Départ. Informatique
LP DEVOPS - Module Modélisation
Conception et Méthodes Agiles

V. Deslandres

TD1 – Modélisation, Conception

EXERCICE 1 - Couches logicielles

L'architecture *n-tiers* ou *multi-niveaux*, dérivée du modèle client-serveur, est très adaptée au fonctionnement web, exploitant les capacités de partage et de diffusion de l'information rendue ainsi disponible.

> Le mot *tier* n'a pas la même signification en français et en anglais. En effet, *tier* (et son pluriel *tiers*) se traduit par niveau ou couche, alors que le mot français *tier* désigne une partie d'un tout divisé en trois.

Pour les applications proposées, on soumet une liste de fonctionnalités : pour chacune, préciser si elle relève plutôt de **l'IHM**, du **métier**, ou de la **persistance des données**.

A- Application Assurances

1. Saisir le login/ pwd en tant que Client
2. Vérifier l'authentification d'un utilisateur
3. Choix du type de sinistre par le client
4. Vérification de la couverture d'une personne dans un sinistre
5. Réplication de la BD
6. Saisie des informations administratives Clients
7. Calcul du montant des cotisations d'une personne
8. Déclaration du sinistre : enchaînement des écrans de saisie
9. Enregistrement d'un nouveau sinistre

B- Application « Formation en ligne »

Aujourd'hui de nombreuses formations sont disponibles en ligne (*e-learning, MOOC*). Les fonctionnalités proposées pour ce type d'outil sont les suivantes (tous utilisateurs et à différents niveaux d'abstraction) :

1. Choix de la formation parmi une liste
2. Inscription d'un utilisateur (saisie des informations)
3. Affectation d'un tuteur pour un apprenant
4. Consultation des pré-requis d'un module
5. Saisir les caractéristiques du module (nombre de crédits, vol. horaire, dates, modes d'évaluation)
6. Valider les saisies des caractéristiques
7. Obtenir les modules accessibles par une formation
8. Accéder au contenu d'un module (accès réservé aux inscrits)
9. Ouvrir une session de *chat* entre un tuteur et ses apprenants
10. Annoter les travaux d'un apprenant

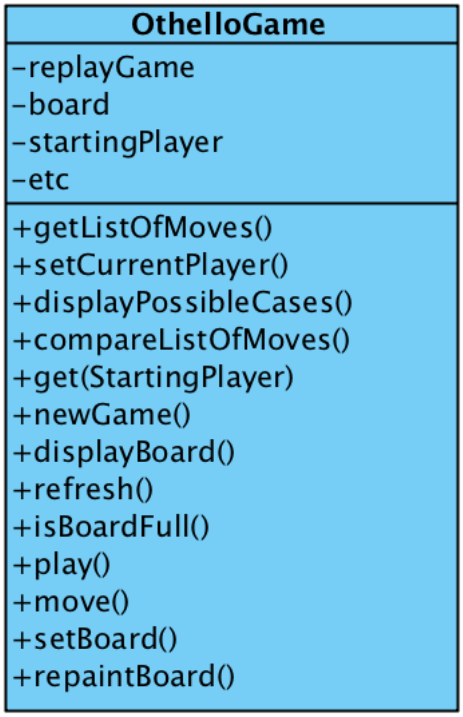
C- Application « Jeu de Bataille navale »

Vous connaissez ce jeu : sur un plateau, le joueur peut placer ses bateaux et déclencher ses tirs. Auparavant il a pu se connecter et attendre un autre joueur. Il reçoit l'information quand c'est son tour de jouer et quand la partie est terminée (tous les bateaux d'un joueur ont été coulés). Il voit les tirs de son adversaire, où il a déjà tiré et le résultat de ses tirs.

Citez quelques méthodes relatives à chaque couche :
- Présentation
- Applicative
- Persistance

EXERCICE 2 - Couplage / cohésion de classes

2.1. `Othello`



Soit la classe du Jeu Othello suivante (schéma) :
Cette classe est-elle **cohésive** ?

Si non, proposer une amélioration.

2.2. `Couplage fort : gestion d'anniversaires`

Imaginons devoir développer une application qui gère les anniversaires : elle stocke les dates d'anniversaires des amis et les rappelle quand nécessaire.

Nous avons donc une classe qui s'occupe de charger et d'enregistrer les dates d'anniversaires dans un calendrier partagé.

Nous développons une classe **`Calendar`** avec donc une méthode `add` qui pourrait s'écrire comme ceci :

```
public class Calendar {
	public void add(Personne personne) {
	/* TODO */
	}
}
```

où Personne est une classe réutilisée de l'équipe de développement, avec les propriétés usuelles comme le nom, le prénom, l'âge, la date de naissance ainsi que des méthodes concernant la personne : envoyer un message, envoyer un SMS.

On constate donc que **`Calendar`** est **fortement couplée** à Personne. Est-ce bien nécessaire ?

a) Pour diminuer le couplage, on décide de construire une interface **IPersonne** avec juste les méthodes de **Personne** (`extract interface` sous l'IDE) :

```
interface IPersonne {
    int getAge();
    DateTime getDateNaissance();
    void setDateNaissance(DateTime uneDate);
    void EnvoyerEmail(String msg);
    void EnvoyerSMS(String msg);
    String getNom();
    void setNom(String unNom);
    // idem pour le prénom
}
```

Du coup, la signature de `add` devient :

```
public void add (IPersonne personne) {
```

b) Vous apprenez du client que **Calendar** ne va jamais utiliser les fonctions d'envoi de messages et de mails à une Personne. **Calendar** a juste besoin de connaître l'âge et le nom de la personne, càd. d'affecter une date de naissance à une personne amie, et de la consulter. **IPersonne** est-elle donc adaptée à notre situation ?

c) Votre chef mentionne d'autre part que **Calendar** devra aussi gérer les anniversaires des membres de Facebook, qui propose une interface semblable à **IPersonne** :

```
interface IAmiFacebook {
    int getAge();
    DateTime getDateNaissance();
    void setDateNaissance(DateTime uneDate);

    // un pseudo, pas le nom complet
    String getPseudo() ;
    void setPseudo(String p) ;

    void EnvoyerMessage(string msg);

}
```

→ Trouver les éléments communs entre **Personne** et **IAmiFacebook** (définir leur contrat), et proposer une amélioration de la conception de **Calendar** visant à diminuer le couplage entre les 3 classes.

EXERCICE 3 – Relation entre le Modèle Relationnel et UML

Vous avez en charge une étude pour la réalisation d'une base de données de gestion d'une association. Vous ne disposez que du modèle relationnel (MR) ci-dessous :

```
Compte (#num:entier, categorie:{A|R}, email:chaine, =solde():réel,
        =alerter() :vide)
Recette(#num=>Compte, #date:date, montant:reel, annee:entier)
Dépense(#num=>Compte, #date:date, montant:reel, seuilAlerte:entier,
        annee:entier, =alerter():vide)
```

Et des explications suivantes :

« Les comptes A (actifs) sont les comptes courants de l'association. En cas de solde négatif, une alerte automatique est envoyée au email associé (titulaire du compte). Les comptes R (réserve) sont des comptes sur lesquelles les opérations sont rares et n'interviennent qu'en cas de problème de trésorerie ou d'investissement important. En cas de dépense supérieure à une valeur seuil, une alerte est aussi envoyée au titulaire du compte »

Les méthodes mentionnées (*solde(), alerter()…*) sont ici associées aux relations. Bien entendu ces méthodes ne seront pas implémentées en SQL LDD et devront être programmées à part, dans des procédures stockées par exemple.

3.1 - Réalisez le schéma UML permettant d'obtenir ce Modèle Relationnel.
Nous proposons d'analyser la qualité de ce modèle :

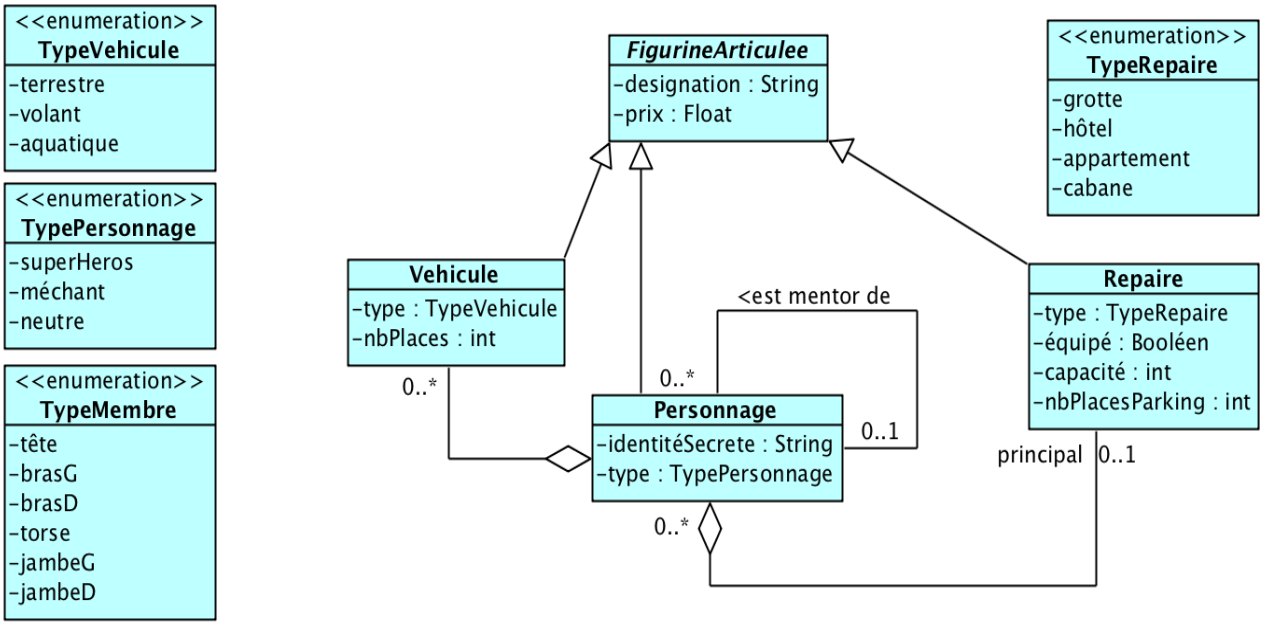
a - Serait-il judicieux de faire intervenir une **relation d'héritage** à la place de l'attribut catégorie pour discriminer les comptes A et R ?

b - Donner les **responsabilités** de la classe *Compte* et de la classe *Dépense*. Qu'en pensez-vous ? Il semble y avoir 2 erreurs.

On imagine d'autre part que dans l'association, il y a des stagiaires présents de 6 à 9 mois, à qui on ouvre des comptes, et des permanents qui sont là depuis plus de 5 ans. Que pensez-vous donc de la responsabilité **relative aux alertes** actuelle ? Proposez des améliorations.

3.2 - Dans une BD de plus d'un million d'enregistrements d'une BD de figurines MATTEL (cf image ci-dessous), la requête la plus utilisée par l'application Web sert à lister tous les super-héros avec toutes leurs caractéristiques, avec leurs véhicules et leurs repaires (et également toutes leurs caractéristiques), puis à la trier par ordre de prix.

a- Expliquez les raisons du temps de réponse de cette requête (parfois quelques secondes).



b- Quelles solutions d'optimisation en BD pourrait-on proposer ?

3.3 – En analyse on utilise souvent des énumérations pour définir des ensembles possibles de valeurs pour les attributs (ex. enum TypeCompte = {actif, réserve}. Comment pensez-vous enregistrer une **énumération** dans une BD **?**

EXERCICE 4 – Inversion de Contrôle

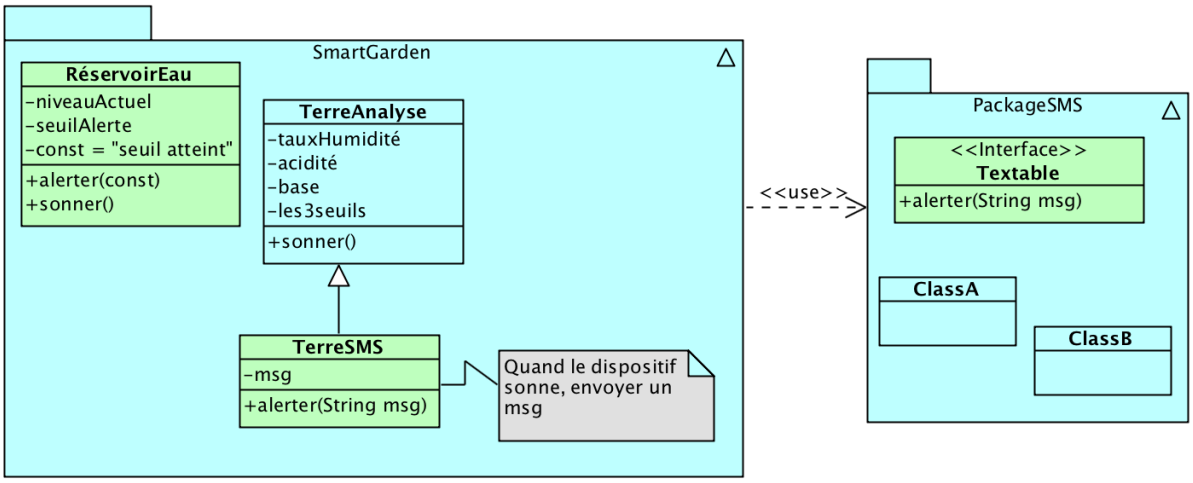
Dans un jardin connecté, différents dispositifs sont installés, certains fabriqués par nous, d'autres sont des composants propriétaire. Un suivi à distance est possible par notifications de SMS. Ainsi :

- Un **réservoir** fait maison (en vert) sonne quand il atteint un seuil d'alerte : on a ajouté une puce qui envoie alors un SMS au jardinier : "Seuil atteint" ;

- Un dispositif propriétaire d'analyse de la terre se met à sonner quand il y a une anomalie sur les mesures (humidité, acidité, base).

- On crée alors, en extension de ce capteur (en vert), un composant qui va **transmettre l'information au jardinier via un SMS.**

On a donc le S.I. suivant :



Mais on aimerait que le déclanchement d'une alerte puisse aussi se faire par une personne : on installe donc un nouveau composant, un **carillon**, qu'un employé du jardin actionne quand il constate un problème.
Et comme pour l'analyse de la terre, on programme le carillon pour qu'il puisse transmettre un SMS aux jardiniers.

4.1. Modifier le **DCL** en conséquence.

4.2. Quel **inconvénient** possède cette architecture ? Imaginez qu'on ajoute un autre composant, puis un autre, etc.

4.3. Notre PKG SmartGarden dépend donc d'une interface **alerter()** par SMS (développée par nos soins). On aimerait que ce soit l'inverse, que le package technique d'alerte dépende du package Métier. Comment procéder ?

Indice : écoutez bien ! La tonalité de sonnerie de chacun des dispositifs est différente.

EXERCICE 5 – Principe de Liskov

Voici un code représentant un contrat pour l'acquisition de données :

```
public interface IDevice{
        void open();
        void read();
        void close();
}
```

Chaque matériel d'acquisition peut être différent selon son type d'interface. On a par exemple des interfaces USB, des interfaces réseaux (TCP ou UDP), des interfaces PCI express ou n'importe quel autre type d'interface d'ordinateur.

5.1- Les composants Client de `IDevice` n'ont pas besoin de savoir quel type de matériel il sollicite. Quel avantage cela procure-t-il ?

5.2 -Supposons qu'il n'y ait dans un premier temps que 2 classes concrètes qui implementent `IDevice` interface :

```
public class PCIDevice implements IDevice {
   public void open(){
   // Device specific opening logic
   }
   public void read(){
   // Reading logic specific to this device
   }
   public void close(){
   // Device specific closing logic.
   }
}

public class NetworkDevice implements IDevice {
   public void open(){
   // Device specific opening logic
   }
   public void read(){
   // Reading logic specific to this device
   }
   public void close(){
   // Device specific closing logic.
   }
}
```

Les 3 méthodes (`open`, `read` and `close`) suffisent pour gérer les données sur ces matériels. Imaginons qu'on introduise un nouveau dispositif d'acquisition basé sur une interface USB.

Le problème avec le matériel USB est que quand ou ouvre la connexion, les données de la précédente connexion sont toujours présentes dans le buffer. Ainsi au premier appel à `read()` sur ce matériel, les données de la précédentes sessions sont renvoyées, ce qui fait que l'acquisition des données est corrompue pour la session.

Heureusement, les pilotes de dispositifs USB fournissent une fonction *refresh* qui efface le buffer du matériel USB. Une solution naïve est d'implémenter cette caractéristique au niveau de la classe Client.

On propose donc cette classe **`USBDevice :`**

```
public class USBDevice implements IDevice{
 public void open(){
    // Device specific opening logic
 }
 public void read(){
    // Reading logic specific to this device<br>
 }
public void close(){
   // Device specific closing logic.
 }
 public void refresh(){
   // specific only to USB interface Device
 }
```

```
}
```

Et c'est le composant Client qui vérifie le type de matériel, et si c'est un matériel USB, agit en conséquence : code de la méthode ***acquire()*** d'une classe client :

```
public void acquire(IDevice aDevice){
        aDevice.Open();
        // Identify if the object passed here is USBDevice class Object.
        if(aDevice.GetType() == typeof(USBDevice)){
        USBDevice aUsbDevice = (USBDevice) aDevice;
        aUsbDevice.Refresh();
        }

        // remaining code….
}
```

Que penser de cette solution ?
Quelle amélioration proposer ?