2020

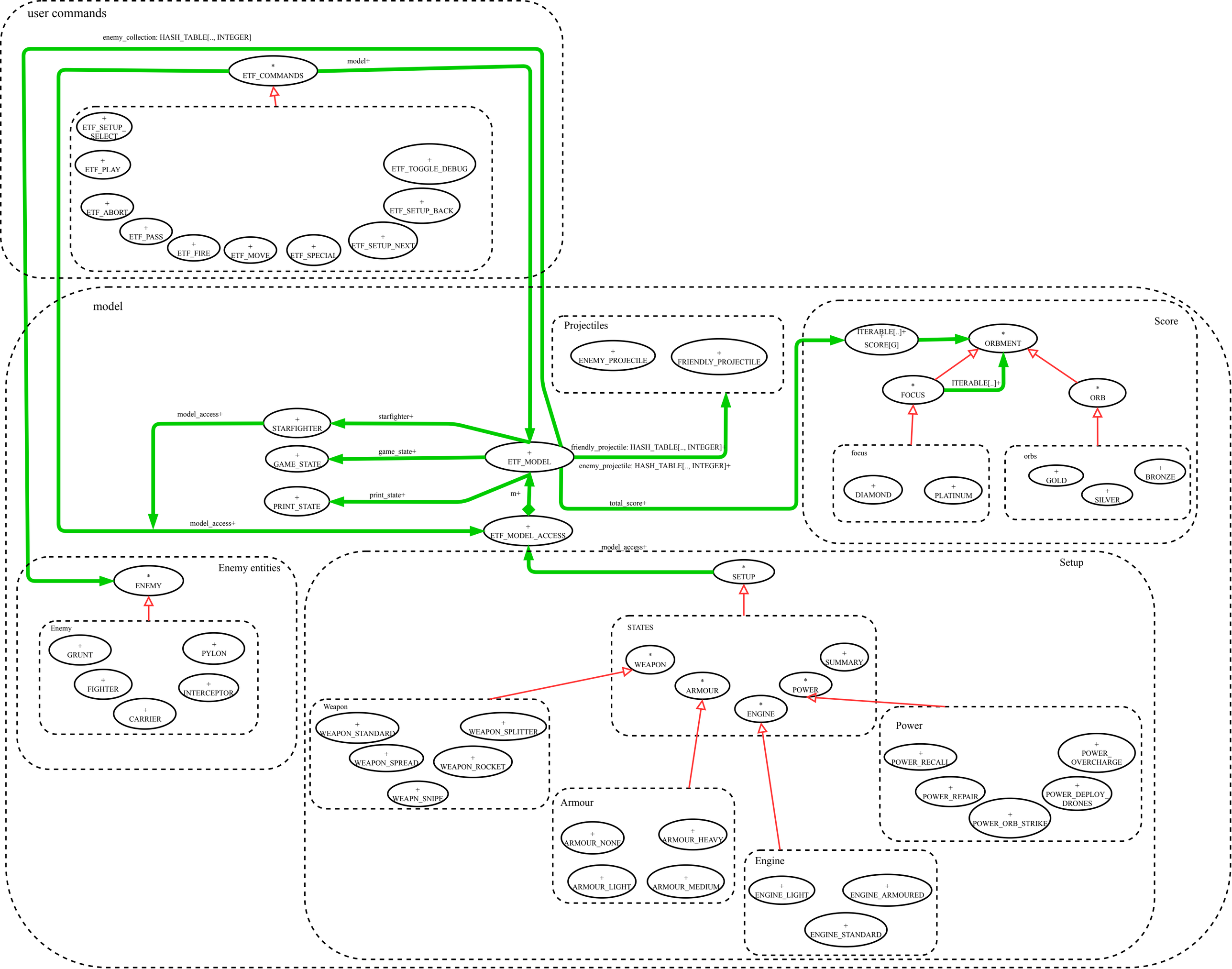
EECS 3311 – A

FINAL PROJECT – FALL 2020

SHIVANG PATEL

EECS LOGIN: - PSHIV11

TITLE: - SPACE DEFENDER 2


user commands
enemy_collection: HASH_TABLE[.., INTEGER]
* ETF_COMMANDS
model+
+ ETF_SETUP_ SELECT
+ ETF_PLAY
+ ETF_ABORT
+ ETF_PASS
+ ETF_FIRE
+ ETF_MOVE
+ ETF_SPECIAL
+ ETF_SETUP_NEXT
+ ETF_SETUP_BACK
+ ETF_TOGGLE_DEBUG
model
Projectiles
+ ENEMY_PROJECILE
+ FRIENDLY_PROJECTILE
Score
ITERABLE[..]+
+ SCORE[G]
* ORBMENT
* FOCUS
ITERABLE[..]+
* ORB
focus
+ DIAMOND
+ PLATINUM
orbs
+ GOLD
+ BRONZE
+ SILVER
model_access+
+ STARFIGHTER
starfighter+
+ GAME_STATE
game_state+
+ PRINT_STATE
print_state+
+ ETF_MODEL
friendly_projectile: HASH_TABLE[.., INTEGER]+
enemy_projectile: HASH_TABLE[.., INTEGER]+
m+
total_score+
model_access+
+ ETF_MODEL_ACCESS
model_access+
Enemy entities
* ENEMY
Enemy
+ GRUNT
+ PYLON
+ FIGHTER
+ INTERCEPTOR
+ CARRIER
Setup
* SETUP
STATES
* WEAPON
* ARMOUR
* ENGINE
* POWER
+ SUMMARY
Weapon
+ WEAPON_STANDARD
+ WEAPON_SPLITTER
+ WEAPON_SPREAD
+ WEAPON_ROCKET
+ WEAPN_SNIPE
Armour
+ ARMOUR_NONE
+ ARMOUR_HEAVY
+ ARMOUR_LIGHT
+ ARMOUR_MEDIUM
Engine
+ ENGINE_LIGHT
+ ENGINE_ARMOURED
+ ENGINE_STANDARD
Power
+ POWER_RECALL
+ POWER_ OVERCHARGE
+ POWER_REPAIR
+ POWER_DEPLOY_ DRONES
+ POWER_ORB_STRIKE

ETF_MODEL+

feature -- supplier attributes

starfighter: STARFIGHTER
total_score: SCORE
game_state: GAME_STATE
print_state: PRINT_STATE

feature -- collections
enemy_collection: HASH_TABLE[ENEMY, INTEGER]
enemy_projectile_collection: HASH_TABLE[ENEMY_PROJECTILE, INTEGER]
friendly_projectile_collection: HASH_TABLE[FRIENDLY_PROJECTILE, INTEGER]
states: ITERABLE[STATE]

feature-- ids
projectile_id, enemy_id: INTEGER
score: INTEGER

feature -- booleans
in_game, in_setup_mode, in_debug_mode: 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
-- SF passes

fire
do
states[weapon_choice].fire
-- 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

feature -- enemy releated features
report_enemies
-- reports all enemies that are still `on_board`

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

preemptive_action(str: STRING)
-- phase 5
-- preemptive action of all enemies `on_board`
-- oldest to newest

action
-- phase 5
-- Enemy action of all enemies `on board`

feature -- queries
enemy_presence [row: INTEGER; column: INTEGER]: INTEGER
-- returns an `id` of an on_board enemy at location [row, column]

enemy_projectile_presence [row: INTEGER; column: INTEGER]: INTEGER
-- returns an `id` of an on_board enemy projectile at location [row, column]

friendly_projectile_presence [row: INTEGER; column: INTEGER]: INTEGER
-- returns an `id` of an on_board friendly projectile at location [row, column]

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

invariant
contradiction:
(in_game and in_setup_mode) = false

starfighter+

STARFIGHTER+

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

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

feature -- model access
model_access: ETF_MODEL_ACCESS

feature -- queries
seen_by_sf (row: INTEGER; column: INTEGER)
-- can starfighter the position [row, column]

feature -- commands

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

add_health (h: INTEGER)
-- adds h to current health

add_energy (e: INTEGER)
-- adds e to current energy

subtract_health (h: INTEGER)
-- subtracts h from current health

subtract_energy (e: INTEGER)
-- subtract e from current energy

apply_health_regen
-- health regeneration of starfighter

apply_energy_regen name_of_attribute
-- energy regeeration of starfighter

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

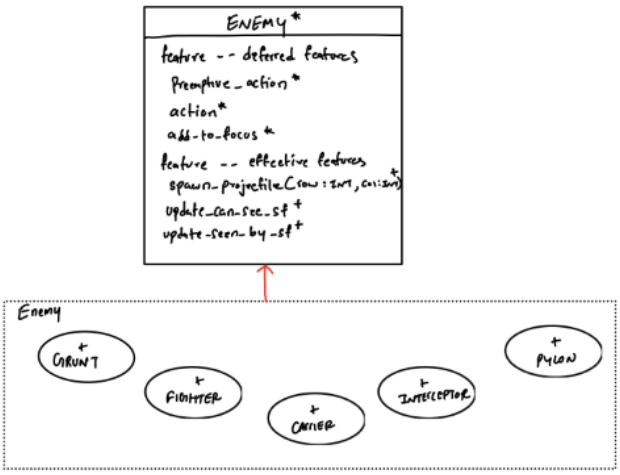
set_old_pos[row: INTEGER; column: INTEGER]
-- updates the starfighter old_pos to [row, column]

invariant
destroyed:
current_health <=0 implies (model.in_game = false)

1. Section: Enemy Actions

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

   ➔ All the enemies perform some preemptive and/or non-preemptive actions based on the command used by STARFIGHTER during the game.



The implementation if the Enemy and its actions are based on the structure shown in the figure above. The features `preemptive action` and `action` are the deferred features with the effective definition given in all its descendant classes based on the respective rules for each enemy in the project requirement. We store `ENEMY` in our ETF_MODEL using the collection class HASH_TABLE [ENEMY, INTEGER] with INTEGER being the `id` of an enemy. Therefore, the HASH_TABLE is the polymorphic collection of ENEMIES. At runtime, we use dynamic binding to perform enemy action based on the retrieved enemy from oldest to newest. Below is the code fragment that is used to retrieve enemies during phase 5.

```
preemptive_action(str: STRING)
    do
        across 1 |..| current.enemy_collection.count is id
        loop
            if attached current.enemy_collection.item (id) as obj and then in_game then
                if attached {ENEMY} obj as e_obj then
                    if e_obj.on_board then
                        e_obj.preemptive_action(str)
                    end

                end

            end

        end
    end

action
    do
        across 1 |..| current.enemy_collection.count is id
        loop
            if attached current.enemy_collection.item (id) as e_obj and then in_game then

                if e_obj.on_board and (not e_obj.end_turn) then
                    e_obj.action
                end

            end

        end
    end
```

Since the enemies have unique IDs associated with them, they can be retrieved with O(1) time complexity from HASH_TABLE. When performing for example, *enemy_obj.preemptive_action("pass"),* the version of `preemptive_action` being executed at runtime is based on the dynamic type if `enemy_obj`. Therefore at runime, we do not have to explicitly cast the retrieved `enemy_obj` to check the type of enemy before using its preemptive action. This helps satisfy the single choice principle by avoiding the unnecessary if…then…else…end statements. How enemy performs its non-preemptive action is based on if its turn ends during the preemptive action or not. All the enemies whose turn does not ends performs the non-preemptive action same as the preemptive action. We use a Boolean variable `end_turn` to keep track of enemies who may or may not end the turn during the preemptive action. This variable is modified using `set_turn (b:BOOLEAN)` feature in the `ENEMY` class thus and its called on the enemy based on its dynamic type at runtime.
After all the enemies performs its preemptive and non-preemptive action we have to reset the `end-turn` variable for it to be usable in next turn of a game. This is done in ETF_MODEL class when all the enemies who are alive are reported at the end of each turn. See code fragment below.

```
report_enemies
    do
        print_state.set_enemy_empty
        across 1 |..| current.enemy_collection.count is id
        loop
            if attached current.enemy_collection.item (id) as obj and then obj.on_board then
                print_state.set_enemy (obj.report_status)
                obj.set_turn(false)

            end


        end
    end
```

Certain enemy's spawn's spawn `enemy projectile` during their action. This feature is implemented at the root level in ENEMY class where the passed projectile is spawned by the respective enemy. thereby avoiding code duplication and maximizing reusability.

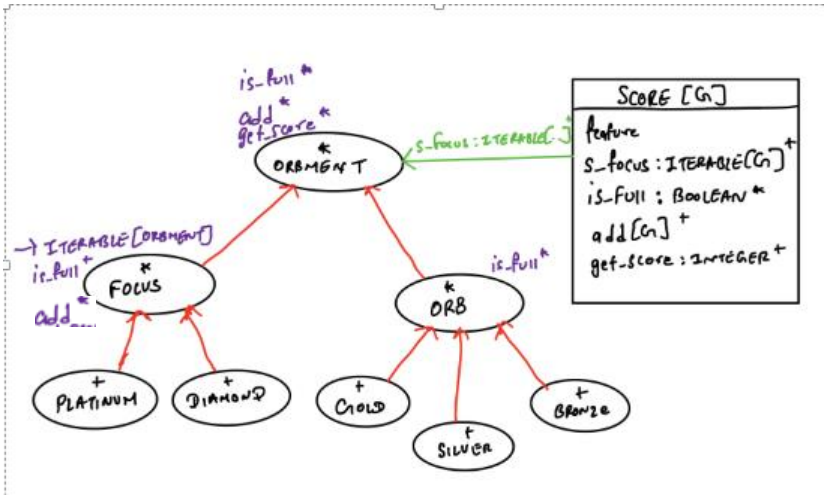
```
spawn_projectile(projectile: ENEMY_PROJECTILE)
    do
        -- spawns projectile

    end
```

The entire ENEMY cluster is designed using the principle based on separation of concern where only the preemptive and non-preemptive actions are defined in the child classed based on their different behaviors whereas spawn_projectile (projectile: ENEMY_PROJECTILE) is implement at root level that can be used by all its descendant classes. This helps satisfy the Cohesion Principle.

2. Section: SCORING of Starfighter

How the scoring of the Starfighter works

➔ We have used Composite design pattern in this project for scoring purpose. The structure of the scoring is shown in the diagram below.



We use recursive nature to calculate the score. As per the requirement the ITERABLE[ORBMENT] in FOCUS is a linear container of fixed size. The features `add` and `is_full: BOOLEAN` are implemented in each of the effective class and behaves differently. The linear container in the FOCUS can contain FOCUS itself (which contains another linear container, thus recursive in nature) or an ORB of type GOLD, SILVER or BRONZE where the `is_full` is true since they being a base case. The feature `is_full` in focus is implemented using the code fragment shown below.

```
is_full: BOOLEAN
    do
        if array.count = max_size then
            -- means we can safely iterate over the array to check if_full on each element
            result :=
                across 1 |..| max_size is index
                all
                    array[index].is_full

                end
        else
            result := false
        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 s_focus is_empty then

Add the ORBMENT at (count + 1)

Elseif s_focus(count) is not full then

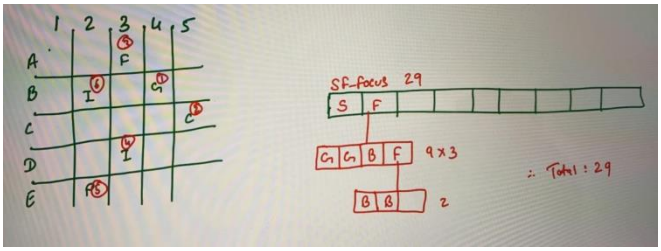
S_focus[count].add(o: ORBMENT)

- *This is a recursive call to the add feature implemented in FOCUS*
- *If it's a base case (i.e GOLD, SILVER, BRONZE) the is_full is always true*

Else

ADD the ORBMENT at (count + 1)

This helps full up the left subtree before adding an element to the next position of s_focus. The example blow follows the algorithm we mentioned to add the enemies being destroyed according to the integer values show in the matrix The right side of figures contains the starfighter focus where S,G and B corresponds to SILVER, GOLD and BRONZE respectively and F corresponds to the enemy focus of either type DIAMOND or PLATINUM. This enemy focus, if not full, is filled before the next element is added to the s_focus.



The enemies when destroyed, may drop an ORB or FOCUS based on its dynamic type. We have implemented the `*add_to_focus*` as a deferred feature in ENEMY class where its effective implementation is show its descendant classes. An example of add_to_focus from GRUNT is shown in the code fragment below

```
add_to_focus
    local
        silver: ORB
    do
        create {SILVER} silver.make
        model.m.total_score.add (silver)
    end
```

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. Since we use ITERABLE[G] in Score component, it can be initialized to any of its descendant classes thereby satisfying Programming from interface 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.

Steps below shows the entire routine of how Scoring works at runtime.

- Feature ***`enemy_projectile_presence(row: INTEGER; column: INTEGER): INTEGER`*** in **ETF_MODEL** is used to retrieve an `**enemy id**` at runtime whenever the collision occurs at **[row, column]** (location).
- `**enemy id**` is used to retrieve an enemy object from **HASH_TABL**E since `**enemy id**` being the `**key**` of enemy object.
- The retrieved enemy, say enemy_object is set_to_off board if its health drops to 0 or below and **enemy_obj.add_to_focus** is executed during that phase.

Based on the dynamic type of **enemy_object**, the corresponding version of **add_to_focus** is called and the appropriate ORBMENT is added to Starfighter FOCUS.

Note: DIAMOND on its creation adds an ORB of type GOLD in its constructor. Similarly, PLATINUM adds BRONZE. Therefore linear container in FOCUS is never empty

The base ORBs of type GOLD, SILVER and BRONZE also contains an integer value points (3, 2, 1 respectively) which is later used in the calculation of the score. The score is recursively calculated using the code fragment shown below.

```
calculate: INTEGER
    do
        across 1 |..| s_focus.count is index
        loop
            Result := result + s_focus[index].get_score
        end
    end
```

^ *Version from class SCORE*

```
get_score: INTEGER

    do
        across 1 |..| array.count is index
        loop
            result := result + array[index].get_score
        end

        if array.count = max_size then
            result := result*3
        end
    end
```

^ Version from Class DIAMOND of type FOCUS

```
    make
            -- Initialization for `Current'.
        do
            points := 1
        end

    get_score: INTEGER
        do
            result := points
        end
```

^ Version from class BRONZE of type ORB

Note: A Multiplier is also used as per the description given in the project requirement