Fitoban

A Sokoban Implementation in Pharo

Michal Černý, Kamil Červený, Michal Cvach, Libor Plíšek

Table of contents

Installation

To install Fitoban: Sokoban Implementation in Pharo, follow these simple steps.

1. Download Pharo from https://pharo.org/. The application was tested using Pharo 6.1, so install this specific version to make sure the application will run as well as possible. The application might work in other versions of Pharo as well, but it wasn't thoroughly tested.
2. Download Bloc (stable) from https://github.com/pharo-graphics/Bloc. You can also use this command in the Pharo Playground:

```
Metacello new
    baseline: 'Bloc';
    repository: 'github://pharo-graphics/Bloc:pharo6.1/src';
    load: #core
```

3. Clone the latest version of Fitoban from (you can use Iceberg) https://gitlab.fit.cvut.cz/cvachmic/OOP-sokoban-in-Pharo
4. Unfortunately, Iceberg can't download graphical and maps resources for now, so you will have to download those manually. Create directories 'maps' and 'graphics' in the directory you will be using Pharo from (the active working directory for Pharo) and then download desired maps from https://gitlab.fit.cvut.cz/cvachmic/OOP-sokoban-in-Pharo/tree/master/maps and copy then into the directory. As for graphics, download everything in https://gitlab.fit.cvut.cz/cvachmic/OOP-sokoban-in-Pharo/tree/master/graphics and copy into the 'graphics' directory. The game won't work without the graphic resources! (You might experience errors if you don't download all the graphical resources.)
5. You're done! You can now start Fitoban from the playground simply using the following command:

```
SGame start
```

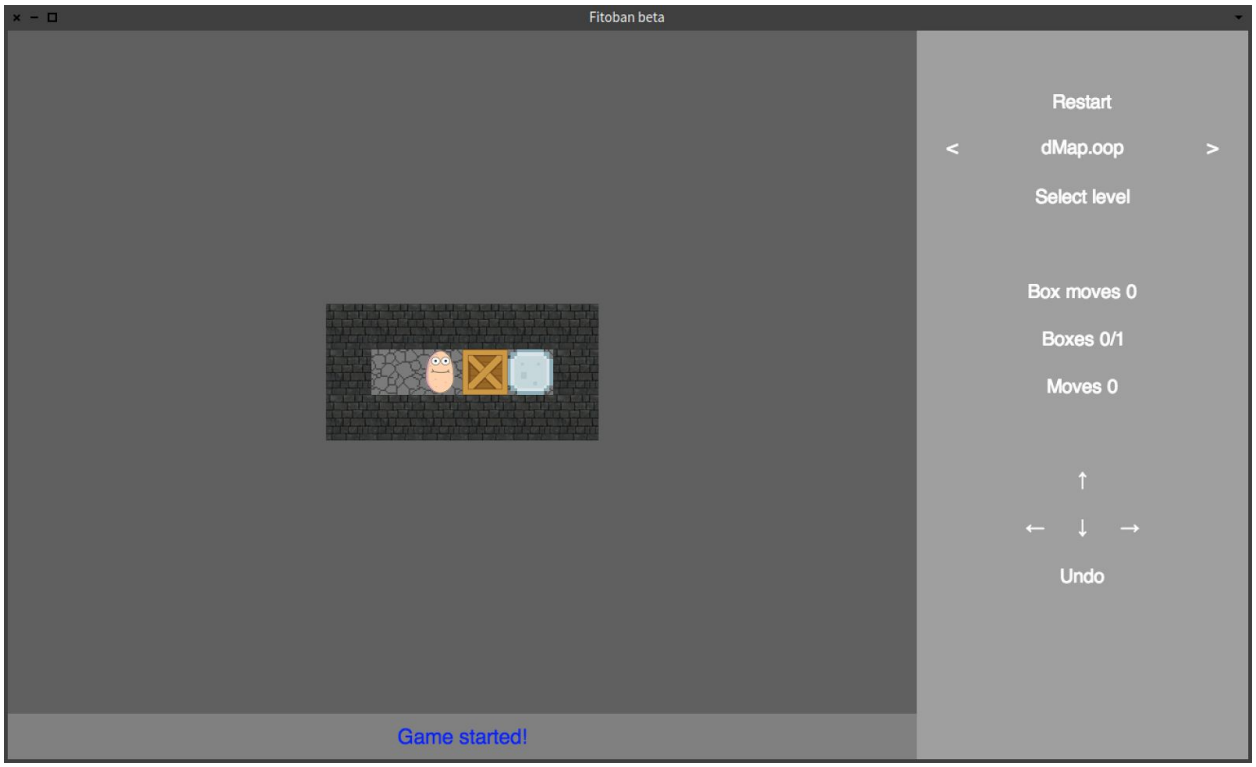
How to play & Examples

Starting the game

To start the game, simply execute the following command in the Playground:

```
SGame start
```

The game window should open and you should be seeing something like this:



You can then simply play the game using the arrows on your keyboard, or using the arrows in the right side of the game window.

Objectives of the game

Game objectives are based on the classic Sokoban game. Your main goal is to push all the boxes to the target tiles.

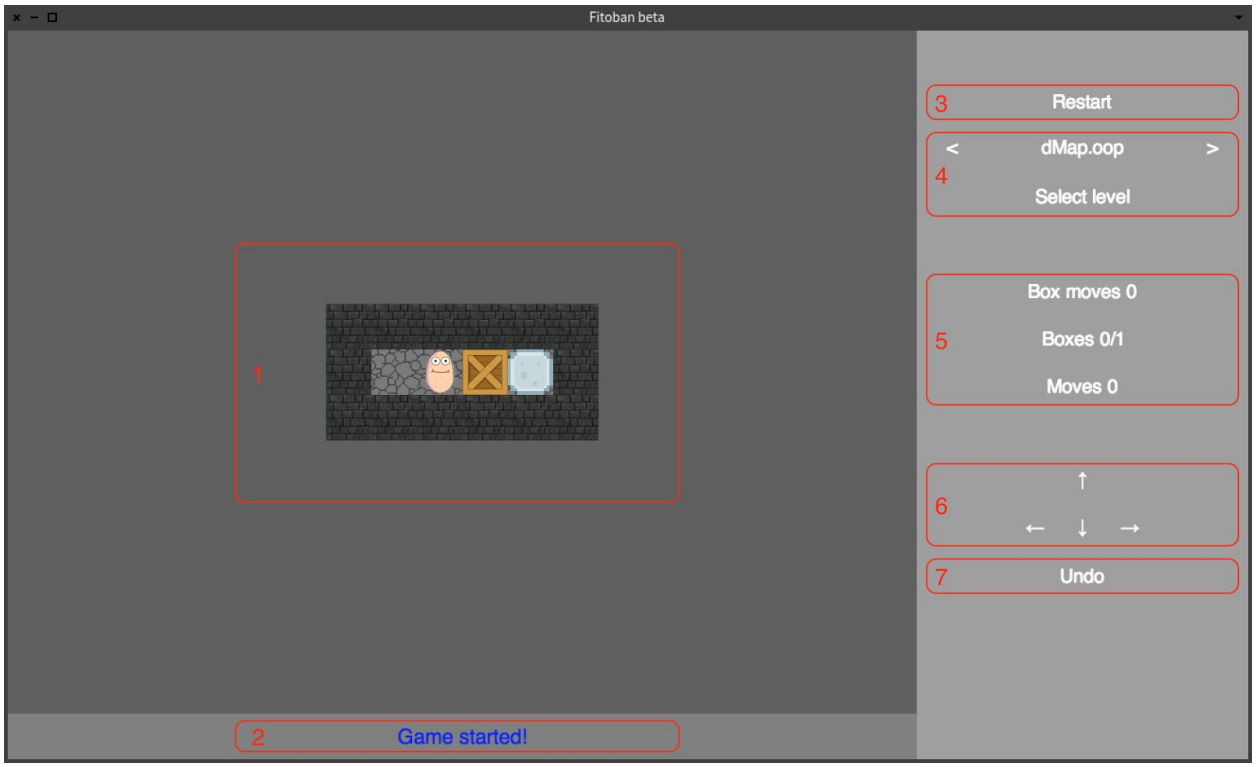
Sokoban isn't only about completing the map. It's also about completing the map in the least possible amount of moves or box pushes. For keeping track of those stats, we present all of those in the right side of the game window.

Also, it can happen that you accidentally make one wrong move and then you would have to replay the whole map. In order to prevent this frustrating situations, just over the stats you will find an 'Undo' button, which can be used to undo your last move (only if you have made any moves, obviously).

That way you can perfect your score until you are happy with your results.

Controls and choosing maps

Our game obviously doesn't offer only one map (in fact, you can actually create your own maps, as will be demonstrated later in the documentation). You can choose which map you want to play, or you can restart the current map as many times as you want, in case you did want to start from scratch.



Here we will describe the basic elements of the game window:

1. The game view itself. Here you can see the current map, and the view updates as you make any moves.
2. We call this element the 'Textstrip'. This element will give you some useful info when necessary. For example it will inform you when you are trying to undo and there aren't any more moves to undo, or it will inform you when you try to load a map which isn't valid.
3. The restart button simply restarts the current map. This is useful for example if you made too much wrong moves, that it's better to start over than undo all the moves.

4. Here you can choose the level you want to play. Use the two small arrows to switch between maps, and then simply click on the name of the map you want to play or the Select level button, and that map will be loaded. Maps are loaded from the ‘maps’ directory in the actual working directory when you open Pharo. Maps must have the ‘.oop’ file extension. The map is validated once you try to load it, so even invalid maps will show in the menu, but if you try to load them, you will be notified (in the Textstrip), that the map you’re trying to load is invalid.
5. The stats, as mentioned on the previous page. These stats will keep track of your moves, box moves (pushes) and boxes currently on target out of all the boxes.
6. The arrows, which you can use to control your character. You can obviously use your keyboard arrows as well, the choice is up to you.
7. The undo button, which you can click to undo your last move, if you have made any moves before. You can also use the U button on your keyboard.

Creating your own maps

If you want to create your own map (or play a map you found somewhere on the internet), it's really easy!

We use a simple text representation for the map, where there are 8 various elements that a map can consist of. Those are:

- **B** for background tiles (tiles that don't really do anything, the player can never get onto these tiles)
- **W** for wall tiles (the player can't walk through walls)
- **F** for empty floor tiles
- **K** for boxes located on floor
- **P** for the player if the player is initially located on the floor
- **T** for an initially empty target tile
- **L** for boxes that are initially located on a target tile
- **R** for the player if the player is initially located on a target tile

There are obviously some constraints that a map must fulfill for it to be a valid map. Some examples of those rules:

- There can only be one player on the map
- There must be the same amount of boxes and targets
- The player shouldn't be able to walk into the background tiles. This means, that there must be a wall between each floor (or target) and background tile.

Take into consideration, that we do not validate, if the map is completable. You can easily create and use a map, that will not be beatable at all.

An example map can look like this:

```
level
h:10
w:10
WWWWWBBBBB
WFFFWWWWWW
WFKKWTTFFW
WFFKFWTFFW
WFKKTTTFWW
WFKPWWWFWB
WFKKTTWFWB
WFFFFFFFWB
WWWFFFFFWB
BBWWWWWWWB
```

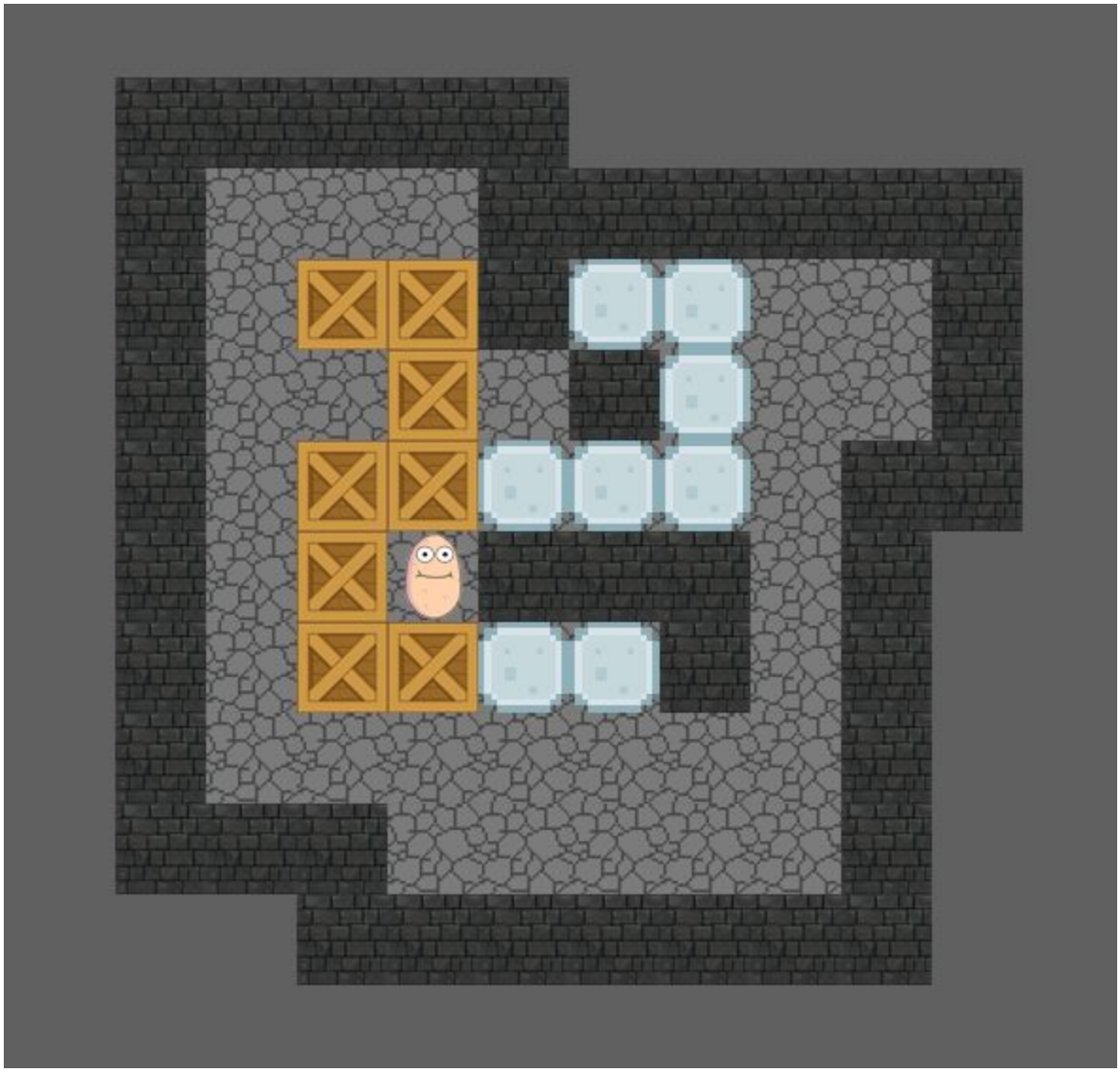
On the first three lines, there is a compulsory header, which consists of the word ‘level’ on a separate line, then the height and width of the map (in format as seen in the example), and then the map itself.

The map itself can only consist of the eight characters described on the previous page, and there must always be a height*width amount of characters. This means, that all maps must be of a rectangle shape. If you want a map that is not rectangular enough, you can always fill the remaining tiles by background tiles, which don’t really do anything gameplay-wise.

Also the maximal size of the map is set to 20 width and 15 height. So if one of the dimensions of your map is bigger, the game won’t load the level.

Once you have the map completed, you just have to save it with the ‘.oop’ file extension and then put it into the ‘maps’ directory in the directory from which you will be starting Pharo (the current working directory for Pharo). After that, you should be able to find your map in the game next time you start the game.

The example map from this page will look as follows (on the next page):



Basic architecture

The Game Window

Here we will describe how the game window works all together. The game window consists of three separate element and those are:

- The **game view**
- The **menu**
- The **Textstrip**

We won't talk about how each of those elements is implemented here, because this will be covered in following chapters. Here we will mostly describe how the window works as a whole.

The SGame class and its role

We should probably mention the **'SGame'** class, which you can find in the 'Sokoban-Game' package. It is the class, that is used to start the game.

It might be useful to implement the **'SGame'** class as a singleton - so that there would be only one running instance of the game at any time. We haven't implemented it as such for now though.

Right now, sending the message start to SGame ('SGame start') works pretty simple. A new instance of SGame is created and then the message 'open' is sent to the instance, which initializes all the necessary game structures and then opens the game window. The last instance of the game can be closed by sending the message close to SGame ('SGame close'), which is useful automatized testing. The class doesn't keep track of previous instances though, so keep that in mind.

The Game Window itself

Talking about the game window, we basically talk about the class **'SGameWindowElement'**, which you can find in the 'Sokoban-Graphics' package. As mentioned in the beginning of 'The Game Window' chapter, it consists of three separate elements.

First element is the game view, which is an instance of **'SMapElement'** (also in the 'Sokoban-Graphics' package). It represents the map itself. The player can see the whole map there, and when he makes a move, the game view is updated to match.

The second element is the menu. The menu is located in the right side of the game window and its implementation can be found in the 'Sokoban-Menu' package. It handles choosing levels, undoing moves, and also displays stats to the player (such as the current amount of moves or pushes).

The third and last element is the Textstrip. It is located in the bottom part of the screen and displays important information to the player (such as error messages when loading invalid maps or when trying to undo a move before making any moves).

Messages are sent directly to the game window, but it delegates them to its parts. The menu handles player input. The menu is the only element (in the game) which captures keystrokes or mouse clicks, so if you are interested in that, you might want to jump to the menu chapter. The Texstrip is pretty simple. You can display string messages in the Textstrip by sending a 'setTexstrip:' message to the game window instance with the string you want to display as the only argument. The game view than handles redrawing, so when a 'redrawElementXYOn:And:' is received, it is delegated to the game view component, which then updates the view.

Graphics

Here we will break down the design of the game view. Most of the graphics related code is located in the ‘Sokoban-Graphics’ package.

The map view

Let’s start by the map view right of the bat. The map view is represented by the ‘**SMapElementClass’** in the ‘Sokoban-Graphics’ package. We decided to use ‘**BlGridLayout**’ for the game view itself, because our game is a pretty simple tile based game. However, if we wanted to have some sort of moving animations, it would probably be better to use ‘**BlBasicLayout**’.

The map itself consists of instances of ‘**STileGraphicalElement**’ (or its subclasses, obviously). There are four most important element and those are:

- **‘SBackgroundGraphicalElement’**
- **‘SFloorGraphicalElement’**
- **‘STargetGraphicalElement’**
- **‘SWallGraphicalElement’**

The most interesting part here are the floor and target elements, because those are the only elements where any redrawing occurs. Both of these elements actually have three versions of their look, which are represented **‘SRawXYFloorGraphicalElement’** and **‘SRawXYTargetGraphicalElement’** where XY is Empty, SokobanOn or BoxOn. Those are the three states those tiles can be in.

So then our approach is, that the **‘SFloorGraphicalElement’** (or the Target element respectively) consists of three of those Raw elements, and always draws the correct one depending on the current state of the tile.
This approach makes adding more states pretty complicated, because you would have to add a new Raw element for each state, and then add it into the actual GraphicalElement, and change the redraw methods, but for a simple game as ours,

where there is no reason to add more states later, this approach seemed sufficient. For bigger projects we wouldn't recommend it though.

Shape based and image based view

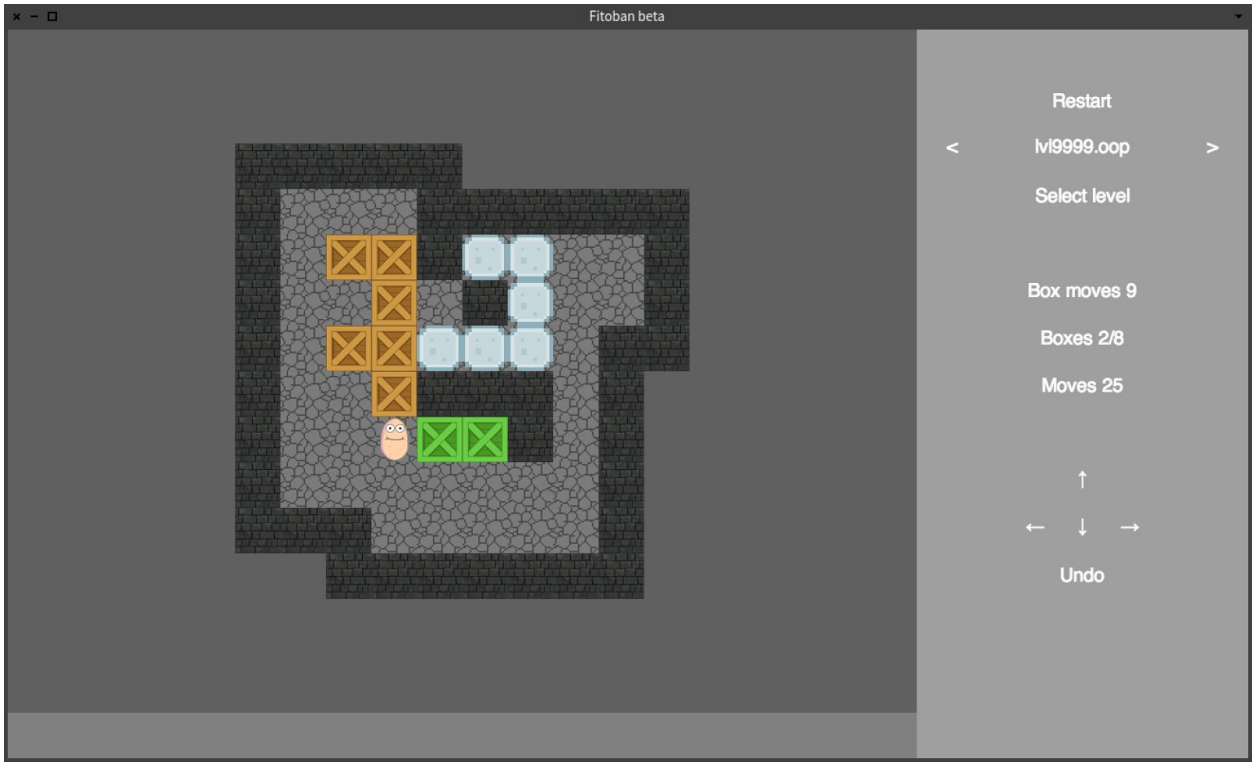
You can notice, that we have a Shape version for some elements in the 'Sokoban-Graphics' package. We started first with the Bloc shape based view for our game, and then added support for image based view later.

We decided to keep the Shape versions of the elements in the package as well, because they can still work with the view pretty easily. The initialization of the tiles when creating the map is actually based on the 'Sokoban-Round' package. There you will find the class **'SRTile'** and its subclasses. You might have notice that it has four subclasses which represent the four elements we have talked about on the previous page.

Each of those subclasses has a method called **'createGraphicalElement'**, which basically tells us what graphical element we have to create for that game logic element. So all you have to do, would you want to use the Shape based elements instead of the image based ones, is changing the elements in those methods to their Shape versions. (Except for example Background is the same for both view versions, so you don't have to change that one)

To compare the two views, here we have a screenshot of the same map using each of the views.

Image based view:



Shape based view:

Menu

The best way to create new instance of menu:

```
| menu grid round |
menu := SMMenuInit new.
grid := menu grid.
…
menu roundInstance: round.
```

SMMenuInit new makes an instance of **SMMenuModel** - there are set:

- selectedLevel = 1
- actLevel = 1
- levelsArray (get from **MapLoader** listLevels function)
- buttonsArray (13x **SMButtonModel**)

SMMenuInit grid return BlElement of whole menu. The grid function makes new instance of **SMMenuElement** and **set it's model** which was already made in **SMMenuInit**. In settings of model are made all button elements (13x **SMButtonElement**) with their model (in this step is the **SMChangeTextAnnouncement** assigned), then the **clickEvent** is assigned and a setSizeLocation function is called (size and location are stored in button's model).

In the last step has to be **round instance assigned** - the menu calls round when some of menu buttons is pressed.

Round

The way to create new instance of round:

```
| window menu map round |
window := SGameWindowElement new.
menu := SMMenuInit new.
map := MapLoader new.
round := SRound new.
round graphicInstance: window.
round menuInstance: menu model.
round mapInstance: map.
window roundInstance: round.
menu roundInstance: round.
round level: 1
```

Commands:

round := **SRound** new. - create new instance of **SRound.**

round graphicInstance: window. - assigns class **SGameWindowElement**.
(it is necessary to draw graphical objects on the playground and to draw states)

round menuInstance: menu model. - assigns class **SMMenuInit**.
(it is necessary to draw stats and counters of moves)

round mapInstance: map. - assigns class **MapLoader**.
(it is necessary to load maps of levels from files)

round level: aLevel. - loads map of level from the file and prepare game (draw graphics objects, draw stats, ...).

round getBoxCnt. - returns the number of boxes of the loaded level

round moveUp. - if it is possible, moves player to the up, redraw the graphic objects, redraw stats and check if player wins or not yet.
(similarly for round moveDown., round moveLeft. and round moveRight.)

round undo. - takes players move back.

Interesting code examples

Polymorphism

Polymorphism can be seen in multiple parts of our project. We will give a few examples here.

Game view redrawing

As we discussed in the 'Basic architecture' chapter, the game view is implemented using '**BlGridLayout**', while the Floor and Target tiles are both implemented using three Raw graphical elements, which represent the empty tile, the tile with the player on, and the tile with a box on. When the player pushes a box, the message '**redrawWithBox**' is sent to the tile the box is moving to. And we don't have to check is that is a floor tile or a target tile, we just simply send the message and both of these elements know how to draw a box on them, so we just let them do their work.

The game logic

An probably even better example is the game logic in the 'Sokoban-Round' package. There you can find the **'SRTile'** class with its subclasses. Each of those have the **'getObject'** or **'moveToMe'** methods. When the player makes a move, those messages are sent to the adjacent tiles, and they can then specify, if the move was valid, or, when the player tries to move into a wall, the Wall element will simply tell him, that the can't.

Singleton

For the image graphics, we use '**.png**' image files for our tiles. It wouldn't be wise to load for example the '**emptyfloor.png**' file each time we create a Floor tile. Instead, we have implemented an '**SImageTile**' class with subclasses for all the elements. The first time an element is created, we actually create an instance of the image tile, which will load the image from disk, but next time the same kind of element is created, we only return the same instance we created the first time.

This can be seen in the 'Sokoban-Graphics' package in the class '**SImageTile**' and its subclasses.

Composite

The game window itself is composed of three elements. Those elements are the game view itself (where you can see the map and the player and boxes move), the menu (on the right, where you can select levels, undo moves and so on), and the textstrip, which is located in the bottom and is used to display information important for the player.

Each of these three elements handles different things, but we do not send messages to them individually. Instead we send messages to the **'SGameWindowElement'** instance (in the package 'Sokoban-Graphics'), and it delegates them to the parts. So for example when the message 'redrawXY' is sent, it is delegated to the game view. If a message 'setTextstrip' is sent (to change the text displayed in the Textstrip element), it is delegated to it as well.

The hook and template concept

This concept can be found at multiple places of our project. First of all, basically every time we do anything graphics related, we use this concept. The ‘drawOnSpartaCanvas:’ method of the **‘BlElement’** class server as the template, and our elements, such as the Textstrip, or the tiles in the gameview, all adjust the behavior for our needs.

Other example of this concept might be the **‘SImageTile’** class (in the ‘Sokoban-Graphics’ package), where the method ‘load’ is defined in the class **‘SImageTile’**, but the method load asks for ‘self filepath’, so each of the subclass can redefine which file it actually wants to load, without having to copy the ‘load’ method into each subclass individually.