OOP Semester project

Sokoban

Gitlab: https://gitlab.fit.cvut.cz/luzindan/oop

Members:

- **Irina Shushkova** @shushiri
- **Maria Karzhenkova** @karzhmar
- **Artem Kandaurov** @kandaart
- **Danil Luzin** @luzindan

Assignment

"The goal of this project is to develop a sokoban game. The application should be able to load levels, collect points... Nicely decoupling the UI from the game model should be considered." (https://docs.google.com/document/d/15wIMy3M8OrFwtzbPNIkSLXWXTTC3aXzNibHkSxZTdXs/edit#)

Installation

Game can be downloaded form our repository : https://gitlab.fit.cvut.cz/luzindan/oop
After all files are downloaded you can start the game by executing the following code:

```
game := Sokoban currentGame.
game start.
```

You can control the game using the WASD keys or by sending move: message to the game instance. 'M' key can be used to open up the menu. 'R' to restart the level.

```
up := Direction up.
down := Direction down.
left := Direction left.
right := Direction right.
```

```
game move: up.
game move: left.
game move: down.
game move: right.
```

List of levels can be found in the /resources folder under the name levelList.txt. Levels that are listed in that file will be loaded during the game.

Level format is text based:

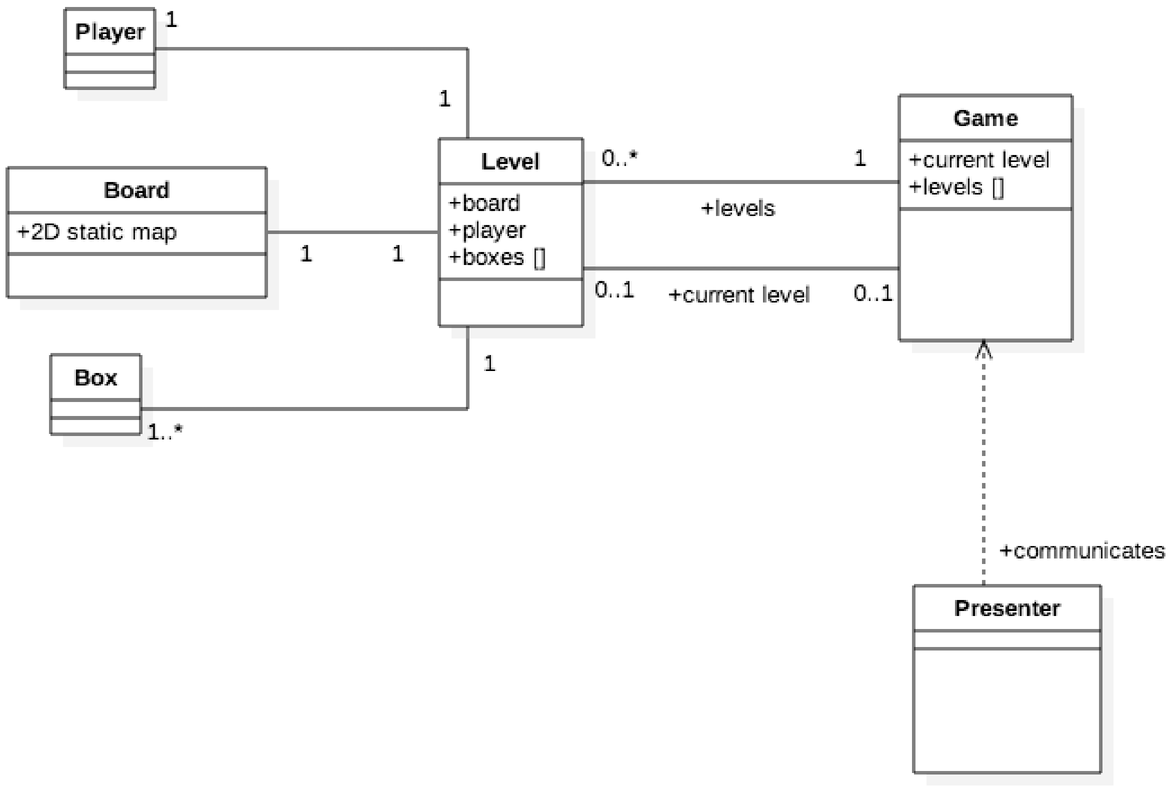
- Width and height are written on the first line as well as optimal score
  width height optimal_score
- Matrix of a level goes after that.

Character meaning:

- w - wall
- f - floor
- p - player
- b - box
- t - target

Base architecture

In order to decouple UI from the game logic we decided to separate graphical part from the game logic in a pretty straightforward way.
Level class is responsible for keeping track of game logic.
Presenter class is responsible for drawing the UI
Sokoban class is responsible for processing the keyboard input and storing level information between sessions.



Pattern usage examples

---

1 Singleton

Since we wanted to keep high-score between session it felt natural to use singleton for the Sokoban class.

Code example:

```
Sokoban class>> new
    self error: 'Please, initialize Game via singleton method currentGame'

Sokoban class>> currentGame
    ^ UniqueInstance ifNil: [ resourcesPath := self getResourcesPath.
                UniqueInstance := super new initialize.
                UniqueInstance initialize ]
```

---

2 & 3 Double-dispatch & Visitor

In order to draw element we use double-dispatch on a cell that can be a floor, wall, player, etc.

Code example(simplified to focus on double-dispatch):

```
Presenter>> calculateRawPresentation
    | cell |
    1 to: height do: [ :h |
        1 to: width do: [ :w |
            cell := self level atX: w Y: h.
            cell acceptPresenter: self ] ].

Floor>> acceptPresenter: presenter
    presenter addFloorElement

Wall>> acceptPresenter: presenter
    presenter addWallElement
….
```

---

4 Null Object Pattern

For keeping high-score of the level and printing it we had to find a solution for levels that were not cleared yet. For that we implementer class UndifinedScore (subclass of Score).
Its default behaviour is shown below:

Normal behaviour of score class:

```
Score>> value
    ^ value
Score>> > other
    ^ value > other value
```

Default behaviour for score of level that wasn't completed yet:

```
UndifinedScore>> value
    ^ '<?>'
UndifinedScore>> > other   #default score acts as infinity when compared with the other score
    ^ true
```

---

5 Polymorphism

Polymorphism is used throughout our project but most obvious place is in elements of the map. As seen in code for double-dispatch we send message **acceptPresenter** to a cell that can be player, box, target, wall or floor cell. Each of the receivers react to that message differently so when we sent the message we shouldn't care which cell will receive it.

Code:

```
Floor>> acceptPresenter: presenter
    presenter addFloorElement

Wall>> acceptPresenter: presenter
    presenter addWallElement
```

Same goes for how we treat UndifinedScore. Since UndifinedScore acts as an infinity by default and level keeps track of high-score by storing instance of class Score (superclass of UndifinedScore) we can do the following:

bestScore will be set to an instance of UndifinedScore for uncompleted levels. Message* **> *is overwritten in the UndifinedScore (see point #4 Null Object Pattern)*

Code:

```
    Level>> isWon
        Boxes do: [ :each |  each onTarget  ifFalse: [ ^ false ] ].
        bestScore > currentScore ifTrue: [ bestScore := Score value: currentScore value ].
        ^ true
```

Message **>** is implemented differently for two different classes, so we can rely on polymorphism to determine the right behaviour.