

IPST-CNAM
Intranet et Designs patterns
NSY 102
Vendredi 5 Mai 2022

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

1ère Session NSY 102

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: |
|------|---------|

| | | |
|---|-----|------|
| Dans la démarche d'architecture d'un Système d'Information, l' Architecture Technique est l'implémentation de la Configuration Architecturale dans une ou plusieurs technologies données. | | Q 1. |
| 1 | OUI | |
| 2 | NON | |

| | | |
|--|-----|------|
| La Configuration Architecturale est la décomposition d'un Système d'Information en composants logiciels et sous-composants. | | Q 2. |
| 1 | OUI | |
| 2 | NON | |

| | | |
|--|-----|------|
| Dans la description de l'architecture technique, un connecteur est un lien de dépendance entre deux composants qui peut être réalisé par le principe du design pattern de l'injection de dépendance. | | Q 3. |
| 1 | OUI | |
| 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 distant. Pour cela il suffit de : | | Q 4. |
| 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 | créer un proxy de A . Ce proxy hérite de UnicastRemoteObject et implémente l'interface de A qui hérite de Remote | |

| | | |
|--|-----|------|
| En RMI de Java, la classe d'appartenance d'un objet distant, hérite de RemoteObject et implémente l'interface Remote | | Q 5. |
| 1 | OUI | |
| 2 | NON | |

Ceci est le diagramme de classe d'un système composé d'un client et de son applicatif que l'on veut rendre distant.

| | | |
|---|--|--|
| 1 | Client est un proxy de ApplicatifImpl | |
| 2 | ApplicatifOD est un Adaptateur de ApplicatifImpl à ApplicatifODInt | |
| 3 | ClientRmi est un proxy de ApplicatifImpl | |

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

| | | |
|---|--|--|
| 1 | On peut créer un nouvel OD dans la JVM1 qui s'exécute sur le port 9102 | |
| 2 | Sur la machine A, on peut créer une nouvelle JVM3 dans laquelle, on crée un nouvel OD qui s'exécute sur le port 9103 | |
| 3 | Dans la JVM2, on peut crée un nouvel objet distribué RMI sur le port 9102 | |

En Java, la méthode **public static Remote toStub(Remote obj)** permet :

| | | |
|---|--|--|
| 1 | d'enregistrer l'objet distant 'obj' dans l'annuaire RMI | |
| 2 | de retourner un proxy de communication qui implémente l'interface de l'objet distant obj | |

Soit un objet, instance de la classe A.
Pour transformer cet objet en un objet distant, il suffit que :

| | | |
|---|--|--|
| 1 | A hérite de la classe UnicastRemoteObject et implémente une interface publique (I) qui hérite de Remote | |
| 2 | A soit un proxy de la classe UnicastRemoteObject | |
| 3 | A soit une agrégation d'une classe B qui hérite de la classe UnicastRemoteObject et implémente une interface publique (I) qui hérite de Remote | |

| | | |
|---|-----|-------|
| Un Design Pattern (DP) ou Patron de Conception est une norme de description des interfaces entre les composants d'une architecture logicielle orientée objet. | | Q 10. |
| 1 | OUI | |
| 2 | NON | |

| | | |
|---|-----|-------|
| Dans un système réparti, le DP Singleton est utilisé pour créer un objet distribué unique sur le réseau | | Q 11. |
| 1 | OUI | |
| 2 | NON | |

| | | |
|---|-----|-------|
| Il existe deux façons de créer un singleton de classe Singleton : - soit d'instancier le singleton dans la définition de la classe - soit d'instancier le singleton dans l'appel de la méthode <i>Singleton getInstance()</i> | | Q 12. |
| 1 | OUI | |
| 2 | NON | |

| | | |
|--|-----|-------|
| Il est toujours possible de créer des setteurs sur un singleton permettant de modifier des attributs du singleton. | | Q 13. |
| 1 | OUI | |
| 2 | NON | |

| | | |
|--|-----|-------|
| Le DP Factory a pour fonction la création d'objet dont les classes héritent d'une même classe abstraite et/ou implémentent la même interface | | Q 14. |
| 1 | OUI | |
| 2 | NON | |

| | | |
|---|-----|-------|
| 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 | |

| | | |
|---|--|-------|
| Ce DP est celui du Factory. | | Q 16. |
| <pre> classDiagram class A class B class C class D A "1" *-- "*" B C .. > D C --> "0..n" B </pre> | | |
| La signification des lettres A, B, C et D est : | | |
| 1 | A=Factory; B = Concrete Product; C=Product (Interface); D=Client | |
| 2 | A=Client; B=Factory; C=Concrete Product; D=Product (interface); | |
| 3 | A = Client; B=Product (interface); C=Concrete Product; D = Factory | |

| | | |
|---|-----|-------|
| Le DP Factory mémorise toujours les produits créés dans une collection. | | Q 17. |
| 1 | OUI | |
| 2 | NON | |

| | | |
|--|-----|-------|
| Le DP "Délégation" est utiliser dans le DP "injection de dépendance" | | Q 18. |
| 1 | OUI | |
| 2 | NON | |

| | | |
|--|-----|-------|
| Si la classe A est un décorateur de la classe B alors les classes A et B héritent toutes deux d'une même classe abstraite. | | Q 19. |
| 1 | OUI | |
| 2 | NON | |

| | | |
|---|-----|-------|
| Dans le DP Observateur, la communication entre l'Observer (consommateur d'évènement) et l'Observable (producteur d'évènement) est nécessairement asynchrone car la communication se fait toujours par l'envoi d'un objet sans valeur de retour. | | Q 20. |
| 1 | OUI | |
| 2 | NON | |

| | | |
|---|---|-------|
| Dans le DP Observateur, la communication entre l'objet observé (producteur d'évènement) et l'observateur (consommateur d'évènement) est : | | Q 21. |
| 1 | synchrone ou asynchrone (choix de conception) | |
| 2 | toujours asynchrone | |
| 3 | toujours synchrone | |

| | | |
|---|--|-------|
| Le DP Observateur , peut être utilisé, dans une architecture MOM, pour réaliser | | Q 22. |
| 1 | le connecteur entre le Provider et les Consommateurs | |
| 2 | le connecteur entre le Producteur et le Provider | |

| | | |
|--|-----|-------|
| Dans le DP Adaptateur, l'adaptateur et l'adapté implémente la même interface | | Q 23. |
| 1 | OUI | |
| 2 | NON | |

| | | |
|---|--|-------|
| Soit le diagramme de classe suivant : | | Q 24. |
| <pre> classDiagram class InterfaceXXX { +void proc(params) } class ClasseXXX { +ClasseXXX() +void proc(params) } class XXX { +XXX() +void proc(params) } InterfaceXXX < .. ClasseXXX InterfaceXXX < .. XXX ClasseXXX *-- XXX </pre> | | |
| 1 | Ce diagramme de classe représente le DP Adaptateur | |
| 2 | Ce diagramme de classe représente le DP Proxy | |

| | | |
|---|-----|-------|
| Dans la communication synchrone, en mode push, via un "canal d'évènement", entre un producteur et des consommateurs, le producteur utilise un proxy de consommateur (et non les consommateurs directement), afin de leurs pousser un évènement. | | Q 25. |
| 1 | OUI | |
| 2 | NON | |

| | | |
|--|-----|-------|
| Un canal d'évènement est constitué de deux DP : un DP Factory permettant de créer des évènements et un DP Iterator permettant de parcourir ces évènements. | | Q 26. |
| 1 | OUI | |
| 2 | NON | |

Soit le schéma suivant :

Les classes et interfaces en rouge encadrées représentent :

| | | |
|---|--|--|
| 1 | un proxy client de communication de ObserverHorloge vers ObservableHorloge | |
| 2 | un pont de communication permettant à un Observable (ObservableHorloge) de notifier les évènements à un Observer (ObserverHorloge) se trouvant dans une autre JVM. | |

Le diagramme suivant :

représente

| | | |
|---|---|--|
| 1 | une injection de dépendance par l'utilisation d'un setteur | |
| 2 | une injection de dépendance par l'utilisation d'un constructeur | |
| 3 | une injection de dépendance par l'utilisation d'un proxy | |

| | | |
|---|---|-------|
| Soit le Design Pattern Observateur décrit (volontairement simplifié) de la manière suivante : | | Q 29. |
| <pre> classDiagram class Observable { +observers: Observer } class ObservableXXX class Observer { } class ObserverXXX Observable < -- ObservableXXX Observer < .. ObserverXXX Observable "0..N" o-- "*" Observer </pre> | | |
| 1 | La classe ObservableXXX notifie les évènements à une instance de Observable | |
| 2 | La classe ObserverXXX implémente la méthode update de l'interface Observer qui est appelée par Observable | |
| 3 | La classe Observable pousse (modèle du "push") les évènements à ObserverXXX | |

| | | |
|--|--|-------|
| Laquelle des descriptions suivantes est un principe de communication synchrone : | | Q 30. |
| 1 | le producteur dépose à son rythme ses évènements dans une file. Le ou les consommateurs peuvent alors récupérer ces évènements | |
| 2 | le producteur pousse ("push") chaque évènement vers chacun des consommateurs via une méthode distante qui retourne un état de consommation | |

| | | |
|---|-----|-------|
| Le principe général d'un MOM (Model Orienté Message) est que tous les composants connectés, à un même canal d'évènement du bus du MOM, en mode Topic, reçoivent tous les évènements publiés dans le fournisseur de service MOM (Provider) | | Q 31. |
| 1 | OUI | |
| 2 | NON | |

| | | |
|---|-----|-------|
| <pre> classDiagram class AppInt { +Object invoke(Object proxy, Method m, Object[] args) } class InvocationHandler { } class App { } class MyServiceHandler { } class Utilisateur { +traitement(Object o) } class DynamicProxy { } AppInt < .. InvocationHandler AppInt < .. App InvocationHandler < .. MyServiceHandler Utilisateur *-- DynamicProxy DynamicProxy o-- InvocationHandler </pre> | | Q 32. |
| Ce schéma est celui du DP Dynamic proxy. | | |
| Le rôle de la classe MyServiceHandler est d'implémenter toutes les méthodes de l'interface AppInt. | | |
| 1 | OUI | |
| 2 | NON | |

Q 33.

```

classDiagram
    class Utilisateur {
        traitement(Object o)
    }
    class DynamicProxy
    class AppInt {
    }
    class InvocationHandler {
        Object invoke(Object proxy, Method m, Object[] args)
    }
    class MyServiceHandler
    class App

    Utilisateur o-- DynamicProxy
    DynamicProxy ..> InvocationHandler
    AppInt <|.. MyServiceHandler
    AppInt <|.. App
    InvocationHandler <|.. MyServiceHandler
    App o-- MyServiceHandler
    
```

L'API RMI de Java utilise le DP DynamicProxy pour :

| | | |
|---|---|--|
| 1 | créer par l'Injection de Dépendance, la dépendance entre la classe MyServiceHandler et App. | |
| 2 | créer le proxy client dans la méthode lookup, permettant la communication avec l'objet. | |

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

Q 34.

```

sequenceDiagram
    participant P as :Producteur
    participant T as :Topic
    participant C1 as :Consommateur
    participant C2 as :Consommateur

    P->>T: publish(m1)
    T->>C1: onMessage(m1)
    T->>C2: onMessage(m1)
    P->>T: publish(m2)
    T->>C1: onMessage(m2)
    T->>C2: onMessage(m2)
    P->>T: publish(m3)
    T->>C1: onMessage(m3)
    T->>C2: onMessage(m3)
    
```

Ce diagramme de transition correspond au mode :

| | | |
|---|-------|--|
| 1 | Queue | |
| 2 | Topic | |

En JMS (Java Messaging System), les producteurs et les consommateurs sont tous des clients d'un composant logiciel appelé JMS Provider

Q 35.

| | | |
|---|-----|--|
| 1 | OUI | |
| 2 | NON | |

Fin du QCM

Suite (Tournez la page)

2. Questions libres (15 points)

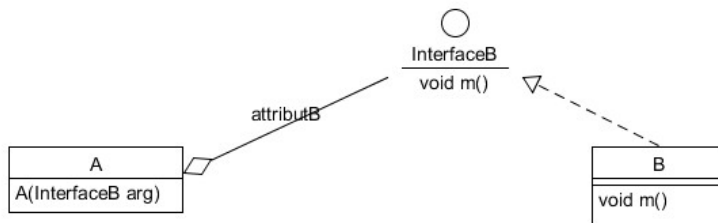
Chaque question est notée sur 5 points.

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

Soit 2 classes A et B dont A utilise B à travers une interface :



Écrire le Diagramme de classe qui permet de rendre distant la dépendance entre A et B en utilisant la technologie RMI de Java.

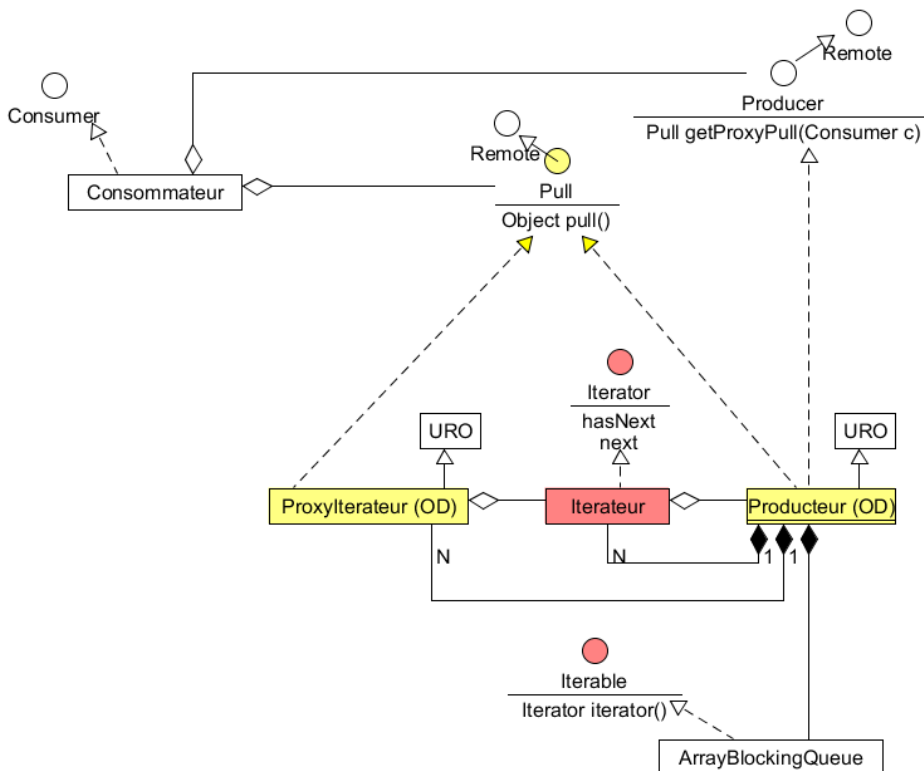
QUESTION NUMERO 2

Expliquez le principe du Design Pattern : Proxy.

Citez 3 cas d'utilisation de ce DP en vous justifiant.

QUESTION NUMERO 3

Soit le Design Patern Pull Asynchrone décrit comme ceci (vu en cours) :



Expliquez le fonctionnement de ce DP.

Fin de la 1^{ère} partie sans document

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

3. PROBLEME [50 points]

Nous envisageons de réaliser un Système d'Information (SI) qui réalise des chaînes de traitements d'images satellites.

Ce SI est composé de 4 COMPOSANTS :

- COMPOSANT 1, appelle Récepteur, qui reçoit les images brutes provenant de différentes Stations de Réceptions d'Image, vérifie leurs qualités brutes, et les envoie au Serveur (COMPOSANT 2) dans le format TIFF (Tagged Image File Format).
- COMPOSANT 2, appelé Serveur, qui gère les images brutes reçues et réalise les traitements d'images demandées par un opérateur.
- COMPOSANT 3, appelé Poste Opérateur, en nombre indéterminé, qui permet, notamment, à un opérateur, de demander au Serveur de réaliser une chaîne de traitements d'image sur une image brute.
- COMPOSANT 4, appelé Archivage qui permet d'archiver dans une base de données les images traités, et de rechercher ces images pour pouvoir les retraitées.

Quand le Serveur reçoit une image brute, il crée un **Produit d'Image** qui contient :

- le numéro d'identification (généralisé par le serveur)
- l'image brute (celle envoyée par le Récepteur)
- la liste ordonnées des différentes images résultats de l'application de chacun des traitements d'image successifs (vide par défaut).
- l'état courant du Produit d'Image : « non traité » (par défaut), « en cours », « traité ».

Par défaut, le Serveur contient tout un ensemble de traitements d'image de base qu'il est possible d'appliquer sur une image (corrections géométriques, corrections de luminosité, de contraste, effets de seuils, filtres radiométriques, fusion d'images, classifications d'image, ...).

Le serveur permet de créer des chaînes de traitements constitués de traitements de base, ou d'autres chaînes de traitement.

Le Poste Opérateur permet de:

- créer une chaîne de traitement, sur le serveur, en précisant la valeur des paramètres de chacun des traitements de base.
- afficher la liste de tous les Produits d'Image connus du Serveur
- sélectionner un Produit d'Image « non traité » ou « traité » pour demander au serveur de réaliser l'exécution d'un traitement de base ou d'une chaîne de traitement.
- sélectionner un Produit d'Image :
 - si « en cours », pour surveiller le déroulement des traitements sur ce produit dans une zone graphique : à chaque fois que le Serveur a fini d'exécuter un traitement de base, il notifie tous les Postes Opérateur qui ont fait la demande de surveillance du Produit d'Image.
 - Si « traité » pour afficher les résultats du Produit d'Image dans une zone graphique
- fermer une zone graphique qui annule la surveillance ou l'affichage du produit d'image associé.
- sélectionner un Produit d'Image « traité » pour en demander l'archivage, ce qui entraîne sa suppression sur le serveur.
- rechercher et choisir un Produit d'Image archivé qui entraîne sa création sur le Serveur à l'état « non traité ». Il est alors possible d'exécuter, à nouveau, une chaîne de traitement sur ce produit.

Sur un Poste Opérateur, chaque surveillance d'un Produit d'Image est réalisée dans une zone graphique qui affiche le résultat de la dernière exécution d'un traitement de base. Quand le Produit d'Image est dans l'état « traité », deux boutons, flèche gauche et flèche droite, permettent d'afficher les différentes images résultats.

Pour démarrer le SI, les composants suivants sont exécutés dans cet ordre : Serveur, Récepteur, Archivage. Ensuite les Postes Opérateur démarrent au grès de leurs utilisations.

1/ [15 points]

Faites le diagramme de communication de ce Système d'Information. Il y a donc 4 composants à décrire qui communiquent entre eux de manière distant.

Commentez votre schéma (rôles des composants et sous-composants, comportement dynamique général, échanges des informations, localisation des données).

Nous rappelons que ce schéma doit permettre de connaître vos choix d'organisation des composants, le sens des communications, et les designs patterns envisagés.

2/ [35 points]

Faites le(s) diagramme(s) de classe UML des [COMPOSANT 2] et [COMPOSANT 3] en mettant en évidence les Designs Patterns utilisés.

Commentez chacun de(s) diagramme(s).

Précisions :

Un composant applicatif [COMPOSANT X] correspond à une JVM ou process. Cela signifie que les COMPOSANTS X communiquent sur le réseau à travers des interfaces distantes.

Ainsi, pour une description précise de vos diagrammes de classe, on fait le choix que toutes les communications distantes entre les composants sont réalisées en RMI (utilisation de la classe URO = UnicastRemoteObject et de l'interface Remote).

Fin du sujet