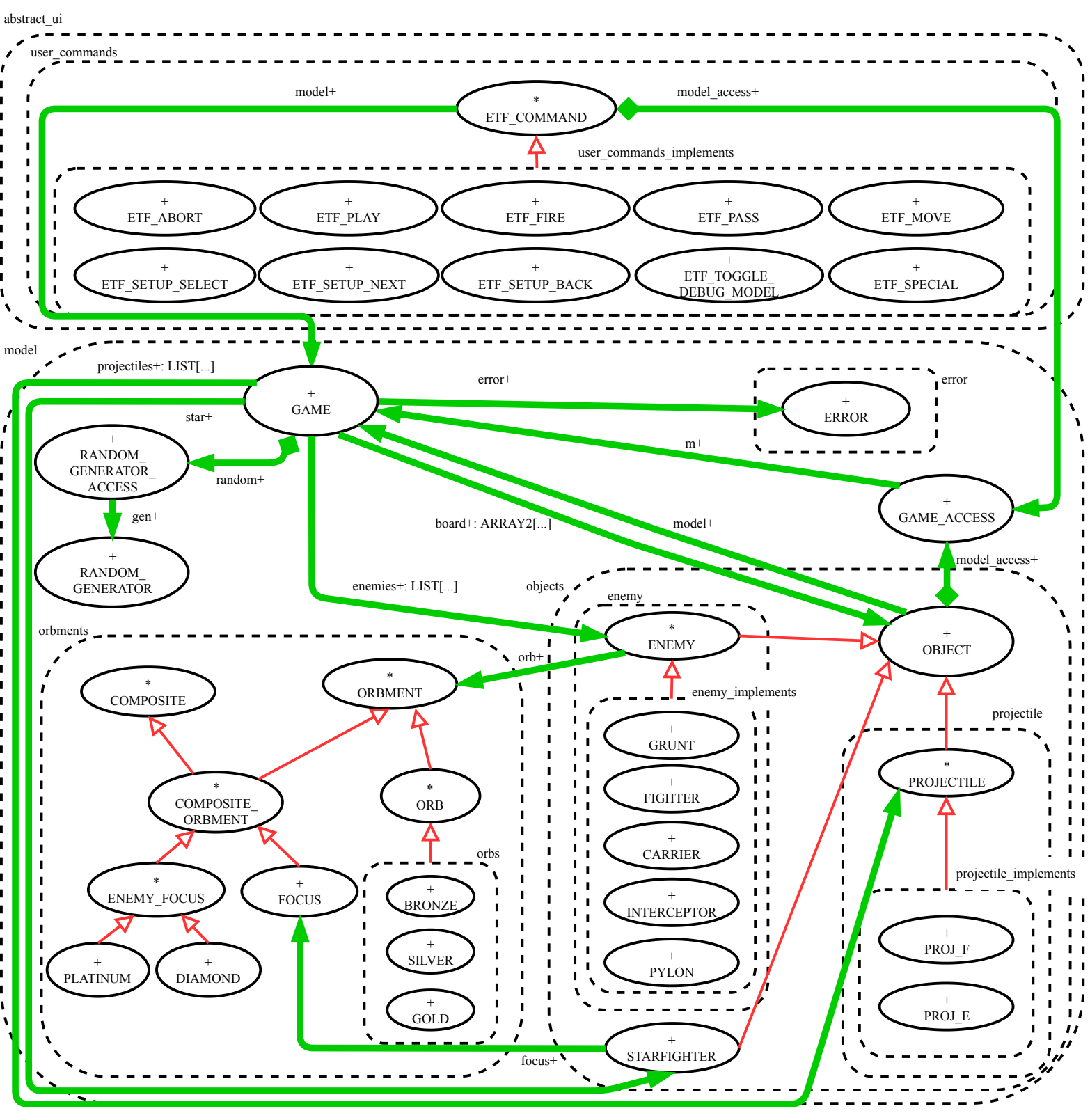
Course	EECS 3311
Section	E
semester	2020 Fall
name	Jiahao Li
Student number	216263949
EECS Prism login	Jiahao18


abstract_ui
user_commands
model+
* ETF_COMMAND
model_access+
user_commands_implements
+ ETF_ABORT
+ ETF_PLAY
+ ETF_FIRE
+ ETF_PASS
+ ETF_MOVE
+ ETF_SETUP_SELECT
+ ETF_SETUP_NEXT
+ ETF_SETUP_BACK
+ ETF_TOGGLE_ DEBUG_MODEL
+ ETF_SPECIAL
model
projectiles+: LIST[...]
+ GAME
error+
+ ERROR
error
star+
m+
+ RANDOM_ GENERATOR_ ACCESS
random+
gen+
+ RANDOM_ GENERATOR
board+: ARRAY2[...]
model+
+ GAME_ACCESS
model_access+
enemies+: LIST[...]
objects
enemy
* ENEMY
+ OBJECT
orbments
orb+
* COMPOSITE
* ORBMENT
enemy_implements
+ GRUNT
+ FIGHTER
+ CARRIER
+ INTERCEPTOR
+ PYLON
projectile
* PROJECTILE
* COMPOSITE_ ORBMENT
* ORB
orbs
projectile_implements
* ENEMY_FOCUS
+ FOCUS
+ BRONZE
+ SILVER
+ GOLD
+ PROJ_F
+ PROJ_E
+ PLATINUM
+ DIAMOND
+ STARFIGHTER
focus+

ENEMY*

feature -- game model
  model+ : **GAME --**the game model object

feature -- ENEMY attributes
  active+: **BOOLEAN** -- indicate the enemy active or not
  ready+: **BOOLEAN** -- indicate the enemy is ready to do action
  health+: **INTEGER** -- indicate the health of enemy
  max_health+: **INTEGER** -- indicate the max health limit of enemy
  h_r+: **INTEGER** -- indicate regeneration of health
  armour+: **INTEGER** -- indicate the armour of enemy
  vision+: **INTEGER --** indicate the view cover of enemy
  pos+: **PAIR[INTEGER,INTEGER]** -- record the position of enemy
  seen_by_starfighter+: **BOOLEAN**
  can_see_starfighter+: **BOOLEAN**
  orb+: **ORBMENT --** the scoring orbment of enemy

feature -- enemy related queries
  in_board+: **BOOLEAN --** is current position on board or not
    ensure
      correct: **Result** = (0<pos.first<model.max_r)∧ (0<pos.second<model.max_c)

  ready_to_act+:**BOOLEAN** --return whether enemy is ready to do action
    ensure
      correct: **Result** = (model.in_game ∧ active ∧ ready)

  display+ : **STRING** -- display the state of enemy

feature {NONE} -- auxiliary command
  move_to+ (r,c:**INTEGER**) -- move to the target postition
    require
      able_to_move: ready_to_act
    ensure
      active_move: active ⇒ in_board ∧ health > 0

feature -- enemy related commands
  make+ (i, r, c:**INTEGER**)
    -- set the basic attributes of enemy
    require
      vaild_id: i > 0
    ensure
      correct_pos: pos.first = r ∧ pos.second = c
      correct_id: id = i

  action***** --normal action
    require
      able_to_act: ready_to_act
    ensure
      case_current_inactive: ¬ active ⇒ (¬ in_board ∨ health = 0)
      case_star_destroyed: model.star.destroyed ⇒ (model.star.health = 0)

  preemptive_action***** --preemive action
    require
      able_to_pre_act: ready_to_act
    ensure
      case_current_inactive: ¬ active ⇒ (**¬** in_board ∨ health = 0)
      case_star_destroyed: model.star.destroyed ⇒ (model.star.health = 0)

  generation+
    require
      allow_to_reg: ready_to_act
    ensure
      not_exceed_max: health ≤ max_health
      full_regen: (health = **old** health + h_r)⇒(**old** health ≤ max_health - h_r)
      regen_to_max: health = max_health ⇒ (**old** health ≥ max_health - h_r)

CARRIER+

feature -- create routine
  make++ (i, r, c:**INTEGER**)
    -- create routine, create a GRUNT object
    require
      vaild_id: i > 0
    ensure
      correct_pos: pos.first = r ∧ pos.second = c
      correct_id: id = i
      correct_orb: orb.value = 2

feature {NONE} -- auxiliary command
  generate+ (r,c:**INTEGER**)
    -- generate a INTERCEPTOR at position [r,c]
    -- reture true if generated INTERCEPTOR is still active
    require
      able_to_move: ready_to_act
      not_occupied_by_enemy: ¬ model.board[r,c].id > 0
    ensure
      succeed: generated

feature {NONE} -- auxiliary query
  generated+:**BOOLEAN** -- Return `TRUE` if this turn succeeful creating of INTERCEPTOR

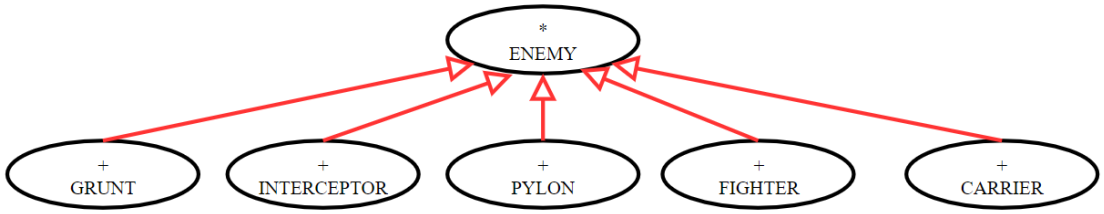
feature -- game related commands
 action**+** -- normal action
  --if can see starfighter, move 1 left, generate 1 INTERCEPTOR at left
  --otherwise, move 2 left
  ensure then
    case_can_see:
      (can_see_starfighter ∧ active) ⇒ (pos.second=(**old** pos.second) - 1) ∧ generated
    case_cant_see:
      (**¬** can_see_starfighter ∧ active) ⇒ (pos.second=(**old** pos.second) - 2)

 preemptive_action**+** --preemive action
  -- if starfighter passes, move 2 left and generate 2 INTERs and end this turn
  -- if starfighter use special, increase `h_r` by 10
  ensure then
    case_pass:
      (model.action = pass ∧ active)⇒ (pos.second=(**old** pos.second) - 2) ∧ ¬ready ∧ generated
    case_special: (model.action = special ∧ active) ⇒ (ready ∧ h_r = **old** h_r + 10)

1. Section: Enemy action

1.1 Design: My design follows the **template** design pattern.
Every different enemy (e.g. GRUNT) will be an object with different actions defined inside its effective class and inherit from the same deferred class *<ENEMY>.* During this section, across all the enemy objects, the action routines will be called by dynamic binding.

1.2 Implementation:
In the enemy action, it is containing 2 action phases: preemptive action and normal action.
Use the pdf document (page 2) `EECS3311_project_detailed_view.pdf` as a reference:
In my model cluster, there is a deferred ***<ENEMY>*** class which contain the common some common effective routines (e.g. *move_to (r, c), generation*) and some common state query of enemy object (e.g. *in_board, display*). There are two deferred routines called `action` and `preemptive action`.



There are total 5 different type of enemies (GRUNT, FIGHTER, CARRIER, INTERCEPTOR and PYLON). Each one of them has an effective class which inherits ***<ENEMY>*** type and implements routines `*action*` and `*preemptive action*`. Different enemy class has different actions implementation.
For an example (page2), the ***<CARRIER>*** does not fire any projectile, but the others do. So ***<CARRIER>*** does not have `fire` related routines.
At the same time, the ***<CARRIER>*** generates *interceptors*, but the others do not. So ***<CARRIER>*** has a unique auxiliary routine called `*generate (r, c)* ` which will generate an *interceptor* at position (r, c).

When the Enemy action section starts, the routine `*enemies_action*` will be invoked.
There are 3 variables is used to indicate whether the actions will be executed or not:
1. ***`active`*** of enemy: active is true implies this enemy is still able to act
2. ***`ready`*** of enemy: ready is true implies this enemy's current turn is not ended or this enemy is not generated in this turn
3. ***`model.star.destroyed`*****:** this indicate the Starfighter is destroyed and the game is over, skip all the actions

```
enemies_action
   -- code fragments of enemies_action
  require
        in_game: in_game
        game_not_over: not model.star.destroyed
  do
        across enemies is e loop
                if e.active and e.ready and not model.star.destroyed then
                        e.preemptive_action
                end
        end
        across enemies is e loop
                if e.active and e.ready and not model.star.destroyed then
                        e.action
                end
        end
  ensure
    check_state:
       across enemies_model is e all not e.active implies (not e.in_board or e.health = 0) end
    case_star_destroyed: model.star.destroyed implies star.health = 0
  end
```

Polymorphism:

The `enemies` is declared as ***<List[ENEMY]>***. The generic type is declared as ***<ENEMY>.***Every time a new enemy is generated, a subclass of ***<ENEMY>*** will be extended into list.

Dynamic binding:

When the `*preemptive_action*` routine is called in the variable `e`. The static type of `e` is ***<ENEMY>.*** The dynamic type of `e` is one of 5 descendants (GRUNT, FIGHTER, CARRIER, INTERCEPTOR and PYLON). For an instance, a ***<CARRIER>*** object is attached in ***<ENEMY>*** `*e*`. When it calls `e.*preemptive_action*`, it is calling the `*preemptive_action*` routine implemented in ***<CARRIER>*** class.

Similar to dynamic binding works on the `*action*` routine.

Runtime:

During this section the enemies will be looped by the order of `*id*` (from 1 to most recently added). If this enemy is not active or its turn is already ended, its action will be skipped. If the Starfighter is destroyed, all the actions will be skipped.

After loop the collection of enemy objects twice, the enemy action section will be addressed.

1.3 Design Principles:

① My design **satisfies** Information Hiding principle.

`enemies_action` has 2 pre-conditions and 2 post-conditions.

All the state variables in the contracts are not likely to be changed. There is a math model of collection of enemy objects called `*enemies_model*` which is **<SET[ENEMY]>** type. Whenever the design of `enemies` change, the clients can only see the `*enemies_model*`. The API content will not be effected.

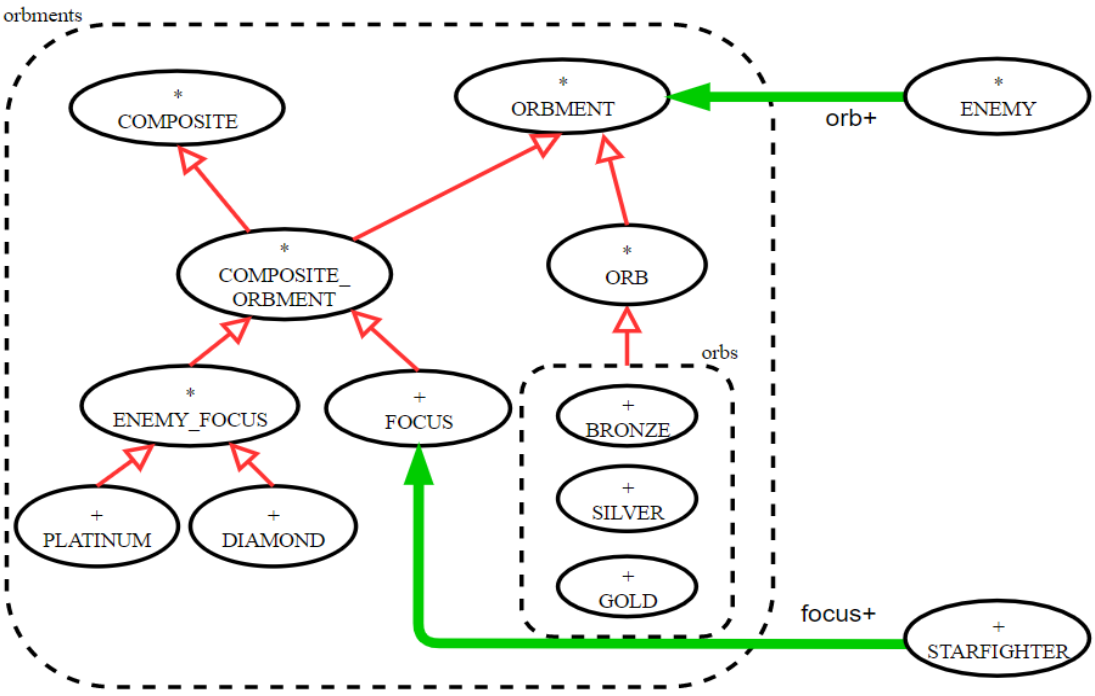
②My design **satisfies** Single Choice Principle
Case 1 (Common routine): If the movement of enemies need to be modified, only need to modify the `move_to` routine in <***ENEMY***>. Because all the enemies share the same movement. Only 1 modification is needed for the common routine.
Case 2 (action): If the preemptive action of ***<GRUNT***> need to be modified, we only need to change the ` preemptive_action` routine in <***GRUNT>.***
When a change is needed, a small place will be change. Because only common routines are inherited from the ancestor <ENEMY>, the action related routines are implemented in different class.

③My design **satisfies** Cohesion Principle.
As I introduced before, the <***ENEMY***> contains enemy related state queries or commands. The cohesion of <***ENEMY***> is maintained.
The descendant of ***<ENEMY***>, for an instance, the <***CARRIER***> class has different routines compared with the other descendant. In all the actions of <***CARRIER***>, it doesn't fire any projectile. So the `fire` related routines are only implemented in the other descendants (grunt, fighter, pylon). And <***CARRIER***> is the only enemy type which generate interceptor in its action. So it has an auxiliary `*generate(r, c)*` routine which implements generate function.
Same for the other classes, they only contain routines which relates to their actions.

④My design **satisfies** Programming from the Interface, Not from the Implementation
When the client (<***GAME***>) use the supplier (<***ENEMY***>) routines, it doesn't cast the <***ENEMY***> into the dynamic type of the object. It calls deferred routines from <***ENEMY***> and then calls the implemented routine by dynamic binding. When the client need enemy to do some actions, it only calls the routines is defined in <***ENEMY***> type.

2. Section: Scoring of Starfighter

2.1 Design: My design follows **Composite** design pattern.
Because the scoring system is a recursive system: While some of orbment are called orb which is an individual orbment, some orbments are composite structure called focus which contains other orbments inside. As a conclusion, the composite design pattern is suitable here.



2.2 Implementation: (use the above compact view diagram as a reference)

We have 6 different types of <**ORBMENT**>.

3 of them are individual <**ORB**> which only have `*value*` routine here.

1 of them is <**FOCUS**> which is the orbment collector of Starfighter without capacity limitation.

2 of them are <**ENEMY_FOCUS**> which has a limited capacity and may has different performance when the number of container reaches the capacity.

Both focus types are inherited from <**COMPOSITE_ORBMENT**> class. This class inherits from <**COMPOSITE**> which is a generic type and it has the `children` as collector and an `add` routine allowing extending the collector.

There are 2 important routines we need to introduce here: **`add (o: OBRMENT) `** and **`value: INTEGER`.**

2.2.1 `add (o: OBRMENT) `: This is the routine allows the <**COMPOSITE_ORBMENT**> to extend the `children` list in the current focus. The `add` routine is implemented in <**FOCUS**> and <**ENEMY_FOCUS**>. The difference between these two routines is the capacity limitation of the <**ENEMY_FOCUS**>.

2.2.2 `value: INTEGER`: This is the query routine which will calculate and return the score of Starfighter.

The three <**ORB**>s' (<**BRONZE**>, <**SILVER**>, <**GOLD**>) version of `value` just return the stored value in it.

The <**COMPOSITE_ORBMENT**> version will call the `*value*` routine in every <**ORBMENT**> in children recursively and add them up. Call the `*value*` routine recursively means the <**ORBMENT**> could have a dynamic type of <**ENEMY_FOCUS**>.
The <**ENEMY_FOCUS**> call the `Precursor` to get value and check if the number of children reaches the capacity. If so, it will return the `*value*` * `*multiple*` as return value.

Runtime:
The `focus` attribute, type <**COMPOST_ORBMENT**>, is the focus of <**STARFIGHTER**>.
When there is an enemy destroyed, `*focus.add(orb)*` will be invoked.
When we need the score of Starfighter, just call `*focus.value*` and it will return the result score.

2.3 Design Principles:
① My design **doesn't satisfy** Information Hiding principle strictly.
In the <**COMPOSITE**> generic class, there is an attribute called `children` which is <**LIST**[**T**]> type. This will be used in the contract of `*add(o:* **ORBMENT**)` routines.
Reasons why clients can see the implementation of `children`:
1. This is just a collector of objects. Once the structure is defined, there is no need to change the implementation of this <**LIST**> type.
2. In the `*add*` routine, the API need to show the last element in `*children*` is addable or not. So the sequence of the collector matters here. The <**SET**> type can't be helpful in showing the order of objects.
Although the clients can see the `*children*` implementation here, it is not likely necessary changeable.

②My design **satisfies** Single Choice Principle
All the code can be reused.
If there is a new type of <**ORBMENT**>:
1.If it is an orb, inherit the <**ORB**> and make some change in the `*make*` create routine will do the work.
2.If it is a focus, inherit the <**ENEMY_ORBMENT**> and make some change in the `*make*` create routine.
If there is a new recursive system need to be introduced:
The <**COMPOSITE**> class can be reused in the new system.

③My design **satisfies** Cohesion Principle.
There are 2 type of orbments: orb and focus.
For every orb, it only has the `*value*` routine which return the score of itself.
For every focus, it contains not only `*value*` but `*children*` and `*add*` routines.
Every class only has the functional related routines.

④My design **satisfies** Programming from the Interface, Not from the Implementation
The type of focus of Starfighter is defined as <**COMPOSITE_ORBMENT**> which have the API of `value` and `add`. These two routines are all the function we need.
Although it is an <**ORBMENT**> type, given the fact that the `add` routine will be called usually, better to define it as an <**COMPOSITE_ORBMENT**>.
All in all, the focus has static type of <**COMPOSITE_ORBMENT**> and dynamic type of <**FOCUS**>. When program we call `add` or `value`, it will call the effective routine implemented in <**FOCUS**> by dynamic binding. We use the <**COMPOSITE_ORBMENT**>'s API as interface. This is how the "Programming from the Interface, Not from the Implementation" maintains.