Comme précédemment, -25% pour les petites fautes, sans cumuler jusqu'à 0 s'il n'y a que ça, -50% pour les fautes plus sérieuses

1 Implantation des cellules et des batteries

1. Quel design pattern pouvez-vous utiliser pour implanter les classes Cell et Battery ?

100 dès qu'il y a écrit « composite », j'ai mis 50 à un type qui avait fait le bon dessin sans nommer le pattern

1.2 Dans un package pobj.hb, donnez le code de la classe Cell

```
package hb;

public class Cell implements IBattery{
private boolean lastFul;
private int nbCycles;
private int capacity;
private int fullCapacity;

public Cell(int capamax){
	fullCapacity = capamax;
	capacity = capamax;
	nbCycles = 0;
	lastFul = true;
}
	@Override
	public boolean charge() {
		if (capacity==fullCapacity)		return false;
		else {
			capacity++;
			if (capacity==fullCapacity)lastFul=true;
			return true;
		}
	}

	public boolean discharge() {
		if (capacity==0)		return false;
		else {
			capacity--;
			if (capacity==0) {
				lastFul=false;
				nbCycles++;
			}
			return true;
		}
	}

	public int getCurrentCapacity() {
		return capacity;
	}

	public int getNbCycles() {
		return nbCycles;
	}

	@Override
	public int getFullCapacity() {
		return fullCapacity;
	}
```

```
}
```

50% pour le mécanisme qui gère effectivement les cycles (seule « difficulté », perso j'utilise le boolean lastFul), 15% pour le *implements*, 10% pour le reste des attributs , 25% pour le reste des méthodes (à pondérer par les fautes)

Mes stats : un seul a une solution (compliquée) pour gérer les cycles, un autre a essayé un truc faux (il ne compte que les charges complètes) j'ai pas mal de 35% (ne gèrent pas le pattern ou extends) ou 50%.

1.3 Dans un package pobj.hb, donnez le code de la classe Battery tel que spécifié jusqu'ici

```
package hb;

import java.util.ArrayList;

public class Battery implements IBattery{
private ArrayList<IBattery> elems;


public Battery(){
	elems = new ArrayList<IBattery>();
}
public void add(IBattery b){
	elems.add(b);
}


	@Override
	public int getCurrentCapacity() {
		int retour = 0;
		for (IBattery c : elems){
			retour += c.getCurrentCapacity();
		}
		return retour;
	}

	@Override
	public int getFullCapacity() {
		int retour = 0;
		for (IBattery c : elems){
			retour += c.getFullCapacity();
		}
		return retour;
	}

	@Override
	public int getNbCycles() {
		int max = 0;
		for (IBattery c : elems){
			if (c.getNbCycles()>max) max = c.getNbCycles();
		}
		return max;
	}

}
```

-25% si pas l'implements
-25% si pas de liste d'éléments (-10% pour Arraylist<Cell> plutôt que <Ibattery> et OK pour une ArrayList non paramétrée)

-25% si autres attributs inutiles
+35% par méthode juste (borner à 100%)
Je mets 10% (au lieu de 35) sur getFullCapacity() à ceux qui font une multiplication d'une capacité élémentaire par le nombre de cellules

2 Implantation du chargement et déchargement d'une batterie

2.1 Quel design pattern peut-on utiliser pour implanter les deux approches proposées ci-dessus ?

Stratégie
0 ou 100. J'ai deux décorateurs et un adaptateur

2.2 Proposez une implantation complète de ces deux approches. Vous penserez à proposer un nouveau constructeur pour la classe Battery qui prendra en argument un booléen indiquant si on choisit une
approche ou l'autre.

Dans Battery:

```
private AccuStrategy myStrat ;

public Battery(boolean bycycles){
	elems = new ArrayList<IBattery>();
	if (bycycles) myStrat = new AccuCyclesFirst();
	else  myStrat = new AccuCapacityFirst();
}
	@Override
	public boolean charge() {
		return myStrat.charge(elems);
	}

	@Override
	public boolean discharge() {
		return myStrat.discharge(elems);
	}
```

Interface:

```
package hb;

import java.util.ArrayList;

public interface AccuStrategy {
	public boolean discharge(ArrayList<IBattery> elems);
	public boolean charge(ArrayList<IBattery> elems);
}
```

Les deux stratégies :

```
import java.util.ArrayList;

public class AccuCapacityFirst implements AccuStrategy {

	public boolean charge(ArrayList<IBattery> elems) {
		int cellmin = 0;
		int capamin = Integer.MAX_VALUE;
		for (int c=0;c<elems.size(); c++){
			 if (elems.get(c).getCurrentCapacity()<capamin)
				{
				cellmin = c;
				capamin = elems.get(c).getCurrentCapacity();
				}
```

```
            }
            return elems.get(cellmin).charge();
      }

      public boolean discharge(ArrayList<IBattery> elems) {
            int cellmax = 0;
            int capamax = 0;
            for (int c=0;c<elems.size(); c++){
                  if (elems.get(c).getCurrentCapacity()>capamax)
                        {
                        cellmax = c;
                        capamax = elems.get(c).getCurrentCapacity();
                        }
            }
            return elems.get(cellmax).discharge();
      }
}
public class AccuCyclesFirst implements AccuStrategy {

      public boolean charge(ArrayList<IBattery> elems) {
            int cellmin = 0;
            int cyclesmin = Integer.MAX_VALUE;
            for (int c=0;c<elems.size(); c++){
                  if (elems.get(c).getNbCycles()<cyclesmin)
                  {
                        cellmin = c;
                        cyclesmin = elems.get(c).getNbCycles();
                  }
            }
            return elems.get(cellmin).charge();
      }

      public boolean discharge(ArrayList<IBattery> elems) {
            int cellmin = 0;
            int cyclesmin = Integer.MAX_VALUE;
            for (int c=0;c<elems.size(); c++){
                  if (elems.get(c).getNbCycles()<cyclesmin)
                  {
                        cellmin = c;
                        cyclesmin = elems.get(c).getNbCycles();
                  }
            }
            return elems.get(cellmin).discharge();
      }
}
```

25% pour l'interface (ou 0 dès que c'est faux). C'est sympa...
25% pour les ajouts dans la classe (5 = attribut, 10 = constructeur, 10 = methodes charge et discharge qui délèguent)
25% pour chaque strategie (j'ai mis 10 dès que ca avait l'air de faire quelque chose)

3 Construction des batteries complexes

3.1 Quel design pattern peut-on utiliser pour réaliser la construction de batteries complexes ?

Factory.
J'ai mis 50 à ceux qui m'ont mis une AbstractFactory. C'est trop gentil?

3.2 Proposez une implantation permettant de construire une batterie aléatoire conforme à la spécification

ci-dessus.

```
package hb;

import java.util.Random;

public class BatteryFactory {
	private static Random generateur = new Random();

	public static Cell createRandomCell(){
		return new Cell(generateur.nextInt(40)+10);
	}

	public static Battery createRandomBattery(){
		Battery b = new Battery(generateur.nextBoolean());
		int nbElems = generateur.nextInt(5);
		for (int i=0;i<nbElems;i++){
			if (generateur.nextBoolean()) b.add(createRandomCell());
			else b.add(createRandomBattery());
		}
		return b;
	}
}
```

4 Batteries rapides

4.1 Quel design pattern peut-on utiliser pour réaliser la construction de ces batteries rapides ?

Plutôt Adapter, éventuellement Decorateur
je suis sympa, je mets 100 s'ils ont mis un des deux mots (un seul a essayé de coder, il a le début correct d'un adapter, je lui mets 10).
J'ai un singleton, c'est original. Un observer, aussi.

4.2 Proposez une implantation de ce nouveau type de batteries. Vous ne donnerez que la méthode public
int discharge(int unitesDemandees); qui prend en paramètres le nombre d'unités d'énergie qu'on souhaite récupérer et renvoie le nombre d'unités effectivement fournies.

Débrouillez-vous, soyez sympa, 10 dès qu'il y a au moins un bout de quelque chose...