Discussion du couplage et de la cohésion des modules

Couplage et cohésion sont deux notions clés en programmation orientée objet, essentielles pour créer un logiciel bien structuré, facile à entretenir et à faire évoluer.

La cohésion, c'est l'idée dans laquelle les éléments sont liés dans un module. Si un module est très cohésif, ça veut dire qu'il s'occupe d'une seule fonction ou tâche spécifique et qu'il inclut seulement ce qui est nécessaire pour cette fonction. Avoir une forte cohésion rend les modules plus faciles à comprendre, à réparer et à réutiliser.

Le couplage, cependant, c'est la dépendance entre différents modules. Quand le couplage est faible, chaque module peut fonctionner de manière indépendante. Cela rend chaque module du logiciel plus facile à maintenir, à tester et à réutiliser, car les modifications ne causent pas d'effets négatives sur les autres modules.

Dans notre système, nous avons organisé les classes de manière qu'elles soient bien structurées et qu'elles n'interagissent pas trop les unes avec les autres. Nous avons rassemblé des attributs et des méthodes pertinents dans des classes spécifiques. Par exemple, la classe «Reservation» possède uniquement les tâches pour gérer les réservations et la classe «InformationClient» qui s'occupe des informations des clients.
Dans ce cas, la conception permet de maintenir une bonne cohésion et un faible couplage.

Pour éviter que nos classes ne dépendent trop les unes des autres, nous avons utilisé l'héritage. Cela signifie que nous avons des classes de base, comme «Voyage» et «Section», et nous créons des versions plus spécifiques pour des choses comme «VolAerien», «ItineraireNaval», «TrajetTrain», ou les différentes sections, «SectionAvion», «SectionTrain» et «SectionPaquebot». Cela nous permet de partager des attributs et des méthodes communs entre les classes sans qu'elles aient besoin de dépendre entre elles.

Discussion du fardeau des classes

Quand une seule classe dans un programme essaie de gérer plusieurs aspects distincts d'une application, ça peut la compliquer, la rendre difficile à gérer, à améliorer et augmenter le risque d'erreurs, ce qui est un fardeau des classes. Pour éviter ce problème, il est important de bien organiser le code en suivant les règles de la programmation orientée objet. Cela implique la séparation des préoccupations, une cohésion élevée et un couplage faible.

Dans un système qui gère les réservations, une classe qui s'occuperait à la fois de

prendre les réservations et de gérer les paiements pour des services comme les vols, les paquebots et les trains pourrait vite devenir trop lourde à maintenir. Pour éviter ça, on peut :

Utiliser l'héritage et le polymorphisme : Des classes de base peuvent être créées pour les fonctionnalités partagées, ce qui aide à éviter de répéter le même code et rend les classes plus simples.

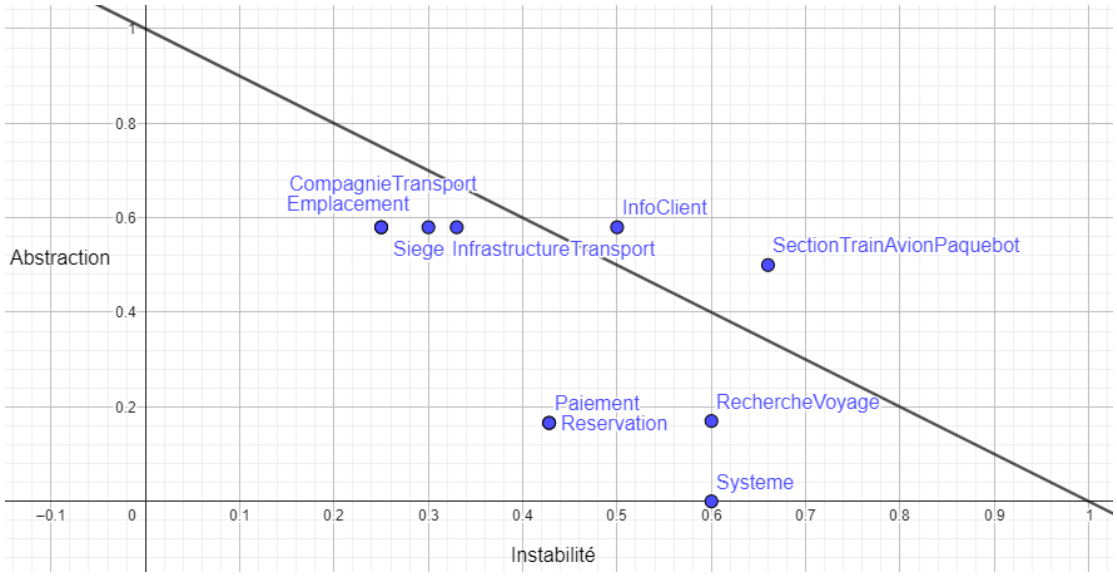
Encapsuler : Regrouper les données avec les méthodes qui fonctionnent sur les données, ce qui rend la classe plus cohérente.

Créer des classes d'auxiliaires : Pour les tâches spécifiques qui ne sont pas en lien avec la logique principale, on peut faire appel à des classes d'auxiliaire. Par exemple, pour valider ce que l'utilisateur entre ou pour le formatage des données.

Faire collaborer les classes : Au lieu d'une super-classe qui fait tout, on peut avoir plusieurs classes qui travaillent ensemble pour accomplir une tâche, comme une classe pour réserver et une autre pour payer dans le but de compléter une réservation complète.

En résumé, pour que notre code soit propre et facile à entretenir, il est crucial d'éviter que les classes se retrouvent avec trop de responsabilités. En mettant en pratique des principes de bonne organisation, on peut éviter le fardeau des classes.

Graphe IA et justification



À première vue, on constate que les composants ne s'écartent pas de manière significative de la droite de la Main Sequence. Cela montre une répartition équilibrée entre l'abstraction et la stabilité dans le système, que ce qui est un bon signe pout la maintenabilité et l'évolutivité.

En observant de plus près, nous constatons que les classes telles que CompagnieTransport, Emplacement, InfoClient, et Siege InfrastructureTransport se distinguent en tant que fondations abstraites solides. Cela indique une bonne pratique de conception, où les abstractions sont stables et moins susceptibles d'être changées fréquemment, ce qui est un signe de bonne architecture logicielle. Ces éléments sont bien intégrés dans le système, servant leurs rôles respectifs de manière assez efficace.

InfoClient et SectionTrainAvionPaquebot semblent équilibrés, alignés près de la ligne de la Main Sequence, un bon équilibre entre l'abstraction et la stabilité. Cela peut indiquer une assez bonne conception de ces classes, où ces classes sont suffisamment flexibles pour être étendues ou modifiées, mais aussi suffisamment stables pour ne pas entraîner de changements fréquents dans le reste du système.

En ce qui concerne les composants comme Voyage, Paiement, RechercheVoyage et Reservation, situés plus bas sur l'axe de l'abstraction, leur position plus proche de la zone d'instabilité indique qu'ils sont fonctionnels et orientés vers des tâches spécifiques. Ces classes sont des parties du système qui interagissent fréquemment avec l'utilisateur ou des systèmes externes, ce qui explique leur position.

Enfin, la classe Systeme semble être très instable et concrète, ce qui pourrait indiquer son rôle central dans le système avec une grande dépendance à d'autres composants. La place qu'occupe Systeme montre bien qu'il doit souvent changer pour et s'ajuster quand les besoins du système évoluent.

Dans l'ensemble, le diagramme montre que la plupart des composants sont dans ou proches de la zone idéale, avec une attention particulière portée à l'abstraction et à la stabilité. Les quelques composants situés en dehors de cette zone jouent des rôles différents qui exigent une manière de conception plus complexe.

Justification de l'application des principes de conception

Single Responsibility Principle (SRP) : Nous avons procédé à l'intégration de contrôleurs pour le volet administratif. De nouvelles classes de contrôleurs ont été développées pour gérer les classes suivantes : Compagnie, Voyage, Infrastructure, Transport, Paiement et Réservation. Ces contrôleurs encapsulent les méthodes et attributs nécessaires à la gestion administrative. Liskov Substitution Principle (LSP) : Les classes Voyage, CompagnieTransport, InfrastructureTransport, Section et Emplacement respectent ce principe. Toutes les sous-classes héritant de ces classes

mères maintiennent l'intégrité des attributs et méthodes, sans les altérer. Par exemple, les sous-classes SectionAvion, SectionTrain et SectionPaquebot sont conformes à la classe Section car ce sont bien des Sections et leurs méthodes et attributs match avec les ceux de la classe mère.