Evidencia #3: Implementación de un simulador de una máquina expendedora (segunda parte)

Diseño de las estructuras de datos

Mi base de datos consta de archivos de texto los cuales se dividen en maquina (inventario y monedas) y transacciones. Existe una de estas listas por máquina. En mi caso estaremos trabajando con 20 máquinas donde simulamos una empresa alrededor de México. Estos archivos tienen datos que nunca serán guardados de manera local en el programa, sino que toda mutabilidad se hace directamente sobre escribiendo en ellos. A continuación, una demostración de lo que tiene nuestra primera máquina.

-> machine1.txt

((a1 pepsi 15 40) (a2 sabritas 15 40) (a3 mantecadas 17 30) (b1 bubalu 4 60) (b2 sprite 17 7) (b3 halls 10 70) (c1 emperador 13 55) (c2 arizona 24 0) (c3 adobadas 15 35) (d1 ruffles 13 35) (d2 pinguino 12 30) (d3 gansito 15 14) (e1 nito 17 27) (e2 crujitos 14 40) (e3 milkyway 13 35))((1 200 350) (2 100 300) (5 225 250) (10 100 200) (20 127 150) (50 20 100))

-> transaction1.txt

(((a1)(1 2 5 1 20))((a2)(10))((d3)(10 50 1 2 5))((gg)(1 1 2 5))((b1)(2 1 1))((a3)(20))((c1)(10 1 2))((e1)(5 5))((d2)(5 5 20))((e2)(1 1 2 2 1 1 10))((b3)(5 2 2 2))((d1)(1))((d1)(20))((a1)(10 5))((e3)(10 50 5))((a1)(5 1 2 10 5)))

Para hacer estas 20 máquinas opte por hacerlas semi automáticas, donde tengo una función que me crea estas listas y luego simplemente las creo dentro de mi computadora. A continuación, una demostración.

(spit "machine10.txt" (prn-str (list (list 'a1 'pepsi '15 (rand-int 50))(list 'a2 'sabritas 15 (rand-int 50))(list 'a3 'mantecadas 17 (rand-int 50))(list 'b1 'bubalu 4 (rand-int 50))(list 'b2 'sprite 17 (rand-int 50))(list 'b3 'halls 10 (rand-int 50))(list 'c1 'emperador 13 (rand-int 50))(list 'c2 'arizona 24 (rand-int 50))(list 'c3 'adobadas 15 (rand-int 50))(list 'd1 'ruffles 13 (rand-int 50))(list 'd2 'pinguino 12 (rand-int 50))(list 'd3 'gansito 15 (rand-int 50))(list 'e1 'nito 17 (rand-int 50))(list 'e2 'crujitos 14 (rand-int 50))(list 'e3 'milkyway 13 (rand-int 50)))))

(spit "machine10.txt" (pr-str (list (list 1 (rand-int-min-max 80 250) 350)(list 2 (rand-int-min-max 60 230) 300)(list 5 (rand-int-min-max 40 200) 250)(list 10 (rand-int-min-max 30 160) 200)(list 20 (rand-int-min-max 20 120) 150)(list 50 (rand-int-min-max 10 80) 100))) :append true)

Esto genera:

((a1 pepsi 15 24) (a2 sabritas 15 34) (a3 mantecadas 17 44) (b1 bubalu 4 26) (b2 sprite 17 7) (b3 halls 10 12) (c1 emperador 13 37) (c2 arizona 24 23) (c3 adobadas 15 31) (d1 ruffles 13 10) (d2 pinguino 12 20) (d3 gansito 15 2) (e1 nito 17 38) (e2 crujitos 14 31) (e3 milkyway 13 39))

((1 237 350) (2 280 300) (5 167 250) (10 178 200) (20 100 150) (50 14 100))

Diseño del autómata

Para el diseño del autómata opte por usar el recomendado en clase. Esta toma como estado actual la suma del estado anterior y el símbolo de transición. De esta manera el autómata puede ser autogenerado.

-> Diseño original visto en clase

Implementación del autómata

Lo uso en mi código principalmente para calcular el cambio que recibe una compra ya que itera por las monedas dadas y las va sumando en un estado que después compara contra el precio del producto y determina el cambio que recibe. La función give-change es una adaptación de revisa-estados, una función vista previamente en avances. Esta usa una transición para ir iterando, haciendo así uso del autómata. Regresa un valor numérico, ya que es un estado final. Posteriormente en el código este valor numérico se convertirá en una lista de monedas que regresaremos.

-> Implementación en el código

```
118   ; automata
119   (defn transition [state coin]
120     (+ state coin))
121
122   ;originalmente funcion revisa-estados, para el automata
123   (defn give-change [transaction-coins actual-state product-price]
124     (if (empty? transaction-coins)
125       (cond (> actual-state product-price) (- actual-state product-price)
126             :else 0)
127       (give-change (rest transaction-coins) (transition actual-state (first transaction-coins)) product-price)))
```

Mutabilidad de datos

Mi programa maneja los datos todos de manera remota, es decir solo se modifica archivos de texto externos a mi código, respetando totalmente el paradigma funcional y la visión de Clojure al no tener mutabilidad dentro de mi código. En mi experiencia, aunque es una manera muy limpia de mantener el código, puede en momentos ser un poco ineficiente. Hablo de ineficiencia ya que, según lo implementado, mi código tiene que abrir, escribir y cerrar los archivos por cada moneda y cada compra, lo que puede ser costoso en el tiempo de ejecución.

Para leer mis archivos uso las siguientes funciones

```
;lee y regresa la lista del investario y monedas. Se guarda en un arreglo las dos strs de mis lineas de maquina
(defn read-machine-file-by-lines [file-to-read]
  (with-open [rdr (clojure.java.io/reader (clojure.java.io/resource file-to-read))]
    (reduce conj [] (line-seq rdr))))

;funcion auxiliar que convierte de srt a lista
(defn get-machine-info [machine-file]
  (map read-string machine-file))


(defn get-products [machine-file]
  (first (get-machine-info (read-machine-file-by-lines machine-file))))


(defn get-coins [machine-file]
  (second (get-machine-info (read-machine-file-by-lines machine-file))))
```

Para hacer la mutabilidad de mis datos uso las siguientes funciones

```
;Función para sobreescribir una lista en la base de datos, modificando los productos
(defn persist-products [new-products old-coins machine]
  (spit (clojure.java.io/resource machine) (prn-str new-products))
  (spit (clojure.java.io/resource machine) (pr-str old-coins) :append true))

;Función para sobreescribir una lista en la base de datos, modificando las monedas
(defn persist-coins [new-coins old-products machine]
  (spit (clojure.java.io/resource machine) (prn-str old-products))
  (spit (clojure.java.io/resource machine) (pr-str new-coins) :append true))
```

Como mencionado anteriormente y evidenciado en los pantallazos, se manejan mis archivos de forma independiente al código, por lo que un cambio en la moneda o el inventario no debería de gestionar un cambio en mi código, sino solo en mis archivos de texto.

Paralelización

La paralelización la uso una sola vez en mi código. La uso para recorrer todas mis maquinas al mismo tiempo con la función pmap, acompañada de doall.

```
199   ;aqui probamos la paralelización, pretendiendo correr las transacciones de todas las maquinas al mismo tiempo
200   (time (doall (pmap open-and-read-transactions files)))
```

Este proceso se pudiera implementar de manera recursiva o incluso con un simple map, pero como se puede ver en el video, el uso del pmap particularmente en este código mejora de manera significativa la rapidez de nuestro código.

Resultado de tiempo con map: 17997 msgs

Resultado de tiempo con pmap: 5309 msgs

Speedup: 12688 msgs = 13segundos

Lo considero un resultado muy significante tomando en cuenta que solamente son 20 máquinas. Me puedo empezar a imaginar la efectividad en cuanto escalan la cantidad de máquinas y transacciones por máquina. En el caso particular de mi implementación del pmap, se puede considerar un *éxito*.

Ventajas y desventajas de Clojure

Ventajas

1. La principal ventaja y la razón de esta evidencia es claramente la combinación del paradigma funcional con la programación en paralelo.
2. Multiparadigma.
3. Puede emplear librerías de Java, lenguaje popular.
4. De los lenguajes funcionales más utilizados en el ambiente laboral y académico.
5. Código elegante, limpio y fácil de documentar. (depende del programador, pero en general se presta para implementaciones elegantes)

Desventajas

1. No tiene mucha información o comunidad en internet para apoyarte.
2. Inmutabilidad de datos puede dar ineficiencias.
3. La instalación y ajustar tu IDE para poder programar.
4. Es un lenguaje que sin previo conocimiento de Raket o algún programa similar, puede ser difícil de aprender.

5.

Código

```clojure
(ns primer-proyecto.core
  (:gen-class)
  (:require [clojure.java.io :as reader]))

;lee y regresa la lista del investario y monedas. Se guarda en un arreglo las dos strs de mis lineas de maquina
(defn read-machine-file-by-lines [file-to-read]
  (with-open [rdr (clojure.java.io/reader (clojure.java.io/resource file-to-read))]
    (reduce conj [] (line-seq rdr))))

;funcion auxiliar que convierte de srt a lista
(defn get-machine-info [machine-file]
  (map read-string machine-file))


(defn get-products [machine-file]
  (first (get-machine-info (read-machine-file-by-lines machine-file))))


(defn get-coins [machine-file]
  (second (get-machine-info (read-machine-file-by-lines machine-file))))

;Función para sobreescribir una lista en la base de datos, modificando los productos
(defn persist-products [new-products old-coins machine]
  (spit (clojure.java.io/resource machine) (prn-str new-products))
  (spit (clojure.java.io/resource machine) (pr-str old-coins) :append true))

;Función para sobreescribir una lista en la base de datos, modificando las monedas
(defn persist-coins [new-coins old-products machine]
  (spit (clojure.java.io/resource machine) (prn-str old-products))
  (spit (clojure.java.io/resource machine) (pr-str new-coins) :append true))

;funcion que regresa un solo producto, dado un id y la lista entera de productos
(defn get-product-by-id [product-id products]
  (cond (empty? products) '()
        :else
        (if (= (ffirst products) product-id) (first products) (get-product-by-id product-id (rest products)))))

;te trae el id del producto que quiere comprar una transaccion
(defn get-transaction-product-id [transaction]
  (ffirst transaction))
```

```clojure

;te regresa el precio de un producto
(defn get-product-price [product]
  (nth product 2))

;te trae la secuencia de monedas insertadas por el cliente en una transaccion
(defn get-transaction-coins [transaction]
  (fnext transaction))

; invierte la lista de monedas
(defn get-reversed-coins [machine]
  (reverse (get-coins machine)))

;te regresa una moneda en especifico dado el tipo de moneda y todas las monedas
(defn get-coin-by-coin-type [coin-type coins]
  (cond (nil? coins) '()
        :else
        (if (= (ffirst coins) coin-type) (first coins) (get-coin-by-coin-type coin-type (rest coins)))))

;función auxiliar para reemplazar un valor en una lista por la posicion dada en posicion dada
(defn replace-nth [list n item]
  (if (= n 0)
    (cons item (rest list))
    (cons (first list) (replace-nth (rest list) (- n 1) item))))

;te regresa la lista de productos actualizada por inventario, esta lista será escrita en nuestra base de datos
(defn update-product-stock-and-return-all-products [products product-id new-stock]
  (map (fn [x] (if (= (first x) product-id) (replace-nth x 3 new-stock) x)) products))

;te regresa la lista de monedas actualizada por cantidad, esta lista será escrita en nuestra base de datos
(defn update-coin-inventory-and-return-all-coins [coins coin-type new-inventory]
  (map (fn [x] (if (= (first x) coin-type) (replace-nth x 1 new-inventory) x)) coins))

;primera condicion. Valida si hay inventario de un producto
(defn check-product-inventory [product-id list-of-products]
  (cond (empty? list-of-products) false
      :else
       (if (and (= (ffirst list-of-products) product-id) (>= (last (first list-of-products))1)) true
          (check-product-inventory product-id (rest list-of-products)))))
```

```clojure

;segunda condicion. Valida si se insertaron suficientes monedas como para hacer la compra.
(defn enough-money-to-purchase? [product-id products transaction-coins]
  (cond (empty? products) false
      :else
       (if (and (= (ffirst products) product-id) (<= (get-product-price (get-product-by-id product-id products)) (apply + transaction-coins))) true
          (enough-money-to-purchase? product-id (rest products) transaction-coins))))

; aumentamos el numero de monedas en nuestro inventorio por cada compra
(defn increase-coin-inventory [transaction-coins machine]
  (if (empty? transaction-coins) '()
       [(persist-coins (update-coin-inventory-and-return-all-coins (get-coins machine) (first transaction-coins)
                                                                   (+ (fnext (get-coin-by-coin-type (first transaction-coins) (get-coins machine))) 1))
                       (get-products machine) machine)
        (increase-coin-inventory (rest transaction-coins) machine)]))

; A partir de aqui se usa funciones para reducir el inventario de monedas por el cambio que se le da al cliente
(defn decrease-coin-inventory [change-given machine]
  (if (empty? change-given) '()
       [(persist-coins (update-coin-inventory-and-return-all-coins (get-coins machine) (first change-given)
                                                                   (- (fnext (get-coin-by-coin-type (first change-given) (get-coins machine))) 1))
                       (get-products machine) machine)
        (decrease-coin-inventory (rest change-given) machine)]))

; Esta funcion reduce el inventario del producto comprado
(defn diminish-product-stock [product-id machine]
  (persist-products (update-product-stock-and-return-all-products (get-products machine) product-id
                                                                  (- (last (get-product-by-id product-id (get-products machine))) 1))
                    (get-coins machine) machine))
```

```clojure

 ;funcion auxiliar para cambiar de numero a lista de monedas en el cambio
 (defn insert-n [list item n]
   (if (= n 0)
     (cons item list)
     (cons (first list) (insert-n (rest list) item (- n 1)))))

 ;en esta funcion recibimos el numero de cambio como entero y lo convertimos a una lista de monedas, las cuales seran restadas de nuestro inven
 (defn return-change-in-coins [change coins-in-reverse change-given-so-far]
   (cond (<= change 0) change-given-so-far
         (>= (quot change (ffirst coins-in-reverse)) 1)
             (cond (>= (second (first coins-in-reverse)) 1)
                 (return-change-in-coins (- change (ffirst coins-in-reverse)) coins-in-reverse
                                         (insert-n change-given-so-far (ffirst coins-in-reverse) (count change-given-so-far))))
         :else (return-change-in-coins change (rest coins-in-reverse) change-given-so-far)))

 ; automata
 (defn transition [state coin]
   (+ state coin))

 ;originalmente funcion revisa-estados, para el automata
 (defn give-change [transaction-coins actual-state product-price]
   (if (empty? transaction-coins)
     (cond (> actual-state product-price) (- actual-state product-price)
           :else 0)
     (give-change (rest transaction-coins) (transition actual-state (first transaction-coins)) product-price)))
```

```clojure
 ;Se empiezan a hacer los cambios en la base de datos, para este momento la compra ya paso las condiciones.
 (defn perform-changes-in-inventories [transaction machine]
   (println "Tu compra fue aceptada")
   (diminish-product-stock (get-transaction-product-id transaction) machine)
   (decrease-coin-inventory (return-change-in-coins
                             (give-change (get-transaction-coins transaction) 0
                                          (get-product-price (get-product-by-id
                                                              (get-transaction-product-id transaction) (get-products machine))))
                                          (get-reversed-coins machine)
                                          '()) machine)
   (increase-coin-inventory (get-transaction-coins transaction) machine))

 ;Funciones de reporte

 ; regresa la mutiplicacion de la moneda por el numero en inventario y lo suma
 (defn sum-of-coins [machine]
   (apply + (map (fn [x] (* (first x) (fnext x))) (get-coins machine))))

 ;con esta funcion se pretende revisar que producto tiene un inventario de menos de 5, en ese caso, se imprime la lista con los nombres de los productos
 (defn stock-of-product-low [machine]
   (map second (filter (fn [x] (<= (last x) 5)) (get-products machine))))

 ;funcion que regresa las monedas casi llenas en el inventario, el filtro es 20 monedas
 (defn coins-with-inventory-almost-full [coins]
   (cond (empty? coins) coins
         :else
         (map first (filter (fn [c] (<= (- (last c) (second c)) 30)) coins))))
```

```clojure
;funcion que regresa monedas con un inventario menor a 30
(defn coins-with-inventory-almost-empty [coins]
  (cond (empty? coins) coins
        :else
        (map first (filter (fn [x] (<= (fnext x) 30)) coins))))

;agarra una transaccion y valida condiciones en ella. Si las cumple, se efectuan cambios en la base de datos, pues se hace la compra
(defn perform-purchase-of-transaction [transaction machine]
  (if (check-product-inventory (get-transaction-product-id transaction) (get-products machine))
    (if (enough-money-to-purchase? (get-transaction-product-id transaction) (get-products machine) (get-transaction-coins transaction))
      (perform-changes-in-inventories transaction machine)
    (println "Te falta dinero, no fue aceptada la transaccion"))
  (println "No existe tu producto o se acabo el inventorio, lo sentimos")))

(defn machine-report [machine]
  (println "---Se terminaron las transaciones--- \n")
  (print "1. Las ganancias netas de hoy fueron ") (println (sum-of-coins machine))
  (print "2. Los productos con poco inventario son ") (println (stock-of-product-low machine))
  (print "3. Las monedas con el inventario casi lleno son: ") (println (coins-with-inventory-almost-full (get-coins machine)))
  (print "4. Las monedas con inventario menor a 30 son ") (println (coins-with-inventory-almost-empty (get-coins machine)))
  (println "GRACIAS POR USAR LA MAQUINA, NOS VEMOS!")
  (println "\n"))

; Con esta funcion leemos todas las trnsacciones y empezamos a iterar por cada una de ellas
(defn iterate-on-transactions [transactions machine]
  (if (empty? transactions) (machine-report machine)
      [(perform-purchase-of-transaction (first transactions) machine)
       (iterate-on-transactions (rest transactions) machine)]))

; se declara globalmente el archivo de las trasancciones, se abre, lee y cierra. Debe de recibir un str
(defn open-and-read-transactions [transaction-machine-files]
  (def transactions (read-string (slurp (clojure.java.io/resource (first transaction-machine-files)))))
  (iterate-on-transactions transactions (second transaction-machine-files)))
; ------------------------------------------------------------------------------------------------------------------------
```

```clojure
(def files '(("transaction1.txt" "machine1.txt")("transaction2.txt" "machine2.txt")("transaction3.txt" "machine3.txt")
             ("transaction4.txt" "machine4.txt")("transaction5.txt" "machine5.txt")("transaction6.txt" "machine6.txt")
             ("transaction7.txt" "machine7.txt")("transaction8.txt" "machine8.txt")("transaction9.txt" "machine9.txt")
             ("transaction10.txt" "machine10.txt")("transaction11.txt" "machine11.txt")("transaction12.txt" "machine12.txt")
             ("transaction13.txt" "machine13.txt") ("transaction14.txt" "machine14.txt")
             ("transaction15.txt" "machine15.txt") ("transaction16.txt" "machine16.txt")
             ("transaction17.txt" "machine17.txt") ("transaction18.txt" "machine18.txt")
             ("transaction19.txt" "machine19.txt") ("transaction20.txt" "machine20.txt")))

;aqui probamos la paralelización, pretendiendo correr las transacciones de todas las maquinas al mismo tiempo
(time (doall (pmap open-and-read-transactions files)))
```

Pruebas realizadas

Como se realizo en la evidencia pasada, para confirmar que funciona este código estaremos revisando nuestro primer y ahora también ultimo archivo y viendo si hubo cambios deseados en ellos.

-> Machine1.txt

((a1 pepsi 15 37) (a2 sabritas 15 40) (a3 mantecadas 17 29) (b1 bubalu 4 59) (b2 sprite 17 7) (b3 halls 10 69) (c1 emperador 13 54) (c2 arizona 24 0) (c3 adobadas 15 35) (d1 ruffles 13 34) (d2 pinguino 12 29) (d3 gansito 15 13) (e1 nito 17 27) (e2 crujitos 14 39) (e3 milkyway 13 34))

((1 206 350) (2 100 300) (5 231 250) (10 104 200) (20 131 150) (50 20 100))

-> transaction1.txt

(((a1)(1 2 5 1 20))((a2)(10))((d3)(10 50 1 2 5))((gg)(1 1 2 5))((b1)(2 1 1))((a3)(20))((c1)(10 1 2))((e1)(5 5))((d2)(5 5 20))((e2)(1 1 2 2 1 1 10))((b3)(5 2 2 2))((d1)(1))((d1)(20))((a1)(10 5))((e3)(10 50 5))((a1)(5 1 2 10 5)))

-> machine20.txt

((a1 pepsi 15 8) (a2 sabritas 15 8) (a3 mantecadas 17 1) (b1 bubalu 4 5) (b2 sprite 17 12) (b3 halls 10 39) (c1 emperador 13 28) (c2 arizona 24 16) (c3 adobadas 15 4) (d1 ruffles 13 34) (d2 pinguino 12 1) (d3 gansito 15 20) (e1 nito 17 33) (e2 crujitos 14 15) (e3 milkyway 13 45))

((1 241 350) (2 202 300) (5 93 250) (10 82 200) (20 117 150) (50 89 100))

-> transaction20.txt

(((a3) (50 10))((e1) (1 2 5 2 10))((d2) (50 2 2))((d3) (10 10 50 20 20 20))((b3) (20 20 20))((a2) (1 2 1 1 50))((b2) (20 10 5))((e2) (5 5 20 5))((a1) (5 5 5 10 10))((a1) (2 10 5 5 1 1))((a1) (5)))

Resultados después de correr el programa una vez

-> machine1.txt

((a1 pepsi 15 37) (a2 sabritas 15 40) (a3 mantecadas 17 29) (b1 bubalu 4 59) (b2 sprite 17 7) (b3 halls 10 69) (c1 emperador 13 54) (c2 arizona 24 0) (c3 adobadas 15 35) (d1 ruffles 13 34) (d2 pinguino 12 29) (d3 gansito 15 13) (e1 nito 17 27) (e2 crujitos 14 39) (e3 milkyway 13 34))

((1 206 350) (2 100 300) (5 231 250) (10 104 200) (20 131 150) (50 20 100))

-> machine20.txt

((a1 pepsi 15 6) (a2 sabritas 15 7) (a3 mantecadas 17 0) (b1 bubalu 4 5) (b2 sprite 17 11) (b3 halls 10 38) (c1 emperador 13 28) (c2 arizona 24 16) (c3 adobadas 15 4) (d1 ruffles 13 34) (d2 pinguino 12 0) (d3 gansito 15 19) (e1 nito 17 32) (e2 crujitos 14 14) (e3 milkyway 13 45))

((1 243 350) (2 202 300) (5 100 250) (10 88 200) (20 117 150) (50 90 100))

1. En primer lugar, podemos ver cómo tanto en la primera como en la segunda se ven números diferentes en el dato correspondiente al inventario (cuarto elemento dentro de la lista). Si ponemos en exposición a pepsi, producto que se compra en ambas maquinas podemos ver que el numero tuvo un decremento.
2. Podemos ver como también las monedas han cambiado en ambos archivos, simulando el incremento en caso de recibir y decremento en caso de regresas cambio.
3. Si vemos en la lista de la última máquina, el elemento b1 se mantiene exactamente igual, esto se debe a que nunca se compró este producto.
4. También podemos ver el elemento de mantecadas en la ultima lista la cual se mantiene en ceros, lo que nos hace saber que no se estará vendiendo productos en caso de no haber en existencia, así evitando siempre los números negativos en inventario, ya que sería un caso imposible.
5. En general, saber que se esta generando estos cambios tanto en la primera como en la ultima lista nos habla de un funcionamiento idéntico en todas las maquinas en existencia, por lo que las pruebas realizas en el avance de la situación problema anterior siguen vigentes para esta.
6. Como se pudo mostrar en el video, también probamos la paralelización al ver el desastre en el que se convirtió nuestra terminal, lo cual es buen signo de que múltiples cosas se están ejecutando al mismo tiempo.
7. **Disclaimer**: En esta evidencia no se da reporte final de todas las maquinas, como esta indicado en las instrucciones. Se genera en su lugar un reporte individual de cada máquina, tal y como en la evidencia pasada.

Enlace al video: https://drive.google.com/file/d/1zJKLFHjo_v_C3ahmuQdoqlM0_Ts2AVnQ/view?usp=sharing

Evidencia de aprendizaje

Para esta evidencia decidí cargar con los aprendizajes de mi evidencia pasada y trabajar de manera limpia y organizada con mi código. En cuanto a habilidades personales, la organización, responsabilidad y superamiento personal llegaron a su punto más satisfactorio en esta evidencia. En cuanto a lo técnico, me considero un programador mucho más completo después de completar esta evidencia. Reforcé de manera recurrente mi pensamiento recursivo, así como la programación en paralelo y funcional. Tengo el criterio para saber cuando se debe o no paralelizar un proceso y de las ventajas y desventajas de las herramientas que estoy usando. Aprendí a generar código fácil de leer para todos, de manera que los comentarios resultan ser incluso apoyo extra. Como es la última evidencia de esta materia no me puedo despedir sin decir que definitivamente cosechamos lo que sembramos. Hubo muchos momentos en los que la respuesta ni era inmediata y podía caer en la desesperación, pero eso solo me enseño resiliencia y fuerza de voluntad. Después de recorrer toda esta materia no puedo más que recordar una frase dicha por Samwise Gamyi en El señor de los anillos: “ But in the end, it's only a passing thing, this shadow. Even darkness must pass. A new day will come. And when the sun shines it will shine out the clearer. Those were the stories that stayed with you. That meant something, even if you were too small to understand why. But I think, Mr. Frodo, I do understand. I know now. **Folk in those stories had lots of chances of turning back, only they didn't.** They kept going. Because they were holding on to something.”