EECS3311

Software Design – PROJECT Text-based Game (DUNGEON_GAME)

ver. 1.0

Group Members:

Mohammad Hossein Zarifi
(Student no: 213320437)

Rachel Chang
(Student no: 213 530 266)

Azadeh Farokhshahi
(Student no: 213421706)

Table of Contents

Table of Contents
1 Introduction 3
1.1 Game description ……...3
1.2 Design pattern**...……..……..……..3**
2 Operation on Map …..... 8
2.1 Enquiry operations 8
2.2 Read operations 9
2.3 Write operations 9
2.4 Other ……...9
3. Contracts ……...10
4. Appendix ……..12

1 Introduction

1.1 Game Description

Welcome to the Dungeon game! It is a text-based game programmed by a group of students.
This game is programmed with an intention of applying the different design patterns of programming learnt in class. You would find three types of design in this game program, that is, the MVC, singleton and Iterator.

In this section, you will get an insight of what the game is all about, and its rules to help you have an enjoyable journey in the land of computers. Don't worry you don't need to have computer knowledge to play this game! So let's get started!

This game consists of two levels - level 0 and level 1.

In level 0 - you need to defeat an enemy

- to fight, you need to build viruses and send it, this will lower the enemy's health points and you win. Your health points are in the form of 'System status', you need to make sure it's higher than the enemy's health points at all times!
- ***NOTE:-*** If you need to make a choice, make sure you type in the correct number against the options given to you. The instructions and the story of the game are displayed as you go on with the game.

In level 1 - you are to find the mini boss and beat it

- In this level, you find the element of randomness as you look around the room.
- Your goal is to find the mini boss and fight!
- Other alternative is to find his minions- the enemy coder and beat 5 of them to win this game.

If you pass both level 0 and level 1, you have saved the village and won the game!

1.2 Design pattern

The design patterns that have been used in this game are MVC, Singleton and Iterator.

MVC: MVC was selected In order to reduce the complexity of the code and Increase flexibility during the implementing the game.

View: There is a View (DUNGEON_VIEW) class which acts as an interface between the system and user. View is responsible for getting the input from the user and also represents visualization of the data that Model contains.

Model: DUNGEON_MODEL class in this game contains all the data and attributes of the system. All the features in this class are public since they are going to be used by other part of the system, mainly controller. DUNGEON_MODEL also contains routines as accessors and mutators in order to give controller the ability to read and write its attributes.

Controller: DUNGEON_CONTROLLER acts as the controller of this system. It acts on both Model and View. It controls the data flow in/out from Model and updates the View whenever there is a change in the data. DUNGEON_CONTROLLER has one instance object of each View and Model. There is no relationship between View and Model and all of the communication between View and Model is made through Controller.
You can see the MVC part of the system in the following diagram:

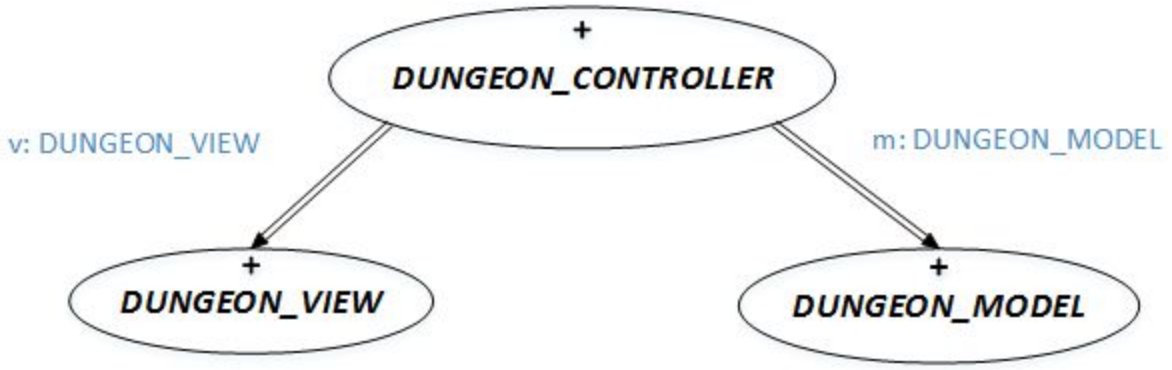

Figure 1: MVC pattern used in the design of the game

Singleton: User should only interact with one instance of the system (Dungeon game) during the whole life cycle of the game. For this reason singleton was selected as one of the important pattern to be used in the system. In Singleton pattern there should be a Singleton class which has only one instance in the system and there should be one global access point to the Singleton class. There is a Singleton class (APPLICATION) and Singleton Accessor (APPLICATION_ACCESSOR). APPLICATION is the root class of the system and everything starts from there and only one instance of it should exists, therefore APPLICATION would act as our Singleton. The access point of the singleton class is APPLICATION_ACCESSOR.
In the following snippet of code you see how just one instance of the application is being made through the APPLICATION_ACCESSOR.

```
class
        APPLICATION_ACCESSOR

feature{ANY} -- Initialization
        make: APPLICATION
                once
                        create Result.make
                end
end --APPLICATION_ACCESSOR
```

As you can see from above diagram, APPLICATION_ACCESSOR make use of the keyword ONCE in eiffel to create one and only one instance of APPLICATION class.

Following are diagrams (short form and expanded form) for Singleton design pattern in the system.

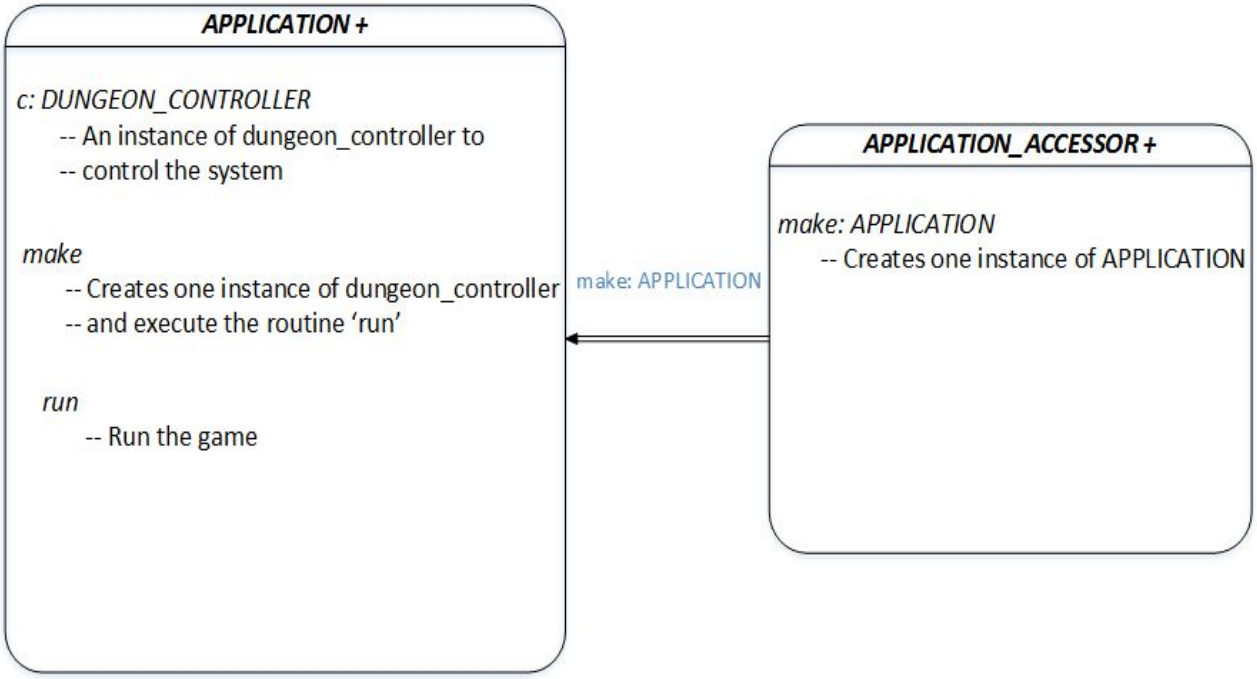

Figure 2: Expanded BON diagram of APPLICATION and APPLICATION_ACCESSOR

Figure 3: BON diagram of APPLICATION and APPLICATION_ACCESSOR (short)

Iterator: In this game user get a set of items at the beginning of the game. This set of items should be stored in a list and can be searched later on during the game. Therefore, we make use of Iterator pattern to make it easier to traverse through the list of items. The class ITERATOR_ON_ARRAYED_LIST[E] inheriting from deferred class ITERATOR acts as the iterator in this system. ITERATOR_ON_ARRAYED_LIST would accept an ARRAYED_LIST[STRING] (since the data structure we are using to store items is ARRAYED_LIST[STRING]) and then it would be easy to iterate through it by using Iterator routines. The following snippet of code shows how we iterate through items of list and prints the items in the list (inventory).

```
do

print("--Inventory-- %N")
        from
                iterator.start
        until
                iterator.is_off
        loop

                v.inven_read_inventory_list(iterator.item.to_string_8)
                iterator.next
        end
        v.inven_read_bit_coin(m.cash)
End
```

The following are the diagrams related to Iterator pattern in the game:

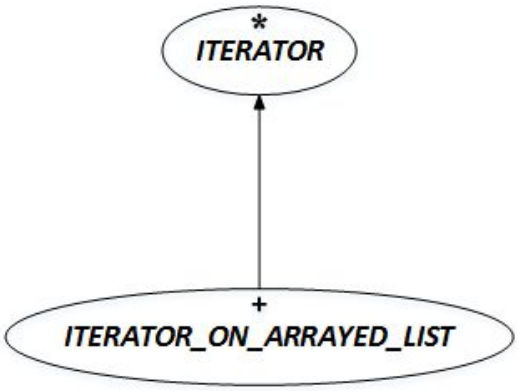

Figure 4: BON diagram of Iterator in the game

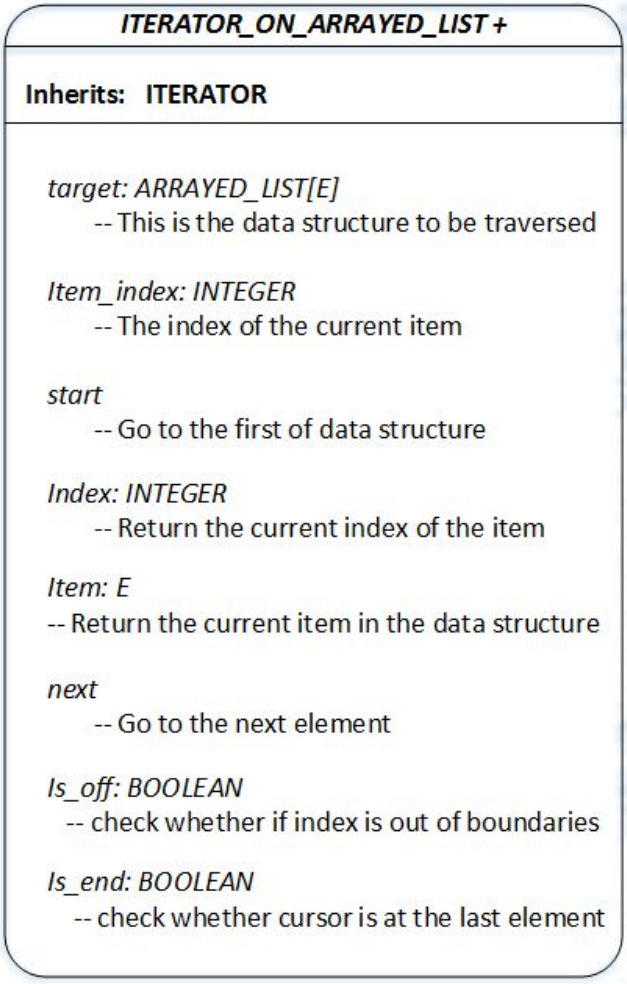

Figure 5: BON class diagram of the ITERATOR_ON_ARRAYED_LIST

Following diagram shows the general picture of the classes in the game.

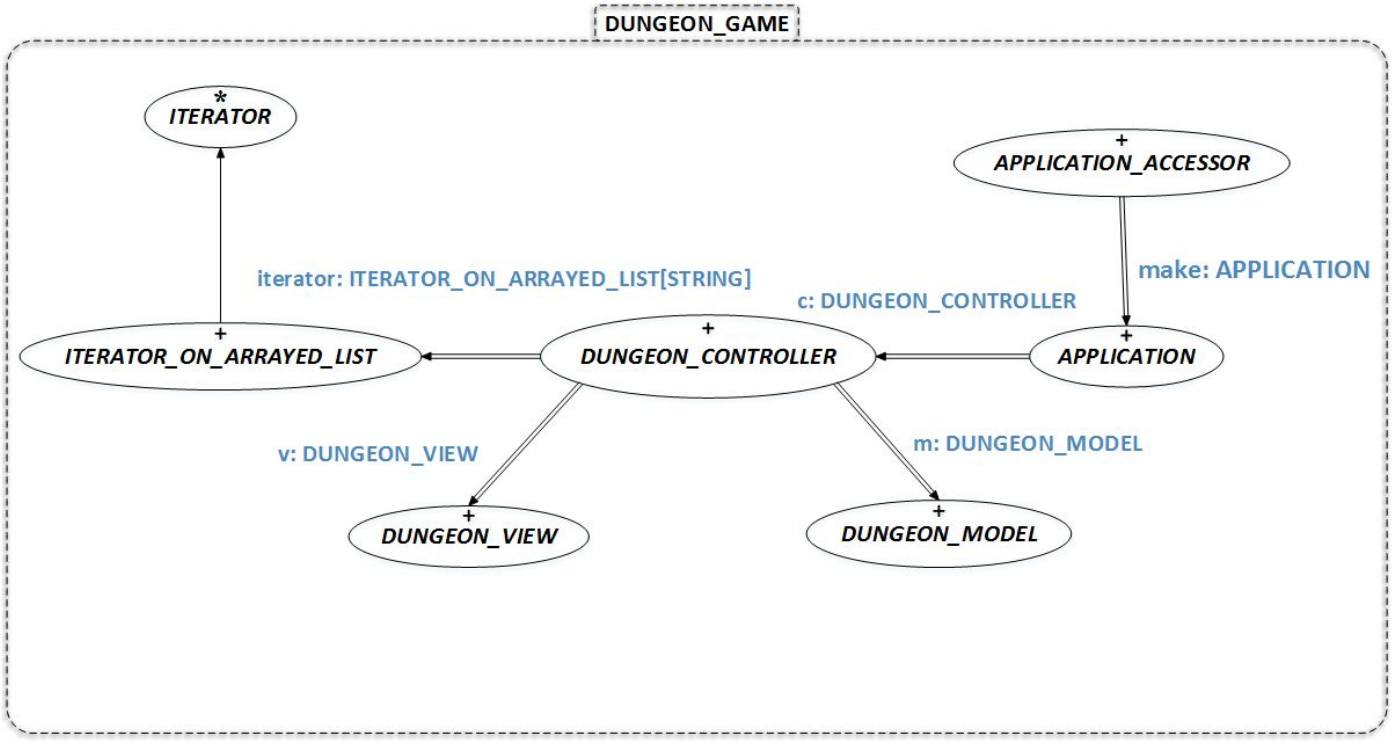

Figure 6: The BON diagram of the game in cluster level

2 Operation on Iterator

2.1 Enquiry operations

These operations retrieve information about the Iterator.

2.1.1 Is the index of the item valid?

is_off:BOOLEAN

-- It returns true if the index of the item is valid.
-- Otherwise it returns false

2.1.2 Is the current item is the last item?

is_end:BOOLEAN

-- It returns true if it is the end of the arrayed list.
-- Otherwise it returns false

2.2 Read operations

There are two read operations in this implementation.

2.2.1 What is the index of the item?

index:INTEGER

-- It returns the index of the item.

2.2.2 What is the item?

item:E

-- It returns the Item.

2.3 Write operations

These operations make changes on information about the Iterator. There is no write operation in this implementation.

2.4 Other

2.4.1 Assign 1 to the index of the first item.

start

-- Make the first item's index equals to 1

2.4.2 Move to the next item and add 1 to the index.

next

--Move to the next item and add 1 to the current index.

3 Contracts

3.3 dungeon_controller.e

purchase
ensure: purchaseno_set_properly: m.get_purchaseno = io.last_integer
-- this feature calls the view function to display purchase options for the
--player to choose from and sets the player's input in the model class in an
-- which could be used accessing the model class for later use.

main_Opt: void
ensure: mainopt_set_properly: m.get_mainoption = io.last_integer
-- this feature calls te view function- mainopt_instruction to display the
--various instructions a player could choose from. This feature also checks
--if the input is valid according to the instructions given to the player and
--saves the input in the model feature- mainOption for later use.

class_Choose
ensure: playerclass_set_properly: m.get_playerclass = io.last_integer
-- This feature calls the view function- class_instruction to display which
--identity a player would like to be recognized as and give the player
--appropriate skill set which would be useful in the game. This feature also
--saves the input and sets it in the model class which could be useful later
--in the program/game.

merchant
ensure: buy_set_properly: m.get_buy = io.last_integer
-- This feature calls the view function- merchant_instruction to display the
--various item the player can purchase and update the player's list of items
--(inventory) according to what the player has chosen to purchase; which
--the player could use later in the game. This feature also checks and
--saves the input.

doBattle(name:STRING; sentHP:INTEGER; sentAttack:INTEGER;
sentDefense:INTEGER): BOOLEAN
require: valid_hp_arg: sentHP > 0 **and** valid_attack_arg: sentAttack >
and valid_defense_arg: sentDefense > 0

-- This feature is all about how the enemy attacks the player, how the
--attack effects the player's status, and when the player attacks the enemy,
--this feature updates the enemy's status accordingly. This feature is called
--when the player encounter's an enemy.

3.4 dungeon_model.e

All of the accessors in the DUNGEON_MODEL class have postcondition to ensure that the returned value is correct. The mutators are also checks that if the correct value was set to the corresponding attributes. The following are examples of an accessor and mutator.

get_system: INTEGER
ensure: result = system

set_mainOption(n:INTEGER_32) : void
ensure: mainOption = n

4 Appendix

Contributions:

Mohammad:

- Contributed in the idea of the game layout- how the game is going to be built:- what challenges could be faced by the player.
- Presented the idea of the design patterns to be implemented in the game and also implemented the controller part.
- Inserted contracts wherever needed.
- Tested the code, fixed errors in the program and made necessary changes.
- Made sure that every step is understandable to the player.
- Contributed in the report - Built the BON diagram for the game, the layout of the report.

Rachel:

- Contributed in the idea of the game layout- the storyline of the game
- implement and separate the model part of the code for the design.
- Inserted contracts and comments.
- Helped fix errors and offered possible solutions.
- Handled exceptions for inputs - making the user aware of it and allowing the user to retry.
- Test the game by playing it and find possible bugs.
- Contributed in the report - handled the introduction and game description.

Azadeh:

- Contributed in the idea of the game layout- what would be the items and accessories of the game to help the player
- Implement the view part of the code.
- Checked overall code and inserted comments where deemed necessary.
- Contributed in the report - Included interface and ADT details in the report, mentioned important methods and contracts of the code
- Helped in testing the final code.