Assignment 3

Team number: 14
Team members:

Name	Student Nr.	Email
Jop Zitman	2670863	j.zitman@student.vu.nl
Tiberiu Iancu	2659445	t.iancu@student.vu.nl
Sebastian Cristian Iozu (Chris)	2663383	s.iozu@student.vu.nl
Yingdi Xie	2669853	y7.xie@student.vu.nl

Formatting conventions:
- Functions: *italics()*
- Attributes: `consolasAttribute`
- Classes: **BoldClass**
- Constructors: <<create>>(param1, param2)
- Abstract classes: mark with {abstract}
- Static: <u>staticField</u>

Summary of changes of Assignment 2

Author(s): Jop Zitman, Chris Iozu, Yingdi Xie, Tiberiu Iancu

Implementation:
Most importantly, we updated the **GameState** to have less responsibilities. This means that all operations were separated and put inside single **Command** classes. Furthermore, the **Buffer** and **Sequence** logic has been given its own class and the scoring logic has been put inside a static class. Getters and setters have been added for attributes that could be made read only or private. Finally, many refactorings have been done to address the feedback that was given in the peer review.

Class Diagram:
The main feedback was that the diagram was disconnected (largely due to Cell associations not being modelled). All associations have been added, regardless of their importance to the understanding of the diagram. In the diagram, many associations to LibGDX classes have been removed, as they were either incorrect or did not help understand the diagram (in the corresponding text, we added links to the original documentation of these classes). In this version, we only specified multiplicities where it wasn't implied by 1. Next, we greatly improved the class diagram text, with the main goal of conveying our design decisions. All methods, attributes, and associations have been properly explained, and alternative design decisions have been mentioned (e.g. use of extending Game compared to extending ApplicationAdapter).

Object Diagram:
The object diagram has been updated to match the class diagram. We also found a better way to model **Cell** objects. The abstract class Game has been completely removed.

Sequence Diagram:
The Game Rendering sequence diagram has been updated to account for the new abstract methods of GameScreen and the renaming of methods of GUIelement. The User Input Handling sequence diagram has been updated to account for the new **Command** classes. These diagrams now cover the most important parts of the project.

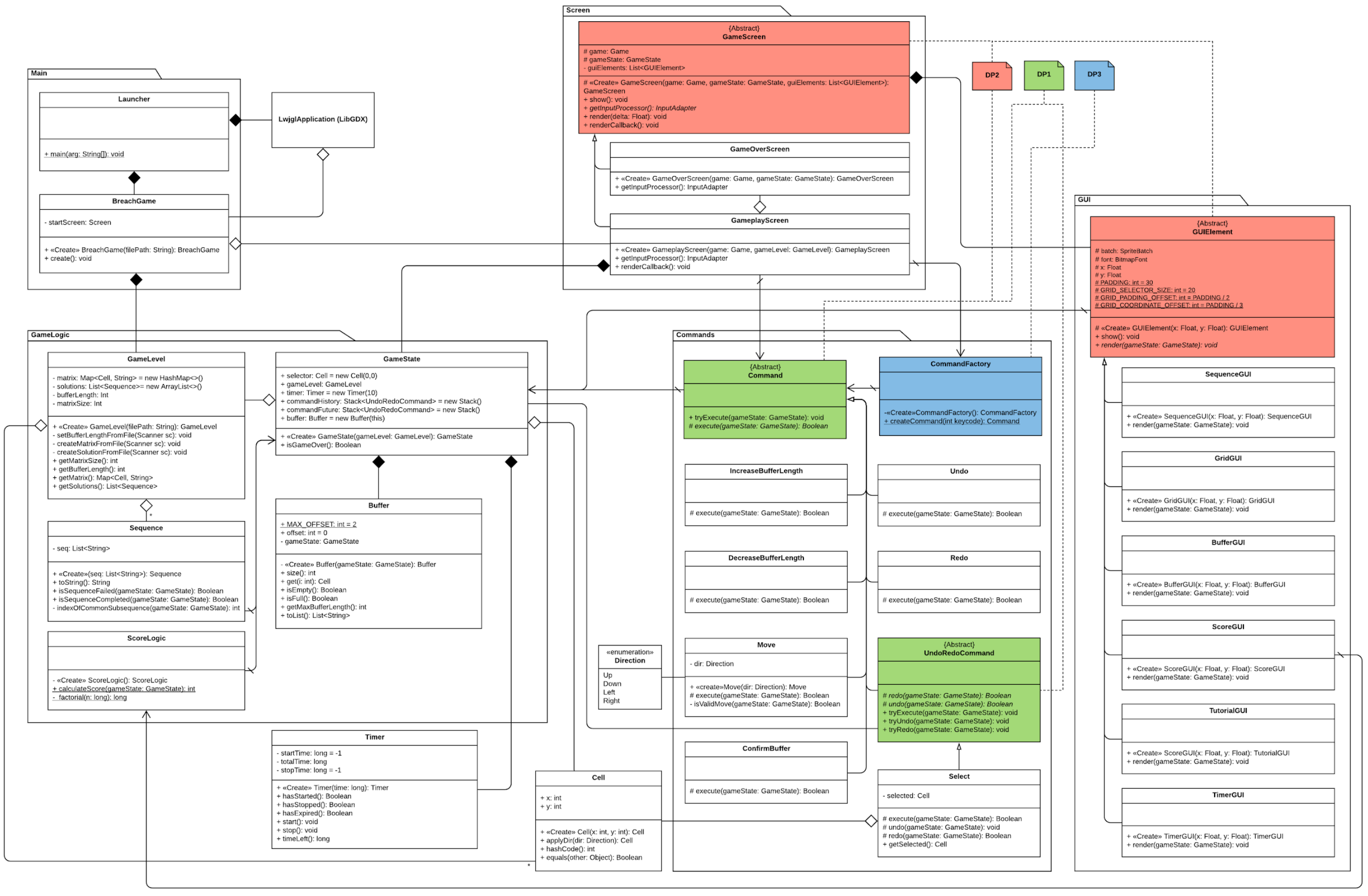
State Machine:
The state machine for the game's state has been updated, because the responsibility of the GameState class has changed so much. The state machine now mentions commands, and it accounts for the fact that

the user can only change the buffer size before starting the game. The GameLevel state machine text has been expanded upon.

Application of design patterns

Author(s): Jop Zitman, Chris Iozu, Yingdi Xie, Tiberiu Iancu

Link to image



	DP1
Design pattern	Command
Problem	The player's key command was handled inside the **GameplayScreen** class which in turn invoked the corresponding operations of the **GameState** class. The auxiliary functions for command handling were also implemented inside **GameState**. It resulted in a huge area of responsibility for **GameState** and also coupled the command's actions with the way of executing the actions, thus making the implementation less readable and maintainable.
Solution	We created an abstract class **Command** for all commands except "Escape" which exits the game application rather than modifying the game's concrete state. The abstract child class **UndoRedoCommand** was created for the undo/redo-able commands (i.e., "Select" in the current version and in the future releases can be extended to other commands such as "In-/DecreaseBufferLength"). The concrete command classes implement the actual actions taken when the command is triggered. Besides, **GameState** maintains a history of commands with a stack construct.
Intended use	At run-time, when the player presses a key, a command object will be dynamically created via a **CommandFactory** (see below). Then, *tryExecute()* method will be called which carries out the execution. The "commandHistory" inside **GameState** supports

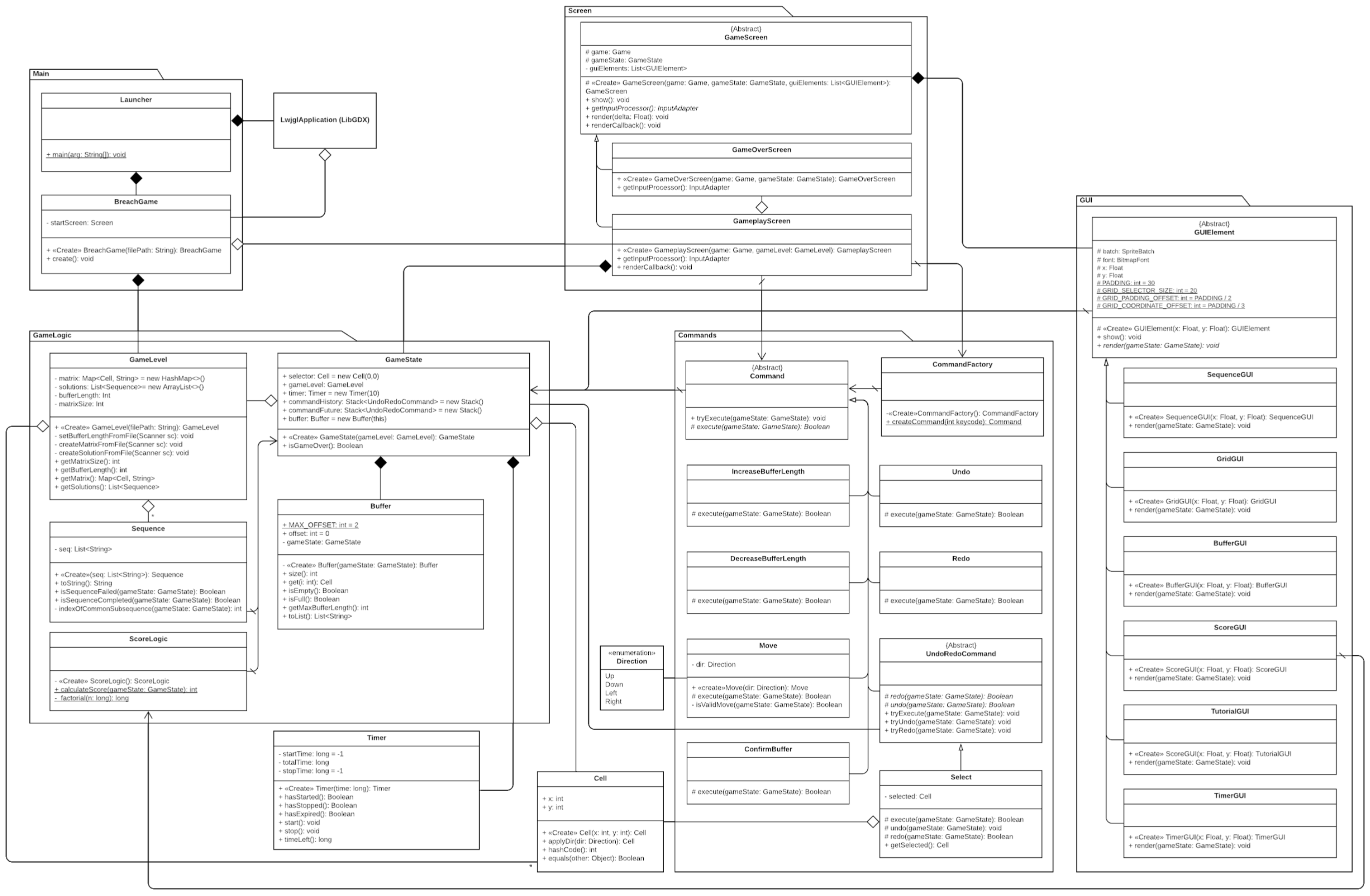
	reverting the last action and the last undone execution for replayable commands. The details for intended use is also illustrated in the “User Input Handling Sequence Diagram” (see below).
Constraints	One constraint is that for the concrete command classes to be able to alter the game’s state, a reference of **GameState** needs to pass into the execution operations so that the attributes of **GameState** (i.e., the references of objects managing the game’s state) are accessible.

	DP2
Design pattern	Template Method
Problem	Some classes contained similar operations and operations of a similar structure. For example, the **GameplayScreen** and **GameOverScreen** classes both implemented *render()* and *show()* methods and inside these methods there was similar code for the rendering and displaying of GUI elements. Similarly, all commands implement “do”, “undo” and/or “redo” of some actions which modify the game’s state.
Solution	We applied the Template Method to extract the similar logic and implement it in a super class. Specifically, we created an abstract class **GameScreen** which implements the similar steps inside the *render()* and *show()* methods while providing hooks for the child classes to implement the class-specific logic. We also constructed abstract classes **Command** and **UndoRedoCommand** for the similar structure in command handling, whereas the different actions were extended in the concrete command classes.
Intended use	At run-time, when the operations of an instance of **GameScreen** are invoked, the similar steps implemented in this superclass will be executed. Then, the methods which serve as a hook and are specified in the subclass such as **GameplayScreen** and **GameOverScreen** will be invoked and executed. The similar intended use applies for the command classes.
Constraints	One constraint is that the classes which inherit the abstract superclasses are forced to implement and specify the operations serving as a hook. Besides, the structure of the methods implemented in the superclass must be followed and need to be modified to suit the needs of the subclasses in some situations.

	DP3
Design pattern	Factory Method
Problem	The handling of all user commands were instrumented in the **GameplayScreen** and **GameOverScreen** classes. This part of code interleaved with the implementation for rendering and displaying the game screens which are the main responsibilities of the screen classes, and thus the code turned out to be less manageable and readable.
Solution	We created a dedicated **CommandFactory** class so that the command objects and the handling of user commands are separate from the main part of the game screen classes. Besides, a concrete command object can be created depending on the player’s keypress at run-time. It also allows our game to be more customizable in the future.
Intended use	When the player presses a key, the keycode retrieved from the **InputAdapter** of LibGDX is used as the argument for the **CommandFactory** to create a corresponding command object. In case of an invalid keypress, the “null” value will be returned.

Class diagram

Author(s): Jop Zitman, Chris Iozu, Tibi Iancu, Yingdi Xie
Link to image



Launcher

Launcher is used for initializing the application. This includes configuring a new window, creating a BreachGame object, and passing the file path from the command argument. This class is implemented to separate application initialization from game initialization.

Operations

- *main()* - the entrypoint for this application.

Associations

- **LwjglApplication** - Composition. Class included in and required by LibGDX to be used for the mentioned application creation. In this project, only the **Launcher** can create application windows.

BreachGame

BreachGame is used for game initialization and more importantly it extends the Game class from LibGDX to provide the ability to switch between game screens (which is not provided in the alternative ApplicationAdapter). The following implementation is a direct “consequence” of using these screens.

Attributes

- `startScreen`: **Screen** - a store for the first screen so that the *create()* method can acquire it.

Operations

- *create()* - sets the screen to `startScreen` and is called by LibGDX when it first acquires this **BreachGame** object (through the **Launcher** above).

Associations

- **GameplayScreen** - Shared aggregation. **GameplayScreen** is initialized in the constructor and saved in `startScreen`, but after LibGDX has called the *create()* method, the screen ownership is moved to LibGDX's abstract class **Game**, which has a composition relationship to all Screens.

GameScreen {Abstract}

Each screen extends the **ScreenAdapter** class from LibGDX which gives a basis for the methods needed for rendering. This class is a base class for the different screens and provides encapsulation for rendering GUIElements and the configuration of an input handler.

Attributes

- `game`: **Game -** a reference to the LibGDX library used to switch screens and set the input adapter.
- `gameState`*:* **GameState** - a reference of **GameState** so that each **GUIElement** can read from it.
- `guiElements`: **List<GUIElement> -** a list of GUI elements that must be rendered on the screen.

Operations

- *show()* **-** called by LibGDX before the first render (i.e. after setting the screen). This call is forwarded to each of the guiElements and sets input handler by calling the abstract getInputProccesor (InputProcessor).
- *getInputProcessor()* - an abstract method to get an InputProcessor for keyboard input (required implementation by subtypes).
- *render(*delta: float*)* - called by LibGDX for each render. **GameScreen** again forwards the call to all **GUIElements**. Furthermore, in this method **GameScreen** checks if the game is over and conditionally switches to a **GameOverScreen**.
- *renderCallback()* - an empty method that is called in each render by *render()*. Subtypes can override it to execute code for each render.

Associations

- **GameplayScreen** - Generalization. This class manages the creation of all the gameplay ui elements such as the grid, buffer, sequences, etc. It also configures the **CommandFactory** as an input processor. It overrides the render callback to check if the game is over on each render (so that it can switch screens to **GameOverScreen**.
- **GameOverScreen** - Generalization. This class manages the creation of all the game over ui elements such as the score and completed sequences. It configures a simple input processor and does not override the render callback.

GUIElement {Abstract}

GUIElements are used to modularize certain elements of the user interface. The attributes below are included in this class so that they can be reused in every extending class.

Attributes

- `batch`*:* **SpriteBatch -** object used to render elements on screen.
- `font`: **BitmapFont -** object used to render text.
- `x`: **Float -** horizontal position of element on screen.
- `y`*:* **Float -** vertical position of element on screen.
- `PADDING`*:* **Int -** used to add space between elements.
- `GRID_SELECTOR_SIZE`*:* **Int -** represents the size of the selector (blue box that hovers on a cell)
- `GRID_PADDING_OFFSET`*:* **Int -** represents an offset added on top of `PADDING` to increase/reduce the position of the selectable area (gray box that hovers on top of the selectable row/column)
- `GRID_COORDINATE_OFFSET`*:* **Int -** scales the selectable area to fit an entire row/column

Operations

- *create()* - sets the horizontal and vertical positions of the element.
- *show()* - called on first render by a screen. Initializes the SpriteBatch and BitmapFont objects.
- *render(*gameState: **GameState***)* - called on each render by a screen. Draws the GUI element using information from the **GameState** object. This is an abstract method, suggesting that the extending classes should override and implement it.

Associations

- **GameState** - Directed Association. **GUIElement** should access **GameState** to retrieve the information needed in rendering of the UI, whereas **GameState** cannot and is not interested in the fields of a **GUIElement**.
- **GridGUI** - Generalization. **GUIElement** for rendering the grid and hover effects.
- **TimerGUI** - Generalization. **GUIElement** for rendering the timer (in white color before the start and in red during the timing).
- **ScoreGUI** - Generalization. **GUIElement** for rendering the score and corresponding win/lose texts.
- **BufferGUI** - Generalization. **GUIElement** for rendering the current buffer and the contained strings.
- **SequenceGUI** - Generalization. **GUIElement** for rendering each of the requested sequences (in green once completed, in red once failed and in white elsewise).

GameLevel

GameLevel encapsulates all information about the game's setup. It is initialized when the game is loaded and does not change throughout the game. The getters support retrieval of the relevant setup information and enforce their immutability.
Ideally we believe that `matrixSize` and `bufferLength` should be final attributes, but we decided against it to support splitting up the constructor in multiple methods.

Attributes

- `matrix`: **Map**<**Cell**, **String**> - represents the matrix where the player can navigate and select cells. It is initialized as a mapping from **Cell** to the alphanumeric letters residing in the cell.
- `solutions`: **List**<**Sequence**> - represents a list of sequences to be matched by the selected cells in the buffer.
- `bufferLength`: **int** - represents the buffer's size, in other words, the maximum number of cells which the player can select during the game.
- `matrixSize`: **int** - represents the width and height of the matrix in cells. It is used to draw the GUI and check whether a move is out of bound.

Operations

- *create(*filePath*:* **String**) - using the filePath, it creates a Scanner for the file and parses it using the functions below. It throws a "**FileNotFoundException**" in case that the file path is not valid.
- *setBufferLengthFromFile(*sc: **Scanner***)* - reads the buffer length from the scanner and saves it into `bufferLength`.
- *createMatrixFromFile(**Scanner** sc)* - parses a matrix of variable size from the scanner and saves it into matrix. It also saves the read matrix size into `matrixSize`.
- *createSolutionFromFile(**Scanner** sc)* - parses the solution sequences from file and saves them into `solutions`.
- *getMatrixSize():* **int** - getter for private attribute matrixSize.
- *getBufferLength():* **int** - getter for private (non-final) attribute `bufferLength`. Same thing for final attribute here.
- *getSolutions():* **List**<**Sequence**> - getter for private attribute solutions. It makes a shallow copy so that the map can't be changed.
- *getMatrix():* **Map**<**Cell, String**> - getter for private attribute solutions. It makes a shallow copy so that the map can't be changed.

Sequence

This class is implemented to encapsulate the sequence logic and encapsulate its internal representation of **List**<**String**>. **GameLevel** creates a bunch of Sequences from a file which the **SequenceGUI** then uses when rendering (color-coding based on the completion/failure of sequences) and the **ScoreLogic** uses to calculate the scores.

Attributes

- `seq`: **List**<**String**> - internal representation of the sequence.

Operations

- *create()* - copies the given **List**<**String**> to `seq`.
- *toString()*: **String**- override default Java behaviour to prettyprint the sequence.
- *isSequenceCompleted(*gameState: **GameState***):* **Boolean** - checks whether a sequence has been completed. A completed sequence is one which is matched by consecutive cells in `buffer`.
- *isSequenceFailed(*gameState: **GameState***):* **Boolean** - checks whether a sequence cannot be completed. If the length of unmatched cells of a sequence are longer than the remaining free slots in `buffer`, it is failed.
- *indexOfCommonSubsequence*(firstSequence**: List**<**String**>**,** secondSequence: **List**<**String**>*):* **int** - This function takes two sequences (lists of strings) as input and returns the index of the longest postfix in firstSequence that is matched by the prefix of secondSequence. -1 is returned in case of no common subsequences. This is useful for comparing a sequence to the buffer (represented as a sequence).

Associations

- **GameState** - Directed Association. The methods require access to gameState for comparing the sequence to the current buffer.

GameState

This is the actual core of the game and keeps track of the global state. This class does not hold much logic or any interactions; it's really a state. **GUIElements** use it to read the current state and **Commands** use it to change the current state.

Attributes

- `selector`: **Cell** - represents the selected cell when the player moves around inside the grid.
- `gameLevel`: **GameLevel** - the loaded gameLevel copied from the constructor and saved in the game's state (for easy access in classes such as **Move** for bounds checking).
- `timer`: **Timer** - a custom timer implementation that is used to determine whether the game has ended and to calculate the current score.
- commandHistory: Stack<**UndoRedoCommmand**> - a history of executed undoable commands is saved to allow for undoing.
- `commandFuture`: Stack<**UndoRedoCommmand**> - a 'future' of commands that were undone. These are saved for when users want to redo commands.
- `buffer`: **Buffer** - an object that is used to retrieve information about the buffer.

Operations

- *create()* - it sets the game level to **GameLevel** and sets the selector to the default position.
- *isGameOver():* **Boolean** - checks if the current state corresponds to a gameover state.

Associations

- **GameLevel** - Shared aggregation. **GameState** holds a reference to an instance of **GameLevel** to access information about the game's setup. **GameLevel** can exist independent of **GameState**.

- **TimerLogic** - Composition. **GameState** holds a reference to an instance of **TimerLogic** to access the timer's information and start the timer. **TimerLogic** should not exist outside **GameState**.

Buffer

This class allows for easy access to the buffer as a list of strings. The actual buffer elements are stored inside the **Select** commands as **Cells** and need to be parsed to **List**<**String**> for easy access. Furthermore, we separated this logic from the **GameState** class to better separate their concerns.

Attributes

- `offset`: **int** - a variable that determines an offset to how many elements can be contained in the buffer. The default buffer size can be found in `GameLevel` and this offset increases/decreases that value. See also IncreaseBufferLength, DecreaseBufferLength.
- `MAX_OFFSET`: **int** - the maximum offset allowed.
- `gameState`: **GameState** - reference to the **GameState** saved. Every operation needs access to the game state so it is easier to reference it as a variable.

To improve encapsulation, operations such as get, size, isEmpty have been added to reduce the complication of accessing the buffer (`buffer`.*toList().size()* vs `buffer`.*size()*). Accessors of the `buffer` shouldn't need to know that the buffer needs to be converted to a list first.

Operations

- *create()* - saves the `GameState` to the private gameState variable.
- *size():* **int** - conveniency operation for accessing the size of the buffer.
- *get():* **int** - conveniency operation for accessing an element in the buffer.
- *isEmpty():* **Boolean** - conveniency operation for determining if the buffer is empty.
- *isFull():* **Boolean** - conveniency operation for determining if the buffer is full.
- *toList():* **int** - conveniency operation for the buffer as a list.
- *getMaxBufferLength():* **int** - returns the sum of the buffer size as specified in the **GameLevel** and the buffer offset.

Associations

- **GameLevel** - Shared aggregation. **GameState** holds a reference to an instance of **GameLevel** to access information about the game's setup. **GameLevel** can exist independent of **GameState**.
- **TimerLogic** - Composition. **GameState** holds a reference to an instance of **TimerLogic** to access the timer's information and start the timer. **TimerLogic** should not exist outside **GameState**.

ScoreLogic

This class is responsible for the score calculation on the **GameOverScreen**. This class is implemented as a static class to separate its logic from the **GameState** but keep it accessible for all classes that have access to the **GameState**, such as the **ScoreGUI**.

Operations

- *create()* - an empty private constructor so that this "static class" can't be initiated.
- *calculateScore(*gameState: **GameState***):* **int** - static class implementation of the aforementioned score calculation.
- *factorial*(n: **long**): **long** - implementation of factorial because Java doesn't have a built-in one.

Associations

- **GameLevel** - Shared aggregation. **GameState** holds a reference to an instance of **GameLevel** to access information about the game's setup. **GameLevel** can exist independent of **GameState**.
- **TimerLogic** - Composition. **GameState** holds a reference to an instance of **TimerLogic** to access the timer's information and start the timer. **TimerLogic** should not exist outside **GameState**.

Command

While **ScoreLogic** and **Buffer** mostly take observing roles, the **Command** is built with the changing of `gameStates` in mind. The commands are separated from the **GameState** to separate responsibilities (i.e. **GameState** keeps state and **Command** edits state). Furthermore, implementing new **Commands** can be done without the knowledge of other existing commands, improving overall encapsulation.

Operations

- *tryExecute(*gameState: **GameState***)* - tries to execute the abstract method *execute()*. This separation is built so that every command is required to execute the "extra" code implemented in *tryExecute()*.
- *execute(*gameState: **GameState***):* **Boolean** - actually executes the command. This is an abstract method, suggesting that the extending classes should override and implement it.

Associations

- **GameState** - Directed Association. **Command** and its subtypes should have access to **GameState** so that they can edit the corresponding fields and access necessary information to complete the command, whereas **GameState** cannot and is not interested in the fields of a **Command**.
- **GameplayScreen** - Directed Association. **GameplayScreen** executes commands that were returned by the **CommandFactory** without the **Command** knowing of the **GameplayScreen**.
- **IncreaseBufferLength** - Generalization. Increases the buffer length (only if the game hasn't started and maximum length has not been reached).
- **DecreaseBufferLength** - Generalization. Decreases the buffer length (only if the game hasn't started and minimum length has not been reached).
- **ConfirmBuffer** - Generalization. Makes the user selected buffer elements final (in the implementation by stopping the timer).
- **Move** - Generalization. Moves the user's selector to a certain direction.
- **Undo** - Generalization. Undoes the last undoable command (**UndoRedoCommand**). The last **Command** will be popped from the `commandHistory`, its *tryUndo()* method is called and it is pushed onto the `commandFuture`.
- **Redo** - Generalization. Redoes the last redoable command (**UndoRedoCommand**). The last **Command** will be popped from the `commandFuture`, its *tryRedo()* method is called and it is pushed onto the `commandHistory`.
- **UndoRedoCommand** - Generalization. Abstract class for commands that should be able to be undone/redone.

UndoRedoCommand

Commands that should be undo-/redoable should extend this class instead of **Command**. **UndoRedoCommand** extends **Command** and provides template methods so that for each execution/undo/redo the according `commandHistory`/`commandFutures` are updated. This is important because in practice, once you execute a new command, you should not be able to redo any previous commands anymore.

Operations

- *redo(*gameState: **GameState***):* **Boolean** - called by *tryRedo()* for redo specific code.
- *undo(*gameState: **GameState***):* **Boolean** - called by *tryUndo()* for undo specific code.
- *tryExecute(*gameState: **GameState***)* - tries to execute the abstract method *execute()*. This separation is built so that every command is required to execute the "extra" code implemented in *tryExecute()*.
- *tryUndo(*gameState: **GameState***)* - tries to execute the abstract method *undo()*. This separation is built so that every command is required to execute the "extra" code implemented in *tryUndo()*.
- *tryRedo(*gameState: **GameState***)* - tries to execute the abstract method *redo()*. This separation is built so that every command is required to execute the "extra" code implemented in *tryRedo()*.

Associations

- **GameState** - Binary Association. **Command** and its subtypes should have access to **GameState** so that they can edit the corresponding fields and access necessary information to complete the command. Furthermore, **GameState** keeps track of a stack of **UndoRedoCommands** and thus also wants access to **UndoRedoCommands**.
- **Select** - Generalization. Saves the current selector into the command, if the buffer isn't full.

CommandFactory

A simple class that allows for the creation of commands based on user input. As described in the design patterns, it improves maintainability by the separation of concerns. The screen's main role is rendering while the **CommandFactory**'s role is to create commands.

Operations

- *create()* - an empty private constructor so that this "static class" can't be initiated.
- *createCommand(*keycode: **int***)* - creates **Command** object corresponding to the keycode.

Associations

- **GameplayScreen** - Directed Association. The **GameplayScreen** can access this factory to create new commands. The **CommandFactory** does not need access to the **GameplayScreen** to fulfill its task.
- **Command** - Shared aggregation. **CommandFactory** creates **Command** objects but is not responsible for their lifetime.

Timer

Timer is a simple timer implementation. In this project, it is primarily used to keep track of whether the game timer has expired (i.e., gameover). Furthermore, the time left is used to calculate the final score.

Attributes

- `startTime`*:* **long** - unix time in milliseconds, initialized when the timer is started.
- `totalTime`*:* **long** - time in milliseconds until the timer ends.
- `stopTime`*:* **long** - unix time in milliseconds of when the timer has stopped.

Operations

- *create()* - creates a new **TimerLogic** object with proper totalTime (in ms).
- *hasStarted():* **Boolean** - returns whether the timer has been started.
- *hasStopped():* **Boolean** - returns whether the timer has been stopped.
- *hasExpired():* **Boolean** - returns whether the timer has run out.
- *start()* - start the timer.
- *stop()* - stop the timer.
- *timeLeft():* **long-** returns the amount of seconds left until the timer ends.

Cell

A class that helps with coordinates and directions. It is used in **GameLevel** in a **Map**<**Cell, String**> to specify what coordinates belong to what values in the grid. Furthermore, the use of **Cell** helps in the **Move** command where the accessor can easily obtain the coordinate given a certain direction. Alternative would be a tuple, but that would unencapsulate the logic of *applyDir*.

Attributes

- x*:* **int** - represents a horizontal coordinate (starting on the left by LibGDX).
- y*:* **int** - represents a vertical coordinate (starting on the bottom by LibGDX).

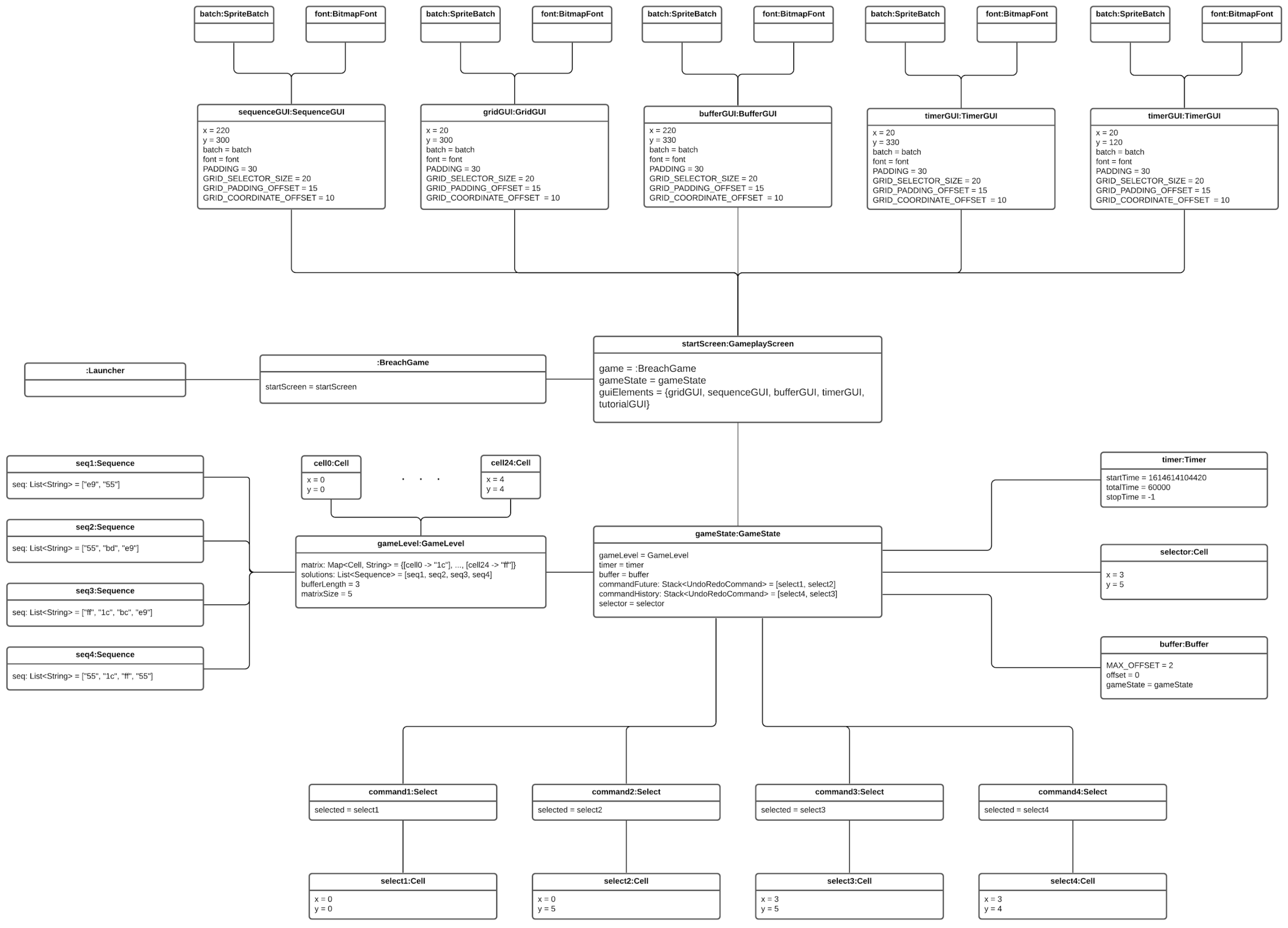
Operations

- *create()* - creates a new **Cell** object and saves x and y.
- *applyDir(*dir: **Direction***)*: **Cell** - this function returns a new **Cell** object representing the outcome of moving in the given direction.
- *hashCode()*: **int** - overridden function used internally by `matrix` **HashMap** in **GameLevel** to store level information. Without implementing this function, two different **Cell** objects with the same coordinates would be mapped to different objects in `matrix`.
- *equals(*givenCell: **Object***)*: **Boolean** - overridden function that facilitates the comparison of different **Cell** objects.

Object diagram

Author(s): Tibi Iancu, Chris Iozu

[Link to image]



The diagram captures a snapshot of the system during level 1 of the Breach Game. In this model, the user has already performed 4 select actions, namely on cells with coordinates (0, 0), (0, 5), (3, 5), (3, 4), then issued 2 undo commands. The active components in the system during the snapshot are described below.

- **:Launcher** - Entry point of the program; responsible for initializing **BreachGame**.
- **:BreachGame** - This is the game object required by LibGDX and it gets created in *main()*. `startScreen` is set to a new **GameplayScreen** at the start of the program execution.
- `startScreen:` **GamePlayScreen** - Initializes `gameState` and `guiElements` when the main screen is supposed to be shown.
- `guiElements:` List<**GUIElement**> - List of **GUIElement**'s containing all necessary information for rendering the main game screen. Each **GUIElement** is initialized with its own x and y coordinates (that represent the position on the screen). They all share the static constants `PADDING`,

GRID_SELECTOR_SIZE, GRID_PADDING_OFFSET and GRID_COORDINATE_OFFSET, that are initialized upon class creation.

- gameState**: GameState -** Holds all necessary game logic information. On game start, gameLevel, timer, buffer, commandFuture and commandHistory are initialized.
- gameLevel: **GameLevel -** Keeps track of game level data. matrix, solutions, bufferLength and matrixSize are parsed from the given file and never change.
- seq1, seq2, seq3, seq4: **Sequence -** These are the sequences loaded in gameLevel that the user needs to complete in order to gain points. Each one is represented by a list of **Strings**.
- timer: **Timer -** startTime is initialized with 1614614104420 (UNIX time in milliseconds) of when the user selected the first cell. The total time the user has to complete the level is 60000 milliseconds, as shown in totalTime. stopTime equals -1, as the timer hasn't been stopped yet.
- selector: **Cell** - Hold the current position of the cursor. In this instance it is at coordinates (3, 5)
- buffer: **Buffer -** Keep track of MAX_OFFSET, the maximum number of cells the user can extend the buffer by and offset (= 0 in our case), the number of cells the user has already extended the buffer by.
- commandFuture and commandHistory: Stack<**UndoRedoCommand**> - These two stacks hold relevant information as to what moves the user has made and what moves have been undone respectively. In this case, the user performed actions command1, command2, command3 and command4, then chose to undo command4 and command3; as such the stacks contain [command1, command2] and [command4, command3] respectively.
- command1, command2, command3, command4: **Select** - These are the four select commands issued by the user. Internally they store which cell was selected, so undo's and redo's can later be done. Note: these are the only commands that appear in the object diagram. That is because whenever a command of another type is issued, it is immediately consumed.
- select1, select2, select3, select4: **Cell -** Each one of these corresponds to one of the issued **Select** commands and they memorize which cell was selected.

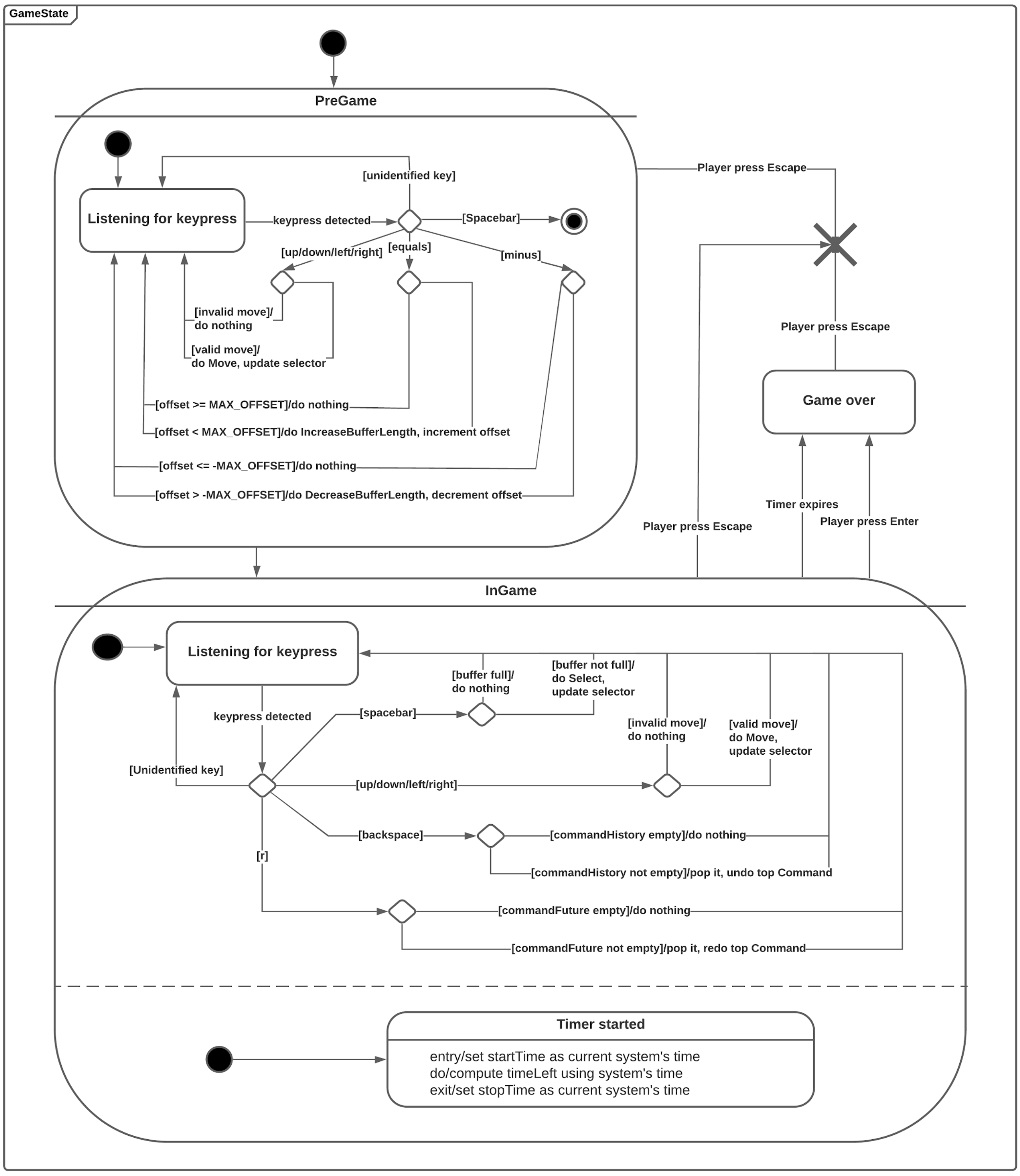
State machine diagrams

Author(s): Yingdi Xie, Jop Zitman, Chris Iozu

In this section, we will describe 2 UML state machine diagrams for two classes of our game. The first diagram depicts the transition between different states for the **GameState** class, that is, the state changes of one main class encapsulating the overall game's information. The second diagram illustrates the transition between states for the **GameLevel** class, which is responsible for parsing game files and initializing the game's setup.

GameState

Link to image


GameState
PreGame
Listening for keypress
keypress detected
[unidentified key]
[Spacebar]
[up/down/left/right]
[equals]
[minus]
[invalid move]/ do nothing
[valid move]/ do Move, update selector
[offset >= MAX_OFFSET]/do nothing
[offset < MAX_OFFSET]/do IncreaseBufferLength, increment offset
[offset <= -MAX_OFFSET]/do nothing
[offset > -MAX_OFFSET]/do DecreaseBufferLength, decrement offset
Player press Escape
Player press Escape
Game over
Timer expires
Player press Enter
Player press Escape
InGame
Listening for keypress
keypress detected
[Unidentified key]
[spacebar]
[buffer full]/ do nothing
[buffer not full]/ do Select, update selector
[up/down/left/right]
[invalid move]/ do nothing
[valid move]/ do Move, update selector
[backspace]
[commandHistory empty]/do nothing
[commandHistory not empty]/pop it, undo top Command
[r]
[commandFuture empty]/do nothing
[commandFuture not empty]/pop it, redo top Command
Timer started
entry/set startTime as current system's time
do/compute timeLeft using system's time
exit/set stopTime as current system's time

The state machine diagram above illustrates the transitions between different states of the **GameState** class. As shown in the class diagram, **GameState** holds the instances of several classes representing parts of our game and is responsible for encapsulating and keeping track of the current game's state, and thus the transitions of its states represent the dynamics when the application is running. The state changes are explained as follows:

- **PreGame** - **GameState** transits to this state immediately after it is initialized by **GameplayScreen**. This state is distinct from the next state (i.e., "InGame") in that different key pressing events will trigger the execution of **Commands** and the change in **GameState**. For instance, pressing "Equals/Minus" keys will modify the game's state while having no effect in the "InGame" state. On entry, the game starts listening for a keypress. Once a keypress is detected, the key is checked against the following guard conditions:
    - If the key pressed is "Up/Down/Left/Right", it transits to a decision state where it checks if the move is valid. If so, it moves the `selector`; elsewise, it does nothing and transits to the listening state.
    - If the key pressed is "Equals", it compares the `buffer`'s offset with the maximum offset. If the offset is smaller, it executes the **IncreaseBufferLength** command; elsewise, it does nothing.
    - If the key pressed is "Minus", it compares the `buffer`'s offset with the negative of maximum offset. If the offset is greater, it executes the **DecreaseBufferLength** command; elsewise, it does nothing.
    - If the key pressed is "Spacebar", it goes to the final state of "PreGame".
- **InGame** - a state while the game is running. On entry, it splits into two concurrent substates.

  In one substate, it listens to and handles the user input. Specifically, when a keypress is detected, the transitions depend on the following conditions:
    - If the key pressed is "Spacebar", it checks if the buffer is full. If so, it does nothing and goes back to the listening state. If it is not full, the **Select** command is executed. Note that when "Spacebar" is pressed, the activities taken here are different from those in "PreGame".
    - If the key pressed is "Backspace", it examines whether the `commandHistory` is empty. If so, it does nothing; otherwise, it pops and undoes the top **Command**.
    - If the key pressed is "r", it checks if the `commandFuture` is empty. If so, it does nothing; otherwise, it pops and redoes the top **Command**.
    - If the key pressed is "Up/Down/Left/Right", the transitions are the same as in "PreGame".

  The other substate is where the **Timer** is started and "running" during the game. Specifically, on entry, it saves the current system's time to `startTime`. Then, it computes the remaining time which is used for the rendering of **GameplayScreen**. Before the substate exits, the system's time is saved to `stopTime` which is used to calculate the player's score at the end of the game.
- **Game over** - Whenever the player presses "Escape", the "PreGame" and "InGame" states will exit and the game's state transits to the termination. Besides, the exit of "InGame" can be triggered by pressing "Enter" or the **Timer** becoming expired. Then, the game goes to "Game over" where the game's state is used for the rendering of **GameOverScreen**. As the player presses "Escape", it exits the game and goes to the final terminating state.

The specific details of the actions taken by the system during the transitions are omitted for the sake of simplicity and understandability. These details are described in the Class diagram section as well as in the User Input Handling Sequence section.

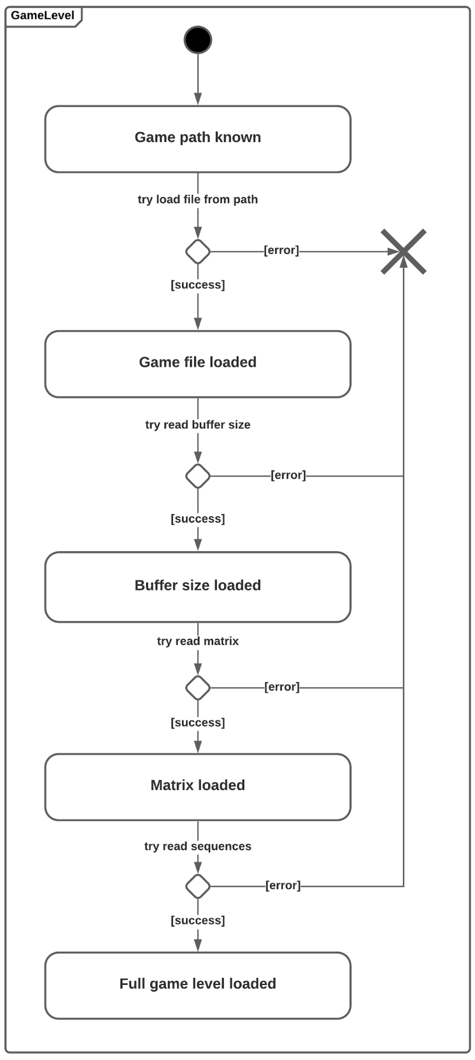
GameLevel

Link to image

The state machine diagram above illustrates the transitions between different states of the **GameLevel** class. The overall transitions represent the state change when the system loads and parses a text file and initializes the corresponding game's setup, which include the following states:

- **Game path known** - Once created by **BreachGame**, **GameLevel** transits to this state where the file path is known from the argument passed by **BreachGame**.
- **Game file loaded** - After it tries to load the file from the path and succeeds, it transits to this state where **File** and **Scanner** objects are constructed and will be used for subsequent states.
- **Buffer size loaded** - The exit from the "Game file loaded" state is triggered by the "try read buffer size" event and in case of success, it goes to this state where `bufferLength` is set.
- **Matrix loaded** - The exit from the last state is triggered by the "try read matrix" event where it tries to scan through the text lines and read the strings into `matrix`. In case of success, it goes to this state where the `matrix` is loaded.
- **Full game level loaded** - The exit from the previous state is triggered by the "try read sequences" event where it tries to scan and read the texts into a list of **Sequences**, (i.e., `solutions`). In case of success, it goes to this state where **GameLevel** is fully loaded.

The error handling in the **GameLevel** class is very simple: try to parse the file and throw exceptions on irregularities. The diagram clearly shows at which points during execution the loading of a file can fail.



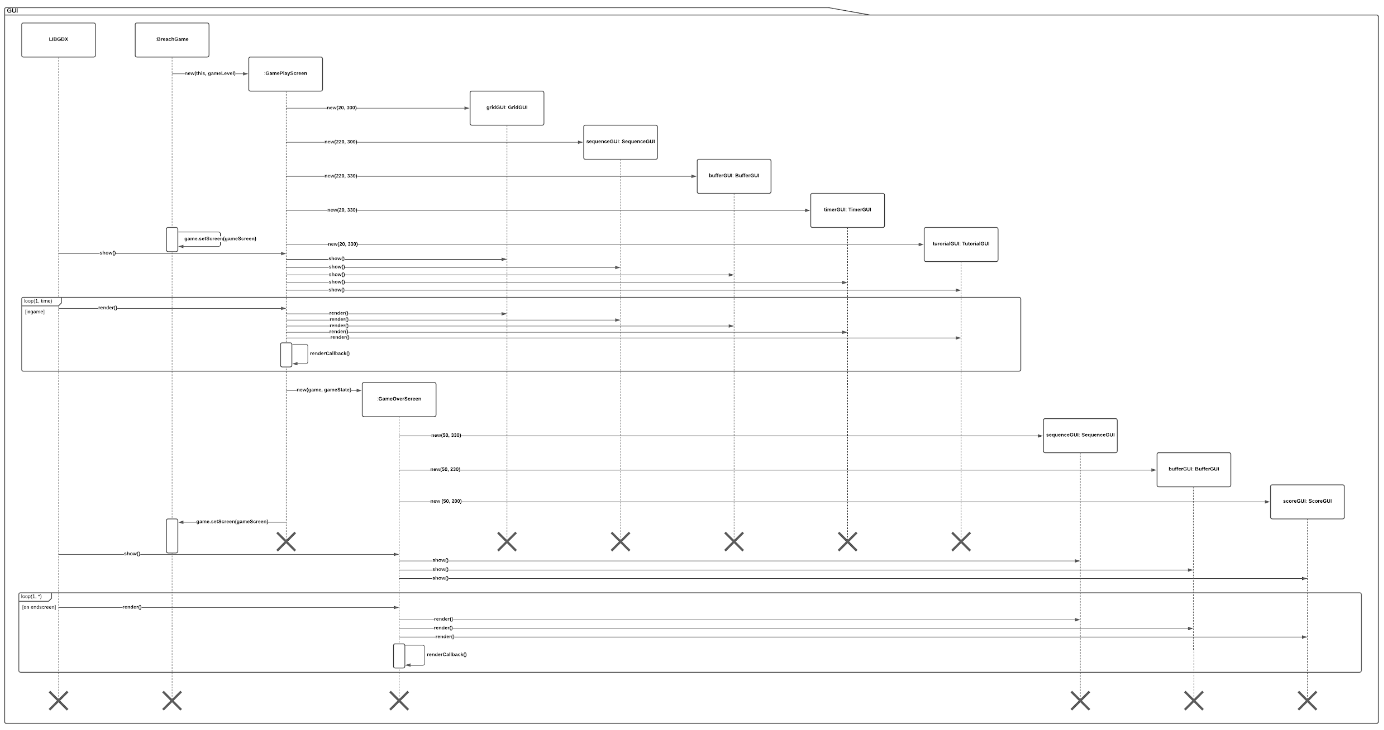
Sequence diagrams

Author(s): Chris Iozu, Jop Zitman

In this section, we will describe 2 UML sequence diagrams for our game. The first diagram depicts the initialization and rendering of different game screens during the game. The second diagram illustrates the interaction between the player and our system while the keyboard input from the player is handled.

Game Rendering Sequence

Link to image



The sequence diagram above depicts the interaction between the different screens and the multiple GUI elements that are part of our system. This diagram covers the initialization of the **GameplayScreen** and the **GameOverScreen**, which extend the **ScreenAdapter** class. Both of these screens actively use multiple **GUIElements** such as **SequenceGUI**, **BufferGUI**, **GridGUI**, **TimerGUI**, **ScoreGUI, TutorialGUI**.

In this diagram, we assume that the **BreachGame** object has just been made. We intentionally hide the interaction with the **GameState** class and user input (like how escape can terminate current execution), and obscure the conditions necessary to switch between the "ingame" and "on endscreen" such as to emphasize the rendering sequence. These conditions will be properly described in User Input Handling Sequence. The rendering objects used by the different GUI elements (such as **SpriteBatch** and **BitmapFont**) are also hidden to make the diagram more understandable (these objects also don't add more value to the diagram).

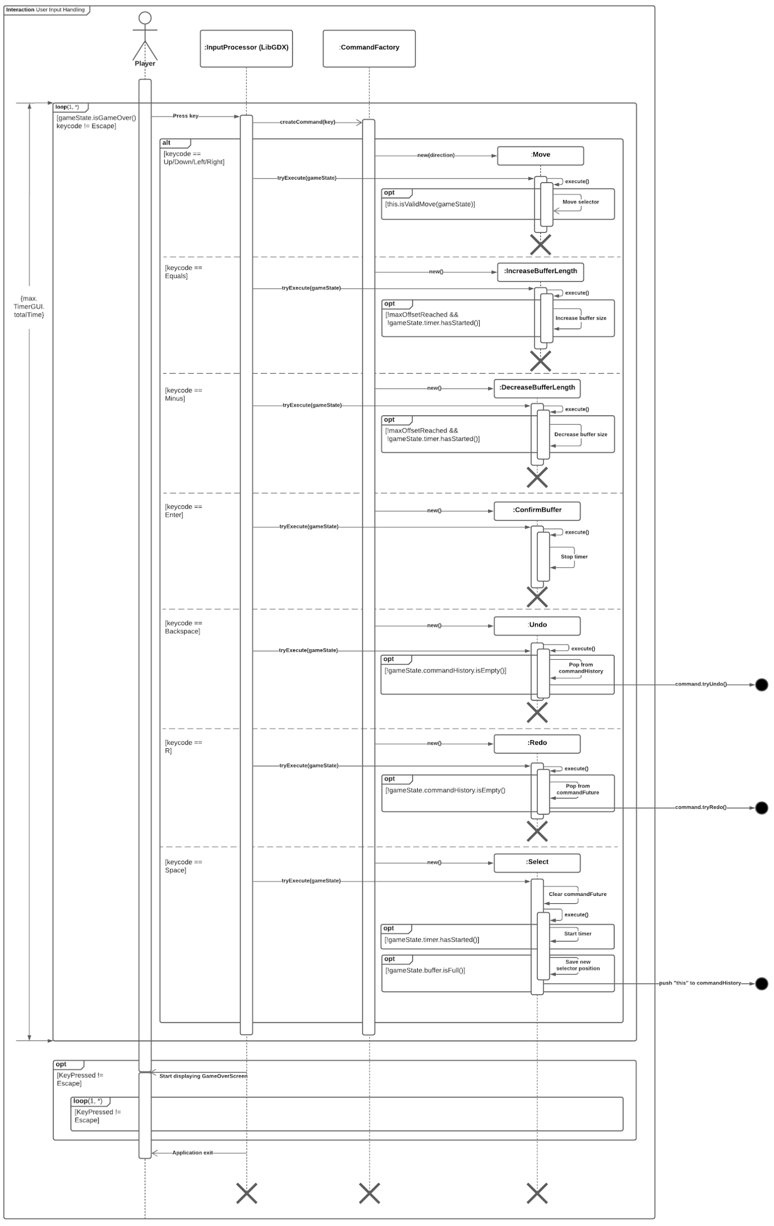
Below is a more detailed description of how the screens are rendered:

- GameplayScreen initialization and rendering:
    - The `breachGame` object invokes the `gameScreen` object, which in its constructor starts initializing the to be used GUI elements: `gridGUI`, `sequenceGUI`, `bufferGUI`, `timerGUI`, and `tutorialGUI`.
    - Once the `gameScreen` object has fully initialized, the `breachGame` object decides to hand over execution to the `gameScreen` object, by calling *setScreen(*`gameScreen`*).*
    - LibGDX now starts the rendering sequence by calling *show()* on the current screen: `gameScreen`. The `gameScreen` object forwards this call to all the used GUI elements. These calls are used for initializing rendering objects such as **SpriteBatch** and **BitmapFont** (omitted in this diagram).

    - LibGDX now starts to repeatedly (loop until not ingame) call *render()* on the current screen: `gameScreen`. This call is again forwarded to all of the used GUI elements. The GUI elements each internally use a **GameState** to draw on the screen (see the [**GameState**]’s associations). Furthermore, a *renderCallBack()* method is executed for screen-specific execution.
- The **GameOverScreen** initialization is very similar to the **GameplayScreen** initialization with the following exceptions:
    - Instead of the `breachGame` object, the `gameScreen` object initializes the next screen: `gameOverScreen`.
    - The `gameOverScreen` only initialized the following GUI elements: **SequenceGUI**, **BufferGUI**, and **ScoreGUI**.
    - After setting the screen to `gameOverScreen`, the `gameScreen` is terminated by LibGDX.
    - The loop renders while the endscreen needs to be shown.

User Input Handling Sequence

Link to image



The sequence diagram above depicts the interaction between the player and 2 integral parts of our system (i.e., LibGDX’s InputProcessor and the **CommandFactory** class) following the initialization of the game. During the game, the user input handling sequence must run in a loop where the player sends a user input by pressing a key and the game handles the user input depending on which key the player pressed. This input-process-and-output loop must iterate at least once and should not end until the timer triggers a timeout or the player presses a key that terminates the game or advances to the next screen.

Specifically, once the game is correctly loaded and initialized, we assume that the relevant components of the game such as the **BreachGame** and **GameplayScreen** classes already exist. Furthermore, in this diagram we omit the rendering sequence such as to emphasise the user input handling sequence. The class that handles the player’s input is **InputProcessor**, initialized by the class **GameScreen** using the abstract method *getInputProcessor()* implemented by **GameplayScreen** and **GameOverScreen**.

Once the **InputProcessor** detects a keypress, it asks the **CommandFactory** for a **Command** to execute. If the **CommandFactory** recognises the key, it returns the **Command** related to that key and the **InputProcessor** invokes the **Command**’s *tryExecute()* method; elsewise, the input is ignored and no **Command** is executed. Note that if the key pressed is a meta-command (i.e., “Escape”), libGDX terminates the application (through the loop condition).

Below is a more detailed description of each of the **Command** subtypes:

- If the pressed key is “Up/Down/Left/Right”, the **Move** command is executed, where in case of a valid move, the selector is advanced, and in case of an invalid move, the input is ignored.
- If the pressed key is “Equals” or “Minus”, the **IncreaseBufferLength** or **DecreaseBufferLength** (respectively) command is executed. Both commands check if the maximum buffer offset has been reached and whether the timer has already started, where if both return false, the buffer length is updated.
- If the pressed key is “Enter”, the **ConfirmBuffer** command is executed, which stops the timer. Note that if the timer has already stopped, the timer will simply ignore the request. This relates to confirming the buffer since the timer state is considered inside the `gameState`.*isGameOver()* in the loop condition. When the game is over, the buffer is frozen and thus confirmed.
- If the pressed key is “Backspace”', the **Undo** command is executed where if the `commandHistory` is nonempty, the top **UndoRedoCommand** is popped and its *tryUndo()* method is called. Since the popped command’s specific type is not determined by the **Undo** command, it is difficult to model their interactions and thus we consider the *tryUndo()* call a lost message.
- If the pressed key is “R”, the **Redo** command is executed where if the `commandFuture` is nonempty, the top **UndoRedoCommand** is popped and its *tryRedo()* method is called. This method call is also considered lost.
- If the pressed key is “Spacebar”, the **Select** command is executed where first the `commandFuture` is cleared. The **Select** command begins by starting the timer if it has not started yet and then if the buffer is not full, the new selector position is saved. Finally the command pushes itself onto the `commandHistory` stack. Since the lifetime of this Select is now controlled by the `commandHistory` stack, the object is not immediately destroyed and the push interaction is shown as a lost message. In practice, the object is destroyed when it is inside the `commandFuture` when it is cleared, but modelling this would greatly reduce the understandability of the diagram, if at all possible.

A detailed explanation of the handling of the player’s meta-commands is as follows:

- If the pressed key is “Escape”, the *exit()* operation of the **Application** class in the LibGDX library is immediately invoked and the application is terminated.

The descriptions above assume that the player sends a meaningful input to our system which will then trigger the handling sequence. The **InputAdapter** class ensures that in case of other keypresses from the player, the user input will be ignored and the game’s state will stay consistent across the timeline.

Finally, a note about the *tryExecute()* method call, which may seem meaningless in this diagram. It is mainly used inside **UndoRedoCommand**s where the `commandFuture` stack is cleared before execution and only if the command successfully executes (e.g. **Select** command: a cell has been successfully selected), the command gets added to the `commandHistory` stack. It is used in all **Commands** to provide uniform access to execution.

Implementation

Author(s): Tibi Iancu, Yingdi Xie

Demo: https://youtu.be/Cz1b49fj-sA

UML to Implementation Strategy

We continued to follow the “Agile Principle” in our design process where the system evolved iteratively and went through testing and modification at every stage of the development. Specifically, the class diagram served as the starting point of reflection on the design and implementation. New classes were created and added to the diagram; several classes were redesigned and refactored. The state machine diagrams helped us to think through the state transitions of the largely-changed class **GameState**. The sequence diagrams guided us to reason about the interactions between different classes during the game. Thus, the diagrams allow us to reason and check the correctness of the process before the next iteration of the implementation. The team continued to split the workload evenly and stay engaged in the weekly meetings where we discussed the progress, modified the diagrams when necessary and refactored the code based on new diagrams.

Key Solutions for Implementation

- **Selector movement**: the selector is an instance of the class **Cell**. Moving the selector in a direction can be easily done through the function *applyDir()* from the **Cell** class, which returns a new **Cell** object, which is called from the **Move** command.
- **Undo/Redo**: as explained in the **UndoRedoCommand** section, undoing/redoing an action is done through the *tryUndo()* and *tryRedo()* functions inside **UndoRedoCommand**. When executing a command, it is pushed in the `commandHistory` stack, so that undoing can be performed by simply popping the top of the stack and calling the *tryUndo()* method. Similarly, *tryUndo()* pushes the undone command into the `commandFuture` stack, so redoing an action can be done by popping the top of the stack and executing the command. Note: when executing a **Select** command, `commandFuture` is cleared, since redo only makes sense when the previous move was an undo.
- **GUI**: The GUI is rendered from the **ScreenAdapter** *render()* function that **GameplayScreen** and **GameEndScreen** implement, which individually calls the *render()* function of each **GUIElement** that has to be rendered.
- **Timer**: when the timer is started, the **Timer** object remembers the start time. Subsequent calls to the *timeLeft()* method can be satisfied using the formula `time_left = start_time + total_time - current_time`, where the current time is obtained through a call to *System.currentTimeMillis()*. *hasStarted()* can then be implemented by checking if `startTime` is greater than 0 (i.e. the *start()* method had been previously called). We have also implemented timer stopping functionality through the *stop()* method. When it is called, `stopTime` is assigned the current time. In **GameOverScreen** we can calculate the score based on `stopTime - startTime.`
- **Buffer**: Internally, the buffer class doesn't store the actual buffer contents, rather it uses `commandHistory`, which stores **Select** commands to extrapolate what it should contain.
- **Sequence**: We store the sequences that the user has to complete as a list of **Sequence**, based on which we also calculate the score and render the text on screen.

Main Execution

The execution of the program starts in the class **main.java.Launcher**, where the *main()* method is called. It takes the level file name as argument and creates a **BreachGame** object using this file name.

Jar Location

The JAR file is outputted by Gradle in `/out/artifacts/software_design_vu_2020_jar` and is called `software-design-vu-2020.jar`.

Time logs

We haven't kept time logs, but everyone worked on the assignment.