REGLE à respecter : Créer UN projet Java PAR exercice (à cause des paquetages à prévoir à chaque fois). Par ailleurs vos projets devraient tous intégrer la librairie Junit5.

Série 0 : Refactoring

Exercice 1 : Révisions sur l'encapsulation

Récupérez et lire le code de l'énoncé auprès de votre enseignant. Il s'agit d'une classe Pile qui a été écrite sans encapsuler ses données.

Que peut-on dire sur les problèmes du code client (ici la fonction main) qui ne respectent pas l'encapsulation ?

Supposons qu'on remplace les données de la classe Pile par une liste chainée (LinkedList) quelle est la conséquence sur le code client ?

Corrigez la classe Pile et réécrivez le code client (le main) tant qu'il est petit. Quels bénéfices en a-t-on tiré ?

Ecrivez une nouvelle classe Pile avec une LinkedList (au lieu d'un tableau) en gardant la taille MAX d'une pile. Que constatez-vous pour le code du client et celui les tests. Des changements à prévoir ?

Exercice 2 : Révisions sur le polymorphisme

Récupérez et lire le code de l'énoncé auprès de votre enseignant. Il s'agit d'un embryon d'application de vente en ligne des différents articles : livres, dvd, etc.

Remarquez que plusieurs tests unitaires ne passent pas (ils sont mis en commentaire). Il faudra qu'ils passent après votre réécriture du code. De plus les méthodes getPourTous() et memeArticle() sont très mal écrites.

Quels sont les défauts de ces méthodes ?

Corrigez ces problèmes.

Il faudra "cacher" au site de vente les articles (donc 2 paquetages).

Le site décide de vendre en plus, des BlueRay (par rapport aux DVDs la taxe passe à 0,30). Ajoutez cette possibilité ! Maintenant ça prend 2 minutes ! Ajouter un test (au site de vente) permettant d'insérer un BlueRay pour enfant et de vérifier qu'il est bien comptabilisé dans la liste des produits pour TOUS.

Faites le diagramme UML final des classes et des paquetages.

Exercice 3 : Cacher ! Toujours cacher

Le programme comprend deux classes Entreprise et Employe. La masse salariale de l'entreprise est la somme des **payes** (à ne pas confondre avec le **salaire** qui ne comprend pas les suppléments éventuels) de tous les employés. Actuellement, les secrétaires sont les seuls métiers/postes pour lesquelles les heures supplémentaires sont payées.

Donc, il faut écrire une classe Secretaire qui matérialise cette particularité du calcul de paye (donc prévoir une méthode getPaye()). Rectifiez les tests unitaires fournis.

D'autres types de postes seront embauchés à brève échéance avec des règles de calcul de paye spécifiques à chacun.

Il faudra "cacher" l'entreprise de ses métiers (2 paquetages). Donc une interface IEmploye serait la bienvenue…

Ecrivez une classe Planton dont les heures sup sont payées double !!!

Faites le diagramme UML final des classes et de leur paquetage.

Exercice 4 : le Pattern Composite

Récupérez le code de l'énoncé auprès de votre enseignant. Ce code vise, à terme, à effectuer certaines simulations concernant la population de différents territoires.

Pour le moment, il y a deux sortes de territoires, les régions et les villes, mais il doit être possible d'en ajouter facilement. Par ailleurs, dans le code actuel, les régions ne peuvent contenir que des villes mais on veut pouvoir fusionner des régions ce qui revient à ce que la plus peuplée contienne la moins peuplée. Comme le montre la classe Population fournie et les tests unitaires qui vont avec. Le code de ces 2 classes fournies est au départ en commentaire. Vous le dé commenterez une fois votre travail terminé.

Modifiez le code pour qu'il soit évolutif et que ça marche !

Faites le diagramme UML final des classes et de leur paquetage.

Série 1 : Patterns structurels

Exercice 1 : Calcul de coût de livraison de commandes : le Pattern Bridge

Nous nous intéressons dans cet exercice au calcul de coûts de livraison de commandes pour un e-commerçant.

Le coût de livraison d'une commande représente 10% de son montant. Cette tarification peut évoluer dans le futur qui pourrait par exemple prendre en compte le poids des commandes.

Ce qu'il y a à faire:

1. Récupérer le code de départ de cette application et le lire.
2. Refaire la conception du calcul du coût de livraison avec le pattern Bridge.
3. Ajoutez un mode de calcul du coût de livraison basé sur le poids (< 2kg 2 euros sinon 9 euros)
4. Faire le nouveau diagramme de classes.
5. Ecrire le code Java des classes.
6. Ecrire des tests de validation pour les 2 modes de calcul de cout.

Exercice 2 : Emballage de commandes : le Pattern Decorator

On gère les commandes d'un e-commerçant représentées par le diagramme de classes ci-contre.

On voudrait proposer aux clients la possibilité d'emballer ou pas leurs commandes avec du papier d'emballage moyennant un surcoût de la commande.

Il existe deux qualités de papier d'emballage : du papier ordinaire et du papier cadeau facturé 2 et 6 euro, respectivement, en plus du montant de la commande.

Travail demandé :

1. Proposer une solution pour l'emballage de commandes à l'aide du pattern Decorator.
2. Coder la solution en Java.
3. Valider le code en testant les cas suivants :
    a. Une commande ordinaire sans emballage.
    b. Une commande urgente sans emballage.
    c. Une commande ordinaire emballée avec du papier ordinaire.
    d. Une commande urgente emballée avec du papier ordinaire.
    e. Une commande ordinaire emballée avec du papier cadeau.
    f. Une commande urgente emballée avec du papier cadeau.
    g. Une commande ordinaire emballée deux fois avec du papier ordinaire.
    h. Une commande ordinaire double emballée avec du papier ordinaire puis avec du papier cadeau !

Série 2 : Patterns comportementaux

Exercice 1 : Calcul du TTC de commandes (cas 1) : le Pattern Template Method

Nous développons une application pour le calcul du montant TTC de deux types commandes : des commandes françaises et des commandes à l'international. Indépendamment des types de commandes, le TTC s'obtient en additionnant le montant HT, la TVA et les frais de livraison. Cette formule peut être amenée à évoluer plus tard.

Le taux de TVA des commandes françaises est actuellement de 20% et celui des commandes à l'international est de 10% (pour encourager vente à l'exportation). Par ailleurs, les frais de livraisons sont fixés pour chaque commande française alors que ceux des commandes à l'international sont définis par pays.

On précise que les types de commandes devraient être définies à court terme et que la formule de calcul du TTC évolue régulièrement mais reste globalement la même pour tout type de commandes.

Travail demandé :

1. Améliorer la maintenabilité du code à l'aide du design pattern Template Method.
2. Ici la "template method" c'est le calcul du prix TTC !
3. Vérifier la non régression fonctionnelle du code à l'aide des tests fournis.

Exercice 2 (<u>Optionnel</u>) : Calcul du TTC de commandes (cas 2) : le Pattern Template Method

Le code Java de cet exercice est proposé pour le calcul du montant TTC de commandes. Deux types de commandes sont considérés, des commandes Françaises (destination en France) et des commandes à l'international (destination hors de France). Les commandes Françaises peuvent contenir plusieurs produits mais toutes du même secteur d'activité tandis que les commandes à l'international sont limitées à un seul produit.

Indépendamment du type de commande, le montant TTC est calculé en additionnant le montant HT et le montant de la TVA. Le taux de la TVA des commandes françaises est défini par secteur d'activité et celui des commandes à l'international est fixe. Une réduction est appliquée sur le montant HT des commandes françaises de clients fidèles et sur les commandes à l'international d'un montant supérieur à un certain seuil minimum.

On précise que les types de commandes devraient être définies à court terme et que la formule de calcul du TTC évolue régulièrement mais reste globalement la même pour tout type de commandes.

Travail demandé :

1. Ajouter quelques tests supplémentaires dans la classe de tests
2. Améliorer la maintenabilité du code à l'aide du design pattern Template Method.
3. Ici la "template method" c'est de nouveau le calcul du prix TTC mais cette fois on peut avoir le droit à des réductions…
4. Assurez de la non régression du code à l'aide des tests fournis.

Exercice 3 : Gestion de commandes : le Pattern State

On gère les commandes de clients, de l'enregistrement à la livraison. Une commande n'est pas systématiquement payée à son enregistrement. Cependant, elle ne peut être livrée qu'après paiement. Les commandes ne sont plus remboursables après livraison.

Ces règles de gestion peuvent évoluer dans le futur (par exemple, la livraison peut devenir possible même sans paiement et le remboursement peut devenir possible jusqu'à deux semaines après livraison).

Pour la date de livraison on mettra simplement la date courante (new Date()) pour une commande livrée.

Travail demandé :

1. Proposer un diagramme de classes à base du pattern « State » améliorant la maintenabilité de cette application.
2. Le coder en Java. Pour cela écrire une classe abstraite **EtatCde** qui implémentera le comportement par défaut des méthodes qui font changer l'état d'une commande ou qui sont utilisées dans les tests. Les voici :
   - payer() : on plante (throw) par défaut
   - livrer() : on plante (throw) par défaut
   - rembourser() : on plante (throw) par défaut
   - estPayee() : on retourne false par défaut
   - getDateLivraison() : on retourne null par défaut.

Les classes états dériveront de la classe EtatCde et du coup les méthodes précédentes seront surchargées ou pas !
On évitera absolument un setter pour modifier l'état d'une commande. C'est à la méthode de retourner le nouvel état d'une commande quand il y a lieu.
3. Valider votre solution à l'aide des tests fournis que vous devez réadapter.

Série 3 : Patterns de Création

Exercice 1 : Gestion des services d'une entreprise : le pattern Factory

L'application demandée dans cet exercice permet seulement de gérer les services d'une entreprise.
L'entreprise est structurée en deux types de service : service administratif et service technique.
L'application doit permettre l'ajout et la suppression de services pendant l'exécution. Par ailleurs, d'autres types de service peuvent être considérés dans de futures évolutions.

Travail demandé :
Reconcevoir la fonction d'ajout de nouveaux services en utilisant le Pattern Factory:

a. Lire (et compléter si nécessaire) les tests de validation d'ajout de services.
b. Modéliser ce pattern sur le diagramme de classes de l'application.
c. Remanier le code de l'application à l'aide du Pattern Factory. La factory **ServicesFactory** doit être injectée via le constructeur de la classe Entreprise.
d. On fera attention à minimiser les dépendances et donc à ajouter toute interface utile !
e. Quel est le coût d'ajout d'un nouveau type de service ? Ajouter un service de type loisir !

Exercice 2 : Embauche de personnel : le Pattern Factory Method

L'application de cet exercice permet d'enregistrer l'embauche de personnels.
L'entreprise est organisée en deux types de services : des services administratifs et des services techniques. Actuellement, un seul service administratif (le service RH) et deux services techniques (Informatique, réseaux et télécom) composent cette entreprise. L'ajout et la suppression de services sont hors de nos préoccupations dans cet exercice.
L'entreprise emploie actuellement deux types de personnel : des personnels administratifs affectés aux services administratifs et des personnels techniques affectés aux services techniques de l'entreprise. Tous les personnels sont décrits leurs matricule, nom et salaire d'embauche.
Le congé des administratifs se prend par mois entier. La paye de fin de moins est majorée de 10% pour ceux ayant travaillé le mois.
La paye de fin de mois des techniciens est calculée en fonction de leur salaire d'embauche et du nombre d'heure travaillées.
Il n'est cependant pas impossible de créer dans le futur de nouveaux types de services (probablement un service Commercial) avec des personnels spécifiques (commerciaux).

Travail demandé :
1. Etudiez d'abord consciencieusement le code fourni !

2. Bien réfléchir afin de pouvoir remanier le code de la fonction d'embauche à l'aide du pattern « Factory Method ».
3. Vérifier que les tests continuent de passer (après de légères adaptations).
4. Vérifier que l'application ne présente plus de dépendances de classes instables. Pensez aux interfaces si nécessaire !
5. L'entreprise crée un service commercial pour promouvoir ses produits. Ce service emploie des commerciaux. Le personnel commercial suit exactement les mêmes règles que les administratifs.
    a. Coder ce changement. C'est vite fait normalement !
    b. Compléter les tests réalisés plus haut pour valider cette évolution.

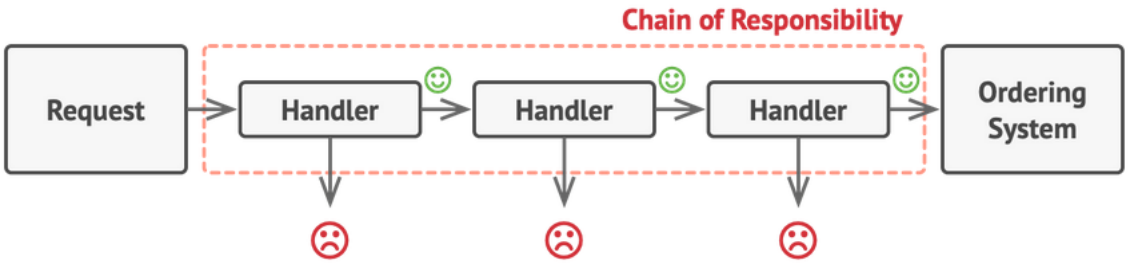
Série 4 : Patterns comportementaux (le retour)

Exercice 1 : GPS : le Pattern Observator

Afin d'illustrer l'implémentation du pattern Observator réalisons une petite application permettant de se positionner grâce au GPS.
Considérons que notre application est reliée virtuellement à un récepteur GPS. On va concevoir une classe nommée Gps qui va stocker les informations du récepteur (positionnement, précision…). Puis 3 autres classes (AfficheResumeDD, AfficheComplet et AfficheResumeDMS) permettant d'afficher de 3 façons différentes ces informations.
Comme dans la définition du pattern Observator, on trouve également deux interfaces Observateur et Observable. Pour résumer la classe Gps sera l'observable et les classes AfficheResumeDD, AfficheComplet et AfficheResumeDMS seront ses observateurs.
La classe Gps implémente l'interface IObservable et peut ainsi avoir des observateurs à son écoute. La liste des **observateurs** à l'écoute est stockée dans un **tableau dynamique** en attribut.
De plus, cette classe possède deux champs un correspondant à la **position** (latitude, longitude) définie comme un tableau de 2 **double** [lat,lon] et l'autre à la **précision** (**entier** de 1 à 10). La précision indiquera la fiabilité du positionnement (plus on utilise de satellites plus la précision est forte). On pourra modifier ces mesures à tout moment via la méthode setMesures() de la classe GPS.
La classe **AfficheResumeDD** n'affiche que la position (au format DD), la classe **AfficheComplet** affiche en plus la précision et enfin la classe **AfficheResumeDMS** affiche seulement la position au format DMS. Rappel : ces 3 classes sont les observateurs…
Une partie du code vous est fournie. Notamment un main() incomplet à compléter. N'oubliez pas les paquetages adéquats.
Faites le diagramme de classes UML résultant de votre travail.

Exercice 2 : Le roi des Orcs : le Pattern Chain of Responsability

Le roi des Orcs dispose d'une chaine de commandement assez simple. Il s'adresse à son commandant qui lui-même s'adresse à son officier et qui lui peut s'adresser à un soldat.



Le roi fait des requêtes ! Il y en a trois : défendre le château (DEFEND_CASTLE, réalisée par un Commandant), torturer un prisonnier (TORTURE_PRISONER, réalisée par un

Officier) et collecter la taxe (COLLECT_TAX, réalisée par un Soldat). Plus une qui ne sera prise en charge par personne FAIRE_A_MANGER !

Une classe abstraite **RequestHandler** devrait gérer le chainage des responsables via un champ **next** de type RequestHandler. Seul le soldat n'a personne après lui donc un champs **next** à null. Cette classe est héritée par nos 3 handlers le Commandant l'Officier et le Soldat !

Une classe **Request** devrait disposer de 3 champs **requestType** (type de requête) **requestDescription** (la chaine décrivant la requête) et un booléen **handled** (requête prise en charge oui ou non).

C'est le roi des Orcs qui impose le chainage de responsabilité. Il donne un ordre à son commandant qui transmet éventuellement à son officier qui lui, renvoie si nécessaire à un soldat. Et enfin, un soldat n'a personne à qui renvoyer la balle (null).

Pensez aux paquetages. Proposez une répartition des classes.

Série 5 : Additif

Exercice 1 : Distributeur d'argent : le Pattern State

Vous venez d'être embauché par la mairie de Saint-Denis pour mettre en place une machine à la pointe de la technologie : un distributeur au-to-ma-tique de billets !

1. On souhaite donc créer un distributeur de billets. Ce dernier peut être vu comme une machine d'états avec trois états : pas de carte insérée (EtatPasDeCarte), en attente d'opération (EtatAttenteOperation), et en attente de retirer des espèces (EtatAttenteRetrait). On pourra faire quatre actions sur la machine : insérer une carte, entrer un code, retirer des espèces, retirer la carte. Évidemment, selon l'état de la machine, les actions auront différentes conséquences.
2. La classe Carte est une classe avec un constructeur qui prend un code secret, et une méthode qui répond si le code entré est bon.
3. Si le Client entre trois fois un mauvais code, sa carte doit être avalée. Codez votre Distributeur.
4. La machine semble bien marcher, mais certains clients se plaignent. En effet, certaines fois, la machine ne donne pas d'argent... Rapidement, les ingénieurs de la machine comprennent que lorsque la machine n'a plus de billets, et ne peut pas donner d'argent. On doit donc rajouter un état à la machine, qui sera l'état vide (EtatVide). Cependant, vous voyez bien que rajouter un état change beaucoup de code et n'est donc pas très pratique. Testez avec un Client que tout fonctionne.
5. Ajoutez un état EtatMachineVide pour empêcher des retirer de l'argent quand la machine est vide. Vous allez donc ajoutez au Distributeur une variable de classe permettant de connaître son stock d'argent, et vous vérifierez, lorsque vous donnerez de l'argent à un Client, qu'il y a assez d'espèces dans le Distributeur. Était-ce compliqué ? Normalement non !!!
6. Ajoutez aussi une méthode lire_solde_compte(), qui permettra au client d'accéder à son solde lorsque le Distributeur est en mode En_Attente_Pour_Retirer_Argent. Était-ce compliqué ? Normalement non !!!

Remarque : On part sur une solution sans utiliser le mécanisme des exceptions. On affichera par contre un message adéquat selon les nombreux cas de figure rencontrés.

Exercice 2 : Vente de desserts : le Pattern Decorator

Nous allons concevoir une application permettant de gérer la vente de desserts. Celle-ci doit permettre d'afficher dans la console le nom complet du dessert choisi et son prix. Les clients ont le choix entre deux desserts : crêpe ou gaufre. Sur chaque dessert ils peuvent ajouter un nombre quelconque d'ingrédients afin de faire leurs propres assortiments. Pour commencer notre exemple, nous choisirons uniquement deux ingrédients (le chocolat, la chantilly) mais il faut garder à l'esprit que l'ajout de nouveaux ingrédients doit être simplifié en vue d'une évolution. Le système de tarification est simple. Une crêpe (nature) coûte 1.50€ et une gaufre 1.80€. L'ajout de chocolat est facturé 0.40€, la chantilly 0.60€.

Pour rappel on peut "décorer" autant de fois qu'on veut son dessert et dans un ordre quelconque !
Un dessert a un libellé (String) et un prix (double). Ce libellé sera "mis à jour" en fonction des décorations survenues.
Les desserts évoluent ! Ajoutez un ingrédient de décoration le sucre à 0,10€ la portion. Etait-ce difficile, long ? Non évidemment !

Exercice 3 : Journal des opérations bancaires: le Pattern Singleton

Afin de mettre en pratique le pattern Singleton, prenons un court exemple d'implémentation dans le milieu bancaire. Tout d'abord nous allons concevoir une classe CompteBancaire qui permet de déposer ou de retirer de l'argent sur un compte.
Mais nous souhaiterions pouvoir afficher les opérations (effectuées ou refusées) dans la console en cas de litige.
Voici les 5 cas où un message de log doit être ajouté au journal.

- Dépôt positif
- Dépôt négatif ou nul (erreur de saisie)
- Retrait négatif ou nul (erreur de saisie)
- Retrait positif (avec solde suffisant)
- Retrait positif (avec solde insuffisant)

Cette application devra rapidement évoluer et il est fort probable que par la suite d'autres classes soient concernées par cette journalisation. Pour cela, vous devez implémenter une classe distincte nommée Journalisation reprenant le pattern Singleton. Ainsi nous allons garantir que notre programme va utiliser une seule et même instance de la classe Journalisation et donc un unique journal.
Enfin, vous écrirez une classe Main permettant de tester quelques cas de figure et d'afficher le journal à chaque étape.