# Documentation (HOWTO)

This is the documentation for the third part of the Dungeon of Doom coursework. Its purpose is to guide the user (whoever checks and grades the project) in successfully compiling and running the code.

1. Compilation – since it is required that the code is compiled from the shell / command prompt, there are some additional steps that need to be taken as opposed to using an IDE.
   a. Once the shell/command prompt has been opened, change the current working directory to the directory of this file as it contains the project files.

      ```
      cd <your path here>/DoD/
      ```

   b. Depending on whether you're using Windows or Unix/Linux use the following command to compile the code. I have provided a list of all source files in the **sources.txt** so there will not be a problem compiling directories recursively.

      ```
      javac –d bin @sources.txt
      ```

2. Running
   a. Running the client. It takes an optional parameter "size" which determines the size of the UI.

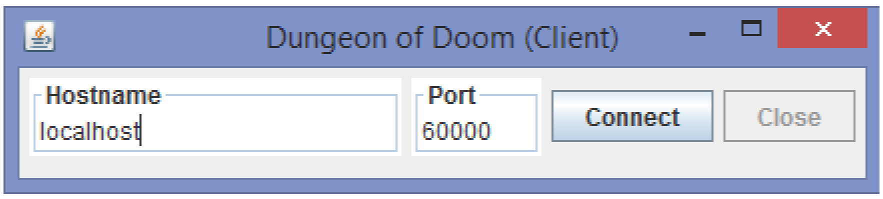
      ```
      javaw –cp bin client.Client [size]
      ```

   b. Running the server. It takes 2 optional parameters – which map to use for this server and what the size of the UI will be.

      ```
      javaw –cp bin server.Server [map file] [size]
      ```

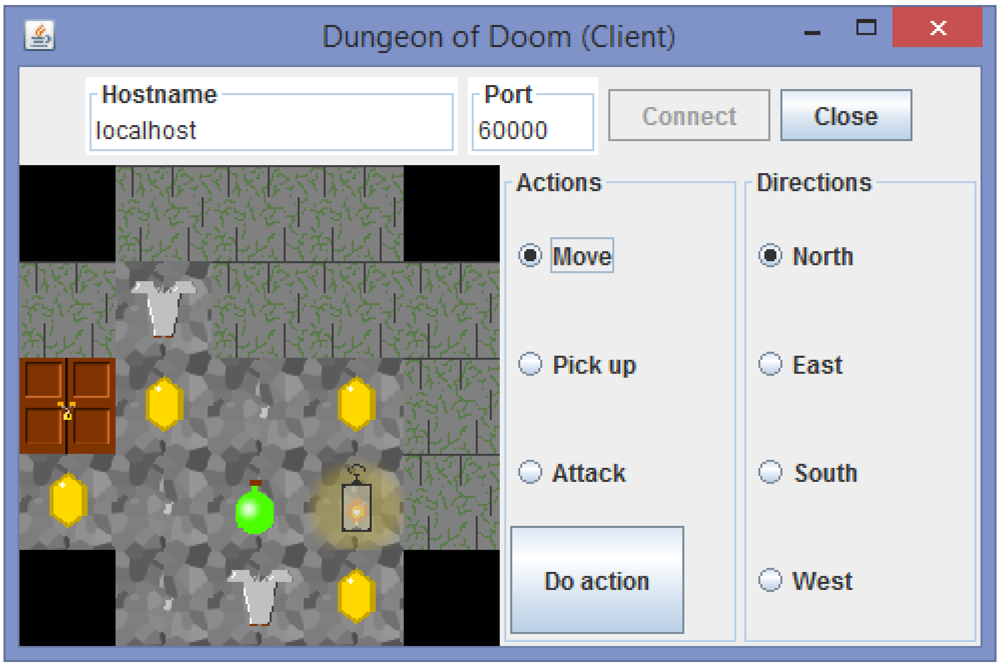
## Screenshots

Client (not connected to server)

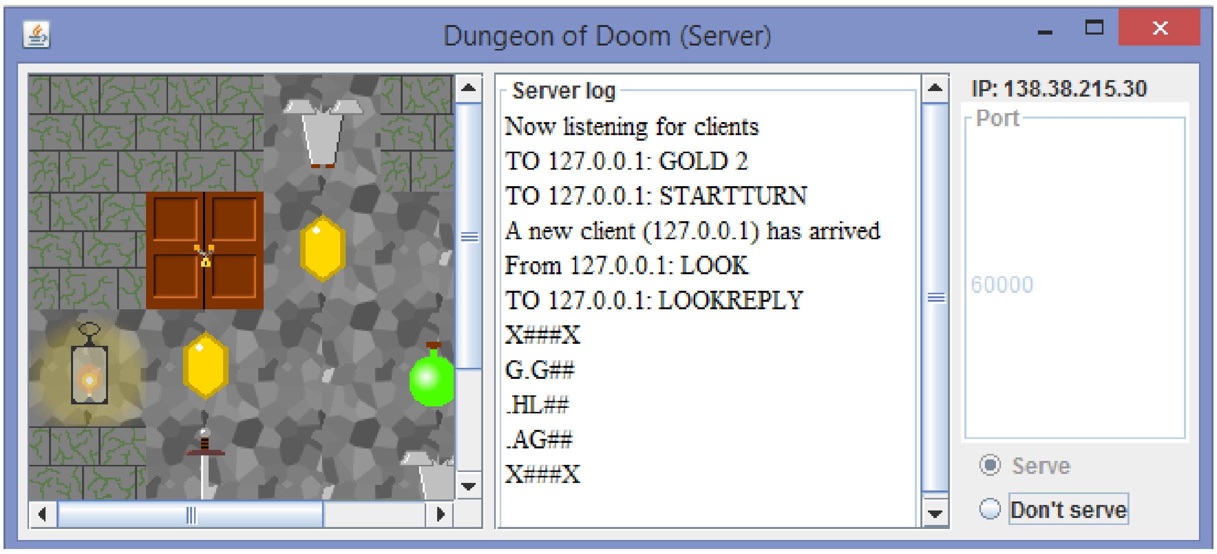


Hostname: localhost
Port: 60000

Client (connected to server)



Server



```
Server log
Now listening for clients
TO 127.0.0.1: GOLD 2
TO 127.0.0.1: STARTTURN
A new client (127.0.0.1) has arrived
From 127.0.0.1: LOOK
TO 127.0.0.1: LOOKREPLY
X###X
G.G##
.HL##
.AG##
X###X
```

## Analysis

While the project as a whole works there is a myriad of flaws of varying impact in the design of the code. The first which comes to mind is the custom GUI classes (**GameCanvas, GamePanel** and **ServerPanel**). All of these are not robust as I'd like them to be due to the fact that they rely on pre-established width-height ratio and resizing setPrefferedSize() them would break their layout. This could have been avoided if I had overridden that method in order to ensure that their subcomponents are always the right size. Moreover, **GamePanel** and **ServerPanel** contain a **GUIClient** and **ServerLogic** objects respectively. At first glance there is nothing wrong with that, but both **GUIClient** and **ServerLogic** contain references to their respective panel object as well.In my opinion this causes high coupling which is an undesired trait. Nevertheless, it had to be this way, since both **GamePanel** and **ServerPanel** have widgets which generate user input to be sent by the client or update the state of the server. On the server side I tried solving this by creating the **ServerUI** interface and made **ServerPanel** to implement it. On the client side, however I had no time to implement a solution, which was:

1. Having a **GamePanel** to store the input generated by its user in a buffer which would be read by additional thread in **GUIClient.**
2. Having **GamePanel** implement an interface called **ClientUI.**

The second part of this solution raised an interesting question. If there is a hierarchy of **ClientUI** derivative classes to be used by **AbstractClient** derivatives, are classes such as **GUIClient** needed? **GUIClient** just specifies how its output should be displayed to the user, but when each **AbstractClient** derivative lets its **ClientUI** decide how to display the output, **GUIClient** loses its purpose as a specialized subset of **AbstractClient**. I guess, it’s a matter of preference and requirements – e.g. if we need a bot client it can still extend **AbstractClient** but it will let its **ClientUI** decide how the output will be handled.