Fourth Iteration Project (I4): Conversational Game (Final Product)

In this fourth iteration (I4) you should complete the project, employing the concepts, skills and tools that have been developed during the course. In this iteration we will first finish the implementation of the basic framework needed to support Conversational Adventures and then an original conversational adventure should be implemented.

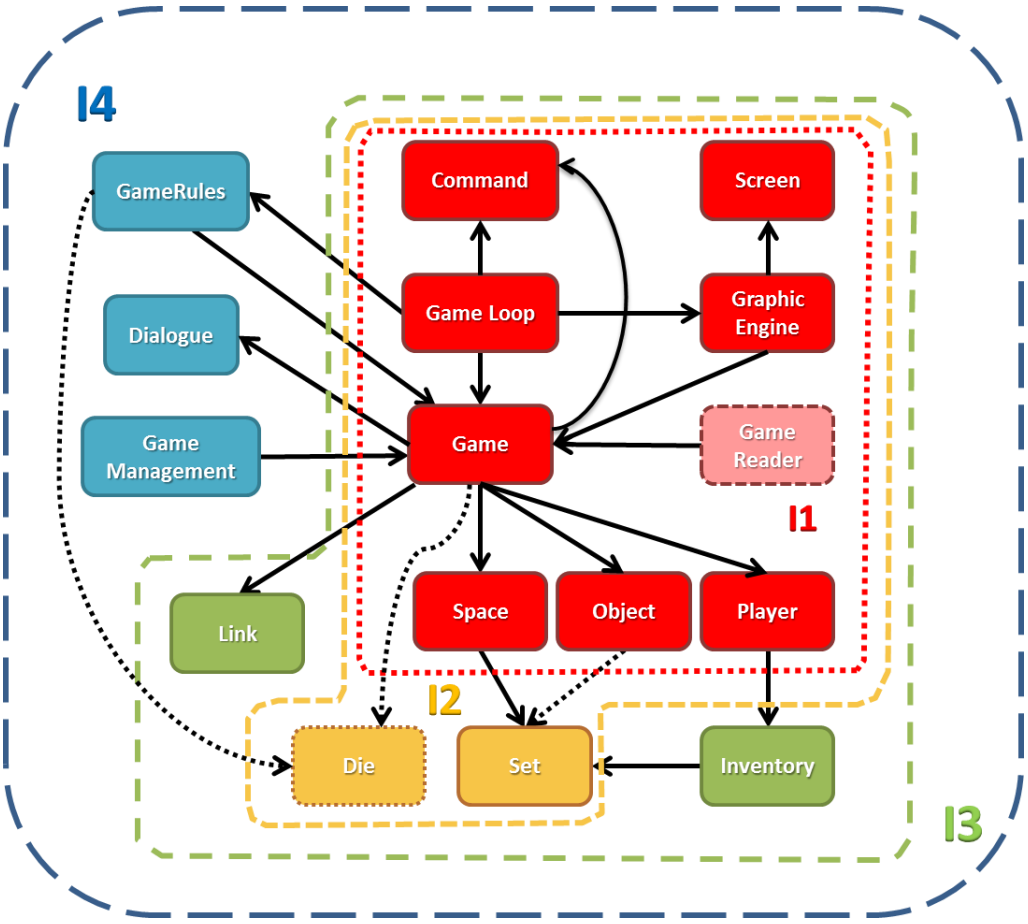

Fig. 1. Modules considered in the fourth iteration (I4) and final project development

Figure 1 illustrates the modules in which we will work on I4 (in blue) as well as the material produced in previous iterations (I1, I2 and I3).

As summarized in the figure, in this iteration we will add another three modules

- *GameManagement*: keeps track of the game status and can save and load it.
- *Dialogue* provides a friendly and natural interface.
- *GameRules* provides the rules that describe how the game status changes during the game

These modules will be discussed in detail in this document.

As result of iterations 1, 2 and 3 you should have code that can:

1. Load spaces, links, objects and players from data files.
2. Manage everything needed to implement basic conversational games using all the elements mentioned in the previous point, as could be the Goose Game.
3. Support user interaction with the system, interpreting commands to: (a) move the player, (b) manipulate objects, (c) inspect spaces and objects and (d) exit the program (besides using a die).
4. Support the generation of a log file that may register the executed commands and the corresponding results.
5. Show the game status at all times including: player position, space descriptions, location of objects on the game map (board), objects in the player's inventory and the last value of the die.
6. Free all resources used before the end of the program execution.

As a result of I4, we expect:

1. An application (ConvGame) that allows the implementation of a Conversational Adventure using the system developed along the course and the extensions incorporated in this latest iteration. The program should use all the features developed in the previous iterations conveniently corrected, improved and expanded.
2. An adventure that contains at least 10 spaces and 20 objects, with their corresponding data files (spaces, links, objects, players, etc.)
3. To correct, improve and extend the functionality of the starting platform in accordance with the requirements that are indicated below and the particular needs of the adventure you are going to develop.
4. A user guide including: (a) all the information needed to play the game, (b) a complete example that shows all the functionalities developed. The example should include instructions on how to navigate from the first space to the last one as well as a file of commands to follow that path (use the format used by the "*.ent" files provided for testing I3).
5. Programs (test for each module, named as "xxx_test.c" where xxx is the module name) and documentation for all modules describing the design as well as test reports.
6. Makefile that compiles the project and generates the documentation: (a) compile with "-Wall -ansi -pedantic" and "-g"; (b) do not forget to compile and link each module along with its testing program ("*_test); (c) manages the project files (it should at least delete the objects and executables).
7. Technical documentation in HTML format generated by doxygen.
8. Project management during I4, using Gantt diagrams and meeting minutes.

Objectives of the Fourth Iteration and Final Project

The purpose of this fourth iteration of the project (I4) is to put into practice the skills learned during the course on teamwork, project management, use and design of libraries, programming, testing, debugging, documentation, etc.

The improvements and modifications required (Requirements R1, R2, etc.) as well as the activities and tasks to be performed are:

1. [R1] Modify the Space module to allow spaces to be illuminated (or not). Object/Space descriptions can only be read in illuminated space. In order to keep record of the "illumination state" add a new Boolean field to the corresponding data structure and implement the necessary methods to manipulate its values (like set and get functions). Add the illumination state to the print function.

2. [R2] Modify again the Space module so that it incorporates, in addition to the connections to the four cardinal points ("north", "east", "south" and "west") other two "up" and "down" for maps with various levels (floors). Implement/modify the necessary functionality to manipulate and print these fields.

3. [R3] Modify the Space module to include a detailed description. The former description should be kept. Use the former description to describe the current space when the game state is shown and use the new (more detailed) description as output of the "inspect space" command. Implement the necessary modifications to manipulate and print the new fields.

4. [R4] Modify the command to examine the current space implemented in I3 ("inspect space") to display the new detailed description if the space is illuminated or nothing otherwise.

5. [R5] Modify the Object module so that it incorporates support for the following properties (do not forget to add/modify the corresponding functions for handling and printing them):

    a. **Movable** indicates whether the object can be moved from its original location. Use a Boolean field that should be added to the corresponding data structure. By default objects cannot be moved. The player may only take items for his inventory if they are movable.

    b. **Moved** indicates whether a movable object has been moved from its original location. Use a Boolean field added to the corresponding data structure. By default objects are not moved. The content of this field is ignored if the object cannot be moved.

    c. **Hidden** indicates whether the object is hidden. By default objects are visible. The space description where the player is located will only show information about visible objects, although you may handle (take, drop, inspect, etc.) them normally.

d. **Open** indicates whether the object can open a particular link specified by its ID. By default, objects cannot open links, in this case the default value is NO_ID.

e. **Illuminate** indicates if the object can illuminate a space. Add a Boolean field to the corresponding data structure, which is TRUE if the object can illuminate and FALSE otherwise. By default, the objects do not illuminate.

f. **TurnedOn** indicates if an object that can illuminate a space is on or off. Add a Boolean field to the corresponding structure. By default, set the value to FALSE. The value may be switched to TRUE only for objects that can illuminate.

6. [R6] Modify the Object module so that in addition to the existing description it incorporates an alternative one. This new description will be displayed when the object is not in its original location. Implement all the necessary modifications in order to manipulate and print this new field.

7. [R7] Modify the command to inspect objects ("inspect <obj>", where <obj> is the name of an object) to display the appropriate description depending on whether the object is (or not) at its initial location or in another (but illuminated) space.

8. [R8] Create the necessary commands to switch on and off objects that can illuminate ("turnon <obj>" and "turnoff <obj>", where <obj> is the name of an object). For example "turnon lantern" or "turnoff torch".

9. [R9] Add a new command to open links with objects ("open <lnk> with <obj>", where <lnk> is the name of the link and <obj> the name of the object). For example, "open door with key" or "open wall with TNT".

10. [R10] Modify, if necessary, the data files corresponding to the modified modules and all modules affected by the changes, e.g. loading functions of spaces and objects from files.

11. [R11] Create an adventure that includes at least 10 spaces and 20 objects, with a story line and its corresponding data file (including spaces, links, objects, player, etc.).

12. [R12] Create a user guide including:

    a) all the information needed to play the game;

    b) a map of the game including spaces, links and location of objects, similar to the example included for Goose Game in I3;

    c) information on how to go from the first space to the last one. Follow a path that shows all the features and functionalities implemented; and

    d) a file with the list of commands to follow the path described in (c) following the format used by the ".ent" files provided to test the I3 program.

13. [R13] Implement a GameManagement module. Rename the module GameReader as GameManagement and modify it so in addition to loading data for initializing a game it may also save the current game state and reload it later. Create a function for saving the game state (gamemanagement_save) and another for loading it (gamemanagement_load). The game_save function should store in one or more files (whose name should be parameterized) the current game state, that is, the content of the Game structure. Regarding gamemanagement_load, the goal is to fill the Game structure with the saved data. Both functions should be able to use a filename as a parameter.

14. [R14] Add two new commands to allow the user to save and load games. The command to save a game (save) should allow the users to supply the name of the file or files where the game will be saved. The command to load the games (load) should allow users to supply the name of the file or configuration files with the data to be read.

15. [R15] Create a Dialogue module. This module should satisfy the following requirements:

    a. For each command executed by the user, the system should show a sentence that informs users if this command has been executed successfully or not. For example, if the user types the command "go west" and execution is successful, then a sentence like "*You've gone west. Now you are in <space_description>.*" (where <space_description> is the description of space) should be shown. If the command has not been carried out successfully, the system will produce a message like "*You cannot go west. Try another action.*"

    b. The module should check if the user has tried to execute the same command two consecutive times or if he has tried to execute a command that does not exist. In these cases, the system should answer in the first case with a message such as "*You have tried to do this before without success.*"; and in the second case with a message such as "*This is not a valid action. Try again*."

    Students may add as many rules as they wish to the dialogue module. The minimum number of rules is one per command plus one to deal with command repetition and another to detect wrong/inexistent commands.

16. [R16] Implement a GameRules module. To give a non-deterministic aspect to the game, in addition to user actions, you will implement actions run by the game itself. Based on the Command module with user commands, you must implement a GameRules module adding actions that may be executed by the game without any user intervention, such as lighting some areas, close or open certain links, change the links of some space, hide or relocate an object, etc. In order to implement this feature, after running a number of user instructions a random number (using the die module) will be generated that will activate one of the game rules. Add a special rule called "NO_RULE" so that some calls to the GameRules module do not have any effect. Students may add as many rules as they wish, with a minimum of six of their choice.

17. In addition to the above requirements, the students must perform the following activities:

    a. **Modify, if necessary, those modules affected by the introduced changes**. Be careful to maintain the previous functionality and incorporate the new proposed one.

    b. **Implement and/or complete the tests programs** as well as the test reports for all the modules.

    c. **Modify the Makefile** file in order to incorporate the new modules/programs.

    d. **Debug the code** until it works.

    e. **Document the new source files and update the previously existing**. Update the HTML technical documentation with Doxygen.

    f. **Manage the project during I4**, performing meetings (documenting them with meeting minutes that include agreements for the team members, tasks assignment, schedule and delivery conditions), schedule for the project iteration (tasks, resources, times, chronogram with Gantt chart), as well as monitoring that schedule with the corresponding modifications, if necessary, written in the meeting minutes and chronograms.

Criterion correction

The contribution of this practice to the final mark is 40%. The mark of this deliverable is calculated according to the following criteria:

C: If C is obtained in all columns.

B: If obtained, at least four Bs and the rest Cs. Exceptionally with only three Bs.

A: If you get at least four As and the rest Bs. Exceptionally with only three As.

Any submission that does not satisfy the requirements of column C will obtain an score lower than 5.

RUBRIC	C (5-6,9)	B (7-8,9)	A (9-10)
Compilation and delivery	(a) All the required files have been delivered on time (b) It is possible to compile and link all the sources in order to create the game and the test programs using Makefile.	In addition to previous column: (a) The Makefile to be delivered allows to manage the project files (cleaning temporary and executable files, generating TGZ with sources, data, and makefile, usage help, etc.) (b) Compilation and linking do not report error messages neither *warnings* using `-Wall` flag	In addition to previous column: (a) Compilation and linking do not provide neither error messages nor *warnings* using `-Wall -pedantic` flags (b) It is possible to produce the technical documentation using Makefile and Doxygen under the default task.
Functionality	Requirement from R1 to R12 are satisfied	In addition to previous column: a) Requirements R13 and R14 are satisfied. alternative requirements may be agreed with the teacher b) The adventure game works making use of the above mentioned modules	In addition to previous column: (a) Requirement R15 and R16 are satisfied b) The adventure game works making use of the above mentioned modules
Tests	At least two unit tests have been implemented for each functions of every implemented module.	At least two unit tests have been implemented for each functions of every implemented module.	At least two unit tests have been implemented for each functions of every implemented module.
Coding style and documentation	(a) Variables and functions have names that help to understand their purpose (b) All constants, global variables, public enumerations, and public structs are documented (c) The code is properly indented[1] (d) All files and functions have been properly commented. In particular, the author field should be unique	In addition to previous column: (a) Module interfaces have not been violated. (b) Doxygen comments implemented for: • File headers (all files). • Function headers (public functions). • Struct and similar data structures (public) (c) Check for error in arguments and returns for all functions used for resource allocation or update.	In addition to previous column: (a) Coding style is homogeneous[2] (b) There is an explanatory comment for each local variable when it is needed (c) It is possible to produce the technical documentation in HTML using Makefile and Doxygen
Project Management	At least one meeting report is submitted describing the work organization for I4. The report should include a Gantt diagram.	In addition to previous column: At least two meeting reports are submitted describing the work organization at the beginning and at the end of I4. The reports should include a Gantt diagram.	In addition to previous column: One meeting report per week is submitted describing the work organization. The reports should reflect and justify the changes in organization as the work progresses. The reports should include a 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.