ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Programação Orientada a Objetos

1º Trabalho Prático

<46022> Hugo Dias, <46055> João Martins,

<46074> Ricardo Margalhau

Instituto Superior de Engenharia de Lisboa – Verão 2018 / 2019

Introdução

Com este trabalho foi nos dada oportunidade para aplicar os conhecimentos adquiridos até ao momento das unidades curriculares de PG e POO.

O objetivo do trabalho foi de implementar o jogo *Sokoban* usando uma arquitetura *MVC* (*Model-View-Controller*) de modo a que mais tarde este jogo seja mais facilmente implementado noutra plataforma sem necessidade de mudar o modelo do mesmo (ex. plataforma *Android*).

Neste trabalho apenas houve a necessidade de implementar certas partes do modelo e da visualização, sendo que o restante código foi nos dado como fonte de partida para o começo da realização do mesmo.

Numa primeira parte deste relatório analisaremos a arquitetura proposta para as classes do *Model* e do *View* que foram implementadas e na segunda parte serão explicadas todas as funcionalidades base e adicionais do trabalho.

1. Análise da Arquitetura

Este trabalho veio com algumas classes já realizadas previamente e segundo o diagrama *UML* que nos foi fornecido no enunciado do trabalho foi nos possível averiguar a funcionalidade das classes que estavam por acabar (ex. *Level, Cell, StatusPanel, ...*) e quais as classes que necessitavam de ser implementadas de raiz.

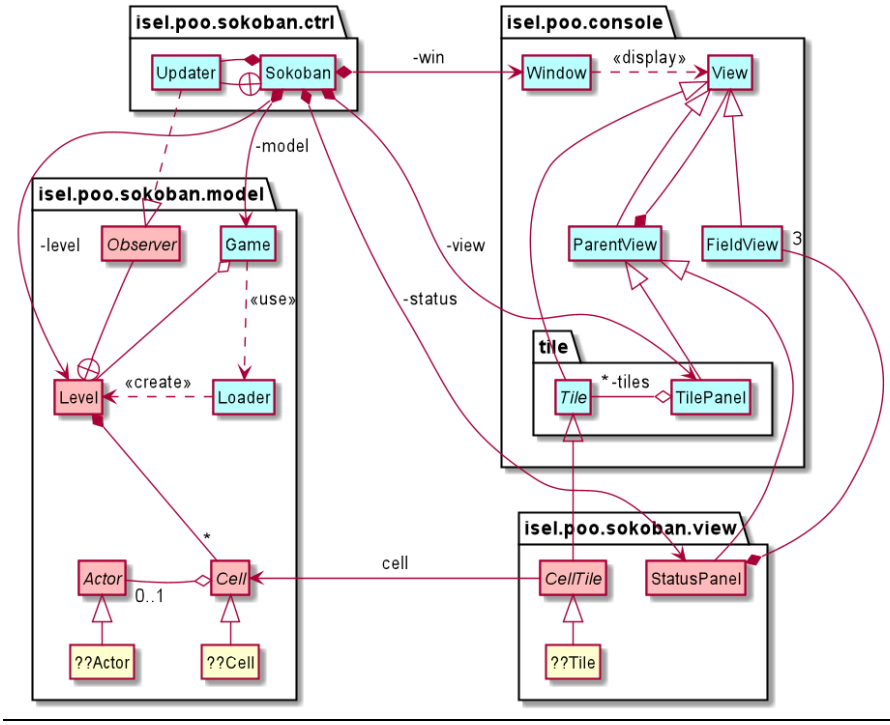

Figura 1 - Diagrama UML fornecido no enunciado

Como o objetivo do modelo é poder ser implementado noutras plataformas sem ter a necessidade de mudar o código, como tal, este deve ser independente do controlo e da visualização. Contudo houve necessidade de comunicar com o controlo ou a visualização e para tal propósito usámos um *Listener* (neste caso um *Observer*) que consiste na implementação duma *interface* e da chamada aos mesmos, pelo que assim encontramo-nos perante um tipo de programação denominado de *event-driven*.

Para as restantes classes que eram objeto de estudo por parte de cada grupo (*Actors*, *Cells* e *Tiles*) realizámos um diagrama *UML* de modo a facilitar toda a organização estrutural do trabalho.

Para os *Actors* temos a seguinte arquitetura:

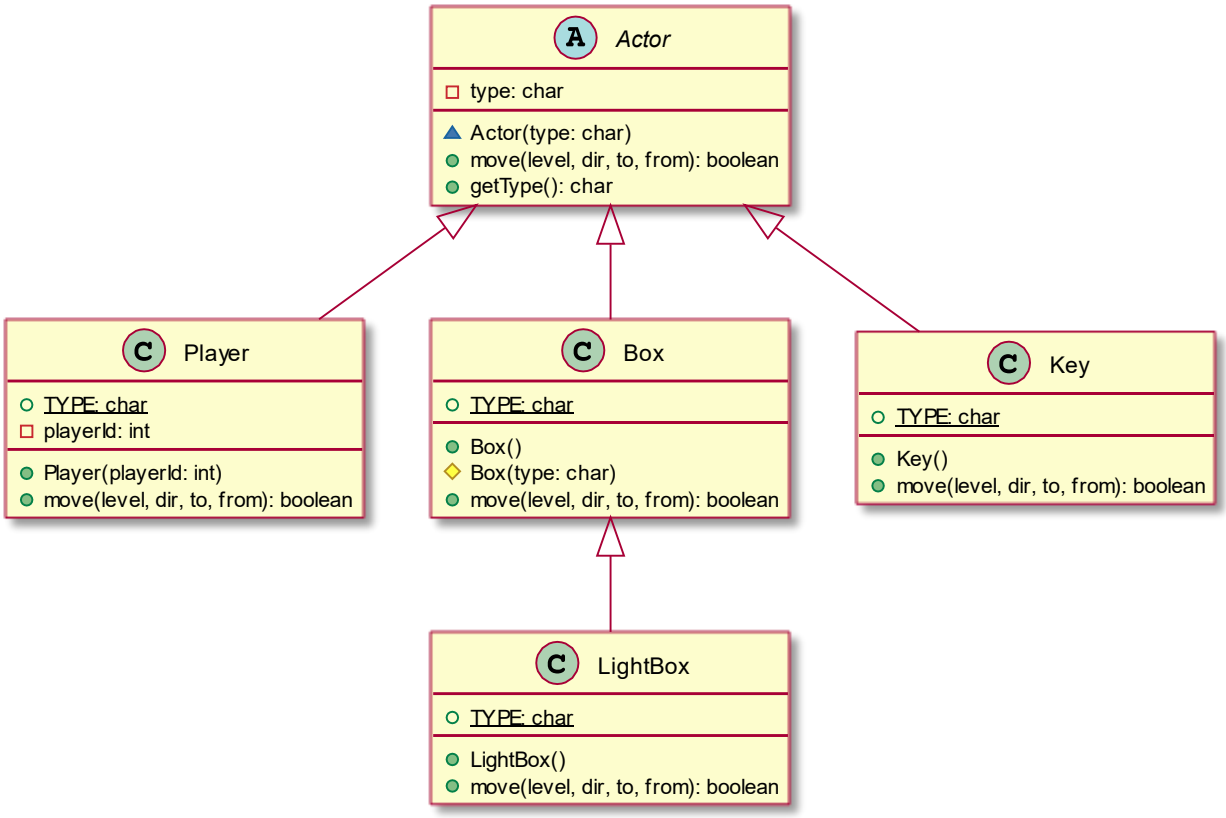

Figura 2 - Arquitetura UML dos Actors

Considerámos que esta arquitetura é viável visto que nos permite ter flexibilidade para implementar outros tipos de atores no futuro caso o jogo assim necessite. Neste momento temos 4 tipos diferentes de atores em que cada um contém uma variável que indica qual o caracter correpondente (é o caracter usado para o ficheiro *levels.txt*) e se for caso fazemos *override* ao método *move* visto que é este que decide se o ator pode mover-se ou não com base no mapa do jogo.

Para a arquitetura das células do jogo temos vários tipos de células sendo que alguns deles derivam de outros tipos de células como nos mostra a Figura 3 (ex. *DownCell* deriva de *DirectionalCell* e como uma *DirectionalCell* só pode ser posta no chão então *DirectionalCell* deriva de *FloorCell*).

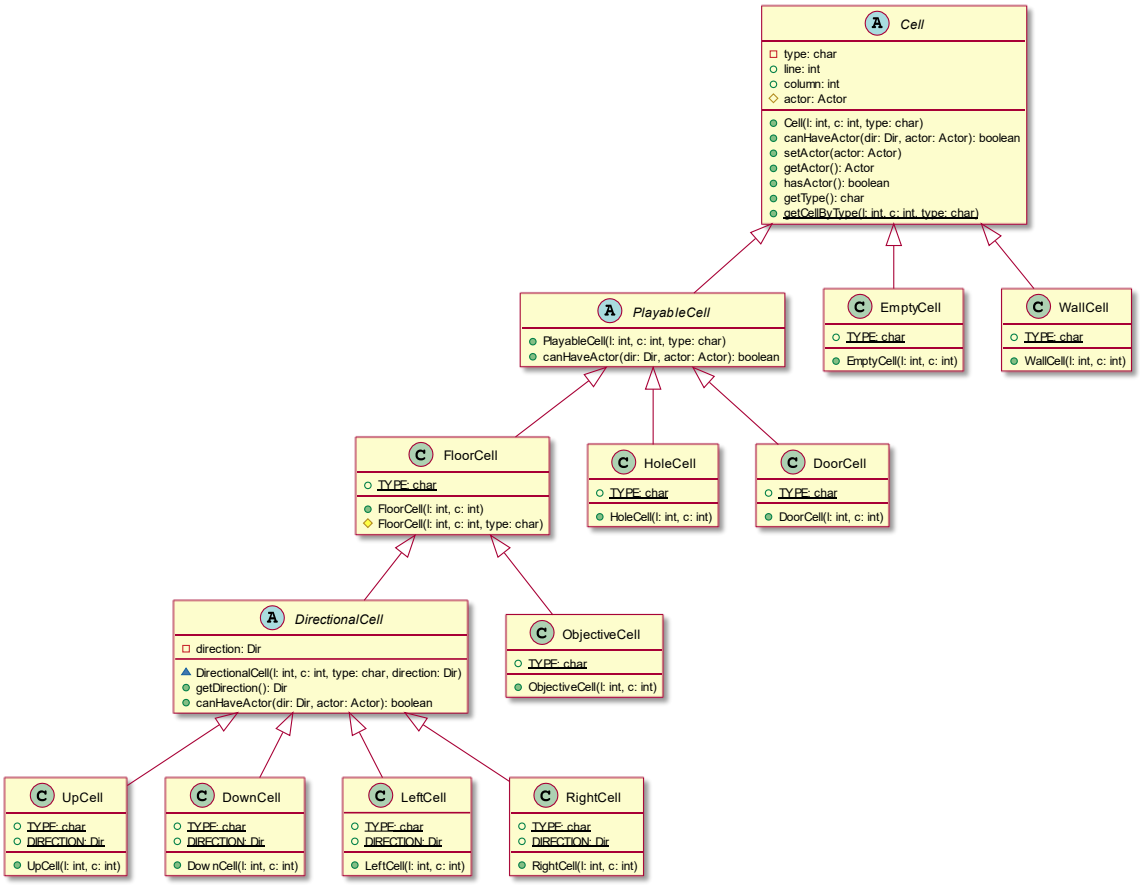

Figura 3 - Diagrama UML das Cells

Finalmente, para os *CellTiles* temos uma arquitetura semelhante ao dos *Cells* pois cada *CellTile* tem que corresponder à sua célula mas apenas tratando da visualização de cada célula do jogo (ex. o *PlayableCellTile* tem métodos destinados a mostrar o *Actor* que está na célula, se for caso disso).

1. Análise da Arquitetura

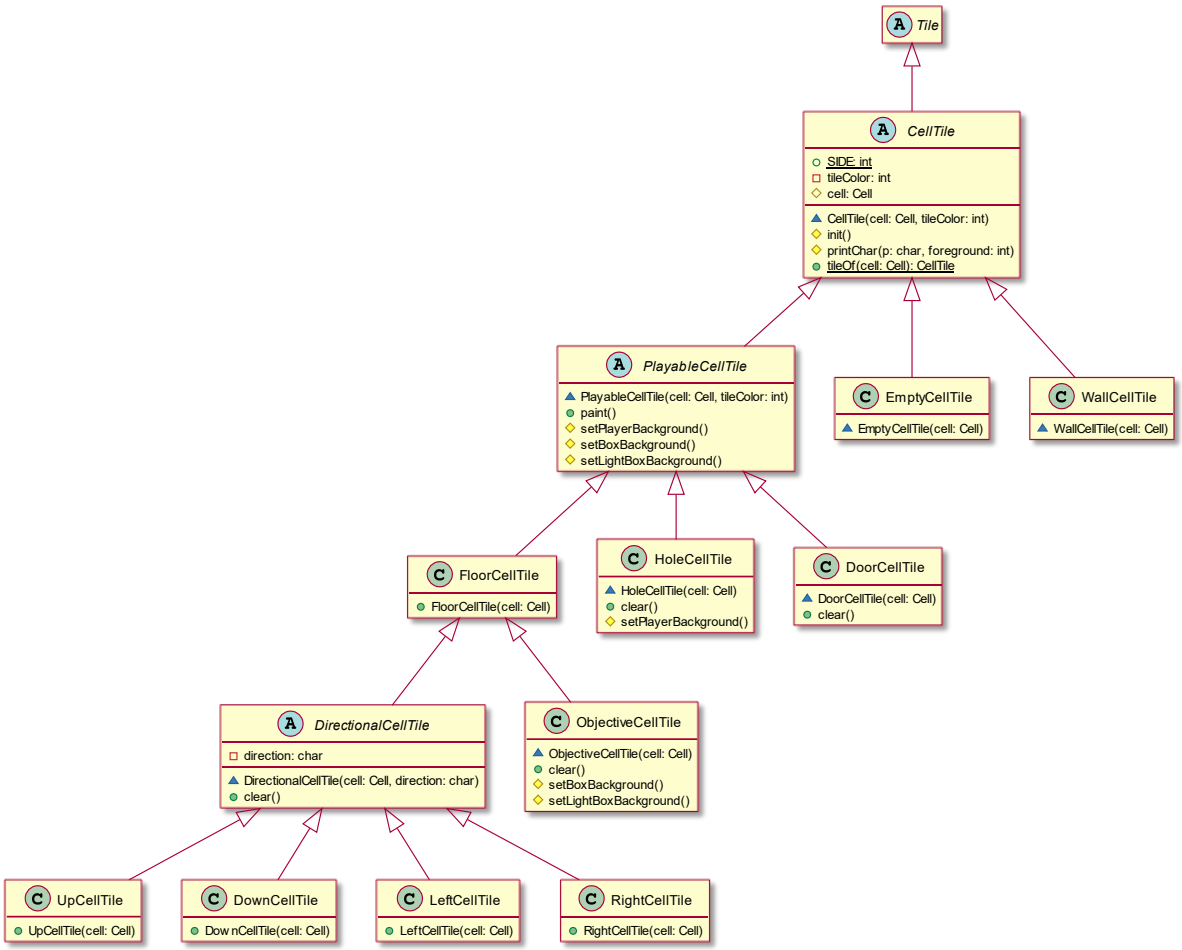

Figura 4 - Diagrama dos CellTiles

2. Funcionalidades

2.1 *Status Panel*

Para implementar o *StatusPanel* criámos primeiramente a classe com o mesmo nome na parte da visualização, como está explicito no diagrama *UML* do enunciado e com esta a derivar de *ParentView*. Com a classe feita eram precisos 3 campos estáticos privados de modo a representar visualmente três propriedades: O nível atual, o total de movimentos do jogador naquele nível e o número de caixas que faltam para completar o mesmo nível.

Além disto foi ainda preciso fazer *setters* de modo a alterar as propriedades durante a execução do jogo, bem como foi preciso alterar o método *paint* para que este também pintasse o conjunto de comandos que são possíveis com o teclado.

2.2 *Cell Tiles*

De modo a implementar a classe abstrata *CellTile*, derivada de *Tile*, começámos por implementar o método estático *tileOf* que converte um objeto do tipo *Cell* no seu *CellTile* correspondente. Para fazer isto usámos o *instanceof* para verificar de que tipo é a célula e com isto criar a representação visual da mesma.

No construtor desta classe é recibida a célula do modelo e o *background* do *tile* que foi criado, com este *background* a ser inicializado no método *init* da classe através do *setBackground*.

Nesta classe temos ainda o método *printChar* que vai colocar, assumindo que cada lado do *tile* tem n de lado, o caracter especificado como parâmetro do método em todas as posições do quadrado da *tile* (ou seja, coloca o mesmo caracter $n \times n$ vezes). Este método recebe ainda *foregroundColor* como parâmetro, sendo que esta cor determina a cor do caracter a ser escrito.

2.3 *Cell*

A classe abstrata *Cell* contém o método estático *getCellByType* que se destina a converter um caracter na sua célula correpondente e, para tal, usa um *switch* que verifica os caracteres possíveis até encontrar um que corresponde. Caso não chegue a encontrar, o método retorna *null*.

Nesta classe temos ainda vários campos como:

- *type* – destina-se a guardar o tipo da célula
- *line* – guarda a linha em que a célula se encontra
- *column* – guarda a coluna em que a célula se encontra
- *actor* – guarda o ator que está na célula

No construtor de *Cell* é passado como parâmetros a linha, a coluna e o caracter que representa o tipo da célula.

Esta classe tem ainda mais 5 métodos sendo os quais: o *canHaveActor* que recebe *Dir* e *Actor* como parâmetros e retorna *true* se a célula em que este método for chamado suportar o ator e a direção que foram passados; o *setActor* e *getActor* que servem de *setters* e *getters* respetivamente para o campo *actor*; o *hasActor* que verifica se a célula contém um ator; e o *getType* que retorna o caracter de tipo da célula.

2.4 *Actor*

Na classe abstrata *Actor* é definido um construtor onde é passado como parâmetro o tipo do ator (caracter que é usado para o identificar), que é guardado num campo desta, e que pode ser acedido exteriormente através do método *getType*.

Existe ainda o método *move* que é responsável por decidir como é que os atores das classes que derivam *Actor* se devem mover (este método deve ser redefinido consoante o seu uso) e movê-los se tal for possível.

Para mover os atores, na classe abstrata apenas temos de verificar se a célula para onde vai suporta o ator e, se suportar, devemos atualizar essa célula e tirar o ator da célula onde estava. Após isto devemos ainda atualizar as células através da chamada do *updateCell* do *Level*.

2.4.1 *Player*

A classe final *Player* tem um campo publico, também final, que guarda o identificador corresponde a cada objeto do tipo *Player*. Esta classe redefine ainda o método *move* da classe *Actor* pois um jogador apenas se pode mover caso a próxima célula estiver vazia (*FloorCell* sem *Actor*), ou caso a próxima célula contenha um *Actor* diferente de *Player* também ele movível.

2.4.2 *Box*

A classe *Box* redefine o método *move* do abstrato *Actor* pois só podemos mover uma *Box* caso a próxima célula esteja desocupada. Ao mover a *box* atualizamos ainda o contador de *boxes* do *Level* através da chamada ao método *incrementBoxCount* quando a *Box* entra ou sai dum *ObjectiveCell*.

2.4.3 *LightBox*

A classe final *LightBox* deriva da classe *Box* redefinindo o método *move* para que caso a próxima célula contenha uma *LightBox* consiga mover estas duas numa vez só.

2.4.4 *Key*

A classe final *Key* redefine o método *move* pois esta só se pode mover caso a próxima célula não contenha nenhum ator e se a próxima célula for uma *DoorCell* então chama o método *replaceCell* do *Level* para substituir a porta pelo chão.

2.5 *Level*

A classe *Level* é a responsável pela lógica do nível onde são implementados métodos que permitem colocar células, mover atores, etc…

Nesta classe começámos por fazer o construtor que recebe 4 parâmetros: o número do nível, a sua altura, a sua largura e o máximo de jogadores para este

nível – e com este construtor foram criados 2 *arrays* de células: o *playersCell* que tem de dimensão o número máximo de jogadores e que guarda a referência para as células em que os jogadores estão, e o *array board* que se trata de um *array* bidimensional que serve para armazenar todas as células do nível independentemente se contém um jogador ou não.

No método *reset* desta classe alteramos todas as referências que estão dentro do *array* de jogadores e do *array* bidimensional para apontarem para *null* (de modo a colocar novas referências nas próximas chamadas ao método *put*). Este método ainda coloca os campos *players* (que guarda o número de jogadores), *moves* (que guarda o número de movimentos) e *boxes* (que guarda o número de caixas) a zero e o campo *manIsDead* a *false*.

O método *put* da classe *Level* destina-se a preencher uma posição da *board* com a célula que corresponde ao caracter e, sendo assim, são passados como parâmetros neste método: a linha da *board*, a coluna da *board* e o caracter que representa um *Cell* ou *Actor*. Neste método começamos por transformar o caracter que nos é passado nos parâmetros para um objeto derivado de *Cell*, sendo que para tal usamos o método estático previamente feito *getCellByType* da classe *Cell*, e caso o retorno for *null* então temos que o caracter trata-se de um ator ou então não existe, pelo que ao verificar esta condição fazemos um *switch* por cada tipo de ator existente. Adicionalmente, se na posição do *array* já existir uma célula e a chamada *Cell.getCellByType* retornar *null* então queremos adicionar à célula já existente o ator (ou nada se não for um ator válido), caso o *board* não contenha nenhuma célula naquela posição então assumimos, sabendo que o tipo nos parâmetros não foi uma célula, que se trata de uma *FloorCell*.

O método *moveMan* do *Level* é responsável por mover o jogador especificado com *playerId* nos parâmetros deste para a $posição\_atual + vetor\_de\_direção$ e para mover o jogador vamos obtê-lo primeiramente através do *array* de *playerCell* (obtendo a célula através do *playerId* e depois o ator chamando o método *getActor* da classe *Cell*) e em seguida chamamos o método *move* do *Player* (é preciso fazer um *cast* explicito a *Player* visto que *getActor* retorna uma referencia para *Actor* e não *Player*). Se a chamada ao método *move* de *Player* for bem-sucedida (ou seja, retorna *true*), então devemos guardar a nova célula que contem o jogador na posição respetiva do *array playersCell*, aumentar o número de jogadas e verificar se o jogador está morto (morre caso a nova célula seja to tipo *HoleCell*) executando os respetivos eventos.

Os métodos *updateCell*, *replaceCell*, *boxMoved* e *incrementBoxCount* também foram criados no *Level* com o objetivo de poderem ser chamados pelos atores quando estes se estão a mover pelo mapa (visto que os atores não têm acesso direto ao *Observer* nem ao campo *boxes*).

No *Level* temos ainda diversos *getters* e *setters* para os diversos campos, sendo que também temos um *getter* para obter a célula que está numa posição da *board* com base nas suas coordenadas.

2.5.1 Observer

No *Level* temos ainda uma *interface* pública que contém os métodos que devem ser implementados por um *Observer*. Cada *Level* tem capacidade para um e só um *Observer*, que neste caso está implementado no módulo de controlo na classe *Sokoban*.

2.6 *Dir*

No enumerador *Dir* estão implementados os vários vetores possíveis (*UP*, *DOWN*, *LEFT*, *RIGHT*), sendo que este *enum* contem um construtor que indica qual o movimento tomado por cada um dos 4 vetores, sendo esses movimentos denominados por *delta line* (*dl*) e *delta column* (*dc*).

Isto permite-nos facilitar o cálculo da nova posição em métodos como o *moveMan* do *Level*.

Conclusão

Neste trabalho concluímos todos os objetivos obrigatórios e opcionais que nos foram indicados no enunciado. Para além destes incluímos algumas funcionalidades da nossa autoria como, por exemplo: modo 2-jogadores, efeitos sonoros e ajuste do *SIDE* da *CellTile* quando o *level* excede as dimensões.

Este trabalho permitiu-nos adquirir prática nos conceitos novos que temos vindo a aprender na unidade curricular POO como: *Polimorfismo, Herança, Interfaces, ...*