## Lab XII.

## XII.1 Arquiteturas microkernel (plugins)

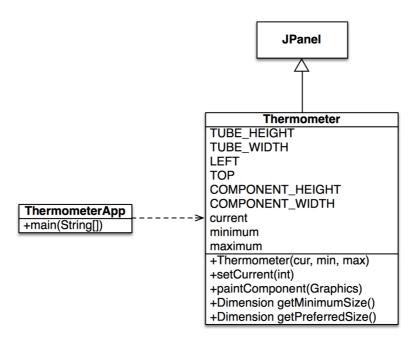
Tome como referência o seguinte código (*IPlugin.java* e *Plugin.java*). Construa um conjunto de classes que implementem a interface *IPlugin* e que permitam adicionar funcionalidades ao programa principal.

```
// IPlugin.java
package reflection;
public interface IPlugin {
        public void fazQualQuerCoisa();
// Plugin.java
package reflection;
import java.io.File;
import java.util.ArrayList;
import java.util.Iterator;
abstract class PluginManager {
   public static IPlugin load(String name) throws Exception {
      Class<?> c = Class.forName(name);
      return (IPlugin) c.getDeclaredConstructor().newInstance();
   }
}
public class Plugin {
   public static void main(String[] args) throws Exception {
       File proxyList = new File("bin/reflection");
       ArrayList<IPlugin> plgs = new ArrayList<IPlugin>();
       for (String f: proxyList.list()) {
         if (f.endsWith(".class")) {
           try {
              plgs.add(PluginManager.load("reflection."
                             + f.substring(0, f.lastIndexOf('.')));
            } catch (Exception e) {
               System.out.println("\t" + f + ": Componente ignorado. Não é IPlugin.");
            }
         }
       Iterator<IPlugin> it = plgs.iterator();
       while (it.hasNext()) {
          it.next().fazQualQuerCoisa();
   }
}
```

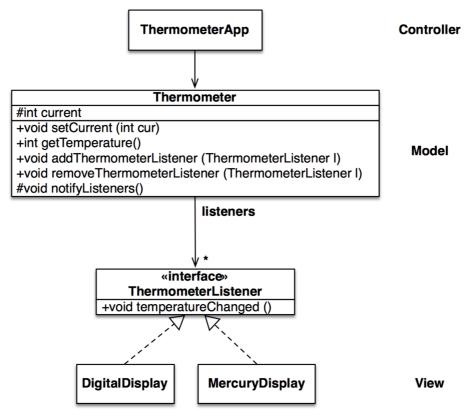
## XII.2 Arquitetura Model-View-Control

No ficheiro *thermo.zip* encontrará uma solução para representar um termómetro em Java Swing. Esta abordagem, numa única classe, torna difícil mudar a implementação e adicionar, por exemplo, uma segunda forma de visualização.





Numa segunda implementação, *thermoMVC.zip*, optou-se por organizar o código segundo um modelo *Model-View-Control*, que usa o padrão *Observer*.



O termómetro está agora dividido em um modelo e duas representações (observações) diferentes. Com esta arquitetura, podemos facilmente modificar a nossa *ThermometerApp* para utilizar a visualização que queremos, podemos usar ambos, ou poderemos usar múltiplas réplicas de uma ou mais visualizações. Podemos criar estas variações com pequenas alterações na *ThermometerApp* (controlo) e nenhuma para a classe termómetro

(modelo).

O Controlo neste exemplo é bastante simples, sendo representado por um campo de texto que permite inserir uma nova temperatura. Isso é implementado com um *ActionListener* – que está associado ao campo de texto. Este *listener* faz uma chamada direta ao Termómetro que regista a mudança no modelo e, em seguida, usa *notifyListeners* para informar todos os *observers* para que possam atualizar a interface (view).

O objetivo deste trabalho é analisar o código fornecido e criar um terceira representação do termómetro (usando Swing, a consola, um ficheiro, etc.).

## XII.3 Serialização JSON de objetos arbitrários

(opcional, para quem queira estudar java reflection)

Um dos usos mais comuns das funcionalidades *Reflection* do java é encontrado na Serialização de objetos arbitrários, por exemplo para o formato JSON. De facto, a biblioteca JSON-Lib utilizada nesta cadeira utiliza precisamente esta estratégia. Explore as funcionalidades de *Reflection* do java de maneira a serializar objetos de forma recursiva. A sua implementação deverá ser capaz de exportar os atributos e métodos públicos, tendo em atenção o seu tipo de retorno (outros objetos, arrays, etc). Note que utilizando esta estratégia é possível criar um método *default* para serializar todo o tipo de objetos, independentemente da sua implementação.

```
public class Ship {
       private String name;
       private int size;
       private String[] passageiros;
       private Owner owner;
       public Ship(String name, int size) {
              super();
              this.name = name;
              this.size = size;
       }
       public String getName() {
              return name;
       public int getSize() {
              return size;
       public Owner getOwner() {
              return owner;
       }
       public void setOwner(Owner owner) {
              this.owner = owner;
       }
       public String[] getPassageiros() {
              return passageiros;
       }
}
```



```
public class PDSSerializer {
                   public static String fromObject(Object o){
                            //Class <u>cl</u> = o.getClass();
                            //Explore os metodos
                            //cl.getMethods();
                            //cl.getFields();
                            //<u>Veja</u> o <u>javadoc</u> <u>das</u> classes: Class, Method, Field, Modifier
                   }
         }
         public static void main(String[] args) {
    Ship s = new Ship("BelaRia", 200);
    s.setOwner(new Owner("Manuel"));
    s.setPassageiros(new String[]{"Manuel", "Amilcar"});
                   System.out.println( PDSSerializer.fromObject(s) );
         }
$ java -jar Serializer.jar
Name: BelaRia
Price: 200
Owner: {
Name: Manuel
Passageiros: [Manuel, Amilcar]
```

