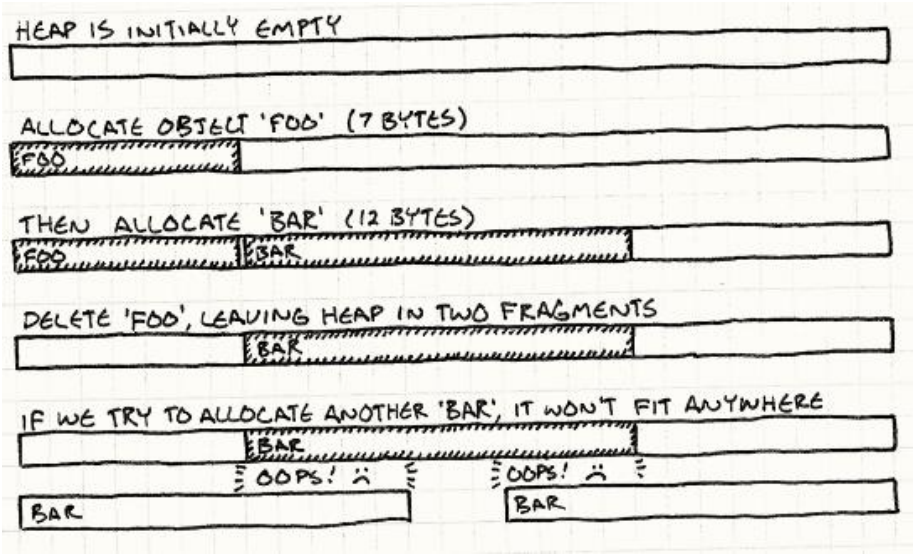
5.1.7. Padrão *Object Pool*

A performance e a otimização são assuntos recorrentes em muitos jogos, pois a otimização de um jogo pode garantir que jogos possam rodar em dispositivos menos potentes, bem como, usar ao máximo um dispositivo para rodar gráficos pesados e bonitos, por exemplo. Ou até mesmo um simples travamento rápido de jogo pode tornar a experiência do jogador frustrante. Esta ideia é reforçada por Nystrom (2014), o qual afirma que jogos são softwares críticos de desempenho.

Mesmo em linguagens com recursos de gerenciamento de memória, como o *Garbage Collector* na linguagem de programação C#, é preciso ter cuidado com a alocação e liberação de memória. Em conformidade com isto, a *Unity Technologies* afirma que ao instanciar um grande volume de objetos, tem-se o risco de causar pequenas pausas em um jogo, provocadas por um pico de coleções de lixo, feitos pelo *Garbage Collector* (Krogh-Jacobsen, 2022)

Em adição a isto, segundo *Krogh-Jacobsen (2022)*, picos de coleção do *Garbage Collector* normalmente estão acompanhados de número de criação e destruição de objetos devido à alocação e liberação de memória.

Figura 29 – Representação do processo de fragmentação de memória.



Fonte: (Nystrom, 2014).

E, nesse sentido, a fragmentação de memória interligada a este contexto, pois ao alocar e liberar espaço dinamicamente, pode-se deixar pedaços pequenos e vazios de memória espalhados na *heap*, coleção responsável memória. É possível visualizar o processo de fragmentação de memória na figura 29, o qual Nystrom (2014) mostra que depois de alocar e

desalocar objetos na memória, podem ficar pequenos espaços vazios na memória, os quais podem ser pequenos demais para serem aproveitados por objetos maiores, mesmo havendo espaço total liberado para eles. Deste modo, necessita redistribuir a memória ou aumentar seu tamanho total, o que pode gastar processamento ou memória desnecessariamente.

O padrão *Object Pool* ajuda a reduzir as chamadas do *Garbage Collector*, pois este reutiliza objetos já criados para evitar este processo de alocação e liberação de memória. A *Unity Technologies* corrobora isto ao afirmar que: "*Object Pooling* é uma forma de otimizar seus projetos e diminuir a carga que recai sobre a CPU ao criar e destruir rapidamente novos objetos. É uma boa prática e um padrão de projeto a se ter em mente para ajudar a aliviar o poder de processamento da CPU para lidar com tarefas mais importantes e não ser inundado por chamadas repetitivas de criação e destruição." (Unity, 2023).

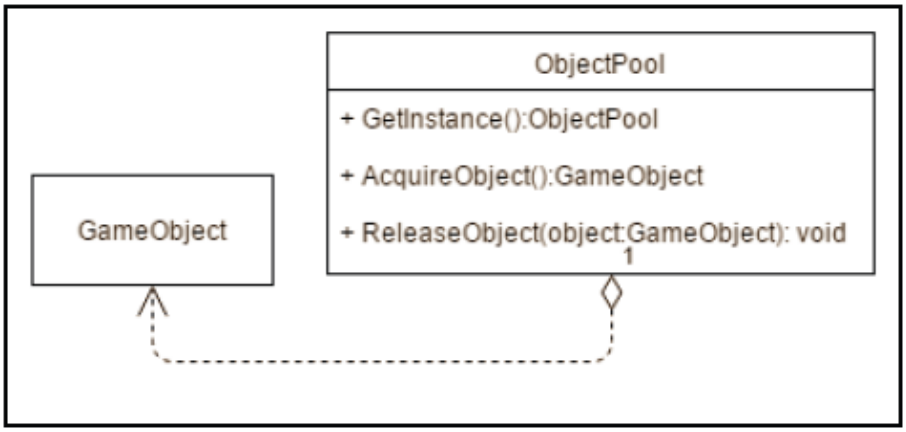
i. Definição

Diferentemente de outros padrões aqui mencionados neste trabalho, este é um padrão voltado a otimização. A *Unity Technologies* define este padrão como: "O *Object Pooling* é uma técnica de otimização para aliviar a CPU ao criar e destruir muitos *GameObjects*." (Unity 2023). Já Nystrom (2014), define este padrão como uma estratégia para melhorar o desempenho e o uso de memória reutilizando objetos de uma coleção (*Pool*) fixa, em vez de alocá-los e liberá-los individualmente.

De forma mais prática, o padrão *Object* Pool consiste em definir uma classe (*Pool*) que mantenha uma coleção de objetos reutilizáveis. De modo que quando solicitado, a *pool* reutiliza objetos previamente criados que estão disponíveis na coleção.

Cada objeto desta *pool* suporta uma consulta para saber se este está "ativo" no momento. Quando a pool é inicializada, ela cria toda a coleção de objetos antecipadamente e inicia todos para o estado desativado. Esta por sua vez, quando recebe uma requisição de objeto, procura um objeto disponível, inicializa-o como "ativo" e o retorna para quem solicitou. Quando o objeto não for mais necessário, este volta ao estado "desativado". Dessa forma, os objetos podem ser criados e destruídos livremente sem a necessidade de alocar memória ou outros recursos em tempo de execução.

Figura 30 – Representação UML do padrão *Object Pool*.



Fonte: Doran e Casanova (2017).

Na Figura 30, contém uma representação simples, em forma de diagrama, do padrão *Object Pool*. Pode-se visualizar que a Classe *ObjectPool* disponibiliza um método para um possível cliente requisitar instâncias de um objeto, neste caso um *GameObject*. Assim como também disponibiliza um método para liberar o objeto de volta para a pool. É válido ressaltar que estes métodos precisam respeitar a lógica anteriormente discutida, bem como, esta é uma representação simples deste padrão. Contudo, a depender do contexto, é possível este padrão ser implementado em conjunto com outros padrões para ficar mais abstraído, tal qual, facilitar acesso e garantir instância única, como o padrão *Factory* e o *Singleton*, respectivamente.

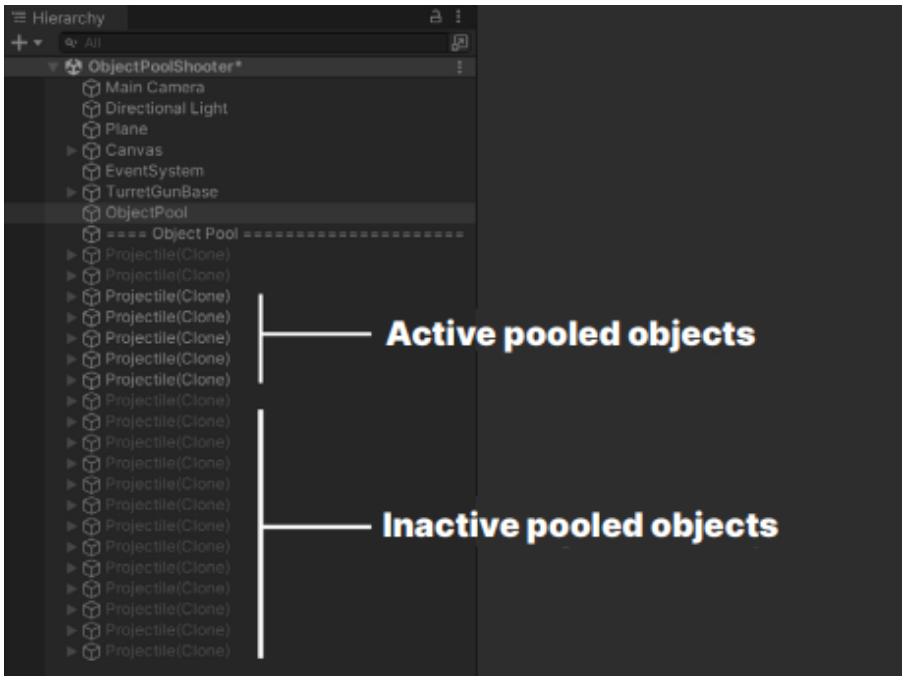
ii. Implementação

Esta seção abordará uma implementação exemplo do padrão *Object Pool* demonstrada no *e-book Level Up Your Code With Game Programming Patterns*, a qual utiliza de uma biblioteca da própria *Unity*, *UnityEngine*.Pool, que dispõe uma coleção genérica, adaptada ao contexto *Unity*, feita para simplificar o processo de criação de uma *pool*, bem como, dispor uma coleção otimizada baseada em pilha para manipular os objetos reutilizáveis. Em seguida, será demonstrado apenas um exemplo mais genérico com o intuito de apenas ilustrar uma possível implementação mais reutilizável e escalável.

A classe *RevisedGun* ilustrada no anexo M, representa a classe de uma arma, a qual cria instâncias de um projétil utilizando o padrão *Object Pool*. Este é um exemplo comum em jogos para se utilizar deste padrão, visto que em pouco tempo uma arma pode criar várias instâncias de um projétil, assim como, se dentro do jogo tiverem várias armas atirando ao mesmo tempo, pode-se causar uma sobrecarga ao instanciar e destruir diversos projéteis. Desta forma, faz-se

sentido utilizar deste padrão para reutilizar projéteis e evitar este cenário ao criar objetos repetidos. Na figura 31, consegue-se visualizar estes projéteis na hierarquia de cena da Unity, os quais, dentre eles, alguns estão ativos e outros desativados, esperando para serem utilizados.

Figura 31 – Hierarquia de cena da *Unity* ilustrando projéteis numa *pool* de objetos.



Fonte: Lin (2021).

Ainda neste exemplo, no anexo M, é perceptível que a classe utiliza de uma *pool* de projéteis, a qual utiliza da coleção *ObjectPool<T>* disponibilizada pela *Unity* na documentação (ref. 7.4). Mesmo tendo essa coleção pronta, ainda é necessário passar alguns *callbacks* no construtor da coleção para adicionar lógica a pool. Vejamos: (1) rotina para criar o objeto em questão ao inicializar; (2) rotina para executar ao receber um objeto; (3) rotina para executar ao liberar um objeto; (4) rotina caso seja necessário destruir um objeto. Além disso, outros parâmetros configuráveis como o tamanho inicial da *pool* e tamanho máximo da pool.

Entretanto, esta implementação não escala bem, pois não dá suporte para facilmente trocar o tipo de objeto ou disponibilizar formas de reutilizar a classe, bem como não depende de abstrações, além de que toda a lógica de criação está na própria classe da arma. Desta forma, é possível fazer melhorias no exemplo do anexo M para ficar de acordo os princípios *SOLID*, como combinar com o padrão *Factory* para tornar o padrão *Object Pool* mais escalável, manutenível e flexível. O anexo N demostra uma forma de realizar isto.

iii. Prós e Contras

Apesar deste padrão aumentar o desempenho, ao promover uma economia de recursos ao reutilizar objetos e, consequentemente, reduzir as interrupções do *Garbage Collector*, como também diminuir a fragmentação de memória, é importante medir o custo-benefício de eu uso, pois este introduz complexidade.

Nystrom (2014) reforça que é preciso gerenciar a memória de forma adequada ao contexto, pois a depender do tamanho da *pool*, é possível desperdiçar memória caso a *pool* seja maior do que o necessário, como também, em caso de ser menor do que o necessário pode se ter comportamentos indesejados ao chegar na capacidade máxima da *pool*, como não retornar um objeto, a depender da implementação.

Ademais, existem algumas estratégias para lidar com estes casos, como por exemplo, num cenário de uma *pool* de partículas, atingir a capacidade máxima e faltar uma partícula pode não fazer falta visualmente para um usuário. Já numa *pool* de áudios, é possível causar estranheza não tocar o som que é associado ao *feedback* de um inimigo, neste caso, pode-se desativar o último objeto da coleção da *pool* ou o som mais baixo e então reutilizá-lo. Como também, é possível instanciar objetos fora da *pool* para os casos extras e depois destruí-los quando não são mais necessários.

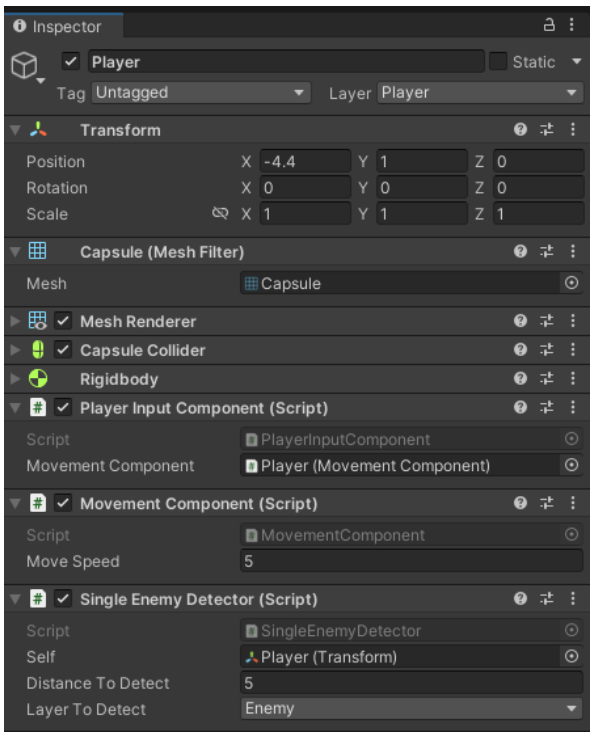
Vale ressaltar ainda que é imprescindível que os objetos requisitados, em algum momento, retornem para a *pool* quando estiverem inutilizados. Ao se esquecer disso pode ocasionar a falta de objetos na *pool*. Além disso, deve-se reciclar o objeto apropriadamente para não deixar resquícios da última utilização. Caso contrário, um objeto reutilizado pode ter algum mal funcionamento não previsto, como por exemplo, um projétil pode ser inicializado na posição de mundo errada.

5.1.8. Padrão *Component*

Conforme Nystrom (2014) aponta em seu livro *Game Programming Patterns* no capítulo sobre o padrão *Component*, diversos conceitos introdutórios de arquitetura de software nos dizem que diferentes domínios em um programa devem ser mantidos isolados uns dos outros, ou seja, domínios como física, renderização, inteligência artificial, som e outros, devem estar em classes separadas. Neste contexto, o padrão *Component* se encaixa muito bem, visto que é bastante útil para desacoplar classes de domínios diferentes, assim como, reduzir classes extremamente grandes e difíceis de trabalhar, separando-as em pequenos componentes independentes e reutilizáveis.

O padrão *Component* é bastante utilizado no mundo dos jogos, inclusive, é utilizando como base este padrão que a *Unity Engine* baseia sua arquitetura. Conforme descrito em sua documentação, um componente é a classe base para tudo anexado a um *GameObject*, a qual adiciona funcionalidade, e um *GameObject* é a classe base para todas as entidades em Unity Scenes (Unity, 2023). Um exemplo de uso é visível na figura 32, a qual ilustra um *GameObject* de um jogador (*Player*) com alguns componentes atrelados no inspector: *Player Input Component*, *Movement Component*, *Single Enemy Detector*, *Capsule Collider* e outros.

Figura 32 – Visualização do *GameObject Player* na janela de Inspector no editor da *Unity*.



Fonte: autoral.

i. Definição

Nystrom (2014) define o padrão *Component* como uma forma de permitir que uma única entidade abranja vários domínios sem acoplar os domínios entre si. Dito de outra forma, uma única entidade deve abranger múltiplos domínios, de maneira que o código para cada um desses domínios fique separado na sua própria classe, isto é, seu próprio componente. Desta forma, a entidade se torna essencialmente um recipiente para os diferentes componentes.

Utilizar componentes significa trabalhar com a composição de objetos (componentes) para compor objetos complexos. Para ilustrar esse conceito, pode-se pensar num objeto de jogo (entidade), que representa o personagem de um jogador, o qual pode ter vários componentes

atribuídos para definir seu comportamento, como um componente para responder *inputs*, outro componente para adicionar movimentação, colisão, renderização, animação, detectar inimigos, inventário etc.

Entretanto é válido ressaltar que apesar deste padrão reforçar comportamentos mais genéricos e reaproveitáveis, assim como, a separação de domínios, é inevitável haver interações entre alguns componentes. Segundo Nystrom (2014), componentes perfeitamente desacoplados que funcionam isoladamente são um bom ideal, mas não funcionam na prática. O fato de esses componentes fazerem parte do mesmo objeto implica que fazem parte de um todo maior e precisam ser coordenados. Isso significa comunicação.

Desta forma, alguns componentes precisam interagir com outros componentes, como por exemplo, um componente de detecção de inimigos provavelmente vai precisar interagir com um componente de colisão, entretanto, não há uma regra: isto pode ser feito por referenciação direta, compartilhando algum estado comum na entidade comum, utilizando de outros padrões como o *mediator* para intermediar mensagens entre classes etc. De acordo com Nystrom (2014), não há uma melhor resposta para isto e que, provavelmente, em um projeto real, acaba-se utilizando um pouco de cada forma, contudo, ele reforça que prefere sempre começar simples e ao longo do projeto melhorar a forma de comunicação quando surgir a necessidade.

Outro ponto relevante é que a separação de domínios em componentes diferentes permite o reuso destes em entidades completamente diferentes. A modo de exemplo, uma entidade que representa uma porta, assim como um inimigo, pode utilizar de um componente de vida. Embora cada um seja de natureza diferente, eles podem compartilhar de componentes iguais, bem como, podem adicionar ou remover componentes em tempo de execução para modificar seu comportamento.

Para complementar, Nystrom (2014) afirma que os componentes são basicamente *plug-and-play* para objetos. Eles nos permitem construir entidades complexas com comportamento rico, conectando diferentes objetos componentes reutilizáveis em soquetes da entidade.

ii. Implementação

No livro *Gaming programming patterns*, Nystrom mostra uma implementação simples para representar a ideia do padrão *Component*. Inicialmente, ele demonstra uma classe monolítica, a qual chama de Bjorn, esta por sua vez, está com bastante responsabilidade (Disponível no anexo O). O método *Update* desta classe roda todo frame e faz as seguintes

coisas: Verifica se o input do joystick está sendo movimentado, a partir disso, altera o vetor velocidade, verifica se há colisão após modificação, verifica se a sprite precisa ser alterada caso esteja andando numa outra direção, como também manda atualizar o gráfico. Tudo num mesmo método.

Desta forma, Nystrom (2014) propõe extrair uma interface em comum entre os diferentes domínios, que no caso é o próprio método *Update*, com diferentes parâmetros para cada caso (anexo P). Assim, ele separa o código em três classes que implementam a interface *Update*: *InputComponent*, responsável pela leitura de *input*; *PhysicsComponent*, responsável por realizar cálculos físicos; e o *GraphiciComponent*, responsável por fazer atualizações gráficas. Após essa refatoração, embora haja formas de melhorar o código-exemplo, o método na classe Bjorn ficou bem mais limpo, pois extraiu-se o código de diferentes domínios para classes separadas, as quais implementam uma interface, ou seja, são mais fáceis de serem trocadas por outros componentes que implementam a mesma interface, bem como, a classe Bjorn apenas repassa as chamadas para seus componentes, tomando uma forma de entidade.

De uma forma similar funciona o padrão *Component* na *Unity*, embora seja um pouco mais complexa, esta complexidade está bastante abstraída pelo próprio editor da *Unity*. De acordo com a Unity Technologies: “Para personalizar e adicionar componentes no Editor, você pode escrever seus próprios scripts. Para criar um componente com script, você precisa escrever o script e anexá-lo a um *GameObject*. Os scripts anexados a um *GameObject* aparecem na janela do Inspetor do *GameObject* porque o editor os trata como componentes integrados” (Unity, 2023).

Dessa forma, assim como visto na figura 32, pode-se anexar componentes a *GameObjects* (entidade), podendo utilizar de componentes previamente criados pela Unity ou implementar seus próprios scripts. Para isso, basta herdar da classe *MonoBehaviour*, a qual funciona uma classe base, definindo métodos comuns dos quais componentes usam para lidar com o *lifecycle* da *Unity*. Pode-se visualizar, no anexo Q, um exemplo de implementação de um componente de movimento que está anexado no *game object Player* ilustrado na figura 32. Resumidamente, este o script define uma movimentação básica utilizando de um componente built-in de física da *Unity*.

iii. Prós e Contras

Este padrão permite desacoplar o código em componentes separados, de forma que possam ser reutilizados em diferentes entidades, compondo o comportamento de uma entidade de forma customizada, bem como, permite a adição e remoção de componentes em tempo de

execução. Além disto, a criação de novos componentes, não necessariamente impactam na modificação de componentes pré-existentes. Ou seja, componentes dão uma grande flexibilidade alta para compor objetos complexos. Inclusive, esta abordagem tem se demonstrado bastante útil para jogos, visto que diversas *engines*, como a *Unity*, *Unreal* e outras, se baseiam neste padrão para arquitetar suas soluções, tal como afirmado por Suscheuski (2019).

Entretanto, de acordo com Nystrom (2014), é necessário também ter cuidado com a forma que esses componentes são referenciados, visto que uma alta complexidade de relacionamento entre componentes pode tornar o código mais desafiador, assim como, pode levar a um alto nível de indireção para obter componentes em tempo de execução e isto em loops internos com desempenhos críticos, pode levar a um desempenho ruim.

Um exemplo disto é utilizar o método GetComponent<T> para obter referência de um componente dentro de um método *Update()*, o qual, como mencionado anteriormente, roda em todo frame da aplicação. Isto será melhor detalhado na seção deste trabalho sobre boas e más práticas na Unity.

A utilização de componentes está diretamente ligada ao princípio de priorizar composição ao invés de herança, o qual foi anteriormente mencionado na seção sobre conceitos prévios. Portanto, é plausível sustentar que potencialmente, devido a independência de componentes, se obtém mais eficientemente comportamentos genéricos ao compor objetos utilizando componentes, bem como, tornam o código mais encapsulado e mais fácil de manter, conforme colocado por Marcelo no artigo "*Developing games with object composition: A case study using the Unity3D platfor*m" (Barbosa et al., 2015).

É válido ressaltar ainda que, em projetos grandes, a composição de componentes num sistema de jogo, como a Unity, pode levar a alguns problemas relacionados à alta dependência entre componentes ou instâncias de objetos de jogo, conforme apontado por Barbosa et al. (2015), entretanto, para este tipo de situação, pode-se utilizar do padrão de injeção de dependência na arquitetura do sistema, visto que este pretende fornecer uma maneira flexível de gerenciar associações indiretas entre componentes dependentes, eliminando a responsabilidade do programador de lidar de fazer estes vínculos de forma direta. (Barbosa et al., 2015).

5.1.9. Padrão *Decorator*

No mundo de jogos é bem comum existir situações nas quais é desejável adicionar pequenas funcionalidades extras ou pequenas modificações a um certo elemento do jogo. Isto

é bem comum em jogos os quais provém upgrades para o jogador, isto é, adicionar melhorias ou acessórios a um determinado elemento de jogo, como por exemplo uma arma, uma armadura, um carro, entre outros.

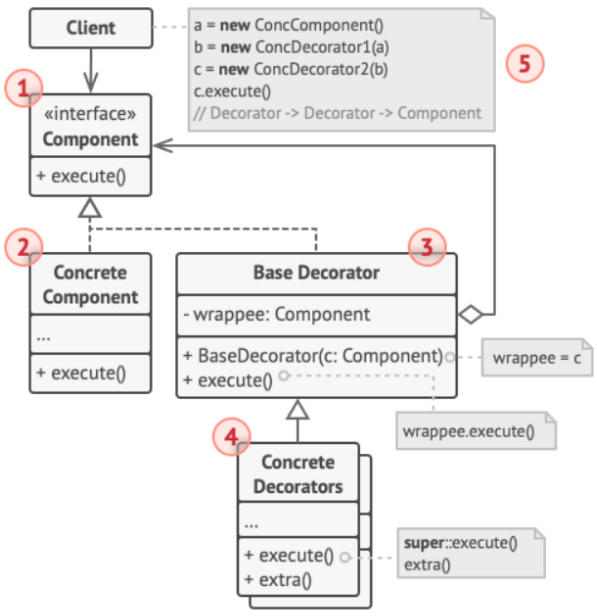
Um exemplo disto é o jogo *Archero*, o qual consiste em um jogo mobile do gênero arcade em que o jogador explora diversos calabouços enquanto vai matando monstros e ganhando upgrades em seu arco e flecha. A mecânica do jogo se baseia nessas melhorias, visto que suas flechas podem ficar mais rápidas, mais fortes, aplicar efeitos ao oponente, ricochetear em paredes etc.

Dito isto, uma maneira de implementar este tipo de comportamento é com o padrão *Decorator*, pois este provê uma forma de adicionar pequenas funcionalidades extras ou pequenas modificações (de forma dinâmica) a um objeto de modo que não altera a responsabilidade original dele.

i. Definição

O padrão *Decorator* também foi um dos padrões introduzidos pelo grupo *Gang of Four* no livro "*Design Patterns: Elements of Reusable Object-Oriented Software*", o qual o definem como um padrão estrutural que permite anexar responsabilidades adicionais a um objeto dinamicamente, utilizando de "decoradores". Estes fornecem uma alternativa flexível à subclasse para estender a funcionalidade (Gamma et al., 1994).

Figura 33 – Representação UML da estrutura do padrão *Decorator*.



Fonte: Dmitry Zhart (2023).

Na figura 33, é observável uma representação da estrutura, a qual está dividida em cinco partes: (1) A interface de um componente, a qual será implementada tanto pelo componente concreto, quanto pelo decorador; (2) A classe concreta de um componente, classe que define um comportamento básico de um componente e que pode ser decorada; (3) A classe base de decoração, esta é a chave do padrão visto que a classe contém uma referência para uma interface *Component* a qual usa para delegar as operações da interface de forma encapsulada; (4) Decoradores concretos, estes são responsáveis por definir comportamentos extras ao sobrescrever os métodos da classe base de decoração, porém, a priori, mantém as chamadas para a interface encapsulada; (5) O cliente, quaisquer classes que façam a composição de componente e decoração, bem como, utilize da interface (Refactoring guru, 2023).

ii. Implementação

Esta seção abordará um exemplo simples de implementação do padrão *Decorator* no ambiente *Unity* de desenvolvimento. O caso de a ser demonstrado aplica decorações a uma classe que representa uma flecha, adotando melhorias na produção destas, também denominadas como *buffs* ou *upgrades*. Ou seja, há uma determinada mistura com um sistema de *buffs* improvisado para o exemplo, porém este não deve ser o foco do exemplo em análise.

Nesse contexto, o referido exemplo consiste na definição de uma interface chamada de *IArrow* (anexo R) para representar uma flecha que, por questões de exemplo, a interface consiste em apenas métodos para retornar o dano causado pela flecha, bem como retornar a velocidade e a direção, juntamente com um método para configurar informações relacionadas ao arco que atira a flecha, como o dano do arco e a direção de disparo.

A partir disto, uma implementação concreta de flecha é feita, a classe *Arrow* (anexo S), que apenas implementa a interface anteriormente mencionada, definindo valores para os campos referidos e, também, permite ser serializável pela *Unity*.

De maneira similar, a classe *ArrowDecorator* (anexo T) - classe base para decorações - implementa a interface *IArrow*, no entanto, utiliza de uma referência do tipo *IArrow*, recebida via construtor, para repassar as chamadas da interface implementada para a referência recebida de forma encapsulada.

Com base nisso, é factível criar as classes decoradoras. Nos anexos U e V, é possível observar as classes *IronArrow* e *LighterArrow*, ambas herdam da classe *ArrowDecorator*, deste modo, para criar uma instância delas, é necessário passar uma instância de *IArrow* - uma classe

para ser decorada. Percebe-se também que, neste contexto, cada uma sobrescreve um método, ambas apenas adicionam um bônus nas propriedades de dano e velocidade, respectivamente. Isto é feito de modo que ainda se mantém uma chamada para a interface original, que está encapsulada via herança do *ArrowDecorator*, a fim de respeitar o comportamento da classe original, apenas adicionando uma modificação, responsabilidade ou efeito.

Tendo isso em vista, o padrão *Decorator* está implementado, entretanto, ainda é necessário mostrar a parte relacionada ao cliente, ou seja, a parte que consome este padrão. Todavia, para usufruir deste padrão na *Unity*, foi necessário fazer algumas adaptações devido ao fato dos componentes da *engine* necessitarem herdar de *MonoBehaviour* para fazerem parte de uma entidade de jogo, além do mais, não é possível instanciar uma classe *MonoBehaviour* via palavra-chave *new*.

Por isto, foi utilizada uma abordagem para separar a lógica de uma flecha atrelada ao *framework* da *Unity* (*lifecycle*, serialização e mensagens *Unity*) em uma classe (*ArrowBehaviour*) e a parte lógica de uma flecha em outra (Arrow). Esta abordagem é conhecida como *Humble Object*. Desta forma, a classe *ArrowBehaviour* lida com a lógica da Unity e repassa para a classe *Arrow*, a qual agora pode ser instanciada via palavra-chave *new*. Assim, é possível ter uma praticidade maior na utilização do padrão *Decorator*.

Dito isto, pode-se visualizar, no anexo W, a classe *ArrowBehaviour*, que lida com a parte de *lifecycle*, serialização e mensagens da *Unity*, como já mencionado, bem como, repassa as chamadas para a interface *IArrow*, que ela mesmo criou. Embora a própria classe pudesse ter instanciado decorações para a flecha, esta ainda não é a classe responsável por isto neste contexto.

No anexo X, é possível visualizar uma classe responsável por criar flechas, a *ArrowFactory*. Além de instanciar uma *ArrowBehaviour*, a classe recebe uma lista de *buffs* ativos (do tipo *IArrow*) e troca a instância interna de *ArrowBehaviour*, como se fosse uma linha de montagem. Por fim, a título apenas de ilustração, é permitido visualizar no anexo Y como a classe é decorada de fato, mostrando o método *ApplyBuff* (*IArrow buffReceiver*).

Diante do exposto, é válido ressaltar que as decorações feitas (*IronArrow* e *LighterArrow*) foram apenas ilustrações simples para aplicar decorações num determinado objeto. É importante mencionar, ainda, que elas poderiam ter sido mais elaboradas, pois, neste caso, foi utilizado de *buffs* apenas para exemplificar sua utilidade num contexto mais real de jogo, entretanto este padrão não se resume a apenas este cenário.

Em última análise, é importante pontuar que, devido ao padrão *Decorator* implementado neste contexto, tornou-se possível adicionar extensões ou modificações no

comportamento de uma flecha em tempo de execução, sem precisar alterar a classe *Arrow* ou a Classe *ArrowBehaviour*, como também não foi preciso utilizar de subclasses. Para isto, basta criar novas classes que herdam de *ArrowDecorator* que decoram a interface e, apenas neste exemplo, associar ao sistema de *buffs*.

iii. Prós e Contras

Conforme exposto neste tópico, o padrão em análise provê uma forma de estender o comportamento de um objeto sem necessariamente fazer subclasse, além de permitir adicionar e remover responsabilidades em tempo de execução. Dessa forma, é possível dividir uma classe em várias classes menores, de modo que, ao adicionar novos comportamentos, não haverá modificações em classes prévias, ou seja, segue-se em harmonia com os princípios da responsabilidade única e com o princípio aberto-fechado, consoante apontado por Adrian Bilescu (2023), em seu artigo "*Investing in Code Quality: The Decorator Pattern and Its Role in Implementing SOLID Principles*".

De acordo com Cuong Le (2016), em seu trabalho "*Design Patterns - Implementation in video game programming*", é recomendado o uso do padrão *Decorator* quando existe a necessidade de adicionar responsabilidades a objetos, de forma que não envolva outros componentes, como também, estas responsabilidades devem ser modificações leves, de maneira que o comportamento central do objeto permaneça o mesmo.

Em atenção a isso, ao se utilizar de um grande número de decoradores, possivelmente haverá uma sobrecarga de complexidade. Contudo, se bem utilizado, Cuong Le (2016) defende que este padrão provê uma flexibilidade aprimorada em comparação a subclasses, visto que evita heranças profundas e classes complicadas.

Por outro lado, existem pontos negativos, posto que, segundo Bilescu (2023), é possível que este padrão possa não ser cabível em toda e qualquer situação, uma vez que nem sempre é possível adicionar comportamentos que não dependam da ordem de composição, como também, quando for necessário fazer modificações internas de estado. Além disso, ao utilizar de muitos decoradores, tem-se a possibilidade de adicionar complexidade nas interações das classes, bem como, é tangível ter um alto nível de indireção, o que pode impactar em cenários críticos de performance.

5.2. Boas práticas na *Unity*

A *Unity* é muito popular entre os desenvolvedores de jogos, inclusive, uma de suas prováveis causas de sua popularidade, é a facilidade de acesso, visto que a *Unity* proporciona

um mecanismo fácil de trabalhar. Assim sendo, este capítulo se dedica a explorar as boas práticas identificadas na pesquisa previamente mencionada, no contexto da Unity, com o objetivo de fornecer um guia para nortear desenvolvedores.

i. Torne toda cena executável

De acordo com Tulleken (2016) e Juego (2021), no seu artigo "*7 Ways to Keep Unity Project Organized: Unity3d Best Practices*", é uma boa prática tornar toda cena executável a fim de evitar ter que trocar de cena para rodar o jogo e testar mais rapidamente.

Entretanto, segundo Tulleken (2016), isto pode ser complicado se existirem objetos que persistem entre carregamentos de cena. Ele afirma que uma das maneiras de fazer isso é ao utilizar de *Singletons* para objetos persistentes entre cenas que serão carregados quando não estiverem presentes na cena (Tulleken, 2016).

ii. Use *prefabs* frequentemente

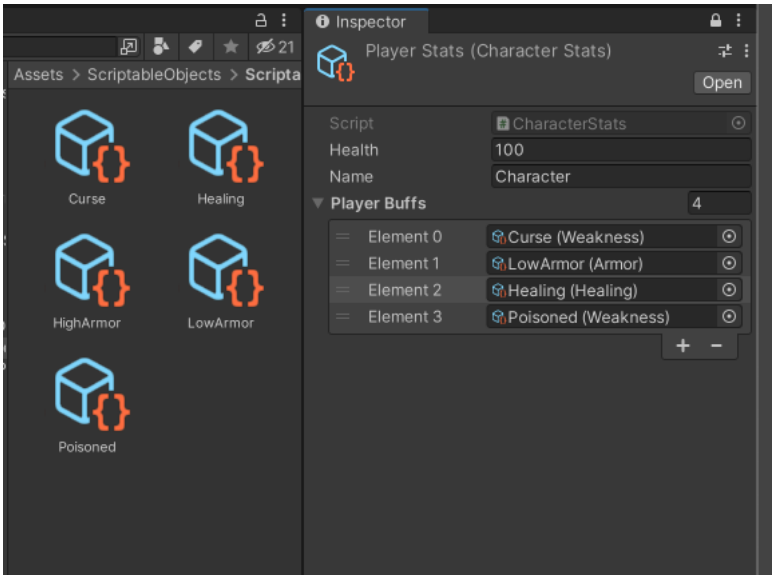
Vários dos recursos de estudo utilizados neste trabalho mencionam o uso de *prefab* de forma natural, visto que é um recurso bastante comum de ser utilizado na *Unity*, uma vez que este mecanismo, como mencionado anteriormente, é utilizado para criar objetos pré-configurados e reutilizáveis. Entretanto, pelo menos três destes materiais reforçam o uso deste mecanismo na *Unity* de forma frequente para facilitar a composição de cena (Tulleken, 2016; Bucher, 2017; Juego, 2021).

De acordo com Tulleken (2016), os únicos objetos de jogo em sua cena que não devem ser *prefabs* (ou parte de um *prefab*) são as pastas. Mesmo objetos usados apenas uma vez devem ser *prefabs*, uma vez que isso torna mais fácil fazer alterações já que torna o objeto isolado ao contexto da cena.

iii. Use *scriptable objects*

Conforme a *Unity Technologies* 2023) define, *ScriptableObject* é uma classe serializável da *Unity* que permite armazenar grandes quantidades de dados compartilhados independentemente de instâncias de *script*. A figura 34 demonstra a visualização de um *scriptable object* no editor da *Unity* através do *inspector*.

Figura 34 – Visualização de *scriptable objects* no editor.



Fonte: Anuj Shrestha (2022).

A *Unity Technologies* (2023) incentiva o uso de *scriptable objects* para armazenar valores ou para configurar objetos, ao invés de utilizar *MonoBehaviours* neste propósito, pois previne duplicações de dados, visto que as configurações podem ser reutilizadas em outros contextos.

Em consonância com isso, Tulleken (2016) em seu artigo "*50 Tips and Best Practices for Unity (2016 Edition)*" recomenda o uso de *scriptable objects* na *Unity* em vários cenários, como para guardar informações de *level*, para configurar objetos no inspector e para especializar *prefabs*.

Além disso, *scriptable objects* tornam o editor mais versátil, podendo servir de objeto intermediário para conectar componentes, além de poder ser utilizado de diversos modos. Um exemplo disto é que os padrões de projeto *Command* e *Observer* podem ser implementados em conjunto com *scriptable objects* para prover uma forma mais conveniente de plugar comandos ou eventos (*observables*) com o editor (Unity Technologies, 2023).

Inclusive, estas formas diversas são mais bem detalhadas no *e-book "Create modular game architecture in Unity with ScriptableObjects*", o qual mostra formas de modificar a arquitetura de seu jogo para incluir *scriptable objects* a fim de tornar a arquitetura mais flexível e modular. Este seria o próximo padrão a ser abordado na seção de padrões de projeto, entretanto devido ao corte necessário para definir o escopo deste trabalho - o qual foi mencionado anteriormente - não foi possível cobrir, visto que este padrão teve apenas três menções.

iv. Utilize o *profiler* para analisar possíveis problemas de performance

O *profiler* é uma ferramenta que você pode usar para obter informações de desempenho sobre seu jogo na *Unity*. É possível executá-lo no Editor para obter uma visão geral da alocação de recursos enquanto desenvolve seu aplicativo, como também, pode conectá-lo a dispositivos em sua rede ou dispositivos conectados à sua máquina para testar como seu aplicativo é executado na plataforma de lançamento pretendida.

Este reúne e exibe dados sobre o desempenho do seu aplicativo em áreas como CPU, memória, renderizador e áudio. É uma ferramenta útil para identificar áreas de melhoria de desempenho em seu aplicativo e iterar nessas áreas. É possível identificar coisas como como seu código, ativos, configurações de cena, renderização de câmera e configurações de construção afetam o desempenho de seu aplicativo. Ele exibe os resultados em uma série de gráficos, para que você possa visualizar onde ocorrem os picos de desempenho do seu aplicativo (Unity Technologies, 2023).

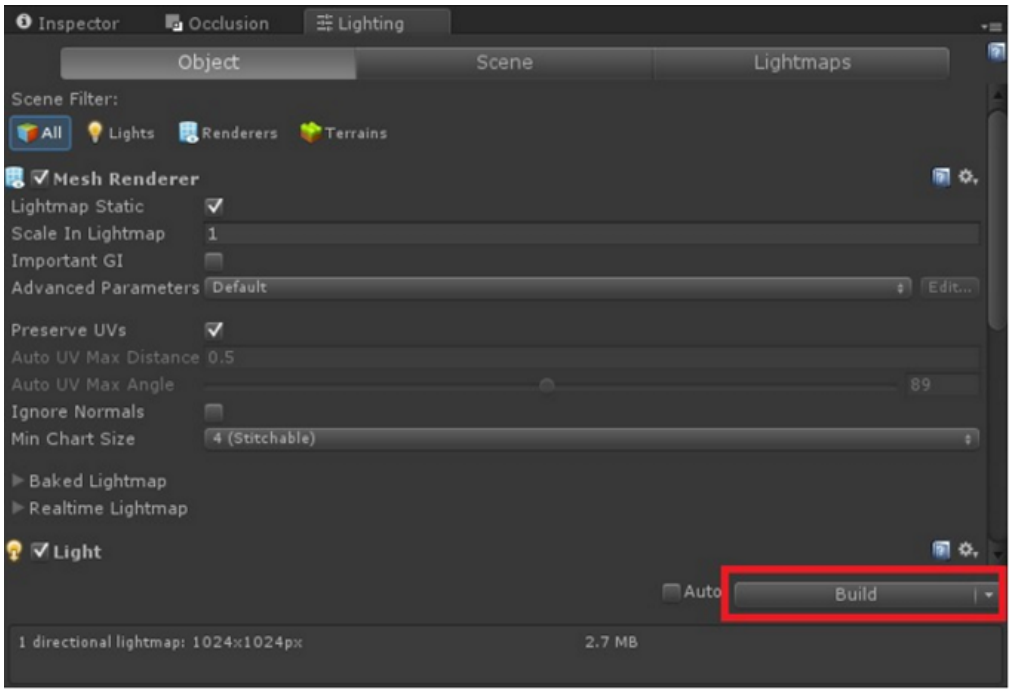
Dito isto, diversas fontes encontradas na pesquisa incentivam o uso desta ferramenta para análise (Kundurthy, 2016; Tulleken, 2016; Blåfield, 2021) como também, Unity (2023) também reforça ao afirmar: "O *Unity Profiler* fornece informações de desempenho sobre seu aplicativo, mas não poderá ajudá-lo se você não o usar. Crie um perfil do seu projeto no início do desenvolvimento, não apenas quando estiver próximo da entrega. Investigue falhas ou picos assim que eles aparecerem".

v. Revise as configurações de qualidade e otimização

Ao compor uma cena na *Unity*, possivelmente existem objetos que não precisam estar rodando na qualidade máxima ou não precisam de soluções complexas e custosas, as quais podem ser simplificadas. A falta desse tipo de ajuste pode estar desnecessariamente gastando processamento ou memória do seu jogo e estas configurações impróprias ou inadequadas, também é considerada uma má prática para alguns desenvolvedores, conforme Borelli et al. (2020). Dito isto, Kundunrthy (2016) recomenda a revisão de algumas configurações, como o uso de *light mapping*, *occlusion culling*, *level of detail* (LOD), *batching* e *atlas texture*.

Light mapping é uma técnica que utiliza dados de luz previamente calculados, os quais são armazenados em uma cache, desta forma os dados são apenas acessados em tempo de execução ao invés de calculados em tempo de execução, trazendo melhorias impactantes se comparadas ao uso de luz em tempo real (Kundunrthy, 2016).

Figura 35 – Visualização das configurações de luz, a qual indica o uso de light mapping e indica como pré-calcular os dados de luz.



Fonte: Praveen Kundurthy (2016).

Occlusion culling é um recurso disponível na *Unity* para otimizar a renderização de objetos que não estão sendo vistos pela câmera ou quando há objetos obstruindo a visibilidade, desativando sua renderização, desta forma, economiza *drawcalls*, ou seja, reduz o processamento gráfico, bem como, o uso de memória (Kundunrthy, 2016).

Level of detail (LOD), é um recurso disponível na *Unity* para trocar objetos que estão muito distante da câmera para objetos mais simples, como por exemplo, uma árvore que está distante não precisa ser renderizada ou ter os vértices com a mesma qualidade que uma árvore próxima a câmera. Deste modo, a *Unity* permite configurar níveis de detalhe para cada objeto, de forma que quando atinja o limiar, ela automaticamente substitua por objetos mais simples e, de forma análoga, o inverso. Ao aplicar este tipo de configuração pode reduzir a sobrecarga que uma cena pode ter (Kundunrthy, 2016).

Batching (lote) combina objetos do jogo em uma única *draw call*. Você obtém os melhores benefícios do processamento em lote ao planejar quais objetos serão agrupados em lote numa única *drawcall*. A *Unity*, para os materiais iguais, automaticamente aplica o agrupamento em lote, entretanto para alguns objetos é necessário atribuir manualmente se é um *batching* estático ou dinâmico, ou seja, objetos que são estáticos ou que se movem, respectivamente. Neste mesmo contexto, Kundunrthy (2016) ainda aponta que pode-se utilizar

de *atlas texture*, uma forma de combinar diversas texturas em uma única textura compactada e otimizada que reduz o número de *draw calls* ao agrupar no mesmo lote.

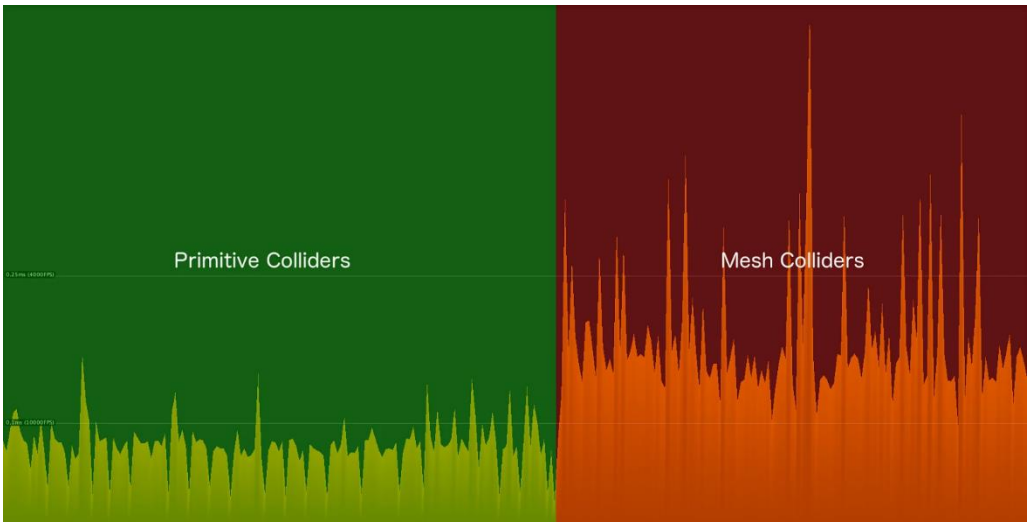
Outro ponto válido para se atentar, segundo (Yin, ref. 18), é o tamanho das texturas (e imagens) importadas no projeto, visto que possivelmente estas não precisam estarem sendo renderizadas na qualidade máxima e a *Unity* oferece diferentes algoritmos de compressão para redimensionar as imagens de forma mais apropriada.

Somado a isso, Kundunrthy (2016) recomenda também habilitar a opção de *mipmapping*, a qual reduz a resolução da imagem caso a imagem esteja distante da câmera.

Por fim, Borelli (2020) e Aguiar (2023) reforçam a revisão da escolha dos *colliders*, componentes de colisão, visto que componentes complexos de colisão usam de mais recursos computacionais do que componentes simples e primitivos. Desta forma, é uma boa prática trocar os *colliders* complexos por *colliders* mais simples, se for possível realizar a troca sem impactar negativamente no *game design* do jogo.

Para complementar, pode-se visualizar na figura 36, a fim de ilustrar a diferença, a qual consiste num experimento realizado por Aguiar (2023).

Figura 36 – Ilustração da diferença entre o uso de *colliders* primitivos e *colliders* complexos, como o *mesh collider*.



Fonte: Aguiar (2023).

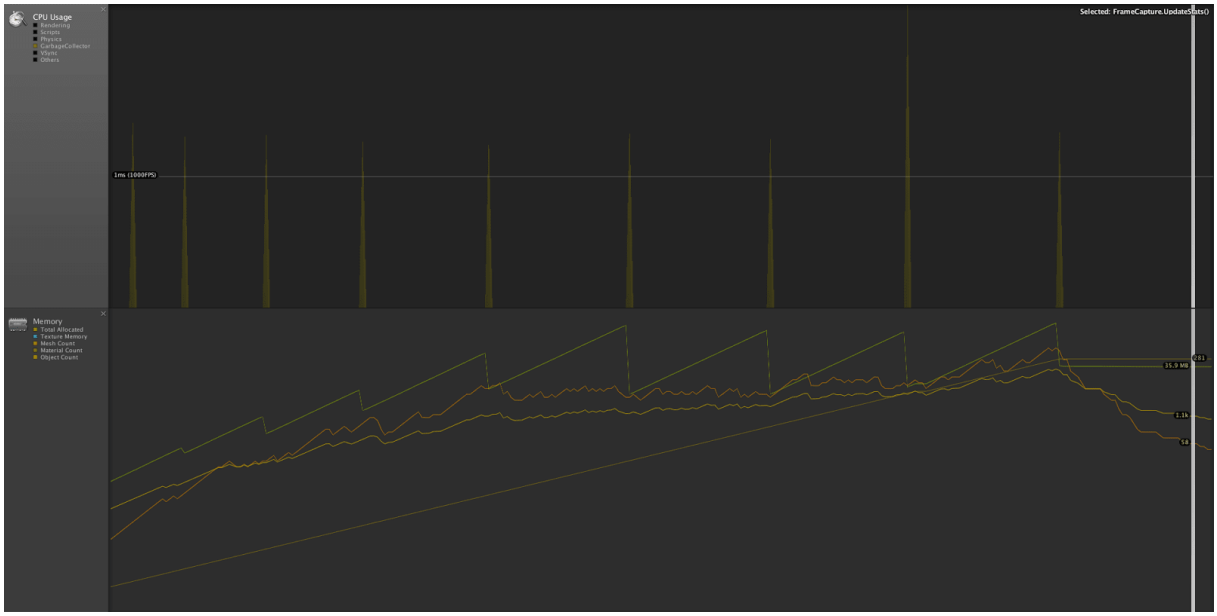
vi. Considere utilizar do padrão *object pool* ao invés de criar objetos dinamicamente

É importante considerar o uso de *object pool* quando estiver instanciando objetos dinamicamente, pois o uso deste padrão, pois apesar de introduzir uma complexidade, traz

vantagens em relação a fragmentação de memória e a diminuição de esforço do *Garbage Collector*, conforme discutido na seção padrão *object pool* e pontuado por Aguiar (2023), assim como, na documentação da *Unity (*Unity Technologies, ref. 11). Além disso, o *Garbage Collector* fica mais lento à medida que o uso de memória aumenta, visto que tem mais memória para escanear e liberar dados não utilizados.

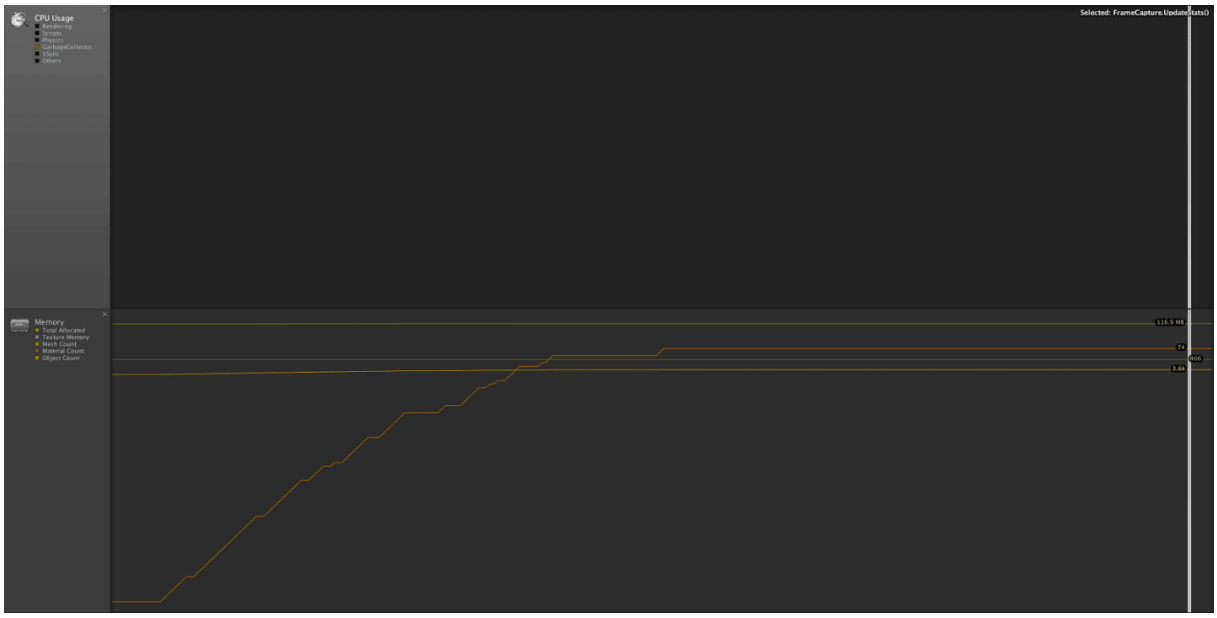
De forma complementar, pode-se visualizar - nas figuras 37 e 38 - uma comparação, com relação ao não uso e ao uso de *object pool*, respectivamente, do *profiler* da *Unity* - ferramenta para analisar diversos aspectos do jogo como consumo de memória, cpu, *Garbage Collector* etc. Na parte superior da figura 37, é possível observar picos de atuação do *Garbage Collecto*r, bem como, é possível observar na parte inferior a alocação dinâmica de memória, mostrando diversas variações ao longo do tempo ao alocar a mesma. Em contrapartida, na parte superior da figura 38, é possível observar que não há chamadas ao *Garbage Collector*, bem como a alocação de memória se mantém estável.

Figura 37 – Visualização do *profiler* da *Unity*, a qual demonstra as alocações para o *Garbage Collector* (na parte superior) e alocação de memória (na parte inferior).



Fonte: Aguiar (2023).

Figura 38 – Visualização do *profiler* da *Unity*, a qual demonstra as alocações para o *Garbage Collector* (na parte superior) e alocação de memória (na parte inferior).



Fonte: Aguiar (2023).

vii. Realize cache de componentes e objetos

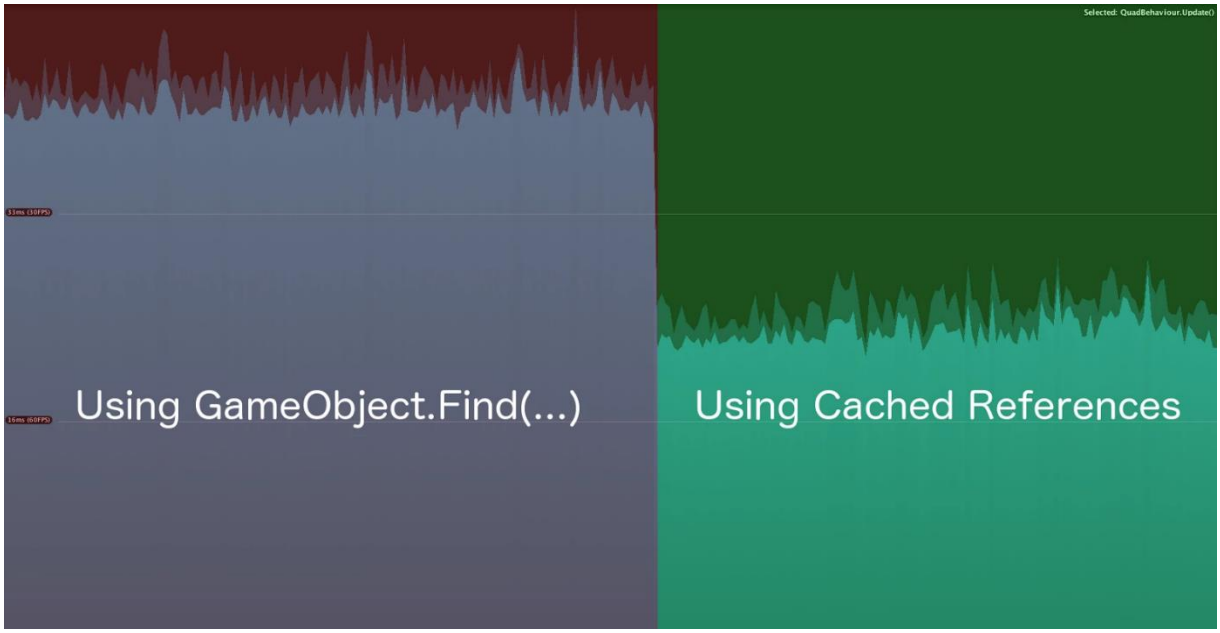
A *Unity* tem uma arquitetura baseada na composição de componentes em objetos de jogo e é normal componentes precisarem acessar funcionalidades de outros componentes, visto que é necessário separar os domínios para aumentar o reuso destes, conforme discutido na seção sobre o padrão *Component*. Contudo, é importante guardar as referências destes componentes (caso haja reuso na classe), pois realizar operações de busca toda vez que precisar utilizar de uma funcionalidade de um componente terceiro desperdiça esforço da CPU. O mesmo se aplica para objetos.

Aguiar (2023) faz uma comparação ao realizar e não realizar cache de componentes e objetos, enfatizando a relevância de realizar cache, bem como, pontua a importância de evitar algumas chamadas *built-in*, como *Transform.position* que por baixo dos panos realiza uma operação de *GetComponent()* para procurar o componente *Transform*, responsável por guardar dados de posição, rotação e escala do objeto de jogo.

O mesmo autor menciona um teste realizado que move 800 caixas, as quais se movem alterando a posição dos objetos no *Update()* via *Transform.position*, em um caso é realizado cache e no outro não (Aguiar, 2023). Deste modo, ao comparar, percebe-se uma queda da média de tempo gasto em um *script* de 30 ms para 23 ms. Ainda no mesmo artigo, ele faz também

uma comparação com referência de objetos e mostra uma média de diferença de 41 ms para 23 ms (Aguiar, 2023), a qual pode ser visualizada na figura 39.

Figura 39 – Visualização do *profiler* da *Unity*, a qual demonstra o tempo gasto em *scripting* ao realizar e não realizar cache de referências de componente.



Fonte: Aguiar (2023).

viii. Cuidado ao manusear *Materials*

A *Unity* compartilha materiais entre os objetos a fim de economizar memória e apenas limpa os materiais que a própria IDE criou ou quando troca de cena, de acordo com a documentação oficial (Unity, 2023).

Entretanto, segundo a *thread* do Reddit "*What are some bad practices to avoid when using Unity*?" isto não é muito evidente e normalmente é uma fonte de *memory leak*, visto que chamadas para *Renderer.material* - forma de adquirir uma referência do material de um objeto - cria uma cópia do material compartilhado, a qual não é automaticamente limpa, desta forma, é necessário destruir o material ao destruir um objeto.

5.3. Más práticas na Unity

De acordo com Ibrahim (2023) - *Tech lead* com anos de experiência em *Unity*, mesmo que existam diversos recursos facilitadores - principalmente para prototipar - para fazer projetos

maiores que vão para produção, algumas destas práticas facilitadoras podem rapidamente causar desordem no código.

Desta forma, ao embarcar no mundo de desenvolvimento de jogos com a *Unity*, é fundamental conhecer não apenas as melhores práticas, mas também as armadilhas que podem surgir ao longo do caminho, a fim de evitar práticas que possam prejudicar a eficiência do desenvolvimento, como também, a qualidade do jogo desenvolvido.

i. Falta de separação de conceitos

Um script *MonoBehaviour* que implementa, ao mesmo tempo, diferentes responsabilidades, dificulta a evolução dos projetos, visto que não há a separação dos conceitos de forma efetiva.

Mesmo a *Unity* sendo baseada numa arquitetura de componentes (Padrão *Component*), conforme explicado anteriormente, a qual tenta separar diferentes domínios em componentes isolados de modo que possam ser reaproveitados para compor entidades diferentes, cabe ao programador utilizar do princípio da responsabilidade única para quebrar classes grandes em classes menores de forma coesa. Ao criar classes que não separem bem seus conceitos, diversos módulos podem ficar acoplados, de forma que fique difícil reutilizar o mesmo componente para outros contextos. Quando as responsabilidades estão bem partidas, naturalmente haverá uma maior reusabilidade dos componentes.

Segundo Borelli et al. (2020) , um erro comum é criar uma classe associada ao *Player* (jogador) que implementa toda lógica, como ler *inputs*, determinar o estado do jogador, mover o jogador, entre outras. Existem formas de contornar estes acoplamentos, utilizando do padrão *Observer*, *Command*, *State* e outros, como abordado na seção de padrões de projeto.

ii. Acoplamento de objetos via Inspector

Conforme apontado previamente, na *Unity* é possível acoplar *scripts* (*MonoBehaviour*) em outros objetos via editor, utilizando a ferramenta chamada de *Inspector*. *MonoBehaviours* dos quais utilizam de variáveis públicas ou privadas com o atributo [*SerializeField*] permitem que o editor crie um campo para associação de valores ou referências de outros objetos, apenas arrastando ou configurando valores.

Esta prática embora seja bastante útil para alguns contextos, quando mal utilizada, pode levar a problemas de manutenção: (1) Para o código, não há visibilidade do acoplamento, bem como, não há valor atribuído. Só é possível visualizar via *Inspector* ou ao quebrar em tempo de execução. Por isso é necessário adicionar checagens para verificar se o valor foi carregado

apropriadamente (é possível avisar, antes de executar utilizando realizando sobrecarga no método *OnValidate()* que roda apenas no editor); (2) Se houver refatorações de nome de classe, as referências podem ser perdidas, sendo necessário reatribuir manualmente.

Contudo, este é um tema polêmico pois existem outros desenvolvedores que defendem o uso desta prática, visto que seu custo-benefício pode ser melhor que outras práticas (uso do *GameObject.Find()* ou sistema de mensagens da *Unity*) em termos de performance, bem como, é bastante conveniente. Além disso, ao utilizar com o sistema de *prefabs* ou *script objects* da *Unity*, permite trocar de objeto, implementação, entre outras coisas, apenas com um simples arrastar de objetos no *Inspector*, como também, as informações ficam salvas num arquivo separado reutilizável (*.prefab* ou *.asset*). Em uma pesquisa realizada por Borelli et al. (2020), retornou-se 31% de respostas positivas, 34% negativas e 35% neutras quanto a esta prática, devido aos fatores anteriormente mencionados.

iii. Dependência de componente não explícita

As classes do tipo *MonoBehaviour* não permitem construtores, ou seja, quando há dependência, não dá para explicitar via construtor, algo bastante usual na programação orientada a objetos. Desta forma, é comum acoplar objetos via editor ou procurar por referências no método *Awake()* que é chamado ao carregar o componente.

Ao procurar por referências, é comum utilizar do método *GetComponent()* para retornar uma referência de um componente, visto que esta forma é mais performática pois não percorre toda hierarquia da cena, apenas percorre os componentes atrelados a entidade correspondente.

Entretanto, precisa que o programador tenha previamente configurado a entidade da qual o componente está anexado, caso contrário, pode-se retornar um componente não esperado ou *null*, causando potenciais *bugs*. Isto pode ser evidenciado pela *thread* do Reddit "*What are some bad practices to avoid when using Unity?*", como também, é mencionado por Tulleken (2016).

Uma forma de explicitar esta dependência é utilizar do atributo [*RequireComponent()*], pois este atributo avisa ao editor que ali há uma dependência e que se não tiver atribuída, no momento que for adicionado o componente em questão, a dependência também é adicionada. É válido ressaltar que a própria *Unity* em sua documentação comenta que o uso deste atributo pode ser útil para evitar erros de configuração no componente (Lin, 2021).

iv. Chamadas exaustivas em contexto crítico de performance

Existem alguns recursos na *Unity* que facilitam determinadas coisas para o programador, entretanto, alguns destes recursos, embora práticos são custosos (como por exemplo, a referência estática para a câmera principal, *Camera.main*), e podem ser ainda mais perigosos se executados num contexto crítico, como por exemplo o método *Update()* que roda a todo frame da aplicação.

Por isso, é importante evitar chamadas exaustivas em contextos críticos. Se o intuito for utilizar dessas chamadas, é melhor chamar num contexto mais seguro, que execute apenas uma vez, como por exemplo o método *Awake()* que executa uma única vez ao carregar um componente (no mesmo exemplo de *Camera.main*, poderia guardar a referência, ao invés de utilizar esta chamada em todo corpo do código). No entanto, o ideal é utilizar de recursos mais otimizados, se possível que não sejam exaustivos, mas nem sempre é possível, vide método *Instantiate()*, única forma de instanciar *GameObjects*.

Ellis (2019) define uma lista de chamadas exaustivas que devem ser evitadas em contexto crítico como: métodos relacionados a *GetComponent*, *FindObjectOfType* e *AddComponent*; Métodos baseados em *strings* como *Invoke*, *SendMessage* e *Find*; Camera.main e comparações com *null* para objetos do tipo Unity.Object.

Além disso, é válido ressaltar que a instanciação e destruição de objetos via *Instatiate()* e *Destroy()*, podem ocasionar pausas para o *Garbage Collector* entrar em ação, bem como, são consideradas chamadas custosas, visto que os objetos de jogo podem ter diversos componentes pesados, como por exemplo, componentes gráficos e malhas de colisão. Desta forma, é boa prática evitar estas chamadas em contextos críticos e a própria *Unity Technologies* reforça o uso do padrão *object pool* para estes casos (Unity, 2023).

De forma complementar, a pesquisa de Borelli et al. (2020), reforça as afirmações anteriores, bem como, também diz que cálculos pesados de física não devem ser executados durante o método *Update()*, seria mais apropriado de ser executado no *FixedUpdate()*.

v. Estratégia de temporização frágil

De acordo com Borelli et al. (2020) um erro típico é a atualização do objeto do jogo em uma atualização baseada em quadros - por exemplo, um movimento fixo é realizado a cada quadro - tornando a velocidade da animação dependente da taxa de quadros e, portanto, variando em dispositivos diferentes, ou no mesmo dispositivo em contexto diferente.

Em outras palavras, quando um desenvolvedor provê uma movimentação via método *Update()*, o qual roda em todo quadro da aplicação, mas não multiplica pela diferença de tempo entre os quadros, significa que a movimentação fica dependente de quadros e não do tempo.

Desta forma, dispositivos com diferentes FPS (frames por segundo), atualizam a movimentação diferentemente.

Como mencionado, uma forma simples é multiplicar pela diferença de tempo, porém também pode-se vincular o cálculo a um método como o *FixedUpdate()* que sempre é executado na mesma diferença de tempo de forma fixa, isto é, não haveria variações entre os quadros.

vi. Uso do *any state* no componente *AnimationController*

Conforme mencionado por Nardone et al. (2023), o uso da transição de estado *any* no componente de animação da *Unity*, ou seja, o uso de uma transição que pode vir de qualquer estado dado uma determinada condição em uma máquina de estados na animação, é considerada por alguns desenvolvedores como uma má prática, visto que pode ocasionar comportamentos inesperados, pois é realizada a transição para o estado independente de qual estado está, inclusive, se mal configurado, pode transicionar para o próprio estado vinculado ao *any*.

vii. Atribuir diretamente a velocidade do objeto e sobrescrever força

Na *Unity* existe um componente chamado *Rigidbody*, o qual lida os cálculos de física da *engine*, ou seja, este é o componente que calcula gravidade, velocidade, forças aplicadas e etc (Unity Technologies, 2023).

Dito isto, ao utilizar deste componente para aplicar cálculos físicos num objeto de jogo e alterar diretamente sua velocidade, ao invés de aplicar força diretamente, pode ser considerado uma má prática para alguns desenvolvedores de acordo com Borelli et al (2020) (a não ser que seja intencional), pois alterar a velocidade diretamente implica em sobrescrever as forças as quais estão sendo aplicadas no objeto no momento. Isto pode causar comportamentos inesperados, como arremessos inesperados, atravessar paredes, entre outros.

6. Conclusão

Em última análise, é essencial enfatizar a importância da *Unity* como uma das ferramentas mais populares no desenvolvimento de jogos digitais. Isso se deve a uma série de motivos, como sua licença acessível, suporte para várias plataformas, uma comunidade ativa, uma variedade de recursos que simplificam o processo de criação de jogos, entre outros motivos.

Tal destaque é ainda mais notável no contexto brasileiro, no qual o setor de jogos tem experimentado um crescimento significativo, como também a ampla adoção desta *engine* como

uma das principais ferramentas utilizadas na produção de jogos no país, segundo a pesquisa realizada pela *AbraGames*, consoante demonstrado neste trabalho.

No entanto, como mencionado ao longo deste estudo, o desenvolvimento de jogos é uma tarefa complexa e multidisciplinar, que envolve diversos elementos interdependentes. Um dos principais desafios é manter o código do jogo eficiente, organizado e sustentável ao longo do tempo. A falta de estrutura adequada pode resultar no temido "código espaguete", dificultando a manutenção e evolução do projeto.

Nesse contexto, os padrões de projeto desempenham um papel fundamental no combate à falta de modularização, fornecendo soluções comprovadas aos problemas recorrentes de código e permitindo que os desenvolvedores criem sistemas mais flexíveis, escaláveis e de fácil manutenção.

Assim sendo, conduziu-se uma revisão da literatura de forma sistemática, a fim de obter tanto a perspectiva de acadêmicos quanto a perspectiva de indivíduos que utilizam a *Unity* no cotidiano para obter os padrões de projeto mais recomendados para esta, assim como as boas e as más práticas que circundam este entorno.

Em virtude de todo o exposto, esta monografia contribui para a disseminação de diretrizes que auxiliam os desenvolvedores de jogos - os quais utilizam a plataforma *Unity* - oferecendo orientações práticas e formas de implementar soluções mais eficazes, tornando o desenvolvedor mais apto a identificar condições problemáticas, incluindo armadilhas comuns, além de aprimorar o julgamento crítico de quando estas soluções são apropriadas, visto que elas podem introduzir complexidade.

Consequentemente, com esses conhecimentos em mãos, os profissionais da área têm a oportunidade de elevar a qualidade de seus jogos, contribuindo para o contínuo crescimento e sucesso da indústria de jogos digitais no Brasil.

7. Referências

AKHTAR, S. Implementing a Command Design Pattern in Unity. Disponível em: https://faramira.com/implementing-a-command-design-pattern-in-unity/. Acesso em: 02, ago., 2023.

AGUIAR, R. Unity 3D Best Practices: Physics. 2023. Disponível em: https://x-team.com/blog/unity-3d-best-practices-physics/. Acesso em: 20, ago., 2023.

AMAT, C. Everything You Need to Know About Singletons in Unity. 2020. Disponível em: https://www.youtube.com/watch?v=mpM0C6quQjs. Acesso em: 19, ago., 2023.

BARBOSA, M. B.; RÊGO, A. B.; MEDEIROS, I. Developing games with object composition: A case study using the Unity3D platform. **Computing Track – Short Papers**, 2015.

BILESCU, A. Investing in Code Quality: The Decorator Pattern and Its Role in Implementing SOLID Principles. Disponível em: https://www.codementor.io/@adrianbilescu/investing-in-code-quality-the-decorator-pattern-and-its-role-in-implementing-solid-principles-24jb2i9ghf. Acesso em: 05, set., 2023.

BLÅFIELD, J. Optimizing mobile games in a Unity environment. 2021. 35 páginas. Monografia (Curso de Information and Communications Technology) - JAMK University of Applied Sciences, Jyväskylä.

BORELLI, A.; NARDONE, V.; LUCCA, G. A.; CANFORA, G. PENTA, M. D. Detecting Video Game-Specific Bad Smells in Unity Projects. **MSR '20: Proceedings of the 17th International Conference on Mining Software Repositories**, p. 198 – 208, 2020.

BUCHER, N. Introducing Design Patterns and Best Practices in Unity. 2017. Disponível em: https://dl.acm.org/doi/10.1145/3077286.3077322. Acesso em: 08, ago., 2023.

CARTAXO, B.; PINTO, G.; SOARES, S. Rapid Reviews in Software Engineering. Contemporary Empirical Methods in Software Engineering. **Springer**, p. 356 – 383, 2020.

DEALESSANDRI, M. What is the best game engine: is Unity right for you? Disponível em: https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you. Acesso em: 19, jun., 2023.

DILLET, R. Unity CEO says half of all games are built on Unity. 2018. Disponível em: https://techcrunch.com/2018/09/05/unity-ceo-says-half-of-all-games-are-built-on-unity/. Acesso em: 22, set., 2023.

DRAKE, J. 19 Great Games That Use The Unity Game Engine. Disponível em: https://www.thegamer.com/unity-game-engine-great-games. Acesso em: 19, jun., 2023.

DORAN, J. P.; CASANOVA, M. **Game development patterns and best practices: better games, less hassle**. Birmingham, Uk: Packt Publishing Ltd, 2017.

ELLIS, M. Unity Performance Best Practices with Rider, Part 1. 2019. Disponível em: https://blog.jetbrains.com/dotnet/2019/02/21/performance-indicators-unity-code-rider/. Acesso em: 02, set., 2023.

FORTIM, I. Pesquisa da indústria brasileira de games 2022. ABRAGAMES: São Paulo, pp. 68, 2022. Disponível em: https://www.abragames.org/pesquisa-da-industria-brasileira-de-games.html. Acesso em: 18, jun., 2023.

GALACH, P. How to implement State Machine in Unity. Disponível em: https://www.patrykgalach.com/2019/03/18/design-pattern-state-machine/. Acesso em: 01, set., 2023.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Design Patterns: Abstraction and Reuse of Object-Oriented Design. **Lecture Notes in Computer Science**, vol 707, p. 406 – 431, 1993.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns: elements of reusable object-oriented software**. Boston: Addison-Wesley, 1994.

GAROUSI, V.; FELDERER, M.; MANTYLA, M. V. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering **Information and Software Technology**, p. 1 – 22, 2018.

HACHE, C. Top 7 Design Patterns Every Unity Game Developer Should Know. 2023. Disponível em: https://www.linkedin.com/pulse/top-7-design-patterns-every-unity-game-developer-should-charles-hache/. Acesso em: 20, ago., 2023.

HUSSAIN, A.; SHAKEEL, H.; HUSSAIN, F.; UDDIN, N.; GHOURI, T. L. Unity Game Development Engine: A Technical Survey. **University of Sindh Journal of Information and Communication Technology**, v. 4 (2), p. 73 – 81, 2020.

IBRAHIM, M. Structuring Your Unity Code For Production - Important Best Practices. 2023. Disponível em: https://www.codementor.io/@mody/structuring-your-unity-code-for-production-important-best-practices-25bmix6f3q. Acesso em: 16, ago., 2023.

JUEGO, S. 7 Ways to Keep Unity Project Organized: Unity3d Best Practices. Disponível em: https://www.juegostudio.com/blog/7-ways-to-keep-unity-project-organized-unity3d-best-practices. Acesso em: 20, set., 2023.

KARPOVICH, A.; PYATKI, D. Improving the performance of unity 3d mobile games. **Electronic collected materials of xi junior researchers' conference**, p. 154 – 156, 2019.

KROGH-JACOBSEN, T. Level up your code with game programming patterns. Disponível em: https://blog.unity.com/games/level-up-your-code-with-game-programming-patterns. Acesso em: 02, set., 2023.

KUNDURTHY, P. Software Performance Optimizations for Games: Best Practices. 2016. Disponível em: https://www.intel.com/content/www/us/en/developer/articles/technical/unity-software-performance-optimizations-for-games-best-practices.html. Acesso em: 04, set., 2023.

LAFRITZ, J. Model-View-Controller Family. Disponível em: https://blog.devgenius.io/model-view-controller-family-3a0d869d81ea. Acesso em: 03, set., 2023.

LEVCHENKO, A. Unity ECS: How Does It Work and Why You Should Use It. Disponível em: https://www.visartech.com/blog/what-is-entity-component-system-ecs-and-how-to-benefit-in-unity/. Acesso em: 19, jun., 2023.

LIN, W. **Level up your code with game programming patterns**. Unity, 2021.

MARTIN, R. C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. Pearson, 2017.

NARDONE, V.; MUSE, B.; ABIDI, M.; KHOMH, F.; DI PENTA, M. Video Game Bad Smells: What They Are and How Developers Perceive Them. **ACM Transactions on Software Engineering and Methodology**, p. 1 – 35, 2023.

NYSTROM, R. **Game Programming Patterns**. 2014.

REDDIT. "What are some bad practices to avoid when using Unity?". Disponível em: https://www.reddit.com/r/Unity3D/comments/9yg57s/what_are_some_bad_practices_to_avoid_when_using/. Acesso em: 02, set., 2023.

REFACTORING GURU. Factory Method. Disponível em: https://refactoring.guru/design-patterns/factory-method. Acesso em: 04, set., 2023.

RICHARDS, M.; FORD, N. **Fundamentals of Software Architecture**. O'Reilly Media, 2020.

SCHARDON, L. What is Unity? – A Guide for One of the Top Game Engines. Disponível em: https://gamedevacademy.org/what-is-unity/. Acesso em: 19, jun., 2023.

SHAH, V. Reasons Why Unity3D Is So Much Popular In The Gaming Industry. Disponível em: https://medium.com/@vivekshah.P/reasons-why-unity3d-is-so-much-popular-in-the-gaming-industry-705898a2a04. Acesso em: 19, jun., 2023.

SUSCHEUSKI, D.; BURACHONAK, I. Architectural design pattern entity-component-system. **Electronic collected materials of xi junior researchers' conference**, p. 144 – 146, 2019.

TULLEKEN, H. 0 Tips and Best Practices for Unity (2016 Edition). Disponível em: https://www.gamedeveloper.com/design/50-tips-and-best-practices-for-unity-2016-edition-. Acesso em: 11, set., 2023.

UNITY. Control of an object's position through physics simulation. Disponível em: https://docs.unity3d.com/ScriptReference/Rigidbody.html. Acesso em: 02, set., 2023.

UNITY. Slider Scripting API. Disponível em: https://docs.unity3d.com/2022.2/Documentation/Manual/script-Slider.html. Acesso em: 03, ago., 2023.

UNITY. Pool.ObjectPool_1 Class. A stack based IObjectPool<T0>. Disponível em: https://docs.unity3d.com/ScriptReference/Pool.ObjectPool_1.html. Acesso em: 15, set., 2023.

UNITY. Creating Components with Scripts. Disponível em: https://docs.unity3d.com/Manual/CreatingComponents.html. Acesso em: 26, ago., 2023.

WEIMANN, J. Unity Bots with State Machines - Extensible State Machine/FSM. Jason Weimann. YouTube. https://www.youtube.com/watch?v=V75hgcsCGOM. Publicado em 26 de abril de 2020.

8. Anexo de Figuras

Figura A – Parte de uma implementação simples de FSM na *Unity* para controlar um zumbi.

```
using UnityEngine;
public class ZombieController_Unrefactored : MonoBehaviour
{
    public enum ZombieStates
    {
        Attack,
        Patrol,
        Idle
    }
    [field: SerializeField] public Animator Animator { get; private set; }
    [field: SerializeField] public EnemyDetector EnemyDetector { get; private set; }
    [field: SerializeField] public MovementComponent MovementComponent {get;private set;}
    [field: SerializeField] public float PatrolTime { get; private set; }

    public ZombieStates currentState { get; private set; }
    public Timer PatrolTimer { get; private set; }

    private bool isFirstFrameOfPatrolState;
    public void Update()
    {
        switch (currentState)
        {
            case ZombieStates.Attack:
                Attack(); // TO-DO
                break;
            case ZombieStates.Patrol:
                if (isFirstFrameOfPatrolState)
                {
                    PatrolTimer.Start();
                    Animator.SetBool(nameof(PatrolState), true);
                    EnemyDetector.enabled = true;
                    isFirstFrameOfPatrolState = false;
                }
                Patrol();
                break;
            case ZombieStates.Idle:
                Idle(); // TO-DO
                break;
        }
    }
    public void Awake()
    {
        PatrolTimer = new Timer(PatrolTime);
    }
   //...
```

Fonte: autoral.

Figura B – Continuação da Figura A, a qual mostra a implementação de uma FSM simples na *Unity* para controlar um zumbi.

```
//...
public void Awake()
    {
        PatrolTimer = new Timer(PatrolTime);
    }
    private void Patrol()
    {
        if (PatrolTimer.HasFinishedTimer())
        {
            currentState = ZombieStates.Idle;
            PatrolExit();
            return;
        }else if (EnemyDetector.HasDetectedEnemiesThisFrame())
        {
            currentState = ZombieStates.Idle;
            PatrolExit();
            return;
        }
        MovementComponent.RandomMove();
    }
    private void PatrolExit()
    {
        PatrolTimer.Reset();
        Animator.SetBool(nameof(PatrolState), false);
        EnemyDetector.enabled = false;
        isFirstFrameOfPatrolState = true;
    }
    private void Idle()
    {
        // checar transições de estado e ações...
    }
    private void Attack()
    {
        // checar transições de estado e ações...
    }
}
```

Fonte: autoral.

Figura C – Definição da interface *IState* na *Unity*, a qual os estados devem implementar conforme padrão *State*.

```
public interface IState
{
    // executa lógica ao entrar no estado
    public void Enter();
    // executa lógica por frame, como também, lógica de transição*
    public void Update();
    // executa lógica ao sair do estado
    public void Exit();
}
```

Fonte: autoral.

Figura D – Exemplo de implementação de uma *State Machine* conforme o padrão *State*.

```
using System;

[Serializable]
public class StateMachine
{
    //guarda estado atual
    public IState CurrentState { get; private set; }

    //Inicializa o estado
    public void Initialize(IState startingState)
    {
        CurrentState = startingState;
        startingState.Enter();
    }

    //troca de estado
    public void ChangeState(IState nextState)
    {
        if (nextState == CurrentState)
            return;
        CurrentState.Exit();
        CurrentState = nextState;
        CurrentState.Enter();
    }

    //repasse para a chamada do Update do estado atual
    public void Update()
    {
        if (CurrentState == null)
        {
            return;
        }
        CurrentState.Update();
    }
}
```

Fonte: autoral.

Figura E – Implementação exemplo de um Zumbi *Controller*, responsável apenas por definir os estados e inicializá-los.

```
using UnityEngine;
public class ZombieController : MonoBehaviour
{
    [field: SerializeField] public Animator Animator { get; private set; }
    [field:SerializeField] public EnemyDetector EnemyDetector { get; private set; }
    [field: SerializeField] public MovementComponent MovementComponent { get; private set; }
    [field: SerializeField] public float PatrolTime { get; private set; }

    public StateMachine StateMachine { get; private set; }
    public PatrolState PatrolState { get; private set; }
    public AttackState AttackState { get; private set; }
    public IdleState IdleState { get; private set; }

    private void Start()
    {
        //Inicializando variáveis
        PatrolState = new PatrolState(this,EnemyDetector,MovementComponent,Animator,PatrolTime);
        AttackState = new AttackState(); // pode ter outros parâmetros também
        IdleState = new IdleState(); // pode ter outros parâmetros também
        StateMachine = new StateMachine();
        StateMachine.Initialize(IdleState);
    }

    private void Update() => StateMachine.Update();
}
```

Fonte: autoral.

Figura F – Implementação exemplo do estado de patrulha de um zumbi.

```
using UnityEngine;

public class PatrolState : IState
{
    private ZombieController zombie;
    private MovementComponent movementComponent;
    private Animator animator;
    private EnemyDetector enemyDetector;
    public Timer PatrolTimer { get; private set; }

    public PatrolState(ZombieController zombie, EnemyDetector enemyDetector,
        MovementComponent movementComponent, Animator animator, float patrolTime)
    {
        this.zombie = zombie;
        this.movementComponent = movementComponent;
        this.animator = animator;
        this.enemyDetector = enemyDetector;
        PatrolTimer = new Timer(patrolTime);
    }

    public void Enter()
    {
        PatrolTimer.Start();
        animator.SetBool(nameof(PatrolState),true);
        enemyDetector.enabled = true;
    }

    public void Update()
    {
        if (enemyDetector.HasDetectedEnemiesThisFrame())
        {
            zombie.StateMachine.ChangeState(zombie.AttackState);
        }
        else if(PatrolTimer.HasFinishedTimer())
        {
            zombie.StateMachine.ChangeState(zombie.IdleState);
        }

        movementComponent.RandomMove();
    }

    public void Exit()
    {
        PatrolTimer.Reset();
        animator.SetBool(nameof(PatrolState), false);
        enemyDetector.enabled = false;
    }
}
```

Fonte: autoral.

Figura G – Implementação da classe *Transition*, responsável por combinar o par (Estado, Condição).

```
public class Transition
{
    public Func<bool> Condition { get; }
    public IState To { get; }

    public Transition(IState to, Func<bool> condition)
    {
        To = to;
        Condition = condition;
    }
}
```

Fonte: autoral.

Figura H – Adição de transições na classe *StateMachine*.

```
public class StateMachineWithTransition
{
    public IState CurrentState { get; private set; }

    private Dictionary<Type, List<Transition>> transitions = new Dictionary<Type, List<Transition>>
(); private List<Transition> currentTransitions = new List<Transition>();

    private static List<Transition> EmptyTransitions = new List<Transition>(0);

    public void Update()
    {
        var transition = GetTransition();
        if (transition != null)
            ChangeState(transition.To);

        CurrentState.Update();
    }

    public void ChangeState(IState state)
    {
        if (state == CurrentState)
            return;

        CurrentState?.Exit();
        CurrentState = state;

        transitions.TryGetValue(CurrentState.GetType(), out currentTransitions);
        if (currentTransitions == null)
            currentTransitions = EmptyTransitions;

        CurrentState.Enter();
    }

    public void AddTransition(IState from, IState to, Func<bool> predicate)
    {
        if (transitions.TryGetValue(from.GetType(), out var transitionList) == false)
        {
            transitionList = new List<Transition>();
            transitions[from.GetType()] = transitionList;
        }

        transitionList.Add(new Transition(to, predicate));
    }

    private Transition GetTransition()
    {
        foreach (var transition in currentTransitions)
            if (transition.Condition())
                return transition;

        return null;
    }
}
```

Fonte: autoral.

Figura I – Definindo transições na classe *Zombie Controller* e adicionando-as a *StateMachine*.

```
public class ZombieControllerWithStateMachineTransition: MonoBehaviour
{
    [field: SerializeField] public Animator Animator { get; private set; }
    [field: SerializeField] public EnemyDetector EnemyDetector { get; private set; }
    [field: SerializeField] public MovementComponent MovementComponent { get; private set; }
    [field: SerializeField] public float PatrolTime { get; private set; }

    public StateMachineWithTransition StateMachine { get; private set; }

    private void Start()
    {
        //Iniciando variáveis
        var PatrolState = new PatrolState_Refactored(EnemyDetector, MovementComponent, Animator,
PatrolTime);
        var AttackState = new AttackState(); // pode ter outros parâmetros também
        var IdleState = new IdleState(); // pode ter outros parâmetros também
        StateMachine = new StateMachineWithTransition();
        //definindo funções de transição
        Func<bool> IsPatrolTimeOver() => () => PatrolState.PatrolTimer.HasFinishedTimer();
        Func<bool> IsEnemyOnRange() => () => EnemyDetector.HasDetectedEnemiesThisFrame();
        // ...
        // definindo transições
        StateMachine.AddTransition(PatrolState, IdleState, IsPatrolTimeOver());
        StateMachine.AddTransition(PatrolState, AttackState, IsEnemyOnRange());
        //...
        StateMachine.ChangeState(IdleState);
    }
    private void Update() => StateMachine.Update();
}
```

Fonte: autoral.

Figura J – Removendo transições de estado e dependência do Zombie *Controller* no estado de patrulha visto anteriormente.

```
public class PatrolState_Refactored : IState
{
    private MovementComponent movementComponent;
    private Animator animator;
    private EnemyDetector enemyDetector;

    public Timer PatrolTimer { get; private set; }

    public PatrolState_Refactored(EnemyDetector enemyDetector,
        MovementComponent movementComponent, Animator animator, float patrolTime)
    {
        this.movementComponent = movementComponent;
        this.animator = animator;
        this.enemyDetector = enemyDetector;
        PatrolTimer = new Timer(patrolTime);
    }

    public void Enter()
    {
        PatrolTimer.Start();
        animator.SetBool(nameof(PatrolState), true);
        enemyDetector.enabled = true;
    }

    public void Update()
    {
        movementComponent.RandomMove();
    }

    public void Exit()
    {
        PatrolTimer.Reset();
        animator.SetBool(nameof(PatrolState), false);
        enemyDetector.enabled = false;
    }
}
```

Fonte: autoral.

Figura K – Código de um modelo que representa a vida de um item ou personagem na *Unity*.

```
public class Health: MonoBehaviour
{
    public event Action HealthChanged;

    private const int minHealth = 0;
    private const int maxHealth = 100;
    private int currentHealth;

    public int CurrentHealth { get => currentHealth; set => current-
Health = value; }
    public int MinHealth => minHealth;
    public int MaxHealth => maxHealth;

    public void Increment(int amount)
    {
        currentHealth += amount;
        currentHealth = Mathf.Clamp(currentHealth, minHealth, max-
Health);
        UpdateHealth();
    }

    public void Decrement(int amount)
    {
        currentHealth -= amount;
        currentHealth = Mathf.Clamp(currentHealth, minHealth, max-
Health);
        UpdateHealth();
    }

    public void Restore()
    {
        currentHealth = maxHealth;
        UpdateHealth();
    }

    public void UpdateHealth()
    {
        HealthChanged?.Invoke();
    }
}
```

Fonte: Lin (2021).

Figura L – Exemplo de implementação concreta de produto e fábrica na *Unity*.

```
public class ProductA : MonoBehaviour, IProduct
{
    [SerializeField] private string productName = "ProductA";
    public string ProductName { get => productName; set => productName
= value ; }

    private ParticleSystem particleSystem;

    public void Initialize()
    {
        // any unique logic to this product
        gameObject.name = productName;
        particleSystem = GetComponentInChildren<ParticleSystem>();
        particleSystem?.Stop();
        particleSystem?.Play();
    }
}

public class ConcreteFactoryA : Factory
{

    [SerializeField] private ProductA productPrefab;

    public override IProduct GetProduct(Vector3 position)
    {
        // create a Prefab instance and get the product component
        GameObject instance = Instantiate(productPrefab.gameObject,
position, Quaternion.identity);
        ProductA newProduct = instance.GetComponent<ProductA>();

        // each product contains its own logic
        newProduct.Initialize();

        return newProduct;
    }
}
```

Fonte: (Unity, 2023).

Figura M – Exemplo de implementação do padrão *Object Pool* na *Unity*, utilizando de biblioteca pronta feita pela própria *Unity*.

```
using UnityEngine.Pool;

public class RevisedGun : MonoBehaviour
{
    …

    // stack-based ObjectPool available with Unity 2021 and above
    private IObjectPool<RevisedProjectile> objectPool;

    // throw an exception if we try to return an existing item, already
in the pool
    [SerializeField] private bool collectionCheck = true;
    // extra options to control the pool capacity and maximum size
    [SerializeField] private int defaultCapacity = 20;
    [SerializeField] private int maxSize = 100;

    private void Awake()
    {
        objectPool = new ObjectPool<RevisedProjectile>(CreateProjec-
tile,
            OnGetFromPool, OnReleaseToPool, OnDestroyPooledObject,
            collectionCheck, defaultCapacity, maxSize);
    }

    // invoked when creating an item to populate the object pool
    private RevisedProjectile CreateProjectile()
    {
        RevisedProjectile projectileInstance = Instantiate(projec-
tilePrefab);
        projectileInstance.ObjectPool = objectPool;
        return projectileInstance;
    }

    // invoked when returning an item to the object pool
    private void OnReleaseToPool(RevisedProjectile pooledObject)
    {
        pooledObject.gameObject.SetActive(false);
    }

    // invoked when retrieving the next item from the object pool
    private void OnGetFromPool(RevisedProjectile pooledObject)
    {
        pooledObject.gameObject.SetActive(true);
    }

    // invoked when we exceed the maximum number of pooled items (i.e.
destroy the pooled object)
    private void OnDestroyPooledObject(RevisedProjectile pooledObject)
    {
        Destroy(pooledObject.gameObject);
    }

    private void FixedUpdate()
    {
        …

    }
}
```

Fonte: (Unity, 2023).

Figura N – Exemplo de melhoria a Figura M, ao utilizar de *generics*, bem como, aplicar o padrão *Factory* em conjunto com o padrão *Object Pool* na *Unity*.

```
public abstract class ObjectPoolFactory<T> : MonoBehaviour, IObjectPool<T>
    where T : MonoBehaviour, IPoolableProduct<T>
{
    [SerializeField]
    protected T productPrefab;
    // stack-based ObjectPool available with Unity 2021 and above
    protected IObjectPool<T> objectPool;

    // throw an exception if we try to return an existing item, already in the pool
    [SerializeField] protected bool collectionCheck = true;
    // extra options to control the pool capacity and maximum size
    [SerializeField] protected int defaultCapacity = 20;
    [SerializeField] protected int maxSize = 100;

    protected void Awake()
    {
        objectPool = new ObjectPool<T>(CreateProduct,OnGetFromPool, OnReleaseToPool,
                    OnDestroyPooledObject,collectionCheck, defaultCapacity, maxSize);
    }

    Object Pool Callbacks

    IObjectPool interface
}

public interface IPoolableProduct<T> where T : MonoBehaviour
{
    public IObjectPool<T> ObjectPool { get; set; }

    public void ReleaseToPool();
}
```

Fonte: autoral.

Figura O – Código em c + + para exemplificar uma classe monolítica antes de aplicar o padrão *Component*.

```
void Bjorn::update(World& world, Graphics& graphics)
{
  // Apply user input to hero's velocity.
  switch (Controller::getJoystickDirection())
  {
    case DIR_LEFT:
      velocity_ -= WALK_ACCELERATION;
      break;

    case DIR_RIGHT:
      velocity_ += WALK_ACCELERATION;
      break;
  }

  // Modify position by velocity.
  x_ += velocity_;
  world.resolveCollision(volume_, x_, y_, velocity_);

  // Draw the appropriate sprite.
  Sprite* sprite = &spriteStand_;
  if (velocity_ < 0)
  {
    sprite = &spriteWalkLeft_;
  }
  else if (velocity_ > 0)
  {
    sprite = &spriteWalkRight_;
  }

  graphics.draw(*sprite, x_, y_);
}
```

Fonte: (Nystrom, 2014)

Figura P – Extração de componentes da classe monolítica anteriormente demonstrada, em c + +.

```
class Bjorn
{
public:
  int velocity;
  int x, y;

  void update(World& world, Graphics& graphics)
  {
    input_.update(*this);
    physics_.update(*this, world);
    graphics_.update(*this, graphics);
  }

private:
  InputComponent input_;
  PhysicsComponent physics_;
  GraphicsComponent graphics_;
};
```

Fonte: (Nystrom, 2014)

Figura Q – Implementação exemplo de um componente de movimentação na *Unity*.

```
[RequireComponent(typeof(Rigidbody))]
public class MovementComponent : MonoBehaviour, IMoveable
{
    [SerializeField]
    private float moveSpeed;

    private Rigidbody rb;
    private Vector3 direction;

    private void Awake()
    {
        rb = GetComponent<Rigidbody>();
        rb.constraints = RigidbodyConstraints.FreezeRotation;
        rb.useGravity = false;
    }

    private void FixedUpdate()
    {
        direction *= moveSpeed * Time.fixedDeltaTime;
        rb.MovePosition(rb.position + direction);
    }

    public void Move(Vector3 direction)
    {
        direction.y = 0; // do not move in y-axis

        if(direction.magnitude > 1)
        {
            direction.Normalize();
        }

        this.direction = direction;
    }

    public void RandomMove()
    {
        Move(GetRandomDirection());
    }

    private static Vector3 s_direction;
    public static Vector3 GetRandomDirection()...
}
```

Fonte: autoral.

Figura R – Interface *IArrow*, utilizada de exemplo na *Unity* para implementar uma interface de um componente do padrão *Decorator* na *Unity*.

```
public interface IArrow : IBuffable
{
    public void Setup(Vector3 arrowDirection, float bowDamage);
    public float GetDamage();
    public float GetSpeed();
    public Vector3 GetDirection();
}
```

Fonte: autoral.

Figura S – Classe *Arrow*, utilizada de exemplo na *Unity* para implementar um componente concreto de interface do padrão *Decorator* na *Unity*.

```
[Serializable]
public class Arrow : IArrow
{
    private float bowDamage;
    [SerializeField]
    private float arrowDamage = 5;
    [SerializeField]
    private float speed = 10;
    private Vector3 direction;

    public Arrow(float speed, float arrowDamage)...

    public Arrow(Arrow arrowToClone)...

    #region IArrow Interface

    public float GetDamage(){ return arrowDamage+bowDamage; }

    public Vector3 GetDirection() { return direction; }

    public float GetSpeed() { return this.speed; }

    public void Setup(Vector3 arrowDirection, float bowDamage)
    {
        this.direction = arrowDirection;
        this.bowDamage = bowDamage;
    }
    #endregion
}
```

Fonte: autoral.

Figura T – Classe *ArrowDecorator*, utilizada de exemplo na *Unity* para implementar uma classe base de decoração do padrão *Decorator*.

```
public class IronArrow : ArrowDecorator
{
    private readonly float bonusDamage;

    public IronArrow(float bonusDamage,IArrow arrow) : base(arrow)
    {
        this.bonusDamage = bonusDamage;
    }

    public override float GetDamage()
    {
        return base.GetDamage() + bonusDamage;
    }
}
```

Fonte: autoral.

Figura U – Classe *IronArrow*, utilizada de exemplo na *Unity* para implementar uma classe concreta de decoração do padrão *Decorator*.

.

```
public class IronArrow : ArrowDecorator
{
    private readonly float bonusDamage;

    public IronArrow(float bonusDamage,IArrow arrow) : base(arrow)
    {
        this.bonusDamage = bonusDamage;
    }

    public override float GetDamage()
    {
        return base.GetDamage() + bonusDamage;
    }
}
```

Fonte: autoral.

Figura V – Classe *LighterArrow*, utilizada de exemplo na *Unity* para implementar uma classe concreta de decoração do padrão *Decorator*.

```
public class LighterArrow : ArrowDecorator
{
    private readonly float bonusSpeed;

    public LighterArrow(float bonusSpeed, IArrow arrow) : base(arrow)
    {
        this.bonusSpeed = bonusSpeed;
    }

    public override float GetSpeed()
    {
        return base.GetSpeed() + bonusSpeed;
    }
}
```

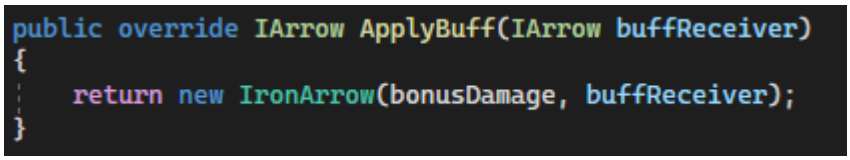
Fonte: autoral.

Figura X – Classe *ArrowFactory*, utilizada de exemplo na *Unity* para fabricar componentes decorados do padrão *Decorator*.

```
public class ArrowFactory : BuffManager<IArrow>
{
    [SerializeField]
    private ArrowBehaviour arrowBehaviour;

    public virtual IArrow CreateArrow(Transform parent,
                                      Transform arrowOrigin,
                                      Vector3 direction,
                                      float bowDamage = 0)
    {
        var newArrowBehaviour = Instantiate(arrowBehaviour,
                                            arrowOrigin.position,
                                            arrowBehaviour.transform.rotation,
                                            parent);

        newArrowBehaviour.CurrentArrow.Setup(direction, bowDamage);
        foreach (var buff in BuffList)
        {
            var arrowBuffed = buff.ApplyBuff(newArrowBehaviour.CurrentArrow);
            newArrowBehaviour.CurrentArrow = arrowBuffed;
        }

        return newArrowBehaviour.CurrentArrow;
    }
}
```

Fonte: autoral.

Figura Y – Método *ApplyBuff()*, utilizado de exemplo na *Unity* para retornar uma decoração aplicada a um componente do padrão *Decorator*.



```
public override IArrow ApplyBuff(IArrow buffReceiver)
{
    return new IronArrow(bonusDamage, buffReceiver);
}
```

Fonte: autoral.