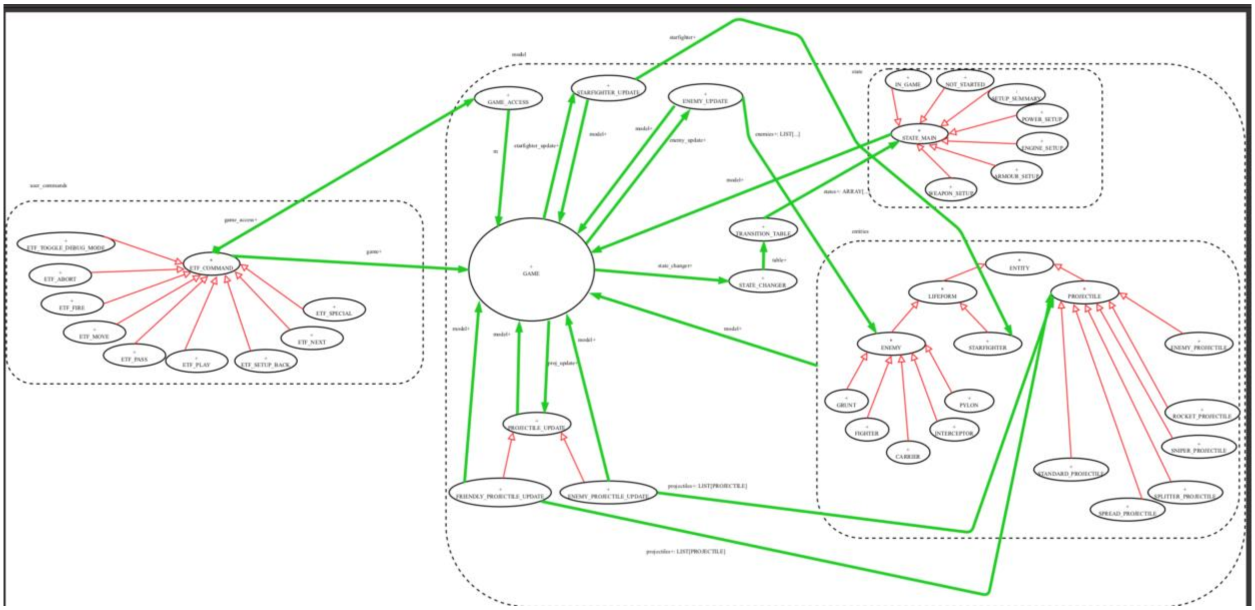
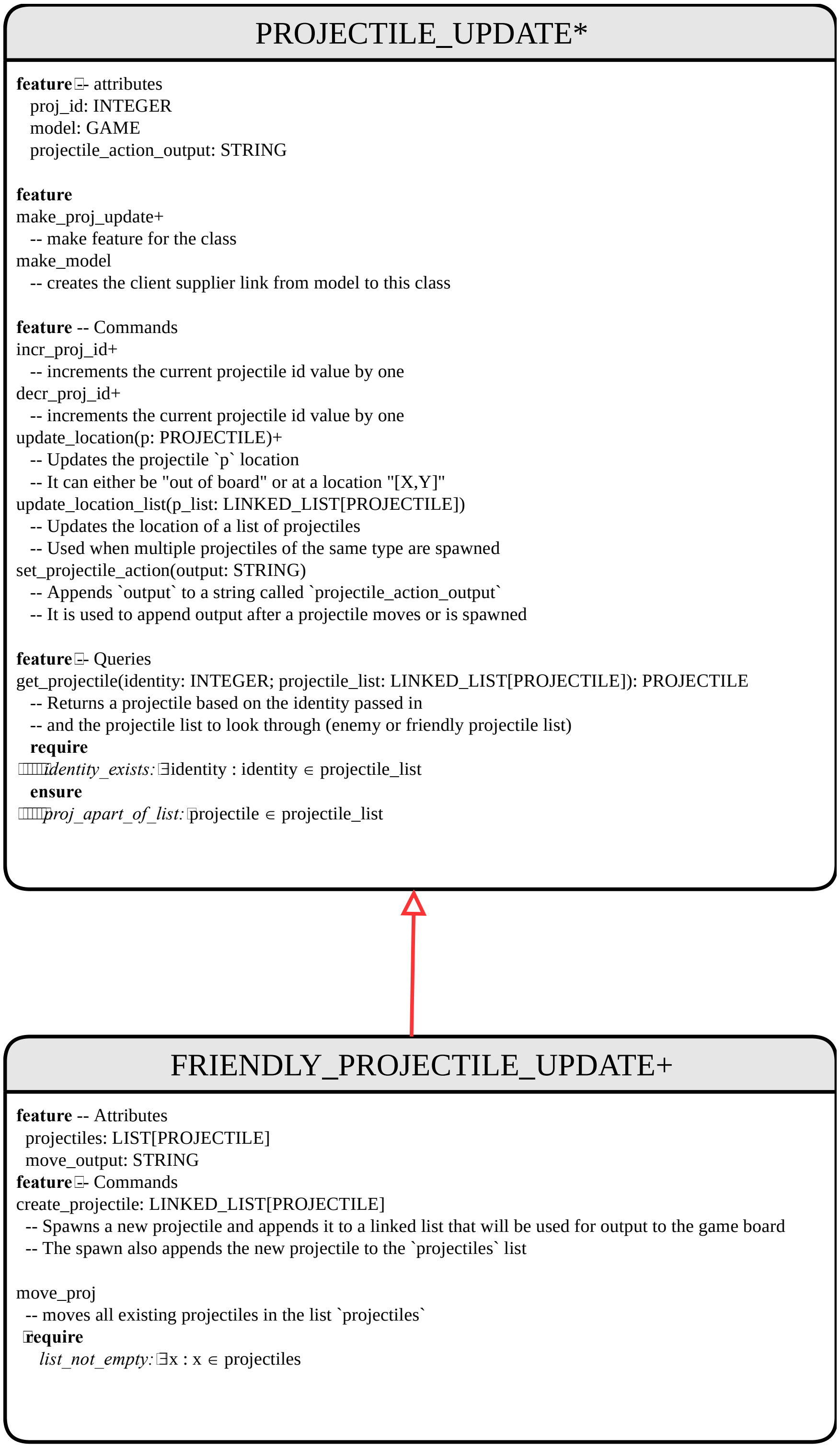
Alborz Gharabaghi
216442428
3311 Project Report



Alborz Gharabaghi
216442428
3311 Project Report



PROJECTILE_UPDATE*

feature -- attributes
proj_id: INTEGER
model: GAME
projectile_action_output: STRING

feature
make_proj_update+
-- make feature for the class
make_model
-- creates the client supplier link from model to this class

feature -- Commands
incr_proj_id+
-- increments the current projectile id value by one
decr_proj_id+
-- increments the current projectile id value by one
update_location(p: PROJECTILE)+
-- Updates the projectile \`p\` location
-- It can either be "out of board" or at a location "[X,Y]"
update_location_list(p_list: LINKED_LIST[PROJECTILE])
-- Updates the location of a list of projectiles
-- Used when multiple projectiles of the same type are spawned
set_projectile_action(output: STRING)
-- Appends \`output\` to a string called \`projectile_action_output\`
-- It is used to append output after a projectile moves or is spawned

feature -- Queries
get_projectile(identity: INTEGER; projectile_list: LINKED_LIST[PROJECTILE]): PROJECTILE
-- Returns a projectile based on the identity passed in
-- and the projectile list to look through (enemy or friendly projectile list)
require
identity_exists: ∃identity : identity ∊ projectile_list
ensure
proj_apart_of_list: projectile ∊ projectile_list

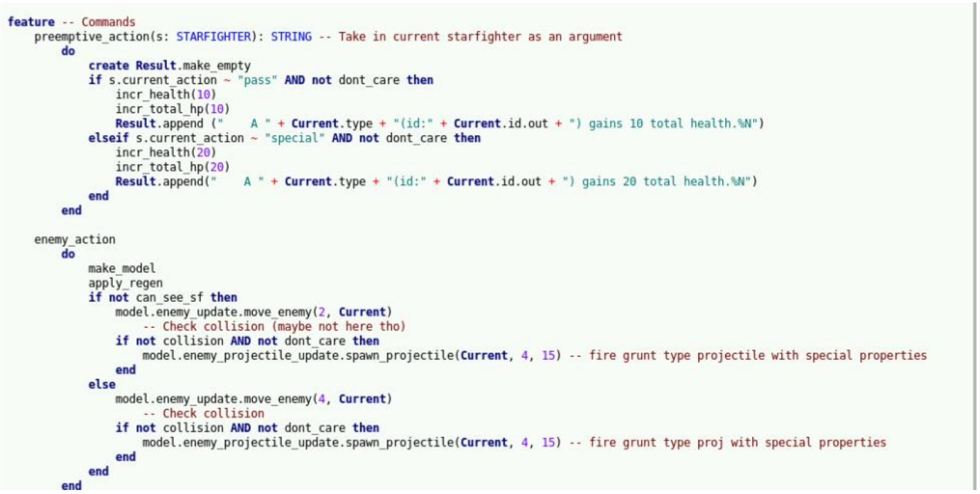
FRIENDLY_PROJECTILE_UPDATE+

feature -- Attributes
projectiles: LIST[PROJECTILE]
move_output: STRING
feature -- Commands
create_projectile: LINKED_LIST[PROJECTILE]
-- Spawns a new projectile and appends it to a linked list that will be used for output to the game board
-- The spawn also appends the new projectile to the \`projectiles\` list

move_proj
-- moves all existing projectiles in the list \`projectiles\`
require
list_not_empty: ∃x : x ∊ projectiles

Alborz Gharabaghi
216442428
3311 Project Report

In phase 5 of a turn, here is how my enemies perform pre-emptive and non-pre-emptive actions. First, I have a deferred class called ENEMY, where both pre-emptive and non-pre-emptive actions are deferred. These two deferred features are to be implemented by each type of enemy, including Grunt, Fighter, Interceptor, Carrier, and Pylon. Here's an image of the Grunt's pre-emptive and non-pre-emptive actions:



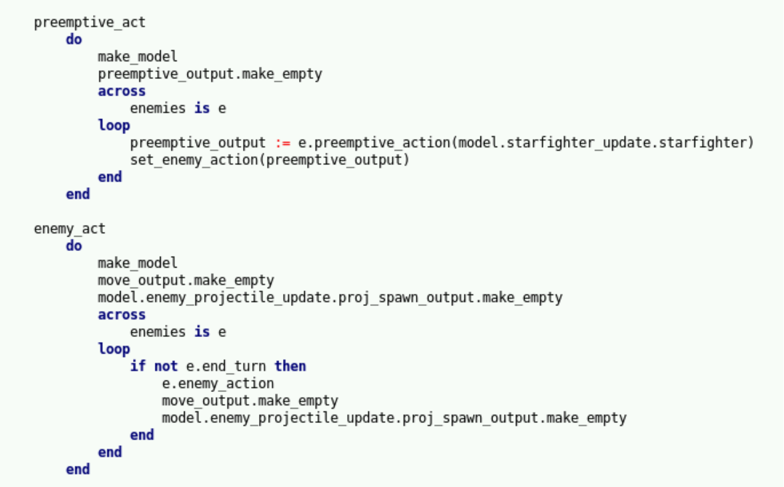
```
feature -- Commands
    preemptive_action(s: STARFIGHTER): STRING -- Take in current starfighter as an argument
        do
            create Result.make_empty
            if s.current_action ~ "pass" AND not dont_care then
                incr_health(10)
                incr_total_hp(10)
                Result.append ("    A " + Current.type + "(id:" + Current.id.out + ") gains 10 total health.%N")
            elseif s.current_action ~ "special" AND not dont_care then
                incr_health(20)
                incr_total_hp(20)
                Result.append("    A " + Current.type + "(id:" + Current.id.out + ") gains 20 total health.%N")
            end
        end

    enemy_action
        do
            make_model
            apply_regen
            if not can_see_sf then
                model.enemy_update.move_enemy(2, Current)
                    -- Check collision (maybe not here tho)
                if not collision AND not dont_care then
                    model.enemy_projectile_update.spawn_projectile(Current, 4, 15) -- fire grunt type projectile with special properties
                end
            else
                model.enemy_update.move_enemy(4, Current)
                    -- Check collision
                if not collision AND not dont_care then
                    model.enemy_projectile_update.spawn_projectile(Current, 4, 15) -- fire grunt type proj with special properties
                end
            end
        end
```

As you can see, the pre-emptive action returns a string which will be specific to the pre-emptive action of the grunt. The enemy action is also implemented, and it makes a call to the model enemy_projectile_update spawn feature, where it will spawn an enemy projectile with the specified movement and damage numbers. The same goes for every other enemy type, this is a great demonstration of dynamic binding and polymorphism as well as inheritance going on, we defer a general feature and have it implemented in different ways for each enemy type. We can also see that the class if quite cohesive as everything that is related to a Grunt and its actions are the only thing present in the class.

In terms of when the actions are called, during a pass, move, special or fire command from the user, when it comes to an enemie's actions a call is made to enemy_update.preemptive_act and enemy_update.enemy_act in order. We can see in the following image how these features are implemented:

Alborz Gharabaghi
216442428
3311 Project Report

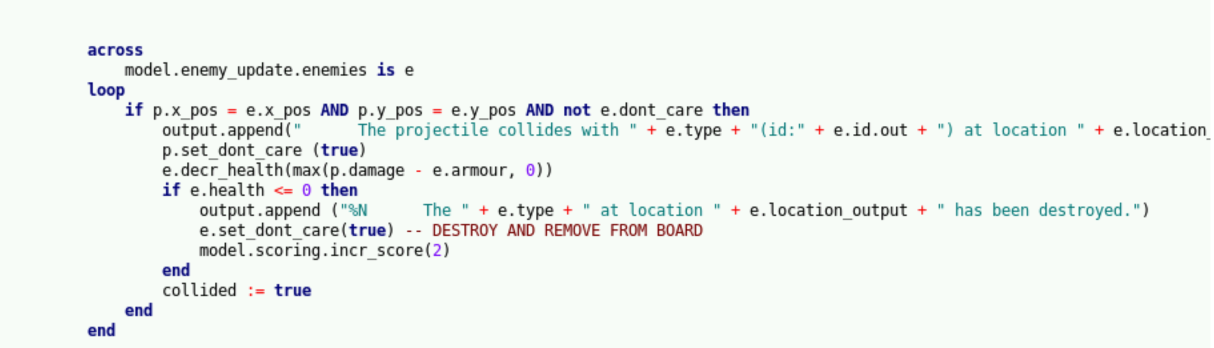


```
preemptive_act
    do
        make_model
        preemptive_output.make_empty
        across
            enemies is e
        loop
            preemptive_output := e.preemptive_action(model.starfighter_update.starfighter)
            set_enemy_action(preemptive_output)
        end
    end

enemy_act
    do
        make_model
        move_output.make_empty
        model.enemy_projectile_update.proj_spawn_output.make_empty
        across
            enemies is e
        loop
            if not e.end_turn then
                e.enemy_action
                move_output.make_empty
                model.enemy_projectile_update.proj_spawn_output.make_empty
            end
        end
    end
```

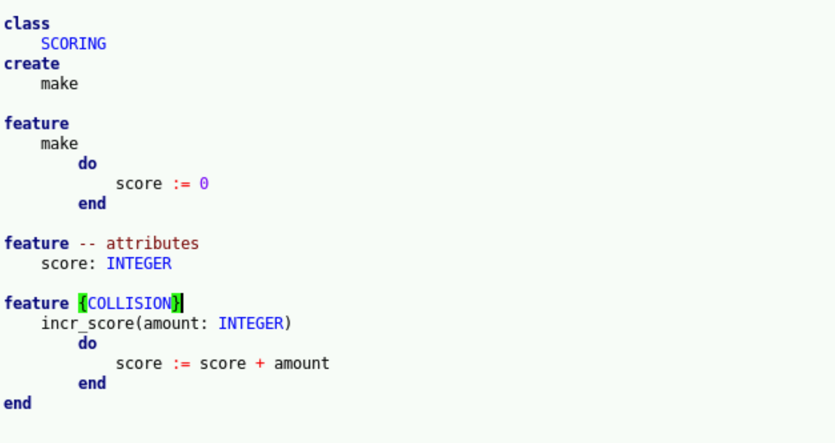
We will use an across loop to go through every enemy that is currently in the game, and call its respective pre-emptive action; this goes the same for a non-pre-emptive action. Every time an enemies pre-emptive action is finished, it's output is appending to a `preemptive_output` STRING, where that will be outputted when in debug mode. There isn't much programming from the interface going on here, but there is polymorphism as every enemy `e` is type ENEMY, and calling e.preemtpive_action will call the dynamic type of the enemy `e` pre-emptive action. The class ENEMY_UPDATE which has the code in it above is cohesive as it deals with updating the enemies actions, which in this case pertains to performing two types of actions. Also if we plan on changing how the preemptive_act feature operates, we will only have to change it in ENEMY_UPDATE since that's the only place where the code exists.

For how scoring of the Starfighter works, it operates in the following way. I made a class called `SCORING`, where it keeps track of a score integer that is increased or decreased based on what enemies are killed. The way we know how much to increase or decrease this value is reliant on my COLLISIONS class, which deals with entities dying (thus giving score).



```
across
    model.enemy_update.enemies is e
loop
    if p.x_pos = e.x_pos AND p.y_pos = e.y_pos AND not e.dont_care then
        output.append("      The projectile collides with " + e.type + "(id:" + e.id.out + ") at location " + e.location_
        p.set_dont_care (true)
        e.decr_health(max(p.damage - e.armour, 0))
        if e.health <= 0 then
            output.append ("%N      The " + e.type + " at location " + e.location_output + " has been destroyed.")
            e.set_dont_care(true) -- DESTROY AND REMOVE FROM BOARD
            model.scoring.incr_score(2)
        end
        collided := true
    end
end
```

The above code fragment is from my COLLISION class, the following situation deals with seeing if a friendly projectile `p` has collided with any type of enemy `e`. If the collision results in the death of an enemy, we do model.scoring.incr_score(2). The following call refers to the singleton instance of model which contains the SCORING class in the form of a client supplier relation ship, and we simply increase the score value inside of SCORING by an integer amount; in this case 2 for killing a Grunt. This abides by single choice principle since if we ever want to change the type of the `score` value in scoring or the incr_score feature, we ONLY have to do it in the SCORING class, not inside of COLLISION where there might be multiple calls of incr_score. Cohesion is present in the SCORING class because it contains nothing but a `score` integer as well as ways to increment it and decrement it.



```
class
    SCORING
create
    make

feature
    make
        do
            score := 0
        end

feature -- attributes
    score: INTEGER

feature {COLLISION}
    incr_score(amount: INTEGER)
        do
            score := score + amount
        end
end
```

As we can see, it is quite cohesive. We can also see some information hiding going on as we restrict the feature where we incr_score to the COLLISION class only, in order to avoid any other classes calling it when they have no business doing so. The COLLISION class is solely responsible for changing the score.