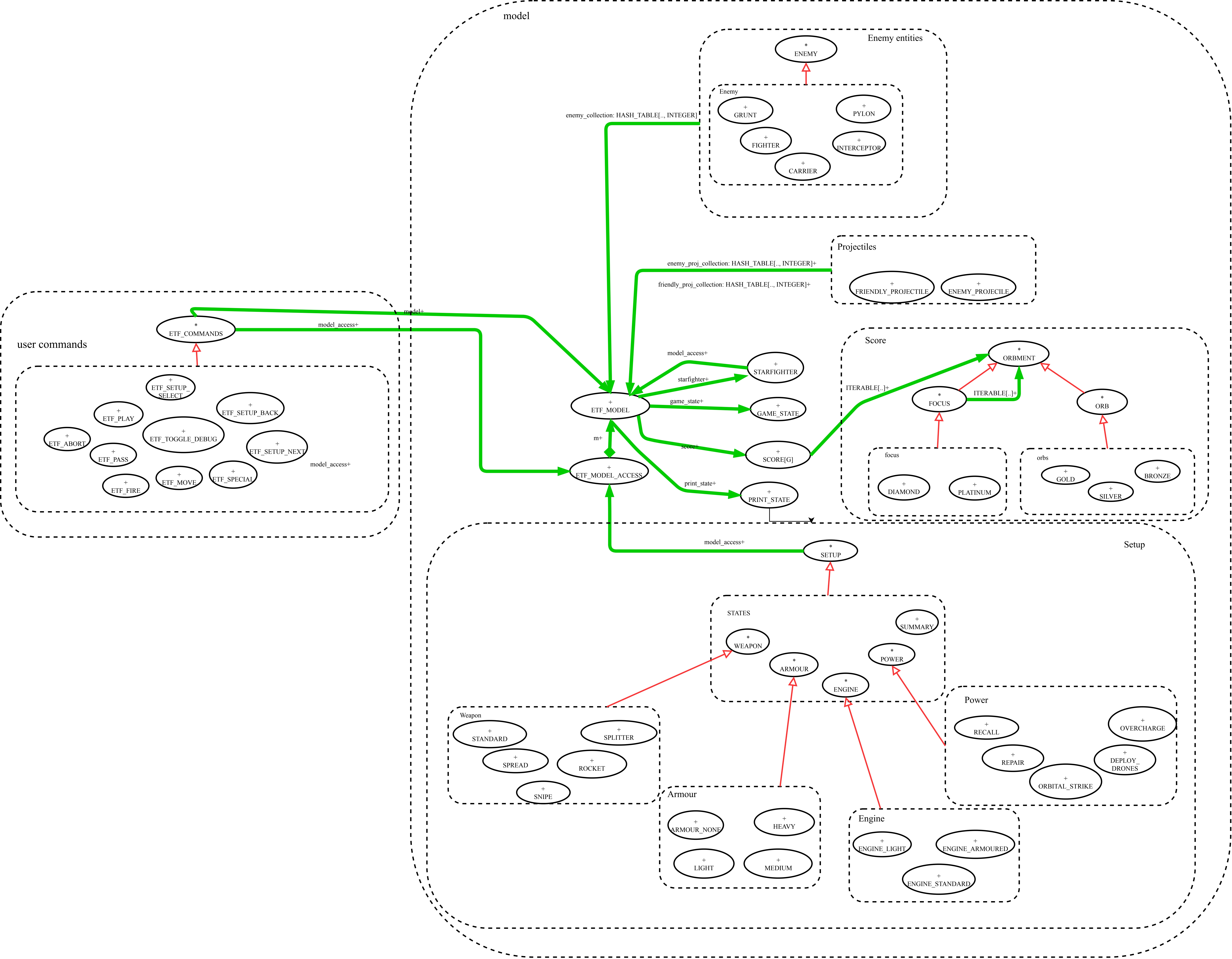
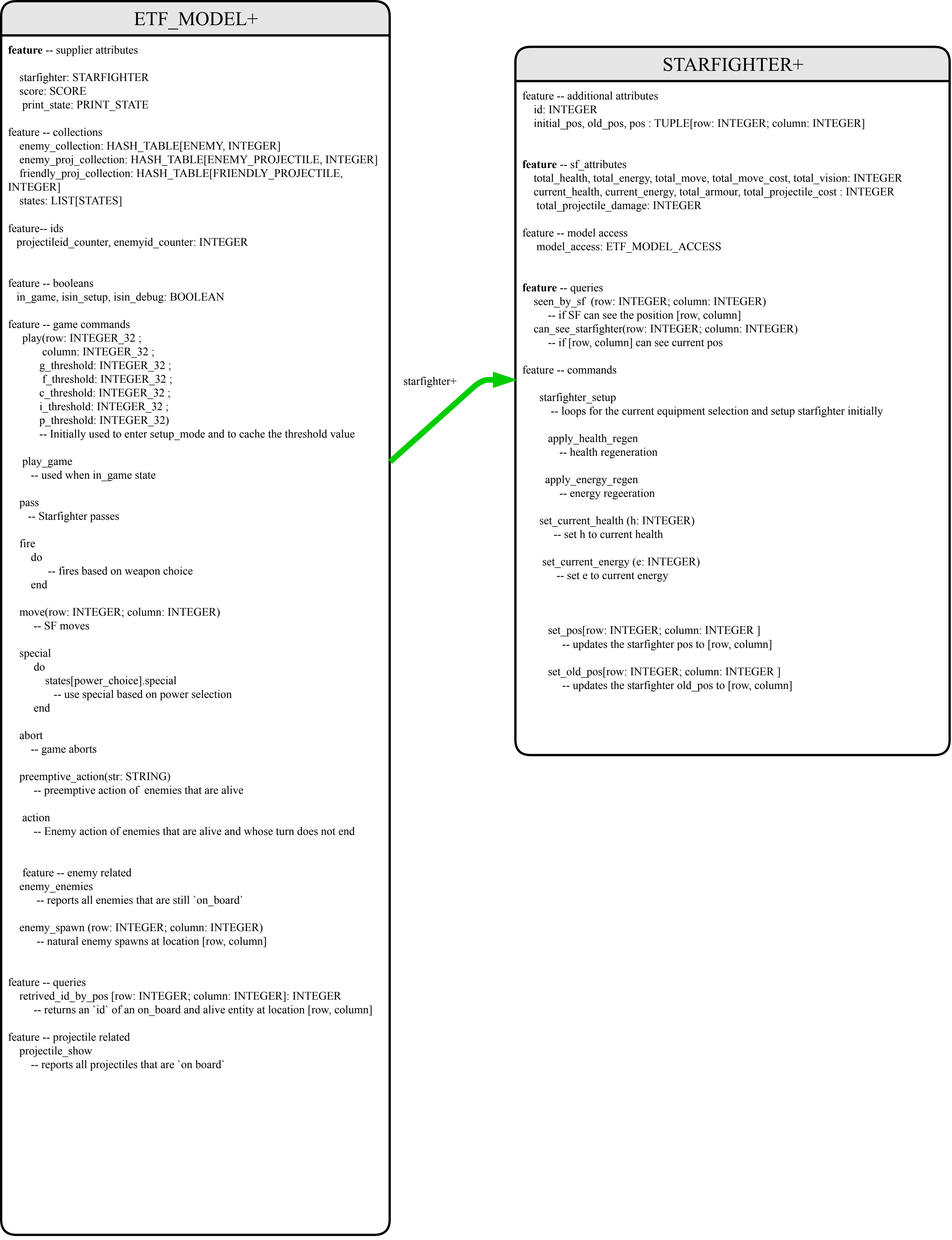
EECS 3311 – SOFTWARE DESIGN
FALL 2020-2021
Kaumilkumar Patel
Student no: 216008914
Prism Login: kaumil97


model
Enemy entities
*
ENEMY
Enemy
+
GRUNT
+
PYLON
+
FIGHTER
+
INTERCEPTOR
+
CARRIER
enemy_collection: HASH_TABLE[.., INTEGER]
Projectiles
enemy_proj_collection: HASH_TABLE[.., INTEGER]+
friendly_proj_collection: HASH_TABLE[.., INTEGER]+
+
FRIENDLY_PROJECTILE
+
ENEMY_PROJECILE
user commands
*
ETF_COMMANDS
model+
model_access+
+
ETF_SETUP_
SELECT
+
ETF_PLAY
+
ETF_SETUP_BACK
+
ETF_ABORT
+
ETF_TOGGLE_DEBUG
+
ETF_PASS
+
ETF_SETUP_NEXT
+
ETF_MOVE
+
ETF_SPECIAL
+
ETF_FIRE
model_access+
model_access+
+
STARFIGHTER
starfighter+
+
ETF_MODEL
game_state+
+
GAME_STATE
m+
score+
+
SCORE[G]
+
ETF_MODEL_ACCESS
print_state+
+
PRINT_STATE
Score
*
ORBMENT
ITERABLE[..]+
*
FOCUS
ITERABLE[..]+
*
ORB
focus
+
DIAMOND
+
PLATINUM
orbs
+
GOLD
+
BRONZE
+
SILVER
model_access+
*
SETUP
Setup
STATES
+
SUMMARY
*
WEAPON
*
ARMOUR
*
POWER
*
ENGINE
Weapon
+
STANDARD
+
SPLITTER
+
SPREAD
+
ROCKET
+
SNIPE
Power
+
RECALL
+
OVERCHARGE
+
REPAIR
+
DEPLOY_
DRONES
+
ORBITAL_STRIKE
Armour
+
ARMOUR_NONE
+
HEAVY
+
LIGHT
+
MEDIUM
Engine
+
ENGINE_LIGHT
+
ENGINE_ARMOURED
+
ENGINE_STANDARD



```
ETF_MODEL+

feature -- supplier attributes

   starfighter: STARFIGHTER
   score: SCORE
    print_state: PRINT_STATE

feature -- collections
   enemy_collection: HASH_TABLE[ENEMY, INTEGER]
   enemy_proj_collection: HASH_TABLE[ENEMY_PROJECTILE, INTEGER]
   friendly_proj_collection: HASH_TABLE[FRIENDLY_PROJECTILE,
INTEGER]
   states: LIST[STATES]

feature-- ids
  projectileid_counter, enemyid_counter: INTEGER


feature -- booleans
  in_game, isin_setup, isin_debug: BOOLEAN

feature -- game commands
    play(row: INTEGER_32 ;
         column: INTEGER_32 ;
        g_threshold: INTEGER_32 ;
         f_threshold: INTEGER_32 ;
        c_threshold: INTEGER_32 ;
        i_threshold: INTEGER_32 ;
        p_threshold: INTEGER_32)
        -- Initially used to enter setup_mode and to cache the threshold value

    play_game
      -- used when in_game state

   pass
     -- Starfighter passes

   fire
      do
          -- fires based on weapon choice
      end

   move(row: INTEGER; column: INTEGER)
       -- SF moves

   special
       do
          states[power_choice].special
            -- use special based on power selection
       end

   abort
      -- game aborts

   preemptive_action(str: STRING)
       -- preemptive action of  enemies that are alive

    action
       -- Enemy action of enemies that are alive and whose turn does not end


    feature -- enemy related
   enemy_enemies
        -- reports all enemies that are still `on_board`

   enemy_spawn (row: INTEGER; column: INTEGER)
        -- natural enemy spawns at location [row, column]


feature -- queries
   retrived_id_by_pos [row: INTEGER; column: INTEGER]: INTEGER
       -- returns an `id` of an on_board and alive entity at location [row, column]

feature -- projectile related
   projectile_show
      -- reports all projectiles that are `on board`
```

```
STARFIGHTER+

feature -- additional attributes
   id: INTEGER
   initial_pos, old_pos, pos : TUPLE[row: INTEGER; column: INTEGER]


feature -- sf_attributes
   total_health, total_energy, total_move, total_move_cost, total_vision: INTEGER
   current_health, current_energy, total_armour, total_projectile_cost : INTEGER
    total_projectile_damage: INTEGER

feature -- model access
    model_access: ETF_MODEL_ACCESS


feature -- queries
   seen_by_sf  (row: INTEGER; column: INTEGER)
        -- if SF can see the position [row, column]
   can_see_starfighter(row: INTEGER; column: INTEGER)
        -- if [row, column] can see current pos

feature -- commands

     starfighter_setup
        -- loops for the current equipment selection and setup starfighter initially

        apply_health_regen
           -- health regeneration

       apply_energy_regen
           -- energy regeeration

     set_current_health (h: INTEGER)
         -- set h to current health

      set_current_energy (e: INTEGER)
          -- set e to current energy



        set_pos[row: INTEGER; column: INTEGER ]
           -- updates the starfighter pos to [row, column]

        set_old_pos[row: INTEGER; column: INTEGER ]
           -- updates the starfighter old_pos to [row, column]
```

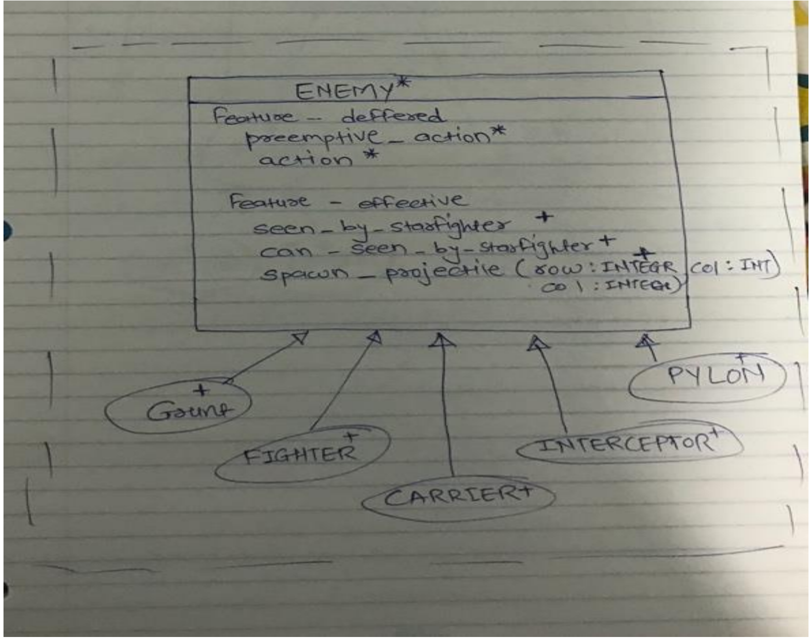
Section: Enemy Actions

How enemies perform action in Phase 5 of a turn, including both preemptive and non-preemptive actions

Given that the game is not over and starfighter is not destroyed, the enemy action phase works in two sub phases.

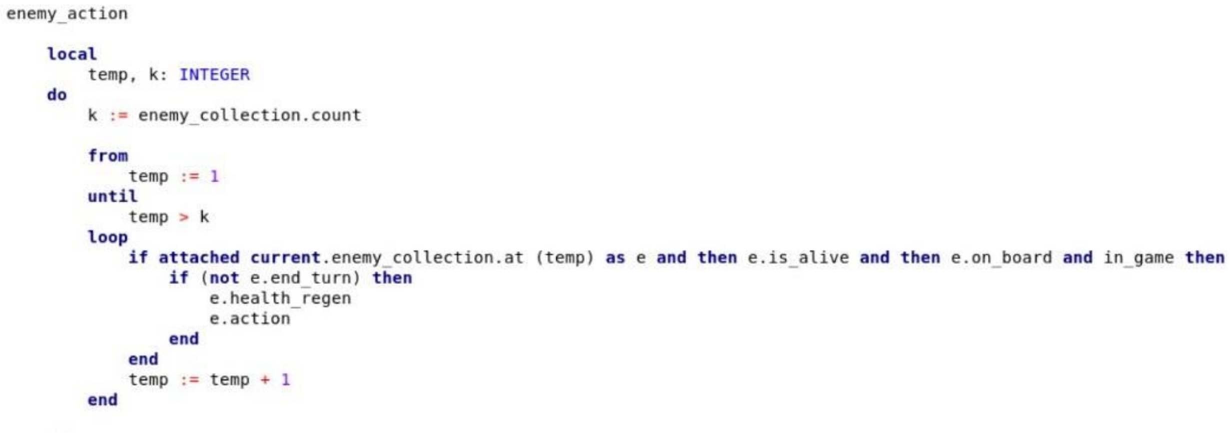
1. Preemptive action: This action is based on the game's 'turn'. And only apply to the enemies that was already on the board. (Not spawned on current turn).
2. Non-preemptive action: This action is based on the vision of the starfighter and the vision of the enemy. (whether enemy can see starfighter or not).

Here is the abstract view of enemy cluster.



Each enemy has different behaviour while performing preemptive and non-preemptive action with the same type of attributes. So, for maintaining the single choice principle, I come up with single level inheritance. The features 'preemptive action' and 'action' are the deferred methods with the effective definition given in all effective descendent classes. To track the occurrence of the enemy's, I created the polymorphic HASH_TABLE [ENEMY, KEY] in the ETF_MODEL.
Where the key is the integer value given to ENEMY by key generator.
This key is the unique ID of ENEMY. Because of the associated unique ids, we can easily retrieve the enemy object via key with the constant time complexity.

Here is the preemptive and non-preemptive action at ETF_MODEL level



```
enemy_action

    local
        temp, k: INTEGER
    do
        k := enemy_collection.count

        from
            temp := 1
        until
            temp > k
        loop
            if attached current.enemy_collection.at (temp) as e and then e.is_alive and then e.on_board and in_game then
                if (not e.end_turn) then
                    e.health_regen
                    e.action
                end
            end
            temp := temp + 1
        end
```

By Iterating the hash table and calling the preemptive action, dynamically at runtime, each enemy perform their preemptive action. By executing a successful turn command, the instance of turn is reported to all enemy. The instance of the turn is checked by enemy to decide which preemptive action should be performed or no preemptive action needed.

The pre-state value of hashtable’s count is cached to restrict my cursor, so I can avoid the preemptive action of newly spawned enemy while performing turn.

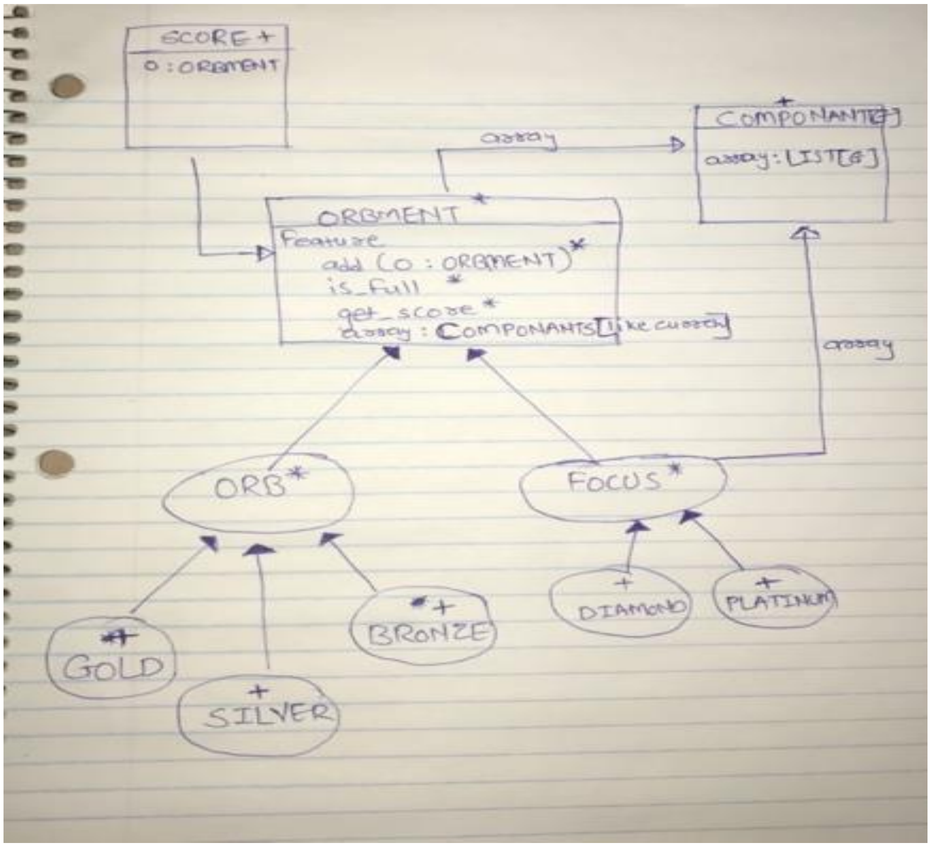
Note that the method is always checking the whether the ‘in_game’ variable true or not. This checking is for stopping the preemptive action while the game ended between iteration.

Second, the action (non preemptive) is based on vision. Enemy have access to know the starfighter location via model_access and vice versa. Enemy has a variable called ‘end_turn’ which can be set by ‘set_turn’ method.
This variable mainly used for deciding whether to perform action or not regardless of vision.

The method for action works similarly as preemtive action, but always check the “end_trun” is true for the enemy.
As described in the diagram above certain feature works similarly, so implemented on abstract class level to avoid code duplication and maximize reusability.

It’s important to understand the separation of concern while doing project. As we discussed, there are only two effective features implemented in each enemy class and all other same features at the upper level to satisfy the Cohesion Principle.

Section: Scoring of Starfighter
How the scoring of the starfighter works



I have implemented scoring using the Composite design pattern. The BON diagram for scoring is shown below.
I used recursive nature of programming to calculate the score. As per the requirement the COMPONANT[ORBMENT] in ORBMENT is a linear container. Similarly, the FOCUS has the linear container but it's fixed size. Science, the FOCUS is a fix sized set of orbment, we have to restrict it's size by making 'is_full' feature. Treating each individual orb as a different component, the composite pattern is most relevant design pattern can be used to solve this problem.

The feature `is_full: BOOLEAN` is implemented in each of the effective class and behaves differently. The linear container in the FOCUS can be created by taking one of ORB of type GOLD, SILVER or BRONZE or FOUCS itself. Depending on FOCUS type you can add up to 4 ORBMENTS in a container.

The feature `is_full` in focus is implemented using the code fragment shown below.

```
is_full: BOOLEAN
do
    if array.count < 4 then
        result := false
    elseif array.count = 4 and not (attached {FOCUS} array[4]) then
        result := true
    else
        if attached{FOCUS} array[4] as f_obj then
            result := f_obj.is_full
        end
    end
end
```

This feature helps us add the ORBMENT from left to right i.e filling up the left subtree before adding it to the right. The s_focus in class SCORE is of unlimited capacity. The linear container in it is initialized by G of type ORBMENT. The ORBMENT are being added to s_focus using the algorithm below:

```
If container not (is_full) then
    Add the ORBMENT at (count + 1)

Elseif container (count) is not (is_full) then
    container[count].add(o: ORBMENT)
     This is a recursive call to the add feature implemented in FOCUS

     If it is a base case (i.e GOLD, SILVER, BRONZE) the is_full is always true

 Else
    ADD the ORBMENT at (count + 1)
```

Since our linear container is of type LIST statically, we follow the principle of Programming from interface.

The `add` feature being called at runtime is based on the dynamic type `SILVER`, thus dynamic binding occurs at runtime and this helps satisfy the Single choice principle.

The whole scoring component works on the principle of separation of concern where only the relevant features are implements in the given classes and thereby satisfying the Cohesion principle.