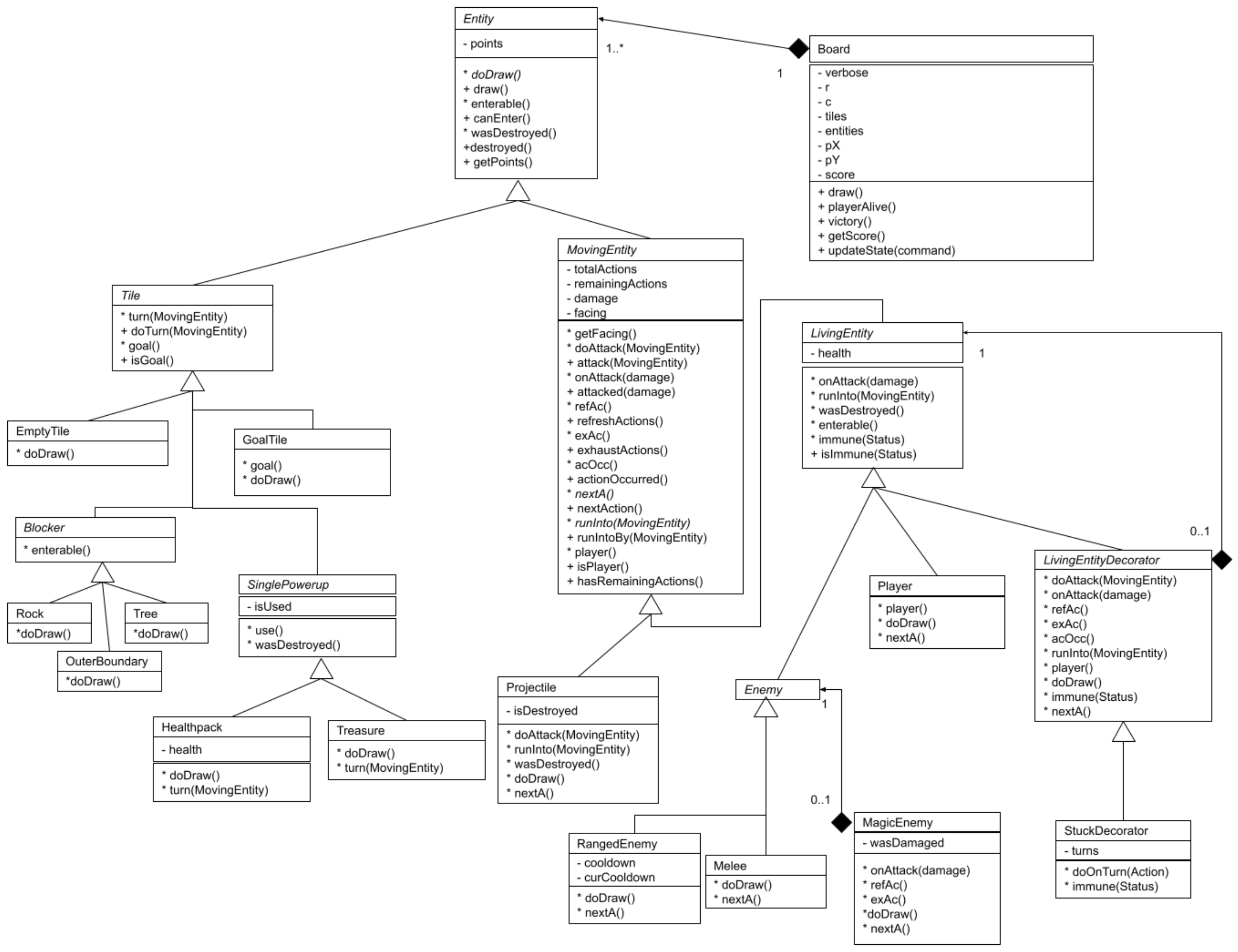
Full Documentation

Jaswin Hargun

UML Diagram



Entity
- points
* doDraw()
+ draw()
* enterable()
+ canEnter()
* wasDestroyed()
+destroyed()
+ getPoints()
1..*
1
Board
- verbose
- r
- c
- tiles
- entities
- pX
- pY
- score
+ draw()
+ playerAlive()
+ victory()
+ getScore()
+ updateState(command)
Tile
* turn(MovingEntity)
+ doTurn(MovingEntity)
* goal()
+ isGoal()
EmptyTile
* doDraw()
GoalTile
* goal()
* doDraw()
Blocker
* enterable()
Rock
*doDraw()
Tree
*doDraw()
OuterBoundary
*doDraw()
SinglePowerup
- isUsed
* use()
* wasDestroyed()
Healthpack
- health
* doDraw()
* turn(MovingEntity)
Treasure
* doDraw()
* turn(MovingEntity)
MovingEntity
- totalActions
- remainingActions
- damage
- facing
* getFacing()
* doAttack(MovingEntity)
+ attack(MovingEntity)
* onAttack(damage)
+ attacked(damage)
* refAc()
+ refreshActions()
* exAc()
+ exhaustActions()
* acOcc()
+ actionOccurred()
* nextA()
+ nextAction()
* runInto(MovingEntity)
+ runIntoBy(MovingEntity)
* player()
+ isPlayer()
+ hasRemainingActions()
Projectile
- isDestroyed
* doAttack(MovingEntity)
* runInto(MovingEntity)
* wasDestroyed()
* doDraw()
* nextA()
LivingEntity
- health
1
* onAttack(damage)
* runInto(MovingEntity)
* wasDestroyed()
* enterable()
* immune(Status)
+ isImmune(Status)
0..1
Player
* player()
* doDraw()
* nextA()
LivingEntityDecorator
* doAttack(MovingEntity)
* onAttack(damage)
* refAc()
* exAc()
* acOcc()
* runInto(MovingEntity)
* player()
* doDraw()
* immune(Status)
* nextA()
Enemy
1
0..1
RangedEnemy
- cooldown
- curCooldown
* doDraw()
* nextA()
Melee
* doDraw()
* nextA()
MagicEnemy
- wasDamaged
* onAttack(damage)
* refAc()
* exAc()
*doDraw()
* nextA()
StuckDecorator
- turns
* doOnTurn(Action)
* immune(Status)

Overview of Classes

I will go through a high level overview of each individual class, though I will not go too in depth to avoid making this section too long, so if you want to see how I implemented something, feel free to check in the corresponding .h or .cc file. To make this easier, all of the function names in the UML correspond to the function names in the code (though some parameter and field names have been omitted to make it easier to understand the UML diagram).

Note that since I used NVII everywhere in my design, there are a lot of public functions that only exist to provide a public, non virtual interface by calling protected virtual functions.

Board

In its fields, Board tracks the number of rows and columns of the board, the x and y coordinates of the player, and the score. It also has a verbose field, which determines whether the board is printed every time an Entity takes an action or just after all Entities have had their turn. The most important fields are tiles and entities. These are 2D vectors that contain all of the tiles on the board and all of the MovingEntities on the board respectively. Note that every square in Board must have a Tile (even if the tile is just EmptyTile or OuterBoundary). This is not the case for entities, which can have entries that are nullptr if there is no MovingEntity on the square. This restriction that all squares have 1 tile and at most 1 MovingEntity is why this set of 2 vectors is used to store this data.

The most important function to understand in Board is updateState. It takes a character as a parameter, which is the command entered by the player. From here, it allows every Entity on the Board to take its turn. Turns occur in a specific order. First, the Player gets to do their turn. Then, all MovingEntities take their turn, starting from the entity at the top left of the board (if there is one) before doing the turns of each Entity to the left and below (essentially, the direction of turns is the same as the direction in which we read English text: left and down). After this, Tiles get their turn (again in the same order, from left to right and from top to bottom). Finally, any Tiles or Entities that were destroyed during the turn that haven't been deleted are deleted and the board is printed out. Note that no destroyed entities should be cleaned up here (since they should be deleted when they are killed), but this adds an extra layer of redundancy, and it is still necessary to delete Tiles that were destroyed. Note that since each square must have a Tile, deleting a Tile consists of replacing it with an EmptyTile (rather than a nullptr, as is done when deleting an Entity).

The rest of Board's functions are getters, a function to output the board to the terminal as ASCII characters (draw), and some private helper functions.

Entity

Entity is the parent class for most classes in my implementation as it is used to enforce a common interface. Each entity stores an integer field called points, which is the number of points the Player earns if they destroy this Entity. Note that most Entities have 0 points, and some cannot even be destroyed by the Player (such as the EmptyTile).

Each Entity has a set of functions. draw calls doDraw, which outputs an ASCII character to cout in order to display the Entity. canEnter calls enterable, which returns true if it is possible to enter this Entity. Some Entities, like EmptyTiles and Projectiles, can be entered while others, like Enemies or Blockers, cannot. destroyed calls wasDestroyed, which returns true if this entity was just destroyed. Note that this does not mean destruction in the C++ sense, where a destructor is run. Instead, it is defined differently for different Entities. For example, a LivingEntity is destroyed if they are killed (meaning their health reaches 0), while a Projectile is destroyed when it hits something. Whenever an entity is destroyed, Board adds its points to its score and deletes it (in the C++ sense this time). This ensures that points are awarded for any destruction of an Entity worth points. For example, even if a RangedEnemy accidentally shoots their friend and kills them, the score will increase just like if the Player had killed that enemy.

Tile

This one is pretty self explanatory. Each Tile has a isGoal function, which calls goal and allows Board to use virtual dispatch to determine if the Player is standing on a Goal (in which case they win) or some other type of Tile. One thing that might seem weird is that Tiles have a doTurn function, which takes a MovingEntity as an argument. This allows for features such as powerups, which trigger when a Player is standing on them on their turn. It also makes it much easier to implement status tiles, such as poisonous tiles, in the future.

EmptyTile

This is the “default” tile. It does nothing and blocks nothing from standing on it or passing through it.

Blocker

All blockers are functionally the same, serving as impediments that MovingEntities cannot enter or go through. The only difference between the 3 types is appearance.

GoalTile

This tile is essentially an EmptyTile with the added stipulation that the Player wins if they are standing on this tile.

SinglePowerup

This is an abstract base class for powerups that are single use. Essentially, when a Player stands on this tile, it does something and then marks itself as destroyed to be turned into an EmptyTile by Board.

Healthpack

This class is constructed with a parameter health, which is the amount of health that it adds to the Player when it stands on it. Note that it can heal a Player beyond their initial health. It also increases the score slightly.

Treasure

When the Player stands on this tile, it adds a large amount to their score before being turned into a regular EmptyTile.

MovingEntity

Note that in the same header as this class, 2 enumeration classes are defined: Action and Direction. Direction can be up, down, left, or right, while actions include doing nothing, turning to face a certain direction, moving, performing a melee attack, shooting a projectile, teleporting, or cloning oneself.

Each MovingEntity has fields totalActions and remainingActions. These represent the number of actions this entity can perform per turn and the remaining number of actions remaining in this turn respectively. It also has field damage, which is the amount of damage done by this entity when it attacks, and facing, which is the direction in which this entity is facing (which determines things like the direction this entity hits and shoots projectiles).

Each MovingEntity has a few getters, as well as setters that set remainingActions to 0 or to totalActions. They also have a function isPlayer, which serves a similar purpose as Tile's isGoal in that it allows clients to determine if a MovingEntity is a Player without using something like dynamic_cast.

The most important function here is nextAction, which calls nextA. I did not include all of the parameters in the UML diagram because there are a lot. This function essentially tells the Entity what tiles and MovingEntities are around it, as well as where it is relative to the player and what the player's most recent action was. The MovingEntity takes this information and returns the Action it would like to take and which Direction it wants to move.

Projectile

Projectiles store a boolean isDestroyed, which determines whether a projectile is destroyed or not. Projectiles, once fired, continue to move in the same direction until it hits something, at which point it is destroyed (note that this includes other Projectiles). Importantly, while all MovingEntities can have more than 1 action per turn, Projectiles are the only ones that have this property by default, meaning they move faster than other MovingEntities by default.

LivingEntity

This entity has health, and once the health reaches 0, it is considered to have been destroyed. Most of its functions are simply overriding parent functions, but it is worth noting isImmune. This takes an element of the Status enumeration class, which is defined in the same header, and determines whether the LivingEntity is immune to that status. The advantages of this are discussed further later in the document.

Enemy

This abstract base class doesn't actually have any special functions. It just exists in case there is ever a need to refer to a group of enemy pointers in the future. This was originally planned to be necessary for Magic enemies but the design was changed (as discussed later in this document).

MeleeEnemy

This enemy moves until it is next to the player, at which point it turns to face the player and attacks.

RangedEnemy

This enemy moves until it is on the same row or column as the player, at which point it turns to face the player and fires a projectile. To prevent this enemy from spamming projectiles too much, it has a cooldown whenever it fires a projectile, which is stored in its fields.

MagicEnemy

This class uses templates to add functionality to any Enemy class provided. It adds the ability to teleport and clone oneself. When a MagicEnemy is attacked, it has a ⅓ chance to teleport somewhere else, a ⅓ chance to make a clone of itself somewhere else (which has the same health, damage, and number of points), and a ⅓ chance to do what a normal enemy would do in that situation. Other than this ability to clone itself and teleport, MagicEnemies act exactly like the enemy that they are templated with.

Player

This class represents the player of the game. It ignores its surroundings when determining its actions and instead relies only on the input of the player.

LivingEntityDecorator

This abstract base class implements the Decorator pattern. It stores a unique_ptr to a LivingEntity that it decorates. It may seem like it has a lot of functions, but all of these simply override the parent class's functions to call the respective function on the stored pointer.

StuckDecorator

This decorator adds the Stuck status effect to a LivingEntity. This status effect prevents the entity from moving for a set amount of turns before they break free. Note that this only affects movement, which means entities that are stuck can still freely shoot and attack.

Resilience to Change

I structured my program with multiple possible changes in mind. I made a few design choices in pursuit of that goal that I will explain below.

Following the NVII Idiom

I made sure to follow the NVII idiom everywhere in my code. This helped me implement certain features and checks. For example, nextAction in MovingEntity is a public interface function that calls a protected virtual function nextA. This allows MovingEntity to take care of decrementing the total remaining actions in the turn rather than forcing the client to take care of it. Design choices such as this reduce the potential that a future coder (or myself) might make a mistake when creating a new child class and forget to implement this feature, which would break the game's logic. If there is ever a need to add or enforce a new invariant, these public functions can be altered to enforce the invariant (similar to what MovingEntity does with nextAction).

Making a Separate Player Class

When I was first planning the design for this project, I considered having Player be a variable owned by the game. However, making Player a separate class makes the design much more flexible. If a new function is added to MovingEntity or LivingEntity, no changes are needed to Board to have that change applied to Player as well. This also makes it much easier to add multiplayer in the future. To add multiplayer, all one would have to do is create two Player objects on the board rather than one (and change the game score to be stored by the Player class rather than the Board class if you want to have 2 scores so players can compete). To store the game score in the Player class, all one would have to do is add an extra field and a public function that allows the Player to be notified when their score should increase, as well as changing Board's getScore function to query the players rather than checking its own field.

Templating Magic Enemies

I originally considered making MagicEnemy an abstract base class that inherited from Enemy. However, this would have meant all enemy types would need to have an extra, extremely similar class that inherits from MagicEnemy rather than just from Enemy. Instead, I used templating to make a single MagicEnemy template that could be applied to ranged and melee enemies. This has the added advantage of making it much easier to add new enemy types. With the templating implementation, if a new type of enemy is added as a child of Enemy, it can immediately be made Magical, meaning it will get the teleport and clone functionality.

Using Enumeration Classes

I used enumeration classes in a few different places. The two examples that I think impact the design's flexibility are Action and Status. These are defined in movingentity.h and livingentity.h respectively.

I originally planned to pass the board to each individual class and let it handle moving, attacking, etc. However, I quickly realized this was infeasible. The code to perform common actions such as moving would be repeated in numerous places, which would both be bad style and make it much harder to add new features (since these would need to be duplicated in numerous places). It would also be much harder to enforce invariants about the board (such as the fact that the edges are a special type of blocker) if every child class had access to the board and the ability to change it at will.

This led me to use the structure I have now. The Board gives each MovingEntity the chance to make its turn, and the MovingEntity looks at the entities around it and the position of the player to decide what action it wants to take (and in which direction). Then, it returns this Action and Direction to the Board - essentially requesting to make this action. This means all of the code for handling the request is contained in Board. This not only reduces duplicate code, but it makes it much easier to add new Actions. If, for example, I wanted to add an ExplosiveProjectile class which would explode and damage everything in a certain radius, I could add an Explode action. None of the other classes would need to be changed (unless I wanted to give them the option to explode as well), so the only extra code needed would be in Board, which would need to be altered to handle requests by MovingEntities to Explode.

Another important effect of this system is the ability for other objects to intercept this request before it makes it to Board. For example, the StuckDecorator class will intercept requests to move, and if it decides the LivingEntity that is decorated has not escaped being stuck, it will replace requests to move with requests to just face a certain direction before passing on the altered request to Board.

I have been focusing on Action, but Status has a similar effect. The Stuck status means a LivingEntity is stuck in place . While all of the logic of being stuck (namely, preventing a certain number of movements) is handled by the StuckDecorator, the Status enumeration class makes it much easier to handle things like immunity. For example, if a Quicksand tile applied the Stuck effect every time a LivingEntity ended its turn on the tile, it would be impossible for entities with less than 1 action per turn to escape before the effect is reapplied. The immunity function prevents this by ensuring the Stuck status isn’t applied more than once.

My original plan to implement this immunity function was to check if the LivingEntity was decorated by a StuckDecorator, but this would be very inflexible. For example, it would make it difficult to implement power ups that confer immunity or to add other reasons that a LivingEntity might be immune to a status effect since such changes would require changes to the immunity checking logic. Using an enumeration class solves this problem. The isImmune() function of LivingEntity will return false for all Statuses by default, and child classes or decorators can modify this behavior (as StuckDecorator does). This means if one wants to add a new Status or a new form of immunity, all one needs to do is add an entry to the Status enumeration class and a new Decorator that handles this logic.

One more way enumeration allows for extension is by making it easier for Statuses to interact with Tiles and Projectiles. For example, say someone wanted to make projectiles that pass through Poison tiles poisonous, meaning they would poison any LivingEntity that they hit. All one would need to do is add the ability for a Projectile to apply a decorator if the LivingEntity is not immune (which, as mentioned earlier, is easy to check with this implementation).