Fourth Iteration Project (I4): Conversational Game

Introduction

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, each team could add every functionality they consider important for its original conversational adventure.

Figure 1 illustrates the modules in which we will work on I4 as well as the material produced in previous iterations (I1, I2 and I3). This time the development of the basic modules will be finished while other special functionalities could be added according to the designed game.

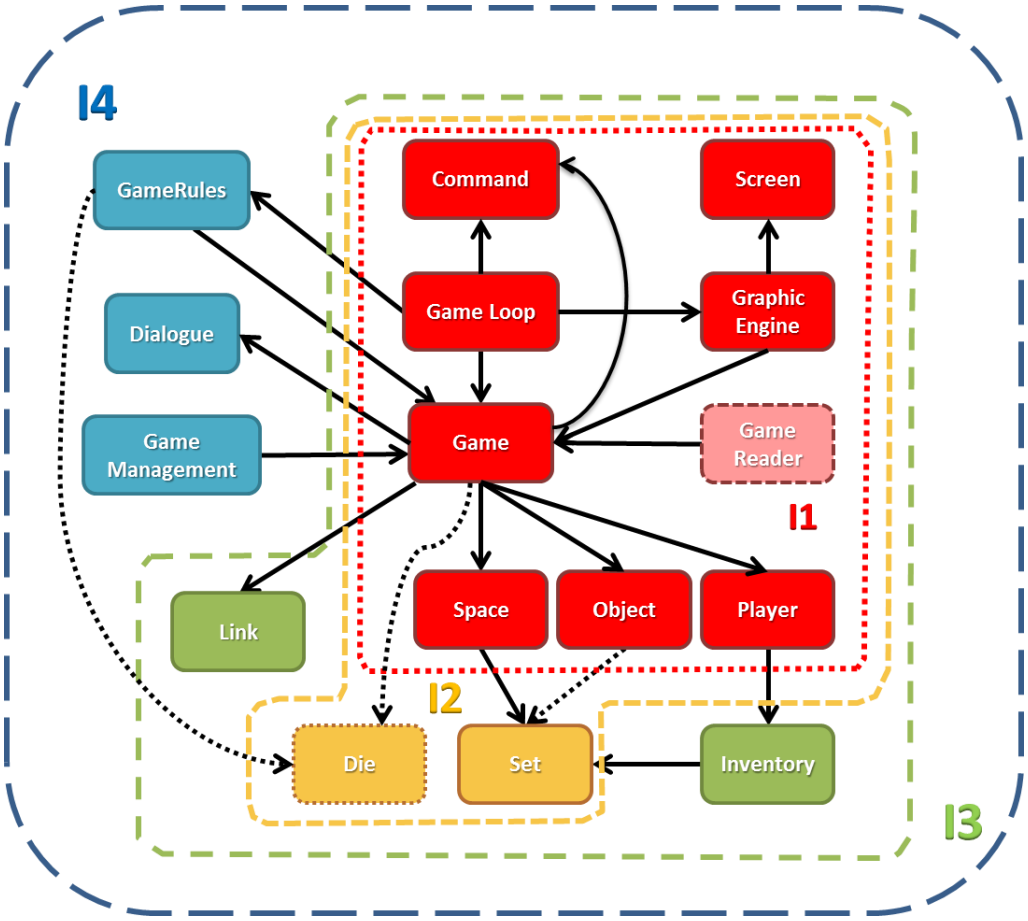

Figure 1. Modules considered in the fourth iteration (I4), corresponding to the final project development.

The modules obtained from I1 are represented in red in Figure 1. In yellow appears the modules developed in I2. In green are shown the modules resulting from I3. Finally, in blue are depicted the modules that could be developed after I4, in which we will also complete and used the modules from previous iterations (red, yellow and green).

Three examples about possible modules that could be added in this iteration are shown in the figure. The first one is `GameManagement`, that keeps track of the game status and can save and

load it. The second is `Dialogue` which provides a friendly and natural interface. Last, `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 every moment: 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 (including 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 all the information needed to play the game, a map, instructions on how to navigate from the first space to the last one appreciating all the developed functionalities and characteristics of the game, and a command file to follow this path.
5. Test programs for each module (`*_test.c`) and documentation for all modules describing the design as well as test reports.
6. Makefile that manage automatically the project: (a) exigent compilation parameters (`-Wall -pedantic`) and for debugging (`-g`); (b) compilation and linkage of each module along with its testing program (`*_test`); (c) automatic generation of the technical documentation.
7. Technical documentation in HTML format generated by Doxygen.
8. Project management during I4, using Gantt diagrams and meeting minutes.

Objectives

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 and space descriptions can only be read in illuminated space, and it will be necessary to illuminate them somehow to have access to that information. For doing this, it should be added a new Boolean field to the corresponding data structure and implemented the necessary methods to manipulate its values (like `set` y `get`) and print its content for debugging (`print`).
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 functions to manipulate and print these fields. In this iteration we will maintain the command `move`, but we will erase the commands `back`, `next`, `left` y `right`.
3. [R3] Modify the `Space` module to include a more detailed description that the existing one. The former description should be kept. Use the former description to describe the current space when the game state is shown (it will appear always) 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 new 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 new Boolean field in the corresponding data structure. By default, objects CANNOT be moved. The player may only take items for his inventory if they are movable.
    b. `dependency`, for indicating if the object depends on another object for taking it, for example, we need to have in the inventory "`torch`" for taking "`fire`". The new field will be of type `Id` and an object can only depend on one object. By default, objects have NO dependencies, so it takes the value `NO_ID`.
    c. `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`.
    d. `illuminate` indicates if the object can illuminate a space. Add a Boolean field to the corresponding data structure, which will be `TRUE` if the object can illuminate and `FALSE` otherwise. By default, the objects do not illuminate.
    e. `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] Create the necessary functions for deciding if an object can be taken or not based on its dependencies. It should be taken into account that if an object is dropped, the objects that depend on it should be also dropped. Modify the corresponding callbacks that call these functions.

7. [R7] 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`") in a similar way as done previously for take an object or move the player in between spaces.
8. [R8] 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`" o "`open wall with tnt`") following the example of the previous developed commands.
9. [R9] 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.
10. [R10] 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.).
11. [R11] 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. Instructions for going from the first space to the last one following a path that shows all the features and functionalities implemented.
    d. A file with the list of commands to follow the path described before. An example of this file could be:

```
move north

move west

inspect space

inspect lantern

take lantern

turnon lantern

move west

drop lantern

…

exit
```

12. [R12] 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 (`game_management_save`) and another for loading it (`game_management_load`). The `game_management_save` function should store in one or more files the current game state, that is, the content of the `Game` structure. Regarding `game_management_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.

13. [R13] 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 info in the `Game` structure 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.
14. [R14] Create a module `Dialogue`. To simulate a more friendly interaction between the user and the computer across the conversational adventure, this module should satisfy:
    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 "`move west`" and the execution is successful, then it can be shown a sentence like "`You've moved west. Now you are in <space_description>.`" where `<space_description>` is the description of the space. If the command has not been carried out successfully, the system will produce a message like "`You cannot move 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 done 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.
15. [R15, OPTIONAL] Create a module `GameRules`. 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 can 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, relocate an object, etc. In order to implement this feature, after running a certain number of user instructions, a random action is executed (using the `Die` in `Game`). 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.
16. In all the previous cases, in addition to the above requirements, the students must perform the following activities and tasks:
    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. Also implement integration tests for checking the correct global performance of the game. For doing so, some files like the one described in the requirement R11d. can be used, in such a way that they work as functionality demos, following the guidelines given in I3.

c. **Modify the `Makefile`** file in order to incorporate the new modules/programs, and to automate the project compilation and linkage.
d. **Debug the code** until it works correctly with the modified and new modules.
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.

Assessment Criteria

The final score for this assignment is part of the final grade according to the percentage set for I3 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 four Bs are obtained and the remaining rows are Cs. Exceptionally with only three Bs.
- **A**: If the project gets at least four As and the remaining rows are Bs. Exceptionally with only three As.

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

The optional module (R15) or any alternative to it (not required by the teacher, but accepted by him), will be assessed to raise the grade once the column A of the rubric table has been reached.

Rubric table (with some footnotes):

	C **(5 - 6,9)**	**B** **(7 - 8,9)**	**A** **(9 - 10)**
Compilation and delivery	a) All the required files have been delivered on time. AND (b) It is possible to compile and link automatically all the sources to get the game and tests executable files using `Makefile`.	In addition to the previous column: (a) Compilation and linking do not provide error messages neither warnings when compiling with `-Wall -pedantic`. AND (b) The files follow the same directory's structure suggested in I3.	In addition to the previous column: (a) The delivered `Makefile` produces the technical documentation using Doxygen under a default task.
Functionality	Requirements from R1 to R11 are satisfied.	In addition to the previous column: (a) Requirements R12 and R13 are satisfied. AND (b) The adventure game works making use of the above-mentioned functionality.	In addition to the previous column: (a) Requirement R14 is satisfied. In professor's orders R15 or an alternative one must be completed. AND (b) The adventure game works making use of the above-mentioned functionality.
Tests	At least two unit-tests have been implemented for each new function of `Space` and `Object` modules.	In addition to the previous column: At least two unit-tests have been implemented for each new function of every module involved in requirements R11 and R12.	In addition to the previous column: (a) At least two unit-tests have been implemented for each new and modified function of every involved module in requirement R15, and if some optional module has been implemented, it should have tests for all its functions. AND (b) Modify the integration tests for checking also the new functionality introduced.

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]. AND (d) Source files and functions contain a header, are correctly commented, and include all the required fields, including data about unique author.	In addition to the previous column: (a) Module interfaces are NOT violated. AND (b) Doxygen comments are included in: • EVERY file header. • Function prototypes. • Enumerated types and data structures. AND (c) Check for errors in arguments and returns of functions used for resource allocation or update.	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) Technical documentation is correctly generated in HTML format using Doxygen.
Project Management	At least one meeting report is submitted describing the work organization of the team for I4. The report should include a Gantt[3] diagram showing the team planning, that must be submitted during the first week of this iteration.	In addition to the previous column: At least two meeting reports with Gantt[3] diagrams are submitted describing the work organization at the beginning and at the end of I4.	In addition to the previous column: One meeting report with Gantt[3] diagram is submitted for each week of the duration of I4. The reports should reflect and justify the changes in organization as the work progresses.

[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.

[3] Gantt charts should cover the complete iteration period and include all the tasks that are planned or that had been done during that period, indicating the designated person to each task.