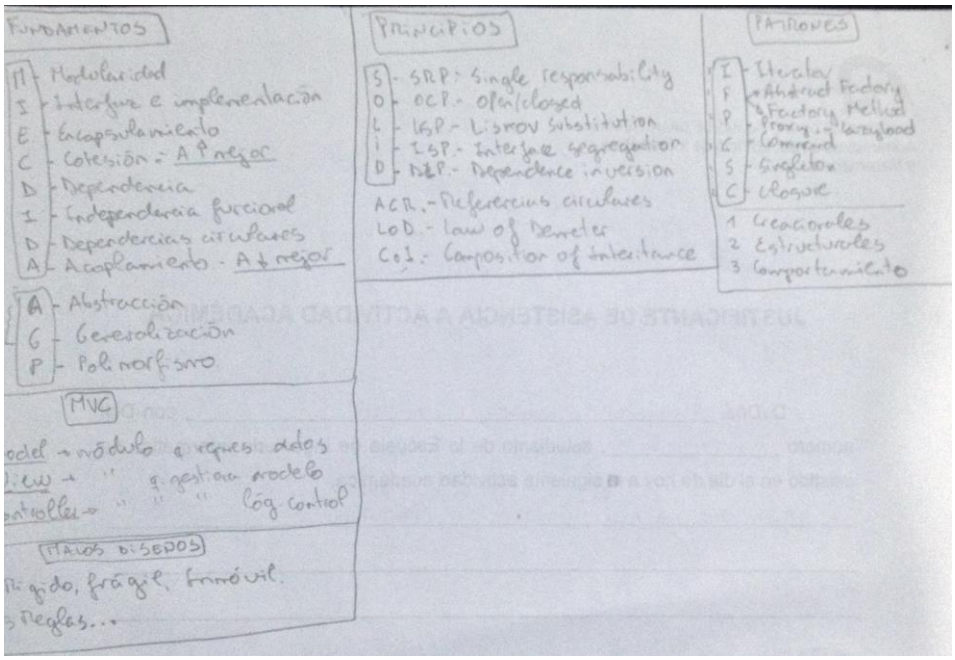
Resumen total IS2

Ingeniería del Software II (Universidad de Las Palmas de Gran Canaria)

IS2, la guía del aprobado



Tema 1.- Descomposición modular

La complejidad de un problema es mayor que la complejidad de cada una de sus partes por separado.

Modularidad

La modularidad es una estrategia de desarrollo de software en la que los componentes o módulos son recombinables para construir diferentes aplicaciones de una misma familia.

La **modularidad** es también la propiedad de los entornos de programación que permite subdividir una aplicación en partes más pequeñas, cada una de las cuales debe ser tan independiente como sea posible de las restantes partes. Pensar en piezas de Lego.

Módulo: Un **módulo** es cada una de las partes de un programa que resuelve uno de los subproblemas en que se divide el problema original. Cada módulo tiene una tarea bien definida para la que puede necesitar a otros módulos.

Interfaz e implementación

Un módulo actúa como una caja negra para lo que se necesita diferenciar la parte externa, interfaz, de la interna, implementación.

Encapsulamiento

Los módulos deben especificarse y diseñarse de tal forma que la información contenida en su interior sea inaccesible a otros módulos que no la necesiten. Esto se consigue cuando un cliente de un módulo no es capaz de saber más de lo que hay en la interfaz.

Cohesión

Hace referencia a la forma en que agrupamos unidades de software (módulos, subrutinas...) en una unidad mayor. Por ejemplo: la forma en que se agrupan funciones en una biblioteca de funciones o la forma en que se agrupan métodos en una clase, etc. Un módulo cohesivo ejecuta una tarea sencilla de un procedimiento de software. A mayor cohesión, mejor. El módulo será más sencillo de diseñar, programar y probar. **Todos los elementos que constituyen un módulo están relacionados.**

Comentado [1]: El significado se me hacía corto y decidí añadir un poco más. Me encontré con esta web (http://www.alegsa.com.ar/) y tiene un diccionario de informática buenísimo.

Comentado [2]: Vamos, que todos los métodos que trabaje el módulo han de centrarse en la idea general del módulo, cohesión a nivel de módulo

Dependencia

Es un módulo requerido por otro módulo para poder funcionar correctamente

A depende de B
A es cliente B
B actúa como servidor de A
A es superior de B
B es subordinado de A

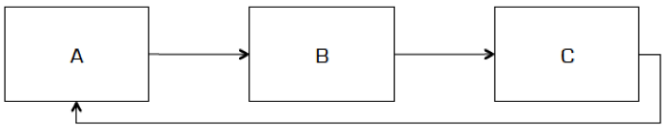
Independencia funcional

Evitar la excesiva interacción con otros módulos. Cada módulo se debe centrar en una función específica de los requisitos y tener una interfaz sencilla, cuando se ve desde otras partes de la estructura del software.

Dependencias circulares

En general son nocivas ya que:

- Se reduce o incluso hace imposible la prueba o la reutilización de un módulo
- Un cambio local en un módulo se puede extender a otros módulos
- Pueden causar recursiones infinitas y/o filtraciones de memoria



Acoplamiento

Grado de interdependencia entre las unidades de software (módulos, funciones, subrutinas, bibliotecas, etc.) de un sistema informático. El acoplamiento da la idea de lo dependiente que son las unidades de software entre sí, es decir, el grado en que una unidad puede funcionar sin recurrir a otras.

El acoplamiento mide la interconexión entre los módulos de una estructura de programa. A menor acoplamiento, mejor. Se reduce la propagación de errores y se fomenta la reutilización de los módulos.

Criterios de un buen diseño

- Evitar el código duplicado
- Minimizar el acoplamiento y maximizar la cohesión
- Conseguir la ortogonalidad: módulos no relacionados se deben poder alterar de forma independiente
- Mantener los módulos que necesiten modificarse a la vez lo más juntos posible dentro del código.

Tema 2.- Abstracción y polimorfismo

La abstracción es un fundamento del software por el cual se aísla toda aquella información que no resulta relevante a un determinado nivel de conocimiento. **Consiste en captar las características esenciales de un objeto, así como su comportamiento**.

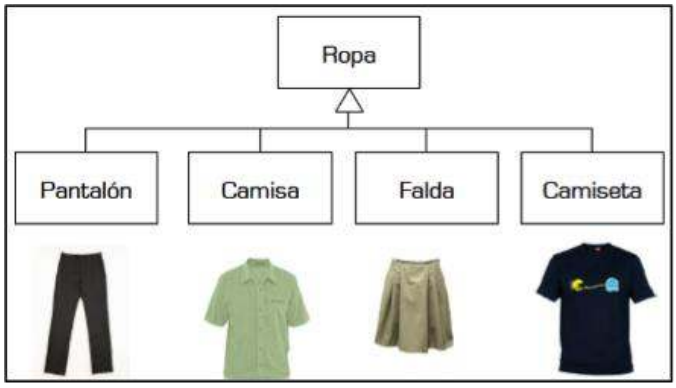
Comentado [3]: Esta relacionado con la herencia. Se puede usar si podemos decir "A es un B".
Si fuese "A contiene un B" hablamos de composición

La abstracción está relacionada con la formación de conceptos:

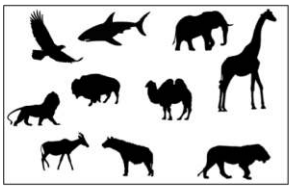
- Los conceptos se forman mediante el reconocimiento de similitudes entre objetos.
- El progreso en la formación de conceptos va de lo particular a lo general.

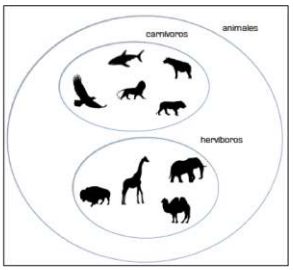
Generalización

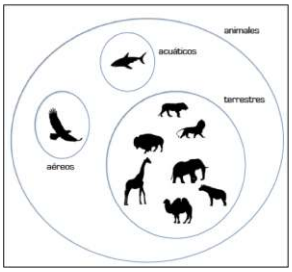
Los conceptos se adquieren por generalización. Los conceptos concretos son primarios, ya que constituyen la base para la adquisición de conceptos más abstractos.



Representación de conceptos

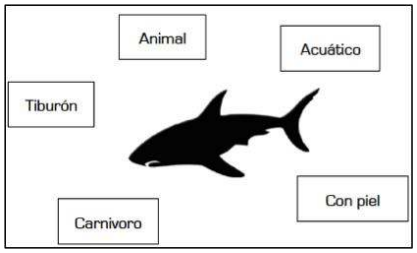






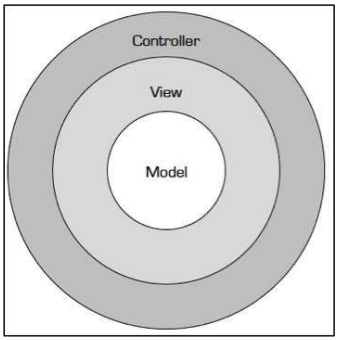
Polimorfismo

La palabra polimorfismo proviene del griego y significa que posee varias formas diferentes. **Está relacionada con la jerarquía de clases y permite que objetos diferentes sean tratados con la misma interfaz**.



Tema 3.- MVC

Es un patrón de arquitectura o **estilo arquitectónico**, no confundir con patrón de diseño.



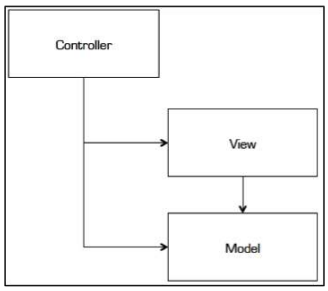
Responsabilidades

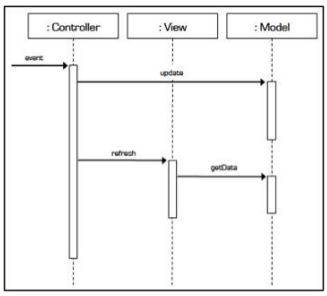
Es necesario distinguir los módulos entre sí de forma que existan responsabilidades bien diferenciadas.

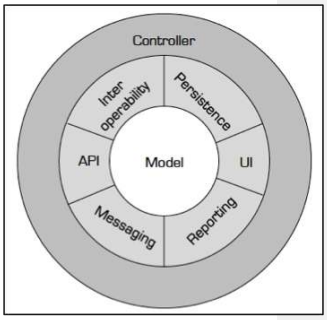
- *Model* -> Módulos que representan los datos.
- *View* -> Módulos que gestionan el modelo con otros sistemas/usuario.
- *Controller* -> Módulos que gestionan la lógica de control.

Dependencias permitidas

Siempre del exterior al interior. No a los lados (tachas entre módulos **no**) ni del centro al exterior.







Tema 4.- Dependencias

Malos diseños

- *Rígido*. Es difícil hacer un cambio ya que todos los módulos están muy imbricados.
- *Frágil*. Cuando se realiza un cambio, otras partes del sistema dejan de funcionar.
- *Inmóvil*. Es difícil reusar el código porque todo está muy ligado a la aplicación actual.

Comentado [4]: ¿Y esto que significa? jajaja

Comentado [5]: según el diccionario: imbricado, da

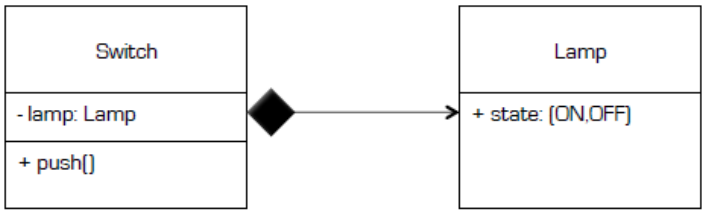
adj. biol. [Hoja, semilla o escama] superpuesta parcialmente a otra del mismo tipo:
la merluza tiene las escamas imbricadas.

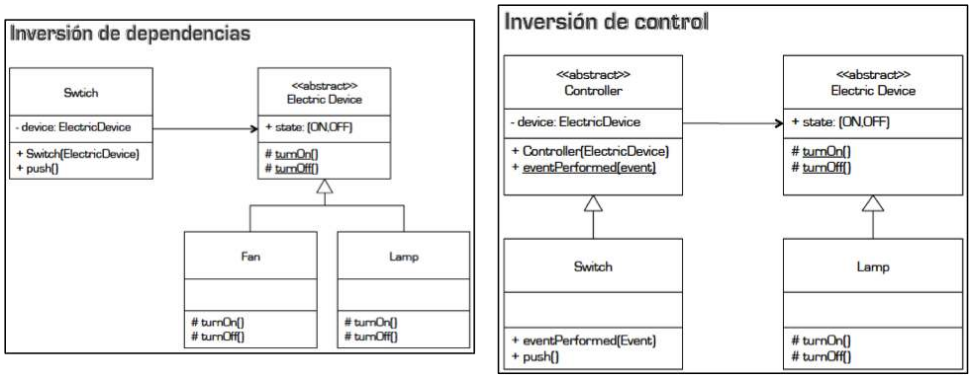
zool. [Concha] de superficie ondulada:
las vieiras son moluscos de concha imbricada.

Comentado [6]: Sí, también tuve que buscarlo xD

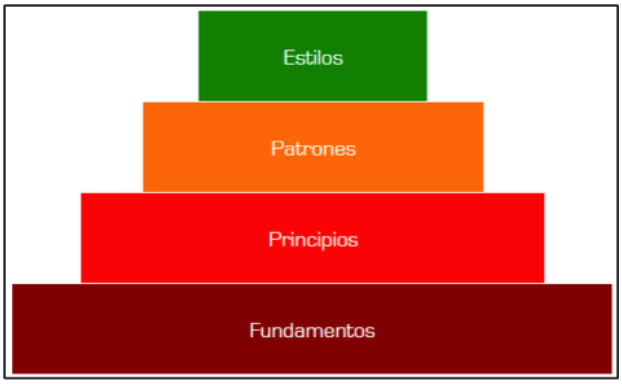
Reglas

1. Los módulos de alto nivel no deben instanciar sus componentes.
2. Las clases de alto nivel no deben depender de los componentes que implementen los detalles.
3. Las abstracciones no deben depender de los detalles sino los detalles de las abstracciones





Tema 6.- Fundamentos, principios y patrones



Fundamentos

- Modularidad
- Abstracción
- Acoplamiento
- Cohesión

Principios (SOLID)

Establece los cinco principios básicos de la programación orientada a objetos y diseño. Este acrónimo tiene bastante relación con los patrones de diseño, en especial, con la alta cohesión y el bajo acoplamiento.

El objetivo de tener un buen diseño de programación es abarcar la fase de mantenimiento de una manera más legible y sencilla, así como conseguir crear nuevas funcionalidades sin tener que modificar en gran medida código antiguo.

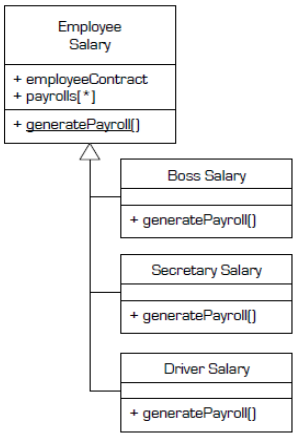
S-Responsabilidad simple (Single responsibility) [SRP]

Este principio trata de destinar cada clase a una finalidad sencilla y concreta.

O-Abierto/Cerrado (Open/Closed) [OCP]

Principio que consiste en que el diseño debe ser abierto para poderse extender pero cerrado para poderse modificar. (pero cerrado a las modificaciones)

mirar aquí:



Las clases que cumplen con OCP tienen dos características:

1. Son abiertas para la extensión; es decir, que la lógica o el comportamiento de esas clases puede ser extendida en nuevas clases.
2. Son cerradas para la modificación, y por tanto el código fuente de dichas clases debería permanecer inalterado.

Podría parecer que ambas características son incompatibles, pero eso no es así.

Comentado [7]: http://jms32.eresmas.net/web2008/documentos/informatica/documentacion/logica/OOP/Principios/Oop_Solid_OCP/2012_09_04_SOLID_PrincipioAbiertoCerrado.html#Refh1_OCP_Principio_Abierto_Cerrado

L-Sustitucion Liskov (Liskov substitution) [LSP]

Este principio habla de la importancia de crear todas las clases derivadas para que también puedan ser tratadas como la propia clase base. Cuando creamos clases derivadas debemos asegurarnos de no reimplementar métodos que hagan que los métodos de la clase base no funcionasen si se tratasen como un objeto de esa clase base.

I-Segregación de Interfaces (Interface segregation) [ISP]

Este principio dice que cuando se definen interfaces estos deben ser específicos a una finalidad concreta. Por ello, si tenemos que definir una serie de métodos abstractos que debe utilizar una clase a través de interfaces, es preferible tener muchos interfaces que definan pocos métodos que tener una interfaz con muchos métodos.

D-Inversión de dependencias (Dependency inversion) [DIP]

El objetivo de este principio es conseguir desacoplar las clases y el uso de abstracciones para conseguir que una clase interactúe con otras clases sin que las conozca directamente. Es decir, las clases de nivel superior no deben conocer las clases de nivel inferior. Dicho de otro modo, no debe conocer los detalles. Existen diferentes patrones como la inyección de dependencias o service locator que nos permiten invertir el control.

En Informática, *Inyección de Dependencias* (en inglés Dependency Injection, DI) es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase quien cree el objeto. El término fue acuñado por primera vez por Martin Fowler.

También:

- Avoid Circular References (ACR)
- Law of Demeter (LoD) -> La noción fundamental es que dado un objeto, éste debería asumir tan poco como sea posible sobre la estructura o propiedades de cualquier otro (incluyendo sus subcomponentes).
- Composition over Inheritance (CoI)

Comentado [8]: RESUMEN: Desacoplar las clases, usando la abstracción para que una clase de nivel superior interactúe con con otra de nivel inferior sin conocer sus detalles.

Comentado [9]: Es un PATRÓN, aunque está en la parte de principios

Comentado [10]: La inversión de dependencias es un principio de diseño,de hay su nombre "Dependency inversion principle".

Lo que si es un patrón es la inyección de dependencias, ya que es el modo en el que se programa.

SOLID es un principio de diseño.

Comentado [11]: Sisi, me refiero a eso. Fíjate que lo que está sombreado para el comentario es lo de Inyeccion de dependencia

Comentado [12]: No profundizamos mucho en clase con esto, no?

Comentado [13]: no

Patrones de diseño

Creacionales	Estructurales	Comportamiento
Singleton	Proxy	Command
Factory	Closure	Iterator
	Listener	

Patrón de diseño Iterador (Comportamiento)

Define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección. Algunos de los métodos que podemos definir en la interfaz Iterador son:

Primero(), Siguiente(), HayMas() y ElementoActual().

Ejemplo

En el MoneyCalculator creamos una clase llamada PrimeNumbersCollection que nos daba números primos mientras los fuéramos pidiendo. Para poder recorrer con un "for each" o mediante iteradores de forma externa dicha colección debemos hacer los siguiente:

1- Hacer que la clase implemente Iterable<T>
2- Crear un método público iterator()
3- En ese método debe devolver un nuevo Iterador e implementar los métodos abstractos
4- Rellenar los métodos hasNext(),next(),remove().

```
public class ImplementaIterable implements Iterable<T> {
    ...
    @Override
    public Iterator<T> iterator() {
       return new Iterator<T>() {
            @Override public boolean hasNext() {}
            @Override public T next() {}
            @Override public void remove() {}
       };
    }
}
```

Patrón de diseño Abstract Factory y Factory Method (Creacional)

Una **factoría** es un objeto que maneja la creación de otros objetos. Las **factorías** se utilizan cuando la creación de un objeto implica algo más que una simple instanciación. Consiste en utilizar una clase constructora abstracta con unos cuantos métodos definidos y otro(s) abstracto(s): el dedicado a la construcción de objetos de un subtipo de un tipo determinado.

Abstract Factory nos permite crear, mediante una interfaz, conjuntos o familias de objetos (denominados productos) que dependen mutuamente y todo esto sin especificar cual es el objeto concreto.

Factory Method es una simplificación del Abstract Factory, en la que la clase abstracta tiene métodos concretos que usan algunos de los abstractos;

Patrón de diseño listener (Estructural)

No lo vimos en clase.

Patrón Proxy (Estructural)

Es un patrón estructural que tiene como propósito proporcionar un subrogado o intermediario de un objeto para controlar su acceso. Se usa cuando se necesita una referencia a un objeto más flexible o sofisticada que un puntero.

LazyLoad

LazyLoad implementa el patrón Proxy, usado para aplazar la inicialización de un objeto hasta el punto en el que sea necesario su utilización. Contribuye a la eficiencia del programa sí está implementado correctamente. (La idea contraria es EagerLoading).

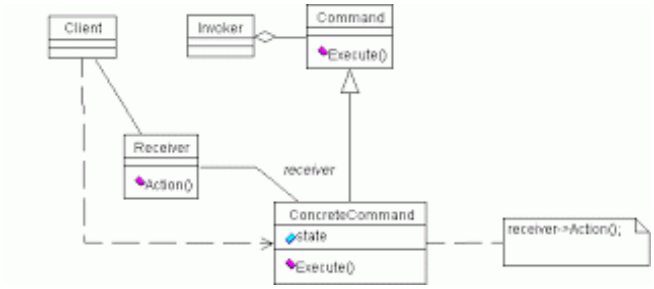
En el ImageBrowser, lo utilizamos como método para la carga de imágenes ya que se basa en ir cogiendo lo que se va necesitando y no cargar todo el contenido del golpe.

Patrón command (Comportamiento)

La intención del patrón es encapsular un comando en un objeto de tal forma que pueda ser almacenado, pasado a métodos y devuelto igual que cualquier otro objeto.

Se utiliza porque a veces se quiere poder enviar solicitudes a objetos sin conocer exactamente la operación solicitada ni el receptor de la solicitud. En general, un objeto botón o menú ejecuta solicitudes, pero la solicitud no está implementada dentro del mismo.

Ejemplo:



Patrón de diseño singleton (Creacional)

(Instancia única) Está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto.

Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

El patrón singleton se implementa creando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o

privado). Este patrón de diseño permite recorrer una estructura de datos sin que sea necesario conocer la estructura interna de la misma

1.- Se establece un constructor privado y se declara una variable private static de la clase
2.- Se hace un método getInstance(); Que devuelve o crea la instancia si no lo esta.

Ejemplo:

```
public class MySingleton{
        private static MySingleton instance;

        private MySingleton(){}

        public static MySingleton getInstance(){
                if(instance == null)
                        instance = new MySingleton();
                return instance;
        }
}
```

Patrón de diseño closure (Estructural)

Se usa para asociar una función con un conjunto de atributos o variable que persisten en la invocación de la función. Es un mecanismo fundamental en programación orientada a objetos para mejorar la modularidad de las aplicaciones. Se aplica este concepto a mejorar la modularidad de la aplicación del histograma. En este caso, el closure se define como una interface llamada AttributeExtractor con un método que debe implementarse como clase anónima para extraer el atributo que permite generar el histograma.

Comentado [14]: Me quedé igual que antes de leerlo.... :(

Comentado [15]: Este es el mas difícil

Ejemplo

```
public interface AttributeExtractor<E, A> {
   public A extract(E entity);
}

--- Main --

...builder.build(new AttributeExtractor<Mail, String>() {
        @Override
        public String extract(Mail entity) {
           return entity.getDomain();
        }
      });
```

```
--- HistogramBuilder --

public <A> Histogram build(AttributeExtractor<T, A> extractor) {
      Histogram<A> histogram = new Histogram<>();

      for (T item : items) {
         A attribute = extractor.extract(item);
         histogram.put(attribute, histogram.get(attribute) + 1);
      }
      return histogram;
   }
```

Estilos

Model View Controller

Descrito aquí.

Model View Presenter

Más información aquí.

Tema 7.- Responsabilidades

Se deberían diseñar clases con una sola responsabilidad para facilitar los cambios y aumentar las posibilidades de reutilización.

Diseño por contrato (Design by contract)

Los componentes se deben diseñar asumiendo que, cumpliéndose ciertas condiciones de entrada, se deben garantizar ciertas condiciones de salida, así como propiedades que se mantienen invariantes a pesar del procesamiento.

Anexo

Repaso UML, modelo de clase

Atributos y métodos

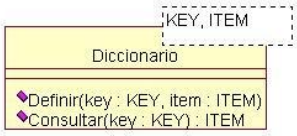
- Public, tiene como prefijo un '+'
- Private, tiene como prefijo un '-'

- Protected, tiene como prefijo un '#'

Los métodos abstractos van subrayados, indicando que se han de implementar en las clases que hereden.

Clase parametrizada

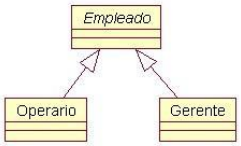
```
public Diccionario<Key,Item>{
        public void Definir(Key key, Item item){}
        public Item Consultar(Key key){
        /*
        //Aqui busca en el diccionario una key y
devuelve su item
        */p
        }
}
```



Clase abstracta

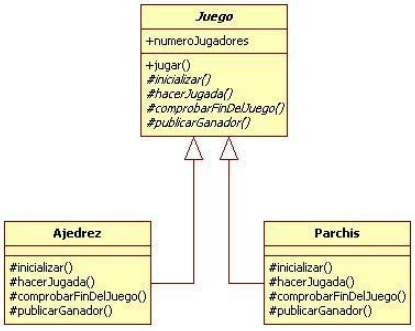
El nombre de la clase abstracta **se puede (normalmente no se hace)** especificar **subrayado,** con cursiva, entre <<>> o indicando que es abstract ({abstract}). Las líneas de las flechas (clases que heredan de la superior) son discontinuas sí se trata de una interfaz.

```
public abstract class Empleado{}
public class Operario extends Empleado{}
public class Gerente extends Empleado{}
```



MAS SOBRE ABSTRACT

```
public abstract class Juego{
        public int numeroJugadores;
        public juego(int jugadores){
                numeroJugadores=jugadores;
        }
```

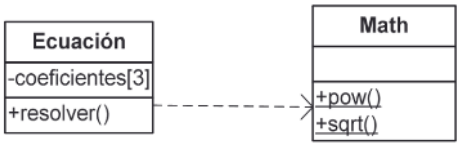


```
        public int getNumeroJugadores(){
                return numeroJugadores;
        }

        public void jugar(){//codigo}
        protected abstract void inicializar();
        protected abstract void hacerJugada();
        protected abstract void comprobarFinDelJuego();
        protected abstract void publicarGanador();
}

public class Ajedrez extends Juego{
        public Ajedrez (int jugadores){
                super(jugadores);
        }
        protected void inicializar(){}
        protected void hacerJugada(){}
        protected void comprobarFinDelJuego(){}
        protected void publicarGanador(){}
        /* Ejemplo : Cuando va @Override
        / si aquí nos hubiesen dicho que la clase ajedrez tiene un método jugar
        / llevaría override porque en el padre ya existe un método con código para
        / esta función . En los casos anteriores se trata de polimorfismo.
        /@Override
        / public void jugar(){}
        */
}
public class parchis extends Juego{/*igual que ajedrez*/}
```

Comentado [16]: Cierto, a pesar de que NetBeans marca warning si no pones override en métodos no implementados

Dependencia/instanciación de clase (uso)

Relación débil, muestra la relación entre un cliente y el proveedor de un servicio usado por el cliente.



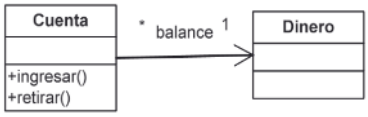

```
public class Ecuación{
        private [] int coeficientes;
        // cuidao que tiene q ser - de 4
        public Ecuacion(int [] coeficientes){
                this.coeficientes=coeficientes;
        }
        public [] int get(){return coeficientes}
        public resolver(){
        /*
        / Aquí es donde utilizara los métodos de Math. Ejemplo, Math.pow() o Math.sqrt()
        */
        }
}
public class Math{
        public static double pow(){}
        public static double sqrt(){}
        // es static por sentido común… con una dependencia Ecuacion usa a Math
        // el tipo es double porque no esta especificado
}
```

Asociación

Permite asociar objetos que colaboran entre sí. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro. Pueden ser bidireccionales o unidireccionales.

```
public class Cliente{
```

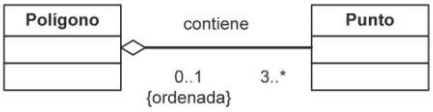
```
        private List OrdenCompra; // es una lista por la multiplidad
}// faltan getters
public class OrdenCompra{
        private Cliente cliente;
}// faltan getters
```




```
public class Cuenta {
        private Dinero dinero;
        public void Ingresar (){} // es void por intuición (no lo especifica) [DEBERIA poner → + ingresar() : Void]
        public Dinero retirar (){} // es Dinero por intuición (no lo especifica) [DEBERIA poner → +retirar() : Dinero]
        public Dinero getDinero(){return dinero;}
}
```

Agregación

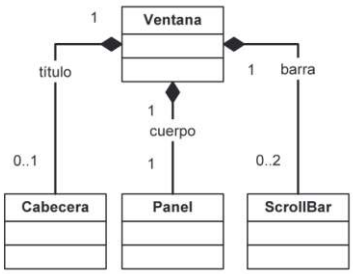
Un todo y sus partes. Las partes pueden formar parte de distintos agregados.

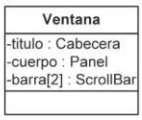



```
public class Poligono{
        private List puntos;
        public Poligono(List puntos2){ // se tiene que controlar que sean 3 como mínimo
                for (Punto punto : puntos2){
                        puntos.add(punto);
                }
        }
        public List get(){return puntos};
}
public class Punto {}
```

Composición

Las partes sólo existen asociadas al compuesto, solo se puede acceder a ellas a través del compuesto.

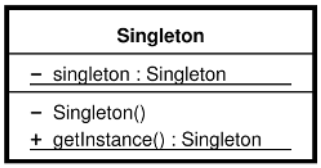




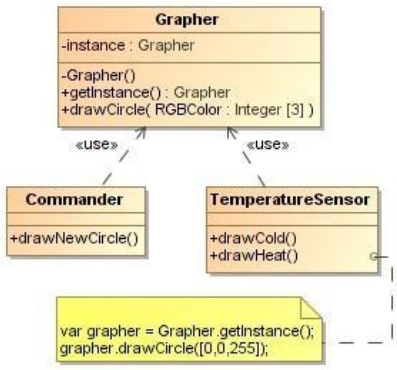

```
public class Ventana{
        private Cabecera titulo;
```

```
        private Panel cuerpo;
        private ScrollBar[]  barra;

// Constructores de Ventana (tener cuidado porque el título y la barra pueden ser 0)
        public Ventana (Cabecera titulo, Panel cuerpo,ScrollBar [] barra) { //controlar
barra - 3
                this.titulo=titulo;
                this.cuerpo=cuerpo;
                this.barra=barra;
        }
        public Ventana (Cabecera titulo, Panel cuerpo) {// Sin barra
                this.(titulo,cuerpo,null);}
        public Ventana (Panel cuerpo,ScrollBar [] Barra) {//Sin titulo
                this.(null,cuerpo,Barra);
        }

        public Cabecera getTitulo(){
                return titulo;
        }

        public Panel getPanel(){
                return cuerpo;
        }

        public [] ScrollBar getBarra(){
                return barra;
        }
}
public class Cabecera{}
public class Panel{}
public class ScrollBar{}
```

Singleton




```
public class Singleton{
        private static Singleton singleton;

        private Singleton(){}

        public static Singleton getInstance(){/// Uso
-> Singleton.getInstance()
                if(singleton == null)
                        singleton= new Singleton();
                return singleton;
        }
}
// More singleton
public class Grapher{
        private static Grapher instance;
        private Grapher(){}
        public Grapher getInstance(){
                if (instance==null)
                        instance=new Grapher();
                return instance;
        }

        public void drawCircle (Integer []
        RBGColor){}
}
```

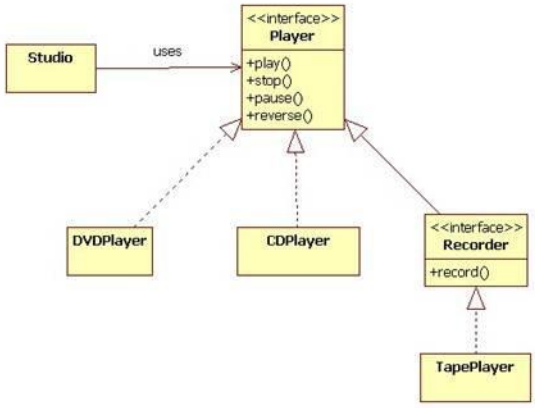



```
public class Commander{
        public void drawNewCircle(Integer [] RBGColor){
                /*Aqui usará drawCircle de Grapher*/
                Grapher.getInstance().drawCircle(RGBColor);
        }
}
public class TemperatureSensor {
        public void drawCold(){
                Grapher.getInstance.drawCircle([/*valores RBG para tonalidad fria*/]);
        }
        public void drawHeat(){
                Grapher.getInstance.drawCircle([/*valores RBG para tonalidad calida*/]);
        }
}
```

Comentado [17]: Listo

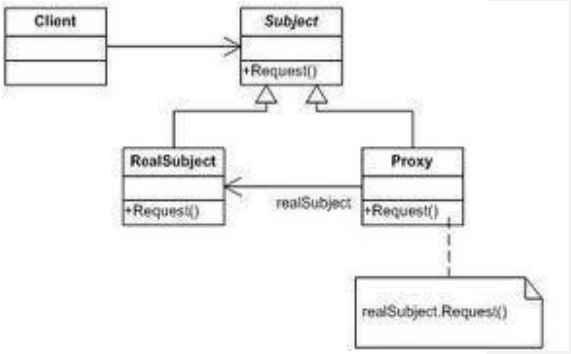
Interface

```
public interface Player {
        public void play();
        public void stop();
        public void pause();
        public void reverse();
}

public DVDPlayer implements
Player{}

public CDPlayer implements Player {}

public interface Recorder extends
Player {
        public void record();
}
public TapePlayer implements Recorder {}
```



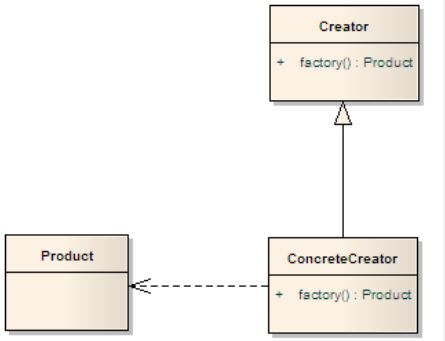
Uml y código java del patrón proxy

```
public class Client{
        private Subject subject;
}

public abstract class Subject{
        public abstract void Request();
}

public class Proxy extends Subject{
        private RealSubject realSubject;
        public void Request(){
                realSubject.Request();
        }
}

public class RealSubject extends Subject{
        public void Request(){}
}
```



Uml y código java del patrón Abstract Factory y Factory Method

```
public abstract class Creator /*Abstract
Factory*/{
        public abstract Product factory();
}

public class ConcreteCreator extends Creator{
        public Product factory(){
        /*Aquí se usa product*/
        }
}
```



NOTA: Realmente el Abstract Factory(Creator) debe ser de un tipo genérico para sólo tener que crear los derivados (Factory Method) de esa factoría respecto al tipo que ellos quieran. En este caso lo más normal es que ConcreteCreator no se llamará así sino ProductCreator.