ISEL | DEETC

Programação Orientada por Objectos

3º Trabalho Práctico

Semestre de Verão de 2009/2010

Autores:

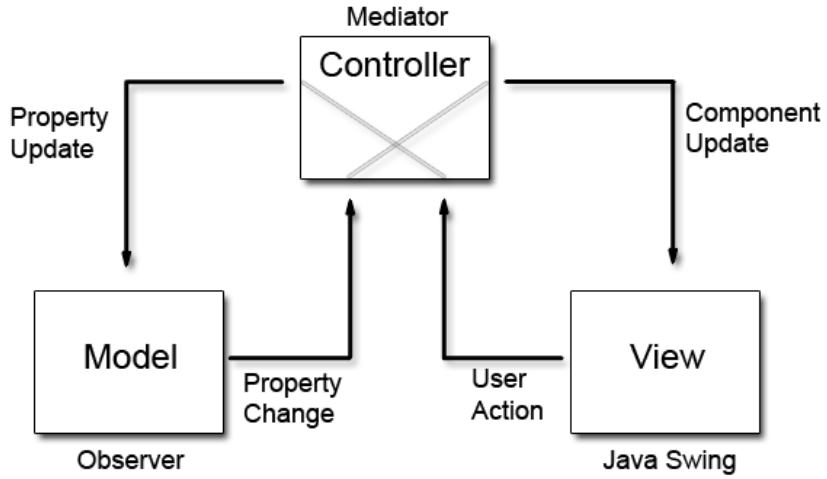
33595 – Nuno Sousa

1 - Introdução

O trabalho consiste em fazer uma versão do jogo SameGame com alguns requisitos específicos.

Como sugerido no enunciado, e sendo a opção que faz mais sentido tentei projectar o UML do jogo de modo a implementar o modelo MVC – Model View Controler.

Após estudar os prós e contras deste modelo, e de modo a tornar a User Interface ainda mais independente do Modelo de Dados resolvi implementar o modelo MVC modificado em que a interface nunca acede directamente ao modelo de dados como mostrado na figura seguinte:



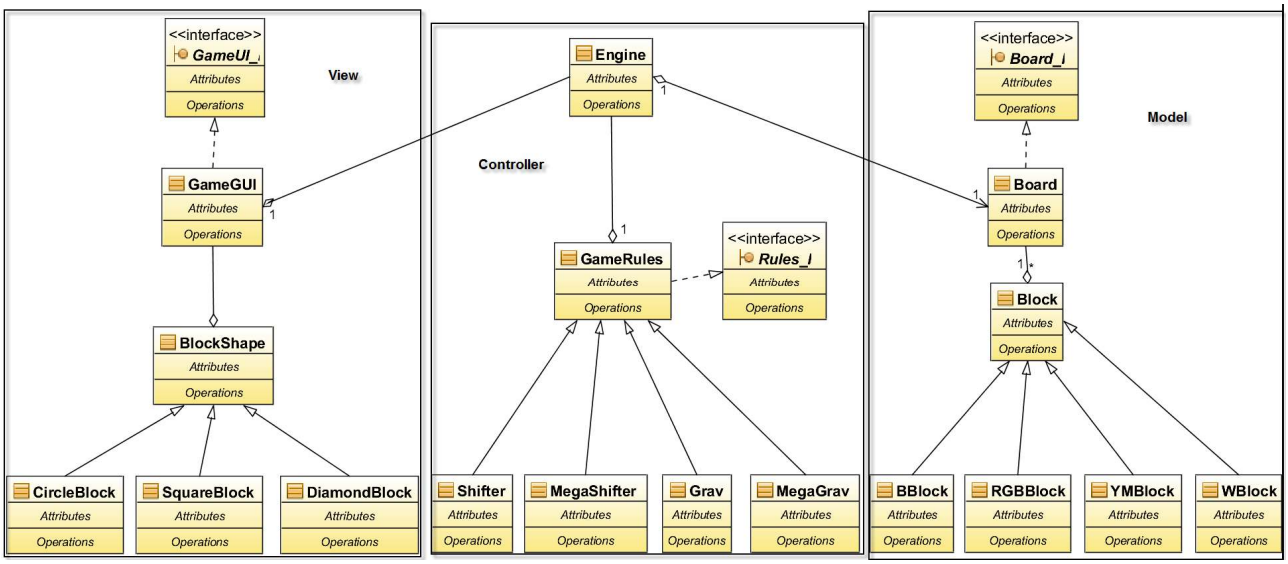
Este modelo permitie uma idependência maior entre o modelo de dados e a interface embora tenha o problema do controler (engine) ter de implementar os métodos que consultam o modelo de dados para informar a interface do estado dos mesmos quando a interface pedir.

Uma alterção efectuada a este modelo no projecto realizado prende-se com o modelo de dados não informar o controlador da alteração dos mesmos uma vez que os métodos que alteram os dados são muito específicos e por isso o controlador sabe que ao invocar esses métodos os dados irão, quase de certeza, ser alterados e por isso informar a interface do facto.

Uma vez que é possível ter vários tipos de jogo, vários tipos de blocos e vários desenhos de blocos implementei uma solução que permite criar novos tipos de jogo, blocos ou desenhos de blocos e apenas é ncessário colocá-los dentro das pastas respectivas para que o jogo ao iniciar carregue os mesmos e fique pronto a utilizá-los. O Engine é responsável por carregar as opções e disponibilizar estas ao Modelo de Dados e à Interface.

2 - UML

Após analisar o tipo de jogo e os requisitos apresentados o UML de onde parti para o desenvolvimento do trabalho foi o seguinte:



Inicialmente as classes “Mega” extendiam as “normais” mas após consultar a professora chegámos à conclusão que isso não permitia criar um tipo de jogo que fosse “mega” sem exisitir o jogo base.

2 – Estrutura da Aplicação

Uma vez que a aplicação está separada em 3 partes vamos analisar cada uma independentemente:

1) O modelo de Dados

O modelo de dados assenta num Board que tem a grelha com o tamanho da grelha do jogo e que é responsável por todas as operações de manipulação dos blocos dentro da grelha, nomeadamente, adicionar, remover, mover, seleccionar (guardando uma lista de blocos selecionados).

Uma vez que seria necessário no âmbito do jogo ter métodos que efectuassem o Shift dos blocos para baixo e para o lado foram implementados no board estes métodos de modo a que fossem estanques e, deste modo, evitassem o lançamento de excepções quando da criação dos jogos.

Os blocos são estruturas muito simples que apenas guardam as informações sobre a sua cor e as suas regras de selecção. Para implementar as regras de selecção usei um array 3x3 de booleanos que indica ao board em que sentido deve efectuar a selecção dos blocos e a posição central indica se o Board deve ter em conta se o bloco contíguo é igual ou não ao bloco em questão.

Este método torna a selecção dos blocos possível de ser efectuada com métodos recursivos bastante mais simples do que se tivéssemos de ter em conta todos os problemas de selecção dos blocos.

O Board também disponibiliza métodos para guardar o estado do jogo para permitir efectuar undo.

Uma vez que existem blocos especias (Brancos e Pretos) que permitem a resolução do jogo sem grande dificuldade na inicialização do tabuleiro foi tido em conta essa questão e foi ponderado o valor aleatório desses blocos para aparecerem menos vezes que os restantes. Além disso, existe um valor máximo para este tipo de blocos aparecer no board.

No modelo de dados encontra-se também a classe para gerir os HighScores.

Foi incluída a funcionalidade de rotação do tabuleiro de jogo no Modelo de dados para poder ser utilizada pelo engine sem ter de ter a preocupação de como efectuar a mesma de acordo com as regras do próprio tabuleiro.

2) O Controller (Engine)

O Engine controla as funções básicas de jogo e tem a ligação ao modelo de dados e à Interface.

As regras do jogo foram desenvolvidas à parte para permitir mudar de jogo sem ter de reiniciar o mesmo. Estas regras é que sabem exactamente o que fazer quando se clica num bloco (seleccionar, remover, shift, etc.)

É da responsabilidade do engine (através da classe FileAccess) recolher os dados gravados no ficheiro, os tipos de jogo e blocos disponíveis nas respectivas pastas e gravar os dados no ficheiro ao sair do jogo.

Caso exista erro a ler as opções do jogo, ao ler o jogo gravado e/ou ao ler os high scores, dependendo de onde existiu o erro o jogo é iniciado com opções de defeito permitindo continuar a jogar um jogo novo mesmo que os ficheiros tenham erro.

Uma vez que o modelo de dados disponibiliza a maioria dos métodos necessário para implementar os jogos tornou-se muito fácil criar as regras dos mesmos.

Foi criada a classe Abstracta para poder estender da classe Observable para que o UI pudesse ser notificado das alterções ao ambiente de jogo.

3) A interface gráfica

Na interface gráfica optei por dividir em várias classes o ambiente de modo a ser mais fácil controlá-lo. Implementa Observer e adiciona-se como observador do Engine para actualizar quando for necessário. Penso que esta solução é melhor do que estar de x em x milissegundos a actualizar tudo.

Do mesmo modo do que para o tipo de blocos, foram criados vários designs gráficos para os mesmos. A interface gráfica foi implementada de modo a poder ter facilmente vários layouts, e blocos de várias formas.

A questão principal no desenvolvimento da parte gráfica (que foi a que ocupou mais tempo) foi o tempo que demora a testar soluções, e verificar o funcionamento correcto das frames, panels, etc.

Apenas não foi terminado o desenvolvimento de perguntar ao jogador o nome, no caso de ter obtido highscore e a visualização dos mesmos. A estrutura está concluída mas mostra os highscores (quando adiciona o mesmo) na consola.

Conclusão

Este trabalho levantou muitas questões quer a nível da melhor estrutura de classes que iria permitir desenvolver o mesmo de modo a poder ser alterado sem grandes problemas, quer a nível de optimização de programação e dos algoritmos para permitir efectuar as operações exigidas sem uma grande quantidade de código.

Tentou-se também fazer com que erros de ficheiros, de classes inacessíveis, etc. Não impedissem o jogo de funcionar excepto quando se tratam de ficheiros essenciais ao jogo e que não possam ser substituídos.

Permitiu utilizar quase toda a matéria da unidade curricular de POO, nomeadamente:

- algoritmos recursivos
- acesso a ficheiros para escrita e leitura
- utilização de estruturas dinâmicas (neste caso LinkedLists)
- tipos de dados genéricos
- a herança e o polimorfismo está presente em toda a estrutura do trabalho
- tratamento de excepções

Foi um trabalho que me ensinou bastante como organizar as classes de um projecto mais complexo e que fui aprendendo sempre com as opções/erros tomados.

Bibliografia

Savitch, Walter - Java: An Introduction to Problem Solving and Programming, 4/E, Prentice Hall

http://en.wikipedia.org/wiki/SameGame

http://java.sun.com/javase/6/docs/api/

http://download.oracle.com/javase/6/docs/api

http://blogs.sun.com/JavaFundamentals/entry/java_se_application_design_with

http://en.wikipedia.org/wiki/Model-view-adapter

http://en.wikipedia.org/wiki/Observer_pattern#Java