

DATA

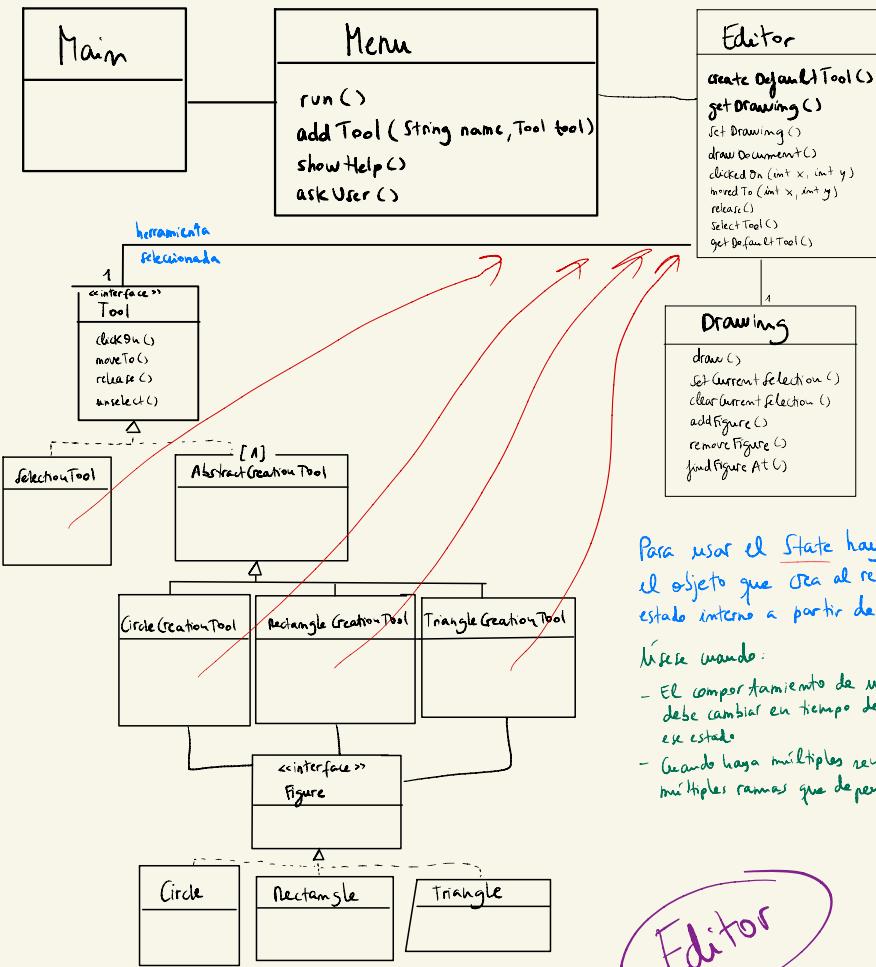
Anotaciones Varias



Eduardo Blasco Billra

el editor cambia su estado interno con cada interacción herramienta

STATE



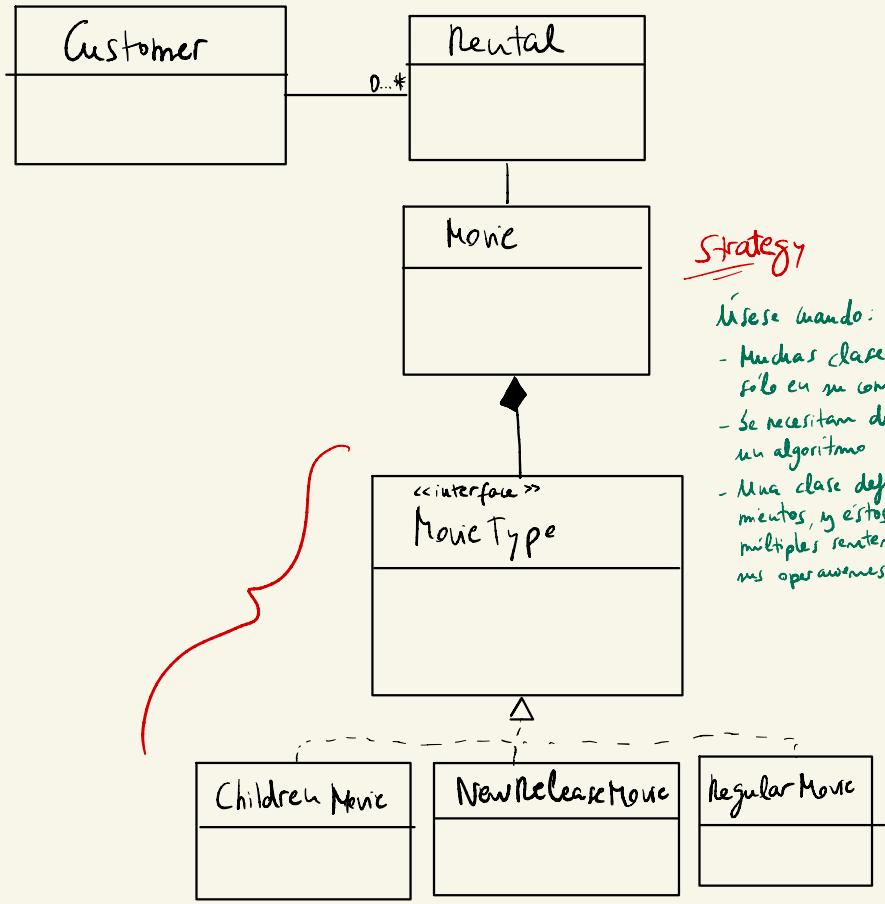
Para usar el State hay que mirar que el objeto que crea al resto modifique su estado interno a partir de estos objetos.

Nótese cuando:

- El comportamiento de un objeto depende de su estado y debe cambiar en tiempo de ejecución dependiendo de ese estado
- Cuando haya múltiples ramificaciones condicionales con múltiples ramas que dependen del estado del objeto.

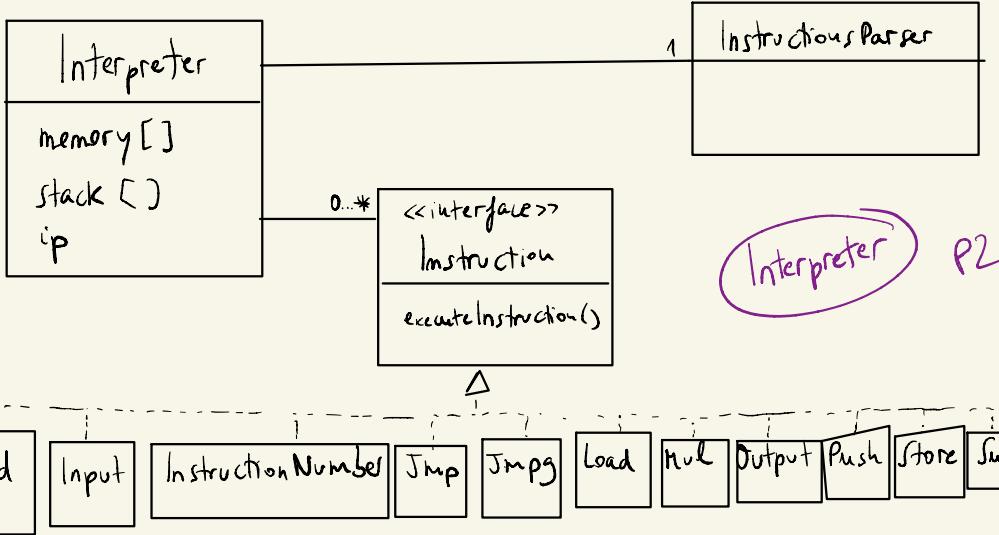
Editor

P3



Video Club

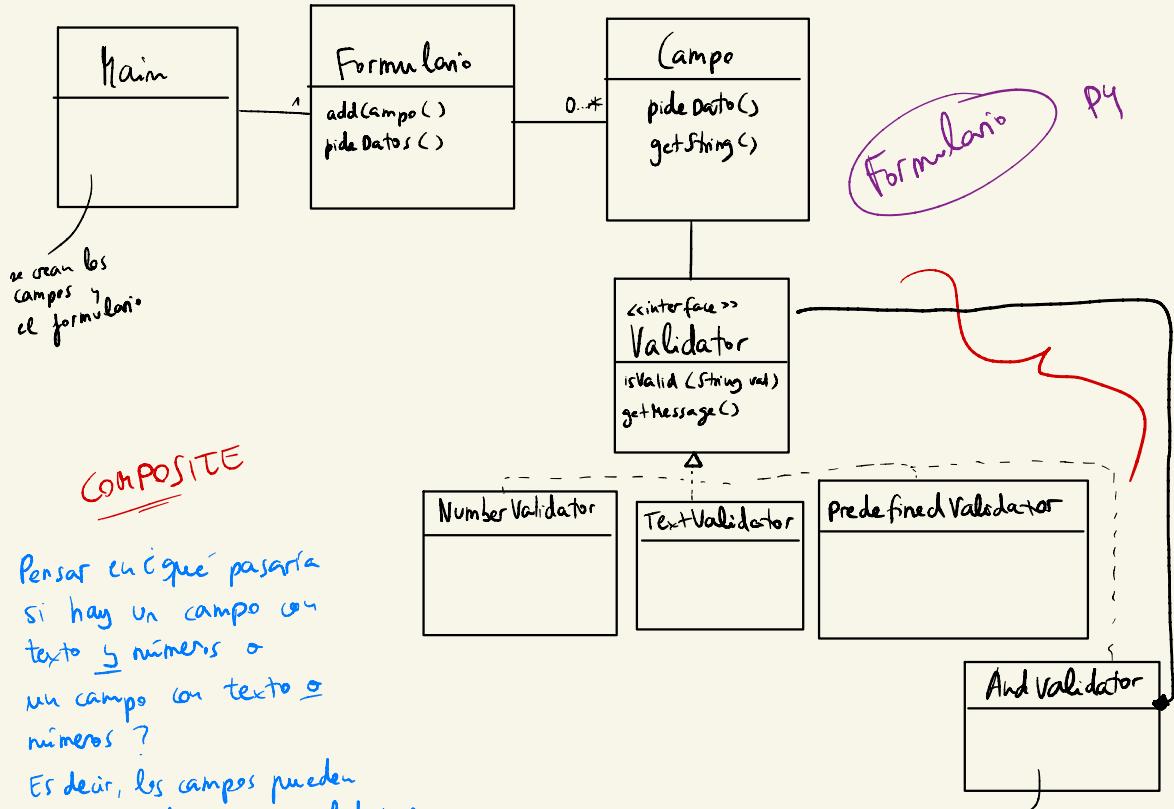
PD



Command

Máscara mando se quiere:

- Parametrizar objetos con una acción a realizar.
- Especificar, poner en cola y ejecutar peticiones en diferentes instantes de tiempo
- Permitir deshacer
- Permitir registrar los cambios para posibles caídas del sistema.

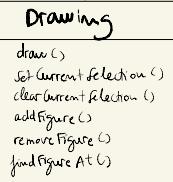
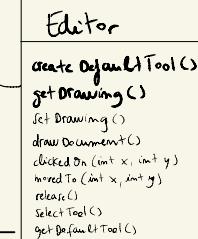


Usarse cuando:

- Se quiera representar jerarquías de parte-todo
 - Se quiera que los clientes puedan diferenciar entre composiciones de objetos y objetos individuales.
- Composto por
varios campos
TEXT AND NUMBER
TEXT OR NUMBER

el editor cambia su estado interno cada vez que la herramienta

STATE



Action Manager

```

executeAction(Action a)
clear() -> redoableActions.clear()
addUndoableAction()
canBeUndone()
canBeRedone()
undo()
redo()
    
```

<<interface>> Action

```

execute()
undo()
    
```

CreateFigure Action

```

figure, Drawing drawing
execute()
drawing.addFigure(figure)
undo()
drawing.removeFigure(figure)
    
```

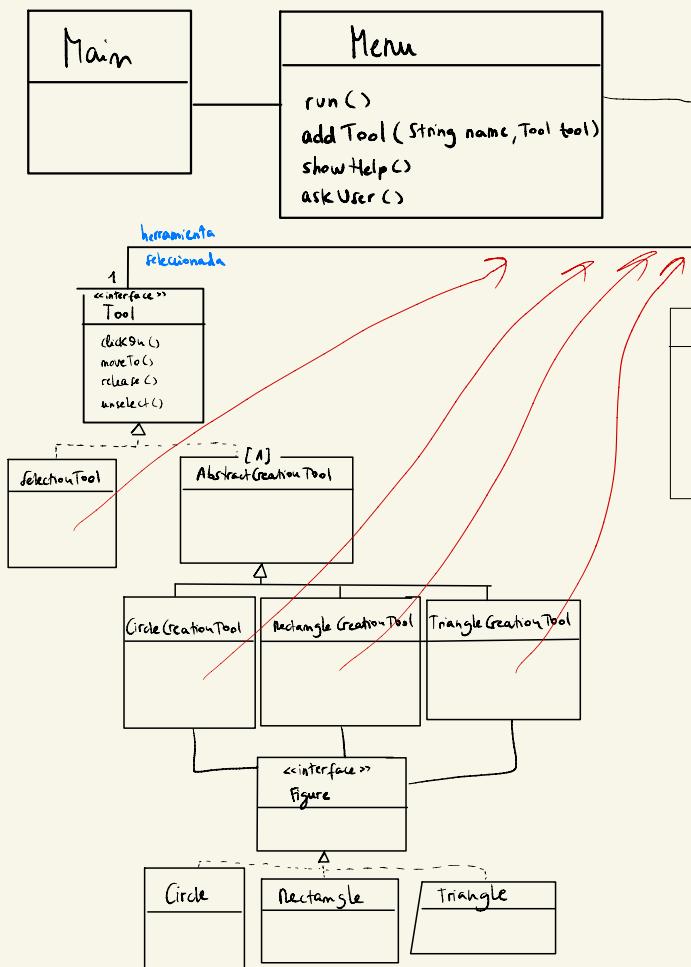
MoveFigureAction

```

figure, int x, int y
execute()
figure.moveTo(x, y)
undo()
figure.moveTo(-x, -y)
    
```

Nótese cuando se quiera:

- Parametrizar objetos con una acción a realizar
- Poner en lista, especificar y ejecutar acciones en distintos instantes de tiempo
- Permitir deshacer
- Permitir registrar los cambios



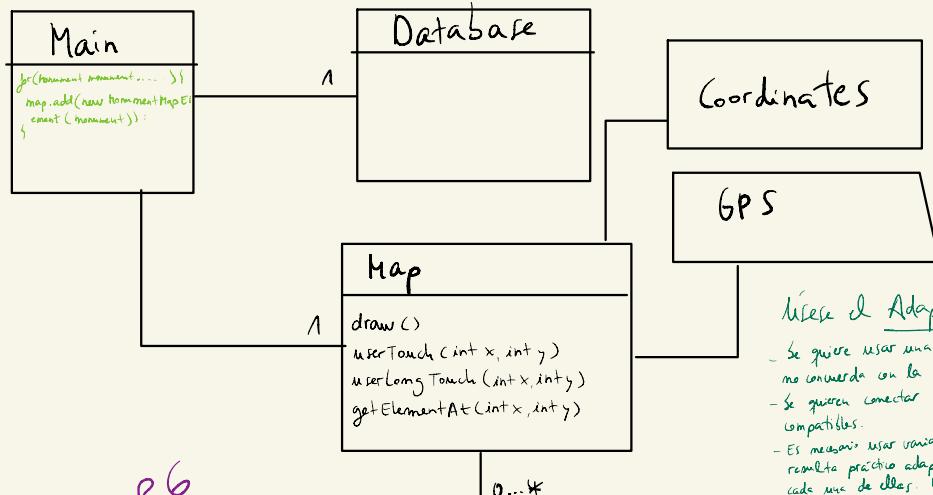
Para usar el State hay que mirar que el objeto que crea al resto modifique su estado interno a partir de estos objetos.

Nótese cuando:

- El comportamiento de un objeto depende de su estado y debe cambiar en tiempo de ejecución dependiendo de ese estado
- Cuando haya múltiples sentencias condicionales con múltiples ramas que dependen del estado del objeto.

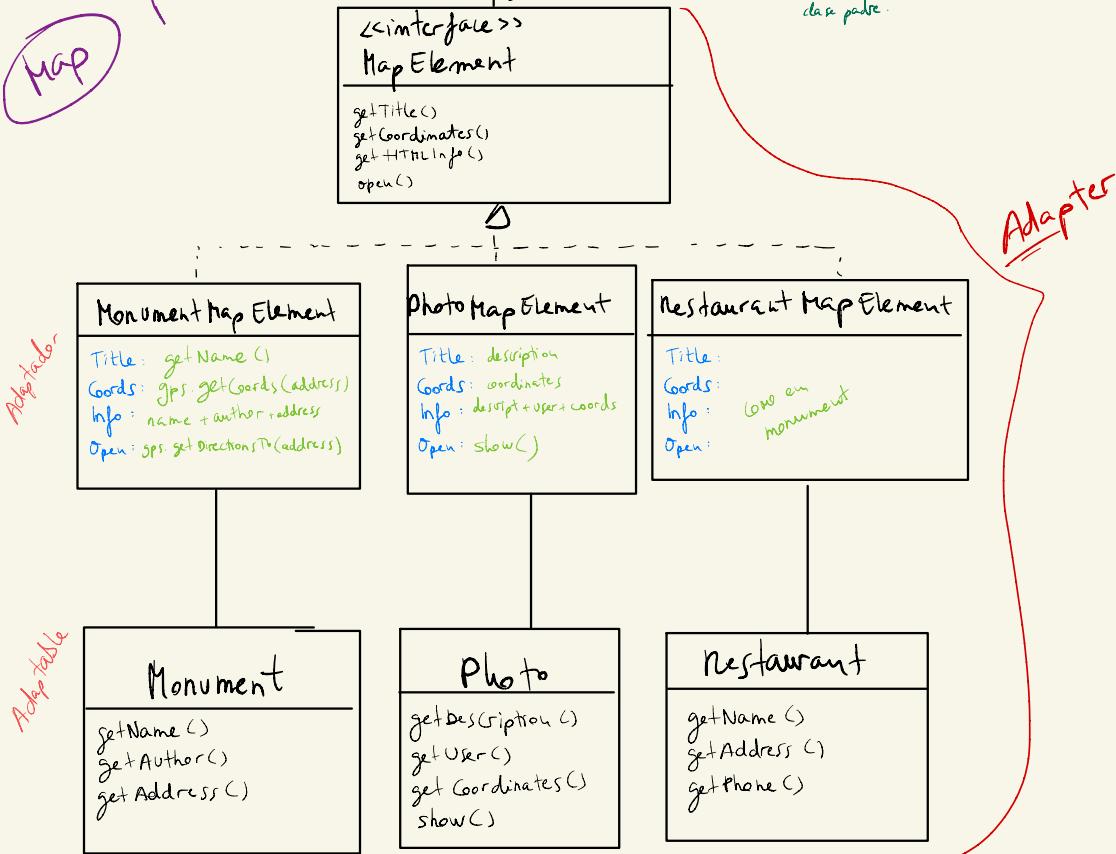
P5

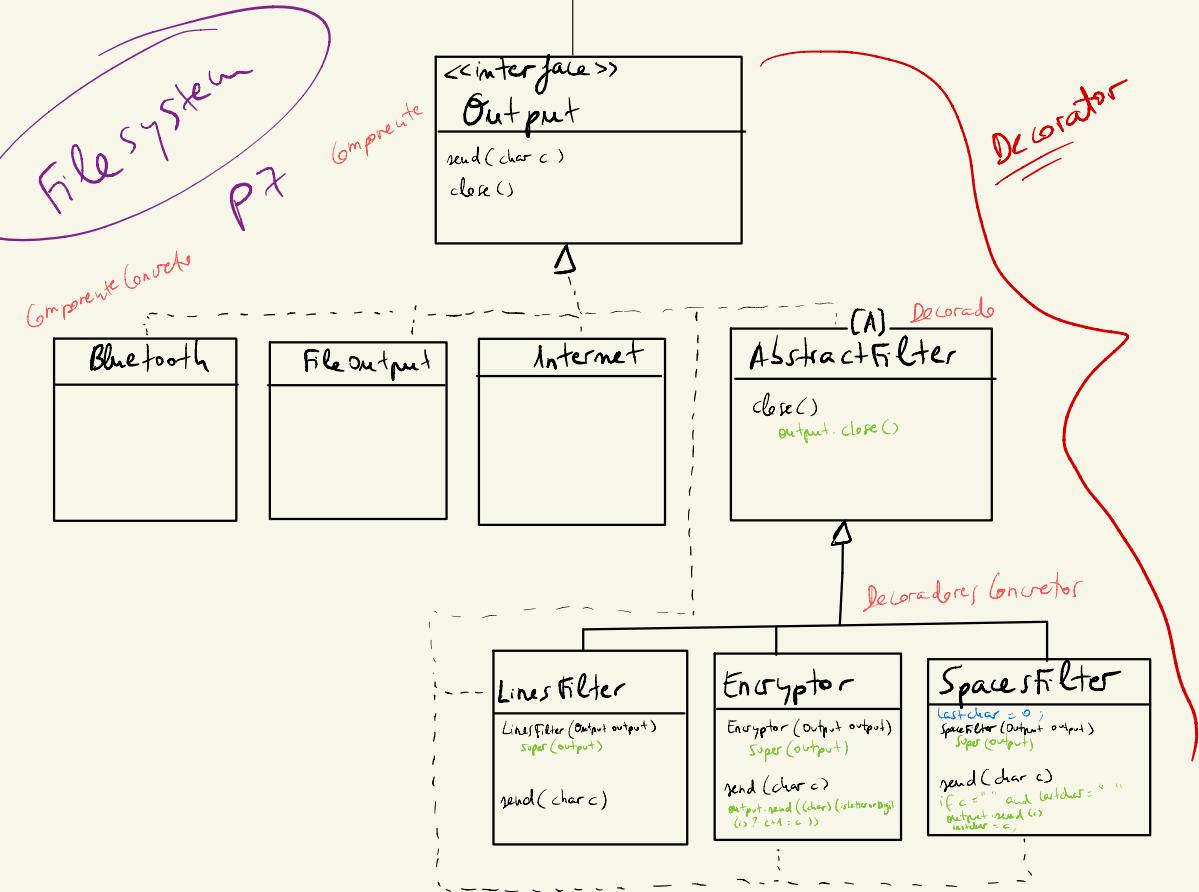
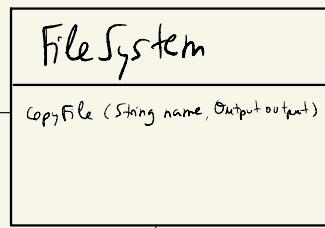
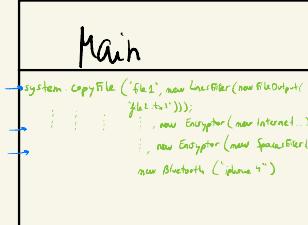
Editor



Se usa el Adapter cuando:

- Se quiere usar una clase existente y su interfaz no concuerda con la que necesita.
- Se quieren conectar clases que no tienen por qué ser compatibles.
- Es necesario usar varias subclases existentes, pero no resulta práctico adaptar su interfaz heredando de cada una de ellas. Puede adaptar la interfaz de un clase padre.





Usar el Decorador:

- Para añadir objetos individuales de forma dinámica y transparente, sin afectar a otros objetos.
- Para responsabilidades que pueden ser retiradas.
- Cuando la extensión mediante la herencia no es viable.

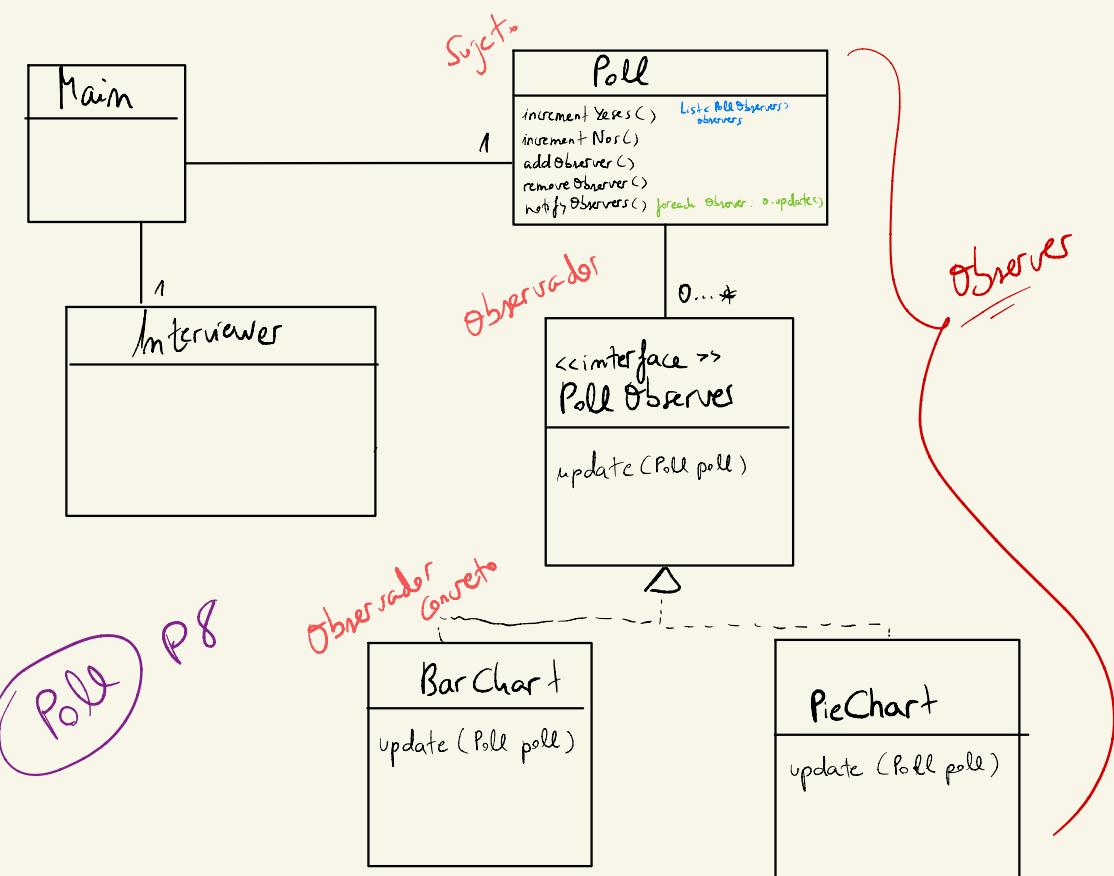
Funcionamiento
(como en Main)

Un Decorador

Un Decorador

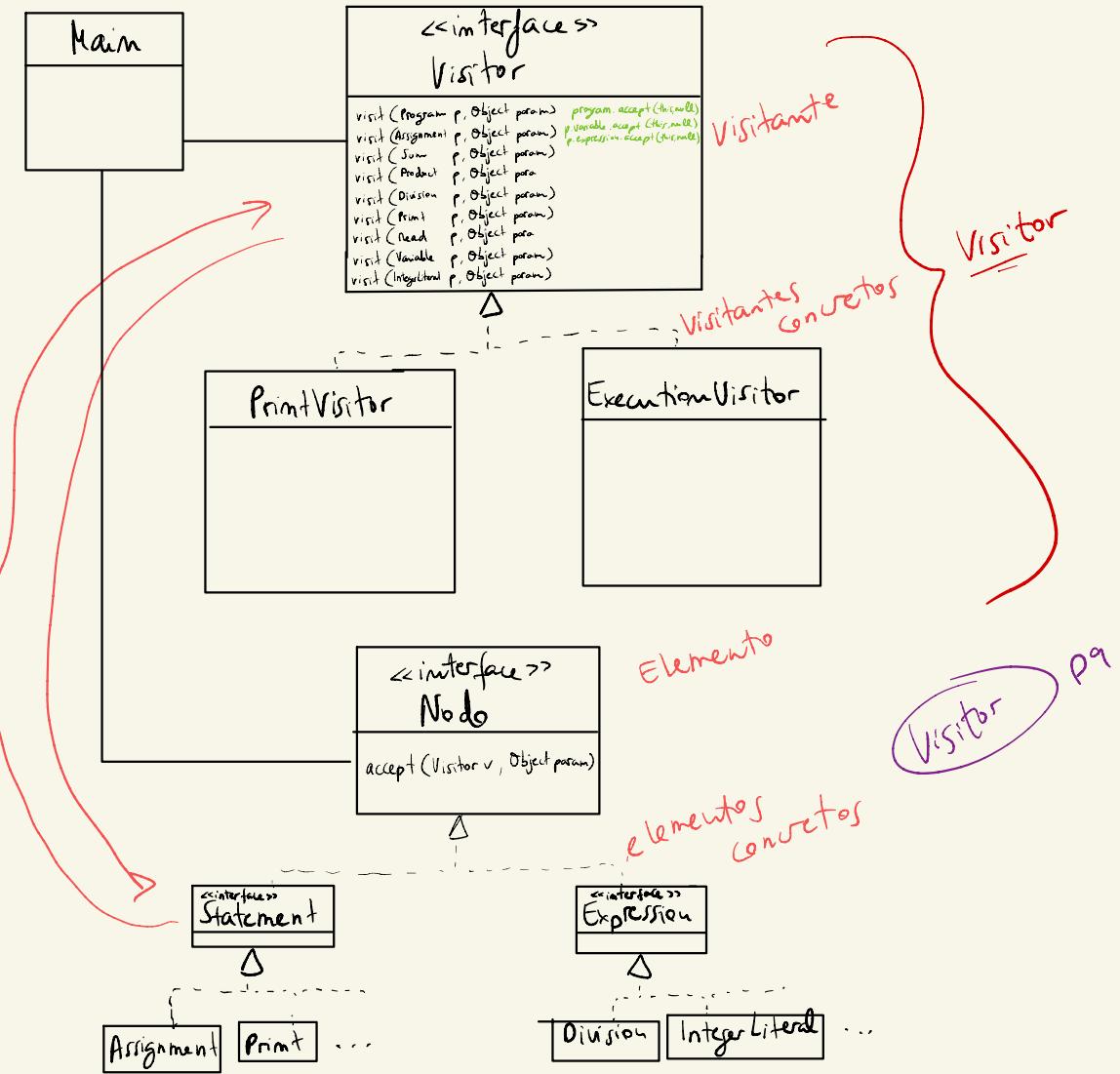
Un Componente

new Encryptor (new SpaceFilter (new Bluetooth (">")))



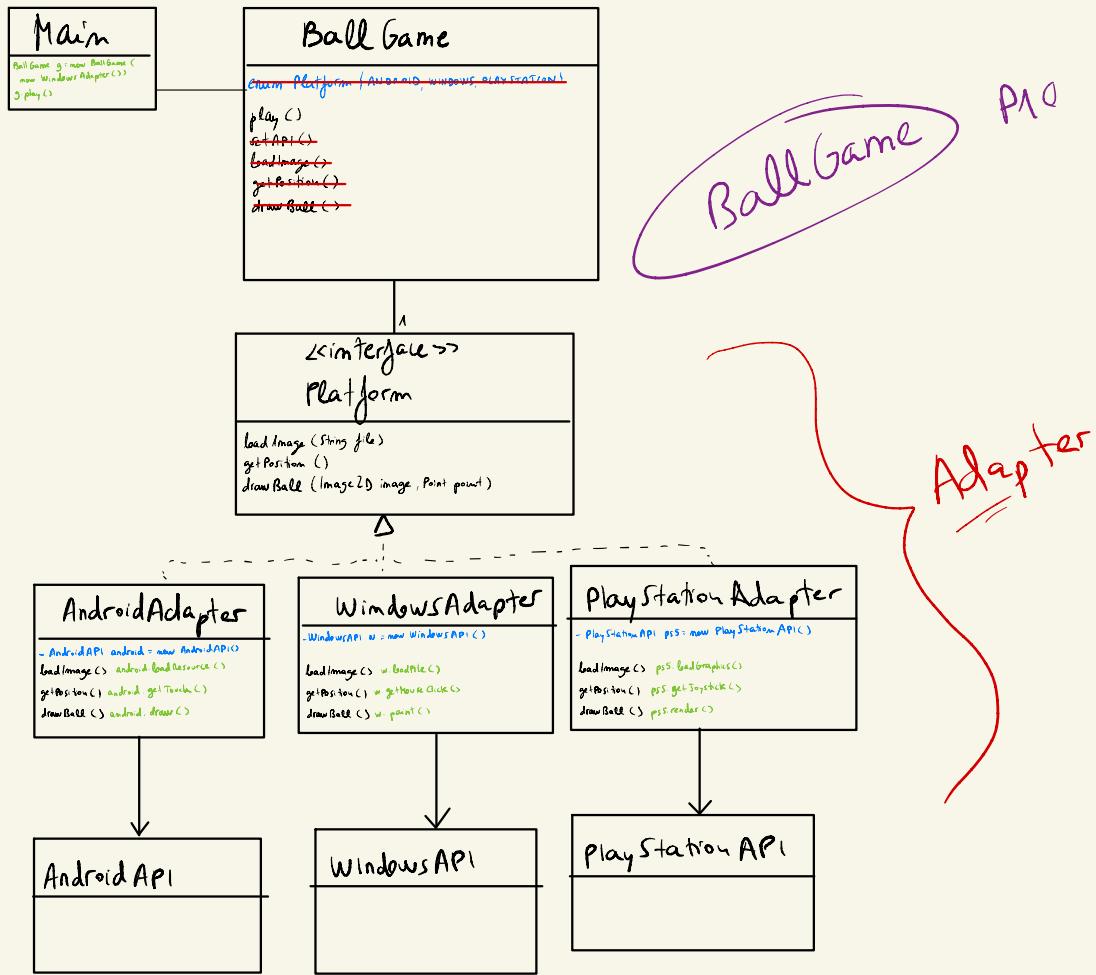
Usar el observer cuando:

- Cuando una abstracción tiene dos aspectos y uno depende del otro.
- Cuando un cambio en un objeto requiere cambiar otros, y no sabe cuántos objetos necesitan cambiarse.
- Cuando un objeto debería ser capaz de notificar a otros sin hacer imposiciones sobre quienes son dichos objetos.



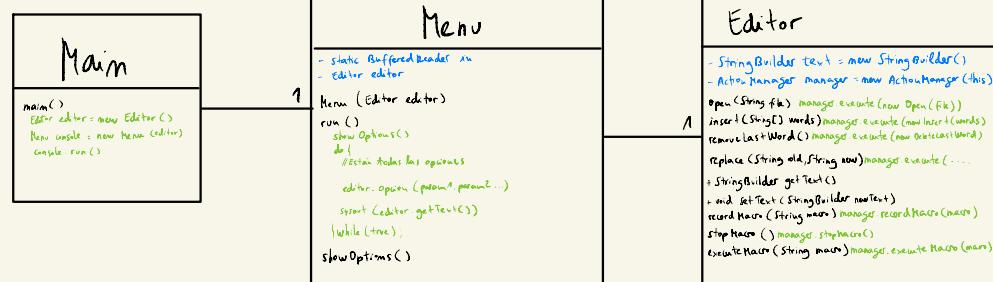
Usarse el patrón visitor cuando:

- Una estructura de objetos contiene muchas clases de objetos con diferentes interfaces y queremos realizar operaciones sobre esos elementos.
- Se necesitan realizar muchas operaciones distintas y no relacionadas sobre objetos de una estructura de objetos. Visitor permite juntar operaciones relacionadas definiéndolas en una clase.

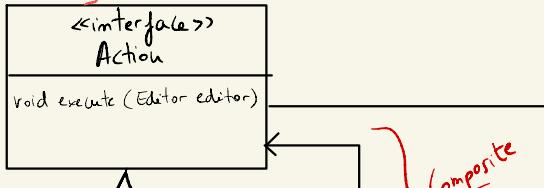


Usar el Adapter cuando:

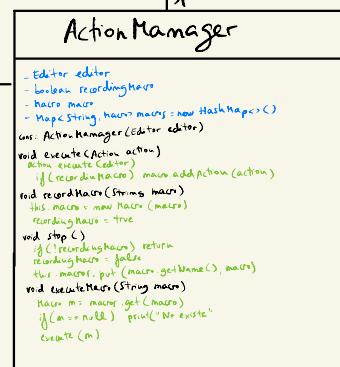
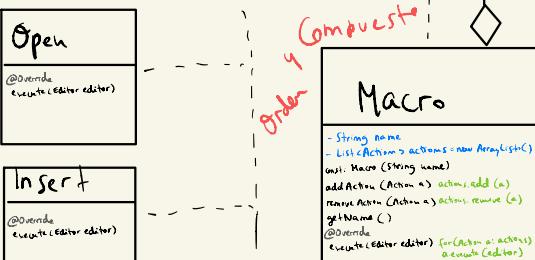
- Se quiere usar una clase existente y su interfaz no convenga con la que necesita
- Se quiere crear una clase no reutilizable que coopere con clases no relacionadas o que no han sido previstas, es decir, que no tienen por qué tener interfaces compatibles.
- Si es necesario usar varias subclases existentes pero no es práctico adaptar su interfaz heredando de cada una de ellas.



Orden y Componente



Órdenes y hojas



Patrones usados:

• Command

Participantes

1 Orden → Action

Métodos:

- execute → execute (Editor editor)

2 Orden Concreta → Open, Insert, Replace, DeleteLastWord, Macro

Métodos

- execute

- En Macro → addAction (Action a)

• Composite

Participantes

1 Componente → Action

Métodos

- Operación → execute (Editor editor)

2 Hoja → Open, Insert, Replace, DeleteLastWord

Métodos

- Operación → execute (Editor editor)

3 Composito → Macro

Métodos

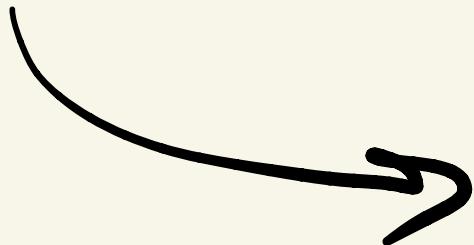
- Operación → execute ()

- addAction (Componente) → addAction (Action a)

P12

Ejemplo Examen

Revisitar
de las
prácticas



Customer

- String name
- List< Rental > rentals
- cons: Customer (String name)
- + void addRental
- + String getFname()
- + String statement()
 - calcula el precio para cada alquiler en función del tipo de película que sea.
- Para cada alquiler se suma su costo y se le resta una deducción por el número de días alquiler y así sucesivamente.
- Finalmente sumamos el total del alquiler y los freguentes para un resultado final.
- getTotalAmount()
- getTotalFrequentRenterPoints()

Rental

- Movie movie
- int daysRented
- const: Rental (Movie m, int days)
- + int getDaysRented()
- + Movie getMovie()
- + int getAmount()
 - return movie.getRentalPrice(daysRented)
- + int getFrequentRenterPoints()
 - return movie.getFrequentRenterPoints() + (daysRented * 2)

P1 Videoclub

Usa un Strategy

0...*

Movie

- static final int CHILDREN = 2 ==> ChildrenMovieType()
- static final int NEW_RELEASE = 0 ==> NewReleaseMovieType()
- static final int REGULAR = 1 ==> RegularMovieType()
- String title
- int priceCode
 - NewRelease type: NewRelease MovieType
 - title: Movie (String title, int priceCode)
 - + getAmount()
 - return priceCode * daysRented
 - + getFrequentRenterPoints()
 - return 2 * daysRented
 - + getMovieType()
 - return type.getMovieType(title, daysRented)

Contexto

MovieType

- + getAmount (int daysRented)
- + getFrequentRenterPoints (int daysRented)

Estrategia

Estrategia A

RegularMovieType

Realiza su implementación correspondiente

Estrategia B

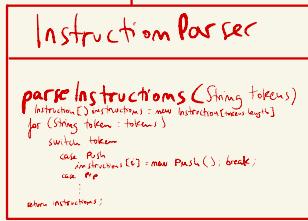
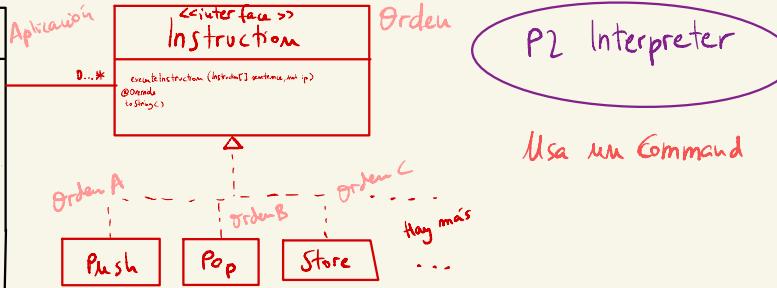
NewReleaseMovieType

Realiza su implementación correspondiente

Estrategia C

ChildrenMovieType

Realiza su implementación correspondiente



Main

```
main ()  
Formulario f = new Formulario()  
f.addCampo(new Campo("text", "Nombre"))  
f.addCampo(new Campo("text", "Apellidos"))  
...  
Validator producto = new ValidatorLister(CampoNombre, nombreValidator);  
Validator precio = new Campo("text", "Precio");  
Validator cantidad = new Campo("text", "Cantidad");  
f.pideDatos();
```

Formulario

```
-List<Campo> campos = new ArrayList<Campo>()  
+addCampo (Campo campo)  
+pideDatos()  
for (Campo c : campos)  
c.pideDatos();  
print (c.getStrong());
```

Py Formulario

Usa un Composite

Campo

```
+String finalValidator: null : new ValidatorLister()  
+String finalValidatorWhere: new ValidatorLister()  
+String regular  
+String regex  
+Validator nombre  
+String tipo (String tipo, String texto)  
+void pideDatos()  
+String[] finalValidatorWhere : new ValidatorLister()  
+String regular  
+String regex  
+Validator nombre  
+String tipo (String tipo, String texto)  
+void pideDatos()  
+String[] finalValidatorWhere : new ValidatorLister()  
+String regular  
+String regex  
+Validator nombre  
+String tipo (String tipo, String texto)  
+void pideDatos()  
+String[] finalValidatorWhere : new ValidatorLister()  
+String regular  
+String regex  
+Validator nombre  
+String tipo (String tipo, String texto)
```

Validator

```
boolean isValid (String value)  
getMensaje ()
```

Componente

Hoja

CampoText TextValidator

```
boolean isValid (String value)  
for (char c : value.toCharArray())  
if (Character.isLetter(c))  
return true  
else  
return false
```

CampoNúmero NumberValidator

```
Cambia los  
implementación
```

CampoPredefinido PredefinedValidator

Cambia los
implementación

Compuesto

[A] Compounded Validator

```
list<Validators validators = new ArrayList<Validators>()  
list<Validators> Validator (Validators ... validators)  
as Validators merge (Validators validators)  
add (Validators validators)  
getvalidators()  
getMensaje()  
getMensaje (String getMensaje())
```

no entiende Hoja

LessThanValidator
(new LengthValidator)

Hoja

And Validator
(new Validator (Validator ... validators))
+void add (Validator validator)
+void add (Validator validator)

Hoja

Or Validator
(new Validator (Validator ... validators))
+void add (Validator validator)
+void add (Validator validator)

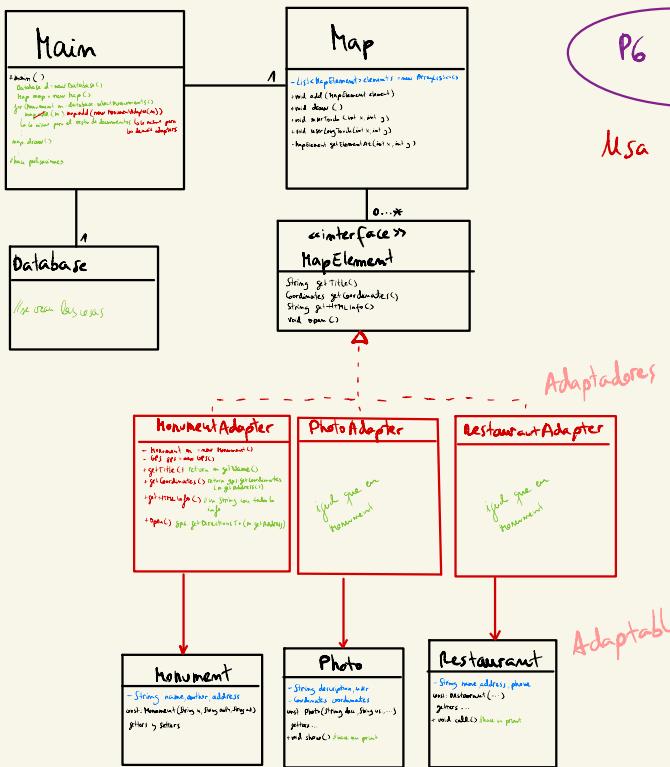
no entiende Hoja

LengthValidator

```
+int min  
+int max  
+String regular (int min, int max)  
+String regular (int min, int max)  
+String regular (int min, int max)
```

GreaterThanOrEqualToValidator

```
+int min  
+String regular (int min)  
+String regular (int min)  
+String regular (int min)
```

P6 Maps

Usa un Adapter

Main

```

    > main()
    FileSystem file = new FileSystem();
    file.writeFile("file name LineFilter (from FileSystem)", "out.txt");
    file.readFile("out.txt");
    file.appendFile("file name LineFilter (from FileSystem)", "out.txt");
    file.appendFile("file name LineFilter (from FileSystem)", "out.txt");
    file.appendFile("file name LineFilter (from FileSystem)", "out.txt");
    file.appendFile("file name LineFilter (from FileSystem)", "out.txt");

```

FileSystem

```

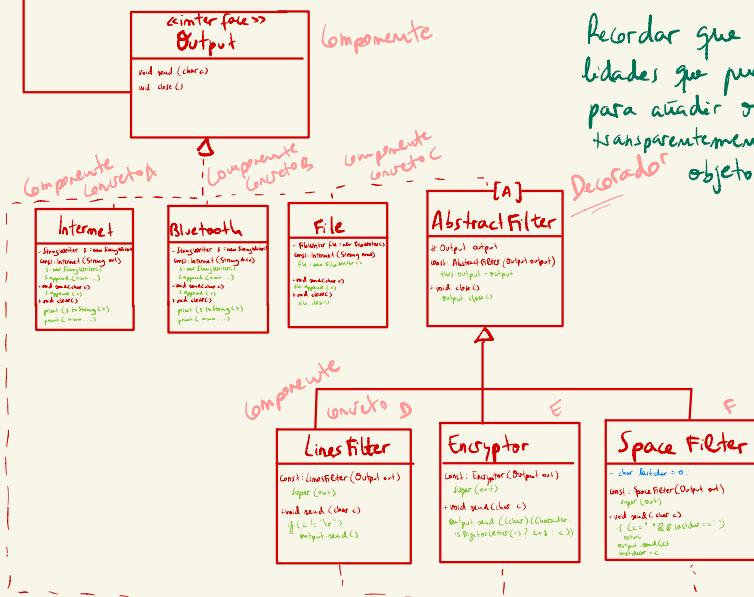
    -> writeFile (String name new LineFilter)
    -> readFile ("file name LineFilter (from FileSystem)")
    -> appendFile ("file name LineFilter (from FileSystem)", "out.txt")
    -> appendFile ("file name LineFilter (from FileSystem)", "out.txt");
    -> appendFile ("file name LineFilter (from FileSystem)", "out.txt");
    -> appendFile ("file name LineFilter (from FileSystem)", "out.txt");
    -> appendFile ("file name LineFilter (from FileSystem)", "out.txt");

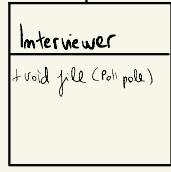
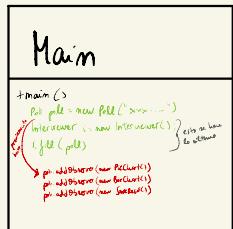
```

P7 FileSystem

Usa un decorador

Recordar que se usa para responsabilidades que pueden ser retiradas y para añadir objetos dinámicamente transparentemente (sin afectar a otros objetos)

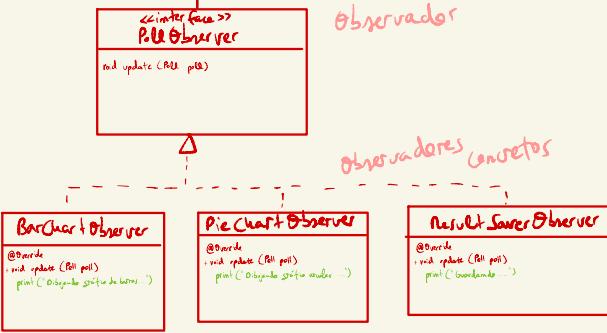


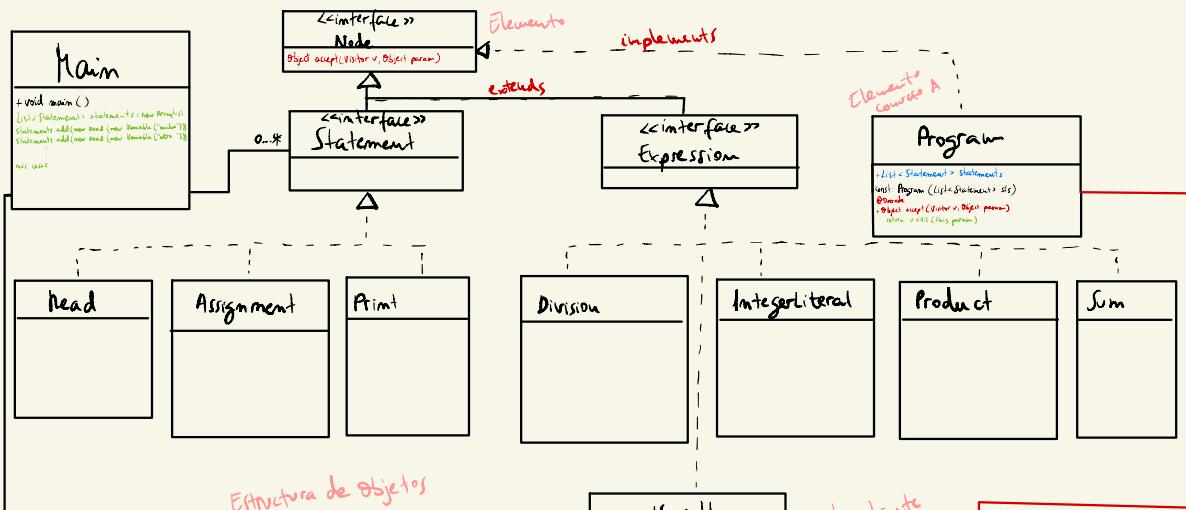


Sujeto
Pf Poll System

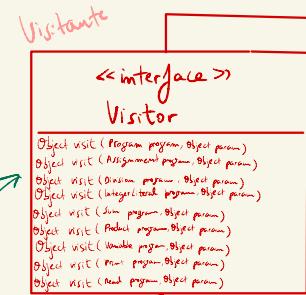
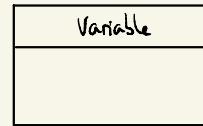
Usa un Observer

Observer





Estructura de objetos



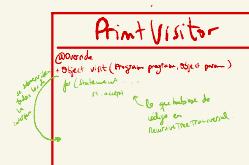
P9 Visitor

Usa el patrón Visitor

Es como una
simulación de
sobrecarga de
métodos (como
en C++)

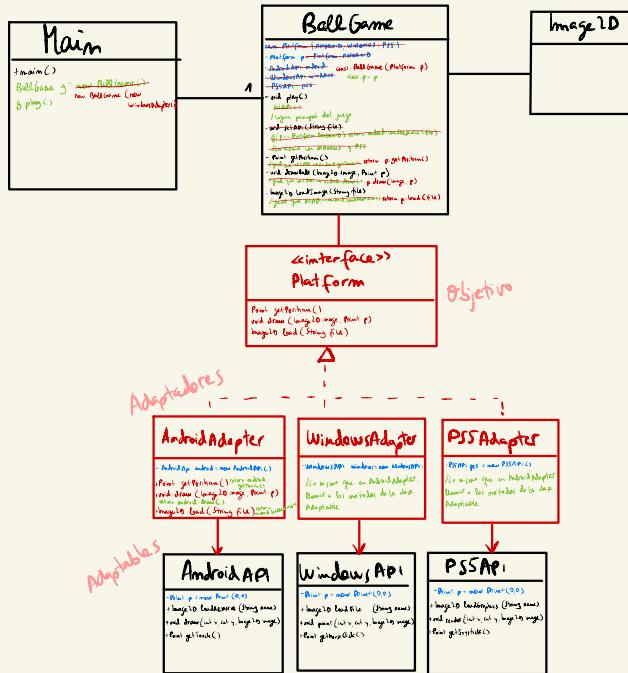
Visitor Concreto

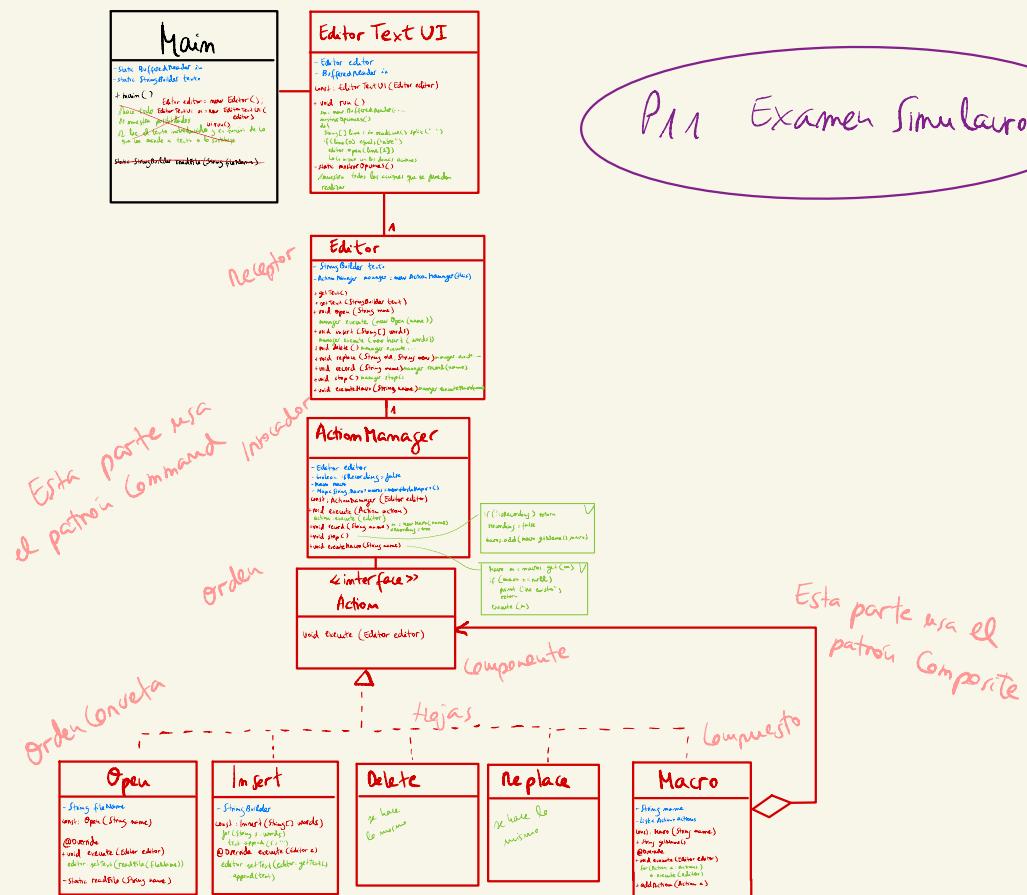
Visitor Concreto A
Visitor Concreto B



P10 Ball Game

Usa el patron Adapter





Main

```
main()
{
    Machine m = new Machine();
    m.addEvent(Event.Event("Event1", 10));
    m.run();
}
```

Machine

```
class Machine implements Runnable
{
    List<Event> events;
    int currentEventIndex;

    public Machine()
    {
        events = new ArrayList<Event>();
        currentEventIndex = 0;
    }

    void addEvent(Event event)
    {
        events.add(event);
    }

    void run()
    {
        while(true)
        {
            if(events.size() > 0)
            {
                Event currentEvent = events.get(0);
                System.out.println("Current event: " + currentEvent);
                currentEvent.execute();
                events.remove(0);
            }
            else
            {
                System.out.println("No events available");
            }
        }
    }

    void print()
    {
        for(Event event : events)
        {
            System.out.println(event);
        }
    }
}
```