Assignment 2

Ex. 1.1: Command Pattern

The Command Pattern got used for two different parts in our code. The first one is responsible for controlling the board state. Three commands exist for this task: Loading a board state, saving a board state, and commanding the execution of a turn. This comes parallel to a new feature where we can undo the last executed move, thus it makes sense to implement the pattern. The second one gets introduced parallel to Ex. 3 as well since it completely covers a freshly implemented extension. This will be the option to choose from different themes for the layout of the board. There is a command for each possible theme and additionally the possibility to undo the last one.

The idea for an undo move came before the decision to implement the Command Pattern. However, the Command Pattern is a logical choice considering our circumstances where we have to execute a move and now also have to undo this move. In the lecture, the example was based on a remote control with physical buttons. Although we don't have a remote control, we still have buttons, just not physical ones. For that reason, we take our UI and metaphorically treat it as our remote control. Thus, we expanded the usage of the pattern to cover all buttons. Loading a game and saving a game do not have an implementation to undo their actions though, thus leaving their undo implementation for a future expansion.

Before the implementation of the pattern, we simply accessed the board and moved the piece to execute a move. Now, with the command pattern, we have a CommandTurn class that implements two methods, execute and undo. It also provides a constructor with which we hand over some arguments. These are the turn, it's active player and an enemy piece if one got captured. To execute a turn, we create a new Command with the necessary variables. We then call the execute method which takes care of moving the piece and pushing the move on our own stack. To undo the last move we pop the last two entries on the stack (if two are pushed, if only one is pushed we only consider the last entry) and create opposite turns. These new turns then get executed by also moving the pieces on the board. This method also takes care of replacing pieces that got captured by a prior jump.

Loading a board state and saving the current board state also get executed by the press of a button. Since none of our 'buttons' are equal, we don't need to implement a method that binds each command to buttons. We directly create the command variables for loading, saving, and undoing the last move in the constructor of the GameHandling class. We then only need to access the variable when needed and call the execute or undo method.

The implementation of the Command Pattern for the theme selector allows to easily add or remove themes, change existing themes and allow for undoable operations without affecting other code. It is a nice way to implement the theme selector with encapsulation in mind.

The GameHandling class acts as the client and is responsible for creating concrete commands and setting their receivers. For instance, to add the green theme, the GameHandling class first creates the receiver for the corresponding command, namely GreenTheme and then passes it to the command "GreenThemeCommandOn".

After the commands have been created, and their receivers have been set, the themeSelector class, which acts as the invoker, sets the commands by assigning the created commands to an internally stored array of commands along with an index.

These operations are all done in the initializeThemeSelector() method of the GameHandling class.

Now, whenever the user presses a button on the theme selector page corresponding to a theme, then the invoker uses its pressButton() method and the index corresponding to the pressed button, which corresponds to a command that is stored in the themeSelector class, to ask that specific command to carry out the desired request by calling its execute() method.

The commands then ask the receivers corresponding to the commands to perform the actual action, which would be to change the colors of some of the UI components.

Each receiver, i.e. each theme, has different colors stored and corresponding to their colors, it updates the colors with the help of the GameHandling class, since the GameHandling class has access to the JavaFX scenes and is able to modify the properties, such as the color, of the UI components.

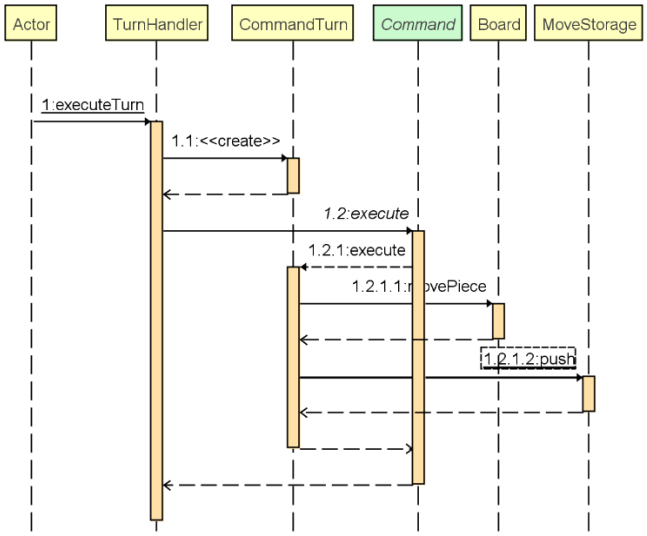
The commands themselves don't contain an undo operation, since in this context to undo a theme, this would mean, to revert to the previously selected theme, however, what theme was previously selected is only known by the invoker, namely the ThemeSelector, and not by the commands. Hence the ThemeSelector class, along with an array of commands, also stores a stack consisting of the previously selected themes.

Whenever the user selects a different theme than the one already selected, then the corresponding command is pushed onto the stack.

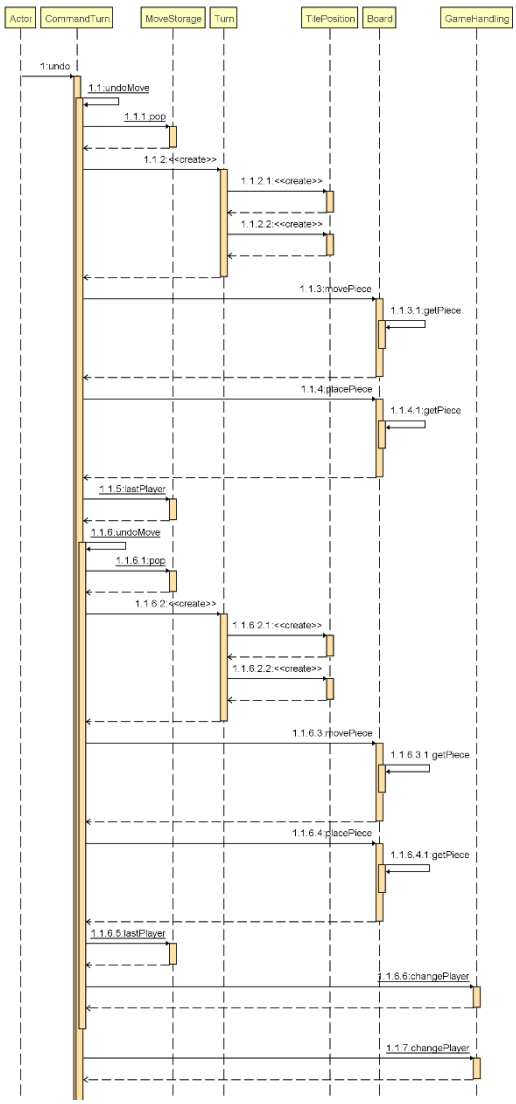
And whenever the user presses the undo button, given that the stack is not empty, that is, a theme has been selected previously, the ThemeSelector pops the command corresponding to the currently active theme to check what command was previously selected, executes that command, and overwrites the current theme.

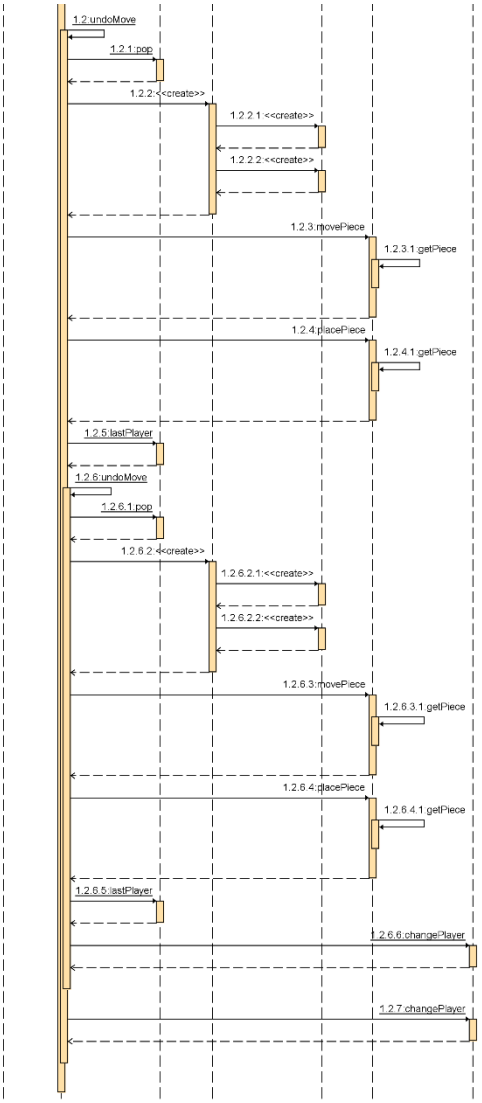
Ex. 1.2: Command Pattern

CommandTurn.execute()

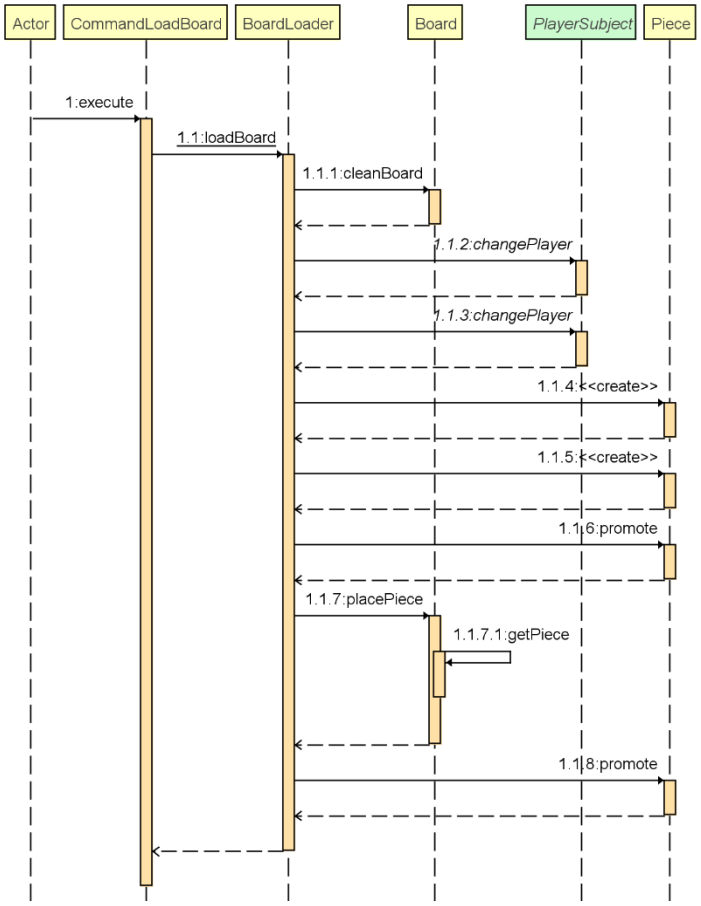


CommandTurn.undo() (Part 1 & 2)

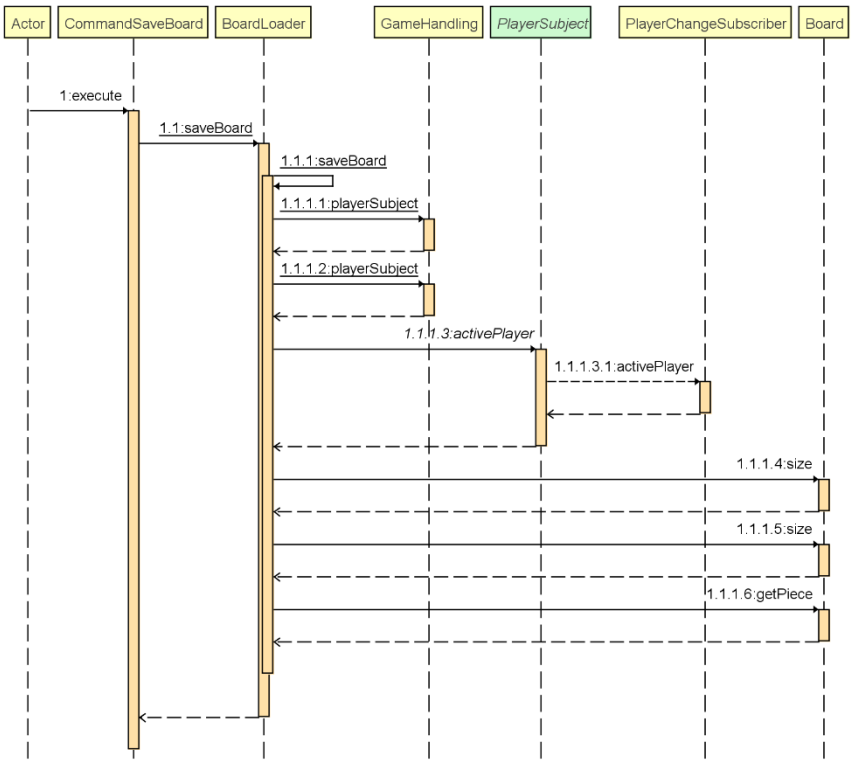




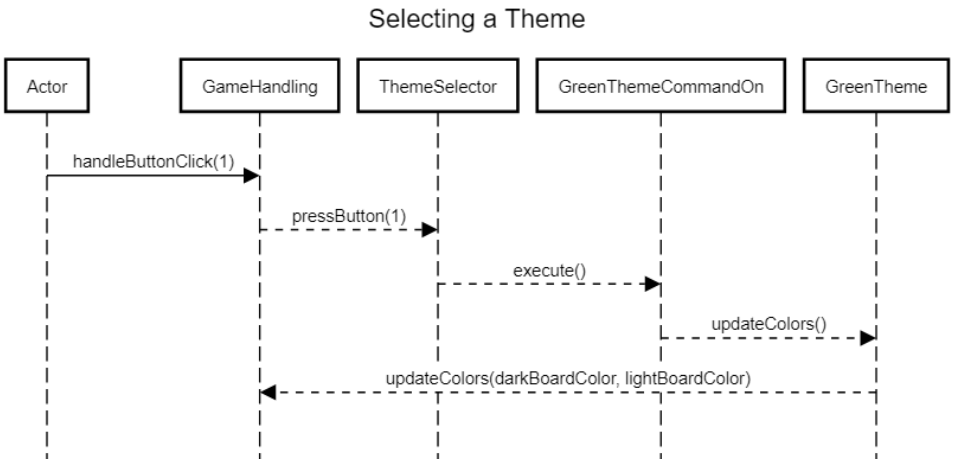
CommandLoadBoard.execute()



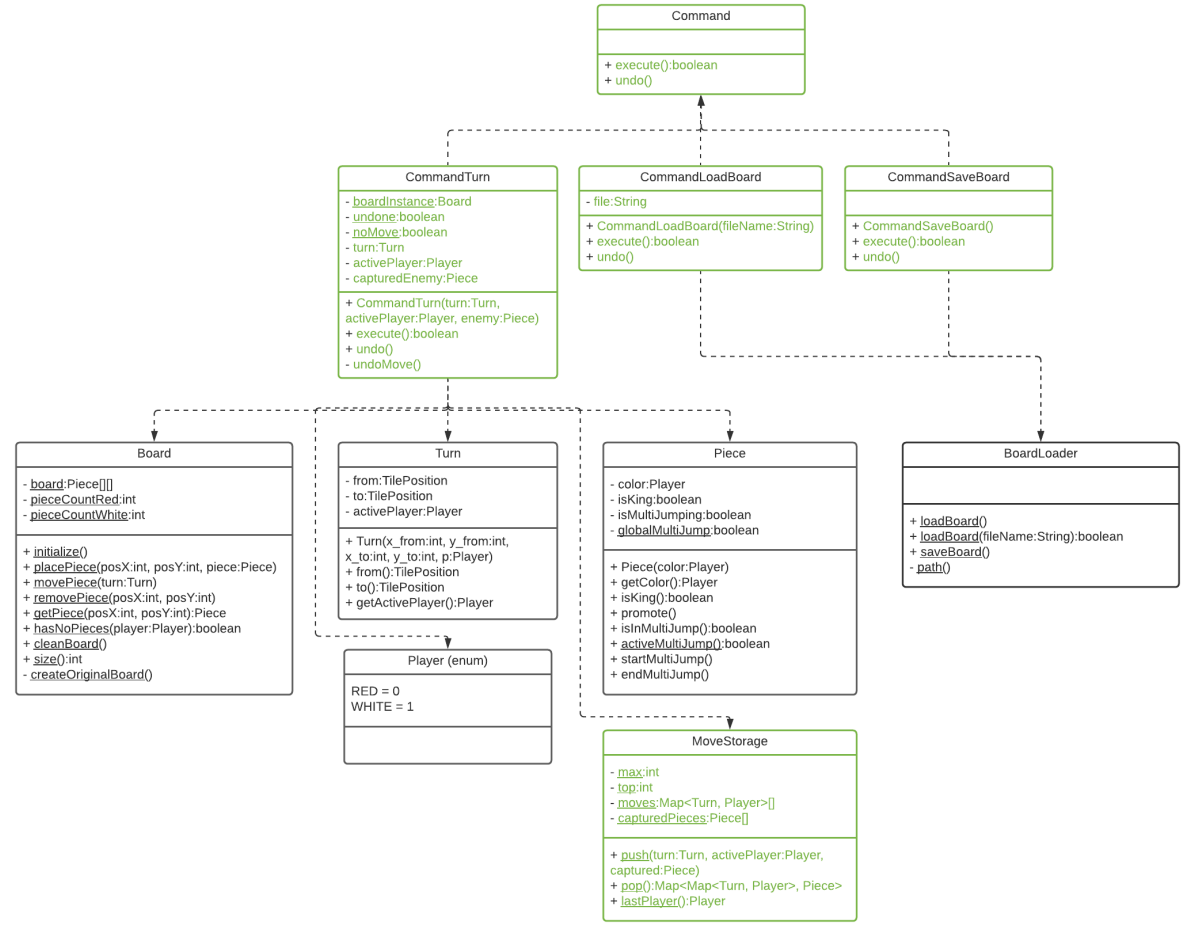
CommandSaveBoard.execute()

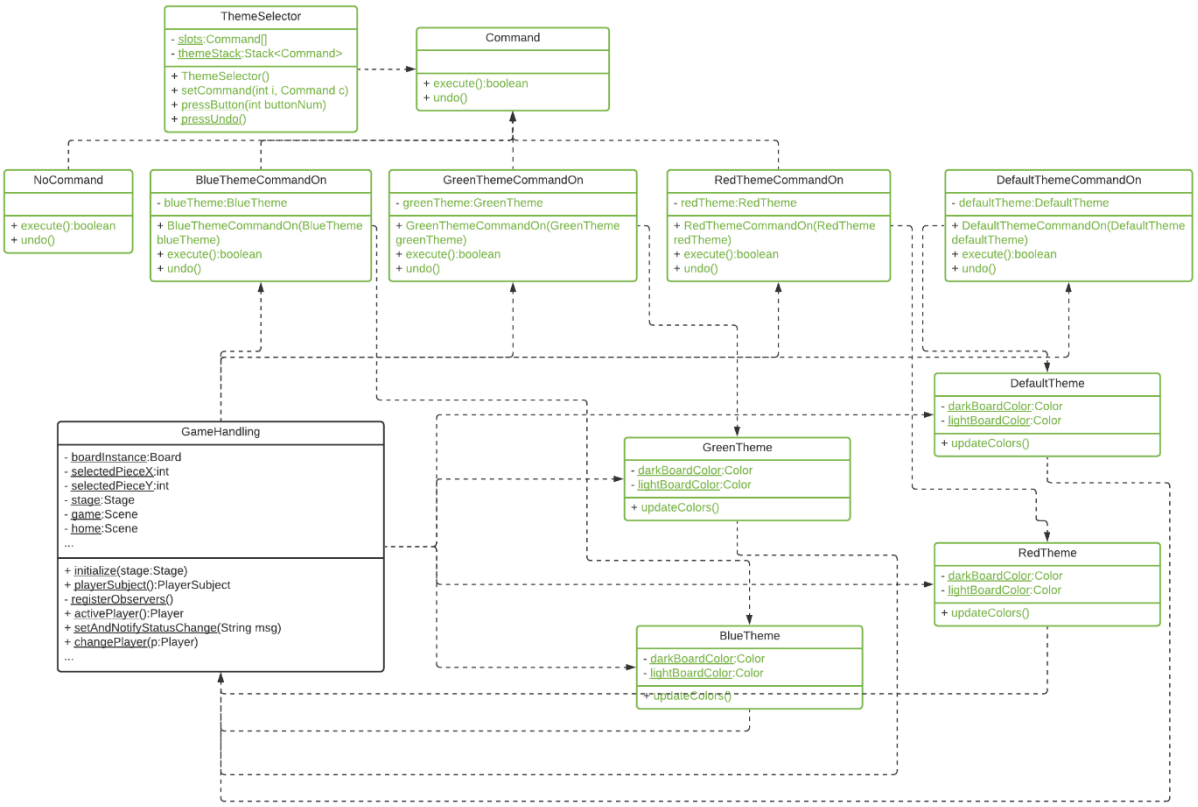


ThemeSelector, e.g. the green theme



Ex. 1.3: Command Pattern



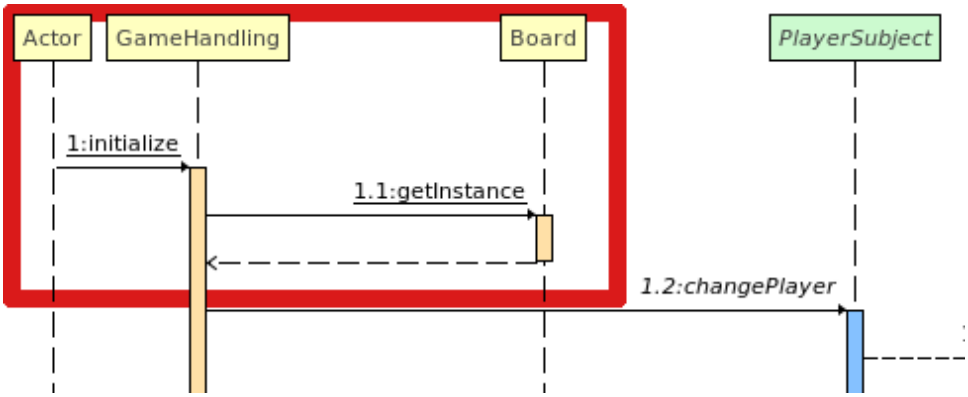


Ex. 1.1: Singleton Pattern

We chose the Singleton Pattern as a means to reflect the fact that the Board class exists globally and is shared as an instance. We changed it to being an instantiable singleton class by first removing the static identifiers from all the methods and then setting the constructor's visibility to *private*. To allow only a single instance of the Board to exist, the now private constructor is only reachable through the new "getInstance()" method which is thread-safe and makes sure that at any given time, the only existing Board instance is returned, de-facto making it statically available again. The instance reference is also stored locally in the Board class for future retrievals. This reflects that the Board class is the single source of truth when it comes to game data.

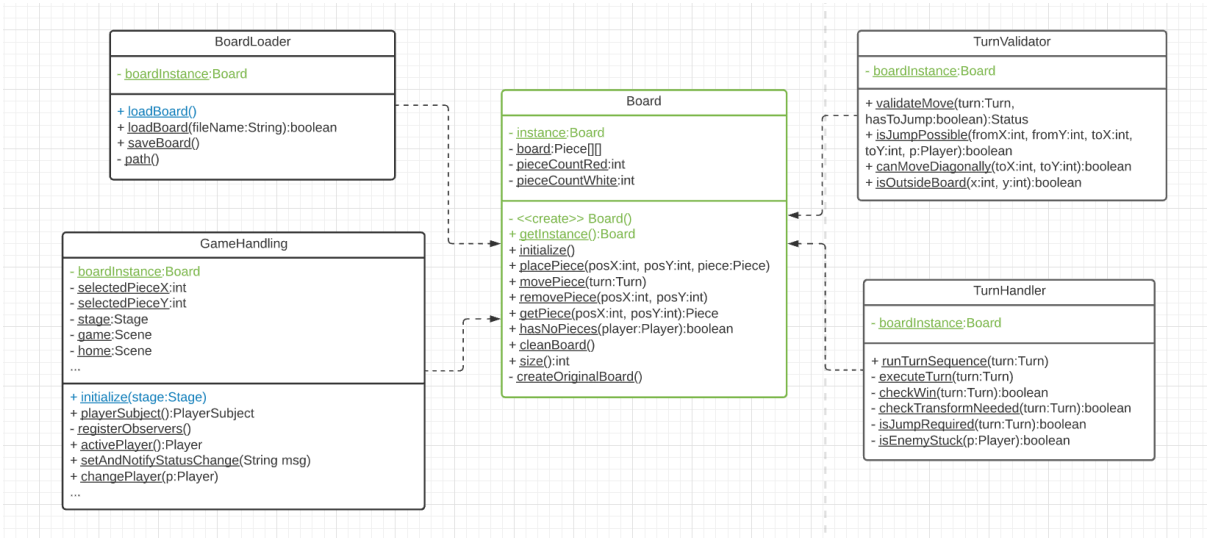
The accessing classes themselves also keep a reference to the board instance if it is needed in multiple methods within the class.

Ex. 1.2: Singleton Pattern



The class is now accessed uniformly through the getInstance() method which resolves and returns the board instance. We can see this in the above example and is done equally across all accessing code locations.

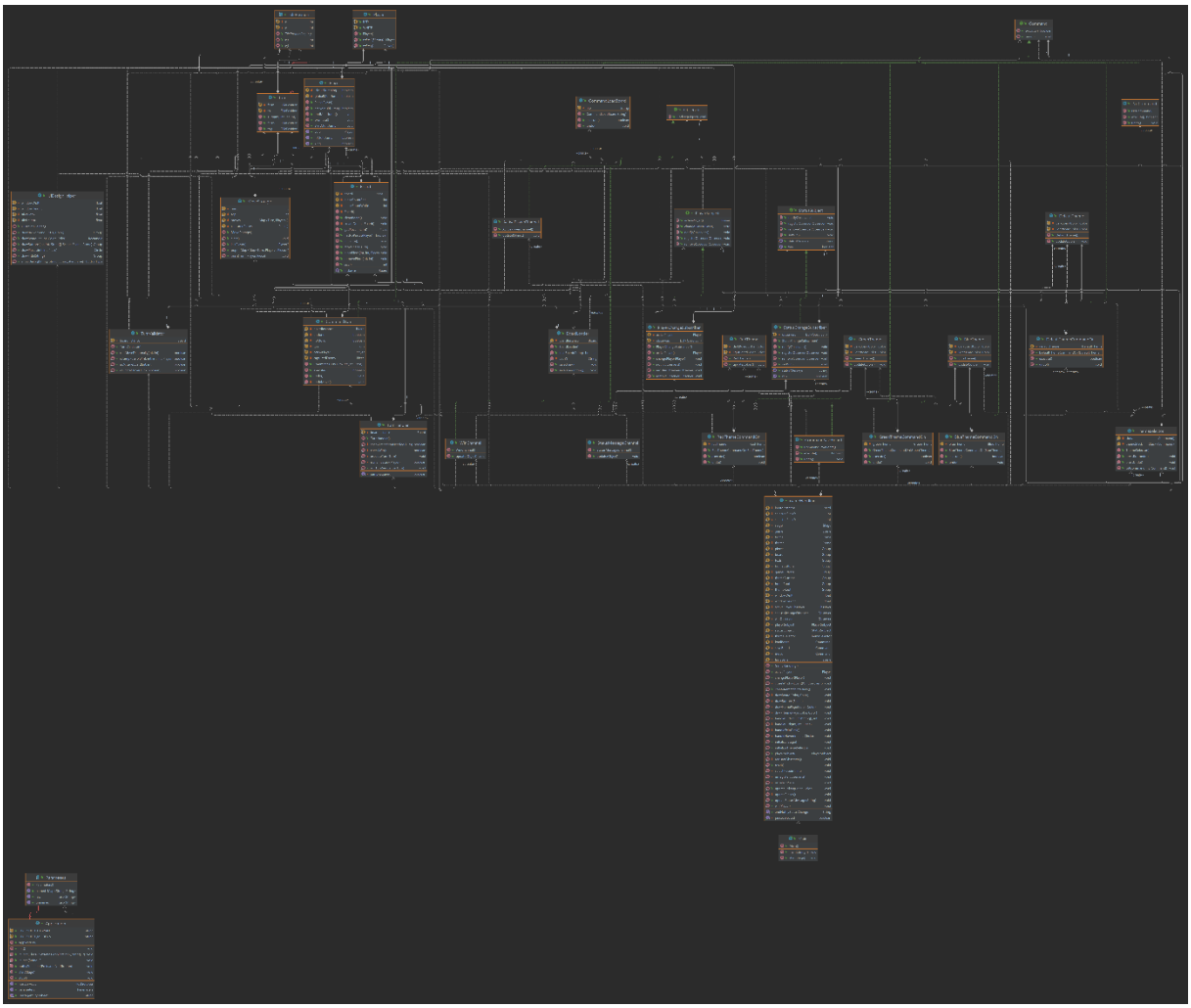
Ex. 1.3: Singleton Pattern



Green variables and classes mark involved references and new code fragments respectively, blue-marked methods call the getInstance() method. Where no blue methods are present, declaration and assignment is done simultaneously.

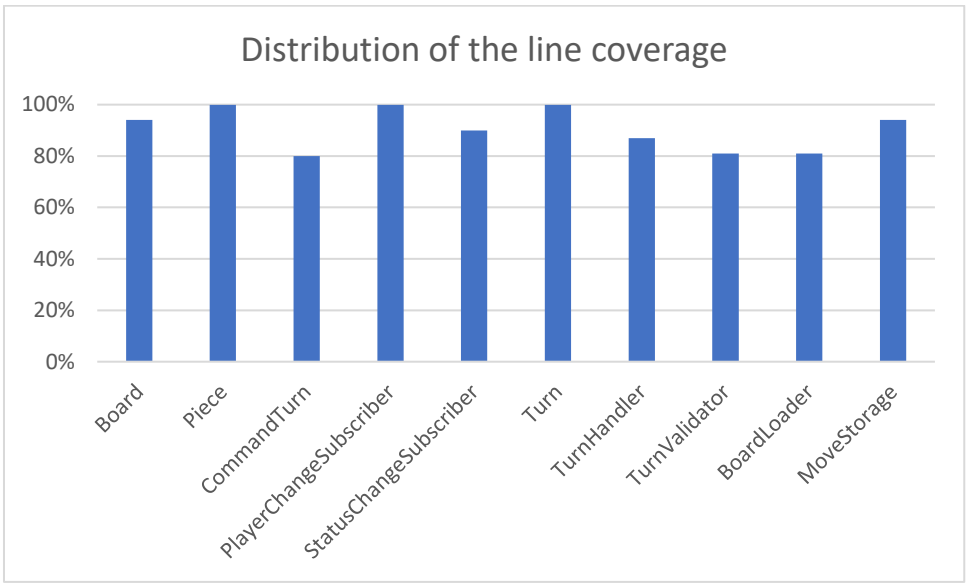
Ex. 2.1

- **Board:** In the Board class the whole game state is saved and every move is executed. Therefore, it is very important that every single method does what it is designed for. Otherwise, invalid changes could happen which would lead to an invalid game state.
- **Piece:** The Piece object stores every detail like isKing or color of the piece. Furthermore, there are some special values to control the turn flow like isInMultiJump. With a faulty implementation it is not guaranteed that the game follows the rules.
- **CommandTurn:** The CommandTurn Class controls the flow of the turns. It saves the last 2 turn on a stack, executes the turn and undo the move if required. It is important to test this class since no moves could be executed if the code would not work as expected.
- **PlayerChangeSubscriber:** Is needed for the control flow. Every time it is the other players turn, this class notifies every observers. Without a correct implementation, the active player could not be changed.
- **StatusChangeSubscriber:** Similarly, to PlayerChangeSubscriber, StatusChangeSubscriber is needed for the control flow. It brodcasts messages for the UI, such that the user can see what he did wrong etc. Without a correct implementation, the user would not get any messages.
- **Turn:** The Turn object contains the move to be executed. It provides the form a to locations.
- **TurnHandler:** The TurnHandler starts the whole turn sequence with all checks and returns the status of the turn. This class must not have errors since the game logic would break.
- **TurnValidator:** The TurnValidator checks if the attempted turn, whether a jump or a move, is valid. This class must not have errors since the game logic would break.
- **BoardLoader:** The BoardLoader provides methods for the player to load board states from the past or save current states for the future. Since the initial board is modifiable and is also loaded via that class, it is crucial for the game.
- **MoveStorage:** The MoveStorage Class is basically our stack. It provides methods to push and pop turns such we can revert them in the future if needed. It therefore must store all the turns executed. It is also crucial since turns cannot be reverted if the class fails.



Ex. 2.3

The average line coverage of our 10 most important cases is 90.67%. Overall, we have a line coverage of 46% since there are some classes which cannot be tested. Especially everything concerning the UI cannot be tested since JavaFx does not allow unit testing in that way we learned it in class.



Ex. 3.1

The first extension is the implementation of the possibility to undo moves. This requires a stack that we had to create in the first place where we can push and pop turns. The new class 'MoveStorage', which provides two main methods 'push' and 'pop', takes care of this.

To implement the idea, we decided to use the Command Pattern in parallel. This means we first created a 'Command' interface that provides two empty methods, 'execute' and 'undo'. Then, we created the 'CommandTurn' class that takes care of executing all the turns and storing them on the stack. More important though is the new part where we use it to undo the last move. 'Undo' accesses the stack, pops the last entry, and breaks it down to the core objects. So, we receive a turn-object, player-object, and a potential piece-object if one got eaten. With this information we can create an opposite turn that reverts the last move, changes the player if necessary and replaces the eaten piece.

This sequence gets called twice since a player most likely notices his mistake only right before his next turn when the enemy has already done his turn. Thus, we need to revert two single turns so the player can play another move. If one move consists of a multi jump, the complete multi jump will get reverted. If only one move has been done, only this single move gets reverted and if no move has been done, nothing can be reverted. Most importantly though, if a move has been undone, the player cannot undo another move, it's only his last move that he can revert.

Since we already have other 'buttons' in the UI, it makes sense to implement the pattern for them as well. They take care of loading a board state and saving the current board state. Two other new classes 'CommandLoadBoard' and 'CommandSaveBoard' have been created for this. Instead of directly accessing the BoardLoader, we now leave that task to the pattern. The 'undo' methods for these classes have not been implemented since we neither have the buttons or input possibility for this nor do we see the necessity to implement this yet. It may leave room for a future extension though since loading another state does not save the current state automatically.

With more than one button in the UI, we can relatively simply imitate the shown remote control from the lecture. Since the buttons are not completely equal, implementing a method that assigns the commands to a button would be unnecessary, thus we directly assign it where needed in the GameHandling class. This means we create global variables of the type 'Command' (l. 71-73) and assign each corresponding command to them in the initialize method (l. 116-118). When the UI recognizes a button press, the 'execute' (or 'undo') method gets called (l. 259, 266 & 295).

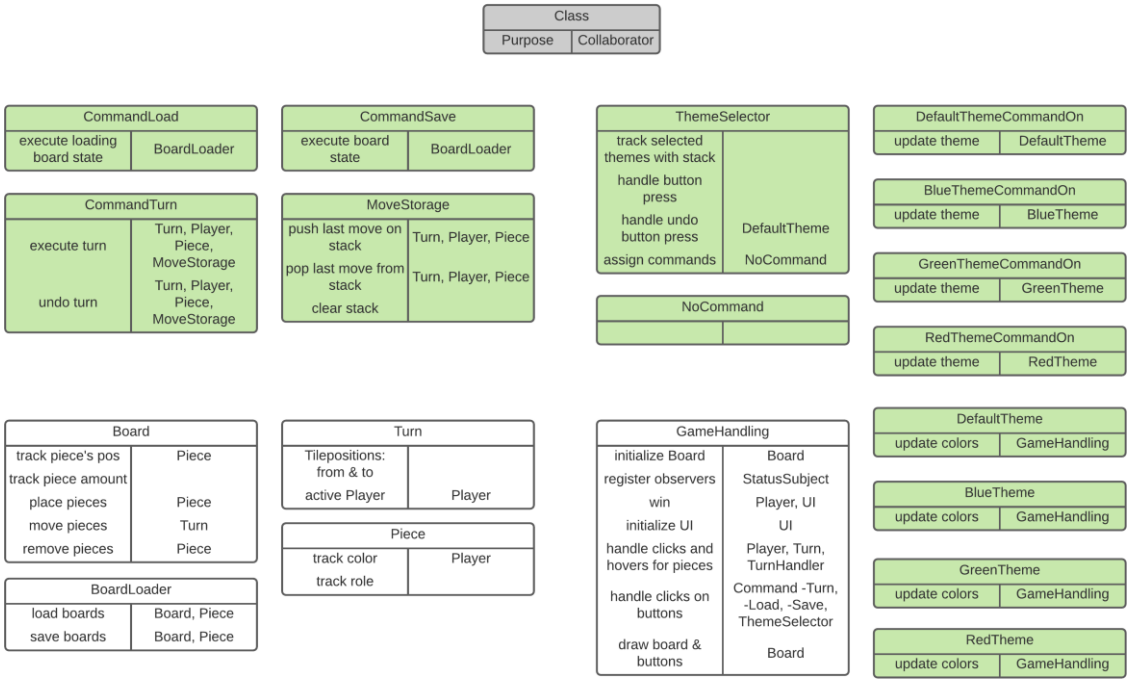
Executing the turn is a different story. Since this is not a direct button press but more of an indirect press through the piece and since it is the only call of a command in the TurnHandler class, it makes no sense to create global variables and initiate them separately whatsoever. For that reason, we simply call the constructor and then the 'execute' method directly afterwards when a turn needs to be executed (l. 68-69).

The Theme Selector allows the user to change the design of the UI. In particular by modifying the colors of some UI components.
On the main menu, a new button is added which leads to the Theme Selector, there the user can select a theme from a list of predefined themes.
The different themes modify the colors of the boards that are seen on the theme-selector page, the main menu as well as the game page.
On top of that, the user is also given the choice to revert to previously selected themes via the undo

button. Selecting the same theme more than once in a row doesn't update the previously selected theme, which allows to always go back to the last distinct theme.

To store the themes and the commands corresponding to the themes, new packages have been added to keep things organized.

Ex. 3.2



The green cards, again, are newly created cards for this assignment. Black cards have existed before already. Only the cards for the extensions and their dependent cards are depicted here.