

# COSC 2P13—A2—Would You Like to Play a Game?

For this assignment, you'll be creating a full software package that includes a server and GUI-based client, as well as supporting direct text connection, to support a simple networked multiplayer card game.

## CoExistence

CoExistence is the newest game that's sweeping the nation! It's simple to learn, but difficult to master! (Actually I'm lying about all of this; it's the most thinly-obfuscated game of rock-paper-scissors ever)

It's ostensibly a card game, where two players are each dealt **unit cards** of varying types, and then take turns using those units to *attack* the other player's units. The units are simple:

- The **axe** cuts true; straight through the handle of the **hammer**, securing one point of glory
- The **hammer** is power; it overwhelms the **sword**, securing one point of glory
- The **sword** is nimble; it dances about the **axe**, striking its wielder and securing one point of glory
- The **arrow** is cowardly but reaches far; it can eliminate any other unit, but earns no glory!
  - Any other unit can similarly defeat it, but again earning no glory!

A standard CoExistence **deck** consists of 12 cards: 4 of each type listed above.

(Other decks are legal by the game's rules, so long as they add up to at least 12)

At the start of each round, the deck is *shuffled*, and 6 cards are *dealt* to each player.

For the first round, the starting player is chosen randomly. Both players take turns choosing a unit to perform an attack, and a target. Only killing blows are permitted (e.g. a hammer may not attempt to attack an axe).

A player may pass any round, and *must* pass if unable to defeat any enemy units.

The round ends when both players have consecutively passed (whether voluntarily or of necessity).

Rounds repeat until either one player **wins** by hitting **9** points total, or the round number hits round 5, indicating that *both* players **lost**.

## The CoExistence Server and messaging format

You'll be creating a server that waits for pairs of players to join, pairs them up, and spins them off into a separate game thread (readying for the next incoming player). CoExistence is designed to be simple to play without *needing* to download a standalone client. To achieve this, it uses a standardized plaintext frame. The server uses a grid-based format for communicating with clients, and receives very simple commands in response.

### Server-to-Client format

One possible sample of a mid-game state is:

01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40		
/	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	\						
02		A		B		C		D		E		F																													
03	/	-	-	-	\	/	-	-	\	/	-	-	\	/	-	-	\	/	-	-	\	/	-	-	\	[	=	X	=	]	-	-	-	-	-						
04	[	=	]																							<	7	>	o	0											
05																										I		v		v											
06																										L		-	-	-	-	-									
07	\	-	-	-	/	\	-	-	/	\	-	-	/	\	-	-	/	\	-	-	/	\	-	-	/	[	=	X	=	]											
08																																					[	1	]		
09	<	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	>	R	3	<				
10																																							[	4	]
11	/	-	-	-	\	/	-	-	\	/	-	-	\	/	-	-	\	/	-	-	\	/	-	-	\																
12	[	<	7	>	]																																				
13	I																									I		P	l	a	y	e	-								
14	L																									L		r	W	o	n	!									
15	\	-	-	-	/	\	-	-	/	\	-	-	/	\	-	-	/	\	-	-	/	\	-	-	/												v				
16	A			B		C		D		E		F																													
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-					
18	M	e	s	s	a	g	e	L	o	g																															
19	\	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	/					

Some important details:

- It is always a fixed size
  - Newlines only indicate the boundaries of rows, so their potentially varying size per-OS is not considered here
- It is in a human-readable format, to ensure anyone *could* play from e.g. netcat
  - Note: this is **mandatory**!
- The first four units of the bottom row are the axe, hammer, arrow, and sword
- Both players are not given (quite) the same message frame:
  - Each player is always listed at the bottom of the received frame (i.e. you'll ‘see the game’ from *your perspective*)
  - This means the rows of cards need to have their positions swapped along with the scores, as well as flipping the direction of the current-player-indicator (it uses a  $\vee$  to point to the ‘current player’, and a carat —  $\wedge$  — to point to the ‘other player’. When it’s game over, both ends of each arrow are simply a —
  - As a general tip, if you internally use something like an integer to indicate ‘whose turn it is’ — say, 0 and 1 — then you can quickly get the other player’s side by subtracting the ‘current turn’ from 1
    - ◆ i.e. if it’s player 1’s turn, then  $1-1=0$ , so 0 is ‘the other player’; if it’s player 0’s turn, then  $1-0=1$ , so 1 is ‘the other player’
    - ◆ If this tip doesn’t help? Then just ignore it! Or ask your instructor to draw it on the board
- For the sake of a client program, you only need to anticipate one point of failure with respect to frames coming from the server: it’s theoretically possible for a frame to ‘reset’ partway through, so if you see the /-----\ that’s the start of a message, assume you’re ‘starting over’ and ignore any partial frames you’ve received

You might notice not *all* of the message frame's contents are entirely functional:

- The header is required, to establish the start of a frame
  - The twelve  $3 \times 3$  tile areas can be used to identify unit cards
  - The player-identifier area is entirely optional (I left mine blank for all sample executions)
  - The two turn-indicators can be relied upon to always match each other, so you can use either
  - The scores and round number are always in the same place
  - The message log is fixed-width, even if most of those values are simply spaces
    - Notice it isn't quite the full width of the frame
  - Everything else? There needs to be something there (to keep byte-counts constant), but it's permissible to be lines, all spaces, illustrations, etc.
  - On that note, though the message log should be displayed to the user, a GUI client doesn't need it for anything internally (since you can infer turn, round, etc. based on the rest of the frame)

## Client-to-Server format

These are always just basic commands. Each is two characters, then newline-terminated.

- For making a move, it's the letter corresponding to the 'from', followed by the letter corresponding to the 'to'. e.g. CE uses the third column's unit to attack the fifth column's enemy unit
  - PS passes the turn
  - All commands are canonically uppercase, but due to possibilities of human error/laziness, allow for any combination of upper- and lowercase
  - From the perspective of the server, *basic* resilience is necessary:
    - If a user tries submitting a syntactically correct but illegal move? Indicate as much within the message log (`INVALID MOVE`) and leave it still on the same player's turn
    - If a user tries submitting a nonsensical or incomplete move? Again, log it (`SYNTAX ERROR`) but play remains on the same player's turn, who should simply 'try again'

- If a player *disconnects*, there's no need to worry about recovering from that; simply ensure the entire server's program doesn't (somehow) crash
- If you're even remotely following the provided sample code, this will be a non-issue

## Software Components

To be clear, you must create your IntelliJ project to allow for multiple independent roles. This includes having more than one **run configuration** configured (remember to click to have it stored within the project folder). If you are at all unclear about *any* of this, tips will be given during the allotted lecture time.

### The Server

This is where most of the 'game' will reside. It's up to you e.g. how many classes to break this up into. Note that this doesn't mean 'every design decision is correct' — it means you're to figure it out. Some things have multiple reasonable approaches; others not.

### The GUI Client

You're to write a simple Java Swing client to let a user play the game.

It's okay if you have no experience with Swing (it's actually assumed that you *don't*); this is a good opportunity to learn a new API! For something 'turn-based' like this, it should be a bit easier than many other GUI-based tasks.

Note that part of your task will be to 'parse' a message from the Server. Keep the requirements above in mind, including which sections really 'matter'. e.g. lines 02 and 16 are *solely* for the benefit of text-based players; your code can simply ignore those lines completely.

You'll need to be able to identify Units based on  $3 \times 3$  panels of text, so it'll *probably* be easier if you 'pre-load' all 19 lines before you look too hard at the details. Considering a frame could be restarted at any time, this would also make that aspect easier.

There's no singular correct means for how this should look. Part of 'the point' is that there's a single standardized format for message frames, but then how those messages are used is largely up to the individual implementation. This is remarkably common in 'real' software development.

That said:

- There's no reason to permit a player to enter a move while it's not their turn
- It doesn't matter whether you display pictures for each of the unit types, or text, but don't just copy over the message frame's contents.
  - e.g. saying "Axe"? when it's the text pattern of an axe? Acceptable! So is an illustration of an axe
  - The point is that you're supposed to be extracting semantic meaning from each frame
- Similarly, the indication of turn should be based on extracting that info, to affect the display
- It's perfectly fine to just display the log as the log, so long as it's trimmed appropriately

### The Terminal Client

There isn't one.

Seriously, the 'terminal client' is just... the terminal. Like netcat or something equivalent.

## Notes:

- Although it's acknowledged above that other decks are legal, that's just because, as-is, the game doesn't allow for very 'deep' gameplay. A 'proper' implementation would probably balance the types a bit more, or simply shake up the distributions to avoid falling into the pitfall of always doing the same gameplay. But though you're *permitted* to support that, it's **completely** outside the scope of this assignment to do so (no marks, no bonus marks, nuttin')
- Reminder: not every byte of the Server-to-Client format needs to actually be *used*:
  - Clients are not required to acknowledge player names if they don't want to
  - Implementing servers and Client GUIs are not required to allow for custom player-names (or similarly for multi-line custom player-names)
    - ◆ If you *do* wish to allow you, you could e.g. let a client specify a name in a Java-like convention (say -----\\n=====\\nEmily\\n=====\\n-----) and then have the server replace the text \\n with \\n characters. That'd require ensuring everything stayed within the 5x5 namebox though, so it's up to you whether you're interested in that. Again, it's outside the scope
  - Refer to the diagram above to see which message cells are really required
    - ◆ (You can recognize that the reason we're covering the format and special cases twice is because it matters a lot, right? Yes? Great!)
  - Again, there'll still be *some* data included in each of those locations
    - ◆ i.e. the 40x19 dimensions are immutable, so unused cells can be e.g. spaces, but they have to be *something*!

## Suggestions on how to begin

First, realize that this isn't one task, but rather several.

- It's accepting, and forking off, pairs of sockets into separate game instances
  - It's 'dealing out' the cards, initializing counters, etc. for a given game
  - It's allocating enough variables to represent a 'game state'
- It's creating a basic loop for the gameplay
  - It's **strongly** advised to write the text-based version *first*
    - ◆ Once the entire game is working, writing a GUI client is barely more work than just
- We'll be devoting the end of the next two lecture times to discussing details of this assignment, so you're strongly encouraged (even more so than usual) to attend
  - Otherwise, please come see your instructor in person
  - Related: no nonsense emails of "can we use this library?" "Are we allowed to —?"

## Submission

Bundle up your entire assignment into a single .zip file and submit it through Brightspace.

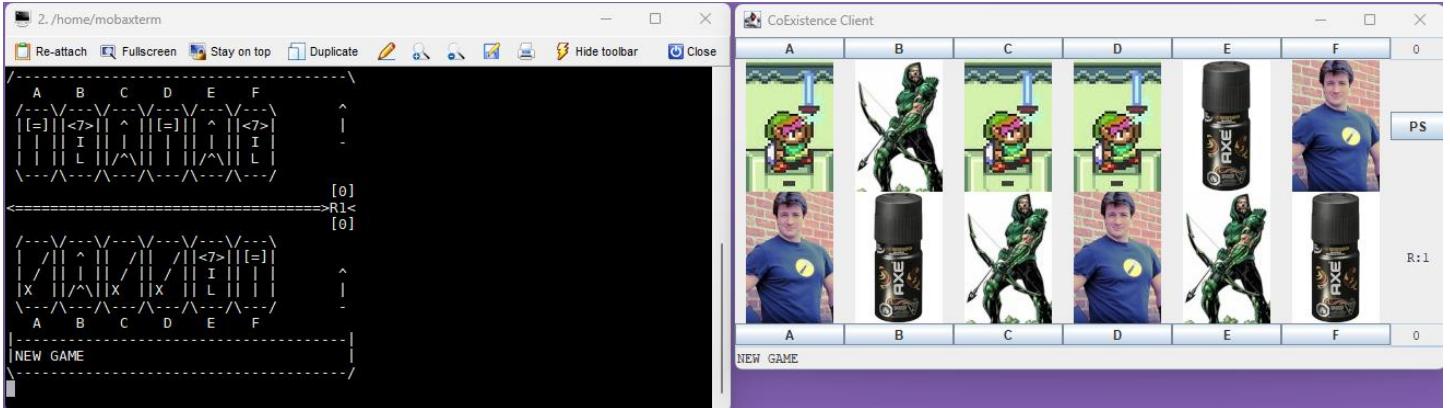
Some reminders:

- It **needs** to be a .zip file, and nothing else
- Friendly reminder: you are developing on a single computer for all of this, or including a note explaining what/why. This nonsense of having fragments of projects or bits and pieces of other work or incomplete submissions is officially over

- Remember that your marker will not put in more work than you did: you're including two run configurations, *included with your project files*. To test properly, your GUI client should be configured to allow for multiple concurrent instances
- Stick to Java 11, both in terms of language features and JDK
  - If you're unsure of what this means, ask your instructor
  - If you're unsure of why this matters, ask your instructor
  - If you're unsure of *anything*, ask your instructor
- Remember that you can submit multiple times, so even if you have something done but are considering still tweaking things, you can submit first, and then resubmit later
  - This is particularly valuable in case you lose track of time and miss that final submission. You'll still have submitted *something*, rather than losing 20% of your final grade
  - Friendly reminder: no matter what happens, no late submissions will be accepted. No note, self declaration, or unfortunate story about your car having been eaten by a grue will change that

## Sample Execution

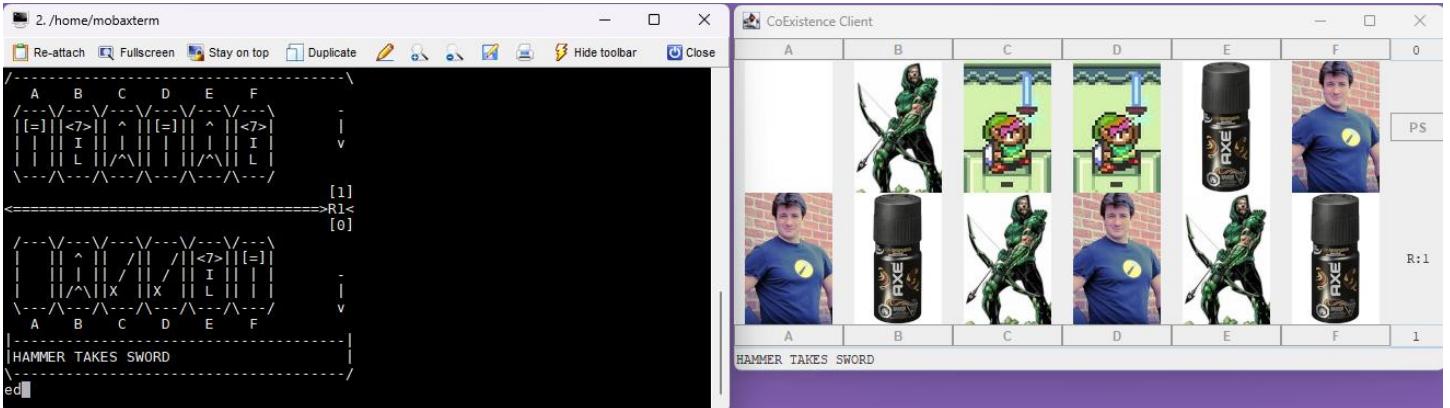
Here's an example of a text client playing against a GUI client. So long as you follow the prescribed conventions, you can do that! (Actually, so long as you follow them in full, I can connect to **your** server with **my** client!)



Notice the sword in the A column of the left (text) view's bottom row aligns with the top row in the right.

Also, since it's the right player's turn, the buttons are active. Both sides are displaying the same log.

Let's say the right player attacks from D to A:



Since it's the left player's turn, the buttons are disabled on the right side.

You can also see the right player has 1 point for having defeated a unit.

The left player wants to make use of the single axe before it's gone, and so typed the ed. We get:

The terminal window shows a game board with columns A through F and rows 1 through 6. The board contains various symbols representing characters and actions. The last move shown is "AXE TAKES HAMMER".

The CoExistence Client window shows a 6x6 grid of icons. Columns A-F represent the board columns, and rows 1-6 represent the board rows. Row 1 (R:1) shows icons for Axe, Archer, and Hammer. Row 2 (R:2) shows icons for Hammer, Axe, and Archer. Row 3 (R:3) shows icons for Hammer, Axe, and Archer. Row 4 (R:4) shows icons for Axe, Hammer, and Archer. Row 5 (R:5) shows icons for Axe, Hammer, and Archer. Row 6 (R:6) shows icons for Axe, Hammer, and Archer.

And now both scores are tied up. The right player doesn't like the control the left's Arrow could afford, and so takes it out:

The terminal window shows the same game board as before, with the last move being "AXE TAKES HAMMER".

The CoExistence Client window shows the same grid of icons. Row 1 (R:1) shows icons for Axe, Hammer, and Archer. Row 2 (R:2) shows icons for Hammer, Axe, and Archer. Row 3 (R:3) shows icons for Hammer, Axe, and Archer. Row 4 (R:4) shows icons for Axe, Hammer, and Archer. Row 5 (R:5) shows icons for Axe, Hammer, and Archer. Row 6 (R:6) shows icons for Axe, Hammer, and Archer.

That wasn't really such a great move. Besides earning no points, it leaves the left's Axe available to take out the right's (Captain) Hammer:

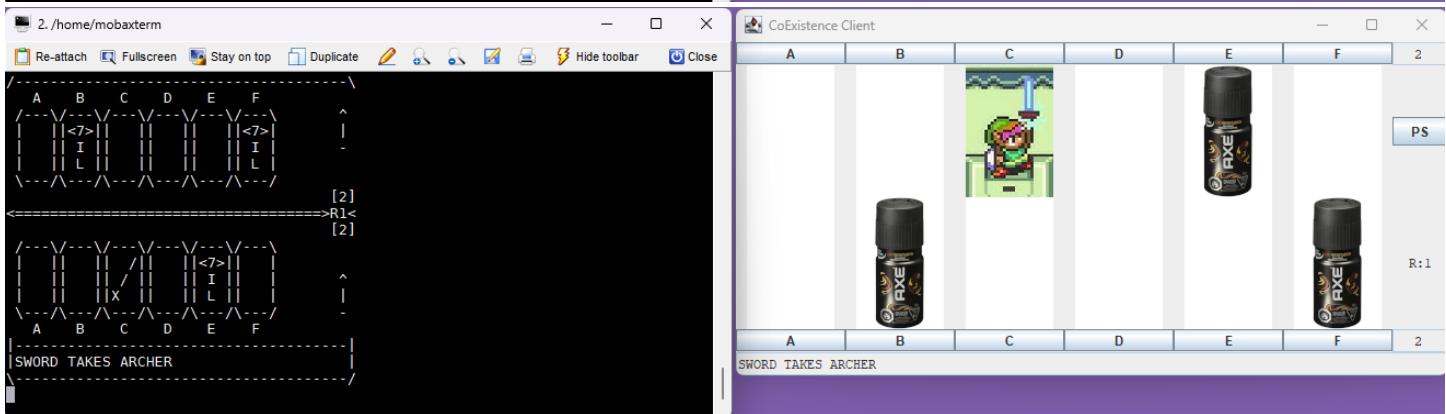
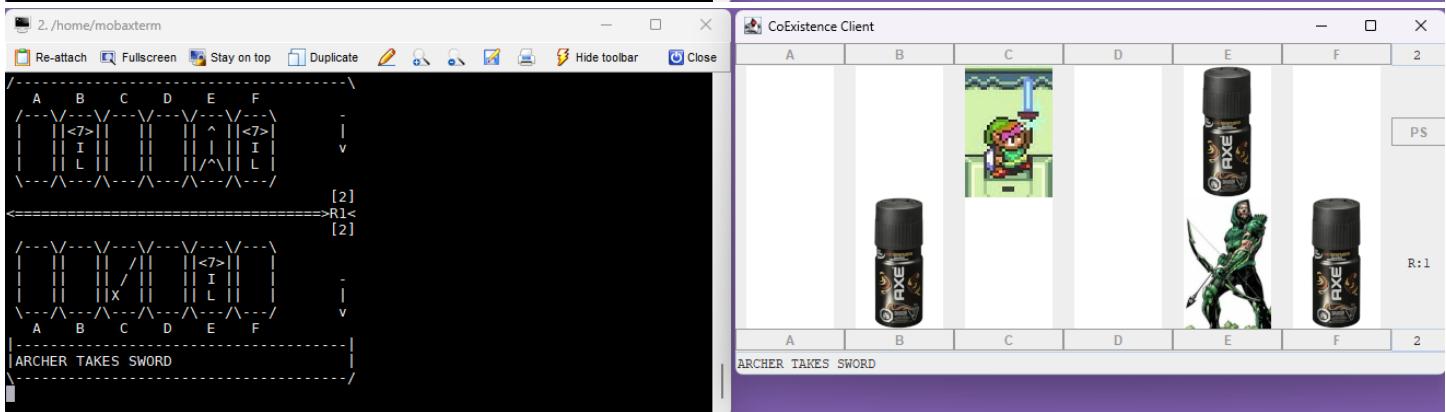
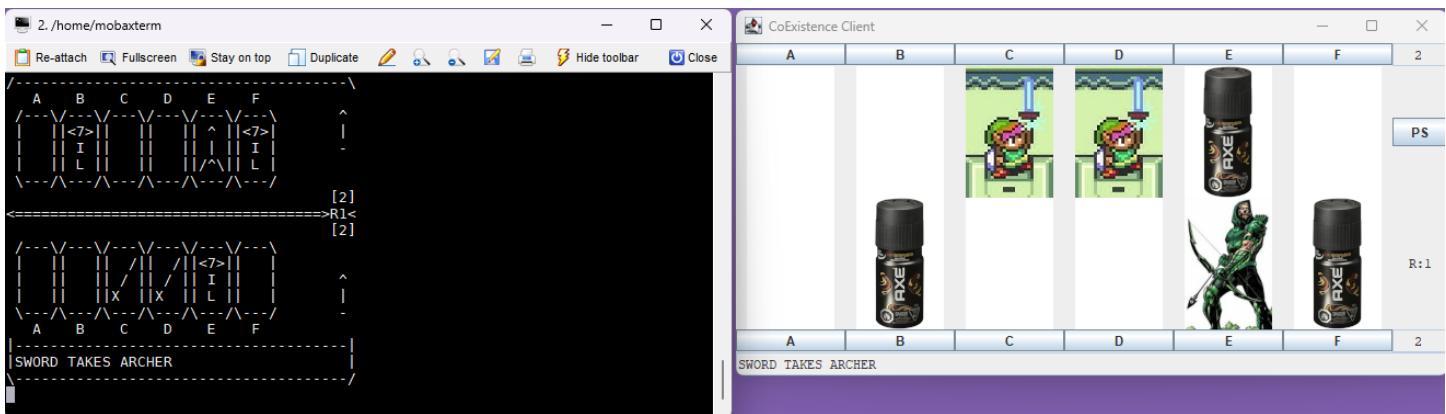
The terminal window shows the game board with the last move being "AXE TAKES HAMMER".

The CoExistence Client window shows the grid of icons. Row 1 (R:1) shows icons for Axe, Hammer, and Archer. Row 2 (R:2) shows icons for Hammer, Axe, and Archer. Row 3 (R:3) shows icons for Hammer, Axe, and Archer. Row 4 (R:4) shows icons for Axe, Hammer, and Archer. Row 5 (R:5) shows icons for Axe, Hammer, and Archer. Row 6 (R:6) shows icons for Axe, Hammer, and Archer.

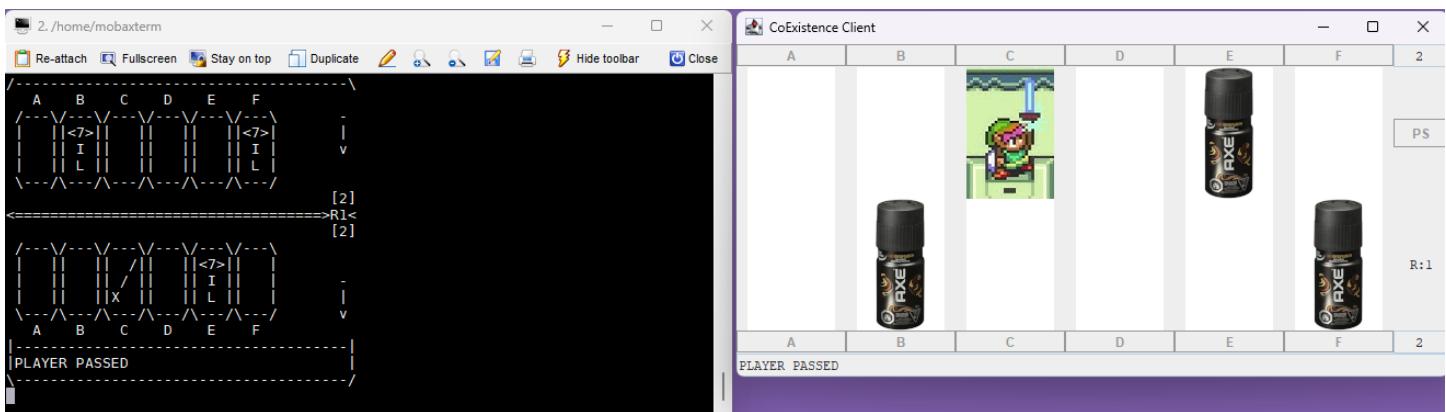
Okay, let's zip through a couple turns:

The terminal window shows the game board with the last move being "AXE TAKES HAMMER".

The CoExistence Client window shows the grid of icons. Row 1 (R:1) shows icons for Axe, Hammer, and Archer. Row 2 (R:2) shows icons for Hammer, Axe, and Archer. Row 3 (R:3) shows icons for Hammer, Axe, and Archer. Row 4 (R:4) shows icons for Axe, Hammer, and Archer. Row 5 (R:5) shows icons for Axe, Hammer, and Archer. Row 6 (R:6) shows icons for Axe, Hammer, and Archer.

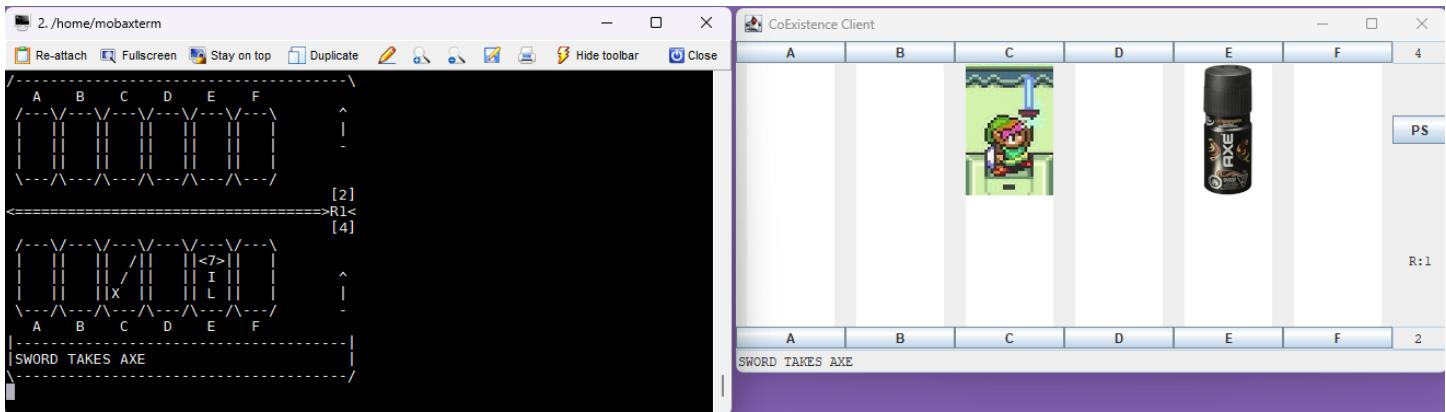
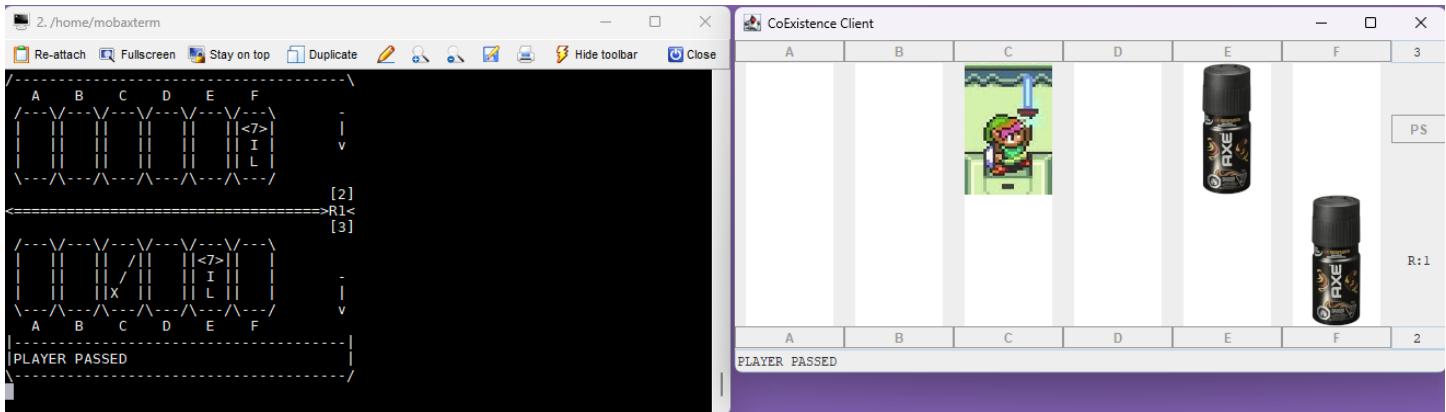
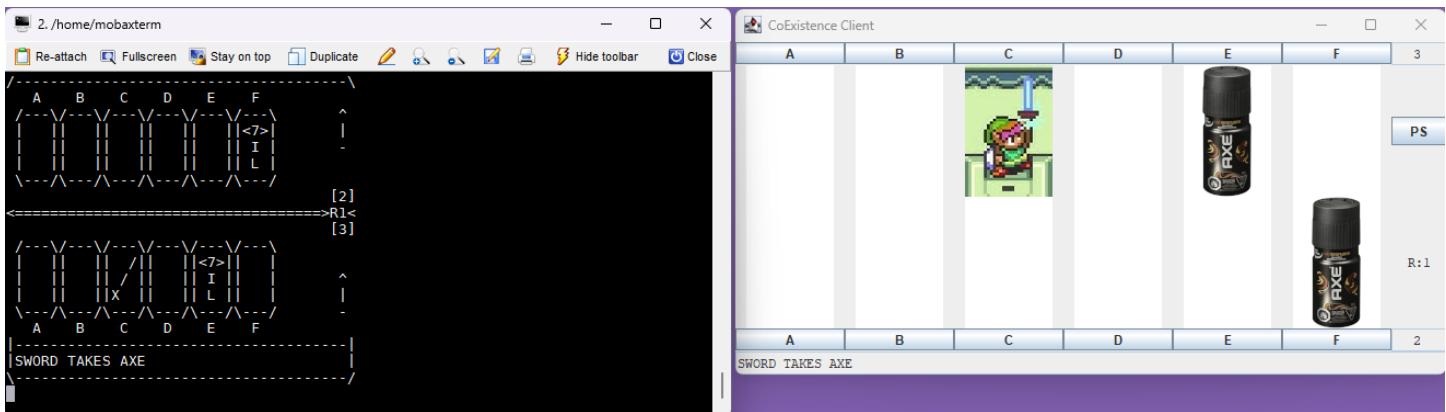


You'll notice the right player's kinda stuck, so *passing* is the only option.

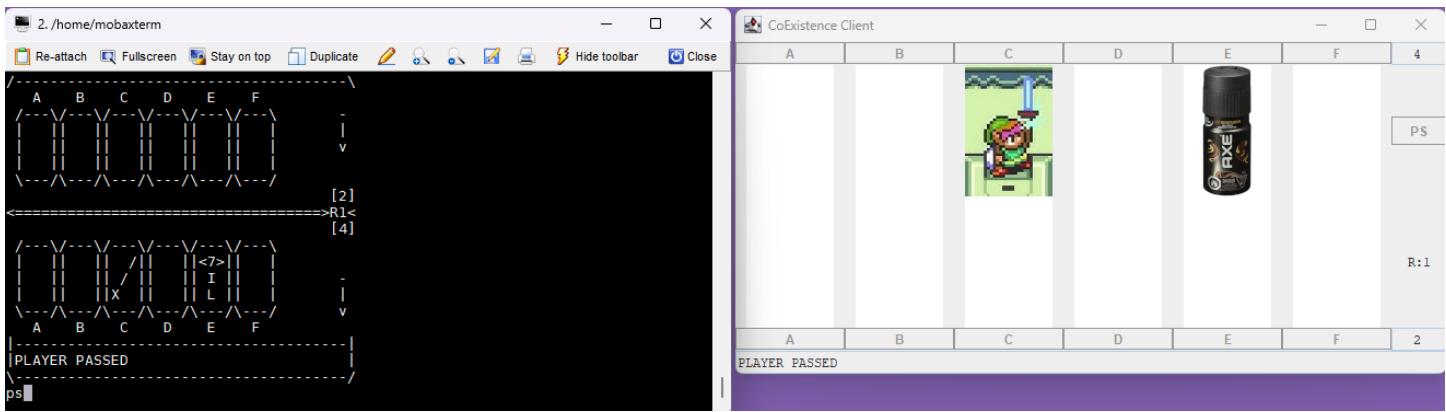


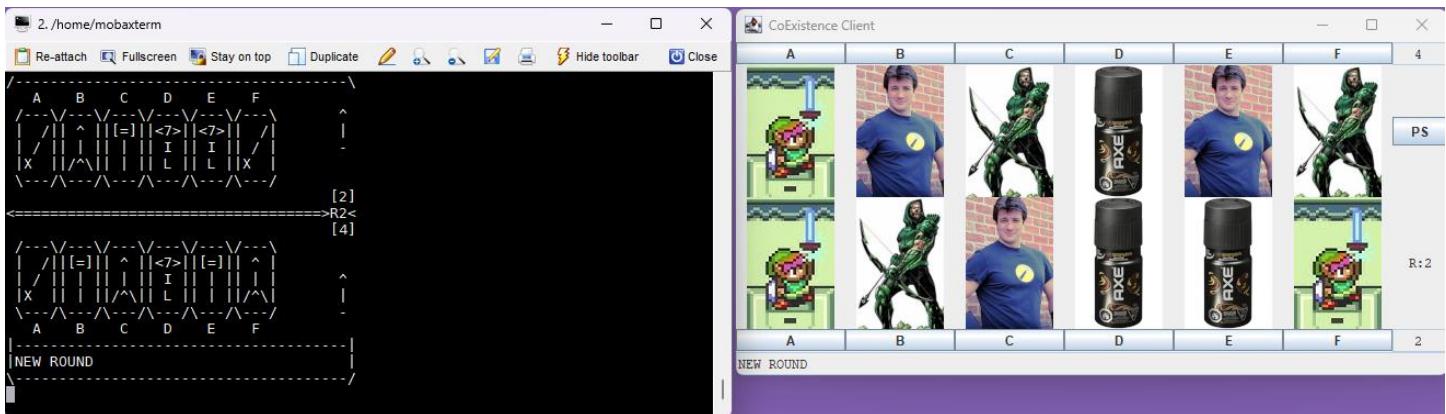
Left could also pass (to move to the next round), but... points are nice?

Instead, the left player eliminates the two axes:

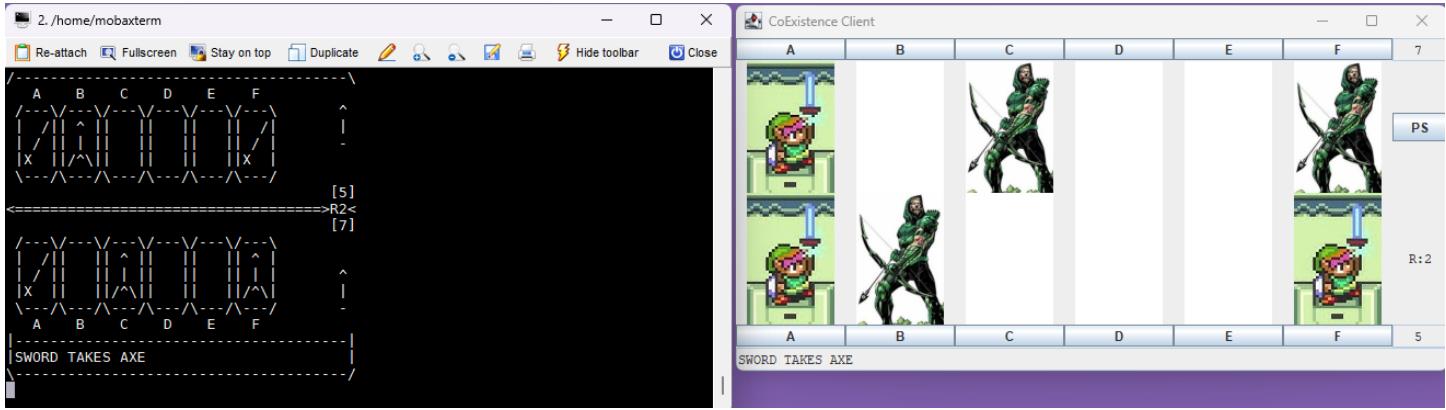


And now, both players have no choice but to pass:



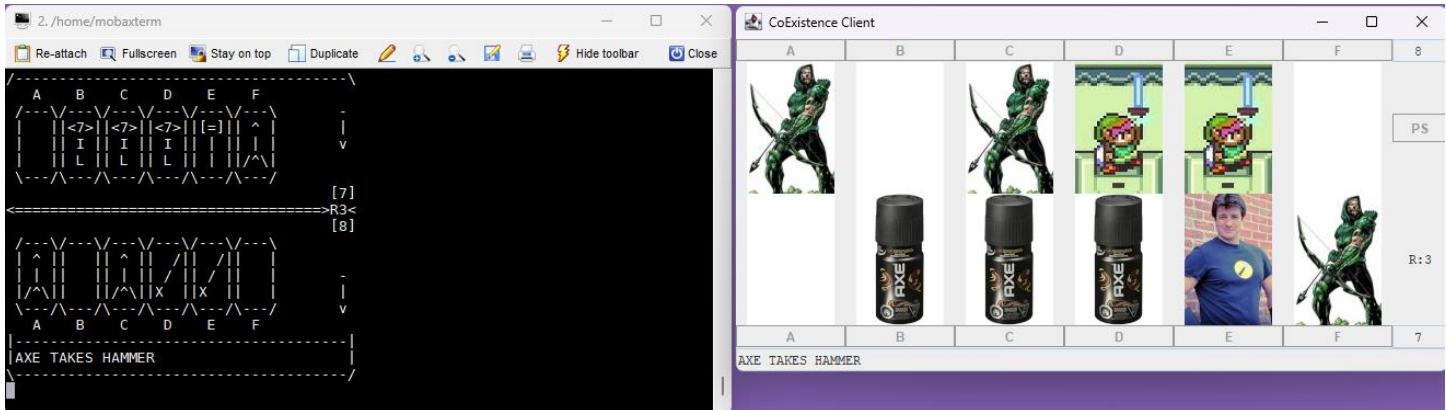


Notice the *round counter* has incremented. For expediency, let's just skip even showing a few turns:

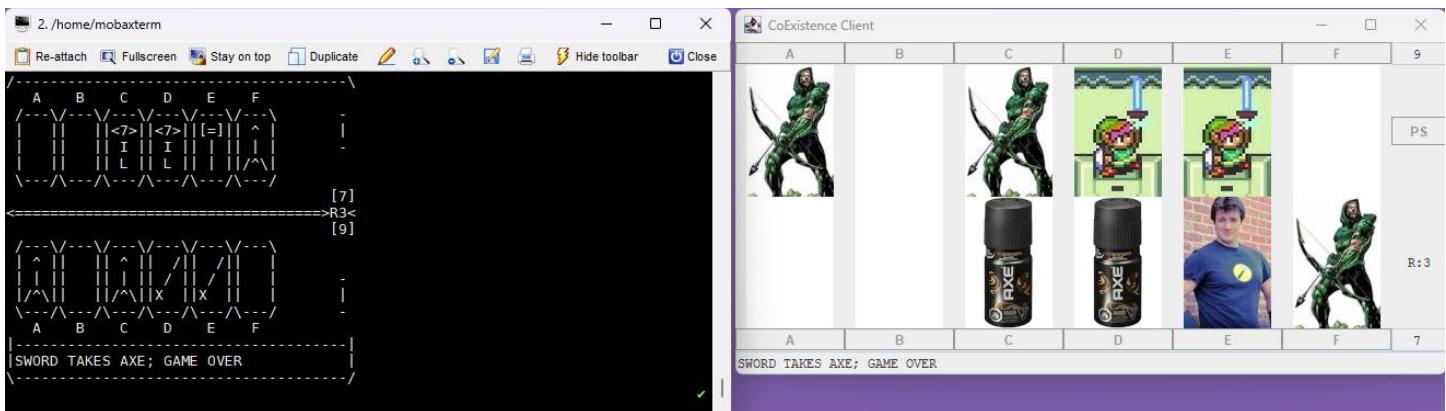


After a minor massacre, they might just both pass to start the next round, or maybe only one would.

Anyhoo, let's skip to the end of the game:



The score is 8:7, and it's left's turn. Left could just take out an Axe:



*If* a GUI client wanted to detect that the game was over, there's no need to consider the log: the turn indicator has neither  $\wedge$  nor  $\vee$  (rather it has only  $-$ ). That means nobody gets to make a move (and thus, game over). You'd see the same for a tie (Round Number reaching 5).