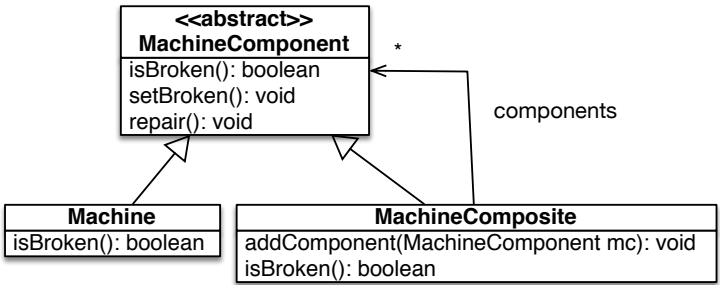
Donat el següent disseny, en el que no marquem més que els aspectes rellevants pel problema que heu de resoldre:



Amb les següents implementacions:

```
1 public abstract class MachineComponent {
2   protected boolean broken = false;
3   public void setBroken() { broken = true }
4   public void repair() { broken = false; }
5   public abstract boolean isBroken();
6 }
7
8 public class Machine extends MachineComponent {
9   public boolean isBroken() { return broken; }
10 }
11
12 public class MachineComposite extends MachineComponent {
13   private List<MachineComponent> components =
14                                        new ArraysList<>();
15   public void addComponent(MachineComponent mc) {
16     components.add(mc);
17   }
18   public boolean isBroken() {
19     if (broken) { return true; }
20     for(MachineComponent mc: components) {
21       if (mc.isBroken()) { return true; }
22     }
23     return false;
24   }
25 }
```

El que voldrem es poder monitoritzar les instàncies de `MachineComponent`, per exemple, creant una interfície gràfica que ens permeti mostrar quan un component canvia d'estat, és a dir, quan `isBroken` passa de cert a fals o de fals a cert.

Per això ens caldrà poder detectar, **des de la capa presentació** quan, per un `MachineComponent`, **el mètode `isBroken` passa de valer cert a valer fals o viceversa**.

Apartat a (2 punts)

Quin patró fareu servir i per què?

Apartat b (2 punts)

Com haurem de modificar les classe `Machine` i `MachineComposite`, serà millor començar per un disseny en el que aquestes classes són els més independents possible. Per tant, per aquest apartat i el següent partirem d'un disseny amb les mateixes implementacions que en el cas anterior (duplicades a les subclasses si cal) i les declaracions següents:

```
1 public abstract class MachineComponent {
2   // Només declaracions de mètodes abstractes
3 }
4 public class Machine extends MachineComponent {
5   // Implementacions dels mètodes abstractes
6 }
7 public class MachineComposite extends MachineComponent {
8   // Implementacions dels mètodes abstractes
9 }
```

Implementeu la classe `Machine` per a que funcioni correctament. Potser us cal també modificar la **declaració** de la classe `MachineComponent`.

Tingueu en compte que si a una màquina ja trencada li fem un `setBroken` **no ha canviat res** (ja que el mètode `isBroken` continua retornant cert). El mateix passa si fem `repair` sobre una màquina que funciona.

Apartat c (4 punts)

Aplicar-ho al cas de `MachineComposite` serà una mica més complicat, ja que les seves instàncies poden trencar-se (resp. reparar-se) no per elles mateixes, sinó pels seus components.

Per exemple, si a una màquina que ja té un component trencat se li trenca un altre, no ha canviat res (la màquina composta continua trencada). O bé, si es marca com trencada una màquina que té un component trencat, tampoc ha canviat res.

Per simplificar una mica el problema **podeu considerar que no hi ha components repetits i que la relació components forma un arbre**.

PISTA: Pot ser molt útil que la classe guardi informació sobre els components que té trencats.

Implementeu la classe `MachineComposite` resultant per a que funcioni correctament. Potser us cal també modificar la **declaració** de la classe `MachineComponent` i/o `Machine`.

Apartat d (1 punt)

Arregleu el codi dels apartats b i c de manera que no hi hagi codi duplicat, etc, etc.

Pels mètodes que no canvien d'implementació podeu fer servir {...} per la seva implementació i, en cas d'ambigüitat, indicar de quina classe prové el codi.

Apartat e (1 punt)

Mostreu el diagrama de classes final de la vostra solució.