



Ecole Militaire Polytechnique
الشهيد عبد الرحمان طالب

ÉCOLE MILITAIRE POLYTECHNIQUE

FILIÈRE : GÉNIE INFORMATIQUE

SPECIALITÉ : RÉSEAUX ET SÉCURITÉ INFORMATIQUE

RAPPORT

TP 2 :Génie logiciel

Élèves :

Benzemam ABD EL HAKIM

Belkhira MOHAMED ALI

Haou ACHREF

Layoune BRAHIM

Enseignant :

Cdt MERABTI

17 mai 2021

Table des matières

1	introduction :	2
2	Activité 1.1 :	
	Test d'une implémentation basique du TimerService	3
2.1	partie 1 :	3
2.2	partie 2 :	4
2.3	Solution :	4
2.4	Définition du patron de conception "délégué" :	4
3	Activité 1.2 :	
	Ajout des interfaces graphiques	5
3.1	Définition du patron de conception "Etat" :	5
3.2	Objectives :	6
3.3	Conception :	6
3.4	Execution de l'application :	8
3.5	Test :	9
4	Conclusion :	9

1 introduction :

Dans ce Tp nous allons continuer avec la montre intelligent que nous avons entamé dans les Tp précédents et en vas faire face a différents problèmes durant la phase de développement que nous allons essayer de résoudre grâce aux patrons de conception qu'on a vue en cours et même on va aussi développer des interfaces graphique qui illustrent le fonctionnement de notre montre intelligente en utilisant "java-Swing" et finalement on cherche a avoir une implantations d'une montre intelligente qui fonctionne parfaitement .

Probleme avec l'implémentation précédente du design patern observable :

1. Il faut faire qu'une insertion a la fois si non on aura des problèmes.
2. Unsubscribe au momment de notification va remédier à un problème (NULL POINTER EXCEPTION) qu'on peut le résoudre avec l'Iterator.
3. Un code qui la meme chose se trouve un peut partout car plusieurs observables donc on doit notifier dans plusieurs places. pour résoudre ce problème on fait une implementation d'une classe observable ,et les autres obserbables herite de cette classe.
4. cette implimentation ne prend pas en considération le single responsabilité (chaque chose a une seule place et chaque place est reservé a une seule chose).
5. problème lié au langages modernes qui utilisent le Garbage Collector qui est le processus responsable à récupérer la mémoire inutilisée pendant l'exécution en détruisant les objets inutilisés, dans le cas de l'implementation de l'observer faite dans le TP précédent ce processus va générer des problèmes de gestion de mémoire qu'on peut les résoudre avec les weak references mais c'est une solution difficile et nécessite la connaissance large du Garbage Collector.
6. contient des accès conquérants (sélection critique) qui peut etre resolu en utilisant les sémaphores mais cette solution bloque le fonctionnement du système ou bien utiliser des threads a part mais pas util quand on a un grand nombre.

Objectifs principales de TP :

1. trouver une solution globale aux problèmes de l'implémentation de l'observer en utilisant le patrone de conception "delegué"
2. Gérer une application comportant plusieurs états en utilisons le patron de conception "State".

2 Activité 1.1 :

Test d'une implémentation basique du TimerService

Dans cette activité on a cloné un project qui est fournie dans l'énoncé du tp grâce à l'outil de gestion de source "git" grâce à la commande "git clone"

Après avoir cloner le project on remarque que les modules présent sont

1. core-lookup
2. launcher
3. timer-service
4. time-service-impl

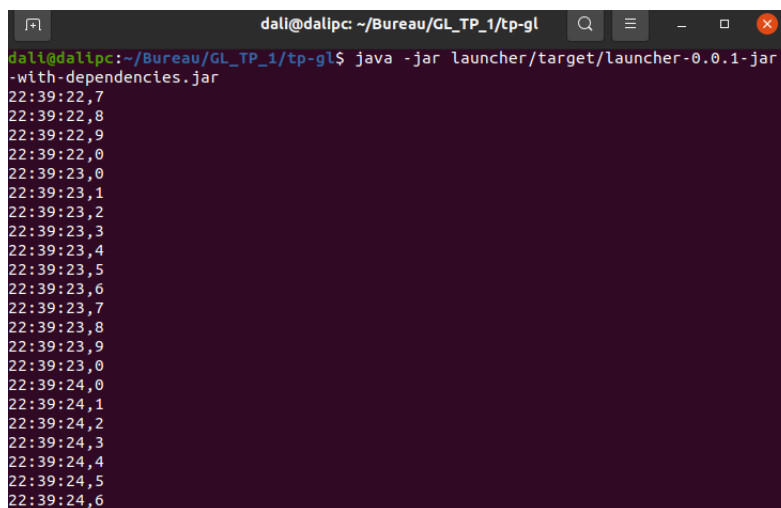
remarque :

le module dummy-timer-service-impl nous offre une classe nommé DummyTimerServiceImpl qui hérite de la classe TimerTask qui nous permet d'utiliser un qui nous sera très utile et la class DummyTimerServiceImpl aussi nous offre une implémentions **INITIALE** du TimerService dans la quelle nous allons utiliser une liste pour gérer les observateurs.

2.1 partie 1 :

Obervation :

Après compilation et exécution du code en Remarque le fonction ment normale de la montre qui nous donne le temps actuelle et continue à calculer le temps grâce à la fonction "**testDuTimeService()**" ou on test le bon fonctionnement de notre Timer avec la fonction propertyChange() de la class "**AfficheurHeureSurConsole**".



```
dali@dali: ~/Bureau/GL_TP_1/tp-gl
dali@dali:~/Bureau/GL_TP_1/tp-gl$ java -jar launcher/target/launcher-0.0.1-jar-with-dependencies.jar
22:39:22,7
22:39:22,8
22:39:22,9
22:39:22,0
22:39:23,0
22:39:23,1
22:39:23,2
22:39:23,3
22:39:23,4
22:39:23,5
22:39:23,6
22:39:23,7
22:39:23,8
22:39:23,9
22:39:23,0
22:39:24,0
22:39:24,1
22:39:24,2
22:39:24,3
22:39:24,4
22:39:24,5
22:39:24,6
```

2.2 partie 2 :

Observation : on est demander de commenter la ligne 30 et de décommenter la ligne 31 comme dans la figure suivante :

```
30 //ts.addTimeChangeListener(new AfficheurHeureSurConsole());
31 | ts.addTimeChangeListener(new CompteAREbour(5 + (int)(Math.random() * 10)));
32
```

ici le changement principale est dans l'utilisation de la classe `CompteAREbour` et ça fonction de `propertyChange()` qui est implémenter par les deux classe `compteAREbour` et la classe `AfficheurHeuresurconsole` d'où on a changer l'objet listner de l'afficheur-heursur console vers un objet `compteArRbourt`

on a ajouter une boucle de 20 iteration pour l'ajout du listner compte a rebour on remarque qu'une erreur a fait apparence qui est la suivante

```
Exception in thread "Timer-0" java.util.ConcurrentModificationException
    at java.base/java.util.LinkedList$Itr.checkForComodification(LinkedList.java:970)
    at java.base/java.util.LinkedList$Itr.next(LinkedList.java:892)
    at org.emp.gl.time.service.impl.DummyTimeServiceImpl.updateSecode(DummyTimeServiceImpl.java:107)
    at org.emp.gl.time.service.impl.DummyTimeServiceImpl.updateDixiemeDeSecode(DummyTimeServiceImpl.java:99)
    at org.emp.gl.time.service.impl.DummyTimeServiceImpl.timeChanged(DummyTimeServiceImpl.java:85)
    at org.emp.gl.time.service.impl.DummyTimeServiceImpl.run(DummyTimeServiceImpl.java:49)
    at java.base/java.util.TimerThread.mainLoop(Timer.java:556)
    at java.base/java.util.TimerThread.run(Timer.java:506)
dali@dali:~/Bureau/GL_TP_1/tp-gl$
```

l'erreur rencontrer est "ConcurrentModificationException" qui arrive lorsque on essaye de supprimer un élément d'une liste quand on est entrain de boucler sur cette liste avec une boucle "for" dans notre cas quand le premier listner `CompteARoubour` attein la valeur 0 (on peut l'apercevoir dans le capture précédent)il va appeler la méthode `removeTimeChangeListener` qui va causer le problème voila l'image :

```
if (compteArebours == 0) { // se désabonner du TimerService !
    Lookup.getInstance()
        .getService(TimerService.class)
        .removeTimeChangeListener(this);
}
```

2.3 Solution :

on va utiliser le patron de conception de delegation `timer-service-impl-withdelegation` qui a une class `TimerServiceImplWithDelegation` qui va déléguer la gestion des observateur à un objet de Type `PropertyChangeSupport`

2.4 Définition du patron de conception "délégué" :

Le délégué est un design pattern en programmation objet ou un objet au lieu de traiter l'une de ces taches il va déléguer cette tâche à un objet associé crée par des professionnels dans ce domaine. Il crée ainsi une inversion de responsabilité. L'objet aidant est appelé

le delegate. Le patron de délégation est l'un des patron fondamental d'abstraction qui implique d'autres modèles tels la composition (appelle aussi agrégation).

le patron de class du Delegué consiste a delegué notre travaille a une class qui est spécialisé dans le domaine qui va gere tout le travaille (ajoutlistner,removelistner,et d'informer les listner par la methode "firepropertychange") sans se soucier de l'implementation et ceci grâce a **"PropertyChangeSupport"**

Les modification apporté au code seront fait suivant les etapes suivante :

1. on declare notre delegué :

```
PropertyChangeSupport delegate = new PropertyChangeSupport(this);
```

2. et apres on lui donne la tache d'ajouter et de supprimer les listner

```
public void addTimeChangeListener(TimerChangeListener pl) {
    delegate.addPropertyChangeListener(pl);
}

@Override
public void removeTimeChangeListener(TimerChangeListener pl) {
    delegate.removePropertyChangeListener(pl);
}
```

3. et puisque notre delegué va prendre tout le travaille il va aussi se charger de prendre en compte les mise a jours(properttychange) grance a la fonction textbf"firePropertyChange".

```
delegate.firePropertyChange(TimerChangeListener.DIXEME_DE_SECONDE_PROP,
    oldValue, dixiemeDeSeconde);
```

4. Et finalement après ces changement le problème de la boucle n'est plus présent car le délègue a bien fait sont travaille sans qu'on connaisse les détails de l'implementation donc la programmation est devenu facile :).

3 Activité 1.2 :

Ajout des interfaces graphiques

maintenant on a une implementation du TimeService qui est fonctionnelle et on cherche a finaliser notre montre intelligent donc on pass a la partie interface graphique. Dans cette activité on va utiliser le patron de conception Etat(state).

3.1 Définition du patron de conception "Etat" :

État est un patron de conception comportemental qui permet de modifier le comportement d'un objet lorsque son état interne change. L'objet donne l'impression qu'il change

de classe.

3.2 Objectives :

1. une interface graphique permettant d'afficher l'heure -GUI Display- afficher l'heure les minutes et les secondes.
2. une interface graphique -GUI Control- qui contient les boutons qui nous permetts de controller les fonction relatives a notre montre : -le button MODE : qui permet de selectionner le mode qu'on veut incrementer seconde,minute ou heure.
-le button INCREMENT : qui fait l'icrementation.
-le button CONFIG : pour activer le mode de changement.

Les deux interface graphique sans independents chaque un est executer seulle.

Rappele :

on va travailler avec java-swing car java-fx nous a poser des problemes, Swing est une API qui fait partie de la bibliothèque Java Foundation Classes (JFC), C'est une API dont le but est similaire à celui de l'API AWT mais dont les modes de fonctionnement et d'utilisation sont complètement différents, et nous offre l'ioutilte Drag and Drop qui nous facilite la tache pour cree les interfaces

3.3 Conception :

Dans cette partie nous allons donner le diagramme de classe de notre montre ce diagramme explique bien les relation entre les classe est les interface utiliser dans cette implementation et comment on a utiliser les deux design patterns Delegation et Etat (State).

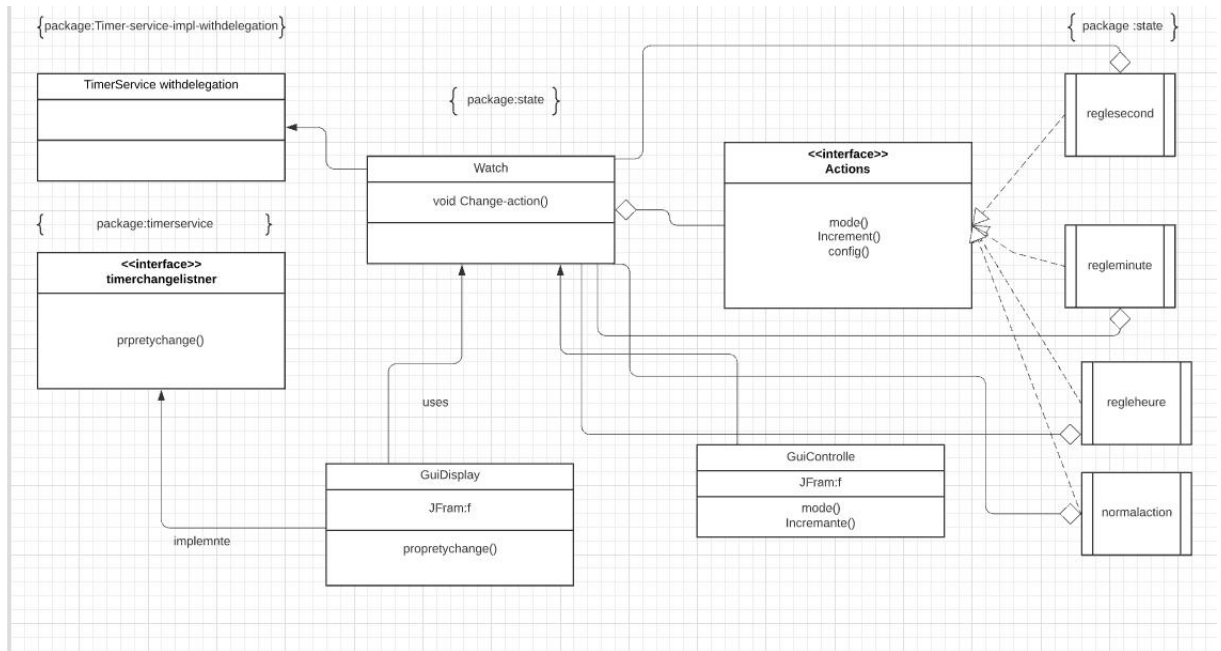
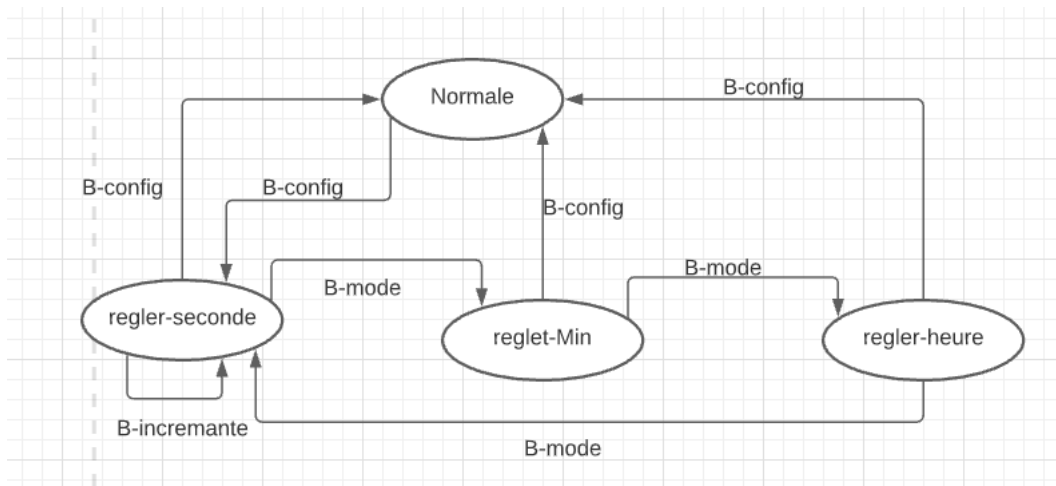


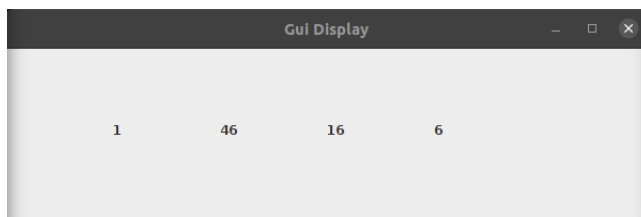
diagramme a étate :



3.4 Execution de l'application :

Lorsque on fais l'execution de notre application les deux interface graphique seront afficher comme montrer dans le deux figures ci-dessous :

interface graphique GUI Display :



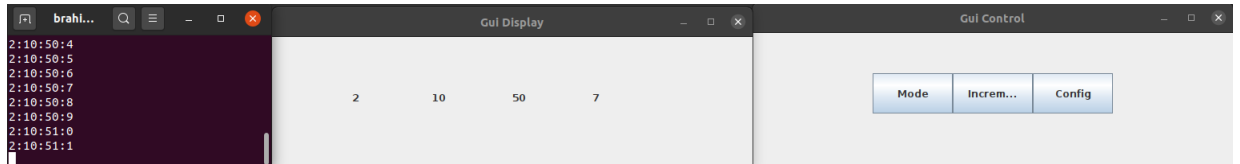
Interface graphique GUI control :



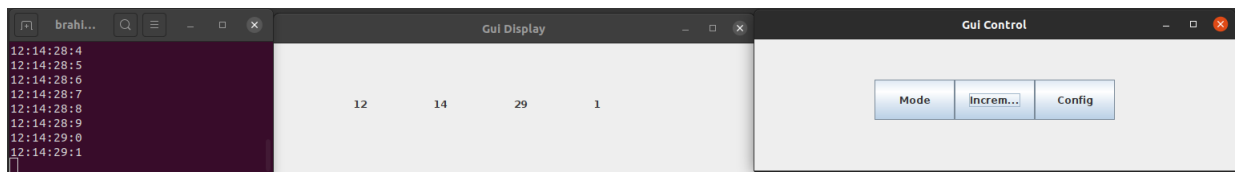
3.5 Test :

On va faire un test sur notre application dans cette partie 1- On execute notre montre. 2-puis on va cliquer sur le button config pour choisit le mode d'incrémenter les heures par 10. Les deux figure ci-dessous donne ce qui passe :

Avant incrementation :



Après incrementation :



4 Conclusion :

Pour conclure en va parler des patron de conception qui nous on aider dans notre tp parmi eux ya Le patron de conception Etat (State) qui a plusieurs avantage parmi eux Principe de responsabilité unique, Organisez le code lié aux différents états dans des classes séparées, Principe ouvert/fermé, Ajoutez de nouveaux états sans modifier les classes état ou le contexte existants, Simplifiez le code du contexte en éliminant les gros blocs conditionnels de l'automate.

La délégation peut également être une technique de conception/réutilisation puissante, Le principal avantage de la délégation est la souplesse d'exécution - le délégué peut facilement être modifié lors de l'exécution(aspect dynamique)