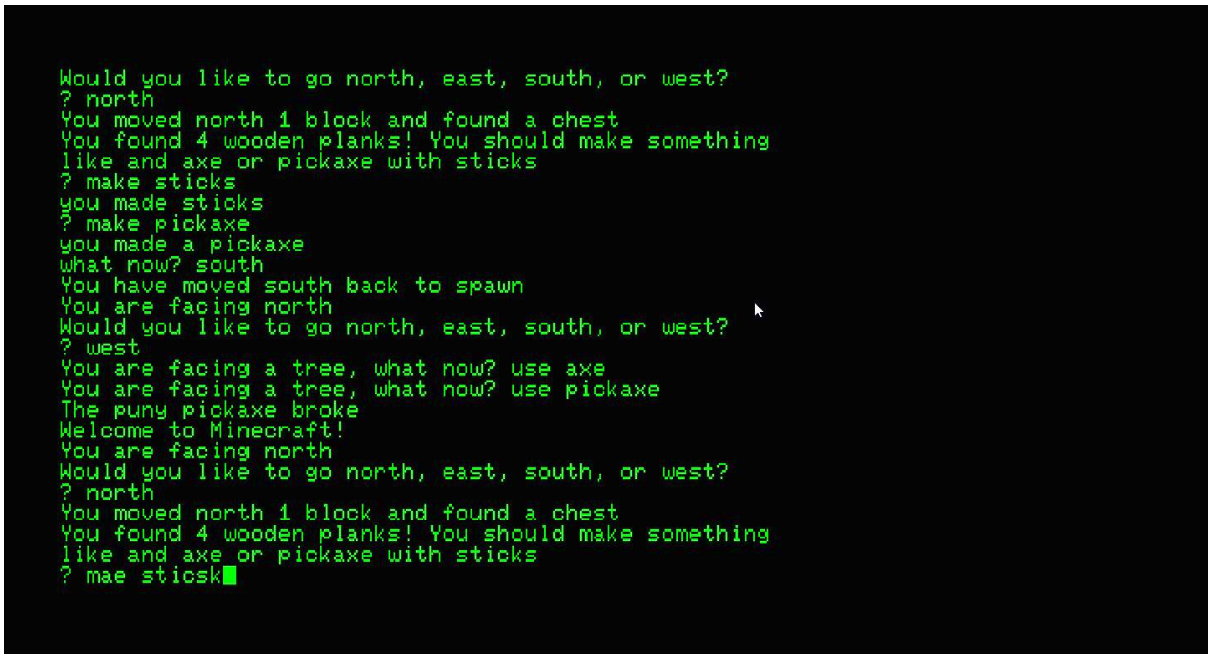
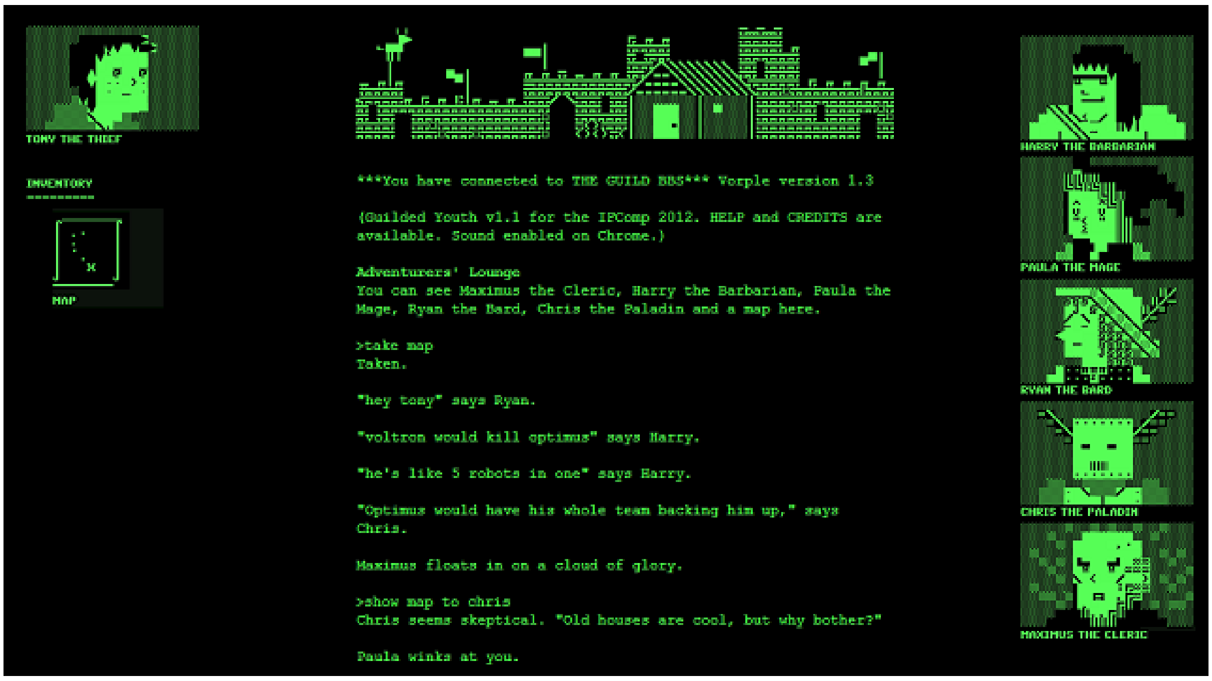
Project Specifications of the Programming Methodology Course 2020/2021 (MZ)
Bachelor's Degree in Computer Science - Sapienza University of Rome

Java Text Adventure - A text

adventure A text adventure is a video game that allows a player to explore a virtual world through keyboard commands:



```
Would you like to go north, east, south, or west?
? north
You moved north 1 block and found a chest
You found 4 wooden planks! You should make something
like and axe or pickaxe with sticks
? make sticks
you made sticks
? make pickaxe
you made a pickaxe
what now? south
You have moved south back to spawn
You are facing north
Would you like to go north, east, south, or west?
? west
You are facing a tree, what now? use axe
You are facing a tree, what now? use pickaxe
The puny pickaxe broke
Welcome to Minecraft!
You are facing north
Would you like to go north, east, south, or west?
? north
You moved north 1 block and found a chest
You found 4 wooden planks! You should make something
like and axe or pickaxe with sticks
? mae sticsk
```



```
***You have connected to THE GUILD BBS*** Vorple version 1.3

(Guilded Youth v1.1 for the IFComp 2012. HELP and CREDITS are
available. Sound enabled on Chrome.)

Adventurers' Lounge
You can see Maximus the Cleric, Harry the Barbarian, Paula the
Mage, Ryan the Bard, Chris the Paladin and a map here.

>take map
Taken.

"hey tony" says Ryan.

"voltron would kill optimus" says Harry.

"he's like 5 robots in one" says Harry.

"Optimus would have his whole team backing him up," says
Chris.

Maximus floats in on a cloud of glory.

>show map to chris
Chris seems skeptical. "Old houses are cool, but why bother?"

Paula winks at you.
```

The minimum elements of the game to code are: the world, an environment of the world (which we will call a room, even if it is an open environment), the characters and the player.

The room

The room is the basic element of the world. Each room has a name, a textual description (and / or graphic, see project for two students) and keeps information on:

- the objects contained in the room (an object can be in only one room at any given time);
- the characters and / or player present in the room (each player or character can only be in one room at any given time);
- the access points to other rooms (referred `links` below).

The world

The world is a set of rooms and has a name and a textual description (and / or graphic, see project for two students).

Character

Each character has a name. A character who has an inventory of items. The character has a number of methods that allow him to interact with the room he is in and with the other characters in the room.

Player

The player, like the other characters, has an inventory of items, which is initially empty. There is only one player in the game, so it is always possible to get a reference to the player.

Loading the world

The world is loaded with the `Mondo.fromFile (Path)` which returns a world loaded and instantiated from the specified file. The format of the input file is as follows:

```
[world: name of the world]
description <tab> textual description of the world
start <tab> name of room 1

[room: name of room 1]
description <tab> textual description
objects <tab> object1 , object2, ..., objectson
characters <tab> character1, ..., charactersom
links <tab> N: door 1, S: room 3, O: room 2

[room: name room 2]
< tab> ......
description
links <tab> E: room 1
```

```
[room: name of room 3]
…
etc.

[links]
door1 <tab> Door <tab> room name 1 <tab> room name 2
window1 tab> Window <tab> room name 4 <tab> <
teleportation garden1 tab> Teleportation <tab> magic room <tab> planet X

[objects]
sushi in the bowl < Keyfish
Key tab> door1 <tab> <
key 2 <tab> Key <tab> port2
etc.

[characters]
Harry <tab> Character
Mary <tab> Character
bobby <tab> Dog
etc.

[player]
Zak <tab> Player
```

where " **`<tab>`** " represents the tab character (including spaces, inserted here only to better separate fields). In particular, there are six types of sections:

- the section relating to the world which contains the name immediately after the keyword `world:`, the description in the format `description` **`<tab>`** description of the world and the room from which to start the world in the format `start` **`< tab>`** `room name 1`
- room (`room: unique name of the room` in square brackets) which contains the following fields:
  - `description` **`<tab>`** description of the room
  - `objects` **`<tab>`** list of names of objects separated by comma
  - `characters` **`<tab>`** list of names of comma separated characters
  - `links` **`<tab>`** comma separated list of character pairs `: name of another room or of a link type object` (e.g. a door) where the character can be `N, S, O, W, E` (`O` and `W` have the same meaning), indicating the direction in which you have to go to enter that room.
- list of links (doors, windows, etc.) that allow a connection between two rooms in the `[links]`. The lines in this section contain four fields: link name, link type class (`Door`, `Window`, etc.), names of the two rooms connected by the link.
- list of objects with their association to the Java class of which they are instance in the [`objects`] section in the format: `object name` **`<tab>`** `Java class` (in some cases it is possible that a third parameter is provided that the object with which it can interact defined object).

- list of characters [`characters`] in the format: `character name` **`<tab>`** `Java class`.
- section `Player [player]` : contains the player's name and class. You can only take one player for granted (and fail otherwise).

The order of these sections is not predetermined (for example, `characters` and `objects` do not necessarily appear at the bottom of the file, as shown above).

The method for loading must check (and issue the consequent error) if: 1) the same object or character is inserted in more rooms, 2) if the objects specified in each room are always associated with a corresponding Java class (otherwise they will not be able to be instantiated); 3) any other errors that can make it impossible to create the world (eg two rooms or objects with the same name).

The file extension for the graphic part is left to the group of students who will implement it (see below).

Textual interaction engine

This is the heart of the game. It is an engine that transforms a text command into a call to a corresponding player method. For example, if I write:

```
go west
```

this will result in the method call `Personage.west`(). Note that a text command will not necessarily directly result in a call to a method with the same name. For example, you could also go west with commands like:

```
Either
go OR
go west
```

(the important thing is that the commands in the provided scripts work). The latter command, for example, could translate to `Character.goi (Direction.WEST)`. When a command is not recognized, the engine raises an exception. As just shown, commands can also provide parameters. For example:

```
take sushi in the bowl
```

will have to match the parameter after `take` with all the names of objects, characters, rooms etc. and, for example, invoke the `character.get (object)`. Similarly:

```
open door with key1
```

The elegance of the engine implementation is subject to evaluation (a cascade of if is not ideal). It is possible but not necessary to introduce a concept of stopword which will be discarded in case the command recognition fails. For example:

```
open the door with key1
```

The execution of each command through the text interaction engine must return a textual confirmation of the success or failure of the requested operation.

Starting the game

The game is started with the following instructions:

```
Game g = new Game ();
Mondo m = Mondo.fromFile (fileName);
g.play (m);
```

classes `Game` and `World`; the objects to be implemented can instead use an ad hoc subpackage):

```
package it.uniroma1.textadv;

public class Test
{
      public static void main (String [] args) throws Exception
      {
            Game g = new Game ();
            Mondo m = Mondo.fromFile ("minizak.game");
            g.play (m);
      }
}
```

Fast forward

game The game allows you to make a game in fast forward mode by entering a script that contains the sequence of text commands, by calling the `play` with two parameters, `World` and `Path`. For example:

```
Path scriptDiGioco = Paths.get ("minizak.ff");
g.play (m, scriptDiGioco);
```

`Minizak game`

The project must work correctly on minizak .game and .ff files:
(v1.0.5, also includes game map)

Command localization (3 extra points)

It is possible to get 3 extra points by implementing the localization of the commands (take , go to, etc.). In this case, the world has a `Game.localize (language)` (where language belongs to a `Language` values `EN` and `IT`) that can switch to the required language (resulting in a change in the handling of commands by the text engine , which will have to use some form of mapping from the default to the required language, if different from the default).

Creating your own story (4 extra points)

In addition to making minizak work, you can implement your own story (.game) and the corresponding fastforward (.ff) file, with ad hoc classes for specific game objects (these classes however, they should be included in the same package used for minizak objects).

Hidden games, bonus rooms, Easter eggs (4 extra points)

It is possible to develop an Easter egg, a hidden game or a bonus room by modifying Zak's world (or inserting it into your own story).

Evaluation

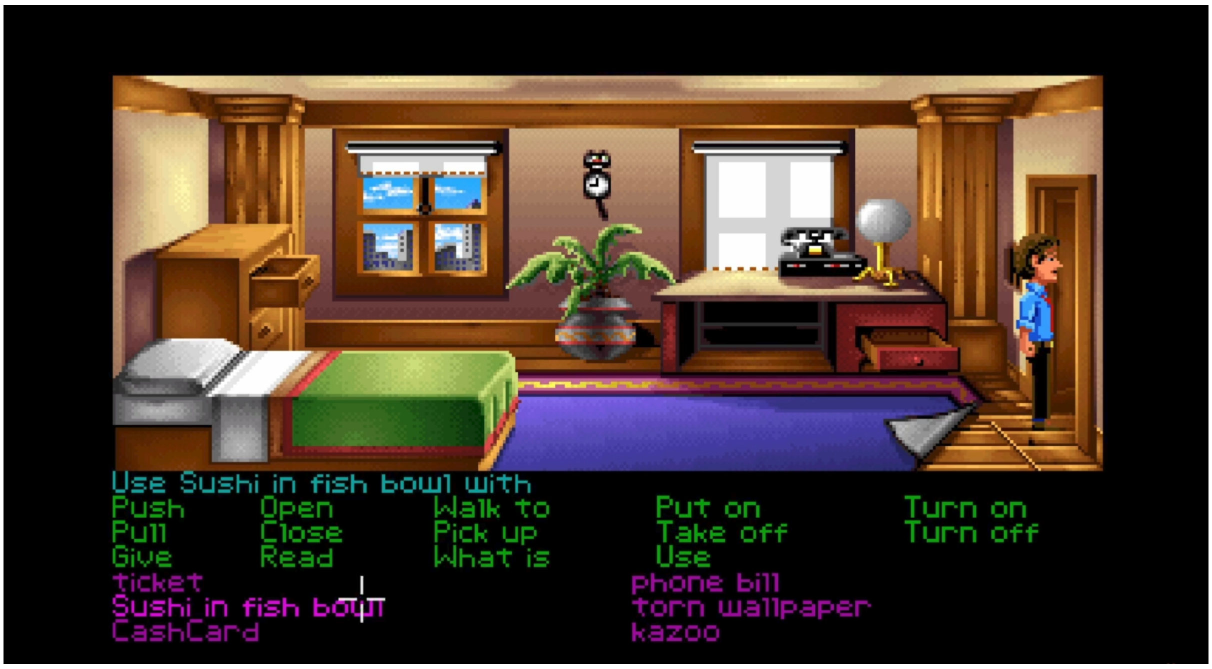
To achieve sufficiency the project must run correctly. The following elements will be evaluated:

- architecture**,**object **modeling** and the**organization and cleanliness of the code** in terms of **correctness and extensibility**, especially in relation to the appropriate use of **naming** (e.g. **camelCase**), **encapsulation** and **visibility of fields and methods**, **inheritance** and **polymorphism**, the correct use of **hashCode** and **equals**.
- code documentation using **javadoc** (on all class headers and methods required by this specification, including parameter documentation), exported from Eclipse / IntelliJ.
- the use of Java 8 (**lambda expressions** and / or **reference to methods** and **streams**).
- using **design patterns**.

Scores (from 0 to 34)

- To obtain a minimum score (18) the project must be able to be carried out, **without absolute hard-wired paths** (for example, file paths or other information entered within the classes) and must be able to carry out the visit correctly of the world (based on an indicative base script that explores the world, without doors, windows, etc.). It is possible to use this simplified version to verify the achievement of sufficiency (form the validity of what has been developed also at the code level):
- It is possible to obtain a grade> 30 only if an appropriate use of reflection and design patterns is made.
- **Realization in groups of two students:** realization in a team is possible if one of the two students creates the graphical interface for the underlying textual engine. You leave ample freedom regarding 1) the technology of the graphics library (as long as it is a Java library, such as JavaFX, LibGDX or others), 2) the graphical interaction mode. The only important constraints are that 1) the graphical interface uses the textual engine created by the other student and the corresponding feedback, 2) the

graphical interface still allows you to interact with the world **ALSO** by typing textual commands.


Use Sushi in fish bowl with
Push
Pull
Give
Open
Close
Read
Walk to
Pick up
What is
Put on
Take off
Use
Turn on
Turn off
ticket
Sushi in fish bowl
CashCard
phone bill
torn wallpaper
kazoo