4) a)

```
public class TreeSleeperDecorateObservable
                implements ITreeSleeper, Iobservable {

    protected Observable obs;
    protected ITreeSleeper treeSleeper;

    public TreeSleeperDecorateObservable (ITreeSleeper treeSleeper,
                                          Observable obs) {
        this.obs = obs;
        this.treeSleeper = treeSleeper;
    }

    public void sleepInTree () {
        treeSleeper.sleepInTree();
    }

    public void awakening() {
        treeSleeper.awakening();
    }

    public String getSleepLong() {
        return treeSleeper.getSleepLong();
    }

    public String getSleepShort() {
        return treeSleeper.getSleepShort();
    }

    public boolean getSleepState() {
        return treeSleeper.getSleepState();
    }

    public void setSleepTab() {
        treeSleeper.setSleepTab();
        notifyObservers();
    }

    public void setAwakeLib() {
        treeSleeper.setAwakeLib();
        notifyObservers();
    }

    public void addObserver(Observer observateur) {
        this.obs.addObserver(observateur);
    }

    public void notifyObservers(Object o) {
        this.obs.setChanged();
        this.obs.notifyObservers(o);
        this.obs.clearChanged();
    }

    public void setObservable(Observable obs) {
        this.obs = obs;
    }
}
```

4) b) À la place d'un décorateur, on aurait pu créer une autre classe abstraite "AbstractTreeSleeperObservable" qui aurait implément l'interface IObservable. Le résultat attendu aurait été le même mais il aurait fallut dupliquer tout le code de la classe AbstractTreeSleeper dans la nouvelle classe observable en plus de l'implémentation de l'interface IObservable. Ce genre de méthodes n'est pas conseillé car elles empêchent d'étendre le comportement des classes existantes à la volé. De plus, ce genre de méthodes ne respectent pas le pincipe de responsabilité unique.

4) c) Non, il suffit d'utilisen la classe que l'on vient de créer ("TreeSleeperDecorateurObservable") afin de rendre un HammouthAdapte observable :

```
new TreeSleeperDecorateurObservable (new HammouthAdapte,
                                     new SujetObservable );
```

4) d) Il est intéressant d'utiliser le pattern Shategy dans ce contexte car cela nous permet de changer rapidement et à la volé le comportement observable d'un objet.

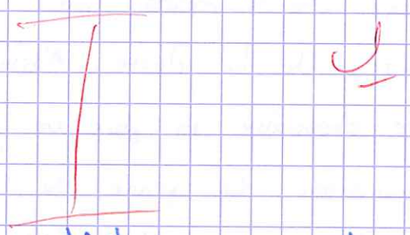
5) Le pattern composite ne serait pas utile dans ce contexte il n'y a pas de notion d'objet "composant" et "composite" qui ressort de la solution utilisée. Pour réveillen une ensemble de marsupiaux et leurs immitateur, une collection d'objet de type AbstractTreeSleeper serait donc tout à fait indiqué.

Benoit GALATI
4 IRC

18/03/2013

Conception Orientée Objet DP 1

6) L'utilisation d'une fabrique abstraite permettrait de centraliser l'instanciation des objets concrets dans une seule et même classe pour plus de clarté.



7 (a) Il n'y a aucun intérêt à ce que le client puisse accéder à l'objet ITreeSleeper qui a été décoré car le Décorateur TreeSleeperDecorable Observable possède déjà toutes les fonctionnalité de l'ITreeSleeper

7, b) Le fait de donner accès à l'objet Observable du client lui permettrait de connaître des information supplémentaires au sujet des observeurs (leurs nombres, etc...) mais cela représente peu d'intérêt pour le client. Il vaut mieux étendre le nombre de fonctions disponibles via l'interface IObservable plutôt que de donner directement accès à l'objet Observable.

2) sleepInTree() et awakening() mettent en œuvre le pattern template method car leurs comportements dépendent de l'implémentation, effectuée dans les sous-classes Opossum et Mansupilami, des méthodes setSleepLib() et setAwakeLib().

3) a) Le pattern Adapter permet à un client de considérer un Mammouth comme un AbstractTreeSleeper.

b)

```
Class MammouthAdapte extends AbstractTreeSleeper {
  protected mammouth;

  public MammouthAdapte (Mammouth m) {
    this.mammouth = m;
  }

  public void setSleepLib() {
    setSleepLong("...");
    setSleepShort("...");
  }

  public void setAwakeLib() {
    setSleepLong("...");
    setSleepShort("...");
  }
}
```