

# In-Class Exercises

ETH Zurich

November 20 2013

## 1 Contracts

ETH students recently designed a special kind of oven for cooking potatoes. Here are some facts about such an oven:

- each oven is equipped with a door which is either open or closed;
- the oven is fairly small, therefore only one potato can fit inside;
- it is only possible to put a potato in or take one out when the door is open;
- to start or stop cooking, one has to use the start/stop switch;
- for safety reasons, the oven would not start cooking if its door is open or there is nothing to cook;
- the door cannot be opened during cooking: cooking has to be stopped first.

The following class *POTATO\_OVEN* models such an oven. Please fill in the missing contracts (preconditions, postconditions, and class invariants), so that each fact from the informal specification above is reflected in the class interface.

Please note the number of dotted lines does not indicate the number of missing contracts.

deferred class

*POTATO\_OVEN*

feature -- Access

*potato\_to\_cook*: *POTATO*  
-- The potato inside the oven.

feature -- Status report

*is\_door\_open*: *BOOLEAN*  
-- Is the oven door open?

*is\_cooking*: *BOOLEAN*  
-- Is the oven cooking?

*is\_empty*: *BOOLEAN*

```
    -- Is the oven empty?  
deferred  
ensure  
  Result = (potato_to_cook = Void)  
end  
  
feature -- Basic operation  
  
  open_door  
    -- Open the door.  
  require  
  
    .....  
    .....  
    .....  
    .....  
  
  deferred  
  ensure  
  
    .....  
    .....  
    .....  
    .....  
  
  end  
  
  close_door  
    -- Close the door.  
  require  
  
    .....  
    .....  
    .....  
    .....  
  
  deferred  
  ensure  
  
    .....  
    .....
```

```
.....  
.....  
  
end  
  
put (a_potato: POTATO)  
    -- Put 'a_potato' into the oven.  
require  
  
.....  
.....  
.....  
.....  
  
deferred  
ensure  
  
.....  
.....  
.....  
.....  
  
end  
  
remove  
    -- Remove the potato.  
require  
  
.....  
.....  
.....  
.....  
  
deferred  
ensure  
  
.....  
.....  
.....
```

.....  
**end**

*switch\_on*

-- Turn on the start/stop switch.

**require**

.....  
**deferred**

**ensure**

.....  
**end**

*switch\_off*

-- Turn off the start/stop switch.

**require**

.....  
**deferred**

**ensure**

```
    end  
invariant  
.....  
.....  
.....  
.....  
end
```

## 2 Inheritance

Below you see the class `GAME_CHARACTER`. The class represents game characters. There are three types of game characters: dragon, marshmallow man and zombie. Every character has a health level in the range of 0 to 100, where 0 means that the character is dead and 100 that it has full strength. Since zombies are dead by definition, their health level stays at 0 at all times. Each of the character types has a damage potential that it can inflict on others. For all of them the damage doubles if the character is angry.

Listing 1: Class `GAME_CHARACTER`

```
class
2  GAME_CHARACTER

4  create
   1  make
6
   2  feature -- Initialization
8
   3  make (t: INTEGER)
10      -- Initialize with type 't'.
   4  require
12      t_valid: (t = marshmallow_man xor t = dragon xor t = zombie) and not
14      (t = marshmallow_man and t = dragon and t = zombie)
   5  do
16      type := t
18      if type = zombie then
20          health := 0
18      else
22          health := 100
20      end
   6  ensure
22      type_set: type = t
   7  end
24
   8  feature -- Access
26
   9  type: INTEGER
28      -- Type of character
30
  10  health: INTEGER
32      -- Health of character (0: dead, 100: full strength)
34
  11  damage: INTEGER
36      -- Damage that the character can do
   12  do
38      if type = zombie then
40          Result := zombie_damage
38      elseif type = marshmallow_man then
42          Result := marshmallow_man_damage
40      else
44          Result := dragon_damage
42      end
```

```

44   if is_angry then
      Result := Result * 2
    end
46   ensure
      zombie: not is_angry and type = zombie implies Result = zombie_damage
48   angry_zombie: is_angry and type = zombie implies Result = 2*zombie_damage
      dragon: not is_angry and type = dragon implies Result = dragon_damage
50   angry_dragon: is_angry and type = dragon implies Result = 2*dragon_damage
      marshmallow_man: not is_angry and type = marshmallow_man implies Result =
          marshmallow_man_damage
52   angry_marshmallow_man: is_angry and type = marshmallow_man implies Result = 2*
          marshmallow_man_damage
    end
54 feature -- Status report
56   is_dead: BOOLEAN
58   -- Is the character dead?
      do
60     Result := (health = 0)
      ensure
62     Result_set: Result = (health = 0)
      end
64   is_angry: BOOLEAN
66   -- Is the character angry?
      -- (Then it can do more damage!)
68 feature -- Element change
70   set_health (h: INTEGER)
72   -- Set 'health' to 'h'.
      require
74     h_valid: h >= 0 and h <= 100
       h_for_zombie: type = zombie implies h = 0
76     do
          health := h
78     ensure
          health_set: health = h
80     end
82   set_angry (b: BOOLEAN)
      -- Set 'is_angry' to 'b'.
84     do
          is_angry := b
86     ensure
          is_angry_set: is_angry = b
88     end
90 feature -- Constants
92   marshmallow_man: INTEGER = 1

```

```

    -- Marshmallow man
94
    dragon: INTEGER = 2
96    -- Dragon

98    zombie: INTEGER = 3
    -- Zombie (is always dead)
100
    zombie_damage: INTEGER = 1
102    -- Damage that a zombie does

104    dragon_damage: INTEGER = 2
    -- Damage that a dragon does
106

    marshmallow_man_damage: INTEGER = 3
108    -- Damage that a marshmallow man does

110 invariant

112    type_valid: (type = marshmallow_man xor type = dragon xor type = zombie) and not (
        type = marshmallow_man and type = dragon and type = zombie)
    health_valid: health >= 0 and health <= 100
114    zombie_always_dead: type = zombie implies health = 0

116 end
```

The above code does not exhibit a nice object-oriented design and it can hardly be called reusable. Redesign the code such that it uses inheritance instead of the `type` attribute to represent the three types of game characters. Write a **deferred** ancestor class `NEW_GAME_CHARACTER` and effective descendants `ZOMBIE`, `MARSHMALLOW_MAN`, and `DRAGON` that inherit from `NEW_GAME_CHARACTER`.

Your design should

- result in the deletion of the `type` attribute.
- result in the same behavior for the three types of game characters as the original code of class `GAME_CHARACTER`.
- include semantically equivalent contracts as the original code of class `GAME_CHARACTER`.

If a feature stays the same in your re-factored code as in the original code, please indicate it by giving the full feature signature and adding a comment `-- See original`.

Example:

```
is_dead: BOOLEAN
    -- See original.
```





end





