Server-side classes

1 CommandLineUser

The abstract *CommandLineUser* class handles parsing of text commands, operating on the *GameLogic* instance. The class is abstract to enable the actual retrieval and output of text commands and responses to be handled in different ways, with the abstract *run* method designed to do the command retrieval (perhaps with a new thread in future) and *doOutputMessage* to do the response output.

In the event of a network game, a *NetworkUser* class which inherits *CommandLineUser* could be created and instantiated once per player, with a thread, to handle the players.

The *CommandLineUser* class implements the *PlayerListener* interface, to enable it to "listen" to a player class. This will become invaluable in a network game with each instance listening to another player.

2 GameLogic

The *GameLogic* class handles the overall control of the game and has a public interface which matches the available map commands (which by this stage have already been parsed). The class relies on multiple other classes, which handle other functionality and simplify the logic.

The class does not return any responses, except for *clientLook* and *getGoal*. If no exception is thrown, the command is assumed to be succesful and another class is responsible for returning success and handling the exception. Otherwise a *CommandException* is thrown. Note that, the class may operate on a *Player* causing it to inform the *PlayerListener* resulting in a response being output to a player, e.g. TREASUREMOD.

3 CommandException

A checked exception to handle invalid commands, which otherwise parsed successfully, e.g. walking into a wall.

4 Player

The *Player* class handles the state of each player, such as location, hit points, amount of gold, items and action points. In a network game, there would be multiple instances of *Player* but only one instance of *GameLogic* and *Map*. The *Player* is "listened" to by a *PlayerListener* which is sent to the constructor. This enables commands such as MESSAGE and TREASUREMOD to be output in response to something happening to a player or the *sendMessage* method. In a

multiplayer game, the LOSE message could be handled in this manner by changing the *PlayerListener* interface.

The player is also an *GameItemConsumer* meaning that items can operate on the player. This is how gold or hitpoints are added to the player.

5 Map

The *Map* class handles reading the map from a file, and mainains an array of *Tile* instances. The *GameLogic* class uses its methods to query it, e.g. to look up a tile or check if a location is valid.

6 Tile

The *Tile* class represents a Tile on the map, contaning an enum, *TileType*. The class has helper fuctions such as *isWalkable* to handle game logic. A tile may also contain a *GameItem*.

7 Location

The *Location* class stores a 2D location and has two helper methods to assist readability in generating an off set location or location at a compass direction.

8 CompassDirection

An enum to handle the different compass directions

9 GameItem

The *GameItem* class represtents an item which can be picked up from a tile by a player and is inherited by the different items. The item may be "retainable", such as a sword, in which case the player holds the item (and can pick up no more) or non-retainable, in which the item immediately dissapears and the player may collect any number of the same item).

When a player picks up the item, something may happen, through the *processPickUp* method which is executed on the *Player* class instance. This method operates through the *GameItemConsumer* interface, implemented by the *Player* class.

Gold is non-retainable, as the player may hold more than one, but the *processPickUp* methods increments the gold count. Health potion (the *Health* class) is also non-retainable and instantly increments the player hitpoints by one. Currently, the only effect of a retainable item is to increase the player's look distance (as featured by the *Lantern* class), but the interface can be expanded to

support more items. The *Armour*, *Lantern* and *Sword* classes are all retainable, but only the *Lantern* class has an effect as attacking has not been implemented.

10 NetworkUser

Extends *CommandLineUser***.** Facilitates communication between the server and the client subsystems. All information sent/received via a socket on a separate thread.

11 ServerLogic

Listens for clients and initiates a thread running a *NetworkUser* for every arriving client, if it is currently listening. Additionally, reports the currently used IP. Redesigned to utilize an implementation of *ServerUI* to report its output to the user.

12 ServerUI

An interface the implementations of which decides how the output of a *ServerLogic* object will be displayed to the user.

13 ServerPanel

Implements *ServerUI*. Represents server output on a graphical panel and allows for toggling the server state (listening/not listening) and setting the port on which the server will listen. Uses **GameCanvas** to visualize the overview of the map.

14 GameCanvas

Visualizes the current state of the dungeon's map from a **char[][]**. Used by *ServerPanel*. Reworked version of *GamePanel* from client side.

15 Server

The main class of the server side. Creates a window containing a *ServerPanel*. Also cleans-up unclosed *NetworkUser* threads if closed abruptly.