Compte rendu des 7 séances de POO

Ismail Alouani

Novembre 2022

1 POO: TD0

1.1 Propagation des modifications:

On a autobus.c et ps_standard.c, si on parle de fonctions internes qui gère l'intéraction entre passager et autobus, c'est autobus.c qui se charge de modifier leurs caractères par les fonctions de type __ab_demander_montee_assise qui va faire monter le *p si dans *a, il existe une place assise disponible.
Le remplacement des attributs assis et debout de la partie ps_standard entraînerait l'invalidité du code présent dans la partie autobus, car celui-ci contient des fonctions qui modifient explicitement ces deux attributs.
Exemple de la fonction __ab__montee_demander_debout:
*p.assis=false;
*p.debout=true;
Ainsi ces 2 parties ne sont pas indépendantes.
La solution pour rendre les 2 code indépendants:
Une solution serait d'implémenter une fonction dans ps_standard.c permettant de modifier l'état du passager. Ce faisant, les fonctions internes de autobus pourront y faire appel au lieu de modifier directement la structure. Ainsi, elles ne dépendent plus de la forme de la structure ps_standard.
On a appliqué le principe d'encapsulation menant à l'abstraction. En fait, la donnée du passager reste interne au code ps_standard, même si on l'utilise explicitement dans les fonctions internes de autobus, on ne fait que l'appel, le contenu on l'ignore, c'est de l'encapsulation de données. Avec les fonctions la modification se fait à l'intèrieur de ps_standard.c et cela permet de modifier le caractère du passager soit abstraire la donnée qui se présume ici à des méthodes rattachées à elle-même.
Ces deux structures et ces fonctions ne sont pas déclarées dans les fichiers en-tête car ceux-ci sont directement utilisés par le code client. Cela permet de masquer le contenu des structures.
Ce fichier a pour but de déclarer les structures tout en les gardant privé du code client qui ne fait appel qu'aux fichiers autobus.h et ps_standard.h.(__internes.h)
Il permet de faire le lien avec les fonctions de ps_standard qui utilise autobus et invérsement soit de se présumer à du code interne qui est pas utilisable directement par le client soit simple.c. Cela correspond au concept de séparer la réalisation du code client qui utilise des fonctions qui font appel à la réalisation et AUSSI faire varier les utilisations sans impacter le code client(de même).
Cette méthode n'est qu'un intention car les prototypes de fonctions déclarées dans un fichier d'en-tête ne sont pas contraignants, et peuvent toujours être modifiés sans obtenir plus qu'un avertissement lors de la compilation. Ce n'est qu'une intention car on pourrait de même programmer pas de cette façon suivant le paradigme de C et obtenir le même résultat mais pour l'évolution du code en cas de changement de passager cela va impliquer bcp de modifications à faire méthode précedemment décrite.

1.2 Extension: ajout d'un nouveau caractère au passager:

Pour ajouter ce caractère, il faudrait modifier les fonctions ps__monter_dans ainsi que la fonction __ps__nouvel_arret. La première est une fonction du client qui indique la montée sans préciser de quel caractère s'agit-il et la deuxième le caractère à la descente appelé par allerArretSuivant qui est une fonction client d'autobus qui appelle cette fonction. Tout cela est logique.
Etant donné que le code de la partie autobus fait appel à ces deux fonctions on ne peut pas toucher au prototype de ces fonctions. Par conséquent, une façon possible d'ajouter ce nouveau comportement serait d'ajouter un attribut caractère à la structure ps_standard permettant ainsi d'adapter les fonctions ps_monter_dans et __ps_nouvel_arret en fonction du caractère du passager.
On peut par exemple créer des fonctions variantes des ses fonctions pour chaque caractère (ps__monter_dans_std, ps__monter_dans_indécis,
__ps__nouvel_arrêt_std,__ps__nouvel_arrêt_indécis).

Il suffit ensuite de modifier le corps de ps__monter_dans et __ps__nouvel_arrêt de façon à ce qu'elles appellent une des variantes en fonctions du caractères du passager passés en paramètre.
Pour l'initialisation on va ajouter un argument dans la fonction qui alloue une place pour le passager et ajuster son comportement précédent en fonction du caractère.
Dans le cours, cela faut appel au 2ième concept: modifier la réalisation sans impacter le code client(simple.c dans ../(./)).
Un avantage de cette solution est qu'elle est assez facilement implémentable en plus d'être simple à comprendre. En contrepartie, cette solution implique de créer deux nouvelles fonctions à chaque fois qu'on souhaite ajouter un nouveau caractère ce qui peut être assez contraignant.

1.3 Le decompte des places dans autobus:

Dans le code d'autobus, la réalisation du décompte des places assises et des places debout utilise la technique du copier/coller/modifier. Donner une solution de réutilisation qui évite le copier/coller de cette réalisation.

Pour éviter le code "copier/coller" écrit dans les fonctions internes de l'autobus (avec pour nom : __ab__....), il est possible d'utiliser une fonction généralisant (factorisant) le code écrit dans les fonctions internes.
Concrètement, on ajoute une fonction prenant en paramètres: l'état final du passager (par un booléen par exemple), un paramètre indiquant si l'autobus est à l'arrêt ou non (encore un booléen), et bien entendu les références des structures (pointeurs en C). Ainsi en modifiant directement le code C, on obtient la modification suivante: Le code de gauche est le code d'origine. Et celui de droite le nouveau passant par une fonction généralisant l'intérieur des fonctions internes de l'autobus.
Remarque: On aurait pu aussi utiliser des énumérations pour coder le changement d'état (montée/arrêt, assis/débout) cela rendrait le code encore plus lisible.

1.4 Algorithmes sur les tableaux:

Compacter les éléments revient à décrémenter l'indice de tous les passagers situés après le passager qui descend. Cependant, le parcours du tableau est réalisé par un indice i, que l'on incrémente à chaque passager. On voit alors sur la figure 1 que la compression se fait après traitement du passager descendant, ce qui amène le passager suivant à se retrouver à la place d'indice déjà traitée. Ainsi, la descente d'un passager fait que le suivant n'est pas traité et ne change pas d'état même s'il le devrait.

Dans l'exemple de simple.c, lors de la descente de Kaylee, si l'on compresse le tableau alors Jayne prendra la place d'indice précédemment occupée par Kaylee, avant que l'indice de parcours ne soit incrémenté, amenant Jayne à ne pas descendre du bus malgré son arrêt 4.

1.5 Une dépendance cyclique:

4.5 Une dépendance cyclique Montrer qu'il y a une dépendance cyclique entre la partie autobus et la partie ps_standard.
Comme la partie 4.1 Propagation des modifications l'explique, l'application est découpée en trois parties :
- le scénario (programme principal) qui utilise le code de l'autobus et du passager standard ;
- l'autobus qui utilise le code du passager standard ;
- le passager standard qui utilise le code de l'autobus.

En effet, le passager a besoin de savoir s'il reste des places assises ou debout dans un autobus pour pouvoir monter et modifier son état. Concernant le bus, il a également

besoin de savoir si un passager monte dans l'autobus ou pas et s'il est assis ou debout. La dépendance est donc bien cyclique, ce n'est pas une partie qui utilise l'autre mais bien chaque partie qui a besoin des informations sur un attribut de l'autre pour fonctionner, voire même modifier un attribut de l'autre partie.
Rq: lien a_UN structurel et dépendance cyclique vis-à-vis de la création d'instances avec malloc() et le problème retenu.
Dans le cas de la compilation séparée, une dépendance d'utilisation cyclique est problématique si on n'utilise pas de pointeur sur les structures ou dans le passage de paramètres. Expliquer pourquoi ?
Dans le cas de la compilation séparée, une dépendance d'utilisation cyclique est problématique si on n'utilise pas de pointeur sur les structures ou dans le passage de paramètres. En effet, dans le cas où on doit modifier un attribut de la structure de l'autre partie dans une fonction, si l'on n'utilise pas de pointeur, la modification ne sera que local, dans une copie de la structure mais ne sera pas retranscrite dans la structure en elle-même. Un pointeur nous permet donc de pouvoir modifier une information dans une autre structure sans perdre la modification. (Voir la remarque)

2 POO: TD1

2.1 Le kit de développement JAVA

Quelle est la dernière version du langage Java en exploitation ? La dernière version du langage Java est la version 18 (plus précisément la 18.0.2.1).
Noter le suffixe du fichier généré par le compilateur.
Le fichier généré au moment de la compilation porte le suffixe ".class". Noter la version du compilateur utilisée (option -version).
La version du compilateur utilisée est la 1.8.0_252. 1.3 Exécuter: Quel est le comportement de cette commande si vous donnez comme argument ExecutionExemple.class ?
En donnant comme argument ExcecutionExemple.class le compilateur produit l'erreur suivante : "Erreur : impossible de trouver ou charger la classe principale ExecutionExemple.class".
Noter la version de la JVM utilisée. La version de la JVM, sur une machine de l'école, est la 1.8.0_252 et la dernière de JAVA on le rappelle est la 18.0.2.1.

2.2 Tester les objets/instances:

En quoi une affectation (ou un effet de bord) dans une assertion Java peut-être problématique ?
Une affectation dans une assertion Java peut être problématique, car la variable va être testée sur la valeur qu'elle vient de recevoir au moment de l'assertion (ce qui ne présente pas un très grand intérêt).
Le principal problème vient surtout du fait que si la compilation est faite sans l'option -ea alors la variable ne recevra pas d'affectation, rendant le code incohérent (produisant potentiellement une erreur si la variable n'est pas initialisée ailleurs).
3.2 Ecriture du premier test: Comment s'écrit l'instanciation de la classe ? L'instanciation d'une classe ClassExemple se fait à l'aide du mot clé new suivit d'un constructeur de la classe :
* new ClassExemple(arg1, arg2, ...);
Comment déclarer une variable pour contenir une telle instance ?
En Java, la déclaration d'une variable suit la forme type nomVariable. Si on reprend la classe de la question précédente et qu'on souhaite déclarer la variable inst1 visant à contenir une instance de ClassExemple, on obtient :
¿ ClassExemple inst1;
Comment s'écrit l'appel à une méthode de (un envoi de messages à) cette instance ? L'appel d'une méthode d'une instance se fait via l'opérateur "." (= point) suivis de parenthèses contenant les arguments. Par exemple, l'appel à une méthode m1 de l'instance inst1 se fait de la façon suivante :
* inst1.m1(arg1, arg2, ..)

Comment appeler la méthode d'instance testDansIntervalle() (ou la méthode d'instance testDehors()) dans le code de la méthode main() de la classe de test ?
Pour appeler la méthode d'instance testDansIntervalle() (respectivement testDehors()) dans le la méthode main() il faut instancier un objet (par exemple test) de la classe TestJauge (respectivement TestPosition()) puis appeler la méthode sur l'objet : * TestJauge test = new TestJauge();//sans arguments aussi avec un constructeur //publique (fait par le compilateur par défaut si il en existe pas)
* test.testDansIntervalle();
Et cela soit se faire à chaque test(possibles effets de bords...).

2.3 Réaliser le reste des cas de test:

En comparant la construction des classes Jauge et Position, expliquer pourquoi une instance de la classe Position est-elle un objet constant ?
Une instance de la classe Position est un objet constant car aucune de ses méthodes ne permet sa modification (d'après les descriptions des méthodes dans sa documentation). De son côté, la classe Jauge possède les méthodes incrémenter et décrémenter, modifiant l'état d'un de ses attributs d'après la documentation.

2.4 Un coup d'oeil sur la source ExecutionExemple.java:

Quel traitement effectue l'opérateur + ? Utiliser la documentation de la classe String.
Selon la documentation, l'opérateur + permet la concaténation de 2 chaînes de caractères ainsi que la conversion d'autres objets en chaîne de caractère.
Dans l'A.P.I., la classe StringBuffer représente aussi une chaîne de caractères. D'après la documentation de ces deux classes, quel est la distinction entre ces deux représentations d'une chaîne de caractères ?
La distinction entre les classes String et StringBuffer est qu'un objet instancié de type StringBuffer peut modifier sa chaîne de caractère représentée. De sorte qu'elle est mutable en revanche pour le type String en java.
Donner l'instruction qui permet d'obtenir à partir d'une chaîne constante (instance de String) une chaîne non constante (instance de StringBuffer) puis donner l'instruction inverse.
L'instruction permettant d'obtenir une instance de String à partir d'une instance de StringBuffer se fait via le constructeur String prenant pour paramètre un objet StringBuffer.
public String(StringBuffer buffer)
L'opération inverse suit la même logique et utilise le constructeur StringBuffer prenant un objet String en paramètre :
public StringBuffer(String str)
En partant de la documentation de la classe System, quelle classe définit le traitement println utilisé dans le code ?
La méthode toString() pour le premier cas fonctionne aussi.
Le traitement println est défini par la classe PrintStream. Dans cette classe, vous trouverez plusieurs définitions de l'identificateur println.
C'est le mécanisme de surcharge.
Comment ces définitions sont-elles différenciées ?
Ces définitions sont différenciées par le type de l'argument pris en entrée de la fonction println. En effet, chaque type ayant des représentations différentes, il est logique qu'il existe une définition par type (int, float, Object, char, etc.) afin d'afficher correctement la valeur de l'argument passer à println. Quelle définition va être appelée dans notre exemple ?
Notre exemple visant à afficher un message informant le status des tests, la définition utilisée est donc celle prenant en entrée un objet de type String.
Quelle conséquence a le mot-clef final dans la déclaration d'une variable ?
Si une variable est déclarée avec le mot-clé final, sa valeur est non modifiable. Dans le cas d'une référence (exemple : une référence vers un objet), le mot-clé final implique

qu'on ne peut pas modifier la référence stockée dans la variable. En revanche, le contenu de l'objet référencé reste modifiable.
En résumé:
Si nous utilisons le mot-clé final avec un objet, cela signifie que la référence ne peut pas être modifiée, mais son état (variables d'instance) peut être modifié.
Rendre une variable de référence d'objet final est un peu différent d'une variable finale. Dans ce cas, les champs d'objet peuvent être modifiés mais la référence ne peut pas être modifiée. Notez qu'il s'applique également aux collections.
Lorsqu'une collection est marquée comme finale, cela signifie que seule sa référence ne peut pas être modifiée, mais que des valeurs peuvent être ajoutées, supprimées ou modifiées.
Exemple plus instructif et avancé: une classe en final n'est pas héritable (notion d'overridding).

3 POO: TD2

3.1 part 1:

Pour ce deuxième Td de Programmation Orientée Objet, nous avons créé deux équipes :
- La première équipe composée de Ismail et Nathan s'est occupée dans un premier temps de corriger la classe Position dont le .java nous a été donné lors de cette séance. En effet, le TD précédent, nous ne disposions que du .class et nous avons effectué des tests unitaires montrant qu'il y avait un problème dans le code des fonctions, ce qui s'est vérifié lorsque nous avons pu avoir accès au code. Lorsque nous voulions créer une instance de position dehors, le programme créait une instance position debout ce qui causait une partie de l'échec des tests. L'autre partie des échecs était dû à la fonction estDebout() qui vérifiait la valeur de ASSIS et non pas de DEBOUT. Suite à la correction de ces deux erreurs, le premier groupe a refait passer les tests unitaires à notre classe corrigée. Tous les tests passant au vert, ils ont pu avancer sur la partie suivante.
- La deuxième équipe, composée de Maxime et Martial, s'est occupée en parallèle de corriger la classe Jauge qui avait également des erreurs lorsque nous passions les tests unitaires la séance précédente. La fonction incrementer() renvoyait valeur -=MAX ce qui n'est pas le comportement attendu par la fonction et que nous avons donc corrigé, de plus la fonction estVert() ne vérifiait pas si la jauge dépassait le maximum mais seulement si elle était supérieure à 0. Une fois la correction des erreurs effectuée, les tests unitaires étaient enfin validés et nous avons donc pu nous répartir la tâche pour l'écriture des classes Autobus et PassagerStandard.

3.2 part 2:

Afin d'écrire en parallèle les classes Autobus et PassagerStandard qui, comme nous l'avons montré lors des TD précédents, ont une dépendance cyclique. Nous avons utilisé une interface pour chaque classe permettant de pouvoir tester la classe à l'aide des autobus ou d'un passager factice afin de pouvoir développer en parallèle sans avoir à attendre l'autre groupe pour faire fonctionner la classe.
La première équipe s'est occupée de la classe PassagerStandard en commençant par écrire une version minimale fonctionnelle. Cette version minimale compile mais ne passe pas les tests puisqu'elle ne fait rien. Pour écrire la classe PassagerStandard, le groupe s'est basé sur le diagramme de classe afin d'avoir le squelette de la classe qu'il doit reproduire en se basant sur l'interface. Ce diagramme nous donne l'information sur les attributs de la classe et les méthodes de la classe en précisant les types des variables d'entrée de chaque méthode et également le type renvoyé en sortie, précisant également pour chaque attribut et méthodes si celui-ci est publique(+), privé(-) ou

aucun des deux( ). Le groupe a ensuite décidé de rédiger l'ensemble des fonctions pour passer les tests associés à la classe étant donné que tout se déroule dans la fonction monterDans() en suivant cette fois-ci le diagramme de séquence à la montée d'un passager. Suite à l'écriture du code, l'équipe a pu tester que les tests unitaires étaient validés sur la classe PassagerStandard avec un autobus factice.

La deuxième équipe s'est occupée de la classe Autobus, de manière similaire à la première équipe, elle a commencé par l'écriture d'une version minimale qui compile mais ne passe pas les tests en se basant sur le diagramme de classe. Pour passer les tests, cette équipe a préféré avancer petit à petit en écrivant du code afin de passer un premier test, puis lorsque celui-ci est validé, elle a écrit le code pour le deuxième test en conservant le premier test valide. Jusqu'à ce que tous les tests soient valides. On peut se demander s'il existe des traitements non testés par les tests qui nous sont fournis, après analyse des différentes classes de tests, on peut voir que tous les cas présents dans le diagramme de séquence ont été testés, cependant certaines fonctions présentes dans les différentes classes ne sont pas testées directement par les tests, notamment chercherEmplacementVide() et chercherPassager(p : Passager) qui sont des méthodes privées de la classe Autobus ou encore la méthode ToString de chaque classe. Dans la classe PassagerStandard, on trouve également la méthode nom() qui n'est pas testé actuellement.

3.3 La source de la classe String :

Pour employer l'opérateur == à la place de la méthode equals(), quelle hypothèse est faite sur les instances de String représentant les chaînes de caractères littérales ?
L'opérateur == vérifie si deux objets sont identiques, c'est-à-dire s'il s'agit en réalité du même objet. Tandis que la méthode equals() ne vérifie que l'égalité entre la sémantique des deux objets.
Par exemple, si l'on crée deux instances String avec la même chaîne de caractère, l'opérateur == renverra faux tandis que la méthode equals() renverra vrai car il ne s'agit pas du même objet bien qu'ils aient la même sémantique.
Pour employer l'opérateur == à la place de la méthode equals, il faut donc faire l'hypothèse que les instances de String représentant les chaînes de caractères littéral possèdent la même référence mémoire, c'est-à-dire qu'il s'agit du même objet, ou bien, tout simplement éviter l'instanciation, les considérant comme types avec valeur directement affectée.

Le code de la méthode equals() dans la classe String est le suivant :

```
public boolean equals(Object anObject)
if (this == anObject)
return true;

if (anObject instanceof String)
String anotherString = (String)anObject;
int n = count;
if (n == anotherString.count)
char v1[] = value;
char v2[] = anotherString.value;
int i = offset;
int j = anotherString.offset;
while (n– != 0)
if (v1[i++] != v2[j++])
return false;

return true;
```

return false;

Tout d'abord, la méthode equals() vérifie si les deux instances ne font pas référence au même objet. Dans ce cas là, les deux instances sont donc forcément égales puisqu'elles sont identiques. Cependant si elles ne sont pas identiques, il faut vérifier si elles sont égales. Pour cela, on vérifie dans un premier temps si l'objet avec lequel nous allons nous comparer est bien une instance de String (Object étant forcément une instance de String puisqu'il appelle cette méthode). Si c'est le cas, on regarde chaque caractère et on renvoie faux si les caractères sont différents, dans le cas où aucun caractère n'est différent, on renvoie vrai puisque cela signifie que les chaînes sont identiques.
Pour vérifier l'égalité, on regarde donc s'il y a une différence entre les chaînes de caractères.
Expliquer pourquoi l'instruction String anotherString = (String) anObject est-elle nécessaire ?
L'instruction String anotherString = (String) anObject est nécessaire car elle nous permet de conserver de façon privé les informations de anObject sans que l'instance puisse être modifiée pendant l'exécution de la méthode equals(), mais également de s'assurer que nous utilisons le même type.
La conversion de type échouera-t-elle à la compilation ou à l'exécution ?
Lorsqu'une conversion de type échoue, cela se déroule à l'éxécution. Soit cela provoque une erreur, dans ce cas, l'éxécution s'arrête. (Soit cela entraîne un avertissement permettant à la compilation de continuer tout en indiquant une potentielle erreur dans le code.)

4 POO: TD3

4.1 Modifications de la réalisation:

Si la classe Position ne construisait pas d'objets constants, combien faudrait-il d'instances pour représenter la position d'un passager dans la classe Passager Standard ?
Comparez avec le nombre d'instances dans le code actuel.
Supposons que la classe Position ne construit pas d'objets constants, donc il y a une ou plusieurs méthodes qui pourraient modifier l'état de l'objet (la valeur de ses variables d'instance), donc logiquement à chaque fois qu'on veut modifier l'état du passager on utilise une méthode d'instance spécifique. Donc une seule instance dans Passager Standard suffit.
Conclusion : Si Position ne construit pas d'objets constants, il faut autant d'instances de Position que d'instances de Passager Standard.
Si on compare avec le nombre d'instances dans le code précédent, on retrouve le même résultat à chaque instanciation de Passager Standard, l'attribut Position est instancié suivant sa position debout, assis ou dehors avec le constructeur adapté dans la classe Position.
Rq: Si on modifie l'état du passager, on utilise la méthode de classe publique assis par exemple pour rendre le passager assis et là, on retourne un new Position(ASSIS) donc à chaque modification, on instancie à nouveau la classe pour modifier l'état courant du passager contrairement à l'instanciation de Passager Standard initial où il faudrait choisir parmi les 3 soit une instanciation de Position.

Expliquer vos modifications ?
A quel moment la classe Position est-elle instanciée ?
(PARTAGE d'INSTANCES)

Pour permettre le partage d'instances, il suffit de créer trois instances correspondantes aux positions debout, assis et dehors dans la classe Position. Ils seront déclarés dès le début de la classe avec le mot clé: private final static ; c'est à dire ca serait indépendamment de l'instanciation de la classe tout entière Position. Prenons un exemple dans le fichier test de Position, on a mis dans une fonction test : Position p=Position.assis() ; c'est à quoi sert le mot clé static : à utiliser les attributs d'une

classe sans pouvoir les instancier avant. C'est un attribut de Position donc il doit être déclaré en private pas en public (il y a des getters pour accéder à la valeur de ce dernier depuis un fichier test par exemple de l'extérieur). Ainsi reste le mot final qui veut signifier le fait que cet élément ne peut être changé à la suite du programme, si on veut modifier l'état du passager (on parlera après de cela) on va appliquer: return Position.Dehors pour faire sortir le passager par exemple, mais aucune méthode de Position ne pourrait changer l'attribut de Position assis=new Position(ASSIS) qui sera instancié à l'intérieur de la classe Position elle-même.
Ainsi, nos modifications consistent à construire trois attributs assis, dehors et debout dans la classe Position en instanciant les trois avec le constructeur qui prend en paramètre DEHORS, ASSIS ou DEBOUT.

On va supprimer le constructeur public qui ne sert plus rien à présent, car dans Passager Standard, si on veut une position, on y accède directement avec le mot clé static et en plus les 3 instanciations (attributs) faites avant sont construites avec le constructeur privé : private Position(courant e). Les trois attributs DEHORS, ASSIS et DEBOUT sont déclarés en static aussi car les instanciations static ont besoin d'arguments static aussi pour que cela marche. Mais l'attribut int courant, on le laisse tel quel.

On ajoute 4 autres modifications, les vérificateurs de l'état du passager. Là, au lieu de faire un test booléen du genre : courant==DEHORS par exemple, on testera directement l'objet instancié avec les 3 instances possibles, cela donnerait par exemple pour l'état assis : this==Position.Assis.

Reste les 3 fonctions qui retournent une nouvelle instance debout, assis ou dehors. Ici, au lieu de retourner une nouvelle instanciation avec le mot clé new (ces fonctions servent avant à modifier l'état du passager au niveau de l'autobus à chaque arrêt) on retourne tout simplement Position.Dehors pour dehors() par exemple. Les 3 fonctions seront déclarées en static pour la raison expliquée avant. Pour la méthode String, rien à signaler. Pour être plus clair, les vérificateurs dont la modification a été signalée dans les paragraphes précédents n'ont pas besoin d'un mot clé static car ils n'agissent pas sur les 3 instances déclarées en static alors que les nouvelles fonctions retournent une instanciation qui est censée être static et utilisée ultérieurement donc ça doit être déclaré en static par contre une simple comparaison au sein de la classe n'a pas besoin de l'être.

Ensuite, on passe les tests de testPosition après avoir fait les modifications nécessaires dans ce fichier et tout est OK jusqu'à là.
Passons au fichier Passager Standard, comme expliqué précédemment, le constructeur va faire : this.position=Position.Dehors sans instancier la classe Position pour y faire un attribut objet car c'est static. Les vérificateurs ne changent pas, ils utilisent les mêmes vérificateurs de Position avec maPosition.estInterieur() par exemple pour tester si le passager est à l'intérieur du bus ou non que les modificateurs de Position qu'on a expliqués avant. Reste les changements changerEnAssis et de même pour changer l'état en dehors ou en debout. Il faudrait changer la syntaxe en : maPosition=Position.assis() et non pas maPosition=maPosition.assis() car on ne crée pas d'instances à chaque modification d'état comme on a dit au début, on partage les instances, soit 3 instances suffisent pour implémenter Passager Standard et l'utiliser après.

Avec le fichier faussaire Faux Véhicule.java, le Passager Standard.java, les 2 classes de départ : la jauge et la position ou bien que Position pour l'instant (test faussaire) ; les tests passent en vert. Tout est ok pour la classe Passager Standard.

C'est écrit dans le sujet que la classe construit des objets constants et que ce partage doit être mis en œuvre par la classe, d'où le mot final dans les 3 instances de départ qui définissent les 3 états possibles d'un passager standard.

Supposons qu'on implémente une classe sans constructeur, le compilateur javac va

mettre un constructeur par défaut ou bien remplir un tableau dont la taille est déjà connue par des null s'il n'est pas rempli, ce qui veut dire que le compilateur a besoin d'assez d'informations sur les attributs de la classe avant même de produire le .class après. Donc, en s'appuyant sur cet argument, c'est au moment même de la compilation, au moment de produire les fichiers .class correspondants que le compilateur instancie ces 3 classes pour voir à quels attributs il en à faire.
Il faudrait un peu plus optimiser le code de Position.java. Pour cela, les 3 premiers attributs DEHORS, ASSIS, DEBOUT qui servent à l'instanciation de nos trois instances de Position pour permettre le partage vont être mis en commentaire, reste à régler le problème des paramètres à l'instanciation. Pour cela, on supprimera les arguments à l'instanciation et on mettra en commentaire le constructeur privé. Ainsi l'instance à l'intérieur de la classe correspondante à l'état ASSIS va s'écrire :
private final static Position ASSIS=new Position();

Avec le mot ASSIS, on saura l'état du passager sans avoir à ajouter des attributs et un constructeur alors que le nom de la variable signifie l'état en question. Pour les autres fonctions, commençant par les getters estDebout par exemple ça va retourner :
this==Position.Debout
et les fonctions qui modifient l'état indirectement en retournant la bonne instance et qui sont bien sûr déclarées en static, ils ne prennent aucun argument et retourne pour assis: Position.Assis.

Cela a été déjà fait partiellement, on a plus besoin de courant comme attribut que des 3 instances ; même le constructeur vu le nom de la variable peut ne rien faire, car l'instanciation appelle le constructeur sans arguments. On peut le retirer mais c'est mieux qu'il persiste dans le code car le compilateur va introduire un constructeur par défaut qui ne fait rien dans le cas contraire.
RQ: dans la méthode String du fichier Passager Standard.java, on retourne "test" tout simplement pour simplifier le code car le but est de tester Passager Standard le vrai code en s'appuyant sur Position avec un fichier faussaire: fauxautobus.java qui a été mis en place pour tester le code qui dépend d'un autre sans que l'autre tandem ait fini sa partie.

4.2 Boutez vos neurones:

Le 1er exemple retourne true pour les deux. Il n'y a pas d'instanciation en utilisant la classe String. En revanche pour le deuxième, ça doit retourner false false car on instancie les classes ici, les emplacements mémoire sont différents, un equals(anObject Object) à la place d'un double égal retournerait true dans le 1er cas du 2ème cas de figure et même pour le deuxième vue l'égalité de la sémantique. On précise dans ce dernier cas que "Kalki" est une instance implicite.

Retournons à notre question. La classe String du kit JDK où les méthodes sont en static dispose d'une méthode particulière appelée : intern(). Cette méthode permet de déclarer un String avec par exemple String s=s1.intern(); sachant que s1 est instancié et de retourner vrai même s'il y eu instanciation. Cela permet de lever le problème de non-partage des instances. Par exemple, explorons le code suivant :
1. public class InternExample
2. public static void main(String args[])
3. String s1=new String("hello");
4. String s2="hello";
5. String s3=s1.intern();
//maintenant, ca sera la même chose que s1
6. System.out.println(s1==s2);//faux car les références de variables pointent vers différentes instances
7. System.out.println(s2==s3);//vrai car les références de variables pointent vers les

mêmes instances
La méthode intern tirée de la documentation de la classe String de JAVA permet d'instancier sans pouvoir que les références des variables instanciées pointent vers différentes instances et avoir une égalité au niveau des instanciations après le test avec les double égal(==). Après, on a réorganisé notre dépôt en créant 3 répertoires : le src qui contient les vraies code sources comme Position/Jauge/Autobus/Passager-Standard.. et le tst qui contient les fichiers qui servent aux tests faussaires ainsi que les tests du début testPosition/testJauge et enfin le fichier build qui contiendrait tous les fichiers .class issus de la compilation des .java et un .gitkeep pour que même si le répertoire soit vide, il soit retrouvé en important le repository git dans une machine.(On verra la commande par la suite qui permet de faire ceci)

4.3 Arborescence de fichiers:

Expliquer l'option -d.
On compile les fichiers sources de cette manière: javac -d build src/*.java . L'option -d permet de spécifier où mettre les fichiers compilés (.class de src/*.java) après compilation. Ça vient du mot anglais directory qui signifie folder ou répertoire de travail. Ça permet aussi de créer un répertoire correspondant au paquet ici tec dans le répertoire build sachant que tous les fichiers dans src et tst commencent par package tec;.(en particulier)
Expliquer l'option -cp.
Pour compiler les fichiers tests avec javac avant compilation, il faudrait gérer les dépendances. En gros, il faudrait indiquer où trouver les fichiers dont un autre fichier en a besoin. C'est le but de l'option de compilation -cp. Avec javac -d build -cp build tst/*.java, on spécifie au compilateur que les fichiers tests après compilation les mettre dans le répertoire build et que n'importe quel fichier dans tst qui dépend d'une autre classe va se retrouver dans le répertoire build. D'où l'utilité de l'option -cp à la compilation sur le terminal. (Cela permet aussi de prendre en compte le package tec et d'en créer un répertoire build/tec contenant les fichiers .class principalement de test compilés.)
En résumé, -d permet de savoir où placer les .class et d'en créer le packetage par un placement sous-répertoir ; l'option -cp permet de savoir le class path soit le chemin à emprunter pour chercher la classe le packetage ça fonctionne récursivement en profondeur.Aussi, cela permet une éxécution depuis un répertoire parent mais là via tec.* définissant le nom de la classe en question le -cp s'empare du répertoire et cherche le package dans ce répertoire pour faire les dépendances et éxécuter(pas d'éxécution en profondeur ici). C'est équivalent à indiquer où le package des classes de dépendances se trouvent et son nom pour pas chercher en profondeur via la ligne d'éxécution.
Donner la commande qui lance l'exécution de la classe TestJauge à partir de la racine de l'arborescence. La commande comme expliquée avant est la suivante:(-cp fait référence à CLASSPATH le chemin vers les classes dont dépendent les fichiers à compiler sur place) java -ea -cp build/ TestJauge TestJauge dépend de Jauge.class qui a été compilé et mis dans build avec : javac -d build src/*.java et javac -d build/ -cp build/ tst/*.java (-cp important ici car dépendances vis-a-vis des fichiers sources soit les .class de src compilés ; cela est parfaitement équivalent à : make source et make test pour avoir les .class correspondants) . Cela sera lancé depuis la racine de l'arborescence avec le fichier dans le répertoire test.

Puis on lance l'exécution avec java -ea pour activer les assertions. (Cela se fait à l'exécution pas à la compilation)
On a produit un Makefile adapté qui en plus de compiler dispose d'un nombre de target qui appelle les lignes de compilations puis exécute la cible en question. Tous les tests passent en vert, tout est OK.
Quelle instruction faut-il ajouter au début de chaque fichier source ?
Pour que toutes les classes appartiennent au même paquetage, pour des raisons d'utilisation du paquetage et de ses classes particulières après éventuellement, il faut

ajouter la commande suivante au début de chaque fichier avant le public class ou bien class: package tec; Le package après le lancement du Makefile s'appellera tec, et une classe x sera appelée par tec.x ou bien import tec.* pour utiliser la classe x sans préciser tec.x mais seulement x. (Voir les TDs suivants)

Comment déclarer une classe avec une portée interne au paquetage ?
Rappel:
Les modificateurs de visibilité d'une classe par rapport à une autre ou un package (possible) sont indiqués devant l'en-tête d'une classe (ou d'une méthode ou devant le type d'un attribut dans un autre contexte de portée de données). Lorsqu'il n'y a pas de modificateur, on dit que la visibilité est la visibilité par défaut. Une classe ne peut que :

- avoir la visibilité par défaut (sans aucun mot clé): elle n'est visible alors que de son propre paquetage.
- se voir attribuer le modificateur public : elle est alors visible de partout.

Pour déclarer une classe avec une portée interne au paquetage, il faut pas de mot clé avant le class... au début du fichier (visibilité par défaut) ; une déclaration avec public rendrait la classe accessible par tous les fichiers à l'extérieur (même pas appartenant au paquet du tout et au répertoire par des systèmes de dépendance au niveau de la ligne de compilation). C'est comme les champs d'une classe.

Si un champ d'une classe A :

- est private, il est accessible uniquement depuis sa propre classe ;
- a la visibilité paquetage (par défaut), il est accessible de partout dans le paquetage de A mais de nulle part ailleurs ;
- est protected, il est accessible de partout dans le paquetage de A et, si A est publique, grosso modo dans les classes héritant de A dans d'autres paquetages ;
- est public, il est accessible de partout dans le paquetage de A et, si A est publique, de partout ailleurs.

Résumé: pour rendre une classe interne au paquet, il faut pas utiliser de mot clé comme public,etc.. juste déclarer la classe normalement (class Position... dans tec par exemple pour la rendre interne à tec). La classe Autobus par exemple doit être déclarée comme accessible depuis l'extérieure du paquet soit un public class Autobus au début de Autobus.java. (comme indiqué dans la feuille de td3.).

Quelle contrainte y-a-t-il sur l'organisation des fichiers des classes compilées ?
Le principal problème qui se retrouve en réorganisant les fichiers classes compilées est de pouvoir gérer les dépendances et lancer les tests pour chaque classe ou bien les tests de recette tout à la fin en ajoutant les bonnes options à la compilation et en précisant où les classes (fichiers compilés) doivent se retrouver et en prendre compte avec l'option -cp pour exécuter un test sinon cela ne marcherait pas. Il faudrait compiler dans les bons répertoires et préciser le package tec dans chaque fichier pour éviter certaines erreurs de compilation. C'est pour cela qu'un Makefile sera opérationnel, dynamique et utile dans ce cas. Quand on a une classe publique compilée on pourrait l'utiliser ailleurs pour d'autres manipulations et donc le package auxquelle elle appartient perd son vrai sens en quoi tous ses fichiers construisent un projet unique, ça pourrait être une problématique, de même disposer des .class seulement comme dans le td1 ne permet pas une exploitation approfondie du code et du débogage donc une réutilisabilité très limitée de la part d'un client externe.

Après un make source puis un make test on compile les tests à l'aide des .class des sources compilées dès le début. Le nom complet d'une des classes de test est par exemple: build/tec/TestPosition.class depuis le répertoire principal. OU: build/tec/TestPassagerStandard depuis le répertoire principal. etc.

4.4 Le packetage tec:

1. Donner la commande pour compiler les fichiers du répertoire src et vérifier l'emplacement des fichiers compilés.
javac -d build src/*.java
ls build/tec ou bien ls build pour visualiser le tec où se trouvent les .class issus de la compilation des fichiers sources. On retrouve tous les fichiers en extension .class dans le folder ./builder/tec vu les package.tec ajoutée à chaque début de fichier source(et les options de compilation incluses).
2. Donner la commande pour compiler les fichiers du répertoire tst et vérifier l'emplacement des fichiers compilés.
javac -d build -cp build tst/*.java
ls build/tec
Et bien, on retrouve tous les fichiers tests en extension .class, les dépendances ont été bien prises en compte.

3. Donner la commande qui lance l'exécution de la classe de test à partir du répertoire du projet1.

On commence tout d'abord par compiler les sources : make source. Tous les fichiers sources sont en .class dans le répertoire build.
Après, on exécute la commande suivante:
javac -d build -cp build/ Simple.java (Simple.java est dans le répertoire principale)
Puis, vient l'exécution du test global via: java -ea -cp build/ Simple (pour déterminer le $classPATH(gestion de dépendances) value et activer les prises en compte des assertions à l'exécution (-ea)) On retrouve le résultat suivant (pris du terminal):
les positions a chaque arrêt des passagers en cours dés le début.
Rq: Cette classe de test finale est particulière, en fait elle n'appartient pas au package donc on la retrouve dans build/ mais pas dans build/tec c'est pour cela que la gestion d'indépendances intervient avec -cp (profondeur 0 pas comme les autres classes tests dans build/(TEC) ; indique même où trouver la classe comme dit précedemment).

4.5 Rassemblement et test d'intégration:

Pour avoir un bon affichage après l'éxécution des tests de recette (make source, javac -d build -cp build Simple.java puis java -ea -cp build Simple) ; il faudrait ajuster les toString d'Autobus et des deux classes PassagerStandard et Jauge. Pour cela on adopte le même format d'affichage qu'en C pour avoir le même format d'affichage sur le terminal.

1 Pour autobus, la méthode d'instance publique toString est écrite de cette façon ou plus précisément elle retourne: "[arret " + arretCourant + "] " + "assis" + jaugeAssis.toString() + " debout" + jaugeDebout.toString();

2 Pour jauge, de la même manière, le retour de la fonction est: "GO" + valeur + "GF";

3 Pour passager standard, ca sera: nom + "GO" + maPosition.toString() +"GF"; avec maPosition.toString() retournant le mot indiquant en miniscules l'état de ce passager. Rq: Tous les toString des 3 classes sont sans arguments fonctionnels.

5 POO: TD4

On a fait jusqu'à là le traitement des points suivants: 3.Intégration et rassemblement (Intégration de la partie fonctionnelle(encapsulation))
2.Autobus et PassagerStandard (les classes Autobus et PassagerStandard(encapsulation))
1.Approche Objet jauge et position (encapsulation ; transports en commun())
0.présentation existant (de même)
et maintenant:
Masquage d'informations (remaniement(substitution d'objets))
Evoquation de la notion de polymorphisme.

5.1 Séparer les méthodes avec des accès différents :

Pour ce quatrième TD de Programmation Orientée Objet, nous avons choisi deux sous-équipes/tandems:
- Le binôme Guillaume et Ismail s'est occupé dans un premier temps de la partie "passager" du remaniement. Puis ils se sont occupés de la solution dite "abstract" pour respecter les nouvelles contraintes d'utilisation du code client.
- Le tandem composé de Martial et Maxime a travaillé sur le remaniement de la partie "vehicule" du code. Ensuite ils ont implémenté la solution dite "interface" pour appliquer les contraintes sur le code client.

Dans les deux tandems, le remaniement a été plutôt facile même si j'estime que du temps a été perdu. Il s'agissait simplement d'ajouter deux nouvelles interfaces et de modifier/bouger quelques méthodes et définitions de classes.

La difficulté était de coordonner les deux binômes pour fusionner les codes (comme pour le TD 2) afin de tester l'ensemble. Car pour lancer les tests globaux/client (issus de Simple.java), il fallait attendre plusieurs fois que les deux tandems finissent la modification du code.

Il y avait interdépendance des parties de codes, c'est la principale raison de la perte de temps.

1.2 Remaniement:
On dispose actuellement des méthodes accessibles par le client soit Usager et Transport et ceux non accessibles via Passager et Vehicule (prochainement INTERNES AU PACKAGE).

Expliquer pourquoi il est nécessaire d'effectuer la conversion Vehicule b = (Vehicule) t; dans le code de la méthode monterDans(Transport t) (passagerstandard etend passager,usager et utilise le code d'autobus qui etend transport,vehicule) même si l'instance contenue dans le paramètre t possède bien les méthodes attendues. Le compilateur voit initialement l'argument t en type (INTERFACE ; ATTENTION) Transport car il est paramètre de la méthode monterDans(Transport t). (c'est une fonction client) Or durant l'écriture concrète de cette méthode (dans PassagerStandard) on fait appel à des méthodes de l'interface Vehicule (aPlaceAssise, aPlaceDebout, monteeDemanderAssis et monteeDemanderDebout). Donc il faut préciser au compilateur que l'objet passé en paramètre peut faire appel à des méthodes de Vehicule, sinon il ne reconnait que les appels de méthodes uniquement de Transport (c'est-à-dire seulement allerArretSuivant).

Cette conversion de type peut-elle échouer ?
Il est évident que cette conversion de type peut échouer.
Par exemple, si on passe en paramètre de la méthode monterDans(Transport t) un objet instancié à partir d'une classe concrète héritant de Transport mais pas de Vehicule, alors la conversion échouera.
En cas d'échec, cet échec est-il détecté à la compilation ou à l'exécution ?
Lors de la conversion, le compilateur agit de manière générique sur Transport t. Tout

ce qu'il sait, c'est que les appels de méthodes de Transport sont autorisés (en prenant en compte la portée des méthodes). En effet, le compilateur ne sait pas à l'avance de quels types possède l'objet passé en paramètre, il sait juste qu'il doit hériter de l'interface Transport. Ainsi (comme vu dans le TD 2 avec les String), en cas d'échec, la conversion échouera à l'exécution.

Il est toujours possible au client (ici la classe Simple) d'utiliser les méthodes déclarées dans les deux interfaces internes au paquetage.
Montrer de quelle manière le client peut accéder à ces méthodes. Expliquer les raisons de cet accès.
Le client (code extérieur au paquetage tec) peut accéder aux méthodes de Vehicule et Passager de manière détournée. En effet, les classes concrètes Autobus et PassagerStandard héritent respectivement des deux interfaces précédentes. De plus, ces classes ont une portée publique. Et le sujet ne demandant pas de modifier la portée des méthodes de celles-ci (mais seulement la portée des interfaces) ces méthodes sont donc aussi de portée publique. Ainsi, le client instancie des objets de type PassagerStandard et Autobus, il peut donc appeler les méthodes de Passager et Vehicule à partir de ces objets.

5.2 Les classes concrètes PassagerStandard et Autobus :

Cette partie a été intéressante pour tout le monde et les deux groupes ont rattrapé leur retard de temps pris durant la partie précédente. En effet, le fait de chercher comment implémenter les deux solutions a été un exercice intellectuel appréciable (surtout pour le coordinateur qui a dû réfléchir sur les deux solutions en parallèle).

5.2.1 Ces 2 classes concrètes sont publiques :

Pour cette solution, le binôme concerné a simplement changé une des interfaces en classe abstraite, puis modifié presque toutes les portées des méthodes dans les classes abstraites et concrètes, et enfin adapté le code avec la nouvelle structure. Bien évidemment nous avons effectué des tests au niveau du code client, permettant de vérifier si les contraintes (pour le code client) sont respectées. Malheureusement ces tests ont été écrits dans le code temporairement car ils provoquent des bugs (tests d'appels de méthodes non visibles) et il faudrait les remettre. Pour cela nous attendons l'utilisation d'une structure permettant de vérifier si une partie de code arrête le programme (les exceptions et les try-catch) dans les futurs TDs.

5.2.2 Ces 2 classes sont internes au package tec :

Le groupe concerné a rendu les classes concrètes non visibles pour le code client, et ajouté une nouvelle classe ni instanciable, ni héritable contenant des méthodes de classes permettant d'obtenir une nouvelle instance d'objet utilisable par le client. Pour les tests, même remarque que pour la solution précédente, les tests de vérification des contraintes ont été effectués mais retirés du code actuel. Nous fournissons le service d'instanciation grâce à la classe publique tec.FabriqueTec :
1. Elle contient deux méthodes de classe fairePassagerStandard() et faireAutobus().
2. Chaque méthode de cette classe effectue l'instanciation d'une des classes concrètes internes au paquetage.
Donner le prototype de ces deux méthodes.
Les prototypes des deux méthodes sont:
public static Usager FairePassagerStandard(String nom, int destination)
public static Transport FaireAutobus(int nbPlaceAssise, int nbPlaceDebout)
3. La classe tec.FabriqueTec ne doit ni être instanciée, ni servir de classe de base.
De quelle manière assurez-vous ces deux contraintes ?
Pour la contrainte d'instanciation il n'existe pas de mot clé comme "static" pour une

classe, il faut donc interdire l'accès au(x) constructeur(s). Par défaut si le constructeur n'est pas défini, le compilateur en ajoute un de portée publique, sans instructions et sans paramètres. Il faut donc forcer le constructeur à être de portée privée. Pour la contrainte d'héritage, il suffit d'ajouter le mot clé "final" à la classe, ce qui empêchera tout héritage par une autre classe.
Rq: Après l'instanciation, on peut pas accéder aux méthodes car on a pas importé les bons packages (passager et vehicule) ça c'est bon sans sourcier des portées des données encapsulées dans ces 2 types de classes.

5.2.3 Comparaison+Adaptation potentielle :

Pour les deux solutions, l'ajout de nouvelles classes concrètes héritant de Passager ou Vehicule nécessite l'implémentation des méthodes héritées (non concrètes), ce qui peut être long à coder pour chaque nouvelle implémentation de ces classes concrètes. De plus, séparer les méthodes en multipliant les classes non concrètes (interfaces/-classes abstraites) n'est pas difficile à mettre en place, mais peut rendre l'organisation moins compréhensible et les héritages plus complexes.

Pour la solution "abstract" (2.1), ajouter de nouvelles classes concrètes implique de vérifier la portée de chaque méthode et attributs de la classe (car ces classes sont instanciables partout). C'est une tâche plus ou moins complexe, mais le développeur peut choisir la portée des attributs et méthodes sans toucher les interfaces/classes abstraites.
Pour la solution "interface" (2.2), l'implémentation de nouvelles classes concrètes est un peu plus simple que pour la solution précédente, la portée peut rester publique dans les nouvelles classes concrètes (car dans tous les cas ces classes ne seront pas instanciables). Cependant, le développeur n'aura pas le choix sur la portée des fonctions et attributs lors de l'implémentation, il devra modifier les interfaces/classes abstraites pour changer cela. De plus pour chaque nouvelle classe concrète le développeur devra ajouter une méthode d'instanciation (dans FabriqueTec par exemple) dérivée pour pouvoir utiliser les objets souhaités (car non instanciables par le client/code externe).

6 POO: TD5

Substitution d'objets: ajout de nouveaux caractères : Pour ce cinquième TD de Programmation Orientée Objet, nous avons choisi deux sous-équipes/tandems:
- Le binôme Guillaume et Maxime s'est occupé de la partie "Interface" du code.
- Le binôme Ismail et Nathan a travaillé sur la partie "Abstract" du code.

6.1 Héritage de la classe concrète PassagerStandard:

Quelles méthodes sont à redéfinir dans les classes dérivées ?

Les caractères des passagers influent uniquement sur leur comportement lors de la montée et lors d'un nouvel arrêt. Par conséquent, les méthodes à modifier sont nouvelArret() et monterDans().
(en général, modifier ceux du client spécifiques au passager et après voir)

Pour le passager indécis, le comportement est différent pour ces deux fonctions, qui vont donc être modifiées, tandis que le passager stressé a le même comportement à la montée que le standard, donc seule nouvelArret() va changer.

Si la méthode nouvelArret() est redéfinie, le test pour la sortie du passager risque d'être dupliqué dans le code des classes dérivées.
Pour éviter cette duplication, comment peut-on appeler cette méthode dans la classe de base?

La fonction nouvelArret() de la classe de base vérifie uniquement si l'arrêt en cours correspond à la destination du passager. Cela va aussi être le cas quel que soit le caractère du passager.
Avant chaque action spécifique au comportement du passager, on va donc appeler la méthode nouvelArret() de la classe de base. Pour cela, on utilise le mot-clé super, qui permet l'utilisation des méthodes de la classe de base au lieu de celles de la classe actuelle : super.nouvelArret(args,....) va donc appeler la méthode de la classe PassagerStandard au lieu de celle de PassagerIndecis ou PassagerStresse.

La réalisation de la classe PassagerStresse nécessite l'information sur la destination. De quelle manière est-il recommandé de fournir l'information de destination aux classes dérivées ? Expliquer pourquoi ?
Les attributs privés ne sont pas directement accessibles aux classes dérivées. Il est donc nécessaire d'ajouter un getter dans la classe de base. Celui-ci permet d'éviter de passer les attributs en public, ce qui permettrait de les modifier depuis l'extérieur de la classe.

Expliquer le prototype du constructeur des classes dérivées et comment appeler le constructeur de la classe de base.

Le constructeur des classes dérivées ressemble à : public PassagerIndecis(String , int) Celui-ci prend les mêmes paramètres que le constructeur de la classe de base. En effet, ce dernier va justement être appelé grâce au mot-clé super, qui correspond dans notre cas au constructeur de la classe PassagerStandard et prend donc lesdits paramètres en argument(s).

6.1.1 Changement du code de la classe de base :

Après la mise en production, le client change la spécification du comportement de la classe PassagerStandard :
- à la montée, demande une place debout sinon reste dehors ;
- sort un arrêt avant sa destination.

Quelles modifications pour le code des classes dérivées de PassagerStandard ?
Avec ces modifications, les méthodes monterDans() et nouvelArret() vont être modifiées pour les deux classes :
- PassagerIndecis a maintenant le même comportement que PassagerStandard à la montée, sa méthode monteeDans() va donc être modifée en un simple appel à super.monteeDans(args,....).
- PassagerStresse à l'inverse va maintenant avoir un comportement différent de PassagerStandard à la montée et va donc aussi voir sa méthode monteeDans() modifée.
Dans les deux cas, le comportement lors d'un nouvel arrêt du passager standard n'est plus commun aux deux autres caractères (aucune ne descend un arrêt avant sa destination). Le code de la méthode nouvelArret() de la classe de base ne peut donc plus être réutilisé par les classes dérivées, et la duplication de code pour la vérification de l'arrêt ne peut plus être évitée.

6.2 La classe abstraite PassagerAbstrait :

Plutôt que d'adopter la classe PassagerStandard comme classe de base à tous les caractères, il est préférable de remanier le code en définissant une classe abstraite PassagerAbstrait. Les classes de caractères héritent de cette classe abstraite. La classe abstraite contient le code commun/général et les classes dérivées le code variable particulier à chaque caractère.

2.1 Définir ce qu'il faut paramétrer
2.2 Remanier la classe PassagerAbstrait
Déclarer deux méthodes abstraites choixPlaceArret() et choixPlaceMontee() dans la

classe PassagerAbstrait.
Donner la portée de ces deux méthodes ?
La portée de ces deux méthodes dépend de la partie qui les a implémentés.
Pour la partie Abstract, ces méthodes ont une portée interne au paquetage pour les besoins de cette partie.
Pour la partie Interface, ces méthodes ont une portée publique. Ce n'est cependant pas nécessaire, car il suffit qu'elles ne soient pas de portée privée.

Comment éviter la redéfinition des deux méthodes monterDans() et nouvelArret() dans les classes dérivées.
Pour ne pas redéfinir ces deux méthodes dans les classes dérivées, il suffit d'effectuer un appel à la méthode choixPlaceMontee() (resp. choixPlaceArret()) dans le corps de la méthode monterDans() (resp. nouvelArret()), tout ceci dans la classe abstraite. Ainsi, les deux méthodes abstraites choixPlaceMontee() et choixPlaceArret() vont provoquer l'appel aux méthodes du même nom définies dans les classes dérivées. Le code des méthodes monterDans() et nouvelArret() est donc commun à toutes les classes, et n'a donc pas besoin d'être réécrit.

Remanier le code des classes concrètes de passager qui ne contiennent maintenant que la redéfinition des deux méthodes choixPlaceMontee(), choixPlaceArret() et leur constructeur.
À la manière du test de recette Simple, réaliser une autre classe servant de test de recette utilisant tous les caractères.
Remarque : Dans la branche avec la fabrique, l'ajout des deux caractères ne doit pas modifier le source de la fabrique. Astuce: on utilise des enumérations (collections) de sous-types de passager..
Étant donné que seules les classes PassagerAbstrait, PassagerIndecis, PassagerStresse et PassagerStandard sont modifiées, les diagrammes suivants ne représenteront que ces dernières, le reste étant similaire au début de la partie 2.

6.3 Boutez vos neurones :

Expliquer précisément comment l'appel à la méthode de la classe de base (par exemple monterDans()) va appeler la méthode (ici choixPlaceMontee()) redéfinie dans la bonne classe dérivée.

Prenons l'exemple de monterDans(). Cette méthode définie dans la classe abstraite PassagerAbstrait contient un appel à la méthode abstraite choixPlaceMontee(). Lors de la compilation, l'adresse de la fonction à appeler ne sera pas connue (d'où le mot-clé abstract). Lors de l'exécution, l'appel à la méthode monterDans() dans une classe dérivée va remplacer la fonction inconnue choixPlaceMontee() par la fonction du même nom redéfinie dans cette classe.

6.4 Factorisez les tests :

4.1 La classe de tests de PassagerAbstrait
4.2 Paramétrer l'instanciation
Nous avons besoin d'une méthode abstraite dans la classe TestPassagerAbstrait qui sera redéfinie dans les classes dérivées pour instancier la classe à tester.
Dans la classe TestPassagerAbstrait, définir la méthode:
abstract protected PassagerAbstrait creerPassager(String nom, int destination); Pourquoi prendre comme type de retour PassagerAbstrait et non pas Passager ou Usager ?
Si l'on utilise Passager ou Usager comme type de retour, alors seules les fonctions disponibles dans ces classes en particulier seront accessibles pour les tests. Par exemple, si le type de retour est Usager, alors seule la méthode monterDans sera accessible, rendant les tests d'autres fonctions impossibles.
Le type de retour PassagerAbstrait permet donc d'avoir accès à toutes les fonctions à

tester.(inconvénient d'une déclaration en tableau directe prenant en compte les contraintes du POLYMORPHISME)

7 POO: TD6

Substitution d'objets : Lien est-UN ou a-UN : Ce sixième TD fait suite au TD précédent où nous avons dû modifier nouvelArret() et monterDans() en ajoutant choixPlaceArret() et choixPlaceMontee() afin de faciliter la mise en œuvre de différents caractères de passagers. Ce TD demande donc de mettre en œuvre différents caractères à la montée ainsi qu'aux changements d'arrêts.

Avant de pouvoir commencer le sujet, nous avons également dû revenir sur l'arborescence de notre dossier principal qui ne correspondait pas à ce qui était attendu, en effet la base de l'arborescence contenait toujours des dossiers sources et build datant des TD où nous n'avions pas séparé le travail en deux dossiers : abstract et interface.

Après s'être remis à niveau, nous avons pu entamer le TD et répartir le travail des équipes. La première équipe composée de Nathan et Ismail s'occupe d'implémenter les différents caractères à la montée ainsi qu'aux changements d'arrêts sur la partie Abstract.

La seconde équipe composée de Maxime et Martial s'occupe d'implémenter les différents caractères à la montée ainsi qu'aux changements d'arrêts sur la partie Interface.

Avant de commencer à écrire le code, les différentes équipes ont dans un premier temps lu le sujet afin de comprendre ce qui nous était demandé et les subtilités impliquées lors de l'écriture. Après avoir réfléchi par groupe et s'être concerté ensemble pour être sûr que nous étions d'accord sur ce qui nous était demandé de faire, les équipes ont pu coder les différents caractères à la montée ainsi qu'aux changements d'arrêts sur leur partie respective.

Combien faut-il construire de classes concrètes pour effectuer les combinaisons ?
Si l'on décide de prendre cette solution est de redéfinir les deux méthodes abstraites en utilisant l'héritage. Cela signifie que nous avons une classe concrète par caractère. Autrement dit, nous en avons autant que de combinaisons possibles. Ici, nous devons implémenter 4 caractères différents à la montée et 5 aux changements d'arrêts, nous avons donc 4*5=20 classes concrètes afin d'effectuer toutes les combinaisons.

En quoi cette solution n'est-elle pas satisfaisante ? Cette solution n'est pas satisfaisante car elle implique de réécrire le code des différents caractères en permanence. Sur les 20 classes concrètes que nous pouvons coder, chaque caractère à la montée est réécrit 5 fois et chaque caractère au changement d'arrêt est réécrit 4 fois ce qui fait beaucoup de réécriture. En plus de devoir réécrire les différentes méthodes cela signifie que le code sera beaucoup plus dur à modifier car si l'on souhaite modifier un de ces caractères, nous devons penser à le modifier dans chacune des combinaisons où il est impliqué alors qu'il serait beaucoup plus simple de n'avoir qu'un endroit à modifier par méthode.

7.1 Développement du lien a-UN :

La hiérarchie pour la méthode choixPlaceArret():
En termes d'objets, donner la différence entre les liens a-un et est-un qui explique l'ajout de ce paramètre.

Un lien a-un correspond à la définition d'un attribut d'une classe P de type d'une autre classe R. La classe P peut alors utiliser les méthodes de la classe R en utilisant son attribut. Un lien est-un correspond à une relation de type/sous-type entre une classe P qui implémente une classe R. Cela signifie alors que la classe P a les mêmes

propriétés que la classe R et que les méthodes de la classe R sont directement présentes dans la classe P.

Dans l'ancienne version, la classe de la méthode choixPlaceArrêt possédait un lien est-un avec PassagerAbstrait et héritait donc de ses propriétés et méthodes. Avec le passage au lien a-un, PassagerAbstrait est une instance contenu dans un attribut mais la classe ne possède pas les méthodes de PassagerAbstrait. Pour utiliser ses méthodes, elle doit donc faire appel à l'attribut contenant l'instance de PassagerAbstrait expliquant l'ajout de ce paramètre.

La classe abstraite et ses classes dérivées:

De quelle manière la classe abstraite modifie-t-elle son constructeur pour respecter l'encapsulation ?

Que ce soit dans la partie Abstract ou bien dans la partie Interface, on ajoute au constructeur un nouveau paramètre de type Comportement, spécifiant le comportement d'un passager lors de l'arrêt. Comme expliqué dans la question précédente, ce changement de paramètre s'explique par le fait que les méthodes choixPlaceArret ont désormais un lien a-un et doivent donc être passées en paramètre du constructeur.

En prenant comme exemple la combinaison MonteeSportif-ArretPrudent, expliquer pour les deux versions développées, de quelle manière et dans quelle partie du code est créée cette combinaison.

Dans l'exemple de la combinaison MonteeSportif-ArretPrudent, quelle que soit la version utilisée, la combinaison est faite dans le constructeur de la classe MonteeSportif-ArretPrudent.
Le caractère à la montée est récupéré par héritage avec extends MonteeSportif tandis que le caractère au changement d'arrêt est récupéré par une instanciation (ArretPrudent.obtenirInstance() qui est de type private final static donc le getter est de type public static obtenirInstance() avec un constructeur privé pour limiter les réutilisations trop abusives du code.)

Dans le cas où la combinaison MonteeRepos-ArretNerveux n'est pas permise, expliquer la conséquence dans les deux versions développées.

Dans le cas où la combinaison MonteeRepos-ArretNerveux n'est pas permise, cela signifie que nous aurons une erreur si nous essayons de créer une telle combinaison.
Il faudrait alors ajouter une condition dans le constructeur de MonteeRepos de sorte que celui-ci retourne une erreur et quitte le constructeur si l'on essaye d'instancier un type ArretNerveux.
(Dans le type de passager (la classe ne contenant que le constructeur qui étend le type de montée ; c'est plus pratique car c'est là où on connaît l'instance soit le type de l'arret donné)).

7.2 Boutez vos neurones :

Héritage multiple:

Retrouvez-vous le même problème que dans la section 1.1 ?

Avec l'héritage multiple, nous ne retrouvons plus le même problème qu'à la section 1.1. En effet, étant donné que nous pouvons hériter du caractère à la montée ainsi que du caractère au changement d'arrêt, nous pouvons donc créer une classe pour chaque caractère et établir deux liens "est-un" pour chaque passager. Le problème de

duplication de code que nous avions est donc résolu avec cette solution.

Expliquer la différence de construction de la combinaison entre un lien est-un (la relation de type/sous-type) et un lien a-un dans un langage comme Java.

Lors de l'utilisation d'un lien "est-un", on s'attend à utiliser "extends" ou "implements" dans la définition de la classe concrète, tandis que pour le lien "a-un" il faut ajouter un attribut dans la classe, puis un paramètre du même type dans le constructeur).

Les classes des caractères à l'arrêt.

Quelle propriété en déduisez-vous sur leurs instances ?

Etant donné que ces instances n'ont pas d'attributs, on peut en déduire que ce sont des objets constants car leur comportement ne peut pas être modifié.
Resume:
En général, dans ce TD, on y allé un peu plus loin. On a combiné lien est-Un et lien a-Un. Les caractères à la montée définissent (1) 4 classes dont le lien est est-UN avec les classes concretes donc les classes concretes etendent ces classes. Pour que l'extension soit limitée, le constructeur va prendre en argument rien et retourner super(,,"caractereArret") et donc la montée qui étend PassagerAbstrait et le passager qui en hérite a accès au constructeur de la montée qui fait la même chose avec le 3ième argument comme constructeur. Il va modifier statuer l'état général initial d'un de ses attributs décrivant les descentes soit un attribut ArretComportement type(−) dans PassagerAbstrait et l'interface donc ArretComportement et les 5 classes de comportements differentes a l'arret implémentant cet interface. Reste à ajouter comme on a dit au début: dans monterDans on a appel à la fonction choixPlaceMontee qui reste abstraite aux yeux de notre développement (pas d'héritages tjrs) mais dans nouvelArret ca va appeler choixPlaceArret via comportement(lien a-UN) soit un appel comme ceci: comportement.choixPlaceArret(Passager p, Vehicule v, int distanceDestination); //on a ajouté un 3ième argument Passager p car avant on héritait donc on savait le passager abstrait et celui-là mais là on va pas hériter donc on DOIT LE PRéciser soit AJout dans la liste des arguments: Passager p. Voilà.

des td:
1.C et java.
2.Les tests unitaires et découverte.
3.Développement et tests fonctionnels.
4.intégration et rassemblement(partage d'instances)
5.Remaniement(ajout de caractères)
6.Substitution d'objets(masquage d'informations au client + factorisation des tests)
7.Héritage inconvénient et avantage du lien a-UN qui fait le lien entre les 2 liens est-UN. On peut combiner différentes combinaisons en créer de nouvelles sous combinaisons unitaires sans pouvoir être amené à réecrire le code à chaque fois..