Into The Breach (ish)

Assignment 2
Semester 1, 2024
CSSE1001

Due date: 24 May 2024, 16:00 GMT+10

1 Introduction

In this assignment, you will implement a (heavily) simplified version of the video game "Into The Breach". In this game players defend a set of civilian buildings from giant monsters. In order to achieve this goal, the player commands a set of equally giant mechanical heroes called "Mechs". There are a variety of enemy and mech types, which each behave slightly differently. Gameplay is described in section 3 of this document.

Unlike assignment 1, in this assignment you will be using object-oriented programming and following the Apply Model-View-Controller design pattern shown in lectures. In addition to creating code for modelling the game, you will be implementing a graphical user interface (GUI). An example of a final completed game is shown in Figure 1.

2 Getting Started

Download `a2.zip` from Blackboard — this archive contains the necessary files to start this assignment. Once extracted, the `a2.zip` archive will provide the following files:

`a2.py` *This is the only file you will submit* and is where you write your code. *Do not* make changes to any other files.

`a2_support.py` *Do not modify or submit this file*, it contains pre-defined classes, functions, and constants to assist you in some parts of your assignment. In addition to these, you are encouraged to create your own constants and helper functions in `a2.py` where possible.

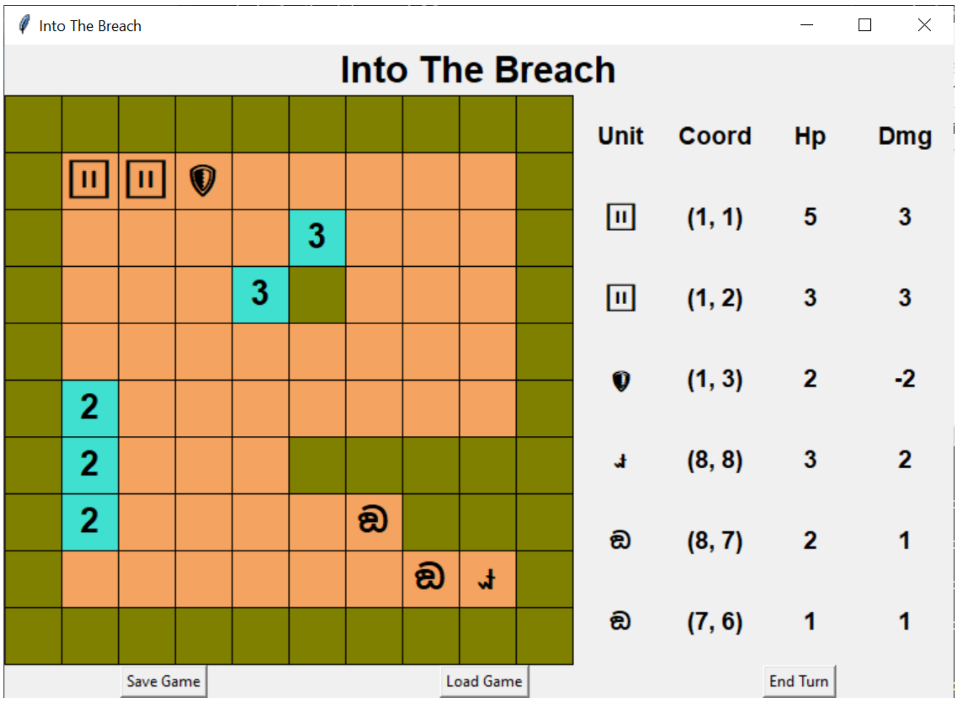
`levels/` This folder contains a small collection of files used to initialize games of *Into The Breach*. In addition to these, you are encouraged to create your own files to help test your implementation where possible.

3 Gameplay

This section describes an overview of gameplay for Assignment 2. Where interactions are not explicitly mentioned in this section, please see Section 4.

3.1 Definitions

Gameplay takes place on a rectangular grid of *tiles* called a *board*, on which different types of *entities* can stand. There are three types of tile: *Ground* tiles, *mountain* tiles, and *building* tiles. Building



Unit	Coord	Hp	Dmg
⏸	(1, 1)	5	3
⏸	(1, 2)	3	3
🛡	(1, 3)	2	-2
ⅎ	(8, 8)	3	2
ඞ	(8, 7)	2	1
ඞ	(7, 6)	1	1

Figure 1: Example screenshot from a completed implementation. Note that your display may look slightly different depending on your operating system.

tiles each possess a given amount of *health*, which is the amount of damage they can suffer before they are *destroyed*. A building is destroyed if its health drops to 0. A tile may be *blocking*, in which case entities cannot stand on it. Tiles that are not blocking may have a maximum of one entity standing on them at any given time. Ground tiles are never blocking, mountain tiles are always blocking, and building tiles are blocking if and only if they are not destroyed.

Entities may either be *Mechs*, which are controlled by the player, or *Enemies*, which attack the player's mechs and buildings. There are two types of mech; the *Tank Mech* and the *Heal Mech*. There are also two types of enemy; the *Scorpion* and the *Firefly*. Each entity possesses 4 characteristics:

1. *position*: the coordinate of the tile within the board on which the entity is currently standing.
2. *health*: the remaining amount of damage the entity can suffer before it is *destroyed*. An entity is destroyed the moment its health drops to 0, at which point it is immediately removed from the game.
3. *speed*: the number of tiles the entity can move during its movement phase (see below for details). Entities can only move horizontally and vertically; that is, moving one tile diagonally is considered two individual movements.
4. *strength*: how much *damage* the entity deals to buildings and other entities (i.e. the amount by which it reduces the health of attacked buildings or entities).

The game is turn based, with each turn consisting of a player movement phase, an attack phase, and an enemy movement phase. During the player movement phase, the player has the option to move each of the mechs under their control to a new tile on the grid. During the attacking phase, each mech and enemy perform an attack: an action that can damage mechs, enemies, or even buildings. Each enemy, mech, and building can only receive a certain amount of damage. If a mech or enemy is destroyed before they attack during a given attack phase, they do not attack during that attack phase. During the enemy movement phase, each enemy chooses a tile as their objective, and then moves to a new tile on the grid such that they are closer to their objective. The order in which

mechs and enemies move and attack is determined by a fixed priority that will be displayed to the user at all times.
A *valid path* within the board is a sequence of movements into vertically or horizontally adjacent non-blocking tiles which do not contain an entity. The length of a valid path is the number of movements made within it. Note that each entity can only move through *valid paths* of length less than or equal to their maximum path length (speed).
A game of Into The Breach is over when either:

1. The player *wins* because at the end of an attack phase, all enemies are destroyed, at least one mech is not destroyed, and at least one building on the board is not destroyed.

2. The player *loses* because at the end of an attack phase, all buildings on the board are destroyed, or all mechs are destroyed.

3.2 Game phases

The game begins with a board of tiles, with entities occupying non-blocking tiles (at least one mech and at least one enemy). The exact set of tiles and entities is given by the level file used to initialise the game. Next to the board of tiles, a list is presented. Each element of the list displays an entity, alongside its position, current health, and current strength. The list is ordered by entity priority, with the highest priority entity appearing at the top (see Figure 1 for an example).
The following four phases repeat until the end of the game:

1. Player movement phase: This is the main phase of the game where all user interaction occurs. The user may click on any tile on the board. The action taken after a tile is clicked is summarized in Table 1. See Figure 2 for an example of the movement system. During the player movement phase, the user may also click one of the three buttons:

   - If the user clicks the *Save* button, they should be prompted to enter a name for their save file via a filedialog. Upon entering a name and clicking to save the file, a new level file should be created based on the current game state. If a mech has been moved before the save button is clicked, the user is warned instead via an error message box.
   - If the user clicks the *Load* button they should be prompted to select a saved file with a filedialog. When they select a file gameplay should restart as if the selected level file was the file used to initialise the game.
   - If the user clicks the *End Turn* button, the current player movement phase is ended, and the program moves onto the attack phase.

2. Attack phase: During the attack phase each entity, in descending order of priority, makes an *attack*. An attack affects a certain set of tiles depending on the entity making it. See Table 2 for the tiles affected by each entity. If a building tile is affected by an attack, then that building loses health equal to the strength of the attacking entity. If an entity is on a tile affected by an attack, then that entity is affected in a manner depending on what entity is performing the attack. See Table 2 for the effects of attacks for each entity. If an entity is destroyed during the attack phase by an entity with higher priority, it does not attack and is removed from the game. After each entity has performed an attack, the program immediately moves to the enemy movement phase.

3. Enemy movement phase: During the enemy movement phase, all enemies are assigned an *objective*. An objective is the position of a tile on the board and is assigned based on the type of entity as described in Table 3. Each enemy, in descending priority order, then moves to the tile that minimizes the length of the shortest path from itself to it's objective. Note that the enemy can only move to tiles reachable via valid paths of length no greater than it's speed. If there exists no valid path from an enemy to its objective, the enemy does not change position.

After every enemy has moved, the display is updated and the program moves to termination checking.

4. Termination checking: If all enemies are destroyed, at least one mech is not destroyed, and at least one building on the board is not destroyed, the user has won and a victory message is displayed via an info messagebox. If all buildings on the board are destroyed or all mechs are destroyed, the user has lost and a defeat message is displayed via an info messagebox. Both victory and defeat messageboxes ask the user if they wish to play again. If the user does want to play again, then the game is reinitialised using the level file and gameplay starts again from the beginning. If the user does not want to play again the program closes the game window and exits gracefully. If no messageboxes were displayed then the program immediately returns to the player movement phase.

Clicked Tile	**Action to take**
Tile containing a mech that has not moved during the current movement phase	Tiles which the mech can move to are highlighted in green. Valid tiles are those to which a valid path can be formed from the mech's position with length less than or equal to the mech's speed.
Tile highlighted by clicking a tile containing a mech that has not moved during the current movement phase	The relevant mech is moved to that tile.
Tile containing an enemy, or Tile containing a mech that has moved during the current player movement phase	Tiles which will be attacked by that entity during the following attack phase are highlighted in red.
Any other tile.	Nothing.

Table 1: Effect of clicking tiles during player movment phase. Every time the user clicks a tile, all previous highlighting is removed.

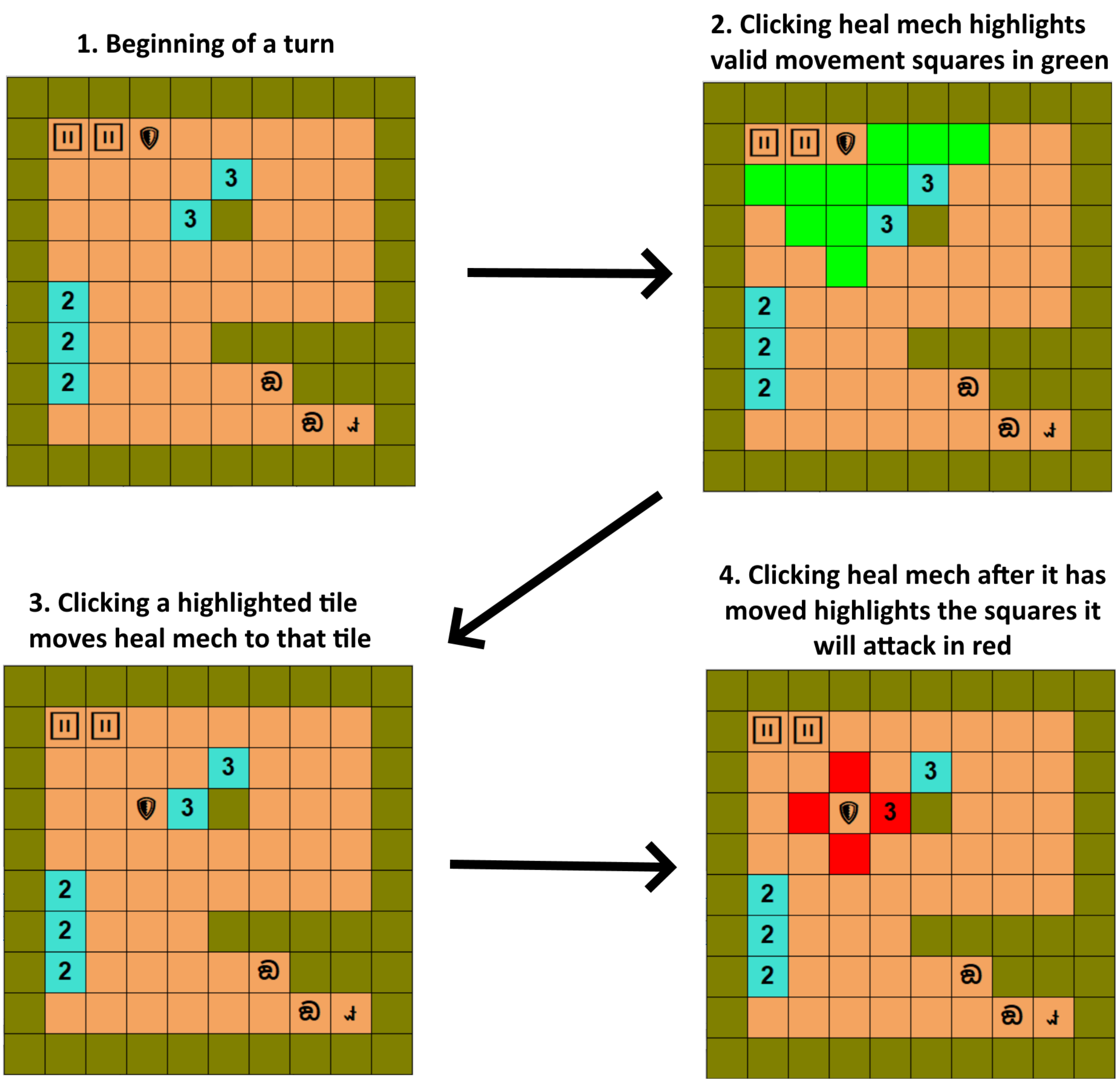

1. Beginning of a turn
2. Clicking heal mech highlights valid movement squares in green
3. Clicking a highlighted tile moves heal mech to that tile
4. Clicking heal mech after it has moved highlights the squares it will attack in red
3
3
2
2
2

Figure 2: Movement of a mech during the player movement phase. The user clicks on the Heal Mech, and then clicks on one of the highlighted squares. Clicking the heal mech again highlights the squares it will attack.

Entity	**Tiles Affected**	**Attack Effect**
Tank Mech	The two sets of five tiles extending in a horizontal line from the tank mech: beginning from the tile directly left of the tank mech and extending left, and beginning from the tile directly right of the tank mech and extending right respectively.	Receive damage equal to strength of tank mech.
Heal Mech	The four tiles directly adjacent to heal mech (not including diagonals)	If target is a mech, recover health equal to strength of heal mech. Do nothing otherwise.
Scorpion	The four sets of two tiles extending in horizontal and vertical lines from the scorpion: beginning from the tile directly left of the scorpion and extending left, beginning from the tile directly right of the scorpion and extending right, beginning from the tile directly above of the scorpion and extending upward, and beginning from the tile directly below scorpion and extending downwards respectively.	Receive damage equal to strength of scorpion.
Firefly	The two sets of five tiles extending in a vertical line from the firefly: beginning from the tile directly above of the firefly and extending upwards, and beginning from the tile directly below the firefly and extending downwards respectively.	Receive damage equal to strength of firefly.

Table 2: Entity attack behavior

Enemy	**Assigned Objective**
Scorpion	Position of tile containing mech with the greatest health. If two mechs are tied for greatest health, choose position of tile containing the mech with the highest priority.
Firefly	Position of building tile with the least health amongst the buildings that are not destroyed. If two buildings are tied for the least health, choose the position of the building tile in the bottommost row. If there is still a tie for lowest health, choose the position of the building tile in the rightmost column.

Table 3: Enemy objectives

4 Implementation

***NOTE:* You are not permitted to add any additional import statements to `a2.py`. Doing so will result in a deduction of up to 100% of your mark.** You must not modify or remove the import statements already provided to you in `a2.py`. Removing or modifying these existing import statements may result in your code not functioning, and may result in a **deduction of up to 100% of your mark.**

Required Classes and Methods

You will be following the Apple Model-View-Controller design pattern when implementing this assignment, and are *required* to implement a number of classes in order to do so.

The class diagram in Figure 3 provides an overview of *all* of the classes you must implement in your assignment, and the basic relationships between them. The details of these classes and their methods are described in depth in Sections 4.1, 4.2 and 4.3. Within Figure 3:

- Orange classes are those provided to you in the support file, or imported from TkInter.
- Green classes are *abstract* classes. However, you are not required to enforce the abstract nature of the green classes in their implementation. The purpose of this distinction is to indicate to you that *you* should only ever instantiate the blue and orange classes in your program (though you should instantiate the green classes to test them before beginning work on their subclasses).
- Blue classes are *concrete* classes.
- Solid arrows indicate *inheritance* (i.e. the "is-a" relationship).
- Dotted arrows indicate *composition* (i.e. the "has-a" relationship). An arrow marked with 1-1 denotes that each instance of the class at the base of the arrow contains exactly one instance of the class at the head of the arrow. An arrow marked with 1-N denotes that each instance of the class at the base of the arrow may contain many instances of the class at the head of the arrow.

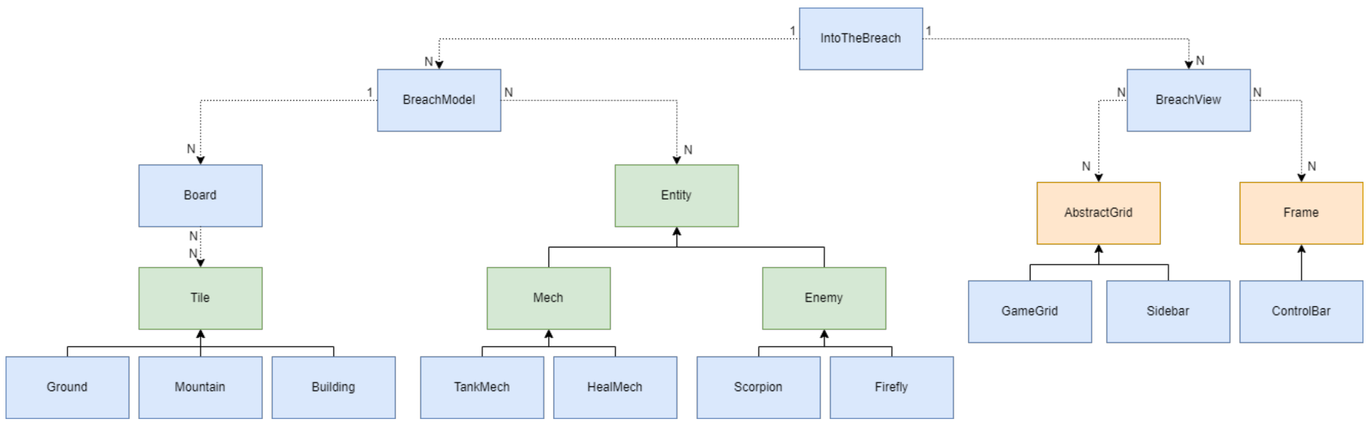


Figure 3: Basic class relationship diagram for the classes in assignment 2.

The rest of this section describes the required implementation in detail. You should complete the model section before attempting the view and controller sections, ensuring that everything you implement is tested thoroughly, operating correctly, and passes all relevant Gradescope tests. You will not be able to earn marks for the controller section until you have passed all Gradescope tests for the model section.
NOTE: It is possible to recieve a passing grade on this assessment by completing section 4.1, providing all hidden tests are passed, and no marks are lost on style (See section 5.2 for more detail on style requirements)

4.1 Model

The following are the classes and methods you are *required* to implement as part of the model. You should develop the classes in the order in which they are described in this section and test each one (including on Gradescope) before moving on to the next class. Functionality marks are awarded for each class (and each method) that *work correctly*. You will likely do very poorly if you submit an attempt at every class, where no classes work according to the description. Some classes require significantly more time to implement than others. The marks allocated to each class are not necessarily an indication of their difficulty or the time required to complete them. You are allowed (and encouraged) to write additional helper methods for any class to help break up long methods, but these helper methods *MUST* be private (i.e. they must be named with a leading underscore).

4.1.1 `Tile()`

`Tile` is an abstract class from which all instantiated types of tile inherit. Provides default tile behavior, which can be inherited or overridden by specific types of tiles. Abstract tiles are represented by the character `T`. The `__init__` method does not take any arguments beyond `self`.

`Tile` should implement the following methods:

- `__repr__(self) -> str`

  Returns a machine readable string that could be used to construct an identical instance of the tile.

- `__str__(self) -> str`

  Returns the character representing the type of the tile.

- `get_tile_name(self) -> str`

  Returns the name of the type of the tile (i.e. the name of the most specific class to which the tile belongs).

- `is_blocking(self) -> bool`

  Returns True only when the tile is blocking. By default tiles are *not* blocking

Examples:

```
>>> tile = Tile()
>>> tile
Tile()
>>> str(tile)
'T'
>>> tile.get_tile_name()
'Tile'
>>> tile.is_blocking()
False
```

4.1.2 `Ground(Tile)`

`Ground` inherits from `Tile`. `Ground` tiles represent simple, walkable ground with no special properties. `Ground` tiles are never blocking and are represented by a space character (' ').

Examples:

```
>>> ground = Ground()
>>> ground
Ground()
>>> str(ground)
' '
>>> ground.get_tile_name()
'Ground'
>>> ground.is_blocking()
False
```

4.1.3 `Mountain(Tile)`

`Mountain` inherits from `Tile`. `Mountain` tiles represent unpassable terrain. `Mountain` tiles are always blocking and are represented by the character `M`.

Examples:

```
>>> mountain = Mountain()
>>> mountain
Mountain()
>>> str(mountain)
'M'
>>> mountain.get_tile_name()
'Mountain'
>>> mountain.is_blocking()
True
```

4.1.4 `Building(Tile)`

`Building` inherits from `Tile`. Building tiles represent one or more buildings that the player must protect from enemies. Building tiles have an integer health value and can be destroyed. A building tile is destroyed when its health drops to zero. The health value of a building can never increase above 9. Building tiles are blocking only when they are not destroyed. Building tiles are represented by their current health value, as a string.

In addition to the `Tile` methods that must be supported, `Building` should additonally implement the following methods:

- `__init__(self, initial_health: int) -> None`

  instantiates a building with the specified health. A precondition to this function is that the specified health will be between 0 and 9 (inclusive).

- `is_destroyed(self) -> bool`

  Returns True only when the building is destroyed.

- `damage(self, damage: int) -> None`

  Reduces the health of the building by the amount specified. Note that `damage` is not constrained to be positive. The health of the building should be capped to be between 0 and 9 (inclusive). This function should do nothing if the building is destroyed.

Examples:

```
>>> building = Building(5)
>>> building
Building(5)
>>> str(building)
'5'
>>> building.is_destroyed()
False
>>> building.is_blocking()
True
>>> building.damage(-10)
>>> str(building)
'9'
>>> building.damage(15)
>>> str(building)
'0'
>>> building.is_destroyed()
True
>>> building.is_blocking()
False
>>> building.damage(-1)
>>> str(building)
'0'
```

4.1.5 `Board()`

`Board` represents a structured set of tiles. A board organizes tiles in a rectangular grid, where each tile has an associated (row, column) position. (0,0) represents the top-left corner, (1,0) represents the position directly below the top-left corner, and (0, 1) represents the position directly right of the top left corner. The methods that must be implemented in `Board` are:

- `__init__(self, board: list[list[str]]) -> None`

  Sets up a new `Board` instance from the information in the `board` argument. Each list in `board` represents a row of the board. The first list represents the top-most row of the board, and the last list represents the bottom-most row of the board. The first character of each inner list represents the left-most tile on that row, and the last character of each inner list represents the right-most tile on that row. Each character should be mapped to the tile that the character represents.

  A precondition to this function is that each list (each row) within the given `board` will have the same length. Another precondition to this function is that the given array will contain at least one row. The final precondition to this function is that each character provided will be the string representation of one of the tile subclasses described in previous sections.

- `__repr__(self) -> str`

  Returns a machine readable string that could be used to construct an identical instance of the board.

- `__str__(self) -> str`

  Returns a string representation of the board. This is the string formed by concatenating the characters representing each tile of a row in the order they appear (left to right), and then concatenating each row in order (from top to bottom), separating each row with a new line character.

- `get_dimensions(self) -> tuple[int, int]`

  Returns the (#rows, #columns) dimensions of the board.

- `get_tile(self, position: tuple[int, int]) -> Tile`

  Returns the Tile instance located at the given position. A precondition to this function is that the provided position will not be out of bounds, that is,
  `(0,0) <= position < self.get_dimensions()`

- `get_buildings(self) -> dict[tuple[int, int], Building]`

  Returns a dictionary mapping the positions of buildings to the building *instances* at those positions. This dictionary should only contain positions at which there is a building tile.

Examples:

```
>>> tiles = [[" ","4"],["6","M"]]
>>> board = Board(tiles)
>>> board
Board([[' ', '4'], ['6', 'M']])
>>> str(board)
' 4\n6M'
>>> board.get_dimensions()
(2, 2)
>>> board.get_tile((0,1))
Building(4)
>>> board.get_buildings()
{(0, 1): Building(4), (1, 0): Building(6)}
```

4.1.6 `Entity()`

`Entity` is an abstract class from which all instantiated types of entity inherit. This class provides default entity behavior, which can be inherited or overridden by specific types of entities. All entities exist at a given (row, column) position, and possess integer health, speed, and strength values. Note: it is not the role of an entity to determine if the position it occupies exists or is valid. Like buildings, entities can be destroyed. An entity is destroyed when its health drops to zero. Entities can be friendly (that is, under player control), or not. Abstract entities are represented by the character `E`. `Entity` should implement the following methods:

-
  ```
  __init__(
  self,
  position: tuple[int, int],
  initial_health: int,
  speed: int,
  strength: int
  ) -> None:
  ```

  Instantiates a new entity with the specified position, health, speed, and strength.

- `__repr__(self) -> str`

  Returns a machine readable string that could be used to construct an identical instance of the entity.

- `__str__(self) -> str`

Returns the string representation of the entity. The string representation of an entity is a comma separated list containing (in order): the character representing the type of the entity; the row currently occupied by the entity; the column currently occupied by the entity; the current health of the entity; the entity's speed; and the entity's strength.

- `get_symbol(self) -> str`

  Returns the character that represents the entity type.

- `get_name(self) -> str`

  Returns the name of the type of the entity (the name of the most specific class to which this entity belongs).

- `get_position(self) -> tuple[int, int]`

  Returns the (row, column) position currently occupied by the entity.

- `set_position(self, position: tuple[int, int]) -> None`

  Moves the entity to the specified position.

- `get_health(self) -> int`

  Returns the current health of the entity

- `get_speed(self) -> int`

  Returns the speed of the entity

- `get_strength(self) -> int`

  Returns the strength of the entity

- `damage(self, damage: int) -> None`

  Reduces the health of the entity by the amount specified. Note that the amount of damage suffered is not constrained to be positive. The health of the entity should be capped to be non-negative. The health of the entity should not be capped to any maximum value. This function should do nothing if the entity is destroyed.

- `is_alive(self) -> bool`

  Returns True if and only if the entity is not destroyed.

- `is_friendly(self) -> bool`

  Returns True if and only if the entity is friendly. By default, entities are *not* friendly

- `get_targets(self) -> list[tuple[int, int]]`

  Returns the positions that would be attacked by the entity during a combat phase. By default, entities target vertically and horizontally adjacent tiles. When overriding `get_targets` in subclasses, see Table 2. Note: The order of elements in this list does not matter.

- `attack(self, entity: "Entity") -> None`

Applies this entity's effect to the given `entity`. By default, entities deal damage equal to the strength of the entity. When overridding the `attack` method in subclasses, refer to Table 2. Note: as the `attack` method is defined as part of the definition of the `Entity` class, the typehint for `entity` will need to be wrapped in double quotes or else python will throw a syntax error. The type of `entity` is still `Entity`.

Examples:

```
>>> e1 = Entity((0,0),1,1,1)
>>> e1
Entity((0, 0), 1, 1, 1)
>>> str(e1)
'E,0,0,1,1,1'
>>> e1.get_symbol()
'E'
>>> e1.get_name()
'Entity'
>>> e1.is_friendly()
False
>>> e1.get_health()
1
>>> e1.get_speed()
1
>>> e1.get_strength()
1
>>> e1.get_position()
(0, 0)
>>> e1.set_position((24,4))
>>> e1.get_position()
(24, 4)
>>> e1.get_targets()
[(24, 5), (24, 3), (25, 4), (23, 4)]
>>> e1.get_health()
1
>>> e1.damage(2)
>>> e1.get_health()
0
>>> e1.is_alive()
False
>>> e1.damage(-4)
>>> e1.get_health()
0
>>> e2 = Entity((1,0),2,1,1)
>>> e2.get_health()
2
>>> e1.attack(e2)
>>> e2.get_health()
1
```

4.1.7 `Mech(Entity)`

`Mech` is an abstract class that inherits from `Entity` from which all instantiated types of mech inherit. This class provides default mech behavior, which can be inherited or overridden by specific types of mechs. All mechs can be *active* (that is, able to be moved by user input), or not. Mechs are always active upon instantiation. Additionally, all mechs also keep track of their *previous* position, that is, the position they were at before the most recent call to `set_position`. Mechs of any type are always friendly. Abstract mechs are represented by the character `M`.

In addition to the `Entity` methods that must be supported, `Mech` should additionally implement the following methods:

- `enable(self) -> None`

  Sets the mech to be active.

- `disable(self) -> None`

  Sets the mech to not be active.

- `is_active(self) -> bool`

  Returns true if and only if the mech is active.

Examples:

```
>>> mech = Mech((0,0),1,1,1)
>>> mech.get_symbol()
'M'
>>> mech.get_name()
'Mech'
>>> mech.is_friendly()
True
>>> mech.is_active()
True
>>> mech.disable()
>>> mech.is_active()
False
>>> mech.enable()
>>> mech.is_active()
True
```

4.1.8 `TankMech(Mech)`

`TankMech` inherits from `Mech`. `TankMech` represents a type of mech that attacks at a long range horizontally. Tank mechs are represented by the character `T`.

Examples:

```
>>> tank = TankMech((0,0),1,1,1)
>>> tank.get_symbol()
'T'
>>> tank.get_name()
'TankMech'
>>> tank.get_targets()
[(0, 1), (0, -1), (0, 2), (0, -2), (0, 3), (0, -3), (0, 4), (0, -4), (0, 5), (0, -5)]
```

4.1.9 `HealMech(Mech)`

`HealMech` inherits from `Mech`. `HealMech` represents a type of mech that does not deal damage, but instead supports friendly units and buildings by *healing* (that is, increasing health); that is, `HealMech` objects ‘damage‘ friendly units and buildings by a *negative* amount. In order to achieve this, the `get_strength` method of the `HealMech` should return a value equal to the *negative* of the heal mech’s strength. A heal mech does nothing when attacking an entity that is not friendly. Heal mechs are represented by the character `H`.

Examples:

```
>>> heal = HealMech((0,0),1,1,2)
>>> heal.get_symbol()
'H'
>>> heal.get_name()
'HealMech'
>>> heal.get_strength()
-2
>>> friendly = TankMech((1,1),1,1,1)
>>> not_friendly = Entity((1,1),1,1,1)
>>> friendly.get_health()
1
>>> heal.attack(friendly)
>>> friendly.get_health()
3
>>> not_friendly.get_health()
1
>>> heal.attack(not_friendly)
>>> not_friendly.get_health()
1
```

4.1.10 `Enemy(Entity)`

`Enemy` is an abstract class that inherits from `Entity` from which all instantiated types of enemy inherit. This class provides default enemy behavior, which can be inherited or overridden by specific types of enemies. All enemies have an objective, which is a position that the entity wants to move towards. The objective of all enemies upon instantiation is the enemy’s current position. Enemies of any type are never friendly. Abstract enemies are represented by the character `N`.

In addition to the `Entity` methods that must be supported, `Enemy` should additionally implement the following methods:

- `get_objective(self) -> tuple[int, int]`

  Returns the current objective of the enemy.

- `update_objective(self, entities: list[Entity], buildings: dict[tuple[int, int], Building]) -> None`

  Updates the objective of the enemy based on a list of entities and dictionary of buildings, according to Table 3. The default behavior (that is, the behavior in the abstract `Enemy` class) is to set the objective of the enemy to the current position of the enemy. If no valid objective exists, then the enemy’s objective should not change.

  A precondition to this function is that the given list of entities is sorted in descending priority order, with the first entity in the list being the highest priority.

Examples:

```
>>> enemy = Enemy((0,0),1,1,1)
>>> enemy.get_symbol()
'N'
>>> enemy.get_name()
'Enemy'
>>> enemy.get_objective()
(0, 0)
>>> enemy.set_position((3,3))
>>> entities = [TankMech((0,1),1,1,1), HealMech((0,2),2,1,1)]
>>> buildings = {(1,0): Building(1), (1,1): Building(2)}
>>> enemy.update_objective(entities, buildings)
>>> enemy.get_objective()
(3, 3)
```

4.1.11 `Scorpion(Enemy)`

`Scorpion` inherits from `Enemy`. `Scorpion` represents a type of enemy that attacks at a moderate range in all directions, and targets mechs with the highest health. Scorpions are represented by the character `S`.
Examples:

```
>>> scorpion = Scorpion((0,0),1,1,1)
>>> scorpion.get_symbol()
'S'
>>> scorpion.get_name()
'Scorpion'
>>> scorpion.get_targets()
[(0, 1), (0, -1), (1, 0), (-1, 0), (0, 2), (0, -2), (2, 0), (-2, 0)]
>>> entities = [TankMech((0,1),1,1,1), HealMech((0,2),2,1,1)]
>>> buildings = {(1,0): Building(1), (1,1): Building(2)}
>>> scorpion.update_objective(entities, buildings)
>>> scorpion.get_objective()
(0, 2)
```

4.1.12 `Firefly(Enemy)`

`Firefly` inherits from `Entity`. `Firefly` represents a type of enemy that attacks at a long range vertically, and targets buildings with the lowest health. Fireflies are represented by the character `F`.
Examples:

```
>>> firefly = Firefly((0,0),1,1,1)
>>> firefly.get_symbol()
'F'
>>> firefly.get_name()
'Firefly'
>>> firefly.get_targets()
[(1, 0), (-1, 0), (2, 0), (-2, 0), (3, 0), (-3, 0), (4, 0), (-4, 0), (5, 0), (-5, 0)]
>>> entities = [TankMech((0,1),1,1,1), HealMech((0,2),2,1,1)]
>>> buildings = {(1,0): Building(1), (1,1): Building(2)}
>>> firefly.update_objective(entities, buildings)
>>> firefly.get_objective()
(1, 0)
```

4.1.13 `BreachModel()`

`BreachModel` models the logical state of a game of *Into The Breach*.

`BreachModel` should implement the following methods:

- `__init__(self, board: Board, entities: list[Entity]) -> None`

  Instantiates a new model class with the given board and entities. A precondition to this function is that the provided list of entities is in descending priority order, with the highest priority entity being the first element of the list, and the lowest priority entity being the last element of the list.

- `__str__(self) -> str`

  Returns the string representation of the model. The string representation of a model is the string representation of the game board, followed by a blank line, followed by the string representation of all game entities in descending priority order, separated by newline characters.

- `get_board(self) -> Board`

  Returns the current board *instance*.

- `get_entities(self) -> list[Entity]`

  Returns the list of all entities in descending priority order, with the highest priority entity being the first element of the list.

- `has_won(self) -> bool`

  Returns True iff the game is in a win state according to the game rules (see section 3).

- `has_lost(self) -> bool`

  Returns True iff the game is in a loss state according to the game rules (see section 3).

- `entity_positions(self) -> dict[tuple[int, int], Entity]`

  Returns a dictionary containing all entities, indexed by entity position.

- `get_valid_movement_positions(self, entity: Entity) -> list[tuple[int, int]]`

  Returns the list of positions that the given entity could move to during the relevant movement phase. Note that this function does not check if the entity has already moved during a given movement phase. The list should be ordered such that positions in higher rows appear before positions in lower rows. Within the same row, positions in columns further left should appear before positions in columns further right. You should make use of `get_distance` from `a2_support.py` when implementing this method.

- `attempt_move(self, entity: Entity, position: tuple[int, int]) -> None`

  Moves the given entity to the specified position only if the entity is friendly, active, and can move to that position according to the game rules (see section 3). Does nothing otherwise. Disables entity if a successful move is made.

- `ready_to_save(self) -> bool`

  Returns true only when no move has been made since the last call to `end_turn`.

- `assign_objectives(self) -> None`

  Updates the objectives of all enemies based on the current game state

- `move_enemies(self) -> None`

  Moves each enemy to the valid movement position that minimizes the distance of the shortest valid path between the position and the enemy's objective. If there is a tie for minimum shortest distance, the enemy moves to the position in the bottom-most row. If there is still a tie for minimum shortest distance, the enemy moves to the position in the rightmost column. If there is no valid path from an enemy to its objective, the enemy does not move. Enemies move in descending priority order starting with the highest priority enemy. You should make use of `get_distance` from `a2_support.py` when implementing this method.

- `make_attack(self, entity: Entity) -> None`

  Makes given entity perform an attack against every tile that is currently a target of the entity. The effect on each tile is described under the attack phase heading in section 3

- `end_turn(self) -> None`

  Executes the attack and enemy movement phases as described in section 3 (ignoring the display update), and then sets all mechs to be active.

Examples:

```
>>> board = Board([['M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M'], ['M', '
 ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'M'], ['M', ' ', ' ', ' ', ' ', '3', '
', ' ', ' ', 'M'], ['M', ' ', ' ', ' ', '3', 'M', ' ', ' ', ' ', 'M'], ['M', '
 ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'M'], ['M', '2', ' ', ' ', ' ', ' ', '
', ' ', ' ', 'M'], ['M', '2', ' ', ' ', ' ', 'M', 'M', 'M', 'M', 'M'], ['M', '
2', ' ', ' ', ' ', ' ', ' ', 'M', 'M', 'M'], ['M', ' ', ' ', ' ', ' ', ' ', '
', ' ', ' ', 'M'], ['M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M']])
>>> entities = [TankMech((1, 1), 5, 3, 3), TankMech((1, 2), 3, 3, 3), HealMech
((1, 3), 2, 3, 2), Scorpion((8, 8), 3, 3, 2), Firefly((8, 7), 2, 2, 1), Firefl
y((7, 6), 1, 1, 1)]
>>> model = BreachModel(board, entities)
>>> str(model)
'MMMMMMMMMM\nM        M\nM    3   M\nM   3M   M\nM        M\nM2       M\nM2   M
MMMM\nM2     MMM\nM        M\nMMMMMMMMMM\n\nT,1,1,5,3,3\nT,1,2,3,3,3\nH,1,3,2,3
,2\nS,8,8,3,3,2\nF,8,7,2,2,1\nF,7,6,1,1,1'
>>> model.get_board()
Board([['M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M'], ['M', ' ', ' ', ' ',
 ' ', ' ', ' ', ' ', ' ', 'M'], ['M', ' ', ' ', ' ', ' ', '3', ' ', ' ', ' ', '
M'], ['M', ' ', ' ', ' ', '3', 'M', ' ', ' ', ' ', 'M'], ['M', ' ', ' ', ' ',
' ', ' ', ' ', ' ', ' ', 'M'], ['M', '2', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '
M'], ['M', '2', ' ', ' ', ' ', 'M', 'M', 'M', 'M', 'M'], ['M', '2', ' ', ' ',
' ', ' ', ' ', 'M', 'M', 'M'], ['M', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '
M'], ['M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M']])
>>> model.get_entities()
[TankMech((1, 1), 5, 3, 3), TankMech((1, 2), 3, 3, 3), HealMech((1, 3), 2, 3, 2
), Scorpion((8, 8), 3, 3, 2), Firefly((8, 7), 2, 2, 1), Firefly((7, 6), 1, 1, 1
)]
>>> model.has_won()
False
```

```
>>> model.has_lost()
False
>>> model.entity_positions()
{(1, 1): TankMech((1, 1), 5, 3, 3), (1, 2): TankMech((1, 2), 3, 3, 3), (1, 3):
 HealMech((1, 3), 2, 3, 2), (8, 8): Scorpion((8, 8), 3, 3, 2), (8, 7): Firefly(
(8, 7), 2, 2, 1), (7, 6): Firefly((7, 6), 1, 1, 1)}
>>> model.ready_to_save()
True
>>> tank = model.entity_positions()[(1,1)]
>>> tank.is_active()
True
>>> model.get_valid_movement_positions(tank)
[(2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (4, 1)]
>>> model.attempt_move(tank, (2,1))
>>> model.entity_positions()
{(2, 1): TankMech((2, 1), 5, 3, 3), (1, 2): TankMech((1, 2), 3, 3, 3), (1, 3):
 HealMech((1, 3), 2, 3, 2), (8, 8): Scorpion((8, 8), 3, 3, 2), (8, 7): Firefly(
(8, 7), 2, 2, 1), (7, 6): Firefly((7, 6), 1, 1, 1)}
>>> tank.is_active()
False
>>> model.ready_to_save()
False
>>> model.get_board().get_tile((2,5))
Building(3)
>>> model.make_attack(tank)
>>> model.get_board().get_tile((2,5))
Building(0)
>>> heal = model.entity_positions()[(1,3)]
>>> model.attempt_move(heal,(2,2))
>>> model.entity_positions()
{(2, 1): TankMech((2, 1), 5, 3, 3), (1, 2): TankMech((1, 2), 3, 3, 3), (2, 2):
HealMech((2, 2), 2, 3, 2), (8, 8): Scorpion((8, 8), 3, 3, 2), (8, 7): Firefly((
8, 7), 2, 2, 1), (7, 6): Firefly((7, 6), 1, 1, 1)}
>>> tank.get_health()
5
>>> model.make_attack(heal)
>>> tank.get_health()
7
>>> board2 = Board([['M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M'], ['M', '
 ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'M'], ['M', ' ', ' ', ' ', ' ', '3', ' '
, ' ', ' ', 'M'], ['M', ' ', ' ', ' ', '3', 'M', ' ', ' ', ' ', 'M'], ['M', ' '
, ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'M'], ['M', '2', ' ', ' ', ' ', ' ', ' ',
' ', ' ', 'M'], ['M', '2', ' ', ' ', ' ', 'M', 'M', 'M', 'M', 'M'], ['M', '2',
 ' ', ' ', ' ', ' ', ' ', 'M', 'M', 'M'], ['M', ' ', ' ', ' ', ' ', ' ', ' ', '
 ', ' ', 'M'], ['M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M']])
>>> entities2 = [TankMech((1, 1), 5, 3, 3), TankMech((1, 2), 3, 3, 3), HealMech
((1, 3), 2, 3, 2), Scorpion((8, 8), 3, 3, 2), Firefly((8, 7), 2, 2, 1), Firefly
((7, 6), 1, 1, 1)]
>>> model2 = BreachModel(board2, entities2)
>>> model2.entity_positions()
{(1, 1): TankMech((1, 1), 5, 3, 3), (1, 2): TankMech((1, 2), 3, 3, 3), (1, 3):
HealMech((1, 3), 2, 3, 2), (8, 8): Scorpion((8, 8), 3, 3, 2), (8, 7): Firefly((
8, 7), 2, 2, 1), (7, 6): Firefly((7, 6), 1, 1, 1)}
>>> model2.end_turn()
>>> model2.entity_positions()
{(1, 1): TankMech((1, 1), 5, 3, 3), (8, 5): Scorpion((8, 5), 3, 3, 2), (7, 5):
Firefly((7, 5), 1, 1, 1)}
```

4.2 View

The following are the classes and methods you are *required* to implement to complete the view component of this assignment. As opposed to section 4.1, where you would work through the required classes and methods in order, GUI development tends to require that you work on various interacting classes in parallel. Rather than working on each class in the order listed, you may find it beneficial to work on one *feature* at a time and test it thoroughly before moving on. It is likely that you will also need to implement components from the controller class (IntoTheBreach) in order to develop each feature. Each feature may require updates / extensions to the `IntoTheBreach` and `BreachView` classes, and potentially additions to other view classes as well. The recommended order of features (after reading through the following section in its entirety) are as follows:

1. `play_game`, `main`, and title: Create the window, ensure it displays when the program is run and set its title. Gradescope calls `play_game` in order to test your code, so you cannot earn marks for the View or Controller sections until you have implemented this function (See section 4.3 for details).
2. Title banner: Render the title banner at the top of the window.
3. `GameGrid`:
    - Basic tile display.
    - Highlighting tiles.
    - Entities display on top of tiles. Annotating building health on top of buildings.
    - Do *not* bind any commands to mouse buttons at this stage. This will be done when working on the controller.
4. `SideBar`:
    - Basic display (non-functional). Sidebar headings appear correctly. This step could also be done before the `GameGrid`.
    - Functionality. Ability to display entries and update.
5. `ControlBar`
    - Basic display. Buttons are laid out correctly. This step could also be done before both the `GameGrid` and `SideBar`.
    - Buttons are assigned the passed commands (You can assume None is passed in for each command until you complete the relevant feature in the controller section).

4.2.1 `GameGrid(AbstractGrid)`

`GameGrid` inherits from `AbstractGrid` provided in `a2_support.py`. `GameGrid` is a view component that displays the game board, with entities overlaid on top. Tiles are represented by certain colored squares, and entities are displayed by annotating special Unicode symbols (that is, regular plaintext that does not appear on most keyboards) on top of these squares. `a2_support.py` provides the exact colors and unicode symbols for you to display. An example of a completed `GameGrid` is presented in Figure 4. `GameGrid` should implement the following methods:

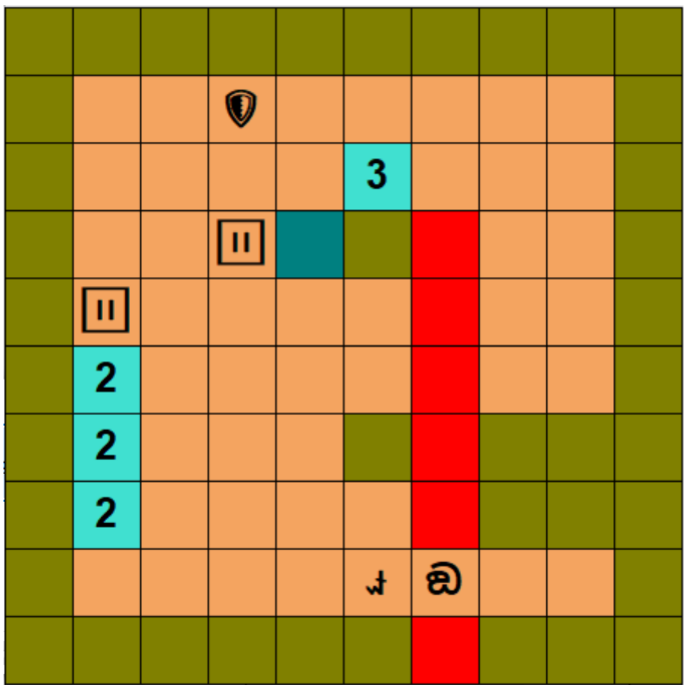


Figure 4: Example of a completed GameGrid partway through a game.

- `redraw( self, board: Board, entities: list[Entity], highlighted: list[tuple[int, int]] = None, movement: bool = False ) -> None:`

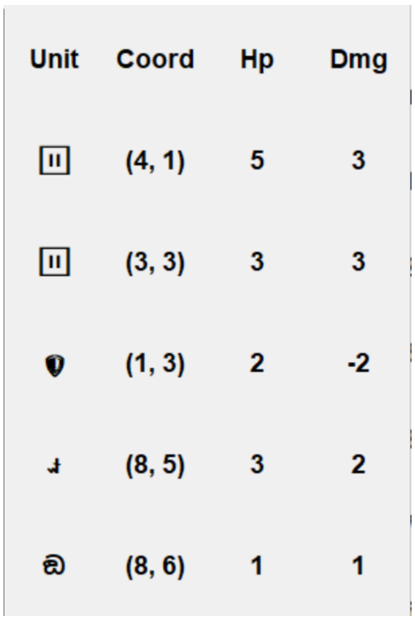
  Clears the game grid, then redraws it according to the provided information. Note that you must draw on the `GameGrid` instance it*self* (not directly onto `master` or any other tkinter widget). Destroyed buildings are colored differently from buildings that are not destroyed. If a list of highlighted cells are provided, then the color of those cells are overridden to be one of two highlight colors based on if `movement` is True (in which case possible moves are being highlighted and tiles should be `MOVE_COLOR` from `a2_support.py`) or False (in which case attacked tiles are being highlighted and tiles should be `ATTACK_COLOR` also from `a2_support.py`). If `highlighted` is None then no highlighting occurs and the `movement` parameter is ignored. The health of every building that is not destroyed is annotated on top of their respective building tiles. The special Unicode character associated with each entity is annotated on top of the tiles located at the position of each respective entity. All annotations appear in the center of their respective cells.

- `bind_click_callback(self, click_callback: Callable[[tuple[int, int]], None]) -> None`

  Binds the `<Button-1>` and `<Button-2>` events on itself to a function that calls the provided click handler at the correct position. Note: We bind *both* `<Button-1>` and `<Button-2>` to account for differences between Windows and Mac operating systems. Note: handling callbacks is an advanced task. These callbacks will be created within the controller, as this is the only place where you have access to the required modelling information. Integrate `GameGrid` into the game before attempting this method.

4.2.2 `SideBar(AbstractGrid)`

`SideBar` inherits from `AbstractGrid` provided in `a2_support.py`. `SideBar` is a view component that displays properties of each entity. Entities appear in descending priority order, with the highest priority entity appearing at the top of the sidebar, and the lowest priority entity appearing at the bottom of the sidebar. A `Sidebar` object is a grid with 4 columns. The top row displays the text



Unit	Coord	Hp	Dmg
⏸	(4, 1)	5	3
⏸	(3, 3)	3	3
🛡	(1, 3)	2	-2
[illegible]	(8, 5)	3	2
ඩ	(8, 6)	1	1

Figure 5: Example of a completed SideBar partway through a game

"Unit" in the first column, "Coord" in the second column, "Hp" in the third column, and "Dmg" in the fourth column. The `SideBar` maintains a constant height, but the number of rows will vary depending on the number of entities remaining in the game. Rows should expand out to fill available space. You do not need to handle visual artifacts caused by too many rows being present. An example of a completed `SideBar` is presented in Figure 5.

`SideBar` should implement the following methods:

- `__init__(self, master: tk.Widget, dimensions: tuple[int, int], size: tuple[int, int]) -> None`

  Instantiates a `SideBar` with the specified dimensions and size.

- `display(self, entities: list[Entity]) -> None`

  Clears the side bar, then redraws the header followed by the relevant properties of the given entities on the `SideBar` instance it*self*. Each entity in the given list should receive a row on the side bar containing (in order from left to right):

  - The special Unicode symbol used to display the entity on the `GameGrid` (provided in `a2_support.py`)
  - The current position of the entity
  - The current health of the entity
  - The damage the entity will deal during a given attack phase

  Entities appear in descending priority order, with the highest priority entity appearing at the top of the sidebar, and the lowest priority entity appearing at the bottom of the sidebar. A precondition to this function is that the given list of entities will be sorted in descending priority order.

4.2.3 `ControlBar(tk.Frame)`

`ControlBar` inherits from `tk.Frame`. `ControlBar` is a view component that contains three buttons that allow the user to perform administration actions. In order from left to right, the `ControlBar`

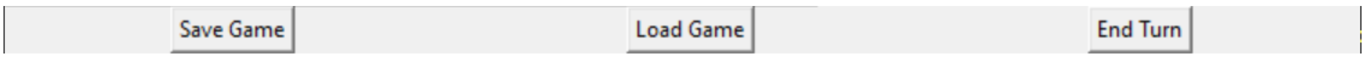


Figure 6: Example of a completed ControlBar

contains a save, load, and end turn button. An example of a completed `ControlBar` is presented in Figure 6. `ControlBar` should implement the following method:

- `__init__( self, master: tk.Widget, save_callback: Optional[Callable[[], None]] = None, load_callback: Optional[Callable[[], None]] = None, turn_callback: Optional[Call None]] = None, **kwargs ) -> None`

  Instantiates a `ControlBar` as a special kind of frame with the desired button layout. Note that the buttons must be created into the `ControlBar` frame it*self*. Each button receives the associated callback as its command. Note: handling callbacks is an advanced task. These callbacks will be created within the controller, as this is the only place where you have access to the required modelling information. Start this task by trying to render display correctly, without the callbacks. Integrate this view component into the game before working on the callbacks. Note that the tk.Button class can accept None as a command, so you can receive full marks for this component without implementing callbacks in the controller.

4.2.4 `BreachView()`

The `BreachView` class provides a wrapper around the smaller GUI components you have implemented, providing a single view interface for the controller. The view should be laid out such that there is a banner at the top of the window, with the `GameGrid` and `SideBar` appearing horizontally adjacent just below it. The `ControlBar` should appear below these two components. `a2_support.py` provides constants for the pixel sizes of each component. The `SideBar` should be the same height as the `GameGrid`. The banner and `ControlBar` should span the width of both the `GameGrid` and `SideBar`. An example of a completed `BreachView` is presented in Figure 1. `BreachView` must implement the following methods:

-
  ```
  __init__(
  self,
  root: tk.Tk,
  board_dims: tuple[int, int],
  save_callback: Optional[Callable[[], None]] = None,
  load_callback: Optional[Callable[[], None]] = None,
  turn_callback: Optional[Callable[[], None]] = None,
  ) -> None
  ```

  Instantiates view. Sets title of the given root window, and instantiates all child components. The buttons on the instantiated `CommandBar` receive the given callbacks as their respective commands.

- `bind_click_callback(self, click_callback: Callable[[tuple[int, int]], None]) -> None`

  Binds a click event handler to the instantiated `GameGrid` based on `click_callback`

- `redraw( self, board: Board, entities: list[Entity], highlighted: list[tuple[int, int]] = None, movement: bool = False ) -> None`

  Redraws the instantiated `GameGrid` and `SideBar` based on the given board, list of entities, and tile highlight information.

4.3 Controller

The controller is a single class, `IntoTheBreach`, which you must implement according to this section. As with the view section, you may find it beneficial to work on one *feature* at a time, instead of working through the required classes and functions in order. You should work on these features in tandem with features from the View section. Each feature may require updates / extensions to the `BreachView` class, and potentially updates to other view classes as well.
The recommended order of features (after reading through the following section in its entirety) are as follows:

1. `play_game`, `main`: Create the window and ensure it displays when the program is run. Gradescope calls `play_game` in order to test your view and controller code, so you cannot earn marks for the View or Controller sections until you have implemented this function.
2. Tile selection (This will require binding mouse buttons in the `GameGrid` class. See section 4.2 for details).
3. Mech Movement
4. Ending turn (this will require passing a function to the `ControlBar` class; see section 4.2 for details).
5. Saving/Loading game (this will require passing functions to the `ControlBar` class).
6. Win/Loss handling

4.3.1 `IntoTheBreach()`

`IntoTheBreach` is the controller class for the overall game. The controller is responsible for creating and maintaining instances of the model and view classes, event handling, and facilitating communication between the model and view classes. The controller will need to track which entity occupied the tile last clicked on by the user in order to correctly highlight tiles on the board (referred to as the *focussed* entity in the below methods). Refer to Table 1 for highlighting rules.

`IntoTheBreach` should implement the following methods:

- `__init__(self, root: tk.Tk, game_file: str) -> None`

  Instantiates the controller. Creates instances of `BreachModel` and `BreachView`, and redraws display to show the initial game state. You can assume that IO errors will not occur when loading a board from `game_file` *during this function.*

- `redraw(self) -> None`

  Redraws the view based on the state of the model and the current focussed entity.

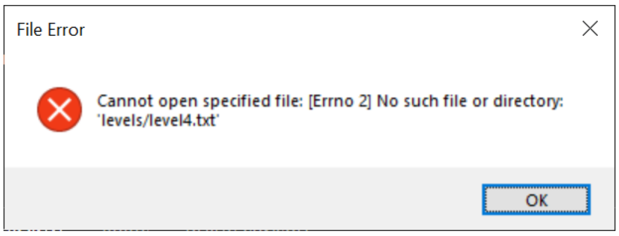
- `set_focussed_entity(self, entity: Optional[Entity]) -> None`

  Sets the given entity to be the one on which to base highlighting. Or clears the focussed entity if `None` is given.

- `make_move(self, position: tuple[int, int]) -> None`

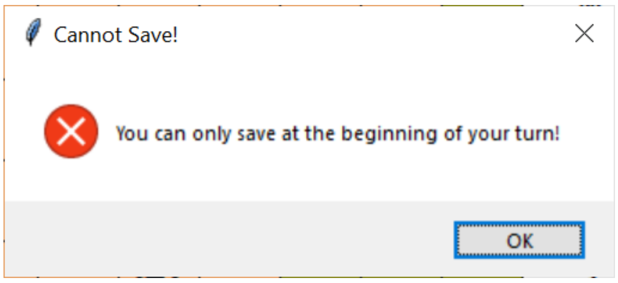
  Attempts to move the focussed entity to the given position, and then clears the focussed entity. Note that you have implemented a method in `BreachModel` that enforces the validity of a move according to the game rules already.

- `load_model(self, file_path: str) -> None`


File Error

Cannot open specified file: [Errno 2] No such file or directory: 'levels/level4.txt'

Figure 7: Example of an IO error messagebox. You may or may not have an icon in the top left corner depending on how you test this function, this will not impact your mark.


Cannot Save!

You can only save at the beginning of your turn!

Figure 8: Example of an invalid save attempt messagebox

Replaces the current game state with a new state based on the provided file. A precondition to this function, is that *if the file opens*, then it will contain exactly the string representation of a *BreachModel*. However, you may **NOT** assume that IOErrors will not occur when opening this file. If an IOError occurs when opening the given file, an error messagebox should be displayed to the user explaining the error that occurred, and the game state should not change. An example of the messagebox that should occur in the event of an IOError is given in Figure 7.

- `_save_game(self) -> None`

  If the the user has made no moves since the last time they clicked the end turn button, opens a `asksaveasfilename` file dialog to ask the user to specify a file, and then saves the current game state to that file. If the user has made at least one move since the last time they clicked the end turn button, shows an error message box explaining to the user that they can only save at the beginning of their turn. An example of this error message box is given in Figure 8. You should make sure to use exactly the messages provided in `a2_support.py`. You do not need to handle IOErrors for this operation.

- `_load_game(self) -> None`

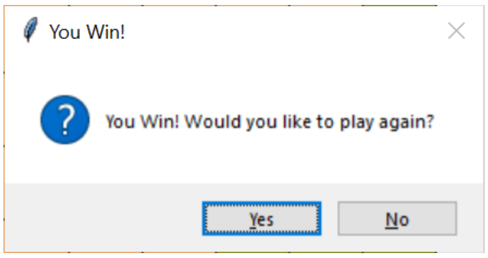
  Opens a `askopenfilename` file dialog to ask the user to specify a file, and then loads in a new game state from that file. If an IO error occurs when loading in a new game state, then a messagebox should be shown to the user explaining the error as described in `load_model`.

- `_end_turn(self) -> None`

  Executes the attack phase, enemy movement phase, and termination checking according to section 3. Examples of the messageboxes that should appear during termination checking are given in Figure 9.

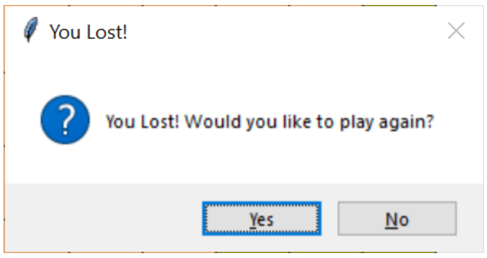
- `_handle_click(self, position: tuple[int, int]) -> None`

  Handler for a click from the user at the given (row, column) position. Applies the game rules specified in Table 1.



You Win!

You Win! Would you like to play again?



You Lost!

You Lost! Would you like to play again?

Figure 9: Examples of win (left) and loss (right) messageboxes

4.4 `play_game(root: tk.Tk, file_path: str) -> None`

The `play_game` function should be fairly short and do *exactly* two things:

1. Construct the controller instance using the given `file_path` and the root `tk.Tk` parameter.
2. Ensure the root window stays opening listening for events (using `mainloop`).

Note that the tests will call this function to test your code, rather than main.

4.5 `main() -> None`

The purpose of the `main` function is to allow you to test your own code. Like the `play_game` function, the `main` function should be fairly short and do *exactly* two things:

1. Construct the root `tk.Tk` instance.
2. Call the `play_game` function passing in the newly created root `tk.Tk` instance, and the path to any map file you like (e.g. `'levels/level1.txt'`).

5 Assessment and Marking Criteria

This assignment assesses course learning objectives:

1. apply program constructs such as variables, selection, iteration and sub-routines,
2. apply basic object-oriented concepts such as classes, instances and methods,
3. read and analyse code written by others,
4. analyse a problem and design an algorithmic solution to the problem,
5. read and analyse a design and be able to translate the design into a working program, and
6. apply techniques for testing and debugging, and
7. design and implement simple GUIs.

There are a total of 100 marks for this assessment item.

5.1 Functionality

Your program's functionality will be marked out of a total of 50 marks. The breakdown of marks for each implementation section is as follows:

- Model: 25 Marks
- View: 15 Marks

- Controller: 10 Marks

Your assignment will be put through a series of tests and your functionality mark will be proportional to the number of tests you pass. You will be given a *subset* of the functionality tests before the due date for the assignment.

You may receive partial marks within each section for partially working functions, or for implementing only a few functions.

You need to perform your *own* testing of your program to make sure that it meets *all* specifications given in the assignment. Only relying on the provided tests is likely to result in your program failing in some cases and you losing some functionality marks. Note: Functionality tests are automated, so string outputs need to match *exactly* what is expected.

When evaluating your view and controller, the automated tests will play the game and attempt to identify components of the game, how these components function during gameplay will then be tested. Well before submission, run the functionality tests to ensure components of your application can be identified. If the autograder is unable to identify components, you will not receive marks for these components, **even if your assignment is functional**. The tests provided prior to submission will help you ensure that all components can be identified by the autograder.

Your program must run in Gradescope, which uses Python 3.12. Partial solutions will be marked but if there are errors in your code that cause the interpreter to fail to execute your program, you will get zero for functionality marks. If there is a part of your code that causes the interpreter to fail, comment out the code so that the remainder can run. Your program must run using the Python 3.12 interpreter. If it runs in another environment (e.g. Python 3.8 or PyCharm) but not in the Python 3.12 interpreter, you will get zero for the functionality mark.

5.2 Code Style

The style of your assignment will be assessed by a tutor. Style will be marked according to the style rubric provided with the assignment. The style mark will be out of 50, note that style accounts for half the marks availible on this assignment.

The key consideration in marking your code style is whether the code is easy to understand. There are several aspects of code style that contribute to how easy it is to understand code. In this assignment, your code style will be assessed against the following criteria.

- Readability
  - Program Structure: Layout of code makes it easy to read and follow its logic. This includes using whitespace to highlight blocks of logic.
  - Descriptive Identifier Names: Variable, constant, and function names clearly describe what they represent in the program's logic. Do not use Hungarian Notation for identifiers. In short, this means do not include the identifier's type in its name, rather make the name meaningful (e.g. employee identifier).
  - Named Constants: Any non-trivial fixed value (literal constant) in the code is represented by a descriptive named constant (identifier).
- Algorithmic Logic
  - Single Instance of Logic: Blocks of code should not be duplicated in your program. Any code that needs to be used multiple times should be implemented as a function.

  - Variable Scope: Variables should be declared locally in the function in which they are needed. Global variables should not be used.
  - Control Structures: Logic is structured simply and clearly through good use of control structures (e.g. loops and conditional statements).

- Object-Oriented Program Structure

  - Classes & Instances: Objects are used as entities to which messages are sent, demonstrating understanding of the differences between classes and instances.
  - Encapsulation: Classes are designed as independent modules with state and behaviour. Methods only directly access the state of the object on which they were invoked. Methods never update the state of another object.
  - Abstraction: Public interfaces of classes are simple and reusable. Enabling modular and reusable components which abstract GUI details.
  - Inheritance & Polymorphism: Subclasses are designed as specialised versions of their superclasses. Subclasses extend the behaviour of their superclass without re-implementing behaviour, or breaking the superclass behaviour or design. Subclasses redefine behaviour of appropriate methods to extend the superclasses' type. Subclasses do not break their superclass' interface.
  - Model View Controller: Your program adheres to the Model-View-Controller design pattern. The GUI's view and control logic is clearly separated from the model. Model information stored in the controller and passed to the view when required.

- Documentation:

  - Comment Clarity: Comments provide meaningful descriptions of the code. They should not repeat what is already obvious by reading the code (e.g. `# Setting variable to 0`). Comments should not be verbose or excessive, as this can make it difficult to follow the code.
  - Informative Docstrings: Every function should have a docstring that summarises its purpose. This includes describing parameters and return values (including type information) so that others can understand how to use the function correctly.
  - Description of Logic: All significant blocks of code should have a comment to explain how the logic works. For a small function, this would usually be the docstring. For long or complex functions, there may be different blocks of code in the function. Each of these should have an in-line comment describing the logic.

5.3 Assignment Submission

You must submit your assignment electronically via Gradescope (`https://gradescope.com/`). You **must** use your UQ email address which is based on your student number
(e.g. s4123456@student.uq.edu.au) as your Gradescope submission account.

When you login to Gradescope you may be presented with a list of courses. Select
CSSE1001. You will see a list of assignments. Choose **Assignment 2**. You will be prompted to choose a file to upload. The prompt may say that you can upload any files, including zip files. You **must** submit your assignment as a single Python file called `a2.py` (use this name – all lower case), and *nothing* else. Your submission will be automatically run to determine the functionality mark. If you submit a file with a **different name**, the tests will **fail** and you will get **zero** for functionality. Do **not** submit **any** sort of archive file (e.g. zip, rar, 7z, etc.).

Upload an initial version of your assignment *at least* one week before the due date. Do this even if it is just the initial code provided with the assignment. If you are unable access Gradescope, contact the course helpdesk (csse1001@helpdesk.eecs.uq.edu.au) *immediately*. Excuses, such as you were not able to login or were unable to upload a file will not be accepted as reasons for granting an extension.

When you upload your assignment it will run a **subset** of the functionality autograder tests on your submission. It will show you the results of these tests. It is your responsibility to ensure that your uploaded assignment file runs and that it passes the tests you expect it to pass.

Late submissions of the assignment will **not** be marked. Do not wait until the last minute to submit your assignment, as the time to upload it may make it late. Multiple submissions are allowed and encouraged, so ensure that you have submitted an almost complete version of the assignment *well* before the submission deadline of 16:00. Submitting after the deadline incurs late penalties. Ensure that you submit the correct version of your assignment.

In the event of exceptional personal or medical circumstances that prevent you from handing in the assignment on time, you may submit a request for an extension. See the course profile for details of how to apply for an extension.

Requests for extensions must be made **before** the submission deadline. The application and supporting documentation (e.g. medical certificate) must be submitted via my.UQ. You must retain the original documentation for a minimum period of six months to provide as verification, should you be requested to do so.

5.4 Plagiarism

This assignment must be your own individual work. By submitting the assignment, you are claiming it is entirely your own work. You **may** discuss general ideas about the solution approach with other students. Describing details of how you implement a function or sharing part of your code with another student is considered to be **collusion** and will be counted as plagiarism. You **may not** copy fragments of code that you find on the Internet to use in your assignment.

Please read the section in the course profile about plagiarism. You are encouraged to complete *both* parts A and B of the academic integrity modules *before* starting this assignment. Submitted assignments will be electronically checked for potential cases of plagiarism.