IPST-CNAM
Intranet et Designs patterns
NSY 102
Vendredi 17 Avril 2015

Durée : **2 h 45**
Enseignants : LAFORGUE Jacques

1ère Session NSY 102

CORRECTION

1ère PARTIE – 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:

Une architecture Client-Serveur 2-tiers utilise un Middleware		Q 1.
1	OUI	**X**
2	NON	

Un ORB (Object Request Broker) est un Middleware		Q 2.
1	OUI	**X**
2	NON	

Une application dite "distribuée" est une application logicielle orientée objet dans lequel des objets sont accessibles de manière distante		Q 3.
1	OUI	**X**
2	NON	

Une architecture 3-Tiers est une architecture dans laquelle les composants métier (une couche) et les composants d'accès aux données (une deuxième couche) sont disjoints (communication distante via un middleware, par exemple)		Q 4.
1	OUI	**X**
2	NON	

Un "objet distribué" est une instance d'une classe d'un programme A dont certaines méthodes peuvent être appelées par un autre programme B.		Q 5.
1	Ces méthodes ne peuvent pas être appelées par A	
2	Ces méthodes doivent être static	
3	Les deux programmes A et B peuvent être sur deux machines différentes	**X**

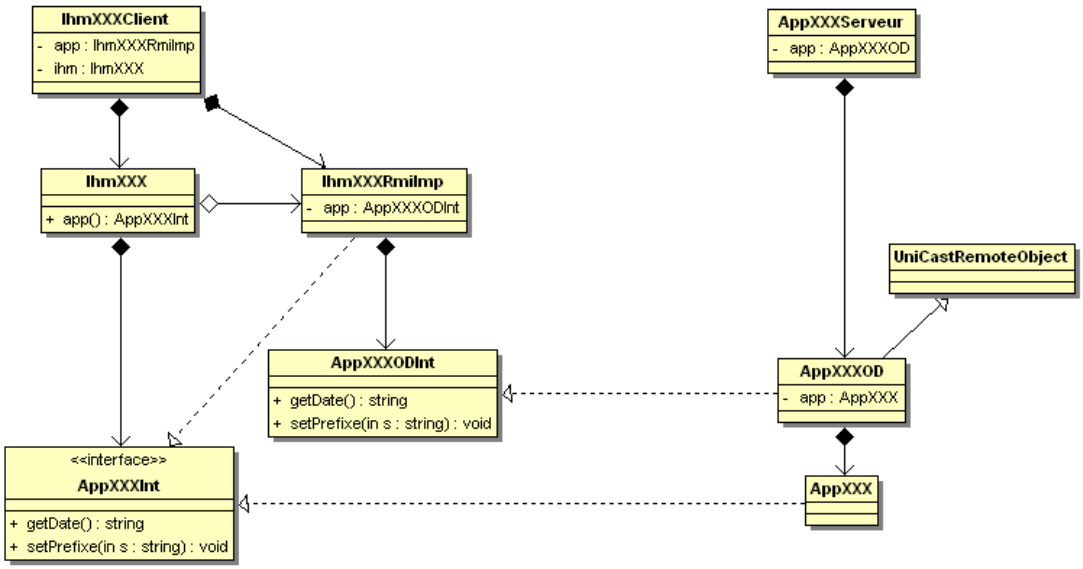
2 clients (A et B) peuvent appeler en même temps la méthode distante m1 d'un même objet distribué.		Q 6.
1	OUI	**X**
2	NON	

Un serveur stocke les stubs de ses objets distribués dans un fichier binaire. Un client reçoit ce fichier, lit les stubs et peut se connecter aux objets distribués du serveur.		Q 7.
1	OUI	**X**
2	NON	

Soit un objet quelconque Obj (instance de la classe A qui n'hérite pas d'une autre classe). En Java RMI, il est très facile de transformer cet objet en un objet distribué. Pour cela il suffit de :		Q 8.
1	faire que la classe A implémente l'interface Remote	
2	faire que la classe A implémente l'interface Serializable, puis écrire cet objet dans un annuaire RMI	
3	faire hériter A de UnicastRemoteObject, mettre les méthodes dans une interface publique (I) qui hérite de Remote, et faire que la classe A implémente l'interface I	**X**

Ceci est le diagramme de classe (Exemple 04 du cours) d'un système composé d'un client IHM (classe IhmXXX) et de son applicatif (AppXXX) que l'on veut rendre distant.

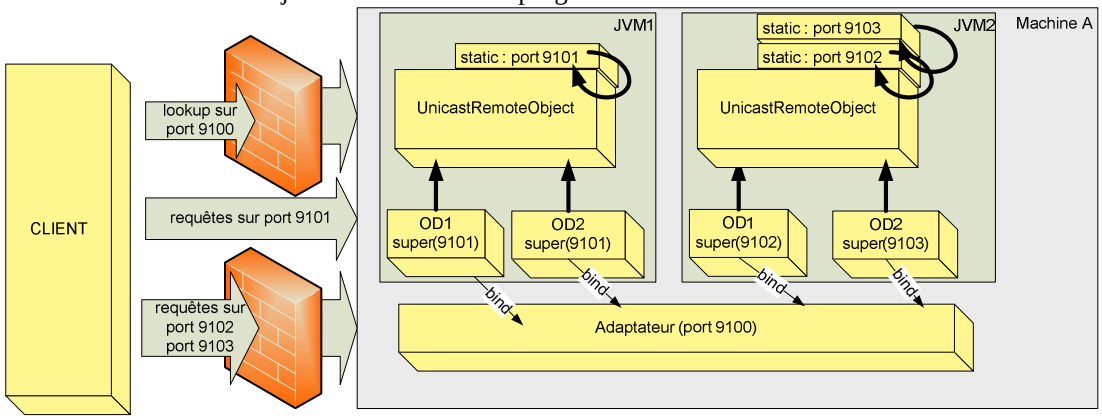
Q 9.



1	IhmXXX est un proxy de AppXXXInt pour AppXXX	
2	AppXXXOD est un Adaptateur de AppXXXODInt pour AppXX	X
3	IhmXXXRmiImp est un proxy de AppXXXODInt pour AppXXX	

Soit le schéma suivant qui représente un fonctionnement possible de plusieurs serveurs de socket des classes UnicastRemoteObject utilisées dans des programmes Java RMI.

Q 10.



1	On peut utiliser une nouvelle machine B dans laquelle on crée une JVM dans laquelle on crée un OD qui s'exécute sur le port 9101	X
2	On peut créer une nouvelle JVM3 dans la machine A dans laquelle, on crée un nouvel OD qui s'exécute sur le port 9104	X
3	Dans la JVM2, on peut créée un nouvel objet distribué RMI sur le port ~~9102~~ 9101	

En Java RMI, le cheminement de la communication (appel distant d'une méthode void) entre un client et un objet distribué se fait de la manière suivante : - le client appelle la méthode d'un "stub" (proxy) qui se connecte à l'adaptateur RMI via un socket - le "stub" écrit sur le socket les informations (nom de la méthode, paramètres, ...) - l'adaptateur lit sur le socket les informations qu'il renvoie à l'objet distribué (serveur de socket) - l'objet distribué lit sur le socket les informations qu'il traduit en l'appel local de la méthode implémentée par l'objet distribué		Q 11.
1	OUI	
2	NON	**X**

Les "patrons de conception" ou Design Patterns" sont des modèles standard et réutilisables de conception de la solution à la problématique de la réalisation d'une partie d'un logiciel informatique.		Q 12.
1	OUI	**X**
2	NON	

Soit le code suivant d'implémentation d'un singleton : Q 13.

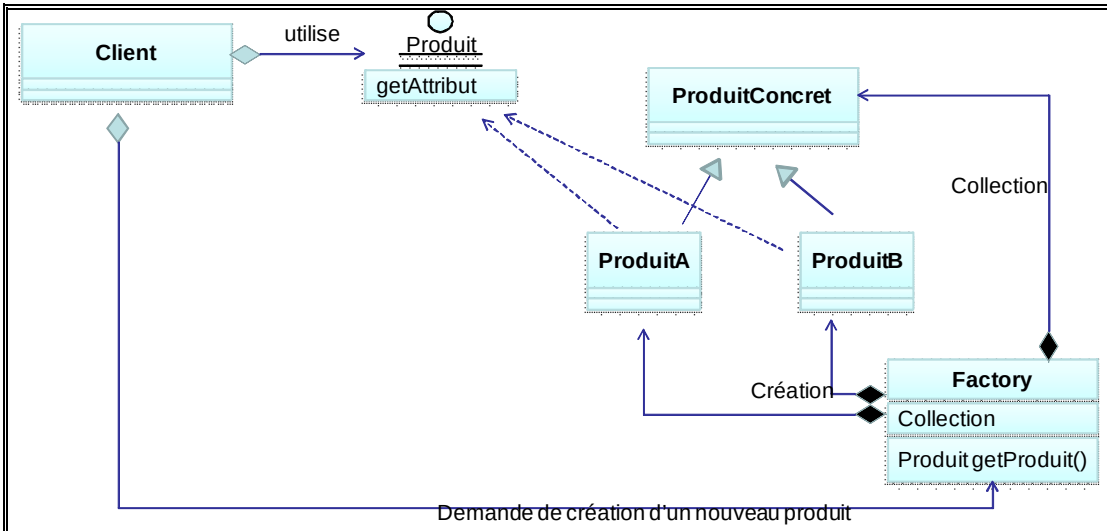
```
public class SingletonXXX
{
    static private SingletonXXX sg = null;

    private SingletonXXX (String[] params) { }

    static public SingletonXXX getSingletonXXX(String[] params)
    {
        if (sg!=null) return sg;
        else return new SingletonXXX (params)
    }
}
```

Ce code est correct.

1	OUI	**X**
2	NON	

Le rôle du DP Singleton est de :		Q 14.
1	créer un objet distribué unique sur le réseau (Singleton d'un Objet Distribué)	
2	limiter le nombre d'instance d'une classe qui dans le cas d'un singleton est 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**

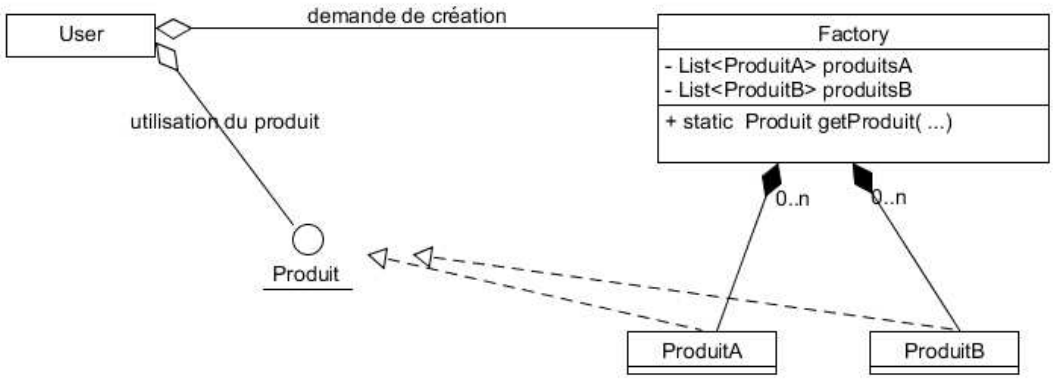
Le rôle du DP Factory est de créer des objets qui sont vus par le reste du programme comme des singletons :		Q 15.
1	OUI	
2	NON	**X**



Q 16.

Ce DP est celui du Factory.
Le rôle des classes ProduitA et ProduitB est :

1	de demander à la classe ProduitConcret de créer (new) des objets de type Produit	
2	de créer (new) des objets de type Produit	X



Q 17.

Ce DP est celui d'un Factory.

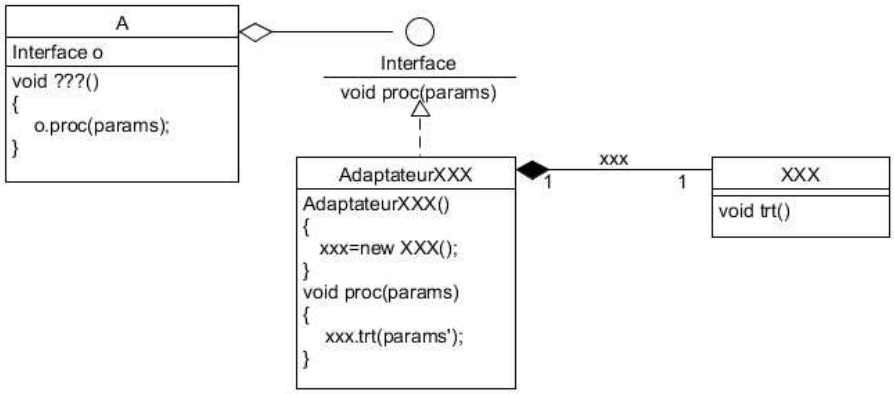
1	OUI	X
2	NON	

Dans le DP Observer, la communication entre l'Observable (producteur d'évènement) et l'Observer (consommateur d'évènement) est :		Q 18.
1	synchrone ou asynchrone (choix de conception)	X
2	toujours asynchrone	
3	toujours synchrone	

La classe prédéfinie JAVA Observable implémente l'interface Observer		Q 19.
1	OUI	
2	NON	X

Le DP Observeur/Observable, peut être utilisé pour réaliser un connecteur Producteur/Consommateur		Q 20.
1	OUI	X
2	NON	

Soit un DP Observeur/Observable, qui est composé de 3 classes ObservableXXX, ProxyObserverXXX, et ObserverXXX. avec : - la classe ObservableXXX qui hérite de Observable et qui notifie les évènements à la classe ProxyObserverXXX - la classe ProxyObserverXX qui implémente l'interface Observer dont la méthode update appelle la méthode update de ObserverXXX - la classe ObserverXXX qui implémente l'interface Observer. Ce DP est-il valide ?		Q 21.
1	OUI	**X**
2	NON	

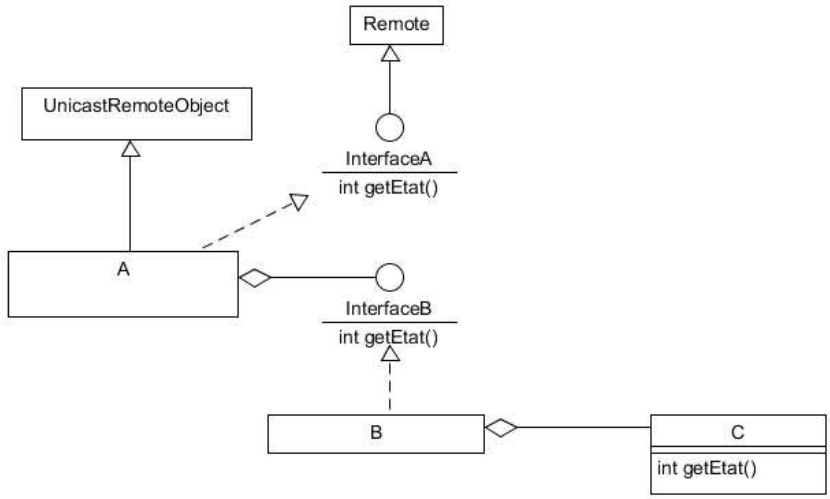
Soit le diagramme de classe suivant :  Ce diagramme de classe représente celui d'un DP Adaptateur		Q 22.
1	OUI	**X**
2	NON	

Le DP proxy est un diagramme de classe dans lequel :		Q 23.
1	deux classes A et B héritent d'une même classe C	
2	deux classes A et B implémentent la même interface, et A a un lien d'agrégation avec B	**X**

Dans la communication asynchrone via un "canal d'évènement" entre un producteur et un consommateur :		Q 24.
1	le producteur utilise un proxy de producteur (et non le producteur directement), afin de lui pousser un évènement	
2	le producteur utilise un proxy de consommateur (et non les consommateurs directement), afin de lui pousser un évènement	**X**

La description suivante est un principe de communication synchrone : chaque producteur dépose son évènement dans une file puis attend que tous les consommateurs aient récupérer l'évènement déposé avant de déposer un nouvel évènement		Q 25.
1	OUI	**X**
2	NON	

Le modèle de communication "Push asynchrone" est un DP dans lequel la classe qui hérite de Observable ~~implémente l'interface Runnable afin de créer un thread qui réalise la notification~~ crée un thread qui réalise l'appel à la notification de l'Observable.		Q 26.
1	OUI	**X**
2	NON	



Q 27.

Ce diagramme de classe est la conception d'un Objet Distribué suivant le modèle :

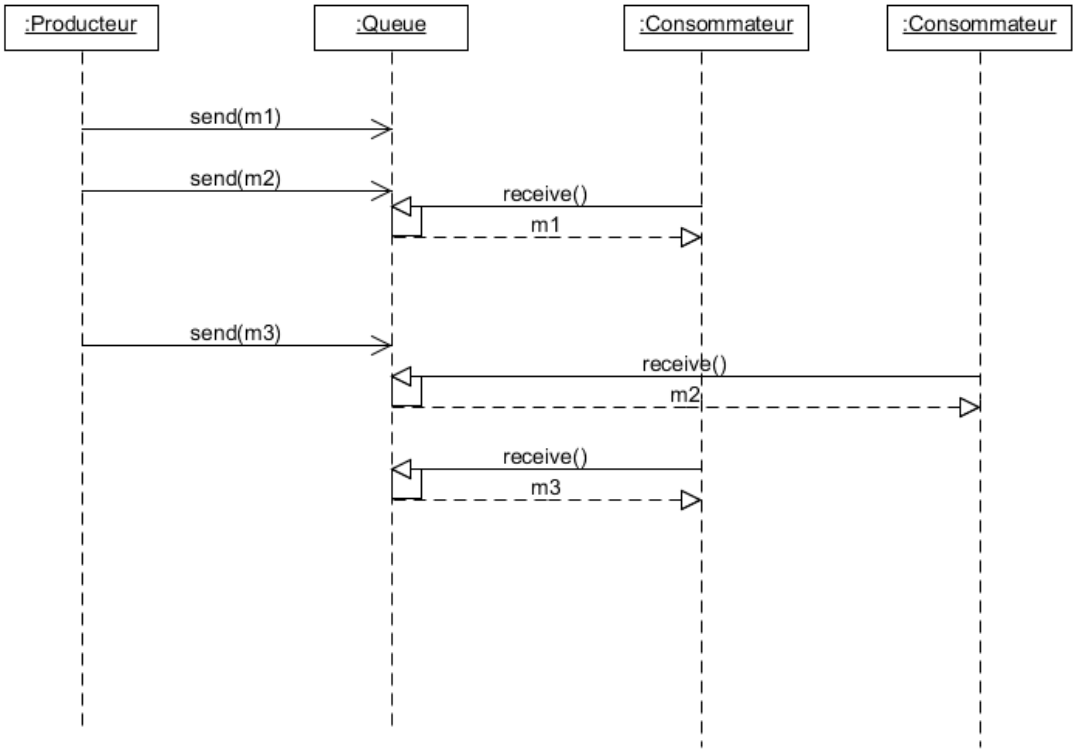
1	d'un DP Proxy	
2	d'un DP Adaptateur	X

Un MOM :		Q 28.
1	est un composant logiciel (Model Orienté Message) qui permet de centraliser l'ensemble des données (Model) d'un système d'information qui sont mises à jour par l'envoi de messages	
2	est une API et des composants dynamiques (Middleware Orienté Message) qui permet certains services d'échanges entre les applications d'un système d'information	X

Le principe d'un MOM est d'utiliser un composant logiciel qui sert d'intermédiaire entre les producteurs et les consommateurs		Q 29.
1	OUI	X
2	NON	

Dans un MOM, les envois de messages sont :		Q 30.
1	toujours synchrone	
2	toujours asynchrone	X
3	synchrone ou asynchrone (en fonction du contexte)	

En JMS (Java Messaging System), il existe (notamment) deux modes de communication : Queue et Topic.		Q 31.



Ce diagramme de transition correspond au mode Queue		
1	Ici, le producteur envoie les messages de manière synchrone à leurs consommations	
2	Ici, le producteur envoie les messages de manière asynchrone à leurs consommations	X
3	Ici, le consommateur consomme les messages de manière asynchrone	X

En JMS (Java Messaging System), les producteurs et les consommateurs sont tous des clients d'un composant logiciel appelé JMS Provider		Q 32.
1	OUI	X
2	NON	

La compilation de fichiers sources écrits en IDL (Interface Definition Language) :		Q 33.
1	permet de générer des fichiers sources qui permettent de mettre en œuvre la programmation des servants d'une application distribuée CORBA	X
2	permet de créer un exécutable qui sert d'intermédiaire de communication entre deux composants d'une architecture CORBA	
3	permet de créer les souches et les squelettes permettant de programmer la communication entre un client et un objet distribué distant	X

Le DP DynamicProxy est utilisé dans l'implémentation de RMI (depuis Java 1.5), pour réaliser les appels distants d'un objet distribué		Q 34.
1	OUI	X
2	NON	

A la différence du DP Proxy, le DP VirtualProxy :		Q 35.
1	crée dynamiquement la classe dont il est le proxy	X
2	peut être le proxy de n'importe quelle classe qui implémente l'interface du proxy	X
3	crée dynamiquement le proxy	

Fin du QCM

Suite (Tournez la page)

2. Questions libres (15 points)

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

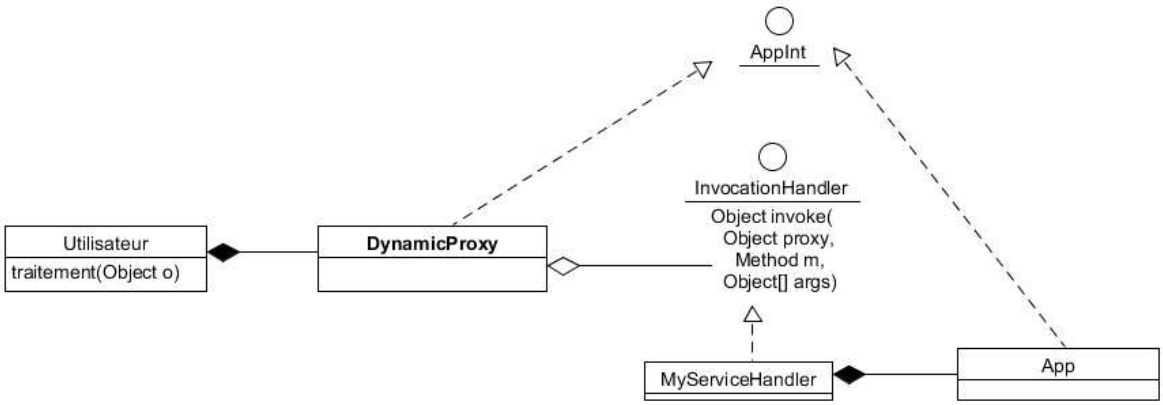
QUESTION NUMERO 1

Dans un MOM (Message Oriented Middleware), il existe deux modes de communication entre les producteurs et les consommateurs. Expliquez ce que sont ces deux modes de communication.

Ces deux mode de communication sont appelés "Queue" et "Topic".
Le principe du mode "Queue" et que le producteur dépose son évènement dans une file de message (intermédiaire). Il est alors à la charge des consommateurs de consommer ces évènements.
Le principe du mode "Topic" est que le producteur donne son évènement à un composant intermédiaire qui connait les consommateurs (ils se sont abonnés au préalable) et leurs envoie l'évènement.

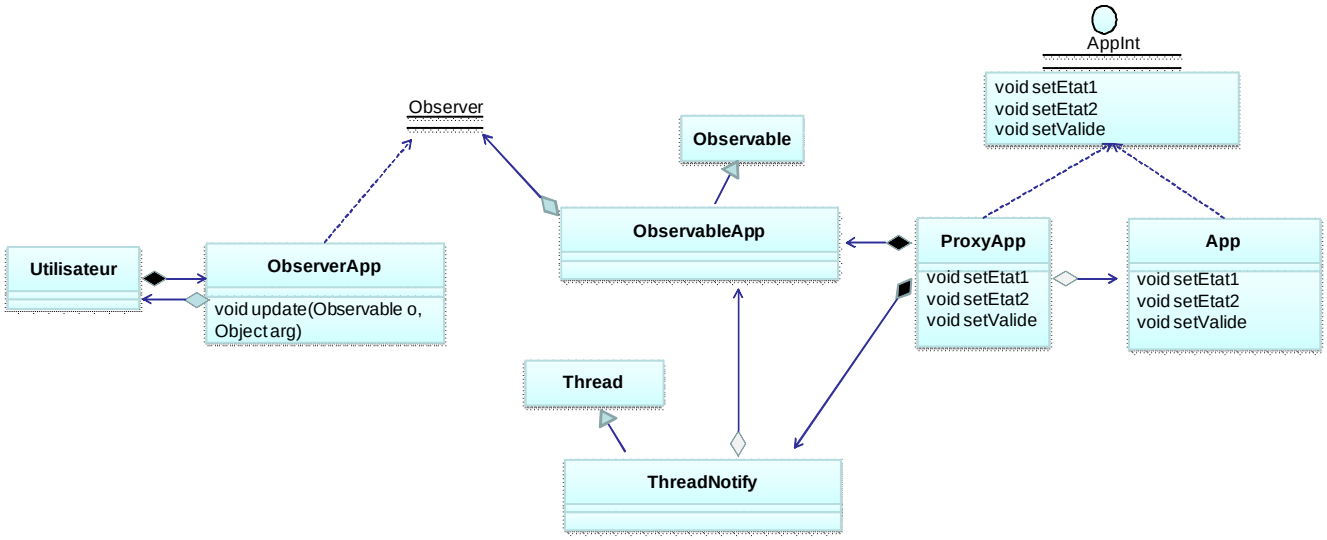
QUESTION NUMERO 2

Expliquez le principe du DP DynamicProxy. Vous pouvez aider votre explication par un diagramme (non obligatoire).
Le DP Dynamic proxy est un DP Proxy dans lequel le proxy n'est pas créé par le programmeur mais créé dynamiquement. Ce proxy implémente toutes les méthodes des interfaces du Dynamic proxy. Le code d'implémentation de toutes ces méthodes est l'appel d'une méthode unique (un invoker) qui doit être implémentée par un objet qui est passé en paramètre du Dynamic Proxy.



QUESTION NUMERO 3

Soit le diagramme de classe suivant vu en cours :



Commentez.

Ce diagramme de classe est la description du DP Observer Push Asynchrone.
Les changements d'état d'un applicatif (App) sont notifier par un Proxy. La notification est faite de manière asynchrone à tous les observers en utilisant un thread.
Il faut que les changements d'état de App soient faites en utilisant les méthodes du ProxyApp.
Le ProxyApp crée un thread ThreadNotify pour chaque notification. Il n'a donc pas besoin d'attendre que tous les Observers aient réalisés leurs traitements (update) : la production des changements d'état est asynchrone à leurs consommations.
L'Observable notifie les observers (ObserverApp).

Fin de la 1ère partie sans document

2ème PARTIE – AVEC DOCUMENT (durée: 1h30)

3. PROBLEME (50 points)

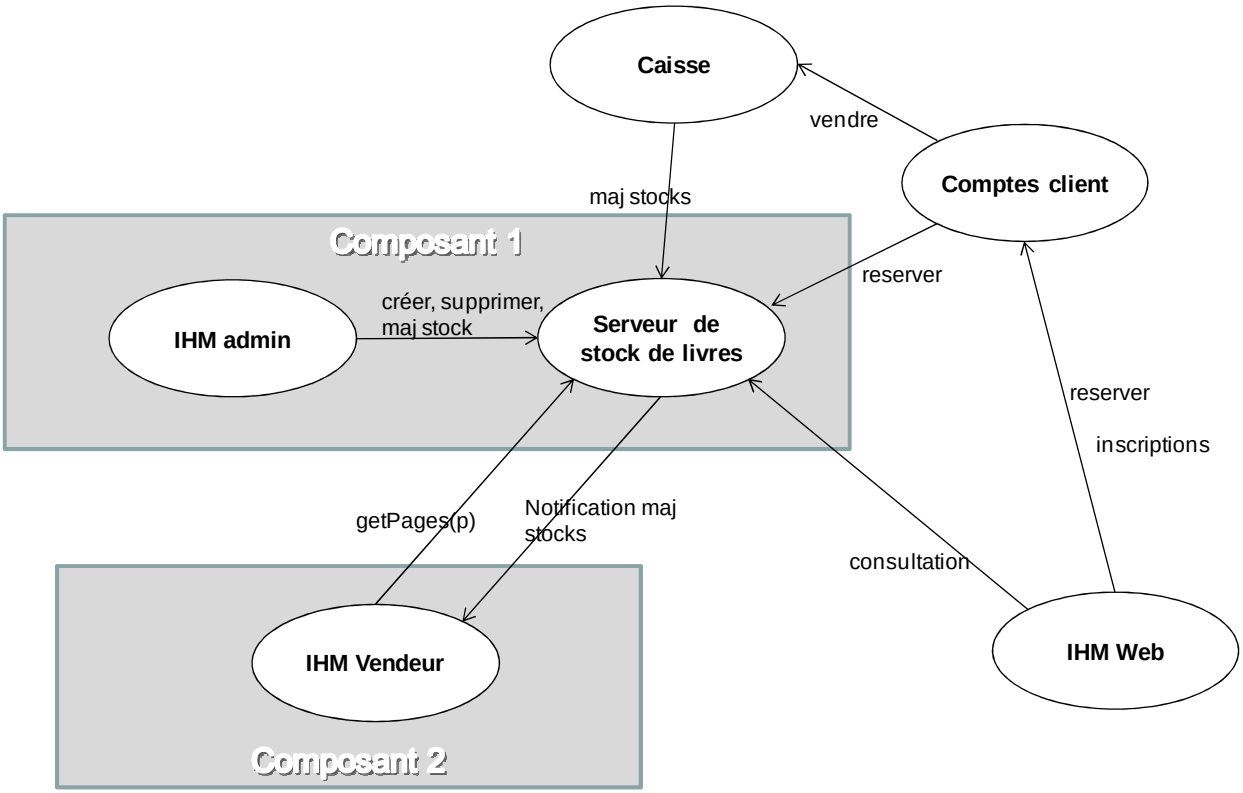
Nous envisageons de réaliser un système d'information permettant de gérer les livres d'un magasin de vente de livre papier qui sont achetés dans le magasin.

Dans le magasin, il y a au minimum :
- un serveur qui gère le stocke de livre (on ne gère pas ici l'aspect base de données. On considère que tous les livres sont en mémoire du serveur) et une ihm (ihm d'administration) qui permet de créer, supprimer un livre ou mettre à jour le nombre en stocke d'un livre. [COMPOSANT 1] (l'ihm et le serveur sont dans le même composant)
- des postes (ihm de vendeur), distants du serveur, utilisés par les vendeurs pour renseigner les clients et donc vérifier si un livre existe et s'il est en stocke. [COMPOSANT 2]
- les caisses de paiement du magasin mettent automatiquement à jour le stocke des livres vendus.

Une personne par internet, utilise une ihm (d'un site internet) permettant de consulter le catalogue des livres du magasin. Il peut alors faire une réservation de livres (s'il est inscrit). Il devra passer les acheter dans un délai de 5 jours.

Pour simplifier le travail à réaliser, nous considérons que :
- les livres chargés en mémoire du serveur sont vus comme un catalogue, de page en page par groupe de 30 livres (par exemple) par page. Pour chaque livre on a sa clef, son titre, ses auteurs et le nombre en stocke.
- l'ihm du vendeur affiche à un moment donné une page en fonction, par exemple, de son numéro. Si, par ailleurs, le nombre en stocke d'un livre est mis à jour alors l'ihm du vendeur est notifiée immédiatement afin de mettre à jour sa page (le livre pouvant être dans la page).
- l' ihm internet demande, en fonction d'un critère de recherche, une liste de livre et l'affiche (seul les livres en nombre de stocke différent de 0 sont retenus) et le client peut alors réserver des livres.

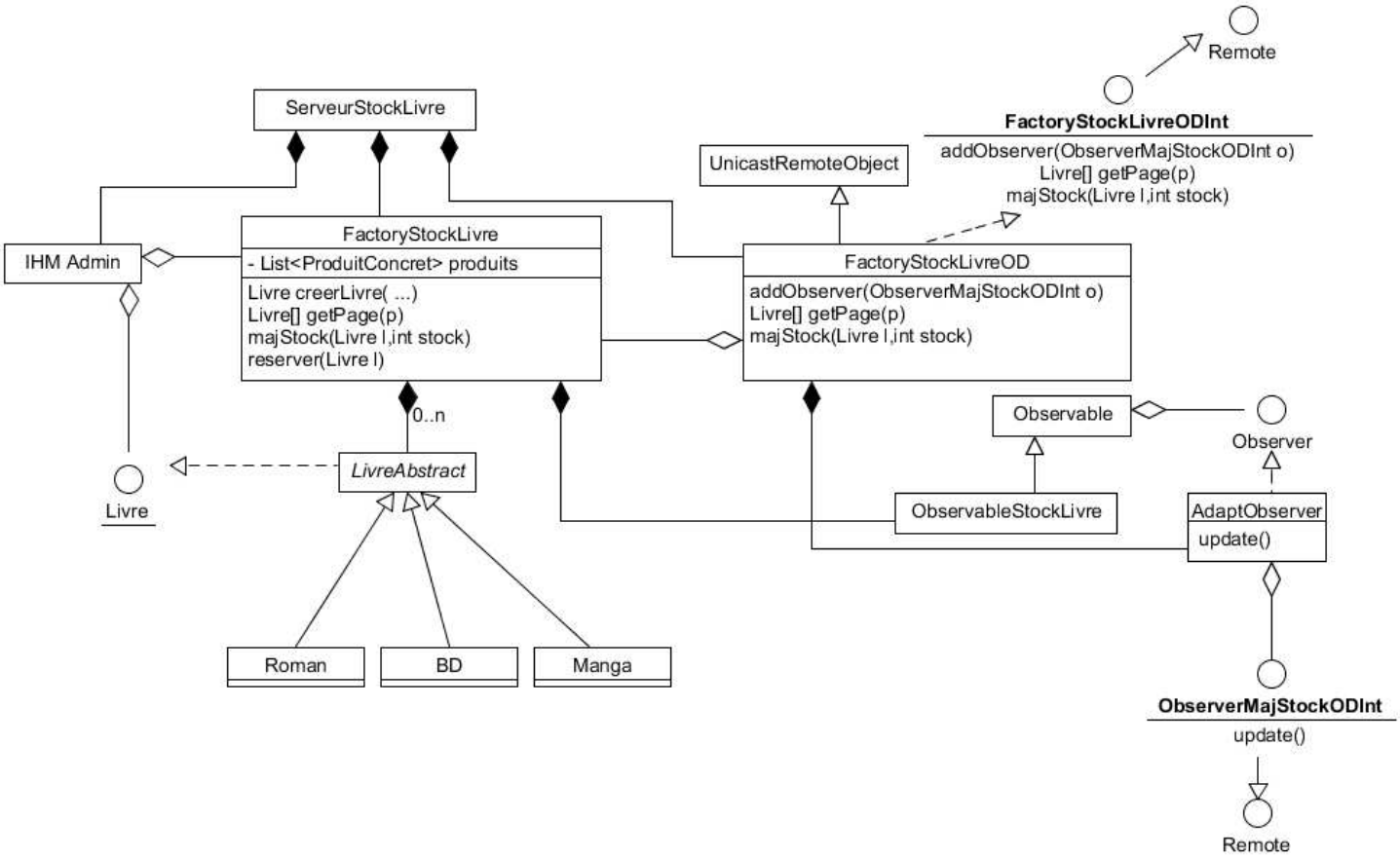
1/ Faite le schéma d'architecture logicielle de votre solution (composants, acteurs, fonctions).
Précisez le rôle de chacun des composants.
5 points



Le **Serveur de stock de livres** gère les livres du magasin. Il permet de mettre à jour le stock des livres (création, suppression, maj stock). Il notifie les maj du stock aux IHM des vendeurs. Il permet de consulter les livres.
L'IHM admin permet de mettre à jour le stock des livres.
L'IHM Vendeur permet de consulter les livres par pages du catalogue. Si dans une page affichée, un stock de livre est mis à jour sur le serveur, la page est notifiée de la mise à jour pour afficher l'état réel du stock.
La **Caisse** met à jour le stock des livres réservés lors de la lecture du code barre du livre.
Le **Compte Client** gère les comptes de chaque client : les infos personnelles du client et les livres qu'il a réservés. Les livres sont réservés (maj du stock temporaire) sur le serveur de stock de livres.
L'IHM Web permet de consulter les livres du magasin, l'inscription d'un client et lui permet de réserver des livres.

2/ Faire le diagramme de classe UML des composants :
[COMPOSANT 1] **15 points**
Ce composant est le serveur de stock. Il est composé de 4 éléments :
- l'IHM administration
- Un Factory de livres qui est utilisé en local par l'IHM Admin : **DP Factory avec classe abstraite**, et autant de classe concrète que de type de livre (Roman, BD, ...)
- Un Factory Objet Distribué qui utilise le Factory de livres local : **DP OD par agrégation**. Cet OD est utilisé par l'IHM Vendeur qui est distant pour consulter les livres (getPage)
- Un Observable distant pour notifier l'IHM Vendeur : **DP Observer/observable** dont l'observer est un adaptateur qui réalise la notification distante.

Remarque : On peut supposer que le composant "Comptes client" fera partie du composant1 d'où la méthode reserver(Livre l) pour réserver en stock un livre. Par contre le composant "Caisse" sera distant d'où la méthode majStock dans l'interface distante du factory pour mettre à jour le stock lors de la vente du livre.



et [COMPOSANT 2] **5 points**

Ce composant est l'IHM Vendeur. Il est composé de 3 éléments :
- l'IHMVendeur qui affiche des pages de livres
- Un **Observer** distant suivant le **DP OD par agrégation** pour mettre à jour la page affichée en recevant les notifications du serveur de stock à travers l'OD
- Une **interface distante** sur le Factory de livre distant pour s'abonner et obtenir les pages à afficher.