Instituto Superior de Engenharia de Lisboa

PROGRAMAÇÃO ORIENTADA POR OBJETOS

Relatório do Primeiro Trabalho Prático

Docente

Prof. José Simão

Alunos

43552 Samuel Costa

43884 Pedro Rocha

Abril 2017

Índice

Introdução

Este relatório, referente ao primeiro de três trabalhos práticos, foi elaborado no âmbito da unidade curricular de Programação Orientada por Objetos e acompanha a implementação de um jogo em consola de texto designado Circuit. Este exercício constituiu uma aplicação da abordagem de desenvolvimento **MVC**, bem como de outros aspetos da programação orientada por objetos, designadamente, os conceitos de herança e polimorfismo.

Ao longo do documento apresentam-se os principais objetivos metodológicos e práticos delineados pelo problema em questão; a metodologia adotada; uma explicação dos procedimentos aplicados no desenvolvimento da aplicação; e, por fim, uma breve reflexão sobre o trabalho realizado em que se discutem os resultados obtidos, explicitando-se, conclusões referentes ao trabalho desenvolvido.

Objetivos

O trabalho prático visa, em termos gerais, aplicar as ferramentas introduzidas em contexto de sala de aula, em particular, os conceitos de herança e polimorfismo, aprofundando as suas potencialidades de utilização, por meio do desenvolvimento de uma aplicação para jogar um jogo chamado Circuit, que permite desenhar as pistas que fazem as ligações entre terminais da mesma cor de um circuito impresso.

Definem-se os seguintes objetivos:

- do ponto de vista metodológico: identificar os procedimentos a aplicar no desenvolvimento da aplicação seguindo a abordagem model-view-controller (**MVC**);
- do ponto de vista prático, empregar e dominar os seguintes instrumentos da linguagem java: (i) respeitar a sintaxe do JAVA e as orientações de conceção de classes; (ii) fazer a utilização adequada de tipos enumerados; (iii) reconhecer a diferença entre estado e comportamento de um objeto; (iv) conceber classes base e classes derivadas; e por último (v) compreender e utilizar convenientemente os serviços fornecidos por uma hierarquia de classes.

Métodos e Recursos utilizados

Para a realização deste trabalho foi utilizada a linguagem JavaTM Versão 8 Update 131. Foi também utilizado o software PlantUML Versão 2017.11, bem como o ambiente de desenvolvimento IntelliJ IDEA Community Edition Versão 2017.1 com os plugins PlantUML Integration Versão 2.6.0 e PlantUML Syntax Check Versão 0.2.0. Todos os elementos da linguagem necessários para a resolução dos problemas propostos foram adquiridos nas aulas da unidade curricular Programação Orientada por Objetos (POO).

Regras do Jogo

O objetivo do jogo é desenhar pistas de um circuito impresso, ligando entre si terminais da mesma cor com pistas. Cada posição do circuito não pode ser ocupada por mais de uma pista e algumas posições no circuito podem ter restrições, isto é, apenas podem ser preenchidas por pista verticais ou horizontais.

O jogo é composto por níveis já compostos, e é possível passar para o nível seguinte, sempre que o jogador tenha completado o nível que está a jogar.

Considera-se o nível completado, quando todas as posições do circuito estão preenchidas com pistas que ligam os terminais da mesma cor entre si e todos os terminais estão preenchidos. Ressalva-se que cada terminal só pode ter uma ligação.

O jogo acaba quando o jogador desiste ou não houverem mais níveis para serem carregados.

MVC (Model-view-controller)

MVC é um padrão de arquitetura, que permite conceber o desenvolvimento de aplicações reduzindo a complexidade, favorecendo a manutenção e a facilidade de leitura, por redução do projeto a três componentes funcionais da aplicação:

Model – (modelo), é um repositório da informação necessária para a aplicação (ou parte dela), mantém constantemente atualizada toda a informação, que poderá passar a outros componentes, ou apenas aguardar que outros componentes lhe peçam informação.

View – (vista), é responsável pela representação gráfica do modelo.

Controller – (controlador), efetua a ligação entre Model e View nas interações do sistema com o exterior, controlando as alterações ao modelo, reagindo a essas alterações, e instruindo a vista para as representar.

Na figura 1 é apresentado um diagrama, baseado na linguagem UML- Unified Modeling Language, que explicita a relação entre as diversas classes que fazem parte do programa desenvolvido, evidenciando essas três componentes da aplicação identificadas durante o desenvolvimento:

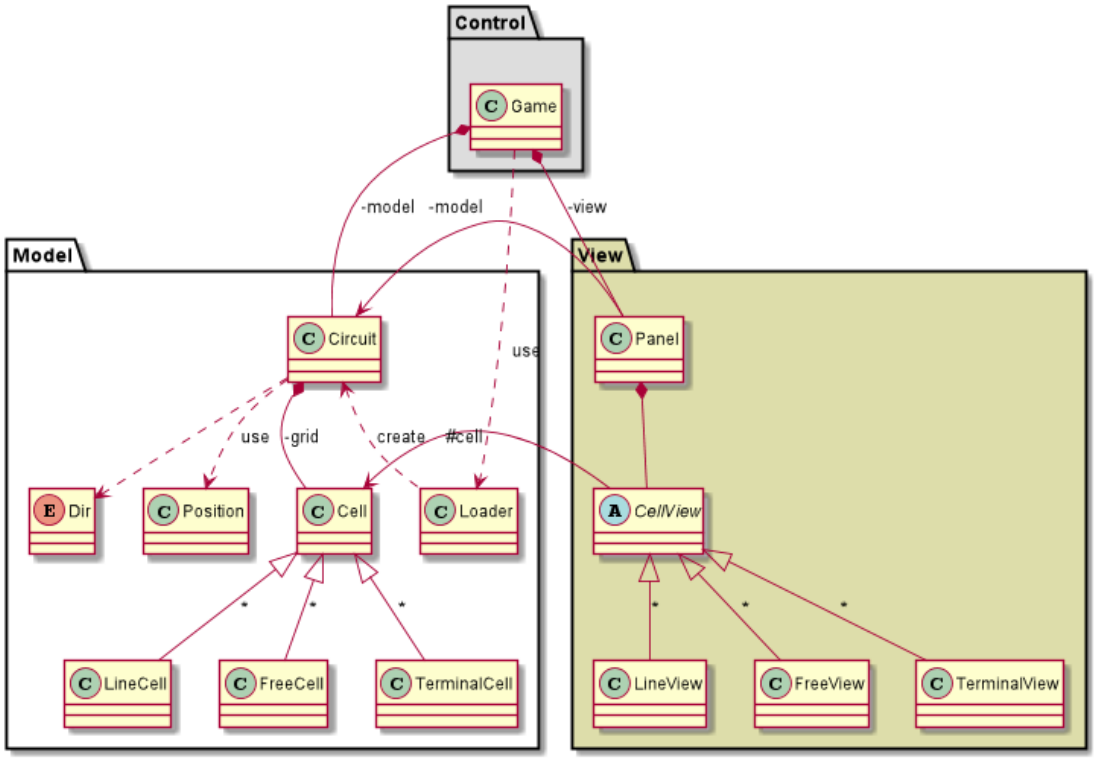

Figura 1 - Diagrama de Classes UML

Abstração do Circuito num *Array*

Para monitorizar o estado do jogo, e facilitar a realização de pesquisas ou modificações ao estado do tabuleiro de jogo, tanto a classe Panel como a classe Circuit têm campos grid, uma matriz bidimensional com as dimensões do tabuleiro. No caso de Circuit, o campo grid é uma matriz de referências para Cell, e em Panel, uma matriz de referências para CellView.

Cada célula tem a informação sobre o seu preenchimento (campo filled e um conjunto de enumerados dirConnected).

No inicio do jogo, só os terminais estão preenchidos, como evidenciado na figura 2, sendo que, à medida que o jogador vai realiza ações com o rato, a informação sobre o preenchimento das células é atualizada em função da interação com o jogador.

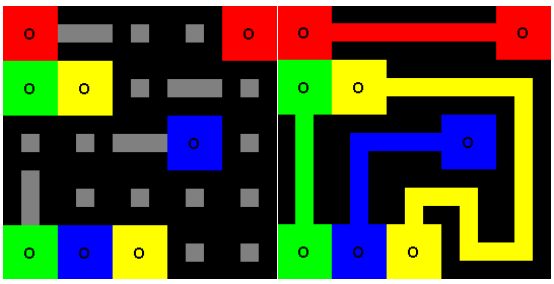

Figura 2 - Tabuleiro no início do jogo e resultado de uma sucessão de ações do jogador sobre a janela de Consola com o rato.

Procedimentos Aplicados

É a organização do programa em packages e nas suas classes, apresentada pelo diagrama de classes UML, que será plasmada na descrição dos procedimentos aplicados. Como este relatório se fará acompanhar do código desenvolvido para a implementação do projeto, esta secção dará ênfase à descrição dos serviços disponibilizados nas diferentes classes e do seu contributo para a execução da aplicação, isto é, não são feitas referências explicitas ao código escrito, salvo em duas ocasiões, como se verá, na classe Circuit.

A sequência em que é feita a descrição pretende facilitar a correspondência entre a composição do projeto e a execução do programa.

Controller

A classe *Game* é o núcleo da componente Controller do projeto. Esta classe gere o cumprimento das regras do jogo, traduzindo-as para a linguagem de programação. (ver "Regras do jogo" no capítulo anterior).

De cada vez que é carregado um novo nível, é chamado o método load() da classe Loader sobre um objeto dessa classe iniciado com uma leitura do ficheiro de níveis.

Enquanto o jogador estiver a jogar um dos níveis, o jogador pode interagir com o programa por meio da tecla ESC do teclado ou por uma ação sobre o rato (tratada como MouseEvent).

As coordenadas da consola em que ocorreu o MouseEvent são convertidas em coordenadas na grelha e guardadas numa referência para Position. Se a posição for válida o evento do rato é processado, consoante o tipo de evento e a posição da grelha em que ocorreu.

Podem existir três tipos de ação sobre o rato, e estas que geram reações diferentes no programa. Quando o rato é pressionado, a célula selecionada aparece com destaque. Quando o rato é arrastado, a célula selecionada é mostrada com destaque, a anterior é mostrada sem destaque e são afetadas as células com ligações ou desconexões consoante o seu estado atual. Quando é libertado o rato, a célula selecionada é mostrada sem destaque. No caso de o rato ter sido libertado na posição em que foi acionado, a célula é desconetada na direção pressionada.

Model

a) Loader

É por recurso a um objeto da classe Loader que cada nível é carregado, como ficou explicado acima. O método load(int) retorna uma referência para um objeto do tipo Circuit referente ao circuito do nível passado como parâmetro. Este método lança uma exceção do tipo LevelFormatException se o carregamento não tiver sido efetuado com sucesso, sendo mostrada

ao utilizador uma mensagem especificando o erro, e em alguns casos, em que linha do ficheiro de leitura foi encontrado o erro.

b) Cell

Com a classe abstrata Cell pretende-se que todas as células de tipos derivados de Cell (FreeCell, LineCell, TerminalCell) tenham um comportamento similar que será detalhado em cada implementação:

IsFull() – retorna indicação de que a célula esta ou não cheia, útil para saber se o nível esta preenchido e para evitar mais conexões com outras células.

canConnectFrom() – retorna indicação de ser possível ou não a célula ser conectada de um determinado terminal e de uma determinada direção.

canDisconnetfrom() – retorna indicação de ser possível ou não a célula ser desconectável de um determinado terminal e de uma determinada direção.

ConnectFrom(), disconnecFrom() – efetua a conexão/desconexão de uma terminal e direção.

c) Dir

No tipo enumerado Dir, que representa as direções de conexão possíveis, foram implementados os métodos de apoio:

deltaLin() – devolve a variação de posição de linha da célula no circuito em função da direção de conexão em relação ao centro da célula, isto é, se a célula tem uma conexão com a direção UP, implica que a célula conectada a si esta na posição linha+ deltaLin (linha + -1).

deltaCol() – O mesmo que deltaLin() mas para colunas.

complement() – devolve a direção contrária.

d) Position

Na classe Position, que representa uma determinada posição de Cell no Circuito, foram implementados alguns métodos de apoio:

equals(Position) – devolve true se o campo line e col são iguais entre duas posições.

newPositionFromDirection(Position, Dir) – devolve uma nova posição tendo por referência a posição atual e uma direção.

e) Circuit

A classe Circuit agrega os serviços disponíveis do package Model para o exterior. Foram implementados alguns métodos de apoio:

isOver() - devolve indicação de que o circuito esta totalmente preenchido (nível completo)

drag() - método invocado pelo Controller após ação do rato drag, que recebe duas posições. Se a célula de onde se pretende ligar estiver preenchida, determina a posição relativa das duas células identificadas pelas posições passadas como parâmetro. Se a posição for contígua, avalia se a célula de destino tem capacidade para ser ligada e nesse caso efetua a ligação entre as duas células nas direções correspondentes. Caso contrário, desliga as duas células. A listagem 1 apresenta parte do código escrito neste método para avaliar se as duas posições são contíguas e qual a posição da célula de destino em relação à célula de origem. Pensou-se que este trecho de código poderia ser implementado como um método relativePosition(Position) na classe Position com tipo de retorno Dir. Porém, uma vez que esta é a única ocorrência deste trecho de código e o método em que está inserido só é chamado num lugar do programa, optou-se por não fazer esta mudança.

```
        ...
if (fillFrom>0 && !cellFrom.isFull()) {
    int dif=0;
    Dir dirFrom = null;
    if (from.getLine()==to.getLine()) {
        dif=(from.getCol()-to.getCol());
        if (dif==1) dirFrom = Dir.LEFT ;
        else if (dif==-1) dirFrom = Dir.RIGHT;
        else dif=0;
    } else if (from.getCol()==from.getCol()){
        dif=(from.getLine()-to.getLine());
        if (dif==-1) dirFrom=Dir.DOWN;
        else if (dif==1) dirFrom=Dir.UP;
        else dif=0;
    }
...
```

Listagem 1 – parte da implementação de drag(Position, Position);

connectCells e disconnectCells – são dois métodos chamados pelo método drag que tentam conectar ou desconectar as células e devolve verdadeiro se foi possível.

unlink – método que é invocado por Controller após uma ação do rato UP. Desconecta a célula na posição passada como parâmetro e verifica se não ficam pistas no ar, isto é, verifica se as pistas estão ligadas ou não a um terminal. Se não estiverem ligadas a um terminal, desconecta toda essa pista.

disconnectIfLastNotTerminal – Recebe como parâmetro o caracter com que a célula de origem está preenchida, a posição da célula de origem na grelha, a direção em que se quer desconectar a célula de origem. Se a célula for um terminal devolve imediatamente falso. Em

seguida, obtém, para cada direção em que a célula de origem está conectada, a posição da célula adjacente nessa direção. É chamado recursivamente este método para a nova posição, com o caracter de preenchimento e a direção complementar à direção em avaliação. Se retornar verdadeiro, são desconetadas as duas células envolvidas. Se a chamada recursiva retornar falso, o método retorna falso. Se a célula de origem não estiver ligada em mais nenhuma direção, retorna verdadeiro. A listagem 2 apresenta em detalhe a implementação deste método.

```
private boolean disconnectIfLastNotTerminal(char fillFrom,Position
pos,Dir dirFrom){
    Cell cellFrom=this.getCellFromPosition(pos);

    if (!(cellFrom instanceof TerminalCell)) {
        for (Dir d : cellFrom.dirConnected) {
            if (d != Dir.EMPTY && d != dirFrom) {
                Position newPos = pos.newPositionFromDirection(d);
                if
(this.disconnectIfLastNotTerminal(fillFrom,newPos,d.complement())) {
this.getCellFromPosition(newPos).disconnectFrom(fillFrom,d.complement());
                    cellFrom.disconnectFrom(fillFrom,d);
                    return true;
                }
                return false;
            }
        }
        return true;
    }
    return false;
}
```

Listagem 2 – implementação do método disconnectIfLastNotTerminal(char,Position,Dir);

View

A classe Panel agrega os serviços disponíveis do package View para o exterior.

No início do jogo, é aberta uma janela de Consola com dimensões adequadas à representação do modelo do jogo, por recurso ao método open(String, int, int). A grelha do modelo é transposta, ou seja, é iniciada uma grelha de referências para tipos compatíveis com CellView, e é mostrado o estado inicial do tabuleiro, por recurso ao método repaint().

O tempo decorrido desde o início do jogo é mostrado na última linha da janela por chamada do método repaintTime(). Para tal, é guardada uma referência temporal de início de jogo, que é subtraída ao instante atual. É decomposto o resultado em minutos e segundos.

Sempre que se quer pintar uma célula do circuito a partir de coordenadas da consola, tal é possível, pois foi escrito um método que converte coordenadas da consola numa posição na grelha getModelPosition(int, int), e outro que retorna uma referência para um tipo compatível com CellView dada uma posição da grelha.

Uma célula pode ser mostrada com destaque (com as posições em que não pode ser preenchida pintadas de cinzento claro), ou sem destaque (com as mesmas posições pintadas a

preto). Para além disso, cada terminal é mostrado de uma cor diferente. Terminais do mesmo tipo são mostrados com a mesma cor. Para além disso, células não preenchidas são mostradas a cinzento escuro nas direções em que podem estar ligadas, mas não estão.

Optou-se por efetuar a pintura das células na janela por camadas, para facilitar a leitura do código escrito. Assim, para pintar células de tipos derivados de CellView, é primeiro averiguado se a célula é para ser mostrada com destaque ou não, e toda a superfície da célula é pintada com a cor correspondente. No caso de objetos do tipo TerminalView, como a vista das células do tipo Terminal não sofre alterações durante o jogo, toda a superfície é pintada com a cor do Terminal, marcando-se de seguida o centro da célula com a letra O. Para objetos do tipo FreeView é pintado o centro, e em seguida as direções em que a célula está ligada. Finalmente, para objetos do tipo LineView, é pintado o centro e as direções em a célula está ligada, da cor do terminal a que se ligou, sendo que, por ultimo se pintam de cinzento escuro as direções que o tipo da célula admite ligação, mas esta não existe.

Conclusões

Com a realização deste trabalho, foi possível estruturar o desenvolvimento em torno de três componentes de um projeto. Constatou-se que a abordagem MVC torna o programa mais flexível, mais fácil de desenvolver, permitindo a um programador desenvolver parte do código sem conhecer todo o código do programa. Permite ainda a reutilização do código. De fato, espera-se que parte do código desenvolvido venha a integrar um dos trabalhos práticos a realizar durante o semestre.

Foi também possível adquirir maior familiaridade com os conceitos de herança e polimorfismo. Em primeira fase, observou-se no método equals(Position), redefinido a partir do método equals da classe Object, um primeiro caso de estudo de polimorfismo.

Tendo como ponto de partida a invariância observada em objetos diferentes, foi possível construir classes de objetos que, apesar de partilharem uma base de atributos e serviços disponíveis (surgindo daí a noção de hierarquia de classes e, assim também de herança), podem implementar serviços e dispor de características distintivas. Foi este o pensamento que orientou a representação das células do jogo e da sua vista, pela conceção de duas hierarquias de classe, em que: se definiram métodos nas classes derivadas por remissão à classe base, ou se redefiniram métodos nas classes derivadas, ou mesmo, simplesmente se desenvolveram métodos na classe base, que pela noção mesma de herança, ficam definidos para as classes derivadas.

A abstração do circuito num array de referências para Cell ou CellView tornou-se, assim, possível pelo polimorfismo, dado que as referências para tipos-base puderam representar adequadamente o comportamento de objetos de classes suas derivadas, dada a sua compatibilidade.

Também se observaram algumas implicações do encapsulamento, pois o diagrama de classes proposto trazia sugestões em termos de dependência e partilha de informação dentro do projeto. Esta noção, que permite estabelecer como que um “biombo” entre o código de cliente e o código do programador, determinando qual a informação que está disponível ou pode ser modificara por aquele, não impôs, neste projeto, uma separação rigidamente instituída numa separação de ficheiros. Por não ter o encapsulamento constituído explicitamente um tema em avaliação neste trabalho, foi preservada essa característica, apesar de se ter tido em conta o encapsulamento.

Em suma, este trabalho mostrou-se um bom exercício de aplicação e entendimento da estrutura de um programa utilizando noções do paradigma da programação orientada por objetos, e a aplicação de um padrão de desenvolvimento que se afigura proveitoso.