PONTIFÍCIA UNIVERSIDADE CATÓLICA DE SÃO PAULO

CURSO DE ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

EDIVAL MARTIN DE SOUZA NETO

A RELAÇÃO ENTRE PADRÕES DE PROJETO E JOGOS DIGITAIS DESENVOLVIDOS NO UNITY

São Paulo
Julho/2017

EDIVAL MARTIN DE SOUZA NETO

A Relação Entre Padrões de Projeto e Jogos Digitais Desenvolvidos No Unity

Monografia apresentada ao Curso de Especialização em Engenharia de Software da Pontifícia Universidade Católica de São Paulo, como requisito parcial para obtenção do título de Especialista em Engenharia de Software, orientado pelo Prof. Dr. Daniel Couto Gatti.

São Paulo
2017

À minha família.

Agradecimentos

Ao meu orientador, Professor Doutor Daniel Couto Gatti, pela disposição durante as orientações e pelos conselhos para o desenvolvimento deste trabalho.

Ao Professor André Pereira, por me apresentar aos Padrões de Projeto e me incentivar a iniciar esta pesquisa.

À minha família, pela paciência e compreensão ao longo dessa jornada.

"Models are not right or wrong; they are more or less useful."
Martin Fowler

Lista de ilustrações

Resumo

Motores de jogos são softwares que disponibilizam bibliotecas e ferramentas utilizadas no desenvolvimento de projetos de jogos digitais, e que ajudam a diminuir a complexidade na implementação desses projetos. Atualmente, um dos motores de jogos mais populares é o Unity, utilizado tanto por pequenos estúdios quanto grandes empresas desenvolvedoras no mercado. Dentro do contexto do Unity, os padrões de projeto - soluções reutilizáveis para problemas específicos e recorrentes em projetos de software orientados a objetos - podem ser aplicados para obter uma estrutura de código mais consistente e uma arquitetura bem definida, respeitando a arquitetura e o fluxo de trabalho do motor de jogo. Este trabalho apresenta uma pesquisa descritiva de alguns dos principais padrões de projeto utilizados em jogos digitais, e como eles se relacionam com as características do Unity. São apresentados os padrões de projeto tradicionais nos projetos de software, além de padrões de projeto específicos em projetos de jogos e em quais contextos poderiam ser aplicados em projetos de jogos. Na conclusão deste trabalho, sugere-se como esses padrões podem ser utilizados em conjunto dentro do Unity e como as características do motor de jogo impactam na decisão do uso de um padrão de projeto.

Palavras-chaves: Padrões de Projeto. Unity. Orientação a Objetos.

Abstract

Game engines are software that provides libraries and tools used in digital game projects development, that help to reduce complexity of implementation in these projects. Nowadays, one of the most popular game engines is Unity, used by both small studios and large developer companies in the market. Design patterns - reusable solutions for specific and recurrent problems in object-oriented software projects - can be applied in Unity to obtain a more consistent code structure and well-defined architecture, respecting game engine's architecture and workflow. This paper presents a descriptive study of some of the major design patterns used in digital games and how they relate to the Unity's characteristics. Traditional design patterns in software projects are presented, as well as specific design patterns in game projects and in which contexts they could be applied in game projects. In conclusion, it is suggested how these patterns can be used together with Unity and how game engine's characteristics impact the decision of using a design pattern.

Keywords: Design Patterns. Unity. Object Oriented.

Sumário

INTRODUÇÃO

Motivação

O desenvolvimento de jogos digitais possui vários desafios inerentes a projetos de software, como a expansão do código e a consequente complexidade em sua manutenção, a dificuldade de se desenvolver código que seja reutilizável e a preocupação com o desenvolvimento de uma estrutura de software consistente. Além disso, fatores como a renderização de imagens em tempo real, gerenciamento de áudio, simulação de física e controle de tomada de decisões são desafios específicos existentes nesses projetos.

Atualmente, para tentar diminuir a complexidade no desenvolvimento de jogos, diversos projetos são implementados utilizando um *Game Engine* (Motor de Jogo, em português) – software que disponibiliza um conjunto de ferramentas e bibliotecas prontas para o desenvolvimento de jogos, como motores de simulação de física, bibliotecas para renderização de imagens e controladores de áudio, por exemplo. Motores de jogos são utilizados por desenvolvedores amadores e profissionais de estúdios de desenvolvimento, desde pequenas equipes e estúdios independentes a empresas de jogos tradicionais com grande apelo comercial.

Um dos motores de jogos atuais mais populares é o Unity, que possui entre as suas principais vantagens o conceito de *Build Once Deploy Anywhere* (construa uma vez, implemente em qualquer parte, em tradução livre), em que o jogo desenvolvido pode ser exportado para diversas plataformas, inclusive consoles como Playstation 4 e Xbox One. O Unity também possui uma licença gratuita para desenvolvedores independentes, facilitando a popularização do motor de jogos entre equipes menores e pequenos estúdios.

O Unity permite que o desenvolvedor crie scripts com código para adicionar características e comportamentos aos objetos do jogo a partir de componentes pré-definidos pelo motor de jogo ou customizados. Contudo, à medida que um projeto no Unity cresce sem o devido cuidado com a estrutura do projeto, o gerenciamento desses objetos e componentes tende a ser mais problemático para os desenvolvedores, tornando a manutenção de código impraticável. Para evitar esses casos, o uso de padrões de projeto é um caminho a ser considerado para a construção de um projeto mais consistente.

Padrões de projeto (*Design Patterns*, em inglês) descrevem soluções reutilizáveis para problemas recorrentes no desenvolvimento de softwares, definidos por meio de experiências em outros projetos e que possam ser aplicadas em situações específicas por diversos projetos, com o objetivo de auxiliar no desenvolvimento de código reutilizável e mais flexível (GAMMA et al., 1995, p. 18).

Assim como os softwares tradicionais, os desenvolvedores de jogos no Unity podem aplicar os padrões de projeto no desenvolvimento de seus jogos. No entanto, é necessário considerar os tipos de problemas enfrentados na implementação de um projeto de jogo, e como a utilização do Unity pode impactar a decisão do uso de um padrão de projeto.

Objetivos

O objetivo deste trabalho é realizar uma pesquisa descritiva sobre o uso de padrões de projeto no desenvolvimento de jogos digitais, relacionando-os com os problemas comuns enfrentados nesses projetos e como o motor de jogo Unity impacta a decisão de implementação desses padrões de projeto.

Método de Pesquisa

Pesquisa Bibliográfica: A primeira parte do trabalho dará ênfase à pesquisa descritiva dos padrões de projeto já consolidados na Engenharia de Software. A pesquisa se aprofundará na descrição dos padrões de projeto criacionais, estruturais e comportamentais catalogados no trabalho apresentado por GAMMA et al. (GAMMA et al., 1995), também conhecidos na literatura como *GoF – Gang of Four*.

Unity: Essa atividade de pesquisa está relacionada ao motor de jogo Unity e a sua estrutura. Pretende-se identificar quais são as características da arquitetura do Unity que reforçam a necessidade de se utilizar padrões de projeto no motor de jogo. Nessa atividade são relacionados alguns conceitos como o padrão *Entity-Component* no fluxo de trabalho do Unity (NYSTROM, 2014).

Pesquisa dos Padrões de Projeto Comuns em Projetos de Jogos: A pesquisa foca nos padrões de projeto recorrentes na implementação de jogos digitais. Alguns padrões utilizados nos jogos são adaptações dos padrões do GoF (NYSTROM, 2014), para cenários de problemas comuns enfrentados no desenvolvimento de jogos digitais. Outras publicações, que possam sugerir aplicações de diferentes padrões para soluções de problemas específicos em jogos digitais, também fazem parte desta atividade da pesquisa.

Relação entre padrões de projeto e o Unity: A pesquisa descreve a aplicação dos principais padrões de projeto em jogos no Unity, com base em outras publicações anteriores, relacionando características do padrão com a implementação no motor de jogo.

Considerações finais: Ao final da pesquisa, as conclusões obtidas pelo estudo serão apresentadas de forma que o leitor possa compreender a aplicação dos padrões de projeto na solução de problemas existentes em projetos de jogos digitais, e de qual forma o uso do Unity pode impactar suas decisões de implementação.

1 Uma Introdução aos Padrões de Projeto

1.1 Padrões de Projeto

Uma das definições de padrão mais difundidas na engenharia de software foi apresentada pelo arquiteto Christopher Alexander: “Cada padrão descreve um problema que ocorre frequentemente em nosso ambiente, e então descreve o núcleo da solução para aquele problema, de forma que você possa usar essa solução um milhão de vezes, sem fazê-la da mesma forma duas vezes.” (Alexander et al., 1977). Embora não seja uma definição específica da engenharia de software, ela nos ajuda a compreender a origem e a necessidade da representação de padrões de projeto para softwares.

FOWLER argumenta que padrões de projeto não são criados, e sim descobertos. Padrões não são ideias originais, mas um conjunto de observações realizadas na área (FOWLER, 2002, p. 10).

Segundo GAMMA et al., “um padrão de projeto nomeia, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizável”. Por meio de soluções representadas por objetos e interfaces, padrões solucionam problemas em um contexto específico no desenvolvimento de software (GAMMA et al., 1995, p. 19).

GAMMA et al. apontam para quatro elementos essenciais de um padrão de projeto:

a) Nome do padrão: auxilia o esclarecimento ao projetar software com um nível mais alto de abstração, além de ser uma referência simples para o problema a qual ele se adapta e à solução proposta pelo padrão;

b) Problema: a situação em que o padrão é mais bem aplicado. Segundo GAMMA et al., em determinadas situações, “o problema incluirá uma lista de condições que devem ser satisfeitas para que faça sentido aplicar o padrão” (GAMMA et al., 1995, p. 19);

c) Solução: As soluções propostas não descrevem um projeto específico ou um estudo de caso, por exemplo, pois são propostas que se adequariam a diversas situações. Um padrão descreverá de forma abstrata um problema de projeto e a sua possível solução, formada por um arranjo geral de elementos (classes e objetos, no cenário de desenvolvimento de software orientado a objetos) (GAMMA et al., 1995, p. 19);

d) Consequências: A utilização de um padrão pode estar relacionada a várias vantagens e desvantagens, que devem ser consideradas no momento da escolha do

> padrão. Para GAMMA et al., essas consequências "são críticas para a avaliação de alternativas de projetos e para a compreensão dos custos e benefícios da aplicação do padrão". As consequências podem estar relacionadas ao impacto sobre a flexibilidade, a extensibilidade ou a portabilidade (GAMMA et al., 1995, p. 19).

Uma vez definidos os elementos que geralmente formam a essência de um padrão, podemos considerar o contexto de padrões de projeto, definidos por GAMMA et al., como:

> O padrão de projeto identifica as classes e instâncias participantes, seus papéis, colaborações e a distribuição de responsabilidades. Cada padrão de projeto focaliza um problema ou tópico particular de projeto orientado a objetos. Ele descreve em que situação pode ser aplicado, se ele pode ser aplicado em função de outras restrições de projeto e as consequências, custos e benefícios de sua utilização (GAMMA et al., 1995, p. 20).

1.2 Solução de Problemas utilizando Padrões de Projeto

Existem muitos desafios a serem considerados durante a implementação de um projeto orientado a objetos, portanto a utilização adequada dos padrões de projeto é parte crucial para enfrentar tais desafios. Contudo, para compreender e identificar o padrão de projeto ideal para a solução de um determinado problema, precisamos primeiramente compreender as características dos projetos orientados a objetos.

Segundo a definição de Gamma et al.:

a) objetos: armazenam os dados e métodos operados sobre eles, executados quando um cliente faz uma solicitação de sua operação. Os objetos são instâncias de uma classe (GAMMA et al., 1995, p. 29);

b) classes: definem a implementação de um objeto, especificam seus dados internos e de sua representação e definem as operações que os objetos poderão executar (GAMMA et al., 1995, p. 29);

c) interface de objeto: "o conjunto de todas as assinaturas definido pelas operações de um objeto" (GAMMA et al., 1995, p. 29). Representa quais são as operações disponíveis para um objeto;

d) polimorfismo: permite que objetos que possuam a mesma interface sejam utilizados sem que o cliente solicitante tenha conhecimento de suas implementações. Segundo Gamma et al., o polimorfismo "simplifica as definições dos clientes, desacopla objetos entre si e permite a eles variarem seus inter-relacionamentos em tempo de execução" (GAMMA et al., 1995, p. 29).

Gamma et al. enfatizam que projetar software orientado a objetos tem como principal dificuldade a decomposição de um sistema em objetos, uma vez que devem

ser levados em consideração fatores conflitantes como encapsulamento, granularidade, dependência, flexibilidade, desempenho, evolução e reutilização (GAMMA et al., 1995, p. 28).

A forma como um objeto é representado em um projeto orientado a objetos depende do tipo de abordagem utilizada na definição de suas classes. As classes de objetos podem representar conceitos simples ou mais abstratos. Segundo Gamma et al., as abstrações são as chaves para possibilitar a flexibilidade em um projeto, e os padrões de projeto facilitam o trabalho de identificar abstrações menos óbvias das quais os objetos possam fazer uso, permitindo a flexibilidade e reutilização no projeto.

Os padrões de projeto podem atuar sobre as interfaces ajudando na identificação de seus elementos-chave e pelos tipos de dados que elas transmitem. Também podem exigir que algumas classes tenham interfaces similares, ou possuam restrições em interfaces (GAMMA et al., 1995, p. 29).

1.3 Padrões de Projeto Catalogados pelo GoF

Os padrões de projeto catalogados por Gamma et al. foram classificados de acordo com duas características: propósito e escopo (Figura 1).

Um padrão de projeto pode ter propósito de criação, estrutural ou comportamental, em que cada propósito define a atuação do padrão. Padrões de projeto de criação atuam com a solução de problemas na criação de objetos. Padrões estruturais dizem respeito ao arranjo de classes e objetos no projeto e como são compostos. Os padrões comportamentais se referem à interação e distribuição de responsabilidades entre classes e objetos (GAMMA et al., 1995, p. 25).

Gamma et al. também definem a classificação dos padrões de projeto por escopo, dividindo-os entre padrões com foco principal nas classes e padrões focados em objetos. Os padrões de classe focam na herança entre classes e subclasses e, devido a essa característica, possuem relacionamentos estáticos formados em tempo de compilação. Os padrões com focos em objetos também aplicam a herança em algum momento, porém o ponto principal do padrão é a comunicação entre objetos, formando relacionamentos dinâmicos em tempo de execução.

1.3.1 Padrões Criacionais

1.3.1.1 *Factory Method*

O padrão de projeto *Factory Method* disponibiliza uma interface geral para criação de objetos, mas a classe do objeto a ser instanciado só é definida pelas subclasses. O nome

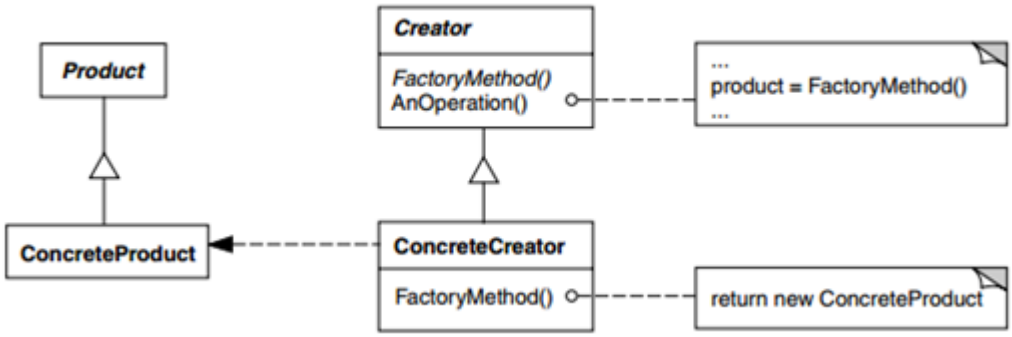
Figura 1 – Classificação dos Padrões de Projeto do GoF

		Propósito		
		Criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method	Adapter (class)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Fonte: (GAMMA et al., 1995, p. 20)

do padrão evidencia a existência de um método responsável pela criação de um objeto, ou sua "manufatura" (GAMMA et al., 1995, p. 113).

Figura 2 – Estrutura do *Factory Method*



Fonte: (GAMMA et al., 1995, p. 113)

Segundo Gamma et al., o uso do *Factory Method* elimina a necessidade de classes específicas vinculadas ao código, lidando apenas com a interface disponibilizada e permitindo a utilização de quaisquer classes concretas disponíveis. Além disso, o padrão

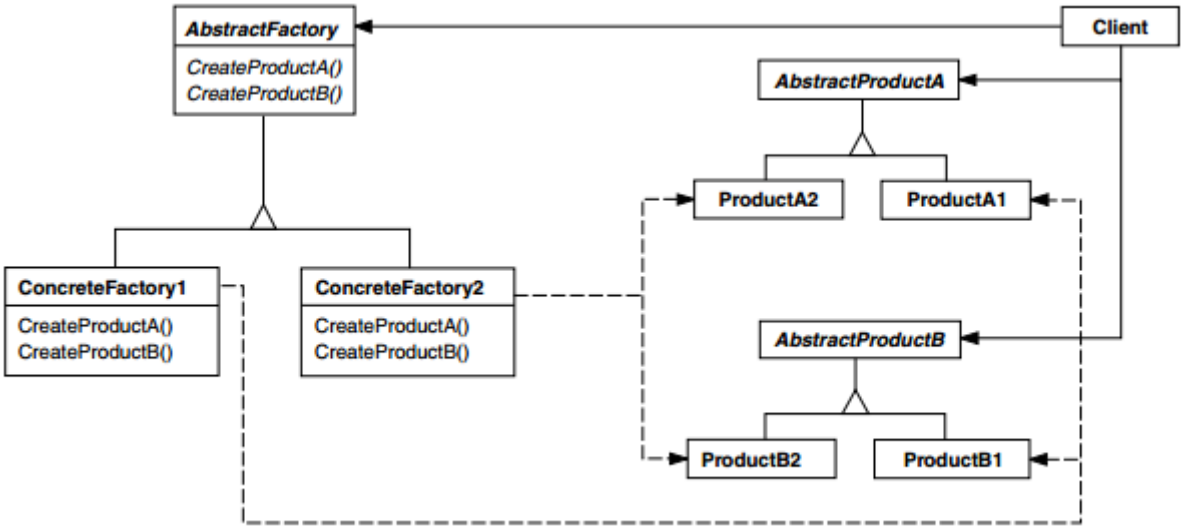
de projeto permite maior flexibilidade do código ao instanciar os objetos diretamente, facilitando a extensão de objetos por meio do método-fábrica (*Factory Method*).

1.3.1.2 *Abstract Factory*

O objetivo do padrão de projeto *Abstract Factory* é permitir a criação de famílias de objetos relacionados ou dependentes por meio de uma interface, sem que o cliente responsável pela chamada da operação tenha conhecimento das classes concretas que estão sendo utilizadas.

Gamma et al. sugerem a aplicação desse padrão em situações que um sistema precisa ser flexível quanto à criação e composição de seus objetos. Outro cenário citado para a utilização do *Abstract Factory* refere-se ao uso de uma biblioteca de classes que disponibilizam apenas suas interfaces, mas não fornecem acesso à sua implementação.

Figura 3 – Diagrama de Classes da Estrutura do *Abstract Factory*



Fonte: (GAMMA et al., 1995, p. 97)

As classes existentes na representação geral do *Abstract Factory* (Figura 3), definidas por Gamma et al., possuem os seguintes objetivos:

a) AbstractFactory: disponibiliza uma interface geral para a criação dos objetos abstratos;

b) ConcreteFactory1, ConcreteFactory2: implementam os métodos que criam objetos concretos;

c) AbstractProductA, AbstractProductB: interface para os tipos de objetos;

d) ProductA1, ProductA2, ProductB1, ProductB2: implementam a sua interface correspondente e definem o objeto concreto com base na sua classe (fábrica) concreta;

e) Client: acessa apenas as interfaces abstratas disponibilizadas nas classes AbstractFactory, AbstractProductA e AbstractProductB.

As principais consequências do padrão *Abstract Factory* são:

a) A fábrica encapsula as responsabilidades e a criação de objetos, isolando as classes concretas e suas implementações;

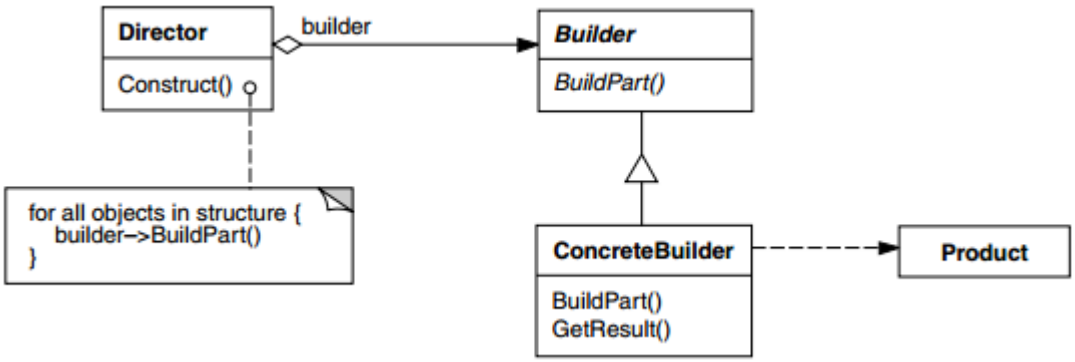
b) aumenta a flexibilidade na troca de famílias de objetos, já que uma fábrica concreta só aparece no momento em que é instanciada, facilitando possíveis mudanças;

c) a estrutura do padrão facilita que objetos sejam utilizados coerentemente, permitindo restrições, tais como impedir que objetos de famílias distintas sejam usados simultaneamente, por exemplo.

1.3.1.3 *Builder*

Segundo Gamma et al., o padrão *Builder* é utilizado na construção de um objeto complexo, em que subclasses têm a responsabilidade de implementar partes da construção do objeto, separando a criação do objeto e as partes que o compõem. Gamma et al. apresentam o conceito do *Builder* e suas classes da seguinte forma (Figura 4):

Figura 4 – Estrutura do padrão *Builder*



Fonte: (GAMMA et al., 1995, p. 105)

A classe *Builder* é a interface abstrata disponibilizada com as assinaturas dos métodos para a composição do objeto. As classes ConcreteBuilder representam as subclasses concretas que implementam a interface *Builder*. Cada ConcreteBuilder implementa partes de um objeto complexo específico e é responsável por retornar o resultado de sua execução

(o objeto construído). A classe Director constrói um objeto *Builder* e faz uma solicitação ao *Builder* quando uma parte do objeto complexo deve ser acrescentada. A classe Product é a representação do próprio objeto a ser construído e retornado pelo ConcreteBuilder.

Algumas das consequências do Padrão *Builder* são:

a) A representação interna de um objeto torna-se mais flexível, uma vez que basta a alteração ou a definição de um novo construtor para a criação de diferentes representações do objeto construído;

b) as partes responsáveis pela construção do objeto são encapsuladas. O cliente não tem acesso às informações de como o objeto é criado, apenas solicita a construção de um objeto ao Diretor.

Bloch sugere o uso do padrão *Builder* quando uma implementação exige muitos parâmetros no construtor da classe. Sem a abordagem desse padrão, variações de objetos com muitos parâmetros tornariam a criação do objeto sobrecarregada de parâmetros iniciais (obrigatórios ou opcionais), que dificultariam o entendimento do código. Outra alternativa seria a chamada de métodos após a criação do objeto para definição de seus atributos internos (conhecidos como setters), porém essa abordagem não é segura, pois tende a gerar *bugs* caso um atributo essencial não seja definido por um desses métodos após a criação do objeto. A abordagem do *Builder* permite que objetos com diferenças características sejam construídos apenas com variações em seus construtores, tornando o código mais legível e seguro, uma vez que sua construção está encapsulada (BLOCH, 2008, p. 15).

1.3.1.4 *Prototype*

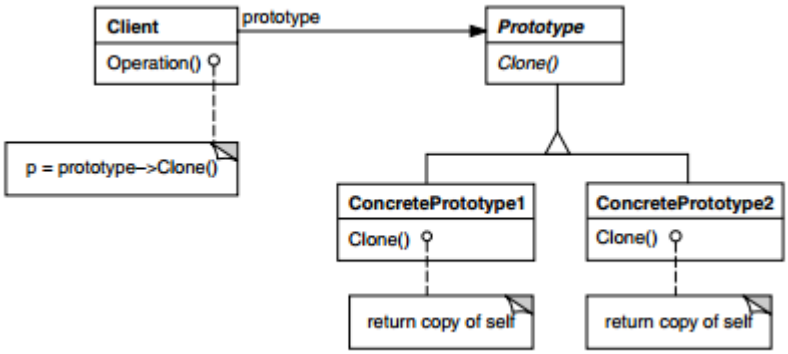
Gamma et al. definem a intenção do padrão de projeto *Prototype* como "especificar os tipos de objetos a serem criados usando uma instância-protótipo e criar novos objetos pela cópia desse protótipo" (GAMMA et al., 1995, p. 121).

Segundo a representação da estrutura do padrão por Gamma et al. (Figura 5), a classe *Prototype* disponibiliza uma interface para permitir a clonagem do próprio objeto. ConcretePrototype1 e ConcretePrototype2 são as classes concretas que implementam as operações para clonar a si próprio, cada uma clonando um tipo de objeto diferente. A classe Client apenas solicita ao protótipo a criação do clone.

Algumas das consequências do padrão *Prototype* são:

a) *Prototype* permite a inclusão e exclusão de objetos em tempo de execução, uma vez que apenas precisamos de uma instância de protótipo (que permita clonar-se) para a representação de uma nova classe concreta de objeto. Essa característica apresenta maior flexibilidade no padrão com relação aos outros

Figura 5 – Estrutura do Padrão *Prototype*



Fonte: (GAMMA et al., 1995, p. 123)

padrões criacionais. Gamma et al. sugerem a aplicabilidade dessa característica com o uso de carga dinâmica, especificando classes a serem instanciadas em tempo de execução;

b) número menor de subclasses no sistema, pois não precisa de classes-fábricas para a instanciação de objetos, apenas as implementações concretas que permitam clonar a si mesmo.

O Padrão *Prototype* possui características semelhantes ao *Abstract Factory* e ao *Builder*, como o encapsulamento da estrutura interna responsável pela construção dos objetos e a facilidade de adicionar novos objetos sem modificar a estrutura existente. Porém, o *Prototype* também pode ser usado em conjunto de outros padrões. Gamma et al. utilizam como exemplo o uso do *Abstract Factory* e *Prototype* juntos, em que uma *Abstract Factory* é responsável por armazenar um conjunto de protótipos que podem ser clonados e retornar novos objetos quando solicitado.

1.3.1.5 *Singleton*

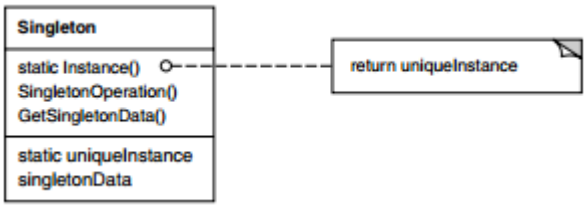
O padrão de projeto *Singleton* tem como objetivo principal garantir que haja apenas uma instância de uma classe no sistema e disponibilizar um acesso fácil a essa instância.

A classe *Singleton* (Figura 6) possui uma operação Instance que permite o acesso à instância *Singleton*, e a classe pode ou não ser responsável por sua própria instanciação. A classe deve garantir que chamadas externas do construtor não permitam instanciar outros objetos da classe *Singleton*.

Algumas das características do padrão *Singleton* são:

a) O padrão *Singleton* permite o acesso a uma única instância utilizando o conceito de encapsulamento, garantindo o controle sobre o objeto acessado, uma opção

Figura 6 – Estrutura do padrão de projeto *Singleton*



Fonte: (GAMMA et al., 1995, p. 130)

melhor que o uso de variáveis globais para representação de instâncias únicas;

b) *Singleton* facilita estender a classe utilizando subclasses que podem ser instanciadas conforme a necessidade em tempo de execução;

c) o acesso centralizado à instância *Singleton* pela operação da classe permite melhor flexibilidade da estrutura para aumentar ou limitar o sistema a um número específico de instâncias.

Alguns autores recomendam cuidado ao utilizar *Singletons*. Yener e Theedom argumentam que o uso excessivo de *Singletons* é um indício de mau uso da orientação a objetos, que pode acarretar em diversos problemas de desempenho com relação à limpeza de memória em linguagens de alto nível (YENER; THEEDOM, 2014, p. 34) . Nystrom sugere que o padrão *Singleton* tornou-se mais popular entre os desenvolvedores devido a sua simplicidade em comparação aos demais padrões apresentados por Gamma et al., o que resultou em um uso excessivo e, por diversas vezes, desnecessário do padrão. Nystrom aponta que o acesso global de um *Singleton* pode dificultar o entendimento de métodos que o utilizam, e que há situações em que projetos são sobrecarregados por *Singletons* que apenas são classes gerenciadoras (*managers*) que fornecem comportamentos a objetos de outra classe. Nesses casos, o ideal seria implementar esses comportamentos na própria classe que é gerenciada (NYSTROM, 2014, p. 81).

1.3.2 Padrões Estruturais

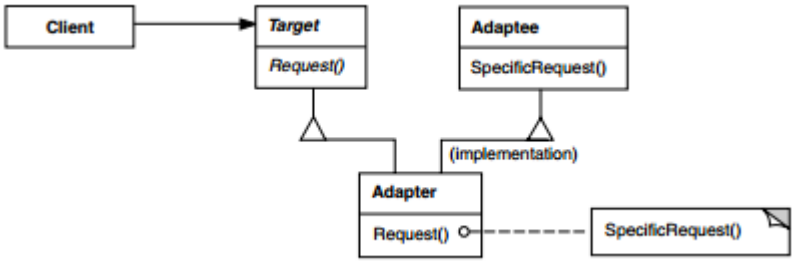
1.3.2.1 *Adapter*

Segundo Gamma et al., o objetivo do padrão de projeto *Adapter* é "converter uma interface de uma classe em outra interface, esperada pelos clientes". A interface adaptada permite que classes com interfaces incompatíveis trabalhem em conjunto.

O padrão *Adapter* possui estruturas diferentes para adaptações de classes e objetos. Adaptadores de classe utilizam o conceito de herança múltipla de classes para adaptar

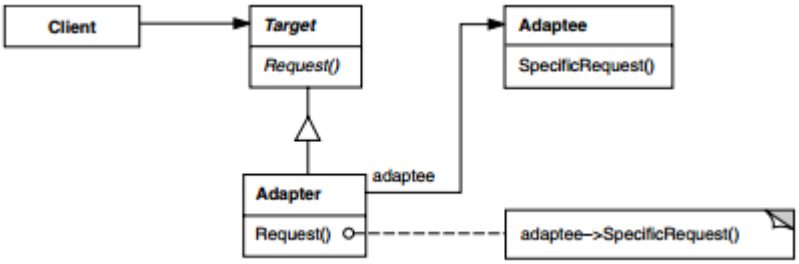
uma interface à outra (Figura 7). Já um adaptador de objetos utiliza a composição de objetos para atingir o objetivo do padrão (Figura 8).

Figura 7 – Estrutura do *Adapter* utilizando herança múltipla (Adaptador de classe)



Fonte: (GAMMA et al., 1995, p. 142)

Figura 8 – Estrutura do *Adapter* utilizando Composição de Objetos (Adaptador de objetos)



Fonte: (GAMMA et al., 1995, p. 142)

Ambas as representações de Gamma et al. utilizam Target, Adaptee e *Adapter* em suas estruturas para compor o padrão. Target disponibiliza a interface acessada pelo Client, Adaptee representa a interface específica a ser adaptada e *Adapter* é responsável por adaptar a interface de Adaptee à interface de Target.

O padrão *Adapter* tem diferentes consequências para adaptadores de classes e adaptadores de objetos. Adaptadores de classes utilizam herança para a adaptação, portanto um adaptador pode substituir operações da classe adaptada, por serem subclasses da classe adaptada. Contudo, a herança da classe concreta a ser adaptada impede que o adaptador adapte uma classe e todas as suas subclasses (GAMMA et al., 1995, p. 143).

Segundo Hall, o uso de *Adapter* empregando a composição de objetos é mais comum que a versão do padrão por herança, por ser uma abordagem mais flexível, já que a herança de classes gera uma dependência de implementação da classe adaptada, enquanto a composição de objetos apenas depende da interface a ser adaptada, podendo variar a sua implementação (HALL, 2014, p. 110).

1.3.2.2 *Bridge*

Gamma et al. definem a intenção do padrão *Bridge* como "Desacoplar uma abstração de sua implementação, de modo que as duas possam variar independentemente" (GAMMA et al., 1995, p. 151).

Existem diversas situações em que o desacoplamento da abstração e implementação é ponto crucial para garantir melhor flexibilidade e reusabilidade no sistema. Gamma et al. citam como exemplo a utilização de classes extensíveis por meio de subclasses, tanto para classes de abstração quanto classes concretas que efetivamente implementam as operações, em que o uso do padrão *Bridge* permite que as classes sejam extensíveis de forma independente.

Outro ponto a ser observado é quando classes implementadas não podem ser recompiladas, portanto alterações na abstração não devem comprometê-las. A estrutura do padrão *Bridge* também permite melhor flexibilidade para alterações na implementação em tempo de execução.

A estrutura do padrão *Bridge* (figura 9) é composta principalmente por Abstraction, Implementor, ConcreteImplementor e RefinedAbstraction. Abstraction é responsável por disponibilizar a interface da abstração e manter uma referência a um objeto da classe Implementor. Implementor representa a interface disponível para implementação, porém sua interface pode ser diferente de Abstraction, pois Implementor disponibilizará as operações para serem implementadas em um nível mais baixo, enquanto Abstraction é responsável por utilizar as operações do objeto Implementor e disponibilizá-las em uma interface de maior nível, repassando as suas solicitações para o objeto Implementor. A classe ConcreteImplementor implementa efetivamente a interface de Implementor, enquanto RefinedAbstraction estende a interface de Abstraction, desacoplando-a da implementação.

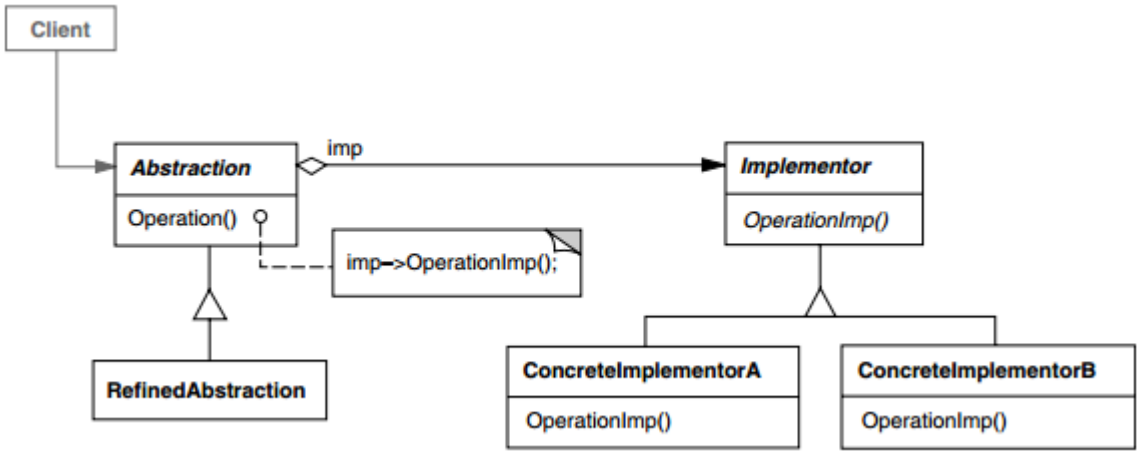
O padrão de projeto *Bridge* dá mais flexibilidade na solução de diversos problemas, em que a abstração e implementação devem ser independentes. *Bridge* remove as dependências que seriam encontradas nessas classes, possibilitando mudanças na implementação em tempo de execução, além de permitir que essas classes sejam estendidas independentemente. A estrutura do padrão *Bridge* também facilita a limitação do que a classe-cliente pode ou não acessar da implementação.

1.3.2.3 *Composite*

Segundo Gamma et al., o padrão de projeto *Composite* "permite aos clientes tratarem de maneira uniforme objetos individuais e composições de objetos.", compondo objetos em estruturas de árvore para representações hierárquicas parte-todo (GAMMA et al., 1995, p. 160).

Composite é um padrão utilizado para representar hierarquias parte-todo e que faci-

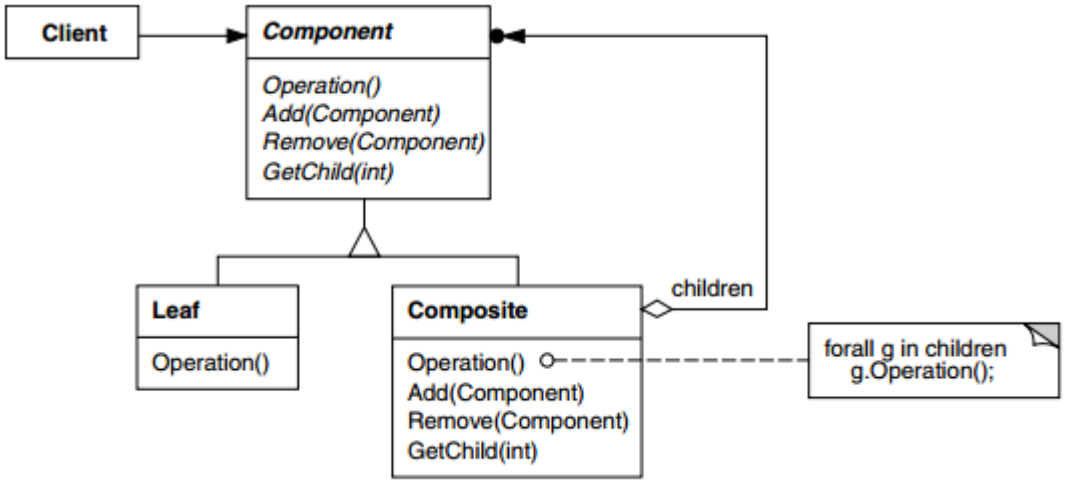
Figura 9 – Estrutura do Padrão *Bridge*



Fonte: (GAMMA et al., 1995, p. 153)

lita a representação de objetos que agregam outros objetos de forma uniforme, compondo-os recursivamente.

Figura 10 – Estrutura do Padrão *Composite*



Fonte: (GAMMA et al., 1995, p. 161)

A estrutura do padrão *Composite* (Figura 10) possui como principais classes: Component, Leaf, Composite e Client. A classe Component disponibiliza a interface para os objetos na composição, e define uma interface para acesso aos seus objetos-filho. A classe Leaf representa um objeto-folha e objetos primitivos na composição da árvore, pois não possui filhos. A classe Composite é responsável por armazenar os objetos-filho e implementar as operações relacionadas aos filhos definidas em Component. A classe Client

utiliza acessa e manipula objetos da estrutura utilizando a interface de Component.

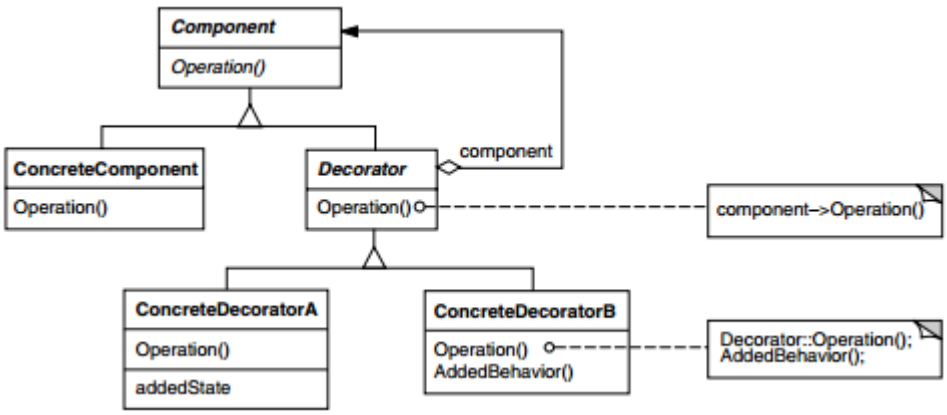
O padrão *Composite* facilita o tratamento de estruturas de objetos que possuem objetos primitivos e objetos compostos (formados por outros objetos), tratando-os todos uniformemente. Isso simplifica o código da classe cliente, uma vez que o cliente não precisa se importar se o objeto é do tipo folha ou se é um objeto composto.

Outra característica do padrão é a facilidade em adicionar novas subclasses como objetos primitivos ou compostos na estrutura, pois a classe-cliente não sofrerá alterações com novos componentes adicionados à hierarquia. No entanto, essa característica pode se tornar problemática em situações em que a hierarquia de componentes deve ser restringida. Nessas situações, é preciso garantir a restrição na composição utilizando verificações internas em tempo de execução (GAMMA et al., 1995, p. 163).

1.3.2.4 *Decorator*

Segundo Gamma et al., o padrão de projeto *Decorator* é "uma alternativa flexível ao uso de subclasses para extensão de funcionalidades". Um *Decorator* atribui novas responsabilidades a um objeto específico de forma dinâmica, sem alterar os demais objetos da classe. O padrão *Decorator* pode ser utilizado quando o uso de subclasses para representação de um grande número de combinações de um objeto se tornaria muito complexa (GAMMA et al., 1995, p. 172).

Figura 11 – Estrutura do Padrão *Decorator*



Fonte: (GAMMA et al., 1995, p. 172)

A estrutura do padrão é formada pelas interfaces Component, ConcreteComponent, Decorator e ConcreteDecorator. Component disponibiliza a interface para objetos que podem ser decorados. ConcreteComponent implementa a interface de Component. *Decorator* também possui uma interface compatível à interface de Component e mantém

uma referência a um objeto do tipo Component, enquanto ConcreteDecorator implementa as características que podem ser adicionadas ao objeto decorado.

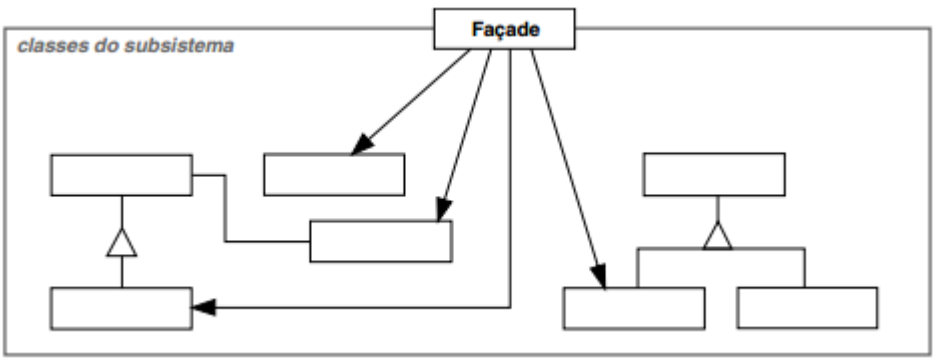
O padrão *Decorator* aumenta a flexibilidade para adicionar ou remover novas funcionalidades a objetos em tempo de execução, em comparação à utilização de herança múltipla estática. Com *Decorator*, acrescentamos características apenas associando-as a um objeto, permitindo combinações independentes entre as características. A mesma solução com herança múltipla padrão exigiria mais subclasses e tornaria a implementação e gerenciamento muito complexos. Além disso, o padrão permite que objetos obtenham apenas as características eventualmente desejadas, evitando uma sobrecarga de atributos e operações obtidos por herança múltipla que não serão utilizados.

Hall relaciona o uso de *Decorator* a um bom exemplo do princípio de responsabilidade única. Esse princípio define que classes com mais de uma responsabilidade devem ser divididas em classes menores com responsabilidade única, pois facilitam a manutenção de código e possibilita que as classes sejam estendidas. De outra forma, classes com múltiplas responsabilidades tendem a tornar a estrutura de um projeto impraticável, pois torna-se difícil de manter o código e implementar novas funcionalidades. Ao utilizar o padrão, cada classe *Decorator* é responsável por implementar uma funcionalidade única (HALL, 2014, p. 184).

1.3.2.5 *Façade*

Gamma et al. apresentam o padrão *Façade* com a intenção de "fornecer uma interface unificada para um conjunto de interfaces em um subsistema." (GAMMA et al., 1995, p. 179). O conceito principal do padrão é disponibilizar uma interface mais simples (um objeto *façade* – fachada) para acesso aos recursos de um subsistema, diminuindo a complexidade de utilizá-lo.

Figura 12 – Estrutura do Padrão *Façade*



Fonte: (GAMMA et al., 1995, p. 181)

A estrutura do padrão é formada pelo objeto *Façade* e as classes do subsistema. As classes clientes fazem as solicitações ao objeto *Façade*, que solicita as operações aos objetos das classes do subsistema para realizarem a solicitação da classe-cliente. As classes do subsistema continuam sendo as responsáveis por executar as ações solicitadas pela classe-cliente, porém o *Façade* pode implementar outras ações necessárias para atender as interfaces das classes do subsistema.

O uso do padrão *Façade* centraliza o acesso às classes de um subsistema utilizando um objeto-fachada, que simplifica o uso do subsistema pelas classes-cliente. Outro ponto positivo do padrão é o baixo acoplamento das dependências entre classes-cliente e classes de implementação dos subsistemas, o que consequentemente facilita mudanças futuras nos subsistemas sem comprometer as classes-cliente.

O padrão *Façade* não impede o acesso às classes do subsistema, apenas facilita o seu uso por intermédio do objeto *façade*. Caso haja a necessidade de acessar diretamente operações específicas de classes do subsistema, as classes-cliente podem referenciá-las diretamente sem o objeto *façade* (GAMMA et al., 1995, p. 182).

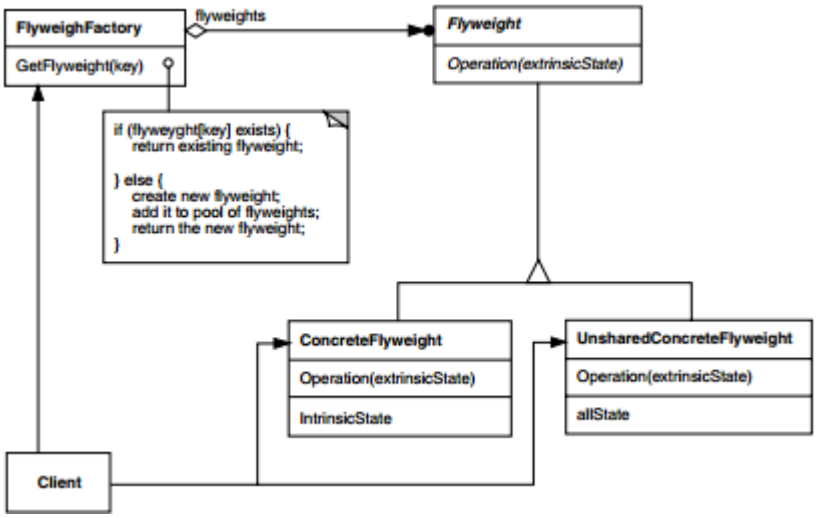
1.3.2.6 *Flyweight*

A intenção do padrão de projeto *Flyweight*, segundo Gamma et al., é "usar compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina" (GAMMA et al., 1995, p. 187).

O conceito principal do padrão *Flyweight* é possibilitar o uso de objetos de forma compartilhada em cenários em que a quantidade de objetos poderia causar gargalos no sistema e prejudicar o seu desempenho. Para atingir esse objetivo, o objeto a ser compartilhado precisa ser independente do contexto em que é utilizado. Gamma et al. apontam que os estados intrínsecos e extrínsecos dos objetos devem ser considerados ao utilizar o padrão. O estado intrínseco do objeto diz respeito ao estado que é independente do cenário em que o objeto é utilizado, portanto pode ser compartilhado. Já o estado extrínseco depende do cenário do *Flyweight*, então não é compartilhado, mas a informação do estado pode ser passada para o objeto por uma classe cliente.

A estrutura do padrão *Flyweight* (Figura 13) é representada pelos seguintes elementos: Flyweight, FlyweightFatory, ConcreteFlyweight e UnsharedConcreteFlyweight. Flyweight define uma interface para que os estados extrínsecos dos flyweights possam ser passados a eles. ConcreteFlyweight é a classe do objeto compartilhado, que implementa a interface de Flyweight e possui apenas o estado intrínseco do objeto. UnsharedConcreteFlyweight representa classes na estrutura do *Flyweight* que não são compartilhadas. Gamma et al. enfatizam que o padrão de projeto *Flyweight* permite o compartilhamento, mas não o obriga ou o garante.

Figura 13 – Estrutura do Padrão *Flyweight*



Fonte: (GAMMA et al., 1995, p. 190)

Na estrutura definida por Gamma et al., FlyweightFactory é responsável por criar os objetos *Flyweight* e gerenciar o compartilhamento de objetos sempre que uma classe Client solicita um novo objeto. Caso o objeto solicitado já exista, ele é compartilhado pelo *Flyweight*, senão o objeto é instaciado pelo *Flyweight* e poderá ser compartilhado posteriormente por outro Client; em ambos os casos o Client apenas possui a referência do objeto *Flyweight*. O Client é responsável por tratar o estado extrínseco do objeto e passá-lo ao *Flyweight*.

Uma vantagem perceptível no uso do padrão *Flyweight* em um sistema é a substituição de um grande número de objetos instanciados por objetos compartilhados entre vários clientes, o que representa uma grande economia no armazenamento de objetos dependendo do cenário em que o padrão é aplicado. Contudo, é preciso considerar alguns pontos importantes antes de tomar a decisão de utilizar o *Flyweight* para solução de um problema.

Gamma et al. sugerem o uso do padrão *Flyweight* quando o custo de armazenamento dos objetos é muito alto, em situações em que muitos objetos poderiam ser substituídos por uma quantidade muito menor de objetos compartilhados com poucos estados extrínsecos e quando a identidade de objetos não é importante no contexto da aplicação.

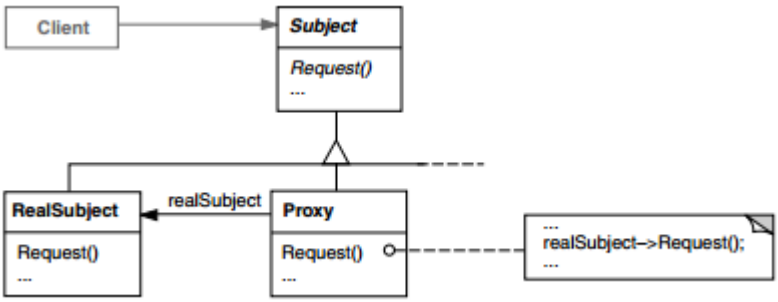
1.3.2.7 *Proxy*

O padrão *Proxy*, segundo Gamma et al., "fornece um substituto ou marcador da localização de outro objeto para controlar o acesso a esse objeto" (GAMMA et al., 1995, p. 198). É um padrão de projeto utilizado para aprimorar ou tornar mais versátil a referência

a objetos.

O padrão *Proxy* possui diferentes aplicações, algumas delas apresentadas por Gamma et al. Um *Remote Proxy* utiliza um objeto representante local para um objeto em outro endereço. Um *Virtual Proxy* utiliza o *proxy* para adiar a instanciação de um objeto de alto custo de memória ou desempenho, apenas instanciado quando houver a necessidade. Um *Protection Proxy* é usado quando é preciso controlar o direito de acesso ao objeto. Um *Smart Reference* é usado como um apontador mais sofisticado, que possui ações quando ao acessar um objeto (GAMMA et al., 1995, p. 200).

Figura 14 – Estrutura do Padrão *Proxy*



Fonte: (GAMMA et al., 1995, p. 200)

Na estrutura do padrão *Proxy* (Figura 14), Proxy mantém uma referência ao objeto real, RealSubject, e pode ser responsável por gerenciá-lo, adicioná-lo ou excluí-lo. Subject disponibiliza uma interface comum para Proxy e RealSubject, o que possibilita o uso do Proxy nos locais em que espera-se o uso de um objeto RealSubject.

Nos casos em que um tipo específico de *Proxy* é utilizado, ele possui responsabilidades adicionais. Em um *Remote Proxy*, *Proxy* precisa gerenciar as operações solicitadas e enviadas pelo objeto real que está em outro endereço. *Virtual Proxy* pode armazenar informações específicas do objeto real para disponibilizá-las a um cliente que solicitá-las, em vez de acessar o objeto. Um *Protection Proxy* pode verificar se o cliente que solicitou uma operação possui acesso a ela.

O padrão *Proxy* disponibiliza um acesso indireto a um objeto, e as diferenças entre os tipos de *Proxy* podem aprimorar o acesso a objetos ou agir como um limitador intencional às operações do objeto, como é o caso do *Protection Proxy*. Fowler enfatiza que o uso de um *Virtual Proxy* permite a representação de um objeto idêntico ao objeto que o cliente pretende acessar. No entanto, o uso inconsequente dessa característica pode gerar problemas de identidade do objeto real em situações específicas, já que o *Proxy* representa indiretamente um objeto, mas não é o objeto real (FOWLER, 2002, p. 201).

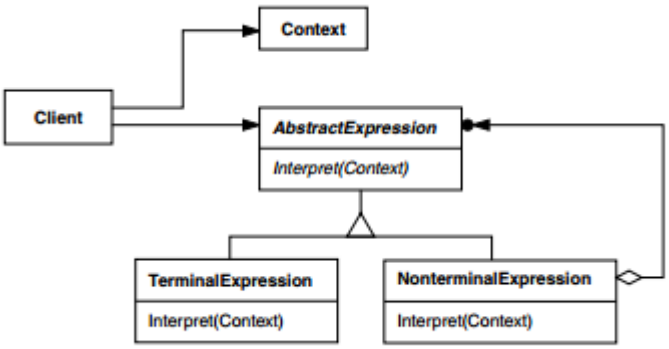
1.3.3 Padrões Comportamentais

1.3.3.1 *Interpreter*

Gamma et al. apresentam o padrão de projeto *Interpreter* como uma solução para problemas que podem ser representados por uma linguagem e a sua interpretação. Segundo Gamma et al., a intenção do padrão *Interpreter* é "dada uma linguagem, definir uma representação para a sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças dessa linguagem" (GAMMA et al., 1995, p. 231).

Problemas que apresentam uma linguagem simples que precisa ser interpretada são os cenários mais propícios para a utilização do *Interpreter*. Gamma et al. sugerem o uso de outras técnicas para a interpretação de linguagens mais complexas, uma vez que o padrão *Interpreter* utilizará subclasses para a representação da gramática da linguagem, o que tornaria a representação de linguagens muito complexas impraticável. Outro fator a ser considerado é a eficiência do *Interpreter* na interpretação da linguagem, principalmente nos casos em que outras implementações podem ser mais eficientes do que a representação por subclasses em árvore de análise sintática (GAMMA et al., 1995, p. 234).

Figura 15 – Estrutura do Padrão *Interpreter*



Fonte: (GAMMA et al., 1995, p. 234)

Gamma et al. definem a estrutura básica do padrão *Interpreter* conforme a figura 15. A classe Client executa o método Interpret, enquanto as classes TerminalExpression e NonterminalExpression implementam o método Interpret disponibilizado pela interface de AbstractExpression. Há uma classe do tipo NonterminalExpression para cada regra da gramática apresentada na linguagem, e a classe é utilizada recursivamente para interpretar essas regras. As representações terminais, ou seja, não formadas por expressões da linguagem, são representadas por classes do tipo TerminalExpression. Gamma et al. utilizam a classe Context como *container* de informações relevantes no contexto do interpretador.

A estrutura utilizada para a implementação do padrão *Interpreter* permite que mudanças sejam feitas com facilidade na forma que a linguagem é interpretada. Gamma et al. enfatizam que, ao utilizar classes para a representação de expressões e literais da gramática de uma linguagem, novas expressões podem ser adicionadas ou expressões já existentes podem ser reutilizadas estendendo as classes da gramática. Embora a solução apresentada se encaixe bem para a representação e interpretação de linguagens de gramática simples, para os casos em que uma linguagem é mais complexa e possui muitas expressões (consequentemente exigindo mais classes em sua implementação), Gamma et al. recomendam o uso de outras abordagens, tais como técnicas para implementação de compiladores (GAMMA et al., 1995, p. 235).

1.3.3.2 *Template Method*

O padrão de projeto *Template Method* tem por objetivo "definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para as subclasses." (GAMMA et al., 1995, p. 301). Por meio de um método-template, o padrão permite que uma estrutura seja definida com operações abstratas que serão implementadas com o código específico de cada subclasse.

Segundo Gamma et al., o *Template Method* pode ser utilizando para situações em que parte do código é única e pode ser compartilhada por todas as subclasses, enquanto outra parte da implementação possui variações. As operações abstratas implementam o código especializado e as demais operações são generalizadas. Outra situação de aplicabilidade apresentada por Gamma et al. é a necessidade de estender subclasses em um estrutura. O padrão permite que um método-template implemente operações *hook* (gancho) como forma de extensão em partes específicas da estrutura de classes. Gamma et al. explicam que uma operação gancho pode ou não possuir uma implementação inicialmente, que pode ser estendida por subclasses quando necessário.

Na estrutura do padrão de projeto *Template Method*, AbstractClass é responsável por declarar a interface das operações abstratas e disponibilizar o método-template, definindo o esqueleto do algoritmo com as chamadas das operações abstratas (PrimitiveOperation1, PrimitiveOperation2). As classes ConcreteClass são responsáveis pela implementação das operações abstratas, com as variações específicas de cada subclasse.

Gamma et al. apontam como principal consequência do *Template Method* a possibilidade de reutilização de código, devido à característica de postergar a implementação de operações variantes às subclasses e generalizar o código que pode ser compartilhado.

Outro ponto apresentado por Gamma et al. ao utilizar o *Template Method* é a escolha entre a implementação de operações abstratas e operações gancho. As operações abstratas de uma classe devem ser redefinidas pelas subclasses, enquanto uma subclasse pode ou não redefinir as operações gancho. A definição dessas operações é importante

para a execução do método-template e na implementação das subclasses.

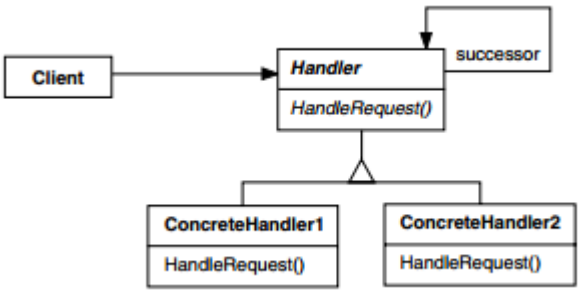
1.3.3.3 *Chain of Responsability*

Gamma et al. definem a intenção do padrão *Chain of Responsability*: "evitar o acoplamento do remetente de uma solicitação ao seu receptor, ao dar a mais de um objeto a oportunidade de tratar a solicitação." (GAMMA et al., 1995, p. 212). Esse padrão apresenta o conceito de "cadeia de responsabilidade" (*chain of responsability*) para possibilitar que uma solicitação seja atendida por um objeto associado a uma cadeia de objetos com uma interface em comum.

No *Chain of Responsability*, um objeto remetente não precisa ter conhecimento do objeto que tratará a sua solicitação. O primeiro objeto da cadeia verificará se pode atender à solicitação ou se a repassará para o objeto seguinte da cadeia. A solicitação é repassada pela cadeia de objetos até que um objeto possa realizá-la.

Segundo Gamma et al., o padrão *Chain of Responsability* pode ser utilizado quando um ou mais objetos podem satisfazer uma solicitação e não há conhecimento do objeto que a tratará; quando uma solicitação será emitida para um objeto e não há a necessidade de especificar o receptor ou quando o conjunto de objetos que podem tratar uma solicitação é especificado dinamicamente (GAMMA et al., 1995, p. 214).

Figura 16 – Estrutura do Padrão *Chain of Responsability*



Fonte: (GAMMA et al., 1995, p. 214)

Na estrutura apresentada por Gamma et al. (Figura 16), Handler é a interface comum aos objetos capazes de tratar uma solicitação, e pode implementar um vínculo para o sucessor. ConcreteHandler1 e ConcreteHandler2 são classes de objetos que podem tratar solicitações e podem acessar o seu sucessor na cadeia.

Client faz uma solicitação a um objeto da cadeia. Se um objeto pode tratar uma solicitação que chegou a sua posição na cadeia, ele a tratará, senão o objeto repassa a solicitação ao próximo objeto na cadeia.

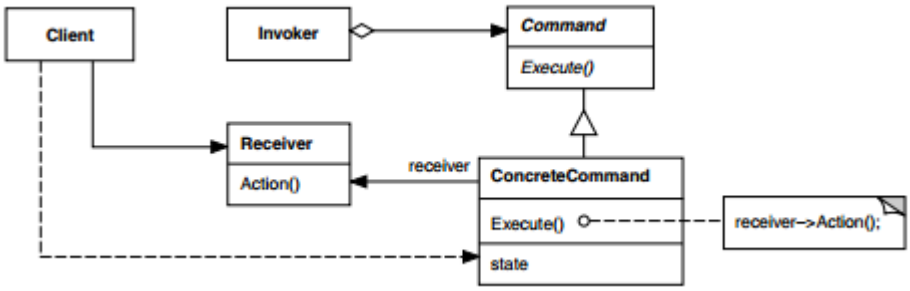
Gamma et al. apontam como uma característica do *Chain of Responsability* o baixo acoplamento entre objetos receptores e remetentes, uma vez que ambos não possuem conhecimento explícito de suas existências, e objetos receptores possuem referência apenas ao seu sucessor na cadeia.

A estrutura do padrão permite que a cadeia de objetos seja flexível, adicionando ou alterando a cadeia em tempo de execução para alterar a capacidade no tratamento de solicitações. Gamma et al. enfatizam o fato que a cadeia de objetos não garante que uma solicitação será atendida. A ausência de uma referência direta ao objeto receptor permite que uma solicitação passe por toda a cadeia de objetos e não seja tratada, por configuração incorreta da cadeia, por exemplo (GAMMA et al., 1995, p. 215).

1.3.3.4 *Command*

A intenção do padrão de projeto *Command*, segundo Gamma et al., é "encapsular uma solicitação como um objeto, desta forma permitindo parametrizar clientes com diferentes solicitações, enfileirar ou fazer o registro (log) de solicitações e suportar operações que podem ser desfeitas." (GAMMA et al., 1995, p. 222). A ideia de encapsular solicitações como objetos permite o desacoplamento entre o objeto que executa ações e o solicitante da operação a ser executada.

Figura 17 – Estrutura do Padrão *Command*



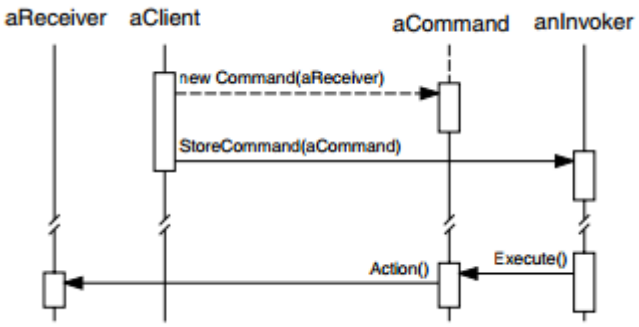
Fonte: (GAMMA et al., 1995, p. 225)

Na estrutura do padrão *Command* apresentada por Gamma et al. (Figura 17), *Command* disponibiliza uma interface comum de execução de uma ação para as subclasses ConcreteCommand, que efetivamente implementarão o método Execute, invocando a ação de um Receiver.

Um Client é responsável pela instanciação de um ConcreteCommand e definir o seu Receiver. O objeto Receiver é o objeto que efetivamente executará a ação pelo método Action. Um Invoker armazena um objeto ConcreteCommand e é responsável pela solicitação da execução a um objeto *Command*. O objeto ConcreteCommand pode possuir uma referência de estado *state* para situações em que um comando deve ser desfeito.

A comunicação entre os objetos no padrão *Command* é representada no diagrama de sequência da Figura 18 (GAMMA et al.). Gamma et al. enfatizam que o diagrama facilita a visualização do desacoplamento entre Invoker, Receiver e a execução da ação.

Figura 18 – Diagrama de Sequência representando o padrão *Command*



Fonte: (GAMMA et al., 1995, p. 226)

Além do desacoplamento entre objetos que executam ações e os solicitantes das ações, a estrutura do padrão *Command* também facilita a adição de novos comandos apenas adicionando novas classes, sem alteração de código existente. O padrão também permite que comandos sejam compostos para a realização de ações (GAMMA et al., 1995, p. 226).

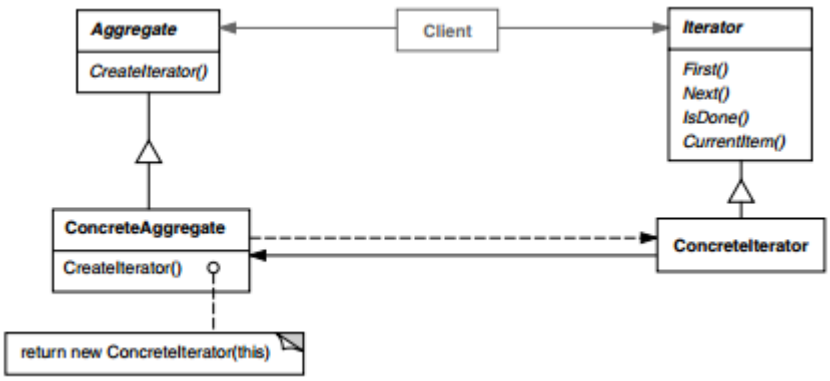
1.3.3.5 *Iterator*

Segundo Gamma et al., a intenção do padrão *Iterator* é "fornecer um meio de acessar, sequencialmente, os elementos de um objeto agregado sem expor a sua representação subjacente" (GAMMA et al., 1995, p. 244).

O padrão *Iterator* mostra-se uma boa opção quando existe a necessidade de acessar um objeto em um agregado (uma lista, por exemplo) e a adição de operações na interface do próprio conjunto para realizar essa tarefa sobrecarregaria a classe do objeto. O padrão permite que a responsabilidade de acesso aos elementos seja destinada a um objeto *Iterator* (iterador). Dessa maneira, o objeto *Iterator* pode possuir diferentes formas de acesso aos objetos, sem que isso afete a classe do objeto agregado.

O objeto *Iterator* possui conhecimento da posição atual na sequência de objetos, e múltiplas implementações de iteradores permitem que a navegação nos objetos possa ser feita de formas diferentes. Segundo Gamma et. al, o padrão também utiliza o conceito de iteração polimórfica, utilizando classes abstratas e concretas de *Iterator* para separar a implementação do agregado e dos objetos *Iterator* (GAMMA et al., 1995, p. 245).

Figura 19 – Estrutura do Padrão *Iterator*



Fonte: (GAMMA et al., 1995, p. 246)

Na estrutura apresentada por Gamma et al., o *Iterator* define uma interface para acesso aos elementos de um agregado (Aggregate). Agreggate disponibiliza a interface para o agregado criar um objeto *Iterator*. ConcreteAggregate implementa a interface para a criação de um ConcreteIterator e retorna o *Iterator* específico. ConcreteIterator implementa as operações concretas de *Iterator* e é responsável por manter a referência ao elemento atual na sequência do agregado.

O padrão *Iterator* permite acessar objetos de um agregado sem que a sua estrutura interna seja exposta. Além disso, o uso do *Iterator* mantém a classe do objeto agregado simplificada, uma vez que as operações para percorrer os elementos do agregado são definidas no objeto iterador.

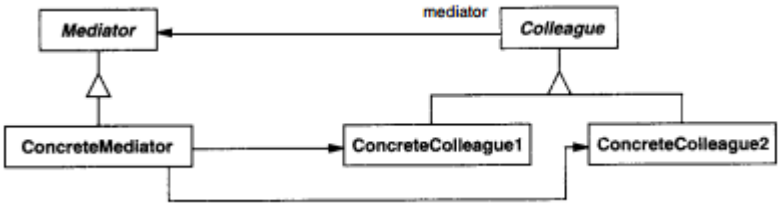
Outra característica do padrão é a facilidade em alterar a forma como os elementos do agregado são percorridos, pois basta alterar a instância do objeto *Iterator* utilizada. O padrão também permite que o número de subclasses sejam ampliadas para variar o acesso aos elementos.

1.3.3.6 *Mediator*

A intenção do padrão *Mediator*, segundo Gamma et al., é "definir um objeto que encapsula a forma como um conjunto de objetos interage" (GAMMA et al., 1995, p. 257). O padrão utiliza um objeto *mediator* (mediador) para gerenciar os relacionamentos e conexões entre diversos objetos, fazendo com que os objetos apenas interajam com o mediador, que será responsável pelo tratamento das comunicações entre esse grupo de objetos.

O padrão *Mediator* é aplicável em situações nas quais as interações entre objetos de um determinado grupo são complexas, de difícil compreensão ou podem complicar a reutilização de objetos (GAMMA et al., 1995, p. 260).

Figura 20 – Estrutura do Padrão *Mediator*



Fonte: (GAMMA et al., 1995, p. 260)

A estrutura do padrão *Mediator* apresentada por Gamma et al. utiliza uma interface *Mediator* para definir as operações que um objeto mediador deve implementar para gerenciar as interações entre os objetos. A classe Colleague define uma interface para ser implementada pelas classes ConcreteColleague.

A classe ConcreteMediator implementa a comunicação entre os objetos ConcreteColleague. Nota-se pelo diagrama que as classes ConcreteColleague1 e ConcreteColleague2 apenas relacionam-se com o ConcreteMediator, mantendo o gerenciamento na classe do mediador. Todas as interações que seriam feitas entre objetos ConcreteColleague são tratadas e repassadas pelo mediador.

Gamma et al. enfatizam que o padrão *Mediator* substitui interações entre objetos muitos-para-muitos para interações um-para-muitos (com o mediador centralizando as comunicações entre objetos), melhorando a compreensão da estrutura de interações entre objetos. No entanto, apesar da vantagem com a diminuição do acoplamento entre os objetos, a centralização das comunicações no objeto mediador pode torná-lo muito complexo e difícil de manter (GAMMA et al., 1995, p. 261).

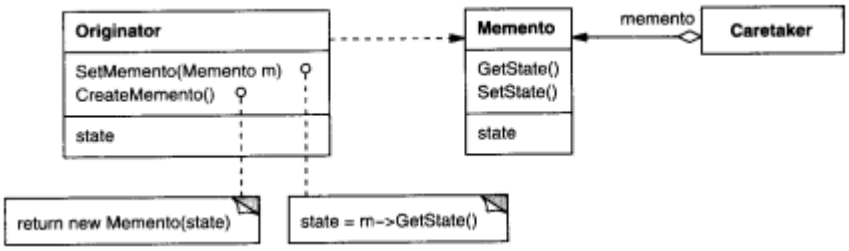
1.3.3.7 *Memento*

O padrão *Memento*, segundo Gamma et al. , é utilizado com o intuito de "sem violar o encapsulamento, capturar e externalizar um estado interno de um objeto, de maneira que o objeto possa ser restaurado para esse estado mais tarde" (GAMMA et al., 1995, p. 266).

Existem situações num projeto de software em que precisamos recuperar estados anteriores de um objeto, como salvar informações de alterações ou desfazer ações, no entanto é necessário respeitar o princípio do encapsulamento de objetos, que diz respeito à limitação do acesso às informações internas inerentes a um objeto. O padrão *Memento* apresenta uma estrutura (Figura 21) que possibilita a recuperação de estados de um objeto sem afetar o encapsulamento, utilizando o conceito de *memento* (recordação) e originador.

Segundo Gamma et al., *memento* (da classe Memento) é um objeto que armazena

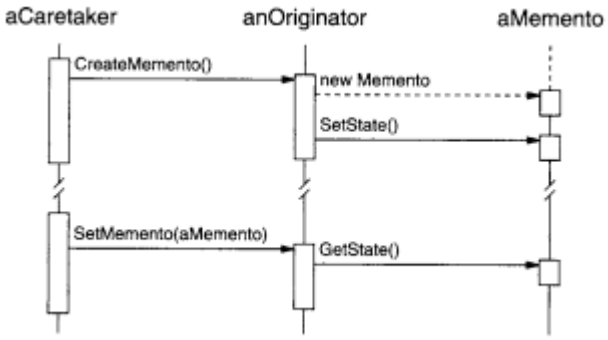
Figura 21 – Estrutura do Padrão *Memento*



Fonte: (GAMMA et al., 1995, p. 268)

o estado interno de um outro objeto, o originador (classe Originator). Em sua estrutura básica, o padrão *Memento* permite que apenas o originador seja capaz de instanciar objetos *mementos* e recuperar estados por meio deles. O Caretaker é responsável por gerenciar os objetos *mementos*, porém tem acesso limitado às suas interfaces. O diagrama de sequência a seguir facilita a compreensão da comunicação entre os objetos *memento*, *originator* e *caretaker* (GAMMA et al., 1995, p. 269).

Figura 22 – Diagrama de Sequência de Exemplo do Padrão *Memento*



Fonte: (GAMMA et al., 1995, p. 269)

Gamma et al. enfatizam que o uso do *Memento* permite o acesso aos estados de um objeto, que outrora seria indisponível, por meio do uso de objetos *memento*, sem afetar o encapsulamento dos objetos. O padrão *Memento* também tira a responsabilidade do objeto originador de manter os estados necessários, já que a responsabilidade é passada ao objeto *caretaker*, simplificando a interface do originador.

Outro ponto levantado por Gamma et al. é referente ao custo computacional do armazenamento de *mementos*. Caso os objetos originadores precisem realizar muitas cópias de informações para repassar um estado ao *memento*, ou a quantidade de acesso e recuperação de *mementos* por um originador é muito alta, o custo de armazenamento e

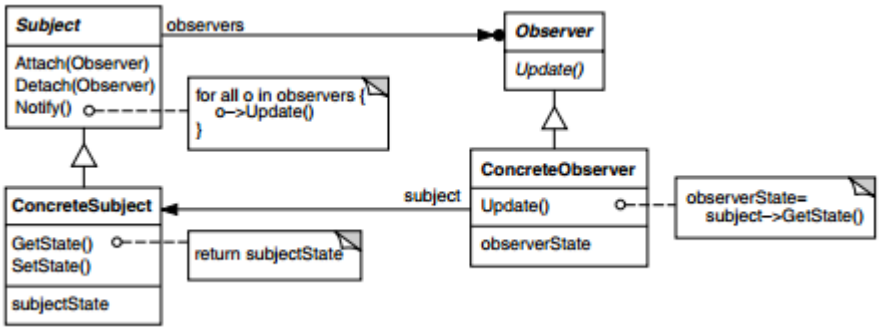
processamento pode se tornar problemático. É necessário analisar com cautela a viabilidade do padrão para garantir que o encapsulamento e o acesso aos *mementos* sejam operações baratas (GAMMA et al., 1995, p. 269).

1.3.3.8 *Observer*

A intenção do padrão de projeto *Observer*, segundo Gamma et al., é "definir uma dependência um-para-muitos entre objetos, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente" (GAMMA et al., 1995, p. 274).

O padrão *Observer* é aplicável nos casos em que a mudança de um objeto afeta vários outros, mas as interações entre esses objetos não devem ser fortemente acopladas. Segundo Gamma et al., outra aplicabilidade do padrão está em abstrações que possuem dois aspectos dependentes, que serão separados em objetos diferentes para facilitar a mudanças e possam variar independentemente.

Figura 23 – Estrutura do Padrão *Observer*



Fonte: (GAMMA et al., 1995, p. 275)

O padrão *Observer* utiliza o conceito de *Subject* (sujeito) e *Observer* (observador) em sua estrutura. Um *Subject* pode possuir muitos *Observers*, que serão notificados sempre que houver uma alteração de estado no *Subject*. Os *Observers* também podem solicitar mudanças no estado do *Subject*.

No diagrama de classes acima, *Subject* disponibiliza uma interface para adicionar e remover *Observers*, além de notificar alterações em seu estado. ConcreteSubject implementa o acesso aos estados de um *Subject*. *Observer* disponibiliza uma interface para atualização dos observadores de um *Subject*. ConcreteObserver implementa a atualização do estado de um *Observer* e mantém uma referência ao *Subject* para manter seu estado consistente ao estado do *Subject*. Sujeitos não conhecem as classes concretas dos observadores, apenas sua interface simples. Isso permite uma série de variações entre observadores e sujeito, tornando o projeto mais flexível e fracamente acoplado.

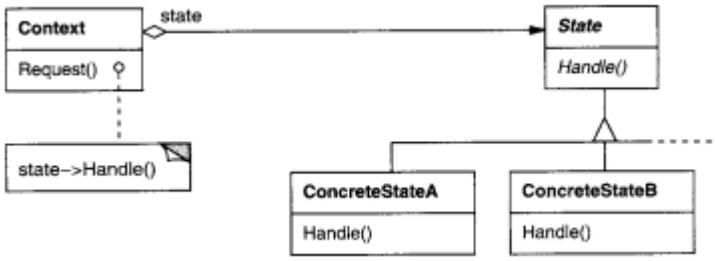
É preciso cautela na implementação do padrão com múltiplos *observers*, pois uma vez que observadores não se comunicam, uma alteração em um *Subject* pode encadear uma sequência lenta e não esperada de atualizações em *Observers* (GAMMA et al., 1995, p. 277). Nystrom também enfatiza o comportamento síncrono da implementação clássica de *Observer*, em que um *Subject* poderia parar a execução de seus observadores, mas sugere que o custo de execução do padrão não é elevado para a maioria das aplicações, exceto projetos de software em que o tempo de resposta é crítico (NYSTROM, 2014, p. 49).

1.3.3.9 *State*

Segundo Gamma et al., o padrão *State* "permite a um objeto alterar seu comportamento quando o seu estado interno muda. O objeto parecerá ter mudado sua classe" (GAMMA et al., 1995, p. 284).

O padrão *State* é utilizado quando um objeto tem diferentes comportamentos de acordo com seu estado, alterados em tempo de execução. O padrão também é aplicável para representar estados como objetos em abstrações que dependem do estado do objeto e possuem muitas variações de operações e condições.

Figura 24 – Estrutura do Padrão *State*



Fonte: (GAMMA et al., 1995, p. 285)

A estrutura do padrão apresentada por Gamma et al. utiliza um objeto Context acessado pelos clientes e mantém uma referência a um objeto ConcreteState que representará o seu estado. *State* define a interface de um estado de Context. As classes ConcreteState implementam o comportamento de cada estado de Context.

O padrão *State* facilita a adição de novos estados a um objeto, já que os comportamentos de cada estado são divididos em subclasses de *State*. Além disso, as transições entre estados são mais evidentes que apenas mudanças de variáveis. A mudança de estado é explicita e representada por um objeto de estado. Um ponto levantado por Gamma et al. refere-se à quantidade de subclasses que a implementação do padrão pode gerar. Em um primeiro momento, tal característica do padrão pode parecer uma desvantagem na implementação, porém o ganho com a facilidade de alteração e adição de novos se

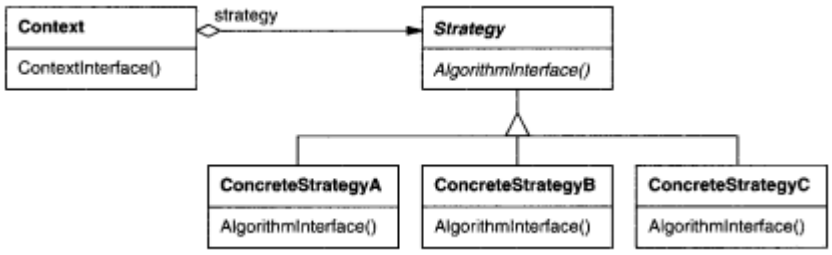
sobressai em comparação a implementações que utilizariam operações condicionais (if e switchs) por todo o código para tratamento dos estados (GAMMA et al., 1995, p. 286).

1.3.3.10 *Strategy*

Gamma et al. apresentam a intenção do padrão *Strategy* "definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis" (GAMMA et al., 1995, p. 292).

O padrão *Strategy* (estratégia) é aplicável quando a única diferença entre um conjunto de classes relacionadas é o seu comportamento. O padrão permite separar esses diferentes comportamentos em estratégias que podem ser utilizadas por uma classe. Outra aplicação do padrão é na implementação de variações de algoritmos. O padrão permite o uso de uma estratégia diferente para cada variação do algoritmo, criando uma hierarquia de classes de algoritmos (GAMMA et al., 1995, p. 293).

Figura 25 – Estrutura do Padrão *Strategy*



Fonte: (GAMMA et al., 1995, p. 294)

Na estrutura do padrão *Strategy* definida por GAMMA et al., *Strategy* é uma interface comum entre todas as estratégias utilizadas. As classes ConcreteStrategy implementam o algoritmo específico por meio da interface de *Strategy*. Context mantém uma referência a um objeto *Strategy*, que pode variar entre diferentes ConcreteStrategy para obter diferentes estratégias.

Segundo Gamma et al., o padrão *Strategy* fornece uma alternativa mais flexível e mais compreensível que o uso de herança e subclasses na implementação de comportamentos. *Strategy* também diminui o número de comandos condicionais no código, uma vez que as estratégias estão separadas em objetos. Se os diferentes comportamentos fossem implementados em uma única classe, eventualmente haveria comandos condicionais por todo o código para definir o comportamento utilizado.

Gamma et al. também apontam algumas deficiências do padrão *Strategy*, como o aumento no número de objetos e a necessidade que os clientes saibam da existência das

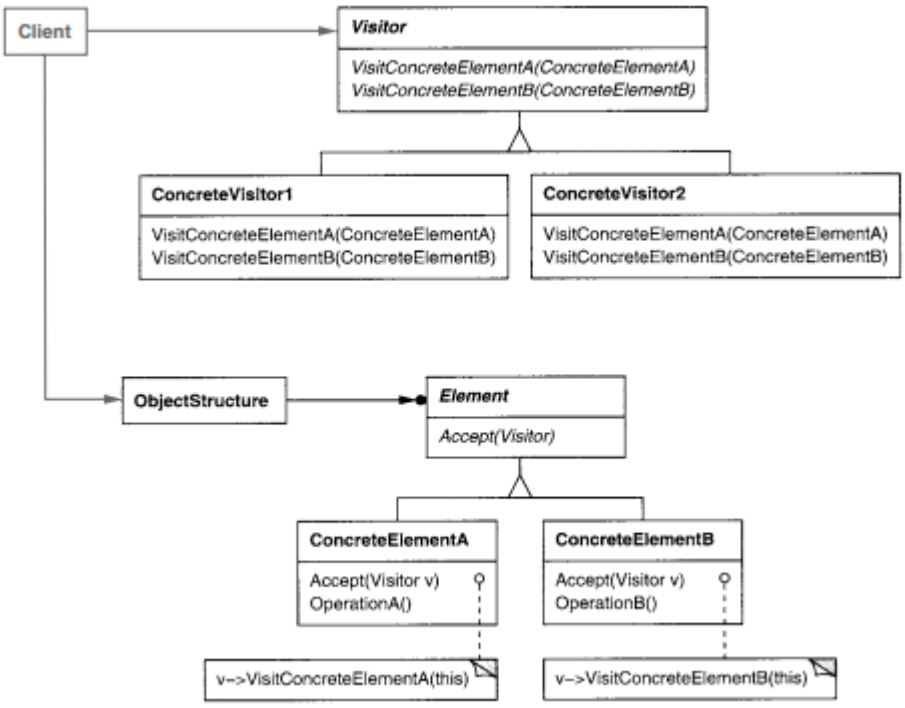
diferenças entre as estratégias para utilizá-las, o que pode expor detalhes de implementação das estratégias ao cliente (GAMMA et al., 1995, p. 296).

1.3.3.11 *Visitor*

Gamma et al. afirmam que a intenção do padrão de projeto *Visitor* é "representar uma operação a ser executada nos elementos de uma estrutura de objetos". Segundo Gamma et al., o padrão "permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera" (GAMMA et al., 1995, p. 305).

O padrão *Visitor* é usado quando uma estrutura de objetos é formada por classes de objetos com interfaces diferentes que precisam realizar operações nos objetos. *Visitor* permite que seja definida uma estrutura paralela à estrutura de elementos para que sejam definidas as operações sobre os objetos (GAMMA et al., 1995, p. 307). Outra aplicabilidade do padrão é separar as operações sobre os elementos da implementação da estrutura de objetos, seja para evitar que as classes de estrutura fiquem sobrecarregadas com as operações.

Figura 26 – Estrutura do Padrão *Visitor*



Fonte: (GAMMA et al., 1995, p. 308)

A estrutura do Padrão *Visitor* apresentada por Gamma et al. tem como base a interface *Visitor*, que será implementada pelas classes concretas ConcreteVisitor. *Visitor* declara uma operação para cada elemento concreto (ConcreteElement) da estrutura de

dados. ConcreteElement implementa um método Accept da interface Element para aceitar um visitante. Quando um objeto da estrutura é visitado, ele invoca o método de visita do visitante que recebe o próprio objeto da estrutura como parâmetro, permitindo que possíveis alterações no estado interno do objeto sejam feitas (GAMMA et al., 1995, p. 309).

Segundo Gamma et al., o padrão *Visitor* facilita a adição de novas operações sobre uma estrutura de objetos, apenas adicionando um novo visitante. No entanto, o uso do padrão dificulta a adição de novos objetos na estrutura de objetos, uma vez adicionar um novo elemento à estrutura causa impacto nas interfaces de *Visitor* e ConcreteVisitor. Gamma et al. orientam a utilizar o padrão apenas nos casos em que a estrutura de objetos é estável e as operações que são aplicadas sobre os objetos sofrem mudanças ou adições constantes. Outro fator a ser considerado na implementação do *Visitor* é o encapsulamento dos objetos da estrutura a ser visitada. Os visitantes acessam os elementos da estrutura para executarem suas operações, o que pode expor o estado interno dos objetos.

2 UMA INTRODUÇÃO AO *GAME ENGINE* UNITY

O Unity é um motor de jogo (game engine) utilizado para o desenvolvimento de jogos 2D e 3D que permite a exportação do jogo para as principais plataformas de dispositivos móveis, VR, desktop e console como Windows, Mac, Linux, Android, iOS, Playstation 4, Xbox One, Wii U, entre outras. É muito popular na comunidade de desenvolvimento de jogos, por permitir que desenvolvedores criem jogos na ferramenta gratuitamente, por meio de uma licença pessoal (Unity Personal Edition), desde que a receita bruta anual dos desenvolvedores não ultrapasse US$ 100 mil. O Unity também possui licenças pagas específicas para empresas de grande porte, abrangendo o público-alvo do motor de jogo.

2.1 Interface do Unity

O Unity possui um editor flexível e personalizável para o desenvolvimento de jogos, que pode ser expandido por meio de extensões desenvolvidas pela comunidade para obter novas funcionalidades no fluxo de trabalho utilizando a API do Unity. A interface básica do Unity é organizada da seguinte forma:

a) Project Window: é a janela responsável por exibir os diferentes assets – itens que podem ser utilizados no desenvolvimento do jogo, como modelos 3D, arquivos de áudio, imagens – disponíveis para uso no projeto;

b) Scene Window: janela que exibe os objetos adicionados a uma cena atual do jogo no Unity, permitindo que o desenvolvedor possa manipular de diferentes maneiras os diversos objetos como câmeras, itens de cenário e personagens, por exemplo;

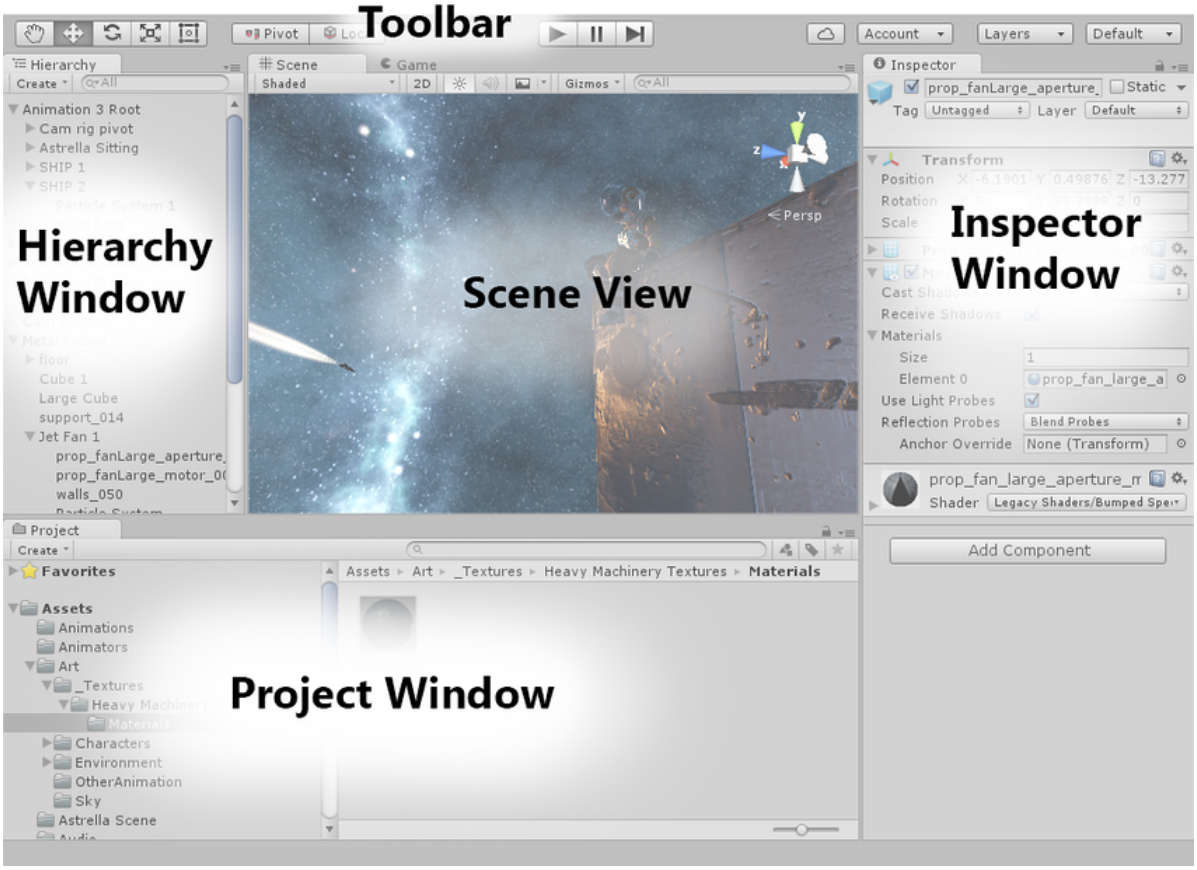
c) Hierarchy Window: a janela exibe os objetos adicionados à cena atual do jogo de forma hierárquica, representando visualmente os objetos que são formados por outros subobjetos;

d) Inspector Window: exibe as propriedades de um objeto selecionado, que podem variar conforme o tipo do objeto;

e) ToolBar: é uma barra fixa que disponibiliza as funções básicas de manipulação no Unity. Toolbar permite alterar a disposição de janelas do Unity com layouts pré-definidos ou salvar o seu próprio layout. Uma das principais ferramentas

disponíveis no Toolbar é o controle para iniciar/pausar a simulação do jogo dentro do motor de jogo.

Figura 27 – Diferentes janelas que formam a interface básica do Unity



Fonte: (UNITY, 2017d)

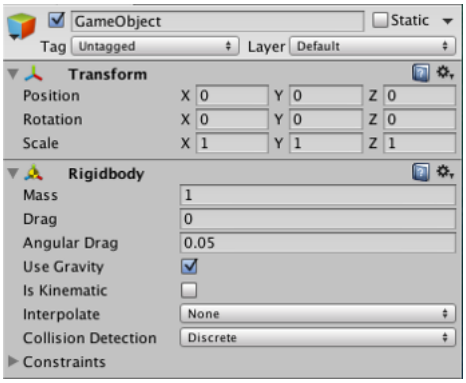
2.2 Fluxo de Trabalho do Unity

O Unity apresenta o conceito de cenas (Scenes) para facilitar a criação de jogos no seu editor. As cenas são utilizadas no Unity como agregadores de objetos do jogo. O desenvolvedor pode utilizar as cenas no Unity de diversas maneiras, mas um exemplo clássico de uso é a construção de uma fase de um jogo. Uma cena pode representar uma fase específica do jogo e diversos objetos são alocados dentro da cena, como personagens controlados pelos jogadores e objetos do cenário (nuvens, árvores, etc.). Outros objetos existentes na cena podem representar efeitos específicos como emissão de partículas, iluminação ou fazer parte da GUI (Graphical User Interface – Interface Gráfica do Usuário) do jogo. Os objetos em cena são chamados de GameObjects. GameObjects sempre possuem um componente Transform associado a eles, que representa sua posição, rotação e escala no ambiente da cena.

Os objetos adicionados às cenas normalmente possuem características e comportamentos (fazer um personagem ser afetado pela gravidade, por exemplo) que são adicionados no Unity utilizando componentes (Components) vinculados ao objeto para obter o comportamento adicional desejado.

O Unity disponibiliza diversos componentes pré-definidos que permitem ao desenvolvedor implementar funcionalidades a um objeto rapidamente e sem a necessidade de escrever linhas de código para isso. A interface de usuário do Unity facilita a inclusão rápida desses componentes por meio de um Inspector, um painel que exibe os componentes associados a um determinado objeto e permite a alteração das propriedades iniciais desses componentes para adaptar o seu funcionamento de acordo com as necessidades do desenvolvedor.

Figura 28 – Exemplo de Inspector de um GameObject, com os componentes Transform e Rigidbody



Fonte: (UNITY, 2017g)

Embora os componentes padrões do Unity facilitem o desenvolvedor iniciante a começar a construção de um jogo na ferramenta, eventualmente será necessário desenvolver os seus próprios componentes.

O Unity permite que os desenvolvedores construam seus próprios componentes utilizando Scripts vinculados aos objetos do jogo. Os scripts também são exibidos no Inspector de um objeto ao qual estão associados.

O Unity utiliza internamente uma versão do Mono para o desenvolvimento dos scripts no Unity Editor. Mono é uma plataforma *open source* (distribuída com a licença MIT) criada com base no .NET Framework da Microsoft. O Mono é multiplataforma, o que permite que desenvolvedores possam criar aplicações com classes compatíveis com as classes do .NET Framework e que executam em ambientes como o Windows, Linux, Mac OS X, Nintendo Wii, Sony PlayStation 3, entre outros (MONO, 2017).

Nos scripts, todos os componentes e objetos referenciados pelo Unity herdam de

uma classe base chamada UnityEgine.Object. Os objetos que compõem uma cena no Unity são objetos da classe GameObject, que derivam de Object.

Ao criar um novo script no Unity, um arquivo de script será criado com uma classe que deriva de MonoBehaviour. Todos os scripts criados no Unity derivam de MonoBehaviour.

Figura 29 – Exemplo da estrutura básica de um Script no Unity

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {

        // Use this for initialization
        void Start () {

        }

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

        }
}
```

Fonte: (UNITY, 2017f)

A estrutura básica de um script recém-criado no Unity possui dois métodos principais: Start() e Update(). Os dois métodos são adicionados automaticamente ao incluir um novo script e, devido à derivação da classe de MonoBehaviour, o motor de jogo irá tratá-los internamente de forma diferente de outros métodos adicionados pelo desenvolvedor e serão executados em momentos específicos. O método Start() é executado pelo Unity antes de o jogo ser iniciado e é utilizado para inicialização de atributos do GameObject. Update() é executado a cada frame do jogo e é utilizado para tratar as interações do objeto ao longo do tempo no jogo, como o controle de movimento de personagens, por exemplo.

Assim como os componentes pré-definidos do Unity possuem propriedades que podem ser editadas no Inspector do objeto, os atributos públicos dos scripts das classes de MonoBehaviour também podem ser exibidos no Inspector para que o desenvolvedor possa manipulá-los na cena e ver os resultados em tempo de execução do jogo.

Os scripts adicionados a GameObjects também são considerados pelo motor de jogo como componentes de um objeto da cena. A forma como os objetos e os componentes do Unity se relacionam faz parte de uma abordagem do motor de jogo que pode ser mais facilmente compreendida com o conceito do padrão *Component*.

2.3 Unity e o Padrão *Component*

O padrão *Component*, também conhecido como *Entity-Component* (Entidade-Componente), é um padrão arquitetural que permite a uma entidade "abranger múltiplos domínios sem acoplá-los entre si" (NYSTROM, 2014, p. 213).

O padrão *Component* define que uma entidade é capaz de possuir múltiplos domínios, que são separados em classes de componentes para evitar o acoplamento, tornando a entidade um *container* de componentes. Evitar acoplamento é uma preocupação recorrente em projetos de jogos, já que o desenvolvimento de um jogo envolve a implementação de uma série de domínios completamente distintos, como o código para o áudio, a simulação de física e a renderização de imagens, por exemplo.

A abordagem do padrão *Component* difere da implementação tradicional de herança de classes e respeita a ideia apresentada por Gamma et al. de utilizar a composição de objetos em um projeto em vez do uso de herança de classes. A vantagem dessa abordagem fica evidente em projetos de jogos, que frequentemente sofrem alterações por intervenções externas e precisam de um código flexível e de alta reusabilidade. Estruturar as classes de objetos em um jogo apenas usando a herança de classes pode tornar o projeto pouco flexível no futuro e resultar em refatorações de código caso haja mudanças no projeto (BILAS, 2002).

Segundo Nystrom, *Component* pode ser utilizado em projetos de jogos para a representação de uma classe base que define as entidades na implementação do jogo. Esse exemplo de uso pode ser observado na classe GameObject do Unity, que representa um objeto na cena e pode agregar diversos componentes pré-construídos pelo motor de jogo para adicionar novos comportamentos ao objeto, além de permitir a criação de novos componentes por meio dos scripts.

Nystrom enfatiza que, separando os diferentes domínios em classes de componentes distintas, o padrão *Component* permite que sejam criadas diversas variações de entidades com componentes reutilizáveis.

Um ponto observador por Nystrom na implementação do padrão *Component* diz respeito à comunicação entre os componentes de uma entidade. Embora o desacoplamento entre os componentes seja importante, eventualmente será necessário que os componentes de um objeto se comuniquem entre si, e isso deve ser levado em consideração na implementação de um projeto. Por exemplo, um componente que representa a barra de energia de um personagem na interface de usuário pode precisar da informação sobre a quantidade de energia disponível no personagem, que estaria disponível no componente que gerencia a sua energia.

Nystrom sugere diferentes abordagens para o tratamento da comunicação entre componentes de um objeto: alterar o estado do objeto que é composto pelos componentes,

pela referência direta entre eles, e utilizando a troca de mensagens entre componentes.

No cenário em que componentes alteram estados do objeto, um componente específico que solicita a informação do objeto não precisa necessariamente saber quem alterou o estado do objeto, apenas possui conhecimento da existência do objeto-entidade. Essa abordagem mantém o desacoplamento entre os componentes, porém ela precisa que todas as informações de componentes compartilháveis por qualquer outro componente sejam disponibilizadas pelo objeto-entidade, o que pode sobrecarregar o objeto com estados que são relevantes apenas para uma parte de seus componentes.

Outra desvantagem dessa abordagem é que a ordem dos componentes pode impactar diretamente o comportamento do objeto. Como os estados do objeto são compartilhados e alterados pelos componentes, a reorganização dos componentes pode fazer com que um componente específico acesse um estado com um valor diferente do esperado, e isso pode resultar em *bugs* difíceis de rastrear.

Ao sugerir a referência direta entre componentes que precisam se comunicar, Nystrom afirma que a comunicação torna-se simples e direta, com apenas uma chamada ao método do componente referenciado, sem poluir a classe do *container* de componentes. No entanto, os componentes que precisam se comunicar tornam-se acoplados por uma referência na classe do componente, perdendo parte da flexibilidade esperada pelo uso do padrão *Component*.

A última abordagem sugerida por Nystrom, a comunicação entre os componentes por meio de mensagens, baseia-se em um sistema de mensagens associado ao objeto entidade, que permita aos componentes que se comuniquem entre si. No exemplo dado por Nystrom, os componentes implementam uma interface que permite receber mensagens e o objeto *container* é responsável por transmitir as mensagens aos componentes. Nystrom afirma que a implementação de um sistema de mensagens em que o *container* de componentes é responsável pela transmissão delas aplica o conceito do padrão *Mediator*, em que o objeto *container* é o mediador entre as comunicações dos componentes. Essa abordagem mantém os componentes fracamente acoplados, apenas com a mensagem transmitida como acoplamento, porém a sua implementação pode não ser trivial.

O motor de jogo Unity já possui uma implementação que permite a transmissão de mensagens entre componentes utilizando um método da classe GameObject chamado SendMessage. O método SendMessage recebe um nome de método como parâmetro. SendMessage chamará todos os métodos dos componentes de GameObject que possuírem o mesmo nome passado por parâmetro.

No exemplo de SendMessage (Figura 30), todos os componentes do gameObject que possuírem um método chamado ApplyDamage terão seu método executado. O segundo parâmetro de SendMessage permite passar valores ao método executado nos componentes.

Figura 30 – Exemplo de uso do método SendMessage

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour {
    void ApplyDamage(float damage) {
        print(damage);
    }
    void Example() {
        gameObject.SendMessage("ApplyDamage", 5.0F);
    }
}
```

Fonte: (UNITY, 2017f)

Embora o funcionamento do SendMessage seja muito próximo ao conceito apresentado por Nystrom, existem objeções ao uso do método devido ao impacto que suas chamadas recorrentes podem causar na performance do jogo (DUNSTAN, 2016). Outra desvantagem de SendMessage é a forma como o parâmetro referente ao método a ser executado é passado. Por se tratar de uma variável *String*, o compilador não alertará o desenvolvedor caso haja algum erro de digitação no nome do método passado, além de ser necessário alterar o valor da *String* caso haja alguma mudança no nome do método. Uma alternativa ao uso do SendMessage é a implementação de um sistema de mensagens próprio utilizando o padrão *Observer* (ZUCCONI, 2015).

2.4 Boas Práticas de Desenvolvimento no Unity

Embora o uso adequado do padrão *Component* coloque o desenvolvedor em direção ao desacoplamento e a alta reusabilidade em um projeto no Unity, ainda existem outras práticas ao longo da evolução de um projeto que podem tornar o gerenciamento dos objetos caótico. Um exemplo é o vínculo entre objetos da cena a um atributo no script de um componente, arrastando o objeto da cena manualmente até o atributo visível no Inspector do objeto que possui o script. A princípio, o uso desta ferramenta do Unity facilita a referência de objetos existentes na cena em componentes de outros objetos. No entanto, projetos maiores normalmente possuem um número muito alto de objetos, o que inviabiliza esse processo manualmente (ANDRUSHKO, 2015). Além disso, os objetos tornam-se fortemente acoplados, tornando o projeto bem menos flexível do que a abordagem de *Component* propõe.

Uma das tentativas para evitar problemas futuros de implementação durante o

desenvolvimento de projetos no Unity é a utilização de boas práticas de programação. A definição de classes com responsabilidade única e corretamente modularizadas são algumas delas (HALL, 2014). Outra abordagem é a separação de componentes de interface e a lógica do jogo (TULLEKEN, 2016).

Essas práticas levam novamente à discussão sobre a implementação de classes e a forma como são projetadas. As abordagens citadas podem utilizar os conceitos já consolidados no desenvolvimento de software orientado a objetos, como interfaces e classes abstratas, para diminuir o acoplamento e desenvolver classes reutilizáveis (BARCELO, 2013).

Os padrões de projeto descritos por Gamma et al. podem ser aplicados para auxiliar na resolução desses problemas específicos no desenvolvimento de projetos no Unity. No entanto, as diferentes características dos padrões criacionais, estruturais e comportamentais e os seus possíveis resultados, positivos ou negativos, devem ser consideradas pelo desenvolvedor antes de definir soluções para as dificuldades encontradas na implementação de projetos no motor de jogo. A combinação do fluxo de trabalho por componentes e o uso adequado dos padrões de projeto é um dos pontos-chave para o desenvolvimento de projetos bem estruturados e com alta reusabilidade no Unity.

3 A RELAÇÃO ENTRE PADRÕES DE PROJETO E O UNITY

3.1 Padrões Criacionais no Unity

3.1.1 *Object Pooling*

Existem situações em projetos de jogos em que a quantidade de objetos que são criados ou destruídos durante o jogo é muito alta, tal como representações de inimigos, projéteis ou objetos que fazem parte dos efeitos visuais do jogo, como efeitos de partícula física. Contudo, a criação e/ou exclusão de muitos objetos podem gerar problemas de desempenho no jogo devido à constante alocação e desalocação dos objetos em memória, gerando possíveis problemas de fragmentação de memória ou queda de FPS (*frames per second* – quadros por segundo, em tradução livre) no jogo.

O padrão de projeto *Object Pooling* propõe o reuso de objetos instanciados em contraponto à criação e destruição de objetos em memória. Por meio de uma classe *pool* (reserva, em tradução livre), objetos são armazenados e podem ser solicitados quando necessário.

A implementação do padrão deve permitir o acesso à disponibilidade dos objetos de um *pool*. Ao ser criado um novo *pool*, todos os seus objetos são instanciados com um estado "disponível". Quando um novo objeto é solicitado ao *pool*, o objeto é retornado e torna-se "indisponível"; quando o objeto não for mais necessário, seu estado torna-se "disponível" novamente. O funcionamento da classe *pool* permite que objetos sejam usados quando necessário, sem que haja a necessidade de mudanças de alocação em memória (NYSTROM, 2014, p. 307).

Nystrom sugere o uso do *Object Pooling* em casos em que a alocação de objetos é lenta e a frequência de criação e destruição de objetos é alta, ambos cenários recorrentes em projetos de jogos. No Unity, o custo de inicialização e remoção de objetos em uma cena pode ser alto devido à forma como a memória do motor de jogo é gerenciada. O gerenciamento de memória no Unity é feito pelo próprio motor do Mono, que possui um sistema de *Garbage Collection* – um processo invisível ao desenvolvedor, que identifica e remove automaticamente a alocação de memória que não é mais utilizada (UNITY, 2017e). Embora o *Garbage Collection* seja vantajoso ao desenvolvedor por gerenciar a memória alocada e evitar que objetos não utilizados ocupem espaço desnecessariamente, um aumento no número de execuções do *Collector* pode exigir um alto processamento da CPU e, consequentemente, pode gerar lentidão no projeto (UNITY, 2017e).

O uso do padrão *Object Pooling* reduz o impacto de execuções do *Garbage Collector*, porém Nystrom sugere cautela ao desenvolvedor que pretende implementar o padrão em um ambiente que já possui um sistema para gerenciamento de alocação de memória, como o Unity. Por exemplo, Nystrom afirma que objetos utilizados do *pool*, que façam referências a outros objetos, podem impedir que o *Garbage Collector* destrua os objetos referenciados, mesmo que não façam parte do *pool*. Uma solução para essa situação é a remoção das referências a outros objetos ao retornar um objeto ao *pool* (NYSTROM, 2014, p. 310).

Outro ponto a ser considerado na implementação de Object Pool diz respeito ao tamanho e flexibilidade do *pool*. Nystrom afirma que o tamanho de um *pool* deve estar de acordo com as necessidades do jogo. Por exemplo, criar um *pool* que comporta 200 inimigos, mas o jogo raramente utilizará mais do que 50 inimigos ao mesmo tempo resulta em um *pool* com vários objetos que não serão utilizados e apenas ocupam espaço em memória.

Pools muito pequenos também podem gerar problemas, uma vez que a quantidade de objetos utilizados ao mesmo tempo pode ultrapassar a disponibilidade do *pool*. Nesse caso, Nystrom apresenta abordagens distintas. Para objetos importantes e que podem exceder o tamanho do *pool*, pode-se utilizar um *pool* dinâmico para evitar o uso de memória por objetos não utilizados. Outra abordagem é a remoção de um objeto, retornando-o ao *pool*, para que um novo seja usado. Essa alternativa pode ser válida para os casos em que a ausência do objeto já existente é menos impactante do que a não aparição do objeto a ser criado, como a não execução de um determinado som no jogo, por exemplo (NYSTROM, 2014, p. 308). Nystrom também aponta os casos em que a criação de um novo objeto, após a indisponibilidade de todo o *pool*, talvez não cause tanto impacto. Nystrom utiliza o exemplo de um *pool* de partículas de efeitos visuais no jogo que ficaria completamente indisponível, porém todas as partículas estariam em execução e visíveis para o jogador, que provavelmente não se incomodaria com a ausência de novas partículas. Para cada caso, as características e as consequências do *Object Pooling* devem ser avaliadas.

3.1.2 *Singleton*

O padrão *Singleton* é utilizado em implementações de projetos no Unity principalmente para facilitar a comunicação entre componentes usando métodos estáticos para acessar objetos por meio de qualquer script. Respeitando o conceito básico apresentado por Gamma et al., a estrutura do padrão permite que apenas uma instância do objeto exista na cena do Unity, e que possa ser acessado a qualquer momento por outros componentes. *Singleton* pode ser uma alternativa mais segura que o uso indiscriminado de variáveis estáticas no projeto (MURRAY, 2014).

O exemplo mostrado por Tulleken (Figura 31) permite que outras classes que derivam da classe *Singleton* também se tornem *Singletons*.

Figura 31 – Exemplo de *Singleton* no Unity

```
public class Singleton<T> : MonoBehaviour where T : MonoBehaviour
{
   protected static T instance;

   //Returns the instance of this singleton.
   public static T Instance
   {
      get
      {
         if(instance == null)
         {
            instance = (T) FindObjectOfType(typeof(T));

            if (instance == null)
            {
               Debug.LogError("An instance of " + typeof(T) +
                  " is needed in the scene, but there is none.");
            }
         }

         return instance;
      }
   }
}
```

Fonte: (TULLEKEN, 2016)

Alguns exemplos de aplicações de *Singletons* no Unity podem ser vistos em objetos de gerenciamento da cena. Murray utiliza o padrão para a implementação de um controlador de instanciação de objetos na cena. Tulleken defende o uso de *Singletons* para classes gerenciadoras de áudio, GUI (Graphic User Interface) e efeitos de partículas, por exemplo, porém não recomenda o uso do padrão para prefabs e objetos que não sejam gerenciadores do jogo (TULLEKEN, 2016).

Outra aplicação do padrão sugerida por Tulleken é o uso de *Singletons* para facilitar a inicialização de objetos compartilhados entre cenas. Esses objetos *Singletons* podem ser persistentes entre as cenas, permitindo o compartilhamento de informações relevantes ao jogo em diversas fases.

Andrushko aponta a vantagem do uso de *Singletons* para acesso a informações de gerenciadores, sem a necessidade de vincular o objeto *Singleton* a outros scripts pelo editor do Unity, evitando erros que poderiam acontecer caso o objeto não fosse vinculado

corretamente entre os componentes no editor (ANDRUSHKO, 2015).

Alguns autores também consideram o *Singleton* como um padrão mal utilizado, em grande parte devido a sua facilidade de implementação por desenvolvedores menos experientes. Um ponto relevante levantado por Nystrom é referente ao uso excessivo de gerenciadores representados por *Singletons*, em situações que o *Singleton* disponibiliza comportamentos que deveriam ser implementados na própria classe gerenciada (NYSTROM, 2014, p. 81).

3.1.3 *Factory Method* e *Abstract Factory*

Dalmau sugere o uso dos padrões *Factory Method* e *Abstract Factory* para centralizar a instanciação de objetos em projetos de jogos (DALMAU, 2004). O padrão *Singleton* também pode ser aplicado em conjunto à implementação de fábricas. Gamma et al. sugerem o uso de *Singleton* como uma melhor abordagem no desenvolvimento de *Abstract Factory*, uma vez que não há necessidade de mais de uma instância de uma determinada fábrica concreta (GAMMA et al., 1995, p. 98).

Um exemplo de implementação de *Factory Method* e *Abstract Factory* no Unity pode ser encontrado no repositório de padrões de projeto implementados por Mann (MANN, 2015), em que os padrões são utilizados para instanciações de GameObjects.

Factory Method e *Abstract Factory* também podem ser aplicados no Unity em conjunto com o Object Pool, em que as fábricas podem solicitar à classe de *pools* os objetos a serem utilizados e apenas retorna o objeto.

3.1.4 *Prototype*

Nystrom utiliza um exemplo do padrão *Prototype* na implementação de um sistema de aparição de monstros em um jogo. Nystrom utiliza *Prototype* o padrão como uma alternativa à construção de uma hierarquia de classes que gerenciariam a aparição de monstros, paralela à hierarquia de monstros (NYSTROM, 2014, p. 61). Essa abordagem condiz com o argumento de Gamma et al. para a utilização de *Prototype* quando houver muitas famílias de produtos (no caso, muitas variações de monstros). O padrão *Prototype* pode ser utilizado em conjunto ao padrão *Abstract Factory*, em que uma fábrica concreta possui uma instância-protótipo para cada variação do objeto e gera novos objetos clonando essas instâncias (GAMMA et al., 1995, p. 99).

3.2 Padrões Estruturais no Unity

3.2.1 *Flyweight*

Muitos jogos utilizam uma grande quantidade de objetos para representação de ambientes como florestas, cidades, etc. Esses objetos representam custo de processamento na execução do jogo, o que gera uma preocupação constante para os seus desenvolvedores. Nystrom utiliza o exemplo de um jogo tridimensional que possui uma floresta formada por inúmeras árvores que possuem um modelo tridimensional e texturas idênticas, informações que poderiam ser compartilhadas entre todas as árvores da floresta, pois não possuem variações. Apenas as informações individuais das árvores não seriam compartilhadas, como a posição, cor e escala (NYSTROM, 2014, p. 35).

O padrão *Flyweight* surge como uma abordagem de compartilhamento de informações em comum entre os objetos, para diminuir o impacto causado no desempenho do jogo. No entanto, a vantagem que o padrão proporciona pode não ter a mesma eficiência em uma implementação de projeto em um motor de jogo, quanto teria num cenário de projeto em C++, como é o caso dos exemplos de Nystrom. De fato, o Unity já possui diversas funcionalidades para a redução de consumo de memória no jogo, utilizando técnicas de renderização, por exemplo, que fazem parte da estrutura interna do motor de jogo. Técnicas como *Occlusion Culling*, por exemplo, que consiste em renderizar para a câmera apenas os objetos que estão visíveis ao seu alcance, e não possuem outros objetos bloqueando a sua visualização. Isso reduz significativamente a quantidade de *draw calls* – chamadas na API gráfica para renderização de imagens do jogo (UNITY, 2017c). Um número alto de *draw calls* pode significar queda de desempenho na execução do jogo. Outra técnica do Unity para diminuição de *draw calls* é conhecida como *Draw Call Batching*, que consiste em agrupar o *draw call* de objetos que possuem o mesmo material. Essa técnica pode ser vista em prática na renderização de árvores no Unity (UNITY, 2017a).

As abordagens do Unity reduzem a necessidade de aplicar o *Flyweight* para a melhoria de desempenho com a renderização de objetos. No entanto, o padrão ainda pode ser aplicado para diminuir o uso de memória com o compartilhamento de dados entre objetos (MANN, 2015).

3.2.2 *Adapter*

Dentro do contexto do Unity, *Adapter* pode ser aplicado para que classes de componentes nativos do Unity com interfaces distintas tornem-se compatíveis. Uma aplicação de *Adapter* no Unity é apresentada por Gbadamosi para diminuição a repetição nos scripts Monobehaviour, implementando um adaptador de objetos para facilitar a alteração de cores de objetos que possuem diferentes interfaces (como objetos das classes Image e SpriteRenderer no Unity), porém precisam compartilhar um comportamento em

comum: a manipulação de cores.

Gbadamosi usa o exemplo de um componente que utiliza o adaptador para implementar o comportamento de fade out (desaparecer) gradualmente nos objetos gráficos. O adaptador utilizado aumenta a flexibilidade para a manipulação de cores em diferentes tipos de componentes e para manipulações mais complexas das cores dos componentes (GBADAMOSI, 2015).

3.3 Padrões Comportamentais no Unity

3.3.1 *Game Loop*

O padrão *Game Loop* consiste em utilizar um loop contínuo durante toda a execução de um jogo, responsável por executar os comandos obtidos do jogador, atualizar os estados do jogo e as imagens exibidas para o jogador (NYSTROM, 2014, p. 126). O padrão também é constantemente utilizado para controlar o FPS de um jogo. Isso é necessário porque o desempenho de velocidade do jogo varia de acordo com a capacidade de processamento do dispositivo em que ele é executado. Computadores muito potentes podem executar um jogo com uma quantidade de quadros por segundo maior do que a esperada pelo desenvolvedor, o que pode prejudicar a experiência do jogador, por exemplo. *Game Loop* permite que o desenvolvedor gerencie o tempo de cada execução do loop para se adequar às suas necessidades. Contudo, a implementação do padrão pode ser tornar complicada à medida que o desenvolvedor necessita de maior controle da execução do jogo (NYSTROM, 2014, p. 136).

Normalmente, *frameworks* e *game engines* já possuem uma implementação do padrão *Game Loop* à disposição do desenvolvedor. O Unity disponibiliza uma interface de loop robusta nos scripts Monobehaviour para que o desenvolvedor gerencie a lógica do jogo, o gerenciamento do input do jogador e a renderização de objetos gráficos.

Cada script de Monobehaviour no Unity pode utilizar os métodos de *Game Loop* para atualização da lógica do jogo da seguinte forma (UNITY, 2017b):

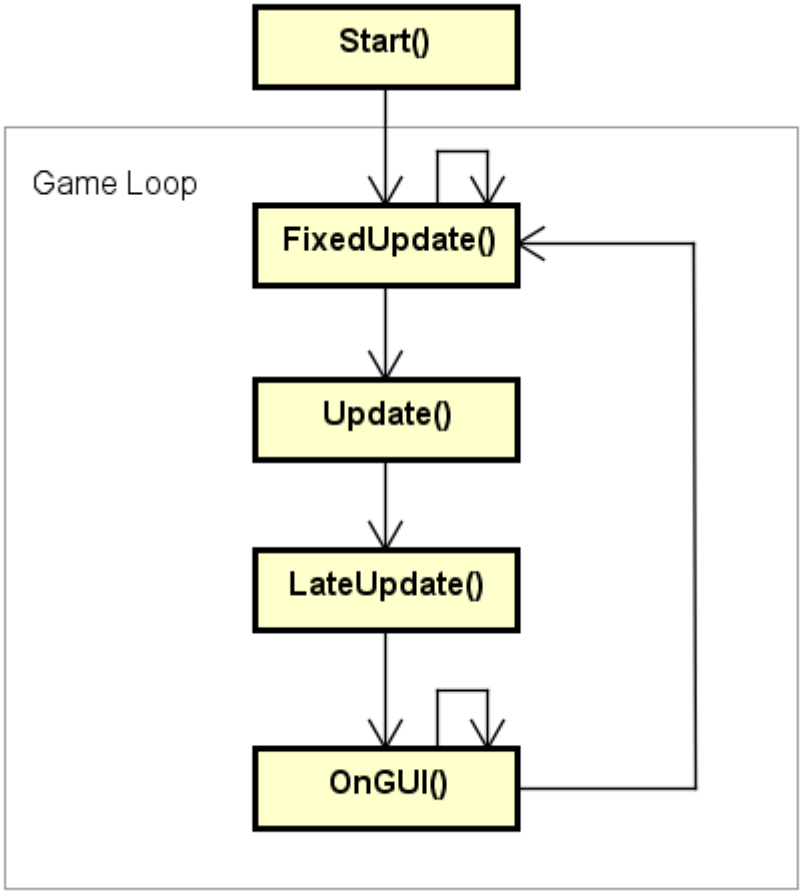
a) FixedUpdate: é executado a uma taxa de frame fixa (podem ser várias execuções por frame) e deve ser utilizado para manipular funções que afetam a física de objetos;

b) Update: é executado a cada frame e é usado para execução da lógica do jogo para os objetos;

c) LateUpdate: também é executado a cada frame e é o último método a ser executado, sendo utilizado principalmente para manipulações de objetos que só devem ser realizadas após a atualização das informações de FixedUpdate e Update.

Existem outros métodos específicos que fazem parte do fluxo de execução do Unity, como o Start, executado apenas uma vez no script e o OnGUI, utilizado para manipulação da interface gráfica do jogo.

Figura 32 – Ordem de execução básica do *Game Loop* no Unity



Fonte: Elaborada pelo autor

O Unity também permite que o desenvolvedor acesse propriedades da classe *Time* para realizar operações de acordo com o tempo decorrido no jogo dentro das funções de Update. É possível realizar a lógica do jogo considerando informações como o tempo necessário para executar o *frame* anterior (*deltaTime*) e alterar a escala de tempo utilizada no jogo (*timeScale*) para simular o efeito de *slow motion*, por exemplo.

A compreensão do fluxo utilizado pelo Unity para a execução do *Game Loop* é importante para obter um bom desempenho na velocidade do jogo e evitar situações inesperadas nas manipulações de objetos que utilizam a física do motor de jogo (UNITY, 2017b).

3.3.2 *Command*

Nystrom demonstra o uso do padrão *Command* em projetos de jogos para gerenciar o controle do jogador. Por exemplo, o jogador utiliza as teclas ou botões do controle do seu dispositivo para realizar ações de um personagem. A entrada de dados do jogador é verificada e, para cada tecla ou botão pressionado pelo jogador, um objeto *command* específico é executado. Portanto, os objetos *command* representam ações a serem executadas dentro do jogo (NYSTROM, 2014, p. 24).

Essa implementação também seria possível apenas verificando a tecla informada pelo jogador e chamando uma função específica para movimentação. No entanto, essa abordagem não permite que o jogador reconfigure as teclas do jogo, por exemplo. O padrão *command* possibilita essa configuração, tornando o gerenciamento do controle mais flexível.

Nordeus adaptou o exemplo de Nystrom em uma implementação do padrão *Command* no Unity (Figura 33).

Figura 33 – Parte da implementação do padrão *Command* no Unity

```
Command buttonU = new FireWeapon();
Command buttonA = new DoNothing();

if (Input.GetKeyDown(KeyCode.U))
{
        buttonU.Execute();
}
if (Input.GetKeyDown(KeyCode.A))
{
        buttonA.Execute();
}
```

Fonte: (NORDEUS, 2016)

Na implementação de Nordeus, a classe abstrata *Command* declara uma interface de Execute(). As classes concretas FireWeapon() e DoNothing() derivam de *Command* e implementam o método Execute(). O método GetKeyDown é disponibilizado pelo Unity para verificação da tecla pressionada, enquanto a execução do comando é delegada aos objetos buttonU e buttonA. Essa implementação adiciona uma camada de abstração em vez de apenas executar uma função específica para cada tecla, permitindo remapear as teclas pressionadas apenas alterando a instanciação de buttonU e buttonA para os objetos *commands* desejados (NORDEUS, 2016).

O padrão *Command* também pode ser aplicado para a implementação de operações

de desfazer e refazer operações em um jogo de estratégia, por exemplo. Cada operação realizada é adicionada a uma lista de *commands*, em que é possível navegar para identificar o comando anterior ou o próximo (caso algum comando já tenha sido desfeito) utilizando uma referência ao *command* atual na lista. O conceito de listas de *commands* também pode ser aplicado para implementar o *replay* em um jogo, reproduzindo novamente todos os comandos já realizados pelo jogador, por exemplo (NYSTROM, 2014, p. 26).

3.3.3 *Observer*

O padrão *Observer* possui diversas aplicações em projetos de jogos. Nystrom demonstra o uso do padrão com o conceito de *achievements* – conquistas que podem ser desbloqueadas em diversas situações durante o jogo. Exemplos de formas de obter *achievements* podem ser derrotar um inimigo, customizar um equipamento ou simplesmente acessando um menu específico do jogo. As distintas situações que podem liberar *achievements* exigem do desenvolvedor uma implementação de um sistema de *achievements* que não torne as classes fortemente acopladas e não misture as responsabilidades de uma classe com a implementação dos *achievements* (NYSTROM, 2014, p. 44).

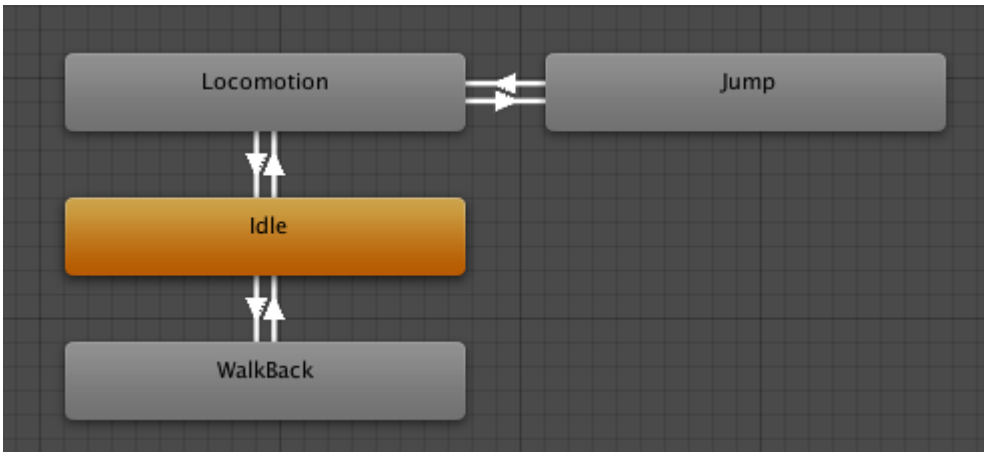
O padrão *Observer* também pode ser aplicado no Unity como um sistema para gerenciamento de eventos entre os objetos, como alternativa ao uso do método SendMessage, que é implementado nativamente pelo motor de jogo (LE, 2016). Essa abordagem também é sugerida por Barcelo para comunicar múltiplos GameObjects e evitar referências concretas a esses objetos (BARCELO, 2013).

3.3.4 *State*

Uma das situações mais comuns na implementação de jogos é a necessidade de gerenciar estados de objetos. A animação de personagens possui estados (parado, andando, correndo), a dinâmica do jogo pode possuir estados (turno do jogador, turno da IA; vitória, derrota), entre outras situações. Implementações simples podem ser feitas utilizando algumas estruturas condicionais (*switches* e *ifs*) e variáveis booleanas. Contudo, não são raras as situações em que o gerenciamento de estados torna-se caótico e o código do projeto acaba repleto de condições para verificação dos estados, sobrecarregando as classes com atributos de estado e gerando código fortemente acoplado e propício a erros.

Nystrom discute a abordagem de *State* em jogos utilizando o conceito de *Finite State Machines*, ou Máquinas de Estado Finito, que representam o conjunto de estados, transições e os eventos que geram as transições de estados na máquina de estados (NYSTROM, 2014, p. 90). Desenvolvedores que utilizam o Unity estão parcialmente familiarizados com o conceito, pois o motor de jogo utiliza máquinas de estados para o controle de animações de GameObjects.

Figura 34 – Máquina de Estados para controle de animações no Unity



Fonte: (UNITY, 2017g)

A implementação de *State* propõe o uso de objetos para a representação dos estados, substituindo as estruturas condicionais espalhadas pelo código e agrupando os atributos relevantes a um determinado estado à sua classe correspondente, gerando estados encapsulados de um objeto (NYSTROM, 2014, p. 99). Essa abordagem simplifica os métodos de *Game Loop* dos scripts no Unity, tornando o código mais limpo e mais fácil de adicionar novos estados (apenas adicionando novas classes de estado).

3.3.5 *Strategy*

O conceito do encapsulamento e separação de algoritmos proposto pelo padrão *Strategy* pode ser um utilizado em uma gama de situações no contexto de projetos de jogos. Figueiredo propõe o uso de *Strategy* para separar as habilidades de inimigos em um jogo em diferentes estratégias. Essa abordagem desacopla os diferentes tipos de habilidades do script dos inimigos (FIGUEIREDO, 2014). Outro uso, sugerido por Barcelo, utiliza interfaces no Unity para representar armas, adicionar e separar o comportamento balístico das armas em diferentes scripts com classes concretas, possibilitando que qualquer objeto possa adicioná-las e obter o comportamento implementado (BARCELO, 2013). Ambas as abordagens respeitam a estrutura de componentes no Unity e utilizam o encapsulamento das estratégias para obter diferentes comportamentos dos objetos.

CONSIDERAÇÕES FINAIS

A maioria dos padrões de projeto utilizados em jogos e que são citados neste trabalho foram apresentados por Gamma et al. Alguns padrões são aplicáveis de forma independente do Unity, com características positivas e implicações parecidas com as suas implementações em outros projetos de software. As aplicações citadas neste trabalho e recorrentes em jogos dos padrões *Strategy*, *Prototype* e *Command*, por exemplo, assemelham-se ao modelo clássico do padrão definidos pelo GoF. Nesse contexto, é imprescindível conhecer a literatura apresentada por Gamma et al. para se familiarizar com as características dos padrões de projeto, compreender em quais situações eles podem ser mais bem aplicados e, em contrapartida, quais são as consequências e implicações do uso de cada um deles no projeto.

Alguns padrões do GoF não foram abordados em aplicações voltadas a jogos, pois possuem poucas menções ou não apresentam detalhes específicos em projetos de jogos citados nos livros e artigos estudados. Aplicações de padrões como Memento, *Composite* e *Bridge* sequer possuem menções nas apresentações dos padrões de projeto de jogos de Nystrom, por exemplo. No entanto, esses padrões não perdem de forma alguma a sua relevância e devem ser considerados para um uso futuro, pois suas aplicações podem se tornar necessárias em abordagens específicas dentro do motor de jogo.

Entre os padrões do GoF citados neste trabalho, a utilização do *Singleton* mostrou-se a mais controversa na literatura pesquisada. O uso do padrão é criticado tanto no trabalho de Nystrom, focado em projetos de jogos, quanto por Yener e Theedom, com foco em softwares tradicionais. Parte das críticas é devido ao uso irrestrito e inconsequente uso do padrão, o que pode ser visto como um resultado da sua popularidade entre os desenvolvedores, uma vez que a sua implementação é mais simples, se comparada aos demais padrões. Apesar das críticas duras direcionadas ao *Singleton*, seu uso pode ser considerado em projetos de jogos, desde que se observem os pontos de crítica levantados pelos autores e considerando o contexto de cada projeto.

Além dos padrões de projeto clássicos apresentados pelo GoF, também foram pesquisados os trabalhos mais recentes sobre padrões de projeto e como esses se aplicam em projetos de jogos por meio de artigos, publicações em sites e livros sobre desenvolvimento de jogos em Unity. Os padrões fora da lista do GoF e com mais menções de uso em jogos são mencionados no trabalho de Nystrom, em artigos acadêmicos e sites com foco em desenvolvimento de jogos, além dos próprios manuais e tutoriais disponibilizados pelo Unity.

As aplicações dos padrões de projeto clássicos e padrões utilizados em jogos apresentados são frequentemente combinadas para a solução de um determinado problema.

Nesse cenário podemos utilizar como exemplos a junção dos padrões *Object Pooling* e o *Factory Method* para solucionar problemas de performance na criação de objetos no jogo; *State* e *Game Loop* podem ser aplicados em conjunto para definir os diferentes estados possíveis nos objetos em uma iteração no jogo; *Component* e *Builder* juntos permitem a criação de objetos com diferentes variações de componentes.

Compreender a relação entre os padrões de projeto e as características do Unity é essencial para o desenvolvedor que deseja aplicar os padrões clássicos ou recorrentes em jogos para resolução de problemas no desenvolvimento de projetos no motor de jogo. Cada padrão possui características que podem ou não se adequar à estrutura que o Unity disponibiliza.

Assim como *frameworks* de desenvolvimento de software tradicionais, o Unity também possui implementações de padrões dentro do próprio motor de jogo. A estrutura do padrão *Game Loop* é disponibilizada e gerenciada pelo Unity, permitindo que o desenvolvedor mantenha o seu foco na implementação do seu jogo. Outras características do motor de jogo podem diminuir a necessidade da implementação de alguns padrões em situações específicas. A otimização e gerenciamento de objetos gráficos do Unity tornam o uso do padrão *Flyweight* menos recorrente, por exemplo. No entanto, é importante que o desenvolvedor conheça a teoria desses padrões para entender melhor o seu funcionamento no contexto do Unity.

Mesmo que a aplicação de um padrão de projeto represente a solução para um problema encontrado no projeto, a sua implementação no Unity pode não ser trivial para um programador inexperiente, caso não possua a base teórica dos padrões de projeto ou não tenha intimidade com o fluxo de trabalho do Unity. Da mesma forma, fazer alterações em um projeto que já utiliza diversos padrões implementados pode se tornar uma tarefa complexa para desenvolvedores iniciantes. Portanto, conhecer a estrutura do motor de jogo, compreender como GameObjects e componentes se relacionam e de que forma utilizá-los com os scripts são os passos iniciais a serem realizados pelo desenvolvedor que pretende aplicar padrões de projeto no Unity.

Referências

ANDRUSHKO, O. *Methods of organizing the interaction between scripts in Unity*. 2015. Disponível em: <http://www.gamasutra.com/blogs/OlegAndrushko/20150319/239199/Methods_of_organizing_the_interaction_between_scripts_in_Unity.php>. Acesso em: 01/11/2016.

BARCELO, V. *Using abstractions and interfaces with Unity3D*. 2013. Disponível em: <http://www.gamasutra.com/blogs/VictorBarcelo/20131217/207204/Using_abstractions_and_interfaces_with_Unity3D.php>. Acesso em: 01/12/2016.

BILAS, S. *A Data-Driven Game Object System*. 2002. Disponível em: <http://scottbilas.com/files/2002/gdc_san_jose/game_objects_slides.pdf>. Acesso em: 01/11/2016.

BLOCH, J. *Effective java*. [S.l.]: Pearson Education India, 2008.

DALMAU, D. S.-C. *Core techniques and algorithms in game programming*. [S.l.]: New Riders, 2004.

DUNSTAN, J. *Unity Function Performance Followup*. 2016. Disponível em: <http://jacksondunstan.com/articles/3605>. Acesso em: 01/11/2016.

FIGUEIREDO, R. T. Padrões de projeto gof aplicados ao desenvolvimento de jogos eletrônicos. Universidade Federal de Pernambuco, 2014.

FOWLER, M. *Patterns of enterprise application architecture*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2002.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley, 1995.

GBADAMOSI, C. *Unity: How Adapters can help you write fewer Mono-Behaviours*. 2015. Disponível em: <http://www.ticktakashi.com/2015/12/unity-how-adapters-can-help-you-write.html>. Acesso em: 05/07/2017.

HALL, G. M. *Adaptive Code via C#: Agile coding with design patterns and SOLID principles*. [S.l.]: Pearson Education, 2014.

LE, C. Design patterns: Implementation in video game programming. Metropolia Ammattikorkeakoulu, 2016.

MANN, S. *Unity Design Patterns*. 2015. Disponível em: <https://github.com/Naphier/unity-design-patterns>. Acesso em: 05/07/2017.

MONO. *About Mono*. 2017. Disponível em: <http://www.mono-project.com/docs/about-mono/>. Acesso em: 01/07/2017.

MURRAY, J. W. *C# game programming cookbook for Unity 3D*. [S.l.]: CRC Press, 2014.

NORDEUS, E. *Game programming patterns in Unity with C#*. 2016. Disponível em: <http://www.habrador.com/tutorials/programming-patterns>. Acesso em: 05/07/2017.

NYSTROM, R. *Game programming patterns*. [S.l.]: Genever Benning, 2014.

TULLEKEN, H. *50 Tips and Best Practices for Unity (2016 Edition)*. 2016. Disponível em: <http://www.gamasutra.com/blogs/HermanTulleken/20160812/279100/50_Tips_and_Best_Practices_for_Unity_2016_Edition.php>. Acesso em: 01/12/2016.

UNITY. *Draw call batching*. 2017. Disponível em: <https://docs.unity3d.com/Manual/DrawCallBatching.html>. Acesso em: 05/07/2017.

UNITY. *Execution Order of Event Functions*. 2017. Disponível em: <https://docs.unity3d.com/Manual/ExecutionOrder.html>. Acesso em: 05/07/2017.

UNITY. *Occlusion Culling*. 2017. Disponível em: <https://docs.unity3d.com/Manual/OcclusionCulling.html>. Acesso em: 05/07/2017.

UNITY. *The Scene View*. 2017. Disponível em: <https://docs.unity3d.com/2017.1/Documentation/Manual/UsingTheSceneView.html>. Acesso em: 01/07/2017.

UNITY. *Understanding Automatic Memory Management*. 2017. Disponível em: <https://docs.unity3d.com/Manual/UnderstandingAutomaticMemoryManagement.html>. Acesso em: 01/07/2017.

UNITY. *Unity Scripting Reference*. 2017. Disponível em: <https://docs.unity3d.com/ScriptReference/>. Acesso em: 01/07/2017.

UNITY. *Unity User Manual (5.6)*. 2017. Disponível em: <https://docs.unity3d.com/Manual/>. Acesso em: 01/07/2017.

YENER, M.; THEEDOM, A. *Professional Java EE Design Patterns*. [S.l.]: John Wiley & Sons, 2014.

ZUCCONI, A. *5 common mistakes made in Unity*. 2015. Disponível em: <https://www.microsoft.com/en-gb/developers/articles/week04jul15/5-common-mistakes-made-in-unity>. Acesso em: 01/11/2016.