Source Code Document

1 Readme

This is an version of Dungeon of Dooom designed to be used in coursework two. It's intended to be like your own coursework one, but there some extra commands and functionality have been added, such as an lantern and extra items to pick up. The game is still not network or multiplayer ready. It only 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 would have to be protected for a multiplayer game, and a turn-based or real-time system would need to be developed. Note that by *multiplayer* here we mean to include a single human and a single AI.

For the model underlying your GUI in coursework three, you can either use a modified version of this code or of your own coursework one, the choice is yours. This code is designed to be usable as a basis for extending functionality to 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

```
cd bin
```

In the bin sub-directory run:

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

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

```
java 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 subclasses 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.

The *HumanUser* allows a user to play the game from the command line, and *Bot* allows a bot to play 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).

Both *HumanUser* and *Bot* contain identical code, which is obviously bad object oriented design. You should therefore refactor it into a parent class, perhaps called *CommandLineUser*. All shared code and members should be in this class, which should be inherited by *HumanUser* and *Bot*. This class ought to contain an abstract method, outputMessage, since this differs between the two classes, and should not contain a *run* method. In the real world, you may have to carry out a similar refactoring, but the code would not be a direct "copy and paste" replica. When you write a class for serving each client, on the server, you could then inherit from this new abstract class.

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 HumanUser

The uses "stdin" and "stdout" to allow a human user to enter messages on the command line.

3.3 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.

3.4 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 successful 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 multiple 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.

3.5 CommandException

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

3.6 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 hit-points are added to the player.

3.7 Map

The *Map* class handles reading the map from a file, and maintains 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.

3.8 Tile

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

3.9 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.

3.10 CompassDirection

An enum to handle the different compass directions

3.11 GameItem

The *GameItem* class represents 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 disappears 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 hit-points 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.