Kaylee Lewis

Assignment 06: SOLID... Functionally This Time

Due: Mon Mar 18, 2024 11:59pm

Problem 1:

Consider the following Clojure code taken from a *tic-tac-toe* game:

```
(ns tic-tac-toe.core
  (:require [tic-tac-toe.board :refer :all]
            [tic-tac-toe.ai :refer [ai.move]]
            [tic-tac-toe.game :refer [player-1-marker player2-marker]]
            [tic-tac-toe.human :refer [user-move]]
            [tic-tac-toe.display :refer [display-iteration]]))

(defn game-running [board]
    (if-not (winner? board)
        (let [user-move (user-input-move board board-dimensions)
              current-board (move user-move player1-marker board)]
              (display-iteration current-board)
              (game-runner (move (a-i-move current-board)
                                   player2-marker
                                   current-board))))
    (print-winner board))

(defn -main []
    (let [emptyboard (create-empty-board board-dimensions)]
        (game-runner emptyboard)))
```

Explain how this code violates the Dependency Inversion Principle.

This code violates the Dependency Inversion Principle because the ‘tic-tac-toe.core’ namespace is dependent on functions from low-level modules like the ‘tic-tac-toe.board’ and ‘tic-tac-toe.ai’. Directly calling functions like ‘winner?’ and ‘move’ are examples of implementations provided by other namespaces. The code also doesn’t depend on abstractions. The code is directly referencing concrete implementations like ‘user-move’ and ‘ai.move’ instead of abstracting them away into an interface. This code snippet is also bound into a concrete implementation of other modules. Due to this, its then difficult to extend or change the behavior of the system without modifying the high-level module violating the open/closed principle too.

To more adhere to the Dependency Inversion Principle, the code should be refactored to depend on abstractions rather than concrete implementations. Introducing an interface or protocol to define the behaviors needed can achieve this adherence. Concrete implementations from other modules should implement these abstractions allowing for easier testing.

Problem 2

Here’s another piece of code:

```
trait List[+T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
  def prepend[U >: T](element: U): List[U] = new Cons(element, this)
}

class Cons[T](val head: T, val tail: List[T]) extends List[T] {
  def isEmpty: Boolean = false
}

object Nil extends List[Nothing] {
  def isEmpty: Boolean = true
  def head: Nothing = throw new NoSuchElementException
```

```
        def tail: Nothing = throw new NoSuchElementException
}
```

Remembering that the LSP in functional languages is extended to covariance of types, explain how `prepend` is NOT in violation of the LSP and provide an example of a redefinition of `prepend` that would be a violation of the LSP.

prepend is not in violation of the Liskov Substitution Principle. The return type of the prepend method is a supertype of the type parameter T. The prepend method returns a List[U] where U is a supertype of T. This means that the prepend method can be called on a List[T] and the result can be assigned to a variable of type List[T].

```
class BadCons[T](val head: T, val tail: List[T]) extends List[T]
{
  def isEmpty: Boolean = false

  // Violates LSP: Changes return type to a subtype
  override def prepend[U <: T](element: U): Cons[U] = new Cons(element, this)
}
```

This is an example of a redefinition of prepend that would be a violation of the LSP. The prepend method in the BadCons class changes the return type to a subtype of the type parameter T. This means that the prepend method can be called on a List[T] and the result can be assigned to a variable of type Cons[T]. This violates the Liskov Substitution Principle because the return type of the prepend method is a subtype of the type parameter T. This means that the BadCons class is not a subtype of the List trait.

Problem 3

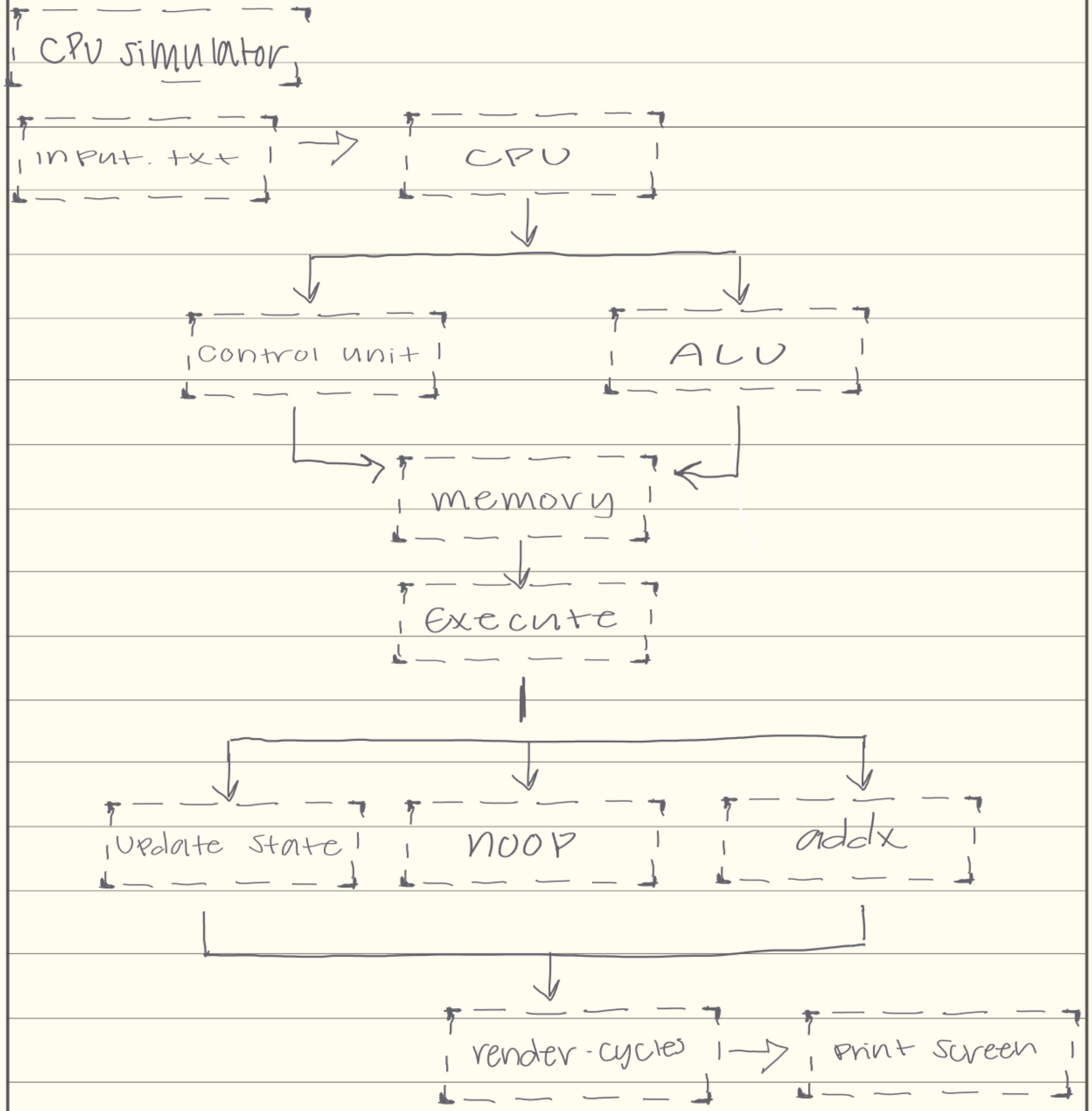
Draw a DFD for the CPU Simulator for the CPU simulator from lecture (and Uncle Bob's book) and then implement the simulator in Scala.

Date: 3/18/2024
Subject: CS 418

Assignment 6 : SOLID... functionally this time

Due : Mon Mar 18, 2024 11:59 PM

Problem 3: Draw a DFD for the CPU simulator



```scala
// Assignment 06: SOLID... Functionally This Time
// Due: Mon Mar 18, 2024 11:59pm
// Problem 3:
// Implement the CPU simulator in Scala.

// https://www.codeconvert.ai/clojure-to-scala-converter

import scala.io.Source

object Problem3 {
    def noop(state: Map[String, Any]): Map[String, Any] = {
        state.updated("cycles", state("cycles").asInstanceOf[Vector[Int]] :+ state("x"))
    }

    def addx(n: Int, state: Map[String, Any]): Map[String, Any] = {
        val x = state("x").asInstanceOf[Int]
        val cycles = state("cycles").asInstanceOf[Vector[Int]]
        state.updated("x", x + n).updated("cycles", cycles ++ Vector(x, x))
    }

    def execute(state: Map[String, Any], lines: List[String]): Map[String, Any] = {
        if (lines.isEmpty) state
        else {
            val line = lines.head
            val nextState: Map[String, Any] = if (line.matches("noop")) noop(state)
                                              else if (line.matches("addx (-?\\d+)")) {
                                                  val n = line.split(" ")(1).toInt
                                                  addx(n, state)
                                              } else Map.empty[String, Any]
            execute(nextState, lines.tail)
        }
    }

    def executeFile(fileName: String): Vector[Int] = {
        val lines = Source.fromFile(fileName).getLines().toList
        val startingState = Map("x" -> 1, "cycles" -> Vector.empty[Int])
        val endingState = execute(startingState, lines)
        endingState("cycles").asInstanceOf[Vector[Int]]
    }

    def renderCycles(cycles: Vector[Int]): List[String] = {
        def loop(cycles: Vector[Int], screen: String, t: Int): List[String] = {
            if (cycles.isEmpty) screen.grouped(40).toList
            else {
                val x = cycles.head
                val offset = t - x
                val pixel = offset >= -1 && offset <= 1
                val newScreen = screen + (if (pixel) "#" else ".")
                val newT = (t + 1) % 40
                loop(cycles.tail, newScreen, newT)
            }
        }
        loop(cycles, "", 0)
    }

    def printScreen(lines: List[String]): Unit = {
        lines.foreach(println)
    }

    def main(args: Array[String]): Unit = {
        val fileName = "input.txt"
        val cycles = executeFile(fileName)
        val screen = renderCycles(cycles)
        printScreen(screen)
    }
```