IPST-CNAM
Programmation JAVA
NFP 121
Jeudi 24 Février 2022

Avec document
Durée : **2 h30**
Enseignant : LAFORGUE Jacques

1ère Session NFP 121

CORRECTION

L'examen se déroule en deux parties. Une première partie de 1h15mn, sans document, consacrée à des questions de cours, et une deuxième partie de 1h 15mn, avec document, consacrée en la réalisation de programmes Java.

Au bout de 1h15mn, les copies de la première partie seront ramassées avant de commencer la deuxième partie.

Pour la première partie, vous devez rendre le QCM rempli et les réponses aux questions libres écrites sur des copies vierges.

Pour la deuxième partie, vous écrivez vos programmes sur des copies vierges. Vous devez écrire les codes commentés en Java.

1ère PARTIE : COURS (sans document)
Durée: 1h15

1. QCM (35 points)

Mode d'emploi :

Ce sujet est un QCM dont les questions sont de 3 natures :

- **les questions à 2 propositions**: dans ce cas une seule des 2 propositions est bonne.
  - +1 pour la réponse bonne
  - –1 pour la réponse fausse
- **les questions à 3 propositions** dont 1 seule proposition est bonne
  - + 1 pour la réponse bonne
  - –½ pour chaque réponse fausse
- **les questions à 3 propositions** dont 1 seule proposition est fausse
  - + ½ pour chaque réponse bonne
  - –1 pour la réponse fausse

Il s'agit de faire une croix dans les cases de droite en face des propositions.
On peut remarquer que cocher toutes les propositions d'une question revient à ne rien cocher du tout (égal à 0).

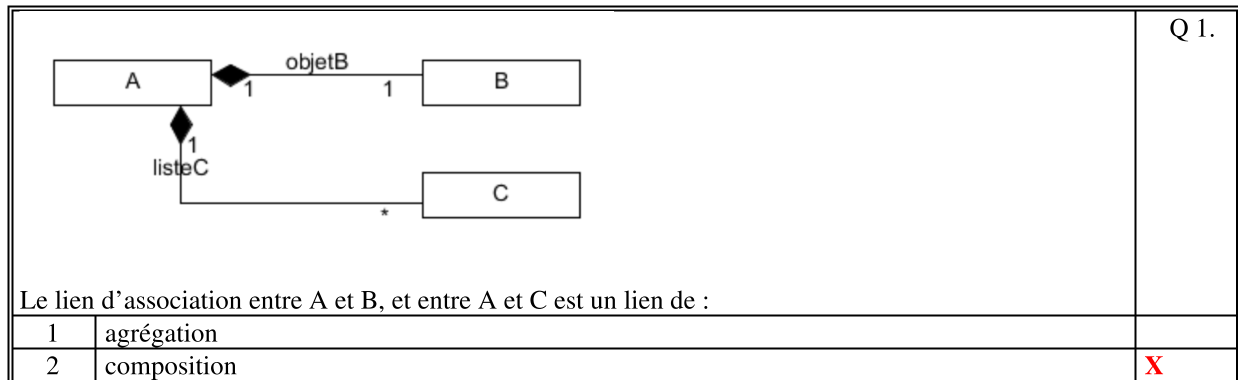
Si vous devez raturer une croix, faites-le correctement afin qu'il n'y ait aucune ambiguïté.

N'oubliez pas d'inscrire en en-tête du QCM, votre nom et prénom.

Vous avez droit à **4 points** négatifs sans pénalité.

NOM:	PRENOM:

Q 1.

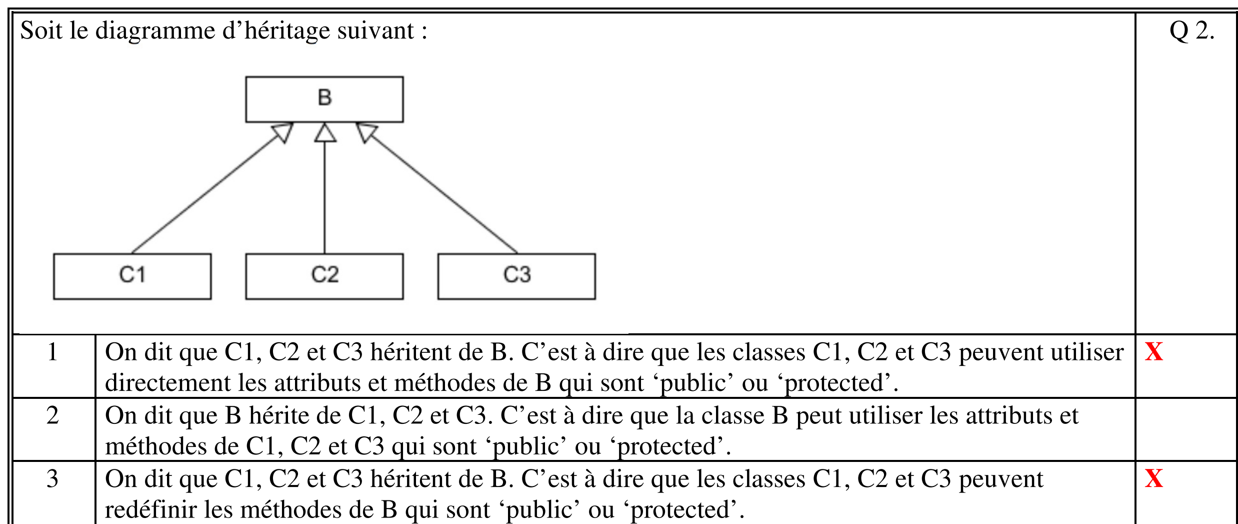


Le lien d'association entre A et B, et entre A et C est un lien de :

1	agrégation	
2	composition	X

Q 2.

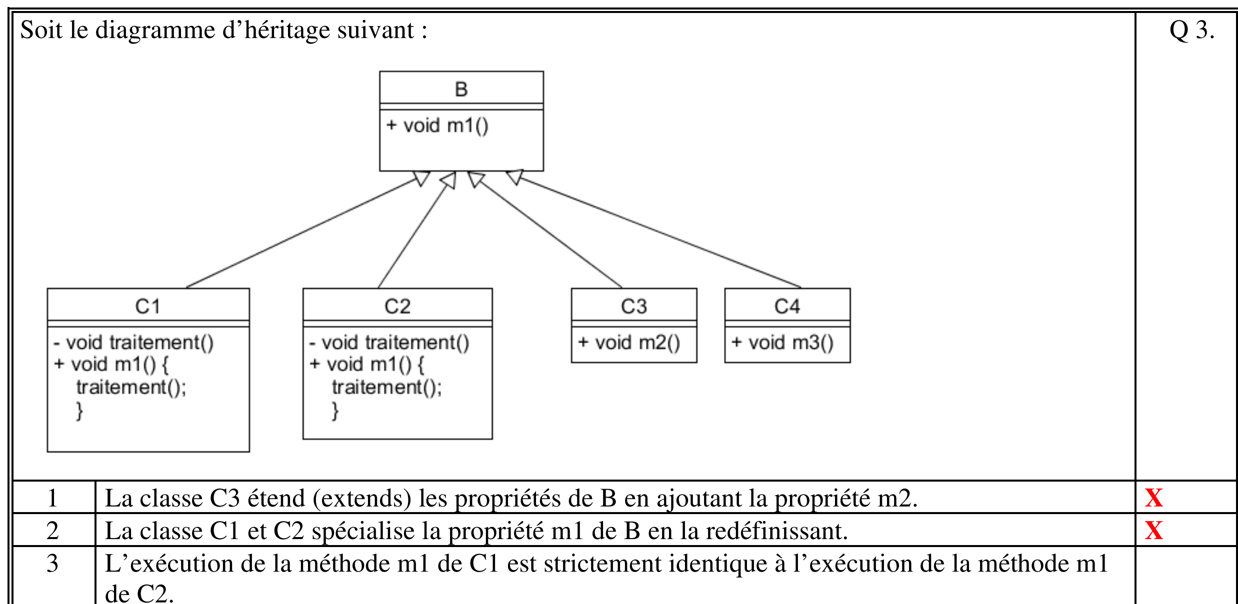
Soit le diagramme d'héritage suivant :



1	On dit que C1, C2 et C3 héritent de B. C'est à dire que les classes C1, C2 et C3 peuvent utiliser directement les attributs et méthodes de B qui sont 'public' ou 'protected'.	X
2	On dit que B hérite de C1, C2 et C3. C'est à dire que la classe B peut utiliser les attributs et méthodes de C1, C2 et C3 qui sont 'public' ou 'protected'.	
3	On dit que C1, C2 et C3 héritent de B. C'est à dire que les classes C1, C2 et C3 peuvent redéfinir les méthodes de B qui sont 'public' ou 'protected'.	X

Q 3.

Soit le diagramme d'héritage suivant :



1	La classe C3 étend (extends) les propriétés de B en ajoutant la propriété m2.	X
2	La classe C1 et C2 spécialise la propriété m1 de B en la redéfinissant.	X
3	L'exécution de la méthode m1 de C1 est strictement identique à l'exécution de la méthode m1 de C2.	

En programmation objet, le principe d'héritage permet de :		Q 4.
1	factoriser les attributs de plusieurs classes en une classe dont elles héritent.	X
2	spécialiser une classe en créant une classe dérivée qui redéfinit certaines des méthodes de la classe d'héritage.	X
3	rendre plus performant en terme d'exécution le programme objet.	

En Java, une classe A peut accéder à un attribut d'une autre classe B (accès direct, sans utiliser de getteur) quand :		Q 5.
1	A hérite de B et que cet attribut est déclaré protected.	X
2	cet attribut est public.	X
3	A hérite de B et que cet attribut est déclaré private.	

En programmation objet, une collection est polymorphe si le type de déclaration des éléments de la collection est une interface ou une classe abstraite.		Q 6.
1	OUI	X
2	NON	

En programmation objet, toutes les méthodes d'une classe abstraite doivent être toutes abstraites.		Q 7.
1	OUI	
2	NON	X

Soit la classe C1 qui hérite de C2. C1 n'est pas une classe abstraite. C2 est une classe abstraite. — Q 8.
Soit les constructeurs de C1 et C2 :

```
public C1(int x){super(x) ;… }
public C2(int y){  … }
public C2(){  … }
```

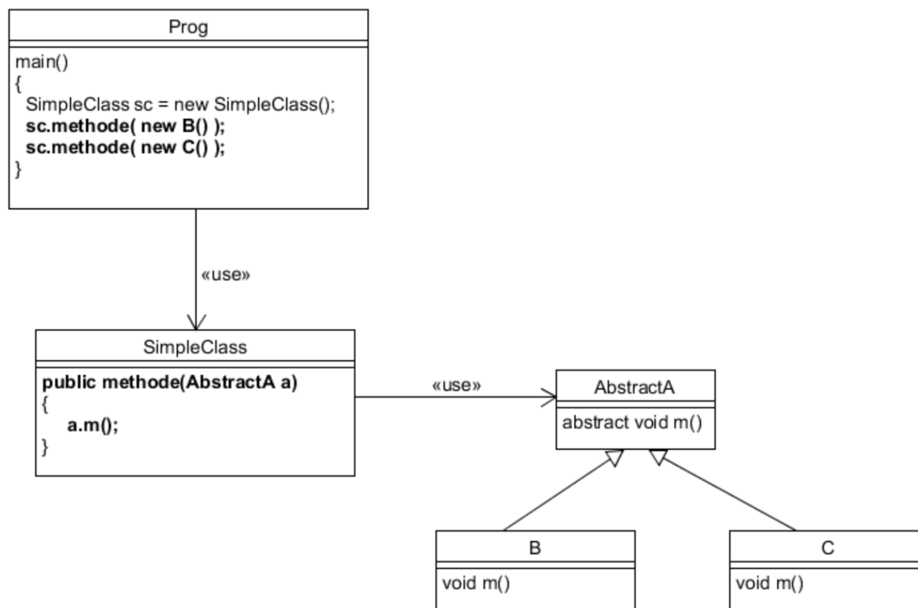
		Q 8.
1	`C1 objetC1 = new C1(100) ;` Ce code est correct.	X
2	`C1 objetC1 = new C1() ;` Ce code est correct	
3	`C2 objetC2 = new C2(10) ;` Ce code est correct	

En programmation objet, un traitement générique est une méthode dont certains des paramètres sont des déclarations d'interface ou de classe abstraite.		Q 9.
1	OUI	X
2	NON	

Soit la classe A qui hérite de la classe B abstraite. Un traitement générique est une méthode de la classe A qui utilise les méthodes abstraites de la classe B.		Q 10.
1	OUI	
2	NON	X

Q 11.

Soit le schéma suivant :

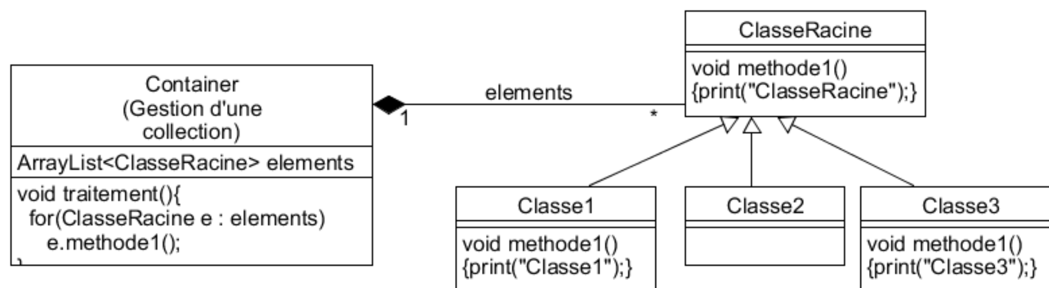


Ce schéma représente une définition d'une méthode abstraite dans laquelle « methode » est la méthode abstraite.

1	OUI	X
2	NON	

Q 12.

Soit le schéma suivant :



Le résultat de l'exécution du code suivant :
{elemnents.add(new Classe1()) ;
elemnents.add(new Classe2()) ;
elemnents.add(new Classe3()) ;
traitement() ;}
est :

1	ClasseRacine ClasseRacine ClasseRacine	
2	Classe1 ClasseRacine Classe3	X
3	Classe1 Classe3	

Q 13.

En JAVA, l'Interface Iterable est une interface qui doit être implémentée par toutes les classes de Collection

1	OUI	X
2	NON	

Le Design Pattern Itérateur (en anglais Iterator) est un DP qui permet de réaliser un processus d'itération en informatique (hasNext(), next()).		Q 14.
1	Ce processus s'applique uniquement sur les collections.	
2	Ce processus peut s'appliquer sur n'importe quelle classe.	X

En JAVA, les classes AbstractList et AbstractSet sont toutes les deux des collections dont les éléments sont indexés (méthode get(i) qui retourne le i-ième élément de la collection).		Q 15.
1	OUI	
2	NON	X

En Java, la classe Collections permet de trier les éléments de n'importe quelle collection (classe qui implémente l'interface List) grâce à la méthode **Collections.*sort*(List<T> list)** Cette méthode fonctionne si la classe d'appartenance, ici T, des éléments de la collection, implémente l'interface :		Q 16.
1	Comparator	
2	Comparable	X
3	Comparer	

En Java, la méthode suivante permet de trier les éléments d'une collection : **Collections.*sort*(List<T> list, Comparator<? super T> c)**		Q 17.
1	Comparator est ici une classe prédéfinie du langage Java qui permet de comparer les éléments de la collection.	
2	Comparator est ici une classe abstraite du langage Java qu'il faut dériver en une classe qui surcharge la méthode int compare(T o1, T o2) qui est utilisée pour comparer les éléments de la collection.	
3	Comparator est ici une interface qui contient la méthode int compare(T o1, T o2) qui est utilisée pour comparer les éléments de la collection.	X

Les traitements réalisés sur les éléments d'une collection polymorphe dont le type d'élément est une classe abstraite sont tous des traitements génériques.		Q 18.
1	OUI	X
2	NON	

Les "patrons de conception" ou Design Patterns" sont des modèles standard et réutilisables de conception d'une solution à la problématique de la réalisation d'une partie d'un logiciel informatique.		Q 19.
1	OUI	X
2	NON	

Le rôle du Design Pattern **Singleton** est la création unique d'une instance d'une classe.		Q 20.
1	OUI	X
2	NON	

Soit le code suivant d'implémentation d'un singleton : **Q 21.**

```
public class SingletonXXX
{
    static private SingletonXXX sg = null;

    private SingletonXXX () { }

    static public SingletonXXX getSingletonXXX()
    {
        if (sg==null) sg =new SingletonXXX();
        return sg ;
    }
}
```

Ce code est correct.

1	OUI	X
2	NON	

Le rôle du DP Singleton est de :		Q 22.
1	limiter les effets de bord dans un programme informatique en centralisant dans une même classe toutes les variables globales du programme.	
2	limiter le nombre d'instance d'une classe qui, dans le cas d'un singleton, sera toujours égal à 1.	X
3	pouvoir accéder à un objet principal et unique de n'importe où dans le code sans avoir besoin de le passer en paramètre ou en attribut d'un objet.	X

Le rôle du DP Factory est de :		Q 23.
1	pour un utilisateur du Factory, d'abstraire la façon dont un objet est créé.	X
2	pour un utilisateur du Factory, de maitriser la façon dont un objet est créé.	

Un principe de base du DP Factory est de retourner l'objet créé sous la forme d'une interface ou sous la forme d'une classe abstraite.		Q 24.
1	OUI	X
2	NON	

Le rôle du DP Factory est de créer des objets qui sont vus par le reste du programme comme des singletons :		Q 25.
1	OUI	
2	NON	X

Le schéma suivant est celui du DP Factory : **Q 26.**

La signification des lettres A, B, C et D est :

1	A = Client; B = Produit concret; C = Produit abstrait (Interface); D = Factory	
2	A = Client; B = Factory; C = Produit abstrait (interface); D = Produit concret	
3	A = Client; B = Factory; C = Produit concret; D = Produit abstrait (Interface)	X

Le rôle du Design Pattern Observateur est de créer, dynamiquement des objets dont la classe d'appartenance implémente l'interface Observer		Q 27.
1	OUI	
2	NON	X

En JAVA, le DP Observateur est implémenté de telle manière que :		Q 28.
1	L'observateur est une classe quelconque qui implémente l'interface prédéfinie Observer	X
2	La classe qui notifie les observateurs est une classe quelconque qui hérite de la classe prédéfinie Observable	X
3	La classe qui notifie les observateurs est la classe prédéfinie Observable qu'il est nécessaire d'instancier.	

En JAVA, le DP Observateur permet de notifier une donnée informatique à tous les observateurs. Cette donnée est :		Q 29.
1	de type Observer	
2	de type Argument	
3	de type Object	X

En JAVA, pour qu'un observateur puisse être notifié, il est nécessaire que :		Q 30.
1	l'observateur implémente la méthode'update' définie dans l'interface Observer.	X
2	l'observateur s'abonne à l'observable.	X
3	l'observateur hérite de la classe prédéfinie Observer.	

Q 31.

Ce schéma représente le DP Modèle-Vue-Controleur. Les lettres A, B et C sont définies ainsi :		Q 31.
1	A = Modèle; B = Vues ; C = Contrôleur	X
2	A = Contrôleur; B = Vues; C = Modèle	
3	A= Modèle ; B = Contrôleur ; C = Vues	

Dans le DP MVC (Modèle, Vue, Contrôleur) la notification d'informations du Modèle vers les Vues peut être implémenté sous la forme d'un DP Observateur.		Q 32.
1	OUI	X
2	NON	

Le socket est un canal de communication permettant de faire communiquer deux logiciels.		Q 33.
1	Ces deux logiciels peuvent être situés sur deux machines différentes connectés en réseau	X
2	Ces deux logiciels peuvent être situés sur la même machine	X
3	Ces deux logiciels doivent être situés sur des machines différentes	

Soit le code du serveur suivant :
`ServerSocket ssoc = new ServerSocket(9100);`
`Socket soc = ssoc.accept();`
`DataInputStream dis= new DataInputStream(soc.getInputStream());`
`String requete = dis.readUTF();`
		Q 34.

1	dès son exécution la méthode *accept* retourne un socket. Le serveur se met alors en attente de lecture avec l'instruction dos.readUTF(), il attend qu'un client écrit sur le socket	
2	dès son exécution la méthode *accept* se met en attente qu'un client crée un socket sur le port 9100.	X

En JAVA, la communication socket se fait en écrivant et en lissant des informations dans le socket. Ces informations :		Q 35.
1	peuvent être de type quelconque.	X
2	doivent être de type Object.	
3	doivent être de type String.	

2. Questions libres (15 points)

Chaque question est notée sur 5 points.
*Vous répondez à ces questions sur une **copie vierge** en mettant bien le numéro de la question, sans oublier votre nom et prénom.*

QUESTION 1 :
Sur les principes de conception d'un programme objet, nous avons listé les différentes étapes à réaliser suivantes :
1/ Lire le cahier des charges
2/ Comprendre le besoin et l'enrichir si nécessaire
3/ Identifier les cas d'utilisation
4/ Définir l'IHM (les écrans et leurs contenus et les actions utilisateurs) et l'interface avec l'Applicatif
5/ Faire le diagramme de classe de la solution
6/ Faire le développement (code) de toutes les classes
7/ Tester unitairement les classes et/ou tester par étapes d'intégration
8/ Faire les tests en fonction des cas d'utilisation identifiés

Parmi ces étapes, quelles sont pour vous les deux étapes les plus importantes à réaliser ? Justifiez votre choix.
(Toute réponse est juste car cela est une appréciation personnelle. Vous serez noté sur la pertinence de votre justification).

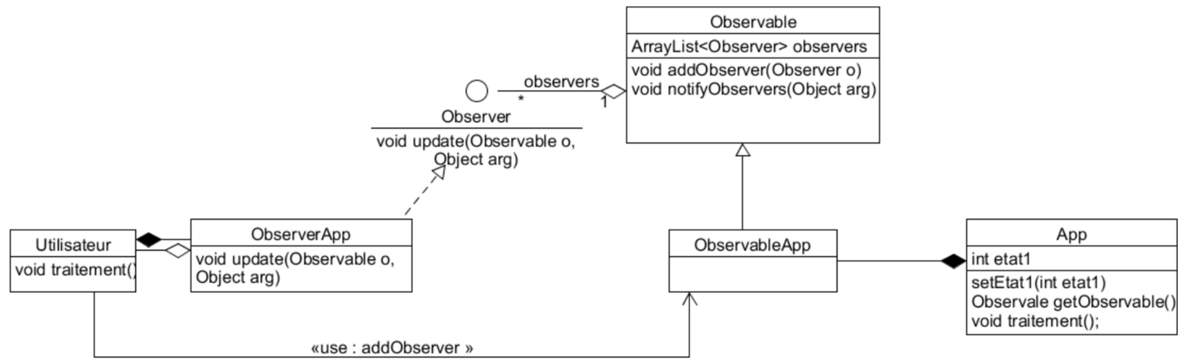
Quelques possibilités :

L'étape la plus importante est de comprendre le besoin cela assure une base solide, une confiance du client, de déterminer un terrain commun connu des deux parties, d'assurer de faire les bons choix, de discuter des alternatives possibles en cas de difficultés, d'assurer une base solide pour l'écriture des scénarios et des tests.

L'étape la plus importante est l'étape du développement car sans cette étape rien n'existe. Elle est celle de tous les dangers, la plus difficile techniquement. Son importance est à la hauteur de sa complexité. Elle nécessite des compétences multiples, une organisation nécessitant de rendre efficace une équipe souvent importante.

L'étape la plus importante est l'étape des tests car si les tests sont conformes cela signifie que le code est correct. Il est normal de faire des bugs. Il est « impardonnable » de les laisser passer. A condition que les tests soient conformes aux besoins.

QUESTION 2 :
a) Faites le schéma de diagramme de classe du Design Pattern Observateur.
b) Expliquez, avec précision, le fonctionnement de ce Design Pattern.
c) Expliquez un cas d'utilisation (autre que celui du projet) de ce Design Pattern.



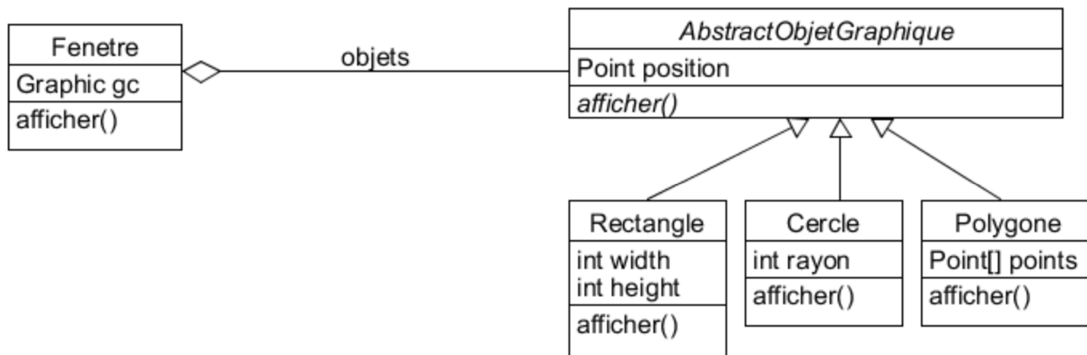
Tout utilisateur crée une instance de ObserverApp qui implémente l'interface Observer. Cette instance s'abonne à l'Observable via la méthode addObserver.
Plusieurs utilisateurs peuvent faire cela.
La méthode addObserver consiste à ajouter l'observateur dans la liste des observateurs gérée par l'Observable.
L'App crée une instance de ObservableApp qui hérite de Observable. L'app va pouvoir notifier un objet à tous les observateurs en utilisant la méthode notifyObservers. Cette méthode boucle sur tous les observateurs en appelant la méthode update qui est implémenté par tous les observateurs.

Un cas d'utilisation peut être, par exemple, des tableaux d'affichage d'horaire d'arrivée des trains répartis dans une gare de grande importance. Tous ces tableaux sont des observateurs qui s'abonnent à un observable utilisé par un poste d'IHM qui permet de saisir les horaires d'arrivée des trains.

QUESTION 3 :
a) Donnez une définition d'un « traitement générique » dans le cadre de la programmation objet.

En programmation objet, un traitement générique, est une méthode qui contient un algorithme qui utilise des méthodes sur des objets qui sont déclarés par des classes abstraites ou des interfaces. Ces objets sont, soit passés en paramètre de la méthode générique, soit des attributs de la classe d'appartenance de la méthode générique.

b) Faites le diagramme de classe d'un exemple concret de l'utilisation d'un traitement générique. Commentez votre diagramme.



La méthode afficher de la classe Fenetre est la méthode générique. On est dans le cas de la réalisation d'un traitement sur une collection polymorphe.

La méthode afficher de la classe Fenetre boucle sur les objets graphiques afin d'appeler la méthode afficher de la classe abstraite AbstractObjetGraphique. En fonction du type concret de l'objet c'est la méthode concrète de chacune des classes dérivées qui est exécutée.

FIN DE LA 1ère PARTIE

2ème PARTIE : PROGRAMMATION (avec document)
Durée: 1h15

EXERCICE sur 20 points

On veut réaliser un programme informatique qui permet de jouer à un jeu de Quiz en solitaire (1 seul joueur).

Le programme sera constitué de deux IHM indépendantes :

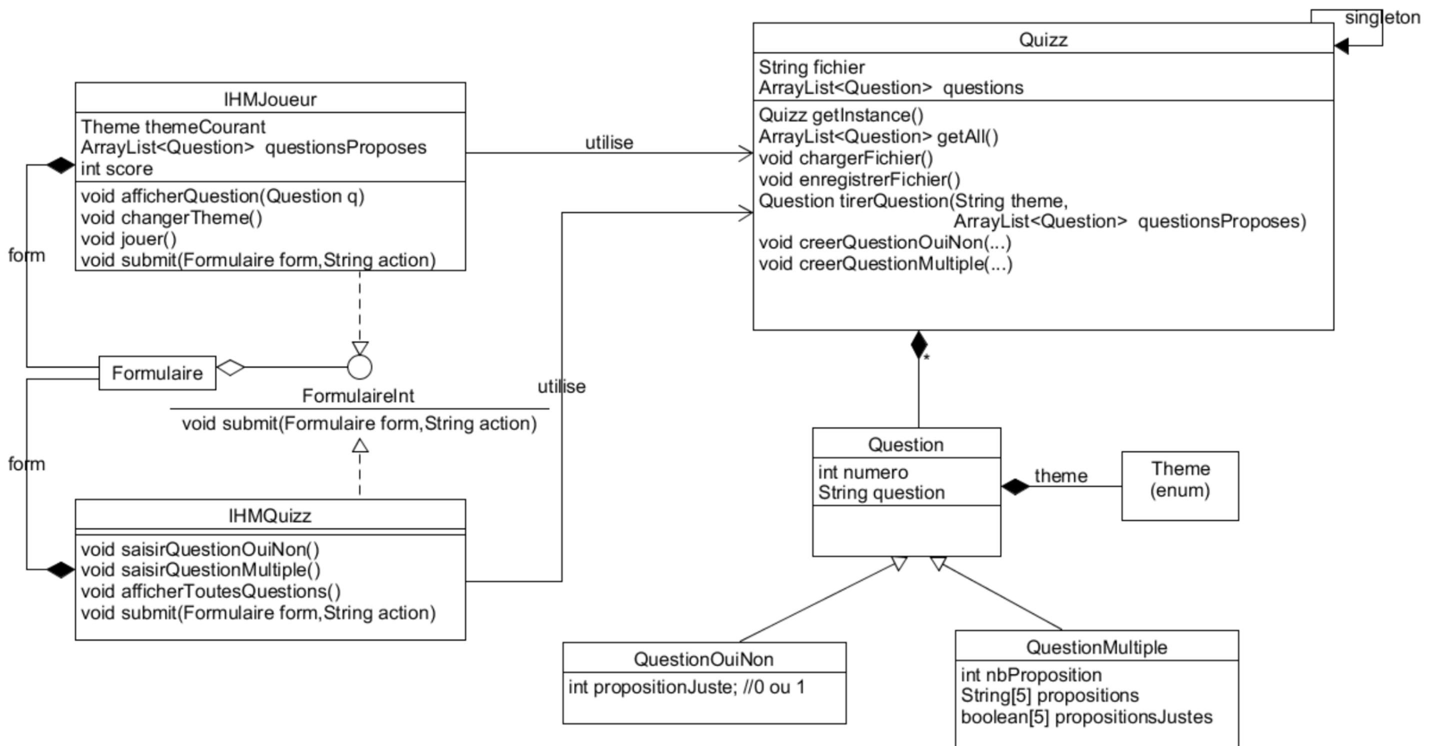
- une IHM permettant de créer les questions et leurs réponses
- une IHM permettant de jouer au jeu.

Les questions sont de deux natures. Celles pour lesquelles il faut répondre par oui ou par non, et celles pour lesquelles, il faut choisir les propositions justes parmi au plus 5 propositions.
Un score est calculé au fur et à mesure de l'exécution du programme : +1 point pour chaque réponse juste, -1 point pour chaque réponse fausse.
Toutes les questions appartiennent à un thème (Histoire, Géographie, Littérature, Loisir).

La première IHM permet d'enrichir les questions qui sont stockées dans un fichier.

La deuxième IHM lit toutes les questions du fichier puis propose, pour un thème choisi par le joueur, une question tirée aléatoirement en ne proposant jamais une même question pendant une exécution du programme.
Le joueur peut changer de thème en cours de jeu.

Faites le diagramme de classe complet de ce programme informatique. Soyez précis dans les attributs et les méthodes des classes. Utilisez des identificateurs clairs. N'hésitez pas à ajouter des commentaires, après le schéma, si vous pensez qu'ils sont nécessaires pour comprendre les choix que vous avez faits.



La classe Quizz est un singleton qui gère toutes les questions du fichier qui sont chargées en mémoire.
La méthode tirerQuestion permet de tirer aléatoire une question d'un thème donné qui ne fait pas partie des questions qui ont été proposées.

La méthode jouer de la class IHMJoueur permet de jouer au quizz. Elle tire une question dans Quizz, affiche la question. En fonction des choix faits dans la question, sa validation calcule le score en comparant les choix et les propositions justes de la question.

Les questions contiennent le texte de la question, les propositions de la question et les propositions justes. Héritage des questions des 2 différents types.

L'IHMQuizz permet de saisir des questions. En validant la question, la méthode submit utilise les méthodes de Quizz pour créer les questions dans la collection.

PROBLEME sur 30 points

On veut réaliser un programme de supervision météorologique très minimaliste.

Faire le code complet de ce programme informatique, implémenté sous la forme du modèle **MV** (il n'y a pas de Contrôleur) et en utilisant le Design Pattern **Observateur** pour réaliser la notification entre le Modèle et les Vues.

Le modèle est une instance unique de la classe **CentraleMeteo** qui contient les méthodes suivantes :

```
public void setTemperature(String nomDuCapteur, Position latlong, double valeur) ;
public void setPression(String nomDuCapteur, Position latlong, double valeur) ;
public void setForceVent(String nomDuCapteur, Position latlong, Angle direction, int force) ;
```

Vous n'écrivez pas le code des classes Position et Angle.

A chaque fois qu'une de ces méthodes est exécutée le modèle notifie, à toutes les vues, un objet de type **Temperature**, **Pression** ou **ForceVent**. Ces classes héritent de la classe **Capteur** qui contient le nom du capteur.
Le modèle gère une collection polymorphe de **Capteur**. *Les méthodes pour gérer cette collection ne sont pas à coder (sauf si vous avez le temps).*

Une vue est une instance de la classe **SupervisionMeteo** qui prend en entrée une liste de noms de capteur. Le rôle de la vue est d'afficher le nom et la valeur de chacun des capteurs en entrée. La valeur est mise à jour en fonction des notifications. La valeur est affichée sous la forme d'une chaîne de caractère qui est la concaténation des informations de Temperature (position, valeur), Pression (position, valeur) ou ForceVent (position, angle, force).

Pour coder la classe **SupervisionMeteo**, vous utilisez la classe prédéfinie **Formulaire** que nous avons utilisée tout au long de l'année.

Pour tester le programme, on crée le programme principal suivant qui crée 2 vues, une pour LYON et une pour PARIS :

```
public class ProgMeteo {
    public static void main(String[] args) {
        ArrayList<String> capteurs = new ArrayList<String>();
        capteurs.add("VENT LYON SUD");
        new SupervisionMeteo(capteurs);  // Création d'une vue qui s'affiche

        capteurs = new ArrayList<String>();
        capteurs.add("TEMP PARIS SUD");
        capteurs.add("PRESSION PARIS NORD");
        new SupervisionMeteo(capteurs);  // Création d'une autre vue qui s'affiche

        // Pour simuler la mise à jour du modèle
        int force=1; double temperature = 30.0; double pression = 170;
        while(true) {
            CentraleMeteo.getInstance().setForceVent("VENT LYON SUD",
                new Position(12.3,-5.6), new Angle(90), force);
            CentraleMeteo.getInstance().setTemperature("TEMP PARIS SUD",
                new Position(2.3,45.6), temperature);
            CentraleMeteo.getInstance().setPression("PRESSION PARIS NORD",
                new Position(2.4,45.6), pression);
            force++; temperature=temperature+0.1; pression = pression + 1;
            try {Thread.sleep(1000);;}catch(Exception ex) { } // Changement toutes les secondes
        }}}
```

10 points

```
public class CentraleMeteo {
	private static CentraleMeteo singleton=null;
	public static CentraleMeteo getInstance() {
		if (singleton==null) singleton = new CentraleMeteo();
		return singleton;
	}
	private Hashtable<String,Capteur> capteurs;
	private MeteoObservable observable;

	private CentraleMeteo() {
		capteurs = new Hashtable<String,Capteur>();
		observable = new MeteoObservable();
	}

	public void sabonner(Observer o) {
		observable.addObserver(o);
	}

	public void setTemperature(String nomDuCapteur, Position latlong, double
valeur) {
		observable.notifier(new Temperature(nomDuCapteur,latlong,valeur));
	}

	public void setPression(String nomDuCapteur, Position latlong, double
valeur) {
		observable.notifier(new Pression(nomDuCapteur,latlong,valeur));
	}

	public void setForceVent(String nomDuCapteur, Position latlong, Angle
direction, int force) {
		observable.notifier(new ForceVent(nomDuCapteur, latlong,d irection,
force));
	}
}
```

2 points

```
public class MeteoObservable extends Observable {
	public void notifier(Object obj) {
		setChanged();
		notifyObservers(obj);
	}}
```

2 points

```
public class Capteur {
	private String nom;
	private Position latlong;
	public Capteur(String nom,Position latlong) {this.nom=nom;this.latlong=latlong ;}
	public String toString() {return nom;}
}
```

2 points

```
public class ForceVent extends Capteur {
	private Angle direction;
	private int force;
	public ForceVent(String nom, Position latlong, Angle direction, int force) {
		super(nom,latlong);
		this.latlong = latlong;
		this.direction = direction;
		this.force = force;
	}
```

```
	public String toString() {
		return latlong.getLatitude()+" "+latlong.getLongitude()+ " "+
direction.getAngle() + " " + force;
}}
```

---

2 points

```
public class Pression extends Capteur {
	private double valeur;

	public Pression(String nom, Position latlong, double valeur) {
		super(nom,latlong);
		this.latlong = latlong;
		this.valeur = valeur;
	}

	public String toString() {
		return latlong.getLatitude()+" "+latlong.getLongitude()+" "+valeur;
	}}
```

---

2 points

```
public class Temperature extends Capteur {
	private double valeur;

	public Temperature(String nom, Position latlong, double valeur) {
		super(nom,latlong);
		this.latlong = latlong;
		this.valeur = valeur;
	}

	public String toString() {
		return latlong.getLatitude()+" "+latlong.getLongitude()+" "+valeur;
   }}
```

---

10 points

```
public class SupervisionMeteo implements Observer , FormulaireInt {
	private Formulaire form;

	public SupervisionMeteo(ArrayList<String> listeCapteurs){
		form =new Formulaire("METEO",this,500,600,true);

		CentraleMeteo.getInstance().sabonner(this);

		for(String capteur:listeCapteurs) {
			form.addText(capteur,capteur, false, "");
		}
		form.afficher();
	}

	public void update(Observable o, Object arg) {
		Capteur capteur = (Capteur)arg;
		form.setValeurChamp(capteur.getNom(), capteur.toString());
     }}
```

(Fin du sujet)