Análise do impacto do uso de padrões de projeto em jogos

Vitor Braga Estevam[1], **Alysson Diniz dos Santos** [1]

[1]Instituto Universidade Virtual – Universidade Federal do Ceará (UFC)

Fortaleza – CE – Brasil

vitorestevam02@gmail.com,alysson@virtual.ufc.br

Abstract. *The proper use of design patterns may promote software quality from early stages of a project. Game development often deals with short deadlines for implementing complex features that can involve multiple areas such as interactivity, animations, and sound design. The objective of this work is to investigate the use of five design patterns (Command, Flyweight, Observer, Prototype, and State) in the game development context. Starting from an analysis of papers published at the Simpósio Brasileiro de Jogos (SBGames), this work presents a mapping between design patterns and game mechanics. As a proof of concept, it presents the refactoring of two small games, taking the mapping as a reference and running static code analysis on the projects, wich enables a discussion about the impact of design patterns in the maintainability of these projects.*

Resumo. *A adequada utilização de padrões de projeto é um dos fatores capazes de promover a qualidade de software desde as etapas iniciais do desenvolvimento de um sistema. O desenvolvimento de jogos digitais, em específico, lida frequentemente com prazos curtos para implementação de requisitos complexos que envolvem múltiplos domínios, como interatividade, animações, áudio, dentre outros. O objetivo deste trabalho é investigar o uso de cinco padrões de projeto (Command, Flyweight, Observer, Prototype e State) no contexto do desenvolvimento de jogos. A partir de uma análise de artigos publicados no Simpósio Brasileiro de Jogos (SBGames), este trabalho apresenta um mapeamento das mecânicas de jogos adequadas aos padrões escolhidos. Outra contribuição é a refatoração de dois jogos de pequeno porte, considerando os padrões de projeto escolhidos. A análise estática das duas versões de cada jogo permitem a discussão acerca do impacto dos padrões na manutenibilidade dos projetos.*

1. Introdução

Um padrão de projeto é uma solução genérica que pode ser usada para resolver problemas comuns do desenvolvimento de um software [Alghamdi 2014]. O livro "Design patterns: elements of reusable object-oriented software", escrito pela Gangue dos Quatro (Gang of Four - GoF) lista 23 padrões e é considerado a referência primária sobre o assunto [Gamma et al. 1995]. O uso de padrões de projeto é consolidado no desenvolvimento de sistemas e a sua utilização pode gerar impactos positivos na qualidade do software [Alghamdi 2014].

Manutenibilidade se refere ao quão favorável a modificações, ajustes e melhorias é um projeto de software [IEEE 1993]. O uso de padrões de projeto é uma das maneiras de

tentar promover a manutenibilidade de um software desde as fases iniciais do seu desenvolvimento. No entanto, a eficiência desta ação pode ser influenciada pela complexidade do padrão e pela sua adequada utilização. [Alghamdi 2014].

Jogos, assim como qualquer tipo de software, têm um desenvolvimento sujeito a problemas que frequentemente estão relacionados à manutenibilidade, como a dificuldade de entender abstrações entre classes ou funções com muitas responsabilidades [Borrelli et al. 2020]. Estes problemas são agravados pelos prazos curtos para desenvolvimento de atividades complexas, o que pode levar a más práticas de programação [Kounoukla et al. 2016].

Neste contexto, alguns pesquisadores [Figueiredo 2015, Kounoukla et al. 2016, Barakat 2019, Paschali et al. 2021] têm investigado os benefícios técnicos da utilização de padrões de projeto no desenvolvimento de jogos digitais. O livro Game Programming Patterns [Nystrom 2014] aprofunda-se na aplicação destes padrões no contexto do desenvolvimento de jogos. O autor propõe treze padrões específicos que consideram aspectos de sequenciamento de ações, desacoplamento, comportamento e otimização. Além da proposição, o autor revisita alguns padrões GoF e argumenta que cinco deles trazem vantagens ao desenvolvimento de jogos: Command, Flyweight, Observer, Prototype e State.

Os padrões GoF já foram amplamente estudados no domínio de desenvolvimento de outros tipos de software, mas nos jogos o assunto não foi tão aprofundado. Além disso, enquanto os padrões GoF propõem soluções genéricas de desenvolvimento que podem ser utilizados em diferentes contextos, os padrões específicos de jogos propostos por [Nystrom 2014] se referem a pontos específicos do desenvolvimento de jogos (otimização e sequenciamento, por exemplo).

Desta forma, o objetivo deste trabalho é investigar o uso de cinco padrões GoF (Command, Flyweight, Observer, Prototype e State) no contexto do desenvolvimento de jogos. Este objetivo geral é abordado a partir dos seguintes objetivos específicos: (i) mapear possíveis aplicações dos padrões GoF escolhidos em mecânicas de jogos; (ii) propor a refatoração de dois jogos considerando a aplicação dos padrões escolhidos; e (iii) mensurar o impacto na manutenibilidade através de uma análise automatizada de código nos jogos antes e depois da refatoração.

As principais contribuições esperadas para este trabalho são duas: (i) um mapeamento dos padrões GoF escolhidos e de mecânicas de jogos adequadas para a sua aplicação; e (ii) uma discussão acerca do impacto mensurável da aplicação destes padrões na manutenibilidade de projetos de jogos. Desta forma, espera-se que este trabalho possa agregar ao corpo de trabalhos sobre o tema, visando analisar se o uso de padrões de projeto pode ser uma solução eficiente para problemas de manutenibilidade no desenvolvimento de jogos.

Este trabalho apresenta: (i) a fundamentação teórica que introduz os principais conceitos que embasam esta pesquisa; (ii) o levantamento bibliográfico que mapeia o uso de padrões de projeto em jogos com base em outros trabalhos na área, resultando em uma relação entre mecânicas e padrões que serve de referência para as próximas etapas; (iii) o desenvolvimento que apresenta a refatoração de dois jogos aplicando os padrões nas mecânicas identificadas e as análises estáticas de código dos jogos refatorados e (iv) as conclusões do trabalho.

2. Fundamentação Teórica

O livro Game Programming Patterns [Nystrom 2014] aprofunda-se na aplicação de padrões de projeto no contexto do desenvolvimento de jogos. O autor revisita alguns padrões GoF e argumenta que cinco deles trazem vantagens ao desenvolvimento de jogos: Command, Flyweight, Observer, Prototype e State. Convém relembrá-los brevemente:

- Command encapsula requisições em objetos, transformando ações em um conjunto de informações [Gamma et al. 1995].
- Flyweight propõe compartilhar recursos entre instâncias para suportar eficientemente grandes quantidades de objetos semelhantes ou iguais [Gamma et al. 1995].
- Observer é uma forma de definir uma dependência de um para muitos entre objetos, de forma que quando o objeto A muda de estado, todos os objetos B são notificados [Gamma et al. 1995].
- Prototype propõe que objetos tenham a capacidade de criar cópias de si mesmo e de seu estado atual [Nystrom 2014].
- State permite que um objeto mude de comportamento quando seu estado muda [Gamma et al. 1995].

A cuidadosa aplicação dos padrões de projeto nas mecânicas de um jogo pode contribuir significativamente com a arquitetura do software que compõe o projeto [Nystrom 2014]. Mecânicas de jogos são as várias ações, comportamentos e mecanismos de controle possibilitados ao jogador no contexto de um jogo [Hunicke et al. 2004]. Comumente, estas mecânicas são descritas em em alto nível, sem a definição de uma maneira específica para implementá-las [Kounoukla et al. 2016].

A arquitetura do projeto de um jogo é um aspecto especialmente sensível, uma vez que seu desenvolvimento pode ser considerado uma tarefa complexa que envolve múltiplas áreas, como: interatividade, animações, áudio, dentre outros. Além disso, jogos normalmente lidam com longos ciclos de evolução e mudança, algumas vezes até anos após o lançamento do produto, o que destaca a importância do nível de manutenibilidade do projeto.

2.1. Análise Estática

Uma forma de analisar a manutenibilidade de um software é via análise estática, que consiste em executar uma busca por problemas no código fonte de um projeto sem a necessidade de executá-lo [Borrelli et al. 2020]. Este processo provê um entendimento da estrutura do código, e pode ser utilizado em fases iniciais do desenvolvimento como um passo para a promoção da qualidade do software.

Os problemas identificados na análise estática podem ser de vários domínios como *bugs*, performance ou, no caso desse trabalho, o nível de manutenibilidade. A resolução destes problemas em estágios iniciais do desenvolvimento pode ajudar os desenvolvedores na etapa de análise dinâmica do código, ou seja, com o software em execução [Motogna et al. 2022].

Ainda que problemas possam ser identificados através de revisões manuais do código, a análise estática normalmente é realizada de forma automatizada. Ferramentas

como Ndepend[1], SonarQube[2] e Code Analysis[3] são capazes de realizar esse tipo de análise em projetos de jogos. Neste trabalho, Code Analysis foi a ferramenta escolhida por ser integrada à IDE Visual Studio e mantida pela Microsoft, mesma empresa à frente da linguagem de programação C#. Esta proximidade ao C# faz com que a Code Analysis se integre com simplicidade à engine de jogos Unity que, por sua vez, é uma das tecnologias padrão do mercado, com comunidade ativa de usuários e com disponibilidade de projetos de código aberto que pudessem ser utilizados nesta pesquisa.

As métricas geradas pelo Code Analysis são: acoplamento, profundidade de herança, complexidade ciclomática, quantidade de linhas de código e índice de manutenibilidade.

O acoplamento é relacionado à interdependência entre partes distintas do código. Em classes, isso se manifesta através de referencias armazenadas em variáveis ou parâmetros, mas não diz respeito à herança [Liukku 2020]. O acoplamento pode ser um indicador de falhas e é desejável que seu valor seja baixo, pois quanto mais alto, mais difícil de manter é o código [Jones 2022]. Além disso, baixo acoplamento pode representar um aumento na reutilização de código [Liukku 2020].

A profundidade de herança representa a quantidade de classes que herdam atributos de outras classes. Representa um problema semelhante ao acoplamento entre classes, já que mudanças na classe pai podem causar quebras e comportamentos inesperados nas classes filhas. No entanto, além de representar complexidade no código, a profundidade de herança também pode representar o reuso de funções passadas da classe pai para a classe filha, o que pode evitar duplicação de código [Jones 2022].

A complexidade ciclomática refere-se ao total de possíveis caminhos que a execução de um código pode seguir e se dá em função da quantidade de blocos de códigos e os possíveis fluxos entre eles. Quanto maior o valor, mais complexo é o código e mais testes serão necessários para cobri-lo. Complexidade ciclomática pode ser um indicador de possíveis falhas no código, no entanto, se analisada juntamente com o número de linhas de código, torna-se um indicador mais forte de potenciais problemas [Jones 2022].

O índice de manutenibilidade é representado por um valor de 0 a 100 que se refere à facilidade de fazer ajustes e melhorias naquele código, sendo 100 um código fácil de manter e 0 um difícil [Jones 2022]. Esse valor se dá em função da complexidade ciclomática, do número de linhas do código e do volume de Halstead. As métricas de Halstead propõem medir a complexidade de um código-fonte através de seus operadores e valores associados por caracterizarem os processos que o programa executa [Thirumalai et al. 2017]. O volume de Halstead refere-se à quanta informação o leitor precisa ter para entender o significado do código e se dá em função da quantidade de operadores totais e únicos presentes nele.

Tendo em vista que o índice de manutenibilidade se dá em função da complexidade ciclomática e das linhas de código, as métricas que foram utilizadas nesse trabalho foram o índice de manutenibilidade, a profundidade de herança e o acoplamento entre classes.

[1]https://www.ndepend.com/docs/analyzing-csharp-unity-app

[2]https://docs.sonarqube.org/latest/analysis/languages/csharp/

[3]https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022

3. Padrões de Projeto e Mecânicas de Jogos

No contexto deste trabalho, os pesquisadores decidiram aprofundar-se nos 5 padrões GoF discutidos por [Nystrom 2014]: Command, Flyweight, Observer, Prototype e State (ver Seção 2). Desta forma, definiu-se a seguinte pergunta de pesquisa: a quais mecânicas de jogos se aplicam os padrões Command, Flyweight, Observer, Prototype e State?

Para investigar a pergunta de pesquisa no contexto do maior evento científico de jogos nacional, foi feita uma pesquisa exploratória nos trabalhos publicados no SBgames[4] desde 2014 até 2021. Os pesquisadores leram o título, o resumo e as palavras chaves dos artigos, buscando por trabalhos que se relacionassem com aplicações práticas dos padrões GoF em jogos.

O único trabalho encontrado no contexto do SBGames foi o artigo intitulado "GoF design patterns applied to the Development of Digital Games"[Figueiredo 2015]. O trabalho relaciona 6 padrões GoF (Builder, Singleton, Flyweight, Observer, Prototype e State) com possíveis mecânicas que podem ser implementadas com eles, apresentando exemplos. Os autores ainda detalham um experimento com um teste AB entre 6 equipes desenvolvendo um mesmo jogo onde 3 equipes foram previamente instruídas acerca do uso de padrões de projeto e 3 equipes não. Apesar do estudo limitado, os resultados indicam que as equipes que utilizaram padrões de projeto produziram software de melhor qualidade, em menos tempo.

Buscando nas referências de [Figueiredo 2015] e pesquisando os trabalhos que citam estas referências, foi possível encontrar dois trabalhos: "A Framework for integrating software design patterns with game design framework" [Barakat 2019] e "Implementing Game Requirements using Design Patterns" [Paschali et al. 2021].

O trabalho de [Barakat 2019] propõe um *framework* que relaciona os padrões de projeto GoF com elementos de game design. O autor propõe que os padrões State, Strategy e Observer sejam utilizados na implementação do *gameplay* e dos seus aspectos formais (regras, padrões de interação, procedimentos e recursos) enquanto que os padrões Prototype e State sejam aplicados na implementação dos elementos dramáticos (personagens controláveis e não controláveis). Apesar da interessante relação dos padrões com aspectos de design de jogos, este é um artigo curto, cuja limitação reconhecida pelos autores requer a validação do *framework* proposto.

O trabalho de [Paschali et al. 2021] estende [Kounoukla et al. 2016] e propõe um repositório de padrões de projeto para jogos, o "GAME-DP Repository". O objetivo do repositório é intrinsecamente relacionado ao do presente trabalho: mapear mecânicas de jogos aos padrões GoF, de forma a prover implementações padrão e avaliar os impactos em termos de extensibilidade e reusabilidade do código. Em experimento realizado com 12 desenvolvedores, os resultados indicaram que a utilização do repositório aumentou a taxa de sucesso, apesar do maior tempo necessário para compreender a documentação. Apesar deste ser um trabalho relevante, é importante ressaltar que o repositório GAME-DP não está disponível online (o link fornecido no trabalho não estava acessível durante a realização deste trabalho) o que restringe o acesso às informações desejadas.

Com a análise de [Figueiredo 2015, Barakat 2019, Paschali et al. 2021] é possível

[4]Simpósio Brasileiro de Games e Entretenimento Digital https://www.sbgames.org

estabelecer uma relação entre os padrões GoF escolhidos, com as mecânicas de jogos a que se aplicam. Este resultado está sumarizado na Tabela 1.

Tabela 1: Mapeamento de padrões e mecânicas segundo [Nystrom 2014], [Barakat 2019], [Figueiredo 2015] e [Paschali et al. 2021]

Mecânica	Padrão	Definição	Exemplo	Autores
Conquistas e Missões	Observer	Definição de objetivos ou conquistas do jogador	Atribuição de uma nova missão principal ou secundária	Nystrom
Elementos de UI	Observer	Comunicação entre as instâncias do jogo com os componentes de interface	Decrescimento de um elemento de interface, como uma barra de vida, quando o jogador sofre danos	Barakat e Paschali
Variação de Comportamento	State	Variação de comportamento de entidades de acordo com seu estado	Um personagem só pode executar determinadas ações se estiver de pé e outras se estiver pulando	Barakat, Nystrom e Figueiredo
Controle de Entidades	Command	Encapsulamento de comandos que uma entidade pode receber	Determinado comando do personagem pode ser mapeado para teclado e para o controle de forma a usar o que estiver disponível	Nystrom
Desfazer Ações	Command	As ações executadas podem ser armazenadas e manipuladas	Um jogo de desenho pode armazenar uma referencia de cada traço feito e pode remover o último de forma independente do restante	Nystrom
Animações de Personagens	State	Gerência e execução das animações dos personagens	Organização de todas as possíveis animações de movimento de uma entidade	Barakat

Replicação de Entidades	Prototype	Capacidade de gerar novas instâncias de um objeto do jogo	Aparecimento de novos inimigos em um novo turno do jogo	Barakat, Nystrom, Figueiredo e Paschali
Repetição de Recursos	Flyweight	Compartilhamento de recursos entre entidades replicadas	Compartilhamento do modelo 3D de um inimigo que aparece em grupo.	Nystrom, Figueiredo e Paschali

O padrão Observer proporciona comunicação sem a necessidade de acoplamento entre módulos [Nystrom 2014]. Desta forma, ele é particularmente interessante para mecânicas que se relacionam com múltiplas dimensões do jogo, como Conquistas e Missões e Elementos de UI. Por exemplo, as conquistas (mecânica Conquistas e Missões) podem se relacionar com atingir o final do jogo, tomar determinadas decisões, acessar diferentes partes da interface, dentre outros. A implementação desta mecânica sem o padrão Observer tende a aumentar significativamente o acoplamento do código e pode, portanto, diminuir a sua manutenibilidade.

O padrão State é adequado para mecânicas cujo funcionamento possa ser coordenado por uma máquina de estado finita. Este é o caso tanto da Variação de Comportamento, quanto da Animação de Personagens. Por exemplo, quando o jogador aperta um determinado botão, a entidade controlada por ele no jogo entra no estado de ataque, agindo conforme aquela situação (mecânica Variação de Comportamento) e exibindo a animação adequada (mecânica Animação de Personagens).

O padrão Command encapsula uma requisição em um objeto, o que relaciona-se com as mecânicas de Controle de Entidades e Desfazer Ações, visto que o padrão permite saber a ordem que as ações ocorreram, reproduzir uma sequência de ações, construir históricos, desfazer movimentos, etc [Nystrom 2014].

O padrão Prototype é citado por todos os autores, uma vez que a sua finalidade é diretamente relacionada à mecânica de Replicação de Entidades. Neste caso, o uso de Prototype evita a duplicação de código e facilita a criação de objetos [Figueiredo 2015] [Nystrom 2014]. Este padrão é tão comum que há suporte nativo para ele em algumas *engines* de jogos, como a Unity que provê nativamente o método Instantiate que clona objetos protótipos chamados Prefabs [Unity 2022].

O padrão Flyweight se relacionado à mecânica de Repetição de Recursos, uma vez que o seu objetivo envolve o compartilhamento de recursos para suportar eficientemente grandes quantidades de objetos semelhantes ou iguais [Gamma et al. 1995]. Por exemplo, em um jogo, um inimigo pode ser uma instância de uma classe que, portanto, consome recursos da máquina, como memória e processamento gráfico. Considerando que fossem necessárias múltiplas cópias deste inimigo, o Flyweight possibilita que essa instância seja separada em dois objetos: um com as informações específicas (nível, vida atual, etc) e outro com as informações gerais que serão compartilhadas (textura, modelo 3D, vida máxima, etc). Esse objeto genérico não possui relação com nenhuma instância especifica e é passado para quantas forem necessárias, economizando, portanto, os

recursos da máquina.

De forma geral, a Tabela 1 responde a pergunta de pesquisa levantada (a quais mecânicas de jogos se aplicam os padrões Command, Flyweight, Observer, Prototype e State?). Através da análise da literatura é possível reforçar o entendimento de [Nystrom 2014] sobre algumas aplicações para os padrões escolhidos através da confirmação de outros autores, como a utilização de State para Variação de Comportamento, Prototype para Replicação de Entidades e Flyweight para Repetição de Recursos. Em outros casos, é possível catalogar outras aplicações para os padrões, além das que são listadas em [Nystrom 2014], por exemplo, Observer para Elementos de UI e State para Animação de Personagens.

Tabulações mais completas que esta, que consideram mais padrões GoF, ou que apresentam exemplos de código foram propostas por [Figueiredo 2015] e [Paschali et al. 2021]. No entanto, na realização desta pesquisa, não foi possível acessar os repositórios relatados por estes pesquisadores. Além disso, é importante ressaltar que, como a Tabela 1 foi criada a partir da análise de trabalhos recentes encontrados na literatura, espera-se que ela possa ser interessante para pesquisadores e desenvolvedores interessados em um documento inicial que possa ajudá-lo na utilização de padrões GoF em projetos de jogos.

4. Desenvolvimento

Uma vez identificadas as mecânicas de jogos associadas aos padrões GoF escolhidos (ver Tabela 1) foi proposta a refatoração de dois jogos com a aplicação desses padrões. Após isso foram realizadas análises estáticas de código para quantificar o efeito dos padrões em termos do nível de manutenibilidade dos projetos.

4.1. Refatoração de Projetos

Para essa etapa, foram identificados dois jogos disponibilizados pela Unity na plataforma *Unity Asset Store* como projetos tutoriais e introdutórios. Estes jogos foram escolhidos por serem de código aberto, o que é requerido para que a refatoração pudesse ser realizada. Além disso, esta escolha se deu pelo escopo reduzido dos jogos, em consideração ao tempo disponível para a realização desta pesquisa.

O jogo 1, *2D Roguelike*[5] (Figura 1A), se trata de um jogo de sobrevivência em turnos baseado em *grids* para um jogador. A cada turno o jogador pode escolher uma direção para se mover. Seus objetivos são evitar o contato com os inimigos (que também se movem em turnos) e coletar comida, que é o recurso utilizado pelo jogador para se mover. Já o jogo 2, *Tanks!*[6] (Figura 1B), consiste em um jogo para dois jogadores que controlam tanques de guerra que podem se mover e disparar projéteis. O objetivo dos jogadores é acertar o oponente até acabar com os pontos de vida dele.

Os jogos escolhidos foram submetidos a um processo de análise e foram jogados múltiplas vezes com a finalidade de identificar a presença das mecânicas definidas na Tabela 1. O resultado desta análise está sumarizado na Tabela 2.

[5]https://assetstore.unity.com/packages/templates/tutorials/2d-roguelike-29825

[6]https://assetstore.unity.com/packages/essentials/tutorial-projects/tanks-tutorial-46209

A)

B)

Figura 1. Jogos escolhidos para refatoração. A) 2d roguelike (Jogo 1), B) Tanks! (Jogo 2)

Tabela 2: Mecânicas dos Jogos 1 e 2

Mecânica	Jogo 1	Jogo 2
Conquistas e Missões	×	×
Elementos de UI	✓	✓
Variação de Comportamento	×	×
Controle de Entidades	✓	✓
Desfazer Ações	×	×
Animações de Personagens	×	×
Replicação de Entidades	✓	✓
Repetição de Recursos	✓	×

No jogo 1, foram identificadas as mecânicas “Elementos de UI” utilizada nos mostradores de comida do personagem principal e da fase atual; “Controle de Entidades” no controle do personagem principal e dos inimigos; “Replicação de Entidades” na geração das fases; e “Repetição de Recursos” entre as múltiplas instâncias de inimigos.

No jogo 2, foram identificadas as mecânicas “Elementos de UI” para mostrar a vida dos personagens; “Controle de Entidades” para controlar os dois personagens jogáveis; e “Replicação de Entidades” ao atirar projéteis.

Em seguida, os pesquisadores analisaram em profundidade o código de cada um dos jogos. Inicialmente, foi necessário a compreensão da sua estrutura e, posteriormente,

a identificação da implementação das mecânicas para a verificação da utilização ou não dos padrões.

Em ambos os jogos, tanto a mecânica de Elementos de UI quanto a de Controle de Entidades estavam funcionando de forma direta, com acoplamento entre os módulos. Desta forma, as duas mecânicas nos dois jogos foram refatoradas para usar, respectivamente, os padrões Observer e Command. Para a mecânica de Replicação de Entidades, o padrão Prototype já estava sendo utilizado, uma vez que ele é o padrão na função de instanciação da engine Unity. Finalmente, apenas para o jogo 1, os recursos compartilhados entre várias instancias de uma mesma classe (mecânica de Repetição de Recursos) foram refatorados para usar atributos estáticos como uma alternativa ao uso do padrão Flyweight na linguagem C#. Essas modificações foram feitas principalmente na estrutura das classes do jogo e na forma como elas se comunicam, de forma que nenhuma regra ou comportamento dos jogos fossem modificados.

A dimensão das refatorações feitas podem ser vistas na Tabela 3, e os códigos refatorados estão disponíveis no Github[7,8] de forma versionada.

Tabela 3: Refatoração dos jogos (os valores referem-se à pasta de *assets* do jogo)

	Jogo 1	Jogo 2
Arquivos modificados	18.33% (22 de 120)	25.75% (17 de 66)
Linhas adicionadas	38.32% (358 de 934)	6.18%(58 de 922)
Linhas removidas	29.76% (278 de 934)	26.13%(241 de 922)

4.2. Comparação das versões

Nessa etapa, as duas versões de cada jogo passaram pela ferramenta Code Analysis. As métricas utilizadas foram índice de manutenibilidade que demonstra a complexidade do código em relação à quantidade de linhas e que expressa melhores resultados quanto maior seu valor; o acoplamento entre classes que se refere às referencias de uma classe dentro de outra; e a profundidade de herança que representa a quantidade de classes que herdam atributos de outras classes. Para as ambas, um valor baixo representa resultados melhores.

Na Tabela 4, estão os resultados das análises das duas versões dos projetos.

[7]https://github.com/VitorEstevam/tanks
[8]https://github.com/VitorEstevam/2d-roguelike

Tabela 4: Comparação dos Jogos

	Métrica	Original	Refatorado
Jogo 1	Índice de manutenibilidade	76	81
	Profundidade de herança	6	6
	Acoplamento entre classes	44	50
Jogo 2	Índice de manutenibilidade	75	81
	Profundidade de herança	5	5
	Acoplamento entre classes	38	44

À principio os resultados mostram-se positivos, pois houve um incremento de 5 e 4 pontos no índice de manutenibilidade. Além disso, a profundidade de herança manteve seu valor, mesmo com novas classes sendo adicionadas aos jogos.

Já o acoplamento entre classes mostra um crescimento de 6 pontos nos dois jogos. Isso se deve à divisão de responsabilidades entre classes, causado principalmente pelo padrão Command que encapsula comportamentos em classes menores e precisa ser referenciado na classe principal. O diagrama simplificado da estrutura de classes do jogo 2 (Tanks!) visto na Figura 2 exemplifica isso.

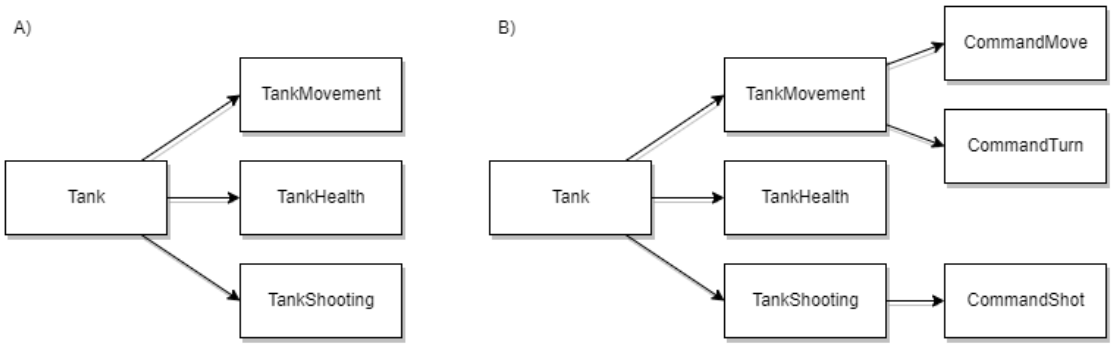

Figura 2. Diagrama de classes simplificado do jogo "Tanks!". A) Antes da refatoração, B) Depois da refatoração

O aumento do índice de manutenibilidade demonstra que o código se tornou menos complexo, ainda que o jogo tenha os mesmos comportamentos. Por conta disso, nos nossos exemplos, o aumento no acoplamento entre classes não demonstra necessariamente um problema na manutenibilidade do código, expressando apenas as relações entre as classes criadas pelos padrões e as já existentes no jogo.

Por mais que os resultados mostrem-se relativamente positivos, os dois jogos usados nas análises tratam-se de projetos pequenos e demonstrativos onde apenas dois padrões puderam ser efetivamente aplicados, o que explica os resultados semelhantes entre os dois. Além disso, os jogos divergem em vários elementos, mas as mecânicas refatoradas foram as mesmas. Os padrões Prototype e Flyweight não precisaram ser implemen-

tados, pois elementos da game engine (Unity) e da linguagem (C#) já faziam seus papéis nas aplicações sugeridas.

Os resultados indicam que pode haver uma melhora no quesito de manutenibilidade do projeto se esses padrões de projeto forem utilizados no desenvolvimento de jogos de pequeno porte.

5. Considerações Finais

O objetivo deste trabalho é investigar o uso de cinco padrões de projeto (Command, Flyweight, Observer, Prototype e State) no contexto do desenvolvimento de jogos. É possível distinguir duas contribuições principais: a primeira trata-se de um mapeamento dos padrões escolhidos e suas possíveis aplicações em jogos, feito através de um levantamento bibliográfico (Tabela 1), enquanto a segunda trata-se de uma comparação de análises estáticas em projetos antes e depois da aplicação de padrões GoF (Sessão 4.2).

Apesar de se tratar de uma pesquisa exploratória, o resultado da análise estática indica que o uso de padrões de projeto pode trazer melhorias na manutenibilidade dos projetos. Apesar disso, é necessário ressaltar que o recorte do experimento é limitado, uma vez que foram utilizados jogos de tamanho reduzido e que compartilhavam as mecânicas que foram refatoradas para usar os padrões. Além disso, o fato das refatorações terem sido feitas pelos mesmos desenvolvedores e na mesma game engine (Unity), pode ter tornado os resultados enviesados.

Desta forma, há possibilidades para aprofundar os resultados encontrados. O mapeamento entre padrões e mecânicas pode ser validado e estendido. É possível, por exemplo, coletar dados com desenvolvedores para verificar os dados da Tabela 1 e encontrar mais mecânicas possíveis. Naturalmente, o mapeamento de padrões também pode ser escalado para abranger mais padrões GoF ou os propostos por outros autores.

Sugere-se que mais trabalhos semelhantes a esse sejam feitos usando o mapeamento proposto para refatorar mais jogos e que jogos de diferentes estilos e complexidades sejam avaliados. Além disso, diferentes pessoas podem trabalhar nas refatorações para que o impacto dos padrões na qualidade possa ser avaliado levando em conta diferentes níveis de experiencia do desenvolvedor. Outra possibilidade são trabalhos visando avaliar outros aspectos da qualidade de um código, como a presença de *bugs*, problemas de performance e a presença de *bad smells* [Borrelli et al. 2020] no código.

Referências

Alghamdi, F e Qureshi, M. (2014). Impact of design patterns on software maintainability. *International Journal of Intelligent Systems and Applications(IJISA)*, 6(10):41–46.

Barakat, N. (2019). A framework for integrating software design patterns with game design framework. *Faculty of Informatics and Computer science,*.

Borrelli, A., Nardone, V., Di Lucca, G. A., Canfora, G., and Di Penta, M. (2020). *Detecting Video Game-Specific Bad Smells in Unity Projects*, page 198–208. Association for Computing Machinery, New York NY USA.

Figueiredo, R. (2015). Gof design patterns applied to the development of digital games. In *Proceedings of SBGames 2015*, Teresina PI Brazil. XIV SBGames.

Gamma, E., Helm, R., Johnson, R., Johnson, R. E., Vlissides, J., et al. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

Hunicke, R., LeBlanc, M., Zubek, R., et al. (2004). Mda: A formal approach to game design and game research. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, volume 4, page 1722. San Jose, CA.

IEEE (1993). *IEEE Standard for Software Maintenance, INSPEC Accession No. 4493167*. IEEE Computer Society.

Jones, M. (2022). Calculate code metrics - visual studio (windows). https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022.

Kounoukla, X.-C., Ampatzoglou, A., and Anagnostopoulos, K. (2016). Implementing game mechanics with gof design patterns. In *Proceedings of the 20th Pan-Hellenic Conference on Informatics*, PCI '16, New York, NY, USA. Association for Computing Machinery.

Liukku, O. (2020). Clean code: Analysing game code using agile programming practices. Master's thesis, Aalto University, Espoo.

Motogna, S., Cristea, D., and Molnar, D. (2022). Formal concept analysis model for static code analysis. *Carpathian Journal of Mathematics*, 38(1):159–168.

Nystrom, R. (2014). *Game Programming Patterns*. Genever Benning.

Paschali, M.-E., Volioti, C., Ampatzoglou, A., Gkagkas, A., Stamelos, I., and Chatzigeorgiou, A. (2021). Implementing game requirements using design patterns. *Journal of Software: Evolution and Process*, 33(12):e2399.

Thirumalai, C. S., Thirunavukkarasu, H., G, V., and K, S. (2017). Software complexity analysis using halstead metrics. In *International Conference on Trends in Electronics and Informatics*.

Unity (2022). Unity docs. https://docs.unity3d.com/. Accessed: 2022-09-18.

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

E84a Estevam, Vitor Braga.
Análise do impacto do uso de padrões de projeto em jogos / Vitor Braga Estevam. – 2022.
13 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Instituto UFC Virtual, Curso de Sistemas e Mídias Digitais, Fortaleza, 2022.
Orientação: Prof. Dr. Alysson Diniz dos Santos.

1. Padrões de projeto. 2. Jogos. 3. Manutenibilidade . 4. Desenvolvimento. I. Título.
CDD 302.23
