TP JAVA n°6: Tutorial pour la mise en place d'une application interactive en mode console

(notions d'architecture logicielle /pattern MVC /DTO /DAO)

Jusqu'à maintenant nous programmions directement les essais («en dur») dans notre programme et aucune interaction avec l'utilisateur n'était proposée.
Nous allons maintenant créer une ***application interactive*** qui proposera dans un premier temps à un utilisateur de ***créer un Patient*** en mode console.
Comme vous l'avez vu en UML, le ***flot de base de ce use case*** est le suivant :

(création d'un patient sans ascendant)

Flot de base

1. le système affiche la liste des informations à saisir,
2. l'utilisateur saisit les informations de la fiche patient et valide
3. le système vérifie la saisie et enregistre le nouveau patient, et affiche un message de prise en compte,
4. l'utilisateur choisit de quitter,
5. le système ferme le use case.

Remarque : Le plan type détaillé complet est donné en annexe 1

La mise en place d'une interactivité a une incidence sur l'architecture logicielle d'une application.

Il est bien sûr évident qu'il faut éviter d'écrire toutes les classes au même niveau et qu'il faut organiser son application en « package » (couche) afin de faciliter le développement (répartition des tâches) et la maintenance de l'application…

Choix d'une architecture logicielle :
Avant de se lancer dans la programmation d'une application, il est donc important de réfléchir à son architecture.
Le concept d'architecture *multi-tiers* (ou *n-tiers*) propose de découper une application en plusieurs ***couches logiques*** spécialisées chacune dans une fonction précise :
→ ***couche présentation :*** présentation des informations à l'utilisateur, interface de saisie (IHM)
→ ***couche application (navigation ou contrôle) :*** gestion du parcours de l'utilisateur entre les différentes parties de l'application.
→ ***couche métier :*** implémentation des traitements directement liés au métier
→ ***couche de persistance métier :*** implémentation des mécanismes permettant de sauvegarder les objets manipulés dans l'application sur un support de persistance et de gérer les accès à ces données.

La dénomination de ces différentes couches ne vous est pas inconnue puisque avant de commencer à implémenter le projet `cabinetMedical`, nous avions déjà organisé notre application en package afin de respecter « le découpage en couches issues de la conception » (cf TP1)

Les couches logiques d'une architecture *n-tiers* (architecture logicielle) peuvent être hébergées sur une ou plusieurs couches physiques de l'infrastructure (architecture matérielle).

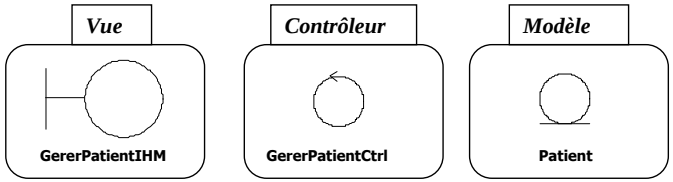
En ce qui concerne l'architecture matérielle pour ce TP, nous travaillerons en architecture centralisée : les parties application, présentation et métier seront situées sur une même machine.

1. Architecture logicielle du projet `cabinetMedical`

Il est important lors de la mise en place d'une architecture de ne pas « réinventer la roue », mais de s'appuyer sur des patterns (modèles de programmation) robustes et simples qui ont fait leur preuve.
Pour écrire notre application, nous utiliserons le **pattern MVC** (**M**odèle **V**ue **C**ontrôleur) qui vous a été présenté en cours d'UML.

1.1 Présentation de l'architecture MVC

Pour expliquer cette architecture, nous allons reprendre les documents produits au cours de la phase d'analyse, notamment le diagramme de séquence qui, comme par hasard, s'appuie sur le pattern MVC



Couche présentation ***Couche application*** ***Couche métier***

Dans le diagramme, nous identifions les 3 composants d'un modèle MVC :
- le **Modèle** (classe `Patient`) qui correspond à la classe ***métier*** qui contient la logique de l'application.
- la **Vue** (classe `GererPatientIHM`) qui permet de mettre en place une (re)***présentation*** visuelle de l'application à l'écran (pour aujourd'hui ce sera un mode console). Elle affiche des informations sur le modèle.
- le **Controleur** (classe `GererPatientCtrl`) qui est en fait le cœur de l'***application***. Il est d'ailleurs représenté au centre du diagramme de séquence.
  Le contrôleur agit sur demande de l'utilisateur et effectue alors les actions nécessaires sur le modèle.
  Il assure ainsi l'interface avec l'utilisateur en faisant exécuter sa demande par la couche métier….
  Et c'est bien là tout l'intérêt du contrôleur : la couche présentation n'a pas le droit d'attaquer directement la couche métier. Ainsi, pour que la vue ne voit pas le modèle, le contrôleur sert de pont entre les deux.

La fonctionnalité `CreerPatient` est une des 4 fonctionnalités proposées par le use case `GererPatient` (créer, modifier, supprimer, rechercher).
Notez qu'il n'y a qu'un seul contrôleur par use case, le contrôleur qui nous intéresse est bien `GererPatientCtrl`. Même si vous travaillez pour l'instant uniquement sur la création d'un patient, nous utiliserons directement les classes `GererPatientCtrl` et `GererPatientIHM` que vous compléterez plus tard en y rajoutant les fonctionnalités de modification, de suppression et de recherche du use case `GererPatient`.

1.2 L'architecture MVC dans notre application

Comme le montre le diagramme de séquence, l'implémentation en JAVA de la création d'un patient ne sera possible que si les **3 couches** (**`presentation`**, **`application`** et **`metier`**) sont respectées.

Etape 1

Travail à faire : Vous devez donc mettre en place 2 nouvelles classes :
- dans le paquetage `com.iut.cabinet.presentation` une classe **`GererPatientIHM`** qui permettra de mettre en place le représentation visuelle à l'écran de l'application (***vue***). Cette classe est disponible sous la zone libre, importez la.
- dans le paquetage `com.iut.cabinet.application` une classe **`GererPatientCtrl`** qui assurera l'interface entre l'utilisateur et le modèle (contrôleur).

En effet, pour le ***modèle***, vous avez déjà développé :
dans le paquetage `com.iut.cabinet.metier` la classe **`Patient`**

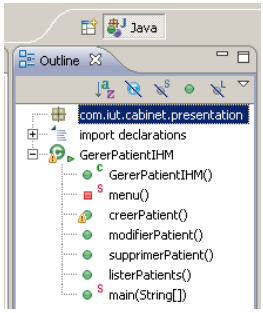
2. Implémentation de l'opération `CreerPatient` (sans ascendant) en mode console

2.1 Classe `GererPatientIHM` de la couche `presentation` :

Récupérer la classe `GererPatientIHM` sur la zone libre et importer la dans le package `com.iut.cabinet.presentation`.

Cette classe contient :
- un *constructeur* `GererPatientIHM()`
- une *méthode privée statique* `menu()` qui permettra d'afficher à l'écran le menu de l'application du type :

```
----------------------------------------
---- Gestion des Patients   ------
----------------------------------------
-- 1. Creer un patient
-- 2. Modifier un patient
-- 3. Supprimer un patient
-- 4. Lister tous les patients
-- 0. Quitter
----------------------------------------
```

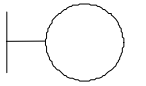


- les méthodes publiques `creerPatient()`, `modifierPatient()`, `supprimerPatient()` et `listerPatients()` qui contiendront le code de chacun des use-case de `GererPatient`.

Théoriquement, notre application devrait pouvoir être exécutée à partir de la classe `EssaiCabMed_console`. Créer cette classe dans le package `com.iut.cabinet.essai`.
Cette classe est le point d'entrée dans notre application, sa première tâche (et la seule) est donc de lancer l'IHM : pour cela elle doit créer un objet de type **`GererPatientIHM();`**

La méthode `main` de la classe `EssaiCabMed_console` n'est donc composée que de l'instruction suivante :

```
new GererPatientIHM();
```



qui permet de créer la vue (**`boundary`**) de notre modèle MVC.
Exécuter la classe `EssaiCabMed_console`.
Pour l'instant, l'application ne sera exécutée qu'en mode console (un simple menu et des saisies au clavier).

Astuce : … afin de faciliter la phase de développement de l'application, vous pouvez écrire le code suivant directement dans la classe `GererPatientIHM` (aller voir en fin de classe… elle s'y trouve déjà…)

```
public static void main(String[] args) {
        new GererPatientIHM();
        }
```

Cela permet, *au cours de la phase de développement* de lancer l'IHM sans avoir à changer de fichier ! ! !

2.2 Mise en place du modèle MVC

↳ Consultez tout d'abord le code du ***constructeur*** de la classe `GererPatientIHM.`
Vous constaterez que le constructeur permet de commencer l'interaction avec l'utilisateur en appelant le `menu`, et en orientant les choix de l'utilisateur vers les autres méthodes de la classe (par exemple, pour l'option 1 du **menu**, la méthode `creerPatient` est appelée)…
Les autres cas, seront bien sûr à compléter plus tard par vos soins afin d'améliorer le projet …

↳ Au lancement de l'application, nous devons donc mettre en place le modèle **MVC** en créant la **Vue**, mais aussi en créant le **Contrôleur**. (L'objet **M**étier viendra plus tard en fonction des options choisies par l'utilisateur)

Rappelez-vous que dans le modèle MVC, le **Contrôleur** est le ***cœur de l'application*** et agit sur le **Modèle** sur demande de l'utilisateur: la **Vue** (`IHM`) dialogue donc uniquement avec le **Contrôleur**, jamais avec le **Modèle**. Pour vous en convaincre consulter le diagramme de séquence de l'annexe 2 :
Une fois la **Vue**(`IHM`) lancée, l'utilisateur peut saisir les données concernant le patient. Lorsqu'il a fini il valide (message 3:`validerSaisie`). La **Vue**(`IHM`) transmet alors ses données via un message au **Contrôleur** (message `4:validerSaisie`), c'est le **Contrôleur** qui se charge de créer un nouveau `Patient` (message `5:Patient`) et son adresse (message `6:Adresse,` message `7:setAdresse`) et d'enregistrer ce Patient dans un support de persistance (message 8 : `store`)

Côté implémentation JAVA, pour que la **Vue** (`IHM`) puisse envoyer des messages au **Contrôleur**, il est nécessaire de déclarer une instance de ce contrôleur (objet de type `GererPatientCtrl`) comme ***attribut de classe*** **`GererPatientIHM`** .

Etape 2

Travail à faire : Dans la classe `GererPatientIHM`, déclarer l'attribut suivant:

```
GererPatientCtrl ctrlUseCase;
```

La mise en place du **Contrôleur** doit se faire en même temps que la mise en place de la **Vue** (`IHM`.)

Etape 2

Travail à faire : Pour que le contrôleur soit créé au démarrage de la vue, l'instanciation de l'objet *`ctrlUseCase`* devra donc être la ***toute première instruction du constructeur*** de la classe `GererPatientIHM.`

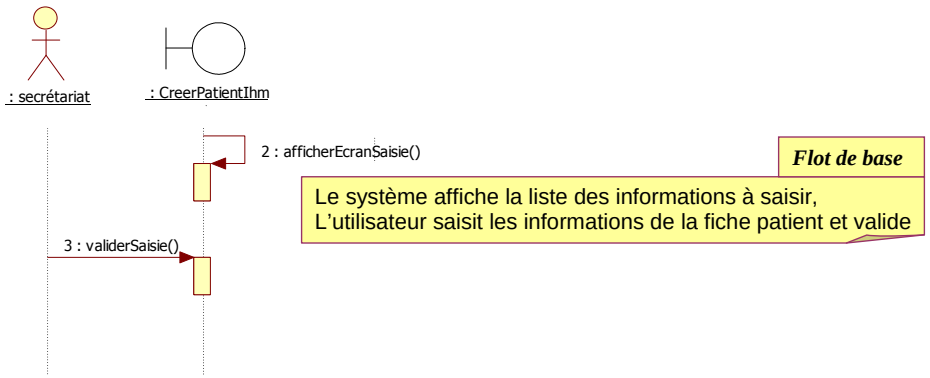
```
ctrlUseCase = new GererPatientCtrl();
```

Remarque : on aurait également pu effectuer l'instanciation du contrôleur en même temps que sa déclaration et écrire directement en une seule ligne:

```
GererPatientCtrl ctrlUseCase = new GererPatientCtrl();
```

2.3 Saisie des caractéristiques d'un `Patient`

Continuons à travailler avec le diagramme de séquence ……et le plan type détaillé



Les étapes suivantes 2 et 3 consistent à ***saisir les caractéristiques du nouveau patient***.
Si vous regardez le code de la classe **`GererPatientIHM`**, vous constaterez que le code de ces 2 étapes est entremêlé dans la méthode `creerPatient`. En effet, dans le cadre de ce TP, nous souhaitons réaliser une application en mode console. Pour plus de convivialité, pour chaque saisie (`validerSaisie`), nous affichons des messages indiquant à l'utilisateur ce qu'il est en train de saisir (`afficherEcranSaisie`)

Mais qui écrit des applications en mode console de nos jours ?

Quand nous réutiliserons ce même diagramme pour implémenter une `IHM` graphique, la décomposition des deux étapes de saisie prendra alors tout son intérêt :

- Tout d'abord, il sera nécessaire d'`afficherEcranSaisie` c-a-d de proposer à l'utilisateur un formulaire à remplir avec tous les champs du `Patient` (sous forme de Java/Swing ou interface HTML)
- Puis, l'utilisateur complétera les champs et un bouton de type Valider permettra de `validerSaisie`.

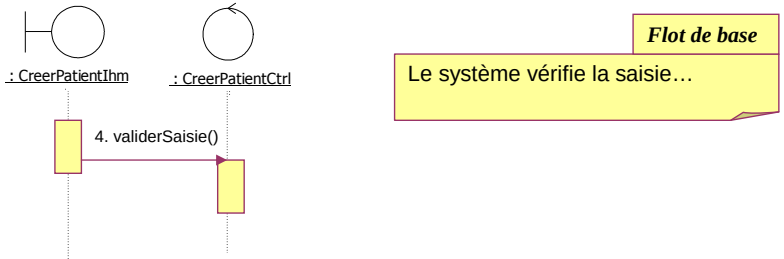
Vous constaterez pour l'instant, que dans le code de la méthode `creerPatient`, aucun objet `Patient` n'a encore été créé. Les caractéristiques du nouveau `Patient` ont été stockées pour le moment dans des variables temporaires.

```
(String nir, String nom, String prenom, Date dateNaissance, boolean isMale,String
telephone, String portable, String email, String numero, String rue, String voie,
String batiment, String codePostal, String ville, String pays, String medecinTraitant,
Integer idPersonne)
```

Le code de saisie des caractéristiques d'un `Patient` a été écrit afin de vous faciliter la tâche…

2.3 Création d'un objet de type `Patient`

Continuons à travailler avec le diagramme de séquence ……et le plan type détaillé



Le message `4.validerSaisie` du diagramme de séquence nous indique que la **Vue** doit transmettre au **Contrôleur** un message contenant les informations saisies.
Côté implémentation JAVA, pour une meilleure lisibilité du code, nous avons décidé que dans la classe `GererPatientCtrl` la méthode `creerPatient` contiendra les traitements que doit effectuer le contrôleur lors de la réception du message `4.validerSaisie` du diagramme de séquence.

Etape 6

Travail à faire : Afin de matérialiser l'envoi du message de la **Vue** au **Contrôleur**, vous devez rajouter dans la couche présentation (classe **`GererPatientIHM`**) à la fin de la méthode **`creerPatient`** c-a-d après la saisie et , une instruction du type :

```
ctrlUseCase.creerPatient(? ? ? ?);
```

Le contrôleur (`GererPatientCtrl`) devra donc contenir la méthode `creerPatient`.
Nous y reviendrons par la suite.

… mais que représente les ? ? ? ? :
La question que l'on peut se poser maintenant est :

Que doit-on transmettre comme argument à la méthode `creerPatient` ***?***

→ *Proposition n°1 :* Transmettre *la liste de variables* saisies précédemment ne serait pas une bonne solution (lourd en écriture et en maintenance)… **pensons objet ! ! !**

→ *Proposition n°2 :* Un objet qui contient les caractéristiques précédentes est un *objet de type* `Patient`…
…. alors pourquoi ne pas utiliser un objet de ce type ?
Rappelez-vous de la présentation du modèle MVC : « ***la couche présentation n'a pas le droit d'attaquer directement la couche métier*** ». En d'autres termes, nous devons éviter de manipuler des objets métier de type `Patient` dans la couche présentation (**`GererPatientIHM`**).

→ ***La bonne solution*** *:* utiliser un ***objet*** qui permet de ***stocker uniquement l'état (caractéristiques)*** d'un `Patient` ***pour transférer ces données jusqu'au Contrôleur***.
Cet objet ne doit pas être un objet métier (afin de maintenir l'indépendance Modèle / Vue) , ***seul son état*** nous intéresse, il n'a pas besoin d'avoir de comportement (c-a-d pas de méthode, juste des attributs)
Un tel objet s'appelle un **`DTO (Data Transfert Object).`**

2.3.a Mise en place des objets DTO dans l'application

↳ **Un DTO** est donc une classe qui possède les mêmes attributs que ceux de la classe métier.

→ Cette classe est composée uniquement **d'attributs et de getteurs/setteurs** (respect du principe d'encapsulation) et des 3 méthodes classiques **toString** et **equals, hashcode** (pour pouvoir manipuler les Collections)

→ Il n'y aura ***aucune méthode de traitement*** dans le DTO (dans le cas de PersonneDTO, la classe ne contiendra pas la méthode calculée getAge).

→ Par contre, comme dans la classe métier correspondante, le DTO redéfinira également les 3 méthodes héritées de la classe Object, à savoir **toString** et **equals, hashcode** (pour pouvoir manipuler les Collections)

→ Un **DTO** est un objet de **Transfert** : c'est une ***copie de l'objet métier***.
Il n'y aura donc aucune vérification sur la nature des données transportées par le DTO (pas d'appel à la méthode verifierNir, ni de test sur l'existence ou non d'une Adresse: ce sont les objets métiers qui se chargent de vérifier ces règles). Un DTO "n'est donc pas risqué" ce qui signifie que le ***DTO ne lèvera aucune exception !!!*** (c-a-d qu'aucune CabinetMedicalException ne doit apparaître dans les DTO de notre application)

→ Un DTO ne sera pas enregistré dans le support de persistance (c'est l'objet métier qui l'est).
Par contre un DTO, comme son nom l'indique est un objet de ***T**ransfert* : il peut être amené à aller sur le réseau (dans le cas d'une architecture multi-tiers). C'est pour cela que **le DTO doit implémenter Serializable.**

→ En ce qui concerne les constructeurs, on peut se contenter dans un premier temps du constructeur par défaut puisqu'on dispose des setteurs pour donner un état à un objet DTO. … mais rien ne vous empêche de définir les mêmes constructeurs que ceux disponibles dans la classe métier…
Prenez dorénavant l'habitude d'écrire une classe DTO pour chaque classe métier nouvellement créée.

→ Les DTO sont donc des classes qui traversent toutes les couches du modèles **MVC** .
- **Modele → DTO → Vue (via Ctrl) :** Lors d'une recherche sur un objet métier, le DTO transporte les données du Modèle que la Vue doit afficher.
- **Vue → DTO → Modele (via Ctrl) :** Lors d'une création d'un objet métier, le DTO contient les données du formulaire de la Vue et les transportent jusqu'au Modèle.

Ainsi, le DTO permet de sécuriser le support de persistance : *en manipulant un DTO, on ne risque pas de modifier les données réelles, puisqu'on travaille* ***juste sur des copies****.*

→ Afin que toutes les couches (**M,V,C**) puissent utiliser les DTO, on place habituellement les DTO dans une ***couche transversale accessible à toutes les couches appelée également « couche de service »*** et représentée dans notre application par le package com.iut.cabinet.**user**.

Etape 3

Travail à faire : Importer dans le paquetage com.iut.cabinet.user, les classes PersonneDTO, PatientDTO et AdresseDTO disponibles sur la zone libre.
Consulter le code de ces classes pour illustrer les explications données précédemment.
Remarques : Les champs adresse et unAscendant de PersonneDTO sont bien sûr respectivement de type AdresseDTO et PersonneDTO. La classe PatientDTO hérite de PersonneDTO.

2.3.b Mise en place des Helper pour favoriser la conversion des objets métiers ↔ objets DTO

Pour pouvoir utiliser les DTO dans notre application, il nous manque une dernière classe à écrire, c'est la **classe Helper**. La classe Helper permet de transmettre les données du DTO à l'objet métier et inversement.
Elle est donc composée initialement de deux méthodes de type :

```
public static MaClasseDTO toMaClasseDTO (MaClasse unObjetDeMaClasse) ;
public static MaClasse toMaClasse (MaClasseDTO unObjetDeMaClasseDTO) ;
```

Remarque : la conversion se fait en appelant un constructeur par défaut et des setteurs.

Etape 4

Travail à faire : Depuis la zone libre, importer dans le paquetage *com.iut.cabinet.**application*** :

→ la classe **HelperPatient** qui propose les méthodes suivantes:

```
public static PatientDTO toPatientDTO (Patient unPatient) {…}
public static Patient toPatient (PatientDTO unPatientDTO) {…}
```

→ ainsi que la classe **HelperAdresse** (puisque utilisée dans la classe HelperPatient)*

→ ainsi que la classe **HelperException**. Une HelperException sera lancée si on transmet en paramètre d'entrée une référence nulle. En effet, il est impossible d'appliquer des getteurs/setteurs sur des références nulles sous peine de bug du programme (NullPointerException). En testant la référence d'entrée et en levant si nécessaire une HelperException, on évite ce genre de bug à l'exécution.

Remarque : Notez également dans le code, que ce n'est que dans le Helper, lors de la transformation **DTO → Metier**, que les exceptions métier (**CabinetMedicalException**) risquent de se déclencher…
Bien sûr, on traite pas les exceptions métier dans les Helper, mais on les fait les remonter…

***Remarques :* zoom sur le code donné :**
↳ Dans un premier temps, notez dans la méthode **toPatientDTO** de la classe **HelperPatient** que la mise à jour du champ Adresse est réalisée grâce à l'instruction suivante :

```
patDTO.setAdresse(HelperAdresse.toAdresseDTO(unPatient.getAdresse()));
```

En effet, l'appel à HelperAdresse est indispensable car il permet de convertir l'objet Adresse du Patient en objet AdresseDTO du PatientDTO

↳ Dans un second temps, pour le champs de type Personne représentant l'ascendant :

```
if (unPatient.getUnAscendant()==null) patDTO.setUnAscendant(null);
                                        // n'oubliez pas ce test à null  à cause du cast
else patDTO.setUnAscendant(HelperPatient.toPatientDTO(
                                                    (Patient) unPatient.getUnAscendant()));
```

2.3.c Création d'un objet PatientDTO dans la couche présentation (GererPatientIHM)

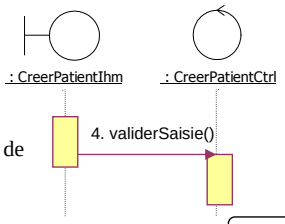
Revenons à la classe **GererPatientIHM**.

Etape 5

Travail à faire : En fin de saisie, dans la méthode creerPatient de la GererPatientIHM, juste avant l'appel à la méthode creerPatient du contrôleur, **stocker les données saisies dans un objet patDTO de type PatientDTO** (avec un ascendant null et un objet de type AdresseDTO que vous aurez préalablement également créé.)

```
PatientDTO patDTO = new PatientDTO(…constructeur à compléter avec les données
  saisies : consulter la classe PatientDTO pour connaître l'ordre des paramètres);
```

2.3.d Transmission au Contrôleur de l'objet `PatientDTO` (créé dans la Vue)



Nous allons maintenant enfin pouvoir compléter l'instruction que nous avions laissée en suspend et qui permet de coder l'étape 4 du diagramme de use case « `ValiderSaisie` » en transférant les données de la vue au contrôleur dans un (des) objet(s) de type `DTO`.

Etape 6

Travail à faire :

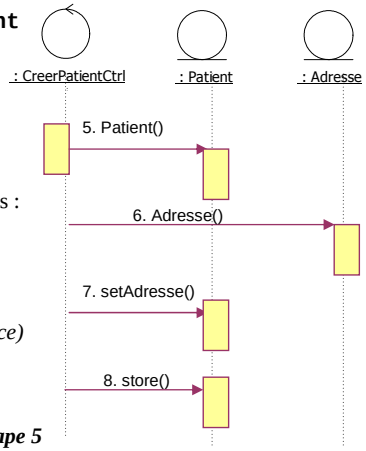
→ Compléter le message envoyé par la **Vue** au **Contrôleur** en transmettant le nouvel objet `patDTO` `PatientDTO` juste créé via la méthode `creerPatient`

```
ctrlUseCase.creerPatient(patDTO);
```

→ ***Afin de supprimer l'erreur de compilation***, vous devez procéder à la déclaration dans la couche application (classe `GererPatientCtrl`) de la méthode `creerPatient` :

```
public void creerPatient(PatientDTO unPatientDTO)
{ … }
```

2.3.e Réponse du Contrôleur au message envoyé par la Vue… … ou le rôle du Contrôleur dans la création d'un `Patient`



Rappelons le rôle du `Contrôleur` dans le modèle `MVC`: le contrôleur « est en fait **le cœur de l'application.**
Il assure ainsi l'interface avec l'utilisateur en faisant exécuter sa demande ***par la couche métier***…. ».

Son rôle lors de la création d'un patient se décompose en deux étapes :

- **étape n°1 : Créer un objet métier à partir du `DTO`**
  (messages 5,6,7 du diagramme de séquence)
  … afin d' …
- **étape n°2 : Assurer la persistance des objets métiers**
  (messages 8.`store` du diagramme de séquence)

Etape n°1 : Créer un objet métier à partir du `DTO`

Nous devons maintenant nous intéresser plus particulièrement à ***l'étape 5 du diagramme de séquence*** où le contrôleur procède à la création de l'objet métier `Patient`
(Les étapes 6 et 7 seront incluses dans notre code de l'étape 5, puisque d'après notre implémentation un `Patient` a comme attribut un objet de type `Adresse`, on peut dire qu'un `Patient` « embarque » une `Adresse`, c-a-d que l'objet `Adresse` devra être créé au préalable…)

Etape 7

Travail à faire :
Compléter la méthode `creerPatient` du Contrôleur `(GererPatientCtrl)` en commençant par déclarer un objet `unPat` de type `Patient` et instancier-le en appelant la méthode `toPatient` de la classe `HelperPatient` prenant comme paramètre l'objet de type `PatientDTO` provenant de la vue :

```
Patient unPat; // Déclaration
unPat = HelperPatient.toPatient(unPatientDTO); // Instanciation
```


→ **Propager les exceptions dans la couche application** `(GererPatientCtrl):` Ne traitez pas les exceptions dans le **C**ontrôleur, laissez-les remonter : elles seront traitées dans la Vue (couche de présentation).
Par exemple, s'il y a un problème avec les données saisies, il faudra demander à l'utilisateur de recommencer la saisie, jusqu'à l'obtention d'une donnée correcte… Il faut donc redonner la main à l'utilisateur. C'est pourquoi, il faut faire remonter l'exception métier à la couche appelante (couche présentation) où elle sera traitée afin de procéder à une nouvelle saisie…

Etape 7

Travail à faire :
→ La méthode **`creerPatient`** de la classe **`GererPatientCtrl`** doit propager (lancer) les exceptions :

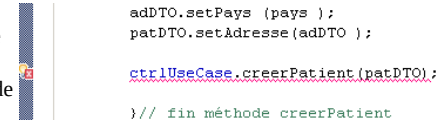
```
public void creerPatient(PatientDTO unPatientDTO)
            throws CabinetMedicalException, HelperException
{ … }
```

→ Traiter dans la Vue (couche présentation : `GererPatientIHM`) les exceptions lancées par le Contrôleur

Etape 8

Travail à faire :
➢ Retour sur dans la couche présentation (paquetage `com.iut.cabinet.presentation`):

→ Dans la méthode `creerPatient` de la classe `GererPatientIHM` , il faut désormais attraper les éventuelles exceptions et informer l'utilisateur de la survenue d'une erreur (pour l'instant, on se contentera d'afficher le message embarqué par l'exception.



```
adDTO.setPays (pays );
patDTO.setAdresse(adDTO );

ctrlUseCase.creerPatient(patDTO);

}// fin méthode creerPatient
```

Aidez-vous de l'assistant d'Eclipse (petite croix rouge) pour générer automatiquement les `try..catch` et compléter les `catch` en affichant le message délivré par de l'exception.

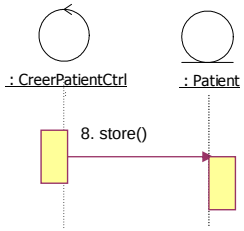
→ A l'aide par exemple d'un booléen et d'une boucle `while` modifiez ensuite votre code pour pouvoir recommencer la saisie tant qu'une exception est récupérée (c-a-d tant que l'on passe dans un bloc `catch` …)

Travail à faire : Tester votre application ! ! ! !
Dans la méthode `creerPatient` du contrôleur (`GererPatientCtrl`), juste après l'appel du `Helper`, procéder à un affichage de votre objet métier `Patient` (`unPat`) pour contrôler la validité de l'étape de création d'un objet métier par le contrôleur (étapes 5,6 et 7 du diagramme de séquence).
Rappel de nir « valides » :

`260058700112367` `297112A10102401`
`191128708545628` `168072B12345652` … *et votre propre nir ! ! !*

Vous pouvez récupérer sur la zone libre un jeu d'essai à personnaliser `JeuEssaiPatient.txt` que vous pourrez utilisé dans votre fichier **`GererPatientIHM`** durant votre phase de développement afin de vous éviter de perdre trop de temps dans la saisie des caractéristiques… Effectuer quand même au moins un test complet avec la saisie des données …

Etape n°2 : Assurer la persistance des objets métiers



Flot de base

Le système … enregistre le nouveau patient, et affiche un message de prise en compte

Cette étape correspond au dernier message du diagramme de séquence (`8.store`)
→ Dans ce TP, nous allons procéder à l'enregistrement d'un nouveau `Patient` dans un fichier.
En réalité, comme dans le TP précédent nous stockerons dans le fichier **`cabMedPersonne.data`** une liste de `Personne` c-a-d une collection de `Personne`.

→ Quelle que soit l'option choisie (créer un patient, modifier un patient, supprimer un patient ou lister tous les patients), nous travaillerons toujours à partir de la liste de `Personne` contenue dans le fichier de persistance. La **création d'un `Patient`** revient donc à ajouter un `Patient` dans la liste de persistance grâce au processus suivant :

- **a.** charger dans une `Collection` la liste de `Personne` initialement contenue dans le fichier
- **b.** créer un nouvel objet de type `Patient`
- **c.** ajouter cet objet à la liste
- **d.** sauvegarder la nouvelle liste dans le fichier(nouvelle liste incluant bien sûr le nouveau `Patient`)

Le **point b** (créer un nouvel objet de type `Patient`) a été traité dans la partie précédente.
Nous allons rapidement traiter le **point c**.

Etape 7 (suite)

Travail à faire : Dans la méthode `creerPatient` du contrôleur (`GererPatientCtrl`) :

→ Déclarer **`maListe`** de manière à manipuler une liste de `Personne` au travers d'une *collection* :

```
Collection<Personne> maListe ;
```

Remarque : *L'utilisation d'une référence de type `Collection` peut être utilisée car les Collections sont des objets polymorphes dont la classe mère est la classe `Collection` (Le surclassement est possible)…*
L'instanciation de `maListe` (`ArrayList` ou `LinkedList` ou autre…) nous importe peu ici puisque cette instanciation a déjà été réalisée lors de la désérialisation (dans le cas d'un support de persistance de type fichier, la méthode de désérialisation n'est autre que `findAllPersonnes` de la classe `PersonneDAOFichier`)
Nous traiterons réellement le point (**a**) à la page suivante … Pour l'instant, nous considérons que nous travaillons directement sur une référence valide de type `Collection` (valide c-a-d référence non nulle).

→ Placer-vous après la création du `Patient` `Patient unPat` (c-a-d après l'appel au `Helper`) et ajouter ce nouveau `Patient` à la collection (`maListe`).

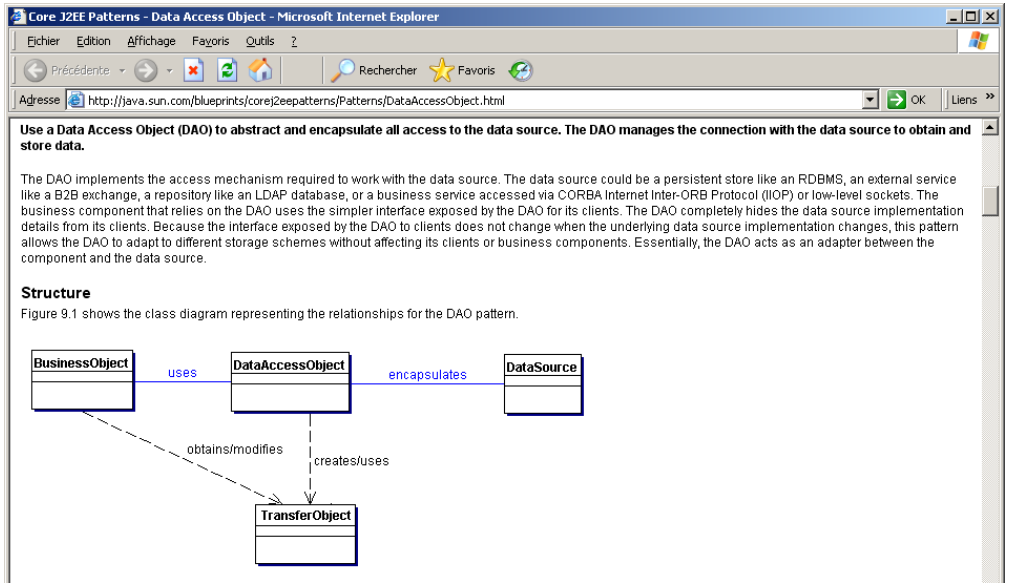
Il reste à traiter les **points a et d**.
→ ***Mais où placer les instructions relatives à la sérialisation/désérialisation ?***
Dans les classes métiers (`Personne, Patient, Professionnel, …`) ?
directement dans le contrôleur (`GererPatientCtrl`) ?
Afin de pouvoir changer aisément de support de persistance (… dans ce TP on va utiliser un fichier, mais plus tard, ce sera une Base de Données …) , ***il faut éviter d'imbriquer les ordres de persistance dans les classes métiers ou le contrôleur.***

La solution consiste à utiliser un pattern **DAO** (**D**ata **A**ccess **O**bject) qui encapsule les opérations de persistance dans une **nouvelle couche** dite ***« couche d'accès aux données »***

La couche métier devient alors indépendante du support de persistance (c-a-d de la gestion des Entrées/Sorties) … Si le support de persistance est modifié, il suffit *juste d'écrire un nouveau DAO* adapté à ce nouveau support : la classe métier restant quant à elle inchangée…

2.4 Persistance des objets métiers à l'aide du pattern DAO (Data Access Object)

Le pattern DAO est sans doute le modèle de conception le plus utilisé dans le monde de la persistance



→ Une **classe de type DAO** doit proposer des méthodes pour implémenter le **`CRUD`**.
Cet acronyme (**`C.R.U.D.`**) désigne les **4 opérations de base de la persistance** :

- **`C`***`reate`* (ou `store`) pour créer une nouvelle entité dans le support de persistance
- **`R`***`etrieve`* (ou `find` ou `load`) pour retrouver une ou plusieurs entités dans le support de persistance
- **`U`***`pdate`* pour modifier une des entités du support de persistance
- **`D`***`elete`* (ou `remove`) pour supprimer une entité du support de persistance

Pour chacune de ces opérations, on peut trouver ***plusieurs variantes de signatures des méthodes correspondantes*** dans les DAOs (c-a-d des méthodes surchargées)

→ Dans le cadre de notre application (architecture centralisée), nous développerons les classes DAO directement dans le paquetage ***metier***.
Le DAO devra porter le même nom que la classe métier.
Afin que tous les objets métier d'une même classe puisse partager le même DAO, nous avons choisi pour ce TP de déclarer toutes les méthode du DAO comme des **méthode statiques.**
Dans le cadre de ce TP, le DAO sera donc une classe dite « tout `static` »

Pour permettre la création d'un `Patient` dans la liste, il sera nécessaire de pouvoir ***sérialiser/désérialiser*** une collection de `Personne` dans un **fichier** c-a-d :
- pour la sérialisation : implémenter le **`C`***`reate`* du DAO souvent codé par une(des) méthode(s) **`store`**
- pour la désérialisation : implémenter le **`R`***`etrieve`* du DAO souvent codé par une(des) méthode(s) **`find`**

Travail à faire : …Rien du tout…
On aurait bien sûr pu écrire une classe `PatientDAOFichier` si on avait souhaité manipuler une `Collection<Patient>` dans le `DAO` … Grâce au polymorphisme, nous avons choisi de manipuler dans cette application directement une `Collection<Personne>`, et donc nous pouvons ré-utiliser la classe `PersonneDAOFichier` qui mémorise une `Collection<Personne>` et qui a l'avantage d'avoir déjà été codée au TP précédent avec ses deux méthodes :

```
public static void storeAllPersonnes (Collection<Personne> uneListe)
public static Collection<Personne> findAllPersonnes ()
```

Il faudra bien sûr penser à transformer plus tard cette `Collection<Personne>` en `Collection<Patient>`

→ **Appel du `DAO`**

Rappel : Dans le processus de création d'un `Patient`, l'appel au `DAO` concernait la sérialisation/désérialisation de la liste c-a-d les points **a** et **d**

- **a. charger** la liste de `Personne` initialement contenue dans le fichier
- b. créer un nouvel objet de type `Patient`
- c. ajouter cet objet à la liste
- **d**. **sauvegarder** la nouvelle liste (incluant le nouveau `Patient`)

Etape 7 (suite)

Travail à faire :
Compléter la méthode `creerPatient` du contrôleur (`GererPatientCtrl`) afin de procéder à la **sérialisation /désérialisation** de la collection `maListe` en passant par la couche d'accès aux données (c-a-d appels aux ***méthodes du DAO***).

→ Pour la ***désérialisation***, compléter la déclaration de `maListe` :

```
Collection<Personne> maListe=PersonneDAOFichier.findAllPersonnes();
```

Une seule ligne suffit puisque la méthode `findAllPersonnes` nous renvoie une référence valide (non nulle) sur une Collection.
Relancer vers la Vue, l'éventuelle `CabinetTechniqueException.`

→ En fin de méthode `creerPatient` , effectuer la ***sérialisation*** de la Collection à l'aide de la méthode `storeAllPersonnes` de la classe `PersonneDAOFichier.`

→ Afin de respecter le plan type détaillé, n'oubliez pas d'informer l'utilisateur dans `GererPatientIHM` après la désérialisation que le nouveau patient a bien été rajouté…

→ Tester…

Flot de base

Le système … enregistre le nouveau patient, *et affiche un message de prise en compte*

Etape 9

Pour tester votre code, dans la méthode `creerPatient` du contrôleur (`GererPatientCtrl`), vous pouvez maintenant afficher « temporairement » (c-a-d uniquement utilisé durant le phase de développement) à la place du `Patient unPat` directement le contenu de votre collection . Rappelons que cet affichage peut se résumer en 1 seule ligne d'instruction :

```
System.out.println(maListe);
```

… à partir du moment où les méthode `toString` ont été redéfinies pour toutes les classes des objets stockés dans la collection…

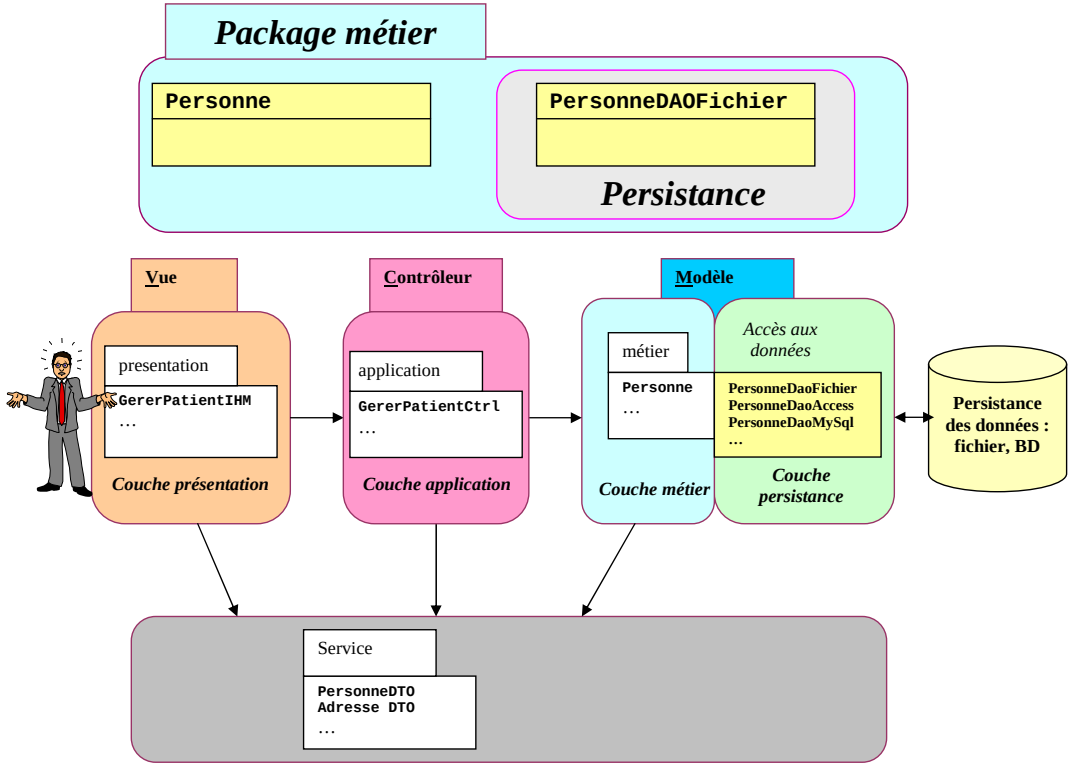
Remarque : Les deux derniers points du plan détaillé du flot de base ont déjà traité dans le constructeur de la classe `GererPatientIHM` par le choix de l'option 0 (`quitter`)
Pour l'instant, nous considérerons que notre use case `creerPatient` est terminé.

Flot de base

4. l'utilisateur choisit de quitter,
5. le système ferme le use case.

A Retenir sur le pattern DAO :
Le pattern DAO permet de s'abstraire de la façon dont les données sont stockées.
Il facilite une éventuelle "interchangeabilité" du support de persistance en encapsulant les accès aux données à des endroits distincts de l'architecture (plutôt que ceux-ci soient dispersés).
Afin de séparer le plus possible les entrées-sorties des classes métiers, il utilise des couches :
→ ***la couche DAO d'accès aux données : « cache » la connexion*** à tout support de persistance
C'est dans cette couche que se trouvent réellement les instructions d'accès aux données.
Les méthodes proposées par cette couche sont celles du CRUD (ajout, lecture, modification et suppression)

Avantage : La couche métier est complètement découplée du support de persistance.
Dans le cadre des bases de données, le programmeur utilisant le pattern DAO n'a plus besoin de connaître le langage SQL pour pouvoir manipuler des données, il lui suffit d'appeler la méthode du CRUD adéquate.



Perspectives futures :
Dans le dernier TP, vous pourrez améliorer votre pattern DAO en utilisant une **« fabrique » de DAO** pour faciliter dans le programme l'interchangeabilité des supports de persistance. En effet puisque de nos jours, le support de persistance utilisé dans les applications est plutôt une base de données qu'un fichier.
Comme la gestion des entrées/sorties peut différer d'un système à l'autre, il paraît donc évident qu'il faut écrire un fichier DAO pour chaque support de persistance….

3. Implémentation du cas : *Lister tous les patients* du use case `GererPatient`

```
------------------------------------
---- Gestion des Patients   ------
------------------------------------
-- 1. Creer un patient
-- 2. Modifier un patient
-- 3. Supprimer un patient
-- 4. Lister tous les patients
-- 0. Quitter
------------------------------------
```

→ Laissons de côté le flot alternatif de la création d'un patient (avec ascendant) pour l'instant, et intéressons-nous plutôt à l'affichage de la liste de tous les patients (option 4 du menu).

Etape 10

Travail à faire : Dans la **Vue** (`GererPatientIHM`), implémenter pour l'instant la méthode `listerPatients` de la manière suivante :

```
public void listerPatients()
      {
        // Déclaration d'une Collection de PatientDTO
          Collection<PatientDTO> maListe=null;

        // Récupération de maListe par appel à la méthode listerPatient
        // du contrôleur de use case
        // … à vous de coder cet appel …

          for(PatientDTO unPatientDTO : maListe)
          {
            // Affichage restreint d'informations concernant le patient
            System.out.println("-----------------------------------");
            System.out.println(" NIR : "+ unPatientDTO.getNir());
            System.out.println(" Nom : "+ unPatientDTO.getNom());
            System.out.println(" Prenom : "+ unPatientDTO.getPrenom());
            System.out.println(" DateNaissance : "+
                               DateUtil.toString(unPatientDTO.getDateNaissance()));
            System.out.println("-----------------------------------");
          }
      }
```

Remarque : Est-ce bien indispensable de rappeler que la vue ne manipule que des objets de type DTO …
Le code proposé ci-dessus est un affichage simplifié des caractéristiques d'un `Patient`, mais si on voulait obtenir toutes les informations contenues dans la méthode `toString` redéfinie dans la classe `Patient`, la boucle `for` pourrait être remplacée par un simple :

```
System.out.println(maListeDTO) ;
```

→Il faut maintenant s'intéresser au contrôleur, le cœur de l'application…

Etape 10

Travail à faire : Dans le **Contrôleur** (`GererPatientCtrl`), écrire une méthode qui permet de renvoyer la collection de `Patient` contenue dans le fichier. Cette méthode aura la signature suivante :

```
public Collection<PatientDTO>listerPatients() { // … à vous de coder }
```

La collection que vous allez récupérer depuis le DAO est une collection de `Personne` et nous souhaitons transmettre une **collection de `Patient`**… Tenez-en compte dans votre code …
Vous relancerez les éventuelles exceptions à la **Vue** qui les traitera…

Travail à faire : ***Tester votre application ! ! ! !***
Commenter tous les affichages que vous aviez écrit dans la méthode `creerPatient` du contrôleur (`GererPatientCtrl`).
Pour tester votre application, il ne vous reste plus qu'à jouer entre les options 1 et 4 du menu :

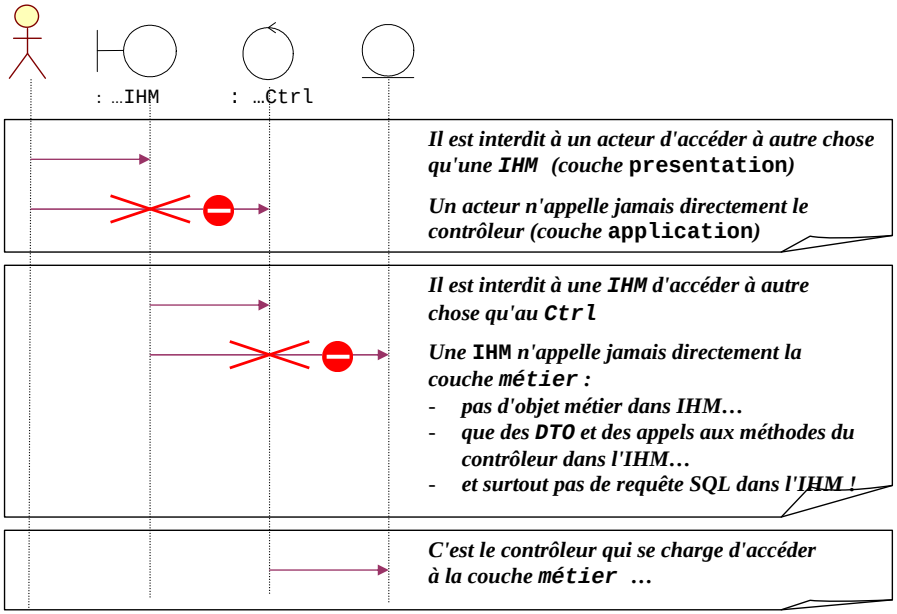
`1.Creer un patient` et `4.Lister tous les patients`

… ce qui doit absolument fonctionner :
→ Créer un patient (flot de base)
→ Lister tous les patients

… Toute amélioration et apport de nouvelles fonctionnalités seront bien évidemment prises en compte dans la notation …

Pour améliorer votre application :

➢ Il vous faut bien garder à l'esprit les quelques "règles" que nous venons de voir : chaque couche a une responsabilité. La mise en place de l'application sous forme de couches logicielles permet d'isoler les comportements/actions de chacune des couches. On ne "saute" pas de couches…



➢ En vous inspirant de tout ce qui a été fait jusqu'à présent, continuez à programmer les différentes options du menu. Afin de compléter les opérations du CRUD, il ne vous reste plus qu'à vous intéresser à ces deux parties.

4. Implémentation du cas : *Supprimer un Patient* du use case `GererPatient`

Intéressez-vous cette fois-ci à l'option 3 (Supprimer un Patient)…. A vous de coder …
Rappelons que quelle que soit l'option choisie (créer un patient, modifier un patient, supprimer un patient ou lister tous les patients), nous travaillerons toujours à partir de la liste de `Personne` contenue dans le fichier de persistance. Nous devons donc toujours charger la liste de `Personne` contenue dans le fichier avant de procéder à une quelconque modification et nous devons également la sauvegarder après modification, si modification il y a.

5. Implémentation du cas : *Modifier un Patient* du use case `GererPatient`

Afin de compléter les opérations du CRUD, il ne vous reste plus qu'à vous intéresser à cette partie…
…. A vous de coder …

6. Prise en compte d'un ascendant en créant un descendant …

Pour créer un `Patient` ayant un ascendant, vous suivrez la démarche indiquée dans le plan type détaillé :

Il faudra bien sûr modifier l'IHM et proposer au bon moment dans l'application, l'option d'ajout d'un descendant… A vous de coder …

Flot Alternatif

Le patient est responsable et possède des descendants

au point 4 du flot de base, l'utilisateur choisit d'ajouter un ou des descendants au patient

a. L'utilisateur ajoute un descendant,
b. Le système affiche la liste des informations à saisir sur la base des informations du patient (meme numéro d'immatriculation)
c. L'utlisateur saisit les informations du patient descendant, et valide,
d. Le système vérifie la saisie, ajoute le nouveau patient dans la liste des descendants du patient en cours,
e. Retour au point 4 du flot de base.

7. Contrôle des champs saisis

Afin de rendre votre application encore plus robuste, vous devez également contrôler les données saisies par l'utilisateur et obliger l'utilisateur à respecter certains formats que vous aurez défini.
Par exemple :
- le `nom` et le `prenom` ne devront contenir que des lettres (pas de chiffres)
- le format du numéro de `telephone` devra posséder 10 chiffres et pourra être saisi suivant l'une des trois syntaxes suivantes : **`xx.xx.xx.xx.xx ou xx-xx-xx-xx-xx ou xxxxxxxxxx`**
- Un `email` devra comporter un `@` et un `.` afin de respecter le format suivant: **xxx@xxx.xx**
- etc …

Ainsi de la même manière que vous avez contrôlé le `nir`, vous pouvez contrôler les autres champs saisis en complétant par exemple la classe `PatientRegle`, voire en créant une classe `PersonneRegle` et en lançant des `CabinetMedicalException`. Vous pouvez écrire vos tests avec de simple **`if`**, mais vous pouvez également utiliser des **expressions régulières** (API **`regex`** qui propose entre autres les classes **`Pattern, Matcher`** : pour en savoir plus effectuer une recherche sous **Google**)

Où doivent être effectués les contrôles de validité des données?

Il sera en fait nécessaire d'effectuer deux types de contrôles pour la validité des données :
- ***un contrôle "de surface"*** dans la couche **`presentation (IHM)`** afin d'apporter un certain confort de saisie pour l'utilisateur
- ***un contrôle "métier" plus approfondi*** dans la couche **`application (Ctlr)`** afin de respecter la cohérence des données à enregistrer dans notre Système d'Information.

→ Contrôle "de surface" des saisies dans la couche `presentation` (`GererPatientIHM`)
permet dans un premier temps une vérification rapide du format des données
Par exemple :
- pour le nir : 15 chiffres pour le nir, seul le 7ème caractère peut être un chiffre ou A ou B ,
- pour un email un `@` et un `.` doivent se trouver dans un email,
- etc …

Attention dans la couche **`presentation (IHM)`** , vous ne devez pas appeler les méthodes écrites dans la (les) classe(s) `PatientRegle`, `PersonneRegle,…`

Pourquoi ?
- D'une part parce que ces classes sont des classes de la couche **`metier`** (`PatientRegle`, `PersonneRegle,…`): elles contiennent des **règles métier (c-a-d un savoir faire de l'entreprise) donc ces règles ne doivent donc pas être exposées dans l'IHM.**
- D'autre part, les différentes couches de votre application (`presentation`, `application`, `Ctrl`) peuvent chacune être déployées sur un serveur différent. Si une classe de la couche `application` comme **`PatientRegle`** , il ne faudrait pas oublier de la redéployée sur le serveur `presentation` (maintenance)

→ Contrôle métier des saisies dans la couche `application` (`GererPatientCtrl`) permet de maintenir la cohérence des données enregistrées en respesctant des règles métiers. **C'est uniquement dans la couche `application` que les règles métier sont et doivent obligatoirement être appelées.**
Dans notre application, cet appel se fait "indirectement" grâce à l'appel de la méthodes du **`Helper`** qui permettent de convertir un objet **`DTO`** en **objet metier**.
Rappel : Les règles métier (méthodes des classes `XXXRegle`) sont appelées par les setteurs. Ainsi, lors de l'instanciation de l'objet métier et de la mise à jour de son état interne, la cohérence des informations est vérifiée à ce moment là par les setteurs.

→ Exemple avec le `nir`:
- ***un contrôle "de surface"*** dans la couche **`presentation (GererPatientIHM)`** : dans un premier temps : 15 chiffres pour le nir, seul le 7ème caractère peut être un chiffre ou A ou B. Pas de calcul du nir pour un contrôle de surface (on n'est pas sensé connaître la règle métier…)
- ***un contrôle "métier"*** dans la couche **`application (GererPatientCtlr)`** l'appel à la règle métier `verifierNIR` est effectué par la méthode `toPatient` de la classe `HelperPatient` qui appelle directement (ou indirectement) la méthode `setNir` de la classe `Patient`

Attention : la première partie du mini-projet `cabinetMedical` est terminée
Pour la fin de la semaine prochaine (jeudi soir minuit dernier délais),
vous enverrez par mail à vos sources à vos enseignants respectifs de TP
c-a-d une version zippée de votre répertoire `cabinetMedical`.
Vous déposerez également une copie de votre ZIP sur la zone libre
libre sous le répertoire ENSEIGNEMENT\DUT2\JAVA

Annexe 1 : Plan type du cas d'utilisation <Créer Patient>

1. Use Case :< créer un Patient >

1.1. Description

Ce use case permet la création d'un patient pour le cabinet médical.

1.2. Flot d'événements

1.2.1. Flot de base

1. Le système affiche la liste des informations à saisir,
2. l'utilisateur saisit les informations de la fiche patient et valide,
3. le système vérifie la saisie et enregistre le nouveau patient, et affiche un message de prise en compte,
4. l'utilisateur choisit de quitter,
5. le système ferme le use case.

1.2.2. Flots alternatifs

1.2.2.1. Le patient est responsable et possède des descendants

au point 4 du flot de base, l'utilisateur choisit d'ajouter un ou des descendants au patient

a. L'utilisateur ajoute un descendant,
b. Le système affiche la liste des informations à saisir sur la base des informations du patient (meme numéro d'immatriculation)
c. L'utilisateur saisit les informations du patient descendant, et valide,
d. Le système vérifie la saisie, ajoute le nouveau patient dans la liste des descendants du patient en cours,
e. Retour au point 4 du flot de base.

1.3. Exigences particulières

Un patient possède obligatoirement une immatriculation.

1.4. Pré-conditions

Une immatriculation est unique dans le système.

1.5. Post-conditions

1.6. Points d'inclusion

Aucun

1.7. Points d'extension

Aucun

1.8. Liste des acteurs participants

Gestionnaire de cabinet de recrutement

Annexe 2 : Diagramme de séquence <Créer Patient>