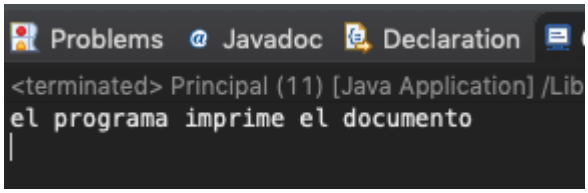
El Principio OCP o principio de apertura y cierre es uno de los principios sólidos. Este principio viene a decir que un programa debe estar abierto a la extensibilidad y cerrado para sus modificaciones. Esto en principio suena pero que muy bien . Sin embargo a veces no es tan sencillo aplicarlo o ver como aplicarlo .Esto es algo habitual que sucede con muchos de los principios de ingeniería de software . El concepto puede ser interesante pero a la gente le resulta a veces difícil ver su aplicación. Vamos a construir un ejemplo de Hola Mundo con el principio OCP. Para ello nos vamos a construir un programa main que según reciba un mensaje por la consola ejecutará una instrucción.

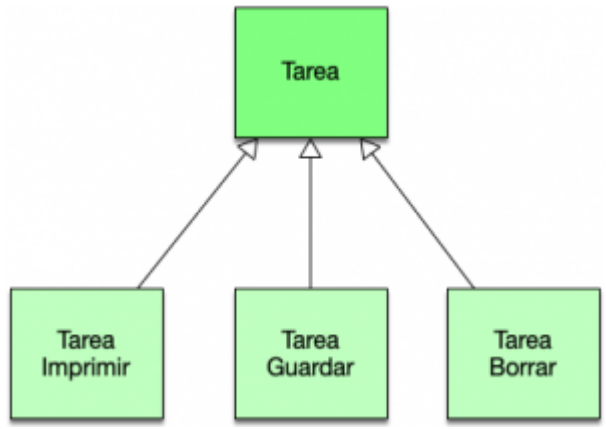
```
package com.arquitecturajava;

public class Principal {

  public static void main(String[] args) {
    String tarea=args[0];
    if (tarea.equals("guardar")) {
      System.out.println("el programa guarda el documento");
    }else if (tarea.equals("imprimir")) {
      System.out.println("el programa imprime el documento");
    }else {
      System.out.println("no hay tarea que ejecutar");
    }

  }

}
```

Si pasamos como parámetro imprimir desde la consola o desde Eclipse . El programa Java se ejecutará y nos mostrará el resultado por la consola.



El principio OCP y la flexiblidad

¿Cómo de flexible es en estos momentos el programa? . La realidad es que muy poco flexible ya que cualquier modificación que queramos hacer estamos obligados a modificar el programa de una forma importante. ¿Podemos hacerlo algo más flexible? . Si , una de las opciones más sencillas es utilizar una jerarquía de clases que gestione Tareas .



Vamos a construirla:

```
package com.arquitecturajava;

public  abstract class Tarea {

  public abstract void ejecutar();
}
```

```
package com.arquitecturajava;
```

```
public class TareaGuardar extends Tarea {

  @Override
  public void ejecutar() {
    System.out.println("el programa guarda el documento");
  }

}
```

```
package com.arquitecturajava;

public class TareaImprimir extends Tarea {

  @Override
  public void ejecutar() {
    System.out.println("el programa imprime el documento");
  }

}
```

Una vez tengamos la jerarquía de clases construida , modificamos el programa principal para que se apoye en ella.

```
package com.arquitecturajava;

public class Principal {

  public static void main(String[] args) {
    String tarea=args[0];
    Tarea mitarea = null;
    if (tarea.equals("guardar")) {
      mitarea= new TareaGuardar();
```

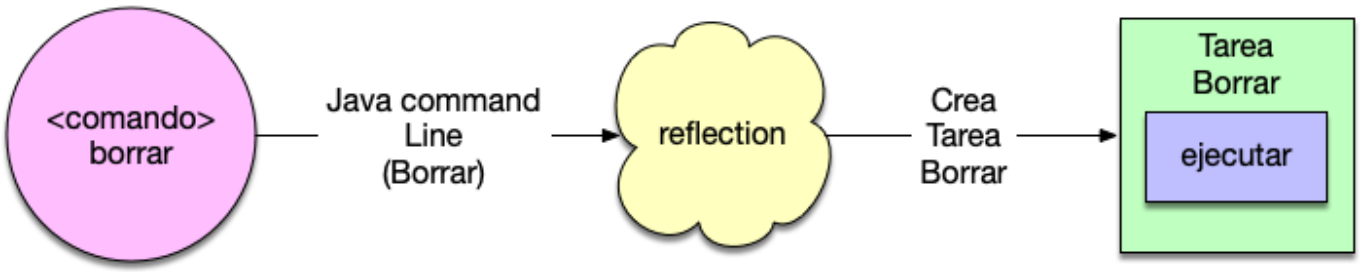
```
    }else if (tarea.equals("imprimir")) {
      mitarea= new TareaImprimir();
    }else {
      System.out.println("no hay tarea que ejecutar");
    }
    mitarea.ejecutar();

  }

}
```

Hemos conseguido construir un programa que ahora es más fácil de extender ya que cada orden se identificada con una nueva Tarea a nivel de la jerarquía de clases . ¿ Ahora bien el programa esta cerrado a las modificaciones? . La realidad es que todavía no cada vez que tengamos que incluir una nueva tarea hay que modificar la estructura if else if .¿ Se puede enfocar de otra forma? . Si podríamos apoyarnos en el API de reflection de Java.

```
package com.arquitecturajava;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class Principal2 {

  public static void main(String[] args) {
    String tarea=args[0];
    Tarea mitarea=null;
    if(tarea!=null) {
       String clase="com.arquitecturajava.Tarea"+tarea.substring(0,
1).toUpperCase() + tarea.substring(1);
        try {
          Constructor<?>
```

```
constructor=(Constructor)Class.forName(clase).getConstructor();
          mitarea= (Tarea) constructor.newInstance();
        } catch (NoSuchMethodException | SecurityException |
ClassNotFoundException | InstantiationException |
IllegalAccessException | IllegalArgumentException |
InvocationTargetException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
        }
    }else {
      System.out.println("no hay tarea que ejecutar");
    }
    mitarea.ejecutar();

  }

}
```

De esta forma cada vez que añadamos un nuevo comando bastará con crear una clase TareaComando .



Es decir si nosotros queremos una tarea que borre ficheros .Primero definiremos como se llama el comando que pasamos por la consola en nuestro caso “borrar”. Será ahora

suficiente con construir una nueva clase Tarea que se denomine TareaBorrar.

```
package com.arquitecturajava;

public class TareaBorrar extends Tarea {

  @Override
  public void ejecutar() {
    System.out.println("el programa borra el documento");
  }

}
```

De esta forma el programa no necesita ser modificado para ir añadiendo tareas adicionales.Simplemente se define el nuevo nombre del comando y por convención construimos una clase TareaComando cumpliendo con el principio OCP. Aquí es donde del API de reflection a través del método newInstance nos echa un cable.

Otros artículos relacionados

1. Adaptadores y patrones y el principio OCP
2. El Principio de Substitución de Liskov
3. Command Pattern en Java y la gestion de tareas
4. https://es.wikipedia.org/wiki/SOLID