Instituto Superior de Engenharia de Coimbra

Departamento de Engenharia Informática e de Sistemas

Programação Avançada
Trabalho Prático
2017/2018

9Card Siege

Mário Rui Silveira Travanca nº 21230267
Maurizio Crocci nº 21230268

Introdução

Este documento foi realizado no âmbito da disciplina de Programação Avançada, e pretende ser uma fundamentação/complemento sobre o processo de desenvolvimento do nosso projeto prático, tendo em conta os objetivos estabelecidos “a priori” pelo enunciado.

As nossas escolhas sobre estruturação, máquina de estados e interface estarão descritas mais abaixo. Será explicado, detalhadamente, todo o processo de estruturação do trabalho proposto bem como a função de cada classe e a sua importância como parte de um todo. No que à Máquina de estados diz respeito, procurámos elaborar um esquema simples, percetível e realista.

Quanto ao tema do trabalho, o jogo, consiste num jogo de cartas em que participa um jogador que tem de defender o seu forte. O jogo mudará de estado consoante a carta que sair e o número que sair num dado virtual.

Arquitetura

A arquitectura da nossa aplicação segue o proposto pelo enunciado. Como tal, o aproveitamento do esquema disponibilizado pelo mesmo era incontornável, pelo que decidimos aproveitar toda a informação que este nos traduzia e adaptar de certa forma à nossa realidade.

Este pequeno esquema demonstra, quase na íntegra, todo o fluxo da nossa aplicação. No entanto, é-nos possível afirmar que o nosso projeto possui 3 principais packages:

- **<UI>** Este package funciona como interface. Toda a informação apresentada ao utilizador está contida nele. É independente da lógica e estados do jogo.
- **<Controllers>** Este package é a nossa base para a lógica de jogo. Possui toda a informação relacionada com a lógica de jogo. Permite-nos aceder ao package <model>, o que já não é possível através do package <ui>.
- **<Model>** Este package possui toda a informação sobre as cartas, spells, dados entre outros. No fundo, permite ao package <controllers> manipular toda a informação do jogo.

Classes

Vamos então explicar, detalhadamente, a função de cada classe. Seguindo o paradigma de cima, vamos dividir este capítulo em packages conforme explicado anteriormente. No entanto, agora haverá mais packages. Seguindo a ordem supracitada, temos:

<UI>

1. **TextUserInterface**: Classe responsável por mostrar ao utilizador o menu de jogo principal. Conforme a opção escolhida, reencaminha o fluxo para uma nova dimensão.

<controllers>

1. **GameState**: Classe responsável por fazer a ligação entre ui e o resto da aplicação. Permite aceder a informação sobre o Estado.

<controllers.states>

1. **IState**: Interface dos Estados
2. **StateAdpater**: Implementa o interface IEstado.
3. **AwaitActionChoice**: espera pela escolha do jogador
4. **AwaitDraw**: espera que tire uma carta
5. **AwaitStart**: espera pelo início
6. **AwaitTunnelMovChoice**: Aguarda pelos movimentos dos tuneis
7. **GameOver**: Fim do jogo

<models>

1. **Game:** jogo
2. **Dice**: Classe responsável pelo lançamento
3. **DRM**: é uma classe enum

<models.cards>

1. **Card**: Classe responsável pela base da Carta
2. **EnemyTrackCard**: Classe representante da carta dos inimigos.
3. **StatusCard**: Classe representante do estado do forte (carta status).
4. **EventCard**: Classe responsável pelos dias dos eventos.
5. **Track**: Classe base para todas os caminhos existentes nas cartas de estado e inimigos.
6. **TrebuchetCount**: é uma track onde só muda o método toString de modo a aparecer segundo a imagem da carta onde se encontra.
7. **Tunnel**: é uma track com metodos adicionais de maneira a simular e facilitar a interacao com o tunel.

<model.events>

1. **EnemyMovement**: Classe base para representar os varios tipos de movimentos. As suas classes derivadas (2 e 3) implementam o método *void **apply**(Game game)* que se responsabiliza por avançar os inimigos.
2. **NoenemyMovement**: Representa a ausência de movimento.
3. **SlowestUnitMovement**: Move a unidade inimiga que se encontre mais longe.
4. **RegularMovement**: Move a unidade inimiga que se especifica na sua criação (WALL, GATES ou TREBUCHET).
5. **Event**: Classe base para representar os eventos. Contém um nome, descrição, pontos de acção e uma lista de ***EnemyMovement***s.
6. **TrebuchetEvent**: Representa um evento de trebuchet.
7. **RegularEvent**: Representa qualquer evento diferente do **TrebuchetEvent**. Todas as seguintes representam os diversos eventos descritos nas imagens fornecidas para as cartas e são classes derivadas de **RegularEvent** e implementam o método *void **apply**(Game game)*
    a. **BadWeather**
    b. **BoilingOil**
    c. **Collapsed**
    d. **CoverofDarkness**
    e. **DeathOfLeader**
    f. **DeterminedEnemy**
    g. **EnemyFatigue**
    h. **Faithbl**
    i. **FlamingArrows**
    j. **GateFortified**
    k. **GuardsDistracted**
    l. **Ilness**
    m. **IronShields**
    n. **Rally**
    o. **RepairedTrebuchet**
    p. **SuppliesSpoiled**
    q. **VolleyOfarrows**

Máquina de Estados

Criamos a nossa máquina de estados guiando nos pelo exemplo que estava no enunciado, fazendo as alterações que achamos necessárias de acordo com as regras do jogo.

Funcionalidades extras / Regras não implementadas

Não fizemos nenhuma funcionalidade extra.

As regras não implementadas foram apenas permitir eventos que limitam o turno a determinadas ações, ex: **BadWeather**.

As escolhas necessárias fazer ao escolher as ações “Rally Troops” e “Additional Action” não estão implementadas através de mudança de estado. No entanto a “Track Selection” está.

Conclusão

Numa análise final a esta primeira fase do trabalho, podemos confessar que não foi fácil a adaptação ao enunciado era confuso e com muitos tópicos/regras. Tivemos diversas dificuldades ao longo do trabalho que com o tempo conseguimos resolver.

Com este trabalho conseguimos adquirir e melhorar a nossa prática para a Programação em Java.