Projeto OOPS

(26/04/2024)

Grupo 7

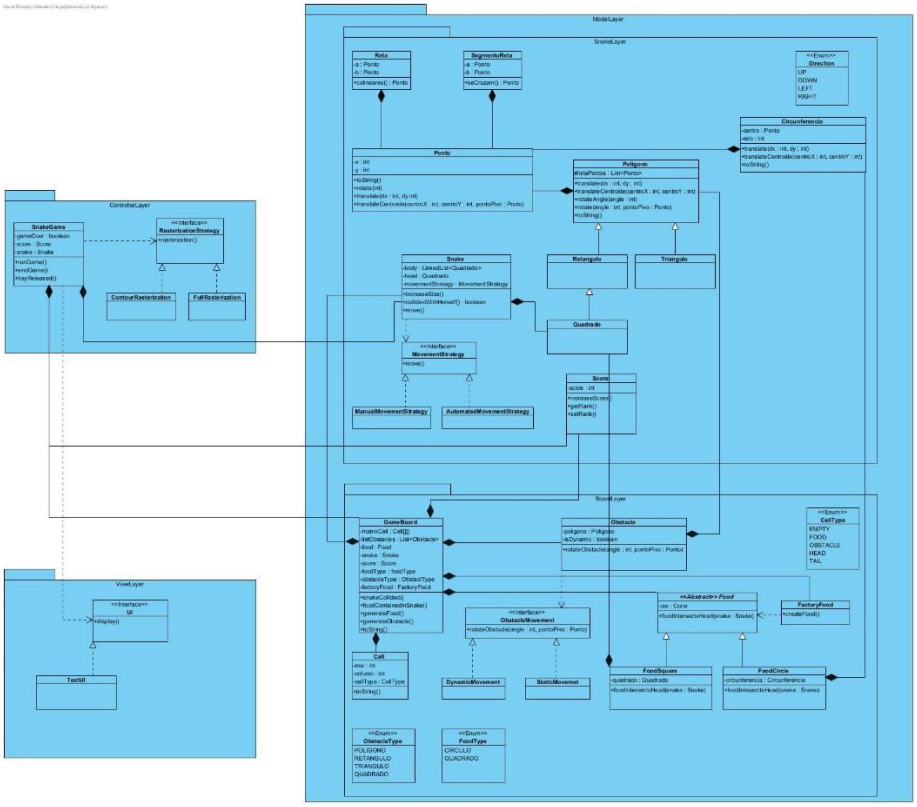
Eduarda Pereira (Nº79749), Hugo Conceição (Nº79722), João Ventura (Nº79882)

Programação Orientada a Objetos

Licenciatura em Engenharia Informática

Faculdade De Ciências e Tecnologias - Universidade Do Algarve

ii) O diagrama de classes UML inicial (ignorando a interface gráfica com o utilizador), incluindo uma breve descrição do diagrama e da responsabilidade de cada classe:



MODELLAYER

SNAKELAYER

DIRECTION

A enumeração referida apresenta as direções possíveis que a snake pode ter: para cima, baixo, esquerda ou direita; consoante os incrementos e decrementos de 90º.

MOVEMENTSTRATEGY

Esta interface define estratégias de movimento da cobra, especificando como a cobra se deve mover em resposta á próxima direção, levando em consideração o corpo atual da cobra, a direção atual e o comprimento da aresta da cabeça. Permite que diferentes estratégias de movimento sejam implementadas de forma flexível, promovendo a modularidade e a extensibilidade do código.

AUTOMATEDMOVEMENTSTRATEGY

A classe AutomatedMovementStrategy implementa a interface *MovementStrategy* e define uma estratégia automatizada de movimento para a cobra no jogo. O método *move()* desta classe é responsável por determinar o próximo movimento da cobra com base na direção atual, na próxima direção desejada, no corpo da cobra e no comprimento da aresta da cabeça. Facilita o controlo da cobra durante o jogo,

permitindo que ela se mova de forma eficiente e evite colisões com obstáculos ou com o próprio corpo.

ManualMovementStrategy

A classe implementa a interface *MovementStrategy,* proporcionando uma estratégia de movimento manual para a cobra no jogo. Define o método *move(),* que recebe a próxima direção que a cobra deve tomar e realiza o movimento correspondente. A estratégia é baseada na manipulação direta da cabeça da cobra, alterando sua posição e orientação de acordo com a direção solicitada. Além disso, a classe verifica se a direção solicitada é oposta à direção atual da cobra para evitar movimentos inválidos. Esta estratégia permite ao jogador controlar diretamente a cobra, fornecendo uma experiência interativa ao jogo.

Circunferencia

A classe Circunferencia, possui campos para armazenar o centro e o raio da mesma. O construtor da classe inicializa uma circunferência com um centro e um raio especificados, lançando uma exceção se o raio fornecido for menor ou igual a zero. A classe também inclui métodos para verificar se a circunferência interseta um polígono ou se está contida dentro de outra circunferência. Além disso, há o método que verifica se a circunferência está contida dentro de um polígono. O método *toString()* retorna uma representação em string da circunferência, indicando o centro e o raio. Esta classe é essencial para o cálculo de interseções e determinação de posicionamento no jogo.

Triângulo

A classe define um triângulo no plano cartesiano e verifica se os pontos fornecidos formam um triângulo válido. O construtor da classe permite criar um triângulo a partir de uma lista de pontos ou de uma string contendo as coordenadas dos vértices do triângulo. A validade do triângulo é verificada garantindo que a lista de pontos contenha exatamente três elementos.

Poligono

Esta classe, desempenha um papel fundamental na representação e manipulação de polígonos. A responsabilidade principal é representar um polígono, garantindo que os pontos fornecidos formam um polígono válido. Para isso, verifica se não existem três pontos consecutivos colineares e se as arestas não se cruzam, com a condição mínima de ter pelo menos 3 pontos. Além disso permite verificar interseções entre polígonos e determinar se um polígono está contido dentro de outro. Também calcula o centróide do polígono, facilitando operações de posicionamento e alinhamento. Outras funcionalidades incluem métodos para aplicar movimentos de rotação e translação, bem como a verificação igualdade entre polígonos, cálculo de hash codes e suporte á clonagem de polígonos.

Ponto

Classe responsável por representar um ponto no plano cartesiano, armazenando as coordenadas correspondentes e possibilitando criar um ponto com coordenadas especificadas. Alguns dos métodos incluem o cálculo de distância entre dois pontos, a aplicação de rotação e translação, cálculo de ângulo entre 2 pontos, verificação de alinhamento ou convexidade, e cálculo de área. Verifica também a igualdade entre pontos, calcula o hash code para fins de comparação e suporta a clonagem.

Retangulo

A classe tem como responsabilidade representar um retângulo e verificar se os pontos fornecidos formam um retângulo válido. Para ser considerado válido, o retângulo deve ter todos os ângulos internos medindo 90 graus e deve ser definido por quatro pontos. Os construtores da classe permitem criar um retângulo a partir de uma lista de pontos ou de uma string que representa os pontos do retângulo. Além disso, a classe contém um método privado para calcular o ângulo interno do retângulo com base em três pontos.

QUADRADO

A classe indicada é uma extensão da classe *Retangulo* e tem como responsabilidade representar um quadrado e verificar se os pontos formam um quadrado válido. Assim, herda as características da classe *Retangulo* e impõe uma condição adicional para garantir que todos os lados do quadrado sejam iguais, de acordo com sua definição geométrica. Os construtores permitem criar um quadrado a partir de uma lista de pontos ou de uma string com as coordenadas dos vértices do quadrado. Durante a criação do quadrado, a classe verifica se os lados são todos iguais.

Reta

A classe representa uma reta no plano cartesiano e é responsável por armazenar dois pontos que definem a reta. O construtor recebe dois pontos e verifica se são diferentes entre si, garantindo a validade da reta. Oferece também um método para verificar se três pontos consecutivos são colineares, ou seja, se estão alinhados na mesma reta. Os métodos *getA()* e *getB()* permitem aceder aos pontos que definem a reta, enquanto *setA()* e *setB()* permitem atualizar esses pontos, caso necessário.

Score

A classe é responsável por representar e administrar a pontuação do jogo. Possui um atributo Score` para armazenar o valor da pontuação e um atributo fileRank para representar o arquivo que contém o ranking dos jogadores. O construtor da classe permite criar um objeto Score com um valor inicial de pontuação especificado. Existem métodos para aumentar a pontuação em uma unidade, obter o ranking a partir de um arquivo, atualizar o ranking para um arquivo e obter ou definir o valor da pontuação. Esta classe é essencial para acompanhar e gerir o desempenho dos jogadores durante o jogo.

SegmentoReta

A classe é responsável por representar um segmento de reta no plano cartesiano e verificar sua validade. O construtor da classe permite criar um segmento de reta com dois pontos especificados. É possivel calcular o comprimento do segmento de reta, verificar se dois segmentos se cruzam e realizar operações como clonagem, igualdade e cálculo do hash code.

Snake

A classe representa a entidade cobra no jogo. Esta classe é responsável por manipular o corpo da cobra, a cabeça, a direção de movimento e a estratégia de movimento. O construtor da classe permite criar uma cobra com base numa lista de quadrados que representam o corpo e definir a direção inicial de forma aleatória. A cobra pode aumentar de tamanho, verificar se colidiu consigo mesma, e mover-se para uma nova direção.

BOARDLAYER

CellType

Esta enumeração define os tipos possíveis de células no tabuleiro. Cada tipo representa um estado diferente da célula: vazia, representa a cabeça ou a cauda da snake, contém comida ou obstáculo.

Cell

A classe representa uma célula que pode ser colocada num tabuleiro. Cada célula possui um tipo específico, como vazio, cabeça da cobra, cauda da cobra, comida ou obstáculo. O construtor da classe cria uma célula vazia por padrão, mas o tipo pode ser alterado posteriormente usando o método *setCellType()*. O método *toString()* retorna uma representação textual da célula com base no tipo, sendo representada por "H" para a cabeça da cobra, "T" para a cauda da cobra, "F" para comida, "O" para obstáculo e "." para célula vazia.

DynamicMovement

A classe implementa a interface *ObstacleMovement*, fornecendo uma implementação para o método *rotateObstacle()*.

FactoryFood

Responsável por criar instâncias de alimentos com base na forma geométrica fornecida como parâmetro, contém o método que aceita um objeto de forma genérica como entrada e retorna um objeto de comida correspondente.
Se o objeto de forma for uma instância de *Circunferencia* é criado e retornado um objeto *FoodCircle*, caso seja uma instância de *Quadrado*, cria-se e retorna-se um objeto *FoodSquare*; e se a forma não for reconhecida é lançada uma exceção *IllegalArgumentException*.

FoodType

A enumeração caracteriza os diferentes tipos de comida disponíveis no jogo: circular ou quadrada.

Food

Esta classe representa um tipo genérico de alimento no contexto do jogo, sendo constituída por um atributo `color` que especifica a cor do alimento (por defeito é amarelo). Existem dois métodos abstratos na classe: o método *foodContainedInHead(Snake snake)* que verifica se o alimento está contido na cabeça da cobra, retornando verdadeiro ou falso; e o *foodIntersectObstacle(Obstacle obstacle)* que verifica se o alimento interseta com algum obstáculo no tabuleiro, retornando verdadeiro ou falso. Essa classe serve como uma base para diferentes tipos específicos de alimentos, como círculos de comida ou quadrados de comida, que implementam esses métodos de acordo com exigências.

FoodCircle

A classe está intimamente relacionada com a classe abstrata *Food*, da qual herda. A classe *Food* define o comportamento geral de um alimento no tabuleiro do jogo, enquanto a *FoodCircle* pormenoriza esse comportamento para um tipo específico de alimento, neste caso, um círculo.

FoodSquare

Tal como, esta classe está intimamente relacionada com a classe abstrata Food, da qual herda. A classe Food define o comportamento geral de um alimento no tabuleiro do jogo, enquanto a FoodCircle pormenoriza esse comportamento para um tipo específico de alimento, neste caso, um quadrado.

GameBoard

A classe apresenta a estrutura principal do ambiente do jogo, entre eles a cobra, os obstáculos e os elementos de comida no tabuleiro. Encapsula a lógica para gerar obstáculos e comida, detetar colisões com a cobra e atualizar a pontuação. Através dos alguns métodos, garante a integridade do jogo verificando as interações da cobra com obstáculos, garantindo que a cobra não colide consigo mesma ou com os limites do tabuleiro, assim como o posicionamento e consumo de comida. No geral, a *GameBoard* orquestra os elementos dinâmicos do jogo, proporcionando a base para uma experiência de jogo eficiente.

Obstacle

A classe representa os obstáculos no tabuleiro do jogo. Cada obstáculo é definido por uma figura geométrica, encapsulada pelo polígono. Dependendo da configuração, um obstáculo pode ser estático ou dinâmico, determinando se ele se move ou não durante o jogo. Regula também interações com a cobra, verificando se a cabeça da cobra colide com o obstáculo ou se está contida dentro dele. Além disso, fornece métodos para rodar o obstáculo, se necessário. Essa classe é essencial para criar desafios adicionais no jogo, adicionando variedade e complexidade ao tabuleiro.

ObstacleType

Esta enumeração oferece uma seleção de tipos de obstáculos no jogo, entre eles poligonos, retângulos, triângulos e quadrados. Tornando o jogo mais desafiante e complexo.

ObstacleMovement

A interface define os movimentos de obstáculos no jogo. O método *rotateObstacle* permite girar um obstáculo em torno de um ponto de pivot, especificando um ângulo de rotação e o ponto em torno do qual o obstáculo deve ser girado. Esta interface é implementada através de classes que controlam o movimento dinâmico dos obstáculos no tabuleiro.

StaticMovement

A classe implementa a interface *ObstacleMovement*, fornecendo uma implementação do método *rotateObstacle*. Ou seja, os obstáculos que usam esse tipo de movimento permanecem estáticos e não são afetados por rotações durante o jogo. De modo geral, esta classe é útil para definir o comportamento de obstáculos que não se movem ou giram.

CONTROLLERLAYER

RasterizationStrategy

Esta interface declara um método *Rasterization()*, que todas as classes que implementam essa interface devem fornecer uma implementação para realizar a rasterização de acordo com a estratégia específica. Isso permite que diferentes estratégias de rasterização sejam facilmente substituídas ou alternadas no sistema, seguindo o princípio do polimorfismo.

ContourRasterization

A classe implementa a interface *RasterizationStrategy*, e converte as coordenadas dos pontos dos contornos em pixels na tela de exibição.

FullRasterization

A classe implementa a interface *RasterizationStrategy* e é responsável pela estratégia de rasterização completa. A rasterização completa envolve o preenchimento de formas, como polígonos, com uma cor específica, garantindo que todos os pixels dentro da forma sejam coloridos corretamente.

VIEWLAYER

UI

Esta interface implementa a User Interface, ou seja, ela é responsável por definir um contrato que específica os métodos que devem ser implementados por qualquer classe que represente esta interface.

TextualUI

A classe TextualUI é responsável por apresentar o jogo de forma textual para o usuário, implementando os métodos definidos na interface que ela herda.

iii) Todas as opções de projeto tomadas, designadamente padrões de projeto:

É possível utilizar padrões de projeto para tornar o código mais eficiente.
Um dos padrões que adotámos inicialmente é o padrão MVC (Model-View-Controller), este padrão divide uma aplicação em 3 componentes principais: o Modelo, a Visão e o Controlador.

Começando pelo Modelo, ele é responsável pela lógica do jogo e pela manipulação dos dados, como a movimentação da cobra, gerar comida e obstáculos, etc. Neste caso o modelo inclui as classes Snake, Food, Obstacle, Score, GameBoard.

Passando para a Visão, a mesma representa a interface do usuário (UI) e como os elementos do jogo são apresentados ao jogador. Isso inclui todo o aspeto gráfico do jogo como a representação visual da cobra, da comida, dos obstáculos, etc e tudo o que interage com o utilizador. A classe responsável por isso é a UI.

Por último, temos o Controlador, este atua como intermediário entre o modelo e a visão, recebe a entrada do usuário e atualiza o modelo ou a visão conforme necessário. Por exemplo, quando o jogador pressiona uma tecla para mover a cobra, o controlador recebe esse input e atualiza a posição da cobra no modelo. No nosso projeto a classe responsável por isso é a classe SnakeGame.

A separação clara entre modelo, visão e controlador oferece várias vantagens:

- **Separação de Responsabilidades**: Cada componente (modelo, visão, controlador) tem uma responsabilidade específica no aplicativo, facilitando a compreensão do código e a manutenção futura.
- **Testabilidade**: Os componentes individuais podem ser testados de forma independente, o que simplifica a escrita de testes unitários e a deteção de bugs.
- **Flexibilidade**: O padrão MVC facilita a extensão e a modificação do aplicativo, pois as alterações em um componente não afetam diretamente os outros componentes.

Adotamos também o padrão Factory Method na classe Food. Este padrão encapsula a criação de objetos deixando as subclasses decidirem quais objetos criar, o uso do padrão oferece uma maneira flexível de criar instâncias de diferentes tipos de comida, como FoodCircle e FoodSquare, sem que o cliente precise conhecer os detalhes da implementação de cada uma.

Além disso o padrão Factory Method facilita a extensão do código, pois podemos adicionar novos tipos de comida simplesmente criando subclasses de “Food”, sem termos de mudar a classe Food. Dando assim a responsabilidade da sua lógica de criação à subclasse em si e tirando essa responsabilidade da classe Food. Utilizamos esse padrão de projeto devido a uma comida poder ser círculo ou quadrado.

Outro projeto de padrão que também usamos foi o padrão Strategy, está implementado no movimento da snake, no modo de rasterização, o movimento do obstáculo, ou seja, na sua rotação e ainda na relação do jogo com a user interface.

O padrão de projeto Strategy é utilizado quando se deseja definir uma família de algoritmos encapsulá-los e torná-los trocáveis. Ele permite que o algoritmo varie independentemente dos clientes que o utilizavam.

O padrão Strategy permite, por exemplo, que a interface “MovementStrategy” represente uma família de algoritmos ou estratégias de movimento para a cobra no jogo. Isto significa que a interface “MovementStrategy” define um conjunto de métodos que são comuns a todas as estratégias de movimento, mas essas estratégias podem ser implementadas de maneira diferentes em classes concretas. Isto torna o código mais reutilizável.

iv) Testes Unitários JUnit (localizados na pasta Testes)