**Slay The Minotaur**

Final Report

Zachary Turner

Michael Ellis

Nathaniel English

Jackson Carlstrom

**Table of Contents**

1. **Introduction………………………………………………………3**
2. **Overall Description……………………………………………3,4**
3. **Software Requirements/Specification……………………..4,6**
4. **Design Process………………………………………………..7,8,9**
5. **User Manual………………………………………………………10**
6. **Introduction**

**Intro**

Video games since their creation have been a source of entertainment for many years. This document is to describe the type of software and modules used to create our project and how we came to create or design it. As all members of this team are gamers, we thought it would be a good idea to make a low version of a video game. The lowest version of a video game we as students could make in the span of time, we had to do this project would have to be a text-based game. So, we all came to the idea that if we had to create a game it would have to be based on something that we have either played or have seen.

**Game-Design**

The game “Slay the Minotaur” like its namesake is a text-based adventure game. Like many other adventure games, the key mechanic is exploration and fighting the occasional mobs or mini bosses before the final showdown. This concept comes from an existing Dungeons & Dragons campaign in which players are supposed to explore their surroundings and find either drops that are either supposed to help or exp to get to the boss.

**Acronyms and Abbreviations**

D&D (dungeons & dragons game)

Drop (items)

Exp (experience)

Mobs (enemy monsters)

GUI (graphical user interface)

1. Game Description

**Story**

The game starts with a lone adventurer ready to fight his way through the labyrinth dungeon to meet the dreaded boss of the labyrinth. If the boss is not defeated and his power grows too powerful, he can rise from the depths of despair and become a harbinger of the end times. As one blessed with both power and strength you must complete this task and destroy the enemies of humanity, or all will suffer. Such an event has happened in the past as one labyrinth was not challenged, and humanity was thrown to the edge of extinction until one blessed with the god’s powers and protection was able to slay the almost infinite hordes of monsters and defeated the dreaded Dragon king.

As the world returned to a peaceful time people never forgot the old stories and how humanity was almost made to face extinction. As a result of this once dreaded and feared era the adventure’s guild was created to train and support the future brave men and women as adventurers. Their secondary mission is to create a guide of the adventure they have on enemies. Finally, to protect the world and them all from destruction.

3.Software Specification/Requirements

**System/Software**

As this is a digital game it can be played on a pc or computer system. The software needed to handle the game is IntelliJ with swing, awt, and other core modules. IntelliJ is an opened source programing tool that one can use to write java programs or learn the core mechanics of java. Swing is a lightweight GUI toolkit that has a lot of key modules needed in the game designing process. Such as picture implementation and external window screens. AWT (abstract window toolkit) is an API that supports the GUI.

**Implementation**

As we use IntelliJ, we have a main that stores and executes the core mechanics of the game. Since it is easier to separate everything from the main and call up the mechanics or events as needed, we implemented subclasses that contain aspects of what any game would need such as enemies’ class and item class. Now, each of these classes has a subclass of their own that specifies what the class is and what they can do. For example: “The enemy class can have a sub of a type of monster like a goblin and what their range of activities they can do”.

**Player**

As an adventurer we have a class that completely controls the adventurer’s actions. Each action is defined as a command that tells you where to go and what to use. Like any standard adventure game, you have a item bag and items that can either increase or decrease your own health or reduce an enemies health points. The item bag stores any items you will come across in the that will aid you in high-risk scenarios. Such as health potions that give a player health if they are dying in the heat of battle.

**Weapons/Armor**

Weapons are standard, as they are what you need to fight the darkness. As a beginner, you have a standard set of weapons, armor, and items to use. Weapons have characteristics like a name, damage modifier, and a description. Armor runs on the same principle as the weapon class, and it just takes a deductible of how much damage is taken by the armor. The reason for this is that both classes are subclasses of the Item class.

**Screenshots**

The following screenshots showcase various classes and methods of the text-based adventure game program, such as how enemies, items, the graphical user interface (GUI), and the enemy subclasses. As can be seen in Figures 1 and 4, the Enemy superclass defines all parameters for the enemies, such as name, health, minimum and maximum damage, experience value, and description. These parameters are then used as arguments in the subclasses, such as in the Minotaur class. This allows for much easier implementation of various enemy archetypes that can then be instantiated as objects of the Enemy type in the main program.

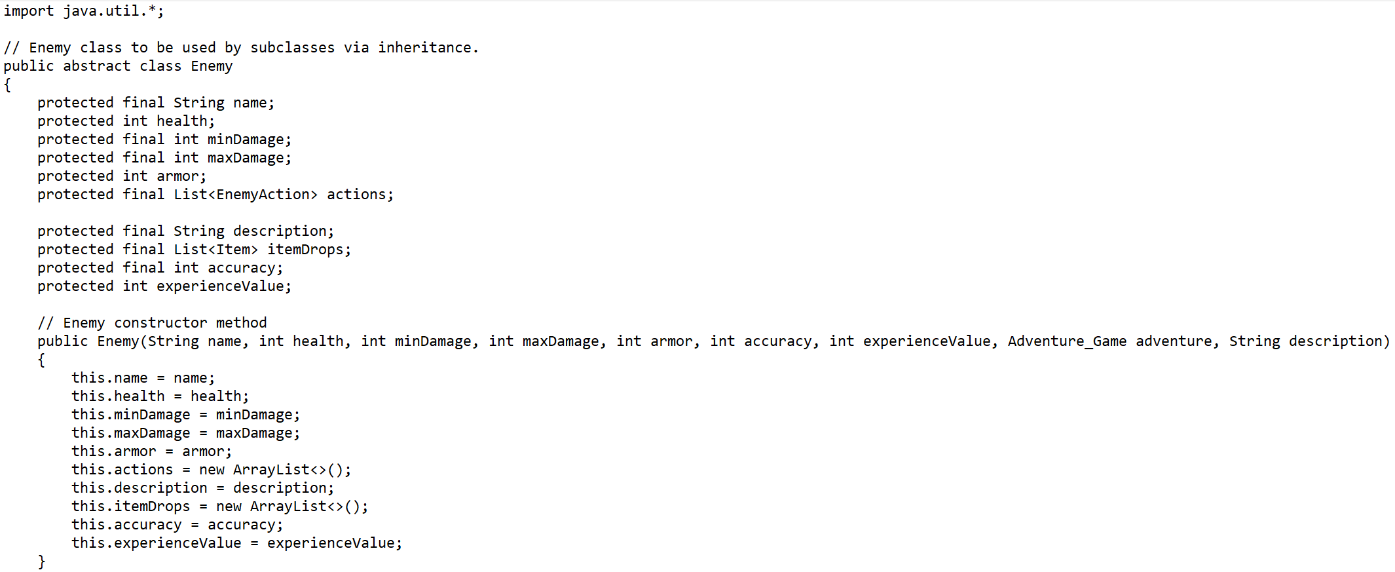
****

Figure 1 – The Enemy superclass and Enemy() constructor method that defines the main methods to be used for Objects of the Enemy type.

Graphical user interface, text, application

Description automatically generated with medium confidence

Figure 2 – The Item class, which implements the Item\_Effect class for using items in-game.

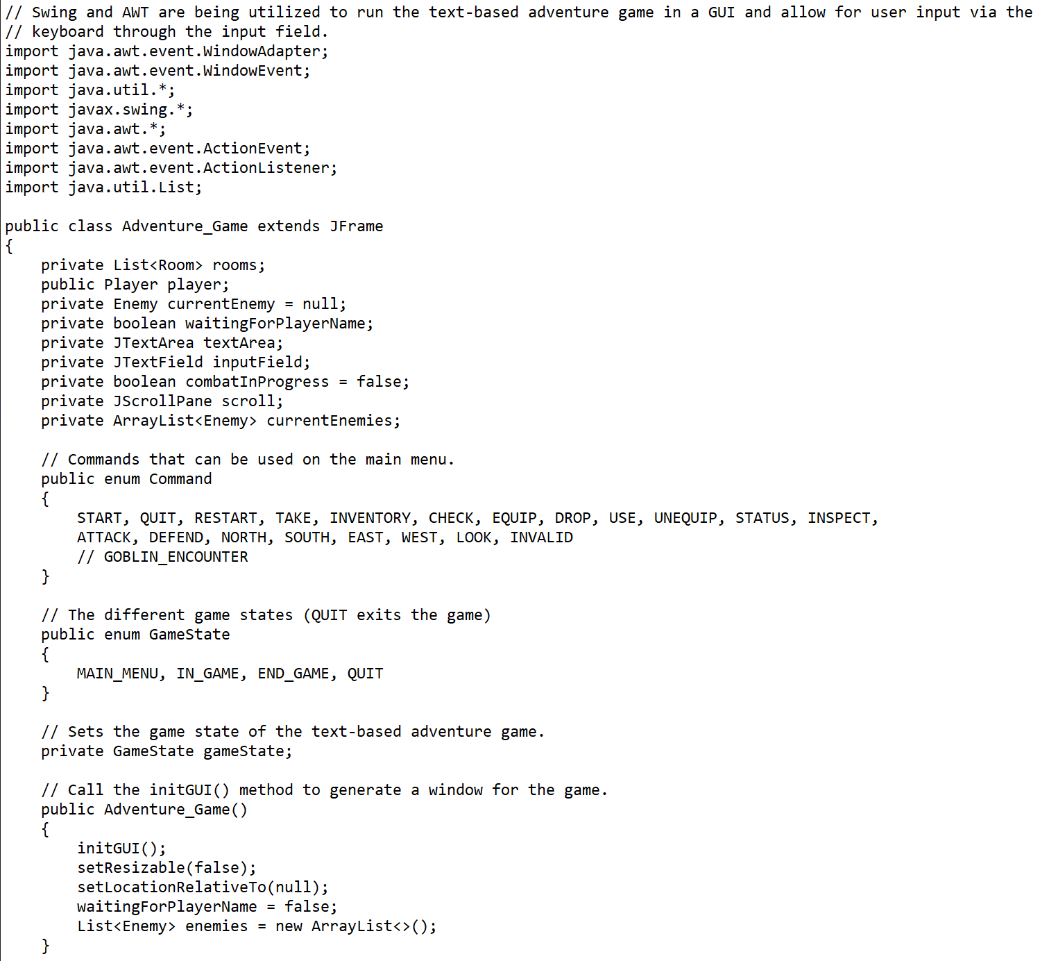


Figure 3 – The Adventure\_Game main class, featuring the Adventure\_Game() constructor method, Commands, and GameStates.

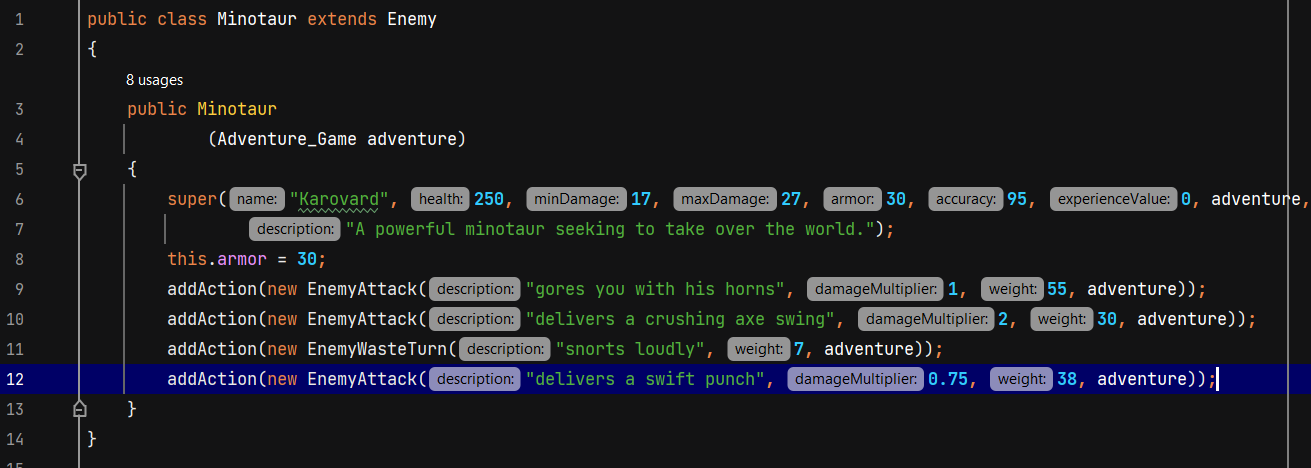


Figure 3 – The Minotaur class, extending the Enemy superclass via inheritance.

**Classes**

The text-based adventure game uses different classes and interfaces to run all of the separate mechanics of the game. In our main class where the bulk of our program is being held is sort of like the engine of the entire game. The other classes act like subsidiaries of the engine, taking care of the smaller tasks and doing them whenever the main needs to be run. This gives the main control power to run its main components and then run the smaller ones without wasting energy and resources in the core.

**Combat**

The combat mechanics of the game are handled by a series of switch statements within the handleInGame() method of the main Adventure\_Game class. These statements define all actions that are available when the boolean variable "combatInProgress" from the startEncounter() method returns true. This variable is only set to true when enemies are present in the room the player enters.

The flow of combat starts with the player typing in what they wish to do, where the only two commands that will progress the flow of battle are "attack (enemy name)" and "defend." A "use (item)" command is in place to allow the player to restore their health by using a health potion, which is then removed from their inventory after use. Once the player attacks an enemy or chooses to defend, all enemies that are still alive will take their turn(s) to choose a random action from the list of actions they have available. They are limited to either attacking or wasting their turn(s). The probability of a specific action being chosen is decided by the designated weight value divided by the total weight value of all actions assigned to the enemy. A base accuracy stat has been implemented for both players and enemies alike to allow for a sense of randomness to combat, where attacks may miss.

Outside of combat, the player has access to most commands, with the following commands being locked:

* Attack (enemy name)
* Defend
* Check (enemy name)

These commands are locked due to only being viable when in combat and thus serve no real purpose outside of this.

Outside of these locked commands, the player is able to look around the room by entering "look around" into the input field. By doing so, the game is prompted to display the room description and a list of all the items contained within the room, which the player may pick up via the "take" command. The player may also drop items from their inventory by entering the “drop” command, which causes the item dropped by the player to be added to the list of items contained within the room, ensuring that items do not disappear upon being dropped. However, this is not the case for health potions, where once used, they are gone.

1. **Design Process**

This project started off with brainstorming amongst us college students on what type of game we should create, as we had roughly three months to work on it. Through brainstorming, we decided on trying out a text-based adventure game, where the main controls of the game would be handled through typed commands by the user. After coming to the decision of creating this type of game, we then considered our options of programming languages, eventually deciding on Java due to it being an object-oriented programming language and offering a vast number of libraries. After conducting a bit of research into what libraries Java offers, we came across Swing and AWT, which offered exactly what we were looking for.

Next, we needed to define how the game would differentiate between game states and how it would handle commands. Game states can be defined as where in the code the game will be running, such as the start menu, main menu, or in-game. For instance, the main menu will be displayed whenever the game state is set to MAIN\_MENU and the actual game starts after the player enters the START command, followed by a name, which changes the game state to IN\_GAME. In other words, these game states are updated depending on certain factors. We also needed a way for the player to perform certain actions, which is where the handleInGame() and parseCommand() methods come in. These two methods define what inputs are registered as valid commands and what actions will be performed depending on the command entered as well as if the player is in combat. With these ideas in mind, we were able to outline the various scenarios the player should be in.

After outlining all the commands that the player should be able to access, we then began working on the classes that would naturally be required for a text-based adventure game. These classes included:

* A Player class with the following parameters:
  + Name
  + Level
  + Health
  + Minimum damage
  + Maximum damage
  + Armor
  + Inventory
* An Enemy class with subclasses that would use the following parameters:
  + Name
  + Health
  + Minimum Damage
  + Maximum Damage
  + Armor
  + Description
* An Item class with subclasses.
* A Room class.
* Classes for enemy actions.

With these classes in mind, the game was procedurally developed, with changes made as the game grew larger and larger. The game needed to be able to handle any inputs from the player through event handlers and switch statements.

A text-based adventure game without a story would prove to be meaningless due to the lack of graphics. A story is the core aspect of any adventure game, as it ties the player into the narrative and gives a sense of progression. It was through this thought process that we decided to use an old Dungeons & Dragons campaign one of our group members had written years ago. Thus, the title “Slay The Minotaur,” a game in which the main boss is a minotaur named Karovard who seeks to take over world, was born.

With the story and game mechanics in mind, all that was left was to create an application capable of displaying information in a graphical user interface, register user inputs as commands, and handle any potentially unexpected user inputs or erroneous data. After this, we focused on the more minute characteristics, such as expanding a basic attack damage to be a damage range for both player and enemy, as well as giving enemies a list of actions to choose from. The room layouts were one of the last features to be implemented, with the appropriate code updated to handle the player’s movement between rooms, available items in rooms, and assigning enemy encounters.

1. **User Manual**

***Overview of the Document***

The document contains a verbal and visual description of features of the game. Screenshots will be used to showcase the execution of in-game commands.  
  
***Controls***

The game will be controlled using the keyboard, where commands will be input in the field located at the bottom of the window. At the very beginning of the game, the player will be prompted to enter “START” to begin the game or “QUIT” to close the game. **Figure 4** shows the start command in action.

Text

Description automatically generated

Figure 4: Starting the game. The Input field is located at the very bottom of the window.

Upon entering “start” into the input field and pressing the Enter key, the player is prompted to enter a name. This name can be anything the player wishes it to be. After choosing a name pressing Enter, the game will begin, displaying a variety of commands available both in and out of combat. **Figure 5** showcases this.

Text

Description automatically generated

Figure 5: The game begins after the player enters a name

The GO command allows the player to move between rooms by indicating the cardinal direction in which they wish to move in: North, South, East, or West. If they enter a room containing an enemy, combat will begin, where the ATTACK command may be used to initiate an attack against the enemy, the DEFEND command may be used to nullify all damage for one turn, or the CHECK command may be used to check the stats of an enemy. **Figure 6** shows these commands in action.

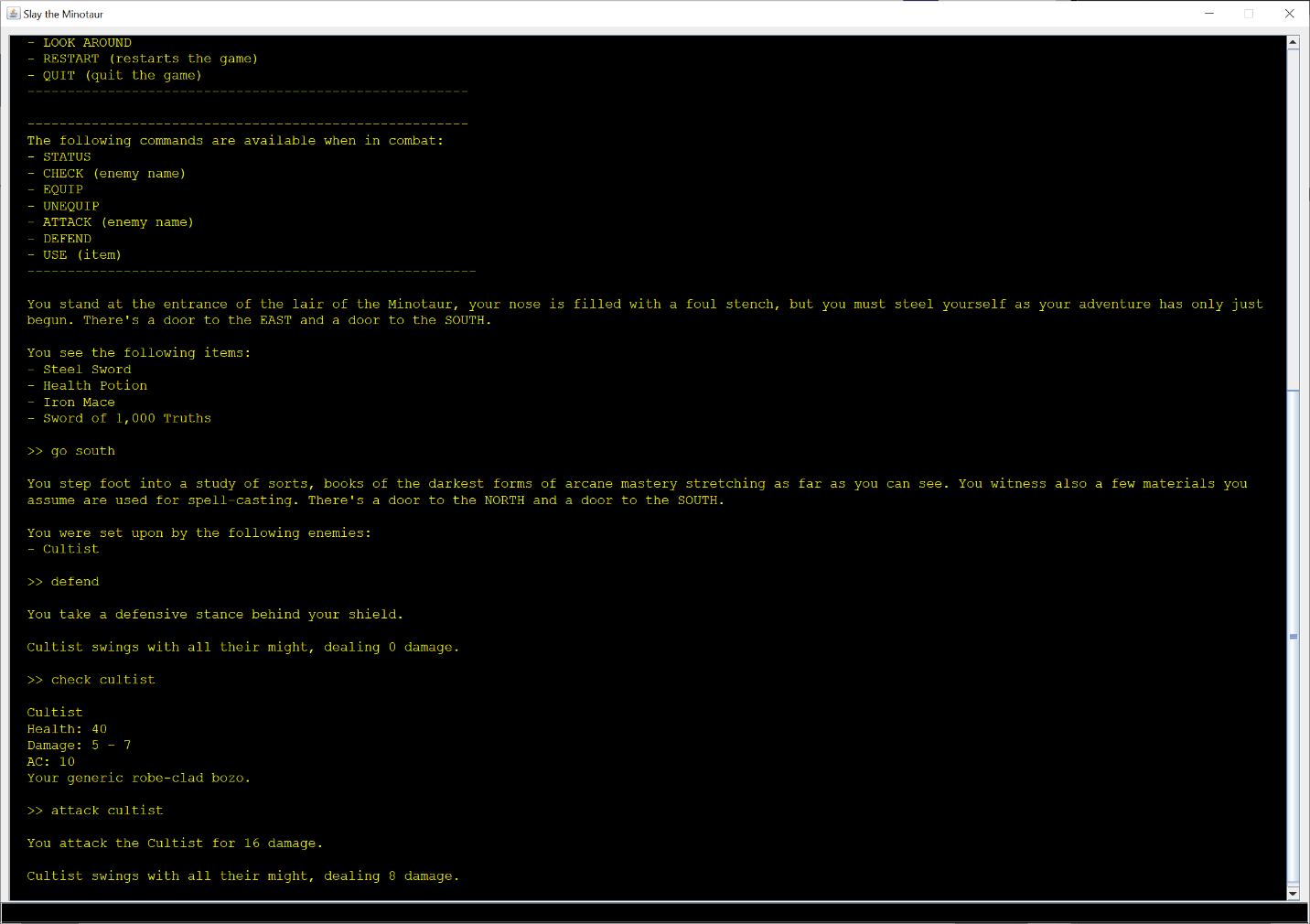


Figure 6: The GO, ATTACK, DEFEND, and CHECK commands being used.

When the player is not in combat, and the room contains items, they may take items by using the TAKE command, followed by the name of the item they wish to take. If the item they wish to take is a weapon or armor, these may be equipped by typing in the EQUIP command, followed by the name of the item. If the item is a health potion, the player may use it by typing in the USE command, followed by the name of the health potion. The player may use the STATUS command is used to view your status, and the INVENTORY command displays the player’s inventory. **Figure 7** shows these commands being used (note that the health potion may only be used if the player is not at max health).  
  
Text

Description automatically generated

Figure 7: The TAKE, EQUIP, STATUS, INVENTORY, and USE commands at work..

When the player wishes to drop an item, they need to enter the DROP command, followed by the name of the item they wish to drop. If the item is a weapon or armor, the player must either equip another item, or use the UNEQUIP command, followed by the name of the item. By using the LOOK AROUND command, the player can get the room description, followed by the list of items contained in the room. If the player wishes to get a description of an item in their inventory, they may use the INSPECT command, followed by the name of the item. **Figure 8** shows these commands at work.

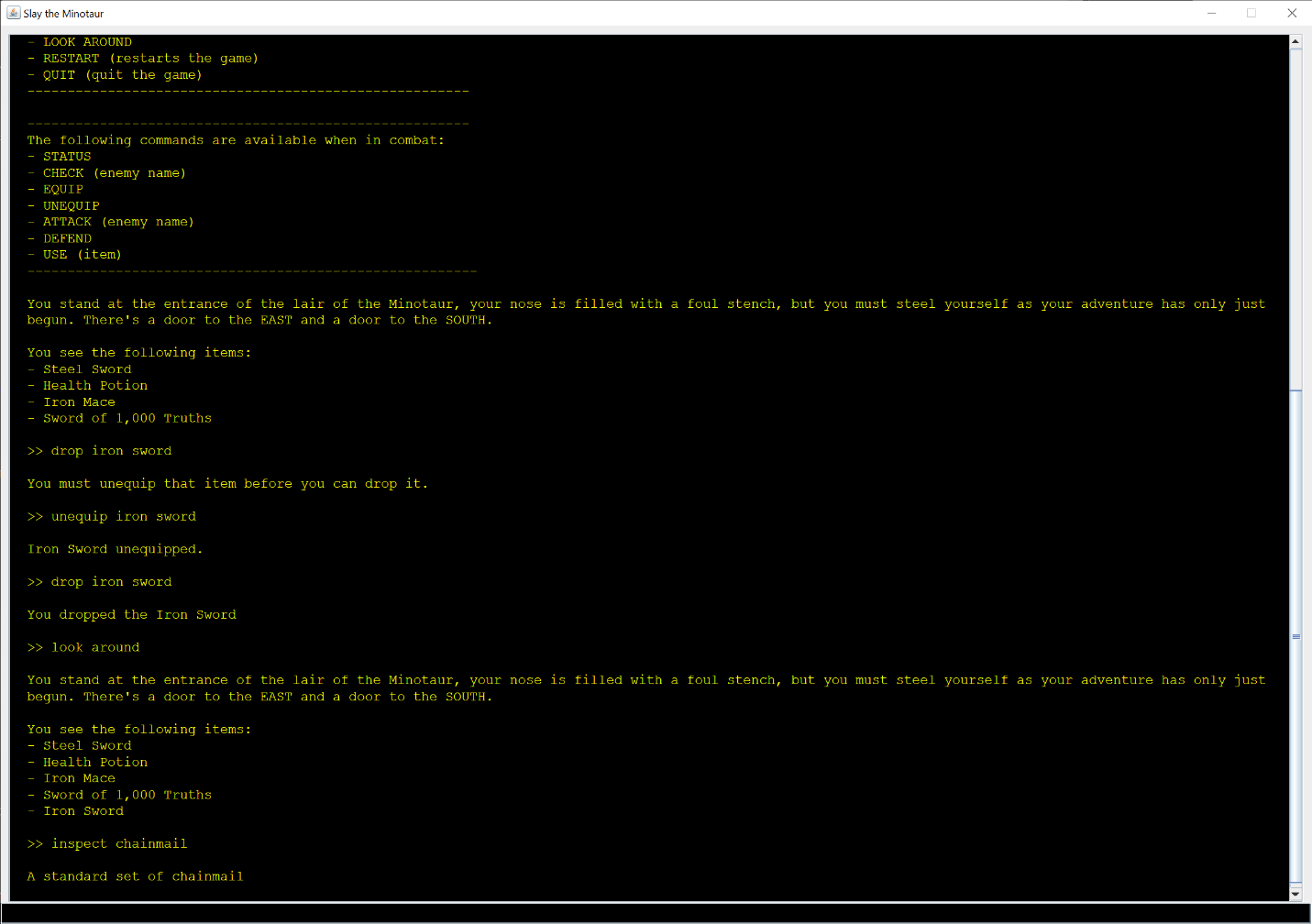


Figure 8: Using the DROP, UNEQUIP, and LOOK AROUND, and INSPECT commands in-game. Notice how in order to drop an item, the player must first unequip it, or equip a different item. Dropped items are added to the room in which they are dropped.