Source Code Document

1 Readme

This is an version of Dungeon of Dooom designed to be ready for coursework two and three. Some extra commands and functionality have been added, such as an lantern and extra items to pick up, but the game is still not network or multilayer ready. The game still supports one player, but the classes have been designed so as to enable easy modification to multiple players. The GameLogic class is not thread-safe, and will have to be protected for a multilayer game, and a turn based system will need to be developed.

The code is also designed as a basis for extending functionality coursework three (excluding the GUI elements, for which separate "view" classes are recommended). New items can be added to the game through extending the abstract GameItem class, and skeleton attacking code.

1.1 Compiling

To compile the source, please use the following command in the directory containing *src* and *bin*:

```
javac -d bin -cp src src/dod/Program.java
```

1.2 Running

```
java -cp bin dod/Program [-b] [map filename]
```

where "-b" specifies that a bot should play instead of the local user e.g.

```
java -cp bin dod/Progam -b example_map
```

2 General Organisation

The classes for handling the Game's internal logic lie in the *dod.game* package. The game is controlled through the *GameLogic* class, which can be viewed as the overall controller. This in turns uses classes *Map*, *Tile*, *Player*, and *Item* and sub-classes in the *dod.game.items* package. In addition, the *CompassDirection* enum and *Location* are used to assist code-readability. Other classes may listen to a player in the game by implementing the *PlayerListener* interface.

Command handling is carried out by the *CommandLineUser* abstract class. This implements the *PlayerListener* interface and hence can "listen in" on a player. This is inherited by *HumanUser*, which allows a user to play the game from the command line, and *Bot*, which plays the game using textual commands instead of operating on *GameLogic* directly (this may seem excessive, but would allow a bot to player over a network).

The game is launched by running the *Program* class, which parses command line arguments from the terminal and instantiates the right classes.

3 Classes

3.1 Progam

The *Progam* class parses command-line arguments, instantiates a new *GameLogic* instance, with a chosen or default map, and instantiates and runs *HumanUser* or *Bot* on the map depending on the command line arguments.

3.2 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.

3.3 HumanUser

The *HumanUser* class inherits the abstract *CommandLineUser* class and uses "stdin" and "stdout" to allow a human user to enter messages on the command line.

3.4 Bot

The *Bot* class is a very basic bot (which moves randomly) which players through textual commands. The commands and responses are output to the command-line so you can see it playing.

3.5 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.

In a multiplayer game, the class will need to be modified to handle multipler users (currently "ENDTURN" and "STARTTURN" occur together). The class is also **not** thread-safe, and will need protection if multiple threads are to operate on it.

Attacking is featured in a method in the class, but has not been implemented.

3.6 CommandException

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

3.7 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.

3.8 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.

3.9 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*.

3.10 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.

3.11 CompassDirection

An enum to handle the different compass directions

3.12 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.