SÉANCE 1

Janssen Ouldsliman Troadec

Janssen
Ouldsliman
Troadec

Haute Ecole de la Province de Liège

2/10/2023

Cas 1:

Vous intégrez une équipe de développement qui développe un logiciel de diagnostiques pour téléphones. Le logiciel est destiné à être utilisé par des centres de réparation. Dans la base de code du logiciel, il existe une interface Charger qui représente un chargeur. La classe StandardCharger implémente cette interface et permet le chargement de la plupart des téléphones. Vous disposez également d'une interface Chargeable et d'un type de téléphone, SimplePhone, qui implémente cette interface.

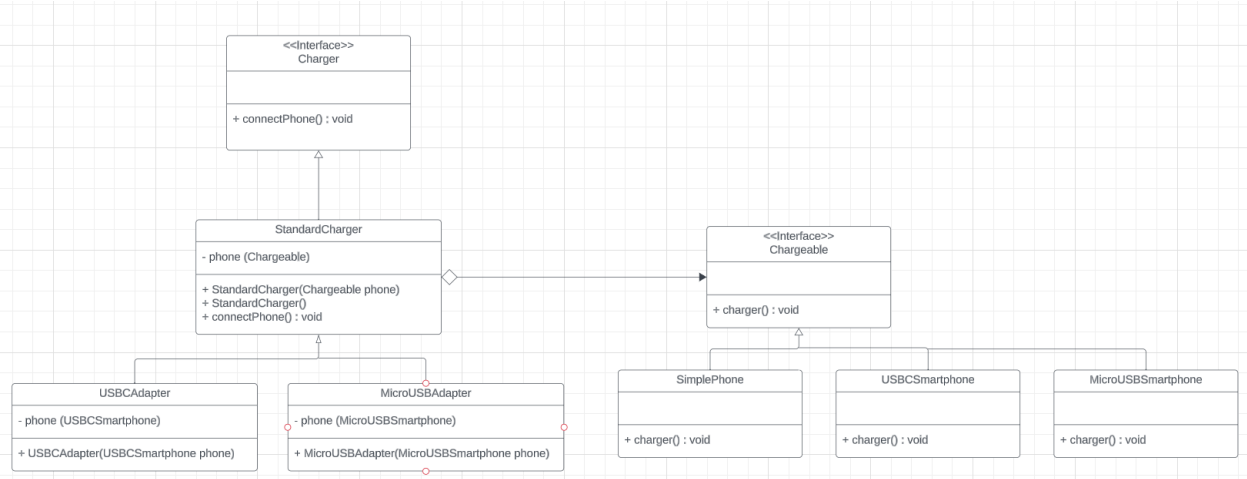
Comment connecter des smartphones qui ont des types de ports de charge différents (USB-C, Micro USB, …) au chargeur standard ? Envisagez plusieurs pistes de solutions, déterminez celle qui vous paraît la plus adéquate. Codez la solution et représentez-la avec un diagramme de classes.

Réponse :

La classe StandardCharger doit savoir manipuler plusieurs types de classes Chargeables différentes et il doit toujours être possible de rajouter des nouveaux types de classes Chargeables sans changer la classe StandardCharger. Cependant, chaque type d'appareille Chargable est très différent l'un de l'autre et a besoin de leur propre manière d'être manipuler. Si un type d'appareil Chargable est implémente dans la classe StandardCharger, alors cette classe devin incompatible avec tous les autres types d'appareils Chargeables. Il est donc nécessaire de construire des classes 'Adaptateur' qui hérite de StandartCharger mais peuvent facilement manipuler une classe Chargeable sans limiter d'autre classes de faire de même avec d'autre classe Chargeable. Avec ces classes 'Adapteurs', il est possible de simplement rajouter de nouveaux classes Chargeable bien différentes l'une de l'autre.

Le parton adaptateur n'est pas parfait car il demande un nouvel adaptateur pour chaque nouveau type de classe Chargeable ce qui peut augmenter la complexité du projet. Cependant cette complexité est nécessaire car comme expliqué plus haut, il est impossible de simplement tous implémenté dans la classe StandardCharger.

Il est possible d'utiliser le patron bridge à la place du patron adapteur mais cela aurait limiter la variabilité des classes Chargeables.



Cas 2 :

Le thème de ce deuxième cas concerne également l'univers de la téléphonie. Imaginez que vous devez concevoir des interfaces et des classes pour représenter des téléphones. Comment pourriez-vous structurer le code sachant qu'il existe des téléphones simples, des smartphones mais aussi que pour chacun de ces types de téléphones il existe plusieurs fabricants ? On imagine qu'un téléphone simple permet uniquement de passer des appels et envoyer des messages texte alors qu'un smartphone permet en plus de surfer sur internet et déprendre des photos.

Ecrire le code correspondant à la réalité ici décrite. En fonction de la structure choisie, justifiez ce qui

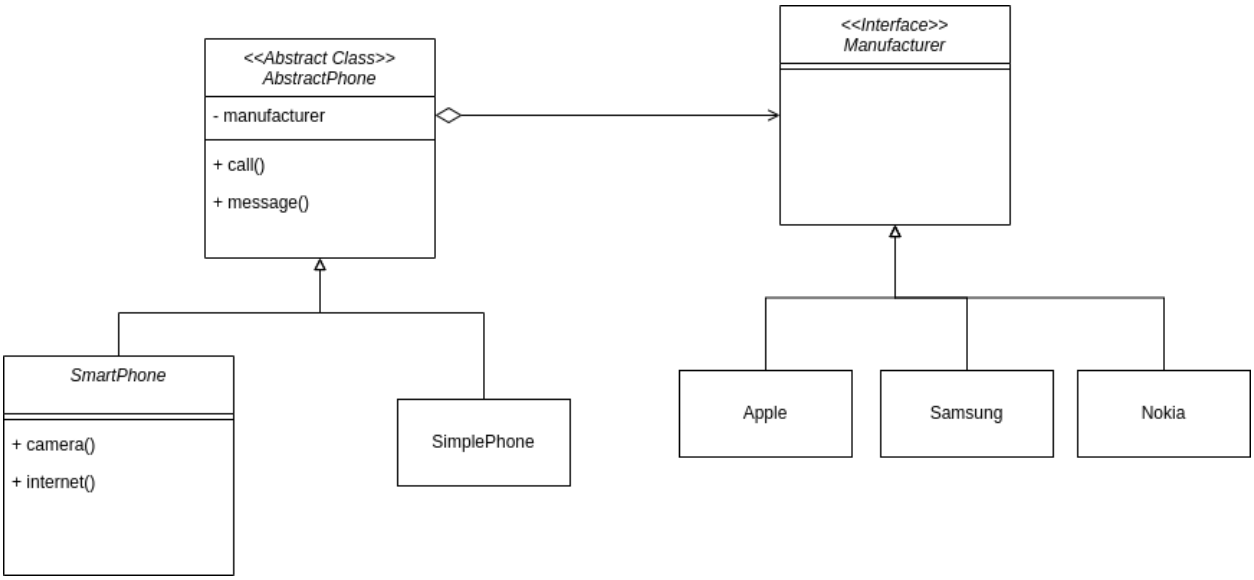
Pourra varier facilement dans votre solution et illustrez ce point avec un « main ».

Réponse :

Pour représenter des téléphones une class mère AbstractPhone à premièrement été créer. Cette class contient 2 fonctions call() et message() qui sont des comportement essentielle pour chaque téléphone. Cette class est abstraite et est seulement utile pour construire d'autre type de téléphone au-dessus. Les deux class SimplePhone et Smartphone hérite de la classe AbstracPhone. Cela permet de partager de la fonctionnalité entre chaque type de téléphone. Vue qu'un smartphone est plus complexe qu'un téléphone classique, la class Smartphone contient aussi les fonctions internet() et camera() que SimplePhone n'a pas.

Pour ajouter l'idée de manufacturer on utilise un bridge. La classe AbstractPhone à une variable membre manufacturer qui prend toutes les class qui implémente la class manufacturer. Ceci permet de limiter le nombre de class car on ne doit pas faire une classe pour chaque combinaison de manufactureur et de type de téléphone.

La création de la classe AbstractPhone ou la remplacé simplement par SimplePhone peut être débattus. On a décidé de créer cette class car elle nous donnait plus de flexibilité dans ce que SimplePhoe pouvait faire.



Cas 3 :

Pour ce troisième cas, le thème se rapporte à un logiciel de configuration pour l'achat de voitures. Le configurateur est destiné à être installé chez différents concessionnaires. Un concessionnaire représente une et une seule marque et propose en général des voitures familiales mais aussi des voitures plus sportives. Les voitures familiales permettent d'installer un siège enfant et de verrouiller automatiquement ou non les portières arrière. Sur une voiture sportive, la carrosserie est équipée d'un aileron et il est possible d'activer un turbo. Un concessionnaire ne peut évidemment pas proposer des voitures familiales d'une marque et des voitures sportives d'une autre marque !

Comment écrire le « main » du configurateur pour que puisse être créé des voitures mais aussi qu'il ne dépende (presque) pas d'une marque de voitures en particulier ?Comment empêcher la création de voitures familiales d'une marque et sportives d'une autre marque au sein de l'application ?

Réponse :

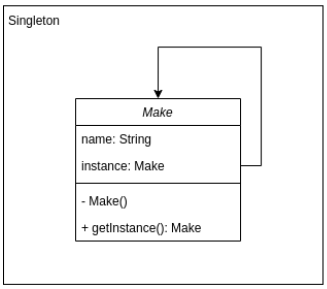
Pour assurer que le garage ne vend aucune voiture de différentes marques, on transforme la classe Make en singleton qui ne sera instancié qu'une seul fois au démarrage et ne sera jamais réinstancié. Cela assure que la marque ne sera toujours correcte pour toutes les voitures. Deux class FamilyCarFactory et SportCarFamily sont créé pour instancier différente voiture de sport ou de famille. Ces deux « factory » hérites d'une class abstraite BaseFactory leur donne la méthode createCar. Cette méthode crée une voiture basée sur le type de la « factory ». Cela est l'application du patron methode factory.

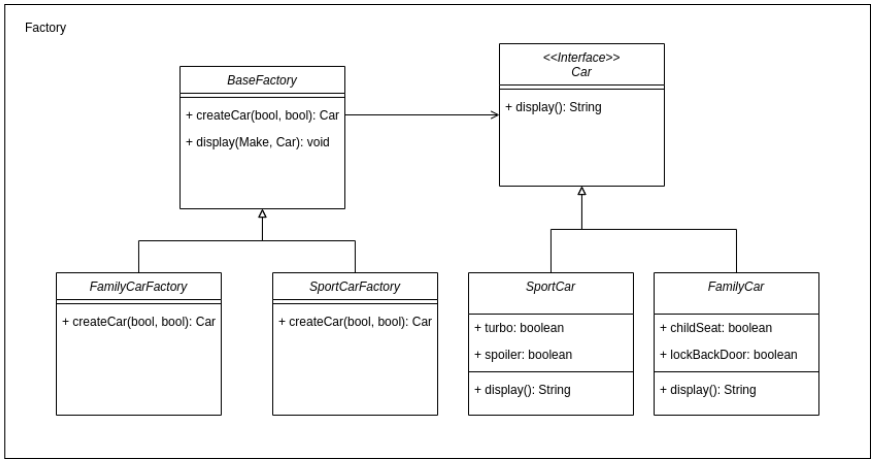
On aurait pu appliquer le patron builder à la place de methode factory qui nous aurait permis plus de flexibilité avec la sélection des différent composant de la voiture mais se patron n'aurait pas pus

permettre la division des voitures en voitures de sport ou voitures familiales. Vue que chaque voiture comporte un nombre relativement petit de composant cela est acceptable.

Il faut faire attention que le patron singleton sera plus difficile à tester dans le cas d'un unit test et devra avoir c'est fonction en mode « sync » si on utilise plusieurs threads dans notre application.

L'application du patron abstract factory est une autre solution possible à ce problème. Cependant, il n'a pas été appliquer car son application aurait demandé la création de différentes class pour chaque marque. Cela est un problème car chaque garage aurait des classes de marque qui lui serait inutile. Ce problème pourrait être remédier avec un wizard d'installation ou avec une version du programme pour chaque marque mais cela semble augmenté largement la complexité du projet. Si un besoin de rajouté beaucoup de fonctionnalités à la classe Make devin nécessaire, il sauras alors peut être plus judicieux d'utilisé le parton abstract factory.





Cas 4 :

Vous développez les composants de base pour créer une suite de logiciels capable de gérer différents types de documents. Dans le code, vous avez défini :

• Document : classe abstraite commune à tous les documents ;

• Application : class abstraite de base pour tout logiciel de la suite.

Un logiciel de votre suite, PDFEditor, permet de gérer des documents Pdf tandis que SimpleTextEditor, permet la manipulation de documents ASCII. Dans le futur, il est prévu de gérer d'autres types de documents. Vous souhaitez néanmoins déjà implémenter les fonctionnalités de création de documents au sein de la classe Application.

Comment procéder pour permettre à Application de créer des documents dont elle ignore la classe

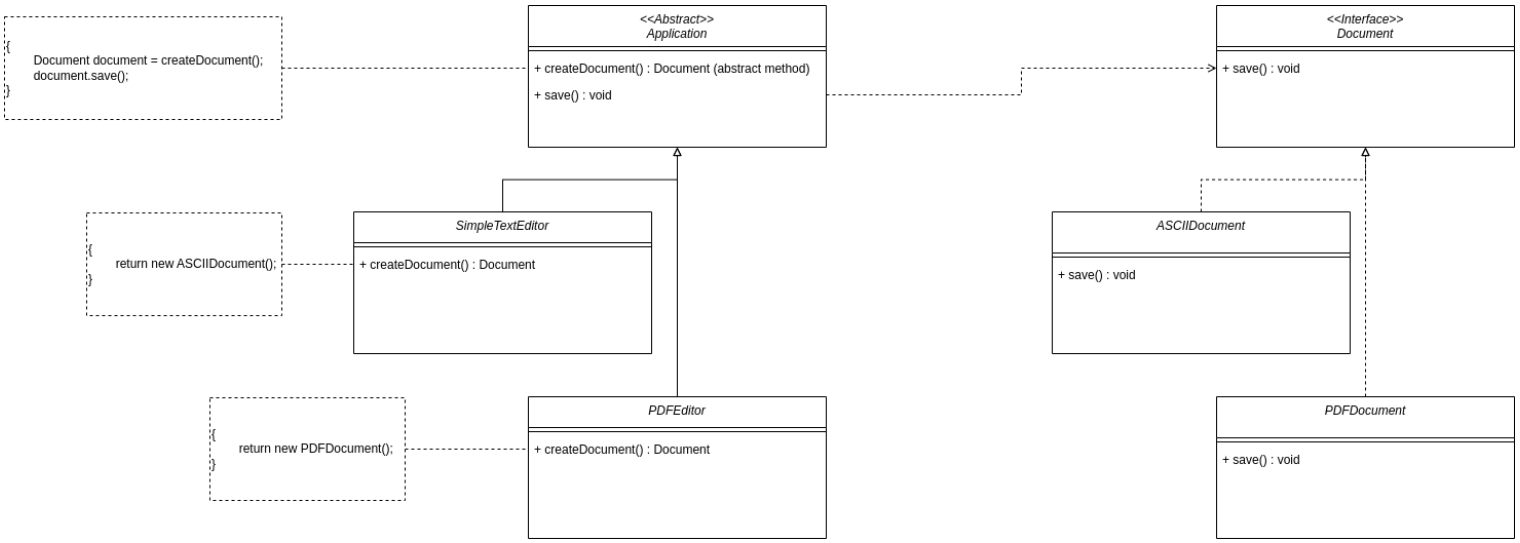
concrète ?

Réponse :

Pour donner la capacité à la class abstraite Application de créer des documents de différent type sans devoir connaître ces différents types, plusieurs classe 'éditeurs' ont été. La classe PDFEditeur hérite des fonctionnalités de la classe Application mais est spécialisé pour la création d'objet PDFDocument. Les

classes SimpleTextEditor et ASCIIDoument fonctionne de la même manière. Pour créer un document voulus il suffit de simplement appeler la fonction surcharger CreatDocument qui est implémenté différemment dans chaque éditeur pour chaque type de document. Le patron fonction factory est alors appliquer mais à la place de « factory », il y a des « éditeurs ».

Ce patron est bien car il permet une bonne séparation entre la class mère Application et les différents types de documents. Cependant, ce patron n'est pas parfait. S'il faut rajouter d'autre type de document, il faut qu'a chaque fois non-seulement créer le document en lui-même mais aussi son éditeur. Cela peu légèrement augmenté la complexité du système.



Contenu