

A continuación encontrará cuatro casos que deben resolverse bajo la misma modalidad de ejercicios que se han practicado en clase.

Cada equipo de trabajo deberá:

- Anotar sus nombres y números de carné en la tabla que se presenta abajo.
- Para cada caso deben resolver los puntos (1..5) y completar la tabla asociada a cada caso, incorporando redacción formal donde se solicita la justificación y proporcionando imágenes de modelos y código **ABSOLUTAMENTE LEGIBLES**.
- Este documento debe renombrarse bajo el nombre **IIExamenGrupoNNMM** y entregarlo en formato **PDF (sin excepción)**.
- El equipo creará un proyecto de programación Java (preferiblemente usando IDE Netbeans) denominado **IIExamenGrupoNNMM** (donde N es 40 si es de la sede CTLSJ o 02 si es de la Sede Cartago, y MM es el número de grupo asignado de trabajo). Construyan en el proyecto cuatro carpetas, cada una con el nombre **CasoX\_Patron** (donde X es el número de caso a resolver y Patron es el nombre del patrón con el que proponen la solución). Ejemplo **Caso7\_Patron**.

Al tec Digital deberá subir un archivo comprimido llamado **IIExamen\_GrupoNNMM** siguiendo la misma nomenclatura del proyecto programado que contiene el proyecto Java y el enunciado del examen en formato **PDF**.

## Caso 1

---

Los objetos `FileInputStream` típicamente representan archivos de texto accedidos en orden **secuencial**, byte a byte. Con `FileInputStream`, se puede acceder a un byte, varios bytes o el archivo completo.

Si se revisa el API IO de Java, la cantidad de objetos especializados en el manejo de archivos de texto es enorme. Hay objetos `BufferedInputStream` que incluye en su funcionamiento un buffer de datos para un mejor rendimiento y añade una funcionalidad de leer una línea, `readLine()`, para leer un renglón a la vez.

Adicionalmente, existe otro tipo de objeto `LineNumberInputStream` que añade la funcionalidad de contar la cantidad de líneas contenidas en el archivo y hasta puede indicar cual es el número de línea actualmente accedido en un momento del tiempo.

### Lo que debe hacer en este caso:

1. ¿Cuál patrón considera usted que fue utilizado para poder proveer todos estos tipos distintos de manejadores de archivos de texto? **(2 puntos)**

El patrón utilizado fue el decorador.

2. Justifique formalmente la selección del patrón para este caso. Aporte del diagrama original de UML correspondiente al patrón. **(3 puntos)**

Además de la misma documentación del API del paquete `Java.io`, es notorio debido a que en su funcionalidad lo que se hace es “pasarle” el mismo objeto a otro que lo “contiene”. Este comportamiento de contener otro objeto que proviene de la misma interfaz es el mismo comportamiento que propone el patrón Decorator.

3. Siguiendo el modelo del patrón, construya detalladamente el modelo de UML que da solución a este planteamiento. Añada al modelo la posibilidad de ofrecer además un tipo de Stream asociado al manejador de archivos de texto que transforme el contenido del archivo en minúsculas ! Para el manejador de archivos que transforma a minúsculas, llame al objeto

LCIStream, esto para no interferir con las propias del API si lo programa en Java.

**(5 puntos)**

**Ver la tabla mas adelante**

4. Programe el funcionamiento del modelo del patrón seleccionado según su propuesta. Para efectos de la programación del caso, construya para representar las clases `FileInputStream`, `BufferedInputStream` y `LineNumberInputStream` bajo los nombres `FISStream`, `BISStream` y `LNISStream` respectivamente con las funcionalidades necesarias para realizar las tareas de leer el contenido de un archivo de texto según lo realiza el API de Java de acuerdo a lo expuesto.

**(10 puntos)**

**Revisar el código adjunto**

5. Evidencias del código producido de los elementos significativos del patrón y su programa de prueba, así como screenshots de funcionamiento. **(5 puntos)**

Corrida de prueba:

```
IIExamenGrupo0110 (run-single) X
compile-single:
run-single:
THIS
IS A
      lineas: 2
TEST
file
BUILD SUCCESSFUL (total time: 0 seconds)
```

Archivo de prueba:

```
1 THIS
2 IS A
3 TEST
4 FILE
5 |
```

FISStream:

```
public class FISStream implements InputStream{

    private FileInputStream stream;

    public FISStream(String filename) {
        try {
            stream=new FileInputStream(filename);
        } catch (FileNotFoundException ex) {
            Logger.getLogger(FISStream.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @Override
    public byte read() {
        try {
            return (byte) stream.read();
        } catch (IOException ex) {
            Logger.getLogger(FISStream.class.getName()).log(Level.SEVERE, null, ex);
        }
        return 0;
    }
}
```

LNISStream:

```
public class LNISStream extends Decorator {  
    private int lines;  
  
    public LNISStream(InputStream stream) {  
        super(stream);  
    }  
  
    @Override  
    public byte read() {  
        byte c = super.read();  
        if (c == '\n')  
            lines++;  
        return c;  
    }  
  
    public int getLineNumber() {  
        return lines;  
    }  
}
```

BStream:

```
*/  
public class LNStream extends Decorator {  
    private int lines;  
  
    public LNStream(InputStream stream) {  
        super(stream);  
    }  
  
    @Override  
    public byte read() {  
        byte c = super.read();  
        if (c=='\n')  
            lines++;  
        return c;  
    }  
  
    public int getLineNumber() {  
        return lines;  
    }  
}
```

| Respuesta Caso 1                | Patrón a utilizar  | Decorator   |
|---------------------------------|--|-------------|
|                                 | Tipo de Patrón   | Estructural |
| Justificación de uso del patrón | El patron Decorator se utiliza para agregarle responsabilidades dinámicamente a un objeto, que es lo que se desea hacer con el FileInputStream. Además, utilizar este patrón nos permite poder crear diferentes decoradores para un tipo de input stream (en este caso para el file input stream). |             |

|  |   |
|--|---|
| Diagrama de UML asociado   | <pre>classDiagram     class Interface {         &lt;&lt;interface&gt;&gt;         +doThis()     }     class CoreFuncionalidad {         +doThis()     }     class OptionalWrapper {         -wrappee         +doThis()     }     class CoreFuncionalidad2 {         +doThis()     }     class CoreFuncionalidad3 {         +doThis()     }     Interface &lt; -- CoreFuncionalidad     Interface &lt; -- OptionalWrapper     OptionalWrapper &lt; -- CoreFuncionalidad2     OptionalWrapper &lt; -- CoreFuncionalidad3     OptionalWrapper "1" --&gt; OptionalWrapper</pre>   |
| Modelo en UML que soluciona el problema                            | <pre>classDiagram     class InputStream {         &lt;&lt;interface&gt;&gt;         +read()     }     class FileInputStream {         +read()     }     class FilterInputStream {         -stream         +read()     }     class BufferedInputStream {         -buffer         +read()         +readLine()     }     class LineNumberInputStream {         -lines         +read()         +getLineNumber()     }     class LCIStream {         +read()     }     InputStream &lt; -- FileInputStream     InputStream &lt; -- FilterInputStream     FilterInputStream &lt; -- BufferedInputStream     FilterInputStream &lt; -- LineNumberInputStream     FilterInputStream &lt; -- LCIStream     FilterInputStream "1" --&gt; FilterInputStream</pre> <p>En el código los nombres son remplazados utilizando el formato solicitado por la profesora.</p> |
| Evidencias de código significativo en la implementación del patrón | Ver arriba  |
| Screehscreens de funcionamiento                                    | Ver arriba  |

## Caso 2

---

En el desarrollo de un proyecto de curso, los estudiantes deberán manejar el acceso de los usuarios al mismo a través de un mecanismo que permita la autenticación del usuario a través de sus credenciales (login y contraseña). Además, se debe y permitir el registro de nuevos usuarios que forman parte de un equipo de trabajo y que requerirán acceder al sistema, por lo que el perfil del nuevo usuario debe completarse con información relevante, como su nombre con dos apellidos, fecha de nacimiento, datos de contacto como dirección física, correo electrónico, número del móvil como mínimo, eventualmente podrían agregarse otros datos a la sección de contactos como código postal, cuenta de alguna red social, entre otros. Los datos de contacto deben tener el formato adecuado y por supuesto no se debe permitir el registro de usuarios menores de edad. Los dos apellidos del usuario son requeridos.

Como es evidente, se requerirá validar algunos de los campos que forman parte del perfil del usuario al momento de registro y además, posteriormente se deberá realizar la validación de las credenciales, sin mencionar que en el proyecto como tal se requiere realizar validaciones que van de acuerdo a la lógica de negocio de la aplicación en desarrollo.

Por esta razón, al momento de la conceptualización del proyecto, se decidió definir una interface que permita estandarizar el llamado a mecanismos de validación independientemente del elemento a revisar, de modo que cuando se requiera una clase en particular, pueda implementar este mecanismo de la forma que le corresponda, según su contexto por medio de clases especializadas que lleve a cabo esta tarea.

Esta interface establece que el método devolverá una lista de hileras que contiene los errores que pudieron ser detectados en el proceso de validación. La interfaz se ha definido de manera totalmente genérica dentro del proyecto, a fin de que pueda ser utilizada con cualquier elemento que requiera implementar mecanismos de validación. Se muestra a continuación la sintaxis de la interface definida:



```
public interface Validator<T> {  
  
    List<String> validate(T info);  
}
```

Se desea que la página o pantalla de registro de usuarios sea sometida al proceso de revisión de las distintas validaciones al momento de intentar registrar un nuevo usuario de modo que si se obtienen errores de distinta índole en el proceso sean desplegados en la pantalla y no se permita el registro para que el usuario corrija.

**Lo que debe hacer en este caso:**

1. ¿Cuál patrón cree usted que el equipo de trabajo utilizó para aportar flexibilidad y uniformidad al llevar a cabo las validaciones de información?

**(2 puntos)**

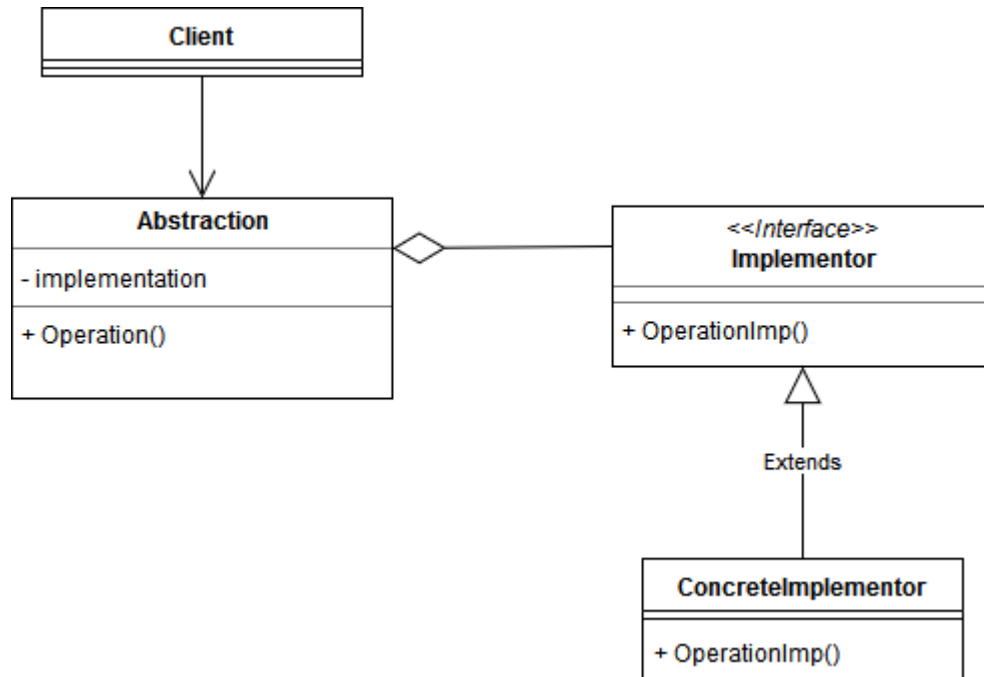
Bridge es el patrón utilizado en este caso.

2. Justifique formalmente la selección del patrón para este caso. Aporte del diagrama original de UML correspondiente al patrón.

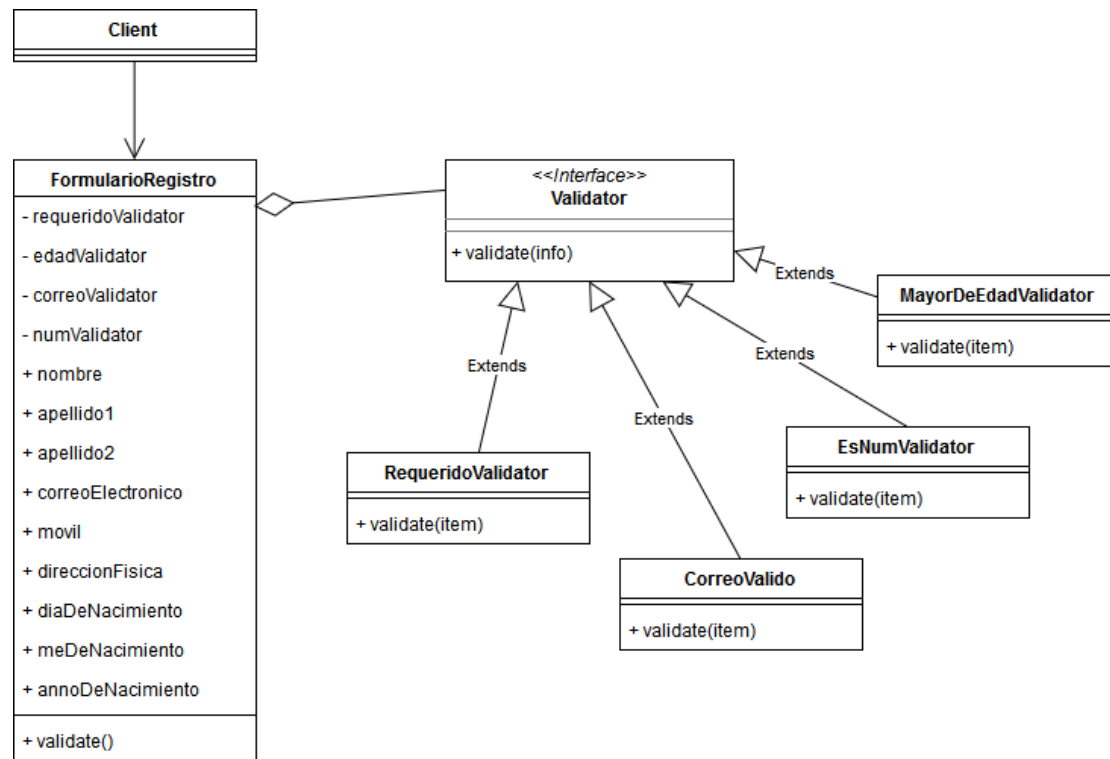
**(3 puntos)**

Primeramente, en el caso se habla de separar la interfaz de la implementación, lo cual es un indicador fuerte para que se realice utilizando el patrón Bridge. Luego se habla de que se creó una interfaz de la cual se crean las versiones necesarias dependiendo del contexto. Por lo tanto, tenemos un objeto que llama a este validator, que probablemente lo contenga. Todo esto apunta a ser un Bridge.

Diagrama original de Bridge:



3. Proponga el diagrama de UML que pudo haber implementado el equipo de trabajo e incorpore los elementos necesarios para ajustarse al requerimiento técnico que debe cumplir el formulario de registro de nuevos usuarios al sistema? Justifique su respuesta. (5 puntos)



Primero, existe un cliente el cual accesa este formulario de registro, este cliente es una vista. Otra manera de verlo es que esta vista contiene esos elementos. Luego se encuentran los validadores requeridos dentro de el formulario, y cuando se necesita validar el formulario se llama cada validador requerido pasándole por parámetro el valor que desea validar.

La cantidad de validadores se puede volver tan grande como la exquisitez con la que se desean filtrar los datos.

4. Programe el funcionamiento del modelo del patrón seleccionado según su propuesta. Desarrolle una pequeña interfaz de usuario que permita el registro de un nuevo perfil de usuario de modo que al momento de intentar registrar un usuario, se solicite al patrón llevar a cabo las validaciones correspondientes mostrando la lista de errores que podrían generarse o bien permitiendo el registro exitoso del nuevo usuario. Recuerde que una validación final debe ser que no se permite la duplicidad de registros.

(10 puntos)

Ver el código adjunto.

|   |  |             |
|---|--|-------------|
| <b>Respuesta Caso 2</b>                 | <b>Patrón a utilizar</b>   | Bridge      |
|   | <b>Tipo de Patrón</b>  | Estructural |
| Justificación de uso del patrón         | <p>Primeramente, en el caso se habla de separar la interfaz de la implementación, lo cual es un indicador fuerte para que se realice utilizando el patrón Bridge. Luego se habla de que se creó una interfaz de la cual se crean las versiones necesarias dependiendo del contexto. Por lo tanto, tenemos un objeto que llama a este validator, que probablemente lo contenga. Todo esto apunta a ser un Bridge</p>  |             |
| Diagrama de UML asociado                | <pre> classDiagram     class Client     class Abstraction {         - implementation         + Operation()     }     class Implementor {         &lt;&lt;Interface&gt;&gt;         + OperationImp()     }     class ConcreteImplementor {         + OperationImp()     }     Client --&gt; Abstraction     Abstraction o-- Implementor     Implementor &lt; -- ConcreteImplementor     </pre>  |             |
| Modelo en UML que soluciona el problema | <pre> classDiagram     class Client     class FormularioRegistro {         - requeridoValidator         - edadValidator         - correoValidator         - numValidator         + nombre         + apellido1         + apellido2         + correoElectronico         + movil         + direccionFisica         + diaDeNacimiento         + meDeNacimiento         + annoDeNacimiento         + validate()     }     class Validator {         &lt;&lt;Interface&gt;&gt;         + validate(info)     }     class RequeridoValidator {         + validate(item)     }     class MayorDeEdadValidator {         + validate(item)     }     class EsNumValidator {         + validate(item)     }     class CorreoValido {         + validate(item)     }     Client --&gt; FormularioRegistro     FormularioRegistro o-- Validator     Validator &lt; -- RequeridoValidator     Validator &lt; -- MayorDeEdadValidator     Validator &lt; -- EsNumValidator     Validator &lt; -- CorreoValido     </pre> |             |

Evidencias de código significativo en la implementación del patrón

```
public class RequeridoValidator implements Validator<Object>{

    public RequeridoValidator() {
    }

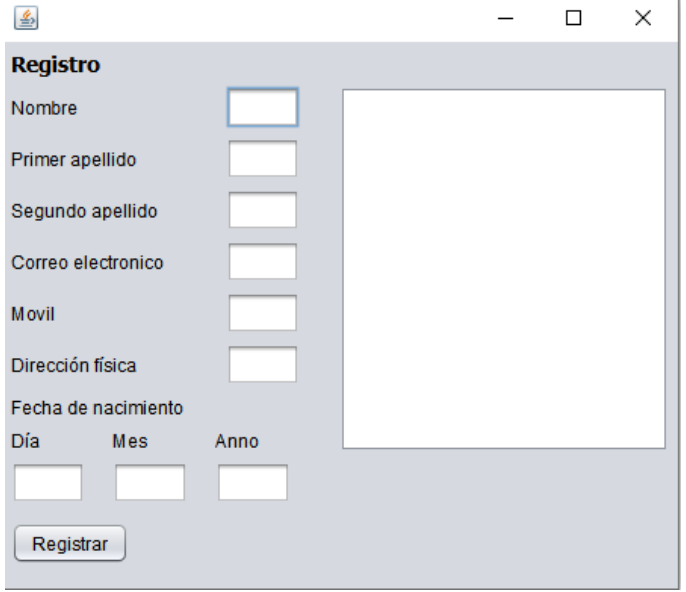
    @Override
    public List validate(Object info) {
        ArrayList<String> res = new ArrayList<String>();
        if (info instanceof String && ((String) info).equals("")) {
            res.add("Cadena vacia recibida");
        } else if (info == null) {
            res.add("Campo vacio recibido");
        }
        return res;
    }
}

public class EsNumValidator implements Validator<String>{

    public EsNumValidator() {
    }

    @Override
    public List validate(String info) {
        ArrayList<String> res = new ArrayList<String>();
        if (info == "")
            res.add("Cadena vacia recibida");
        else {
            try {
                int i = Integer.parseInt((String) info);
            } catch (NumberFormatException | NullPointerException nfe) {
                res.add("Numero en formato desconocido");
            }
        }

        return res;
    }
}
```

|                                |  |
|--------------------------------|--|
|                                | <pre>1 public class FormularioRegistro { 2     RequeridoValidator requeridoValidator; 3     EsNumValidator numValidator; 4     CorreoValidoValidator correoValidator; 5     MayorDeEdadValidator edadValidator; 6 7     String nombre; 8     String apellido1; 9     String apellido2; 10    String correoElectronico; 11    String movil; 12    String direccionFisica; 13    String diaDeNacimiento; 14    String mesDeNacimiento; 15    String annoDeNacimiento; 16 17    public FormularioRegistro() { 18        this.requeridoValidator = new RequeridoValidator(); 19        this.numValidator = new EsNumValidator(); 20        this.correoValidator = new CorreoValidoValidator(); 21        this.edadValidator = new MayorDeEdadValidator(); 22    } 23 24    public List validate() { 25        ArrayList&lt;String&gt; res=new ArrayList&lt;&gt;(); 26        try { 27            res.addAll(requeridoValidator.validate(nombre)); 28            res.addAll(requeridoValidator.validate(apellido1)); 29            res.addAll(requeridoValidator.validate(apellido2)); 30            res.addAll(requeridoValidator.validate(correoElectronico)); 31            res.addAll(correoValidator.validate(correoElectronico)); 32            res.addAll(requeridoValidator.validate(movil)); 33            res.addAll(numValidator.validate(movil)); 34            res.addAll(requeridoValidator.validate(direccionFisica)); 35            res.addAll(requeridoValidator.validate(diaDeNacimiento)); 36            res.addAll(numValidator.validate(diaDeNacimiento)); 37            res.addAll(requeridoValidator.validate(mesDeNacimiento)); 38            res.addAll(numValidator.validate(mesDeNacimiento)); 39            res.addAll(requeridoValidator.validate(annoDeNacimiento)); 40            res.addAll(numValidator.validate(annoDeNacimiento)); 41            SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd"); 42            res.addAll(edadValidator.validate(format.parse ( annoDeNacimiento+"-"+mesDeNacimiento+"-"+diaDeNacimiento))); 43        } catch (ParseException ex) { 44 45        } 46    } 47 }</pre> |
| Screenshoots de funcionamiento |    |

The image displays two sequential screenshots of a web application's registration form, titled "Registro". The form is designed with a light gray background and includes several input fields for user registration. The first screenshot shows the form with the following data: Name (Adam), First Last Name (West), Second Last Name (Smith), Email (empty), Mobile (123), Physical Address (?th AV.), and Birth Date (Day: 1, Month: 11, Year: 1997). A "Registrar" button is visible at the bottom. A red error message box on the right side of the form indicates "Cadena vacia recibida" and "Email invalido". The second screenshot shows the same form after the user has entered the email address "5". The error message box now only displays "Email invalido". The "Registrar" button remains visible at the bottom of the form.

**Registro**

Nombre: Adam

Primer apellido: West

Segundo apellido: Smith

Correo electronico:

Movil: 123

Dirección física: ?th AV.

Fecha de nacimiento: Día: 1, Mes: 11, Anno: 1997

Registrar

Cadena vacia recibida  
Email invalido

**Registro**

Nombre: Adam

Primer apellido: West

Segundo apellido: Smith

Correo electronico: 5

Movil: 123

Dirección física: ?th AV.

Fecha de nacimiento: Día: 1, Mes: 11, Anno: 1997

Registrar

Email invalido





### Caso 3

Application Service Providing o ASP es un modelo de negocio que da soporte a medianas empresas con software empresarial complejo y altamente integrado. Este modelo implementa una técnica y una infraestructura organizativa que garantiza un alto grado de disponibilidad del sistema y la seguridad de los datos. Varios clientes comparten el uso de la infraestructura central y cuentan con un mecanismo de configuración para acceder a sus datos y sus diversas funcionalidades que han sido contratadas.

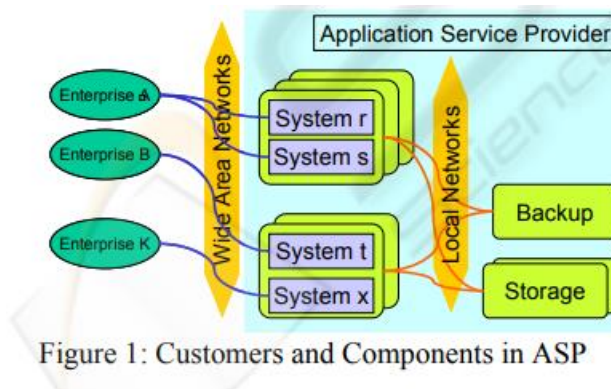


Figure 1: Customers and Components in ASP

La administración técnica de ASP debería ser soportado por un software de monitoreo que abarque una gran gama de componentes de software y hardware, esto porque un ASP normalmente es un sistema distribuido que tiene múltiples servidores, las aplicaciones son basadas en una arquitectura multicapa, tiene una gran exigencia de disponibilidad dado que múltiples usuarios pueden solicitar múltiples recursos en simultáneo, la apariencia de la aplicación puede variar dependiendo del cliente que ingrese, esto es, la misma aplicación puede tener distintas apariencias para ser manipulada por usuarios distintos, entre otros.

A menudo los monitores para componentes específicos son basados en técnicas no orientada a objetos, por ejemplo, parsing de archivos de log o accesos a información interna de ejecución que es llevada a cabo por los system-call de los sistemas operativos.

Entonces en ocasiones un monitor específico para un ASP requiere por ejemplo tener acceso a logs de estado que le sean provistos por un "FileMonitor", y en otras necesita acceder a la tabla de procesos de un sistema operativo que pueden ser obtenidas por un "ProcessMonitor".

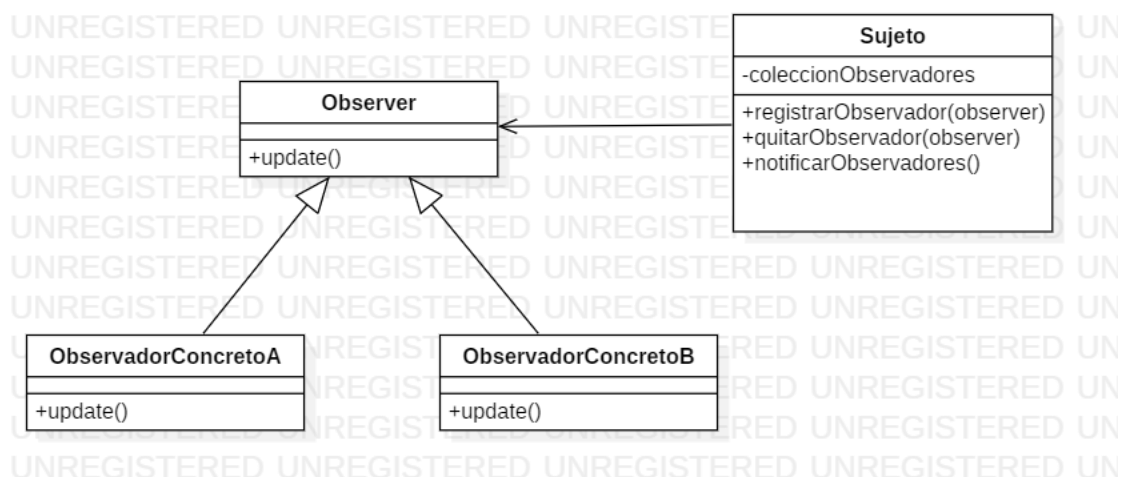
Por otro lado podría requerirse distintos tipos de monitores, uno que esté monitoreando las bases de datos en la zona de almacenamiento (ORCLMonitor, DBaseMonitor), y en ocasiones se deberá conectar otro que rastree el ASP directamente (ASPMonitor).

**Lo que debe hacer en este caso:**

1. ¿Cuál sería el patrón que se puede utilizar para implementar esta estrategia de monitoreo asociada a este tipo de aplicación como es el ASP? **(2 puntos)**

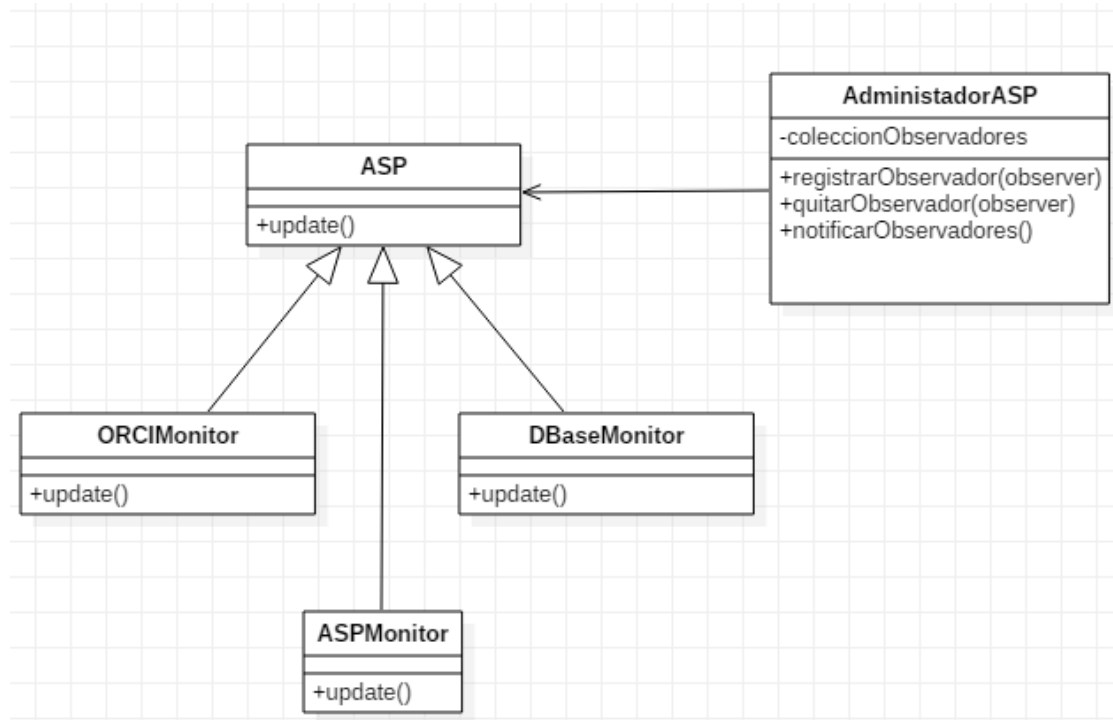
Se usara el patrón de Observer.

2. Justifique formalmente la selección del patrón para este caso. Aporte del diagrama original de UML correspondiente al patrón. **(3 puntos)**



3. Proponga el diagrama de UML que implementa la flexibilidad de poder brindar al ASP el uso de monitores distintos que puedan dedicarse a recuperar información de comportamiento de los componentes de hardware y software involucrados en el ASP? Justifique su respuesta. **(5 puntos)**

Se plantea el uso del patrón Observer debido a que el software de ASP es único y lo que varía es la forma en como se ve, por lo tanto los monitores, dependiendo de lo que se ocupe, trabajarán como observadores, los cuáles pueden ser registrados o dados de baja según sea necesario por los administradores de ASP.



4. Programe el funcionamiento del modelo del patrón seleccionado según su propuesta. Suponga una instancia de un ASP con una configuración particular que debe llevar a cabo en un momento determinado operaciones de monitoreo de comportamiento de accesos a la base de datos y ver los archivos de log, y en otras ocasiones debe analizar el comportamiento de atención de procesos por parte del sistema operativo que le da soporte a la aplicación. **(10 puntos)**

|                                 |                   |                       |
|---------------------------------|-------------------|-----------------------|
| Respuesta Caso 3                | Patrón a utilizar | <i>Observer</i>       |
|                                 | Tipo de Patrón    | <i>Comportamiento</i> |
| Justificación de uso del patrón |                   |                       |

|  |  |
|--|--|
| Diagrama de UML asociado   | <pre>classDiagram     class Sujeto {         -coleccionObservadores         +registrarObservador(observer)         +quitarObservador(observer)         +notificarObservadores()     }     class Observer {         +update()     }     class ObservadorConcretoA {         +update()     }     class ObservadorConcretoB {         +update()     }     Sujeto &lt; -- ObservadorConcretoA     Sujeto &lt; -- ObservadorConcretoB     Observer &lt; -- ObservadorConcretoA     Observer &lt; -- ObservadorConcretoB</pre> |
| Modelo en UML que soluciona el problema                            |  |
| Evidencias de código significativo en la implementación del patrón |  |
| Screeeshoots de funcionamiento                                     |  |

## Caso 4

---

Suponga una casa de producción de contenido que contrata agentes vendedores para escribir contenido para la organización.

Para cada proyecto asignado a un proveedor, el departamento de RRHH proporciona al vendedor un contrato con una serie de cláusulas que contienen términos y condiciones y además un acuerdo de confidencialidad que debe aceptar antes de comenzar a trabajar.

El contenido de los acuerdos sigue siendo el mismo para todos los proveedores y un empleado de Recursos Humanos sólo debe completar el nombre del proveedor antes de enviar un acuerdo al proveedor.

Tanto el formato del contrato como del acuerdo de confidencialidad se encuentran en una base de datos remota distintas que deben ser accedida para solicitar el envío un nuevo contrato para un nuevo vendedor. El vendedor no será contratado si no ha firmado ambos documentos.

El contrato debe ser firmado tanto por el encargado de RRHH y el vendedor.

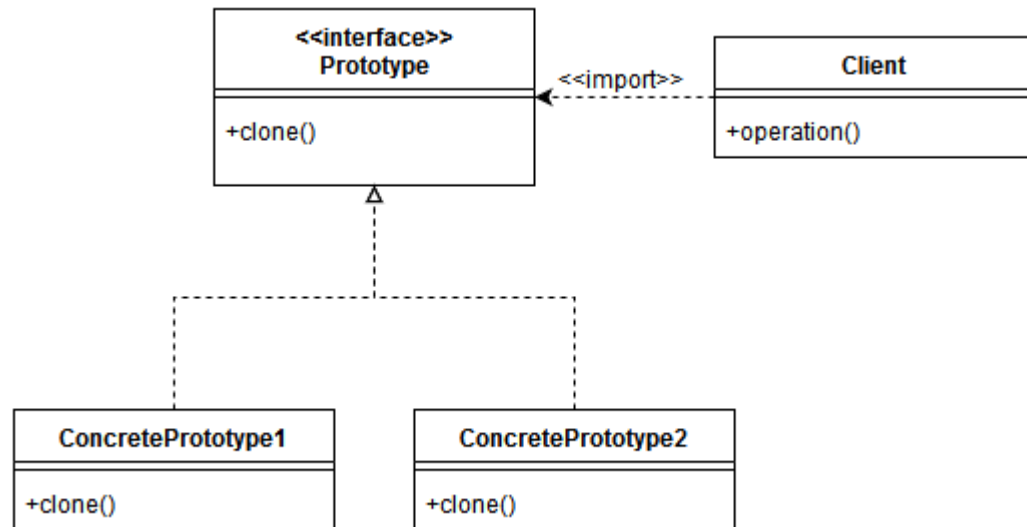
### Lo que debe hacer en este caso:

1. ¿Cuál sería el patrón que se puede utilizar para obtener ambos documentos y contratar al vendedor? **(2 puntos)**

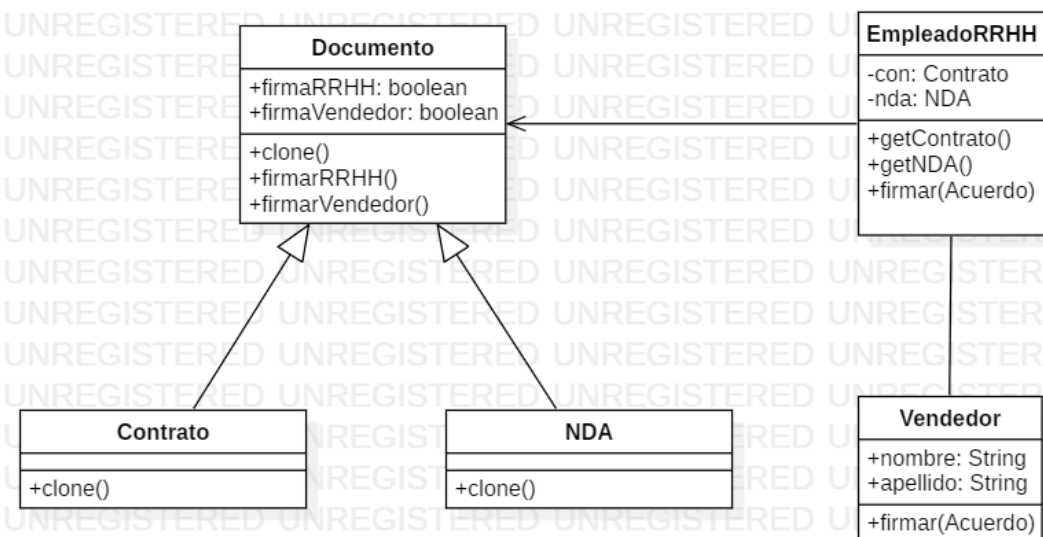
El patrón que se podría usar para poder obtener los documentos desde la base de datos sería el de **prototype**.

2. Justifique formalmente la selección del patrón para este caso. Aporte del diagrama original de UML correspondiente al patrón. **(3 puntos)**

Esta decisión se justifica debido a que cada documento tiene el mismo formato, que está guardado en una base de datos remota, y lo que cambia es que se completan de manera distinta. Con el patrón escogido, es solo necesario clonar el formato vacío que se encuentra en la base de datos.



3. Proponga el diagrama de UML que implementa la solicitud por parte del encargado de RRHH de los documentos que se requieren para la contratación de un nuevo vendedor. **Justifique su respuesta.** (5 puntos)



4. Programe el funcionamiento del modelo del patrón seleccionado según su propuesta. Recuerde que cada documento debe ser obtenido de repositorios (o bases de datos distintas) para que ambos puedan ser completados por las partes, y ambos documentos deben estar firmados por ambos (encargado de RRHH y vendedor) para que se finiquite la contratación. **(10 puntos)**

|  |   |                   |
|--|---|-------------------|
| Respuesta Caso 4   | Patrón a utilizar   | <i>Prototype</i>  |
|  | Tipo de Patrón  | <i>Creacional</i> |
| Justificación de uso del patrón                                    | Cada documento tiene el mismo formato, lo que varía es como se completa, además ya que se tiene este formato en una base de datos remota, se puede tomar el formato como el prototipo.  |                   |
| Diagrama de UML asociado   | <pre> classDiagram     class Prototype {         &lt;&lt;interface&gt;&gt;         +clone()     }     class ConcretePrototype1 {         +clone()     }     class ConcretePrototype2 {         +clone()     }     class Client {         +operation()     }     Prototype &lt; .. ConcretePrototype1     Prototype &lt; .. ConcretePrototype2     Client ..&gt; Prototype : &lt;&lt;import&gt;&gt; </pre>   |                   |
| Modelo en UML que soluciona el problema                            | <pre> classDiagram     class Documento {         +firmaRRHH: boolean         +firmaVendedor: boolean         +clone()         +firmarRRHH()         +firmarVendedor()     }     class Contrato {         +clone()     }     class NDA {         +clone()     }     class EmpleadoRRHH {         -con: Contrato         -nda: NDA         +getContrato()         +getNDA()         +firmar(Acuerdo)     }     class Vendedor {         +nombre: String         +apellido: String         +firmar(Acuerdo)     }     Documento &lt; -- Contrato     Documento &lt; -- NDA     EmpleadoRRHH --&gt; Contrato : -con     EmpleadoRRHH --&gt; NDA : -nda </pre> |                   |
| Evidencias de código significativo en la implementación del patrón | <pre> public class Contrato extends Documento{     public Contrato() {     }      @Override     public Documento copiar() {         return new Contrato();     } }  public class NDA extends Documento{     public NDA() {     }      @Override     public Documento copiar() {         return new NDA();     } } </pre>  |                   |

|                               |  |
|-------------------------------|--|
|                               | <pre>public abstract class Documento {     public String proveedor;     public boolean firmaRRHH;     public boolean firmaVendedor;      public Documento() {     }      public abstract Documento copiar();      public boolean aprobado() {         if(firmaRRHH &amp;&amp; firmaVendedor)             return true;         return false;     }      public void firmarRRHH(){         System.out.println("El documento ha sido firmado por el empleado de RRHH...");         this.firmaRRHH = true;     }      public void firmarVendedor(){         System.out.println("El documento ha sido firmado por el vendedor...");         this.firmaVendedor = true;     } }  public class EmpleadoRRHH {     private final Contrato con = new Contrato();     private final NDA nda = new NDA();      public EmpleadoRRHH() {     }      public Contrato getContrato(){         return (Contrato) con.copiar();     }      public NDA getNDA(){         return (NDA) nda.copiar();     }      public void firmar(Documento doc){         doc.firmarRRHH();     } }</pre> |
| Screenshots de funcionamiento | <pre>public static void main(String[] args) {     // TODO code application logic here     EmpleadoRRHH emp = new EmpleadoRRHH();     Vendedor ven = new Vendedor("Alejandro", "Garita");      Contrato con = emp.getContrato();     NDA nda = emp.getNDA();      emp.firmar(con);     emp.firmar(nda);     if(!con.aprobado())         System.out.println("El documento no ha sido aprobado");     ven.firmar(con);     ven.firmar(nda);     if(con.aprobado())         System.out.println("El documento ha sido aprobado"); }</pre>   |



|  |  |
|--|--|
|  | <div>El documento ha sido firmado por el empleado de RRHH...</div> <div>El documento ha sido firmado por el empleado de RRHH...</div> <div>El documento no ha sido aprobado</div> <div>El documento ha sido firmado por el vendedor...</div> <div>El documento ha sido firmado por el vendedor...</div> <div>El documento ha sido aprobado</div> |
|--|--|

---

**FIN DE LA PRUEBA**