

TD1 : Conception Objet - principes de base

Relations de dépendance entre les classes

Une classe C1 dépend de la classe C2 ($C1 \neq C2$) dans les 3 cas suivants :

- un attribut de C1 est un objet, un tableau, ou une collection d'objets de C2 (attention, cette relation est représentée dans le diagramme de classe par une association),
- dans le code de C1, une méthode de C2 (méthode de classe ou d'objet) est utilisée,
- dans le code de C1, un objet C2 est créé (appel à un constructeur de C2).

Dans le diagramme de classes les dépendances sont représentés par les associations (premier cas), et dans les autres par le lien d'usage (stéréotype <<use>>).

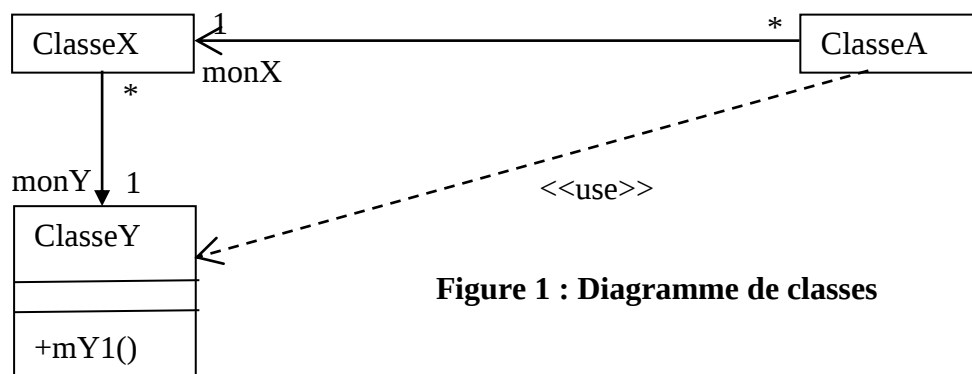


Figure 1 : Diagramme de classes

Questions :

1. Listez les dépendances des classes dans le diagramme de classe de la figure 1.
2. Listez les dépendances des classes dans le diagramme de classe de la figure 2.

Cohésion forte et couplage faible

Cohésion forte : une classe a un ensemble de responsabilités (méthodes) formant un tout cohérent, à savoir que le rôle/responsabilité de la classe peut être décrit en une phrase. La cohésion forte des classes facilite la compréhension des classes, celle de la structure du programme orienté objet, et donc :

- la réutilisation du code,
- la maintenance du code.

Couplage faible : minimiser les dépendances entre les classes. Deux moyens principaux permettent de retirer ou minimiser une dépendance.

1. remplacer une classe par une interface, dans ce cas là on minimise la nature de la dépendance sans la détruire totalement (voir figure 1).
2. technique d'enveloppement ou d'emballage d'une méthode, à savoir remplacer d'un appel direct (`ClasseY objY = monX.getObjetY(); objY.mY1();`) par un appel indirect (`objX.mY1();`). Ainsi la dépendance de la ClasseA avec laClasseY via la méthode m1 est éliminée. Cette technique nécessite que la méthode `mY1()` soit définie dans la classeX (`mY1() { return objX.getObjetY().mY1(); }`).

Le diagramme de classes de la figure 2 représente le même code que celui de la figure 1 après avoir éliminé les dépendances inutiles entre les classes.

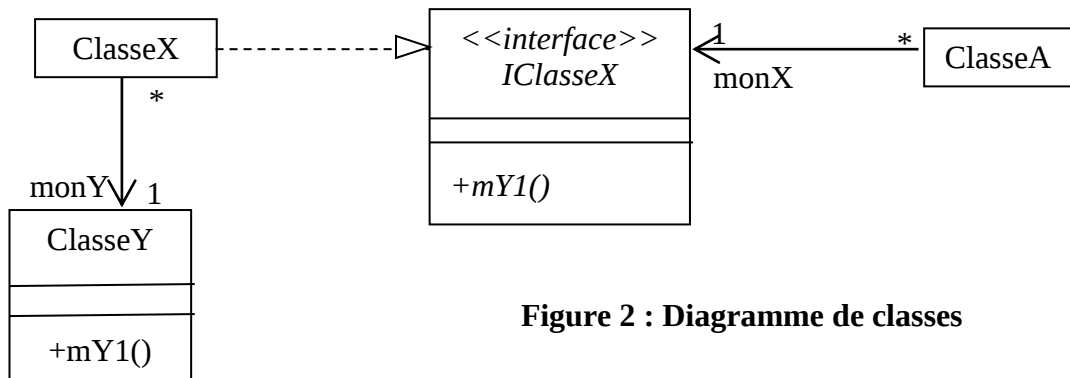


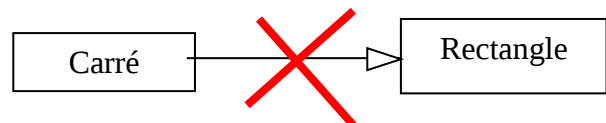
Figure 2 : Diagramme de classes

Principe de substitution de Liskov

Le principe de substitution de Liskov (LSP) est, dans le cadre de la programmation orientée objet, une propriété nécessaire à l'héritage/impémentation. Il a été formulé par Barbara Liskov et Jeannette Wing.

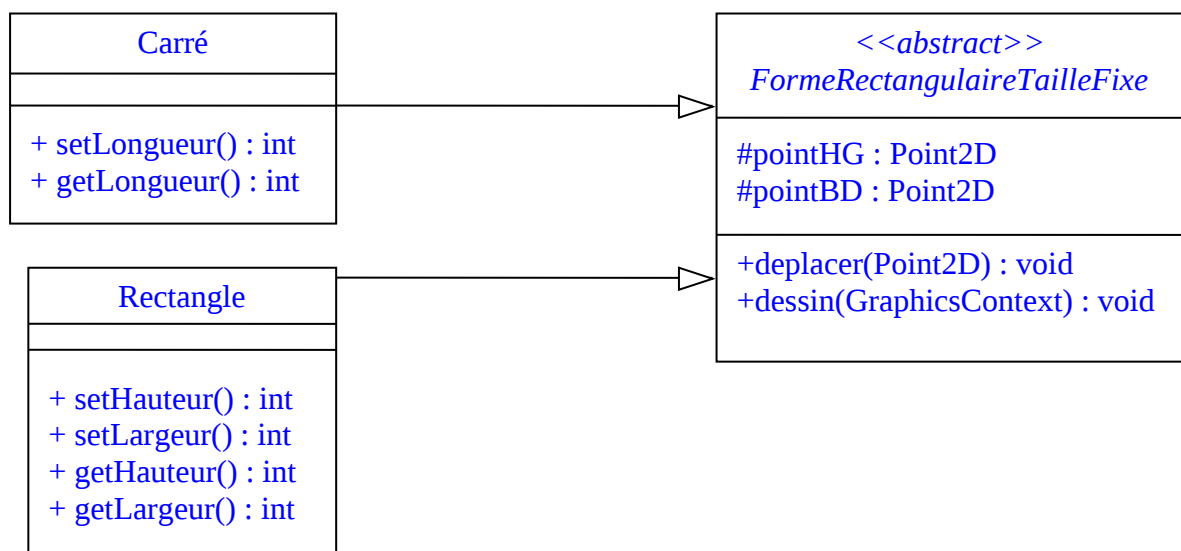
Barbara Liskov a reçu en 2008 le Prix Turing (quasiment l'équivalent du prix Nobel pour les informaticiens).

Une instance de la classe/interface T doit pouvoir être remplacée par une instance de la classe/interface G, tel que G est une sous classe/interface T, sans que cela ne modifie la cohérence du programme.



Le changement de taille d'un rectangle se fait via deux méthodes setLongueur() et setLargueur() ; or le changement de taille d'un carré se fait via une seule méthode setLongueur(). Donc Carré n'est pas une sous classe de Rectangle.

Par contre, Carré et Rectangle sont toutes deux sous-classes de FormeRectangulaireTailleFixe car elles peuvent avoir la même structure interne (2 points) et le même code pour la méthode déplacer et dessin.



Patron de conception



Dans le cadre du génie logiciel, un patron de conception (*design pattern* en anglais) est un concept destiné à résoudre des problèmes récurrents suivant le paradigme objet.

Les patrons de conception décrivent des solutions standard pour répondre à des problèmes de conception des logiciels. On peut considérer un patron de conception comme une formalisation de bonnes pratiques ou de solutions éprouvées.

Il ne s'agit pas de fragments de code, mais d'une manière standardisée de résoudre un problème qui s'est régulièrement posé par le passé. On peut donc considérer les patrons de conception comme un outil de capitalisation de l'expérience appliqué à la conception logicielle.

Inconvénients des Patron de Conception

L'utilisation des Patron de Conception n'est pas des plus simples et ne convient assurément pas à tout le monde. Les modèles de conception requièrent une conception détaillée. Il introduit en outre un niveau approfondi de complexité qui nécessite une attention de tous les instants aux détails. La charge de travail supplémentaire induite ne doit pas être sous-estimée.

Il en résulte que l'usage des modèles de conception peut être trop complexe pour les petites applications, et même pour bon nombre d'applications moyennes.

Les premiers patrons de conception ont été formalisés dans le livre « Design Patterns: Elements of Reusable Software », Gamma, Helm, Johnson et Vlissides en 1994. Ce livre, devenu un best-seller, décrit vingt-trois « patrons GoF » et comment s'en servir - Gang of Four – GoF.

Ils existent d'autres patrons de conception et aussi des anti-patrons de conception (usage à éviter) : code spaghetti, blob, culte du cargo, réinventer la roue, ...

Typologie des Patron de conception

Il existe trois familles de patrons de conception selon leur utilisation :

- Créateurs: les patrons de conception relatifs à la création des instances de classes (objets);
- structuraux: les patrons de conception relatifs à l'organisation des classes;
- comportementaux: les patrons de conception relatifs à la distribution des responsabilités entre classes et aux fonctionnements des algorithmes impliqués.

Singleton : un patron de conception de « création »

Patron de conception : « Singleton » répond à deux exigences :

- garantir qu'une unique instance d'une classe « Singleton » sera créée
- offrir un point d'accès universel à l'instance unique.

Ce patron de conception est tout indiqué pour implémenter des services qui :

- ont un fonctionnement identique au sein de l'application (ex: système de journal centralisé - log -, gestion de la configuration...)
- doivent pouvoir être appelés par toutes les couches de l'application. Il serait peu pratique de passer une référence au service à toutes les classes devant l'utiliser.

Questions

1. Transformez la classe java “SingletonExample” en un singleton. Indice : il faut interdire la création « libre » d’une instance de “SingletonExample”; et il faut autoriser l’accès à l’unique instance de SingletonExample.

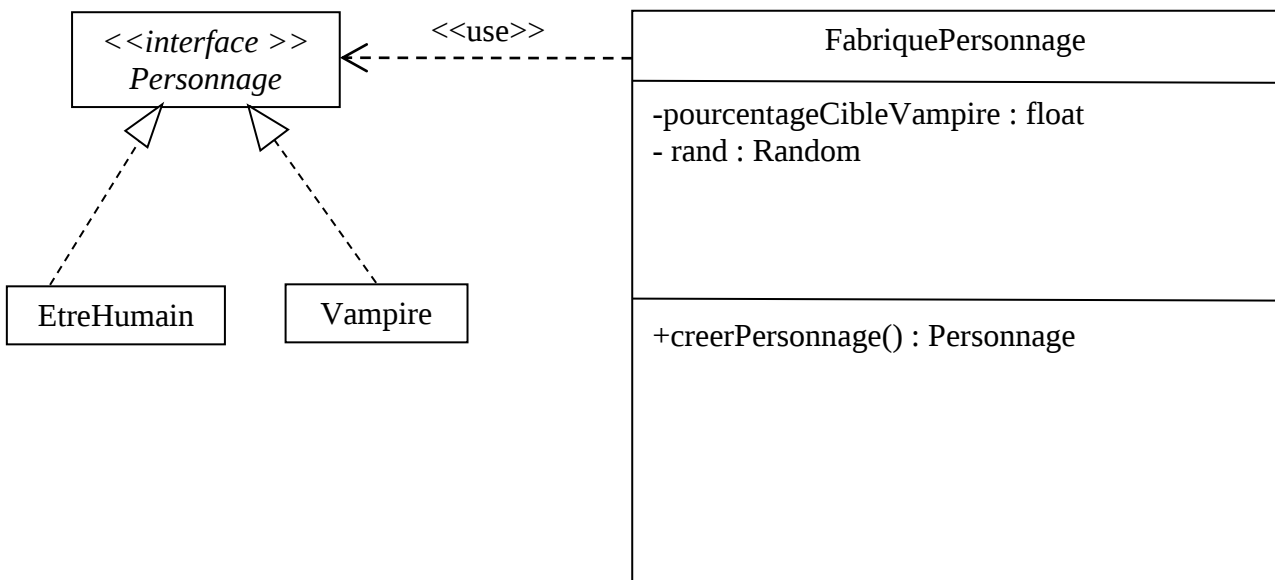
```
public class SingletonExample {  
    SingletonExample INSTANCE = new SingletonExample();  
    SingletonExample() {}  
    SingletonExample getSingletonExample() {  
        return INSTANCE; }  
}
```

2. Ecrivez le diagramme de classes contenant uniquement la classe SingletonExample

Méthode de Fabrication : un patron de conception de « création »

Une fabrique « method factory » crée des objets la classe exacte de l'objet n'est donc pas connue par l'appelant.

Les fabriques étant en général uniques dans un programme, on utilise souvent le patron de conception singleton pour implémenter les fabriques.



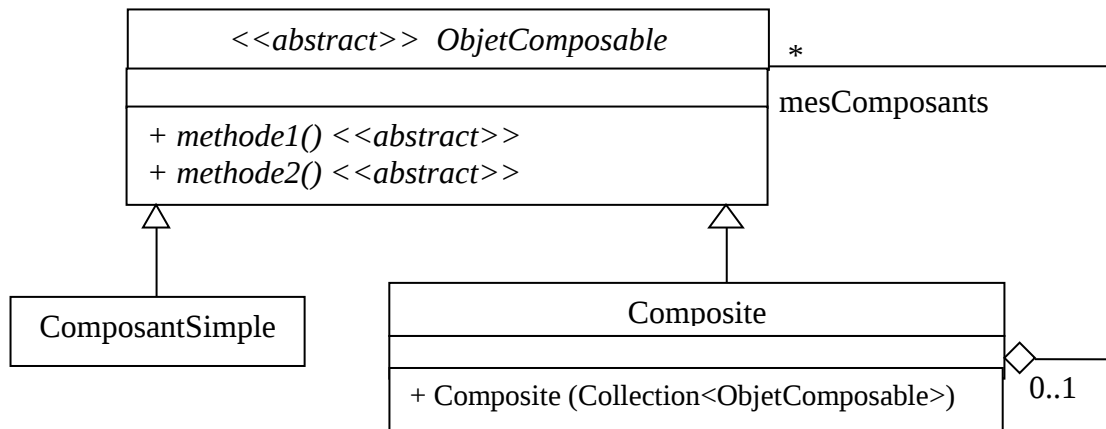
Questions

3. Transformez la classe “FabriquePersonnage” dans le diagramme de classes en un singleton.
4. Ecrivez le code en Java de la méthode *creerPersonnage()*. Cette méthode crée un Personnage qui est retourné. Ce personnage est un Vampire dans *pourcentageCibleVampire* des cas.

information : *rand* est un générateur de nombre aléatoire; *rand.nextFloat()* retourne un réel entre 0 et 1.

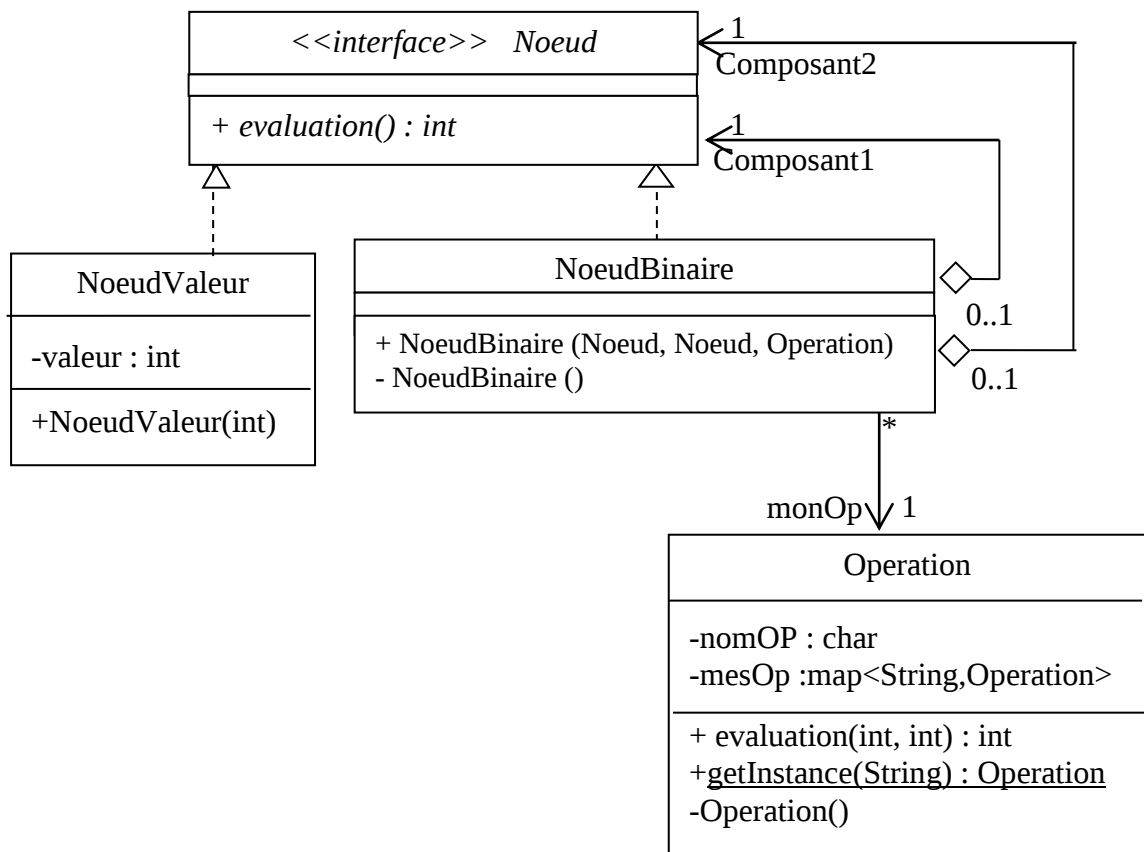
Composite/Composant : un patron de conception structurant

Le composite (groupes d'objet) a des propriétés (méthodes, attributs) similaire à ses composants (objet simple). Le « client » veut traiter les composites comme les composants. Ce patron permet aussi de représenter les structures d'arbres. Diagramme de classe représentant ce patron :



Dans une implémentation de ce patron : l'agrégation peut être un lien de composition et les objets composables peut avoir un nombre quelconque de méthodes abstraits.

Implémentation du patron « Composite » dans un diagramme de classes modélisant les arbres syntaxiques abstraits (AST, abstract syntax tree) représentant des formules arithmétiques sans variable et sans la négation d'une formule :



Questions

5. Ecrivez le code complet de la classe `NoeudBinaire` et `NoeudValeur` en Java.

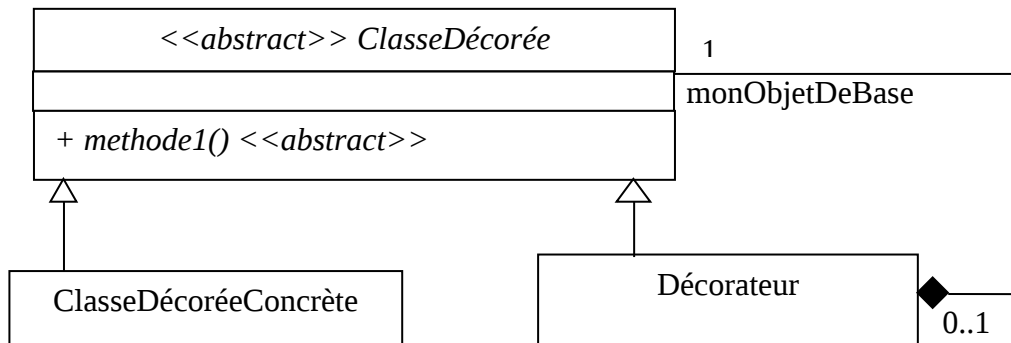
Décorateur: un patron de conception structurant

La façon la plus classique d'ajouter des fonctionnalités à une classe est d'utiliser l'héritage.

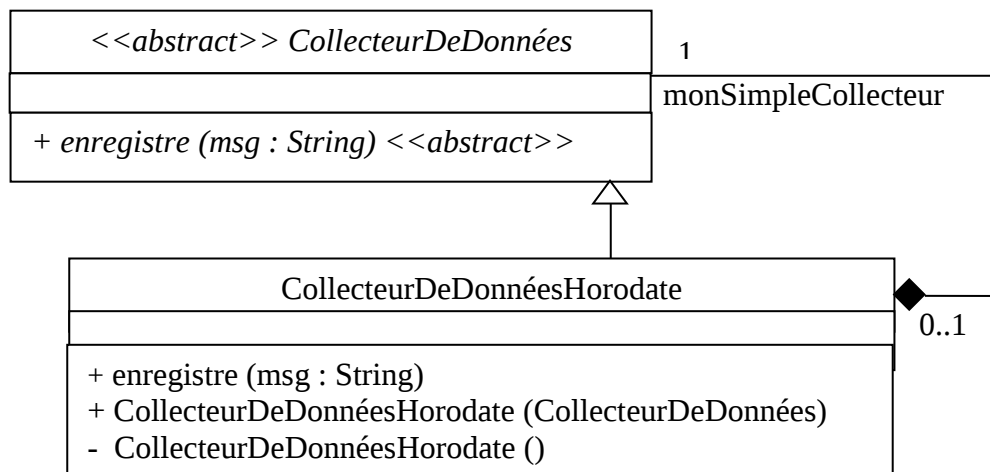
Le patron Décorateur « Decorator » attache dynamiquement des responsabilités supplémentaires à un objet. Il fournit une alternative souple à l'héritage, pour modifier les fonctionnalités de la classe « mère ». Car,

- l'héritage ajoute des fonctionnalités de façon statique,
- la classe « mère » peut être définie comme étant finale,
- En cas d'héritage, la classe fille doit être maintenue en cas de modification apportée à la classe « mère » par son éditeur.

Voici le diagramme de classes correspondant au patron « Décorateur » :



Par exemple nous avons la classe « Collecteur de Données » et on veut ajouter à chaque message construit par le collecteur, l'heure et la date d'émission. Nous voulions aussi définir une classe « collecteur » pour un affichage console, une autre pour enregistrer les informations dans un fichier, une autre pour afficher les informations dans une fenêtre JavaFX, une fenêtre SWING, , ...



Questions

6. Ecrivez le code complet de la classe `CollecteurDeDonnéesHorodate` en Java.
7. Expliquez pourquoi le constructeur sans argument de `CollecteurDeDonnéesHorodate` a la visibilité privée.
8. Utilisez le patron de conception « décorateur » dans la conception des jeux suivants.
Un jeu « entier » consiste à deviner un nombre entier (le mystère). Le joueur va essayer de deviner le mystère en effectuant des tentatives, c'est-à-dire des appels à la méthode `tester(nb : int)` ; le résultat de la dernière tentative est enregistré dans un message accessible via la méthode `getMsg() : String`. Le jeu « Chaud Froid » est un exemple de jeu « entier » ainsi que le jeu « min/max ».

Un jeu « d'entier » est dit « borné » si le nombre de tentatives pour deviner le mystère est limité: si le nombre mystère n'est pas trouvé après ce nombre de tentatives alors le joueur a perdu.

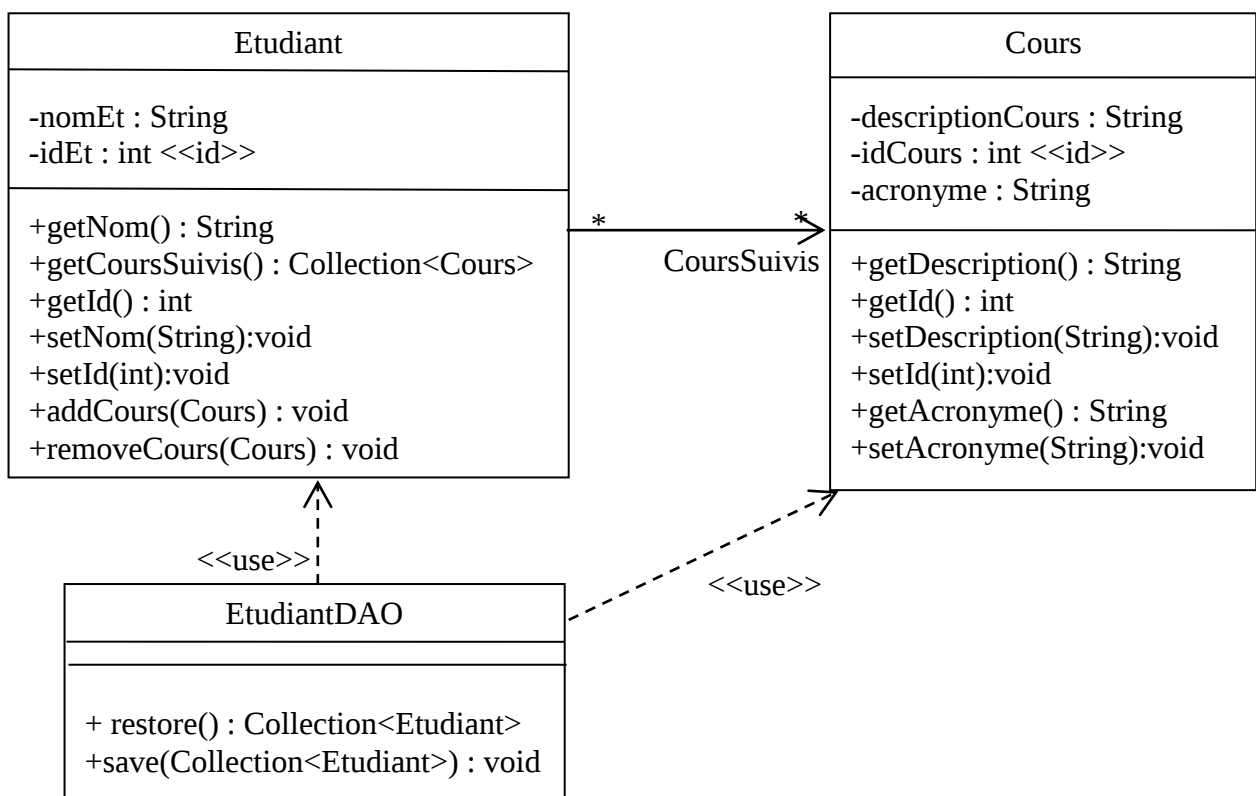
9. Ecrivez le code de la méthode *tester(int)* et celui de la méthode *getMsg()* dans le cas d'un jeu à borné en utilisant les implémentation non bornée.

Objet d'accès aux données (DAO) : un patron de conception structurant

Le patron objet d'accès aux données « Data Access Object » propose de regrouper les accès aux données persistantes dans des classes à part, plutôt que de les disperser. Il s'agit surtout de ne pas écrire ces accès dans les classes "métier".

L'utilisation du patron DAO permet de s'abstraire de la façon dont les données sont stockées au niveau des objets métier. Ainsi, le changement du mode de stockage ne remet pas en cause le reste de l'application. En effet, seules ces classes dites "DAO" seront à modifier (et donc à re-tester).

Cette souplesse implique cependant un coût additionnel, dû à une plus grande complexité de mise en œuvre.



Questions :

9. Listez les dépendances de la classe **EtudiantDAO**, de la classe **Etudiant**, et de la classe **Cours**.
10. Proposez un autre diagramme de classes relatif à « gestion des inscriptions à Ecole Excellence Zelta » limitant les dépendances entre classes.

Observateur : un patron de conception comportemental

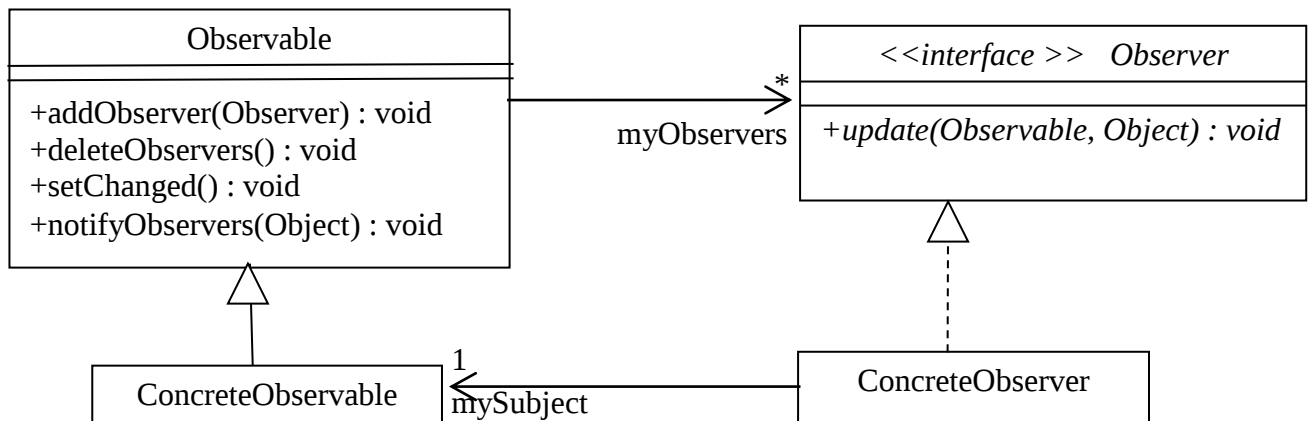
Le patron observateur « observer » est utilisé pour permettre a des objets d'envoyer un signal à d'autre objets qui jouent le rôle d'observateurs. Un « observable » est une instance de la classe *Observable* ou d'une classe fille ; un « observateur » est une instance de classe réalisant l'interface *Observer*.

Un objet désirant observer un « observable » doit s'enregistrer auprès de lui en appelant la méthode *addObserver* de l'observable. L'observable l'enregistre en l'ajoutant à sa collection *myObservers*.

Un observable doit envoyer une notification à ses observateurs chaque fois que son état change. Cette notification nécessite l'appel de la méthode *setChanged*, suivi de l'appel à *notifyObservers*.

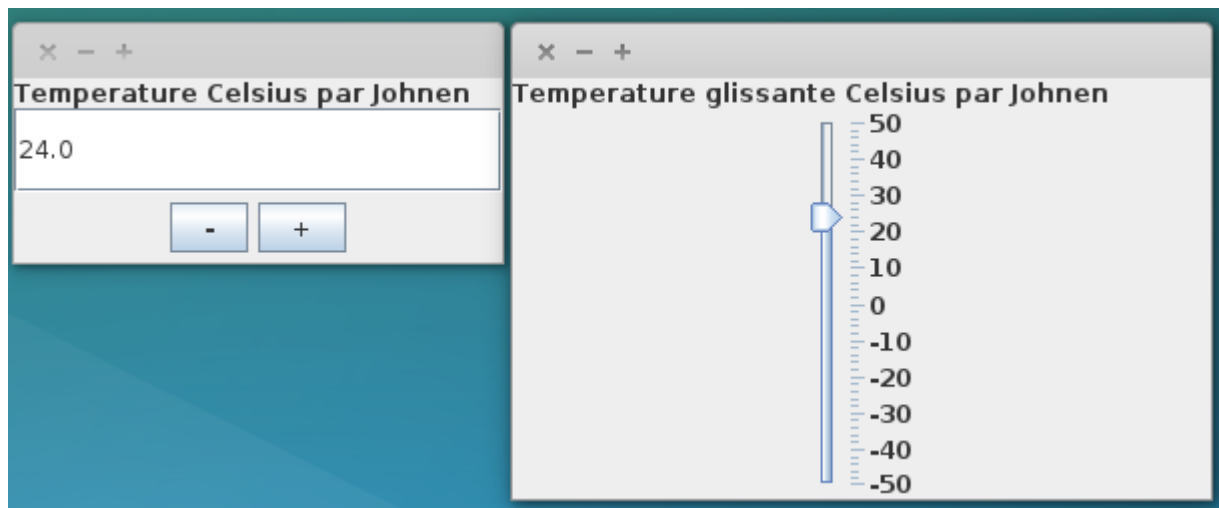
L'exécution de *notifyObservers* appelle la méthode *update* de chacun des observateurs de l'observable (donc la méthode *update* est exécutée par chaque observateur). Durant l'exécution de la méthode *update*, un observer peut obtenir des informations complémentaires par l'appel d'une ou de plusieurs méthodes de l'observable.

Information : La classe *Observable* ainsi que l'interface *Observer* font parties de l'API Java.



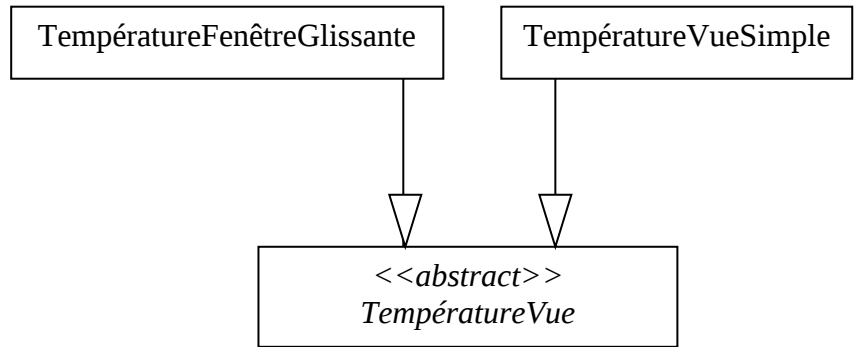
Thermomètre

L'objectif est de contrôler d'une température en degrés Celsius via plusieurs interfaces. La modification de la température dans l'une des vues doit mettre automatiquement à jour l'autre vue.



Questions

11. Compléter le diagramme de classes ci-dessous



Chaînage de méthodes : patron de conception comportementale

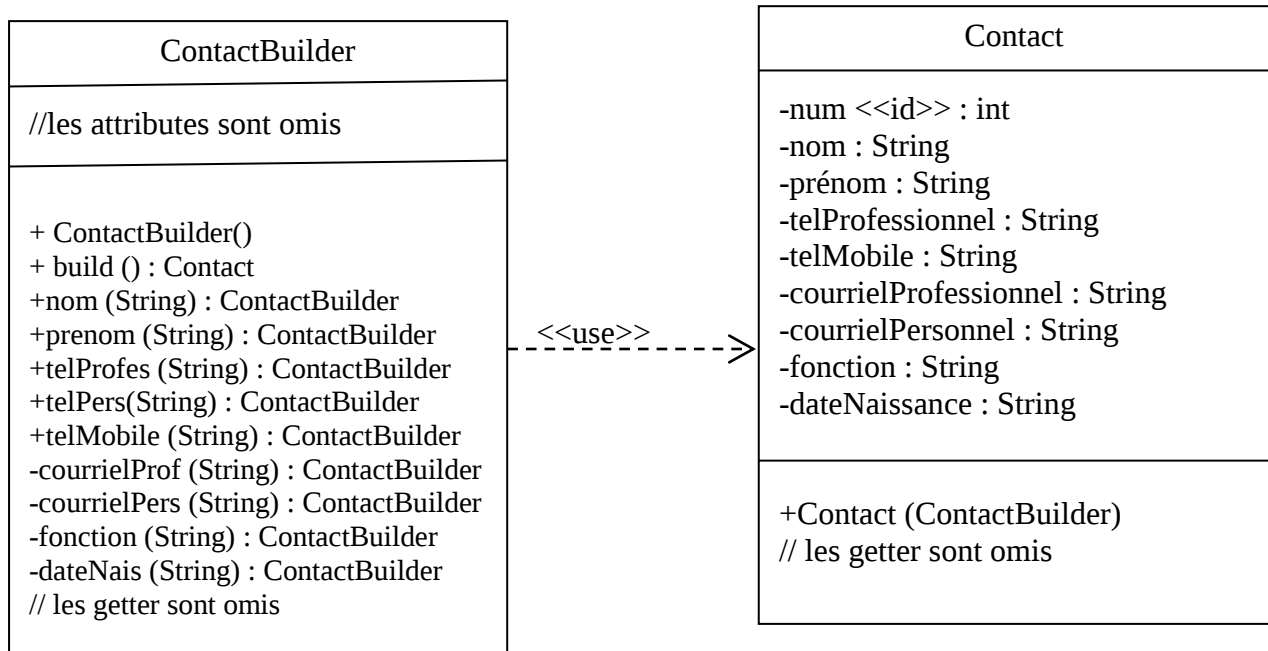
Un **chaînage de méthodes (method chaining)** consiste à agir en une seule instruction sur plusieurs méthodes du même objet, dans un but de plus grande lisibilité.

L'idée principale est qu'au lieu de qualifier chaque méthode par le nom de l'objet correspondant, ce qui conduisait à une lourdeur de style, ce nom d'objet reste par défaut actif dans toute l'instruction en cours. Le contexte est :

- défini par la valeur de retour de la méthode appelée,
- transmis à la méthode suivante;
- terminé par le retour d'un contexte vide (facultatif).

Le code est ainsi à la fois plus concis et plus clair.

Remarque : Ce patron de conception ne fait pas partie des 23 patrons de conception proposés par le « GoF ».



```

ContactBuilder cb = new ContactBuilder () ; // déclaration et instantiation
Contact contact ; // déclaration
cb.prenom("Isa").courrielProf("isa.dut@bord.fr").nom("Dut").fonction("DA") ;
contact = cb.build() ;

```

Monteur : patron de conception de « création » (version simplifiée)

Typiquement, le patron Monteur « Builder » sera préconisé dans les situations où au moins :

- l'objet final est imposant, et sa création complexe;
- beaucoup d'arguments doivent être passés à la construction de l'objet, l'usage du patron « Monteur » permet d'avoir un design et un code lisible;
- certains de ces arguments sont optionnels ou ont plusieurs variations.

Questions

12. Proposer une conception différente de « ContactBuilder » permettant d'ajouter d'autres propriétés au Contact (adresse personnelle, adresse professionnelle, ...) de manière uniforme et simple.