Acosta Porcayo Alan Omar, Gutiérrez Grimaldo Alejandro, Medina Villa Samuel

Práctica 12. Hilos

Resumen

La práctica se enfoca en la implementación del concepto de multitarea mediante hilos en un entorno orientado a objetos. Se abordarán dos enfoques principales: la creación de hilos con la clase *Thread* y la interfaz *Runnable*. La práctica incluirá ejercicios prácticos para aplicar estos conceptos, evaluación de ventajas y desventajas entre ambos enfoques, y la exploración de la ejecución concurrente de tareas, haciendo hincapié en la coordinación efectiva de hilos para mejorar la eficiencia en programas concurrentes.

Introducción

Imaginemos el desafío de desarrollar una aplicación departamental encargada de realizar diversas operaciones complejas. Entre estas funciones críticas se encuentran la descarga del catálogo de precios de productos recién incorporados al inventario, la realización de la contabilidad correspondiente al día anterior y la aplicación de descuentos a productos ya existentes en el sistema.

Inconvenientes de la Ejecución Secuencial

En un flujo de ejecución convencional, estas tareas se llevarían a cabo de manera secuencial, una después de la otra. No obstante, la ejecución secuencial plantea inconvenientes, especialmente cuando la descarga de productos nuevos consume demasiado tiempo, afectando la aplicación oportuna de descuentos. En este escenario, la eficiencia se ve comprometida, y la respuesta a solicitudes específicas, como productos con descuentos, se ve demorada.

Solución Ideal: Hilos (*Threads*)

La solución ideal radica en la implementación de varios flujos de ejecución simultánea para que una tarea no dependa de la finalización de otras. La herramienta fundamental para lograr esto son los hilos, donde cada hilo representa una tarea independiente.

Exploración de Hilos en Java

En este contexto, exploramos el concepto de hilos (*threads*), que son flujos de ejecución individuales dentro de un proceso. En Java, la Máquina Virtual (JVM) tiene la capacidad de manejar multihilos, permitiendo la creación de varios flujos de ejecución simultánea. Esto se gestiona a través de clases y una interfaz específicas, como *Thread* y *Runnable*.

Ciclo de Vida de un Hilo

La práctica se enfocará en comprender y aplicar el ciclo de vida de un hilo, desde su estado inicial hasta su finalización. Se destacan características importantes, como los estados *"new"*, *"runnable"*, *"not running"* y *"dead"* y se explora cómo los hilos pueden regresar al estado *"runnable"* según la situación.

Planificador (*Scheduler*) y Prioridad

Exploramos el papel crucial del planificador (*Scheduler*) de Java, que decide qué hilos deben ejecutarse según su prioridad. La prioridad de un hilo, modificable a lo largo de su vida, determina su orden de ejecución.

Clases y Elementos Clave

Adicionalmente, se abordan las clases *Thread*, *Runnable*, *ThreadDeath*, y *ThreadGroup*, proporcionando una visión integral de cómo Java gestiona los hilos. Características importantes incluyen el método run que define la acción de un hilo, los métodos *start* y stop para iniciar y detener la ejecución del hilo, y la capacidad de agrupar hilos mediante *ThreadGroup*.

Métodos o Bloques Sincronizados

También se explora la importancia de los métodos o bloques sincronizados para evitar conflictos de acceso en entornos de ejecución paralela. Estos aseguran que

Acosta Porcayo Alan Omar Ing. en Computación 320206102
Gutiérrez Grimaldo Alejandro Ing. en Computación 320282098
Medina Villa Samuel Ing. en Computación 320249538

solo un hilo acceda a un método sincronizado al mismo tiempo, evitando pérdida de información.

Objetivos

- Implementar hilos utilizando la clase *Thread*.
- Implementar hilos utilizando la interfaz *Runnable*.
- Desarrollar habilidades para gestionar excepciones específicas de hilos.

Metodología

Hilo.java

```
public class Hilo extends Thread{
  public Hilo(String nombre){
    super(nombre);
  }
  public void run(){
    for(int i= 0; i<5; i++){
      System.out.println("iteracion: " + (i+1) + "de" + getName());
    }
    System.out.println("Termina el " + getName());

  }

  public static void main(String[] args) {
    new Hilo ("Primer hilo").start();
    new Hilo ("Segundo hilo").start();
    System.out.println("Termina el hilo principal");

  }
}
```

Cuenta.java

```
class Cuenta extends Thread{
  private static long saldo = 0;
  public Cuenta(String nombre){
    super(nombre);

  }

  public void run(){
    if (getName().equals("Deposito 1") || getName().equals("Deposito 2")){
      this.depositarDinero(100);
    }else{
      this.extraerDinero(50);
    }
  }

  public synchronized void extraerDinero(int cantidad){
    try{

      if(saldo <= 0){
        System.out.println(getName() + "espera a que depositen lana");
        sleep(5000);

      }

    }catch(InterruptedException e){
      System.out.println(e);

    }

    saldo-= cantidad;
    System.out.println(getName() + "extrajo 50, el saldo es: " + saldo);
    notifyAll();

  }

  public synchronized void depositarDinero(int cantidad){
    saldo += cantidad;
    System.out.println("Se deposito dinero (100), el saldo es: " + saldo );
    notifyAll();

  }

  public static void main(String[] args) {
    new Cuenta("Acceso A").start();
    new Cuenta("Acceso B").start();
    new Cuenta("Deposito 1 ").start();
    new Cuenta("Deposito 2 ").start();
    System.out.println("termina el main");
  }
}
```

Grupo.java

```
public class Grupo extends Thread{
  public Grupo(ThreadGroup G, String n){
    super(G,n);

  }
  public void run(){
    for (int i=0;i<10 ;i++ ) {
      System.out.println(getName());

    }
  }

  public static void listarHilos(ThreadGroup grupoActual){
    int numhilos;
    Thread[]listaHilos;

    numhilos = grupoActual.activeCount();
    listaHilos = new Thread[numhilos];
    System.out.println("Numero de hilos activos en el grupo " + numhilos);
    for (int i=0; i<numhilos ;i++ ) {
      System.out.println("Hilo Activo: " + (i+1) + "=" + listaHilos[i].getName());

    }

  }

  public static void main(String[] args) {

    ThreadGroup grupoH = new ThreadGroup("Grupo de Hilos prioridad normal");
    Thread hilo1 = new Grupo(grupoH, "Hilo 1 con prioridad normal");
    Thread hilo2 = new Grupo(grupoH, "Hilo 2 con prioridad normal");
    Thread hilo3 = new Grupo(grupoH, "Hilo 3 con prioridad normal");
    Thread hilo4 = new Grupo(grupoH, "Hilo 4 con prioridad normal");
    Thread hilo5 = new Grupo(grupoH, "Hilo 5 con prioridad normal");
    Thread hilo6 = new Grupo(grupoH, "Hilo 6 con prioridad normal");
    Thread hilo7 = new Grupo(grupoH, "Hilo 7 con prioridad normal");
    Thread hilo8 = new Grupo(grupoH, "Hilo 8 con prioridad normal");
    Thread hilo9 = new Grupo(grupoH, "Hilo 9 con prioridad normal");
    Thread hilo10 = new Grupo(grupoH, "Hilo 10 con prioridad normal");

    hilo7.setPriority(Thread.MAX_PRIORITY);
    hilo10.setPriority(Thread.MIN_PRIORITY);
    grupoH.setMaxPriority(Thread.NORM_PRIORITY);

    System.out.println("La prioridad del grupo es de : " + grupoH.getMaxPriority());

    System.out.println("La prioridad del Thread es de : " + hilo1.getPriority());
    System.out.println("La prioridad del Thread es de : " + hilo2.getPriority());
    System.out.println("La prioridad del Thread es de : " + hilo3.getPriority());
    System.out.println("La prioridad del Thread es de : " + hilo4.getPriority());
    System.out.println("La prioridad del Thread es de : " + hilo5.getPriority());
    System.out.println("La prioridad del Thread es de : " + hilo6.getPriority());
    System.out.println("La prioridad del Thread es de : " + hilo7.getPriority());
    System.out.println("La prioridad del Thread es de : " + hilo8.getPriority());
    System.out.println("La prioridad del Thread es de : " + hilo9.getPriority());
    System.out.println("La prioridad del Thread es de : " + hilo10.getPriority());

    hilo1.start();
    hilo2.start();
    hilo3.start();
    hilo4.start();
    hilo5.start();
    hilo6.start();
    hilo7.start();
    hilo8.start();
    hilo9.start();
    hilo10.start();

    listarHilos(grupoH);
  }
}
```

Resultados

Problema 1

Modificar el juego de la liebre y la tortuga de la práctica 3 para simular la carrera de manera asíncrona.

Usar métodos propios de la clase Thread start(), join() y yield()

Solución

```
import java.util.Random;

public class Problema1 {
  static final int N = 10;
  static StringBuilder[][] pista = new StringBuilder[N][N];
  static final int ESPERA = 2000;

  class Corredor extends Thread {
    int pos, i, j;
    final char SIMBOLO;

    Corredor(int pos, int i, int j, char SIMBOLO, String nombre) {
      this.pos = pos;
      this.i = i;
      this.j = j;
      this.SIMBOLO = SIMBOLO;
      setName(nombre);
    }

    @Override
    public void run() {
      synchronized (pista) {
        System.out.println("\nTurno de " + getName() + "\tPosicion: " + pos);
        pos = jugada(pos);
        if (pista[i][j].charAt(pista[i][j].length() - 1) == SIMBOLO)
          pista[i][j].deleteCharAt(pista[i][j].length() - 1);
        else
          pista[i][j].deleteCharAt(pista[i][j].length() - 2);

        if (pos == 0) {
          i = 0;
          j = 0;
        } else {
          i = (pos - 1) / 10;
          j = (pos - 1) % 10;
        }

        pista[i][j].append(SIMBOLO);
        imprimirPista();
        System.out.println("Fin del turno");

        try {
          Thread.sleep(ESPERA);
        } catch (InterruptedException e) {
          System.out.println(e);
        } finally {
          Thread.yield();
        }
      }
    }
  }

  static void inicializarPista() {
    int numeroCasilla = 2;

    pista[0][0] = new StringBuilder("S");
    pista[9][9] = new StringBuilder("E");

    for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
        if ((i == 0 && j == 0) || (i == 9 && j == 9)) continue;
        else pista[i][j] = new StringBuilder(Integer.toString(numeroCasilla++));
      }
    }

    pista[0][0].append("HT");
  }

  static void imprimirPista() {
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++)
        System.out.print(pista[i][j] + "\t");
      System.out.println();
    }
  }

  static int jugada(int posActual) {
    Random rand = new Random();
    int dado, posNueva = posActual;

    dado = rand.nextInt(6) + 1;
    posNueva += dado;

    if (posNueva > 100) posNueva = 100;

    System.out.println("Dado: " + dado);
    System.out.println("Avanza de " + posActual + " a " + posNueva);

    switch (posNueva) {
      case 7, 14, 33, 77, 89 -> {
        System.out.println("Casilla Especial!\tTira de nuevo");
        posNueva = jugada(posNueva);
      } case 6, 23, 42, 56, 82, 90 -> {
        System.out.println("Trampa\tRetrocede de " + posNueva + " a " + (posNueva - 5));
        posNueva -= 5;
      } case 36 -> {
        System.out.println("Casilla Especial!\tAvanza de " + posNueva + " a 71");
        posNueva = 71;
      } case 65 -> {
        System.out.println("Casilla Especial!\ttAvanza de " + posNueva + " a 84");
        posNueva = 84;
      } case 10 -> {
        System.out.println("Trampa\tRegresa al inicio");
        posNueva = 1;
      } case 66 -> {
        System.out.println("Trampa\tRegresa de " + posNueva + " a 40");
        posNueva = 40;
      }
    }

    return posNueva;
  }

  public static void main(String[] args) throws InterruptedException {
    int turno = 0;
    Corredor tortuga = new Problema1().new Corredor(1, 0, 0, 'T', "Tortuga");
    Corredor liebre = new Problema1().new Corredor(1, 0, 0, 'H', "Liebre ");

    inicializarPista();

    System.out.println("Pista inicial");
    imprimirPista();

    tortuga.start();
    liebre.start();
    Thread.sleep(ESPERA);

    while (true) {
      turno++;
      tortuga.join();
      liebre.join();
      tortuga.run();
      Thread.yield();
      liebre.run();
      if (liebre.pos == 100 || tortuga.pos == 100) break;
    }

    System.out.println("\nGanador: " + (turno % 2 != 0 ? "Tortuga!" : "Liebre!"));
    System.out.println("Fin del juego");
  }
}
```

Ejecución

```
Turno de Liebre         Posicion: 1
Dado: 2
Avanza de 1 a 3
ST      2       3H      4       5       6       7       8       9       10
11      12      13      14      15      16      17      18      19      20
21      22      23      24      25      26      27      28      29      30
31      32      33      34      35      36      37      38      39      40
41      42      43      44      45      46      47      48      49      50
51      52      53      54      55      56      57      58      59      60
61      62      63      64      65      66      67      68      69      70
71      72      73      74      75      76      77      78      79      80
81      82      83      84      85      86      87      88      89      90
91      92      93      94      95      96      97      98      99      E
Fin del turno

Turno de Tortuga        Posicion: 1
Dado: 2
Avanza de 1 a 3
S       2       3HT     4       5       6       7       8       9       10
11      12      13      14      15      16      17      18      19      20
21      22      23      24      25      26      27      28      29      30
31      32      33      34      35      36      37      38      39      40
41      42      43      44      45      46      47      48      49      50
51      52      53      54      55      56      57      58      59      60
61      62      63      64      65      66      67      68      69      70
71      72      73      74      75      76      77      78      79      80
81      82      83      84      85      86      87      88      89      90
91      92      93      94      95      96      97      98      99      E
Fin del turno
```

Problema 2

Realice un programa que funcione de manera síncrona que simule la interacción de una clase. Primeramente, que cuando un alumno o el profesor entre a clase muestre un letrero “ingreso a la clase”.

Y que cuando un profesor (hilo) diga “buenos días”, los alumnos en clase (mínimo 5 hilos) respondan buenos días.

Usar métodos propios de la clase Thread start(), join(), nofifyAll() y wait()

Solución

```
class Clase {
    public synchronized void entrarClase(String persona) {
      System.out.println(persona + " ingreso a la clase");
    }

    public synchronized void decirBuenosDias() throws InterruptedException {
      System.out.println("Profesor: Buenos dias");
      notifyAll();
    }
}

class Alumno extends Thread {
  private Clase clase;
  private String nombre;

  public Alumno(Clase clase, String nombre) {
    this.clase = clase;
    this.nombre = nombre;
  }

  @Override
  public void run() {
    clase.entrarClase(nombre);
    try {
      synchronized (clase) {
        clase.wait();
        System.out.println(nombre + ": Buenos dias");
      }
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

}

class Profesor extends Thread {
  private Clase clase;

  public Profesor(Clase clase) {
    this.clase = clase;
  }

  @Override
  public void run() {
    clase.entrarClase("Profesor");
    try {
      Thread.sleep(1000);
      clase.decirBuenosDias();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}

public class Problema2 {
  public static void main(String[] args) {
    Clase clase = new Clase();

    Profesor profesor = new Profesor(clase);
    profesor.start();

    Alumno[] alumnos = new Alumno[5];
    for (int i = 0; i < alumnos.length; i++) {
      alumnos[i] = new Alumno(clase, "Alumno " + (i + 1));
      alumnos[i].start();
    }

    try {
      profesor.join();
      for (Alumno alumno : alumnos) {
        alumno.join();
      }
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Ejecución

```
Profesor ingresó a la clase
Alumno 1 ingresó a la clase
Alumno 4 ingresó a la clase
Alumno 2 ingresó a la clase
Alumno 5 ingresó a la clase
Alumno 3 ingresó a la clase
Profesor: Buenos días
Alumno 1: Buenos días
Alumno 3: Buenos días
Alumno 5: Buenos días
Alumno 2: Buenos días
Alumno 4: Buenos días
```

Problema 3

Usando variables volátiles y métodos sincronizados programe dos hilos en java que calculen n coeficientes de la secuencia de Ulam en la práctica 5.

Usar métodos propios de la interfaz runnable

Solución

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.Scanner;

class Ulam implements Runnable {
  private static final ArrayList<Integer> sequence = new ArrayList<>();
  private final int u;
  private final int v;
  private final int n;
```

```
  public Ulam(int u, int v, int n) {
    this.u = u;
    this.v = v;
    this.n = n;
  }

  static synchronized void ulam(int u, int v, int n) {
    sequence.clear();
    sequence.add(u);
    sequence.add(v);

    for (int i = 0; i < n - 2; i++) {
      ArrayList<Integer> sums = new ArrayList<>();

      for (int j = 0; j < sequence.size(); j++) {
        for (int k = j + 1; k < sequence.size(); k++)
          sums.add(sequence.get(j) + sequence.get(k));
      }

      sums.sort(Comparator.naturalOrder());

      for (int j = 0; j < sums.size(); j++) {
        if (sequence.contains(sums.get(j))) {
          sums.remove(j);
          j--;
        }
      }

      while (true) {
        if (sums.size() == 1) {
          sequence.add(sums.get(0));
          break;
        } else if (sums.get(0).equals(sums.get(1))) {
          int temp = sums.get(0);
          while (sums.get(0).equals(temp))
            sums.remove(0);
        } else {
          sequence.add(sums.get(0));
          break;
        }
      }
    }
  }

  static synchronized ArrayList<Integer> getSequence() {
    return new ArrayList<>(sequence);
  }

  @Override
  public void run() {
    ulam(u, v, n);
  }
}

public class Problema3 {
  public static void main(String[] args) throws InterruptedException {
    Scanner sc = new Scanner(System.in);

    System.out.print("Numero 1: ");
    int u = sc.nextInt();
    System.out.print("Numero 2: ");
    int v = sc.nextInt();
    System.out.print("Cuantos numeros? ");
    int n = sc.nextInt();

    Ulam ulam = new Ulam(u, v, n);

    Thread thread1 = new Thread(ulam);
    Thread thread2 = new Thread(ulam);

    thread1.start();
    thread2.start();

    thread1.join();
    thread2.join();

    ArrayList<Integer> ulamSequence = Ulam.getSequence();

    System.out.println("\nSecuencia de Ulam:");
    System.out.println(ulamSequence);

    sc.close();
  }
}
```

Ejecución

```
Numero 1: 1
Numero 2: 2
Cuantos numeros? 10

Secuencia de Ulam:
[1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
```

Conclusiones

La utilización de hilos en Java se presenta como un componente crucial para mejorar la multitarea de forma eficaz. Desde la supervisión de los ciclos de vida hasta la coordinación de operaciones, estos hilos representan la solución fundamental para desarrollar aplicaciones que sean ágiles y capaces de adaptarse al crecimiento. Este método, respaldado por las robustas utilidades proporcionadas por Java, permite alcanzar un rendimiento óptimo y gestionar eficientemente los recursos en el complejo contexto de la programación concurrente.

Acosta Porcayo Alan Omar, Gutiérrez Grimaldo Alejandro, Medina Villa Samuel

Práctica 13. Patrones de diseño

Resumen

La finalidad de la práctica se enfoca en la elaboración de una aplicación en un lenguaje orientado a objetos, integrando el uso de un patrón de diseño específico. Como fase inicial, se llevará a cabo un análisis exhaustivo para comprender diversos patrones de diseño, explorando sus particularidades y aplicaciones. Subsecuentemente, se empleará este conocimiento adquirido durante la creación de la aplicación, seleccionando y aplicando de manera adecuada un patrón de diseño que se adecue a los requisitos del proyecto. Este enfoque no solo aborda la implementación técnica, sino que también destaca la capacidad de tomar decisiones fundamentadas sobre el diseño del software para optimizar la estructura y funcionalidad de la aplicación.

Introducción

En el ámbito de la ingeniería de software, los patrones de diseño se erigen como soluciones recurrentes y generales para los problemas cotidianos en el diseño de software. Es esencial comprender que un patrón de diseño no constituye un diseño de software completo y listo para la codificación; más bien, se presenta como una descripción o modelo que aborda la resolución de problemas y puede aplicarse en diversas situaciones.

Los patrones de diseño desempeñan un papel crucial al agilizar el proceso de desarrollo, al proporcionar un paradigma desarrollado y probado. Un diseño de software efectivo debe anticiparse a problemas que pueden surgir durante la implementación, cubriendo exhaustivamente todos los detalles. La reutilización de patrones de diseño emerge como una estrategia para prevenir problemas sutiles y mejorar la legibilidad del código para aquellos familiarizados con los patrones.

Acosta Porcayo Alan Omar Ing. en Computación 320206102
Gutiérrez Grimaldo Alejandro Ing. en Computación 320282098
Medina Villa Samuel Ing. en Computación 320249538

Patrones de Software

Cada patrón de diseño aborda problemas recurrentes y presenta soluciones centrales que pueden aplicarse en diversas situaciones. En términos generales, un patrón de diseño consta de cuatro elementos esenciales:

1. **Nombre del patrón:** Una descripción concisa del diseño del problema y su solución.
2. **Problema:** La situación en la que se debe aplicar el patrón, explicando el contexto y, a veces, estableciendo condiciones para su viabilidad.
3. **Solución:** La descripción abstracta de los elementos que componen el diseño y cómo resuelven el problema.
4. **Consecuencias:** Los resultados y recompensas de aplicar el patrón, críticos para evaluar el diseño alternativo y comprender el impacto en la ingeniería de software.

Se distinguen tres niveles de patrones de software: arquitectónicos, de diseño y de programación, cada uno abordando diferentes niveles de estructuras de software y hardware.

Patrón de Diseño Modelo-Vista-Controlador (MVC)

El MVC asigna roles a los objetos en la aplicación, definiendo cómo se comunican entre sí. Con capas bien definidas, como el modelo que encapsula la información, la vista que presenta la información y el controlador que actúa como intermediario, el patrón MVC ofrece beneficios en reutilización, claridad de interfaces y facilidad de extensión.

- **Modelo:** Representa la información y la lógica de manipulación de datos.
- **Vista:** Objeto visible para el usuario, mostrando información y permitiendo la edición.
- **Controlador:** Actúa como intermediario entre vista y modelo, gestionando acciones y el ciclo de vida de los objetos.

Consecuencias del MVC

- .Desacopla vistas y modelos, estableciendo un protocolo de suscriptor/notificador.
- Permite acoplar varias vistas a un modelo y crear nuevas vistas sin modificar el modelo.
- Facilita el manejo de entradas del usuario sin alterar la presentación, encapsulando respuestas en el controlador.

Patrón de Diseño Creacional *Singleton*

El *Singleton* aborda la necesidad de una única instancia de una clase, proporcionando control de acceso y reduciendo el espacio de nombres. Permite refinar operaciones y representaciones, así como la flexibilidad de permitir un número variable de instancias.

Condiciones para su uso

- Debe existir exactamente una instancia de una clase accesible desde un punto conocido.
- Cuando una única instancia deba existir para una subclase y los clientes utilicen la subclase sin modificar su código.

Patrón de Diseño Estructural *Composite*

El patrón *Composite* define una jerarquía de clases compuestas por objetos primitivos y compuestos. Permite tratar con estructuras compuestas u objetos de manera uniforme, facilita la creación de nuevas funcionalidades y propicia la reutilización.

Consecuencias del *Composite*

- Define una jerarquía de clases con objetos primitivos y compuestos.
- Simplifica la tarea del cliente al tratar con estructuras compuestas de manera uniforme.
- Facilita la creación de nuevas funcionalidades y puede hacer difícil restringir componentes nuevos debido a su facilidad de adición.

Patrón de Diseño de Comportamiento *Observer*

Este patrón aborda la necesidad de notificar a un grupo de objetos sobre un evento en la aplicación. Al dividir el sistema en clases cooperativas, el *Observer* garantiza consistencia de información entre objetos relacionados, permitiendo un bajo acoplamiento y reutilización.

Problema: Notificar a diferentes objetos sobre un evento en la aplicación, sin saber cuántos objetos deben reflejar el cambio.

Solución: Crear una clase abstracta que maneje un conjunto de objetos observadores. Cuando cambie el sujeto observado, notificar a todos los observadores.

Consecuencias del *Observer*

- Parte el sistema en clases cooperativas para consistencia de información.
- Permite un bajo acoplamiento y reutilización de clases.

Objetivos

- Aplicar un patrón de diseño específico para mejorar la eficiencia y el rendimiento de la aplicación
- Implementar patrones que mejoren la gestión de errores y excepciones en la aplicación
- Implementar una aplicación en un lenguaje orientado a objetos utilizando algún patrón de diseño.

Resultados

Problema 1

Usando el patrón creacional *Singleton* programe “el genio de la lampará”. El genio solo puede ser usado por una persona y el genio solo puede usada como una y sólo una instancia. El genio pregunta tres deseos de una persona y los cumple.

Solución

```
import java.util.Scanner;

class Genio {
  private static Genio instancia;
  public String nombre;

  private Genio(String nombre) {
    this.nombre = nombre;
  }

  public static Genio getInstancia(String nombre) {
    if (instancia == null)
      instancia = new Genio(nombre);

    return instancia;
  }
```

```
  public void pedirDeseo() {
    Scanner sc = new Scanner(System.in);

    System.out.print("Cual es tu deseo?\n> ");
    String deseo = sc.nextLine();

    System.out.println("Pensando...");

    try {
      Thread.sleep(3000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("Tu deseo se ha cumplido!!!\n");

    sc.close();
  }
}

public class Problema1 {
  public static void main(String[] args) {
    Genio genio = Genio.getInstancia("Genio");

    genio.pedirDeseo();
    genio.pedirDeseo();
    genio.pedirDeseo();
  }
}
```

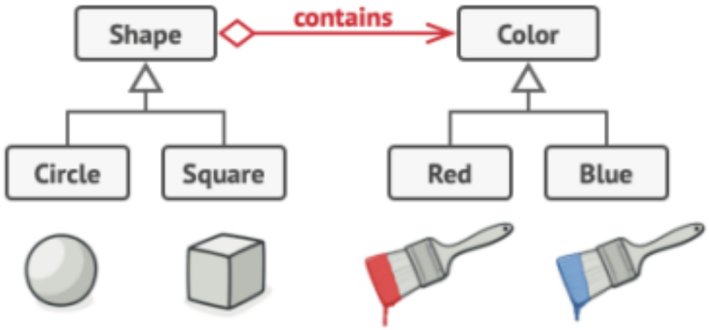
Ejecución

```
Cual es tu deseo?
> Pasar ecuaciones
Pensando...
Tu deseo se ha cumplido!!!

Cual es tu deseo?
> Barcelona gane la champions
Pensando...
Tu deseo se ha cumplido!!!

Cual es tu deseo?
> Pasar el examen de poo
Pensando...
Tu deseo se ha cumplido!!!
```

Problema 2

Implementa el patrón de diseño de estructura *Bridge* que se describe en el problema de figuras y colores en: https://refactoring.guru/design-patterns/bridge

Considere 2 figuras y 2 colores adicionales al problema.



Shape.java

```
public class Shape {
  Color color;
  String nombre;

  public Shape(Color color, String nombre) {
    this.color = color;
    this.nombre = nombre;
  }

  @Override
  public String toString() {
    return "Nombre: " + nombre + "\tColor: " + color;
  }
}

class Circle extends Shape {
  public Circle(Color color, String nombre) {
    super(color, nombre);
  }
}

class Rectangle extends Shape {
  public Rectangle(Color color, String nombre) {
    super(color, nombre);
  }
}

class Triangle extends Shape {
  public Triangle(Color color, String nombre) {
    super(color, nombre);
  }
}

class Square extends Shape {
  public Square(Color color, String nombre) {
    super(color, nombre);
  }
}
```

Color.java

```
public class Color {
  String tipo;

  public String toString() {
    return tipo;
  }
}

class Red extends Color {
  public Red () {
    tipo = "Rojo";
  }
}

class Blue extends Color {
  public Blue () {
    tipo = "Azul";
  }
}

class Green extends Color {
  public Green () {
    tipo = "Verde";
  }
}

class Yellow extends Color {
  public Yellow () {
    tipo = "Amarillo";
  }
}
```

Problema2.java

```
public class Problema2 {
  public static void main(String[] args) {
    Circle circulo = new Circle(new Red(), "Circulo 1");
    Rectangle rectangulo = new Rectangle(new Blue(), "Rectangulo 1");
    Triangle triangulo = new Triangle(new Green(), "Triangulo 1");
    Square cuadrado = new Square(new Yellow(), "Cuadrado 1");

    System.out.println(circulo);
    System.out.println(rectangulo);
    System.out.println(triangulo);
    System.out.println(cuadrado);
  }
}
```

Ejecución

```
Nombre: Circulo 1       Color: Rojo
Nombre: Rectangulo 1    Color: Azul
Nombre: Triangulo 1     Color: Verde
Nombre: Cuadrado 1      Color: Amarillo
```

Conclusiones

Los patrones de diseño en el ámbito de la ingeniería de software proporcionan soluciones comprobadas y recurrentes para enfrentar desafíos típicos en el diseño de aplicaciones. Desde la mejora del rendimiento hasta asegurar la adaptabilidad y mantenibilidad del código, la aplicación efectiva de estos patrones posibilita la construcción de sistemas resistentes y flexibles. Al entender y emplear de manera adecuada patrones creacionales, estructurales y de comportamiento, los desarrolladores tienen la capacidad de perfeccionar la eficiencia, seguridad y usabilidad de sus aplicaciones, lo que a su vez contribuye a un desarrollo de software más eficaz y sostenible.

Referencias

Bridge. (2014). Refactoring.guru. https://refactoring.guru/design-patterns/bridge

Solano, J. (2017, 20 enero). *Manual de prácticas de Programación Orientada a Objetos.* Laboratorio de Computación Salas A y B. http://lcp02.fi-b.unam.mx/