Examen Final

—

Les technique de construction d'architectures logicielles avancées

—

Remarques :

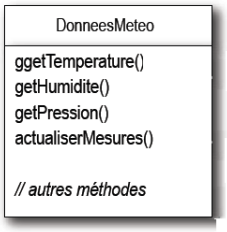
— Les notes du cours manuscrites sont autorisées.

— Les documents imprimés ne sont pas autorisés ainsi que les appareils électroniques (PC, Tablette, téléphone..).

— La lisibilité et la clarté de vos réponses et de votre code sont très importantes. Une réponse pas claire ne sera pas prise en compte lors de la correction.

Exercice 1 :(5 points)

Nous souhaitons mettre en place une station météorologique consultable en ligne. Des données Météo sont récoltées et qui représentent la température, l'hygrométrie et la pression atmosphérique. L'application doit fournir trois affichage : les conditions actuelles, statistiques et les prévisions simples. Noter que les affichages sont actualisés en temps réel dés la réception des nouvelles données. De plus cette station météo doit être extensible. l'interface *DonneesMeteo* est donnée comme suite :



Questions :

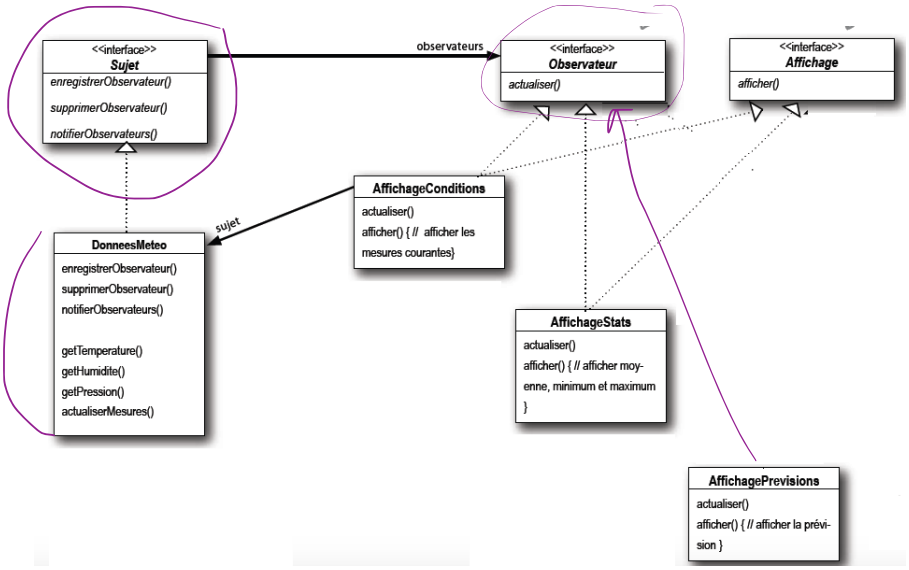
1 (1p) Donner les inconvénients de la solution présenté comme suite :

```
public class DonneesMeteo {

    // déclaration des variables d'instance

    public void measurementsChanged() {

      float temp = getTemperature();
      float humidite = getHumidite();
      float pressure = getPression();

      affichageConditions.actualiser(temp, humidite, pression);
      affichageStats.actualiser(temp, humidite, pression);
      affichagePrevisions.actualiser(temp, humidite, pression);
    }
    // autres méthodes de DonneesMeteo
}
```

Les inconvenients c'est que l'utilisation des objets concrets (affichageConditions, affichageStats et affichagePrevisions) dans la méthode actualiserMesures(), rendent le changement, la modification et la suppression des éléments d'affichage difficile, car cela passe par la modification du programme de la méthode actualiserMesures().

2 (1p) Quel design pattern utilisé pour résoudre le problème.
Le design pattern observer permet l'abstraction des objet d'affichage et la mise a jour dynamique des affichages.

3 (1p) Donner la conception en utilisant le design pattern choisi.

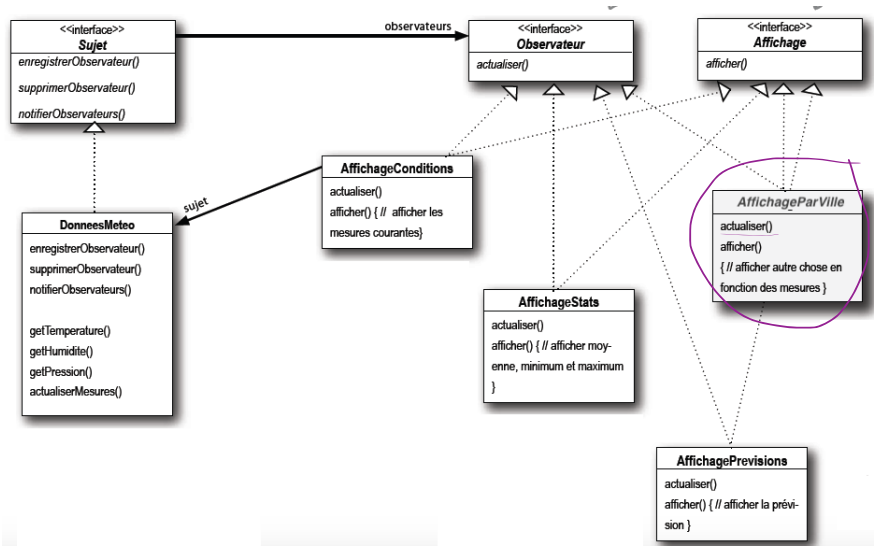


4 (1p) Donner une implémentation simple.

```
public interface Sujet {
     public void enregistrerObservateur(Observateur o)
     public void supprimerObservateur(Observateur o);
     public void notifierObservateurs();
}


public interface Observateur {
     public void actualiser(float temp, float humidite, float pression);

}



public interface Affichage {
     public void afficher();
}

public class DonneesMeteo implements Sujet {
     private ArrayList observateurs;
     private float temperature;
     private float humidite;
     private float pression;

     public DonneesMeteo() {
          observateurs = new ArrayList();
     }

     public void enregistrerObservateur(Observateur o) {
          observateurs.add(o);
     }

     public void supprimerObservateur(Observateur o) {
          int i = observateurs.indexOf(o);
          if (i >= 0) {
               observateurs.remove(i);
          }
     }

     public void notifierObservateurs() {
          for (int i = 0; i < observateurs.size(); i++) {
               Observateur observateur= (Observateur)observateurs.get(i);
               observateur.actualiser(temperature, humidite, pression);
          }
     }

     public void actualiserMesures() {
          notifierObservateurs();
     }

     public void setMesures(float temperature, float humidite, float pression) {
          this.temperature = temperature;
          this.humidite = humidite;
          this.pression = pression;
          actualiserMesures();
     }

     // autres méthodes de DonneesMeteo
}
```

```
public class AffichageConditions implements Observateur, Affichage {
     private float temperature;
     private float humidite;
     private Sujet donneesMeteo;

     public AffichageConditions(Sujet donneesMeteo) {
          this.donneesMeteo = donneesMeteo;
          donneesMeteo.enregistrerObservateur(this);
     }

     public void actualiser(float temperature, float humidite, float pression) {
          this.temperature = temperature;
          this.humidite = humidite;
          afficher();
}

     public void afficher() {
           System.out.println("Conditions actuelles : " + temperature
                + " degrés C et " + humidite + " % d'humidité");
     }
}
```

5 (1p) Nous souhaitons ajouter l'affichage des prévisions par ville. Proposer une solution simple.

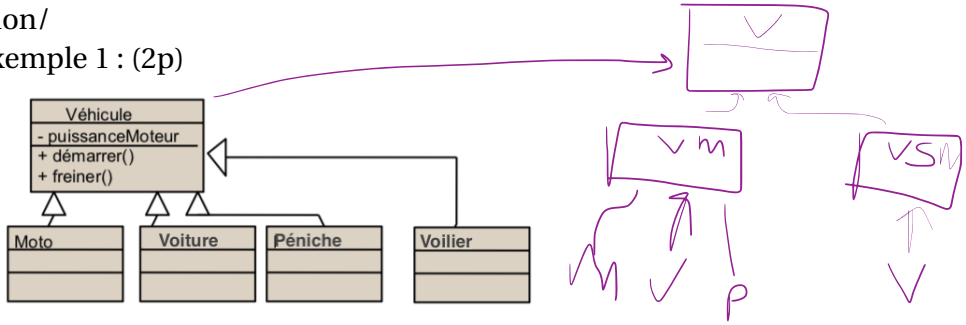


Exercice 2 :(10 points)

1 Donner les différents principes SOLID avec une petite description.
- Single responsability (0.75p) : une seul responsabilité par classe.
- Open/Close (0.75p) : fermer à la modification et ouvert à l'abstraction.
- Liskov substitution (0.75p) : étendre une classe héritée sans modifier le comportement de la classe mère.
- Interface segragation (0.75p) : une interface contient les fonctionnalités d'un seul client.
- Dependency Inversion (1p) : la dépendance entre classes par l'abstraction et non par l'implémentation.

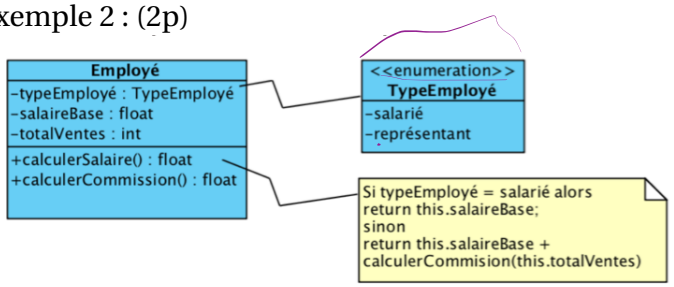
2 Pour chaque exemple donné, définir le principe SOLID non respecter en argumentant et proposer une solution/

1 Exemple 1 : (2p)

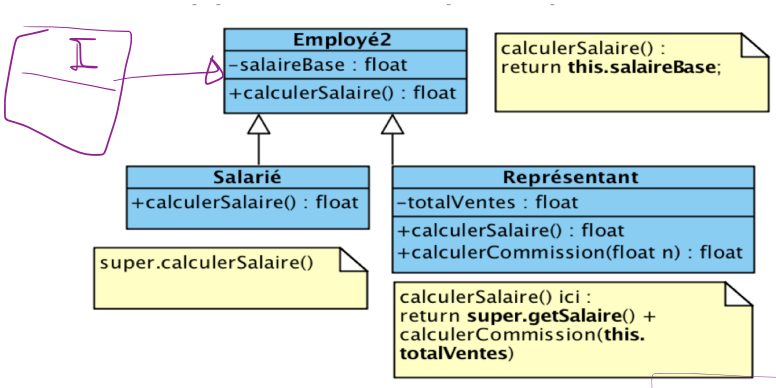


Le principe non respecté est Liskov substitution, car le voilier n'a pas de moteur donc pas besoin des deux fonctions démarrer et freiner.
Pour corriger la conception il faut ajouter un autre niveau d'abstraction pour séparer entre les véhicules a moteur et les véhicules sans moteurs.

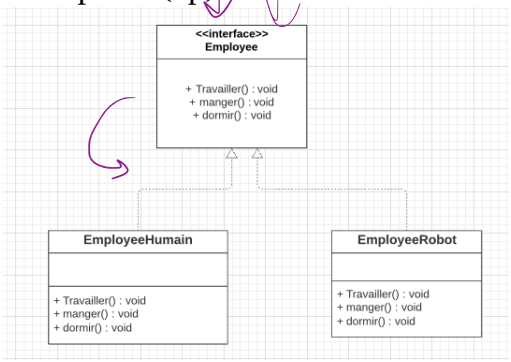
2 Exemple 2 : (2p)



Le principe non respecter est open/close, car si on doit ajouter un nouveau type d'employé il faut changer l'implémentation de la classe Employé.
La correction :



3 Exemple 3 : (2p)



Le principe solide non respecté est interface segrégation, car la classe EmployeeRobot n'a pas besoin n'implémenté les deux fonctions manger et dormir.

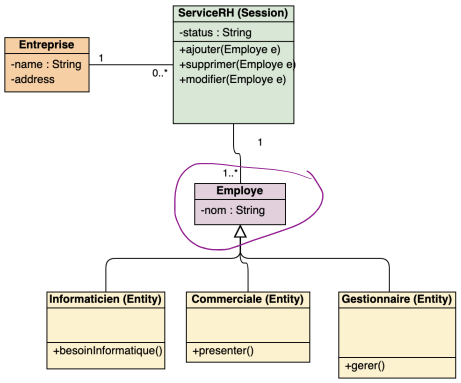
Pour corriger cette erreur il faut couper l'interface en deux interfaces une pour chaque sous classe.

Exercice 3 : (5 points)

Dans le cadre d'une entreprise de prestation de services en informatique. Le service ressource humain gère l'ensemble des employés de l'entreprise composés de : informaticiens, commerciales et de gestionnaires.
Le rôle du commerciale est de présenter les services proposés par l'entreprise aux prés des clients.
Le rôle de l'informaticien est de répondre aux besoins informatique (hardware et software) des clients.
Le rôle du gestionnaire est de gérer l'ensemble des services au sein de l'entreprise.
L'application permet d'ajouter, de supprimer ou de modifier des employés. Nous souhaitons mettre en place un module **EJB** à intégrer dans un système distribué.

Questions :

1 (2p) Proposer une conception en précisant les classes session et entité Bean.



2 (1p) Pour chaque classe session bean préciser si elle est avec état ou sans état en justifiant vos choix.
La classe ServiceRH est une classe session bean avec etat, c'est pas une session partagée, car c'est une session qui va changer l'état de la base de données.

3 (2p) Donner l'implémentation de la classe qui permet de manipuler les employés (ajout/suppression/-modification).

```
@Stateful
public class ServiceRH implements ServiceRHInterface{
@PersistenceContext
   private EntityManager em;
   public void Ajouter(Employe e){
    if(em.find(e.getID())==null ) {
    em.persist(e);
    em.flush();
    }else {System.out.println("L'employé existe déja");}
   }
   public void modifier(Employe newE,){
    Employe oldE = em.find(newE.getID())
    if(oldE !=null ) {
    oldE = newE;
    em.merge(oldE);
    em.flush();
    }else {System.out.println("L'employé n'existe pas");}
   }
  public void supprimer(Employe e){
    Employe oldE = em.find(newE.getID())
    if(em.find(newE.getID())!=null ) {
    em.delete(e);
    em.flush();
    }else {System.out.println("L'employé n'existe pas");}
   }

}
```

Bon courage et bonne continuation.