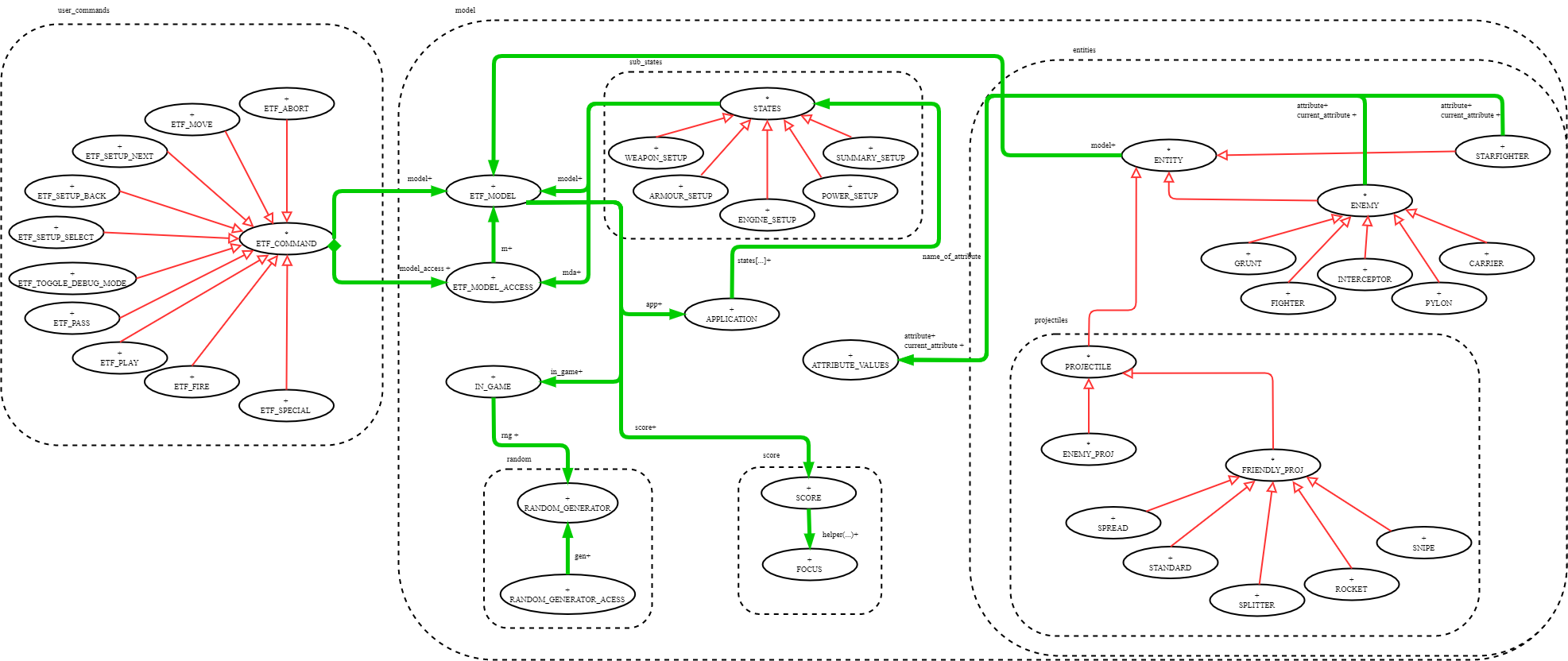
**Course:** EECS 3311 – Software Design

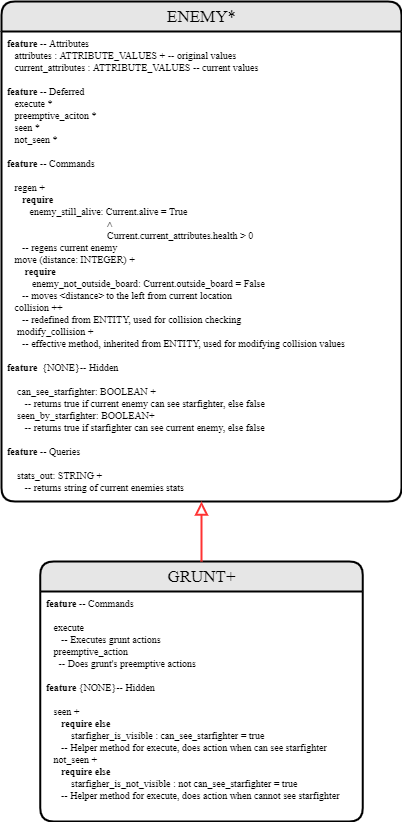
**Semester:** Fall 2020 – 2021

**Name:** Krishaanth Manoharan

**EECS Prism Login:** krish100



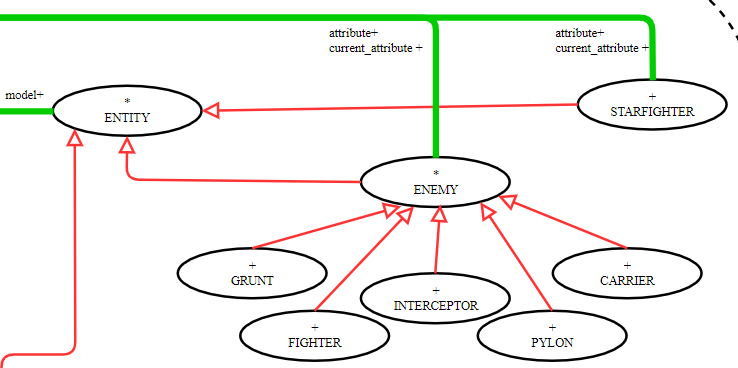
Compact View BON  
Please look at the pdf attached, as it’s more clear on there.



2 Detailed View BON

# 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.

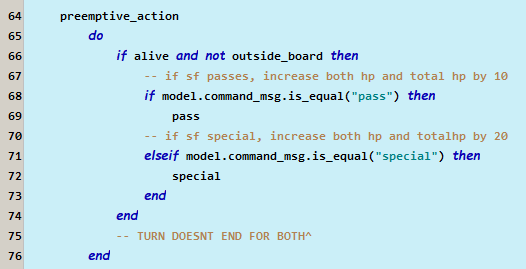
There are also other query methods which also help in outputting toggle messages, such as printing all the current stats of the enemy.

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.

Case1: Pre-emptive actions when turns DON’T end

We will be demonstrating this using grunt.



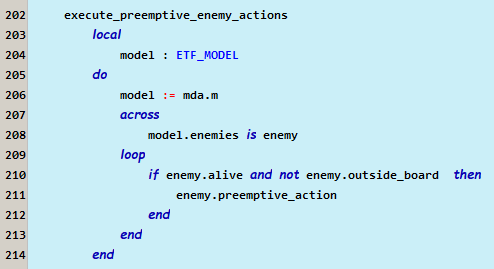
In the above code snippet we first check if grunt is alive and not outside of the board for safety. We then check whether the user used pass method or special (as these two methods cause pre-emptive for grunt). Similarly other enemies check for their own pre-emptive commands.

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

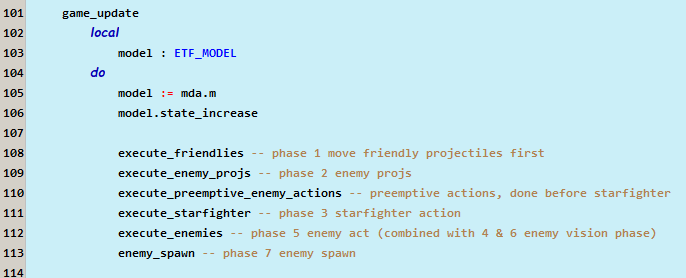
It was decided not to take this approach as in the long run we will still be using if statements to compare, and using objects would in theory take up more space whilst strings will be for the most part a constant O(1) space complexity.

The following snippet is found in\_game state class:



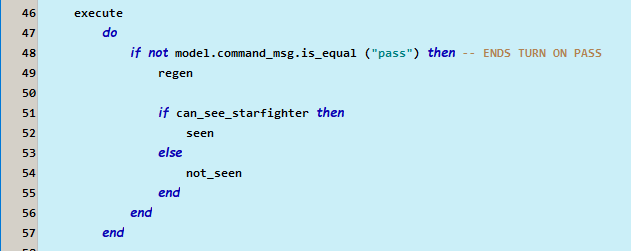
Going back to the in\_game state, this is where we initially call preemptive actions, we simply use dynamic binding to our advantage and call preempetive\_action on every single enemy that is on the board.

The following snippet is found in in\_game state class:



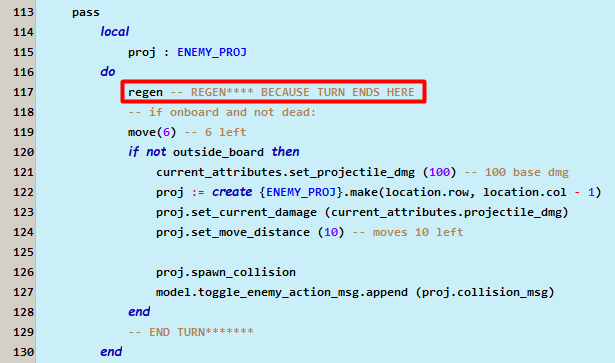
This is where we call pre-emptive enemy actions before starfighters’ act. And this game\_update method is called from each ETF command that is related to ingame actions.

Case2: Pre-emptive actions when turns DO end



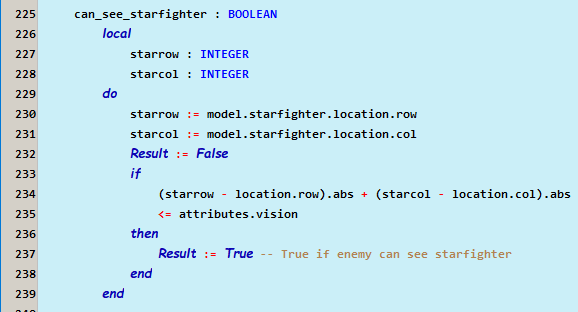
This above snippet is taken from enemy, Fighter. Here in execute we have an if statement where it doesn’t execute the seen and not seen of fighter whenever user inputs pass (which ends fighter’s turn in preempetive).

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:



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

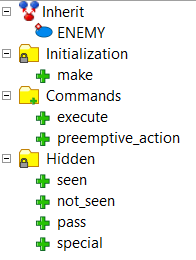
Earlier we also mentioned that can\_see\_starfighter is a command inherited from the deferred class ENEMY. Below you can find the code snippet:



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).

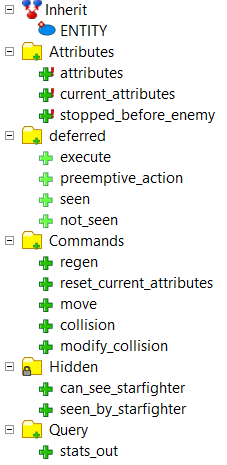
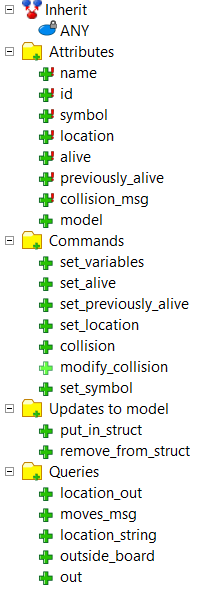
Information Hiding

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.

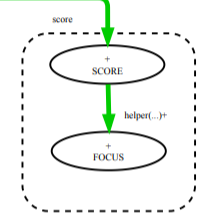
Programming from the Interface, Not from the implementation.

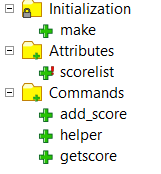
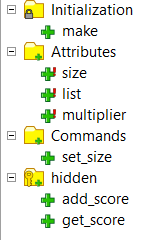
This was satisfied by using deferred classes and methods. I’ve made sure to make everything as similar as possible so enemy actions can be easy to understand.

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:

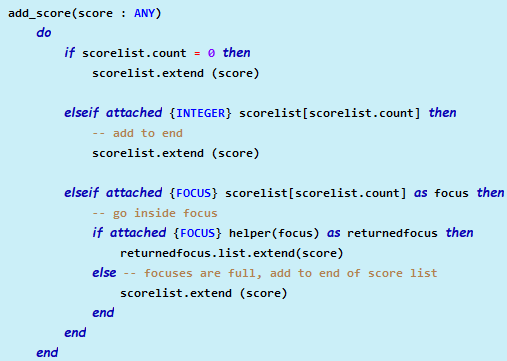
Score uses recursion to add the points, points are described as orbs and focuses. Instead it was decided to use orbs as integers and created a separate class for focuses.



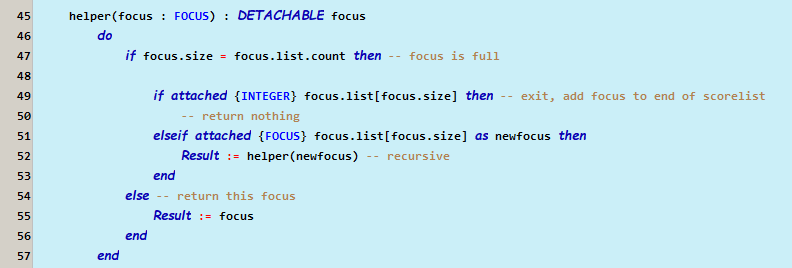
Above is FOCUS features

Above is SCORE features

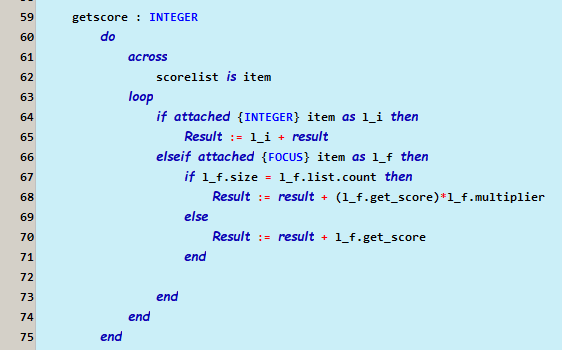
Score makes use of a linked\_list, which stores type ANY. This allows for integers and focuses to be stored into, we also make use of type casting to make sure nothing else gets inserted into score.



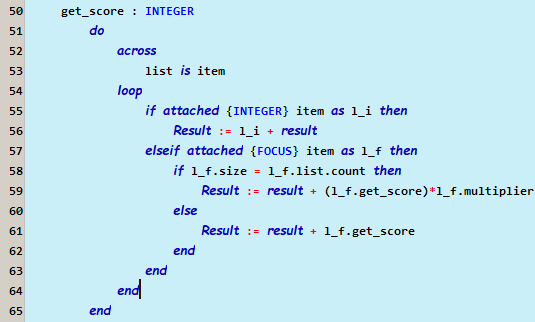
Base case, when list size is 0, add either the ‘orb’ or ‘focus’ into the linkedlist, named as scorelist. If the last item added into the list was an integer, then that means we can directly add to the list. Otherwise then a focus was inserted last, here we use recursion by using a helper method named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



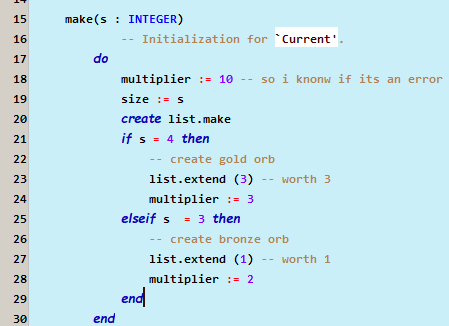
Helper accepts the focus as a parameter then traverses inside, and goes to the ‘bottom/depth’ of the focus. Focus.size is a predefined size which determines the max size of the focus, if its equal to the size of the list in focus then that means the focus is full. We then check whether its an integer 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.



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:

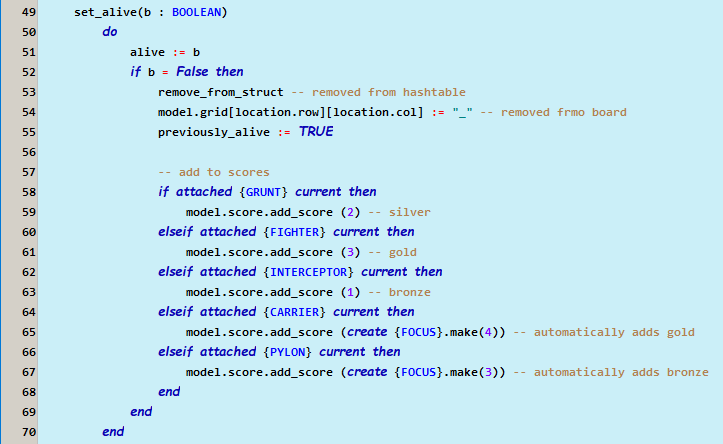


Get\_score, traverses focus’ list and adds if its an integer, other wise go inside the focus and recall get\_score and apply the multiplier.

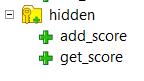


This is found in FOCUS, here we set the multiplier and the ‘size’ upon creation. For example if a carrier gets destroyed, it gives the param 4, and creates the size of 4 and immediately adds a gold orb being integer 3. And sets multiplier to 3. Similarly, Pylon works the same way.

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:



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.