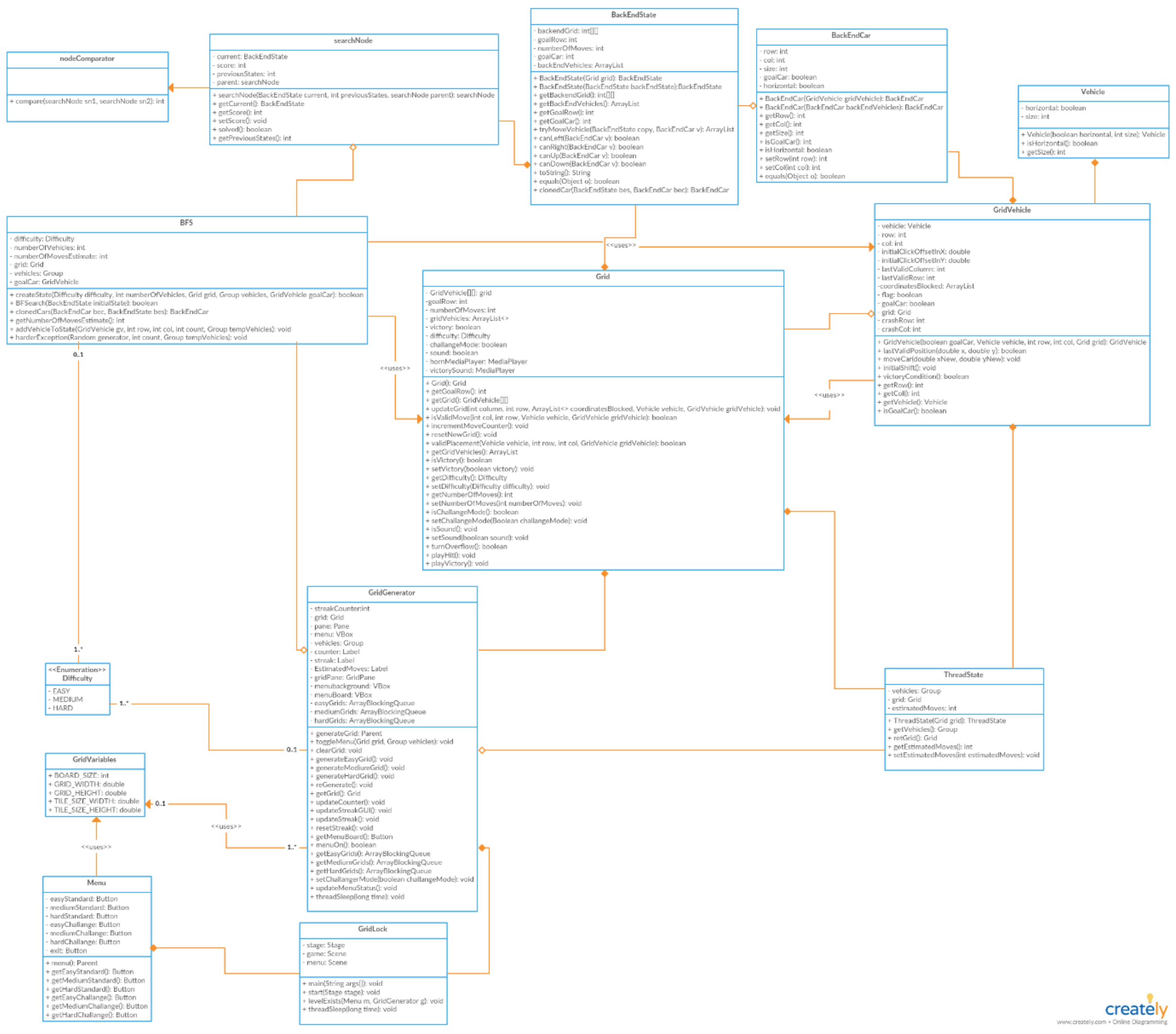
Table of Contents

UML Class Diagram

<u>RESOLUTION IS 2200x2000 SO PLEASE ZOOM IN FOR FURTHER DETAIL OR ACCESS IMAGE AND ZOOM INTO TO SEE CERTAIN SPECTS</u>

The uml design was consistently changed as it had gone along, the main design focus was to have the gridlock class be the bootstrapped class for main. The class will only depend on two different scenes to create the game for the user. The menu scene is in control of the main menu with the tutorial and access to the main game, and the Grid Generator is used to load the game board scene AND generate boards concurrently while the user plays! Smooth gameplay from this. Circular design was avoided, every class has its own unique responsibility, however some classes must be coupled such as grid and grid vehicle as this allows for less responsibility on the grid class however, the grid vehicle when placed on the board needs to update the board itself and when moving it needs to check the board. This will be for both front and back end mixed in to these classes as both rely on each other. The back-end state class will only be used for search algorithm. To avoid using huge amounts of memory by cloning grid vehicles and grids, a more primitive class is used which only contains the information needed to solve the board by machine. The thread state class will be used since the states need to be stored in a class while the game is running.


nodeComparator
+ compare(searchNode sn1, searchNode sn2): int
searchNode
- current: BackEndState
- score: int
- previousStates: int
- parent: searchNode
+ searchNode(BackEndState current, int previousStates, searchNode parent): searchNode
+ getCurrent(): BackEndState
+ getScore(): int
+ setScore(): void
+ solved(): boolean
+ getPreviousStates(): int
BackEndState
- backendGrid: int[][]
- goalRow: int
- numberOfMoves: int
- goalCar: int
- backEndVehicles: ArrayList
+ BackEndState(Grid grid): BackEndState
+ BackEndState(BackEndState backEndState):BackEndState
+ getBackendGrid(): int[][]
+ getBackEndVehicles(): ArrayList
+ getGoalRow(): int
+ getGoalCar(): int
+ tryMoveVehicle(BackEndState copy, BackEndCar v): ArrayList
+ canLeft(BackEndCar v): boolean
+ canRight(BackEndCar v): boolean
+ canUp(BackEndCar v): boolean
+ canDown(BackEndCar v): boolean
+ toString(): String
+ equals(Object o): boolean
+ clonedCar(BackEndState bes, BackEndCar bec): BackEndCar
BackEndCar
- row: int
- col: int
- size: int
- goalCar: boolean
- horizontal: boolean
+ BackEndCar(GridVehicle gridVehicle): BackEndCar
+ BackEndCar(BackEndCar backEndVehicles): BackEndCar
+ getRow(): int
+ getCol(): int
+ getSize(): int
+ isGoalCar(): int
+ isHorizontal: boolean
+ setRow(int row): int
+ setCol(int col): int
+ equals(Object o): boolean
Vehicle
- horizontal: boolean
- size: int
+ Vehicle(boolean horizontal, int size): Vehicle
+ isHorizontal(): boolean
+ getSize(): int
BFS
- difficulty: Difficulty
- numberOfVehicles: int
- numberOfMovesEstimate: int
- grid: Grid
- vehicles: Group
- goalCar: GridVehicle
+ createState(Difficulty difficulty, int numberOfVehicles, Grid grid, Group vehicles, GridVehicle goalCar): boolean
+ BFSearch(BackEndState initialState): boolean
+ clonedCars(BackEndCar bec, BackEndState bes): BackEndCar
+ getNumberOfMovesEstimate(): int
+ addVehicleToState(GridVehicle gv, int row, int col, int count, Group tempVehicles): void
+ harderException(Random generator, int count, Group tempVehicles): void
<<uses>>
0..1
GridVehicle
- vehicle: Vehicle
- row: int
- col: int
- initialClickOffsetInX: double
- initialClickOffsetInY: double
- lastValidColumn: int
- lastValidRow: int
-coordinatesBlocked: ArrayList
- flag: boolean
- goalCar: boolean
- grid: Grid
- crashRow: int
- crashCol: int
+ GridVehicle(boolean goalCar, Vehicle vehicle, int row, int col, Grid grid): GridVehicle
+ lastValidPosition(double x, double y): boolean
+ moveCar(double xNew, double yNew): void
+ initialShift(): void
+ victoryCondition(): boolean
+ getRow(): int
+ getCol(): int
+ getVehicle(): Vehicle
+ isGoalCar(): boolean
Grid
- GridVehicle[][]: grid
-goalRow: int
- numberOfMoves: int
- gridVehicles: ArrayList<>
- victory: boolean
- difficulty: Difficulty
- challangeMode: boolean
- sound: boolean
- hornMediaPlayer: MediaPlayer
- victorySound: MediaPlayer
+ Grid(): Grid
+ getGoalRow(): int
+ getGrid(): GridVehicle[][]
+ updateGrid(int column, int row, ArrayList<> coordinatesBlocked, Vehicle vehicle, GridVehicle gridVehicle): void
+ isValidMove(int col, int row, Vehicle vehicle, GridVehicle gridVehicle): boolean
+ incrementMoveCounter(): void
+ resetNewGrid(): void
+ validPlacement(Vehicle vehicle, int row, int col, GridVehicle gridVehicle): boolean
+ getGridVehicles(): ArrayList
+ isVictory(): boolean
+ setVictory(boolean victory): void
+ getDifficulty(): Difficulty
+ setDifficulty(Difficulty difficulty): void
+ getNumberOfMoves(): int
+ setNumberOfMoves(int numberOfMoves): void
+ isChallangeMode(): boolean
+ setChallangeMode(Boolean challangeMode): void
+ isSound(): void
+ setSound(boolean sound): void
+ turnOverflow(): boolean
+ playHit(): void
+ playVictory(): void
<<uses>>
<<uses>>
1..*
<<Enumeration>>
Difficulty
- EASY
- MEDIUM
- HARD
1..*
0..1
GridGenerator
- streakCounter:int
- grid: Grid
- pane: Pane
- menu: VBox
- vehicles: Group
- counter: Label
- streak: Label
- EstimatedMoves: Label
- gridPane: GridPane
- menubackground: VBox
- menuBoard: VBox
- easyGrids: ArrayBlockingQueue
- mediumGrids: ArrayBlockingQueue
- hardGrids: ArrayBlockingQueue
+ generateGrid: Parent
+ toggleMenu(Grid grid, Group vehicles): void
+ clearGrid: void
+ generateEasyGrid(): void
+ generateMediumGrid(): void
+ generateHardGrid(): void
+ reGenerate(): void
+ getGrid(): Grid
+ updateCounter(): void
+ updateStreakGUI(): void
+ updateStreak(): void
+ resetStreak(): void
+ getMenuBoard(): Button
+ menuOn(): boolean
+ getEasyGrids(): ArrayBlockingQueue
+ getMediumGrids(): ArrayBlockingQueue
+ getHardGrids(): ArrayBlockingQueue
+ setChallangerMode(boolean challangeMode): void
+ updateMenuStatus(): void
+ threadSleep(long time): void
ThreadState
- vehicles: Group
- grid: Grid
- estimatedMoves: int
+ ThreadState(Grid grid): ThreadState
+ getVehicles(): Group
+ retGrid(): Grid
+ getEstimatedMoves(): int
+ setEstimatedMoves(int estimatedMoves): void
GridVariables
+ BOARD_SIZE: int
+ GRID_WIDTH: double
+ GRID_HEIGHT: double
+ TILE_SIZE_WIDTH: double
+ TILE_SIZE_HEIGHT: double
0..1
<<uses>>
1..*
<<uses>>
Menu
- easyStandard: Button
- mediumStandard: Button
- hardStandard: Button
- easyChallange: Button
- mediumChallange: Button
- hardChallange: Button
- exit: Button
+ menu(): Parent
+ getEasyStandard(): Button
+ getMediumStandard(): Button
+ getHardStandard(): Button
+ getEasyChallange(): Button
+ getMediumChallange(): Button
+ getHardChallange(): Button
GridLock
- stage: Stage
- game: Scene
- menu: Scene
+ main(String args[]): void
+ start(Stage stage): void
+ levelExists(Menu m, GridGenerator g): void
+ threadSleep(long time): void
creately
www.creately.com • Online Diagramming

CRC Cards

BackEndCar

This class will be the minimalistic and primitive representative of vehicles on the board, used to solve the board more efficiently, it stores information on the attributes:

1. The vehicle size
2. The vehicle orientation
3. The goal Car flag
4. The vehicle row
5. The vehicle column

Responsibility	Collaborators
- Storing the vehicle details to be used for search from the GridVehicle class - Allowing quick access to the board state vehicles details	- BackEndState - BFS - GridVehicle

BackEndState

This class will represent the board in a minimalistic way for the search. This will allow for a faster search as cloning the primitive board and accessing integers is much faster than for objects and stack panes.

Responsibility	Collaborators
- Storing the 2d matrix representing the grid in a state of 2d integer array. - Stores list of vehicles associated with the board	- BackEndCar - BFS - Grid - GridVehicle

BFS

This class will be used to generate Boards by randomising board states and attempting to solve that state in a more primitive form. It will generate the boards based on difficulty passed in and check if it is solvable.

Responsibility	Collaborators
- This class will be used to generate a state until one is created - It will then attempt to solve the state and return true/false if its solvable	- BackEndState - BackEndCar - GridGenerator - Grid - GridVehicle - Difficulty

Grid

This class will be the representation of the board state for both front and back end, it will contain the list of objects on the 2d grid of cars, and the list of cars available on the board.

Responsibility	Collaborators
- Check if a vehicle is able to move into a slot - Create the grid state initially for user - Control vehicle movement, restriction on vehicle movements - Responsible for enabling sound/game mode (challenge mode)	- GridVehicle - GridGenerator - BackEndState - GridVariables

GridGenerator

This class will be responsible for creating board states and storing them into the queue concurrently, so it can constantly load in boards for the user while user is playing. It will create boards of scaling difficulty and update the game GUI.

Responsibility	Collaborators
- Updating board counters - Loading the menu on request - Creating boards for the user to play on - Loading boards for user on victory condition	- Grid - BackEndState - BFS - ThreadState - GridLock - GridVariables - Difficulty

GridLock

This will be the main class for the game, it will be responsible for loading the GUI scenes, for both the menu and the game scene.

Responsibility	Collaborators
- Loading the main menu - Loading the game state on switch	- GridGenerator - Menu - GridVariables

GridVehicle

This class will be the vehicle displayed on the board which can be moved

Responsibility	Collaborators
- Moving the vehicle on the screen on user drag - Updating the board state on vehicle movement	- Grid - BackEndCar - BFS - GridGenerator - GridVariables

Menu

This class will be scene for the main menu, has buttons to switch the scene to the game scene or exit program

Responsibility	Collaborators
- Displaying tutorial for user - Main access to the game scene - Allow exit from program with exit button	- GridLock - GridGenerator - GridVariables

searchNode

This class will be the main class used in the breadth first search, it will contain the state and a link to the previous state to allow for a path for a solution to be found, and the size of the solution (estimated moves!). it will be responsible for solving a board state and checking if it is solvable

Responsibility	Collaborators
- Creating paths for the search showing how game state changed from its parent - Responsible for creating multiple minimalistic states to check paths	- BackEndState - BFS - NodeComparator

ThreadState

This class will be used to store a state that is able to be instantly loaded from a queue and initialise a new state.

Responsibility	Collaborators
- Stores the Grid and list of vehicles in that state - Allows instant loading (no loading screen) for user, provided there is a threadstate in the queue in the grid generator	- Grid - GridVehicle - GridGenerator - GridLock

Vehicle

This class will be used for storing the vital information for the vehicle. It will store the size, and orientation of the vehicle

Responsibility	Collaborators
- Holds the information for how a car will be represented and allows access to this information	- GridVehicle

GridVariables

This class will be an empty class used to hold all the information for display, in terms of resolution, board size (nxn). These variables will be able to be accessed from anywhere effectively meaning if user wants to change resolution only has to change 1 variable rather than change it in all 5 uses.

Responsibility	Collaborators
- Holds the information for global access to classes that all use the same information - Avoids need for multiple re-declaration of vehicles	- GridLock - Menu - GridGenerator - GridVehicle

NodeComparator

This class will be used for comparing two different states based on the heuristic score of the state. This score will be set in the searchNode class.

Responsibility	Collaborators
- Will re-order the queue based on the node with the lowest score	- searchNode

Difficulty

Responsibility	Collaborators
- sets the difficulty for the state - will be used as a threshold system for making boards based on difficulty	- Grid - GridGenerator - BFS

WalkThrough

User starts game

1. User clicks on the run file
2. A game state loads at the main menu allowing user to select game mode and difficulty
3. Boards will be constantly infinitely generated until queue is full and locks up, and will generate once the queue unlocks etc. this is done through multi-threading
4. Boards will be continuously generated once the program is created until it is full (lock unlock etc), this will even occur when a board is generated on main thread (this occurs when there are no boards in queue since we want ALWAYS to guarantee user will receive a state to play whenever level loaded).

User selects a game mode

1. User selects a game difficulty
2. The scene is switched to the game scene
3. The background is always there, a state is popped off the queue and loaded for the game
4. User is given a game state to play

User presses esc to load menu

1. User presses esc key
2. The scene has the menu turned on/off based on if its there
3. A translucent background is place on top of the game to indicate pause
4. The scene will either add/remove the menu box from the scene whenever esc key is pressed (toggle)

User attempts to drag vehicle

1. User clicks on a vehicle
2. A red border is placed around to indicate the vehicle current selected
3. User is able to move vehicle in the direction of its orientation
4. If user moves vehicle into another vehicle, it will not allow for the vehicle to move, user has to release mouse to snap back to last valid row/column OR the user can choice to move the vehicle in the opposite direction of the collision provided it is a valid move
5. If sound is enabled and crash occurs it will play a small thud sound to indicate invalid move
6. User can toggle the sound through the menu
7. See below for victory

User Victory

1. User drags the goal greenish car into the green victory tile
2. The program will attempt 1 of two things
3. Attempt to pop a state off the queue of SAME difficulty for user to play
4. If there is no state it will create one which will cause ui lag till it is loaded but it will also generate boards constantly on other threads so it will be unlikely for multiple lag
5. User is given continuous gameplay

User decides to exit game

1. User is either on main menu or has pressed esc to open the menu
2. User presses exit button
3. Program exits

User decides to mute the sounds or unmute sounds

1. User presses esc key to access the menu
2. User presses mute button
3. The audio flag is set off/on and a border is placed around button or removed from the button to indicate muted/unmuted (red border == muted)

User decides to switch game modes

1. User presses esc to access menu
2. User presses main menu button
3. Main menu is loaded with the two game modes (3 difficulties standard/challenge)
4. User can select either

Design Summary

Multi-Threading

To allow for almost no delay between loading levels, multi-threading is used to generated boards constantly even while the user is playing. The levels generated are stored onto a queue until the queue has reached maximum capacity and automatically locks, and only unlocks as soon as the queue is no longer at maximum capacity. This is done by having all threads execute board generation with a sleep in a while(true) loop. The sleep is kept in place to make sure the CPU usage is not too difficult.

Challenge mode

Challenge mode is a game mode designed to be punishing for players who make incorrect moves. The user has a limited number of moves determined from the search algorithm to solve the board. Since the search algorithm finds states in SINGLE moves only, i.e each child state only changes a block by 1 tile, it will always be over the minimum number of FULL moves to solve the board. However, since it is a breadth first search with a admissible heuristic, it will be the minimum number of SINGLE moves to solve, however since user is allowed to perform full moves, it will be a bit higher than the minimum number of moves to solve. This means that user can make a few mistakes but not too many. Once the user runs out of moves, it will reset the streak since user had failed, and load a new board from the same difficulty. The incentive of challenge mode is to reach highest streak possible but as it goes on it becomes riskier as too many false moves could reset entire streak, causes more cautious analysis later on to maintain the streak.

The board generation algorithm

The board generation algorithm is done in a way to allow for random boards to be generated that are guaranteed to be solvable. A random state is created by placing vehicles on the board until a configuration has been made, this will be when the number of vehicles for the state randomly chosen between 4-20 is on the board. It will constantly attempt to create this state. The BFS class will then attempt to perform the Breadth First Search to solve the board, if the board has no solution it will redo the entire process, this ensures that at the end of the generate(difficulty)Board function it will return a state of the difficulty chosen. The difficulty is set based on a threshold, the number of moves from the path returned from search will be the threshold, it will check how many moves are performed and see if its within the difficulty threshold. Different states are created by trying to move the vehicle in its orientation in all directions possible creating a state on any move able to be performed.