Metodologías de Programación I

Práctica 5.

Patrones Proxy y Command

Ejercicio 1

Implemente con el patrón Proxy un proxy para cualquier *Alumno*. Este proxy debería actuar como tal para almacenar y devolver todos los datos de un *Alumno*. El proxy debe crear al alumno real (*Alumno* o *AlumnoMuyEstudioso*) al momento de responder una pregunta (método *responderPregunta*).

Ejercicio 2

Ejecute la función *main* del ejercicio 8 de la práctica 4 para comprobar el correcto funcionamiento.

Nota: puede imprimir un mensaje por consola al momento de crear la instancia de *Alumno* o *AlumnoMuyEstudioso* dentro del proxy, para corroborar el momento en que se generan las instancias (Cuando el *Teacher* toma el examen).

Ejercicio 3

Implemente la clase *Aula*:

```
Aula
      teacher                ← Es una variable que almacena un
                               Teacher
      comenzar()             ← Imprime un mensaje por consola e
                               instancia la variable teacher (new
                               Teacher)
      nuevoAlumno(Alumno)    ← Agrega al teacher el alumno recibido
                               como parámetro usando el método
                               goToClass.
      claseLista()           ← Envía el mensaje teachingAClass al
                               teacher.
```

Ejercicio 4

Implemente la interface *OrdenEnAula1:*

```
OrdenEnAula1
      ejecutar()
```

Ejercicio 5

Usando el patrón Command implemente dos órdenes con la interface *OrdenEnAula1*, donde las órdenes trabajaran con un *Aula* como receptor:

- *OrdenInicio*: envía el mensaje *comenzar* al aula.
- *OrdenAulaLlena*: envía el mensaje *claseLista* al aula.

Ejercicio 6

Implemente la interface *OrdenEnAula2:*

```
OrdenEnAula2
      ejecutar(Comparable)
```

Ejercicio 7

Usando el patrón Command implemente una orden con la interface *OrdenEnAula2*, donde la orden trabajará con un *Aula* como receptor:

- *OrdenLlegaAlumno*: envía el mensaje *nuevoAlumno* al aula.

Ejercicio 8

Implemente la interface *Ordenable:*

```
Ordenable
      setOrdenInicio(OrdenEnAula1)
      setOrdenLlegaAlumno(OrdenEnAula2)
      setOrdenAulaLlena(OrdenEnAula1)
```

Ejercicio 9

Modifique las clases *Pila*, *Cola*, *Conjunto* y *Diccionario* para que implementen la interface *Ordenable* y puedan trabajar con las tres órdenes implementadas en los ejercicios anteriores.

- Invoque a la orden *OrdenInicio* cuando la colección recibe el primer elemento
- Invoque a la orden *OrdenLlegaAlumno* cuando se recibe un elemento, enviando el elemento recibido por parámetro, incluso si es el primero que se recibe, y en este caso se invoca a esta orden luego de haber invocado la *OrdenInicio*.
- Invoque a la orden *OrdenAulaLlena* cuando la colección recibe el elemento número 40 y después de haber invocado la *OrdenLlegaAlumno*.

Ejercicio 10

Implemente una función *main* que instancie un *Aula* y una instancia de cada una de las tres órdenes con el aula como receptor. Instancie una *Pila*/*Cola*/*Conjunto*/*Diccionario* y seteele las tres órdenes. Luego llame a la función *llenar* (práctica 3, ejercicio 6) para agregar 20 instancias de *Alumno* y 20 instancias de *AlumnoMuyEstudioso*.

```
main
      pila = new Pila()          ← ó Cola ó Conjunto ó Dicc.
      aula = new Aula()
      pila.setOrdenInicio(new OrdenInicio(aula))
      pila.setOrdenLlegaAlumno(new OrdenLlegaAlumno(aula))
      pila.setOrdenAulaLlena(new OrdenAulaLlena(aula))
      llenar(pila, OPCION_ALUMNO)
      llenar(pila, OPCION_ALUMNO_MUY_ESTUDIOSO)
```

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

Ejercicio 11

Para reflexionar. Luego de haber necesitado adaptar la instanciación de un *Alumno* con Adapter, Decorator y Proxy. ¿Queda en evidencia la necesidad de utilizar fábricas? ¿Qué hubiera pasado si no se hubiera implementado el patrón Factory Method y se poseen muchas funciones donde se necesitan crear instancias de *Alumno*? (piense en todas las funciones que tuvo que implementar con la necesidad de crear instancias de *Alumno*)

Ejercicio 12

Opcional. Implemente un Proxy para *Coleccionables*. La tarea de este proxy debe ser de optimización.

- El proxy debe crear al Coleccionable real cuando se agrega un elemento (método *agregar*). Hasta tanto, los métodos:
  - *cuantos*: devuelve cero.
  - *minimo*: devuelve null.
  - *maximo*: devuelve null.
  - *contiene*: devuelve false
- Una vez creado el sujeto real se redireccionan los métodos *cuantos*, *agregar* y *contiene*.
- Para el caso de los métodos *minimo* y *maximo*, en la primera invocación, luego de creado el sujeto real, se le pide a éste el mínimo y el máximo. El proxy deberá guardar el *Comparable* mínimo y máximo devuelto por el *Coleccionable* real y éstos deberán ser devueltos en las siguientes invocaciones de los métodos *mínimo* y *máximo* del proxy. El proxy le pedirá al *Coleccionable* real nuevamente el mínimo o el máximo cuando el Coleccionable "cambie", es decir, se agrega un nuevo elemento a la colección.

Ejemplo:

```
coleccionable = new PilaProxy() ← ó ColaProxy ó ConjProxy
                                      ó DiccProxy
print(coleccionable.cuantos())   ← Imprime cero
print(coleccionable.minimo())    ← Imprime null
print(coleccionable.maximo())    ← Imprime null

coleccionable.agregar(new Numero(5))  ← se crea la Pila
                                        real
coleccionable.agregar(new Numero(3))
coleccionable.agregar(new Numero(8))
print(coleccionable.cuantos())   ← Imprime 3
```

```
print(coleccionable.minimo())    ← Imprime 3. Se le pide el
                                 minimo a la Pila. El proxy
                                      "cachea" el Numero 3
                                      como mínimo.
print(coleccionable.maximo())    ← Imprime 8. Se le pide el
                                 maximo a la Pila. El proxy
                                      "cachea" el Numero 8
                                      como máximo.
print(coleccionable.cuantos())   ← Imprime 3
print(coleccionable.minimo())    ← Imprime 3. El proxy
                              devolvió el valor "cacheado"
print(coleccionable.maximo())    ← Imprime 8. El proxy
                              devolvió el valor "cacheado"

num = new Numero(15)
coleccionable.agregar(num)       ← El coleccionable cambió.
print(coleccionable.minimo())    ← Imprime 3. Se le pide el
                                 minimo a la Pila. El proxy
                                      "cachea" el Numero 3
                                      como mínimo.
print(coleccionable.maximo())    ← Imprime 15. Se le pide
                                 el maximo a la Pila. El
                                 proxy "cachea" el Numero
                                 15 como máximo.
print(coleccionable.minimo())    ← Imprime 3. El proxy
                              devolvió el valor "cacheado"
print(coleccionable.maximo())    ← Imprime 15. El proxy
                              devolvió el valor "cacheado"
```

Ejercicio 13

Para reflexionar: ¿Qué devolvería el proxy del ejercicio anterior si se hiciera: `num.setValor(1)` luego de la última instrucción del programa anterior?

Ejercicio 14

Para reflexionar: ¿Cómo podría un proxy darse cuenta que un elemento del *Coleccionable* cambió? ¿Qué patrón de diseño ya visto podría utilizarse para resolver este problema?

Opcional: Implemente la solución propuesta utilizando la clase *Numero*.

Ejercicio 15

Opcional. Intercambie las clases de proxies y órdenes implementadas en esta práctica con otro compañero para probar si funcionan clases "externas" en el sistema desarrollado por uno mismo.