Universidade Federal de Pernambuco

Pós-Graduação em Ciência da Computação
Centro de Informática

Padrões de Projeto GOF aplicados ao Desenvolvimento de Jogos Eletrônicos

Roberto Tenorio Figueiredo

DISSERTAÇÃO DE MESTRADO

Recife, 03/2014

“Padrões de Projeto GoF Aplicados ao Desenvolvimento de Jogos Eletrônicos”

Por

Roberto Tenorio Figueiredo

Dissertação de Mestrado

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR: Prof. Dr. GEBER LISBOA RAMALHO

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

Recife, 03/2014

Padrões de Projeto GoF Aplicados ao Desenvolvimento de Jogos Eletrônicos

Roberto Tenorio Figueiredo
Orientador: Dr. Geber Lisboa Ramalho

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

Examinada por:

_____

Presidente, Prof. Dr. André Luís de Medeiros Santos (UFPE / CIn)

_____

Prof. Dr. Charles Andryê Galvão Madeira (UFRN / IMD)

_____

Prof. Dr. Geber Lisboa Ramalho (Orientador - UFPE / CIn)

Agradecimentos

Dedico este trabalho primeiramente a Deus, mestre dos mestres, força maior que me fez chegar até aqui, sempre ao meu lado, me ajudando a vencer todos os desafios que a vida tem me colocado. Agradeço aos mestres da loja branca que me auxiliaram em todos os momentos que precisei e nunca me abandonaram.

A meu pai Roberto Figueiredo, que nunca mediu esforços para me ajudar a ser quem hoje sou e continua sempre me ajudando em cada passo de minha vida.

À minha mãe Fátima Eleonora (*in memorian*), que de onde ela estiver, tenho a certeza que intercede por mim nos momentos mais difíceis que a vida tem me colocado.

À minha esposa Carla Brandão Figueiredo, que tanto me incentivou a fazer este mestrado, me deu forças quando quis desistir e coragem quando precisei lutar.

A toda minha família, que sempre enxergou em mim um exemplo de perseverança na busca da realização dos meus sonhos e dos meus ideais.

Ao meu Tio Marcos Tenório, que me deu abrigo todas as vezes que precisei ir a Recife falar com meu orientador.

Aos meus amigos Augusto César Ribeiro Silva e Márcia Macedo, que me ajudaram durante todo o curso, principalmente nas disciplinas, trabalhando em equipe, me ajudando a construir os projetos e aguentando todas as minhas maluquices.

E ao meu orientador, Geber Lisboa Ramalho, que mesmo de longe, me ajudou a traçar os rumos certos desta pesquisa, com muita paciência, vontade e profissionalismo. Muitas das informações que recebi, certamente serão repassadas aos meus futuros orientados.

Obrigado e um abraço a todos.

“O céu está em vossas mãos, a terra sob os vossos pés, seja qual for vossa ordem, não mandeis geadas, chuva ou neve até que a maravilha que me haveis concedido se materialize”.

Ramsés II

Resumo

No contexto de um mercado cada vez mais exigente e competitivo, as empresas desenvolvedoras de jogos têm adotado bibliotecas e frameworks para realizar funções básicas (como detectar dispositivos de entrada, acessar interfaces de hardware, renderizar imagens, etc.), ganhando tempo e deixando o programador mais focado na lógica do jogo. Neste contexto, a utilização de padrões de projeto poderia ser mais uma ferramenta importante para auxiliar o desenvolvimento de jogos. Infelizmente, ainda há poucos trabalhos devotados a catalogar padrões de projetos em jogos. Um grupo bastante afamado de padrões de projeto é o GoF (*Gang of Four*), composto por vinte e três padrões com uso reconhecido em aplicações comerciais, mas com utilização bastante limitada no desenvolvimento de jogos. Este trabalho analisa alguns padrões de projeto propostos para o desenvolvimento jogos e propõe maneiras de se utilizar todos os padrões GoF neste contexto, não somente mostrando o resultado final, mas apresentado o “antes e depois” da aplicação desses padrões e onde eles estão auxiliando o programador em sua tarefa. Uma avaliação preliminar do uso destes padrões nos jogos foi positiva.

Palavras-chave: Desenvolvimento de Jogos. Padrões de Projeto. GoF

Abstract

In the context of an increasingly demanding and competitive marketplace, companies that develop games have adopted frameworks and libraries to perform basic functions (how to detect input devices, access hardware interfaces, rendering images, etc.), gaining time and leaving the programmer more focused on the game logic. In this context, the use of design patterns could be an important tool to aid the development of games. Unfortunately, there are few works devoted to catalog design patterns in games. A very famous group of design patterns is the GoF (Gang of Four), composed of twenty-three with recognized standards in commercial applications use, but with very limited use in game development. This paper analyzes some proposed standards project to develop games and proposes ways to use all the GoF patterns in this context , not only showing the final result , but presented the “before and after” the application of these standards and where they are assisting the programmer in your task. A preliminary evaluation of the use of these patterns in the games was positive.

Keywords: Game Development. Design Patterns. GoF

Lista de Figuras

Lista de Tabelas

Sumário

1. Introdução

Os jogos eletrônicos começaram em telas monocromáticas onde retas e quadrados representavam todo o universo que esses jogos eram capazes de reproduzir. Nesta época, a programação era feita em Assembler em ambientes de desenvolvimento bastante rudimentar.

O avanço da tecnologia proporcionou computadores com capacidades de processamento e qualidade visual que permite a recriação de ambientes virtuais quase perfeitos. Essa nova realidade fez surgir uma demanda por jogos cada vez mais realísticos, com efeitos visuais que beiram a perfeição. Essa demanda exigiu a profissionalização dos desenvolvedores de jogos e fez surgir um mercado de bilhões de dólares (PERUCIA, BERTHÊM, *et al.*, 2005).

O desenvolvimento de um jogo requer equipes com os mais diversos tipos de conhecimentos, como engenharia de software, computação gráfica, bancos de dados, inteligência artificial, além de artes, música, psicologia e muitas outras (FERNANDES, 2002). Com o objetivo de melhorar a qualidade e a velocidade de produção desses jogos, muitas ferramentas, frameworks e técnicas foram criados (FERNANDES, 2002). A maioria destes elementos facilita apenas a manipulação da arquitetura do hardware, compatibilidade com periféricos, física dos objetos e detalhes de baixo nível, não atuando de maneira eficaz nas áreas de mais alto nível, que são específicas de cada jogo. As ferramentas que propõe facilitar a construção da lógica do jogo, geralmente são muito restritas a um único tipo de jogo e quando são mais genéricas, os benefícios gerados tornam-se mínimos.

1.1. Justificativa

Este trabalho está sendo feito devido à alta competitividade no mercado de jogos, que exige qualidade e agilidade no lançamento de novos títulos aliada a carência de técnicas que auxiliem, de maneira geral, o desenvolvimento da lógica específica de cada jogo.

O desenvolvimento rápido e sem bugs pode ser facilitado pela reusabilidade e o desacoplamento de componentes de software. Existem muitas técnicas e ferramentas na engenharia de software que podem ajudar a promover esses atributos em um jogo. Algumas vêm sendo muito usadas, mas outras nem tanto, com é o caso dos padrões de projeto. Este trabalho vem a complementar a literatura de desenvolvimento de jogos inserindo uma análise da aplicação dos 23 padrões de projeto GoF em jogos.

1.2. Objetivo

O objetivo deste trabalho é propor uma melhoria na reusabilidade e na qualidade das classes criadas para resolver a lógica da programação dos jogos eletrônicos, com a aplicação dos padrões de projeto definidos pela GoF.

1.3. Metodologia

O método utilizado no desenvolvimento deste trabalho foi o exploratório, onde foi feito um levantamento de informações relevantes sobre o tema em referências bibliográficas; e o descritivo, onde foi apresentada a descrição de aplicações práticas, sem interferência do pesquisador.

A coleta dos dados foi feita de caráter documental, ou seja, foi realizado o levantamento de informações em artigos, livros e trabalhos produzidos por terceiros; e experimental, onde foi utilizado um experimento em laboratório de informática com o objetivo de validar a hipótese apresentada.

1.4. Estrutura do Trabalho

Os assuntos e temas abordados nesta dissertação foram divididos em seis capítulos.

O capítulo 2 apresenta a evolução no desenvolvimento de jogos com ênfase nos problemas encontrados e nas soluções que foram surgindo e resolvendo tais problemas. Ele mostra ainda a evolução dos ambientes de desenvolvimento, das linguagens de programação, o surgimento dos motores de jogos e os problemas enfrentados pelos desenvolvedores de jogos na atualidade.

O capítulo 3 apresenta o conceito de Padrões de Projeto, sua origem, evolução e utilização prática. Mostra os padrões GoF, sua classificação e divisão.

O capítulo 4 é o estado da arte, apresentando uma pesquisa detalhada junto a diversos autores que falam da utilização de padrões de projeto em jogos, alguns deles falam do GoF, mas de maneira limitada; outros sugerem seus próprios padrões, onde muitos são na verdade, requisitos e funcionalidades que os autores desenvolveram e chamam de padrões.

O capítulo 5 é a proposta do trabalho, apresentando um-a-um os padrões de projeto GoF e sua utilização em jogos. Cada padrão possui o conceito, quais trabalhos estão relacionados a ele, onde se aplica em jogos e como o padrão pode ajudar a criação e desenvolvimento desses jogos resolvendo de maneira reusável situações que podem surgir durante a programação.

O capítulo 6 é a descrição de um experimento prático, utilizado para validar os conceitos de reusabilidade e qualidade na criação de jogos proposta neste trabalho, com o uso dos padrões de projeto GoF .

2. O Problema

Este capítulo tem como objetivo apresentar alguns dos principais desafios na programação de jogos eletrônicos.

2.1. Evolução da programação de jogos digitais

No início da programação de jogos eletrônicos, as máquinas eram muito limitadas, tanto em diversidade quanto na sua capacidade de processamento. Existia processadores de 8-bits, rodando a cerca de quatro megahertz com memória que girava entre 48 e 64 kilobytes. Os gráficos eram limitados e a terceira dimensão praticamente inexistente e sem capacidade de mapear texturas (PERUCIA, BERTHÊM, *et al.*, 2005).

A programação para essas máquinas exigia um super esforço no sentido de contornar estas limitações com o intuito de obter melhores resultados. Todos os jogos tinham que ser codificados em Assembler, a fim de atingir os requisitos de velocidade e espaço, pois, apesar de existirem linguagens de alto nível, como o C, os compiladores não eram capazes de gerar códigos pequenos o suficiente para ser usado no desenvolvimento de jogos. Isto significava que para portar um jogo entre plataformas era preciso reescrever totalmente o código (PERUCIA, BERTHÊM, *et al.*, 2005). Além disso, o desenvolvimento era feito apenas por pessoas altamente especializadas, devido às dificuldades naturais da linguagem de máquina (ROLLINGS e MORRIS, 2003). A Figura 2-1 mostra um antigo ambiente de programação em Assembler.

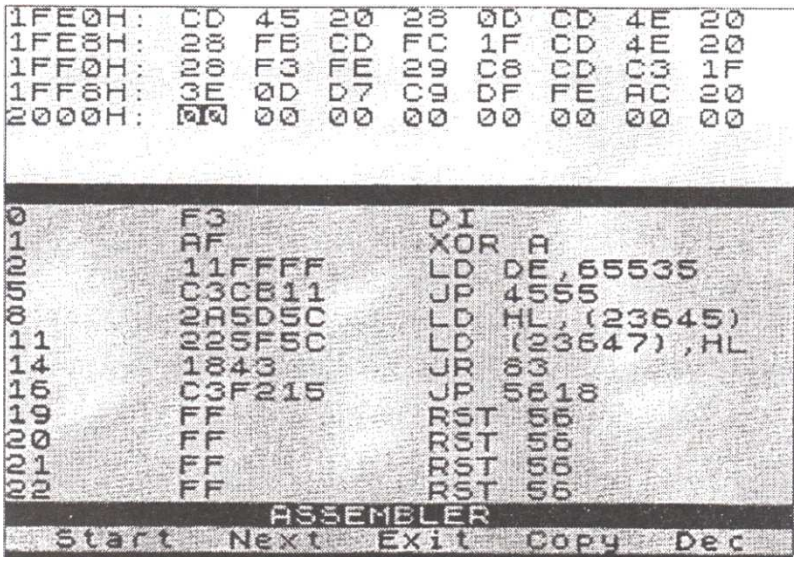

Figura 2-1: Antigo ambiente de programação Assembler
Fonte: (PERUCIA, BERTHÊM, *et al.*, 2005)

No meio destas desvantagens, algumas vantagens eram encontradas, como, por exemplo, a inexistência de variação dentro dos limites de uma plataforma. O programador sabia que a máquina usada no desenvolvimento era exatamente a mesma utilizada pelos usuários finais, assim como acontece com os consoles de hoje, onde a máquina final é única e perfeitamente conhecida pela equipe de programação. Isto evita problemas de escalonamento do código para funcionar em plataformas diferentes, além de evitar problemas com velocidade e espaço, comuns quando um software é testado em uma máquina superior à máquina do usuário final (PERUCIA, BERTHÊM, *et al.*, 2005).

Devido às limitações de hardware mencionadas, as prioridades dos desenvolvedores na codificação dos jogos eram bastante diferentes das prioridades de hoje. Questões como velocidade e otimização ainda são discutidas, porém não são priorizadas tanto quanto costumava ser. Hoje, efeitos visuais realistas e inteligência artificial avançada direcionam o foco principal dos desenvolvedores. Apesar disto, as limitações do passado não eram necessariamente uma coisa ruim. Devido a estas restrições, sem nenhum apelo visual, muita ênfase foi colocada sobre a jogabilidade, tornando os pequenos jogos boas experiências a serem vividas (ROLLINGS e MORRIS, 2003).

Os jogos produzidos hoje em dia possuem gráficos e poder de processamento infinitamente melhores, mas a jogabilidade não cresceu na mesma proporção e em alguns casos até piorou. Muitas ideias interessantes não são colocadas no mercado devido às pressões do “mercado de massa”. O fato é que, em muitos casos, ainda estão sendo jogados os mesmos jogos antigos, porém, em uma embalagem nova e brilhante (ROLLINGS e MORRIS, 2003).

2.1.1. Ambientes de desenvolvimento

No início da programação dos jogos, os códigos eram desenvolvidos diretamente na máquina e o planejamento era feito apenas na cabeça de quem ia programar. Os comandos eram escritos em papéis e depois manualmente convertidos para uma sequência em hexadecimal e posteriormente as sequencias eram digitadas no ambiente, que permitia carregá-las diretamente na memória (ROLLINGS e MORRIS, 2003).

Com o passar do tempo, surgiram montadores e o ambiente de desenvolvimento Assembler, onde os programadores podiam digitar, além das sequencias em hexadecimal, alguns comandos, conhecidos pelo ambiente e o próprio ambiente se encarregava de convertê-los em hexadecimal.

Até a década de 80, a única linguagem de desenvolvimento era o Assembler que consistia apenas em uma tela com dígitos hexadecimais e códigos estranhos que, visualmente, nada significavam (Figura 2-1). Embora tenha facilitado o

desenvolvimento, operar o Assembler ainda era muito difícil. Além disso, o ambiente levava parte da pouca memória que as máquinas tinham na época, o que significava que, para um jogo usar toda a memória disponível, ele teria que ser montado em partes e juntado apenas em tempo de execução (ROLLINGS e MORRIS, 2003).

Depuração também era uma atividade restrita e difícil, porque o próprio depurador exigia alguma memória do sistema, concorrendo com o ambiente de desenvolvimento e o próprio jogo. Esta dificuldade foi, em parte, compensada pelo fato de que o desenvolvedor geralmente conhecia todo o código e a única maneira de obter a velocidade desejada era retirar completamente o sistema operacional e escrever tudo diretamente no hardware.

Com tantas limitações, os desenvolvedores achavam que era impossível um jogo ser implementado em outra linguagem e ter uma performance aceitável. Tudo mudou com o lançamento do jogo DOOM, mostrado na Figura 2-2, totalmente escrito em C e sucesso entre os usuários de jogos (ROLLINGS e MORRIS, 2003).

Figura 2-2: Doom, totalmente escrito em C

Hoje em dia, os dígitos em hexadecimal e os códigos estão escondidos atrás de várias camadas de interface com o usuário. A vida de um desenvolvedor de jogos tem sido facilitada por uma enorme variedade de excelentes compiladores, bibliotecas de otimização de jogos que automaticamente se aproveitam de aceleração de hardware, a depuração remota, vários monitores de depuração, e, em alguns casos, a capacidade de desenvolver o jogo usando um emulador de hardware mais potente, melhor que o hardware alvo.

2.1.2. As linguagens de desenvolvimento

O Assembler praticamente não é mais utilizado pelos desenvolvedores atuais, a maioria dos produtos é escrita em uma linguagem de alto nível, com apenas pequenas partes críticas sendo escritos em baixo nível. Doom mudou o paradigma da programação de jogos de computador para sempre (PERUCIA, BERTHÊM, *et al.*, 2005).

Doom teve seu código compilado no Watcom C/C++ que gerava uma aplicação DOS 32 bits muito rápida e otimizada e impulsionou a indústria a migrar rapidamente de seus TASM (*Borland Turbo Assembler*) e MASM (*Microsoft Assembler*) para o Watcom C/C++ (ROLLINGS e MORRIS, 2003).

O compilador Watcom ter produzido aplicativos de 32 bits para DOS era um grande benefício para os desenvolvedores de jogos. A diferença entre uma aplicação de 16 bits (produzida pelos antigos compiladores) e uma de 32 bits é o modelo de memória. Em um aplicativo de 16 bits, a quantidade de memória disponível para o pedido era limitada a 640Kb (na verdade, um megabyte, mas o sistema operacional utilizava o resto), a menos que o desenvolvedor, através de uma interface lenta e sem recursos, mapeasse a memória expandida da máquina.

A mudança para o modelo de 32-bit resolveu este problema de uma vez por todas, com o uso de ponteiros de 32 bits que permitiram a aplicação fazer uso direto de quatro gigabytes de memória. Com isso o programador ganhou, ao mesmo tempo, uma ferramenta de alto nível para seu desenvolvimento e um modelo de memória simples e flexível para executar seus trabalhos (ROLLINGS e MORRIS, 2003).

O domínio do Watcom C/C++ na indústria de jogos continuou até alguns anos mais tarde, quando a Microsoft voltou sua atenção para o mercado de jogos e sua dependência ao DOS, pois a empresa estava tentando acabar com o DOS (PERUCIA, BERTHÊM, *et al.*, 2005).

Com o lançamento do Windows 3.1 e NT 3.51, todos os aplicativos novos que estavam sendo lançados para o PC foram feitos para o Windows, exceto os jogos, principalmente porque o Windows abstraía do hardware, de tal forma que era impossível obter a velocidade necessária para a execução de um jogo de computador.

Com o objetivo de trazer os jogos para o Windows e enterrar o último suspiro do DOS, a Microsoft lançou o *Game SDK*, ou *Software Development Kit* (precursor do DirectX), que consistia em um conjunto de bibliotecas de programação que facilitava o acesso rápido aos dispositivos de hardware (PERUCIA, BERTHÊM, *et al.*, 2005).

Com o lançamento do Windows 95, a Microsoft “anunciou” a morte dos sistemas de 16 bits e do MS-DOS, porém, a indústria de jogos praticamente ignorava o novo sistema operacional, exceto para verificar a compatibilidade com os seus executáveis

DOS/4GW (extensor 32-bits do DOS, usado pelo compilador C Watcom). Vendo que a indústria de jogos era a única que continuava produzindo para o DOS e tentando trazer essa indústria de uma vez por todas para o Windows, a Microsoft lançou o DirectX, uma biblioteca que permite o acesso direto ao hardware e, se necessário, tirar vantagem de qualquer aceleração fornecida pelo hardware.

A ideologia por trás do DirectX era fornecer uma interface padrão para o hardware de maneira mais simples e eficiente do que a solução embutida no DOS/4GW, porém, inicialmente, não seduziu os desenvolvedores. Vendo a pouca aderência da indústria de jogos, a Microsoft lançou o DirectX 2, que aproveitou o componente Microsoft recém-desenvolvido chamado de *Component Object Model* (COM). O COM é uma técnica orientada a objetos que permite que objetos sejam facilmente acessados a partir de outras linguagens que não aquela em que o objeto foi originalmente escrito.

Com o DirectX 2 e a versão 4 do Visual C++ (compilador para Windows 32 bits mais rápido do que o Watcom) os desenvolvedores sentiram que era o momento de migrar. A partir deste momento, o DirectX amadureceu através de onze versões e não há desenvolvimento de jogos comerciais para PC feitos sem ele, com exceção de alguns casos que são baseados em OpenGL, biblioteca “concorrente” ao DirectX.

O crescente poder computacional, a diversidade de sistemas operacionais e a variedade de estilos de máquinas (PCs, *tablets*, *smartphones*, consoles, entre outros) têm feito a vida do desenvolvedor mais difícil. A variedade de hardwares e a complexidade subsequente de programação da API, agravado pela quantidade de novas informações para absorver e a exigência do mercado são definitivamente maiores do que antes e o trabalho do *Game Designer* tornou-se uma tarefa mais difícil do que outrora.

Segundo Rollings e Morris (2003), no que diz respeito ao desenvolvimento para consoles, a linguagem C só foi realmente utilizada como linguagem padrão a partir da quinta geração, com o surgimento do Playstation da Sony, antes disso, o Assembler dominava. Além disso, as empresas fabricantes dos consoles sempre se mantiveram fechadas, ou seja, escondendo detalhes de seus hardwares e disponibilizando seus manuais e kits de desenvolvimento apenas mediante o pagamento de enormes quantias de dinheiro o que impedia as pequenas e médias empresas desenvolvedoras entrarem neste mercado.

Com o avanço da internet e o crescente interesse em programação para consoles, diversas comunidades virtuais surgiram e começaram a compartilhar informações descobertas sobre os aparelhos, fazendo surgir um grande número de kits de desenvolvimento gratuitos escritos por amadores entusiastas. Em muitos casos, estes kits eram tão bons que os desenvolvedores oficiais passaram a usá-los ao invés dos kits oficiais.

2.1.3. Motor de Jogos

O motor de jogo é uma aplicação que reúne diversas bibliotecas e ferramentas prontas, associadas a um IDE, para o desenvolvimento facilitado e ágil de jogos eletrônicos. Nele, o desenvolvedor pode ter seu foco principal na dinâmica do jogo, tornando transparentes detalhes de hardware, da física dos objetos do jogo e outros detalhes de baixo nível. Além disso, eles fornecem uma abstração do hardware permitindo ao programador criar sem a necessidade de conhecer a arquitetura da máquina dos usuários. Muitos motores são desenvolvidos a partir de bibliotecas existentes, como o DirectX e o OpenGL. Com o motor de jogo é possível criar, além de jogos, diversos tipos de aplicações, como demonstrações e simulações (BUSSO e FERREIRA, 2006).

Antes do surgimento dos motores, os jogos eram programas de computador independentes, escritos sempre "do zero" e todos os detalhes deveriam ser implementados. O termo "motor do jogo" surgiu na década de 90, porém aplicações da década de 80 ganharam o status de motores de jogos devido a forma com que foram utilizados.

Com o sucesso de DOOM e outros jogos de tiro, desenvolvedores passaram a escrever novos jogos utilizando diversos códigos copiados destes, ao invés de reescrevê-los. Com o passar do tempo, diversos desenvolvedores licenciaram o núcleo de seus jogos para que outros pudessem reutilizá-lo nos seus. Com a evolução da ideia, companhias de desenvolvimento passaram a desenvolver seus motores e a maioria de seus jogos era criada com estes motores. Posteriormente, empresas especializadas em criar motores foram surgindo (BUSSO e FERREIRA, 2006). Hoje em dia, existem desde motores gratuitos, até motores que custam alguns milhares de dólares. Entre os motores mais conhecidos estão o Microsoft XNA, *Unreal Development Kit*, *Unity Engine*, *Game Maker*, *RPG Maker*, 3D *Game Builder*, *Game Creator*, *KODU Game Lab*, *Adventure Maker* e *GameKa*.

Atualmente, os motores de jogos são muito utilizados para o desenvolvimento, desde grandes indústrias até desenvolvedores amadores ou de pequeno porte. A força destas ferramentas está principalmente em facilitar a manipulação da arquitetura do hardware, compatibilidade com periféricos, física dos objetos e detalhes de baixo nível, deixando o desenvolvedor livre para trabalhar diretamente na lógica do jogo (BUSSO e FERREIRA, 2006).

2.1.4. DSL (*Domain Specific Languages*)

Além dos motores de jogos, alguns desenvolvedores têm a sua disposição as chamadas DSLs. A DSL não é, necessariamente, uma linguagem de programação, mas é uma

linguagem com poder de criação com foco restrito a um domínio particular. Cria suas aplicações através de notações e abstrações. São mais restritas que uma linguagem de programação de propósito geral, porém, são mais poderosas (FURTADO, 2008).

DSL não possuem um compilador próprio. Esse trabalho de compilação ou interpretação fica cargo da API (*Application Programming Interface*). São geralmente pequenas (micro-linguagens) e podem ser vistas, tanto como linguagem de programação como também linguagem de especificação (FURTADO, 2008). O exemplo mais conhecido de DSL é o HTML (*HyperText Markup Language*), mas existem outras, como LaTex, Sql, Yacc, JavaDoc, entre outras.

Muitos jogos já foram desenvolvidos utilizando DSL, um exemplo é o jogo de perguntas e respostas (*quiz games*) demonstrado por Furtado (2008).

2.2. A indústria e o desenvolvimento hoje

A quantidade de trabalho, que é realizado automaticamente pelo compilador e pelos sistemas operacionais, aumentou dramaticamente. Ao produzir algumas bibliotecas e frameworks para acessar as interfaces de hardware, tornou-se possível concentrar mais na lógica do jogo. Todo o trabalho de inicialização da máquina, detecção de joystick, compatibilidade, entre outros, tornou-se transparente e a produção ficou mais próxima do jogo em si (PERUCIA, BERTHÊM, *et al.*, 2005).

Apesar de todas as facilidades existentes, o alto faturamento da indústria de jogos que, segundo Guerra (2012), girou em torno de 420 milhões de dólares em 2011, somente no Brasil, gerou uma disputa acirrada entre empresas desenvolvedoras, que tentam suprir as demandas de um mercado cada vez mais exigente e competitivo.

2.2.1. Reusabilidade

Um dos conceitos mais buscados pela indústria de jogos é a reusabilidade. Trata-se da produção de componentes de software a serem utilizados em mais de um projeto (ROLLINGS e MORRIS, 2003). Este conceito está diretamente associado ao desacoplamento entre as partes do jogo, o que por consequência, traz uma maior facilidade na correção de bugs e alteração e inclusão de novos elementos em um jogo nas fases finais de seu desenvolvimento (FERNANDES, 2002).

O desenvolvedor de jogos medianos, em geral, acredita que o uso de componentes compartilhados resultará em jogos muito simulares. Isso pode ser verdade, dependendo de que partes são reutilizadas, mas existe uma diferença entre os tipos de reusabilidade.

Esta é a diferença entre o desenvolvimento monolítico e baseada em componentes e as mentalidades que vão com eles.

Desenvolvimento monolítico é a prática de desenvolvimento de um programa constituído em partes separadas, mas as partes estão interligadas de modo a torná-las inseparáveis. Cada parte do sistema é interdependente das demais, porém, as interfaces que as unem dependem do conhecimento interno das partes a serem unidas, o que implica em partes fortemente acopladas.

O contraste de um sistema monolítico é um sistema baseado em componentes, como por exemplo, os sistemas baseados em COM. Nestes casos, as interfaces de conexão entre os componentes não dependem de detalhe internos destes componentes, deixando o sistema fracamente conexo. O DirectX é um exemplo de bibliotecas para jogos que utiliza a tecnologia COM.

A reusabilidade reduz o tempo de desenvolvimento de projetos futuros, pois novos jogos já terão vários detalhes implementados. Esses detalhes já foram testados anteriormente e a possibilidade de bugs é pequena. Reduzindo tempo e bugs, automaticamente reduz-se os custos de desenvolvimento e a qualidade dos jogos melhora.

2.2.2. Desafios

Observando os números, somado à grande concorrência no setor, os desenvolvedores, a cada dia, buscam novas formas de programação que aliem custo baixo, agilidade (tempo de desenvolvimento baixo), qualidade e aceitação em um mercado tão lucrativo quanto competitivo e a reusabilidade tem um papel importante neste contexto.

Existem diversas ferramentas que auxiliam nestas tarefas, como os motores de jogos mencionados no tópico 2.1.3, muitas vezes chamados de frameworks. Esses frameworks já são largamente utilizados na produção e desenvolvimento de jogos, mas existem outras ferramentas e técnicas que podem ajudar que são menos utilizadas, como por exemplo, algumas DSLs (*Domain Specific Languages*) e os padrões de projeto.

Neste contexto surge a hipótese de que padrões de projeto GoF, desenvolvidos para aplicações comerciais, podem ser aplicados no desenvolvimento de jogos eletrônicos, com o objetivo de melhorar a reusabilidade e a qualidade dos códigos implementados, aliado ao uso de frameworks.

3. Padrões de Projeto

O planejamento de aplicações é um processo considerado bastante complicado, porém algumas soluções são propostas pela Engenharia de Software com o intuito de tentar tornar menos árduo esse contexto. Neste capítulo veremos uma introdução sobre padrões de projeto, um dos ramos propostos pelos engenheiros de sistemas.

3.1. Conceito

Segundo Gamma, (et al., 2000), os padrões de projeto representam um avanço considerável para a área de orientação a objeto, uma vez que disponibiliza um catálogo de planos de projeto que admite a reutilização dessas soluções que foram testadas e provaram ser eficientes para a resolução de problemas semelhantes. É dado um nome e uma descrição abstrata para essas estruturas, admitindo uma comunicação de um projeto em termos de uma linguagem de mais alto nível que as notações básicas. Por estar em um nível abstrato, a sua reutilização é admitida no projeto de novas aplicações ou na melhoria de aplicações existentes, independentemente da implementação.

O nível de abstração dos padrões de projetos pode variar bastante, mesmo assim, tem características comuns e em diversos casos estão fortemente relacionados uns com os outros. Apresentam possíveis soluções para problemas que ocorrem no decorrer do desenvolvimento do projeto e tem por objetivo facilitar a reutilização dessas soluções no decorrer do projeto. A classificação dos padrões une os diferentes padrões em categorias para facilitar sua compreensão e utilização.

O padrão de projeto é uma amostra de experimento que já foi amplamente testado e obteve resultados satisfatórios podendo assim ser usado como um guia para resolver problemas particulares para arquiteturas orientadas a objeto. É uma opção reutilizável, que se baseia na abstração das escolhas que foram bem sucedidas utilizadas para atender um problema conhecido e é composta de quatro elementos básicos: o **nome**, o **problema** a ser resolvido, a **solução** para o problema e os **resultados** obtido com a aplicação do padrão (SHALLOWAY e TROTT, 2002).

> "Um padrão descreve um problema que ocorre inúmeras vezes em determinado contexto, e descreve ainda a solução para esse problema, de modo que essa solução possa ser utilizada sistematicamente em distintas situações. Cada padrão tem uma característica diferente para ajudar em algum lugar onde se precisa de mais flexibilidade ou precisa encapsular uma abstração ou de se fazer um código menos casado.” (GAMMA, HELM, *et al.*, 2000)

Um padrão de projeto nomeia, abstrai e identifica aspectos problemáticos comuns e propõe uma solução padrão para esses problemas. O padrão de projeto é usado para identificar as classes e instâncias que participam, seus papéis, colaborações e a distribuição das responsabilidades entre esses participantes. Cada padrão de projeto observa um problema específico ou tópico particular de projeto orientado a objeto que necessita ser resolvido. Ele descreve em que situação deve ser aplicado, para que tipo de problema resolver, especifica também as consequências, custos e benefícios de sua utilização (GAMMA, HELM, *et al.*, 2000).

3.2. Origem

A ideia original de padrões de projetos ou *Design Patterns*, surgiu com o arquiteto austríaco Cristopher Alexander, no momento em que ele propôs um catálogo de padrões para arquitetura. Após ter observado inúmeras cidades medievais, ele teve a certeza da existência de padrões que tornavam essas cidades mais harmoniosas e belas (MARTINS, 2010).

Então ele sentiu-se interessado em traduzir tais padrões de uma maneira que fosse fácil para o entendimento e para a aplicação em projetos futuros de construção ou até mesmo para melhoria dos projetos já existentes, isso resultou na publicação, em 1977, do livro mais famoso por ele escrito "*A Pattern Language: Towns, Buildings, Construction*". Nesse livro, ele apresentou diversas regras e usou ilustrações para apresentar métodos exatos para a construção de desenhos práticos, seguros e atrativos que poderiam ser reutilizados para outros projetos de construção. Um dos maiores benefícios dessa obra foi que, ao deixar esses padrões reaplicáveis, foi de grande valia para auxiliar na tomada de decisões de arquitetura (MARTINS, 2010).

Os padrões de Alexander buscavam oferecer um leque de ideias que foram provadas através de testes, para indivíduos e comunidades com objetivo de serem utilizadas em construções, desta forma, mostrando o quanto belo, confortável e flexível podem ser as construções dos ambientes. Ele argumentava que os métodos utilizados na arquitetura não atendiam as necessidades reais da sociedade e dos indivíduos. Ele queria criar estruturas para melhorar a qualidade de vida das pessoas. A qualidade de vida das pessoas era o objetivo principal de Alexander com a utilização dos padrões.

Alexander encontrou abordagens recorrentes na arquitetura e as capturou em descrições e instruções de como resolver essas abordagens, chamadas padrões. Cada padrão relata um problema que acontece repetidamente em um ambiente, ele descreve o núcleo para a solução do problema, de modo que essa solução possa ser usada inúmeras vezes.

Em 1987, Kent Beck e Ward Cunningham começaram a utilizar os padrões de projeto, inicialmente desenvolvidos para a arquitetura, no desenvolvimento de software, apresentando seus resultados na conferência de OOPSLA (Conferência de Desenvolvimento de Sistemas Orientados a Objeto), naquele mesmo ano. Eles foram

pioneiros da utilização da metodologia de desenvolvimento ágil de aplicações e da elaboração de aplicações baseadas em testes. Foram responsáveis pelo desenvolvimento de um conjunto de padrões para serem utilizados na criação de interfaces de usuário elegantes em Smalltalk (THOMAZINI NETO e JANDI JR, 2006).

Nesse mesmo período Jim Coplien desenvolvia um catálogo de padrões para C++ denominados idiomas. Durante esse período Erick Gamma pesquisava em sua tese de doutorado a respeito do desenvolvimento de sistemas orientados a objeto, e percebeu a importância de acumular de forma explícita as estruturas de projetos que apareciam com mais frequência (THOMAZINI NETO e JANDI JR, 2006).

Na década de 90 os desenvolvedores de sistemas levaram a sério a ideia do uso de padrões, porém, a popularidade desses padrões só veio quando Erich Gamma, John Vlissides, Ralph Johnson e Richard Helm, conhecidos como "*The Gang of Four*" ou GoF, lançaram o livro "*Design Patterns*" em 1995, onde descrevem 23 padrões de projeto para desenvolvimento de software (THOMAZINI NETO e JANDI JR, 2006). Foram considerados os maiores entusiastas dos *Design Patterns.*

O GoF tinha o objetivo comum de estudar e encontrar padrões e soluções amplamente empregadas no desenvolvimento de software para registrá-las de maneira a torná-las uma base para ser utilizada em uma determinada situação futura. O conceito de Padrões de Projeto tem sido utilizado nos mais diferentes ramos, englobando desde o desenvolvimento de software até o uso em arquitetura e educação.

3.3. Utilização Prática

Segundo LARSEN (et al., 2006), a utilização do padrão de projeto é de suma importância por diversos fatores. Entre eles, facilita a manutenção, uma vez que deixa o projeto bem documentado e em um formato internacionalmente conhecido; aumenta a escalabilidade do projeto, ou seja, a habilidade do sistema em manipular e suportar a carga total requerida pelos recursos; e permite a reutilização de todo o material, resultando no aumento significativo da produtividade. Pode ser aplicada em qualquer situação, independente da linguagem de programação que esteja sendo utilizada.

Padrões de projeto são considerados uma forma para representar, registrar e reutilizar microarquiteturas de projetos repetidas vezes quanto forem necessários, e também a experiência acumulada por projetistas durante todo o desenvolvimento do projeto. Eles existem como uma forma que facilita no desenvolvimento de aplicações, fazendo com que de certa forma "economize neurônios", uma vez que são soluções mais ou menos prontas para resolver problemas mais ou menos comuns no dia a dia do programador.

Segundo GAMMA (*et al.*, 2000), o importante dos padrões de projeto é saber com exatidão qual o problema e qual a melhor solução para ele. É importante fazer uma análise do caso e da solução para a situação, pois é através dessa análise e do

conhecimento dos padrões de projetos que você pode decidir qual usar, como usar e porque usar um determinado padrão de projeto, se realmente for a melhor escolha para resolver esse problema.

Ao fazer uso dos padrões de projeto, os sistemas tornam-se mais fáceis de serem lidos, se bem aplicados, ajudando assim na manutenção do sistema, uma vez que é mantida uma linha de raciocínio padrão no desenvolvimento.

Segundo GAMMA (*et al.*, 2000), com uso dos padrões de projeto é determinado um vocabulário comum para a solução dos problemas, fazendo com que seja desenvolvido um projeto com um bom nível de coesão e reusabilidade, o que torna fácil o processo de manutenção do software. Lembrando que a manutenção é justamente o que toma o maior período de tempo de vida de todo projeto de desenvolvimento.

Outro ponto que deve ser levado em consideração, é que na maioria das vezes um projeto não é desenvolvido sozinho. Então se toda a equipe de desenvolvimento já tem conhecimento dos padrões de projeto, caso seja necessário explicar como foi desenvolvida uma funcionalidade qualquer, será economizado bastante tempo, dessa forma não será preciso que “baixe o nível”. Ao invés de dizer “foi feito com que o construtor da classe não pudesse ser visível para as demais e foi criado um método getInstance para retornar a única instancia da classe”, diria simplesmente “foi usado o Singleton”, e todos já entenderiam.

Os padrões de projeto utilizam de forma eficiente herança, polimorfismo, composição, modularidade e abstração, conceitos muito importantes para o desenvolvimento de projetos orientados a objeto, construindo dessa forma um código reutilizável, eficiente, de alta coesão e com baixo acoplamento. Deixa aberto um caminho e um alvo para refatorações. Facilita a compreensão, comunicação entre os desenvolvedores e documentação, uma vez que apresenta um vocabulário comum, facilitando o entendimento das classes do sistema.

O foco principal deste trabalho são os padrões GoF, por isso, no tópico seguinte, veremos um estudo sobre estes padrões.

3.4. Padrões GoF

A '*Gang of Four'* apresentou vinte e três padrões de projeto de software e os dividiu em três categorias, cada uma com um determinado objetivo, são elas: criacional, estrutural e comportamental (GAMMA, HELM, *et al.*, 2000).

Muitos destes padrões são úteis em diversas fases do desenvolvimento e em partes diferentes do código, mas se mostraram muito eficientes em sistemas de múltiplas threads (DEITEL e DEITEL, 2005).

Além dos padrões de projeto, também são apresentados os padrões arquiteturais que auxiliam na definição da arquitetura do sistema e expressam um esquema da organização global da estrutura dessa arquitetura. O sistema é dividido em sub-sistemas pré-definidos e a escolha do padrão arquitetural está associada ao tipo de sistema e os requisitos não funcionais do mesmo (SHALLOWAY e TROTT, 2002). Um padrão arquitetural especifica como subsistemas interagem entre si.

Neste tópico serão discutidas as diferentes categorias de padrões de projeto de software e será descrito os padrões correspondentes a cada categoria.

3.4.1. Padrões de projeto criacionais

Os Padrões de projeto classificados como criacionais, ou de criação, auxiliam questões relacionadas à criação de objetos, como por exemplo, a quantidade de instâncias de um objeto ou o momento exato onde ele deve ser criado, a fim de economizar recursos. Também trata de questões como quais tipos de objetos serão criados ou qual o objeto mais apropriado para realizar uma determinada tarefa (DEITEL e DEITEL, 2005).

Um padrão de criação de classe utiliza o conceito de herança para fazer a variação da classe que será instanciada, já o padrão de criação de objeto repassa a instanciação do objeto para outro objeto. Com o forte uso da herança em sistemas maiores, o uso de padrões criacionais são de grande importância, pois é preferível "Tem-Um" ao invés de "É-Um", deixando assim a estrutura menos acoplada a seu código (GAMMA, HELM, *et al.*, 2000).

Os padrões criacionais, de maneira geral, ocultam sobre quais classes concretas são utilizadas pelo sistema e escondem também de que maneira as instâncias destas classes são criadas e compostas proporcionando maior flexibilidade ao que, como e quando é construído e quem constrói. A estrutura e funcionalidade dos objetos configurados por esses padrões podem variar facilmente. Essa configuração pode ser estática (em tempo de compilação) ou dinâmica (em tempo de execução) (DEITEL e DEITEL, 2005).

São padrões criacionais: *Abstract Factory*, *Builder*, *Factory Method*, *Prototype* e *Singleton*.

3.4.2. Padrões de projeto estruturais

Os padrões de projeto estruturais descrevem formas comuns de organizar classes e os objetos em um software. Eles mostram como classes e objetos serão compostos para desenvolver estruturas maiores desacopladas. Para a criação de interfaces ou implementações os padrões estruturais de classe fazem uso da herança (GAMMA, HELM, *et al.*, 2000).

Os padrões estruturais são largamente utilizados na criação de bibliotecas de classe que são desenvolvidas independente da aplicação e tratam com a estrutura do projeto, facilitando a comunicação entre suas entidades (DEITEL e DEITEL, 2005).

Os padrões estruturais de objetos não compõem interfaces ou implementações, apenas mostram formas de compor objetos para a obtenção de novas funcionalidades. A adaptação desses objetos se tem com sua composição dinâmica, ou seja, composição em tempo de execução (DEITEL e DEITEL, 2005).

São padrões estruturais: *Adapter*, *Bridge*, *Composite*, *Decorator*, *Facade*, *Flyweight* e *Proxy*.

3.4.3. Padrões de projeto comportamentais

Os padrões de projeto comportamentais mostram estratégias para modelar a colaboração entre os objetos e oferecem comportamentos especiais apropriados para diversos tipos de softwares dos mais variados sistemas (GAMMA, HELM, *et al.*, 2000).

Os padrões comportamentais definem como classes e objetos interagem, delegando responsabilidades a outras classes e objetos. Esses padrões desviam o foco do fluxo de controle para que a concentração esteja somente na forma como os objetos são interconectados (DEITEL e DEITEL, 2005). Eles desacoplam a comunicação entre os objetos, fazendo com que seja mais fácil e flexível a manipulação dos objetos sem interferir nas demais classes e objetos comunicantes. Os padrões comportamentais de classes fazem uso da herança para realizar a distribuição do comportamento entre as classes. Já os padrões comportamentais de objeto fazem uso da composição desses objetos (GAMMA, HELM, *et al.*, 2000).

São padrões comportamentais: *Chain of Responsibility*, *Command*, *Interpreter*, *Iterator*, *Mediator*, *Memento*, *Observer*, *State*, *Strategy*, *Template Method* e *Visitor*.

4. Padrões de projeto em Jogos: Estado da Arte

Existe uma grande quantidade de artigos que falam sobre a aplicação de padrões de projeto em jogos, porém, grande parte destes artigos define seus próprios padrões. Entre os artigos que falam sobre a aplicação dos padrões GoF, temos uma quantidade muito limitada de padrões trabalhados. Os artigos contemplam entre um e seis padrões, muitos deles se repetindo entre os artigos, tendo mais de dez padrões que não foram citados em nenhum artigo.

A busca dos trabalhos foi realizada através de uma minuciosa pesquisa no portal de periódicos da CAPES e nos anais de todos os anos de congressos referentes ao tema, como o ICSE (*International Conference on Software Engineerin*), CSEE&T *(Conference on Software Engineering Education and Training*), SEKE (*Conference on Software Engineering and Knowledge Engineering*), CBSoft / SBES (Congresso Brasileiro de Software / Simpósio Brasileiro de Engenharia de Software), SESRes (*Software Engineering and Systems Research*), GDC (*Game Developers Conference*), SBGames (Simpósio Brasileiro de Jogos e Entretenimento Digital), entre outros. Além disso, foram consultadas todas as referências dos artigos encontrados em busca novas referências. Entre as palavras-chaves utilizadas nas buscas, têm-se: "*Design Patterns*", "GoF" e o nome em separado de cada um dos 23 padrões, todos unidos a palavra "*Games*" ou a "*Game Development*".

4.1. Aplicações de padrões de projeto GoF

Em seu trabalho, Ampatzoglou e Chatzigeorgiou (2006) mostram a aplicação e o uso de padrões de projeto em jogos, buscando tornar o código de um jogo mais flexível e reutilizável, baixando custos de manutenção, porém, apesar de fazer citação de onze dos vinte e três padrões GoF, só explica efetivamente quatro (*Strategy*, *Observer*, *State*, *Bridge*). A pesquisa não se resume apenas aos padrões e muda de foco em alguns tópicos.

Já Trindade e Fischer (2008) apresentam os padrões de projeto GoF *Singleton*, *Adapter* e *Observer* e ainda define os padrões *Data Access Object* e *Monitor* onde abordam aspectos estruturais e aplicabilidade de cada padrão, bem como exemplos de implementação, porém, apenas o padrão *Singleton* possui seu exemplo com foco no desenvolvimento de jogos.

Apesar de não falar diretamente sobre a forma de aplicação dos padrões de projeto GoF em jogos, o trabalho desenvolvido por Björk e Holopainen (2005) investiga a correlação entre a aplicação do padrão de projeto e *bugs* encontrados em softwares. Para atingir seu objetivo, foi realizado um estudo empírico sobre jogos desenvolvidos em Java. Esta pesquisa identificou o número de defeitos, a taxa de depuração e os padrões utilizados nos jogos. Os resultados obtidos mostram que o número total de utilização de padrões não está correlacionado diretamente com a depuração. No entanto, alguns padrões de projeto mostram ter um impacto significativo sobre o número de *bugs* reportados. Entre os padrões GoF discutidos estão o *singleton*, *composite*, *adapter*, *observer*, *state*, *strategy*, *template method*, *decorator*, *prototype*, *proxy* e *abstract factory*.

O artigo de Kaae (2001) comenta como os padrões de projeto podem ajudar o programador na fase inicial da programação de jogos, mas não trata diretamente dos padrões GoF, fala apenas sobre como aplicar o padrão arquitetural MVC (*Model-View-Control*). Outro trabalho que mostra a aplicação do MVC em jogos é o trabalho realizado por Wong e Nguyen (2002), porém, apesar do foco de sua pesquisa ser o MVC, ele evidencia o importante uso dos padrões GoF *state*, *strategy* e *visitor* em jogos.

Gestwicki (2007) apresenta um modelo para apoiar a concepção, análise e desenvolvimento de jogos através do uso de padrões de projeto. O modelo consiste em um quadro estrutural para descrever os componentes dos jogos e os padrões de interação que descrevem como os componentes são utilizados pelos jogadores (ou pelo computador). A pesquisa valida o uso de cinco padrões GoF na construção de jogos. São eles: *state*, *facade*, *observer*, *strategy* e *visitor*.

Além destes, alguns artigos que falam do uso de padrões de projeto em jogos são de caráter educacional, ou seja, utilizam jogos para facilitar o aprendizado dos padrões por parte de alunos de graduação. Entre os trabalhos neste sentido, vale destacar o de Silveira e Silva (2006), que mostra como os jogos podem ser utilizados como motivadores no aprendizado de padrões de projeto, sendo seu foco principal os padrões arquiteturais, citando e explicando apenas o padrão *decorator*, mas citando outros sem um estudo de sua aplicação.

Nos trabalhos de Gestwicki e Sun (2007) e Gestwicki e Sun (2008) é apresentado uma abordagem para o ensino de padrões de projeto que enfatiza a orientação a objetos e a integração dos padrões. O contexto de desenvolvimento de jogos de computador é usado para envolver e motivar os alunos, sendo apresentado um estudo de caso com base em EEClone, um jogo de computador no estilo arcade implementado em Java. Estes trabalhos focam nos padrões GoF *singleton*, *facade*, *observer*, *state*, *strategy* e *visitor*.

Segundo Martín, Díaz e Arroyo (2009), o design de software orientado a objeto requer uma combinação de habilidades que não podem ser facilmente transferidas para os

alunos em aulas tradicionais. Seus estudos mostram que podem aumentar a compreensão dos alunos sobre os padrões de projeto através de uma abordagem que consiste no desenvolvimento de uma família de jogos de estratégia de uma forma incremental. No desenvolvimento destes jogos, é evidenciado o uso do padrão arquitetural MVC e dos padrões GoF *observer*, *strategy*, *template method*, *factory method*, *abstract method* e *proxy*.

Ainda nesta linha educacional, Wick (2005) discute a complexidade do aprendizado de padrões de projeto e como essa complexidade pode ser reduzida com o uso de jogos digitais como exemplos práticos da aplicação de padrões. Neste artigo são discutidos os padrões GoF *observer*, *state*, *singleton*, *command* e *visitor*.

A Tabela 1 e a Tabela 2 mostram o resumo das referências aos padrões GoF nos artigos pesquisados. A Tabela 1 mostra essas referências em relação a cada um dos padrões.

Tabela 1: Resumo das referências aos padrões GoF, organizadas por padrão.

Referência	Padrões
Abstract Factory	Ampatzoglou, Gortzis (et al., 2011)
Builder	–
Factory Method	–
Prototype	Ampatzoglou, Gortzis (et al., 2011)
Singleton	Ampatzoglou, Gortzis (et al., 2011) Trindade e Fischer (2008) Gestwicki (2007) Gestwicki e Sun (2008)
Adapter	Ampatzoglou, Gortzis (et al., 2011) Trindade e Fischer (2008)
Bridge	Ampatzoglou e Chatzigeorgiou (2006)
Composite	Ampatzoglou, Gortzis (et al., 2011)
Decorator	Ampatzoglou, Gortzis (et al., 2011) Silveira e Silva (2006)
Facade	Gestwicki (2007) Gestwicki e Sun (2008)
Flyweight	–
Proxy	Ampatzoglou, Gortzis (et al., 2011)
Chain of Responsibility	–
Command	–
Interpreter	–
Iterator	–
Mediator	–
Memento	–
Observer	Trindade e Fischer (2008) Gestwicki (2007) Gestwicki e Sun (2008) Ampatzoglou, Gortzis (et al., 2011) Ampatzoglou e Chatzigeorgiou (2006)

State	Gestwicki (2007) Wong e Nguyen (2002) Gestwicki e Sun (2008) Ampatzoglou, Gortzis (et al., 2011) Ampatzoglou e Chatzigeorgiou (2006)
Strategy	Gestwicki (2007) Wong e Nguyen (2002) Gestwicki e Sun (2008) Ampatzoglou, Gortzis (et al., 2011) Ampatzoglou e Chatzigeorgiou (2006)
Template Method	Ampatzoglou, Gortzis (et al., 2011)
Visitor	Gestwicki (2007) Gestwicki e Sun (2008) Wong e Nguyen (2002)

Tais referências não necessariamente implicam em descrição da aplicação do padrão em jogos, conforme discutido durante o tópico 4.1. A Tabela 2 é similar à tabela 1, porém, foi construída com base nos artigos e mostra o que cada um deles trás sobre o assunto, através dos códigos.

O sinal de mais (+) indica que o artigo apenas cita que o referido padrão pode ser utilizado no desenvolvimento de jogos, sem qualquer explicação da aplicação nesse desenvolvimento. O sinal de sustenido (#) indica que o artigo cita e explica o padrão, mas o foco da explicação não é desenvolvimento de jogos e sim, outros fatores que envolvem jogos de uma maneira geral. O sinal de asterisco (*) indica que o artigo cita e explica detalhadamente o padrão no desenvolvimento de jogos.

Tabela 2: Resumo das referências aos padrões GoF, organizadas por autor.

Artigos	**Abstract Factory**	**Builder**	**Factory Method**	**Prototype**	**Singleton**	**Adapter**	**Bridge**	**Composite**	**Decorator**	**Façade**	**Flyweight**	**Proxy**	**Chain of Responsibility**	**Command**	**Interpreter**	**Iterator**	**Mediator**	**Memento**	**Observer**	**State**	**Strategy**	**Template Method**	**Visitor**
Ampatzoglou e Chatzigeorgiou (2006)							*												*	*	*		
Ampatzoglou, Gortzis (et al., 2011)	#			#	#	#		#	#			#							#	#	#	#	
Gestwicki (2007)										+									+	+	+		+
Gestwicki e Sun (2008)					#					#									#	#	#		#
Silveira e Silva (2006)									#														
Trindade e Fischer (2008)					#	#													+				
Wong e Nguyen (2002)																				+	+		

4.2. Outros trabalhos relacionados

Mesmo sem falar nos padrões de projeto GoF, alguns artigos discutem a importância do uso de padrões de projeto em jogos, definindo seus próprios padrões, baseados em experimentos, porém, os padrões definidos em boa parte destes trabalhos não são verdadeiramente padrões de projeto conforme a definição constante no tópico 3.1. Esses padrões são, na verdade, funcionalidades, cujos autores demonstram a importância de sua presença nos jogos e caracterizam essas funções em forma de padrão, com o objetivo de mostrar que essas funções devem estar presentes nos jogos.

Dos trabalhos apresentados, somente Dahlskog e Togelius (2012), Larsen e Aarseth (2006) e Kirk (2005) podem ser realmente enquadrados no conceito de padrões de projeto definido por Gamma (*et al.*, 2000), apresentado no tópico 3.1. Os demais podem ser enquadrados como "padrões de funcionalidade", por se tratarem mais de requisitos de jogos que do projeto de sua implementação de fato. Os padrões de projeto, definidos por estes autores, não possuem relações diretas com os padrões GoF, ou seja, não são padrões GoF “disfarçados” ou “evoluídos”, e sim, novos padrões que poderão ajudar na implementação de tipos específicos de jogos. A única exceção será apresentada no texto. Apesar deste fato, os padrões GoF também poderão ajudar na implementação de alguns destes padrões, como será apresentado ainda neste tópico.

Para definir um padrão, Mcgee (2007) descreve materiais e métodos para ajudar os desenvolvedores a identificar e formular padrões de projeto baseado nas “boas práticas” de programação. Também sugere como utilizar os padrões gerados como parte na criação de jogos inovadores.

Com o mesmo objetivo de ajudar os desenvolvedores a criar seus padrões de projeto, Loh e Soon (2006) validam uma abordagem que compara dois jogos para identificar quais elementos seriam úteis na conversão de um para outro. Sendo um deles um jogo de tabuleiro tradicional (como exemplo deste, usaram o jogo Risk) e o outro um jogo eletrônico de mesmo tipo (o jogo usado foi o Warcraft III). Os elementos identificados podem virar padrões de projeto e serem utilizados para melhorar a programação de jogos eletrônicos.

Na linha de definição de padrões, Davidsson, Peitz e Björk (2004) propõem uma série de padrões de projeto em jogos voltados para dispositivos móveis. Entre eles, o “*chat forum*”, permitindo jogadores conversarem entre si; “*Coupled Games*”, propondo um acoplamento maior entre jogos, “*Social Rewards*”, que sugere uma premiação para os jogadores com mais amigos; “*Configure Gameplay Area*”, “*Physical Navigation*”, “*Player Physical Prowess*”, “*Player-Location Proximity*”, “*Artifact-Location*” e “*Late Arriving Players*”, onde o programador deve definir estratégias para a entrada de um novo jogador durante a progressão de outros jogadores.

Em seu artigo, Hullett e Whitehead (2010) listam, com exemplos práticos em jogos de primeira pessoa, seus padrões de projeto voltados para auxiliar a jogabilidade, geometria dos objetos, posicionamento de itens na tela e a inteligência artificial. Estes padrões permitem que os designers criem níveis mais interessantes e variados. Entre os padrões têm-se: "*Sniper Location*", que mostra posições onde podem ficar os inimigos atiradores; "*Gallery*", "*Choke Point*", comentando sobre pontos onde a ação é mais ofensiva; "*Arena*", "*Stronghold*", "*Turret*", "*Vehicle Section*", "*Split Level*", "*Flanking Route*" e "*Hidden Arena*", que trata de caminhos secretos e alternativos no jogo.

Homer e Plass (2009) definem alguns padrões de alto nível de aprendizado, como o "construir coisas é divertido e ajuda a aprender", "jogos podem ser envolventes sem visuais deslumbrantes", "diferentes níveis de incentivos aumentam a diversão e o engajamento", entre outros. Estes padrões são mais filosóficos e de alto nível, não fazendo qualquer referência a padrões de projeto de software.

A publicação de Bergström, Björk e Lundgren (2010) explora discussões de projetos em jogos visando melhorar o relacionamento social entre os jogadores em ambiente multiplayer. Os estudos sugerem uma estética através da dinâmica do jogo e os ideais estéticos na jogabilidade que o design deve atingir. Para facilitar a tarefa, padrões são propostos, entre eles, o "*Team Strategy Identification*", mostrando como a equipe de jogadores deve se identificar uns com os outros; "*Team Accomplishments"*, sugerindo como a equipe deve trabalhar em conjunto; "*Guilting*", "*Mutual FUBAR Enjoyment*", "*Spectacular Failure Enjoyment*" e "*Mutual Experiences*" onde são definidos formas dos jogadores trocar suas experiências durante o jogo, melhorando assim a experiência individual de cada jogador.

O trabalho de Beznosyk (et al., 2012) define padrões para desenvolvimento de jogos voltados para a usabilidade. Entre eles, o padrão "*Limited resources*" sugere o quanto os recursos devem ser limitados em um jogo; "*Complementary*"; "*Interaction with the same object"*, que mostra como identificar as formas de interação com objetos do jogo; "*Shared puzzles*"; "*Abilities that can be used on other players*" e "*Shared goals*", definindo como os jogadores podem compartilhar objetos. Estes padrões podem ser bastante úteis nos jogos cooperativos em ambiente remoto.

Segundo Dahlskog e Togelius (2012), a geração de conteúdo e os padrões de projeto podem ser combinados de várias maneiras diferentes no desenvolvimento de um jogo. O trabalho discute como combinar estes dois elementos, usando, como exemplo, o projeto de um jogo de plataforma (Super Mario Bros). O documento propõe diversos padrões como *"Enemy", "2-Horde", "3-Horde", "4-Horde", "Roof", "Gaps", "Multiple gaps", "Variable gaps", "Gap enemy", "Pillar gap", "Valley", "Pipe valley", "Empty valley", "Enemy valley", "Roof"*, porém, apenas alguns deles são realmente descritos. Ao descrever um padrão, Dahlskog e Togelius (2012) apresentam cinco itens: o nome do padrão, o problema que o padrão tenta resolver, a solução proposta pelo padrão, o uso do padrão e comentários sobre a aplicação do mesmo. Os padrões definidos tratam

principalmente da jogabilidade do jogo. Entre os cinco padrões descritos por Dahlskog e Togelius (2012), têm-se:

- Nome do Padrão: 4-*Horde*

- Problema: O jogador pode passar pela fase sem risco. O jogador poderá efetuar um salto muito grande (distante), de maneira que irá saltar sobre vários inimigos sem precisar matá-los ou perder pontos de vidas com eles.
- Solução do padrão: Ao colocar quatro inimigos (4-*Horde*) em uma formação de fila, o comprimento máximo do salto não pode ser suficiente para passar sobre os inimigos, o que obriga o jogador a pousar em qualquer um dos inimigos e realizar um segundo salto sobre os inimigos restantes.
- Uso do padrão: o uso deste padrão é adequado para jogos de plataforma longos. O padrão pode também ser usado em conjunto com os padrões *valley* para limitar a área de destino do jogador. O padrão pode ser utilizado para forçar o jogador a executar ações em uma plataforma elevada permitindo que caia sobre inimigos.
- Comentários: Nem todos os tipos de inimigos são adequados para este padrão. Por exemplo, os inimigos que um personagem não consegue pular sobre ele pode causar fases intransponíveis. Elementos para aumentar o poder do salto devem ser considerados quando se aplica esse padrão. Este padrão não deve ser interpretado como dois padrões 2-*Horde*, a distância entre os inimigos na formação é fundamental.

- Nome do padrão: *Pillar Gap*

- Problema: o jogador pode saltar e atravessar obstáculos verticais sem qualquer dificuldade. Um salto muito alto fará o jogador saltar sobre obstáculos elevados.
- Solução do padrão: Ao colocar uma série de pilares com uma determinada largura um jogador menos habilidoso pode não conseguir ultrapassar em um salto e cair no fosso. O uso de alturas variáveis entre os pilares que o jogador precisa transpor deve variar de acordo com o comprimento do salto.
- Uso do padrão: esse padrão pode ser utilizado em qualquer jogo de plataforma e, no caso do jogo Super Mario Bros, é utilizado perto do final da fase de modo que Mario possa ser suficientemente elevado para alcançar o topo da bandeira no final de um nível.
- Comentários: Tentativas não poderão salvar o jogador, mas um jogador hábil poderá saltar do ponto certo e cair perfeitamente no local desejado.

- Nome do padrão: *Risk and Reward*

- Problema: A disposição das fases é praticamente linear e a escolha do jogador é limitada.
- Solução do padrão: múltiplos caminhos são oferecidos e podem ser escolhidos como recompensas ou com intuito de desviar de um grupo de inimigos. O

jogador pode escolher o caminho especificado de acordo com a sua vontade e sua habilidade.

- Uso do padrão: o uso principal é a criação das possibilidades do jogador seguir em uma fase. O padrão pode também ser usado para introduzir uma recompensa quando o jogador consegue alguma façanha ou terminar o jogo em um nível de dificuldade específico.
- Comentários: essas possibilidades incluem no jogo a tomada de decisão por parte do jogador que pode fazer esse jogador reiniciar uma partida do início, após um término total do jogo, para que as demais decisões possam também ser tomadas e vivenciadas na jogatina.

➢ Nome do padrão: *Enemy valley*

- Problema: O jogador pode saltar e atravessar obstáculos verticais sem risco. O jogador pode atravessar o nível lentamente sem risco.
- Solução do padrão: colocar um inimigo entre dois obstáculos verticais obriga o jogador a enfrentar o inimigo. Se o jogador pula sobre o inimigo e depois salta sobre o obstáculo, o jogador corre o risco de perder uma vida devido à alta velocidade da manobra.
- Uso do padrão: o padrão pode ser utilizado com a maioria dos inimigos.
- Comentários: o padrão faz-se necessário um comprimento bem calculado entre os obstáculos verticais.

➢ Nome do padrão: *Stair up*

- Problema: o jogador precisa estar em uma altura diferente do nível do solo para saltar, pois a altura de seu pulo não é suficiente para transpor o obstáculo.
- Solução do padrão: ao fornecer plataformas ou blocos cuja altura aumenta gradativamente, o jogador pode saltar sobre eles até chegar a uma posição mais elevada que permita transpor o obstáculo.
- Uso do padrão: o padrão é utilizado em qualquer fase onde o jogador precisa saltar um obstáculo, mas não consegue devido a suas limitações na capacidade de salto.
- Comentários: a plataforma de salto não deve ser colocada a uma altura que impeça o alcance do jogador, a não ser que seja a pretensão ao criar tal plataforma.

O trabalho proposto por Larsen e Aarseth (2006) surgiu após uma análise do tempo de desenvolvimento de um jogo e da complexidade de sua programação. O artigo propõe seis padrões para construção de jogos que reduzem o tempo e a complexidade na criação dos mesmos. São eles: *"Multiple Paths"*, *"Local Fights"*, *"Collision Points"*, *"Reference Points"*, *"Defense Points"* e *"Risk Incentive"*. Na descrição de cada um destes padrões, o autor apresenta seis itens: o nome do padrão, a ideia central do padrão, uma descrição geral, como o padrão pode ser útil, consequências do uso do padrão e um

conjunto de padrões relacionados ou interconectados a ele. A descrição dos padrões de Larsen e Aarseth (2006) é mostrada a seguir.

- Nome do padrão: *Multiple Path*
- Ideia central: Cada caminho de um jogo tem de ser completado por um ou mais caminhos, a fim de evitar os estrangulamentos.
- Descrição geral: Em jogos *multiplayer*, é fundamental que todas as principais rotas e objetivos centrais tenham alternativas, ou seja, rotas e objetivos secundários. Se este padrão não é utilizado, os objetivos centrais tornam-se monótonos e a jogabilidade dinâmica de batalhas entre as equipes são paralisadas pelos gargalos da não utilização deste padrão.
- Utilidade: Na fase dos campos elétricos do jogo *Unreal Tournament* 2004, o uso do padrão é muito visível. Em qualquer lugar da fase, o jogador tem opções a escolher.
- Consequências: o *game design* evita gargalos que podem bloquear a jogabilidade e resultar em jogos nos quais não existam vencedores. É importante lembrar que o uso excessivo do padrão, por outro lado, pode resultar em muitos caminhos, onde os jogadores nunca se encontram, pois estão perdidos entre tantas possibilidades.
- Padrões relacionados: existe uma estreita relação com o padrão *Collision Points*. O *Collision Points* pode ser considerado um sub-padrão do padrão *Multiple Path*.

- Nome do padrão: *Local Fights*
- Ideia central: quebra-se a fase em fases menores, isoladas do resto da fase.
- Descrição geral: grandes fases são divididas em partes menores que podem ser jogadas independentemente das demais partes. Grandes áreas abertas podem complicar a jogabilidade. Deve haver lugares onde o jogador possa se esconder e deixar eventuais perseguidores desorientados. O design da fase força os jogadores a se envolverem em batalhas, sem saber a real posição do inimigo.
- Utilidade: ao se criar um corredor, deve-se quebrar o corredor em partes. O risco de atravessar um longo corredor ou área aberta pode parecer muito alto para a maioria dos jogadores. Quebrando a área em áreas menores, dá aos jogadores a sensação de ter apenas que cruzar pequenas áreas. Este padrão de projeto também se torna viável para projetos de grandes áreas livres. Deve-se fazer as áreas ao ar livre em terreno montanhoso, acrescentando colinas, montanhas, lagos e florestas permitindo assim a criação de pequenas sub-áreas.
- Consequências: se grandes áreas estão incluídas em uma fase sem divisões (sem utilizar o padrão de projeto), as áreas podem se tornar um paraíso para bons atiradores de longa distância, tornando a área impossível de ser atravessada pelos demais jogadores.

- Padrões relacionados: este padrão tem uma relação com o padrão *Reference Points.* Os elementos utilizados para dividir o nível podem servir para um duplo propósito, ou seja, além de divisória, também pode ser um ponto de referência.

O padrão *Local Fights* é extremamente semelhante ao padrão de projeto GoF *Proxy*. Pode-se dizer que o padrão *Proxy* é uma generalização do padrão *Local Fights*, sendo esta a única exceção onde se pode afirmar que um padrão definido é semelhante a um padrão GoF.

➢ Nome do padrão: *Collision Points*

- Ideia central: os caminhos dos jogadores adversários devem cruzar, em alguns pontos, o caminho dos jogadores aliados, com o intuito de criar uma tensão no nível.
- Descrição geral: em jogos *multiplayer*, um dos principais elementos é o contato entre os jogadores. Os caminhos de cada equipe, que levam da área de uma equipe para outra ou para um objetivo importante no nível, têm que se cruzarem para que os membros de ambas as equipes possam se enfrentar. Esses cruzamentos são chamados de Pontos de Colisão.
- Utilidade: se a fase criada requer a captura de um elemento em comum, deve-se construir o nível de tal forma que todos os jogadores passem por uma área central. Ao fazer isso, fica garantido que os jogadores convergirão para um determinado ponto em comum.
- Consequências: se o mapa contém apenas um ponto de colisão, é imperativo que o tempo para se alcançar este ponto seja o mesmo em ambas as equipes. Posicionar o ponto de colisão muito perto de uma equipe desequilibra o jogo, deixando-o impossível para as demais equipes.
- Padrões relacionados: existe uma relação entre este padrão e o padrão *Local Fights.* Usando o padrão *Collision Points*, fica garantido que pelo menos um local de luta estará presente no jogo.

➢ Nome do padrão: *Reference Points*

- Ideia central: Sempre se devem fornecer pontos de referência em uma determinada fase para auxiliar a navegação do jogador.
- Descrição geral: o jogador nunca deve sentir-se perdido. O objetivo de um nível *multiplayer* é produzir uma experiência de jogo contínuo para todos os jogadores. Os jogadores devem ser desafiados pelos jogadores adversários e não pela dificuldade de se atravessar um cenário. Os pontos de referência não precisam ser grandes edifícios ou monumentos, mas pode ser algo simples, como uma tubulação ou uma sequência de postes elétricos.
- Utilidade: é interessante aplicar elementos arquitetônicos em uma fase. Deve-se utilizar o poder de pontos de referência para uma boa navegação. Ao usar o padrão também é imperativo que os pontos de referência sejam únicos, pois pontos semelhantes podem gerar confusão entre os jogadores.

- Consequências: o uso excessivo do padrão pode acarretar em uma fase muito trivial e óbvia, portanto, esses pontos de referência devem ser os mais sutis possíveis.
- Padrões relacionados: este padrão tem uma pequena relação com o padrão *Multiple Path.* Tão importante quanto proporcionar pontos de referência é proporcionar mais caminhos para os jogadores, caminhos estes bem distintos dos demais, onde estas diferenças já são consideradas um ponto de referência.

Os padrões *Collision Points* e *Multiple Path* podem ser facilmente alcançados com o uso do padrão de projeto GoF *Abstract Factory*, pois este pode forçar a criação de um número inicial de caminhos e um número mínimo de pontos de colisão entre estes caminhos e gerar erro, caso o programador não instancie a quantidade mínima de caminhos definida pela classe abstrata.

- Nome do padrão: *Defense Points*
- Ideia central: os jogadores ou equipes que defendem objetos devem ser, de certa forma, beneficiados pelo layout arquitetônico do nível.
- Descrição geral: a maioria das fases em jogos de ação *multiplayer* gira em torno de uma equipe atacar o local de outra equipe ou tomar o controle de pontos específicos. Em ambos os casos, os membros das equipes frequentemente precisam defender essas áreas ou objetivos. Os defensores não sabem quando ou de onde os atacantes podem vir, o que é uma desvantagem no jogo. Para compensar esta desvantagem, alguns elementos podem ser incluídos no cenário como forma de compensação, como por exemplo, trincheiras, dunas naturais, entre outros.
- Utilidade: criar áreas ao redor do objetivo principal do nível pode ajudar os defensores em seu objetivo.
- Consequências: se a área de defesa se torna muito poderosa, pode-se atingir um nível onde o atacante não tem nenhuma forma de ultrapassar a área de defesa. Portanto, este é um padrão que deve ser utilizado com moderação. Uma boa maneira de utilizá-lo é combiná-lo com o padrão *Multiple Path,* fazendo as defesas naturais abrangerem apenas a entrada principal do objetivo. Caminhos alternativos devem ser menos fortificados.
- Padrões relacionados: pode haver uma relação entre este padrão e o padrão *Reference Points.* Tanto os atacantes quanto os defensores, usando a área de defesa, por isso, é importante que ela seja facilmente reconhecida, facilitando assim a comunicação sobre os eventos que ocorrem nessas áreas.

- Nome do padrão: *Risk Incentive*
- Ideia central: o acesso aos objetos desejados, que podem ser encontrados em um nível, deve estar relacionado com os elementos de risco daquele nível.
- Descrição geral: armas de alta potência, pacotes de saúde ou atalhos são muito comuns em fases de jogos *multiplayer*, mas, a fim de manter o equilíbrio da fase,

deve haver alguns riscos relacionados a obtenção destes objetos ou caminhos. O tamanho do risco deve ser proporcional ao benefício de se ter aquele objeto no jogo.

- Utilidade: se existe uma rota grande que vai de A para B, deve-se colocar um atalho em um lugar bem visível, de modo a tornar claro, para todos os jogadores, quando os demais jogadores estão tentando tirar proveito deste atalho ou colocar armadilhas antes de uma arma poderosa, fazendo jogadores perderem pontos de vida ao falhar na tentativa de obter tal arma.
- Consequências: usando este padrão, o jogo dá escolhas aos jogadores. Eles vão ter que, individualmente, decidir se vale ou não a pena o risco de tentar atravessar um atalho ou obter uma arma poderosa.
- Padrões relacionados: Este padrão é fracamente relacionado com o padrão *Multiple Path*. Se os vários caminhos em um nível proporcionar alguma forma de atalho, o padrão *Risk Incentive* também deve ser aplicado. Outra relação existe com o padrão *Defense Points*, pois o uso de elementos de defesa pode acarretar algum risco, como por exemplo, levar tempo para entrar e sair de uma área anti-aérea ou limitar o campo visual de uma determinada área.

Os padrões propostos por Kirk (2005), definem elementos de design de alto nível, focados basicamente em jogos de RPG. O trabalho apresenta um conjunto de diversos padrões os quais os descreve com os itens: nome do padrão, justificativa da existência do padrão, nomes sinônimos ao padrão, padrões relacionados, descrição do problema abordado pelo padrão, critérios de quando usar o padrão, consequências do uso do padrão, problemas que podem ocorrer com o uso do padrão, exemplos e uso conhecido do padrão. Os padrões propostos são o *"Alignment", "Anonymous Rule", "Attendance Reward", "Attribute", "Class", "Class Tree", "Conflicted Gauge", "Contest Tree", "Currency", "Endgame", "Failure Reward", "Game Master", "Gauge", "Generalized Contest", "Gift", "Hit Points", "Idiom", "Last Man Standing", "Level", "Loose Coupling", "Modularity", "Narrative Reward", "Negotiated Contest", "Point Spend Attributes", "Priority Grid, "Random Attribute", "Rank", "Resource", "Safety Valve", "Skill", "Skill Tree", "Structured Story", "Success Reward", "Template", "Trait", "Trauma Gauge", "Wound Trait"*. Segue a descrição de alguns deles.

- Nome do padrão: *Alignment*
- Justificativa: Fornecer orientações sobre como um jogador deve gerir o caráter de seu personagem.
- Sinônimos: não aplicável;
- Padrões relacionados: *Attribute, Idiom;*
- Problema abordado: um alinhamento é uma característica comum, que especifica como um jogador deve retratar um personagem ao determinar suas ações. Alinhamentos são geralmente especificados com palavras em vez de números, os mais comuns são: "Bom" e "Mau". Para ampliar o campo de comportamentos

existe uma gama maior de possibilidades. Muitos jogos especificam um grande número de alinhamentos. Além de "Bom" e "Mau", um jogo pode permitir a um jogador escolher entre "legal" e "ilegal" ou "social" e "anti-social", entre outros;

- Quando usar: quando um jogo dá orientações aos jogadores sobre a maneira em que eles devem retratar seus personagens, o padrão de alinhamento faz um trabalho simplório. Outros padrões, tais como o padrão Idiom, têm sido desenvolvidos nos jogos modernos que satisfazem este objetivo de maneira mais eficiente. É altamente recomendado entender o padrão *Idiom* antes de decidir usar o padrão *Alignment*;
- Consequências: embora o padrão *Alignment* prometa promover uma melhor interação dos jogadores, ele não faz isso. Alguns jogos vão oferecer recompensas para que o jogador exerça corretamente seu alinhamento, mas esses sistemas de recompensas são, muitas vezes, falhos. Por exemplo, alguns jogos promovem recompensas exatamente iguais as que um jogador vai ganhar para a realização de outras atividades, fazendo com que o jogador não siga o alinhamento inicialmente escolhido.
- Problemas com seu uso: se o padrão *Alignment* for utilizado, deve ser feito com bastante atenção. Um dos maiores problemas com o padrão é que os jogadores do tipo "mal" tendem a não seguir seu alinhamento com muito mais frequência do que os jogadores do tipo “bom”. A razão é simples, os jogadores precisam fazer algo diabólico para seus colegas jogadores a fim de retratar adequadamente seus alinhamentos “mal”, o problema é que a maioria das pessoas não considera isso divertido e simplesmente não o fazem.
- Exemplo: Dungeons & Dragons v.3.5, RIFTS, Final Fantasy.

➢ Nome do padrão: *Anonymous Rule*

- Justificativa: ocultar a complexidade da jogabilidade, incorporando regras de pouca importância em meio às regras mais importantes.
- Sinônimos: Não aplicável;
- Padrões relacionados: *Modularity*;
- Problema abordado: Uma regra anônima é uma regra inclusa na descrição de outra regra ou entidade do jogo. Tais regras não têm nomes próprios, e por isso são chamadas de "anônimas". Estas regras não descrevem o processo pelo qual duas ou mais regras são simplificadas, através da criação de uma nova regra. Pelo contrário, elas descrevem a adoção de uma regra para a descrição de regras mais complexas. Os *game designers* querem regras simples e que permitam cobrir adequadamente o contexto do jogo, porém simplicidade e máxima cobertura dificilmente caminham juntas. Para manter um número mínimo de regras alguns escritores adicionam peculiaridades e características especiais para regras mais significativas. Frequentemente, essas "peculiaridades" poderiam ser divididas como regras separadas, mas não são porque o *game designer* quer buscar o máximo de simplicidade possível.

- Quando usar: usar regras anônimas não é uma boa ideia do ponto de vista do *game design*, porém, do ponto de vista da equipe do marketing é muito interessante. Se o game designer estiver tentando criar um livro-jogo de RPG com uma forte ênfase na estética visual, regras anônimas podem ajudar pelo fato delas reduzirem o número de páginas e fazer o texto fluir de forma mais suave no layout da tela.
- Consequências: o uso exagerado de regras anônimas pode gerar mais complexidade do que simplicidade. No ponto de vista do *game designer* pode ter ficado simples, mas as regras anônimas, criadas para esconder detalhes, podem camuflar a complexidade deixando o jogo muito complicado para os jogadores menos experientes.
- Problemas com seu uso: quando regras anônimas são inclusas em um jogo, deve-se criar mecanismos para que os jogadores encontrem essas regras de maneira fácil quando necessário. Por exemplo, um jogo usando o padrão *Anonymous Rule* para raças, poderia descrever uma raça de Elfos de maneira mais simples e criar descrições anônimas que explicam ou expandem o conceito da raça.
- Exemplo: Dungeons & Dragons v.3.5, HARP, Nobilis.

➢ Nome do padrão: *Attendance Reward*

- Justificativa: fornecer aos jogadores incentivos e recompensas por encontrar novas sessões (desafios) no jogo.
- Sinônimos: *E.P., X.P., EXP, Character Points, Development Points*
- Padrões relacionados: *Failure Reward, Idiom, Narrative Reward, Success Reward*
- Problema abordado: o uso do padrão *Attendance Reward* é uma forma de recompensar os jogadores por encontrar sessões no jogo. A ideia é incentivar a participação do jogador e garantir que todos usem o máximo de sessões do jogo. Um jogo que usa o padrão *Attendance Reward* segue os seguintes passos: 1) Ele dá ao jogador uma recompensa a cada sessão; 2) A recompensa consiste exclusivamente em executar algo no jogo, ou seja, não é algo físico, tal como dinheiro ou comida; 3) A recompensa não é algo que deve desgastar o personagem para ser usufruída. Se o jogador deve se arriscar então não é realmente uma recompensa, mas sim um desafio obrigatório de algum recurso vital e necessário para o fluxo de jogo.
- Quando usar: este padrão deve ser utilizado quando o jogo roda em mais de uma sessão e quando se quer manter um grupo consistente de jogadores atravessando estas sessões.
- Consequências: o padrão *Attendance Reward* encoraja os jogadores a participar de uma série de sessões de jogo, no entanto, mudar de sessão sem um motivo aparente pode fazer os jogadores deixarem de jogar. Essas recompensas geralmente são de longo prazo e muitos jogadores não gostam de passar muito tempo em um jogo. A teoria afirma que o fornecimento de recompensas "boas"

ou "apropriadas" transforma as atividades que começam por "diversão" em "trabalho" e isto é tudo que os jogadores não querem.

- Problemas com seu uso: o padrão *Attendance Reward* recompensa o jogador que passa muito tempo jogando. Na implementação deste padrão, deve-se ter muito cuidado com contradições entre as sessões. Por exemplo, se um jogo fornecer aos jogadores recompensa em formato de "pontos de experiência" e esses pontos também puderem ser obtidos por outros meios no jogo, como matando monstros, os jogadores imediatistas vão adotar temporariamente os personagens e buscar esses pontos apenas matando monstros, diminuindo significativamente o incentivo da recompensa entre as sessões. A melhor opção neste caso seria ter recompensas diferentes entre aquelas ganhas no jogo daquela ganhas ao participar de novas sessões.
- Exemplo: *Hero System 5th Edition, Nobilis e The World of Darkness*

CONCLUSÃO DO CAPÍTULO

Como foi mostrado no capítulo, o estudo sobre a aplicação dos padrões GoF em jogos é limitado a apenas alguns padrões (vide Tabela 1) e desses, alguns não apresentam o seu uso geral de maneira clara e objetiva.

Quando os autores fazem referência a padrões de projeto em jogos, muitos não estão falando do GoF e a maioria trata funcionalidades como padrões, o que reduz a quantidade de trabalhos relacionados a esse tema.

Esse capítulo mostra uma carência de pesquisa sobre a utilização dos padrões GoF em jogos, o que deixa uma lacuna. Estes padrões são aceitos como facilitadores na programação de aplicações comerciais. Eles atribuem vantagens, principalmente agilidade na programação de novos sistemas com o reuso de código e a redução de *bugs*, já que problemas são resolvidos com soluções testadas a aprovadas em outros projetos (DEITEL e DEITEL, 2005).

5. Proposta do uso de padrões de projeto GoF aplicados ao desenvolvimento de jogos

Este capítulo mostra detalhadamente cada um dos 23 padrões GoF e situações em que podem ser aplicados no desenvolvimento de jogos digitais. Cada um dos padrões tem seu propósito e sua motivação, mas o objetivo final é sempre resolver problemas na programação, facilitando assim o desenvolvimento e possíveis manutenções.

5.1. Abstract Factory

O padrão *Abstract Factory* é um padrão de criação que sugere uma interface para criar famílias de objetos relacionados ou dependentes entre si, sem especificar suas classes concretas (GAMMA, HELM, *et al.*, 2000).

O padrão *Abstract Factory* é citado apenas no artigo Ampatzoglou, Gortzis (et al., 2011) que se limita apenas em constatar a quantidade de *bugs* reportados sem e com o uso deste padrão no desenvolvimento de jogos, não mostrando formas e situações de aplicação do mesmo nesses jogos. Neste trabalho é apresentado um exemplo de onde e como o padrão pode ser aplicado no desenvolvimento de jogos.

5.1.1. Onde se aplica:

Em diferentes tipos de jogos, mas frequentemente em jogos de RPG e de Tiro em primeira pessoa, faz necessária a criação de diversos cenários, ambientes, labirintos, castelos e tudo mais onde o personagem terá que atravessar para atingir seus objetivos. Se não for bem planejada, a criação destes locais se tornará muito custosa e a duplicação de códigos será inevitável.

Um exemplo disso pode ser visto na Figura 5-1, onde podemos ver diversos cenários diferentes, mas com características de programação semelhantes, a Figura 5-1(a) mostra um cenário onde o personagem caminha por um espaço aberto, a Figura 5-1(b) mostra uma situação semelhante, onde o personagem está em uma construção de concreto e a Figura 5-1(c) mostra uma caverna natural fechada. Apesar de visualmente diferentes, os códigos internos são bem parecidos (PERUCIA, BERTHÊM, *et al.*, 2005) e poderiam ser simplificados com a adoção de um padrão de projeto.

Figura 5-1: Jogos em primeira pessoa utilizando diferentes ambientes representados. (a) Espaço Aberto (b) Construção Fechada (c) Caverna Natural

5.1.2. Solução proposta pelo padrão:

Uma forma de facilitar esta criação deste tipo de cenário é o uso do padrão *Abstract Factory*, com ele, todos os métodos de geração de cenários saem da classe principal do jogo e entram na classe abstrata que irá fabricar os cenários. O padrão também propõe uma classe genérica para a geração de cenários, que será uma classe concreta da fábrica abstrata. Na Figura 5-2, sem o uso do padrão, observamos forte acoplamento entre os itens de cenário e o cenário como um todo. Muitos dos elementos com características comuns são refeitos e possíveis modificações e ajustes em itens impactam diretamente o cenário como um todo. Na Figura 5-3, já com o uso do padrão, vemos um fraco acoplamento entre os itens, neste caso, a porta de um cenário, que pode ter seus parâmetros modificados sem a necessidade de qualquer modificação nas classes que geram os cenários.

5.1.3. Diagramas:

→ Sem o uso do padrão

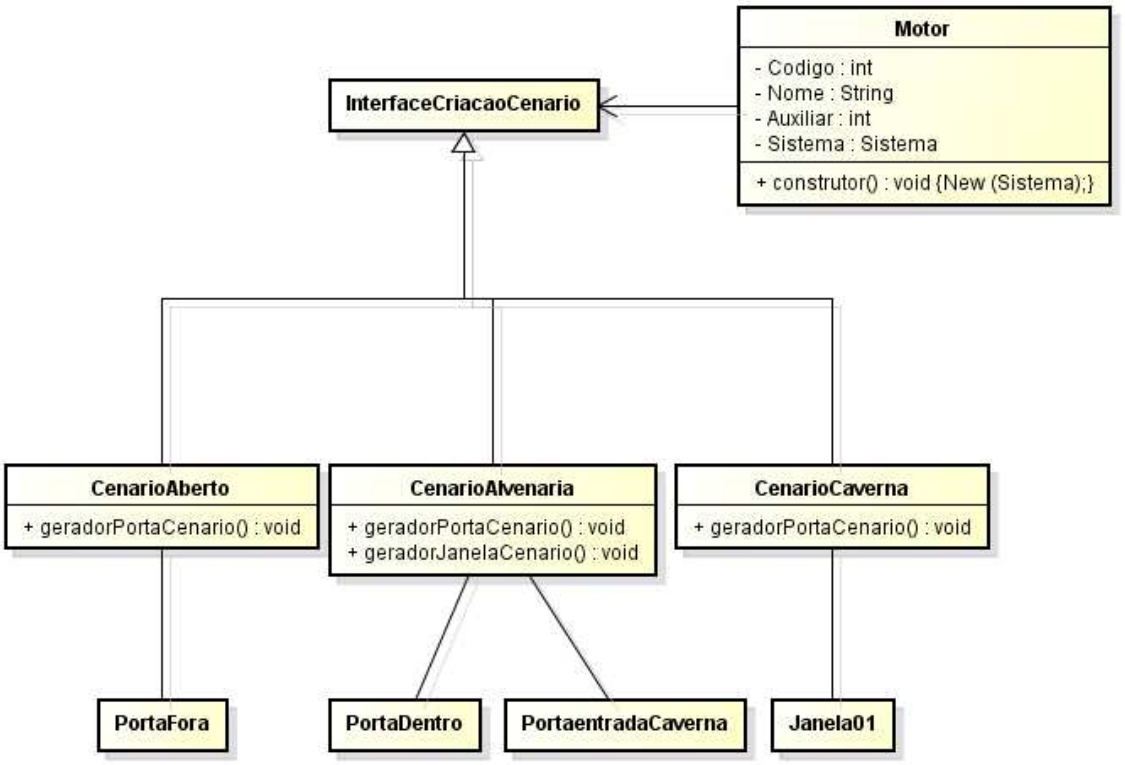

Figura 5-2: Diagrama parcial da geração de cenários sem o uso do Padrão *Abstracy Factory*. Nesta situação percebe-se o forte acoplamento entre os itens do cenário e o próprio cenário.

→ Com o uso do padrão

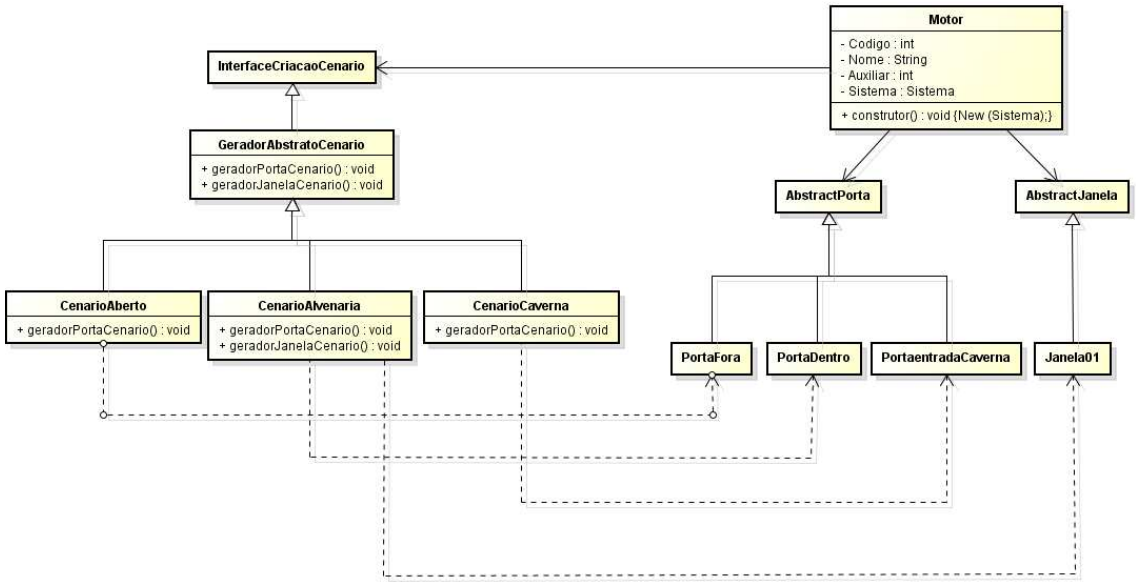

Figura 5-3: Diagrama parcial da geração de cenários com o uso do Padrão *Abstracy Factory*

5.2. Builder

O padrão *Builder* é um padrão de criação que separa a construção de objeto complexo de sua representação, com o objetivo de criar representações diferentes da mesma classe (GAMMA, HELM, *et al.*, 2000).

O padrão *Builder* não possui qualquer referência direta sobre seu uso no desenvolvimento de jogos.

5.2.1. Onde se aplica:

Em diversos jogos, muitas críticas são feitas em relação aos inimigos, por eles serem exatamente iguais ou serem apenas ligeiras variações dos inimigos de fases anteriores. Um exemplo disso pode-se ver no jogo "*Street of Rage*", conhecido popularmente como "Briga de Rua", mostrado na Figura 5-4. Os inimigos *Signal* e *Galsia* aparecem na imagem de maneira muito parecida ou exatamente igual. Esta semelhança se repete em todas as fases do jogo, tendo ligeiras variações na cor, no nome e na força do personagem, tendo poucos inimigos realmente diferentes. O diagrama de classes de um jogo criado para gerar inimigos, sem uso de padrões, pode ser visto na Figura 5-5, onde vemos uma classe "inimigo", identificada como "pai" e várias classes de inimigos, que herdam desse "pai" e alteram alguma característica. A cada nova característica as ser alterada, uma nova classe deve ser criada, ocupando assim mais espaço em memória.

A criação de uma grande diversidade de inimigos é algo custoso e demorado nos jogos, além de ocupar memória e espaço em disco. (PERUCIA, BERTHÊM, *et al.*, 2005).

5.2.2. Solução proposta pelo padrão:

Uma forma de otimizar a criação de inimigos variados, reduzindo o consumo de recursos da máquina é o uso do padrão *Builder*. O padrão propõe a criação de um inimigo complexo, com diversos itens, movimentos, armamentos e roupagens bem diferentes, vários tipos de socos e chutes, além de estilos diferentes de luta. A classe *builder* concreta criará vários personagens, partindo deste inimigo "mestre", separando algumas dessas características e elementos para compor diversos personagens distintos. O diagrama com um exemplo de aplicação do padrão *Builder*, pode ser visto na Figura 5-6.

Figura 5-4: Jogo "*Street of Rage*". Nesta tela é possível notar a semelhança entre os inimigos do jogo. O inimigo *Signal* aparece replicado de jaqueta laranja e de jaqueta amarela e o *Galsia* aparece três vezes, sendo duas de roupagem azul e outra com roupa verde.

5.2.3. Diagramas:

→ Sem o uso do padrão

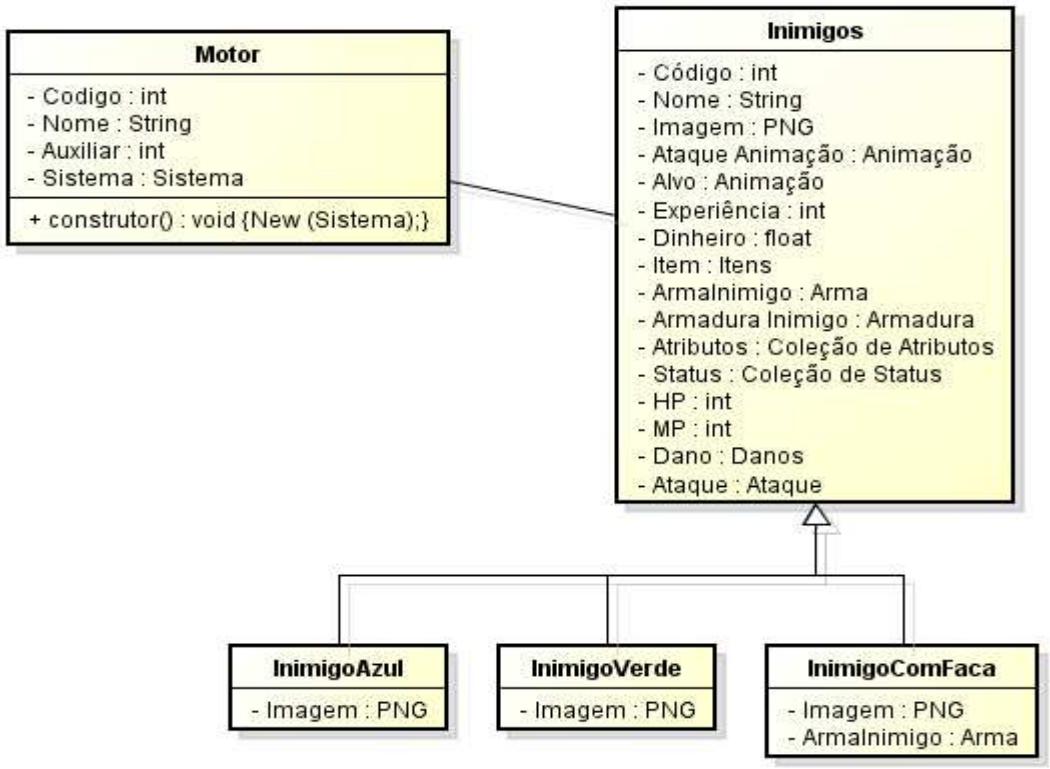

Figura 5-5: Diagrama de Classes parcial de um jogo, que mostra a criação de inimigos sem o uso do padrão Builder. Os inimigos possuem ligeiras variações de um inimigo pai.

→ Com o uso do padrão

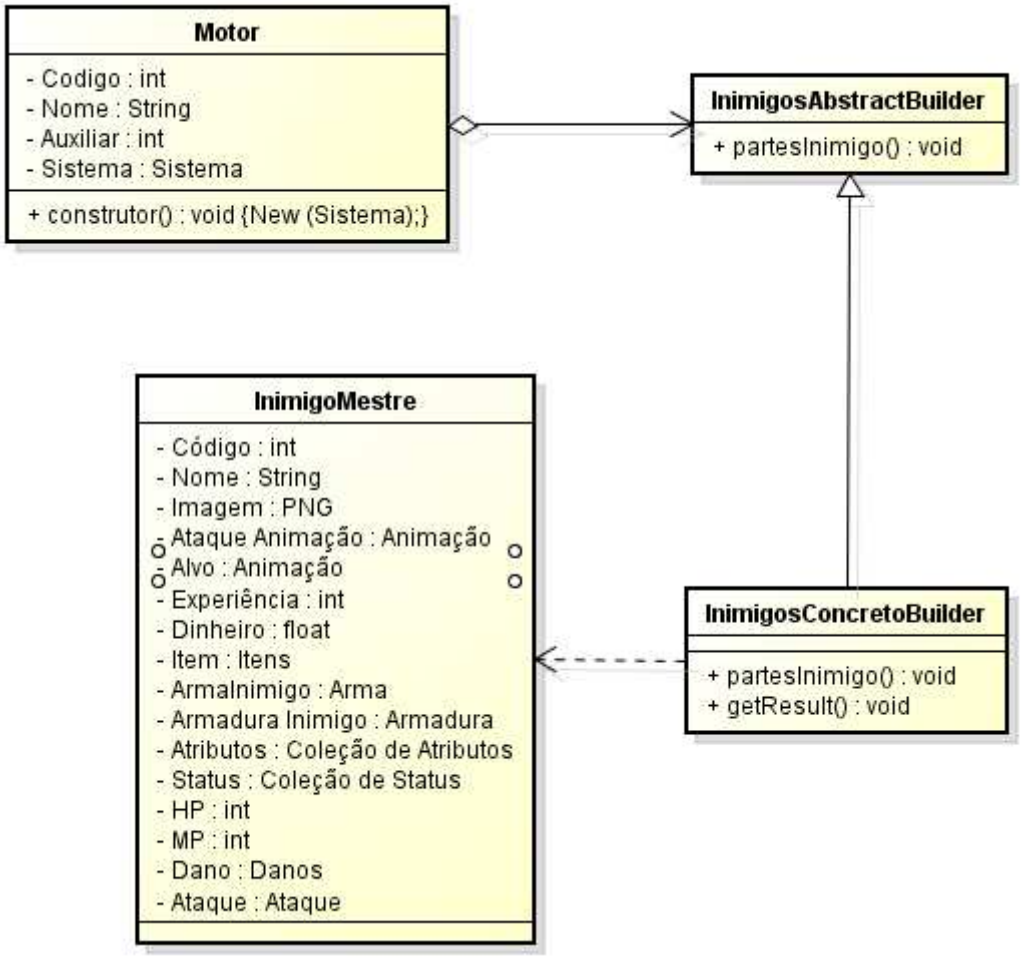

Figura 5-6 Diagrama de Classes parcial de um jogo, que mostra a criação de inimigos com o uso do padrão Builder. Cada inimigo pode ter algumas características que o tornaram único no jogo, sendo que todas estas características provém de uma única classe inimigo.

5.3. Factory Method

O padrão *Factory Method* é um padrão de criação que define uma interface para criar objetos, porém, deixa que as subclasses decidam qual classe deve instanciar esse objeto (GAMMA, HELM, *et al.*, 2000).

O padrão *Factory Method* não possui qualquer referência direta sobre seu uso no desenvolvimento de jogos.

5.3.1. Onde se aplica:

Em empresas de desenvolvimento de softwares em geral, o reaproveitamento de código em diferentes projetos traz alguns benefícios como, por exemplo, redução no tempo de desenvolvimento de novos projetos (DEITEL e DEITEL, 2005). Com jogos isso não poderia ser diferente, contudo, classes planejadas sem esse objetivo limitam ou impedem este tipo de procedimento, porém, buscar a criação de códigos reaproveitáveis pode trazer alguns prejuízos como o gasto maior de tempo no projeto atual e a geração de linhas de códigos desnecessárias.

5.3.2. Solução proposta pelo padrão:

Em jogos, uma boa solução para facilitar o reaproveitamento de código em projetos é o uso do padrão *Factory Method*, que propõe classes que podem redefinir alguns ou todos os métodos que geram algo dentro de um jogo.

Um exemplo disso pode ser visto na geração de cenários e personagens para jogos de tiro, como por exemplo, Quake, mostrado na Figura 5-7(a), Duke Nukem, na Figura 5-7(b) e Doom, na Figura 5-7(c). É possível notar nestes jogos a semelhança entre os cenários e inimigos, mudando somente aparência visual e alguns detalhes de movimentação, detalhes que podem ser facilmente reprogramados com a redefinição de métodos das classes geradores destes itens.

A Figura 5-8 mostra o diagrama parcial de dois jogos gerando seus cenários individualmente, com códigos semelhantes entre eles. Já a Figura 5-9 mostra a aplicação do padrão que concede aos jogos uma classe geradora e um método que fabrica os itens necessários para os jogos que precisam fazer apenas ajustes a seus respectivos sistemas.

Figura 5-7: (a) Quake (b) Duke Nukem (c) Doom. Como é possível observar, os cenários e personagens são muito semelhantes em relação a várias características.

5.3.3. Diagramas:

→ Sem o uso do padrão

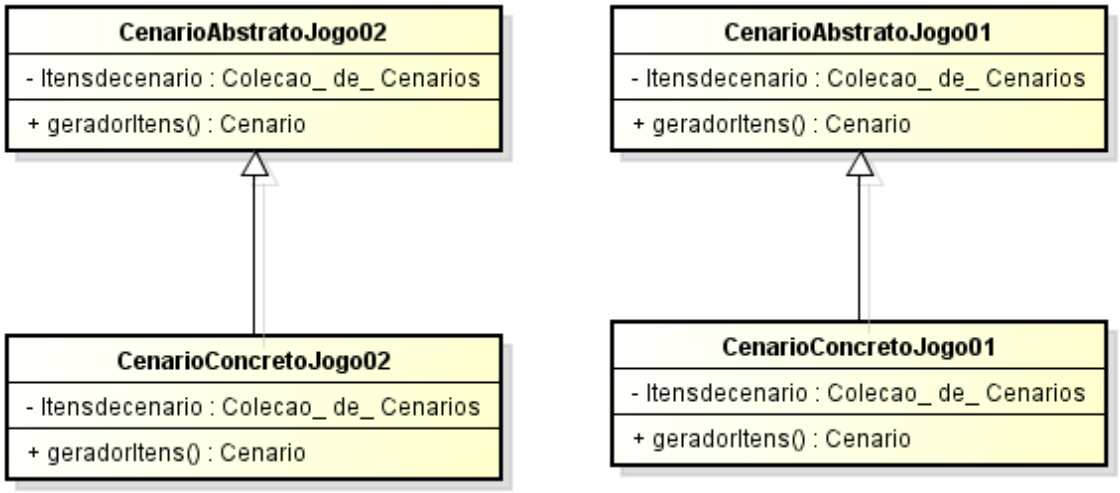

Figura 5-8: Diagrama de classes parcial que mostra a geração de cenários de dois jogos. Cada jogo possui seu método e parte do código foi replicado.

→ Com o uso do padrão

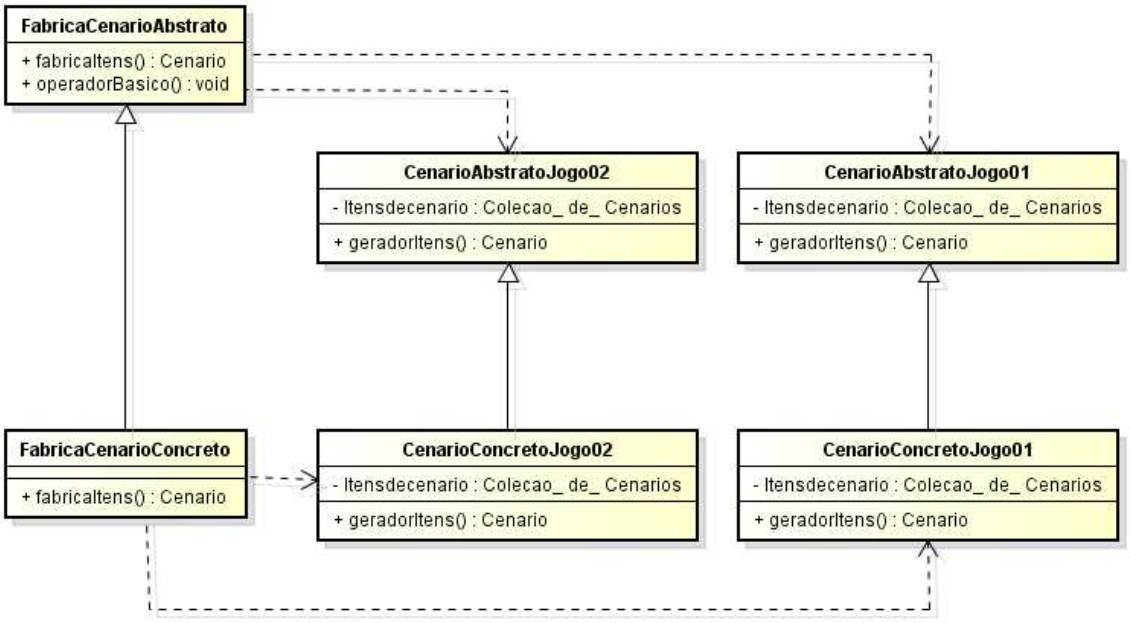

Figura 5-9: Diagrama de classes parcial que mostra a geração de cenários para dois jogos distintos partindo da mesma classe que fabrica cenários e cada jogo subscrever os métodos que necessita para ajustar a classe as suas necessidades.

5.4. Prototype

O padrão *Prototype* é um padrão de criação que define tipos, usando a instância de um objeto como protótipo e gera novos objetos, simplesmente copiando (clonando) esse protótipo (GAMMA, HELM, *et al.*, 2000).
O padrão *Prototype* assim como o *Abstract Factory* é citado apenas no artigo de Ampatzoglou, Gortzis (et al., 2011) que limita apenas em constatar a quantidade de *bugs* reportados sem e com o uso deste padrão no desenvolvimento de jogos. Neste trabalho é apresentado um exemplo de onde e como o padrão pode ser aplicado no desenvolvimento de jogos.

5.4.1. Onde se aplica:

Como foi discutida anteriormente, a criação de inimigos idênticos ou com extrema semelhança podem gerar críticas de seus jogadores, porém, em dispositivos com recursos limitados (celulares, *tablets*, entre outros) essa ideia pode ser a única saída para a criação de grupos de inimigos. Uma dificuldade que pode surgir na criação de inimigos replicados é a duplicação de código em cada unidade destes elementos, gerando esforço na criação e nos possíveis ajustes que esses elementos poderão sofrer durante o desenvolvimento. Um exemplo desta replicação pode ser visto no diagrama de classes da Figura 5-10, que mostra três inimigos replicados em uma fase.

5.4.2. Solução proposta pelo padrão:

Uma solução para este problema é proposta pelo padrão *prototype*. Como se pode perceber na

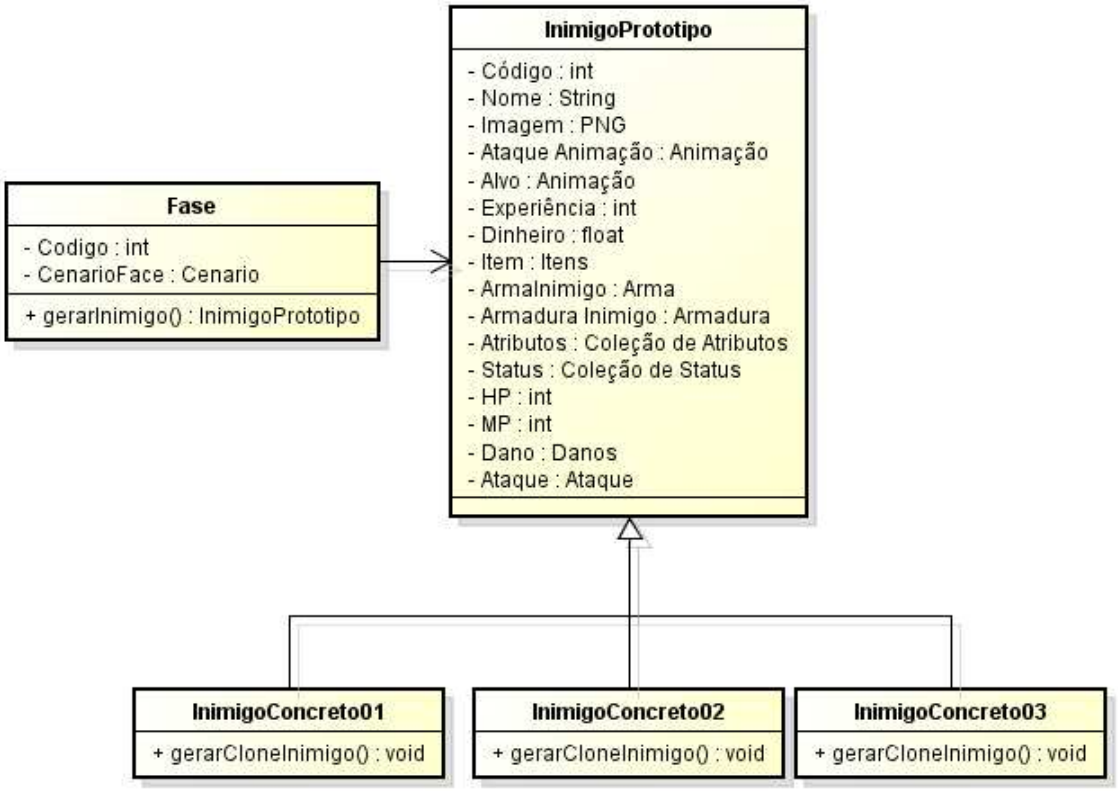

Figura 5-11, o uso do padrão faz com que as classes dos inimigos sejam independentes da fase e a escolha da quantidade de inimigos gerados é feita em tempo de execução e pode ser ajustada de acordo com a necessidade do jogo e a carga de objetos instanciados que o dispositivo suporta. O método *gerarInimigo*() cria um novo objeto solicitando que a classe *InimigoPrototipo* (um de seus filhos) se duplique.

5.4.3. Diagramas:

→ Sem o uso do padrão

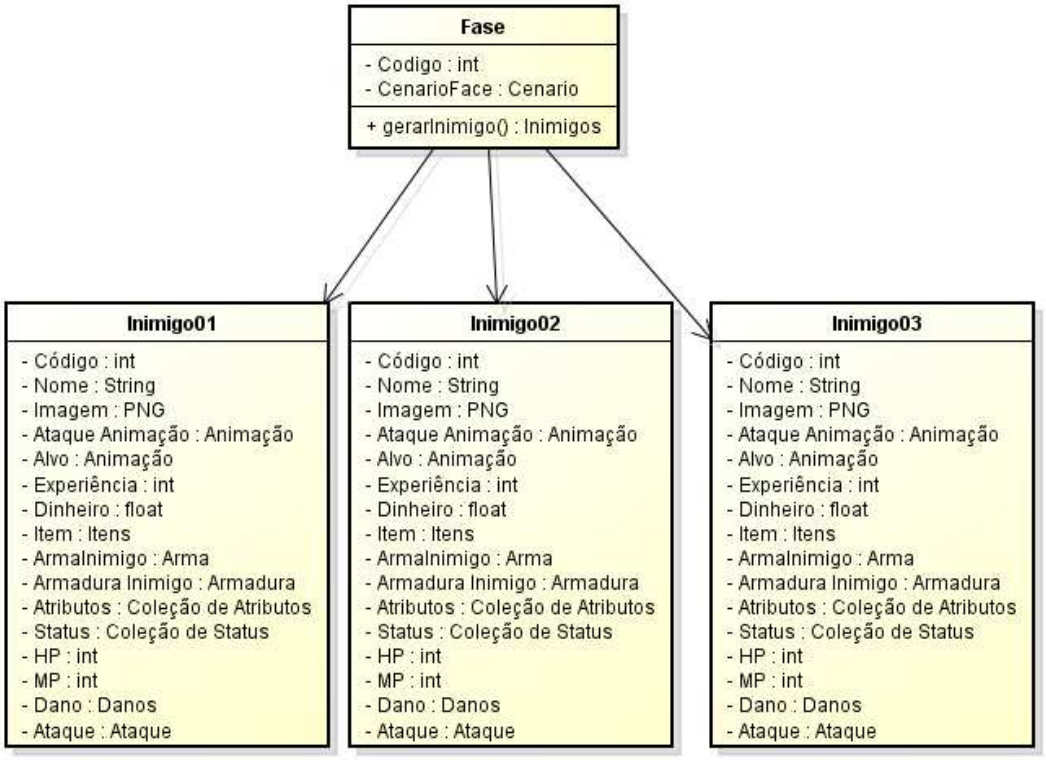

Figura 5-10: Diagrama parcial de um jogo que mostra a geração de inimigos idênticos sem o uso do padrão *Prototype*.

→ Com o uso do padrão

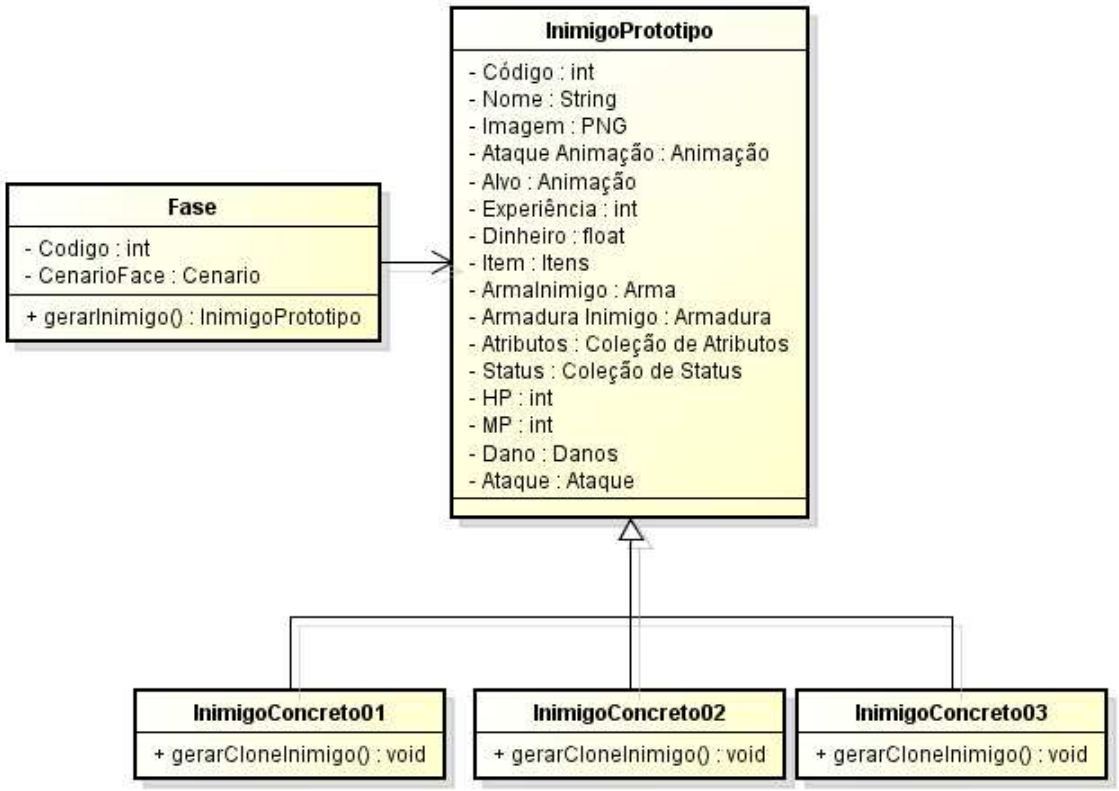

Figura 5-11:Diagrama parcial de um jogo que mostra a geração de inimigos idênticos com o uso do padrão *Prototype*.

5.5. Singleton

O padrão *Singleton* é um padrão de criação com o objetivo de garantir que só existirá uma única instância de uma determinada classe e propõe outra classe que servirá como único ponto de acesso a classe original (GAMMA, HELM, *et al.*, 2000).

O padrão *Singleton* é citado, além do artigo de Ampatzoglou, Gortzis (et al., 2011), que apenas referencia *bugs*, no artigo de Trindade e Fischer (2008), que mostra apenas o uso do padrão em uma *engine* de criação de jogos e não nos jogos em si e no artigo Gestwicki e Sun (2008), que descreve bem o padrão, mas com foco voltado no aprendizado dos padrões para alunos de graduação. Neste trabalho, o foco principal é o desenvolvimento de jogos, o que o diferencia dos demais.

5.5.1. Onde se aplica:

Diversos elementos de um jogo não podem ser replicados em nenhum outro lugar, pois tal replicação poderá gerar inconsistências, tanto lógicas, como no enredo do jogo. Um exemplo disso é o personagem principal que o jogador irá controlar durante o jogo.

Em diversos jogos, o personagem principal só pode aparecer uma vez. Um exemplo de problema que pode acontecer com a duplicação deste personagem é a inconsistência de eventos. Um determinado evento pode acontecer se houver uma colisão entre o personagem e um inimigo, porém, se houver uma replicação de personagens, havendo uma colisão, este evento poderá ocorrer em duplicidade, ocorrer uma vez ou simplesmente não ocorrer, em qualquer uma destas alternativas, o jogador poderá se confundir e rejeitar o jogo.

5.5.2. Solução proposta pelo padrão:

Uma forma de garantir uma única instância de um objeto qualquer é utilizar o padrão de projeto *Singleton*. A Figura 5-12 mostra uma situação sem o uso do padrão, onde uma programação errada em algum método da classe Motor pode gerar mais de uma instância da classe Personagem.

Na Figura 5-13 têm-se a aplicação do padrão *Singleton*, que sugere uma classe para instanciar da classe personagem, através de um método estático que não irá permitir uma segunda instanciação, garantindo assim a unicidade da instância da classe Personagem, evitando possíveis inconsistências e erros.

5.5.3. Diagramas:

→ Sem o uso do padrão

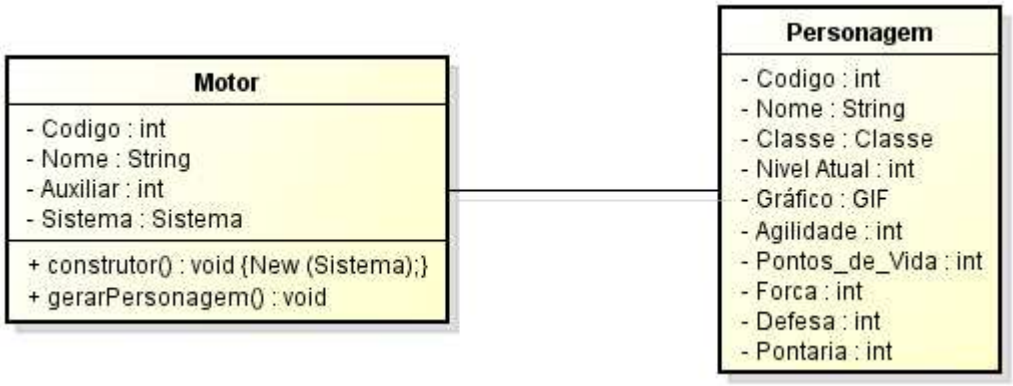

Figura 5-12: Diagrama parcial de um jogo sem o uso do padrão *Singleton*. O motor do jogo pode instanciar mais de um personagem, pois não há garantias da unicidade.

→ Com o uso do padrão

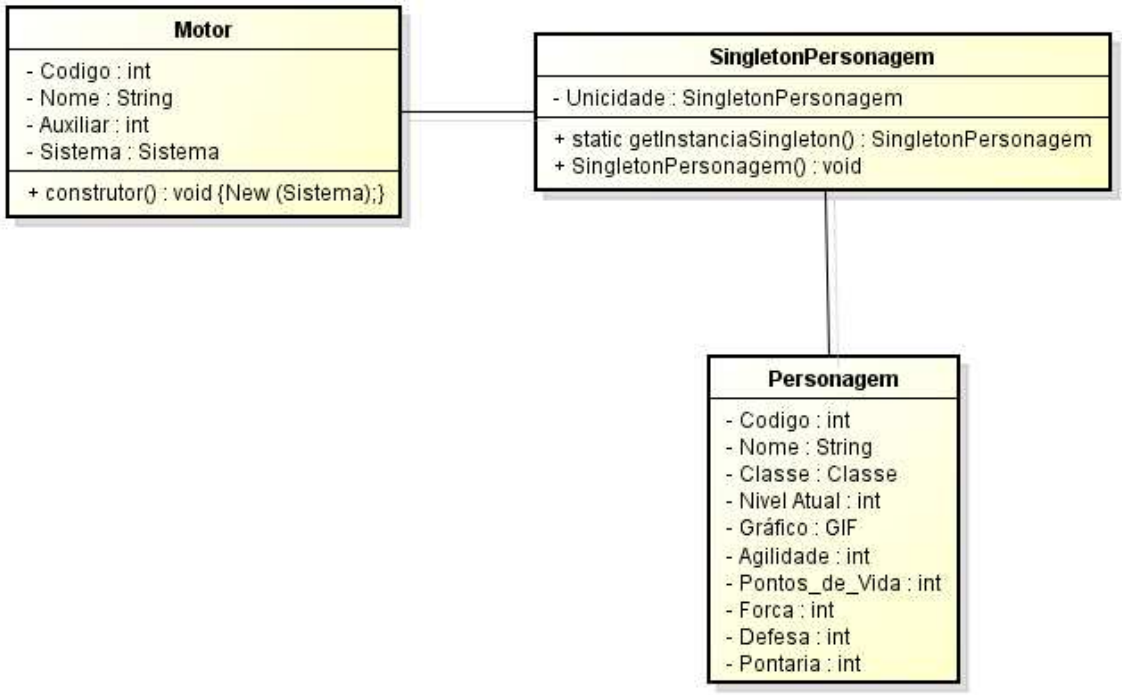

Figura 5-13: Diagrama parcial de um jogo com o uso do padrão *Singleton*. O personagem só pode ser instanciado através da classe *Singleton*, que garante a unicidade da instância da classe Personagem

5.6. Adapter

O padrão *Adapter* é um padrão estrutural que converte uma interface ou um objeto de uma classe em outra interface ou objeto no formato solicitado pelo cliente (GAMMA, HELM, *et al.*, 2000).

O padrão *Adapter* é referenciado nos trabalhos de Ampatzoglou, Gortzis (et al., 2011), que apenas referencia *bugs*, e de Trindade e Fischer (2008), mas sem referência direta de sua aplicação em jogos. Este trabalho descreve uma aplicação direta do uso do padrão no desenvolvimento de jogos.

5.6.1. Onde se aplica:

Um grande problema encontrado hoje no desenvolvimento de jogos é a adaptação do jogo aos mais diferentes tipos de dispositivos. São celulares, *smartphones*, ipads e tantos outros aparelhos, cada um com quantidades diferentes de processamento, memória, tamanho de tela, entre outras características que influenciam diretamente na programação de um jogo. Fazer com que o jogo se adapte, ao máximo possível de dispositivos do mercado, necessita que as classes diretamente relacionadas a essas características mutantes estejam fracamente acopladas com a programação do jogo em si.

Um exemplo do problema pode ser visto na Figura 5-14, que mostra o jogo *Super Volley Ball* sendo executado em um *smartphone Sansung Galaxy* 5, como ele não foi criado para este tipo de dispositivo, o próprio aparelho faz suas adaptações, deixando a superfície visual do jogo pequena, não aproveitando os recursos do aparelho, o que pode causar insatisfação por parte dos jogadores (PERUCIA, BERTHÊM, *et al.*, 2005).

Ao fazer uma análise do diagrama constante na Figura 5-15, vemos que qualquer alteração nos dispositivos e principalmente inclusão de novos dispositivos, causará interferências diretas nas classes que enviam dados para eles, tornando complexa a adaptação entre estes dispositivos e as classes de baixo nível do jogo.

5.6.2. Solução proposta pelo padrão:

A solução deste problema pode ser encontrada com o uso do padrão *Adapter*. A proposta do padrão é fornecer um ponto único de comunicação entre qualquer dispositivo e o baixo nível do jogo. Conforme é mostrado no diagrama da Figura 5-16, o padrão permite que, em uma única classe, o programador possa desativar funções não essenciais do jogo, com o objetivo de adaptá-lo aos dispositivos de pouca memória ou, também em uma única classe, fazer as adaptações necessárias no display, para que o jogo se adapte aos mais diferentes tamanhos de tela dos dispositivos.

Com o uso deste padrão, é possível incluir novos dispositivos, com o mínimo de esforço, no que diz respeito à adaptação deste jogo aos mais diferentes dispositivos oferecidos pelo mercado e a novos dispositivos que poderão surgir no futuro.

Figura 5-14: *Super Volley Ball* sendo executado em um *Sansung Galaxy* 5. A tela do aparelho e a superfície de cobertura do jogo estão marcados em vermelho. A diferença é evidente.

5.6.3. Diagramas:

→ Sem o uso do padrão

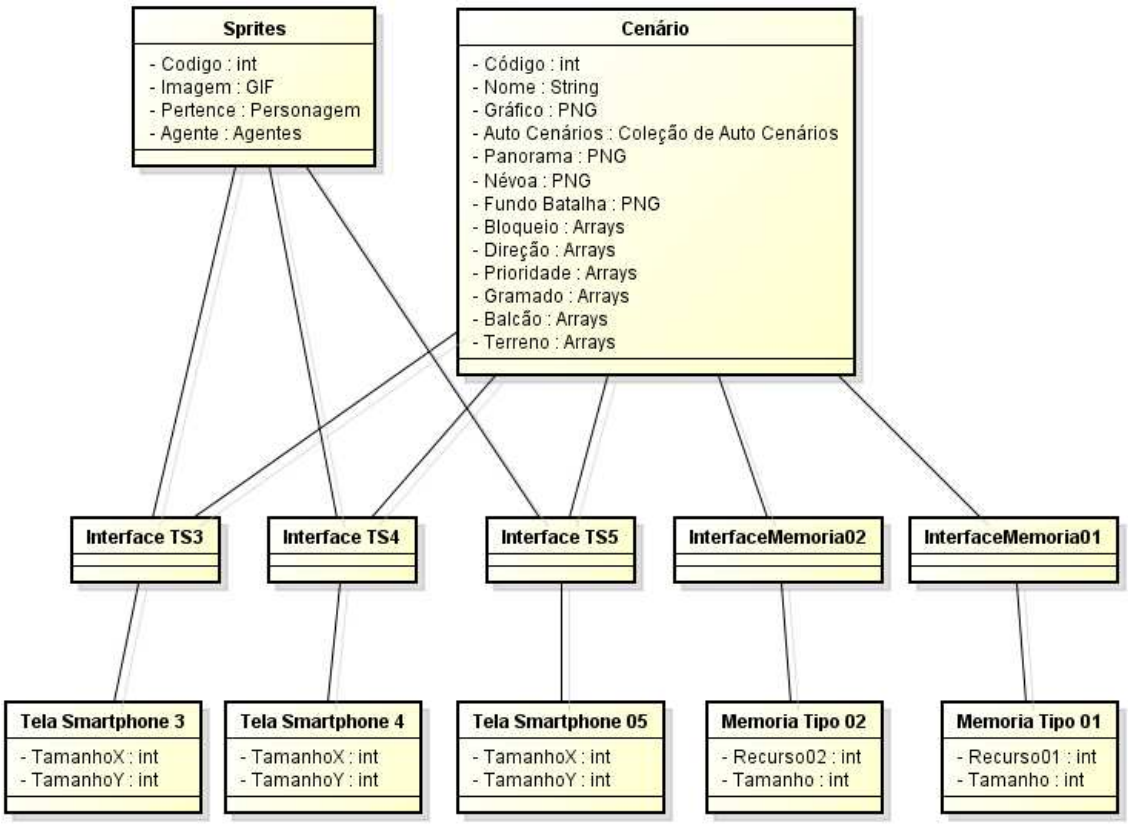

Figura 5-15: Diagrama parcial de um jogo qualquer, sem o uso do padrão Adapter

→ Com o uso do padrão

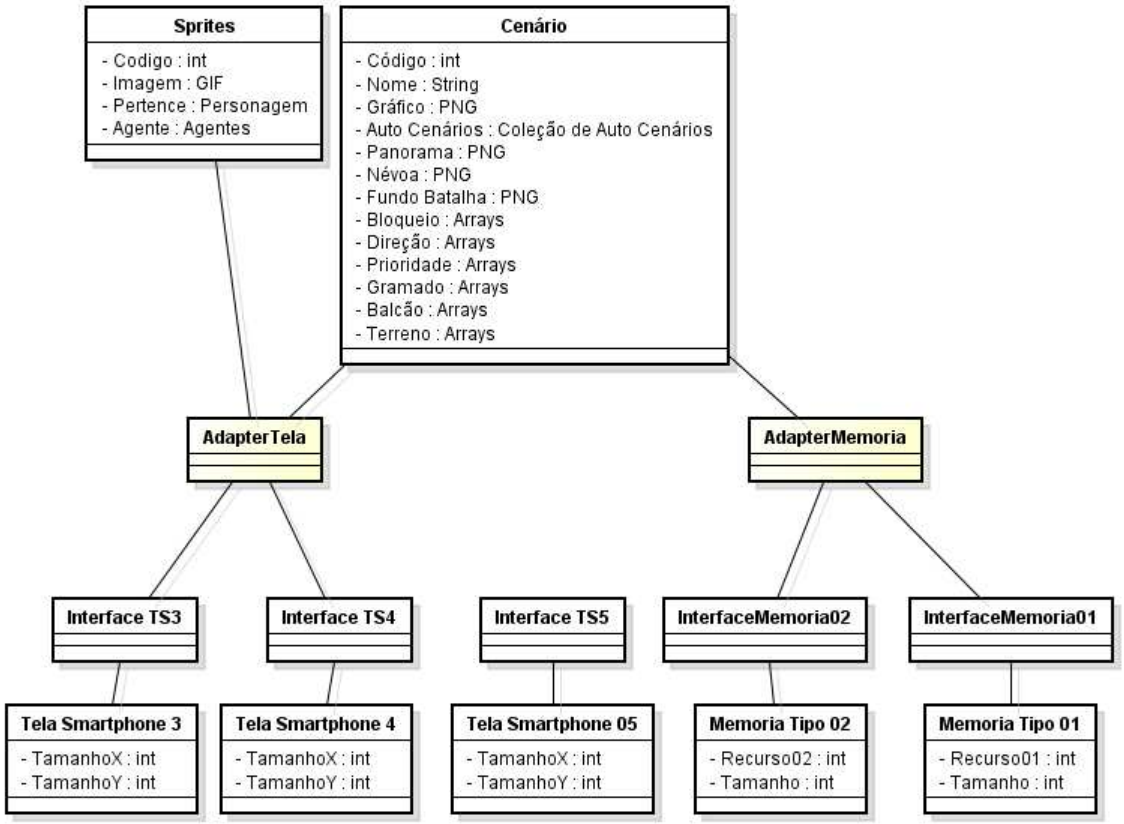

Figura 5-16: Diagrama parcial de um jogo qualquer, com o uso do padrão Adapter

5.7. Bridge

O padrão *Bridge* é um padrão estrutural que desacopla uma abstração de sua implementação, permitindo que os dois possam variar independentemente (GAMMA, HELM, *et al.*, 2000).

O padrão *Bridge* é citado apenas no trabalho de Ampatzoglou e Chatzigeorgiou (2006) que faz uma boa descrição da aplicação do padrão em jogos. Nesta dissertação é analisado o exemplo fornecido por este autor.

5.7.1. Onde se aplica:

Em jogos 3D, os objetos tridimensionais precisam receber textura em uma das fases finais na construção destes objetos. Essa texturização pode ser feita de maneira simples, criando classes para os diferentes tipos de textura e associando essas classes à classe que cria um objeto, conforme ilustrado na Figura 5-17.

Esta forma de implementar gera um forte acoplamento entre as classes das texturas e a classe do objeto, tornado custosa a modelagem de uma textura aos mais diversos objetos de um jogo, bem como a inclusão de novas texturas, pois, estas ações exigem modificações na classe que gera o objeto.

5.7.2. Solução proposta pelo padrão:

Uma solução para fazer este desacoplamento é o uso do padrão de projeto *Bridge*. Este padrão pode ser utilizado em qualquer jogo com objetos em 3D, que podem receber texturas de várias maneiras, como por exemplo, cor fixa, materiais, imagem 2D, entre outros. Ao mesmo tempo, esses objetos podem também variar sua composição, uma vez que podem ser gerados através de uma primitiva (cilindro, esfera, cubo) ou uma importação a partir de um pacote 3D, por exemplo (AMPATZOGLOU e CHATZIGEORGIOU, 2006). Uma modelagem do padrão *Bridge* pode ser vista na Figura 5-18.

Com o padrão *Bridge*, a classe abstrata Objeto3D define uma interface de abstração, que mantém uma referência a um objeto do tipo *BridgeAcabamento*. O *Bridge* define uma interface entre as classes que não precisa corresponder exatamente à interface da abstração. Normalmente a interface implementa apenas as operações primitivas e as classes concretas implementam as operações baseadas nestes primitivas.

As abstrações refinadas (Primitivas3D, Estruturas3D, Modelos3D) estendem a interface definida pela abstração e a implementação concreta (Classes Textura e Materiais) define sua implementação específica. O padrão permite que uma abstração possa ser configurada em tempo de execução e um objeto possa alterar suas implementações em tempo de execução.

Desta forma, o padrão elimina dependências da implementação, deixando o jogo mais extensível.

5.7.3. Diagramas:

→ Sem o uso do padrão

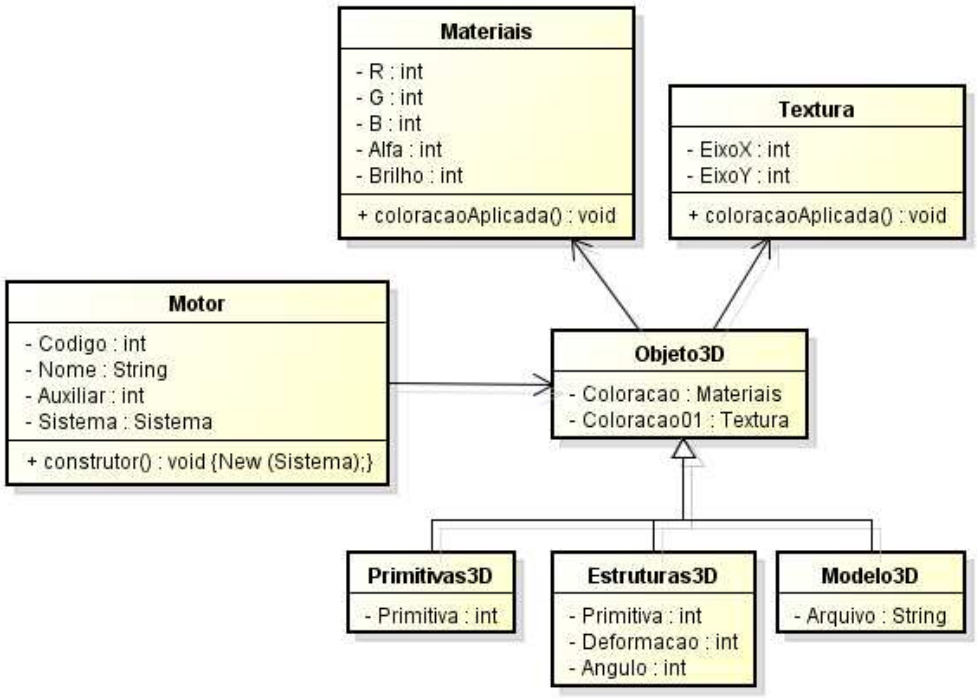

Figura 5-17: Diagrama parcial da criação de uma imagem 3D para jogos, sem o uso do padrão *Bridge*. Pode-se notar o forte acoplamento entre o formato do objeto e sua textura, o que pode gerar muito trabalho quando houver necessidade de remodelação da imagem e a adaptação desta textura nessa imagem e também na inclusão de novas imagens.

→ Com o uso do padrão

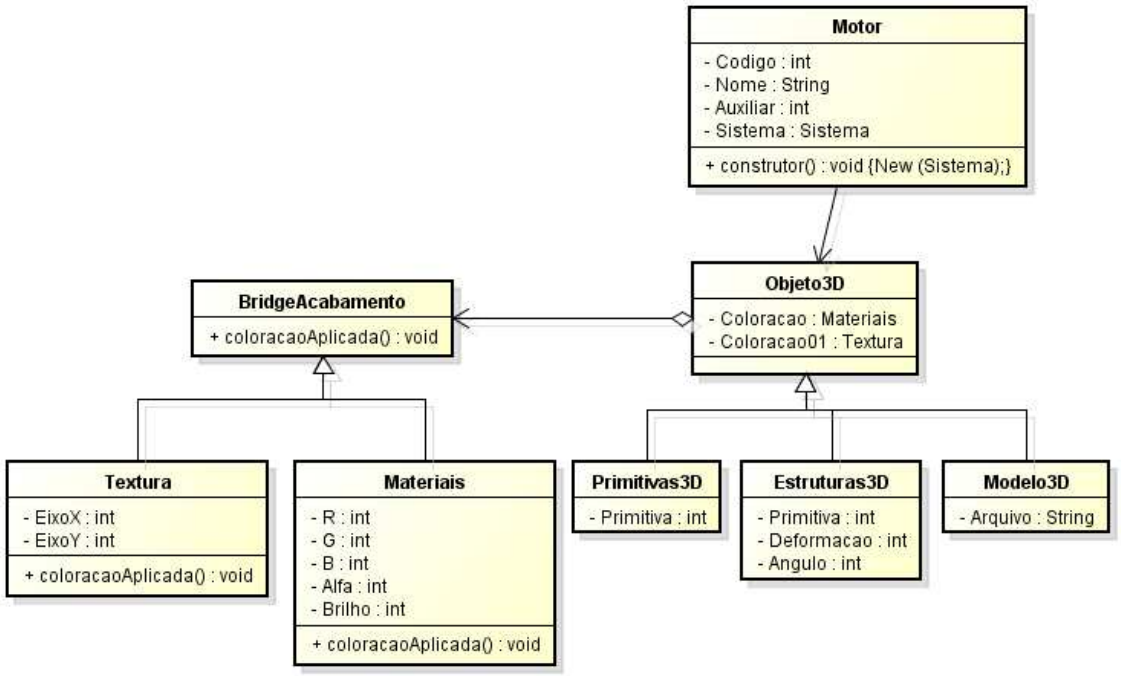

Figura 5-18: Diagrama parcial da criação de uma imagem 3D para jogos, com o uso do padrão *Bridge*. O uso do padrão causou um fraco acoplamento entre a forma de constituição da imagem e sua textura, facilitando possíveis ajustes ou inclusão de novas formas de criação e novas texturas.

5.8. Composite

O padrão *Composite* é um padrão estrutural que permite tratar objetos individuais e composições hierárquicas desses objetos da mesma maneira (GAMMA, HELM, *et al.*, 2000).

O padrão *Composite*, assim como o *Prototype* e o *Abstract Factory* é citado apenas no artigo de Ampatzoglou, Gortzis (et al., 2011) que limita apenas em constatar a quantidade de *bugs* reportados sem e com o uso deste padrão no desenvolvimento de jogos. Neste trabalho é apresentado um exemplo de onde e como o padrão pode ser aplicado no desenvolvimento de jogos.

5.8.1. Onde se aplica:

Em muitos jogos 3D, os personagens são formados por uma série de elementos básicos de criação, como cubos, cilindros, esferas, pirâmides entre outros, chamados de Primitivas Gráficas. Cada grupo desses elementos, repetidos ou não, formam uma Hierarquia Gráfica ou Estrutura Gráfica (PAULA FILHO, 2011). A Figura 5-19, mostra o jogo *Adrenaline Crew*, onde podemos ver muitos elementos formados a partir de primitivas gráficas.

Figura 5-19: Imagem do jogo *Adrenaline Crew*. Nesta imagem é possível identificar algumas Estruturas Gráficas e quais primitivas geraram tais estruturas.

Muitas vezes, os algoritmos que manipulam as estruturas não são os mesmos que manipulam as primitivas, por sua natureza mais complexa. Esta situação gera uma maior dificuldade no momento de manipular esses elementos, tornando o código menos legível e com métodos, muitas vezes, replicados, conforme mostra a Figura 5-20.

5.8.2. Solução proposta pelo padrão:

Uma solução para este problema é o uso do padrão *Composite*. Com ele, elementos complexos (estruturas gráficas) constituídos a partir de elementos mais simples (primitivas) podem ser tratados e manipulados da mesma maneira. As classes que requisitarem geração de hierarquias gráficas poderão fazê-la através da mesma classe abstrata que gera as primitivas gráficas, conforme mostra a Figura 5-21, deixando o código mais simples de ser utilizado e de ser ajustado.

5.8.3. Diagramas:

→ Sem o uso do padrão

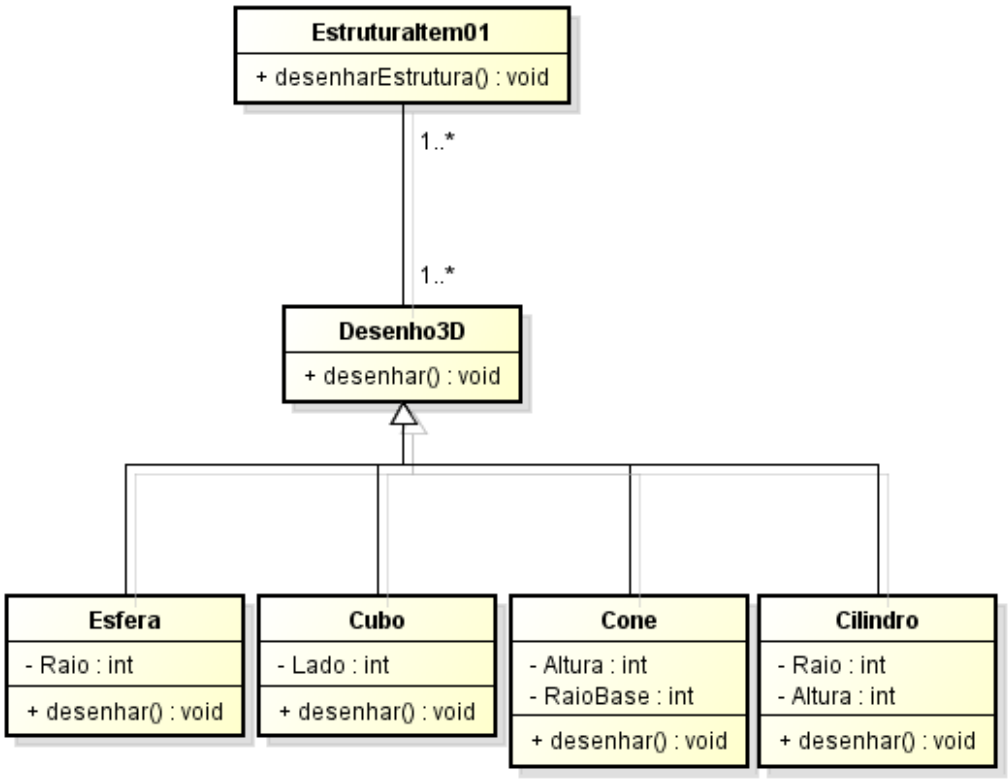

Figura 5-20: Diagrama parcial sem o uso do padrão *Composite*. É possível perceber que a classe *EstruturaItem01* usa diversos desenhos gerados pela classe *Desenho3D*, mas possui métodos distintos, o que poderá gerar duplicidade de código ou custo elevando no momento de um ajuste.

→ Com o uso do padrão

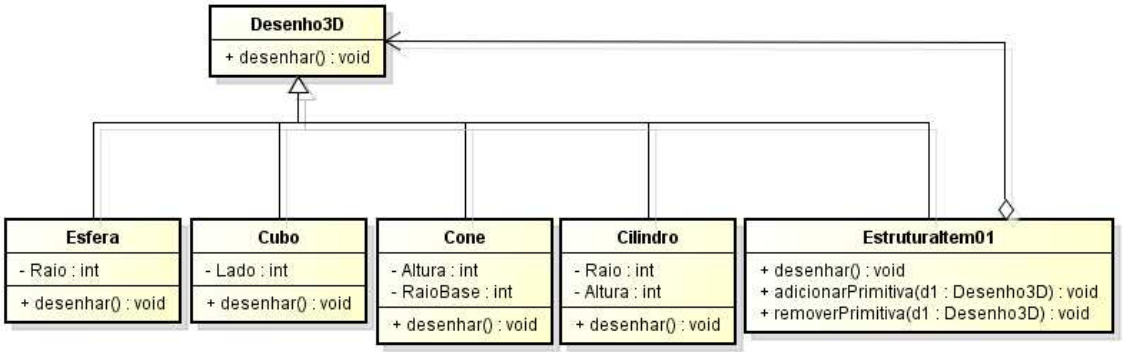

Figura 5-21: Diagrama parcial com o uso do padrão Composite. Neste diagrama nota-se que a classe abstrata Desenho3D está representando tanto primitivas quando estruturas gráficas, que podem agora serem tratados da mesma maneira.

5.9. Decorator

O padrão *Decorator* é um padrão estrutural que agrega responsabilidades adicionais a um objeto em tempo de execução (GAMMA, HELM, *et al.*, 2000).
O padrão *Decorator* é citado, além do artigo de Ampatzoglou, Gortzis (et al., 2011), que apenas referencia *bugs*, no artigo Silveira e Silva (2006), que faz uma boa descrição, mas com foco no aprendizado de alunos de graduação em relação aos padrões de projeto. Neste trabalho, o foco principal é o desenvolvimento de jogos, o que o diferencia dos demais.

5.9.1. Onde se aplica:

Em diversos jogos, principalmente os RPGs, os jogadores podem customizar seu personagem das mais variadas maneiras. Seja alterando atributos como Força e Poder de Ataque, seja modificando e melhorando os itens que o personagem é capaz de utilizar. Um bom exemplo disso é o jogo *Priston Tale*, mostrado na Figura 5-22, onde o personagem inicia com uma arma e armadura simples e podem evoluir estes itens para se tornarem mais poderosos ou ganhem novos atributos, além de terem acesso a outros tipos de arma que também sofrem evolução.

Figura 5-22: Jogo *Priston Tale*. Na imagem, a personagem arqueira, usando arco, botas, bracelete e armadura evoluídas, atacando um inimigo.

O problema é que a cada vez que novos atributos (novas funcionalidades) para uma arma ou armadura forem surgindo, sendo modificados ou excluídos, a classe principal Arma ou Armadura deve ser atualizada, novos atributos surgirão e ajustes nos métodos será inevitável. Um exemplo de diagrama desta situação é ilustrado na Figura 5-23.

5.9.2. Solução proposta pelo padrão:

Este problema é solucionado com o uso do padrão *Decorator*. Com ele é possível adicionar novas características a objetos específicos, ao invés de mexer na classe inteira, com isso, a inclusão de novas características torna-se simples e as classes principais ficam intocadas. No exemplo do diagrama da Figura 5-24, temos as classes principais "Armadura", que pode ter como características especiais ser resistente a foto, frio ou gás e "Arma" que pode ter como características, ser reforçada por fogo, gelo ou gás. A classe abstrata decoradora garante que a modificação ou inclusão de novos atributos não afetará a classe principal, mas sempre será compatível, pois ambas partilham a mesma interface.

A utilização do decorador é transparente à classe "Motor" que, neste exemplo, figura-se como cliente das classes principais Armadura e Arma, tornando o código mais simples de ser modificado e mantido.

5.9.3. Diagramas:

→ Sem o uso do padrão

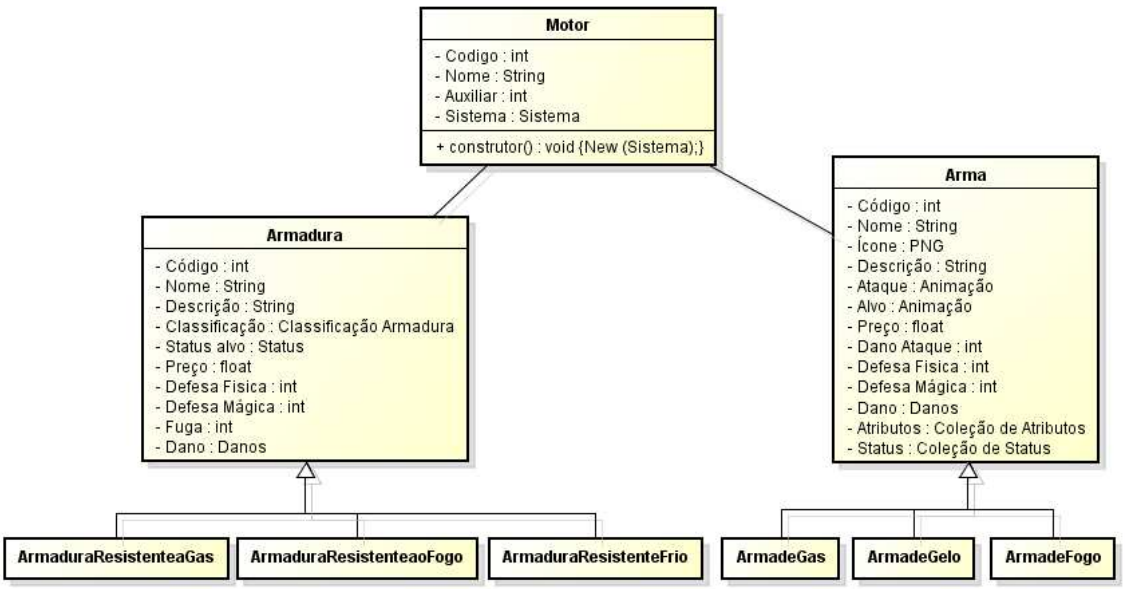

Figura 5-23: Diagrama parcial de um jogo sem o uso do padrão *Decorator*. Novos tipos de armas ou armaduras poderão causar modificações na classe principal.

→ Com o uso do padrão

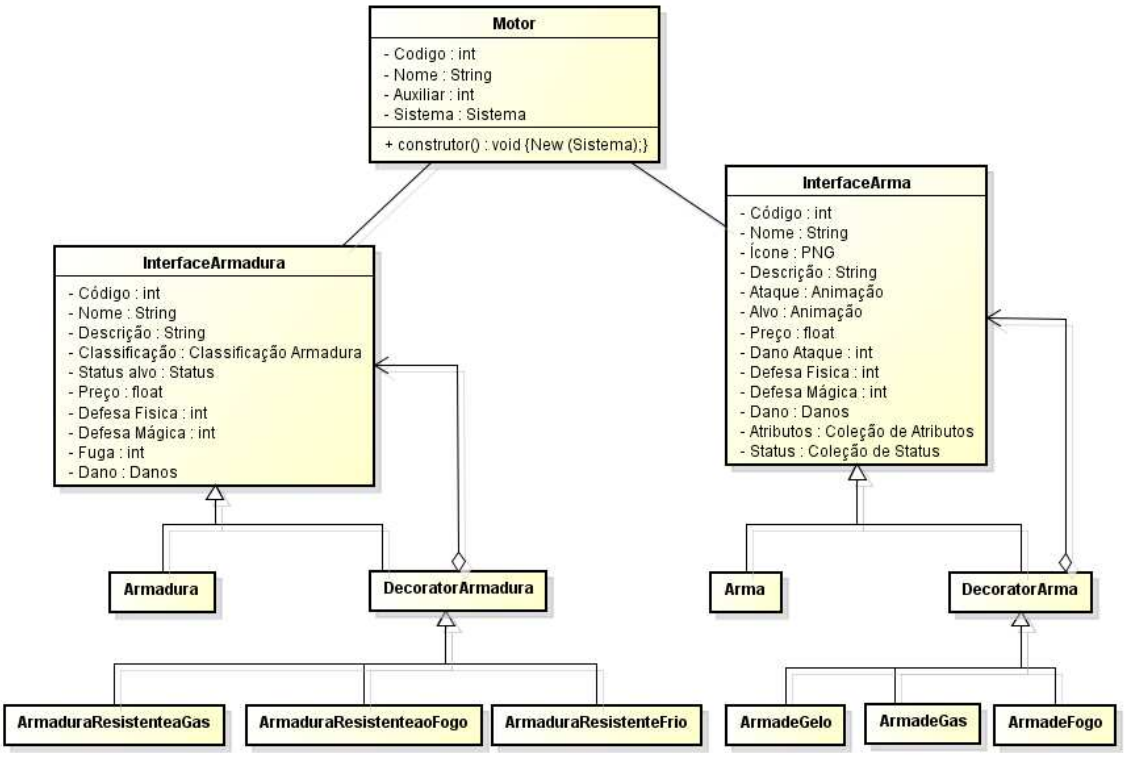

Figura 5-24: Diagrama parcial de um jogo com o uso do padrão *Decorator*. Subclasses que representam novas funcionalidades a uma classe principal deixam de ser dependentes dessa classe, mas precisam manter uma interface compatível com a interface da classe principal.

5.10. Facade

O padrão *Facade* é um padrão estrutural que fornece uma interface única entre um nível e um conjunto de interfaces de um subsistema de um nível inferior (GAMMA, HELM, *et al.*, 2000).

O padrão Facade tem seu uso em jogos descrito no artigo de Gestwicki (2007), mas sem exemplos práticos, e no artigo Gestwicki e Sun (2008), que descreve bem o padrão, mas com foco voltado no aprendizado dos padrões para alunos de graduação. Este trabalho tem o foco principal no desenvolvimento de jogos e apresenta um exemplo prático da aplicação do padrão, o que o diferencia dos demais.

5.10.1. Onde se aplica:

Um grande problema no desenvolvimento de jogos é a programação das ações dos personagens do jogo na tela, fortemente acopladas com o desenvolvimento da arte gráfica. Cálculo da altura de um pulo, *scroll* de tela que acompanha o personagem,

movimentação dos inimigos de acordo com as limitações visuais do cenário e outros elementos gráficos são exemplos de fatores que podem complicar a programação neste sentido, especialmente em jogos atuais, onde a grande variedade no tamanho dos displays limita o mercado para jogos dedicados a um único tipo de ecrã.

Um exemplo das consequências deste problema pode ser visto no jogo *Super Volley Ball*, mostrado na Figura 5-14, onde grande parte da tela não foi utilizada, deixando a superfície do jogo reduzida, com o intuito de preservar a legibilidade e movimentação normal do jogo.

Essa redução não é interessante aos usuários, que, por muitas vezes, deixa de jogar com conta deste tipo de erro (PERUCIA, BERTHÊM, *et al.*, 2005).

5.10.2. Solução proposta pelo padrão:

Uma ajuda na solução deste problema é apresentada pelo padrão Facade, que cria uma classe, responsável diretamente pela comunicação dos itens de baixo nível com a interface gráfica, sendo esta classe a única possibilidade da realização de tal comunicação, tornando as camadas de baixo nível independentes da interface gráfica e de suas particularidades. A Figura 5-25 mostra um exemplo de um jogo de RPG, cuja exibição dos cenários, agentes e itens da tela estão fortemente acoplados com as classes de mais baixo nível, dificultando a manutenção e inclusão de novos elementos na tela. Já na Figura 5-26 é apresentada a classe *Facade*, proposta pelo padrão, que é utilizada como único limite comunicante entre as classes visuais e as classes de programação de baixo nível, tornando fraco o acoplamento entre elas. Para resolver completamente o problema do dimensionamento de tela, este padrão deve ser aplicado juntamente com o padrão *Adapter*.

5.10.3. Diagramas:

→ Sem o uso do padrão

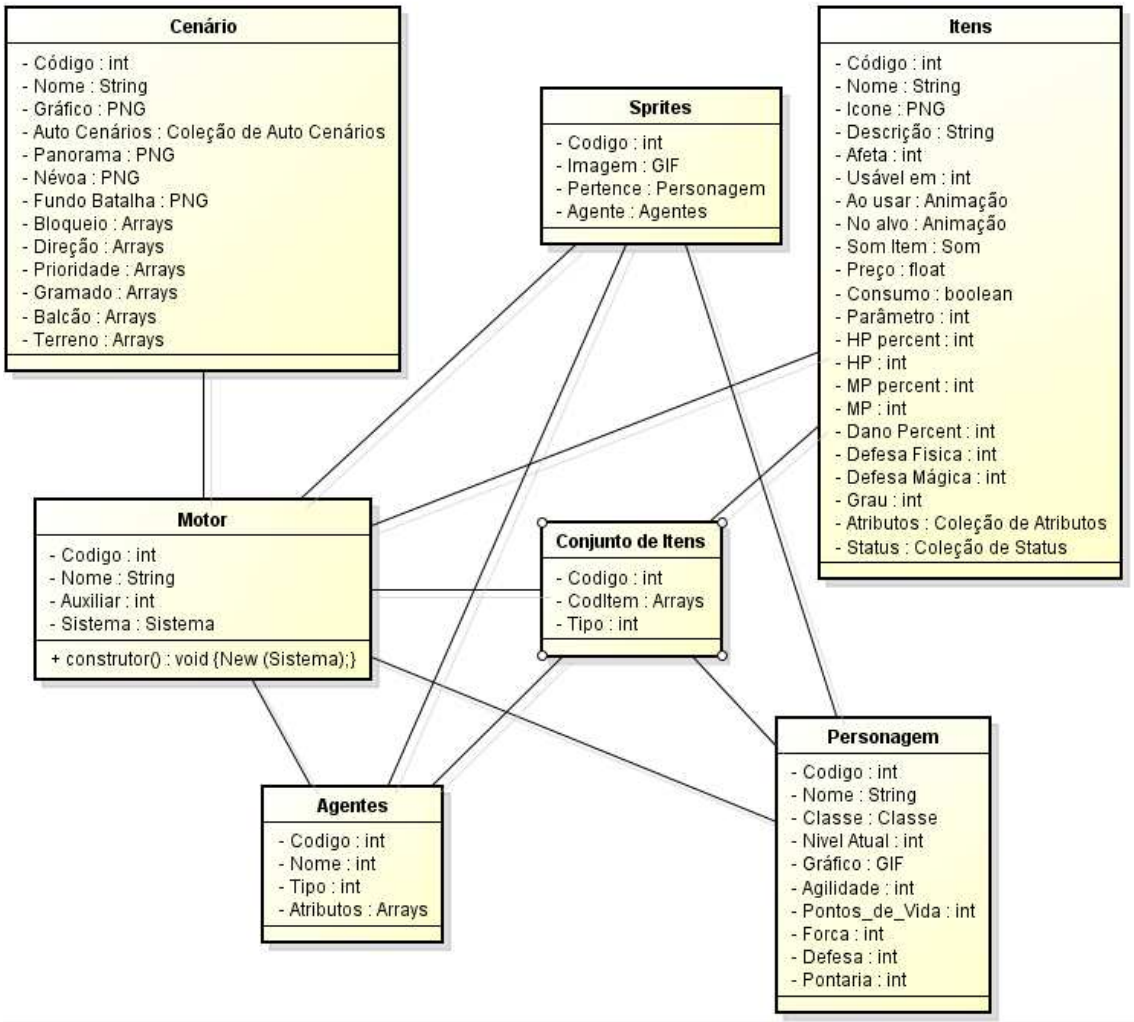

Figura 5-25: Diagrama parcial de um jogo de RPG, sem o uso do padrão *Facade*

→ Com o uso do padrão

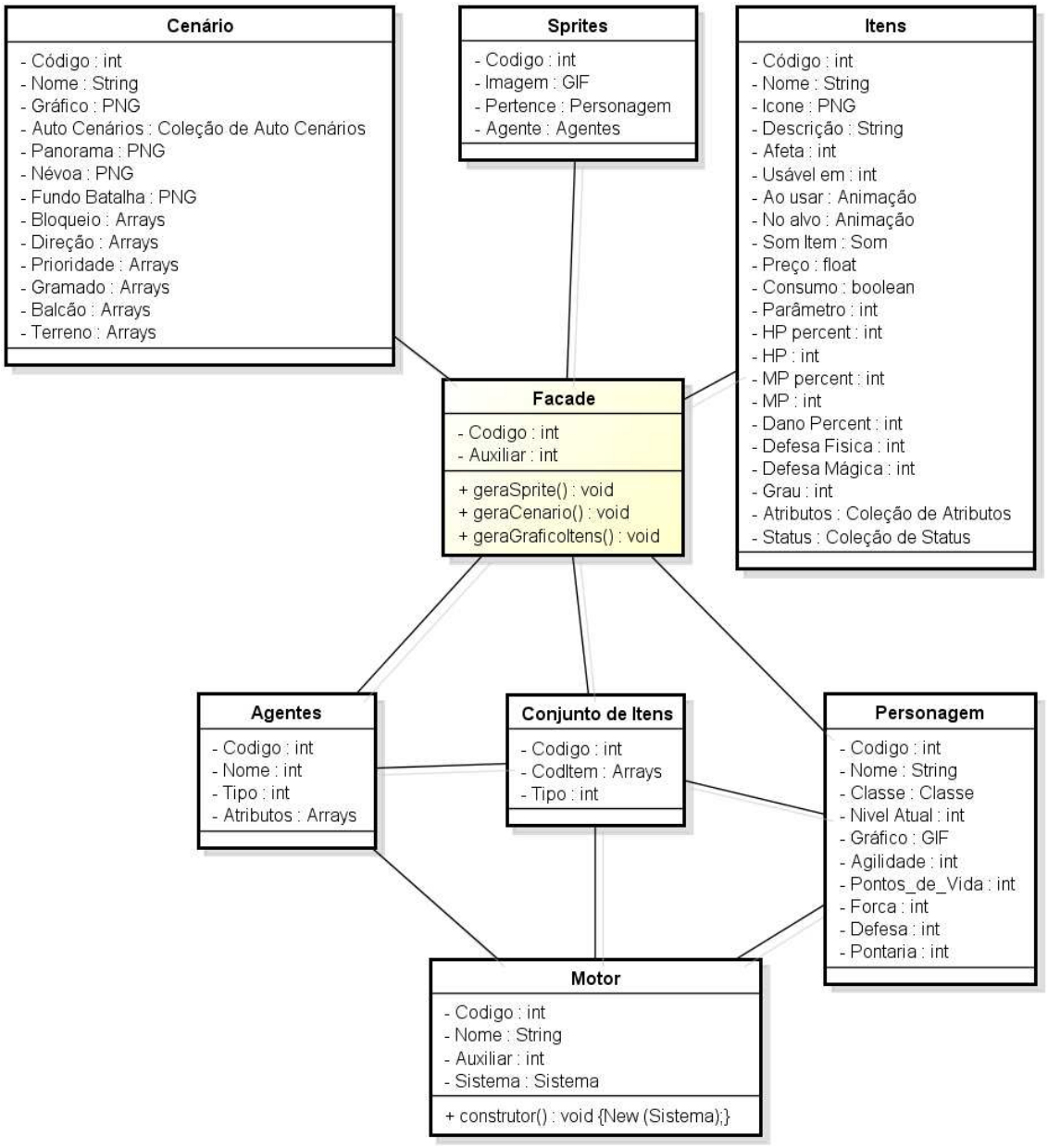

Figura 5-26: Diagrama parcial de um jogo de RPG, com o uso do padrão *Facade*

5.11. Flyweight

O padrão *Flyweight* é um padrão estrutural que usar compartilhamento e subdivisões para suportar grandes quantidades de objetos complexos em quantidades menores de memória (GAMMA, HELM, *et al.*, 2000).

O padrão *Flyweight* não possui qualquer referência direta sobre seu uso no desenvolvimento de jogos.

5.11.1. Onde se aplica:

Cada um dos inimigos ocasionais em um jogo é uma instância de um objeto. Essas instâncias consomem memória com informações, muitas vezes repetidas, pois os inimigos geralmente possuem características em comum. Quando estes inimigos andam em bandos e se aglomeram na tela, como ilustrado no jogo da Figura 5-4, vão gerar diversas instâncias simultâneas na memória real, o que pode ocasionar *slowdowns* e até *overhead*. A primeira solução é limitar a quantidade de inimigos simultâneos, o que pode distorcer o enredo do jogo, além de impactar em outras áreas do jogo, como opções de dificuldade e dinâmica do jogo.

5.11.2. Solução proposta pelo padrão:

Uma solução para reduzir o impacto destes problemas, aumentando a capacidade de gerar inimigos simultâneos, ocupando o mínimo de memória é a utilização do padrão *Flyweight*. Com o padrão, é criado um objeto compartilhado que pode ser utilizado por diversos inimigos simultaneamente. Este objeto não tem vínculo específico com nenhum inimigo e possui apenas as informações que são comuns entre eles. Com isso, cada instância de um inimigo só possui informações exclusivas, deixando os objetos menores, consequentemente, ocupando menos espaço em memória.

O diagrama ilustrado na Figura 5-27 mostra a geração de inimigos pelo motor, sem o uso do padrão. Cada objeto Inimigo terá todas as informações da classe “Inimigo”, ocupando o máximo de memória. Já na Figura 5-28 tem-se a aplicação do padrão *Flyweight*, nela é possível notar a classe *FlyweightAbstrato* que nada mais é do que uma interface na qual os *Flyweights* concretos poderão agregar as informações específicas de cada inimigo às informações gerais do *Flyweight*. Também existe a classe “InimigosInformacoesNaocompartilhadas” que pode guardar as informações que não precisam ser compartilhadas pelo *Flyweight*. O motor mantém referências aos *Flyweights*, pois ele armazena ou gera as informações específicas de cada inimigo, mas só pode gerar instâncias de Inimigos através da classe *FlyweightFabricaDeInimigos*, garantindo assim o compartilhamento apropriado das informações.

5.11.3. Diagramas:

→ Sem o uso do padrão

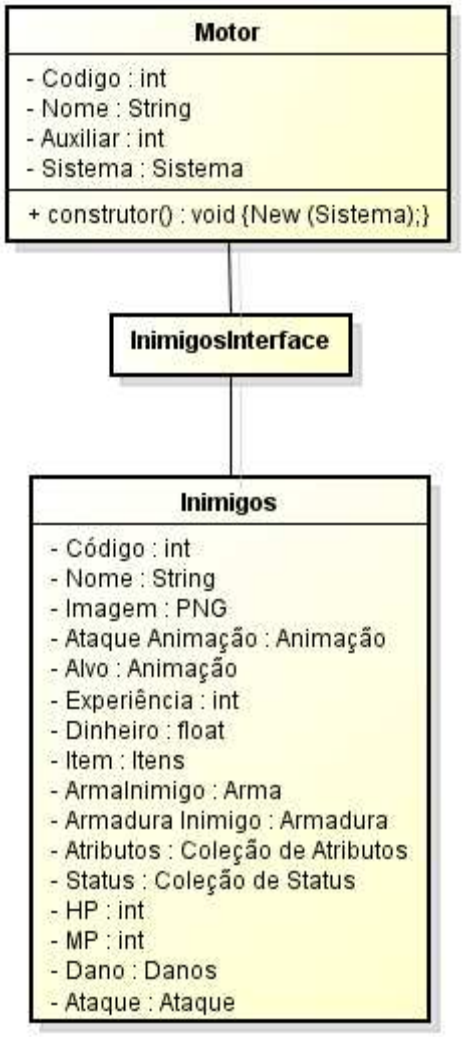

Figura 5-27: Diagrama parcial de um jogo sem o padrão *Flyweight*. O motor apenas gera instâncias completas de inimigos quando necessário.

→ Com o uso do padrão

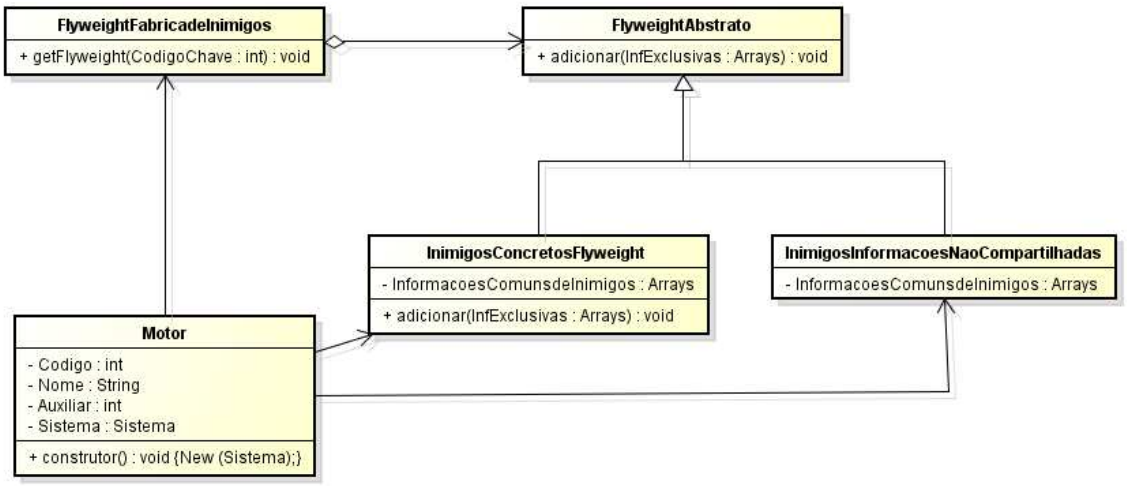

Figura 5-28: Diagrama parcial de um jogo sem o padrão Flyweight. O motor

5.12. Proxy

O padrão *Proxy* é um padrão estrutural que fornece um substituto ou ponto de apoio, onde um objeto possa controlar o acesso a outro objeto (GAMMA, HELM, *et al.*, 2000).

O padrão *Proxy*, assim como o *Composite*, o *Prototype* e o *Abstract Factory*, é citado apenas no artigo Ampatzoglou, Gortzis (et al., 2011) que limita apenas em constatar a quantidade de *bugs* reportados sem e com o uso deste padrão no desenvolvimento de jogos. Neste trabalho é apresentado um exemplo de onde e como o padrão pode ser aplicado no desenvolvimento de jogos.

5.12.1. Onde se aplica:

Em jogos grandes, com múltiplas fases, diversos elementos de cenário e imagens são específicos de uma determinada fase, porém, sem o devido tratamento, o jogo irá instanciar todos os objetos necessários, para gerar todas as fases, no início do jogo, consumindo assim muita memória e até inviabilizando a execução do jogo em diversas máquinas. Este situação é ilustrada na Figura 5-29.

Obviamente, elementos de fases seguintes só precisariam ser instanciados no momento da transição entre as fases. Este tratamento pode ser feito pelo Motor, porém, carregaria o motor do jogo com atributos e métodos que podem interferir em outros métodos do motor, deixando a modificação e inclusão de novos cenários fortemente acoplados a demais processos no jogo.

5.12.2. Solução proposta pelo padrão:

Uma solução para este problema é apresentada pelo padrão *Proxy*. Este padrão oferece um ponto de acesso para a classe que gera os cenários, controlando a instanciação e liberação de memória de seus objetos. A modelagem deste padrão pode ser vista na Figura 5-30.

A classe *CenarioProxy* mantém uma referência que permite o único acesso à classe *CenarioConcreto*, que irá realmente gerar os cenários. Essa referência pode ser feita, pois ambas as classes possuem a mesma interface, a classe Cenario.

5.12.3. Diagramas:

→ Sem o uso do padrão

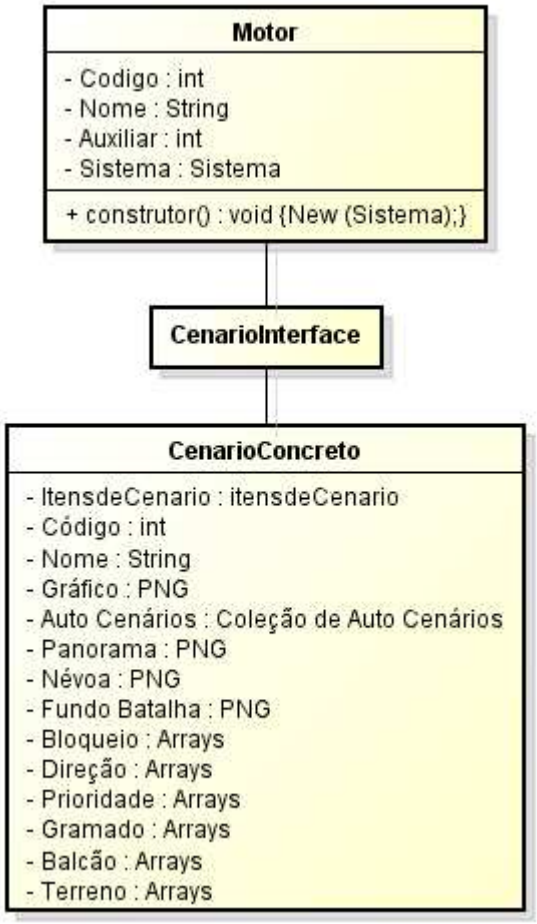

Figura 5-29: Diagrama parcial sem o uso do padrão Proxy. Todos os Itens de cenários serão gerados na inicialização do jogo.

→ Com o uso do padrão

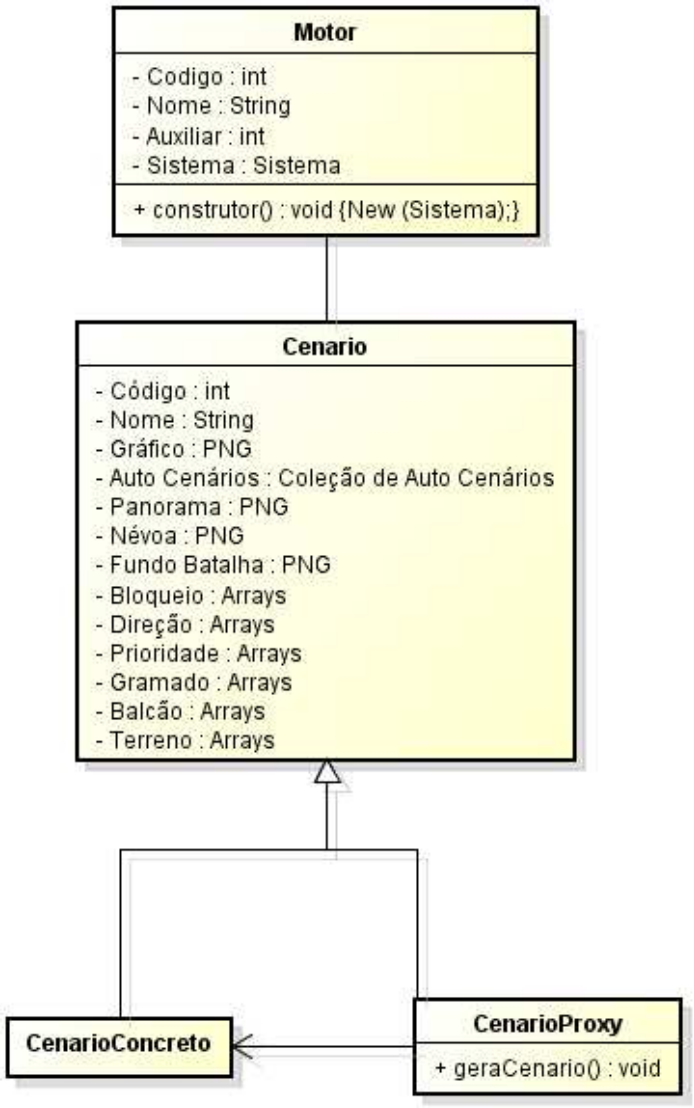

Figura 5-30: Diagrama parcial com o uso do padrão *Proxy*. Os cenários e seus itens serão gerados a partir da classe "CenarioProxy" que só irá instanciá-los quando for necessário e quando não mais for, retira a instância da memória.

5.13. Chain of Responsibility

O padrão *Chain of Responsibility* é um padrão comportamental que compõe objetos em forma de cascata para, através dela, delegar uma requisição até que um objeto que possa atendê-la, ou seja, evita o acoplamento entre o objeto que emite uma requisição do objeto que irá receber, com isso, outros objetos também poderão receber e processar a requisição (GAMMA, HELM, *et al.*, 2000).

O padrão *Chain of Responsibility* não possui qualquer referência direta sobre seu uso no desenvolvimento de jogos.

5.13.1. Onde se aplica:

Vários sites disponibilizam jogos on-line, executados diretamente do browser para os usuários que o acessam. Muitas vezes, algumas funções jogo não funcionam corretamente devido a algumas divergências no tratamento dos browsers com os comandos de script utilizado no jogo. Uma ilustração do diagrama de classes que favorece esta situação pode ser vista na Figura 5-32.

Um exemplo disso acontece no jogo *Marvel Avengers Alliance*, mostrado na Figura 5-31. Nele, o jogador pode incluir amigos de uma rede social, que também jogam, com o intuito de trocar itens do jogo, melhorando assim os personagens e o desempenho no mesmo, porém, testes realizados durante a pesquisa no mês de janeiro e fevereiro de 2013 concluíram que essas requisições de novos amigos não funcionam corretamente no browser *Chrome*, versão 24.0.1312.57 m, a mais atualizada na época, devido a inconsistências entre o jogo e a referida versão do browser.

Figura 5-31: Jogo on-line *Marvel Avengers Alliance*. Ele executa diretamente no browser.

5.13.2. Solução proposta pelo padrão:

Uma solução para este problema é proposta pelo padrão *Chain of Responsibility*. O padrão propõe uma interface em comum para os receptores da requisição (browsers) e cada objeto que representa um browser irá tratar a requisição da maneira mais correta para ele. Esta ação desacopla os objetos que representam o browser do motor do jogo, facilitando a atualização das classes que representam o browser, quando os browsers forem atualizados, como também a inclusão de novos browsers aceitos pelo jogo, sem a necessidade de mexer no motor ou em qualquer outra classe do sistema. O diagrama de classes destra situação pode ser vista na Figura 5-33.

5.13.3. Diagramas:

→ Sem o uso do padrão

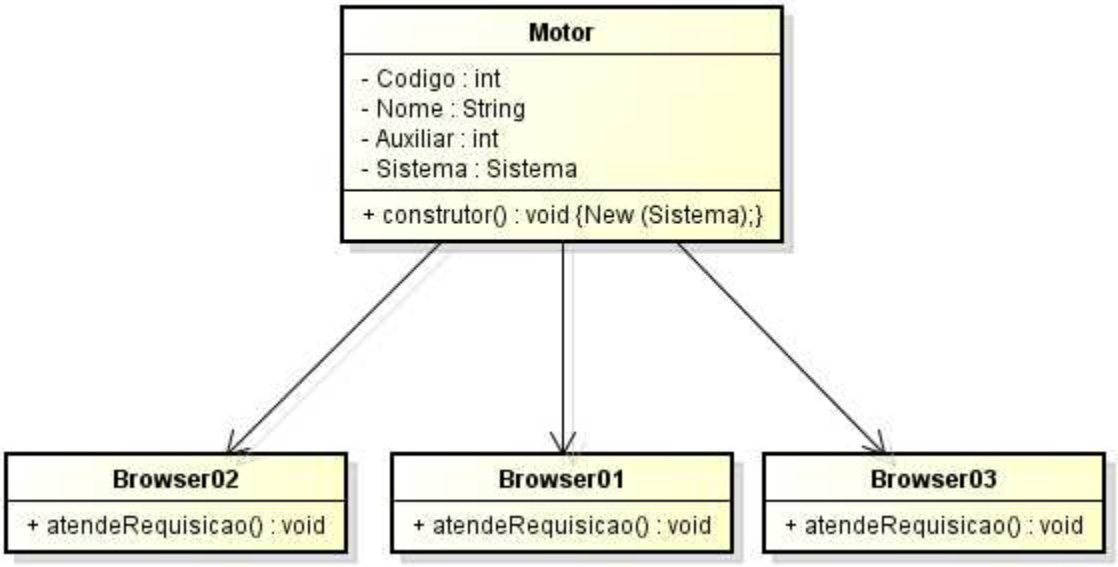

Figura 5-32: Diagrama parcial de um jogo sem o uso do padrão *Chain of Responsibility*. O motor envia diretamente as requisições para cada browser e os browsers a interpretam como quiserem.

→ Com o uso do padrão

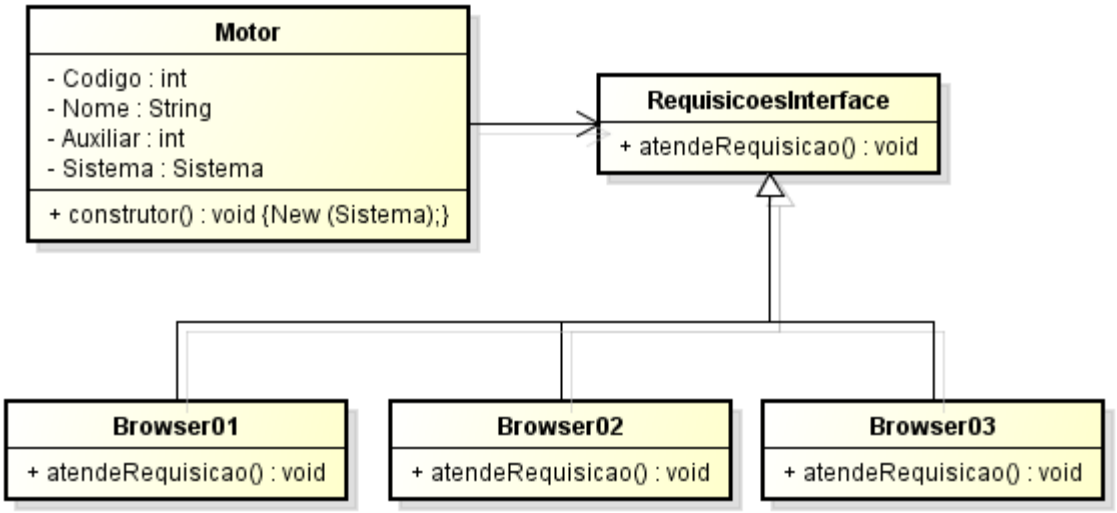

Figura 5-33: Diagrama parcial de um jogo com o uso do padrão *Chain of Responsibility*. O motor envia as requisições para a interface padrão que recebe cada requisição e o objeto que representa o browser a converte corretamente, desacoplando assim os browsers ao motor do jogo.

5.14. Command

O padrão *Command* é um padrão comportamental que encapsula uma requisição como objeto parametrizado, com o objetivo de atender clientes com diferentes requisições e dar suporte a ações reversíveis (GAMMA, HELM, *et al.*, 2000).

O padrão *Command* não possui qualquer referência direta sobre seu uso no desenvolvimento de jogos.

5.14.1. Onde se aplica:

Em jogos de estratégia de tempo real, como no jogo *Age of Mitology* mostrado na Figura 5-34, o jogador precisa criar e evoluir um exército e mandar este exército para a batalha contra exércitos inimigos. Nestas batalhas, muitas estratégias podem ser utilizadas e cada grupo de soldados, escolhidos pelo jogador, tem um papel na batalha.

Um problema acontece no momento de dar a ordem aos soldados de entrar em combate, pois cada soldado é um objeto, que possui o método “atacar”, e uma quantidade excessiva de comandos podem precisar de uma máquina mais eficiente, pois a execução do método “atacar” de cada um dos objetos “soldado” será executada em paralelo, além

disso, alguns soldados podem receber novas instruções e precisam abandonar o comando anterior, portanto, um método de cancelamento de comandos deve estar presente. Um diagrama desta situação é mostrado na Figura 5-35. O motor do jogo envia os comandos ao soldado e ele os executa em cada instância do objeto que recebeu aquela solicitação.

Figura 5-34: Tela do jogo *Age of Mitology*, onde o jogador cria e evolui um exército para conquistar um objetivo ou destruir outros exércitos.

5.14.2. Solução proposta pelo padrão:

Uma solução para este problema pode ser encontrada no padrão *Command*. Os comandos que podem ser executados pelos objetos “Soldado” são encapsulados na classe “Comandos”, que gera uma interface para a classe “ComandosConcretos” que, por sua vez, envia a solicitação para os objetos “Soldado” e estes as executa. Quando o usuário requisita uma ação, o objeto responsável pela ação solicita ao *Command* a sua execução e o objeto “ComandosContretos” aciona o método que ativa as ações. Um diagrama com a aplicação do padrão *Command* pode ser vista na Figura 5-36.

O objeto “ComandosContretos” guarda os comandos realizados, portanto, estes podem ser cancelados ou desfeitos, caso o motor os permita fazê-los. O padrão *Command* também pode ser utilizado para auxiliar a criação de *Checkpoints* ou salvamento da progressão do jogo para continuação posterior.

5.14.3. Diagramas:

→ Sem o uso do padrão

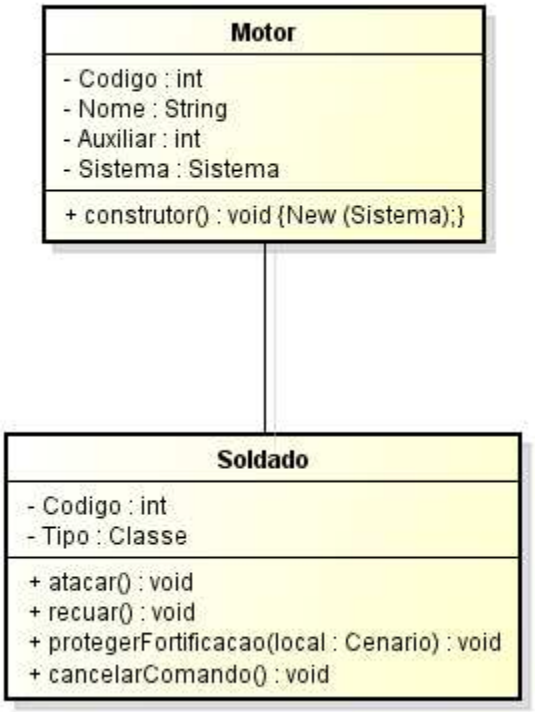

Figura 5-35: Diagrama parcial de um jogo sem o ouso do padrão *Command*. O motor do jogo emite um comando a ser executado pelo soltado, através de seu método e o mesmo executa.

→ Com o uso do padrão

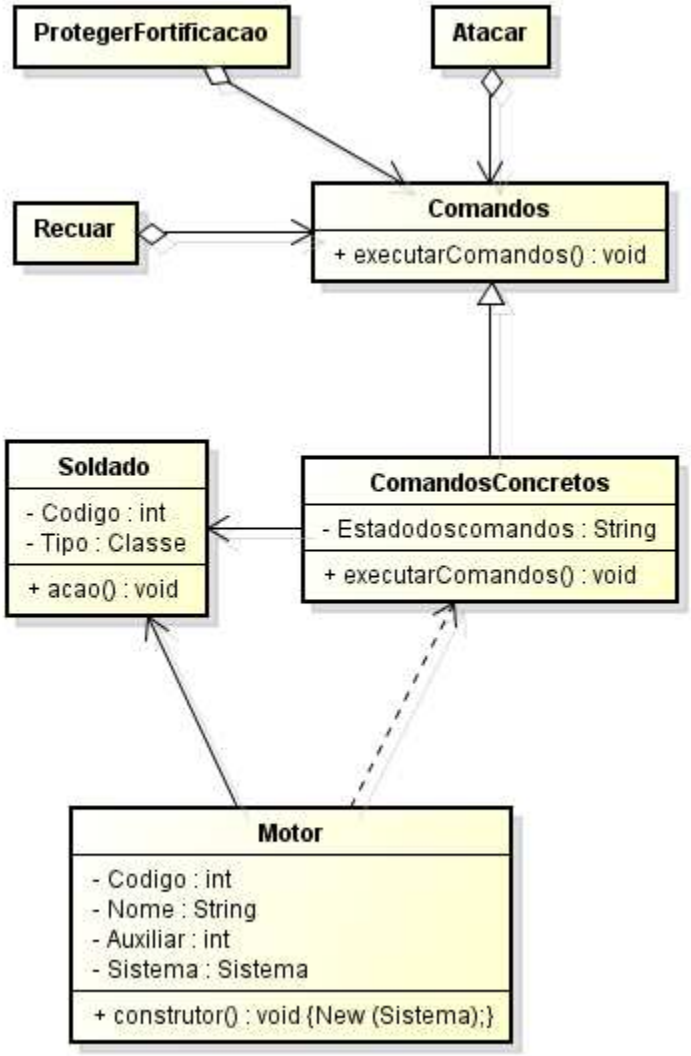

Figura 5-36: Diagrama parcial de um jogo com o uso do padrão *Command*. Os comandos são encapsulados em um objeto que aciona a ação na classe soldado.

5.15. Interpreter

O padrão *Interpreter* é um padrão comportamental que dada uma linguagem, definir uma gramática junto com um interpretador (GAMMA, HELM, *et al.*, 2000).

O padrão *Interpreter* não possui qualquer referência direta sobre seu uso no desenvolvimento de jogos.

Este padrão é amplamente utilizado no desenvolvimento de compiladores e de rotinas que buscam padrões em sequência de Strings. Este padrão também pode ser visto sendo

aplicado a problemas recorrentes em conjuntos estritamente especificados, onde pode ser definida uma linguagem que define esse conjunto.

A pesquisa realizada não mostrou nenhuma aplicação prática direta em jogos, porém, é possível utilizar o *Interpreter* para resolver problemas de raciocínio lógico, que podem ser apresentados como jogos, como o problema da "divisão do vinho" (dividir oito litros de vinho em duas porções de quatro litros, tendo somente a garrafa de oito litros cheia, e duas garrafas vazias, uma de cinco litros e outra de três litros todas sem marcações). Uma ideia para seu uso em jogos pode ser a implementação de um jogo do tipo "Perguntas e Respostas", onde as respostas seriam discursivas e não objetivas, porém, a grande complexidade na verificação das respostas, ainda mantém esta ideia no campo da pesquisa. Em reportagem datada de 30 de abril de 2012 para o site "Tecmundo", o jornalista Wikerson Landim afirma que o MIT está testando uma aplicação capaz de efetuar correções em questões abertas, mas não faz nenhuma referência a jogos eletrônicos digitais.

5.16. Iterator

O padrão *Iterator* é um padrão comportamental que permite executar um acesso sequencial a elementos de um objeto, sem expor sua representação interna (GAMMA, HELM, *et al.*, 2000).

O padrão *Iterator* não possui qualquer referência direta sobre seu uso no desenvolvimento de jogos.

5.16.1. Onde se aplica:

Em jogos de RPGs clássicos, baseado em turnos, como o jogo "Os Federa 4 e a Máquina do Tempo", ilustrado na Figura 5-37, os personagem vão evoluindo de maneira linear durante a progressão do jogo. Uma evolução é chamada de "avanço de nível" e a cada nível avançado, novos atributos são incorporados aos personagens, dependendo de sua classe. Esses atributos são extraídos de uma lista que armazena, em ordem crescente de nível, os atributos que cada classe de personagem deve receber. O problema consiste em ler da lista esses atributos, com o objetivo de incluir ao personagem durante a progressão do jogo.

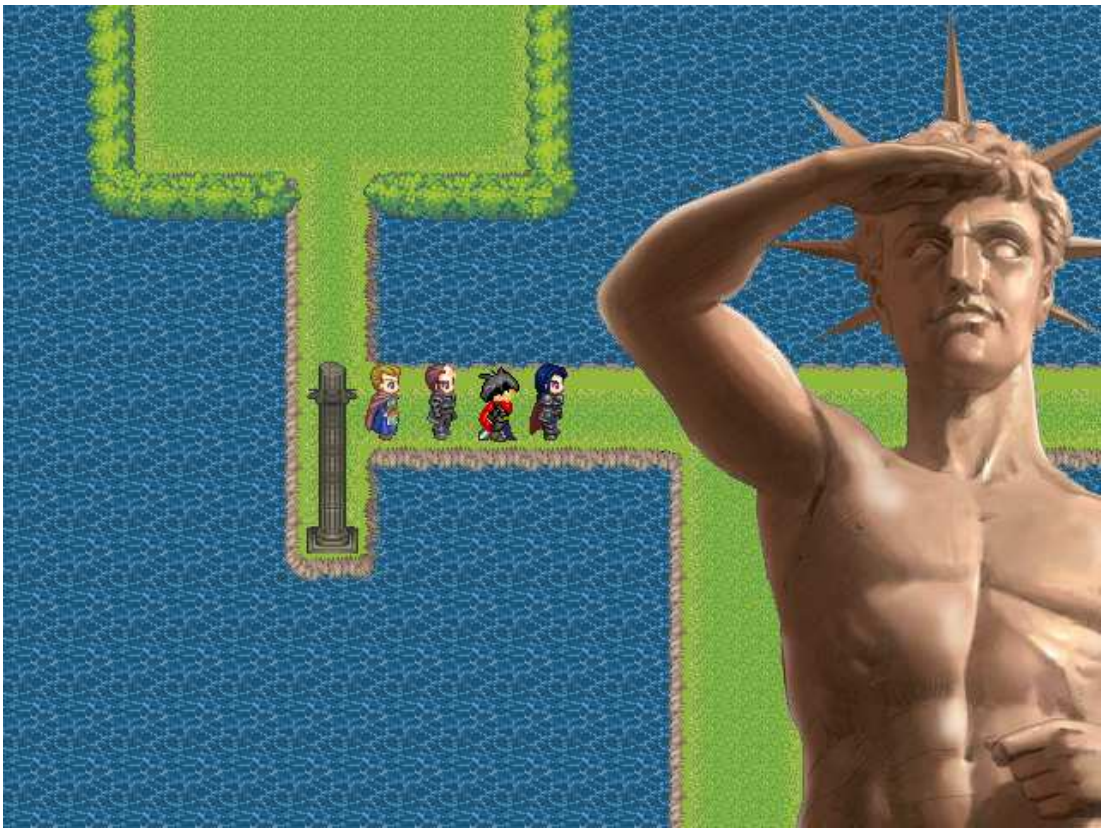

Figura 5-37: Tela do jogo de RPG baseado em turnos, Os Federa 4 e a Máquina do Tempo.

5.16.2. Solução proposta pelo padrão:

Existem diversas formas de efetuar a operação de busca sequencial de informações, uma delas é ilustrada na Figura 5-38 onde quaisquer alterações na lista ou em algum atributo acarretarão em atualização na forma de busca e até no motor se este depender da forma que receberá os resultados.

Uma forma bastante eficiente é apresentada pelo padrão *Iterator*. Uma vantagem de se utilizar o padrão nesta pesquisa, é que a classe motor não precisa saber como a lista de itens está implementada, além de poder existir diferentes formas de varredura, sem necessidade de mexer em qualquer outra classe do jogo que não seja a classe *Iterator*. O diagrama ilustrando o uso do padrão é mostrado na Figura 5-39.

A classe “Atributos_Iterator” define uma interface para acessar e percorrer os elementos da lista de atributos e a classe “Atributos_IteratorConcreto” implementa esta interface e mantém a informação de qual atributo do personagem está sendo acessado naquele nível. A classe “Lista_AtributosConcretos” retorna uma instância do objeto “Atributos_IteratorConcreto” quando é solicitada o início de uma busca.

5.16.3. Diagramas:

→ Sem o uso do padrão

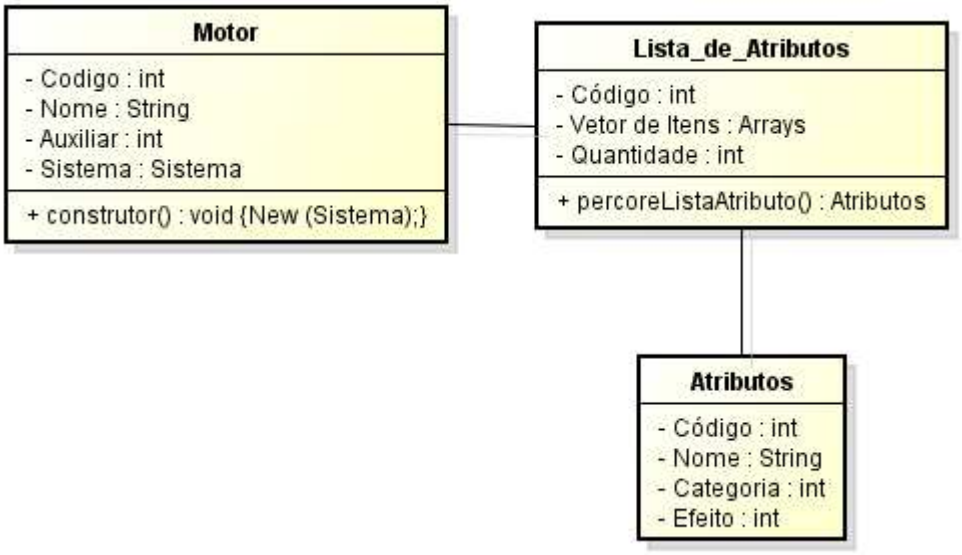

Figura 5-38: Diagrama parcial de um jogo em sem o uso do padrão *Iterator*. O objeto que guarda a lista de itens possui um método para fazer a busca. Qualquer alteração na lista ou em algum atributo acarretará em atualização na forma de busca.

→ Com o uso do padrão

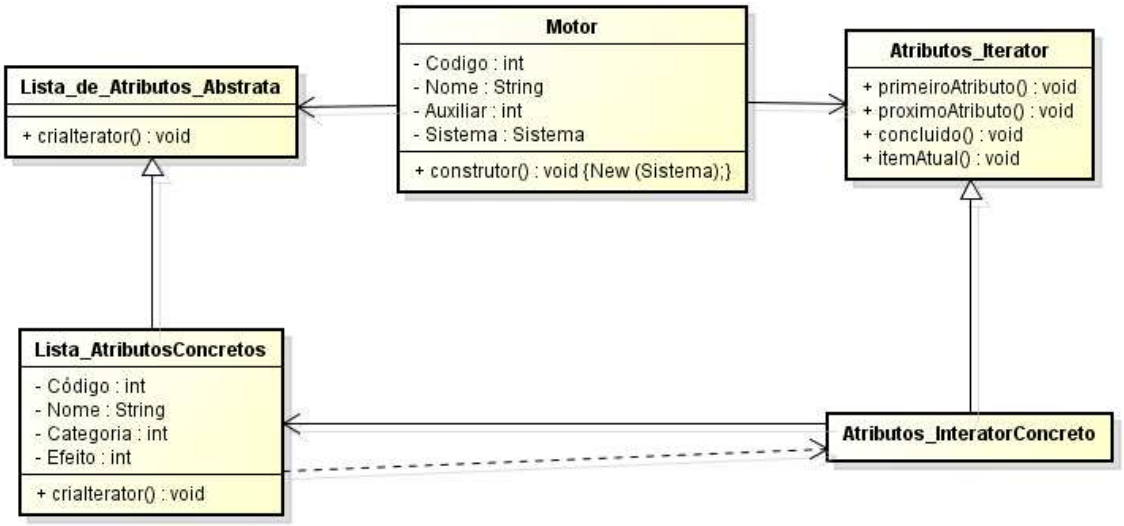

Figura 5-39: Diagrama com o uso do padrão *Iterator*. A lista de atributos e a busca estão desacopladas entre si e também com o motor de controle do jogo.

5.17. Mediator

O padrão *Mediator* é um padrão comportamental que define um objeto que encapsula a forma de interação de um conjunto de objetos (GAMMA, HELM, *et al.*, 2000).

O padrão *Mediator* não possui qualquer referência direta sobre seu uso no desenvolvimento de jogos.

5.17.1. Onde se aplica:

Em diversos tipos de jogos, como RPG e LUTA, cada personagem é um objetos e nos campos de batalha esses objetos se comunicam e trocam diversas informações entre si. A quantidade de objetos é grande e essa comunicação pode gerar um forte acoplamento entre as classes destes objetos.

Um exemplo desta comunicação pode ser visto na tela do jogo *Mortal Kombat III Ultimate*, mostrada na Figura 5-40, o personagem *Reptile* aplica um chute no personagem *Smoke*, neste momento deve ser calculada a colisão e quantos pontos de vida o *Smoke* perdeu, para isso é necessário o valor da força do chute (fraco, médio ou forte) e da defesa do *Smoke*, então é feito o cálculo e os pontos de vida são retirados. Se esses cálculos forem feitos pelos objetos personagens, conforme ilustra o diagrama da Figura 5-41, tem-se um forte acoplamento entre eles e, com o aumento na quantidade de personagens e de formas diferentes de interação entre eles, a complexidade do código será inevitável.

Figura 5-40: Telado jogo de luta *Mortal Kombat III Ultimate.* Na imagem, o ninja verde *Reptile* aplica um chute no oponente *Smoke*.

5.17.2. Solução proposta pelo padrão:

Uma solução para reduzir a complexidade e permitir um fraco acoplamento entre os personagens é o uso do padrão *Mediator*. Conforme ilustrado no diagrama da Figura 5-42, o padrão sugere uma classe que define uma interface de comunicação entre os objetos lutadores. A classe concreta do *Mediator* implementa toda a comunicação entre os personagens, deixando as classes dos personagens apenas com as informações particulares de cada um dele. A classe “Lutador” conhece o *Mediator* e envia requisições sempre que houver comunicação entre seus herdeiros.

5.17.3. Diagramas:

→ Sem o uso do padrão

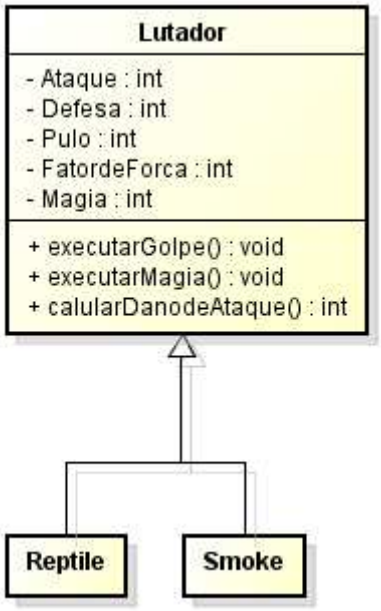

Figura 5-41: Diagrama parcial sem o uso do padrão *Mediator*. A classe principal implementa toda a comunicação entre os lutadores, colisão e força dos golpes e todos os lutadores podem sobrescrever os métodos deixando um forte acoplamento entre as classes de lutadores.

→ Com o uso do padrão

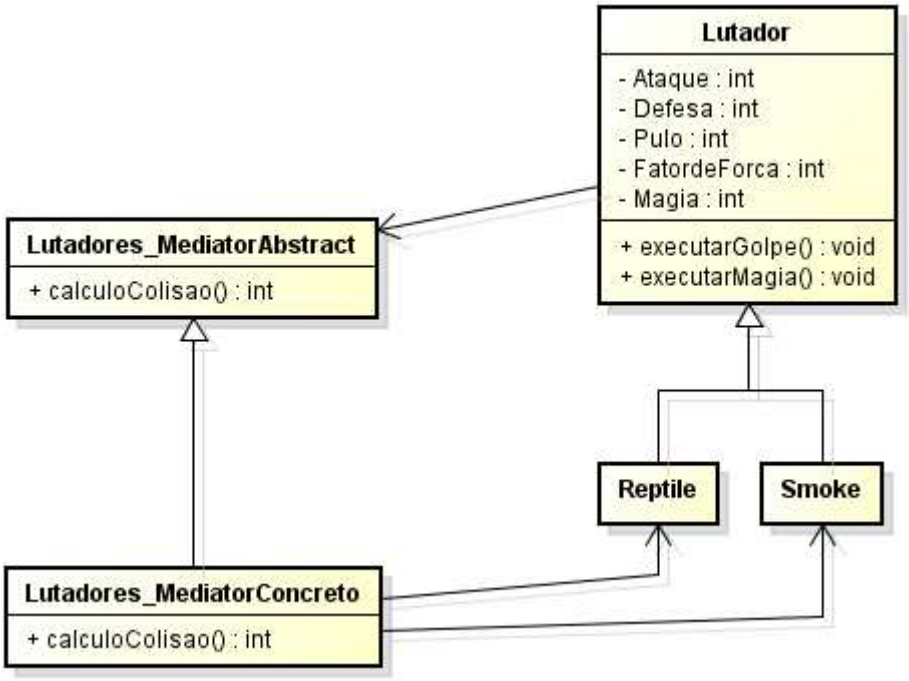

Figura 5-42: Diagrama com o uso do padrão *Mediator*. Os cálculos de colisão e força de ataque estão perfeitamente desacoplados dos lutadores, diminuindo a complexidade dos mesmos e isolando as comunicações entre os objetos de forma a facilitar a inserção de novos ataques. Os objetos lutadores não precisam se preocupar com os demais.

5.18. Memento

O padrão *Memento* é um padrão comportamental que externaliza o estado interno de um objeto, de maneira que o objeto original fica preservado de possíveis alterações (GAMMA, HELM, *et al.*, 2000).

O padrão *Memento* não possui qualquer referência direta sobre seu uso no desenvolvimento de jogos.

5.18.1. Onde se aplica:

É comum em diversos jogos a criação de *Checkpoints* no meio das fases. Um *checkpoint* é um ponto de retorno, caso o personagem perca todos os seus pontos de vida e já tenha passado por um *checkpoint*, a vida seguinte não precisará reiniciar a partida do início da fase e sim do *checkpoint*. É uma forma de garantir que o jogador não irá se aborrecer repetindo toda a fase se perder seus pontos de vida praticamente no final dela. Um exemplo de *checkpoint* pode ser visto no jogo *Super Mario Bros* ilustrado na Figura 5-43.

O problema consiste em criar internamente esse *checkpoint*. Para criar um ponto de retorno é necessário armazenar o estado interno de todos os objetos modificados até aquele momento do jogo, somente assim, o jogo pode ser restaurado da maneira correta do ponto onde parou. Segundo GAMMA (et al., 2000), expor o estado interno dos objetos viola o encapsulamento e prejudica a confiabilidade e extensibilidade geral do jogo. O diagrama com esta situação pode ser visto na Figura 5-44.

Figura 5-43: Tela do jogo *Super Mario Bros*, nela vemos o personagens e um par de traves, com uma ligação. Quando o Mario passa pelas traves o jogo cria um Checkpoint.

5.18.2. Solução proposta pelo padrão:

Uma solução para este problema é proposta pelo padrão Memento, com ele é possível guardar o estado interno do personagem sem expor sua implementação. O diagrama com a aplicação do padrão pode ser vista na Figura 5-45.

O objeto *Checkpoint* requisita um *memento* ao objeto Personagem. O personagem cria um memento e armazena neste *memento* seu estado interno (itens, pontos de vida, entre outros) e repassa o *memento* ao solicitante. Quando o personagem perde a vida, o objeto *Checkpoint* seta o *memento* no objeto Personagem, fazendo com que ele recupere o estado que possuía no momento em que passou pelo elemento de *checkpoint* na fase. O diagrama ilustrando esta operação pode ser visto na Figura 5-45.

5.18.3. Diagramas:

→ Sem o uso do padrão

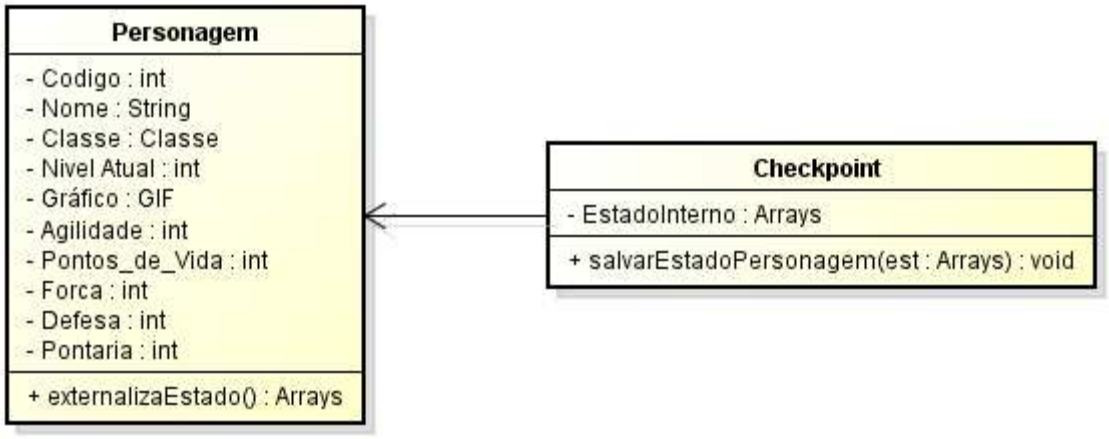

Figura 5-44: Diagrama parcial de um jogo sem o uso do padrão *Memento* na criação de um Checkpoint. O personagem precisa externalizar seu estado interno, o que viola o encapsulamento.

→ Com o uso do padrão

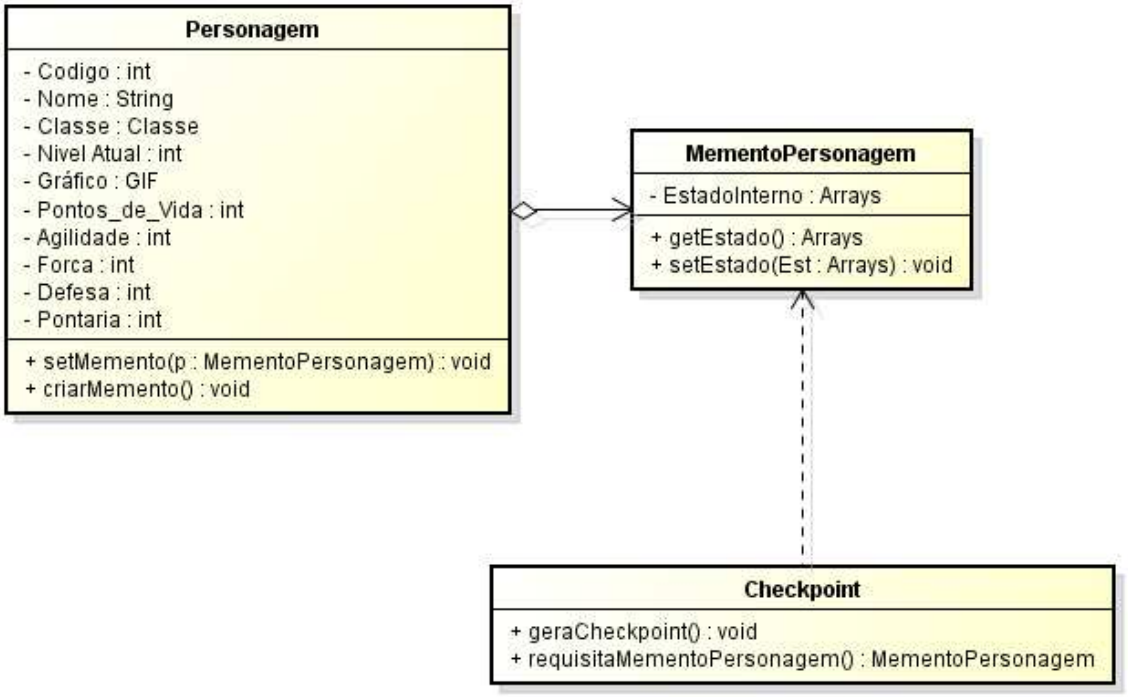

Figura 5-45: Diagrama com o uso do padrão Memento na criação de um checkpoint. O personagem não precisa expor detalhes de sua implementação e o encapsulamento está garantido.

5.19. Observer

O padrão *Observer* é um padrão comportamental que define uma dependência 1-N entre um objeto e seus N dependentes de maneira que, quando o objeto principal mudar de estado, todos os seus dependentes são avisados e atualizados automaticamente (GAMMA, HELM, *et al.*, 2000).

O padrão *Observer* é citado pelos autores Trindade e Fischer (2008), sem referência direta de sua aplicação em jogos; Gestwicki (2007), mas sem exemplos práticos; Gestwicki e Sun (2008), que descreve bem o padrão, mas com foco voltado no aprendizado dos padrões para alunos de graduação; Ampatzoglou, Gortzis (et al., 2011), que apenas referencia *bugs* e finalmente Ampatzoglou e Chatzigeorgiou (2006) que faz uma boa descrição da aplicação do padrão em jogos e usa o jogo de “Futebol” como exemplo de aplicação. Esta dissertação também mostra a aplicação do padrão em jogos, mas utiliza um jogo de “Tiro” como exemplo da aplicação.

5.19.1. Onde se aplica:

Na maioria dos jogos atuais, os agentes inteligentes são largamente utilizados (PERUCIA, BERTHÊM, *et al.*, 2005). Esses agentes geralmente são os inimigos do personagem jogador, onde a Inteligência Artificial é aplicada. A estratégia de ataque dos inimigos é calculada com base nos movimentos do personagem de maneira a evitar seus

ataques e cercá-lo por todos os lados. Um exemplo disso acontece no jogo *Resident Evil 5*, ilustrado na Figura 5-46, onde os inimigos, que são zumbis, atacam em bandos, cercam o personagem e tentam evitar seus ataques quando possível.

O problema acontece quando os objetos dos inimigos precisam receber as informações de movimentação e ataque do jogador, pois qualquer alteração no personagem precisa ser reportada, além disso, a cada mapa, a quantidade de zumbis varia, portanto, não se tem de maneira simples a quantidade de objetos que precisam ser notificados. Obviamente a comunicação não pode afetar a consistência dos objetos comunicantes. Uma forma de criar as comunicações, sem o uso de nenhum padrão é ilustrada na Figura 5-47. Uma rotina é implantada no personagem com uma referência para cada zumbi, repassando um a um as informações necessárias.

Figura 5-46: Tela do jogo *Resident Evil 5*. Os inimigos atacam de acordo com as ações do personagem.

5.19.2. Solução proposta pelo padrão:

Uma solução para efetivar essa comunicação, com fraco acoplamento é a aplicação do padrão *Observer*. A classe do personagem conhece seus observadores a partir de uma lista e possui métodos para incluí-los ou excluí-los, pois estes podem variar com o andamento do jogo. A classe “ObserverZumbi” define uma interface para os objetos que precisam ser notificados. A classe “PersonagemConcreto” armazena as informações interessantes para os zumbis e os notifica sempre que uma mudança ocorrer. Depois de

informado sobre uma mudança, os zumbis consultam o personagem e utilizam a informação obtida para ajustar sua estratégia de ação. O diagrama da aplicação do padrão pode ser vista na Figura 5-48.

5.19.3. Diagramas:

→ Sem o uso do padrão

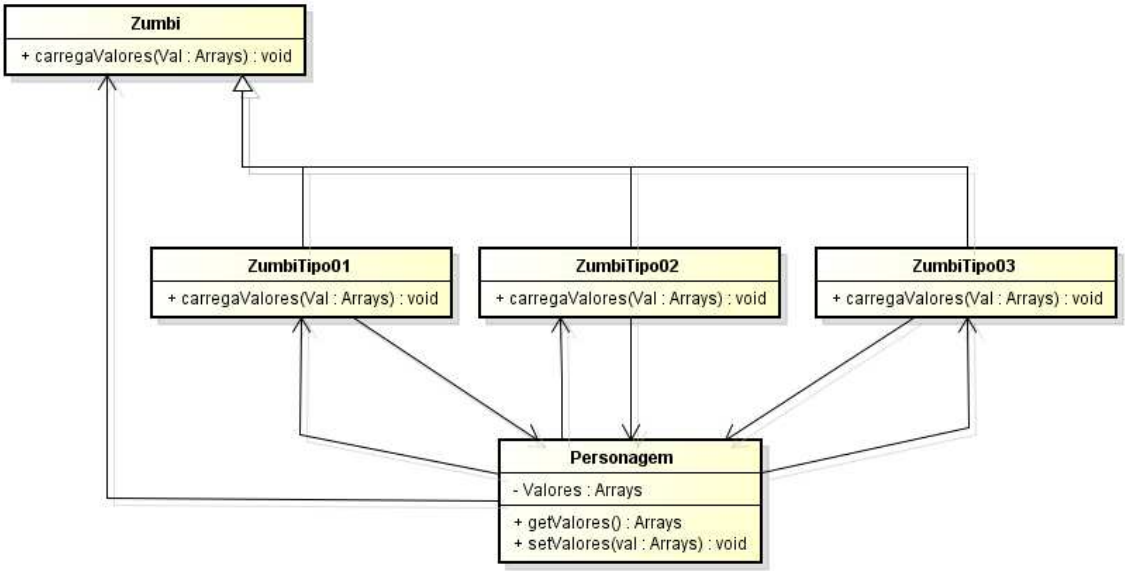

Figura 5-47: Diagrama parcial sem o uso do padrão *Observer*. Verifica-se um forte acoplamento entre a classe do personagem e as classes que representam os zumbis.

→ Com o uso do padrão

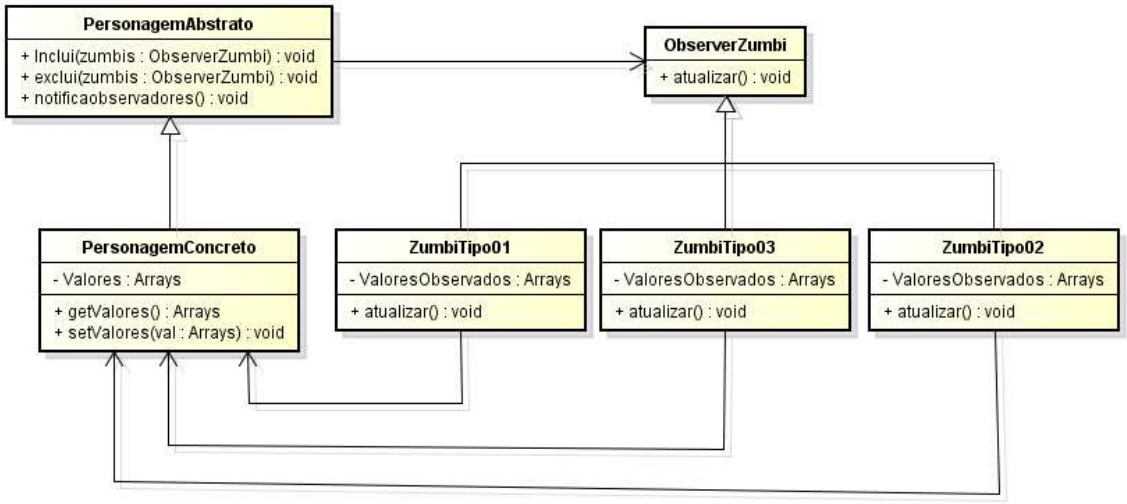

Figura 5-48: Diagrama parcial com o uso do padrão *Observer*. Os objetos dos zumbis são notificados a cada movimentação do personagem, sem o forte acoplamento proposto pelas soluções sem o uso do padrão.

5.20. State

O padrão *State* é um padrão comportamental que permite a um objeto alterar o seu comportamento dinamicamente, quanto o seu estado interno mudar (GAMMA, HELM, *et al.*, 2000).

O padrão *State* é citado pelos autores Gestwicki (2007), mas sem exemplos práticos; Wong e Nguyen (2002), de maneira muito sucinta; Gestwicki e Sun (2008), que descreve bem o padrão, mas com foco voltado no aprendizado dos padrões para alunos de graduação; Ampatzoglou, Gortzis (et al., 2011), que apenas referencia *bugs* e finalmente Ampatzoglou e Chatzigeorgiou (2006) que faz uma boa descrição da aplicação do padrão em jogos e usa o jogo de "Tiro" em primeira pessoa, como exemplo de aplicação. Esta dissertação também mostra a aplicação do padrão em jogos, mas utiliza um jogo de "Aventura" como exemplo da aplicação.

5.20.1. Onde se aplica:

Em um jogo qualquer, os eventos que podem ocorrer com um personagem, vão depender de determinadas características transitórias dos mesmos em um determinado instante.

Um exemplo disto acontece no jogo *Sonic The Hedgehog*, onde o evento do personagem principal esbarrar em um inimigo poderá ter diferentes consequências de acordo com sua característica atual. A Figura 5-49 ilustra diferentes estados do personagem Sonic. Na Figura 5-49 (a) temos seu estado normal sem argolas, ao esbarrar em um inimigo neste estado o personagem perderá uma vida e retornará ao início da fase, caso ainda tenha mais vidas. Na Figura 5-49 (b) temos o personagem com um escudo, ao esbarrar em um inimigo neste estado o personagem simplesmente perderá o escudo, dará um pulo pra trás e o jogo prossegue normalmente. A Figura 5-49 (c) mostra o personagem com a chamada invencibilidade temporária, ao esbarrar em um inimigo neste estado, o inimigo morre e nenhuma consequência será transmitida ao personagem.

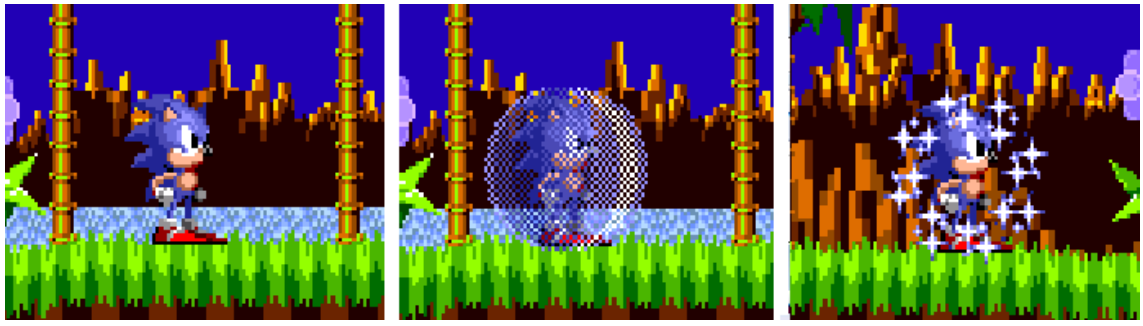

Figura 5-49: (a) Sonic normal (b) Sonic com escudo (c) Sonic com invencibilidade temporária

Outro exemplo pode ser visto em jogos *Marvel Avengers Alliance*. Ao receberem ataques de inimigos ou ajuda de aliados o personagem pode ter ganhos ou perdas temporárias dependendo do que lhe foi dado.

No caso de ganhos, por exemplo, o personagem pode ficar “Resiste e Queimaduras”, estado onde o personagem não pode sofrer queimaduras; “Resistente a Venenos”, estado onde o personagem não pode sofrer envenenamentos; “Resistente a Sangramentos”, estado onde o personagem não sofre danos por conta de sangramentos; “Agil”, estado onde o personagem tem sua capacidade de fuga aumentada em 25%; “Fortalecido” estado onde o personagem tem sua força aumentada em 25%; “Protegido”, estado onde o personagem tem sua defesa aumentada em 25% e “Focado”, estado onde o personagem tem sua pontaria aumentada em 25%.

No caso de perdas, o personagem pode ficar “Queimado”, estado que o faz sofrer danos a cada turno de luta e tem sua defesa reduzida; “Envenenado”, estado onde o personagem sofre danos a cada turno e tem seu ataque reduzido; “Sangrando”, estado onde o personagem sofre danos a cada turno e a cada ataque efetuado; “Lento”, estado onde o personagem tem sua capacidade de fuga reduzida em 25%; “Enfraquecido” estado onde o personagem tem sua força reduzida em 25%; “Exposto”, estado onde o personagem tem sua defesa reduzida em 25% e “Tonto”, estado onde o personagem tem sua pontaria diminuída em 25%.

5.20.2. Solução proposta pelo padrão:

Alterações e reações feitas em objetos personagens e eventos tornam-se independentes do estado que o objeto se encontra. O Objeto *State* se encarrega de avaliar o estado do objeto e fazer as devidas alterações e reações. Na Figura 5-50 podemos ver o diagrama parcial do jogo *Marvel Avengers Alliance* sem aplicação do padrão *State*. É possível perceber que os estados do personagem são atributos e métodos da classe Personagem, causando um forte acoplamento entre os estados e as ações individuais do personagem, deixando a classe mais pesada e com operadores que não serão utilizados constantemente, visto que o personagem não estará sempre queimado, exposto, tonto, enfraquecido, envenenado, sangrando e lendo simultaneamente. Na Figura 5-51 temos o diagrama utilizando o padrão *State*, nota-se agora que os estados do personagem estão desacoplados da classe principal, permitindo que estados possam ser removidos ou incluídos sem qualquer alteração na classe Personagem e cujas ações podem ser modificadas sem qualquer preocupação com o resto do sistema.

5.20.3. Diagramas:

→ Sem o uso do padrão

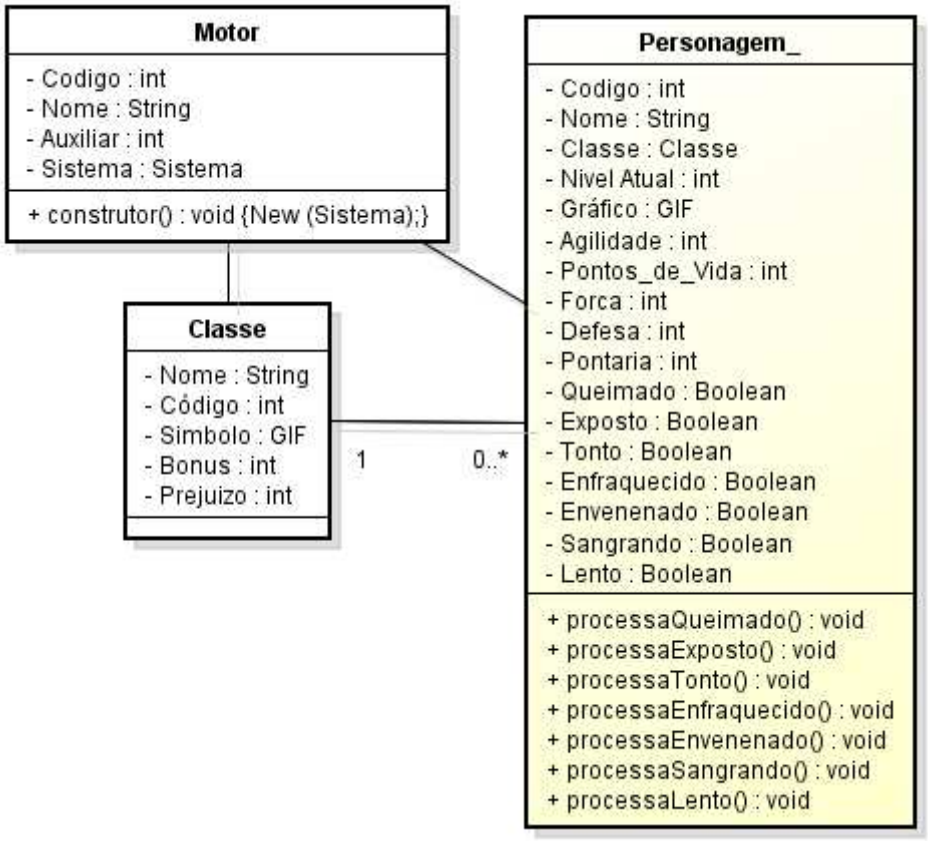

Figura 5-50. Diagrama parcial do jogo *Marvel Avengers Alliance*, sem o padrão *State*, em caso de perdas.

→ Com o uso do padrão

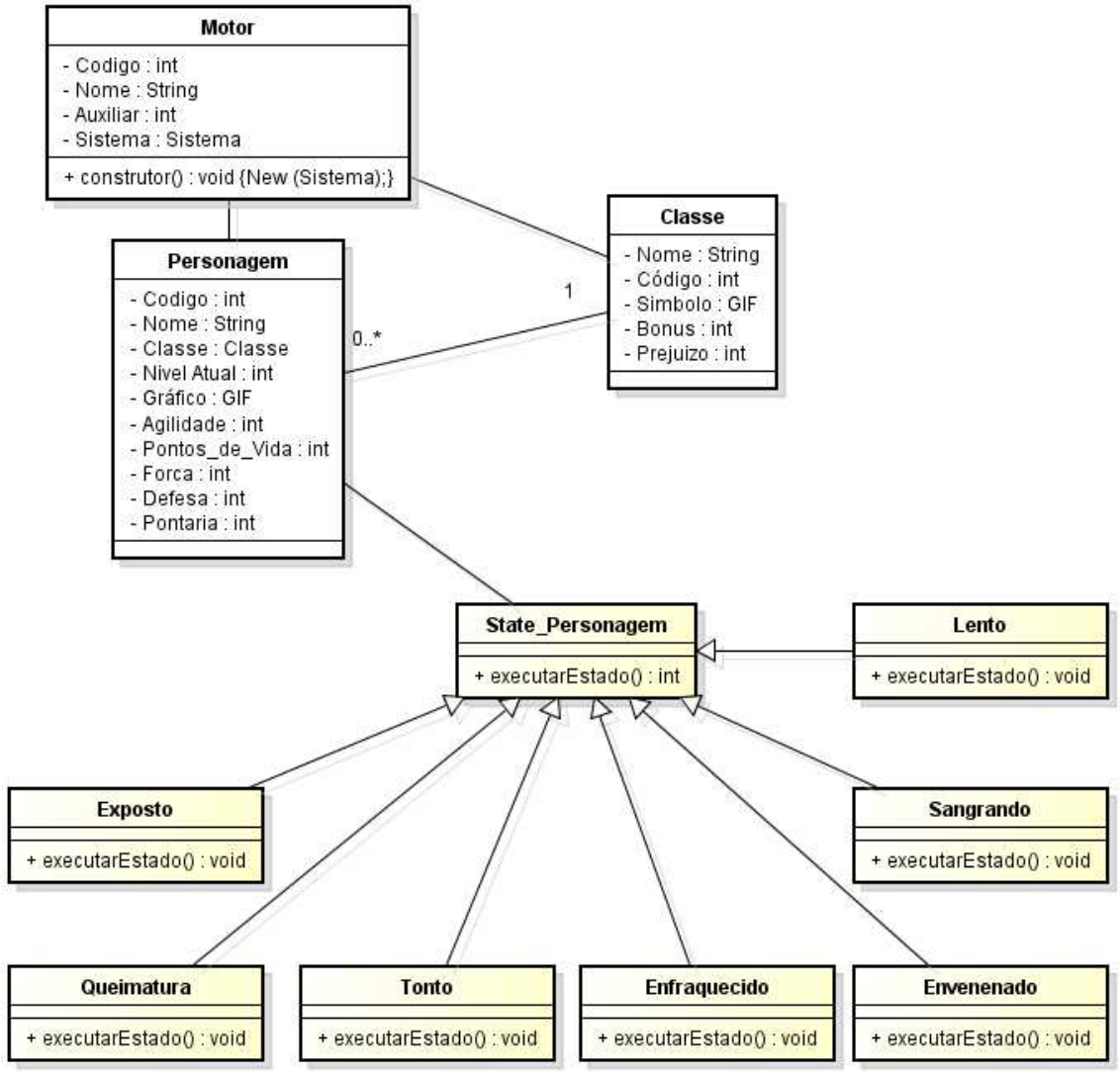

Figura 5-51. Diagrama parcial do jogo *Marvel Avengers Alliance*, com o padrão *State*, em caso de perdas

5.21. Strategy

O padrão *Strategy* é um padrão comportamental que defini um conjunto de algoritmos, encapsula cada um, e promove a comunicação entre eles, sem o acoplamento dos mesmos (GAMMA, HELM, *et al.*, 2000).

O padrão *Strategy* é citado pelos autores Gestwicki (2007), mas sem exemplos práticos; Wong e Nguyen (2002), de maneira muito sucinta; Gestwicki e Sun (2008), que descreve bem o padrão, mas com foco voltado no aprendizado dos padrões para alunos de graduação; Ampatzoglou, Gortzis (et al., 2011), que apenas referencia *bugs* e finalmente Ampatzoglou e Chatzigeorgiou (2006) que faz uma boa descrição da

aplicação do padrão em jogos e usa o jogo de “Xadrez” como exemplo de aplicação. Esta dissertação também mostra a aplicação do padrão em jogos, mas utiliza um jogo de “Ação” como exemplo da aplicação.

5.21.1. Onde se aplica:

É comum em jogos eletrônicos, a presença de diversos inimigos, cada um deles com uma característica de movimentação e ataque peculiar. Alguns inimigos são capazes de pular, outros de correr, enquanto outros podem voar. Alguns atacam com socos, outro com chutes e alguns combinam algumas destas características, um exemplo disto pode ser visto no jogo *Altered Beast*, onde alguns inimigos estão ilustrados na Figura 5-52.

A princípio pensa-se na implementação através de herança, conforme diagrama ilustrado na Figura 5-53, porém, se os inimigos herdam de uma classe pai, com todas as características que um inimigo pode ter, certamente ele herdará características que não são suas. Outra possibilidade é a criação de interfaces para cada uma das características e os inimigos implementarão as interfaces com as características que possuem, porém, quando a quantidade de características ou principalmente de inimigos começarem a passar de poucas dezenas, teremos um emaranhado de ligações e a manutenção destas implementações pode tornar bastante custosa e pouco produtiva.

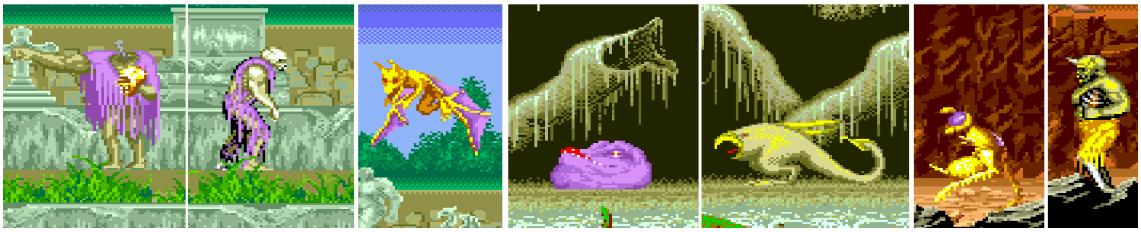

Figura 5-52: Imagens de inimigos do jogo *Altered Beast*. Cada inimigo possui uma característica peculiar.

5.21.2. Solução proposta pelo padrão:

Uma solução para este problema é sugerida pelo padrão *Strategy*. O padrão propõe que todas as características que podem variar entre os inimigos devam ser encapsuladas. Este encapsulamento deverá ser feito através de interfaces e a implementação destas interfaces serão feitas pelas classes concretas das características e não dos inimigos como discutido anteriormente. Desta forma, os inimigos, ao invés de herdar estas características, estão fazendo uma composição delas. Um diagrama ilustrando a aplicação do padrão pode ser visto na Figura 5-54.

A classe composta estará fracamente acoplada à outra e manutenções podem ser feitas sem interferências. É possível ainda, na composição, a utilização da delegação, onde o envolvimento de objetos tem o objetivo de atender uma determinada solicitação.

5.21.3. Diagramas:

→ Sem o uso do padrão

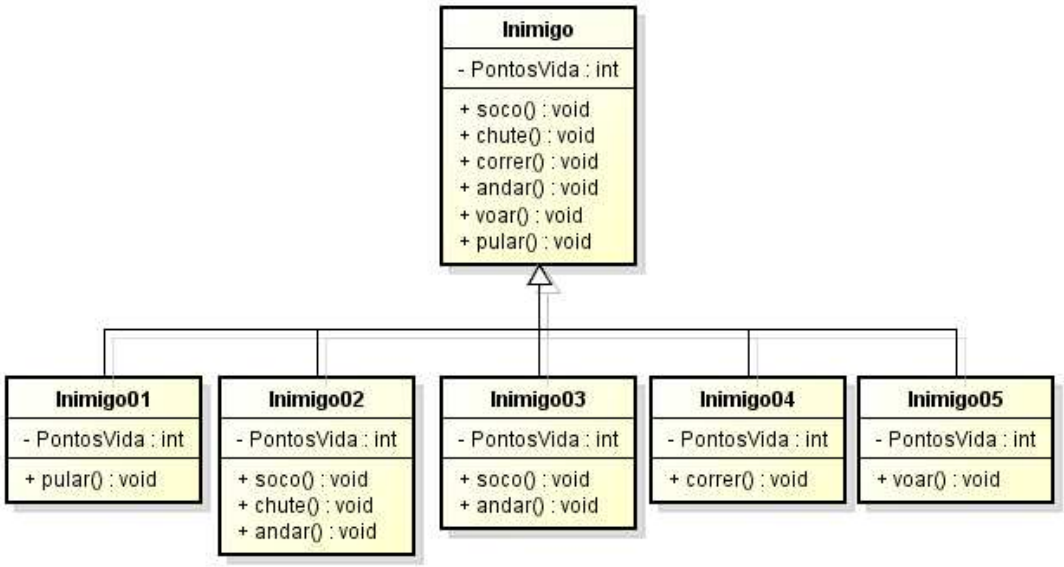

Figura 5-53: Diagrama parcial de um jogo sem o uso do padrão *Strategy*. A herança pode gerar duplicação de código ou métodos vazios, para que a classe filha não tenha a característica herdada que não deveria ter.

→ Com o uso do padrão

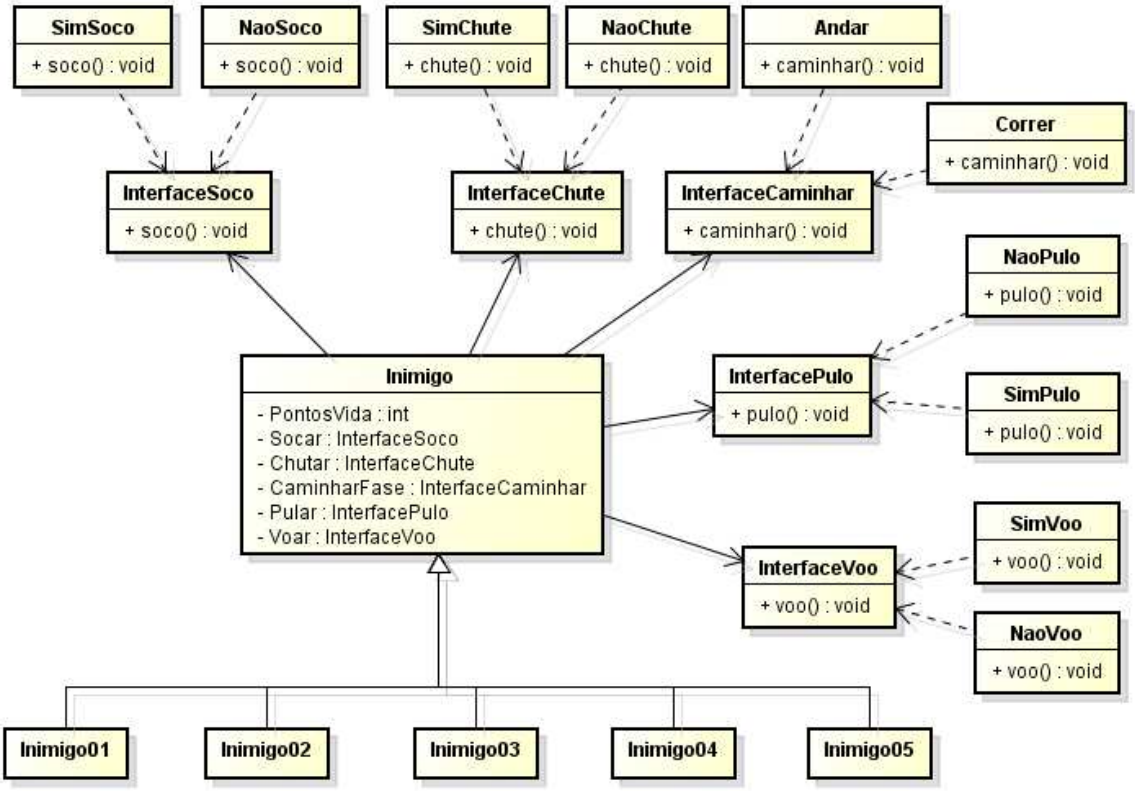

Figura 5-54: Diagrama parcial de um jogo com o uso do padrão *Strategy*. Os inimigos, além das características comuns, serão uma composição de características peculiares.

5.22. Template Method

O padrão *Template Method* é um padrão comportamental que definir o arcabouço de uma classe, deixando alguns atributos e métodos a serem preenchidos pelas subclasses (GAMMA, HELM, *et al.*, 2000).

O padrão *Template Method*, assim como o *Proxy*, o *Composite*, o *Prototype* e o *Abstract Factory*, é citado apenas no artigo Ampatzoglou, Gortzis (et al., 2011) que limita apenas em constatar a quantidade de *bugs* reportados sem e com o uso deste padrão no desenvolvimento de jogos. Neste trabalho é apresentado um exemplo de onde e como o padrão pode ser aplicado no desenvolvimento de jogos.

5.22.1. Onde se aplica:

Em uma grande variedade jogos, as fases são compostas por mapas e muitos destes mapas são bastante similares, mudando apenas algumas características, como disposição das fronteiras e de elementos visuais e localização de itens e inimigos. Um exemplo disso acontece no jogo "*ToeJam e Earl*", mostrado na Figura 5-55, o jogo possui vinte e cinco fases e todas elas possuem características bastante parecidas umas com as outras. A criação destas fases sem um bom planejamento pode gerar multiplicidade de códigos repetidos nas classes geradoras destes mapas. Esta situação é ilustrada na Figura 5-56.

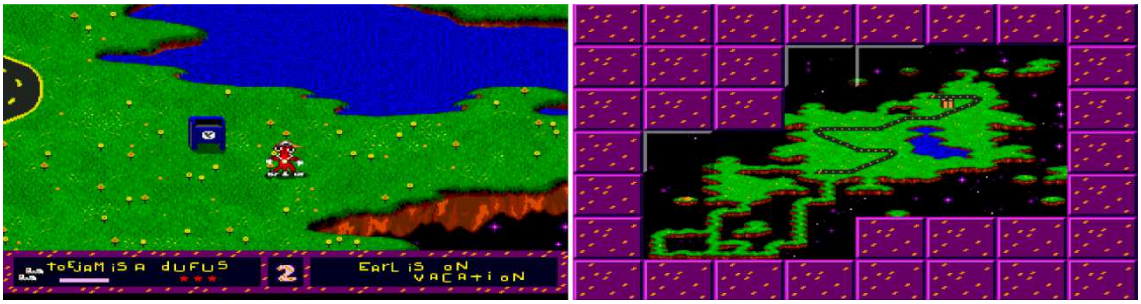

Figura 5-55: Tela do jogo *Toejam e Earl*. O jogo é composto de 25 fases, com mapas bem similares, com modificações apenas em seu formado e na disposição dos itens da fase.

5.22.2. Solução proposta pelo padrão:

A possibilidade de se criar mapas com geometrias, relevo e disposição de elementos diferentes, mas com muitas características similares, sem duplicação de código, é a proposição feita pelo padrão *Template Method*.

O padrão propõe a criação da classe abstrata "GeradorCenario", que declara operações primitivas a serem implementadas pelas sub-classes e implementa o *Template Method* que define o arcabouço dos algoritmos e invoca tanto as operações primitivas quando as

operações concretas da classe “GeradorCenario”. As classes que representam as fases implementam as operações primitivas executando os passos do algoritmo específicos daquela fase. As fases não precisam gerar as partes invariantes do algoritmo, pois isso é feito pela classe “GeradorCenario”.

5.22.3. Diagramas:

→ Sem o uso do padrão

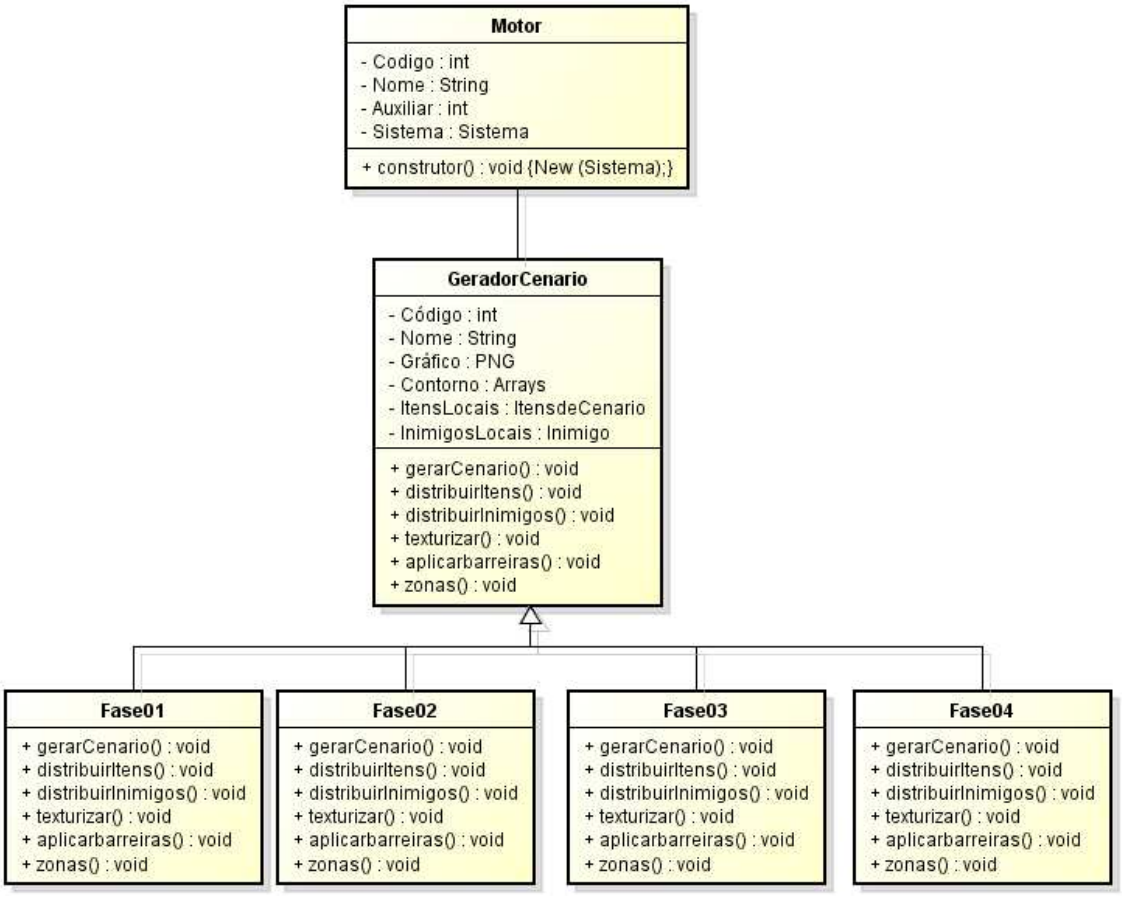

Figura 5-56: Diagrama parcial sem a aplicação do padrão *Template Method*. Códigos repetidos em fases diferentes com a mesma característica são inevitáveis.

→ Com o uso do padrão

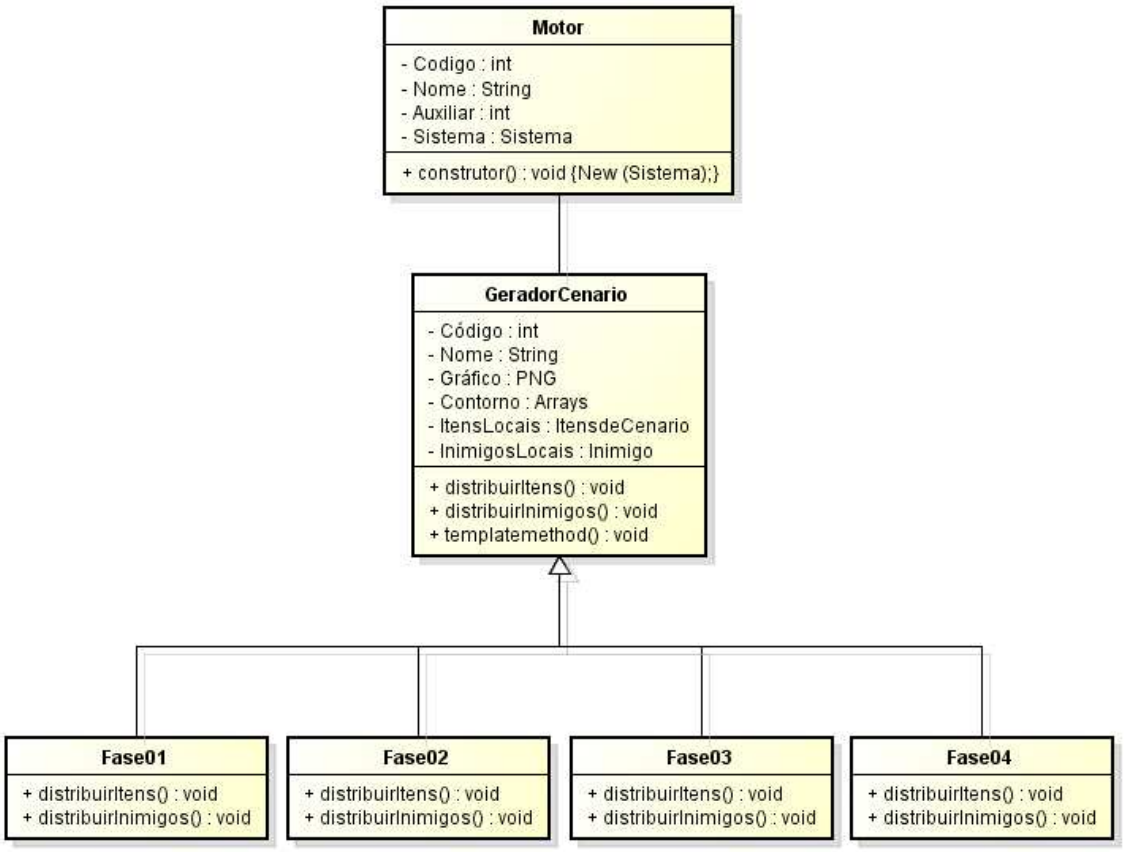

Figura 5-57: Diagrama com o uso do padrão *Template Method*. O método do padrão define um arcabouço onde só serão necessárias as implementações dos algoritmos específicos de cada sub-classe.

5.23. Visitor

O padrão *Visitor* é um padrão comportamental que gera a representação de uma operação a ser realizada sobre os elementos de um conjunto de objetos (GAMMA, HELM, *et al.*, 2000).

O padrão *Visitor* aparece nos artigos de Gestwicki (2007), mas sem exemplos práticos e em Gestwicki e Sun (2008), que descreve bem o padrão, mas com foco voltado no aprendizado dos padrões para alunos de graduação. Este trabalho tem o foco principal no desenvolvimento de jogos e apresenta um exemplo prático da aplicação do padrão, o que o diferencia dos demais.

5.23.1. Onde se aplica:

Um dos algoritmos mais importantes em qualquer jogo eletrônico é o chamado Algoritmo de Colisão.

> "O processo de detecção e tratamento de colisões em jogos consiste em verificar se, após a atualização de suas posições, os objetos da cena estão se sobrepondo e, caso isto aconteça, executar o tratamento que a situação exige. Este processo é realizado em duas etapas: a primeira etapa é a detecção da colisão e a segunda etapa é o tratamento da colisão."
>
> (PERUCIA, BERTHÊM, *et al.*, 2005)

Um grande problema na colisão é que o resultado da colisão depende dos elementos envolvidos nela. Um exemplo pode ser visto no jogo *Kid Chameleon*, mostrado na Figura 5-58. Nele, o jogador deverá colidir com diversos blocos durante o jogo e cada bloco possui um efeito diferente no ato da colisão, além disso, o personagem pode vestir diferentes armaduras e cada uma delas pode interagir diferentemente com cada um dos blocos do jogo, além das colisões com os inimigos.

Uma ideia para a implementação de colisão, no que diz respeito a tentar encontrar qual é a ação correta a ser feita, é codificar os elementos como valores inteiros e depois fazer uma seleção através de um comando *switch* (ou *case*). O problema desta codificação é espalhar o algoritmo de colisão por várias classes no jogo, tornando a manutenção complicada e a difícil verificação, se todos os tratamentos foram realizados. Uma ilustração desta situação pode ser vista na Figura 5-59.

Figura 5-58: Telas do jogo *Kid Chameleon*. O personagem pode vestir diferentes armaduras e cada uma tem sua própria característica.

5.23.2. Solução proposta pelo padrão:

Uma forma de solucionar este problema, sem criar um forte acoplamento hierárquico é o uso do padrão *Visitor*. Com ele a operação de tratar uma colisão é encapsulada na classe *VisitorColisaoInterface*. As classes visitantes concretas possuem o tratamento das

colisões esperadas, com isso, as classes que manipulam os personagens e os blocos não ficam carregadas com esses algoritmos. A aplicação do padrão *Visitor* pode ser vista no diagrama da Figura 5-60.

A classe *VisitorColisaoInterface* declara uma operação que todas as classes concretas precisam implementar. Esta operação é exatamente a que trata a colisão do objeto com outro.

5.23.3. Diagramas:

→ Sem o uso do padrão

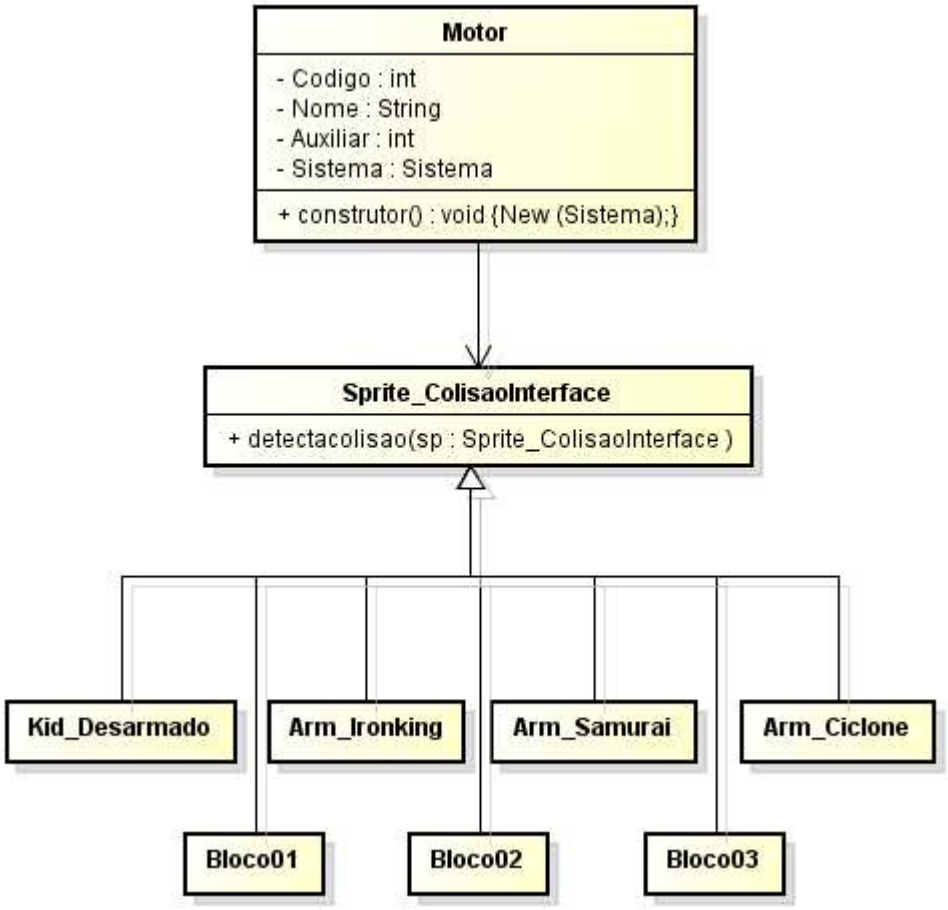

Figura 5-59: Diagrama parcial de um jogo sem o uso do padrão *Visitor*. Cada classe deverá implementar manualmente a colisão com todos os demais elementos em seus métodos.

→ Com o uso do padrão

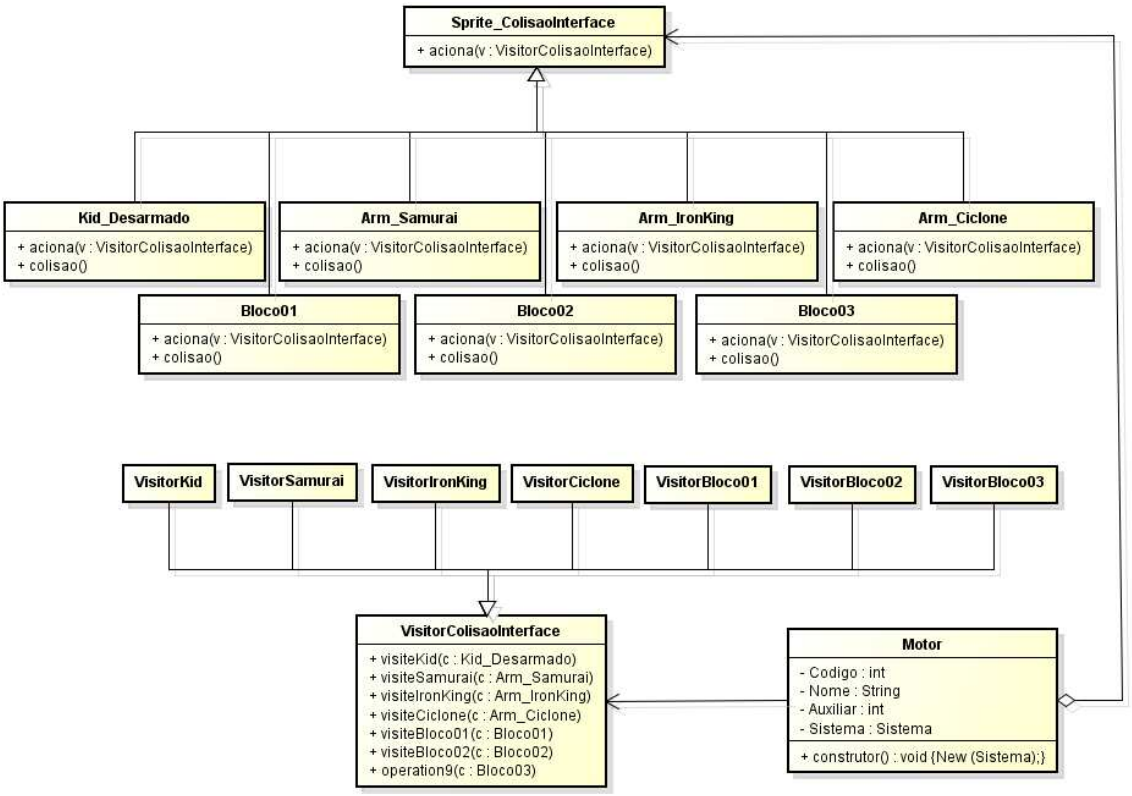

Figura 5-60: Diagrama com o uso do padrão *Visitor*. As colisões são resolvidas nas classes "visitantes" que são fracamente acopladas às classes dos *sprites*. As formas de colisão podem ser modificadas sem a preocupação com o resto do código dos personagens.

CONCLUSÃO DO USO DOS PADRÕES

Com o que foi exposto neste capítulo é possível perceber que os padrões se misturam e se completam na solução de alguns problemas em jogos. O uso de padrões permite criar jogos de fácil manutenção, expansão, evolução e que pode ter suas classes reutilizadas em outros projetos de jogos digitais.

Como nem tudo tem somente vantagens, também é possível perceber que a aplicação da maioria dos padrões causa um aumento no número de classes, o que pode acarretar uma perda de performance geral do jogo. Esse fato pode ser preocupante em dispositivos mais simples e o caso deve ser bem estudado, porém, de forma geral, não acarreta transtornos na maioria dos computadores atuais.

6. Experimento

Para mensurar o impacto da aplicação dos padrões de projeto aqui propostos no desenvolvimento de jogos digitais, foi realizado o experimento que será descrito neste capítulo.

6.1. Objetivos

O objetivo do experimento foi averiguar e quantizar as melhorias da adoção de padrões de projeto em jogos, em relação à redução no tempo de desenvolvimento, a diminuição na presença de *bugs* e na redução de linhas de código.

6.2. Participantes

Para realização do experimento, foram convocados alunos a partir do sexto período do curso de Ciência da Computação da Faculdade de Ciências Aplicadas e Sociais de Petrolina (FACAPE), que já cursaram a disciplina de "Prática de Programação", cujo conteúdo é ministrado com práticas em C#. Todos os participantes responderam o questionário, constante no Anexo 01 – Questionário aos participantes do experimento deste trabalho.

Dezesseis alunos se inscreveram para participar do experimento. A Figura 6-1 mostra o período em que os participantes do experimento estão cursando, dentro do curso de Ciência da Computação.

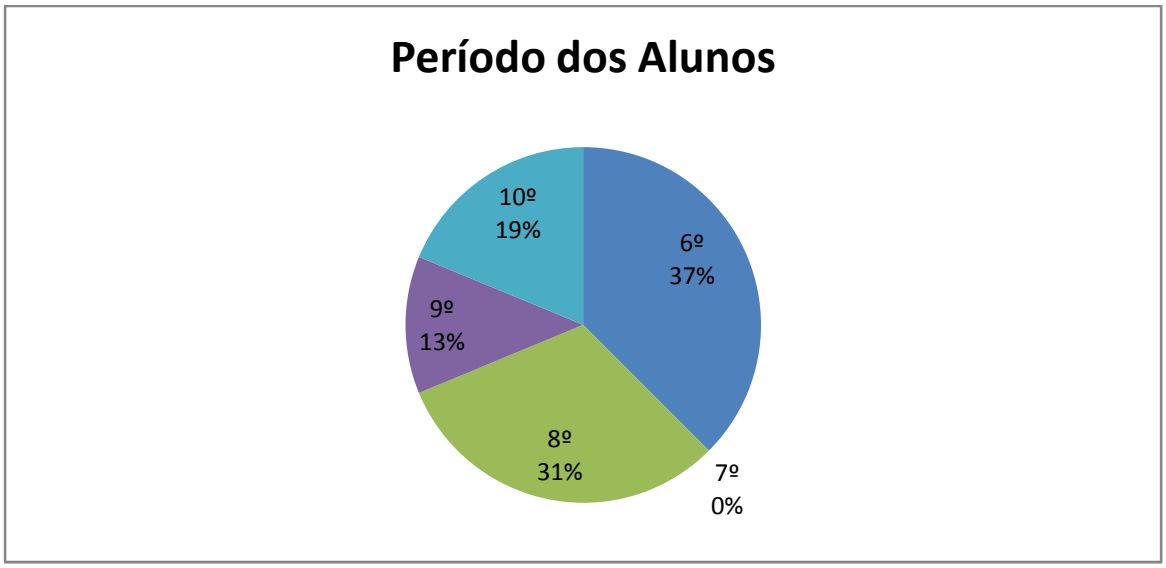

Figura 6-1: Gráfico que mostra a porcentagem de participantes do experimento por período no curso de Ciência da Computação.

A Figura 6-2 mostra o tempo de experiência dos participantes na linguagem de programação C#.

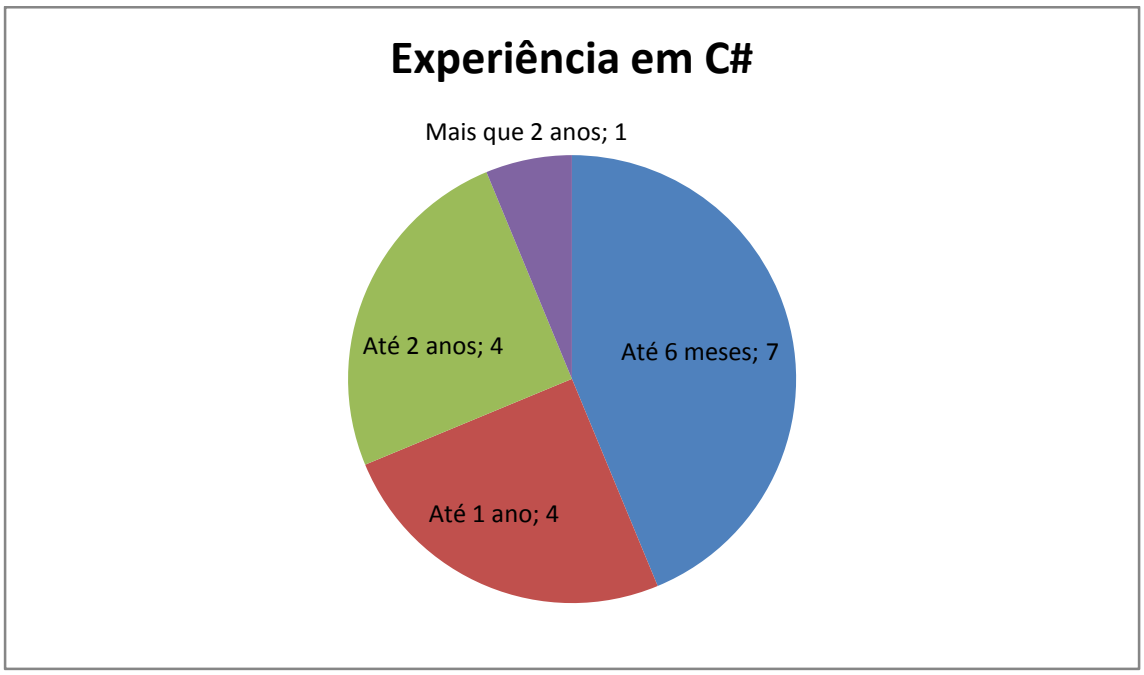

Figura 6-2: Tempo de experiência em C#.

Foi perguntado sobre a experiência do participante no desenvolvimento de jogos, cujo resultado é mostrado na Figura 6-3. Dos alunos que responderam “Sim” a esta pergunta, nenhum deles afirma ter desenvolvido jogos em caráter comercial. O desenvolvimento foi apenas por lazer e de maneira informal.

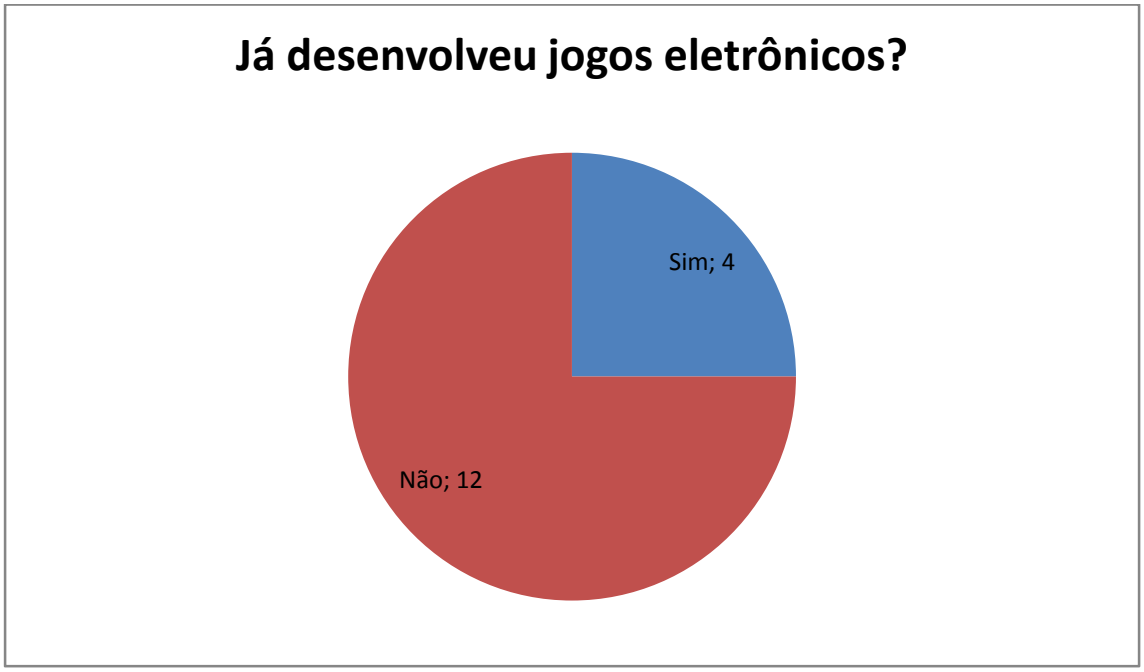

Figura 6-3: Quantidade de Alunos que já participaram do desenvolvimento de jogos eletrônicos.

Foi questionado aos participantes sobre seu conhecimento em padrões de projeto, especialmente o GoF, os resultados são mostrados na Figura 6-4. Dentre os participantes que responderam "Sim" a esta pesquisa, todos afirmam que nunca utilizaram esses padrões no desenvolvimento de jogos. Quando perguntado sobre quais padrões conheciam, foram citados os padrões: *Singleton*, *Observer*, *Adapter*, *Proxy*, *Facade* e *Prototype*.

Figura 6-4: Conhecimento dos participantes em relação aos padrões de projeto GoF.

6.3. Protocolo do Experimento

Este tópico descreve como o experimento foi planejado e executado.

Os alunos inscritos no experimento foram divididos em grupos com três componentes, sendo que um grupo ficou com apenas um componente, totalizando seis grupos. A divisão dos grupos tentou equilibrar o máximo possível o período no curso, a experiência na linguagem C# e a experiência em desenvolvimento de jogos. O participante com um conhecimento superior aos demais, ficou sozinho.

Devido ao tempo que os alunos tinham para participar do experimento e a complexidade envolvida no processo, apenas três padrões de projeto foram utilizados no experimento, são eles: *Singleton*, *Prototype* e *Facade*. Estes padrões foram escolhidos devido à facilidade no seu desenvolvimento e a possibilidade de serem aplicados a um jogo simples e curto.

Coube a cada grupo a implementação de um jogo no estilo nave, bastante simples, mas que oferecesse a oportunidade de se utilizar os padrões de projeto selecionados.

Na véspera da execução do experimento, a três desses grupos (chamados de grupos A, B e C, sendo o grupo C, o grupo com apenas um componente) foi explicado como utilizar os três padrões GoF de maneira geral, mas não especificamente em jogos. Aos demais grupos (chamados de D, E e F) nada foi explicado. A Tabela 3 mostra o perfil de cada um dos grupos em detalhes.

Tabela 3: Perfil dos grupos participantes do experimento.

Aluno	Grupo	Período em que cursa Ciência da Computação	Experiência em C#	Conhecimento básico sobre padrões GoF	Experiência básica no desenvolvimento de jogos
1	A	6	Até 6 meses	Não	Não
2	A	8	Até 6 meses	Não	Sim
3	A	10	Até 2 anos	Sim	Não
4	B	9	Até 1 ano	Não	Sim
5	B	8	Até 1 ano	Sim	Não
6	B	6	Até 6 meses	Não	Não
7	C	10	> 2 anos	Sim	Não
8	D	10	Até 2 anos	Não	Não
9	D	6	Até 6 meses	Não	Não
10	D	6	Até 6 meses	Não	Não
11	E	9	Até 2 anos	Sim	Não
12	E	8	Até 1 ano	Não	Sim
13	E	6	Até 6 meses	Não	Não
14	F	6	Até 6 meses	Não	Não
15	F	8	Até 1 ano	Não	Sim
16	F	8	Até 2 anos	Não	Não

No dia 06 de outubro de 2013 os grupos foram reunidos em um laboratório de informática e a todos foi explicado o seguinte:

> "Cada grupo terá disponível um computador e deverá desenvolver um jogo de nave com as especificações que lhes estão sendo repassadas, na ferramenta Visual Studio 2010, com a biblioteca XNA para desenvolvimento de jogos, ambas previamente instaladas no laboratório. Todos vocês receberam um projeto em andamento e deverão usar tudo que o projeto lhes oferece. Quem terminar primeiro, com todas as especificações, ganhará uma premiação. Mãos a obra."

O projeto em andamento fornecido às equipes D, E e F contou apenas com as imagens a serem utilizadas no jogo, mostradas em conjunto na Figura 6-5, previamente importadas para o Visual Studio. O projeto fornecido para as equipes A, B e C, além das imagens, continha as classes prontas referes aos padrões de projeto *Singleton*, *Prototype* e *Facade* mostradas no anexo 02. Para o padrão *Singleton*, foi entregue a classe "*SingletonConfiguracao*" que garante que a classe onde são geradas configurações do

jogo, não poderá ser replicada, ou seja, existirá uma única instância em todo o projeto do jogo. Para o padrão *Prototype*, foi entregue a classe "*NaveInimigaPrototype*" que possui a capacidade de clonar a si mesma e gerar naves inimigas. Para o padrão *Facade*, foi entregue a classe "*EndidadeFacade*". Esta classe é a única forma que as demais classes do jogo possuem para acessar as classes capazes de fazer com que uma imagem associada a um objeto apareça na tela.

Figura 6-5: Montagem feita com as imagens utilizadas no jogo.

Os requisitos fornecidos aos participantes foram listados no seguinte texto:

> O projeto refere-se ao desenvolvimento de um jogo de nave com uma única tela em C#, com a utilização da biblioteca XNA, no ambiente de desenvolvimento Visual Studio 2010.
>
> O personagem jogador, que é a nave amarela, inicialmente ficará na parte central inferior da tela. A nave poderá se deslocar pela tela pressionando as setas do teclado para a posição correspondente a seta pressionada, mas não pode sair do campo visual da tela. Esta nave é capaz de disparar tiros ao pressionar de uma tecla. Esses tiros partirão de onde a nave do jogador se encontra e seguirão em linha reta até sair da tela pelo topo da mesma. A velocidade do tiro deve ser maior que a da nave jogadora.
>
> Na parte superior da tela em colunas aleatórias surgirão naves inimigas, que serão as naves azuis. Estes inimigos circularão pela tela em uma trajetória e também poderão disparar tiros. Os tiros inimigos partirão da nave inimiga e seguirá em linha reta até sair da tela pela parte inferior da mesma.
>
> Quando o tiro na nave do jogador colide com uma nave inimiga, o jogador ganha 50 pontos e a nave inimiga é destruída (simplesmente desaparece da tela). Quanto um total de 10.000 pontos for atingido, as

naves desaparecerão e surgirá uma mensagem de “Congratulações”, indicando que o jogo terminou e o jogador foi vitorioso. O jogador deve iniciar o jogo com a pontuação zerada.

Caso a nave do jogador colida com uma nave inimiga ou com um tiro da nave inimiga, ele perderá uma vida, sendo que inicialmente contará com um total de três vidas. Caso as três vidas sejam perdidas, as naves desaparecerão da tela e surgirá uma mensagem de “*Game Over. Tecle Enter para reiniciar.*” indicando que o jogo terminou e o jogador foi derrotado no jogo.

O experimento teve início às 08h34min da manhã e nenhum dos participantes sabia da diferença entre os projetos iniciais entregues aos grupos.

6.4. Resultados

Os resultados obtidos no experimento são apresentados na Tabela 4.

Tabela 4: Resultados do experimento

	Uso dos Padrões	Tempo Aproximado	Completude	Bugs	Linhas de Código	Quantidade de Classes
Grupo A	Sim	06h50min	100%	0	246	21
Grupo B	Sim	05h56min + 40min	90%	1	227 + 26	23
Grupo C	Sim	06h05min	100%	0	218	25
Grupo D	Não	07h40min	100%	1	345	15
Grupo E	Não	07h48min + 01h57min	80%	1	207 + 52	1
Grupo F	Não	06h40min	100%	1	251	6

O tempo foi medido através da observação de um relógio convencional. A cada tempo foi somado um valor estimado para o cumprimento dos requisitos faltantes. O cálculo da estimativa foi realizado através de uma "regra de três", entre a quantidade de requisitos atendidos, a quantidade total de requisitos e do tempo auferido.

A completude foi medida de acordo com o atendimento dos requisitos do jogo, a saber:

1. O jogo é inicializado e o background aparece corretamente;

2. Nave jogadora aparece na tela e se movimenta nos quatro sentidos (norte, sul, leste, oeste);

3. A nave jogadora atira e o tiro segue em linha reta em velocidade superior a da nave jogadora;

4. Naves inimigas aparecem aleatoriamente na tela;

5. Naves inimigas atiram e o tiro segue na direção correta;

6. Vidas sendo perdidas a cada colisão da nave jogadora com tiros inimigos ou com os próprios inimigos, sendo que a nave jogadora reaparece na posição inicial a cada uma destas colisões;

7. Contagem de 50 pontos a cada destruição de nave inimiga ocorrendo e sendo visualizada na tela;

8. Nave jogadora não pode sair do campo visual da tela;

9. Mensagem de "Congratulações" quando o jogador atinge 10.000 pontos;

10. Mensagem de “Game Over” quando o jogador colide três vezes com naves inimigas ou com tiros de naves inimigas;

A implementação de cada requisito inclui 10% no total da completude do jogo.

Os bugs foram medidos de acordo com o protocolo de testes dinâmicos realizados com o jogo. O protocolo de testes pede para que cada um dos 10 requisitos apresentados seja testado por duas pessoas diferentes, cada um em uma máquina diferente. A Tabela 5 mostra o planejamento feito para testes dos requisitos do jogo.

Tabela 5: Protocolo de testes do jogo

Requisito	Ação do testador	Resultado esperado
1	Inicializar o jogo	Jogo carregado, apresentando o background na tela e a nave amiga surgindo no centro da parte baixa deste background.
2	Pressionar as teclas de seta do teclado.	A nave jogadora se movimenta na direção pressionada.
3	Pressionar a tecla referente ao tiro do jogador.	Um tiro é disparado, partindo da nave e seguindo em linha reta até o topo da tela.
4	Inicializar o jogo	Naves inimigas irão aparecendo aleatoriamente no topo da tela a cada intervalo de tempo.
5	Inicializar o jogo	Naves inimigas, em intervalos de tempos aleatórios, atiram e o tiro segue em linha reta na direção sul da tela.
6	Mover a nave para causar colisões propositais da nave jogadora com as naves inimigas e com os tiros inimigos.	O jogador perde uma vida e retorna para a posição inicial do jogo.
7	Efetuar disparos certeiros em naves inimigas.	Naves inimigas desaparecem da tela e 10 pontos são somados ao *score* do jogador.
8	Mover a nave jogadora de forma a tentar fazê-la sair da tela nas quatro direções possíveis.	A nave jogadora pára quando chega à margem da tela.
9	Efetuar disparos certeiros em 10 naves inimigas.	Todas as naves e tiros desaparecem da tela e aparece a mensagem: “Congratulações”.
10	Mover a nave para causar colisões propositais da nave jogadora com as naves inimigas ou com os tiros inimigos três vezes.	Todas as naves e tiros desaparecem da tela e aparece a mensagem: “Game Over”.

Para cada um dos resultados esperados não atendidos, um bug é incluído nos resultados do experimento. Caso o requisito não tenha sido implementado, a ausência do resultado esperado não é contada como bug.

A quantidade de linhas de código foi contada de maneira absoluta, contando uma a uma as linhas de código dos jogos resultantes de cada equipe. A esse valor absoluto, foi

somado uma quantidade de linhas estimada para os possíveis requisitos faltantes do jogo daquele grupo. A estimativa foi baseada na quantidade de linhas de código do grupo que implementou todos requisito do jogo, respeitando o uso ou não dos padrões de projeto.

A quantidade de classes foi contada de maneira absoluta, contando uma a uma as classes dos jogos resultantes de cada equipe.

6.5. Análise dos Resultados

Com os resultados verificados na Tabela 4 podemos concluir que, o tempo médio de implementação do jogo com a utilização dos padrões foi de 06h31min, enquanto que a média das equipes que não utilizaram os padrões foi de 08h02min. Em relação à completude, quase todas as equipes conseguiram concluir o projeto, tendo certo equilíbrio entre as equipes que utilizaram padrões e as que não utilizaram. Foi verificado *bugs* em todos os projetos que não utilizaram padrões e apenas um bug nos projetos que utilizaram padrões. Além disso, a média de quantidade de linhas de código das equipes que utilizaram os padrões foi de 717 linhas, enquanto que a média dos projetos que não utilizaram os padrões foi de 855 linhas. Já em relação ao quantitativo de classes, pode-se verificar que a média da quantidade de classes dos grupos que utilizaram os padrões foi de 23 classes. Bem superior a média de 07 classes, auferida dos grupos que não utilizaram os padrões.

Com isso podemos mensurar que o ganho em tempo de desenvolvimento com a utilização de padrões foi em torno de 18,9% e o ganho em linhas de código foi em torno de 16,14%. Além disso, a qualidade do software melhorou, reduzindo a quantidade de *bugs* gerados. Também é possível mensurar que a quantidade de classes geradas teve um aumento de mais de três vezes quando os padrões são utilizados, o que não significa mais trabalho, visto que houve uma redução no número total de linhas digitadas. Estes resultados já eram esperados visto que a literatura sobre padrões de projeto já afirmam tais ganhos em softwares e aplicações comerciais.

Apesar do número de participantes não ser suficiente para que cheguemos a conclusões definitivas, os dados do experimento mostram evidências claras de que o uso de padrões reduz o esforço na programação e o tempo necessário para desenvolver um projeto de jogos, além de melhorar a qualidade do código. Isto se deve a reutilização de código feita de maneira correta proporcionada pelos padrões, que, além de disponibilizar uma solução pronta a um problema recorrente, permite a redução no esforço com a utilização de classes prontas que outros jogos também poderão reutilizar.

7. Conclusão

Este trabalho mostrou os desafios no desenvolvimento de jogos, desde seus primórdios até os dias de hoje e como muitos desses desafios foram vencidos e alguns que ainda estão por vencer. Além disso, foi visto a evolução dos ambientes de programação, desde a linguagem de máquina até o surgimento dos motores de jogos.

Também foi apresentado o conceito e evolução dos padrões de projeto, com ênfase nos padrões de projeto definidos pela GoF, foco principal do trabalho. A importância de se utilizar padrões para resolver problemas conhecidos ficou evidenciada quando se fala em reusabilidade de códigos já testados, o que aumenta a agilidade no processo de desenvolvimento e reduz a quantidade global de *bugs* no jogo.

O trabalho discutiu ainda diversos artigos presentes da literatura que falam sobre padrões de projeto em jogos, seja definindo novos padrões ou empregando padrões existentes em projetos reais.

A grande contribuição do trabalho foi mostrar como os vinte e três padrões de projeto GoF podem ser aplicados no desenvolvimento de jogos, muitos deles totalmente ausentes na literatura de jogos até então. Os diagramas apresentados mostram uma opção de como é feita a programação sem o padrão e com o padrão, trazendo comentários sobre as melhorias que o uso do padrão trouxe ao projeto do jogo.

Entre essas melhorias estão a reusabilidade e a qualidade dos jogos, visto que classes fracamente conexas, utilizadas na solução de problemas, podem ser reaproveitadas na solução do mesmo problema em outros jogos, por se tratar de problemas recorrentes neste tipo de projeto, e como estas classes já foram testadas e aprovadas, induzirá menos *bugs*, conferindo assim, uma maior qualidade ao produto.

Os benefícios de se utilizar padrões de projeto, de uma maneira geral, são confirmados pela literatura. Esta apresenta vantagens aos desenvolvedores e mostra como ajudar a construir uma aplicação confiável com arquitetura testada e a experiência acumulada pela indústria. Além de promover a reutilização de projetos em futuros sistemas, identificar equívocos comuns e armadilhas que ocorrem durante o desenvolvimento, tornar o projeto independentemente da linguagem de programação, estabelecer um vocabulário comum entre projetos e os desenvolvedores e reduzir a fase de projeto no processo de desenvolvimento de sistemas (DEITEL e DEITEL, 2005). O experimento realizado neste trabalho corroborou o que se espera da adoção de padrões de projeto.

A principal dificuldade encontrada no desenvolvimento da pesquisa foi a competição acirrada entre as empresas, que fazem com que os projetos dos jogos estejam sobre sigilo industrial. Pouca documentação existe sobre o processo de criação de jogos

adotados por estas empresas e, dos projetos encontrados, nenhum fala da aplicação dos padrões de projeto GoF. Com isso, a pesquisa focou em artigos acadêmicos publicados em revistas e congressos sobre o assunto, mas muitos dos artigos não possuem o foco direto no desenvolvimento de jogos, tornando escassas as referências que realmente tratavam sobre o assunto.

Para dar continuidade a este projeto, em trabalhos futuros, devem ser feitos experimentos com os demais padrões. Também será preciso aumentar a quantidade de participantes do experimento e avaliar outras métricas como acoplamento e modularidade.

Outra linha de trabalho futuro é realizar uma revisão sistemática da literatura, onde os dados são minerados em repositórios, registrando quais os repositórios que mais possuem referências de padrões de projeto em jogos e os padrões encontrados.

Outra linha de trabalho futuro consiste no estudo da aplicação de outros padrões de projeto em jogos, como os padrões GRASP (*General Responsibility Assignment Software Patterns*) ou ainda a identificação de novos problemas recorrentes no projeto de desenvolvimento de jogos, o que pode sugerir a criação de novos padrões de projeto sempre com o objetivo de facilitar a vida dos desenvolvedores e *Game Designers*.

Referências

[1] AMPATZOGLOU, A. et al. **An empirical investigation on the impact of design pattern application on computer game defects**. In. XV International Academic MindTrek Conference: Envisioning Future Media Environments. Tampere, p. 8. 2011.

[2] AMPATZOGLOU, A.; CHATZIGEORGIOU, A. **Evaluation of object-oriented design patterns in game development**. Journal: Information and Software Technology, Ed. Elsevier, p. 10. 2006.

[3] BERGSTRÖM, K.; BJÖRK, S.; LUNDGREN, S. **Exploring Aesthetical Gameplay Design Patterns – Camaraderie in Four Games**. In: International Digital Media & Business Festival - MindTrek. Tampere, p. 8. 2010.

[4] BEZNOSYK, A. et al. **The Influence of Cooperative Game Design Patterns for Remote Play on Player Experience**. In. X Asia Pacific Conference on Computer Human Interaction. Matsue, p. 9. 2012.

[5] BJÖRK, S.; HOLOPAINEN, J. **Patterns In Game Design**. Ebook: The Game Design Reader, p.410 a 437. 2005.

[6] BUSSO, T. M.; FERREIRA, M. A. G. V. **Soluções Reutilizáveis no domínio de jogos computacionais: a aplicação de padrões de projeto no desenvolvimento de motores de jogos**. Disseretação de Mestrado defendida na Escola Politécnica da Universidade de São Paulo. São Paulo. 2006.

[7] DAHLSKOG, S.; TOGELIUS, J. **Patterns and Procedural Content Generation**. In. Workshop on Design Patterns in Games (DPG 2012) Article No. 1 [S.l.]. 2012.

[8] DAVIDSSON, O.; PEITZ, J.; BJÖRK, S. **Game Design Patterns for Mobile Games**. In. Project report to Nokia Research Center. Finland, p. 57. 2004.

[9] DEITEL, P. J.; DEITEL, H. M. **Java Como Programar**. Tradução de Edson Furmankiewicz. 6. ed. São Paulo: Pearson Prentice Hall, 2005.

[10] FERNANDES, M. **Programação de Jogos com Delphi usando Direct X**. Florianópolis: Relativa, 2002.

[11] FURTADO, A. DSL Tools - Melhore sua produtividade através de linguagens visuais de domínio-específico no Visual Studio.NET. **Linha de Código**, 2008. Disponivel em: <http://www.linhadecodigo.com.br/artigo/1456/dsl-tools-melhore-sua-produtividade-atraves-de-linguagens-visuais-de-dominio-especifico-no-visual-studionet.aspx>. Acesso em: 18 jul. 2013.

[12] GAMMA, E. et al. **Padrões de Projeto**. Porto Alegre: Bookman, 2000.

[13] GESTWICKI, P. V. **Computer Games as Motivation for Design Patterns**. In. 38th SIGCSE Technical Symposium on Computer Science Education. P. 233-237. 2007.

[14] GESTWICKI, P.; SUN, F.-S. **On Games, Patterns, and Design**. In. Symposium on Science of Design. p. 17-18, 2007.

[15] GESTWICKI, P.; SUN, F.-S. **Teaching Design Patterns Through Computer Game Development.** Journal on Educational Resources in Computing (JERIC**)**, New York, 2008.

[16] GUERRA, R. Tecmundo Jogos. **Tecmundo**, 2012. Disponivel em: <http://www.tecmundo.com.br/jogos/31445-como-anda-o-mercado-de-jogos-no-brasil-.htm>. Acesso em: 13 abr. 2013.

[17] HOMER, B. D.; PLASS, J. L. **Educational Game Design Pattern Candidates**. Journal of Research in Science Teaching, New York, p. 16. 2009.

[18] HULLETT, K.; WHITEHEAD, J. **Design Patterns in FPS Levels**. Fifth International Conference on the Foundations of Digital Games. p. 78-85, New York, 2010.

[19] KAAE, R. C. **Using design patterns in game engines**. TietoEnator Consulting A/S. Helsinki. Ebook, 2001.

[20] KIRK, W. J. **Design Patterns of Successful Role-Playing Games**. [S.l.]. Ebook, 2005.

[21] LARSEN, S.; AARSETH, E. **Level Design Patterns**. Copenhagen: IT, Ebook, 2006.

[22] LOH, S.; SOON, S. H. **Comparing Computer and Traditional Games Using Game Design Patterns**. In. CyberGames - International Conference on Game Research and Development. p.237-241, 2006.

[23] MARTÍN, M. A. G.; DÍAZ, G. J.; ARROYO, J. **Teaching Design Patterns Using a Family of Games**. In. 14th SIGCSE Conference on Innovation and Technology in Computer Science Education. p. 268-272, 2009.

[24] MARTINS, E. **Melhoria da Manutenabilidade - Notas de Aula**. São Paulo: [s.n.], 2010.

[25] MCGEE, K. **Patterns and Computer Game Design Innovation**. In. 4th Australasian Conference on Interactive Entertainment. Article No. 16, 2007.

[26] PAULA FILHO, W. P. **Multimídia:** Conceitos e Aplicações. 2ª Edição. Ed. São José: LTC, 2011.

[27] PERUCIA, A. S. et al. **Desenvolvimento de Jogos Eletrônicos – Teoria e Prática**. São Paulo: Novatec, 2005.

[28] ROLLINGS, A.; MORRIS, D. **Game Architecture and Design: A New Edition.** Indianapolis: New Riders, Ebook, 2003.

[29] SHALLOWAY, A.; TROTT, J. R. **Explicando Padrões de Projeto: uma nova perspectiva em projeto orientado a objeto.** Porto Alegre: Bookman, 2002.

[30] SILVEIRA, I. F.; SILVA, L. **Aprendizagem de Padrões de Projeto em Ciência da Computação através de Jogos Digitais**. In.XIV WEI - Workshop sobre Educação em Computação. São Paulo, p. 10. 2006.

[31] THOMAZINI NETO, L. F.; JANDI JR, P. **Padrões de Projeto**. Notas de aula. Jaguariuna: [s.n.], 2006.

[32] TRINDADE, J. M. F.; FISCHER, L. G. **Estudo e Aplicação de Padrões**. Notas de Aula. Universidade Federal do Rio Grande do Sul. Rio Grande do Sul, p. 13. 2008.

[33] WICK, M. R. **Teaching Design Patterns in CS1: a Closed Laboratory Sequence based on the Game of Life**. In. 36th SIGCSE Technical Symposium on Computer Science Education. Eau Claire, p. 487-491, 2005.

[34] WONG, S. B.; NGUYEN, D. **Design Patterns for Games**. In. 33rd SIGCSE Technical Symposium on Computer Science Education. p. 126-130 New York, 2002.

ANEXOS

Anexo 01 – Questionário aos participantes do experimento

Nome: _____

Disciplina que cursa com o prof. Tenorio, caso haja alguma _____

Email: _____

Curso superior em andamento ou concluído:
( ) Ciência da Computação ( )Engenharia da Computação
( ) Outros. Qual? _____

Período que está cursando: _____ (Se já for formado, escreva FORMADO aqui)

Quais linguagens você programa e quanto tempo de experiência? (pode marcar mais de uma opção)

( ) C _____ ( ) C# _____ ( ) Pascal/Delphi _____

( ) Java _____ ( ) C ++_____ ( ) Não sei programar

( ) Outras. Quais _____

Você já desenvolveu jogos? ( ) Sim ( ) Não

Se sim, responda: Quantos Comercialmente? _____ E por lazer? _____

Quanto tempo de experiência em programação de jogos? _____

Quais linguagens você usou? _____

Já ouviu falar em Padrões de Projeto?
( ) Sim, bastante ( ) Sim, apenas em alguma disciplina na Faculdade ( ) Não

Se sim, quais? _____

Já conhecia os Padrões de Projeto GoF?
( ) Sim, bastante ( ) Sim, apenas em alguma disciplina na Faculdade ( ) Não

Se sim, quais? _____

Já usou na prática os Padrões de projeto GoF?
( ) Sim, em projetos comerciais ( ) Sim, apenas em projetos na faculdade ( ) Não

Se sim, quais? _____
Já usou Padrões de projeto GoF em jogos?
( ) Sim ( ) Não Se sim, quais? _____

Anexo 02 – Classes entregues aos grupos A, B e C do experimento.

Classe 01: Entidade

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace JogoNave.Facade
{
   public abstract class Entidade : DrawableGameComponent
   {
      #region Atributos: imagem, posicao, spriteBatch
      protected Texture2D imagem;
      protected Vector2 posicao;
      protected SpriteBatch spriteBatch;

      #endregion
      #region Propriedades: Pisicao, Retangulo

      public Vector2 Posicao
      {
         get { return posicao; }
         set { posicao = value; }
      }

      public Rectangle Retangulo
      {
         get
         {return  new Rectangle((int)posicao.X,(int)posicao.Y,imagem.Width,imagem.Height);}
       }

      #endregion
      #region Construtor

      public Entidade (Game game) : base (game)
      {
         this.posicao = Vector2.Zero;
         this.spriteBatch =
(SpriteBatch)this.Game.Services.GetService(typeof(SpriteBatch));
      }

      #endregion
      #region Métodos Virtuais: LoadContent

      public virtual void LoadContent(string assetName)
      { this.imagem = this.Game.Content.Load<Texture2D>(assetName); }
      #endregion
   }  }
```

Classe 02: EntidadeComplexa

```
using System.Collections.Generic;
using Microsoft.Xna.Framework;

namespace JogoNave.Facade
{
   public abstract class EntidadeComplexa: Entidade
   {
      #region Atributos: componentes
      protected List<GameComponent> componentes;
      #endregion
      #region Propriedades: Componentes

      public List<GameComponent> Componentes
      {
         get { return componentes; }
         set { componentes = value; }
      }
      #endregion
      #region Construtor

      public EntidadeComplexa(Game game) : base(game)
      { this.componentes = new List<GameComponent>(); }

      #endregion
      #region Métodos Nativos: Update, Draw

      public override void Update(GameTime gameTime)
      {
         for (int i = 0; i < this.componentes.Count; i++)
         {
            if (this.componentes[i].Enabled)
            { this.componentes[i].Update(gameTime); }
         }
         base.Update(gameTime);
      }

      public override void Draw(GameTime gameTime)
      {
         for (int i = 0; i < this.componentes.Count; i++)
         {
            if (this.componentes[i] is DrawableGameComponent &&
((DrawableGameComponent) this.componentes[i]).Visible)
            { ((DrawableGameComponent) this.componentes[i]).Draw(gameTime); }
         }
         base.Draw(gameTime);
      }
      #endregion
   } }
```

Classe 03: EntidadeFacade

```
using Microsoft.Xna.Framework;

namespace JogoNave.Facade
{
   public abstract class EntidadeFacade: EntidadeComplexa
   {
      #region Construtor

      public EntidadeFacade(Game game) : base(game)
      {
      }

      #endregion
      #region Metodos Nativos: Draw

      public override void Draw(GameTime gameTime)
      {
         if(this.imagem != null)
         {
            this.spriteBatch.Draw(this.imagem,this.posicao,Color.White);
         }

         base.Draw(gameTime);
      }

      #endregion
   }
}
```

Classe 04: SingletonConfiguracao

```
using Microsoft.Xna.Framework.Input;

namespace JogoNave
{
   public class SingletonConfiguracao
   {
      private static SingletonConfiguracao instancia;

      //CONFIG JOGO
      public int PosicaoTextoIniciarJogoX { get ; set ; }
      public int PosicaoTextoIniciarJogoY { get ; set ; }
      public int DimensaoTelaAltura { get ; set ; }
      public int DimensaoTelaLargura { get ; set ; }
      public int PosicaoTextoFimJogoY { get; set; }
```

```
//CONFIG NAVE
public float VelocidadeTiroNave { get ; set ; }

//CONFIG INIMIGO
public float VelocidadeNaveInimiga { get ; set ; }
public float PosicaoInicialNaveInimigaX { get; set; }
public float PosicaoInicialNaveInimigaY { get; set; }
public int QtdMaximoInimigo { get; set; }
public int TempoCriarNaveInimiga { get; set; }
public int TempoDirecaoNaveInimiga { get; set; }
public int TempoNaveInimigaAtirar { get; set; }

//CONIG JOGADOR
public float VelocidadeNaveJogador { get ; set ; }
public int QtdVidasJogador { get ; set ; }
public float PosicaoInicialNaveJogadorX { get ; set ; }
public float PosicaoInicialNaveJogadorY { get ; set ; }
public float PosicaoVidaJogadorX { get ; set ; }
public float PosicaoVidaJogadorY { get ; set ; }
public float PosicaoScoreJogadorX { get ; set ; }
public float PosicaoScoreJogadorY { get; set; }

//CONFIG TECLADO JOGO
public Keys MoverParaBaixo { get; set; }
public Keys MoverParaCima { get; set; }
public Keys MoverParaEsquerda { get; set; }
public Keys MoverParaDireita { get; set; }
public Keys DispararTiro { get; set; }

private SingletonConfiguracao()
{
   //CONFIG INI JOGO
   PosicaoTextoIniciarJogoX = 155;
   PosicaoTextoIniciarJogoY = 280;
   DimensaoTelaAltura = 600;
   DimensaoTelaLargura = 800;
   PosicaoTextoFimJogoY = 250;

   //CONFIG INI NAVE
   VelocidadeTiroNave = 10.0f;

   //CONFIG INI INIMIGO
   VelocidadeNaveInimiga = 5.0f;
```

```
            PosicaoInicialNaveInimigaX = 400;
            PosicaoInicialNaveInimigaY = 100;
            QtdMaximoInimigo = 10;
            TempoCriarNaveInimiga = 2000;
            TempoDirecaoNaveInimiga = 3000;
            TempoNaveInimigaAtirar = 2000;

            //CONFIG INI JOGADOR
            VelocidadeNaveJogador = 8.0f;
            QtdVidasJogador = 3;
            PosicaoInicialNaveJogadorX = 360;
            PosicaoInicialNaveJogadorY = 500;
            PosicaoVidaJogadorX = 70;
            PosicaoVidaJogadorY = 30;
            PosicaoScoreJogadorX = 710;
            PosicaoScoreJogadorY = 30;

            //CONFIG INI TECLADO JOGO
            MoverParaBaixo = Keys.Down;
            MoverParaCima = Keys.Up;
            MoverParaEsquerda = Keys.Left;
            MoverParaDireita = Keys.Right;
            DispararTiro = Keys.Space;
        }

        public static SingletonConfiguracao Instancia
        {
            get
            {
                if(instancia == null)
                {
                    instancia = new SingletonConfiguracao();
                }
                return instancia;
            }
        }
    }
}
```

Classe 05: NaveInimigaPrototype

```
using System;
using JogoNave.Enums;
using JogoNave.Facade;
using Microsoft.Xna.Framework;

namespace JogoNave.EntidadesJogo
{
   public class NaveInimigaProtoType : Nave
   {
      public override Nave ProtoType()
      {   return this.MemberwiseClone() as NaveInimigaProtoType;  }
      #region Atributos: tempoParaEscolherDirecaoAleatoria, tempoParaComecarAtirar
      private double tempoParaEscolherDirecaoAleatoria;
      private double tempoParaComecarAtirar;
      #endregion
      #region Contrutor

      public NaveInimigaProtoType(Game game, Vector2 posicao) : base(game,posicao)
      {
         Random random = new Random(DateTime.Now.Millisecond);
         DirecaoHorizontalNave direcaoHorizontalNave =
(DirecaoHorizontalNave)(int)random.Next((int)DirecaoHorizontalNave.Esquerda,
(int)DirecaoHorizontalNave.Direita);

         DirecaoVerticalNave direcaoVerticalNave =
(DirecaoVerticalNave)(int)random.Next((int)DirecaoVerticalNave.Cima,
(int)DirecaoVerticalNave.Baixo);

         this.tempoParaEscolherDirecaoAleatoria = 0;
         this.tempoParaComecarAtirar =
SingletonConfiguracao.Instancia.TempoNaveInimigaAtirar;

         this.direcaoHorizontal = direcaoHorizontalNave;
         this.direcaoVertical = direcaoVerticalNave;
         this.velocidade = SingletonConfiguracao.Instancia.VelocidadeNaveInimiga;
      }

      #endregion
      #region Métodos Nativos: Update

      public override void Update(GameTime gameTime)
      {
```

```
            this.tempoParaEscolherDirecaoAleatoria +=
gameTime.ElapsedGameTime.TotalMilliseconds;
            this.tempoParaComecarAtirar +=
gameTime.ElapsedGameTime.TotalMilliseconds;

            this.MudarDirecao();
            this.ControlarTiro();
            this.EscolherNovaDirecaoAletoria();

            base.Update(gameTime);
        }

        #endregion
        #region Métodos Privados: MudarDirecao, ControlarTiro,
EscolherNovaDirecaoAletoria

        private void MudarDirecao()
        {
            //HORIZONTAL
            if(this.Retangulo.Left <= 0)
            {
                this.direcaoHorizontal = DirecaoHorizontalNave.Direita;
            }
            else
            {
                if(this.Retangulo.Right >=
SingletonConfiguracao.Instancia.DimensaoTelaLargura)
                {
                    this.direcaoHorizontal = DirecaoHorizontalNave.Esquerda;
                }
            }

            //VERTICAL
            if(this.Retangulo.Top <= 0)
            {
                this.direcaoVertical = DirecaoVerticalNave.Baixo;
            }
            else if(this.Retangulo.Bottom >=
SingletonConfiguracao.Instancia.DimensaoTelaAltura)
            {
                this.direcaoVertical = DirecaoVerticalNave.Cima;
            }
        }
```

```
        private void ControlarTiro()
        {
            if(this.tempoParaComecarAtirar >
SingletonConfiguracao.Instancia.TempoNaveInimigaAtirar)
            {
                this.TesteCriarTiro(DirecaoTiro.Baixo);

                this.tempoParaComecarAtirar = 0;
            }
        }

        private void EscolherNovaDirecaoAletoria()
        {
            if(this.tempoParaEscolherDirecaoAleatoria >=
SingletonConfiguracao.Instancia.TempoDirecaoNaveInimiga)
            {
                Random random = new Random(DateTime.Now.Millisecond);
                DirecaoHorizontalNave direcaoHorizontalNave =
(DirecaoHorizontalNave)(int)random.Next((int)DirecaoHorizontalNave.Esquerda,
(int)DirecaoHorizontalNave.Direita);

                DirecaoVerticalNave direcaoVerticalNave =
(DirecaoVerticalNave)(int)random.Next((int)DirecaoVerticalNave.Cima,
(int)DirecaoVerticalNave.Baixo);

                this.direcaoHorizontal = direcaoHorizontalNave;
                this.direcaoVertical = direcaoVerticalNave;

                this.tempoParaEscolherDirecaoAleatoria = 0;
            }
        }

        #endregion
    }
}
```