Second Iteration Project (I2): Game of the Goose

Introduction

In this second iteration (I2) we will continue exploring the Game of the Goose as a simple model of Conversational Adventure, as explained in the introduction to this game.

Figure 1 illustrates the modules of the project on which we will work in this new iteration, on the basis of the material developed in the first iteration (I1). As explained in the introduction to the project, it consists on the second approximation to the essential modules of the project.

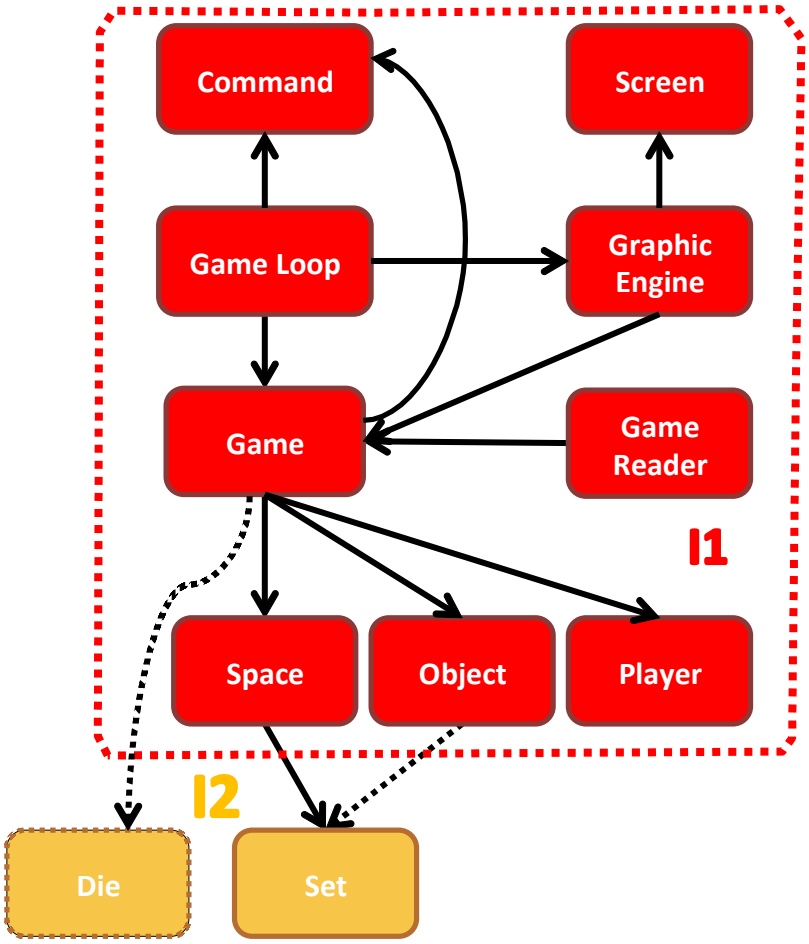

Figure 1. Modules considered in the second iteration (I2) of the project development.

The modules obtained in I1 are shown in red in Figure 1. In yellow we present those modules to be developed in this second iteration. In addition, some red modules will be either expanded or modified. The new modules are two: `Set` and `Die`. The `Set` module will provide the necessary functionality to manage sets of identifiers as those used for objects or spaces. A set is a collection of (non-repeated) components in which the order is irrelevant. Finally, the `Die` module will provide random values within a fix range, like those obtained when we roll a die.

The `Set` module is a good example of a reusable module as it will be used to develop the final application, by providing support to other modules (initially to `Space` and, in further iterations, to `Object` and `Player`). On the other hand, the `Die` module is only necessary for the Goose Game but may not be required for the final project. Its interest relies in the use of functions from the standard C library (such as those for generating pseudo-random numbers and time management).

Starting with the material generated for I1, your project should be able to:

1. Load a game board (spaces) from a data file.
2. Manage everything necessary to implement the game (spaces, player able to carry one object, and location on the board of the object and the player).
3. Support the user interaction with the system by interpreting commands that allow the player to move (ahead and back), to manipulate one object placed on the board (taking it and leaving it) and exiting the program.
4. Move the player through the spaces, allowing to take objects and leaving them, and changing accordingly the status of the game.
5. Show the current position of the player and the adjacent spaces to that one currently occupied, and also specify the location of the object.
6. Free all used resources before the end of the program execution.

In addition to all the functionalities described for I1, for I2 the application should be able to:

1. Load from data files the necessary objects in a similar way to how the spaces are handled.
2. Manage all the extra data needed for implementing the game according to the new requirements.
3. Support interaction with the user. In particular, it should be possible to roll the die and move the player between special spaces (goose, bridge, and death).
4. Display the graphic description of spaces in ASCII (see figure 2), the position for all objects and the last value of the die.

Objectives

The objectives of this second iteration (I2) are twofold. On the one hand, going in Depth int he use of the GNU programming environment (`gcc`, `gdb`, etc.), progressing in the use of techniques and tools for version control, and learning more about the use and design of libraries. On the other hand, practising all these concepts starting with the material developed in I1, modifying the code for improving it and gaining new functionalities.

Improvements and modifications required (requirements R1, R2, etc.) are:

1. [R1] Creating a `Set` module that integrates the functionality needed to manage sets. In particular, every set should be implemented as a data structure with two fields, one to store a vector of Ids, and another to remember the number of them at all times and provided the necessary functions for creating and destroying sets (`create` and `destroy`), add and delete values (`add` and `del`) and print their content for debugging

(`print`). Additionally, you should create a program (`set_test`) to test the module, that should include at least two tests for each function.

2. [R2] Modifying the `Space` module so that it can contain several objects inside (use a `Set` for this task). To achieve this goal, you should substitute, in the `Space` data structure, the actual `object` field by another field (that we will call `objects`) that contains all the objects in the space. In addition to the `Space` data structure, you should modify all primitives using the new field, so they continue working. You should add any necessary function for handling objects properly. For example, functions for adding an object into a space, for identifying objects in that space, or to see if the identifier of an object is in the space.
3. [R3] Creating a `Die` module using ANSI standard C libraries (`stdlib` and `time`) to generate random integers within a range. In particular, the die should be implemented as a structure with an identifier (just in case there is more than one die in the game), two integer values for storing the minimum and maximum values of the range, and an integer to store the number obtained in the last roll, so you can display it when showing the game state. Also in this case, the module should provide the necessary functions for creating and destroying dice (`create` and `destroy`), roll the die (`roll`) and print the die contents (`print`). Additionally, you should create a program (`die_test`) to test this module, that should include at least two tests for each function.
4. [R4] Modifying, if needed, the other existing modules to use the new ones, but maintaining all previously implemented functionality. For example, in the `Game` data structure, replace the existing pointer to `Object` to store the only existing object in the I1, by an array of object pointers to store all those used in the game, as it is done with spaces. In addition, you should add a pointer to `Die` to have a die in the game. In all cases, the game must use the appropriate functions from the new modules to implement the data manipulation primitives.
5. [R5] Adding to the data file (`data.dat`) those data related to the objects that will be used in the game, following the example used to load spaces and using the data line format "`#o:1|*|1`", where object's id, its name and its location are indicated. The new file should include four objects.
6. [R6] Creating a function in `GameReader` for loading objects into the game as we did for loading spaces. Modify the necessary modules so that the objects are loaded from a file at the beginning of the game, as it is done with spaces.
7. [R7] Adding a new command to roll the die (`roll` o `rl`).
8. [R8] Adding two new commands (`left` or `l`, `right` or `r`) to jump among special spaces (geese, bridges, death) for those squares that, in addition to be sequentially linked (north/south), they are specially linked (east/west).
9. [R9] Modifying the command that allows the player to take an object from one space (`take` or `t`), so that we can identify the object we want to take from those available in the space. In particular, this command must include the name or identifier for the object we want to take, for example: "`take o4`" (see Figure 2).

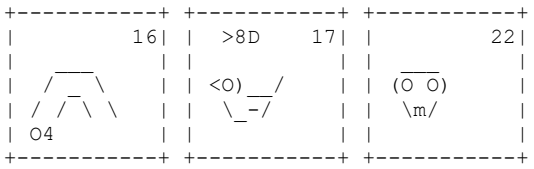

Figure 2. Examples of graphic descriptions (ASCII) space: left bridge with object "O4"; center, goose space and player "> 8D" and right, death space.

10. [R10] Modifying again the `Space` module to include a graphic description (ASCII) using 3x7 characters for the space (see Figure 2). In order to achieve this goal, you may use a field `gdesc` defined as an array composed by three strings, each with seven characters. For example, the graphical description of square 17 of Figure 2 could be:

```
"       "
"<O)__/ "
" \_-/  "
```

And the line of the data file corresponding to that space could be:

```
"#s:17|Square 17 (Goose)|16|-1|18|21|       |<O)__/ | \_-/  |"
```

Again, primitives should be implemented to handle the new field in the existing functions (`create`, `destroy` and `print`) and create the new manipulation functions necessary (`set` and `get`).

11. [R11] Creating a new file with information for loading 25 spaces, all of them sequentially linked (north/south). In addition, the geese squares (5, 9, 13, 17, 21 and 25) must be joined by an additional link (east/west), each one with the following goose. Additionally, the spaces corresponding to bridges (8 and 16) must be linked to each other (also east/west). And finally, the space corresponding to death (22) must be connected (again east/west) with the initial space (1). Moreover, the file must include the graphic descriptions for the geese, bridge, and death (see Figure 2).
12. [R12] Modifying the function for loading spaces to include the new changes in format.
13. [R13] Changing the function that display the game status to show: (a) the graphic description (ASCII) for each space; (b) the space where each object is; (c) several objects in the shown spaces; (d) objects carried out by the player; (e) the last value given by the die; and (f) the last executed command followed by its execution result (OK or ERROR) (see Figure 3). You should take into account that it is possible that you must resize some windows in the user interface.

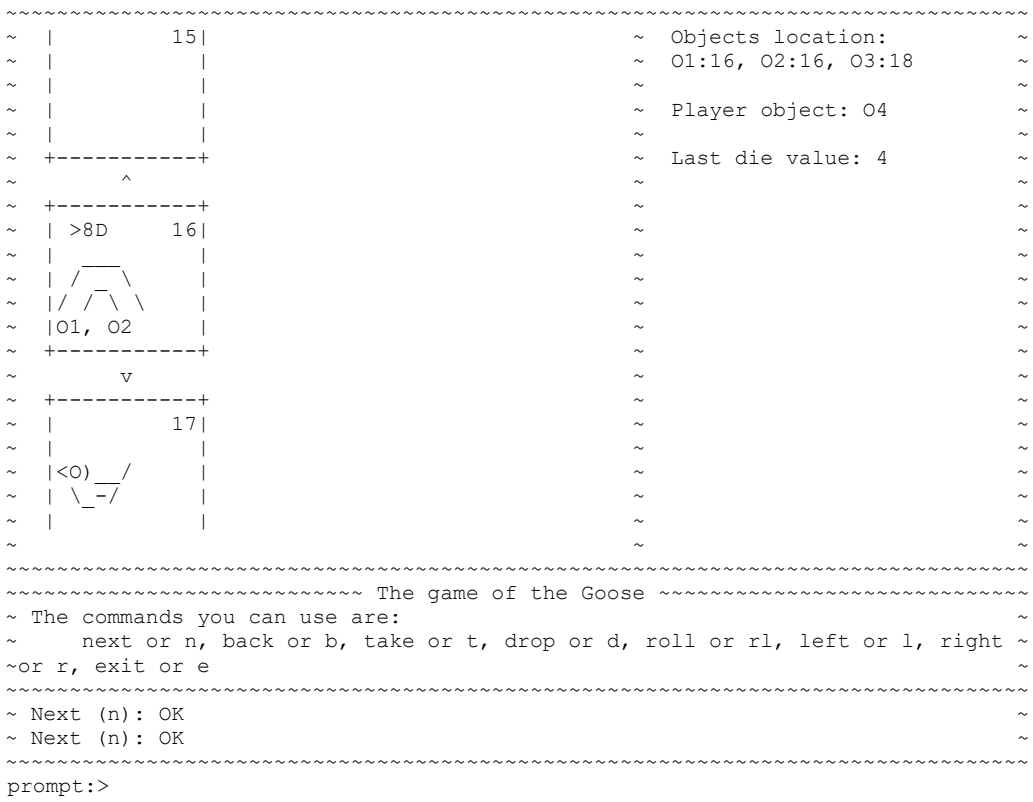

Figure 3. Displaying a game status: top: space 15 which is empty; below, space 16 of type “bridge” with objects "O1" and "O2 and the player “>8D”"; below: space 17 of type goose; right: lines printing information related with: objects location, objects carried by a player and last die value; bottom: the last executed command and their status.

14. [R14] Modifying the main function (`game_loop`) so that it allows to generate a log file (LOG) with traces of execution (apart from its previous functionality). In particular, the LOG file will record a line for each executed command and the result of its execution (OK or ERROR). Thus, if the command was "`take O1`" and it is not possible to take that object, "`take O1: ERROR`" will be written in the LOG file. On the other hand, if the object had been taken, "`take O1: OK`" should be logged. To tell the program that you want to generate a LOG file, the flag `-l` should be added to the command line followed by the LOG file filename. This means that, without LOG, we should invoke the program as usual: "`./game_goose datafile`", where `datafile` is the name of the data file to be loaded, while for generating the LOG file we should write: "`./game_goose datafile -l logfile`", where `logfile` is the name of the LOG file. In order to implement this new functionality the program should: (a) process the input arguments, and if necessary, (b) before starting the game loop, open a file LOG (writing mode) under the name indicated, (c) obtain in each interaction the result of the command execution and write the result in the LOG file as mentioned above, and finally (d) close the LOG file before ending the program to exit the loop game.
15. Modifying the `Makefile` to compile the whole extended project.

16. Debugging the code until proper operation.
17. Modifying the given Gantt diagram in a way that will allow the team organization. The submission will be done during the first week of this iteration. The final version of the activity schedule has to be also submitted, and it will include all the changes done during the weeks that last this iteration to adapt times to the real ones.

Assessment Criteria

The final score for this assignment is part of the final grade according to the percentage set for I2 at the beginning of the course. In particular, the assessment of this deliverable is calculated according to the following criteria:

- **C**: If C is obtained in every row of the rubric table.
- **B**: If at least two Bs are obtained and the remaining rows are Cs. Exceptionally with only one B.
- **A**: If the project gets at least two As and the remaining rows are Bs. Exceptionally with only one A.

Every submission that does not obtain the requirements of column C will obtain a score lower than 5.

Rubric table:

	C (5 - 6,9)	B (7 - 8,9)	A (9 - 10)
Delivery and compilation	(a) All the required files have been delivered on time. AND (b) It is possible to compile and link all the sources to get an executable file using `Makefile`.	In addition to the previous column: Compilation and linking do not report error messages neither warnings when using `-Wall`.	In addition to the previous column: Compilation and linking do not provide error messages neither warnings when compiling with `-Wall -pedantic`.
Functionality	It covers requirements R1, R2, R3 and R4, so that the functionality from I1 is kept.	In addition to the previous column: It covers requirements R5, R6, R7, R8 and R9 having the previous and new functionality.	In addition to the previous column: It covers requirements R10, R11, R12, R13 and R14, having the previous and new functionality.
Testing	The file `space_test` is adapted to the new functionality.	In addition to the previous column: Unitary testing for the Set module (`set_test`)	In addition to the previous column: Unitary testing for the Die module (`die_test`)
Coding style and documentation	(a) Variables and functions have names that help to understand their purpose. AND (b) All constants, global variables, public enumerations, and public structs are documented. AND (c) The code is properly indented[1].	In addition to the previous column: (a) Files and functions include header comments with all required fields. AND (b) Functions have a unique author. AND (c) Module interfaces are not violated.	In addition to the previous column: (a) Coding style is homogeneous[2]. AND (b) There is an explanatory comment for each local variable when it is needed. AND (c) Values for the arguments passed to functions as well as returned values are checked for errors. Resources are properly freed.
Project Management	The Gantt diagram complete.		Final version of the Gantt diagram.

[1] Indentation should be homogeneous. Every code block at the same level must have the same indentation. Besides, either tabulation characters or spaces (always the same number of spaces for each level) must be used, and combinations are not allowed.

[2] At least: the function names should start with the name of the module; the variables, functions, etc. should follow either a camel case or snake case notation, but they should never be mixed; the coding style should always be the same (e.g., K&R, Linux coding conventions, etc.) and never mix different coding styles.