UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO E ENGENHARIA DE COMPUTAÇÃO

ALEX MORAES
BRUNO SANTANA
JOÃO WEIT

Implementação do jogo de tabuleiro War em Typescript usando os paradigmas funcional e orientado a objetos

Relatório apresentado como requisito parcial para a obtenção de conceito na Disciplina de Modelos de Linguagens de Programação

Prof. Dr. Lucas Mello Schnorr
Orientador

Porto Alegre
2018

SUMÁRIO

1 INTRODUÇÃO

Este trabalho tem por objetivo a implementação de uma aplicação em uma linguagem que suporte tanto o paradigma funcional quanto o paradigma orientado a objetos visando a comparação das duas abordagens na solução de um mesmo problema. A aplicação escolhida pelo grupo é uma versão eletrônica do jogo de tabuleiro *War* que será desenvolvida usando a linguagem TypeScript.

TypeScript

TypeScript é um superconjunto da linguagem JavaScript desenvolvida pela Microsoft, projetada com o objetivo de reduzir a complexidade de código produzido em JavaScript. Ao contrário de JavaScript, TypeScript é uma linguagem compilada e o produto final da compilação é código em JavaScript que pode ser interpertado por qualquer motor que suporte o padrão ECMAScript 3 ou superior.

Outra diferença em relação a sua predecessora é a possibilidade de tipagem estática. Quando são adicionadas anotações de tipo às declarações, a checagem de tipos é feita em tempo de compilação. Adicionalmente, existe a possibilidade do uso de classes, interfaces, módulos e *namespaces*. Além disso, sendo uma extensão do padrão ECMAScript 5, qualquer programa em JavaScript é um programa TypeScript válido.

Paradigma funcional

O paradigma funcional é uma maneira distinta da qual estamos acostumados de se descrever uma determinada computação. Diferente do paradigma imperativo, em funcional tudo é representado por funções matemáticas, sem estados ou efeitos colaterais, isto é, toda vez que uma função for executada, deve retornar o mesmo valor e não deve alterar nada externo à ela. Além disso, por ser um paradigma *declarativo*, a ordem de execução do programa não pode ser definida, tornando o mesmo altamente paralelizável.

Paradigma orientado a objetos

O paradigma orientado a objetos visa um forte mapeamento entre objetos do mundo real e objetos do programa, geralmente representados por classes. A orientação a objetos possui como base os conceitos de abstração de dados, encapsulamento, herança e polimorfismo, tornando o código gerado muito mais legível e de fácil reuso. Todas essas características têm como objetivo aproximar o software desenvolvido do mundo real, tornando o sistema extremamente modular e de fácil interpretação quando feito corretamente.

2 VISÃO GERAL DA LINGUAGEM

A linguagem definida para o desenvolvimento do trabalho foi Typescript, uma linguagem de código aberto desenvolvida pela Microsoft, que propõe orientação a objetos e compila em Javascript nativo. Por Javascript ser altamente portável - todos navegadores são capazes de executar javascript - Typescript acabou se tornando uma ótima opção para orientação a objetos em aplicações que necessitam rodar em navegadores. A principal adição do Typescript em relação ao Javascript notoriamente é a noção de orientação a objetos, permitindo a criação de classes, tipos estáticos, interfaces, entre outros - mais detalhes sobre estas e outras funcionalidades de orientação a objetos serão abordadas durante este trabalho. Typescript veio para substituir o Javascript nativo em projetos grandes, tornando as suas manutenções uma tarefa fácil, devido às grandes e já conhecidas vantagens de manutenção em códigos orientados a objetos, que são: abstração, encapsulamento, herança e polimorfismo. Mesmo Typescript sendo uma linguagem relativamente nova - lançado em fevereiro de 2017 -, o seu uso em aplicações comercias vem crescendo consideravelmente, graças à compatibilidade com Javascript, o que propõe migrações parciais de código para a nova linguagem. Uma das aplicações que possui implementações em Typescript é o VSCode, que é uma IDE altamente usada na indústria e foi inclusive a IDE escolhida para realizarmos a implementação deste trabalho.

3 APRESENTAÇÃO DO PROBLEMA

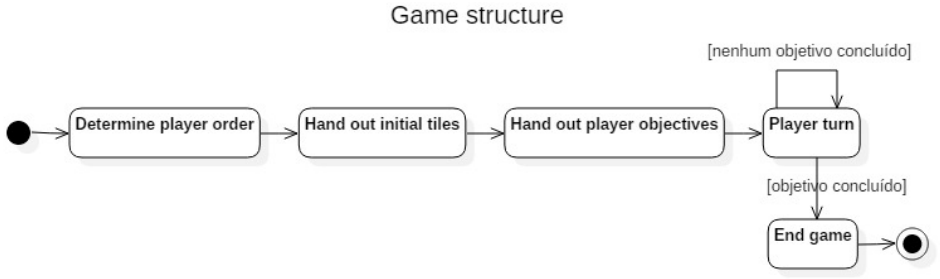
O jogo de tabuleiro *War* (*Risk* na sua versão orignal) é um jogo de estratégia no qual cada jogador controla exércitos em uma missão de dominação mundial. A dinâmica do jogo consiste em expandir o seu domínio sobre o tabuleiro participando de combates contra os exércitos dos outros jogadores (**??**).

Na etapa inicial do jogo, cada jogador escolhe para si, em turnos, um território desocupado do tabuleiro até que não existam mais territórios desocupados. Em cada um dos territórios escolhidos, o jogador coloca uma peça representando o seu exército de ocupaçao no território. Em seguida cada jogador recebe uma quantidade de exércitos definida de acordo com a quantidade de jogadores. Os jogadores também se revezam em turnos posicionando cada um desses exércitos em territórios sob o seu controle. Quando todos os exércitos são posicionados dessa forma, o jogo avança para a próxima etapa.

Na etapa seguinte, os jogadores ser revezam em turnos onde reforçam as suas posições em territórios já ocupados e combatem os exércitos dos territórios dos oponentes. Cada turno começa com o jogador ativo recebendo exércitos de acordo com a quantidade de territórios possuídos e os posicionando em seus territórios. Uma vez que todos os exércitos do turno tenham sido posicionados, o jogador pode escolher qualquer território seu que possui mais de um exército e declarar um ataque contra um território vizinho que seja controlado por um oponente. Em cada turno, o jogador pode escolher atacar quantas vezes quiser desde que possua territórios com mais de um exército.

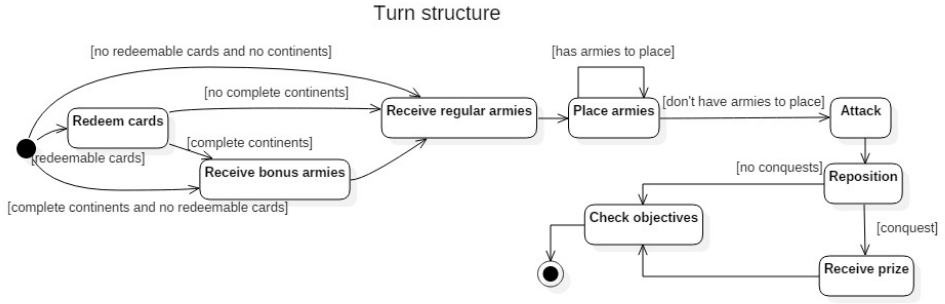
No combate, o jogador atacante escolhe até 3 exércitos no território atacante e rola um dado de ataque para cada exército escolhido. O jogador defensor escolhe até 2 exércitos no território atacado e rola um dado de defesa para cada exército escolhido. A maior rolagem de ataque é então comparada com a maior rolagem de defesa e, caso seja maior, um exército do território atacado é removido do tabuleiro. Caso a rolagem de

Figura 3.1: Representação em máquina de estados das etapas do jogo



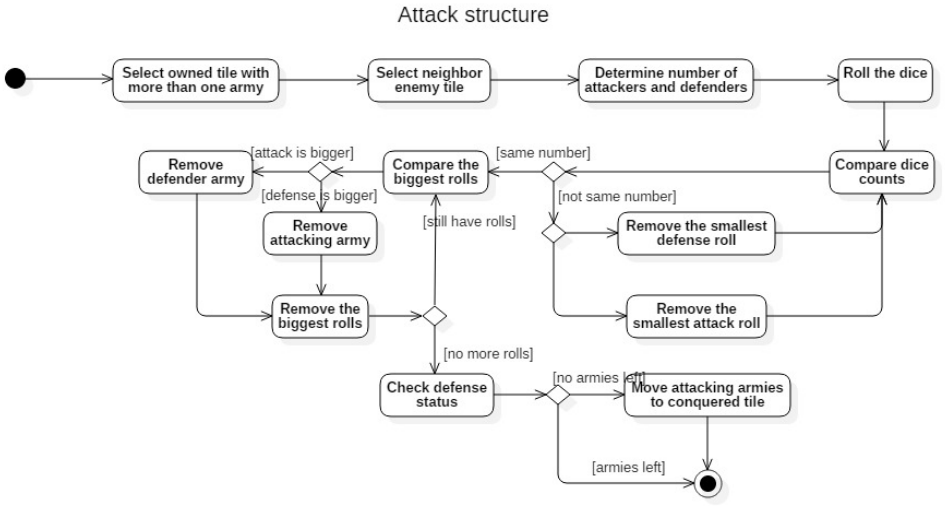
Fonte: Os Autores

Figura 3.2: Representação em máquina de estados das etapas do turno



Fonte: Os Autores

Figura 3.3: Representação em máquina de estados das etapas do combate



Fonte: Os Autores

ataque seja igual ou menor que a rolagem de defesa, um exército do território atacante é removido. Esse procedimento se repete para as próximas maiores rolagens de ataque e defesa enquanto houverem dados para serem comparados. Ao final desse processo, se o território atacado acabar sem exércitos, o jogador atacante move os exércitos atacantes que restaram para esse território.

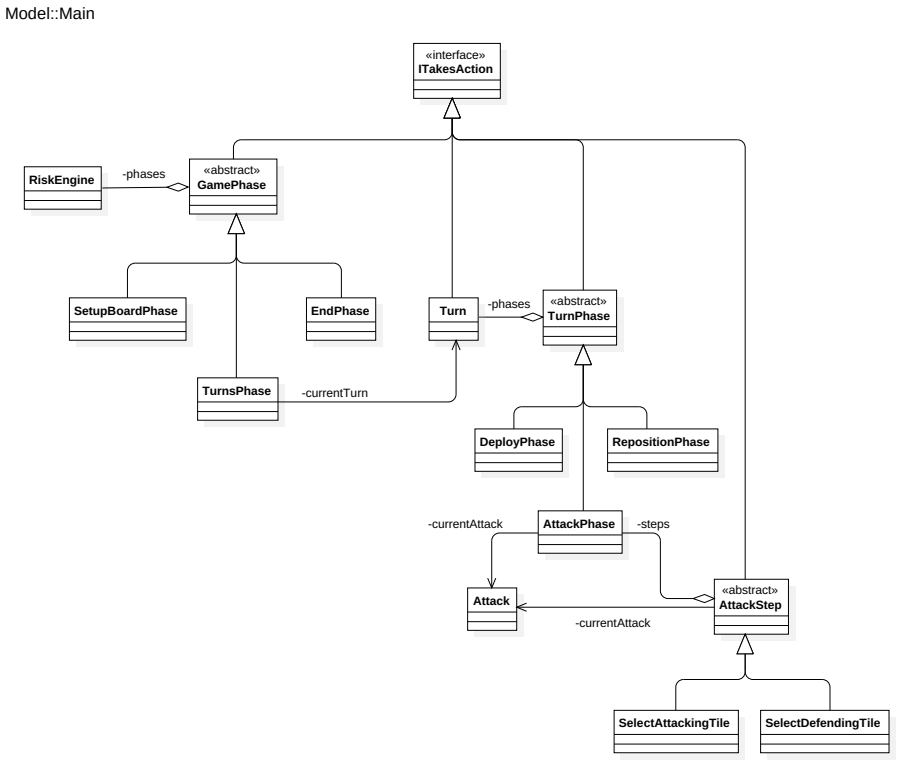
O sequênciamento dessas ações no jogo podem ser representadas por máquinas de estados e essa representação foi a que guiou o desenvolvimento da solução. A Figura 3.1 representa a sequência de etapas do jogo, a Figura 3.2 representa a sequências de fases de um turno e a Figura 3.3 representa as etapas de um combate.

4 ABORDAGEM ORIENTADA A OBJETOS

Classes

As classes são a base da orientação a objetos. A figura 4.1 representa o diagrama de classes UML modelado para este projeto e utilizado como base para implementação do mesmo.

Figura 4.1: Typescript class



Encapsulamento

O encapsulamento envolve a abstração dos métodos e atributos de determinada classe, tornando o software mais flexível à mudanças, já que cada funcionalidade deve ser implementada isolada das demais. Nesta implementação,

Construtores

Os construtores são métodos chamados ao instanciar um objeto, podendo ser herdados de uma classe abstrata à qual a classe em questão herda ou ser definidos na própria classe. Como o conceito de herança foi amplamente utilizado neste projeto, grande parte dos objetos herda o construtor de sua superclasse, mas em alguns casos extendemos o construtor herdado para atingir alguma funcionalidade a mais implementada pela classe filha, como pode ser observado na classe *SetupBoardPhase*, mostrado na Figura 4.2.

Figura 4.2: Construtor da classe *SetupBoardPhase*

```
public constructor(players: Array<IPlayer>, tiles: Array<ITile>){
    super(players);
    this.players = players;
    this.tiles = tiles;
}
```

Destrutores

Em typescript temos o conceito de *garbage collection*, um processo automático que desaloca objetos que não são mais referenciados no programa. Com esta tecnologia, o programador não precisa se preocupar com o gerenciamento de memória e evita problemas como o *memory leak*. Tendo este recurso disponível, destrutores se tornam desnecessários na linguagem. Porém, esta funcionalidade da linguagem também possui desvantagens, tal como o maior uso de recursos computacionais para definir que objetos podem ser desalocados, visto que em linguagens que não possuem garbage collection, o programador que gerencia a desalocação de memória, resultando em menor uso destes recursos. (**??**)

Polimorfismo por inclusão

O polimorfismo por inclusão é aquele em que um objeto pode pertencer a várias classes, tendo um comportamento modificado em relação ao comportamento original. Neste trecho de código, o polimorfismo por inclusão ocorre pois a propriedade *this.phases* é um Array de *TurnPhase* e este Array foi inicializado no construtor com dois elementos, o primeiro sendo o DeployPhase e o segundo o AttackPhase. Estas são classes que modificam comportamentos da classe *TurnPhase* e serão tratados como *TurnPhase* a partir de então. Posteriormente, por exemplo, o método *takeAction()* será chamado para cada elemento do Array de *TurnPhase*, sendo executado a implementação da classe herdada - *DeployPhase* ou *AttackPhase*.

Figura 4.3: Trecho do construtor da classe *Turn*

```
this.phases = [
    new DeployPhase(player),
    new AttackPhase(player)
]
```

Polimorfismo paramétrico

Polimorfismo por sobrecarga

Delegates

5 ABORDAGEM FUNCIONAL

6 CONCLUSÃO

Abordagem orientada a objetos