Programmation Orientée Objet
TD 6 : Lien « est-un » ou « a-un »
Compte-rendu de l'équipe *Lorem Ipsum* (G4E1)

Coordinateur : Aymeric FERRON, Tandem 1 : Rémi DEHENNE, Tom MOËNNE-LOCCOZ,
Tandem 2 : Sébastien DELPEUCH, Aurélien MOINEL

16 octobre 2020

Résumé

Dans le cadre de cette sixième séance de TD des modules PG202–PG203 à l'ENSEIRB-MATMECA, nous souhaitons permettre à chaque type de passager de disposer de *stratégies* pour choisir une place, aussi bien à l'arrivée du bus qu'à chaque arrêt. Contrairement à la séance précédente, on souhaite que ces deux comportements soient indépendants, afin de combiner les stratégies à la montée[1] et à l'arrêt en limitant les duplications de code. Pour cela, on décide d'utiliser des relations d'héritage, ainsi que des associations (agrégations ou compositions).

1 Une première implémentation « naïve »

Dans le but de combiner des comportements à l'arrivée du véhicule et à chaque arrêt desservi, il est nécessaire de séparer les définitions des méthodes `choixPlaceMontee()` et `choixPlaceArret()` dans des classes distinctes. Nous explorons par la suite différentes solutions afin de « lier » des instances de passagers à des comportements.

Une première solution naïve consisterait à faire hériter un `PassagerConcret` à la fois d'une classe définissant un comportement à la montée et un comportement à un nouvel arrêt. L'héritage multiple n'étant pas autorisé en langage Java, cette solution est éludée dans un premier temps. Elle est détaillée en sous-section 4.2.

Pour contourner cette absence d'héritage multiple en ne conservant que des relations de « type/sous-type », une des stratégies doit être définie dans une classe mère ; l'autre stratégie est définie dans une classe fille. Ainsi, en Figure 1, les stratégies de *nouvel arrêt* sont définies dans des classes dérivées des classes qui définissent la *montée*.

1. À des fins de simplification, nous utilisons le terme de « stratégie lors de la montée d'un passager ». En réalité, il faudrait utiliser le terme de « stratégie à l'arrivée du véhicule », puisqu'un passager peut décider de ne pas monter à bord (lorsque le véhicule est plein, par exemple).

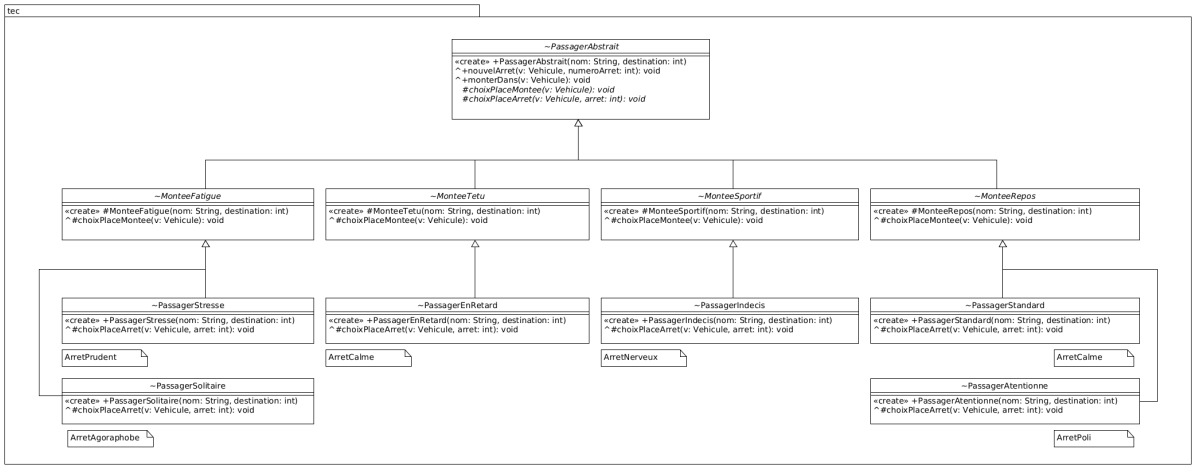

FIGURE 1 : Diagramme de classes de la solution utilisant exclusivement l'héritage

Ainsi, chaque combinaison de stratégies « montée–nouvel arrêt » se traduit par une nouvelle classe à créer. Dans le code actuel, sont définis $n = 4$ comportements à la montée (« fatigué », « têtu », « reposé » et « sportif »), et $m = 5$ comportements à un arrêt (« calme », « nerveux », « prudent », « agoraphobe » et « poli »). Si l'on souhaitait autoriser toutes les combinaisons possibles, il faudrait alors implémenter $m \times n = 5 \times 4 = 20$ classes. En supposant $n = m$, le nombre de classes à créer est alors quadratique selon le nombre de comportements.

Par ailleurs, autoriser toute combinaison de caractères suppose que le code d'un comportement de *nouvel arrêt* doit être dupliqué autant de fois que de comportements de *montée*. Concrètement, le code de la méthode `choixPlaceArret()` serait dupliqué entre les différentes classes `Passager*` [2]. Par exemple, la Figure 1 représente deux passagers `PassagerEnRetard` et `PassagerStandard`, dont les comportements à la montée sont respectivement `MonteeTetu` et `MonteeRepos`, mais qui implémentent tous deux le comportement « calme » à l'arrêt. Par conséquent, le code de la méthode `choixPlaceArret()` est dupliqué entre ces deux classes. Au final, le nombre de méthodes dupliqués est quadratique selon le nombre de comportements.

2 Héritages « en cascade » et agrégation du comportement d'arrêt

Afin d'éviter de définir une méthode `choixPlaceArret()` pour chaque combinaison de comportements « montée–nouvel arrêt », on utilise désormais une relation d'association. Ainsi, l'une des solutions consiste à utiliser une relation d'association entre la classe abstraite `PassagerAbstrait` et un

2. Pour clarifier le propos, les notations `Arret*`, `Montee*` et `Passager*` sont utilisées afin de désigner « toute classe concrète dont le nom commence par « `Arret` », « `Montee` » ou « `Passager` » » (exceptée `PassagerAbstrait`).

comportement de nouvel arrêt. Pour cela, chaque comportement est défini dans une classe qui implémente une interface commune `ComportementNouvelArret`. De ce fait, le `PassagerAbstrait` définit un attribut référençant un `ComportementNouvelArret`. Le comportement à utiliser dépend donc du type concret de l'objet référencé : le comportement à adopter est donc déterminé *dynamiquement* à l'exécution, et non *statiquement* à la compilation.

Plus précisément, cette association correspond à une agrégation : les objets implémentant l'interface `ComportementNouvelArret` sont référencés par un passager, mais ne sont pas instanciés par celui-ci. Par conséquent, une même instance de comportement à l'arrêt peut être affecté à plusieurs passagers, limitant de ce fait le nombre d'objets en mémoire. En effet, les stratégies d'arrêts ne définissent pas d'attributs d'instance, uniquement des méthodes. Par conséquent, ces instances peuvent être partagées entre différents passagers sans risquer de modifier leur état interne et d'entraîner des comportements inattendus. Il est donc possible de ne créer qu'un seul objet par classe de comportement, en utilisant le patron « singleton » : au lieu de créer de nouveaux objets, le code client accède à une unique instance renvoyée par la méthode `getInstance()`. Pour cela, le constructeur de la classe est déclaré *privé* et n'est utilisé que pour créer l'unique instance de la classe, référencée par un attribut statique de la classe de comportement.

Finalement, cette solution permet de palier à la duplication du code des méthodes `comportementNouvelArret()` soulevée en section 1. En revanche, il est toujours nécessaire d'instancier une classe de passagers par combinaison de stratégies « montée–arrêt », comme illustré en Figure 2 et Figure 3. Par conséquent, l'absence d'une classe combinant une stratégie d'arrêt et une stratégie de montée garantit dès la compilation que cette combinaison ne peut pas exister. La solution oblige cependant le développeur à créer une classe par combinaison autorisée. Elle est donc appropriée lorsque l'on souhaite autoriser les combinaisons au cas par cas, selon une logique de liste d'inclusion.

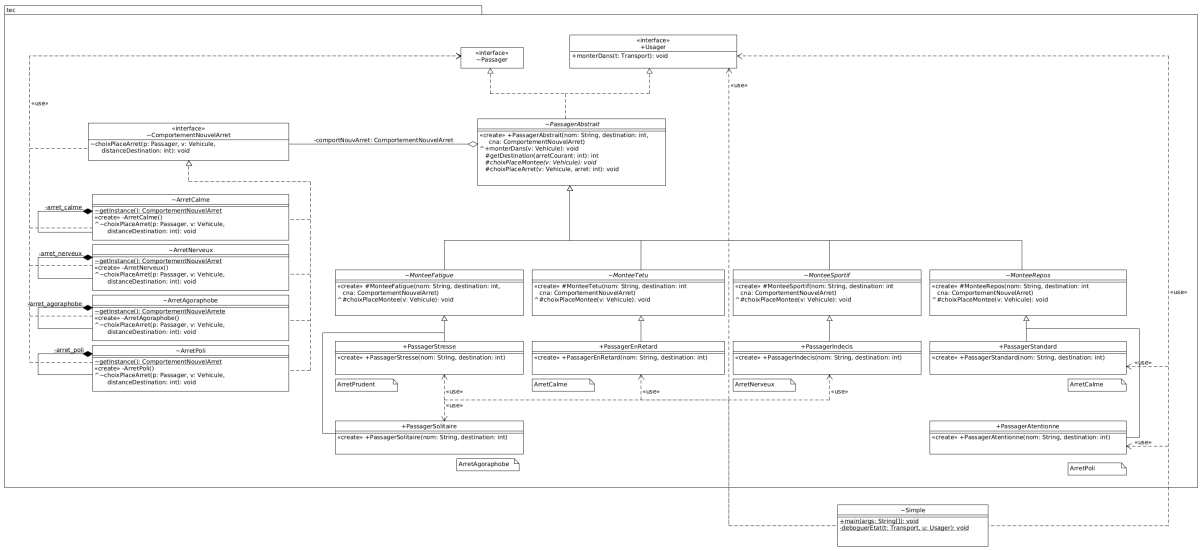

FIGURE 2 : Diagramme de classes de conception de la solution d'héritage successifs et d'agrégation d'une stratégie d'arrêt (version *classe abstraite*)

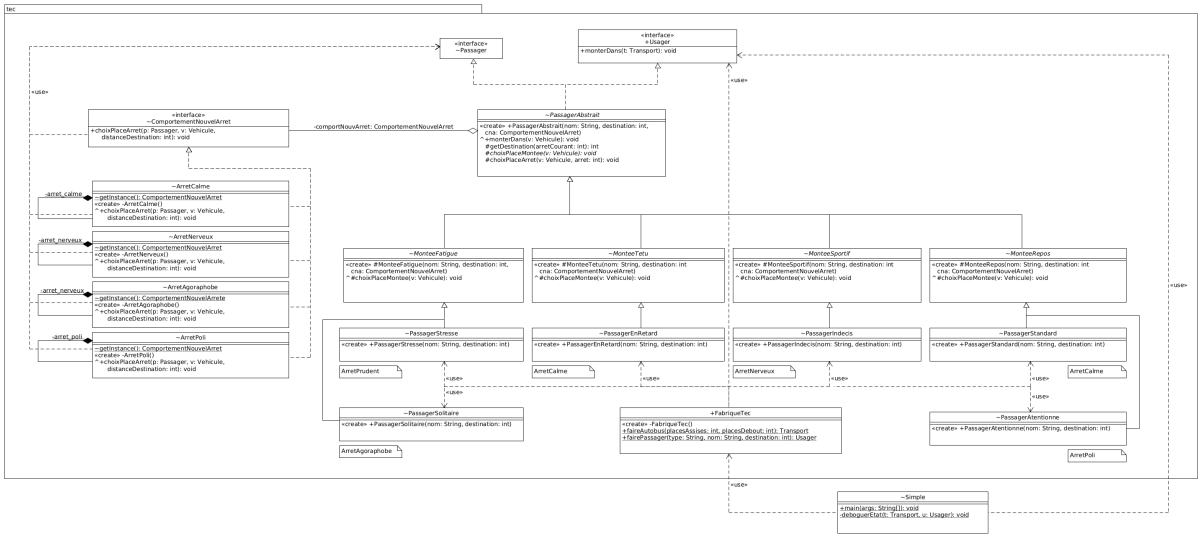

FIGURE 3 : Diagramme de classes de conception de la solution d'héritage successifs et d'agrégation d'une stratégie d'arrêt (version *fabrique*)

Cependant, le changement de la relation de « type/sous-type » par la relation « a-un » implique de modifier le prototype de la méthode `choixPlaceArret()`. En effet, celle-ci n'est plus directement définie au sein des classes concrètes des passagers, il n'est donc plus possible *en l'état* d'accéder aux attributs ou d'appeler les autres méthodes de `Passager`. Ainsi, pour interagir avec l'instance

de `Passager*`, il est nécessaire d'ajouter une référence de `Passager` en paramètre de la méthode `choixPlaceArret()`.

Par conséquent, les comportements de nouvel arrêt sont désormais définis dans des classes externes à une classe `Passager*`. Ils n'ont donc plus accès aux attributs ou méthodes privées de `PassagerAbstrait` : si le code de la méthode avait utilisé des attributs ou méthodes privés, il faudrait rendre ces méthodes visibles au *paquetage*, ou ajouter de nouvelles méthodes à exposer au *paquetage*. Cela alourdirait probablement le code, et autoriserait d'autres classes du paquetage à modifier l'état du `Passager` de manière non autorisée. Dans le cas présent, le code de `choixPlaceArret()` ne faisait appel qu'à des méthodes au moins visibles au *paquetage*.

Par ailleurs, même si l'attribut qui référence le `ComportementNouvelArret` est défini dans `PassagerAbstrait`, on souhaite autoriser les classes dérivées à fixer le comportement à utiliser. Pour ce faire, le constructeur de `PassagerAbstrait` doit permettre aux classes filles de spécifier l'instance de `ComportementNouvelArret` à utiliser, en ajoutant un paramètre `ComportementNouvelArret`. Cette modification est également répercutée dans les différentes classes de `Montee*`.

Enfin, pour que les classes concrètes `Passager*` héritant d'une `Montee*` puissent fixer le comportement d'arrêt à utiliser, le prototype de leur constructeur n'est pas modifié : le client ne doit pas pouvoir préciser le comportement d'arrêt à utiliser lors de l'instanciation d'un `Passager*`. À la place, le constructeur transmet les paramètres `nom` et `destination` au constructeur mère, en y ajoutant « en dur » le comportement d'arrêt à utiliser.

Finalement, cette solution ne répond que partiellement au problème initial : le code des méthodes `monterDans()` n'est pas dupliqué, mais il est toujours nécessaire de créer une classe par combinaison autorisée (potentiellement de manière quadratique selon le nombre de comportements). Toutefois, cette méthode permet de s'assurer à la compilation que le client ne peut pas créer de combinaison interdite, et non à l'exécution, ce qui demanderait davantage de vérifications. De même, elle est plutôt adaptée si l'on souhaite interdire toutes les combinaisons par défaut, puis autoriser quelques combinaisons au cas par cas, selon un principe de liste d'inclusion.

3 Interdiction de combinaisons par levée d'exceptions

On adopte maintenant une approche opposée : on souhaite autoriser toute combinaison par défaut, puis en interdire certaines au cas par cas à l'exécution, selon un principe de *liste d'exclusion*. Cette approche vise également à éviter de créer une classe concrète `Passager*` pour chaque combinaison. On se propose donc de mettre en place une nouvelle implémentation conservant la relation d'association, mais dont la combinaison « montée–nouvel arrêt » n'entraîne pas de création de classes `Passager*`.

En effet, le mécanisme d'héritage utilisé dans la solution précédente résout *à la compilation* les méthodes à appeler, selon un principe de *surcharge de méthode* : on parle alors de « liaison statique ». À l'inverse, avec une relation d'association qui référence une classe possédant des filles, si une méthode est *redéfinie* par les classes dérivées, alors la méthode à exécuter diffère selon la classe concrète de l'instance. Cet aiguillage ne peut être résolu qu'à *l'exécution* : on parle alors de « liaison dynamique ». Cette liaison dynamique nous permet justement de faire varier le comportement d'un passager à l'exécution, sans créer autant de classes qu'il existe de combinaisons différentes.

On implémente désormais ces deux méthodes, avec quelques différences, sur les branches `interface` et `abstract`.

3.1 Implémentation avec fabrique

Dans la solution *interface* illustrée en Figure 4, la fabrique n'instancie plus des `Passager`⋆ dont le comportement de nouvel arrêt est fixé. En effet, c'est la fabrique elle-même qui instancie une des classes de `Montee`⋆, désormais concrètes, en indiquant à leur constructeur le `ComportementNouvelArret` à utiliser.

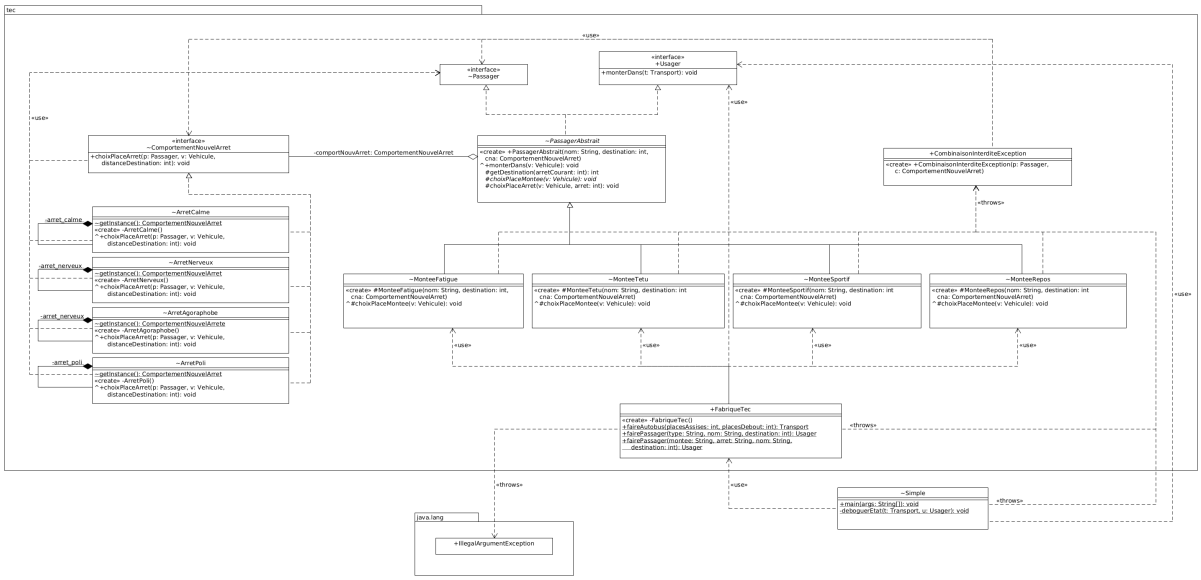

FIGURE 4 : Diagramme de classes de conception de la solution d'interdiction des combinaisons à l'exécution (version *fabrique*)

Le client ne peut quant à lui instancier des passagers que par le biais de la méthode de fabrique `fairePassager()` et n'a pas accès aux classes concrètes de `Montee`⋆ ou aux classes dérivées de `ComportementNouvelArret`.

Pour que le client puisse choisir une combinaison de stratégies « montée–arrêt », on modifie le prototype de `fairePassager()` : au lieu d'une chaîne de caractères `type` (`"PassagerStandard"`, `"PassagerEnRetard"`…), on demande deux chaînes `comportementMontee` et `comportementNouvelArret` correspondant aux noms des classes de comportements (`"MonteeFatigue"`, `"ArretPoli"`…). Dans le cas d'une chaîne de caractères invalide, une exception est levée.

Par ailleurs, chaque constructeur de `Montee*` vérifie que la combinaison « montée–arrêt » est autorisée, et lève une exception sinon. Même si ce choix entraîne une dépendance entre la classe de montée et les classes concrètes `Arret*` qu'elle souhaite interdire, cela évite de réaliser ces vérifications dans la fabrique et de modifier systématiquement `creerPassager()` pour interdire une combinaison. De même, ces vérifications dans les constructeurs de `Montee*` peuvent être réutilisées quasiment à l'identique dans la branche *abstract*.

On aurait pu également décider de ne pas modifier le prototype de la méthode de fabrique `fairePassager()`, afin de ne pas « casser » le code existant. Néanmoins, il aurait fallu, dans le code de la fabrique, associer chaque chaîne `"PassagerX"` à un couple (`MonteeY`, `ArretZ`), ce qui revient à autoriser au cas par cas les combinaisons par une liste d'inclusion. Il serait aussi possible à la fois d'autoriser les anciennes chaînes de caractères `"PassagerStandard"`, `"PassagerStresse"`, `"PassagerIndecis"` et de nouvelles chaînes au format `"<ComportementMontee><Separateur><ComportementArret>"` (par exemple, `"MonteeFatigue-ArretPrudent"`) : on aurait alors regroupé deux paramètres dans un seul, ce qui rendrait le code moins lisible, moins maintenable, et aurait obligé à *parser* les paramètres.

Finalement, on conserve une définition de `fairePassager(String type, String nom, int destination)`, qui ne reconnaît que les chaînes `"PassagerStandard"`, `"PassagerStresse"` et `"PassagerIndecis"` et appelle `fairePassager(String comportementMontee, String comportementNouvelArret, String nom, int destination)` avec les paramètres adéquats. L'ancienne méthode est alors toujours utilisable, mais dépréciée : on ajoute l'annotation `@Deprecated` avant sa définition, ainsi qu'une explication dans un commentaire Javadoc. À la compilation, tout code qui utilise cette méthode dépréciée sera alors informé qu'il doit mettre à jour son code avant qu'elle soit définitivement supprimée.

Ainsi, pour instancier un passager de comportements `MonteeSportif` et `ArretPrudent`, le code client appelle la méthode de fabrique `fairePassager("MonteeSportif", "ArretPrudent", "<Nom>", 5)`. Celle-ci appelle le constructeur `MonteeSportif`, et lui passe en argument l'instance partagée statique renvoyée par `ArretPrudent.getInstance()`. Si le constructeur ne lève pas d'exception, alors la combinaison est autorisée et une instance est renvoyée au client. En revanche, la fabrique doit être modifiée et recompilée à chaque fois qu'une classe de comportement est ajoutée, afin d'associer chaque chaîne à une instance (pour les arrêts) ou à un constructeur (pour les montées). Cependant, il ne s'agit que d'ajouter qu'un cas de **`switch`** par *comportement*, non par *combinaison* de

comportements : ces ajouts ne sont donc que *linéaires* en fonction du nombre de comportements.

3.2 Implémentation avec classes abstraites

L'implémentation pour la branche *abstract* proposée en Figure 5 est relativement similaire. Contrairement à la fabrique, et comme au quatrième TD « Substitution d'objet », les méthodes sont masquées non pas en interdisant au client d'accéder aux classes concrètes, mais en intervenant directement sur la portée des méthodes. De ce fait, on autorise le client à accéder aux classes `Montee*` et `Arret*`, mais pas à appeler certaines de leurs méthodes : pour réaliser cela, `ComportementNouvelArret` est implémenté sous forme de *classe abstraite* et non *d'interface*, puisque les méthodes d'une interface sont nécessairement *publiques* en Java. Ici, le client appelle directement les constructeurs des `Montee*` avec un argument `Arret*`. Ces constructeurs peuvent lever des exceptions lorsque la combinaison n'est pas autorisée.

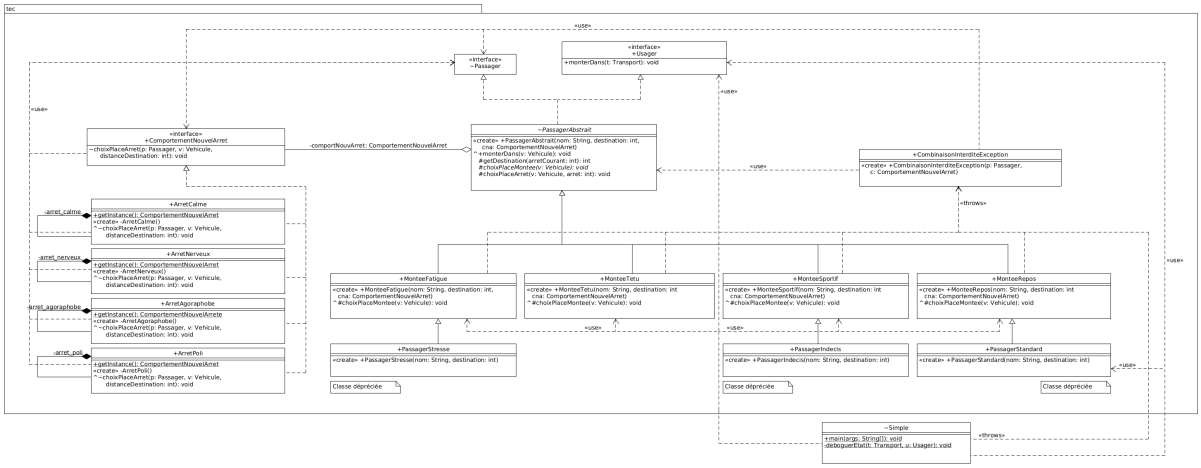

FIGURE 5 : Diagramme de classes de conception de la solution d'interdiction des combinaisons à l'exécution (version *abstract*)

Ainsi, pour instancier un passager aux comportements `MonteeSportif` et `ArretPrudent`, le client appelle le constructeur avec l'instance partagée de `ArretPrudent` : l'instanciation est réalisée par un `new MonteeSportif("<Nom>", 5, ArretPrudent.getInstance())`.

Les classes `PassagerStandard`, `PassagerIndecis` et `PassagerStresse` sont conservées le temps de permettre à un éventuel client de modifier son code et de ne plus les utiliser. Elles sont marquées comme dépréciées et seront à terme supprimées.

Contrairement à la fabrique, ajouter une classe de comportement ne modifie pas le code existant, sauf pour interdire un nouveau comportement d'arrêt dans une classe de montée. Cependant, le client est

très dépendant de l'implémentation : d'une part, si une classe de comportement est renommée ou supprimée, le code client qui l'utilise ne compilera plus. D'autre part, pour créer un passager, il doit instancier une `Montee*`[3] en lui passant en argument une instance partagée de `Arret*` obtenue par un `getInstance()`. On préférerait sans doute fournir au client une couche d'abstraction afin de créer des passagers, comme une fabrique, même si celle-ci possède ses propres inconvénients.

3.3 Nouvelle architecture de tests

Dans les deux versions, les tests doivent être sensiblement modifiés : on ne teste plus un `Passager*` aux comportements fixés, mais on vérifie indépendemment chaque comportement d'arrêt ou de montée, ainsi que leurs combinaisons.

Tout d'abord, le prototype de la méthode abstraite `creerPassager()` de la classe `TestPassagerAbstrait` nécessite un nouveau paramètre de type `ComportementNouvelArret`. Ce constructeur suit en effet l'évolution du constructeur de la classe `PassagerAbstrait` pour instancier un passager.

Ensuite, les comportements de montée et de nouvel arrêt sont testés unitairement : ici, on prend l'exemple des classes `MonteeFatigue` et `ArretPrudent`[4].

D'une part, les tests de `MonteeFatigue` sont écrits au sein d'une classe `TestMonteeFatigue` héritée de `TestPassagerAbstrait`. Elle s'assure que le comportement de `choixPlaceMontee()` est correct en reprenant le code de `choixPlaceMontee()` de l'ancienne classe `TestPassagerStresse`. Il faut également vérifier que le constructeur de `MonteeFatigue` lève bien une exception `CombinaisonInterditeException` lorsque le comportement d'arrêt doit être interdit : on peut par exemple choisir d'interdire la combinaison `MonteeFatigue`–`ArretNerveux`. À l'inverse, il est impératif de vérifier qu'aucune exception n'est levée pour les comportements d'arrêt autorisés. Enfin, dans le but de conserver des tests *unitaires*, tous les tests, saufs les tests des combinaisons, utilisent un objet factice `FauxArret` dont le corps de la méthode `choixPlaceMontee()` est vide.

D'autre part, le comportement de la méthode `choixPlaceArret()` d'`ArretPrudent` est testé par une instance de `TestArretPrudent` héritée de `TestPassagerAbstrait`. Le comportement d'arrêt est associé à une instance de l'objet factice `FausseMontee`, laquelle n'a aucun comportement à la montée. Le corps de `testChoixPlaceArret()` est repris de l'ancienne classe `TestPassagerStresse`.

3. Selon nous, ce nom de classe concrète est particulièrement contre-intuitif pour l'utilisateur.
4. D'autres comportements ont été testés, mais pas la totalité : par manque de temps, `MonteeTetu`, `ArretPoli` et `ArretAgoraphobe` n'ont pas été traités.

4 Solutions alternatives : double agrégation et héritage multiple

Enfin, on examine deux autres solutions envisageables mais interdites par le sujet ou le langage Java, afin de vérifier si elles répondraient réellement au problème posé. Dans un premier temps, on s'intéresse à une implémentation qui utilise une association dans le but de définir *à l'exécution* le comportement de montée du passager. Puis, on s'intéresse à l'héritage multiple, inexistant en Java, qui pourrait éventuellement remplacer l'association `PassagerAbstrait`–`ComportementNouvelArret`.

4.1 Suppression des relations d'héritage par une double agrégation

Une solution interdite par le sujet pour des raisons pédagogiques consisterait à remplacer la relation de « type/sous-type » du comportement de montée par une relation d'agrégation, comme avec le comportement d'arrêt. On créerait ainsi une nouvelle interface `ComportementMontee` dont hériteraient les classes de montée, désormais sans attributs. Ainsi, un attribut d'instance `comportMontee` référençant un `ComportementMontee` serait ajouté au `PassagerAbstrait`, et la méthode `monterDans()` du `PassagerAbstrait` se contenterait d'appeler la méthode `monterDans()` de `comportMontee`.

De ce fait, la classe `PassagerAbstrait` n'aurait plus de méthode abstraite : les comportements d'arrêts seraient définis dans les classes concrètes dérivées de `ComportementNouvelArret`, et ceux de montée dans les classes concrètes dérivées de `ComportementMontee`. Par conséquent, on déclarerait la classe *concrète*, et on la renommerait `PassagerConcret` pour des raisons sémantiques.

Généralement, cette solution de « patron stratégie » est utilisée lorsque l'on souhaite utiliser indistinctement une méthode ou une autre *à l'exécution*. Au lieu de définir la méthode à exécuter de façon statique par un héritage *à la compilation*, cette méthode est déterminée *à l'exécution* par des mécanismes d'aiguillage. Cette solution est donc particulièrement utilisée pour « combiner » des comportements à l'exécution.

Cependant, elle ne semble pas nécessairement plus adaptée que les solutions précédentes dès lors qu'on cherche à interdire certaines combinaisons. En effet, en sous-sections 3.1 et 3.2, chaque classe de montée interdit *localement* certains comportements d'arrêts. En ne conservant qu'une classe `PassagerConcret`, toutes les combinaisons doivent être vérifiées dans le constructeur de `PassagerConcret` : si le nombre de combinaisons interdites devenait important, ce code pourrait devenir long et difficile à maintenir.

4.2 Héritage multiple

Finalement, on s'interroge si l'héritage multiple, interdit en Java, aurait permis ou non de répondre à ce problème de manière adéquate.

Instinctivement, un passager concret comme un `PassagerStandard` pourrait être vu comme une classe dérivée de deux classes mères `MonteeRepos` et `ArretCalme`, lesquelles définissent respectivement des méthodes `choixPlaceMontee()` et `choixPlaceArret()`. À nouveau, on se heurte à des contraintes de liaison *statique* : l'héritage est un mécanisme qui a lieu à la *compilation*, une classe de passager ne peut donc pas définir *dynamiquement* ses comportements de montée et d'arrêt.

Dès lors, même si aucun code n'est dupliqué d'une méthode à l'autre, le nombre de classes à instancier reste quadratique en le nombre de comportements. Cette solution est donc plutôt similaire à celle présentée en section 2 : elle serait adaptée pour autoriser un petit nombre de combinaisons au cas par cas, à la compilation.

Conclusion

Finalement, aucune des solutions ne semble réellement se démarquer des autres. Hormis la solution *naïve* qui duplique énormément de code, le reste des solutions répond à des besoins différents.

D'une part, les implémentations d'héritage « en cascade » (section 2) et d'héritage multiple (sous-section 4.1) semblent pertinentes lorsqu'un petit nombre de combinaisons doit être autorisé. Même si une classe doit être créée pour chaque combinaison acceptée, seul son constructeur doit être défini. De même, un client ne peut pas essayer de créer une combinaison incohérente, puisqu'il n'existe alors pas de classe concrète correspondant à cette combinaison. Cette garantie est donc fournie par le compilateur, ce qui assure au développeur de repérer immédiatement ces erreurs. En contrepartie, le code doit être entièrement recompilé à chaque ajout de combinaison.

D'autre part, les méthodes de « levées d'exceptions » (section 3) et de double agrégation (sous-section 4.2) interdisent les combinaisons incohérentes via des exceptions levées à l'exécution plutôt que des garanties à la compilation, ce qui permet d'étendre le code sans le recompiler entièrement. D'une part, la version *interface* abstrait davantage les détails d'implémentation au client. D'autre part, seule la version *abstract* permet d'autoriser toute combinaison par défaut, et d'en exclure certaines au cas par cas.

Cependant, le sujet ne spécifie pas si les combinaisons doivent être autorisées ou interdites par défaut. Selon les besoins, on choisirait la solution la plus adaptée à la situation. Certaines questions laissent cependant penser qu'on souhaiterait éviter de créer une classe par combinaison. Nous avons donc choisi d'implémenter les solutions de la section 2, la double agrégation n'étant pas autorisée.

Commentaires

Aymeric (Coordinateur) : Au cours de ce TD, je me suis éloigné du code comme mon rôle de coordinateur le voulait et j'ai donné une architecture au rapport tout en gérant la répartition du travail dans les tandems. Comme tout problème, ce TD présentait plusieurs solutions, ayant chacune des avantages et des inconvénients. En C++, nous aurions pu utiliser l'héritage multiple à partir de nos classes de gestion d'arrêt et de montée pour créer des passagers personnalisés. Cette solution me paraît la plus naturelle depuis mon point de vue néophyte. Cependant, j'ai compris que cette solution n'était pas forcément la plus adaptée et qu'elle dépendait véritablement du contexte. D'autre part, en Java, cela s'avère impossible. Nous avons donc pris le parti d'utiliser un mélange d'agrégation et d'héritage afin de parvenir à nos fins, ce qui m'a donné un exemple concret d'utilisation des liens « a-un » en POO.

Rémi (Tandem 1) : Au début de la séance, nous avions choisi d'adopter la solution de création de combinaisons à la compilation pour les divers avantages qu'elle propose (section 2). En lisant le sujet plus en profondeur, nous nous sommes finalement aperçus qu'il ne fallait pas *vraisemblablement* créer une classe par combinaison, mais plutôt préférer les vérifications à l'exécution. Nous avons donc dû retravailler une bonne partie du code pour implémenter les solutions de la section 3 à la place. Nous avons donc pu comparer un certain nombre de solutions, y compris entre les branches *abstract* et *interface*.

Au final, si j'étais persuadé au début du TD que la solution de « double agrégation » aurait été la plus adaptée si elle avait été autorisée, je me suis rendu compte que celle-ci rendait laborieuse l'interdiction de combinaisons.

Toutefois, ces solutions me semblent toutes un peu « lourdes » à implémenter, dues au grand nombre d'interfaces et de classes à créer (au minimum une par comportement). Dans notre cas, il aurait pu être intéressant d'utiliser des mécanismes fonctionnels, comme des lambda expressions.

Tom (Tandem 1) : Dans l'ensemble, ce TD m'a paru plus simple que le précédent. Cependant, mon avis *a priori* sur la solution optimale s'est avéré être incorrect. Ma première hypothèse était en effet qu'une double relation d'association formerait une implémentation optimale. Pourtant, comme nous l'avons montré dans ce rapport, cette solution possède ses inconvénients. Notamment, cette solution n'est pas vraiment adaptée à une situation où les combinaisons de comportements interdites sont dominantes. Du reste, le travail effectué en TD avec mon binôme Rémi pendant le cours m'a apporté la majorité des pistes de compréhension. Il me semble que je conçois maintenant assez bien les intérêts des implémentations possibles, ce qui est très satisfaisant.

Aurélien (Tandem 2) : J'ai trouvé les idées et concepts liés à ce TD pertinents pour aborder les avantages/inconvénients entre association et héritage. En effet, au début du TD je ne mesurais pas l'intérêt de ne pas réaliser directement une double associations entre une classe concrète `PassagerConcret` et les différents comportements « à la montée » et « à l'arrêt ». Cependant, lors de la réalisation des dif-

férentes solutions, j'ai pu me rendre compte que les implémentations faisant intervenir des relations de type/sous-type possédaient un certain nombre d'avantages qu'une double association ne permet pas (toujours dans le cadre du TD). Par ailleurs, il me semble que la gestion des différentes combinaisons interdites, était un bon moyen de manipuler à nouveau les exceptions java en abordant un cas concret d'utilisation.

Sébastien (Tandem 2) : Suite à ce TD j'ai pu comprendre la différence entre les deux types de liens mis en places. De plus, de manière similaire au TD précédent, nous avons étudié tous ensemble l'architecture du code avant de nous lancer dans ce dernier. En outre, ayant l'habitude du C++ et de ses concepts, découvrir comment nous pouvions réaliser les combinaisons de comportement sans utiliser d'héritage multiple fut très intéressant.