## CSE 413 16au Assignment 6 – A Ruby Text Adventure

**Due: Online via the Catalyst Dropbox by 11 pm, Thursday, November 17**

Text adventure games have been a staple of personal computing for decades, existing on practically every computer system from the microcomputers of the 70s through the handheld devices of today. They are fairly expressive while being extremely simple to create and use. They also lend themselves well to an object-oriented design, which we will explore in this assignment.

A text adventure game, like any game, takes in input, uses that input to modify the game state, and then displays output based on the new state. Text adventure games use text for both input and output: players input short phrases that the game can parse as actions, and the game displays narrative text or ASCII art showing the result of the action or giving more information about the scene.

A typical "turn" in a text adventure game might look something like this:

```
+- Outside Cave Entrance -------------------------------------------------+
| You are at the entrance to a large cave mouth in the middle of an evergreen
| forest. A large moss-covered rock sits at the mouth of the cave.
+-------------------------------------------------------------------------+
Player > examine rock

       _---_
     _/     \
    |  ----__-|
    \__--____/

+- rock -----------------------------------------------------------------+
| The rock is over a meter across. Water drips off the fronds of the moss
| that covers its surface.
+-------------------------------------------------------------------------+
```

Here, the game presents a Location in the game to the Player, then prompts for input. The player types in an Action and an Thing to perform that action on, and the game runs the appropriate code for that Action: in this case, printing out an image and some descriptive text for a Thing in the scene.

The capitalized words in the previous paragraph should give you some idea of how the game represents things under the hood – each capitalized word corresponds to a class in the game code. In this assignment, we will provide the base classes for you. Your job will be to create sub-classes that add new behavior to the game. This assignment emphasizes reading, understanding, and extending existing code.

The provided base classes in *game.rb* are as follows:

**Game** – Stores the important pieces of game state, such as the player, current location, and set of possible actions. Responsible for prompting for input, parsing, and formatting output nicely. Subclass this if you want to change how parsing (i.e., analyzing input commands) or output display works, or to add additional objects to the centralized game state.

**Action** – A class that provides a *do*(args) method. Subclass this if you want to add a new action to the game. (Notice how Ruby encouraged an inheritance-based design for this problem, where another language like Racket might have gone with a more functional approach, passing "action" functions around directly).

**Location** – A place in the game. Has a name and description text, a dictionary of "doors" mapping names to other locations (e.g. "east" to another Location to the east), and a set of "things" in the room, which may be an instance of any subclass of Thing. Subclass this to add unique behavior to some locations.

**Thing** – An object in the game. We don't use "Object" because that already exists in Ruby. Specifically, a Thing can have up to three different text strings describing it, and has a *describe*(context) method that takes a *symbol* dictating which kind of description text to return. (A Ruby symbol is a name like :abc, which is quite similar to a quoted Racket name 'abc, and is often used for purposes similar to enumerated types in other languages. See any Ruby reference for more details.). The

default context symbols used in the game are **:brief**, used when the **Thing** needs to be described in a couple of words (e.g. for an inventory listing); **:world**, used when a **Location** is trying to print out descriptions of all of the Things it contains; and **:detail**, used when the player examines something closely.

There are two supplied subclasses of **Thing** in the provided code:

**Monster** – A **Thing** with an inventory and a dictionary of attributes. Can print an ASCII portrait, e.g. when being described in detail, and can be described in a **:combat** context. Has an *attack*(other) method that, by default, tries to decrement the attributes of *other*. Subclass this to create monsters with unique portraits, attributes, attacks, etc. One subclass of Monster is included in the provided code:

**Player** – A special kind of **Monster** that prints second-person instead of third-person messages. Usually, there's only one instance of this, and it's held by the **Game**.

The other supplied subclass of **Thing** is **Item** – A **Thing** that has a *use*(args) method. Subclass this to add items that do unique things when used.

The goal of this assignment is to create a functioning game with interesting content. The first part of the assignment will make sure you understand how Ruby's object system works by having you instantiate classes and create subclasses with specific additions, while the second part will be a more open-ended creative task. You may want to view the requirements of the second part before starting so that you can work towards them while working through Part A.

**Part A**

Before you begin, we recommend that you skim through the provided **Game.rb** file so you get a general sense of how things are implemented and what kind of interface you are provided. You may also want to open up the Ruby docs in a browser window so you can refer to them regularly while working. For this part of the assignment, you should save your code in a file called **hw6a.rb** and make it clear which code goes with which part so that we can give you appropriate credit when grading.

**i)** (Warm-up). At the top of your file, import the game code (hint: look up **require_relative** in the Ruby docs). Create an instance of **Location** with a name and descriptive text. You may want to store this in a variable to use momentarily. Then create an instance of **Game**, passing in the location you just created as the positional argument to Game's constructor.

At this point, you should be able to run the game. Open up a terminal window, navigate to the directory where the **game.rb** and your **hw6a.rb** files are located, and type "*ruby hw6a.rb*" to run the game. You should see a prompt that looks like this:

Enter your name:

??? >

If you don't see a prompt and instead see an error message, go back and fix the errors until it works as described.

After entering your name, the game should display the text you put in the Location you created. At this point, you should try out some basic commands to see the game in action.

**ii)** Now you're going to create your first subclass. In your file, create a new class called **FriendlyMonster** which extends **Monster**. A **FriendlyMonster** is a monster that doesn't want to attack the player. In the **FriendlyMonster** class, override the *attack* method with a new method that does nothing. Make sure that the parameter list matches the superclass' *attack* method, or you'll encounter error messages when the game is running.

Create an instance of FriendlyMonster with at least a name and description. The way you add a Monster (or any Thing) to a Location is simply by creating mappings in the Location's *things* hash. The keys you add to the hash will be used by the game when executing actions, so you may want to add multiple mappings so the player can refer to the same thing by several different names. (Not sure how to add something to a Hash? Check the docs!)

(Also, for your (ungraded) consideration: how is it that *things* is accessible to other classes? Find the line in **game.rb** that makes this possible.)

At this point, when you run the game, you should see a new line of description text appear under the description of the location. Try out some of the built-in commands, using the name you gave the FriendlyMonster as a target. Do any of the commands affect it?

**iii)** Now you're going to add a new kind of **Item.** Create a new class called **Lantern** which extends **Item**. A Lantern is an item that can be turned on and off, and will toggle states when **Use**d. Override the necessary functions from the superclass to make this happen. Lantern items, when **Examine**d, should tell the player whether they are on or off. Figure out what needs to change to make this happen, and do it! (Hint: go look for the **Examine** action and see what it does!)

The provided **game.rb** doesn't have an action for picking up and dropping items in the world, so you'll have to have the player start out with this item in their inventory. Take another look at the **Game** constructor and figure out how to pass in some items for the player to start with. Once you've figured out how, add an instance of the **Lantern** item to their starting inventory with appropriate description strings.

**iv)** Now that you have an item that can do something interesting, let's add a **Location** that is affected by the lighting. Create a subclass of **Location** called **DarkLocation**. A DarkLocation is a location that can't be described unless the Player has a Lantern item that is on – it should print something generic like "It is too dark to see anything." instead of its built in description UNLESS the player has a Lantern turned on, in which case it prints its normal description string.

There are several different ways you can achieve this effect – the details of implementing this are up to you as long as it obeys the above behavior, and doesn't break how the game deals with normal Locations.

Adding a **DarkLocation** to your game will be just like adding a normal **Location**. Instantiate the DarkLocation and save it to a variable, then add entries to another **Location**'s *doors* Hash. (This should work just like adding Things to a location). Once the new location is added to a previous location's *doors* Hash, you should be able to run the game and use the **go** command with any key in the Hash. For example:

```
+- Location 1 ----------------------------------------------------+
| Location 1 description
+---------------------------------------------------------------+

Player > go west

+- Location 2 ----------------------------------------------------+
| Location 2 description
+---------------------------------------------------------------+
```

In the case of a DarkLocation, going to a DarkLocation should result in "Location 2 description" being something like "It is too dark to see anything." unless the Player has a lit lantern.

At this point you should have a pretty good idea how to create and extend objects involved in running the text adventure, so now it's time to implement your own. Save your **hw6a.rb** file with the above changes to submit with the rest of the assignment.

**Part B**

The second part of this assignment is... to build your own game using the framework! This is fairly open ended and you can keep it simple or go very complex. However, there are certain things that you are required to do.

Your code for this part of the assignment should go in a file called **hw6b.rb**. (You may use your code from Part A by importing it with require_relative if you wish)

You must provide a file called **hw6_game.txt** that describes your additions to the game and the way they fulfill the requirements listed below.

For this part of the project you must do at least the following:

i. You must create at least 1 new subclass of **Location** distinct from DarkLocation (it can be a subclass of DarkLocation but you must add some significant new behavior)
ii. You must create at least 2 new subclasses of **Action** to add new actions to the game. You should look at the arguments to **Game.initialize** and the **DefaultActions** array in **game.rb** to see how to hook these actions up to the game's parsing system. The new actions must be possible to run in-game the same way as other actions.
iii. You must create at least 3 new subclasses of **Thing**, **Item**, or **Monster** that add some significant new behavior. (e.g. you can add 1 new Thing, 1 new Item, and 1 new Monster, or 2 new Monsters and 1 new Item, or 3 new Thing subclasses, and so on)
iv. You must *instantiate* at least 8 Locations (possibly instances of your subclass(es) of Location) and network them so that a Player can visit each one
**v.** You must *instantiate* at least 4 Things, Items, or Monsters (possible instances of your subclass(es)) and add them to various Locations in your game

Be sure that you are clear in your **hw6_game.txt** what parts of your file address each numbered item above so that we can give you full credit.

**Extra Credit**

If you finish the above before the assignment is due, and are looking for an additional challenge, here are some things you can add to your game to make it more interesting. You should document which of these things you're adding in **hw6_extra.txt** so that we can give you credit for them. (Note that some of the additions may qualify for required features above. If you are treating them as such, you should say so in your **hw6_game.txt**). A small amount of extra credit will be awarded for extensions that go beyond the basic requirements.

i. For our provided Actions and Items, functionality is added via inheritance, overriding the **do** and **use** methods respectively. Create new subclasses of these that take a block instead, and execute the code in the block when their **do** or **use** is called.
ii. The default parser can only understand 1-word commands, interpreting everything else as arguments. Subclass **Game** and override the **main_loop** method to add support for parsing slightly more complex expressions (that include prepositions like “to” or “up” between the verb and the arguments). (You shouldn't need to know anything in particular about writing parsers, this can remain very simple – just think about what needs to happen if the player types in something like “go to river”)
iii. The base game has no actions for picking up and dropping objects. Create Actions for these (whether they are subclasses or block-based versions from extra credit part i is up to you). The pick-up action should look for a named Item in the current Location, remove it from the Location, and add it to the Player's inventory. The drop action should look for a named item in the player's inventory, remove it, and put it in the current Location. If you decided not to implement extra credit part ii, you may need to look for alternate ways of saying “pick up”.

**What to Turn In**

You should turn in the files that make up your project using the regular course dropbox:

- hw6a.rb – the code from part A
- hw6b.rb – the code from part B
- hw6_game.txt – your description of the additions you have made to the game and how they meet the requirements for the assignment.
- If you do any extra credit, a file hw6_extra.txt that documents the additions to the assignment.