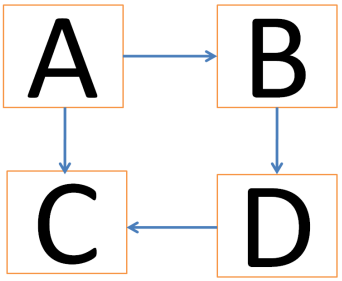
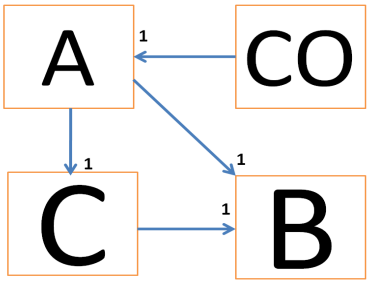
Ejercicios de patrones de diseño

1. ¿Cuál sería el mejor candidato de estos cuatro, según el patrón Controlador?



2. Partiendo de la situación en la que sólo existen objetos de las clases CO, A y C (relacionados entre ellos), se deben ejercer las siguientes responsabilidades:
   - Crear un objeto de la clase B
   - Asociar :C con :B
   - Asociar :A con :B



   - ¿A quién asignar cada una de estas responsabilidades?
   - ¿En virtud de qué patrón o patrones?
   - Trazar un D.I: que ejemplifique el paso de mensajes necesario para realizar las 3 acciones anteriores, como consecuencia de un mensaje f(), llegado a :CO
   - ¿En qué hubiera cambiado la asignación de responsabilidades si no hubiera sido necesario asociar :A con :B

3. Dados los siguientes códigos que representan a programas traductores, escribirlos y compilarlos.

```
public class Traducteur {
      private String motOriginal;
      private String motTraduit;
                  /**
                  * Prepara la palabra a traducir
                  */
      public void preparez(String mot) {
         this.motOriginal=mot;
      }
                  /**
                  * Traduce <b>internamente</b> la palabra anteriormente preparada
                  */
      public void traduire() {
         switch (motOriginal) {
```

```
            case "hola":motTraduit="bonjour";
               break;
            case "adios":motTraduit="adieu";
               break;
            default:motTraduit="mot inconnu";
               break;
         }
      }
         /**
         * Obtiene la palabra traducida previamente.
         */
public String getMotTraduit() {
         return (this.motTraduit);
}
}
```

==========

```
  public class Translator {
      public String translate(String word) {
         String palabra=null;
         switch (word) {
            case "hola":palabra="hello";
               break;
            case "adios":palabra="bye";
               break;
            default:palabra="unknown";
               break;
         }
         return palabra;
      }
   }
```

4. Realizar una clase Prueba1 que utilice la clase “Traducteur” para traducir la palabra “hola” al francés. Mejorar este programa para que traduzca cualquier palabra introducida por teclado mediante la clase Scanner.
5. Igual que el anterior, pero con otra clase llamada Prueba2 que haga lo mismo pero trabajando con el traductor de inglés “Translator”.
6. Definir una clase interfaz ideal de traducción (pista: basada en Translator), y realizar un par de adaptadores para las clases “Traducteur” y “Translator”, basados en esta interfaz.
7. Crear una clase “Prueba3” que acceda al sistema de traducción en inglés, pero a través del Adapter.
8. Crear una clase “Prueba4” que igual que la anterior, pero que acceda al sistema francés.
9. Crear una clase “Prueba5” que haga lo mismo pero escoja entre el sistema francés o inglés en función de que en los parámetros de entrada (args) se reciba la palabra “ing” para activar el sistema inglés o “fra” para el francés.
10. Crear una clase “Prueba6” que decida el adaptador utilizando args, pero no con palabras reservadas específicas, sino con **el nombre del adaptador a utilizar**
    NOTA: Utilizar introspección con el método “Class.forName(“...”).newInstance()”. Es importante indicar la ruta de paquetes completa al adaptador.

11. Crear una clase “Prueba7” igual que la anterior, pero que decida el sistema a conectar en función de lo contenido en un fichero llamado “idioma.ini”, en lugar de los parámetros “args”. Esto permitirá cambiar la traducción de la palabra sin necesidad de recompilar.
12. Crear un adaptador para el subsistema de traducción al italiano proporcionado a continuación. Comprobar que ya no es necesario cambiar nada del código de Prueba7 para poder acceder a estos servicios.

```
public class Traduttore {
private boolean listo;
public Traduttore() {
        listo=false;
}

/**
 * Activa la capacidad de traducción del subsistema
 * Cada vez que se traduce una palabra, se desactiva
 */
public void activare() {
        listo=true;
}
/**
 *
 * @param parola la palabra a traducir
 * @return la palabra traducida
 * @throws WordException si se invoca este método antes de activar
 * la capacidad de traducción mediante el método {@link #activare()}
 */
public String tradurre(String parola) throws WordException {
        String parolaFinale="";
        if (!listo) {throw new WordException();}
        switch (parola) {
        case "hola":
                parolaFinale = "ciao";
                break;
        case "adios":
                parolaFinale = "addio";
                break;
        default:
                parolaFinale = "sconosciuto";
                break;
        }
        listo=false;
        return parolaFinale;

}
}
```

13. Crear una factoría de adaptadores llamada “TraductoresFactory” que ejerza la responsabilidad de decidir qué adaptador utilizar y devolver una referencia al mismo a través de un método ***getTraductorAdapter():ITraductor***.

    NOTA: Mantener un atributo ***adaptador:ITraductor*** en la factoría para no estar “recreeando” el adaptador cada vez que se invoque al getter.
14. Crear una clase **Prueba8** que de haga lo mismo que **Prueba7**, pero utilice dicha factoría para acceder al servicio de traducción.

15. Hacer que la clase **TraductoresFactory** sea un singleton y hacer un programa **Prueba9**, que haga lo mismo que **Prueba8** pero accediendo al singleton.
16. Crear una clase ***Persona***, con atributo ***nombre:String,*** que tenga un método ***saludar()***, cuyo comportamiento, no esté definido en ***Persona,*** sino en una estrategia externa cuya interfaz se llame ***ISaludoStrategy***. Crear un par de Estrategias reales que implementen dicha interfaz, llamadas ***SaludoFormalStrategy*** y ***SaludoAmableStrategy*** que implementen el saludo, mostrando por pantalla “Hola, soy <nombre>” y “Hola, buenos días. Me llamo <nombre>”, respectivamente. Probarlas.
17. Hacer que la creación y el acceso a esta estrategia, se realice a través de una factoría singleton llamada ***StrategyFactory***, la cual decidirá si utilizar una estrategia Formal o Amable, en función del nombre de la clase correspondiente, contenida en un fichero externo “estrategia.ini”
18. Cambiar el criterio de creación de la estrategia en función del minuto que sea. Si el minuto actual es impar, utiliza una estrategia “Formal”, si es par, “Amable”.
    NOTA: Para saber el minuto actual **java.util.Calendar.getInstance().get(java.util.Calendar.MINUTE)**
19. Volver a la forma de trabajar de la factoría en el ejercicio 17 (es decir, a través del contenido de un fichero). Crear un nuevo atributo ***edad:int*** en ***Persona.*** Hacer un constructor sobrecargado para poder introducir esta edad (llegado este punto el programa anterior que trabajaba con las estrategias “Amable” y “Formal” debería seguir funcionando. Crear una nueva estrategia de saludo llamada ***SaludoCompletoStrategy*** que implemente saludar, diciendo “Hola, soy <nombre> y tengo <edad> años”. Comprobar su corrección.
20. Utilizar un patrón composite para crear lo siguiente: una interfaz “IMusico” que tenga métodos **setName(String)** y **tocar()**. Cada músico “tocará” su instrumento, mostrando por pantalla entre paréntesis su nombre y haciendo un sonido onomatopéyico del mismo. Crear un par de clases de músicos solistas (Trompetista, Guitarrista) que implementen dicha interfaz. Hacer una clase “Banda” que sea un composite de músicos. Cuando se le pide a una Banda que toque, lo que hará es pedirle ésta a todos sus miembros que toquen.

    Probar dichas clases desde un main en el que creemos un array de IMusico (llamado “actuacion”) en el que puedan participar tanto solistas como bandas de solistas, como supergrupos (bandas de bandas). Ponerlos a todos a tocar.
21. Hacer que banda sea ahora una interfaz (llamada IBanda) e implementar distintos tipos de banda: p.ej. “BandaDeTrompetas”, en las que cualquier instrumentista que no toque una trompeta) no suena, pero puede haber músicos que no toquen la trompeta, BandaTotal en la que suenan todos, BandaSolistas en la que no se permite que haya bandas de bandas.