1. **TicTacToe Slides with iClicker**
2. **TicTacToe Plagiarism?**

After a comparison of the code turned in by everyone, an alarming amount of the submissions seem to have a disturbing amount of code in common.

Your course staff is looking into this and comparing code by human examination. If you have violated or think you may have violated the policies for this course, you may want to...

1. **Finite State Machines**

This graph represents a FSM with:

* Nodes
* DIrected edges

**Example: Parse “Nice”:**

<https://en.wikipedia.org/wiki/Finite-state_machine#/media/File:Fsm_parsing_word_nice.svg>

We move from state to state in response to inputs to a given state. In the example, while “n” is an acceptable input to the “Start” state, it is not an acceptable input to any of the other states.

It is trivial to hard code a FSM like this. This example is even easy to do in a versatile manner because we can send two strings for comparison which makes it decoupled from the data itself. An object that merely determined if a string matched exactly one case such as the word “nice” is not very useful.

**Example: ATM**

<https://people.engr.ncsu.edu/efg/210/s99/Notes/fsm/>

What if we wanted to be able to create an engine that would take a description of the actions and give user feedback without hard coding anything? A user interface like the ATM.

1. **Adventure (The FSM)**

<https://courses.grainger.illinois.edu/cs126/sp2020/assignments/amazing-adventures/>

We can make a FSM game by coding the machine logic directly into our code but this violates the principle of decoupling our data from our code. It would be a very inflexible design. This is where our JSON schema/data come into play. We can build an engine to process the FSM described by the JSON.

Our schema shows:

* Direction
* Room
* Layout

Let’s look at these in hierarchical order:

**Layout:**

 This is the entire map. It is the highest level construct in this graph. It lists all the nodes which we call “rooms”.

“Layout” contains 0 or more “rooms”.

**Room:**

 This is a node in a graph. It is identified by its “name” field and it has a list of relationships to other rooms which we call “direction”. If there is a direction, the room associated with that direction must exist to have a “correct” map. A room’s “name” must be unique to avoid ambiguity.

“Room” contains 0 or more “directions”.

**Direction:**

 This information uses a string to distinguish each edge. Again, they could be named anything but we use directions that users will associate with a physical layout. For each edge, the name of the node (room) is included.

We would typically use

* “North”, “south”, “east”, “west”
* “Up”, “down”
* “Crazy”, “sane”
	+ Because our user would expect to be able to return to a previous room, we have “sane” to return from “crazy” but nothing requires this from a functionality standpoint.
	+ A room could have a one-way entrance or exit if there is no reciprocal “direction” in the adjoining room. This would allow traps that have no exit.
1. **Start Your Engines!**

We begin in “startingRoom and traverse the graph to find the end. To do this, we move from room to room testing to see if we are in the “endingRoom”. Otherwise, we offer the list of “directions” as options to the user. Our engine uses the selected “direction” (edge) and uses the corresponding room name to find the “room” in the “layout”.

For this task, we will need to:

1. Read in and parse game data (JSON)
2. Take user input from the console
3. Send output to the user through the console
4. Process/interpret user input to move through the adventure

**6) Java InputStream and OutputStream**

**Getting User input (Scanner)** - intereactive

<https://www.w3schools.com/java/java_user_input.asp>

**How to use PrintStream to write to the console:**

* <https://stackoverflow.com/questions/51625966/java-how-to-use-printstream-outputstream-to-print-to-the-command-line>
* <https://www.geeksforgeeks.org/java-io-printstream-class-java-set-1/>
* <https://www.geeksforgeeks.org/printstream-printlnobject-method-in-java-with-examples/>

**Example:**

In this version of David Brewster’s “NumberGuesser”, he uses “System.out” for the outputStream

(lines 19-22)

“Logger” is a PrintStream that takes “System.out” as its destination. This design allows for easy modification to send the output to other destinations including files.

<https://github.com/CS126SP20/NumberGuesser/blob/master/src/main/java/edu/illinois/NumberGuesser4.java>

* We could specify a different destination for our output by calling with a different OutputStream using this form of NumberGuesser4:
	+ public NumberGuesser4(int lo, int hi, OutputStream outputStream)
* Similarly, a different source for input can be specified by using:
	+ public int play(InputStream inputStream)
* **InputMismatchException:**
	+ <https://airbrake.io/blog/java-exception-handling/inputmismatchexception>

======================================]

David’s Comments (edited by MJW):

======================================

* The game engine uses input/output streams, so it is very generic. For example, one could pass an input or output stream whose underlying datasource is a file.
	+ MJW Note: it would be difficult to play the game if your output was going to a file. However, we could send our output to a game AI that would play the game for us. In this way, you could make a game player to exhaustively play the map to detect problems and log results.
* Also, in the tests, he is using ByteArrayInputStreams and ByteArrayOutputStreams.. “The PrintStream used in the NumberGuesser class is for convenience/abstraction: I want to think about writing high-level objects like Strings, ints, floats, etc., not bytes.”

**Example Testing**

Below, “baos” and “bais” in AdventureTest allow our tests to pass input and place the output where it is accessible to our test.

**ByteArrayOutputStream:**

<https://www.geeksforgeeks.org/io-bytearrayoutputstream-class-java/>

**ByteArrayInputStream:**

<https://www.tutorialspoint.com/java/java_bytearrayinputstream.htm>

In NumberGuesser5Test, notice how David is able to create user input to passed into the tests through

ByteArrayInputStream **inputStream**; (line 19)

which gets its data by joining “guesses” into a string and placed in “input”

String[] guesses = { **"1"**, **"2"**, **"3"**, **"4"**, **"5"** };

String input = String.*join*(**"\n"**, guesses) + **"\n"**;

**inputStream** = Utils.*fromString*(input);

**Online Java** <https://www.tutorialspoint.com/compile_java_online.php>

// mike’s test version of David’s code for above environment

import java.io.InputStream;

import java.io.OutputStream;

import java.io.PrintStream;

import java.util.InputMismatchException;

import java.util.Scanner;

import java.util.Random;

/\*\*

 \* `NumberGuesser4`.

 \* @author anonymous.

 \*/

public class NumberGuesser4 {

 private int lo;

 private int hi;

 private PrintStream logger;

 private Integer correctNum;

 public NumberGuesser4(int lo, int hi) {

 // Use System.out by default.

 this(lo, hi, System.out);

 }

 public static void main(String[] args) {

 NumberGuesser4 myGame = new NumberGuesser4(10, 1000);

 myGame.play();

 }

 public NumberGuesser4(int lo, int hi, OutputStream outputStream) {

 this.lo = lo;

 this.hi = hi;

 this.logger = new PrintStream(outputStream);

 this.correctNum = null;

 }

 /\*\*

 \* Attempts to get the guess from the input source.

 \* @param input

 \* @return

 \*/

 private Integer getGuess(Scanner input) {

 try {

 return input.nextInt();

 } catch (InputMismatchException e) {

 logger.println("You didn't input a number.");

 return null;

 } catch (Exception e) {

 // Output unexpected Exceptions.

 logger.println(e);

 return null;

 }

 }

 /\*\*

 \* Determines whether or not the guess was correct.

 \* @param guessedNum

 \* @return

 \*/

 private boolean isCorrectGuess(int guessedNum) {

 if (guessedNum > correctNum) {

 logger.println("Too high!");

 return false;

 } else if (guessedNum < correctNum) {

 logger.println("Too low!");

 return false;

 } else {

 // They guessed the correct number.

 logger.println("Yay! " + correctNum + " was the correct number.");

 return true;

 }

 }

 public int play() {

 return play(System.in);

 }

 /\*\*

 \* Plays the game and returns the number of attempts taken to guess the correct answer.

 \* @param inputStream

 \* @return the number of attempts if successful, else returns -1.

 \*/

 public int play(InputStream inputStream) {

 Scanner input = new Scanner(inputStream);

 Random rand = new Random();

 //correctNum = Utils.randomInteger(lo, hi);

 correctNum = rand.nextInt(1000);

 int numAttempts = 0;

 logger.print("Implementation 4\n================\nType a number: ");

 while (input.hasNextInt()) {

 // Check if input is valid.

 Integer guessedNum = getGuess(input);

 ++numAttempts;

 if (guessedNum == null) {

 continue;

 }

 if (isCorrectGuess(guessedNum)) {

 return numAttempts;

 }

 logger.print("Type another number: ");

 }

 return -1;

 }

}