In-Class Exercises

ETH Zurich

November 28 2012

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
not_cooking: not is_cooking
door_closed: not is_door_open -- optional
deferred
ensure
door_open: is_door_open
end

close_door
-- Close the door.
require
door_open: is_door_open -- optional
deferred
ensure
door_closed: not is_door_open
end

put (a_potato: POTATO)
-- Put ‘a_potato’ into the oven.
require
potato_attached: a_potato /= Void -- optional
empty_oven: is_empty
door_open: is_door_open
deferred
ensure
potato_in_oven: potato_to_cook = a_potato
not_empty: not is_empty -- optional
end

remove
-- Remove the potato.
require
not_empty: not is_empty
door_open: is_door_open
deferred
ensure
empty_oven: is_empty
end

switch_on
-- Turn on the start/stop switch.
require
door_closed: not is_door_open
not_empty: not is_empty
not_cooking: not is_cooking  -- optional
defered
ensure
    is_cooking: is_cooking
end

switch_off
    -- Turn off the start/stop switch.
require
cooking: is_cooking  -- optional
defered
ensure
    not_cooking: not is_cooking
end

invariant

    is_cooking  implies not is_door_open
    is_cooking  implies not is_empty
    -- Or: (is_door_open or is_empty) implies not is_cooking
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`

```plaintext
class
  GAME_CHARACTER
create
make

feature -- Initialization
make (t: INTEGER)
  -- Initialize with type 't'.
  require
    t_valid: (t = marshmallow_man xor t = dragon xor t = zombie) and not
      (t = marshmallow_man and t = dragon and t = zombie)
do
  type := t
  if type = zombie then
    health := 0
  else
    health := 100
  end
ensure
  type_set: type = t
end

feature -- Access

  type: INTEGER
    -- Type of character

  health: INTEGER
    -- Health of character (0: dead, 100: full strength)

damage: INTEGER
    -- Damage that the character can do
do
  if type = zombie then
    Result := zombie_damage
  elseif type = marshmallow_man then
    Result := marshmallow_man_damage
  else
    Result := dragon_damage
  end
```
if is_angry then
  Result := Result * 2
end

ensure
  zombie: not is_angry and type = zombie implies Result = zombie_damage
  angry_zombie: is_angry and type = zombie implies Result = 2*zombie_damage
  dragon: not is_angry and type = dragon implies Result = dragon_damage
  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
  angry_marshmallow_man: is_angry and type = marshmallow_man implies Result = 2*marshmallow_man_damage
end

feature -- Status report

is_dead: BOOLEAN
  -- Is the character dead?
do
  Result := (health = 0)
ensure
  Result_set: Result = (health = 0)
end

is_angry: BOOLEAN
  -- Is the character angry?
  -- (Then it can do more damage!)

feature -- Element change

set_health (h: INTEGER)
  -- Set 'health' to 'h'.
require
  h_valid: h >= 0 and h <= 100
  h_for_zombie: type = zombie implies h = 0
do
  health := h
ensure
  health_set: health = h
end

set_angry (b: BOOLEAN)
  -- Set 'is_angry' to 'b'.
do
  is_angry := b
ensure
  is_angry_set: is_angry = b
end

feature -- Constants

marshmallow_man: INTEGER = 1
--- Marshmallow man

```
dragon: INTEGER = 2
    -- Dragon

zombie: INTEGER = 3
    -- Zombie (is always dead)

zombie_damage: INTEGER = 1
    -- Damage that a zombie does

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

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

invariant

```
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

zombie_always_dead: type = zombie implies health = 0
```
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.
```
deferred class
NEW_GAME_CHARACTER

feature -- Access

health: INTEGER
--- Health of character (0: dead, 100: full strength)

damage: INTEGER
--- Damage that the character can do
do
Result := damage_constant
if is_angry then
Result := Result * 2
end
ensure
not_angry: not is_angry implies Result = damage_constant
angry: is_angry implies Result = 2*damage_constant
end

feature -- Status report

is_dead: BOOLEAN
--- Is the character dead?
do
Result := (health = 0)
ensure
Result := Result = (health = 0)
end

is_angry: BOOLEAN
--- Is the character angry?
--- (Then it can do more damage!)

is_valid_health (h: INTEGER): BOOLEAN
--- Is ‘h’ a valid health for the character?
deferred
ensure
Result implies (h >= 0 and h <= 100)
--- other possibility: no postcondition
end

feature -- Element change

set_health (h: INTEGER)
--- Set ‘health’ to ‘h’.
require
h_valid: is_valid_health (h)
do
health := h
ensure
health_set : health = h
end

set_angry (b: BOOLEAN)
— Set ‘is_angry’ to ‘b’.
do
is_angry := b
ensure
is_angry_set : is_angry = b
end

feature —— Constants

damage_constant: INTEGER
—- Damage that a character does
deferred
end

invariant

health_valid : is_valid_health (health)
—- other possibility: health >= 0 and health <= 100
end

Listing 3: Class ZOMBIE

class
2 ZOMBIE

inherit
6 NEW_GAME_CHARACTER

create
make

feature —— Initialization

make
—- Initialize health 0.
do
health := 0
ensure
health_set : health = 0
end

feature —— Status report

is_valid_health (h: INTEGER): BOOLEAN
—- Is ‘h’ a valid health for the character?
do
Result := (h = 0)
ensure then
    Result = (h = 0)
end

feature -- Constants
    damage_constant: INTEGER = 1
invariant
    zombie_always_dead: health = 0
end

Listing 4: Class DRAGON

class
    DRAGON
inherit
    NEW_GAME_CHARACTER
create
    make
feature -- Initialization
    make
        -- Initialize with health 100.
        do
            health := 100
        ensure
            health_set: health = 100
        end

feature -- Status report
    is_valid_health (h: INTEGER): BOOLEAN
        -- Is 'h' a valid health for the character?
        do
            Result := (h >= 0 and h <= 100)
        ensure then
            Result = (h >= 0 and h <= 100)
        end

feature -- Constants
    damage_constant: INTEGER = 2
end
Listing 5: Class `MARSHMALLOW_MAN`

```asciidoc
class MARSHMALLOW_MAN

inherit NEW_GAME_CHARACTER

create
    make

feature -- Initialization
    make
        -- Initialize with health 100.
        do
            health := 100
        ensure
            health_set: health = 100
        end

feature -- Status report
    is_valid_health (h: INTEGER): BOOLEAN
        -- Is 'h' a valid health for the character?
        do
            Result := (h >= 0 and h <= 100)
        ensure then
            Result = (h >= 0 and h <= 100)
        end

feature -- Constants
    damage_constant: INTEGER = 3
```

end