UNIVERSIDADE ESTADUAL DE CAMPINAS

Instituto de Computação

DANIEL YAN 214793

ERICK MANAROULAS FELIPE 215448

Relatório do projeto “Duo Como Péda”

Relatório de projeto final da disciplina

MC322 CD - Programação Orientada a Objetos

Prof. Dr. Leonardo Montecchi

CAMPINAS

2021

1. **Pacotes principais**
    a. **Main**

    Contém as classes Main, usada para rodar o programa e Tester, que define cenários usados com o intuito de testar e validar seções do código.

    b. **Game**

    Contém quase todos os outros pacotes, é responsável por toda a lógica interna do jogo. Especialmente, contém quatro Singletons, **Board** e **Game**, que são responsáveis pelo gerenciamento do fluxo do jogo e do combate, respectivamente, além de conter todos os outros pacotes abaixo. Também contém o nosso **EffectManager**, que é uma interface entre os eventos e os efeitos que devem ser ativados por eles. E, para a representação visual do jogo, há a classe **TextualGraphicsEngine**, que é responsável por imprimir o estado do jogo no console.

    c. **Menu**

    Contém toda a lógica de interação humano-computador da aplicação, utilizando-se do padrão de projeto Command. Há um menu diferente para defensores e atacantes, tanto em fase de combate quanto na main phase.

    d. **Player**

    Contém a própria classe abstrata Player, que define as ações dos jogadores, e o pacote Deck, que é responsável pela interação entre o Player com suas cartas e a construção de decks utilizando o padrão de projeto Factory.

    e. **Card**

    Contém a implementação mais geral de uma carta, a classe Card. Contém também três outros pacotes:

        1. Spell - Funcionalidade geral de cartas mágicas e as magias específicas, como “Judgement”
        2. Minion - Funcionalidade geral de cartas de combate, além dos minions específicos, como “Poro”. Neste pacote, há também o pacote Champion, que extende a classe Minion além de implementar os champions específicos, como “Garen”. Dentro

do pacote Champion, temos o pacote Mission, que rege como as missões de campeões serão implementadas.

3. Event - Contém as interfaces responsáveis pela emissão de eventos do jogo, MinionEventHandler e SpellEventHandler. Cada evento pede dados diferentes para funcionar. Por exemplo, onPlay pede um Player “owner” e um Card “PlayedCard”, já onKill precisa saber do Owner, de um Minion “Killer” e de um minion “Killed”

f. Stats

Stats representa recursos que podem ser usados no jogo, como Health e Mana. Está em um pacote separado devido a seus elementos poderem ser usados tanto por Cards quanto por Players. Health é uma classe que é usada por minions e players e é responsável pelo gerenciamento dos pontos de vida do usuário. Como minions e players produzem efeitos completamente diferentes, foi criada a interface Killable, que delega a responsabilidade de produzir os efeitos de morte a quem a implementar. Assim, Health gerencia a vida de qualquer coisa que implemente Killable. Além disso, criamos a classe MinionStats, que se torna responsável pelo gerenciamento de tudo relacionado à vida e ao poder dos minions. Assim, podemos delegar essas funcionalidades e simplificar a classe Minion, além de facilitar a inclusão de efeitos temporários como buffs ou negações de poder.

2. Fluxo de Execução

a. Inicialização do jogo

Pelo Tester, o Jogo inicia primeiro criando dois jogadores controlados por pessoas, usando decks prontos e dando as cartas iniciais a elas. Depois, o loop do jogo começa:

1. Começa o Round, dando o token de ataque para um jogador
2. Printa o estado atual do jogo
3. Processa o input do jogador
4. Passa para o turno do próximo jogador
5. Checa se deve haver combate
6. Processa o combate

7. Avança para o próximo round

b. Fluxo dos turnos

Os turnos avançam seguindo as seguintes condições:

1. Na fase principal, caso o jogador passe.
2. Na fase de combate, caso o jogador confirme o posicionamento dos seus minions.

Os rounds avançam caso os dois jogadores passem ou confirmem.

Fig. 1 - O turno é de Player João no momento, então ele não consegue ver as cartas da mão de Maria. As quatro fileiras do meio representam o banco (fileira 0 e 3) e o campo de batalha (fileira 1 e 2)

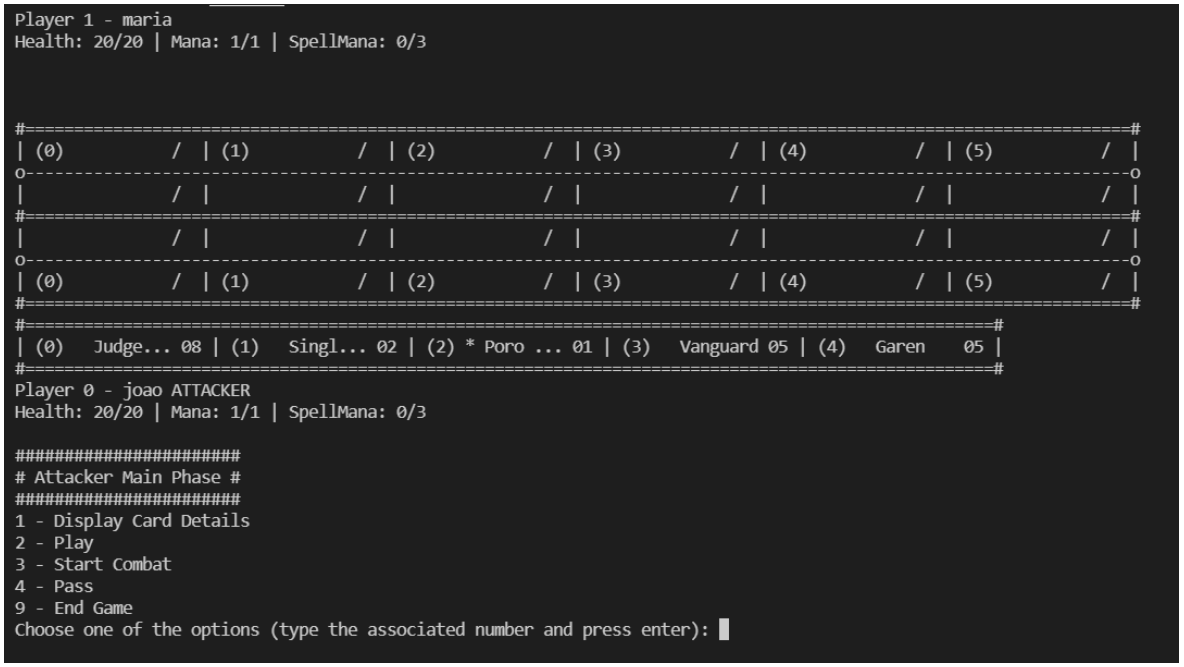
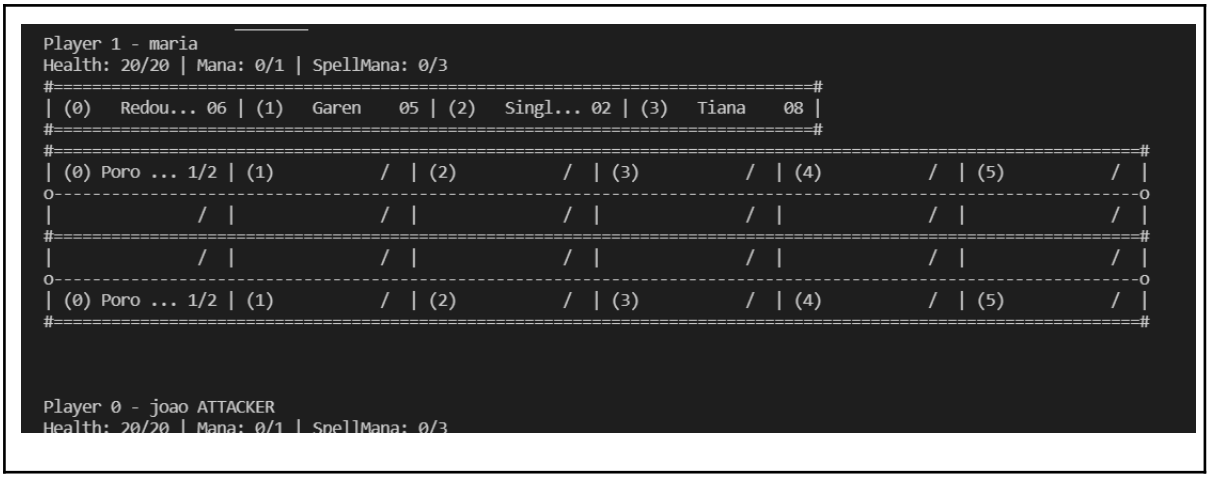


Fig. 2 - *Ambos João e Maria colocaram um poro em seus bancos. João agora pode decidir atacar ou não.*



.

c. Finalização

O jogo pode acabar a qualquer momento caso algum jogador chame o comando de Quit Game, ou caso um dos jogadores morra, assim dando a vitória para o oponente.

3. Processo de desenvolvimento

Começamos o projeto a partir de um diagrama de classes UML com as classes e métodos mais importantes para o jogo. A partir dele, começamos a implementar todas as classes uma por uma. Com o processo de implementação, descobrimos a necessidade de criar novas classes e pacotes. O primeiro UML foi realizado manualmente com a ferramenta Star UML, já o diagrama finalizado foi gerado com auxílio da ferramenta IntelliJ IDEA. Um problema do IDEA é que não conseguimos escolher quais classes queremos expandir, apenas pacotes. Isso faz o diagrama ficar um pouco conturbado, repetindo dependências. A fim de simplificação, geramos um diagrama para cada pacote, explicitando as dependências entre pacotes. Os diagramas são encontrados na pasta **UML** do projeto. Nas páginas abaixo, colocamos uma ilustração dos diagramas.

Fig. 3 - Diagrama Inicial

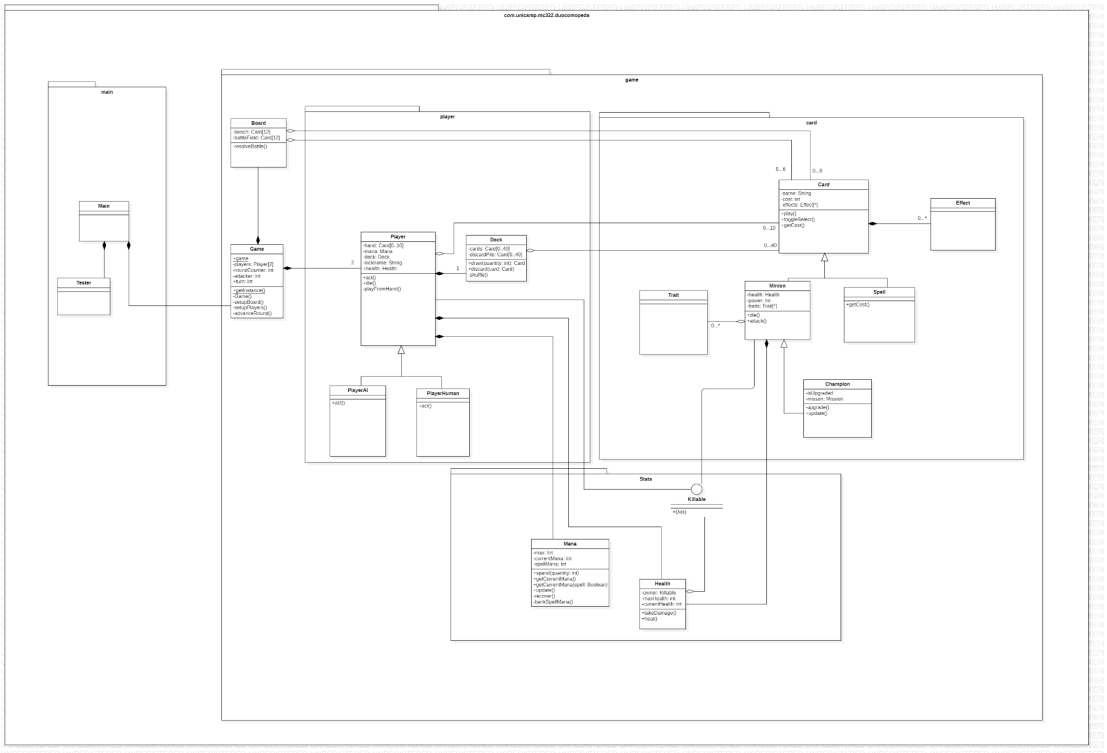


Fig. 4 - Diagrama do pacote Game

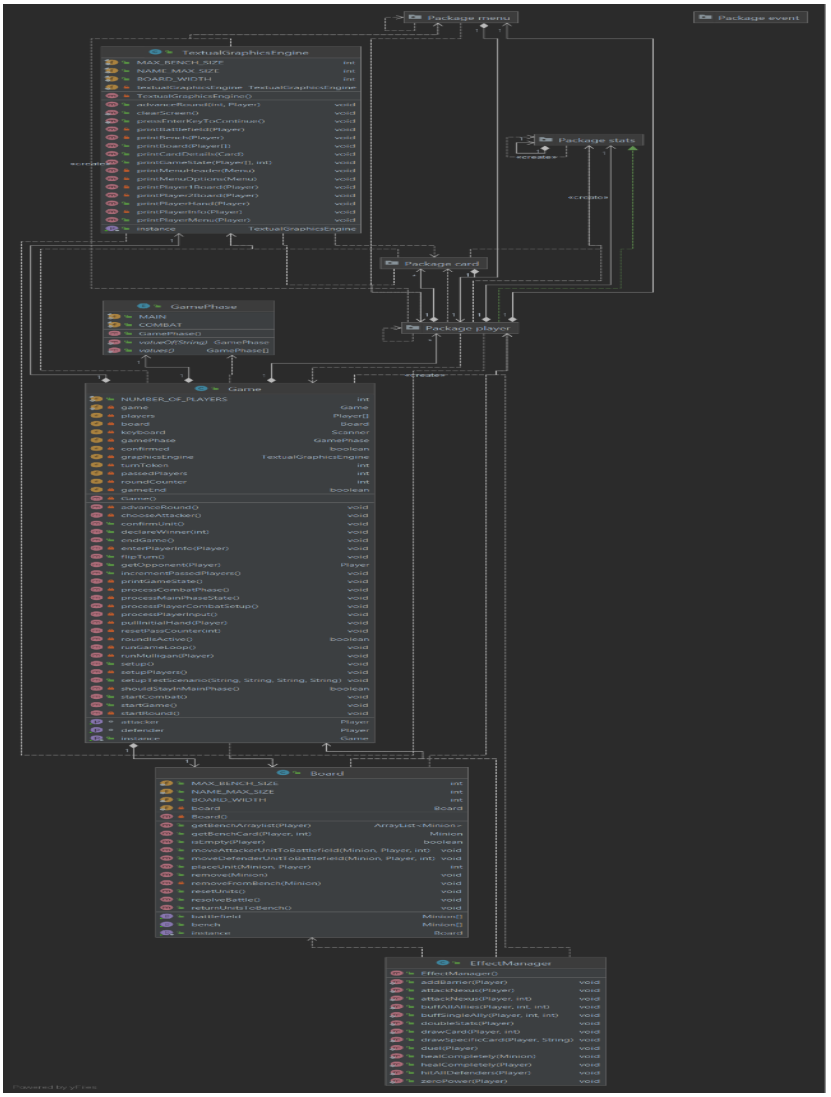
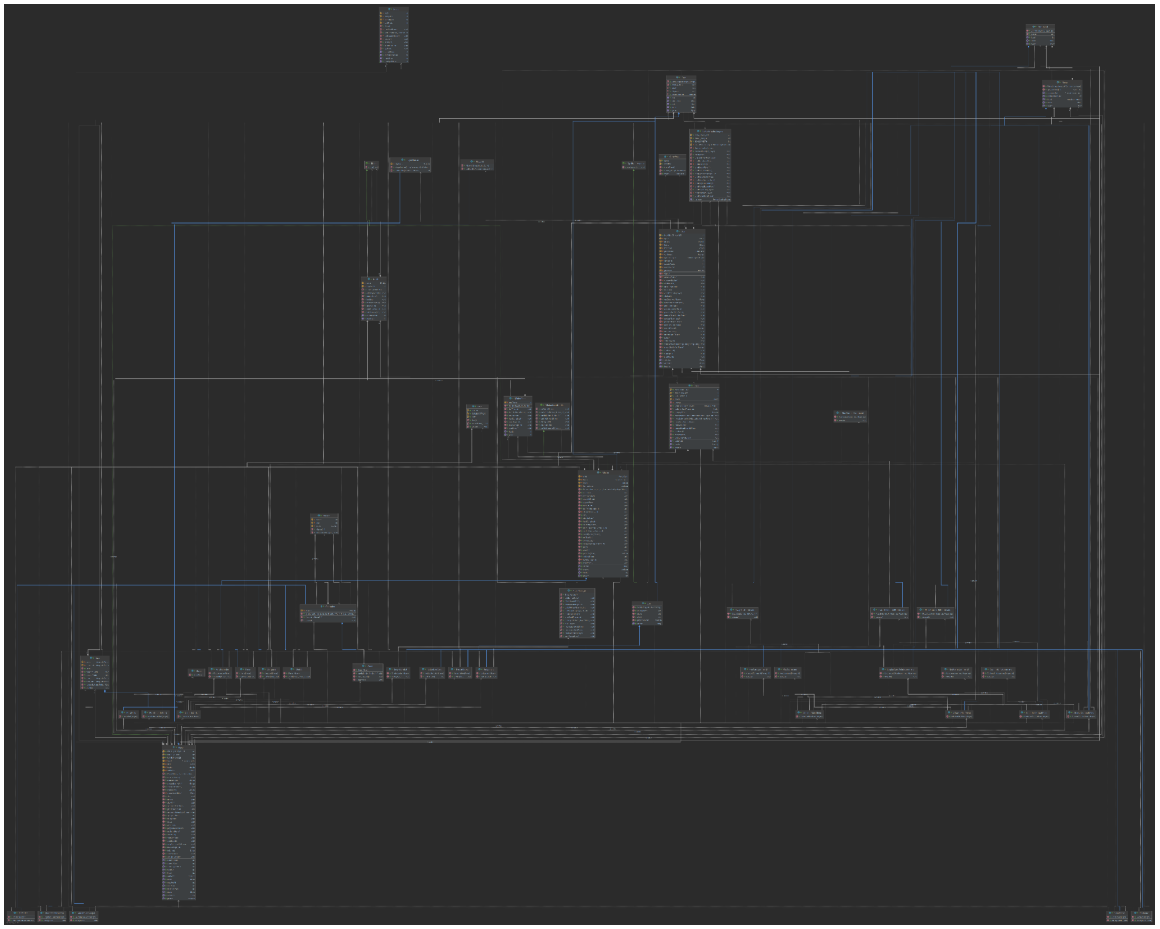
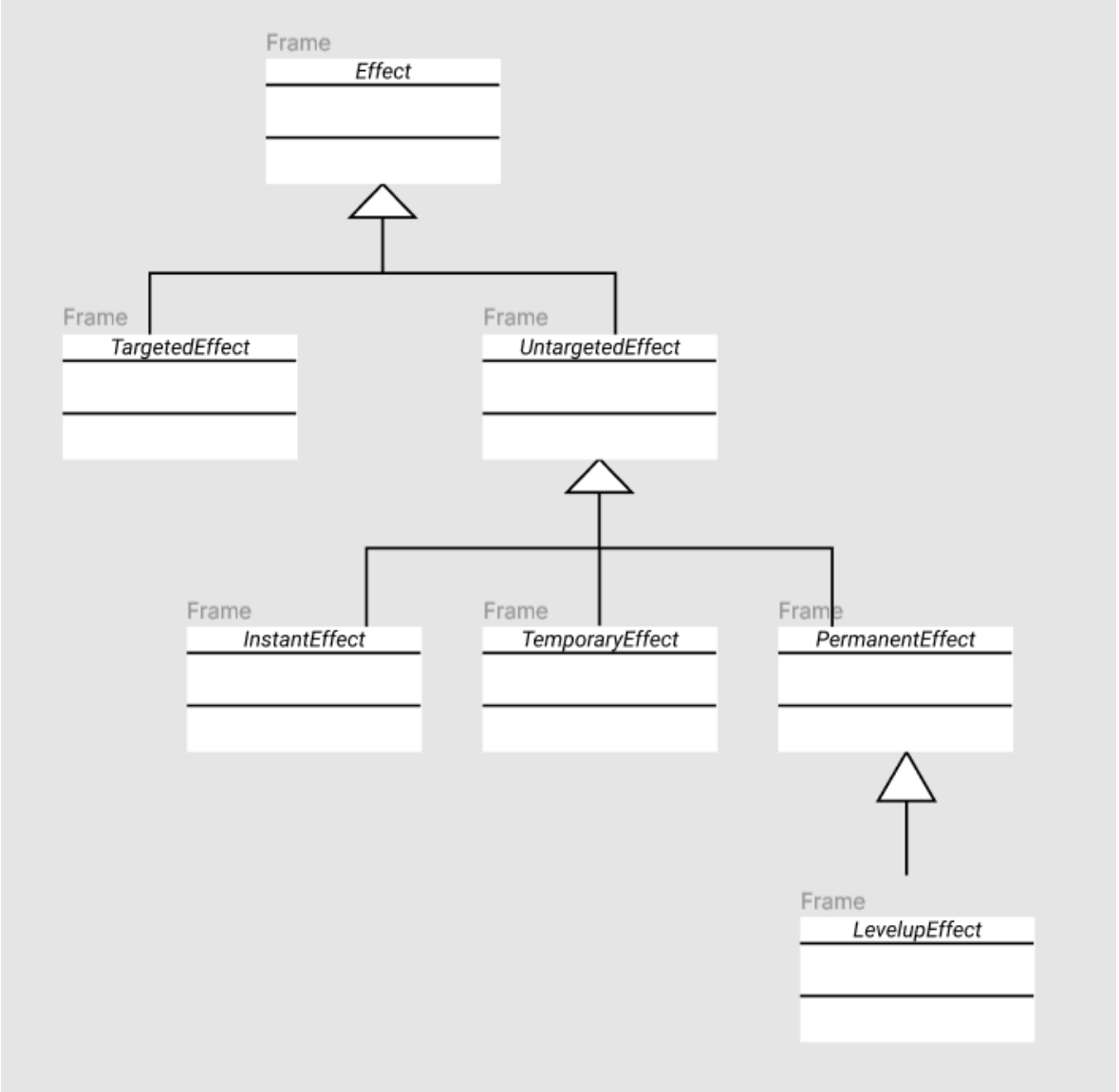


Fig. 5 - Diagrama do pacote Game - Expandido



Uma das nossas maiores dificuldades foi implementar o sistema de efeitos e gatilhos. Inicialmente havíamos projetado um pacote effect, com uma classe para cada Effect. Cada effect seria ativado com um Trigger, que teria classes filhas para cada tipo de vento. Um possível efeito seria drawCard(OnKillTrigger trigger), sendo o OnKillTrigger responsável por carregar as informações relevantes como jogador dono do minion que matou, o minion que matou e o minion que morreu. Pensamos também em uma árvore de efeitos que seguiria o seguinte modelo:

Fig. 6 - Árvore de herança para a classe Effect



Todo esse sistema acabou complicando demais o funcionamento do projeto. Decidimos então optar por um sistema mais simplificado: Criamos um Singleton chamado EffectManager, que carrega consigo métodos estáticos para cada efeito especificado. Criamos duas interfaces, MinionEventHandler e SpellEventHandler que configuram quando os eventos serão lançados e quais informações devem passar. A classe abstrata *Minion* então implementa a interface de MinionEventHandler, porém com os métodos todos vazios. Então, cada especificação de minion, como Vanguard, por exemplo, chama o efeito que quiser

de EffectManager no evento que quiser. Aqui poderíamos ter feito uma relação de herança entre as interfaces, fazendo MinionEventHandler uma especificação de SpellEventHandler ou ambos serem especificações de um possível EventHandler. Como o único evento em comum entre os dois era onPlay(), decidimos implementar as interfaces sem alguma relação. Uma ilustração da situação descrita acima a seguir:

Fig. 7 - MinionEventHandler

```java
package com.unicamp.mc322.duocomopeda.game.event.handler;

import com.unicamp.mc322.duocomopeda.game.card.Card;
import com.unicamp.mc322.duocomopeda.game.card.minion.Minion;

public interface MinionEventHandler {

    public void onPlay(Card playedCard);

    public void onKill(Minion killer, Minion killed);

    public void onDeath(Minion killed);

    public void onHit(Minion attacker, Minion defender, int damage);

    public void onTakeDamage(Minion target, int damage);

    public void onDefense(Minion attacker, Minion defender, int damage);

    public void onRoundEnd();

}
```

Fig. 8 - Implementação vazia da interface MinionEventHandler, com gatilhos corretos e funcionais na classe Minion

```
private void defend(Minion attacker, int attackerDamage) {
    this.onDefense(attacker, this, attackerDamage);
    attacker.takeDamage(this.stats.getPower());
    if (attacker.isDead) {
        this.onKill(this, attacker);
    }
}

// Maybe it's public
private void takeDamage(int amount) {
    if (!barrierActive) {
        this.onTakeDamage(this, amount);
        this.stats.takeDamage(amount);
    }
    barrierActive = false;
}

public void zeroPower() {
    this.stats.zeroPower();
}

// default option for event is to do nothing
public void onPlay(Card playedCard) {

}

public void onKill(Minion killer, Minion killed) {

}
```

Fig. 9 - Classes Vanguard e método buffAllAllies de EffectManager

```java
public class Vanguard extends Minion {

    public Vanguard(Player owner) {
        super("Vanguard", 5, 5, 5, EnumSet.noneOf(Trait.class), owner,
                "Effect: When this card is played, give +1/+1 to all allies");
    }

    @Override
    public void onPlay(Card playedCard) {
        EffectManager.buffAllAllies(getOwner(), 1, 1);
    }

}
```

```java
public static void buffAllAllies(Player player, int power, int health) {
    Board board = Board.getInstance();
    ArrayList<Minion> minions = board.getBenchArraylist(player);
    for (Minion minion : minions) {
        minion.buff(power, health);
    }
}
```

Desta forma, conseguimos realizar os efeitos especificados de uma maneira um pouco mais simples, sem prejudicar o desempenho do jogo.

Uma outra coisa a se notar nos prints é que chamamos um método buff em Minion. Trata-se de uma delegação a um objeto da classe MinionStats que Minion carrega. Fizemos este tipo de delegação ao longo do projeto inteiro, o que cresceu o tamanho de algumas classes mas evitou muitos acessos múltiplos. Decidimos que seria mais intuitivo fazer *minion.buff()* em vez de *minion.getStats.buff()* ou, pior ainda, *{minion.getHealth.setMaxHealth(); minion.incrementPower()}.*