

Metodologías de Programación I

Práctica 2.

Patrones Strategy e Iterator

Ejercicio 1

Implemente cuatro estrategias de comparación para la clase *Alumno* definida en la práctica anterior. Realice una estrategia para que compare alumnos por nombre, otra para que compare por DNI, otra para que compare por promedio y una última estrategia que compare por legajo.

Ejercicio 2

Modifique el ejercicio 16 de la clase anterior para crear alumnos con alguna estrategia implementada en el ejercicio anterior.

Use el método *main* del ejercicio 17 de la clase anterior para comprobar el funcionamiento correcto de la estrategia seleccionada. Note que este método *main* NO debería ser modificado.

Ejercicio 3

Implemente la clase *Conjunto*. Un *Conjunto* es una colección que almacena elementos sin repetición. Es decir, si se intenta almacenar un elemento que ya está en el conjunto, éste elemento no se almacena ya que sino estaría repetido.

Esta clase debe contar con los métodos:

| | |
|----------------------------------|--|
| <code>agregar(elemento)</code> | ← el cual agrega el elemento al conjunto si es que éste no existe |
| <code>pertenece(elemento)</code> | ← el cual devuelve true si el elemento recibido por parámetro ya está dentro del conjunto, o false en caso contrario |

Ejercicio 4

Implemente la clase *Diccionario*. Un *Diccionario* es una colección que almacena elementos, donde cada elemento tiene una clave asociada. Las claves no pueden repetirse.

Esta clase debe contar con los métodos:

| | |
|------------------------------------|---|
| <code>agregar(clave, valor)</code> | ← el cual agrega la asociación clave-valor al diccionario si es que la clave no existe en el diccionario. Si la clave existe, entonces reemplaza el valor asociado. |
|------------------------------------|---|

```
valorDe(clave)          ← devuelve el valor asociada a la
                           clave recibida por parámetro o null
                           si la clave no existe.
```

Sugerencia: implemente una clase auxiliar *ClaveValor* que lo único que hace es almacenar una clave (que puede ser cualquier *Comparable*) y un valor asociado (que puede ser cualquier otro objeto). Luego puede usar la clase *Conjunto* para que un diccionario almacene instancias de la clase *ClaveValor*, las cuales son creadas en el método *agregar* y *valorDe*.

Ejercicio 5

Haga que las clases *Conjunto* y *Diccionario* implementen la interface *Coleccionable*.

En el caso de *Diccionario*, los métodos *minimo*, *máximo* y *contiene* deben hacer referencia a los valores asociados, no a las claves. En el método *agregar* puede usar una clave única por defecto, que maneja el propio diccionario para ir agregando los valores asociados a esas claves únicas.

Ejercicio 6

Haga que las clases *Pila*, *Cola*, *Conjunto* y *Diccionario* implementen la interface *Iterable* vista en teoría.

Ejercicio 7

Implemente una función *imprimirElementos* que reciba un coleccionable y usando el iterador del coleccionable imprima todos los elementos del coleccionable

```
imprimirElementos (coleccionable)
    para todos los elementos elem del coleccionable
        imprimir(elem)
```

Sugerencia: puede re-implementar el método *ToString* (en C#) o *__str__* (en Python) para imprimir los elementos del coleccionable.

Ejercicio 8

Implemente un módulo *main* para crear una pila, una cola, un conjunto y un diccionario de alumnos y luego invoque la función *imprimirElementos* para cada coleccionable.

```
main
    pila = new Pila()
    cola = new Cola()
    conjunto = new Conjunto()
    diccionario = new Diccionario()
    llenarAlumnos(pila)
    llenarAlumnos(cola)
    llenarAlumnos(conjunto)
    llenarAlumnos(diccionario)
    imprimirElementos(pila)
    imprimirElementos(cola)
    imprimirElementos(conjunto)
    imprimirElementos(diccionario)
```

Ejercicio 9

Implemente una función *cambiarEstrategia* que reciba un coleccionable y una estrategia de comparación (implementada en el ejercicio 1) y asigne esa estrategia a todos los elementos del coleccionable

```
cambiarEstrategia (coleccionable, estrategia)
    para todos los elementos elem del coleccionable
        elem.setEstrategia(elem)
```

Nota: en C# deberá hacer un casting de *elem* a *Alumno* para poder cambiar la estrategia.

Ejercicio 10

Modifique el módulo *main* para que cambie la estrategia de comparación a los elementos de un coleccionable e informe mínimo y máximo elemento:

```
main
    pila = new Pila()      ← ó Cola ó Conjunto ó Diccionario
    llenarAlumnos(pila)
    cambiarEstrategia(pila, Estrategia de comparación por
                        nombre)
    informar(pila)
    cambiarEstrategia(pila, Estrategia de comparación por
                        legajo)
    informar(pila)
    cambiarEstrategia(pila, Estrategia de comparación por
                        promedio)
    informar(pila)
    cambiarEstrategia(pila, Estrategia de comparación por dni)
    informar(pila)
```

Este ejercicio, y todos los anteriores que dependen de éste, debe ser entregado en el aula virtual del campus.

Ejercicio 11

Para reflexionar. Si se implementara una nueva clase *AlumnoEgresado*, subclase de *Alumno*, que agrega como estado el atributo *fecha_de_egreso*. Además de agregar la clase *AlumnoEgresado* ¿qué otras modificaciones hay que hacer a todo lo desarrollado hasta acá.

Ejercicio 12

Para reflexionar. Si se posee un alumno con promedio 7.00 como “alumno promedio” y se desea usar para compararlo con alumnos que están en un conjunto (donde algunos tienen mejor promedio y otro un promedio más bajo), además se desea usar la misma función del ejercicio 7 sin modificarla en absoluto. ¿Qué cambios se deben hacer al sistema desarrollado hasta ahora para imprimir solo los alumnos que tengan mejor promedio que mi “alumno promedio”?

Ejercicio 13

Opcional. Implemente el iterador pensado en el ejercicio 12. Piense en que el iterador pueda comparar el “alumno promedio” con los criterios promedio, nombre, legajo o dni, donde en los cuatro casos solo se deben recorrer los alumnos “mayores” al alumno promedio con el criterio elegido.

Sugerencia: Utilice las estrategias implementadas en el ejercicio 1.

Ejercicio 14

Opcional. Desarrolle una función *multiplesIteradores* que permita que tres “clientes” iteren sobre la misma colección.

```
multiplesIteradores()
    pila = new Pila()
    llenarAlumnos(pila)
    iteradores = new Iterator[3]
    fin = new boolean[3]
    for i in (1 to 3)
        iteradores[i] = pila.crearIterador()
        fin[i] = false
    while (not fin[1]) and (not fin[2]) and (not fin[3])
        ite = valor_random_entre1_y_3
        if(not iteradores[ite].fin)
            imprimir(iteradores[ite].actual())
            iteradores[ite].siguiente()
        else
            fin[ite] = true
```

Para reflexionar. ¿Podría haber hecho esto mismo sin iteradores? ¿A qué costo? ¿Qué cambia de este algoritmo si en vez de una pila se usa el diccionario implementado en esta actividad.

Ejercicio 15

Opcional. Intercambie las clases que implementan las estrategias y coleccionables con otro compañero para probar si funcionan clases “externas” en el sistema desarrollado por uno mismo.