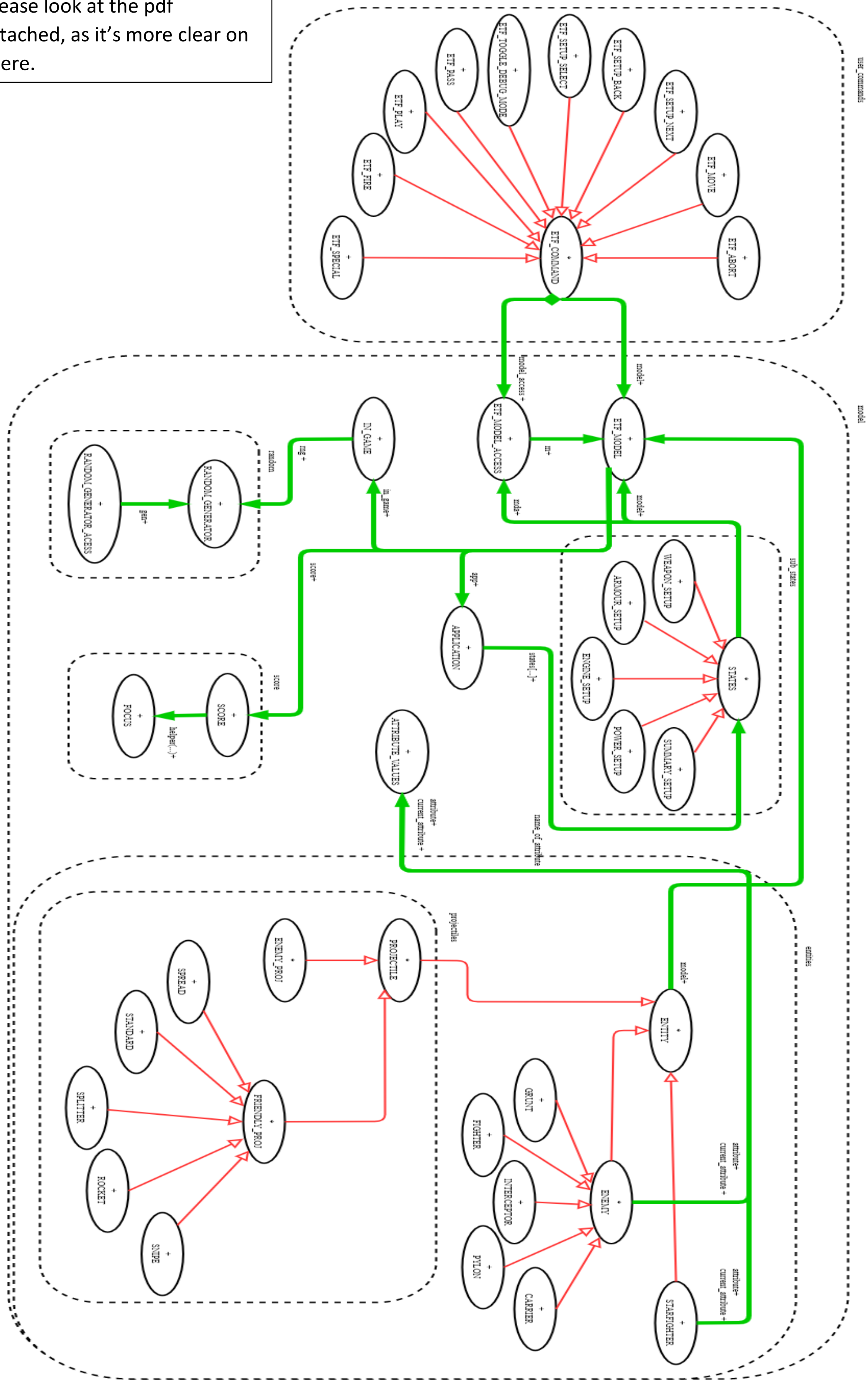
Course: EECS 3311 – Software Design

Semester: Fall 2020 – 2021

Name: Krishaanth Manoharan

EECS Prism Login: krish100

Compact View BON
Please look at the pdf attached, as it's more clear on there.



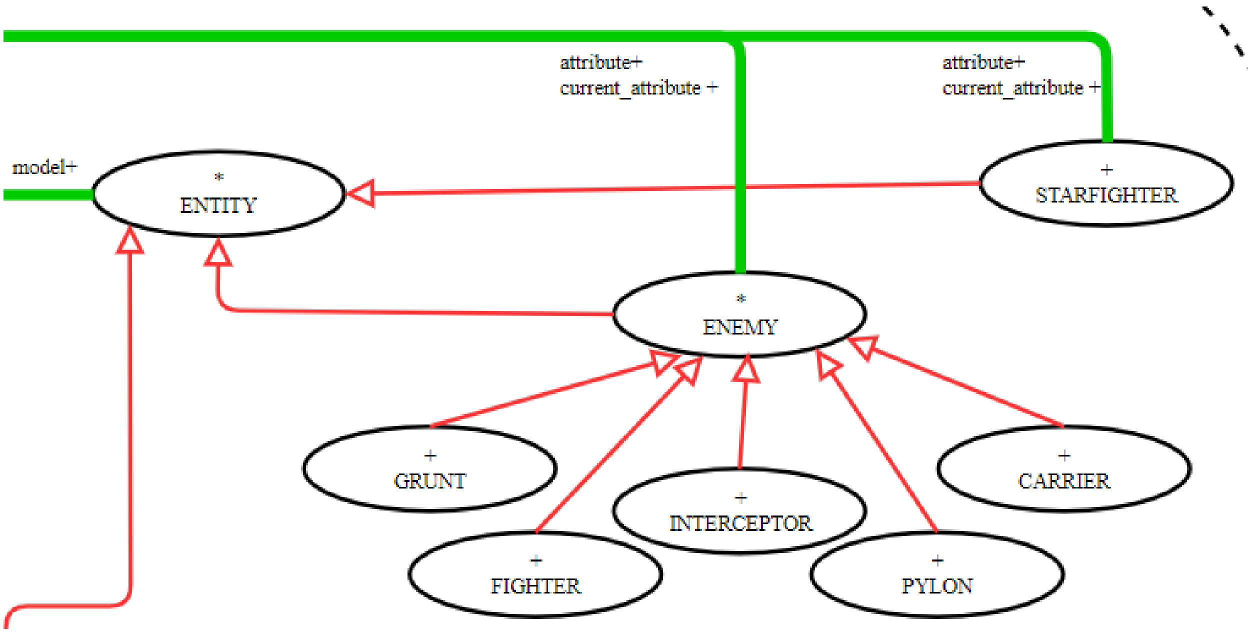
2 Detailed View BON

```
ENEMY*

feature -- Attributes
  attributes : ATTRIBUTE_VALUES + -- original values
  current_attributes : ATTRIBUTE_VALUES -- current values

feature -- Deferred
  execute *
  preemptive_aciton *
  seen *
  not_seen *

feature -- Commands

  regen +
    require
      enemy_still_alive: Current.alive = True
                          ^
                          Current.current_attributes.health > 0
    -- regens current enemy
  move (distance: INTEGER) +
     require
       enemy_not_outside_board: Current.outside_board = False
    -- moves <distance> to the left from current location
  collision ++
    -- redefined from ENTITY, used for collision checking
   modify_collision +
    -- effective method, inherited from ENTITY, used for modifying collision values

feature {NONE}-- Hidden

   can_see_starfighter: BOOLEAN +
     -- returns true if current enemy can see starfighter, else false
   seen_by_starfighter: BOOLEAN+
     -- returns true if starfighter can see current enemy, else false

feature -- Queries

   stats_out: STRING +
     -- returns string of current enemies stats
```

```
GRUNT+

feature -- Commands

  execute
    -- Executes grunt actions
  preemptive_action
   -- Does grunt's preemptive actions

feature {NONE}-- Hidden

  seen +
    require else
       starfigher_is_visible : can_see_starfighter = true
    -- Helper method for execute, does action when can see starfighter
  not_seen +
    require else
       starfigher_is_not_visible : not can_see_starfighter = true
    -- Helper method for execute, does action when cannot see starfighter
```

Section: Enemy Actions

Quick recap of enemy structure:



Enemy is a deferred class that inherits from Entity (another deferred class). It inherits basic things such as collision, location mutator and accessor methods. As well as queries such as checking whether the current entity is outside the board, and printing the location in string format, etc.

Deferred Enemy class, declares 4 deferred methods/commands:

- execute
    - This is for executing enemy actions, which we will further go more in depth.
- seen
    - This command acts as a helper method for execute, and it's dependent on how a specific enemy type gets effected when it *sees* Starfighter.
- not_seen
    - This command acts as a helper method for execute, and it's dependent on how a specific enemy type gets effected when it *doesn't see* Starfighter.
- preempetive_action
    - This command is used for preemptive actions which we will further go more in depth.

Deferred Enemy class, also implements further methods such as:

- regen (to regenerate health); Each effective child sets its regen in its attributes object, so we take that regen and apply it to the current enemy.
- move (to move across the grid/board); since all enemies move from right to left on the board (except for Interceptor – can move vertically), it was decided to create a dedicated move method that all enemies can use. For interceptor we instead redefined move for its own fitting.
- collision ; this method was inherited from Entity but was redefined due to enemies not being able to collide with other enemies, and stops 1 space before.

- modify_collision ; this is a deferred command from Entity which was implemented in Enemy class, this acts as a helper method to collision, as this method provides what to specifically do when e.g enemy collides with friendly projectile or enemy projectile etc. Uses dynamic binding.
- can_see_starfighter : this query method returns a Boolean, true whether the current enemy can see the starfighter or false when it can't see starfighter due to vision
- seen_by_starfighter : this query method returns a Boolean, true whether starfighter can see the current enemy or false when starfighter cannot see the current enemy due to starfighters' vision.

There are also other query methods which also help in outputting toggle messages, such as printing all the current stats of the enemy.

Phase 5: How Preemptive Actions work

Earlier we mentioned the basic structure of an enemy. Now we will look at pre-emptive actions of an enemy.

<u>Case1: Pre-emptive actions when turns DON'T end</u>

We will be demonstrating this using grunt.

```
    preemptive_action
        do
            if alive and not outside_board then
                -- if sf passes, increase both hp and total hp by 10
                if model.command_msg.is_equal("pass") then
                    pass
                -- if sf special, increase both hp and totalhp by 20
                elseif model.command_msg.is_equal("special") then
                    special
                end
            end
            -- TURN DOESNT END FOR BOTH^
        end
```

In the above code snippet we first check if grunt is alive and not outside of the board for safety. We then check whether the user used pass method or special (as these two methods cause pre-emptive for grunt). Similarly other enemies check for their own pre-emptive commands.

The way these commands get passed is from ETF_MODEL, when a user inputs a command the command_msg string is updated with e.g pass or special. Another way this could have been done is by creating pass/special/etc objects, then we can do:

e.g if attached {PASS} then do preemptive action end

It was decided not to take this approach as in the long run we will still be using if statements to compare, and using objects would in theory take up more space whilst strings will be for the most part a constant O(1) space complexity.

The following snippet is found in_game state class:

```
    execute_preemptive_enemy_actions
        local
            model : ETF_MODEL
        do
            model := mda.m
            across
                model.enemies is enemy
            loop
                if enemy.alive and not enemy.outside_board  then
                    enemy.preemptive_action
                end
            end
        end
```

Going back to the in_game state, this is where we initially call preemptive actions, we simply use dynamic binding to our advantage and call preempetive_action on every single enemy that is on the board.

The following snippet is found in in_game state class:

```
    game_update
        local
            model : ETF_MODEL
        do
            model := mda.m
            model.state_increase

            execute_friendlies -- phase 1 move friendly projectiles first
            execute_enemy_projs -- phase 2 enemy projs
            execute_preemptive_enemy_actions -- preemptive actions, done before starfighter
            execute_starfighter -- phase 3 starfighter action
            execute_enemies -- phase 5 enemy act (combined with 4 & 6 enemy vision phase)
            enemy_spawn -- phase 7 enemy spawn

```

This is where we call pre-emptive enemy actions before starfighters' act. And this game_update method is called from each ETF command that is related to ingame actions.

Case2: Pre-emptive actions when turns DO end

```
    execute
        do
            if not model.command_msg.is_equal ("pass") then -- ENDS TURN ON PASS
                regen

                if can_see_starfighter then
                    seen
                else
                    not_seen
                end
            end
        end
```

This above snippet is taken from enemy, Fighter. Here in execute we have an if statement where it doesn't execute the seen and not seen of fighter whenever user inputs pass (which ends fighter's turn in preempetive).

Whereas Grunt does not have that if statement, so regen will be done in execute. Whilst Fighter has 2 regen locations, one in execute and one in the helper method pass, as shown below:

```
    pass
        local
            proj : ENEMY_PROJ
        do
            regen -- REGEN**** BECAUSE TURN ENDS HERE
            -- if onboard and not dead:
            move(6) -- 6 left
            if not outside_board then
                current_attributes.set_projectile_dmg (100) -- 100 base dmg
                proj := create {ENEMY_PROJ}.make(location.row, location.col - 1)
                proj.set_current_damage (current_attributes.projectile_dmg)
                proj.set_move_distance (10) -- moves 10 left

                proj.spawn_collision
                model.toggle_enemy_action_msg.append (proj.collision_msg)
            end
            -- END TURN*******
        end
```

So during the preemptive phase, Fighter regens wheras in the non-preemeptive phase fighter will not due to the if statement.

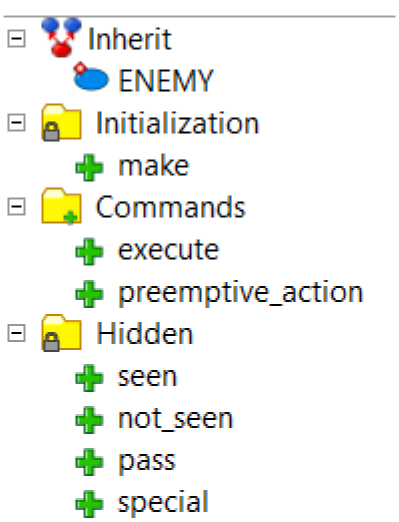
Earlier we also mentioned that can_see_starfighter is a command inherited from the deferred class ENEMY. Below you can find the code snippet:

```
    can_see_starfighter : BOOLEAN
        local
            starrow : INTEGER
            starcol : INTEGER
        do
            starrow := model.starfighter.location.row
            starcol := model.starfighter.location.col
            Result := False
            if
                (starrow - location.row).abs + (starcol - location.col).abs
                <= attributes.vision
            then
                Result := True -- True if enemy can see starfighter
            end
        end
```

Here we simply check the location of starfighter with the current enemy and check whether the distance is less then the vision. This plays a huge role into whether we enemy uses seen vs not_seen actions. As you've seen earlier, basic actions are seen and not_seen, this was execute_enemies (on page 4 pic).

Information Hiding

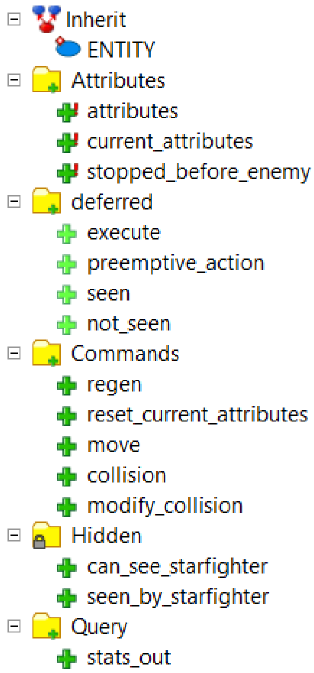
Regarding information hiding, this was attempted by setting variables and commands/queries as hidden by grouping them under a feature {NONE}.



Here we will demonstrate this using enemy, Grunt. It was decided to make seen, not_seen, pass, special as hidden. As these 4 methods act as helpers for execute. By doing so, the client cannot access these methods thus preventing from execution to get ruined.

Similarly, for something like Pylon, extra features such as for regenerating nearby enemies is also hidden. Thus, preventing from clients to access.

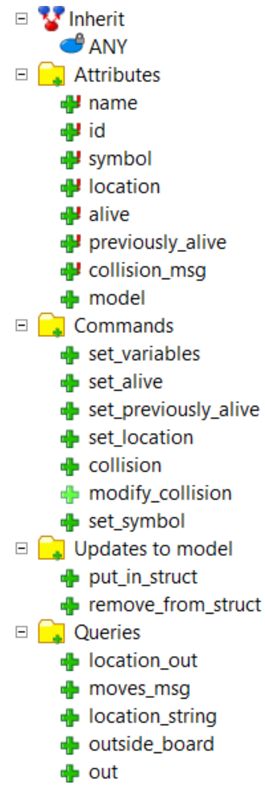
To add on, it was decided to make can_see_starfighter and seen_by_starfighter hidden from client as they don't need access to these two queries, as they act as helpers for execute.



One thing that could be improved on for next time is setting attributes hidden to client. So clients wouldn't be able to change values. The reason why this was visible is due to collision and modification of values upon impact.

execute, move, collision, modifying of collision, regen and attributes, and print queries are all visible to clients.

The screenshot on the right is taken from ENTITY, which ENEMY inherits from. Here we have everything visible as shown on the right, simarily we could have improved this by making specific attributes hidden. One way this is satisfied is due to not having setter methods for all the attributes except for collision related which had to be visible so other entities can see.



Single choice principle

Single choice principle was satisfied by having a single move method for all enemies.This move method is located in the parent deferred class, ENEMY. As we explained earlier all enemies except for Interceptor move from right to left. Interceptor redefines its own type of move method.

Another way single choice principle was satisfied was by having a single collision method which every other child entity class inherits, this goes for enemies, starfighter, and projectiles. Only enemies redefine collision due to it stopping before another enemy.
Since enemies, starfighter, projectiles etc. act differently upon collision, each of the parent classes has an effective modify_collision command where they set its own specifics.

Regen in a way also satisfies single choice principle as all enemies use the same type of regen only effecting health. And there's only 1 regen method which is found in the parent class, ENEMY.

Places where single choice principle violated was the spawning of projectiles. I could have made similarity to regen a fire method where it shoots the specified projectile. The reasoning for why I decided to put fire into each enemy child class was due to changes of the attributes of projectile damage. It was easier to implement it in a specific fire command rather than 1 as there's many attributes to be changed.

Cohesion

Cohesion for the most part was satisfied, as all methods are related to enemies. Move is dedicated to enemy along with collision, regen, and whether enemy can see starfighter.

Attributes are dependent on the current enemy thus satisfying cohesion.

Since enemy inherits from ENTITY, collision is sort of violated due to updates to model. Here we update the locations directly to model, rather a better solution could have been updating from model or another class rather then enemy.

Programming from the Interface, Not from the implementation.

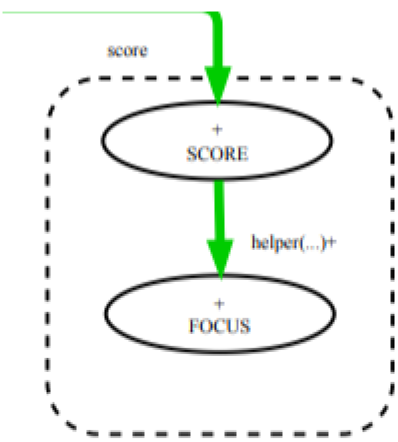
This was satisfied by using deferred classes and methods. I've made sure to make everything as similar as possible so enemy actions can be easy to understand.

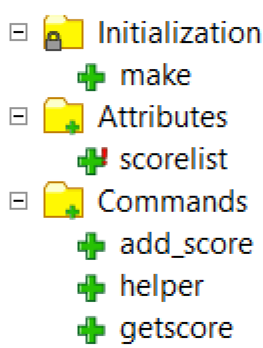
To add on, it was decided to use deferred methods such as modify_collision, preemptive_action, execute, seen, not_seen. This way it allows for clients to apply execute on all entities even when the static class is declared as Entity or Enemy.

Section: Scoring of Starfighter

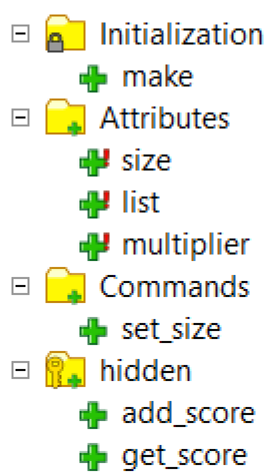
Quick recap of score structure:

Score uses recursion to add the points, points are described as orbs and focuses. Instead it was decided to use orbs as integers and created a separate class for focuses.





Above is SCORE features



Above is FOCUS features

Score makes use of a linked_list, which stores type ANY. This allows for integers and focuses to be stored into, we also make use of type casting to make sure nothing else gets inserted into score.

```
add_score(score : ANY)
    do
        if scorelist.count = 0 then
            scorelist.extend (score)

        elseif attached {INTEGER} scorelist[scorelist.count] then
            -- add to end
            scorelist.extend (score)

        elseif attached {FOCUS} scorelist[scorelist.count] as focus then
            -- go inside focus
            if attached {FOCUS} helper(focus) as returnedfocus then
                returnedfocus.list.extend(score)
            else -- focuses are full, add to end of score list
                scorelist.extend (score)
            end
        end
    end
```

Base case, when list size is 0, add either the ‘orb’ or ‘focus’ into the linkedlist, named as scorelist. If the last item added into the list was an integer, then that means we can directly add to the list. Otherwise then a focus was inserted last, here we use recursion by using a helper method named helper!

```
    helper(focus : FOCUS) : DETACHABLE focus
        do
            if focus.size = focus.list.count then -- focus is full

                if attached {INTEGER} focus.list[focus.size] then -- exit, add focus to end of scorelist
                    -- return nothing
                elseif attached {FOCUS} focus.list[focus.size] as newfocus then
                    Result := helper(newfocus) -- recursive
                end
            else -- return this focus
                Result := focus
            end
        end
```

Helper accepts the focus as a parameter then traverses inside, and goes to the ‘bottom/depth’ of the focus. Focus.size is a predefined size which determines the max size of the focus, if its equal to the size of the list in focus then that means the focus is full. We then check whether its an integer or focus at the end of the focus’ list, if its an integer then return nothing, which signifies all focuses are full thus we can extend to scorelist. Otherwise using recursion we apply helper(focus) on the current focus, till we find the last focus, or integer.

```
getscore : INTEGER
    do
        across
            scorelist is item
        loop
            if attached {INTEGER} item as l_i then
                Result := l_i + result
            elseif attached {FOCUS} item as l_f then
                if l_f.size = l_f.list.count then
                    Result := result + (l_f.get_score)*l_f.multiplier
                else
                    Result := result + l_f.get_score
                end

            end
        end
    end
```

Score also uses recursion to get the actual point integer score. We traverse the linkedlist, if an integer is found directly add to result, other wise if a focus is found, we traverse to the bottom, then apply multipliers. This uses a helper method get_score which is found in FOCUS class, as shown below:

```
get_score : INTEGER
    do
        across
            list is item
        loop
            if attached {INTEGER} item as l_i then
                Result := l_i + result
            elseif attached {FOCUS} item as l_f then
                if l_f.size = l_f.list.count then
                    Result := result + (l_f.get_score)*l_f.multiplier
                else
                    Result := result + l_f.get_score
                end
            end
        end
    end
```

Get_score, traverses focus' list and adds if its an integer, other wise go inside the focus and recall get_score and apply the multiplier.

```
make(s : INTEGER)
        -- Initialization for `Current'.
    do
        multiplier := 10 -- so i knonw if its an error
        size := s
        create list.make
        if s = 4 then
            -- create gold orb
            list.extend (3) -- worth 3
            multiplier := 3
        elseif s  = 3 then
            -- create bronze orb
            list.extend (1) -- worth 1
            multiplier := 2
        end
    end
```

This is found in FOCUS, here we set the multiplier and the 'size' upon creation. For example if a carrier gets destroyed, it gives the param 4, and creates the size of 4 and immediately adds a gold orb being integer 3. And sets multiplier to 3. Similarly, Pylon works the same way.

To add the actual scores upon enemy being destroyed is found in ENTITY, where we set if an entity is alive or not alive, show below:

```
set_alive(b : BOOLEAN)
    do
        alive := b
        if b = False then
            remove_from_struct -- removed from hashtable
            model.grid[location.row][location.col] := "_" -- removed frmo board
            previously_alive := TRUE

            -- add to scores
            if attached {GRUNT} current then
                model.score.add_score (2) -- silver
            elseif attached {FIGHTER} current then
                model.score.add_score (3) -- gold
            elseif attached {INTERCEPTOR} current then
                model.score.add_score (1) -- bronze
            elseif attached {CARRIER} current then
                model.score.add_score (create {FOCUS}.make(4)) -- automatically adds gold
            elseif attached {PYLON} current then
                model.score.add_score (create {FOCUS}.make(3)) -- automatically adds bronze
            end
        end
    end
```

Here we check if the current entity that got destroyed is an enemy type, depending on the enemy type we add the score to variable score, which was declared in ETF_MODEL.

Information Hiding

Information hiding was satisfied by making add_score and get_score hidden in FOCUS. It was set visible to only SCORE and FOCUS. Thus clients won't be able to accidently directly add to the focus, rather they should add to SCORE class.

Clients can see scorelist, add_score, helper, getscore.

Information hiding was violated due to scorelist being visible as a linkedlist, rather we should have created a setter and getter method to prevent users from extending into the list.

Single choice principle

Single choice principle was satisfied. There's no need to duplicate/similar methods as we use recursion to solve score. All methods are separate and differ, this is unlike ENEMY where some methods are similar. If a change is needed only one place will get effected.

But if more types of orbs/focuses are needed to be added changes are needed to be made only in FOCUS if a new orb type is added. And similar if a new focus size is implemented only in the make of FOCUS is changed, thus satisfying single choice principle.

Cohesion

Cohesion is satisfied as all methods are directly involved with SCORE. Similarly, class FOCUS, only has features directly effecting SCORE and FOCUS.

One place cohesion is violated in ENTITY because that's where I've set enemies getting destroyed. As said earlier, ENTITY directly adds the score to ETF_MODEL where score is declared. This violates cohesion as this should have been done in a separate class dedicated for enemy getting destroyed.

Programming from the Interface, Not from the implementation.

Programming from the interface is satisfied as we didn't need to use deferred methods as there's no places where we redefine methods.

Clients can easily use the visible commands to modify and add items to score and get the score points.