\*Unbeatable Clojure **Tic Tac Toe** implementation

Introduction Description

This implementation of Tic-tac-toe features an unbeatable computer opponent using the minimax algorithm. The program is structured with clear separation of concerns across multiple modules handling game state, AI logic, and console interaction. The codebase consists of 324 total lines including 301 lines of source code and 55 lines of comments, complemented by 813 lines of comprehensive testing. I changed the program a couple times but thought I would make this documentation for some interesting notes about the stats for the program and also how I developed the play-game function over time.

## **Index**

## **Architecture**

## **Play-Game Evolution**

## 

## 

## 

## 

## **Architecture**

The application is divided into four main modules:

console 134 lines of code 1 comments

ai 52 lines of code 11 comments

state 72 lines of code 15 comments

main 66 lines of code 20 comments

324 total lines of code, 277 lines of source code, with 47 lines of comments

ai\_testing 298 lines of code

-> 41 passing cases

console\_testing 141 lines of code

-> 25 passing cases

state\_testing 269 lines of code

-> 41 passing cases

main\_testing 105 lines of code

-> 13 passing cases

Original play-game ‘megafunction’

(defn play-game

[player-name stats]

(loop [current-board board-state

winner

(let [is-player-win? (= winner "X");loss condition or 'win' if win was possible.

new-stats (console/update-stats current-stats (if is-player-win? :win :loss))]

(console/print-draw-or-computer-win false is-player-win? player-name)

(console/print-stats new-stats 0)

(if (console/play-again?)

(recur board-state 0 new-stats)

new-stats))

(>= moves-made 9);draw condition

(let [new-stats (console/update-stats current-stats :draw)]

(console/print-draw-or-computer-win true false player-name)

(console/print-stats new-stats 0)

(if (console/play-again?)

(recur board-state 0 new-stats)

new-stats))

:else;if not a loss or draw, the game continues on and the computer takes a turn

(let [row (get-valid-input "Enter your row [0, 1, 2]:" valid-input?)

col (get-valid-input "Enter your column [0, 1, 2]:" valid-input?)

board-after-player-move (make-move current-board row col "X")]

(if (= board-after-player-move current-board)

(recur current-board moves-made current-stats)

(if (= moves-made 8)

(recur board-after-player-move (inc moves-made) current-stats)

(let [[\_ [ai-row ai-col] :as minimax-result] (minimax board-after-player-move "O" 9)]

(if (or (nil? ai-row) (nil? ai-col))

(recur board-after-player-move (inc moves-made) current-stats)

(let [board-after-computer-move (make-move board-after-player-move ai-row ai-col "O")]

(recur board-after-computer-move (+ moves-made 2) current-stats)))))))))))

First major revision: conditional statement with branches for lose, draw, and continue. Lots of coupling still and many arguments in function calls. Easier to read and shorter.

(defn play-game

[player-name stats]

(loop [current-board board-state

moves-made 0

current-stats stats]

(console/print-board current-board)

(let [winner (check-winner current-board)]

(cond

winner

(let [is-player-win? (= winner "X");loss condition or 'win' if win was possible.

[new-stats playing-again?] (console/handle-game-over current-board current-stats winner moves-made is-player-win? player-name)]

(if playing-again?

(recur board-state 0 new-stats)

new-stats))

(>= moves-made 9);draw condition

(let [[new-stats playing-again?] (console/handle-game-over current-board current-stats nil moves-made false player-name)]

(if playing-again?

(recur board-state 0 new-stats)

new-stats))

:else ;if not a loss or draw, the game continues on and the computer takes a turn

(let [[board-after-player-move new-move-count updated-stats] (handle-player-move current-board moves-made current-stats)

[board-after-computer-move final-move-count final-stats] (handle-computer-move board-after-player-move new-move-count updated-stats)]

(recur board-after-computer-move final-move-count final-stats))))))

Second major revision: Using a case in a similar way. Much easier to read and interpret, and the functions take up less space. Still repeated code and side effects in main.

(defn play-game [player-name]

(reset! game-state (assoc board-state :player-name player-name))

(loop []

(let [current-state @game-state]

(console/print-current-board (:board current-state) (:moves-made current-state))

(let [input (when (= (:current-player current-state) "X")

(handle-player-turn))

{:keys [state next-state]} (compute-next-state current-state input)]

(case state

:game-over

(let [[new-stats play-again?] (console/handle-game-over-state next-state)]

(if play-again?

(do

(reset! game-state (assoc board-state

:stats new-stats

:player-name player-name))

(recur))

new-stats))

:invalid-move

(do

(println "Invalid move! Spot already taken.")

(reset! game-state next-state)

(recur))

:continue

(do

(reset! game-state next-state)

(console/print-current-board next-state (inc (:moves-made current-state)))

(recur)))))))

Third Implementation: Still using cases to direct the control of the application. I have attempted to pull state out of the play-game function as much as possible, and try to put as many side-effects and state-transformations in appropriate helper modules rather than in the main module. The result is the easiest version to read for a human, much more scalable and flexible; and even looks better.

(defn play-game

[player-name]

(loop [current-state (assoc state/board-state :player-name player-name)]

(let [\_ (console/print-current-board current-state (:moves-made current-state))

completion-result (state/handle-game-completion current-state)]

(case (:status completion-result)

:quit (:stats completion-result)

:restart (recur (assoc state/board-state

:stats (:stats completion-result)

:player-name player-name))

:continue (let [move (when (= (:current-player current-state) "X")

(console/obtain-player-input))

turn-result (ai/process-turn current-state move)]

(case (:status turn-result)

:invalid (do

(println "Invalid move! Spot already taken.")

(recur current-state))

:continue (recur (:state turn-result))))))))

Current Implementation: I noticed that I had some repeated code evaluating the legality of moves, and I wanted to pull that out of the play game function. This is now taken care of in the obtain-player-input function. It makes this function even cleaner and more focused on just serving as a switch for the game state..

(defn play-game

[player-name]

(loop [current-state (assoc state/board-state :player-name player-name)]

(let [\_ (console/print-current-board current-state (:moves-made current-state))

completion-result (handle-game-completion current-state)]

(case (:status completion-result)

:quit (:stats completion-result)

:restart (recur (assoc state/board-state

:stats (:stats completion-result)

:player-name player-name))

:continue (let [move (when (= (:current-player current-state) "X")

(console/obtain-player-input current-state))

turn-result (process-turn current-state move)]

(recur (:state turn-result)))))))