

## **Section: Enemy Action**

The core of the design is relying upon the fact that enemy actions are triggered by the starfighter's actions. With that in mind, I have defined a routine for each of the 7 phases in a turn in the WAR\_GRID class, and a “perform\_turn” routine in the SPACEWAR class which executes all 7 phases in order.

When the starfighter takes an action (i.e – pass, move, fire, special), the action is always carried out in phase 3. The starfighter stores the last action in a local variable in SPACEWAR class.

For example

```
fire
  local
    setup_access: SETUP_ACCESS
    setup: SETUP
  do
    setup := setup_access.m
    if setup.setup_has_begun = false then
      process_not_in_game_error
    else
      if setup.current_step <= 5 then
        -- during setup
        process_not_in_game_error

        -- in game
      else
        last_command := "fire"
        process_command
      end
    end
  end
end
```

When the “perform\_turn” routine is executed later on, it already knows what was the starfighter’s last action, and so, the routine taking care of phase 5 (enemy action) knows too.

```
perform_turn
  local
    war_grid_access: WAR_GRID_ACCESS
    war_grid: WAR_GRID
  do
    war_grid := war_grid_access.m
    war_grid.perform_phase_1
    war_grid.perform_phase_2
    war_grid.perform_phase_3 (last_command, last_move_to_tgt_row,
      last_move_to_tgt_col)
    war_grid.perform_phase_4
    war_grid.perform_phase_5 (last_command)
    war_grid.perform_phase_6
    war_grid.perform_phase_7
    if war_grid.starfighter.is_dead then
      state_header.name := "not started"
    end
  end
end
```

Given that phase 5 knows what was the starfighter's last action, the enemy can preempt accordingly by checking the relevant actions (based on the type of the enemy).

For example:

```
    preempt_action (pass: BOOLEAN ; fire: BOOLEAN ; move: BOOLEAN ; special:
        BOOLEAN): BOOLEAN
    do
        if is_on_grid then
            if pass then
                health := health+10
                max_health := max_health+10
                report_enemy_gains_total_health (10)
            elseif special then
                health := health+20
                max_health := max_health+20
                report_enemy_gains_total_health (20)
            end
            Result := false
        else
            Result := true
        end
    end
```

Depending on the enemy type and/or the starfighter's last move, the enemy may end its turn. If it doesn't, it will act. First all enemies preempt their actions and the ones that did not finish their turn will be added to a list. Then after all enemies have preempted, the enemies in the list who still need to act will do so.

### **Satisfaction of design principles:**

#### **- Information hiding**

In the enemy class, the methods "preempt\_action" as well as "act" are not hidden. They are considered stable as they are directly deduced by the basic requirements of the game.

On the other hand, the way that the move itself is carried out can be adjusted.

The enemy moving happens in "perform\_move" in enemy class. The routine "perform\_move" is hidden from clients.

I actually changed the design more than once during development and that resulted in changing the content of "perform\_move" but it did not have any effect on clients using the class (i.e – WAR\_GRID). Also, an enemy may spawn a projectile during its act, and that too, "spawn\_projectile", is hidden as it may be changed. For instance, at first I let the projectiles undergo collision inspections before spawning. But later on I saw the need to do it the other way around, so I did not have to change anything in client classes.

#### **- Single Choice Principle**

All enemies, and thus enemy actions, are designed using inheritance. There is a main deferred class, ENEMY, which defines all shared code in one place, in addition to deferred routines that will later on be defined by the individual enemy classes that inherit from ENEMY. Thus ensuring that if a change has to be made, it will be made in one place only.

Note: The individual enemy classes extend ENEMY by adding their own specialized attributes and routines to either beautify or simplify the code.

#### - Cohesion

The attributes in the ENEMY class are shared among all its descendants. Some descendants though, such as ENEMY\_GRUNT, have few more attributes to maintain. Those attributes are defined in the class that needs them rather than in the ancestor class, thus satisfying cohesion.

When an enemy moves for instance, the debug string reporting what happened is the same regardless to the type of the enemy. The routine taking care of that was defined and implemented in the ancestor ENEMY class, where it can be used by any enemy object inheriting from ENEMY.

#### - Programming from the interface

In phase 5, every enemy will preempt, and those that did not finish their turn will still need to act.

Those enemy were collected in a LIST, that was created at run time with ARRAYED\_LIST cast.

Also, all enemies ever created are collected in a LIST as well, called “enemies” in class WAR\_GRID.

The runtime type is also ARRAYED\_LIST. If for any reason in the future we decide to change the run time type to, say LINKED\_LIST, none of the clients of WAR\_GRID would be affected as they are using the interface itself, not the implementation.

-- Entity initialization

create starfighter.make (row\_count)

create {ARRAYED\_LIST[PROJECTILE]} projectiles.make (10)

create {ARRAYED\_LIST[ENEMY]} enemies.make (10)

### **Section: Scoring of Starfighter**

There's a deferred ancestor class called SCORE\_ITEM, that has the descendant classes ORB as well as FOCUS. The idea is that an orb and a focus are both a SCORE\_ITEM.

There are only 2 routines for us to be concerned with:

1. “add\_score\_item”
2. “get\_value”

The starfighter maintains a focus (FOCUS) where scores items (i.e ORB or FOCUS) can be added.

When an enemy dies, it accesses the starfighter's focus and executes “add\_score\_item” on it, adding either an orb or a focus (depending on the enemy type).

Whenever it's time to print the state strings and content, the very last step is to print the starfighter's score, by executing “get\_value” on the starfighter's focus.

#### - add\_score\_item

Traverse the starfighter's focus left to right, if the current item is an orb, go on and ignore.

If the current item is a focus, try to add the score item there recursively.

If successful, end. Otherwise, keep traversing rightward in the original focus.

If there is absolutely no place in any of the focuses within the starfighter's focus, append the score item to the right of the starfighter's focus.

#### - get\_value

Traverse the starfighter's focus left to right, add the value of the current element to the overall value by executing “get\_value” on the current score element. If the current score element is a focus, the routine “get\_value” is executed recursively on the current score element.

If at any point in time we land on a full focus, we multiply its value by the relevant factor depending on the focus type.

## Satisfaction of design principles:

### - Information hiding

In the FOCUS class, there's a method called "is\_full". This method is used to query whether a focus is full during the calculation of the value. If I decide to improve the overall design next week and thus change the way I check if a focus is full, that would not affect any client class because the method is hidden.

On the other hand, "add\_score\_item" as well as "get\_value" are public as a design choice, since they are used by clients such as ENEMY descendants. Their content though can be updated seamlessly because the client class does not see anything but those 2 routines.

### - Single Choice Principle

The class SCORE\_ITEM has the attributes that are common to both ORB and FOCUS classes.

It ensures that if for any reason we decided to change the name or the type of those attributes, we only need to change them in one place – thus satisfying the single choice principle.

Beyond the attributes, there are only 3 deferred methods in SCORE\_ITEM and they're all implemented in the descendant classes.

### - Cohesion

As mentioned above, the SCORE\_ITEM class has 3 deferred methods.

In particular, let's consider "add\_score\_item". While this method makes sense for the FOCUS descendant, it does not make any sense for the ORB descendant.

This violates cohesion because the ORB class has irrelevant functionality.

As a consequence, the ancestor SCORE\_ITEM class also has some irrelevant attributes to any of the two descendants we choose. As seen here

#### feature -- attributes

type: INTEGER

value: INTEGER

**multiply\_factor: INTEGER**

is\_starfighter\_focus: BOOLEAN

t\_orb: INTEGER

t\_focus: INTEGER

The attribute "multiply\_factor" is not needed in the ORB descendant, thus again, violating cohesion.

The reason I designed it that way is for speed of programming, given another equally tough project in EECS3431. But if this was a project conducted in real life (or if EECS3311 was the only course I was taking), then a much better design can be created that would satisfy cohesion by definition.

### - Programming from the interface

The starfighter's focus (and by extension, any FOCUS object), has an attribute of type LIST, with all the focus's content.

The dynamic type of the focus is ARRAYED\_LIST, but all clients of the list in FOCUS use the LIST interface to add or get elements in the focus. Thus, if we change the dynamic type to LINKED\_LIST for example, the clients would not be affected.