EECS 3311 Project
Design Document

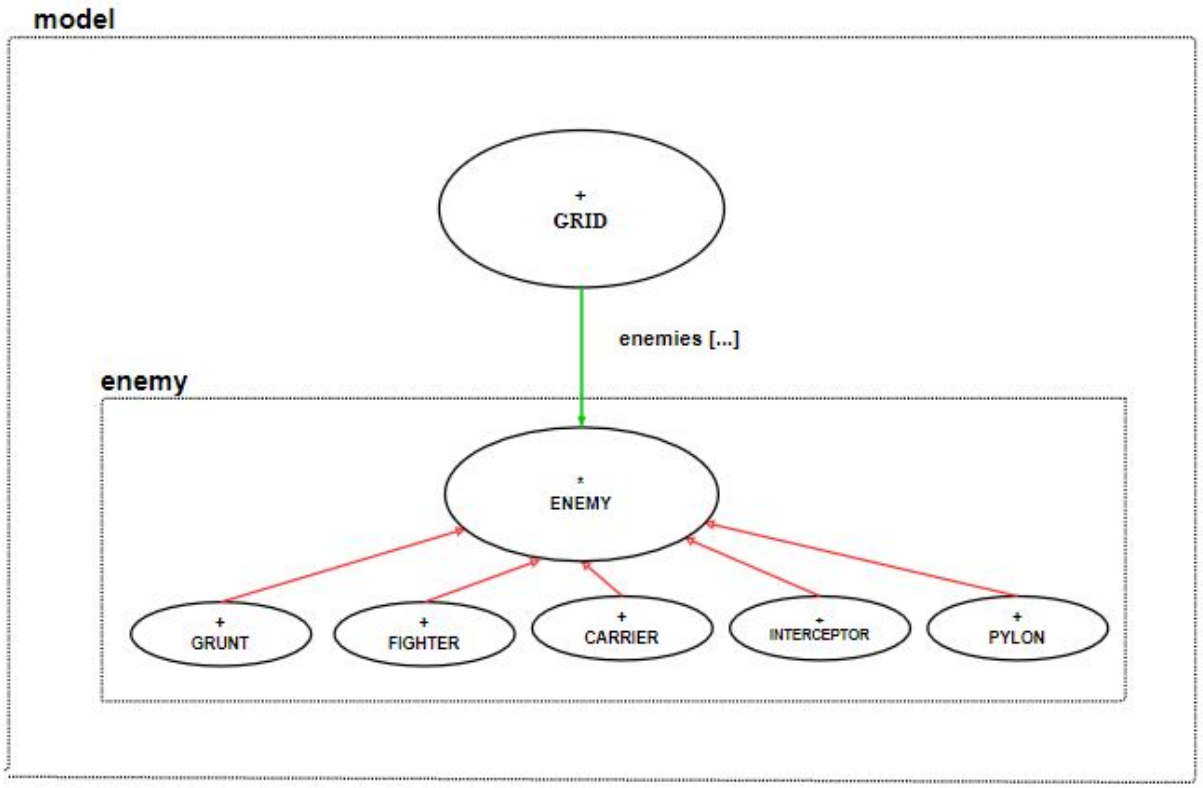
Course **: EECS 3311**
Semester **: Fall 2020**
Name **: Eric Kwok**
EECS Prism Login **: ericn576**

Section : Enemy Actions

How did I design this section?

For the enemy actions in the game I had to approach the problem in a specific way in order to solve the problem. Looking at the problem there are many factors that had to be taken into consideration for it to work. My idea was that I had to be able to isolate these factors into their respective classes that use them for their intended purpose. Before I explain how I designed my solution first we will go through the problem that needs to be solved. So with the current enemies in the game we are tasked with performing their specific phase that is the fifth phase in a starfighter's successful turn. What happens in this phase is first in the order of oldest to newest spawned enemies an enemy will preempt their actions based on what command the starfighter took. These actions can vary and could potentially end a turn for that specific enemy. That's when we get into the next part of the phase where in the same order the enemy will perform a turn depending on whether they can see the starfighter or not. In order for this part to happen the enemies turn must've not ended from its preemptive action. Some of these actions can vary from spawning enemies, firing projectiles, moving, healing other enemies, etc.

Looking at this problem I know that it would eventually have to be used together with other classes in order to interact with the game directly but the first thing that I did was define what an enemy was so that I could easily give enemies their specific actions and features. This is what the enemy hierarchy looks like as well as its client,



Now we will look into how we created these classes and how they will be used in the game

First looking at the enemy class, here we define what an enemy is. The class is deferred as there are certain routines that will be defined in descendant classes. The enemy has many of the same attributes as the starfighter with values such as health, armour, vision, id, etc. These values will be used to keep track of the enemy and its current status. There are also certain things that we need to keep track of unlike the starfighter and that is whether we can see the starfighter or if the starfighter can see us. These attributes will be used and updated frequently at each turn to help us decide what action we must take. Shown below is one of the update methods that we use to update the seem value for an enemy,

```
update_seen_by_starfighter
    do
        seen_by_starfighter := game_info.grid.can_see (game_info.starfighter, row_pos, col_pos)
    ensure
        value_set_correctly : seen_by_starfighter = game_info.grid.can_see (game_info.starfighter, row_pos, col_pos)
    end
```

Several setters are also used in the enemy class to update values as necessary and helper methods that will help us out when debugging our code. Also there are methods used in scoring for enemies (Will be looked at in the scoring system part). Other than that remains the main functionality of the enemy, which are the commands that must be taken at each turn. We have preemptive actions and actions whether the starfighter is seen or not. These are deferred so that specific enemies can implement their functionality on them. Some contracts are still used in order to ensure that the action being taken is correct for example,

```
action_when_starfighter_is_not_seen
    require
        is_not_seen : not can_see_starfighter
    deferred end
```

Here in order for an enemy to take an action the starfighter must not be seen and this is where the attributes we discussed earlier come in handy. After defining our enemy class we now can start implementing its descendant classes which are the types of enemies in the game. Let's look at one of them which is the pylon enemy. So for the pylon there are no preemptive actions so when its a plyons turn it does nothing. For its actions when the starfighter is not seen it will move 2 spaces and if it is not destroyed then it will heal all enemies within its vision range by 10. How I did this was to now implement the deferred method I had in the enemy class and perform this specific action for it. The same idea is then applied to the other actions. We have a boolean attribute that will vary based on whether the enemies turn is over so if it is then it won't perform any more actions,

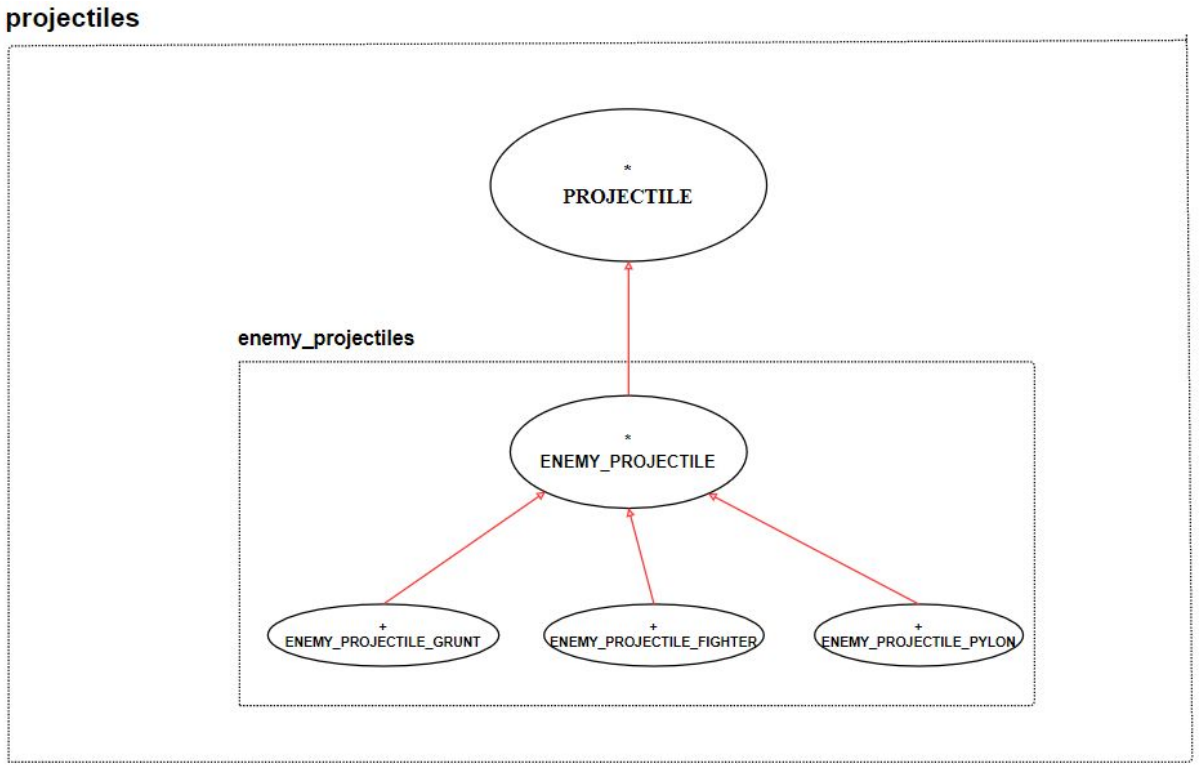
```
is_turn_over = false
```

Let's say for an enemy that does do something in a preemptive action what we will do is that the command will pass a signature which will then indicate to the enemy what action to take in their preemptive action. This is what is looks like,

```
preemptive_action (type : CHARACTER)
```

Some enemies may need to perform a regeneration which is also applied in the turn ordering as specified in the game. This will work the same way as the starfighter except only the health will be healed as there is no energy.

Now that we have our actions in place there are some specific cases that we need to look for. The first one is moving. This is trivial as we just need to move the enemy's position while considering the collisions with other entities. This will be performed in these methods. The next case is firing projectiles where we will have to formulate a way to define these projectiles and their specific actions. This is what the enemy projectiles will look like,



Here we have defined the projectiles as enemy projectiles and in the same way as enemies we first define a general class which is deferred in order to have common routines that will be implemented in descendant classes. In the same way as the enemies, projectiles will have their necessary attributes from id, position, damage, etc. From there in the same way that we defined actions for enemies in the ENEMY class we will have something similar for projectiles where they must "do their turn",

```
do_turn
    -- Turn Action for a Projectile
    deferred end
```

Now with this enemies can use the grid in order to fire projectiles from enemies when necessary. Lastly we have spawning enemies which also is very similar to spawning projectiles however in this case we already have enemies defined so we can just do it in the same way we spawn enemies each turn but in this case we have a specific coordinate in mind and what type of enemy we want to spawn.

Once we have defined an enemy and what it does it's time to actually use it in the game. The GRID class will hold enemies in a list where all the enemies that are spawned will be held. The same thing can be said about the enemy projectiles that the enemies spawn as well.

This can be shown from this snippet in the GRID class,

```
enemy_projectiles : LIST[ENEMY_PROJECTILE]
enemies : LIST[ENEMY]
```

From here the grid in a specific game will keep track of these enemies and projectiles then at each turn will call each of their specific actions. This can be shown below of how they are called,

```
enemy_action
    require
        is_alive : game_info.is_alive = true
    local
        i : INTEGER
    do
        from
            i := 1
        until
            i > enemies.count
        loop
            if enemies.at (i).can_see_starfighter then
                if is_in_bounds (enemies.at (i).row_pos , enemies.at (i).col_pos) then
                    enemies.at (i).action_when_starfighter_is_seen
                end
            else
                if is_in_bounds (enemies.at (i).row_pos , enemies.at (i).col_pos) then
                    enemies.at (i).action_when_starfighter_is_not_seen
                end
            end

            if game_info.starfighter.curr_health = 0 then
                i := enemies.count + 1
            end

            i := i + 1
        end
    end
```

Here all the enemies actions are called based on whether they can see the starfighter or not. This will be called in specific commands in the user commands cluster when a turn is successful and phases can be applied. This then implements the enemy actions aspect of our game.

How does this design satisfy Information Hiding?

This design satisfies information hiding since specific attributes that should only be accessible from the enemies themselves are hidden from other classes to using them. For example the boolean value which determines whether the current enemies turn is over yet is left hidden from other classes from using.

```
feature {NONE} -- Private Attributes
    is_turn_over : BOOLEAN
```

Attributes that are necessary for other classes to use are not hidden and ones that should be hidden from other classes have their implementation hidden. Thus our design satisfies information hiding.

How does this design satisfy the Single Choice Principle?

This design satisfies the single choice principle since our inheritance hierarchy with the enemies and projectiles means that features that are reused over several types of classes will be put in one place so when changes need to be made they can be done so in that place. For example the regeneration of enemies can be defined the same way for all enemies so it would be better to put it in one place where if we need changes it wouldn't require us to go through every enemy's implementation to do so. Thus the single choice principle in our design is satisfied since we have no multiple instances of the same thing.

How does this design satisfy Cohesion?

This design satisfies cohesion since our classes are not duplicating data from each other. From the singleton design pattern which applies a once routine on the ETF_MODEL_ACCESS class it then ensures that only one instance of the grid is created and that grid is then used among subsequent classes ensuring that it is not duplicated and is being shared between them. The classes also satisfy cohesion since each class is focused on its intention meaning that features necessary for that class will be used for that class specifically. For example an enemy specific action will be taken into consideration for that enemy. Each class is focused on its specific functionality. Thus our design satisfies cohesion.

How does this design satisfy Programming from the interface, Not from the Implementation?

This design satisfies programming from the interface, not from the implementation because in cases where we used or declared certain attributes, the interface class was used as it's type until it was implemented where we could decide on how we wanted to implement it. This can be shown from the snippet below,

```
enemy_projectiles : LIST[ENEMY_PROJECTILE]
enemies : LIST[ENEMY]
```

In this case we used the interface rather than then implementation so that we could declare it's implementation when we make the object,

```
create {ARRAYED_LIST[ENEMY_PROJECTILE]} enemy_projectiles.make (0)
create {ARRAYED_LIST[ENEMY]} enemies.make (0)
```

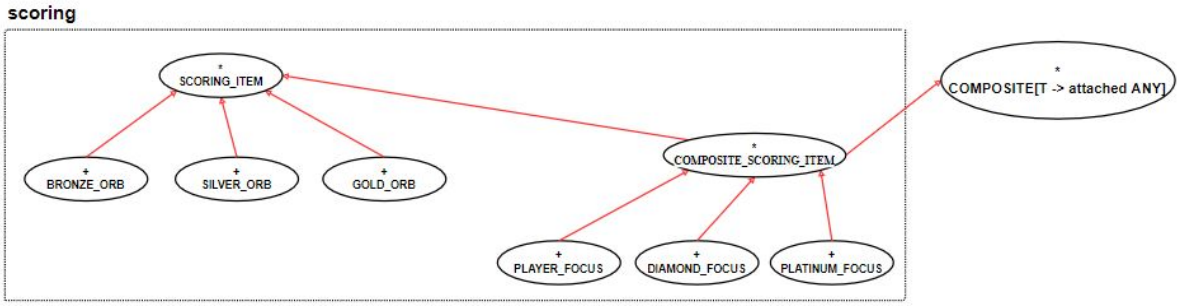
Thus our design satisfies programming from the interface and not the implementation.

Section : Scoring of Starfighter

How did I design this section?

For the scoring of the Starfighter in the space defender game I approached the problem with a way in order to effectively solve the problem. Before I discuss my design for this problem we will break down what exactly needs to be done. So for the scoring of the Starfighter we have orbs where certain enemies can drop and they contain a value that can be added to the starfighters score. Also there are focuses where certain enemies can drop and the player has one as well. Focuses can hold focuses and orbs and certain focuses have multipliers when they are filled with orbs. Only the player's focus will have an unlimited capacity while others have a finite size. When calculating the score of the starfighter should be recursive.

Looking at this problem my approach to this was to use the **Composite Design Pattern** to formulate my solution. This is because this problem uses recursive artifacts so it would be a good idea to build a structure to represent the whole-part heirarchy. After designing the architecture for our scoring system this is what it looks like,



First we will look at the SCORING_ITEM class where items will have a certain value that will be returned when called which will determine the items score. Notice how this item isn't composite meaning that scoring items can't add other items to itself. This class is deferred as its descendants will implement the feature for its specific value to return. Here is a snippet of the feature in this class,

```
feature
    value : INTEGER
        deferred
        ensure
            non_negative_value : Result >= 0
        end
```

As you can see there is a postcondition which will ensure that we don't return any negative scores because all scoring items will have a positive value at least. This will be important when we get to its descendant classes.

Next we have all the orb classes BRONZE_ORB, SILVER_ORB and GOLD_ORB which are descendants of SOCRING_ITEM. These are the items that are not composite and can only hold themselves. Each of them will return a specific value which will correspond to its score. Here is a snippet from the BRONZE_ORB class,

```
feature
    value : INTEGER
        do
            Result := 1
        ensure then
            correct_value : Result = 1
        end
```

From what we can see from this the value feature is now implemented and we can see that it's specific value is 1 in this case. Something to note is the new postcondition on this feature. Here we use subcontracting and from what we can see when we run the program the new postcondition should **ensure more** than the old postcondition. This means it should allow less outputs than the original one.

$$Result = 1 \Rightarrow Result \geq 0$$

Above should be implied from this case and looking at the implication the only way for it to be false would be for the new post to be true and the old post to be false. If the new post was true then the Result would be 1 and 1 is always non-negative, ensuring our design of the postcondition.

Breaking down this design we can see that we use multiple inheritance on the COMPOSITE_SCORING_ITEM class in order to inherit COMPOSITE and SCORING_ITEM features and make sure that SOCRING_ITEM doesn't have any composite features since we only want focuses to be able to add other items. This applies to all other subsequent orb classes.

Now looking at the COMPOSITE class we can see that we have used this class to separate the composite functionality that originally would have been in the SCORING_ITEM features. The good thing about doing this is that generalizing the COMPOSITE will allow us to use it for any applications that the composite is necessary for without implementing it again. The composite will allow the object to be able to add objects to itself. As for the implementation for it I decided to use a linked list to hold those objects. Another thing that I did to it was to add an abstraction via mathematical models to allow for easier implementation changes if I wish to do so in the future. In this case I used a sequence to form the model of all the objects the composite is holding. From there all contracts that we have in other features should use the model to do so rather than the specific implementation. Here is an example,

```
ensure
    model.count ~ (old model.deep_twin).appended (c).count
end
```

Now we will move onto the COMPOSITE_SCORING_ITEM class. Now that we have defined the COMPOSITE and SCORING_ITEM class we can inherit both of them using multiple inheritance. By doing this now we can have composite features and scoring features at the same time without letting non composite scoring items have access to composite features. What we will do in this class is introduce new helper features such as capacity and boolean queries to let us know if this item has a limit or not. The value will be calculated recursively by iterating the children of the item and calling its subsequent values until it reaches the end. This can be shown in the snippet below,

```
feature
    value : INTEGER
        do
            across
                Current.model as cursor
            loop
                Result := Result + cursor.item.value
            end
        end
```

Next we will look into its descendant classes PLAYER_FOCUS , DIAMOND_FOCUS and PLATINUM_FOCUS which will be our composite scoring items. These items will be able to add other items to themselves and some of them even have capacities and multipliers when they reach that capacity. An example of this can be shown in the DIANMOND_FOCUS class where if the focus is at capacity then the score returned by it will be multiplied by 3. This can be done by calling a precursor from COMPOSITE_SCORING_ITEM value feature and multiply it by three if it is full. Here is a snippet of how its done,

```
local
    base : INTEGER
do
    base := Precursor
    if children.count = capacity then
        Result := base * 3
    else
        Result := base
    end
end
```

The other types of focuses will then follow with their specific conditions. Now after doing this we will have our scoring system set up and all we need to do is use it with our starfighter. The STARFIGHTER class will hold two attributes, one for the player's focus and one for all the other focuses. The one that holds other focuses will act as a collection to check the latest focus in the player's focus and if the latest orb needs to be added to it. Here is a snippet,

```
player_focus : COMPOSITE_SCORING_ITEM
focuses : LIST[COMPOSITE_SCORING_ITEM]
```

From there items will be added accordingly when enemies die and the score will be periodically updated at each turn. With this we have implemented the scoring system.

How does this design satisfy Information Hiding?

This design satisfies information hiding since we have hidden the choice of implementation for composite using mathematical models

```
feature {NONE}
    children : LINKED_LIST[T]
```

This way using the model feature from our contracts will allow our implementation to be hidden and when we decide to change the implementation we will only have to adjust the body of the abstraction function. This is good because we are able to hide our design decisions that are likely to change. Thus information hiding is satisfied for our design. None of the other classes in our design have decisions that are likely to change (Ex : value returning an integer).

How does this design satisfy the Single Choice Principle?

This design satisfies the single choice principle since we only have one instance of our COMPOSITE class meaning that changes in the composite feature would only require one place to make those changes since any class that is composite will just have to inherit from that one class. As for the other features inheritance also prevents code reuse and any changes needed to be made can be affected in one place. If inheritance is not used then every class that had a certain feature implemented would have to have the changes applied. Thus the single choice principle in our design is satisfied since we have no multiple instances of the same thing.

How does this design satisfy Cohesion?

This design satisfies cohesion since our classes are not duplicating data from each other. From the singleton design pattern which applies a once routine on the ETF_MODEL_ACCESS class it then ensures that only one instance of the starfighter is created and that starfighter is then used among subsequent classes ensuring that it is not duplicated that is being shared between them. The classes in this design also satisfy cohesion since each class is focused on its intention meaning that features necessary for that class will be used for that class specifically. For example only a composite scoring item will need to take capacity into consideration but not scoring items so only the composite scoring items will use capacity. Each class is focused on its specific functionality. Thus our design satisfies cohesion.

How does this design satisfy Programming from the interface, Not from the Implementation?

This design satisfies programming from the interface but not the implementation because in cases where we used or declared certain attributes, the interface class was used as its type until it was implemented where we could decide on how we wanted to implement it. This can be shown from the snippet below,

```
player_focus : COMPOSITE_SCORING_ITEM
focuses : LIST[COMPOSITE_SCORING_ITEM]
```

Here we used the interface rather than the implementation so that we could declare its implementation when we make the object. As well as the type COMPOSITE_SCORING_ITEM where the most generalized class is declared and a more specific implementation will be chosen when we make the object.

```
player_focus := create {PLAYER_FOCUS}.make
focuses := create {ARRAYED_LIST[COMPOSITE_SCORING_ITEM]}.make (0)
```

Thus our design satisfies programming from the interface but not the implementation.