Mislead

**Objective**

Mislead is a java-based code set for the genre of game known as text adventures or interactive fiction. In a text adventure, the player interacts with the game world by typing simple commands which are interpreted by a parser into game actions. The game then outputs text describing what the player can see and hear. Through exploring and examining the world, the player learns sufficient information to understand the game’s model and solve its puzzles.

At its most basic, traditional interactive fiction allows the player to move between multiple rooms and pick up objects. It also allows them to use those objects to solve puzzles by using them on the environment. The interface of a text parser usually interprets commands in the simple imperative form, such as CLOSE DOOR. In the oldest of games, most actions that did not lead to solving a puzzle were not understood, returning an error message: “You can’t pick that up”, “There’s no need to eat that”, that sort of thing. Most basic puzzles will require implementing commands that contain both a primary and a secondary object. LOCK DOOR WITH KEY, for example.

The player must be capable of listing the objects they are carrying using a command called “inventory” to keep track of progress. This command allows the player to quickly take stock of what they’ve picked up, and thus their options for solving a problem in front of them.

More sophisticated game concepts require the modeling of time, usually by turn count: counting the actions the player has taken. For example, a lit fuse might take a while to burn, or a large monster might occasionally move from room to room.

**Design**

Mislead is separated into two parts: One section, called the “Parser”, is for communicating with the player through the interface, accepting input, and translating it into actions. The other section, the “Model”, models the state and behaviour of the game world, tracking objects, rooms, and the position of the player.

The Parser

The Game Class

The game object, once instantiated, creates a game frame that contains UI elements for the player: A text box that contains the output of all game text, and an input box below it that allows the player to input commands and hit “enter”, which sends the output to a parser object.

It uses Swing. [expand if necessary]

The game class also contains three static fields – one for a player object, one for a parser object, and one for a time object. Information about these three objects is needed throughout the program, so putting references to them in static fields makes them easy to find and access when we need to.

The interface frame also contains a compass that displays available exits from the room the player is standing in, which makes it easier for the player to orient themselves. It changes the state of the compass by polling the state of each of the six exits from the room the player is standing in. (See the “Room” and “Exit” class.)

The Parser Class

Commands from the player are routed to the Parser class, which interprets them using its parse(String in) function. It assumes the command is space character delimited and separates the string into single-word tokens using a StringTokenizer, and then shuffles those tokens into a stack. As the StringTokenizer pushes the words into the stack in order, we get them out of the stack backwards.

This means that a command such as OPEN DOOR WITH KEY gets read as KEY WITH OPEN DOOR. The parser class looks at each word in sequence, figures out what sort of word it is, and performs a behaviour based on what it sees.

The parser contains lists of words: verbs and prepositions are hard-coded, whereas the list of nouns is automatically generated, with each thing created in the game model adding its name to the list using the constructor.

On seeing a noun, the Parser assumes it indicates the object of the action. It then tries to find the thing, looking in the player’s inventory for items the player is carrying, and in the room’s contents for things the player can see. If it can’t find the object, the action fails, telling the player “I can’t see that here.”

Upon seeing a preposition, such as “on”, the Parser then realizes the object it saw earlier is actually the secondary object of the action, and thus moves the object into the secondary object.

Upon seeing a verb, the Parser takes the object and the secondary object (if it has one) and asks the Understand class to turn it into an action.

The Understand Class

The Understand class is a large switch block that takes the objects given it to by the Parser and a string containing a verb and translates it to an action. Its three understand methods correspond to zero, one, or two nouns. In the one or two noun case, it calls the relevant action method of the primary object of the command. In the no-noun case, it calls a special action: Either travelling in a direction (such as NORTH, SOUTH, and so on: Those are verbs in an IF game!) or calling a special action, such as “i” for taking inventory.

The Io Class

The Io class (for “In / Out”) is a utility class whose static method, out(String str) forwards a string to the game object to be printed. It exists to make it easy to change the output from going to the console to going to a game object without having to change the code in more than one place, and because “Io.out(...)” is easy to remember and fast to type.

The Model

The Thing Class

Every object in the game world that the player can interact with inherits functionality from the Thing class. Its task is to contain the default failure behaviour for all actions and functionality, which are then overwritten by individual subclasses.

Each thing has a method for each verb in the game. When the player tries eating a lit torch, the action that gets called is torch.eat(). Even though eating a torch is ridiculous, we’ve promised the Understand class that calling thing.eat() makes sense no matter what thing you give it, so all things need to have some response to such a call. (For the most part, it’s a failure message such as “You can’t eat that.”)

Individual things inherit from the thing class and overwrite their methods with more specific ones that change the game state; for example, to put a hamburger in the game, one would write a Hamburger class that extends from Thing, and have it replace the default eat() method with one that contains the description and effects of eating a hamburger.

All things have a name, which the player and game refer to it by, and a description, which is printed when the player examines the thing closely.

Not every single thing mentioned in the game is modeled as a thing: For example, we don’t instantiate the soil beneath your feet or the sound of rain, even though we might mention them in the description – unless either of those is used in a puzzle.

The Carryable Class

Carryables are a special sort of thing that can be picked up and carried with the player, thus the name: Items the player can get. The get() and drop() functions of a carryable move the item from the room into the player’s inventory, or vice-versa, if appropriate.

The Room Class

Every location in the game that the player can visit is an instance of the Room class. All rooms have a name and a description. The description being how the room is described to them when they enter, or look. On visits after the first, the full description is replaced with a shorter one, since it’s likely that the player already has some idea what’s in there, and is only interested in the most important parts of it.

Every room contains a Thinglist called its contents: All the things the player can interact with in the room.

Each room has six fields for exits, which describe its connections to other rooms. Exit fields are usually null, but if the player can travel in the north direction from a room, there will be an exit to the north.

The Thinglist Class

[Describe it as an arraylist, describe the Find and Has functions, note the Add and Remove functions]

The Exit Class

[It has two sides, it can be open or closed. Note that special exits can extend from this one, such as an exit where leaving through it ends the game. Traveling through it moves the player from the side they’re on to the side they aren’t on.]

The Player Class

[The player exists in a certain location. The player has a special Thinglist called an inventory of things they carry. The player’s north, south, etc functions check to see if there’s a specific exit in that direction. Calling player.getLocation() is how we find the current location that the game is paying attention to, for anything where that matters, such as searching through items the player can see.]

The Time Class

[All successful actions increment a turn counter in the Time class. Things can have tick() methods that describe “what happens with this thing after a certain period of time” – these tick() methods can be scheduled in the Time class, and they occur some time later. Describe the LinkedList that backs the Time class, and the logic we use to make sure it remains sorted.]