COM1001 Introduction to Software Engineering

Pair Programming

Ruby concepts introduced in this lab: *Methods · Parameters · If / conditional statements*

1 Pair Programming

For this lab, we will be using pair programming. Pair programming is an agile software development technique in which two developers work together at one computer. Find a partner to work with for this lab (or let us know if you can't find anyone).

- Try to find a partner with comparable skills to you to avoid a teacher/learner situation.
- Partners sit side-by-side at one computer.
- The "driver" has control of the keyboard/mouse and actively implements the program.
- The "navigator" continuously observes the work of the driver to identify tactical defects (such as syntactic and spelling errors, etc.) and also thinks strategically about the direction of the work.
- At least the driver, and possibly both programmers, are expected to keep up a running commentary; pair programming is also "programming out loud" - if the driver is silent, the navigator should intervene. Say what you are about to do, ask for an implementation idea, ask for a better way to solve the problem at hand, bring up alternative ideas, point out possible inputs that the code doesn't cover, suggest clearer names for variables and subroutines. When people are pairing well, they are talking back and forth almost non-stop.
- When you're the navigator, don't dictate the code. The driver should be actively thinking about how to achieve the current task, not just typing passively. And as the navigator, you should exploit the fact that you don't need to invent the small details; you can and should think at a higher level. Saying "That looks right. How about handling the case where we're passed a null pointer now?" is better than: 'OK, now type if (s == NULL) return ...'
- You should change roles every 10 to 15 minutes, or at natural breaking points. Today's exercise will consist of implementing five different methods, so a good time to switch over will be after each of these methods has been implemented.
- When changing roles, you should just need to move over the keyboard. If you need to re-arrange chairs or re-adjust the monitor, then you are not sitting in a proper pair programming setup!

2 New Ruby Concepts

2.1 Methods

Ruby methods are very similar to methods in Java or any other programming language. Ruby methods are used to bundle one or more repeatable statements into a single unit. In Ruby, method names should begin with a lowercase letter. You can define a simple method as follows:

```
def method_name
   # ...
end
```

Whenever you want to call the simple method, you write only the method name as follows:

```
method_name
```

Similar to Java, you can represent a method that accepts parameters like this (note the missing type declaration):

```
def method_name (var1, var2)
   # ...
end
```

When you call a method with parameters, you write the method name along with the parameters, such as:

```
method_name 25, 30
```

Methods in Ruby automatically return the last value returned by a statement in that function. For example, the following method `foo` returns the string "`Hello World`":

```
def foo
  "Hello World"
end
```

However, it is still valid to use a `return` statement like in Java, and if you want to leave a method before the end then that is what you will have to do.

2.2 Conditional Statements

You should already be familiar with `if` conditions from Java; in Ruby they work similarly. One difference is, that you do not need to (but you may if you want to) put the condition into parentheses. Another difference is that if you have several different cases you use `elsif` in Ruby, rather than `else if` like in Java. The `then` is optional in Ruby. Finally, in Ruby `if` conditions need to be ended with an `end`.

```
if x > 0 then
  puts "x is greater than 0"
elsif x < 0 then
  puts "x is less than 0"
else
  puts "x is 0"
end
```

To create complex conditional statements consisting of several conjoined conditions, you can use the `&&` and `||` operators like in Java. To negate values you can use `not`. In addition, instead of writing

```
if not var
```

you can write

```
unless var
```

3 A Text Adventure in Ruby

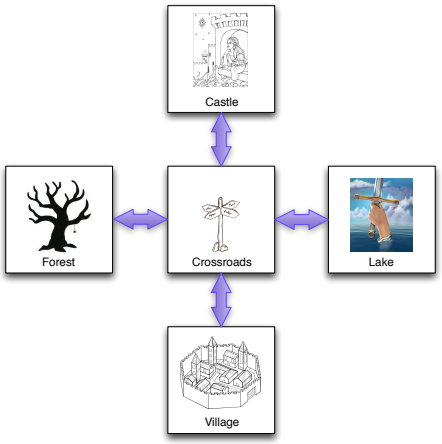
3.1 Text Adventure Games

Text adventures are a type of game popular since the late 70s, and there is still a very active community producing these games today. In a text adventure the player gets shown a textual description of the current location and state in the game, and then uses text input to take actions, solve puzzles, and survive in the game.

If you have not seen text adventures before, you can try classics like "Zork" or "The Hitchhikers Guide to the Galaxy" online at the following website:

`http://textadventures.co.uk`

Your task in today's lab is to implement a thrilling text adventure in Ruby. The adventure starts deep in a dark forest, and you have to rescue a princess who is held captive in a castle by a dragon. To slay the dragon you need a sword and some armour. You can get a sword from the Lady of the Lake, and you can find some armour in the village smithy. Thus, the game consists of five distinct locations:



In text adventures, one typically navigates from location to location using the go command. For example, when starting in the forest, to get to the crossroads one would type:

```
go east
```

When at a location, one can interact with the items at the location using other text commands. For example, to kill the dragon we could use:

```
kill dragon
```

Here is an example of a herioc effort succeeding to slay the dragon and free the princess:

```
You are in a dark forest.  There is a path leading to the east.
go east
You are at a crossroads. To the east you see the shores of a lake. You can hear the buzzling
noise of a village in the south. To the north, a magnificent castle towers over the trees of
the forest.
go south
You are in a busy village.
The smith has a magnificent set of armour on display.
take armour
You put on the armour. The smith alerts the guards to stop the thief, but their weapons are
repeled by the armour and they let you go.
You are in a busy village.
go north
You are at a crossroads.
go east
You are at a misty lake. It is dead silent, and the water is calm.
A mysterious hand thrusts itself up from the water, holding aloft a magnificent sword.
take sword
You take Excalibur and shake the hand of the Lady of the Lake to thank her.
You are at a misty lake. It is dead silent, and the water is calm.
go west
You are at a crossroads.
go north
You are in front of a grand castle. At the top of the tower, you see a princess waving for your
 attention. She is in distress, and clearly a prisoner who needs your help.
A dragon guards the entrance of the castle.
kill dragon
You slay the dragon and rescue the princess. You win!
```

However, in text adventures it is very easy to die. For example, if you do not have a sword and armour, the dragon will kill you:

```
A dragon guards the entrance of the castle.
kill dragon
The dragon eats you and you die.
```

3.2 Implementing the Text Adventure Game

To implement this game, we are going to use one method for each of the locations. That is, create the following five methods: `forest`, `crossroads`, `village`, `lake`, `castle`. Walking from one location to the next thus simply consists of calling the appropriate method for the next location.

Each of the methods should do the following:

- Print a description of the location. (Copy and paste the example text from this labsheet, or write your own dramatic text)
- Read a command from the user, remove the newline (`chomp`), and convert to lower case (`downcase`).
- Take an action based on the user input. That is, create an `if` condition that compares the user input with different commands that should be valid at that location. For example:
  - `go east` - this should be possible in the forest and the crossroads. For all other locations, the possible directions will be different (see map above, and table below).
  - `take armour` or `take sword`: In the village it should be possible to get some armour, and at the lake you can get the sword.
  - `kill dragon`: When at the castle, you can attempt to kill the dragon.
  - If the user types an unknown command, then print "`I don't understand that.`"
- At the end of each action, invoke the method that represents the next location. That is, if the user types `go east` in the forest, then the action will be to call method `crossroads`. If the user typed an invalid command, then invoke the method of the current location again.
- If the dragon is killed or the player dies, print a message stating this and quit the game (using the command `exit`).
- To make it a bit more challenging, the dragon can only be slain if the player possesses both, armour **and** sword. To keep track of whether these items have been picked up, all methods should take two parameters `armour` and `sword` of Boolean value which are initially `FALSE`.
- Feel free to add more drama and commands to your adventure. For example, allow your player to drown in the lake, get killed by the guards in the village, etc.

Method	Description	Commands
`forest`	You are in a dark forest. There is a path leading to the east.	`go east`
`crossroads`	You are at a crossroads. To the east you see the shores of a lake. You can hear the buzzling noise of a village in the south. To the north, a magnificent castle towers over the trees of the forest.	`go north`, `go east`, `go south`, `go west`
`lake`	You are at a misty lake. It is dead silent, and the water is calm. *If sword is not already owned add:* A mysterious hand thrusts itself up from the water, holding aloft a magnificent sword.	`go west`, `take sword`
`village`	You are in a busy village. *If armour is not already owned add:* The smith has a magnificent set of armour on display.	`go north`, `take armour`
`castle`	You are in front of a grand castle. At the top of the tower, you see a princess waving for your attention. She is in distress, and clearly a prisoner who needs your help. A dragon guards the entrance of the castle.	`go south`, `kill dragon`

3.2.1 Getting Started

Start by creating a new project in RubyMine. To be able to run the game, just invoke the first method (`forest`) at the end of the file, with parameters to indicate neither armour nor sword are possessed:

```
forest FALSE, FALSE
```

If you are unsure of how to implement the methods, you can find an example implementation of the *forest* method as a starting point on the MOLE page of the module, and extend that with the remaining four methods.