Contrôle terminal info2 / Semestre 3

M3105 : Conception de programmation objet avancées

Nom du responsable :	**BORNE Isabelle**
Date du contrôle :	**22/10/21**
Durée du contrôle :	**1h30**
Nombre total de pages :	**9 pages**
Impression :	**Recto**
Documents autorisés :	**Polycopiés de cours**
Calculatrice autorisée :	**non**
Réponses :	**Copie + 4 feuilles à remplir**

Bien lire le sujet.
Les 2 exercices sont indépendants
Les pages 3,4, 5 et le diagramme de classes sont à mettre dans la copie

Exercice 1

Le code donné en annexe décrit la simulation d'un compteur de temps (Timer) affiché par deux types d'horloge : une analogique (AnalogClock) et une digitale (DigitalClock).

Le patron de conception Observer peut être appliqué à cet exemple

Question 1

Expliquer comment le patron peut être appliqué : le fonctionnement et le rôle de chaque classe de l'exemple par rapport à la structure générale du patron Observer);

Question 2

Compléter le code directement sur la feuille en ajoutant toutes les lignes de code qui manquent pour faire fonctionner le patron.

Remarques : la classe java.util.Calendar est utilisée juste pour obtenir les heures, minutes et secondes courantes.

Les variables de classes : HOUR_OF_DAY, MINUTE, et SECOND contiennent les valeurs de l'heure courante que l'on obtient par un message get() envoyé à un Calendar.

Le message Calendar.getInstance() retourne un Calendar avec les valeurs des 3 variables de classe remplies avec l'heure courante.

Exercice 2

Question 1

Dessinez le diagramme de classes de l'application en utilisant la feuille où les boîtes des classes sont tracées. Il faut bien sûr ajouter tout ce qui manque sans oublier package, dépendance, etc.

La classe ScenarioAction n'est pas présente sur le schéma et n'est pas à représenter

Question 2

Des patrons de conception ont été utilisés dans cette application.

Pour chaque patron donner son nom, son objectif et, en reprenant la description des patrons dans le cours, donner le rôle joué par chacune des classes de l'application impliquées dans le patron.

Question 3

La classe Action a un attribut *state*, cela fait penser à un patron vu en cours. Indiquer les modifications à faire si on voulait appliquer ce patron : explications des modifications et éventuellement un peu de code.

```
package clock;
import java.util.Calendar;


public class ClockTimer  {
  private Calendar rightNow;

  public ClockTimer(){
    this.rightNow= Calendar.getInstance();// to get the current time
  }
  public int getHour() {
    return this.rightNow.get(Calendar.HOUR_OF_DAY);
  }
   public int getMinute() {
     return this.rightNow.get(Calendar.MINUTE);
  }
    public int getSecond() {
      return this.rightNow.get(Calendar.SECOND);
  }

   // permet de mettre à l'heure le timer
    public void tick() {
      rightNow = Calendar.getInstance();



    }
}
```

```
package clock;



public class AnalogClock   {
  private ClockTimer timer;

  public AnalogClock(ClockTimer timer){
    this.timer=timer;
```

```
    }


  public void draw() {
     int hour = timer.getHour();
     int minute = timer.getMinute();
     int second = timer.getSecond();
     System.out.println("Analog Clock time is "+hour+":"+minute+":"+second);
   }









}
```

```
package clock;



public class DigitalClock  {
  private ClockTimer timer;

  public DigitalClock (ClockTimer timer){
    this.timer=timer;




    }

  public void draw() {
    int hour = timer.getHour();
    int minute = timer.getMinute();
    int second = timer.getSecond();
   System.out.println("Digital Clock time is "+hour+"h "+minute+" minutes and
"+second+ " seconds");
  }




}
```

```
import clock.*;
public class ScenarioClock {
  public static void main(String[] args) {
    ClockTimer timer = new ClockTimer();
    DigitalClock  dClock = new DigitalClock(timer);
    AnalogClock aClock = new AnalogClock(timer);
    timer.tick();
  }
}
```

ANNEXE 2

```
package action;
public abstract class Action{

  protected String msg;
  protected ActionState state;

  public Action() {
    this.state = ActionState.READY;
  }

  public void doStep() throws ActionFinishedException {
    if (this.isFinished()) {
      throw new ActionFinishedException();
    }
    if (this.state==ActionState.READY) {
      this.state = ActionState.IN_PROGRESS;}
      this.reallyDoStep();
    if (this.stopCondition()) this.state=ActionState.FINISHED;
  }

  protected abstract void reallyDoStep();

  protected abstract boolean stopCondition();

  public boolean isFinished(){
    return this.state==ActionState.FINISHED;
  }

  public ActionState getState() {
    return this.state;
  }
}
```

```
package action;
public class PredictedAction extends Action{
  private int waitingTime;

  public PredictedAction(int w) {
    this.waitingTime = w;
  }

  protected void reallyDoStep() {
    this.waitingTime = this.waitingTime -1; }
```

```
  protected boolean stopCondition() {
    return this.waitingTime ==0;
  }
}
```

```
package action;
import java.util.ArrayList;

public abstract class Scheduler extends Action{
  protected ArrayList<Action> theActions;

  public Scheduler() {
    theActions = new ArrayList<Action>();
  }

  protected void reallyDoStep() {
      Action action = this.nextAction();
      if (action!=null) {
        try{
          action.doStep();
        }
         catch (ActionFinishedException e) {
                   System.out.println(e.getMessage());
         }
         if (action.isFinished())  theActions.remove(action);
      }
      else { System.out.println("TheActions vide - action null");
      }
    }

  protected abstract Action nextAction();

  public void add(Action a) {
              theActions.add(a);
  }

  protected boolean stopCondition() {
    return theActions.isEmpty();
  }
}
```

```
package action;
import java.util.Iterator;
```

```
public class FairScheduler extends Scheduler {

   protected Action nextAction() {
     Action a = null;
     Iterator<Action> itActions = theActions.iterator();
     if (itActions.hasNext())
         a = itActions.next();
     return a;
   }
 }
```

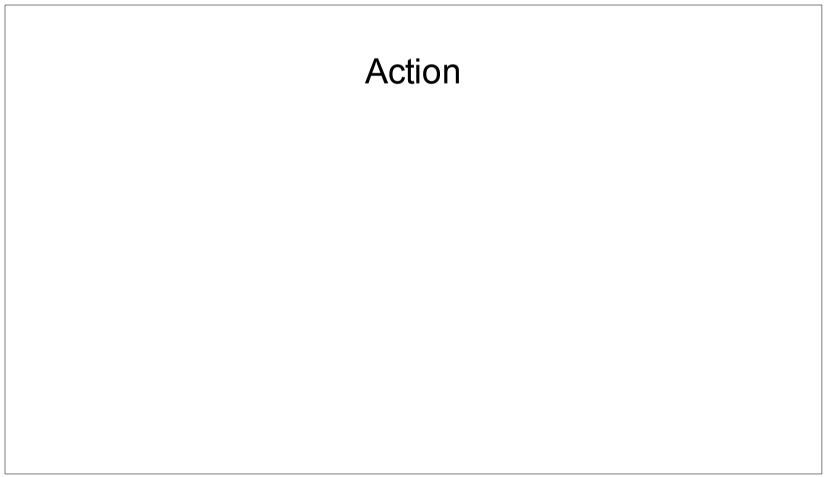
```
 package action;
 public class Scenario extends Scheduler{

   protected Action nextAction() {
     return theActions.get(0);
   }
 }
```

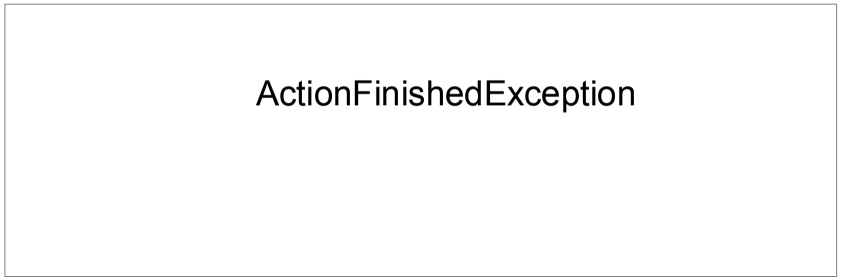
```
package action;
 public class ActionFinishedException extends Exception{

   public ActionFinishedException() {super();}
   public ActionFinishedException(String s) {super(s);}
 }
```

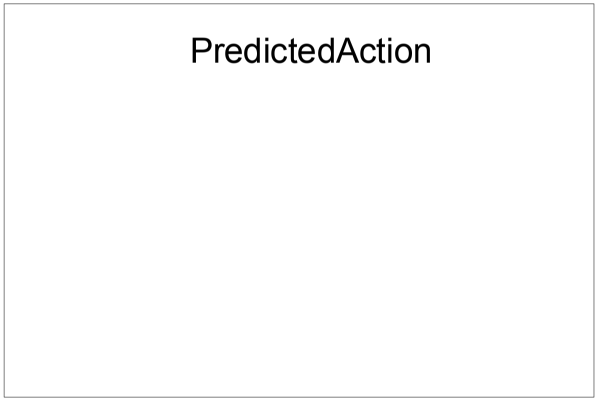
```
package action;
 public enum ActionState{
       READY,
       IN_PROGRESS,
       FINISHED;
 }
```

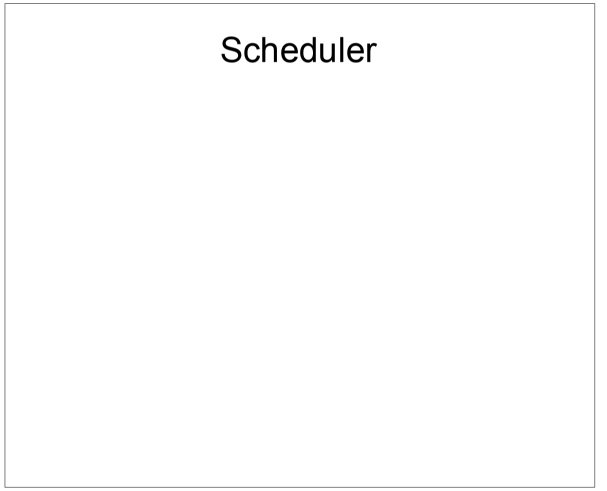
```
import action.*;
 public class ScenarioAction {
   public static void main(String[] args) {
     PredictedAction action1 = new PredictedAction(2);
     FairScheduler scheduler= new FairScheduler();
     Scenario scn= new Scenario();
     scheduler.add(action1);
     scn.add(new PredictedAction(3));
     try{
        scheduler.doStep();
        scn.doStep();    }
      catch (ActionFinishedException e) {
         System.out.println(e.getMessage());  }
   } }
```


ActionState

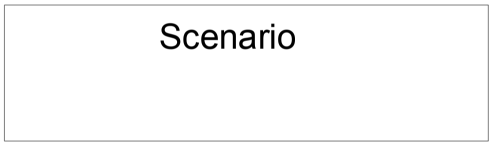

Action

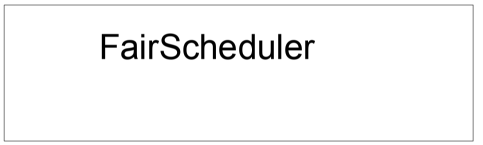

ActionFinishedException


PredictedAction


Scheduler

Nom
Prénom


Scenario


FairScheduler