

CSIS0396A Programming Methodology and Object Oriented Programming (Class A) Assignment 3

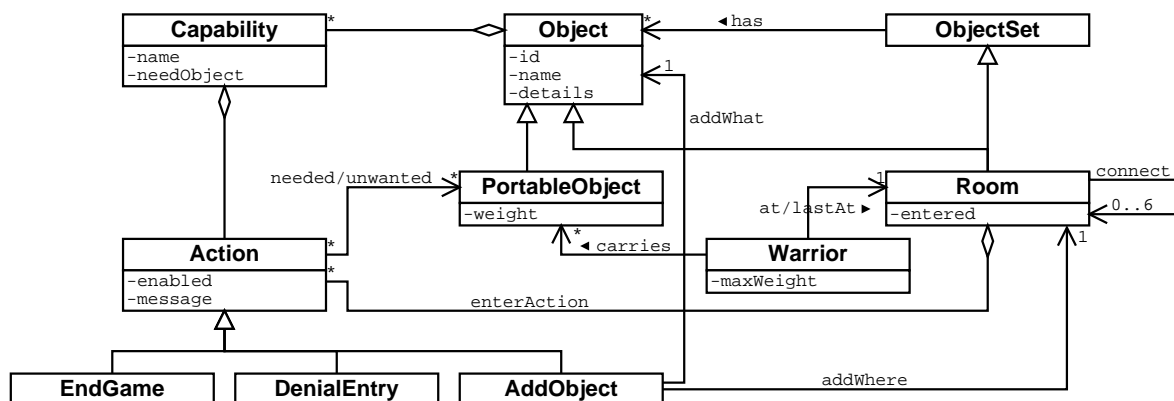
Tutor: [yycau@csis.hku.hk](mailto:yycaw@csis.hku.hk), Deadline April 28, 2002, 11:59pm.

Create a CVS repository and do the following mini-projects in it, using the project name `adventure` as the CVS directory name. It should have a Makefile, which must be created at the beginning of the mini-project. Make sure that you start using CVS at the beginning of the project, and keep meaningful log message throughout. Handin your CVS repository via the hand-in system of our department.

Be reminded to create an empty directory, check-out all the files from the repository and check whether you can make the programs there before you hand-in.

1. Overview

This question requires that you write a text-mode Role Playing Game (RPG). It reads a “maze” from a file and allows the user (player) to explore in the maze¹. The following class diagram is suggested as a reference:



The maze contains **rooms**², interconnected with each other. At any time, the **warrior** (player) is at one of the rooms. A room is an **object**, i.e., *something which has an Id, a name and some detailed information* (the Id is unique within the program, and contains no space for easy reading). Any time the warrior enters a room, the name of the room is shown. The player can issue a command “look”, which shows the detailed description. This also happens when any room is first entered. The player can use movement commands “north”, “south”, “east”, “west”, “up” and “down” to move around the maze.

A room can also contain other objects. We will make the room a set of objects to implement this. Some objects are **portable**, allowing the warrior to get them using the “get” command (e.g., “get shovel”), walk with them and drop them into another room using the “drop” command (e.g., “drop shovel”). When the player enters a room with *portable* objects (or looks at the room), the names of these objects are displayed after the description of the room. A command “inventory” allows the player to see the names of portable objects that the warrior is carrying. The player

¹It is inspired by the Emacs `dunnet` game (type `M-x dunnet`). But don't spend too much time playing that game before you finish this program.

²Here, room is a concept which may represent any place, which is not necessarily a room in our common sense. E.g., it might be a road, a junction, a cave, etc.

can also issue a “look” command (e.g., look tree) to see the detailed description of an object (not necessarily portable) that the warrior carries, or is in the room where the warrior situates at.

Each object may have a set of **capabilities**, which defines *new commands* that can be executed when the warrior is at the object (if the object is a room), at a room containing it, or carrying it. Each capability has a **name**¹ and an associated **action**, to be performed if the command is issued. Also, one may or may not need to tell the name of the object when performing the action (e.g., “shake tree” vs. “dig”). A room can also have actions that are performed when entered.

An action consists of three parts: a list of objects that **must be carried** by the warrior before the action can be done, a list of objects that **must not be carried** before the action is done, and a **message** that is displayed when the action is done. Derived classes of actions performs more specialized things: it may exit the program, move the warrior to the place where he enters a room, or adds an object to a room.

2. The Program

Your program should start by **reading the initial configuration from a file**, specified on the command line (e.g., “adventure MyGame”). The format of the file is described in the appendix. Then the program starts reading and executing commands of the player. Commands are lines consisting of one or two words, converted to lower-case before processing. To summarize:

- north, south, east, west, up, down: movement commands. If there is such a connection from the current room in the specified direction, it updates the current and last room of the warrior, and runs the enter action of the newly entered room.
- look, inventory: look at the room and the belongings of the warrior respectively.
- look *object*, get *portable-object*, drop *portable-object*: look at an object (at the room, or carried by the warrior), get an object at the room, and drop a carried object.
- *capName*: execute the capability associated with the room, an object that is at the room, or an object that is carried by the warrior. The action must require no object specifier. If more than one can be executed, any one of them is executed.
- *capName object*: execute the capability associated with the specified object.

To make our program simple, “*object*” is specified using the Id of the object.

3. Hints and requirements

- **Start as soon as you can.** The instructor actually coded up the thing, requiring him 7 hours—with all his familiarity of the C++ language and STL. You can expect to need a few days to complete it.
- Place most implementation code in .cc files, leaving .hh files only for class declaration.
- By using forward declarations like **class TObject**;, try to avoid header files from including non-standard header files. Otherwise, it is easy to get into a situation that two headers include each another, making the program uncompileable.
- Put the main command loop into the definition of the *Warrior* class, where most of the

¹Guessing names of such commands is usually one of the most funny (or frustrating) things in such games)

needed information can be found easily.

- Have a global list of rooms and a global list of (non-room) objects. To make it easy to find a room and an object, they should really be *map*'s from *string* to rooms and objects.
- Use separate header files for defining object, action, capability, room and warrior.
- A sample program and some sample configurations can be found in the web page. Feel free to add features (classes) to the above to make a more meaningful game, but if you do so, please keep the file format compatible (i.e., make sure your program can still load the file with the above format), and submit a sample file that exhibit the added features.
- Read objects by constructing them using an *istream* as argument.
- The input is line-based. So whenever you use the `>>` operator (e.g., to read a number), use *istream::ignore* to skip the remainder of the line. You will also have to use *getline()* frequently.
- Use STL as far as you can. E.g., the *ObjectSet* can be a *Set<Object*>*. This saves you from implementing the details.
- For the *set* template, you probably need to use *insert*, *erase* and *find* function. For the *map* template, probably you just need to use the `[]` operator.
- To iterate through a STL container like a set, use iterator. You also need an iterator (returned by *find*) to remove elements in a set.

Instruction to hand-in the repository: change directory to the repository (not working directory!), and type “`tar czvf ~/a3.tgz adventure`”. Then hand-in the file `a3.tgz` in your home directory.

Appendix A. The file format

The file consists of 6 parts, in the following order: a list of rooms, a list of connections between the rooms, a list of objects, a list of actions for the rooms, a list of capabilities for the objects, and the description of the warrior. Each list starts by a line specifying the number of items in the list. Here are examples for each type of items in the file. The part of the line after the % sign is not really part of the file, they are just to tell you what each line means.

A.1. Room

There is only one type of rooms, so it is very easy to specify.

```
room ns-road-s                % object type and ID
NS-road South end            % object short name
You are at the South end of a long road, % object description
running from North to South. There are % description can span lines
some trees around.
                                % until a blank line
0                               % whether the room has been entered
```

A.2. Objects

There are two types of objects. An (non-portable) object is stored like this:

```
object tree                % object type
Palm tree                  % object name
They are palm trees with a bountiful
supply of coconuts in them. % object description

3                          % Number of rooms with the object
ns-road-s ns-road ns-road-n % Lists the rooms
```

Portable objects are similar, except that it has additional weight:

```
portable-object brass-key % object type
Brass key                  % object name
I see nothing special about that. % object description

0.5                        % Weight of object
0                          % Number of rooms with the object
```

A.3. Connections

A connection allows the warrior to go **from** a room **to** another room using a **direction** command. So it is specified by these three words:

```
ns-road north ns-road-n
```

Such a connection does not imply the reversed connection (from ns-road-n to ns-road using the south command), so if it is needed, one must specify it separately.

A.4. Room actions

An action of the room is specified this way:

```
ns-road-s                % room for this action to be used
message                  % action type
1                        % enabled action
0                        % no object must be carried
0                        % no object must not be carried
Welcome to adventure!    % message to display
Good luck!               % message can span lines
                          % an empty line terminate it.
```

There are 3 other types of actions. End-game action is one which exits the game if executed:

```
ns-road-s                % room for this action to be used
end-game                 % action type
1                        % enabled action
1                        % one object must be carried
gold                     % id of the object
0                        % no object must not be carried
You win the game! Congratulations! % message to display
                          % an empty line terminate it.
```

Denial-entry action is one which put you back to the last room entered.

```
hall % room for this action to be used
denial-entry % action type
1 % enabled action
0 % no object must be carried
1 % one object must not be carried
brass-key % id of the object
You don't have the key needed to open % message to display
the door. % message can span lines
% an empty line terminate it.
```

Add-object action adds an object to a room. See the example in the next section.

A.5. Object capability

Capability is specified by specifying the object which the capability applies to, the capability name and the action for it:

```
fork % object or room with the capability
dig % name of capability
0 % no object name needed
add-object % action type: add a new object
1 % enabled, disable on first execution
1 % number of objects to be carried
shovel % name of carried object
0 % no object must not be carried
You find something here. % message

brace-key % object added
fork % add to where
```

A.6. Warrior

The warrior has description like this:

```
warrior % object type
10 % maximum weight
2 % carrying two objects
lamp shovel % carried objects
ns-road % current room
```