Informe e2

Principios SOLID:

Principio de inversión de la dependencia:

¿Qué es?

El principio se basa en que los módulos de alto nivel no deberían depender de los de bajo nivel, lo que quiere decir es que los elementos de alto nivel se comunicarán con los de bajo mediante una interfaz en vez de componentes concretos. A su vez los de bajo nivel deben implementar esa interfaz y sobrescribir los métodos.

¿Dónde se usa?

En nuestro código sucede a la hora que declaramos la interfaz Componente, de la cuál podemos usar los métodos declarados en esta, sin tener que saber que implementación tienen. Además, nuestro patrón composición (del que hablaremos más adelante) es un ejemplo de diseño procedimental que presenta un esquema de dependencias donde los módulos de alto nivel, dependen de los módulos de bajo nivel.

Principio abierto-cerrado:

¿Qué es?

El principio se basa en que una clase o método debe de estar abierta a extensiones pero cerrada a modificaciones. Esto quiere decir que si se quiere añadir algo nuevo no debería haber un motivo por el cual el código anterior sea cambiado.

¿Dónde se usa?

Esto es muy fácil de ver para nuestra interfaz componente tal que podemos “extender” estos métodos (al ser una interfaz lo estamos implementando) pero de alguna manera cumplimos este diseño tal que no podemos modificar los métodos definidos en la interfaz, solo podemos “extenderlos” (repito, al ser una interfaz lo estamos implementando).

Patrones de Diseño:

Patrón Composición:

¿Qué es?

Es un patrón de diseño estructural. Este nos permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales. Basicamente, nos permite objetos complejos a partir de otros más simples.

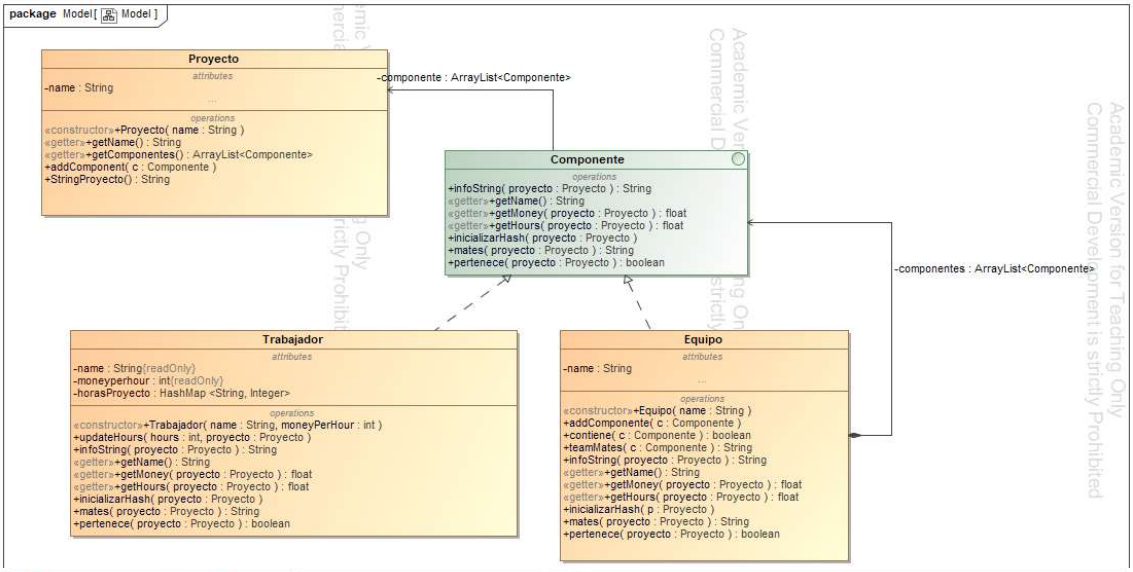
¿Por qué usarlo?

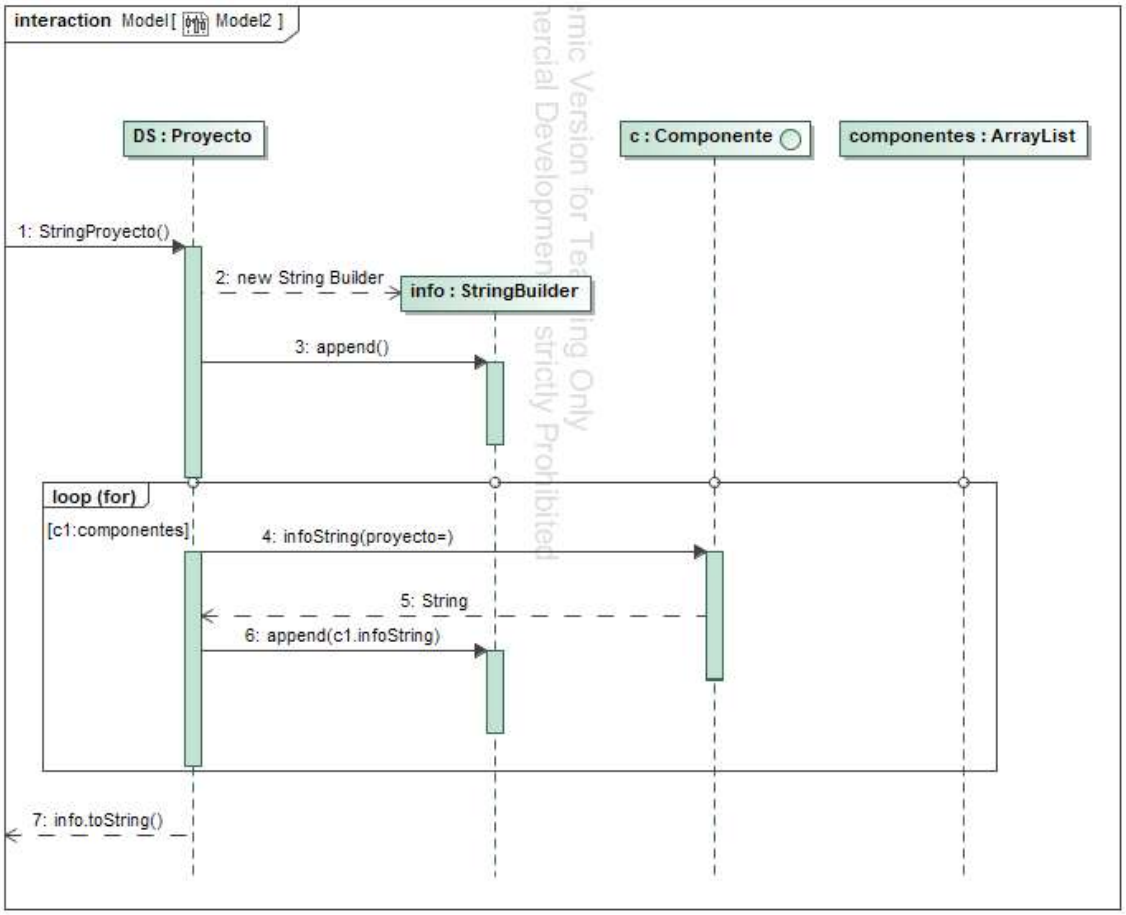
En el ejercicio se nos pedía hacer un proyecto que almacenara una estructura de equipos y trabajadores, y así lo hemos hecho, aplicando composición a estas dos clases este patrón.

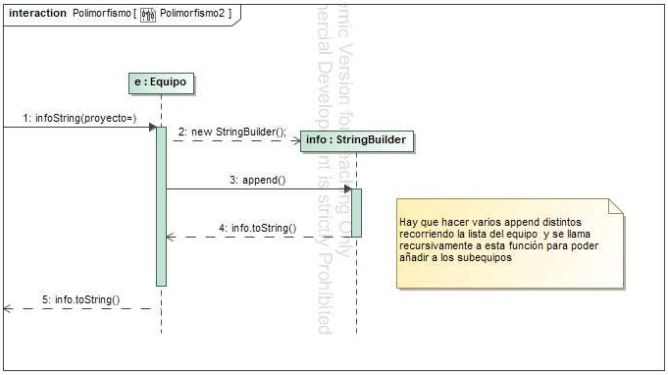
El funcionamiento es simple: tenemos una clase Proyecto, cuyos objetos tienen como objetivo guardar una colección de equipo, y despúes tenemos equipos y trabajadores que son las que utilizamos para aplicar este patrón. Donde trabajadores serán los objetos más simples, que no pueden contener ni más equipos ni trabajadores, y equipos serán más complejos pudiendo contener otros equipos o trabajadores, o ambas cosas.

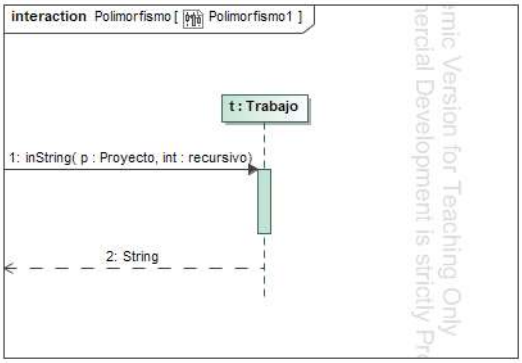
Podríamos haber definido una clase abstracto que despues extendiera equipo pero tal y como era el enunciado nos parecía innecesario ya que solo ibamos a tener equipos como estructura de almacenamiento.

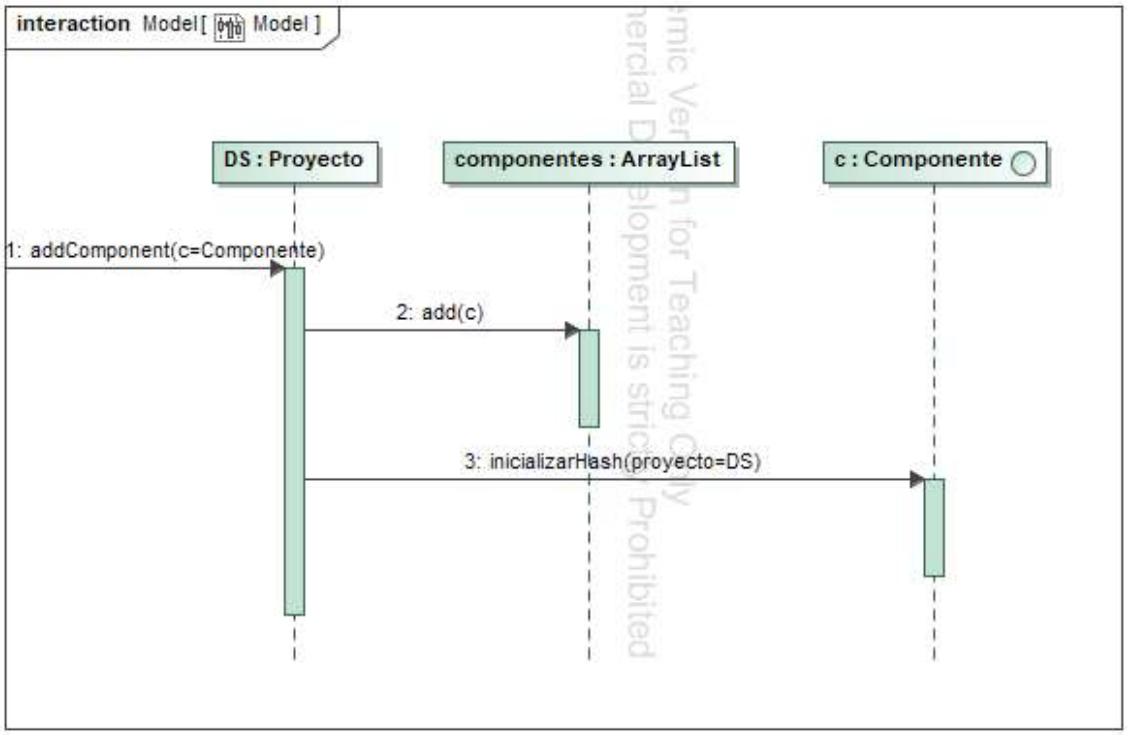
También decidimos que la clase Proyecto no implementase la interfaz ya que sus objetos no son más que una colección de equipo, no podemos tener un proyecto dentro de otro ni dentro de equipo.

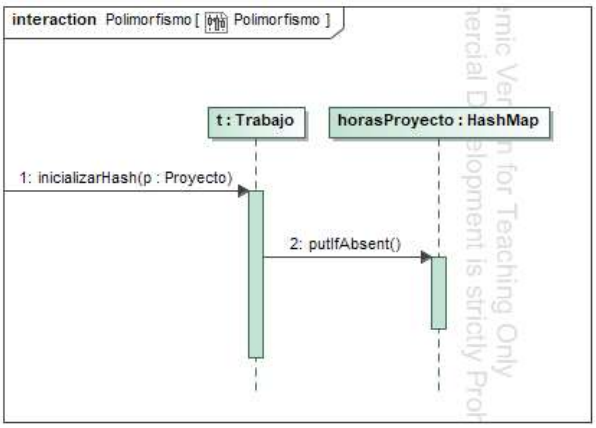

package Model [Model]
Proyecto
attributes
-name : String
...
operations
«constructor»+Proyecto(name : String)
«getter»+getName() : String
«getter»+getComponentes() : ArrayList<Componente>
+addComponent(c : Componente)
+StringProyecto() : String
-componente : ArrayList<Componente>
Componente
operations
+infoString(proyecto : Proyecto) : String
«getter»+getName() : String
«getter»+getMoney(proyecto : Proyecto) : float
«getter»+getHours(proyecto : Proyecto) : float
+inicializarHash(proyecto : Proyecto)
+mates(proyecto : Proyecto) : String
+pertenece(proyecto : Proyecto) : boolean
-componentes : ArrayList<Componente>
Trabajador
attributes
-name : String{readOnly}
-moneyperhour : int{readOnly}
-horasProyecto : HashMap <String, Integer>
operations
«constructor»+Trabajador(name : String, moneyPerHour : int)
+updateHours(hours : int, proyecto : Proyecto)
+infoString(proyecto : Proyecto) : String
«getter»+getName() : String
«getter»+getMoney(proyecto : Proyecto) : float
«getter»+getHours(proyecto : Proyecto) : float
+inicializarHash(proyecto : Proyecto)
+mates(proyecto : Proyecto) : String
+pertenece(proyecto : Proyecto) : boolean
Equipo
attributes
-name : String
...
operations
«constructor»+Equipo(name : String)
+addComponente(c : Componente)
+contiene(c : Componente) : boolean
+teamMates(c : Componente) : String
+infoString(proyecto : Proyecto) : String
«getter»+getName() : String
«getter»+getMoney(proyecto : Proyecto) : float
«getter»+getHours(proyecto : Proyecto) : float
+inicializarHash(p : Proyecto)
+mates(proyecto : Proyecto) : String
+pertenece(proyecto : Proyecto) : boolean

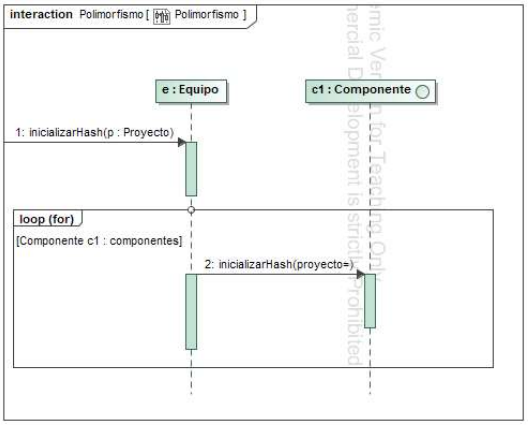

interaction Model [Model2]
DS : Proyecto
c : Componente
componentes : ArrayList
1: StringProyecto()
2: new String Builder
info : StringBuilder
3: append()
loop (for)
[c1:componentes]
4: infoString(proyecto=)
5: String
6: append(c1.infoString)
7: info.toString()

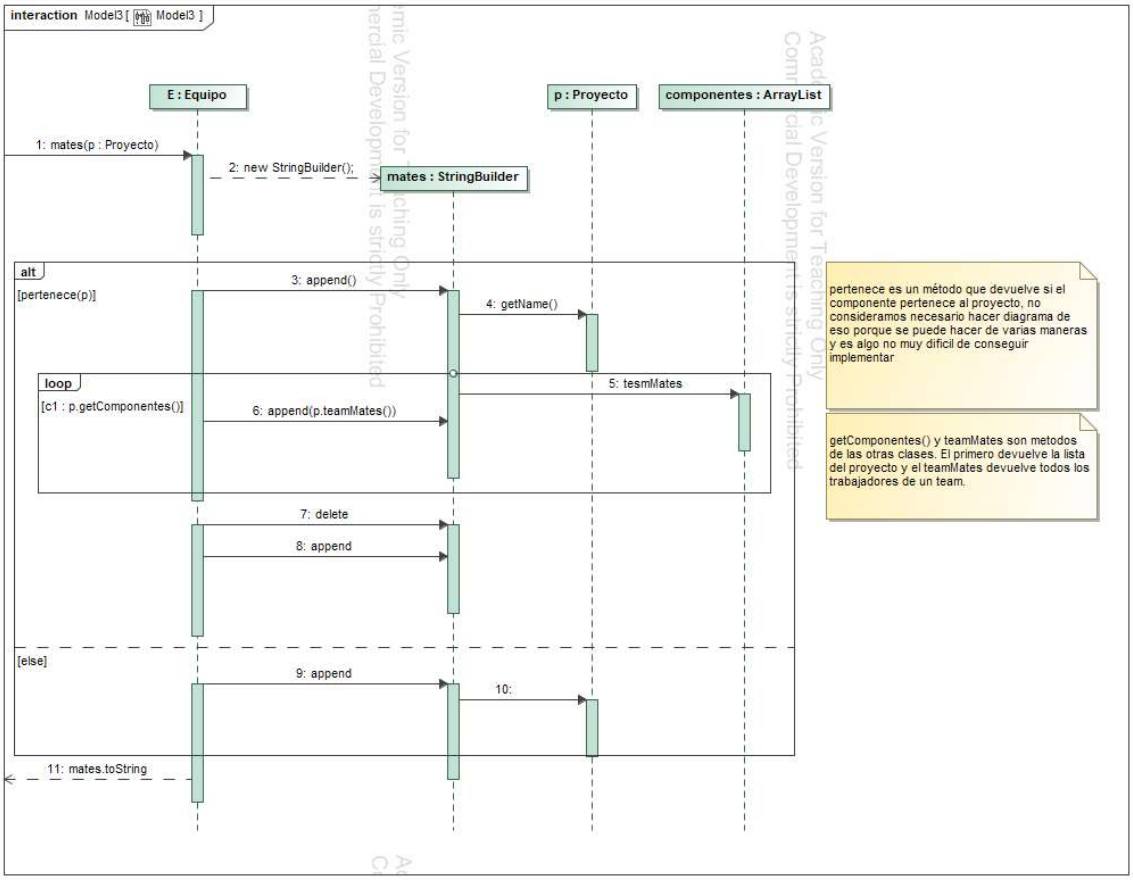

interaction Polimorfismo [Polimorfismo2]
e : Equipo
1: infoString(proyecto=)
2: new StringBuilder();
info : StringBuilder
3: append()
4: info.toString()
5: info.toString()
Hay que hacer varios append distintos recorriendo la lista del equipo y se llama recursivamente a esta función para poder añadir a los subequipos

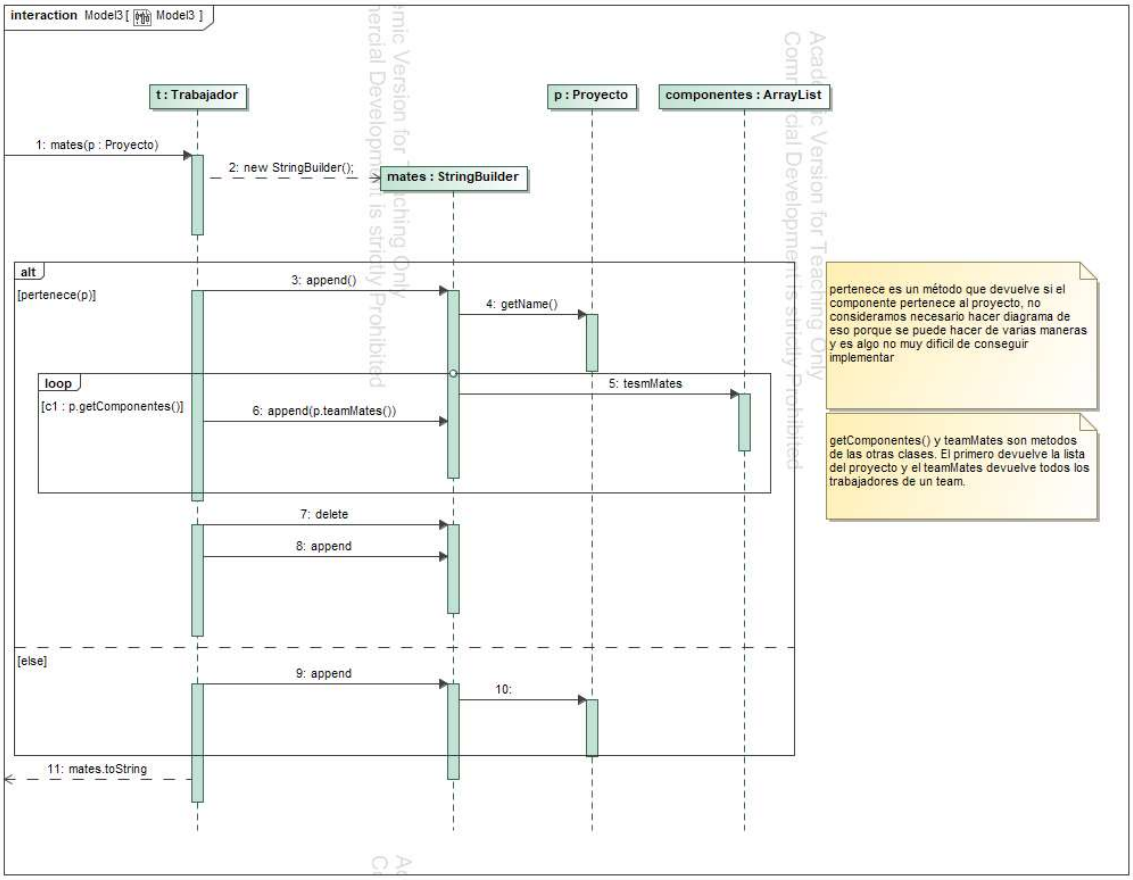

interaction Polimorfismo [Polimorfismo1]
t : Trabajo
1: inString(p : Proyecto, int : recursivo)
2: String


interaction Model [Model]
DS : Proyecto
componentes : ArrayList
c : Componente
1: addComponent(c=Componente)
2: add(c)
3: inicializarHash(proyecto=DS)


interaction Polimorfismo [Polimorfismo]
t : Trabajo
horasProyecto : HashMap
1: inicializarHash(p : Proyecto)
2: putIfAbsent()


interaction Polimorfismo [Polimorfismo]
e : Equipo
c1 : Componente
1: inicializarHash(p : Proyecto)
loop (for)
[Componente c1 : componentes]
2: inicializarHash(proyecto=)


interaction Model3 [Model3]
E : Equipo
p : Proyecto
componentes : ArrayList
1: mates(p : Proyecto)
2: new StringBuilder();
mates : StringBuilder
alt
[pertenece(p)]
3: append()
4: getName()
loop
[c1 : p.getComponentes()]
5: tesmMates
6: append(p.teamMates())
7: delete
8: append
[else]
9: append
10:
11: mates.toString
pertenece es un método que devuelve si el componente pertenece al proyecto, no consideramos necesario hacer diagrama de eso porque se puede hacer de varias maneras y es algo no muy difícil de conseguir implementar
getComponentes() y teamMates son metodos de las otras clases. El primero devuelve la lista del proyecto y el teamMates devuelve todos los trabajadores de un team.


interaction Model3 [Model3]
t : Trabajador
p : Proyecto
componentes : ArrayList
1: mates(p : Proyecto)
2: new StringBuilder();
mates : StringBuilder
alt
[pertenece(p)]
3: append()
4: getName()
loop
[c1 : p.getComponentes()]
5: tesmMates
6: append(p.teamMates())
7: delete
8: append
[else]
9: append
10:
11: mates.toString
pertenece es un método que devuelve si el componente pertenece al proyecto, no consideramos necesario hacer diagrama de eso porque se puede hacer de varias maneras y es algo no muy difícil de conseguir implementar
getComponentes() y teamMates son metodos de las otras clases. El primero devuelve la lista del proyecto y el teamMates devuelve todos los trabajadores de un team.