Université de Lomé
CENTRE INFORMATIQUE ET DE CALCUL

utbm
université de technologie Belfort-Montbéliard

MITST03

LES DESIGN PATTERNS ET LEURS RÔLES

02 Mars 2022

Membres du groupe 1

AMOUZOU Kokou Benjamin

OFFRIDAM Kossi Jean-Claude

EDJAMTOLI Abidé Céline

Chargé du cours

Dr. APEKE Séna

Table des matières

1- Introduction et histoire des design patterns

Les design patterns ou patrons de conception en français sont des solutions classiques à des problèmes récurrents en conception orientée objet. Lorsqu'une question revient plusieurs fois dans différents projets, une personne décide finalement de détailler la solution et de lui donner un nom. C'est souvent comme cela qu'un nouveau patron est créé.

Le concept de patron de conception a d'abord été décrit par Christopher Alexander dans le livre "A Pattern Language: Towns, Buildings, Construction" pour concevoir un environnement urbain. Les patrons de conception de ce livre décrivent la hauteur attendue des fenêtres, le nombre d'étages qu'un bâtiment devrait avoir, la taille des espaces verts d'un quartier, etc.

En 1994, Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (la bande des quatre, ou « Gang of Four » GoF), ont publié le livre "Design patterns: Elements of Reusable Object-Oriented Software". Ils ont examiné plus de 300 projets sur lesquels travaillaient d'autres développeurs et se sont rendus compte que les mêmes problèmes réapparaissent sans cesse.

Ils ont aussi remarqué que ces divers problèmes étaient résolus d'une façon globalement similaire. Ils ont classé les patterns en trois sections : Les patterns de création, de structure et de comportement.

Le livre recense 23 patrons de conceptions à partir desquels de nombreux autres patrons sont nés tout en s'étendant à d'autres paradigmes de programmation.

2- Qu'est-ce qu'un design pattern

« Un design pattern est un modèle de conception qui permet de faire la description d'un problème qui se produit fréquemment et l'architecture de la solution à ce problème de telle sorte que la solution soit réutilisable plusieurs fois. »

« Les patterns de conception sont des solutions de niveaux classe et méthode à des problèmes courants principalement dans le développement orienté objet »

3- Pourquoi apprendre les design patterns

- Les patrons de conception sont une boîte à outils de solutions fiables et éprouvées utilisées en réponse à des problèmes classiques de la conception de logiciels ;

- Les patrons de conception définissent un langage commun que vous et vos collègues pouvez utiliser pour mieux communiquer ;
- Ils permettent d'écrire du code fermé à la modification et ouvert à l'extension : code réutilisable et extensible ;
- Le programme résultant est donc facilement maintenable et évolutif.

4- Les grandes familles de design patterns

Pour chaque grande famille de design patterns nous étudierons trois exemples.

4-1- Les design patterns de création

Les patrons de création fournissent des mécanismes de création d'objets qui augmentent la flexibilité et la réutilisation du code. Ils sont au total cinq d'après la classification GoF : Fabrique, Fabrique Abstraite, Monteur, Prototype et Singleton.

4-1-1- Pattern Singleton

Intention

Le pattern singleton est un patron de conception de création qui garantit que l'instance d'une classe n'existe qu'en un seul exemplaire, tout en fournissant un point d'accès global à cette instance.

Problème

Le singleton règle deux problèmes à la fois :

1. Il garantit l'unicité d'une instance pour une classe lorsque l'on veut contrôler l'accès à une ressource partagée (une base de données ou un fichier par exemple)
2. Il fournit un point d'accès global à cette instance : le singleton vous permet d'accéder à l'objet n'importe où dans le programme, telle une variable globale. Cependant, il protège son instance et l'empêche d'être modifiée.

Un singleton résout nécessairement les deux problèmes.

Solution

Toute mise en place d'un singleton est constituée des deux étapes suivantes :

- Rendre le constructeur par défaut privé afin d'empêcher les autres objets d'utiliser l'opérateur new avec la classe du singleton ;
- Mettre en place une méthode de création statique qui appelle le constructeur privé pour créer une seule fois un objet sauvegardé dans un attribut statique de la classe du singleton.

Quand utiliser le singleton ?

- Lorsque l'une de vos classes ne doit fournir qu'une seule instance à tous ses clients ;
- Lorsque vous voulez un contrôle absolu sur vos variables globales.

NB : Dans un environnement multi-thread, il faut poser un verrou lors de la création du singleton pour empêcher un autre thread de le créer simultanément.

4-1-2- Pattern Fabrique (Factory method)

Intention

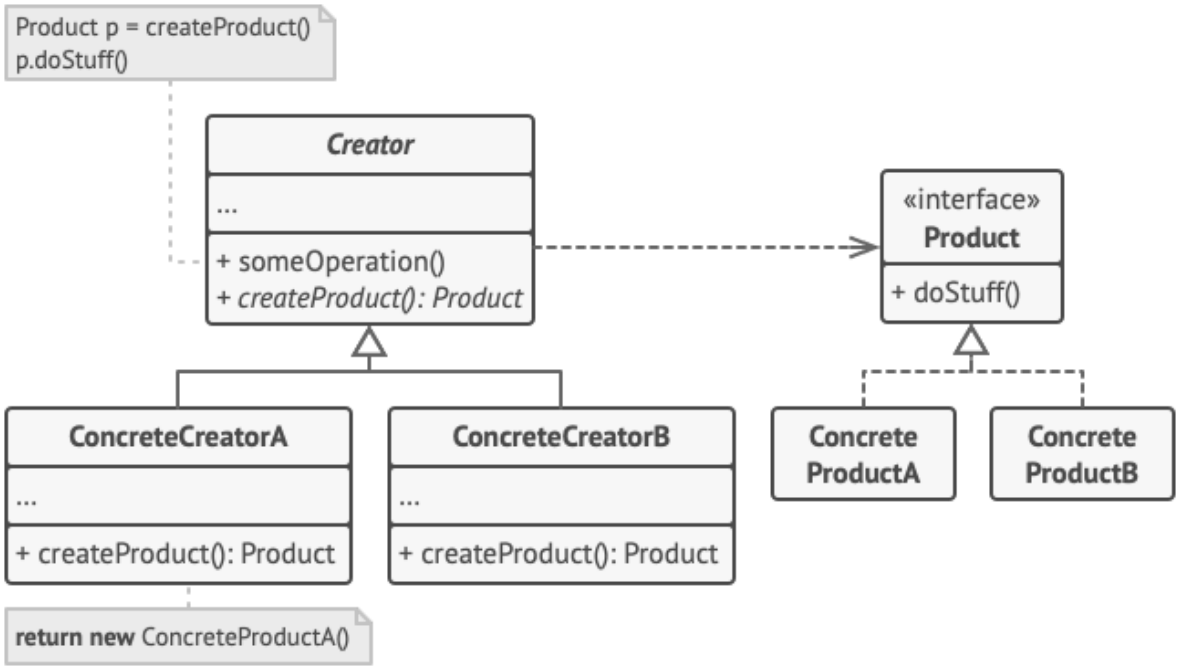
Le pattern fabrique est un patron de conception de création qui définit une interface pour créer des objets dans une classe mère, mais délègue le choix des types d'objets à créer aux sous-classes.

Problème

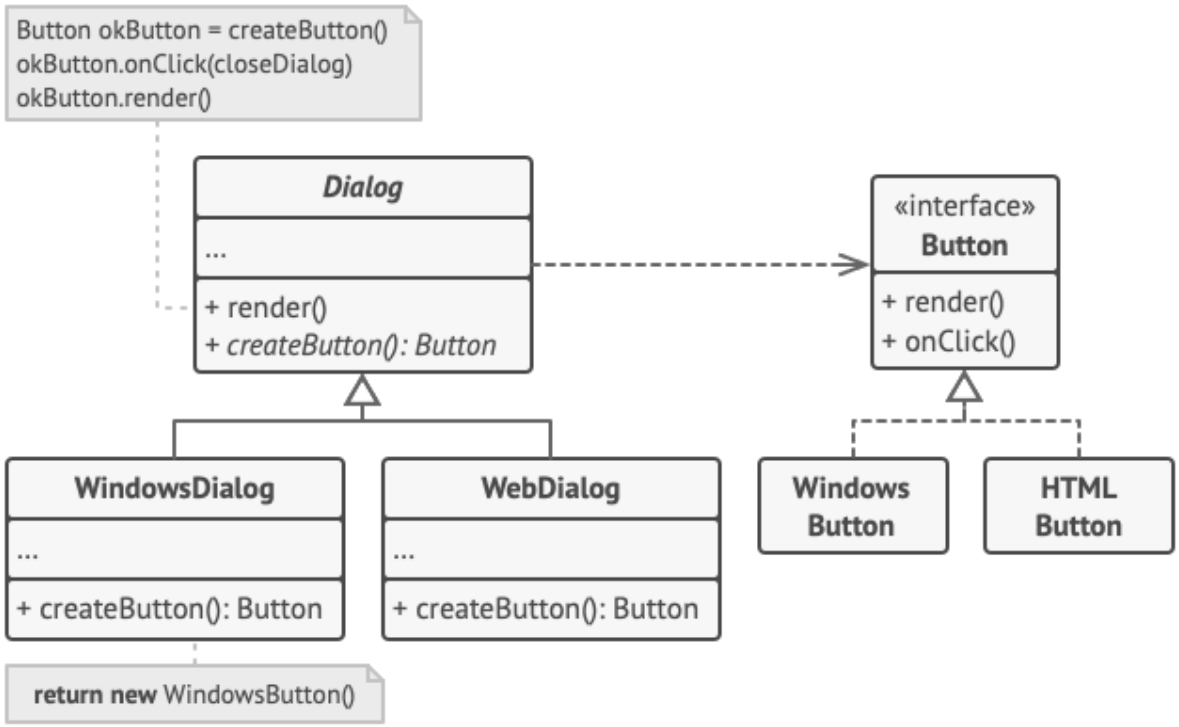
Vous avez créé une application de gestion logistique qui ne gère que les camions dans sa version première. Dans une version suivante, vous souhaitez ajouter la gestion logistique pour les bateaux. Comment le faire de la manière la plus flexible possible pour avoir du code propre en sachant que le code initial est fortement lié aux camions ?

Solution

Vous créez une interface commune à vos différents moyens de transports : camions et bateaux. Vous créez ensuite une classe créateur disposant d'une méthode factory renvoyant un objet implémentant cette interface. Vous créez enfin pour les camions et les bateaux des classes filles de la classe créateur qui redéfinissent la méthode factory de la classe mère.



L'exemple montre comment la fabrique peut être utilisée pour créer des éléments d'une UI (interface utilisateur) multiplateforme sans coupler le code client aux classes concrètes de l'UI.



Quand utiliser le patron Fabrique ?

- Utilisez la fabrique si vous ne connaissez pas à l'avance les types et dépendances précis des objets que vous allez utiliser dans votre code (La

fabrique effectue une séparation entre le code du constructeur et le code qui utilise réellement le produit) ;

- Utilisez la fabrique si vous voulez mettre à disposition une librairie ou un Framework pour vos utilisateurs avec un moyen d'étendre ses composants internes ;
- Utilisez la fabrique lorsque vous voulez économiser des ressources système en réutilisant des objets au lieu d'en construire de nouveaux ;

4-1-3- Pattern Prototype

Intention

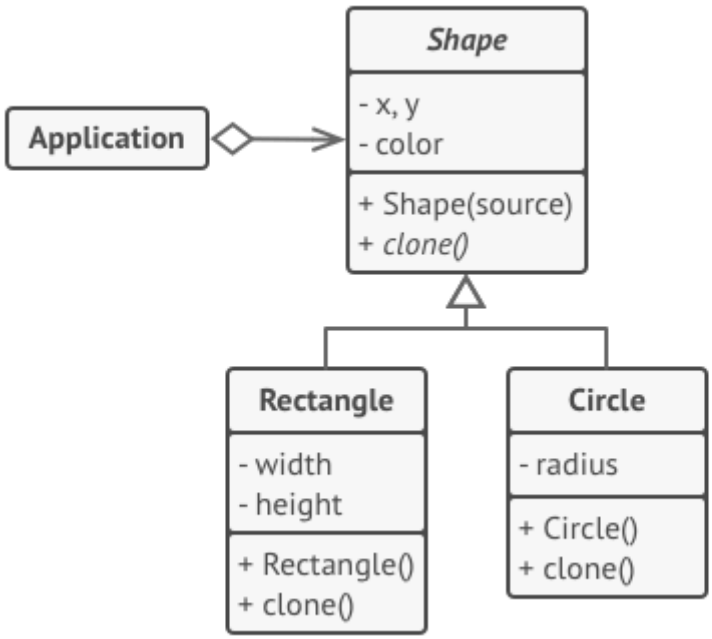
Le pattern prototype est un patron de conception qui crée de nouveaux objets à partir d'objets existants sans rendre le code dépendant de leurs classes.

Problème

Vous souhaitez avoir une copie exacte d'un objet existant. Vous parcourez les attributs de l'objet existant, vous en créez des copies que vous mettez dans un nouvel objet. Le hic, ce sont les attributs privés non accessibles et la possibilité de ne pas connaître la classe de l'objet.

Solution

Vous créez une interface contenant une méthode clone() implémentée par tous les objets devant être clonés.



Quand utiliser Prototype ?

- Utilisez le prototype si vous ne voulez pas que votre code dépende des classes concrètes des objets que vous clonez.

4-2- Les design patterns structurels

Les patterns de structure correspondent à des manières d'organiser les classes ou les objets de façon à ce qu'ils soient faciles à utiliser. Il existe deux types de patterns de structure : ceux qui organisent les classes et ceux qui organisent les objets. Ils sont sept en tout : Adapteur, Pont, Composite, Décorateur, Façade, Poids Mouche et Procuration.

4-2-1- Pattern Adapter

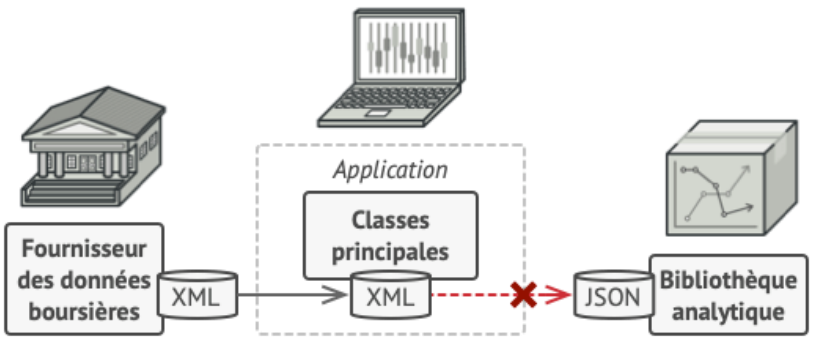
Intention

L'Adaptateur est un patron de conception structurel qui permet de faire collaborer des objets ayant des interfaces normalement incompatibles.

Problème

Vous créez une application de surveillance du marché boursier. L'application télécharge des données boursières depuis diverses sources au format XML et affiche ensuite des graphiques à l'utilisateur.

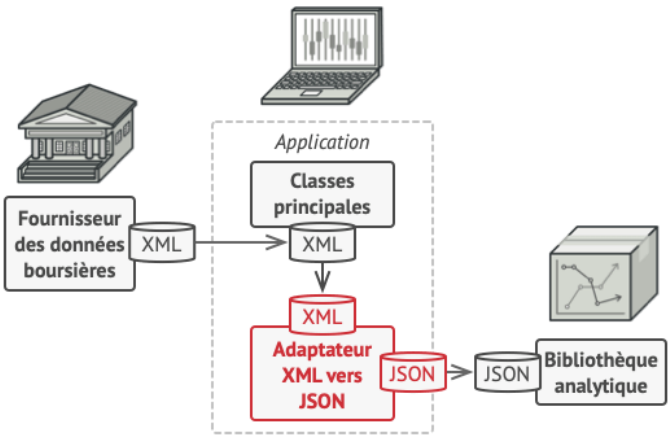
Vous décidez plus tard d'améliorer l'application en intégrant une librairie d'analyse externe. Malheureusement, cette librairie ne fonctionne qu'avec des données au format JSON.



Vous pourriez modifier la librairie afin qu'elle accepte du XML, mais vous risquez de faire planter d'autres parties de code qui utilisent déjà cette librairie. Ou alors, vous n'avez pas accès au code source de la librairie, rendant la tâche impossible.

Solution

Vous créez un adaptateur. C'est un objet spécial qui convertit l'interface d'un objet afin qu'un autre objet puisse le comprendre. L'adaptateur aide également différentes interfaces à collaborer.

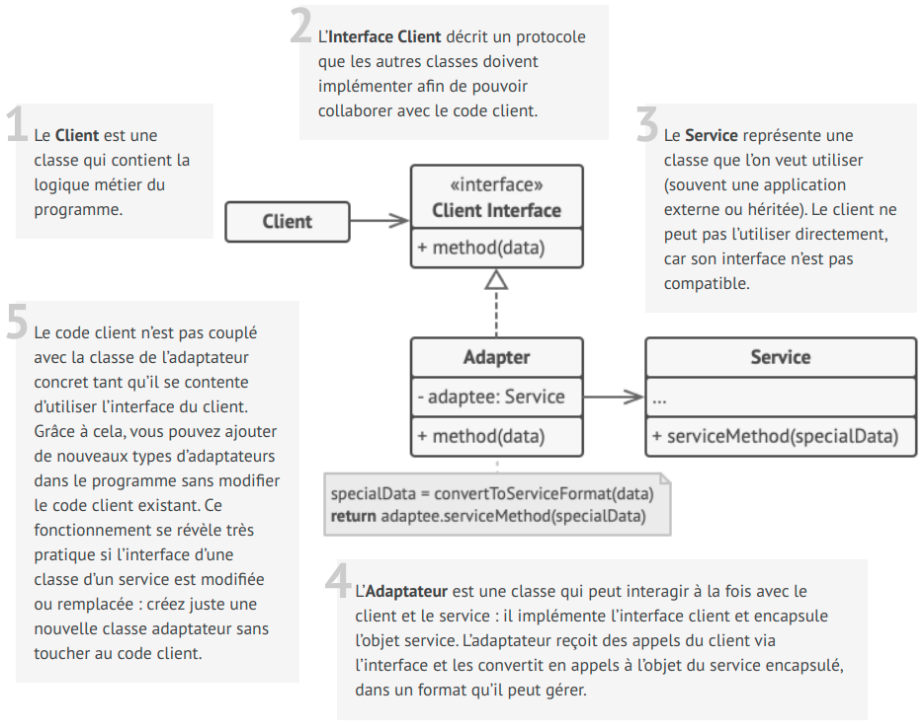


Dans l'exemple du marché boursier, vous pouvez créer des adaptateurs XML vers JSON pour chaque classe de la librairie. Vous ajustez votre code pour communiquer avec la librairie à l'aide de ces adaptateurs. Lorsqu'un adaptateur reçoit un appel, il convertit les données XML en une structure JSON. Il renvoie ensuite l'appel à la méthode appropriée dans un objet d'analyse encapsulé.

Structure

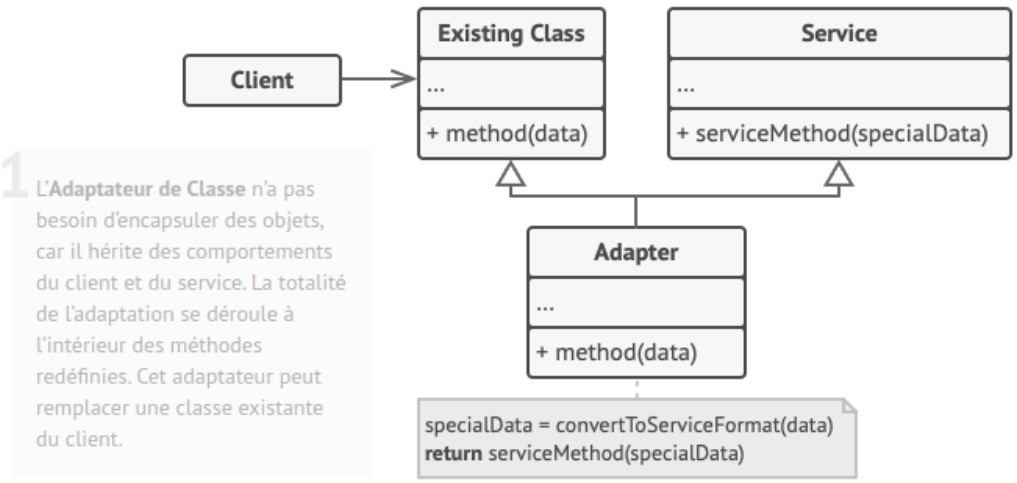
Adaptateur d'objets

Cette implémentation a recours au principe de composition : l'adaptateur implémente l'interface d'un objet et en encapsule un autre.



Adaptateur de classe

Cette implémentation utilise l'héritage : l'adaptateur hérite de l'interface des deux objets en même temps, une approche uniquement possible pour les langages de programmation gérant l'héritage multiple, comme le C++.



Quand utiliser le pattern adapter ?

- Utilisez l'adaptateur de classe si vous avez besoin d'une classe existante, mais que son interface est incompatible avec votre code ;
- Mettez en place l'adaptateur si vous désirez réutiliser plusieurs sous-classes existantes à qui il manque des fonctionnalités communes qui ne peuvent pas être remontées dans la classe mère.

4-2-2- Pattern Composite

Intention

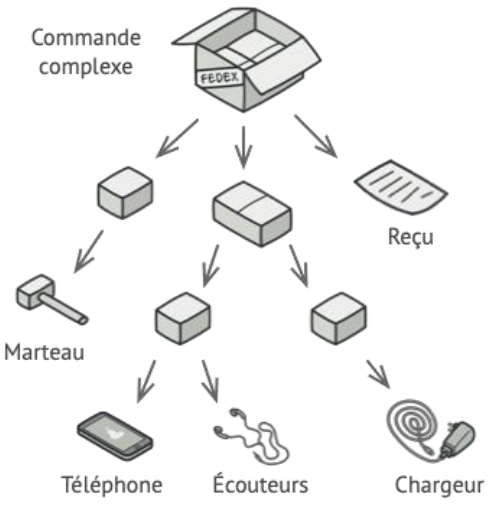
Le pattern composite est un patron de conception structurel qui permet d'agencer les objets dans des arborescences afin de pouvoir traiter celles-ci comme des objets individuels.

Problème

L'utilisation de ce patron doit être réservée aux applications dont la structure principale peut être représentée sous la forme d'une arborescence.

Soient deux objets Produits et Boîtes. Une boîte peut contenir des produits et des boîtes plus petites. Ces petites boîtes peuvent également contenir des produits et d'autres boîtes encore plus petites, et ainsi de suite.

Vous décidez de mettre au point un système de commandes qui utilise ces classes. Les commandes peuvent être composées de produits simples sans emballages et d'autres boîtes remplies de produits. Comment allez-vous déterminer le coût total d'une telle commande ?



Vous pouvez tenter l'approche directe : déballer toutes les boîtes, prendre chaque produit et faire la somme pour obtenir le total. Possible dans le monde réel mais pas évident dans un programme. Il faut connaître à l'avance la classe des Produits et des Boîtes que l'on parcourt, le niveau d'imbrication des boîtes ainsi que d'autres détails. Tout ceci rend l'approche directe compliquée voire impossible.

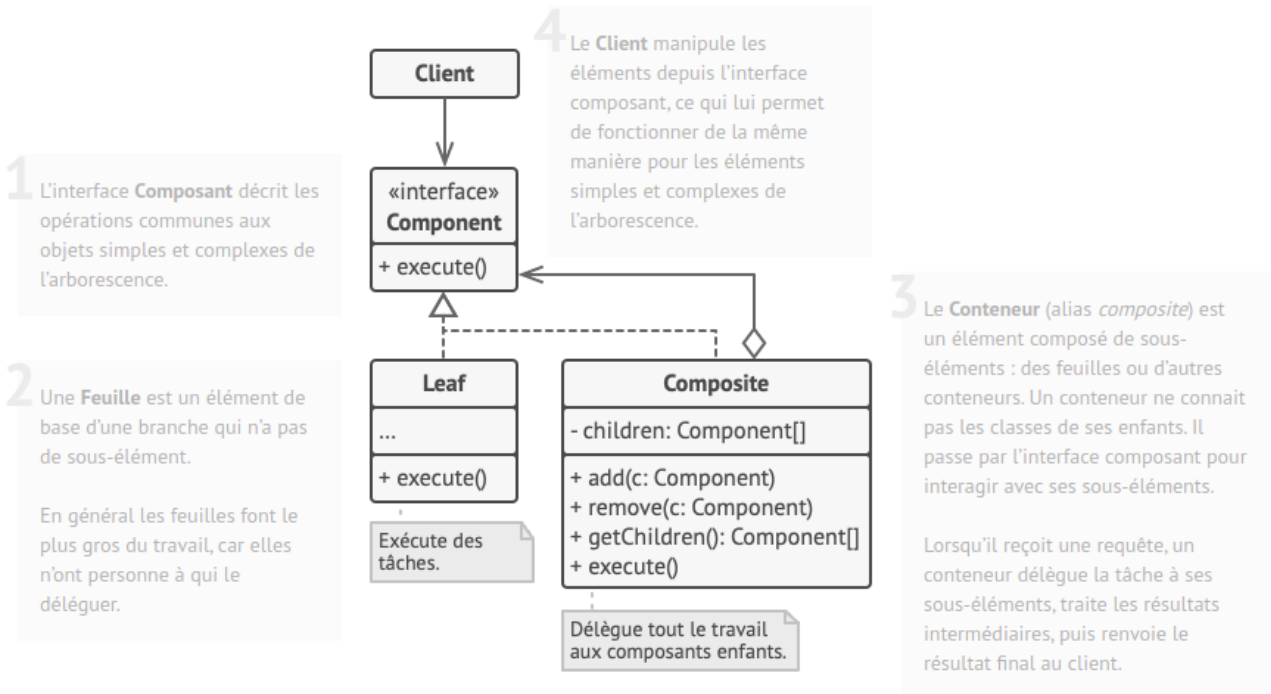
Solution

Le patron de conception composite vous propose de manipuler les Produits et les Boîtes à l'aide d'une interface qui déclare une méthode de calcul du prix total.

Pour un produit, on retourne son prix. Pour une boîte, on parcourt chacun de ses objets, on leur demande leur prix, puis on retourne un total pour la boîte. Si l'un de ces objets est une boîte plus petite, cette dernière va aussi parcourir son propre contenu et ainsi de suite, jusqu'à ce que tous les prix aient été calculés. Une boîte peut même ajouter des frais supplémentaires, comme le prix de l'emballage.

Vous n'avez même pas besoin de connaître la classe concrète des objets de l'arborescence. Vous les manipulez de la même manière grâce à une interface commune. Lorsque vous faites appel à une méthode, les objets s'occupent de faire transiter la requête en descendant vers les feuilles de l'arbre.

Structure



Quand utilisez le pattern composite ?

- Utilisez le composite si vous devez gérer une structure d'objets qui ressemble à une arborescence ;
- Utilisez ce patron si vous voulez que le client interagisse avec les éléments simples aussi bien que complexes de façon uniforme.

4-2-3- Pattern Decorator

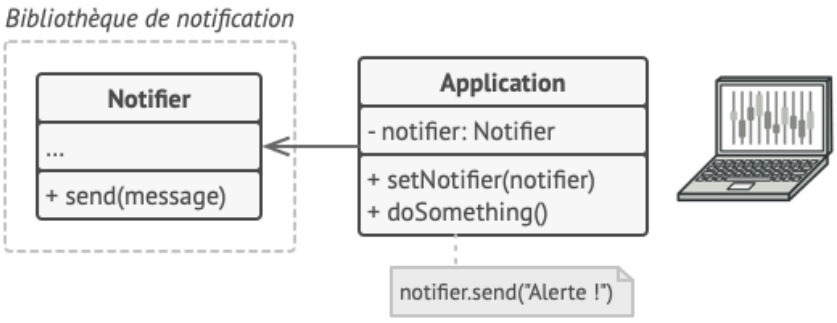
Intention

Décorateur est un patron de conception structurel qui permet d'affecter dynamiquement de nouveaux comportements à des objets en les plaçant dans des emballeurs qui implémentent ces comportements.

Problème

Imaginez que vous travaillez sur une librairie qui permet aux programmes d'envoyer des notifications à leurs utilisateurs lorsque des événements importants se produisent.

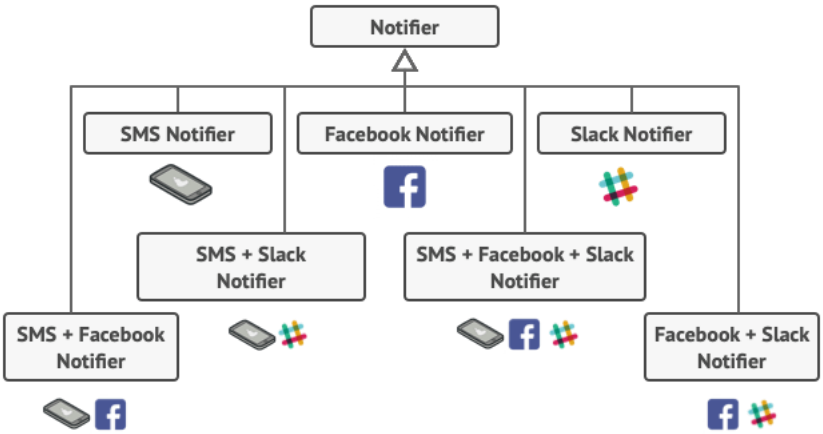
La version initiale de la librairie était basée sur la classe Notificateur qui n'avait que quelques attributs, un constructeur et une unique méthode envoyer. La méthode pouvait prendre un message en paramètre et l'envoyait à une liste d'e-mails à l'aide du constructeur du notificateur. Une application externe qui jouait le rôle du client devait créer et configurer l'objet notificateur une première fois, puis l'utiliser lorsqu'un événement important se produisait.



Plus tard les utilisateurs souhaiteraient recevoir des SMS, des messages sur Facebook ou sur Slack lorsque leurs applications rencontrent des problèmes critiques.

Vous avez étendu la classe Notificateur et ajouté des méthodes de notification supplémentaires dans de nouvelles sous-classes. Le client instancie la classe de

notification désirée et utilise cette instance pour toutes les autres notifications. Certains clients cependant souhaiteraient utiliser plusieurs types de notifications simultanément. Vous avez tenté de résoudre ce problème en créant des sous-classes spéciales qui combinent plusieurs méthodes de notification dans une seule classe. Mais il s'avère que cette approche va gonfler le code de la librairie et celui du client.

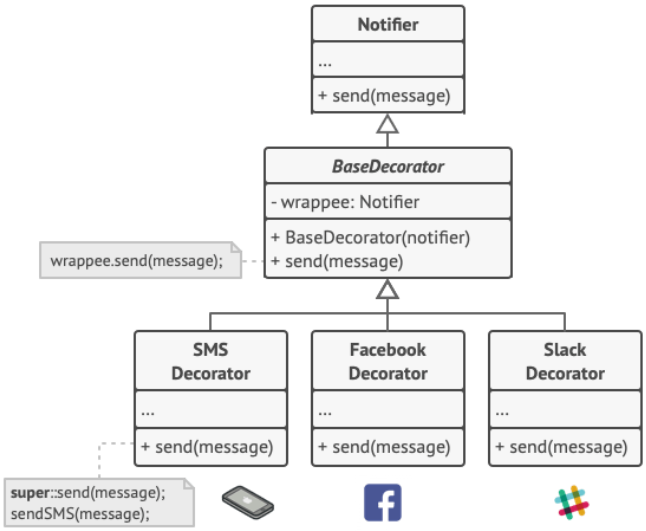


Solution

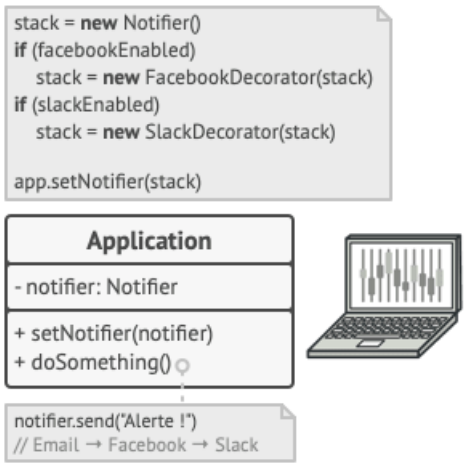
L'agrégation et la composition sont des principes clés dans le décorateur. Le décorateur est également appelé « emballeur » ou « empaqueteur ». Un emballeur est un objet qui peut être lié par un objet cible. L'emballeur possède le même ensemble de méthodes que la cible et lui délègue toutes les demandes qu'il reçoit. Il peut exécuter un traitement et modifier le résultat avant ou après avoir envoyé sa demande à la cible.

Un emballeur implémente la même interface que l'objet emballé. Du point de vue du client, ces objets sont identiques. L'attribut de la référence de l'emballeur doit pouvoir accueillir n'importe quel objet qui implémente cette interface. Vous pouvez ainsi utiliser plusieurs emballeurs sur un seul objet et lui attribuer les comportements de plusieurs emballeurs en même temps.

Reprenons notre exemple et mettons-y une notification par e-mail dans la classe de base Notificateur, mais transformons toutes les autres méthodes de notification en décorateurs.



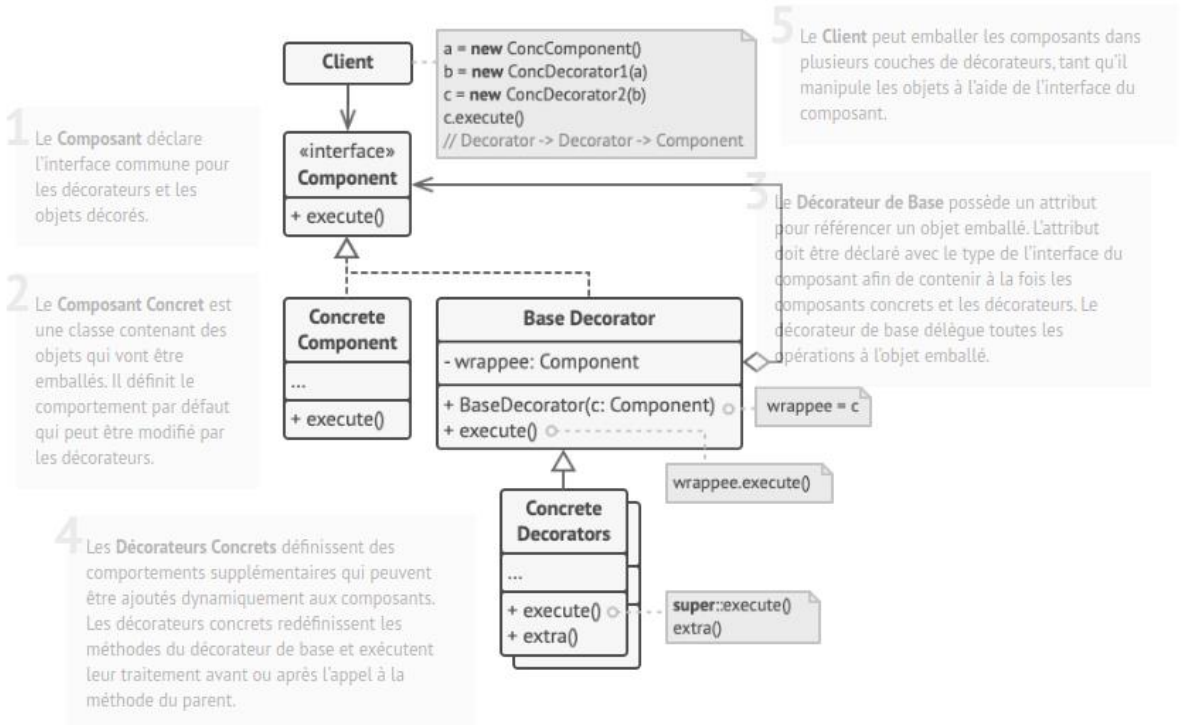
Le code client doit emballer un objet basique Notificateur pour le transformer en un ensemble de décorateurs adapté aux préférences du client. Les objets qui en résultent sont empilés.



Le client traite le dernier objet de la pile. Les décorateurs implémentent tous la même interface (le notificateur de base). Le reste du code client manipule indifféremment l'objet notificateur original et l'objet décoré.

Nous pouvons utiliser cette technique pour tous les autres comportements, comme la mise en page des messages ou la création de la liste des destinataires. Le client peut décorer l'objet avec des décorateurs personnalisés tant qu'ils suivent la même interface que les autres.

Structure



Quand utiliser le pattern décorateur ?

- Utilisez le décorateur si vous avez besoin d'ajouter des comportements supplémentaires au moment de l'exécution sans avoir à altérer le code source de ces objets ;
- Utilisez ce patron si l'héritage est impossible ou peu logique pour étendre le comportement d'un objet.

4-3- Les design patterns comportementaux

Ce sont des patrons qui s'occupent des algorithmes et de la répartition des responsabilités entre les objets. Ils sont onze au total : Chaîne de responsabilité, Commande, Itérateur, Médiateur, Mémento, Observateur, État, Stratégie, Patron de méthode et Visiteur.

4-3-1- Pattern Observateur

Intention

Le pattern observateur est un patron de conception comportemental qui permet de mettre en place un mécanisme de souscription pour envoyer des notifications à plusieurs objets, au sujet d'événements concernant les objets qu'ils observent.

Problème

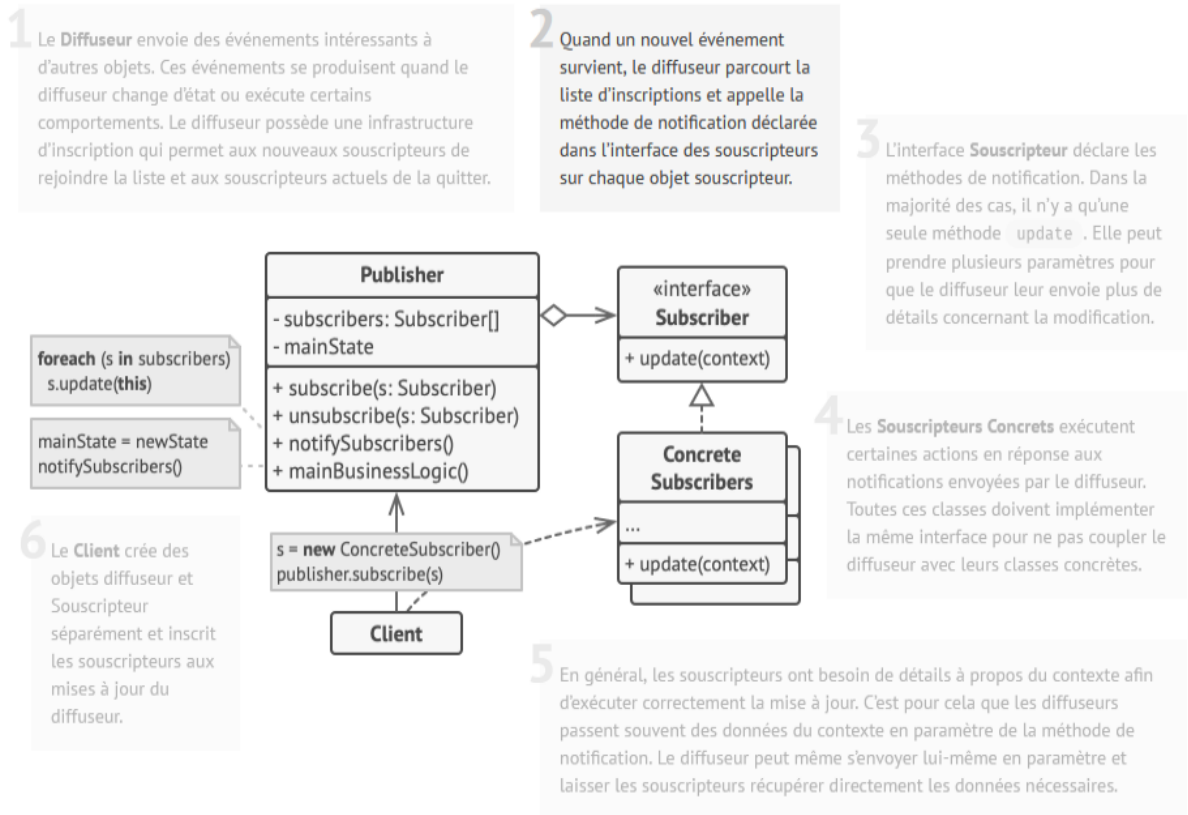
Un magasin souhaite informer régulièrement ses clients lors de l'arrivée de nouveaux articles. Le magasin peut envoyer à chaque fois des emails à tous ses clients. Il risquerait cependant d'agacer ceux qui ne sont pas intéressés par l'arrivée de nouveaux articles. Comment faire alors ?

Solution

Le patron de conception Observateur vous propose d'ajouter un mécanisme de souscription à la classe diffuseur pour permettre aux objets individuels de s'inscrire ou se désinscrire de ce diffuseur. L'objet suivi est appelé diffuseur et l'objet qui suit le diffuseur est appelé souscripteur.

Les applications peuvent comporter des dizaines de classes souscripteur différentes qui veulent être tenues au courant des événements qui affectent une même classe diffuseur. C'est pourquoi il est crucial que tous les souscripteurs implémentent la même interface et qu'elle soit le seul moyen utilisé par le diffuseur pour communiquer avec eux.

Structure



Quand utiliser le pattern observateur ?

- Utilisez le patron de conception Observateur quand des modifications de l'état d'un objet peuvent en impacter d'autres, et que l'ensemble des objets n'est pas connu à l'avance ou qu'il change dynamiquement ;
- Utilisez ce patron quand certains objets de votre application doivent en suivre d'autres, mais seulement pendant un certain temps ou dans des cas spécifiques

4-3-2- Pattern Stratégie

Intention

Le pattern stratégie est un patron de conception comportemental qui permet de définir une famille d'algorithmes, de les mettre dans des classes séparées et de rendre leurs objets interchangeables.

Problème

Vous avez créé une application de navigation pour les voyageurs occasionnels. Vous l'avez développé avec une superbe carte comme fonctionnalité principale, qui aide les utilisateurs à s'orienter rapidement dans n'importe quelle ville.

La fonctionnalité la plus demandée était la planification d'itinéraire. Un utilisateur devrait pouvoir entrer une adresse et le chemin le plus rapide pour arriver à destination s'afficherait sur la carte.

La première version de l'application ne pouvait tracer des itinéraires que sur les routes. Les automobilistes étaient comblés. Mais apparemment, certaines personnes préfèrent utiliser d'autres moyens de locomotion pendant leurs vacances. Vous avez ajouté la possibilité de créer des trajets à pied dans la version suivante. Juste après cela, vous avez ajouté la possibilité d'utiliser les transports en commun dans les itinéraires.

Mais tout ceci n'était que le début. Vous avez continué en adaptant l'application pour les cyclistes, et plus tard, ajouté la possibilité de construire les itinéraires en passant par les attractions touristiques de la ville.

Chaque fois que vous ajoutiez un nouvel algorithme pour tracer les itinéraires, la classe principale Navigateur doublait de taille. La moindre touche apportée aux algorithmes impactait la totalité de la classe, augmentant les chances de créer des bugs dans du code qui fonctionnait très bien.
Ajouter une nouvelle fonctionnalité vous demandait de modifier une classe énorme, créant des conflits dans le code produit par les autres développeurs.

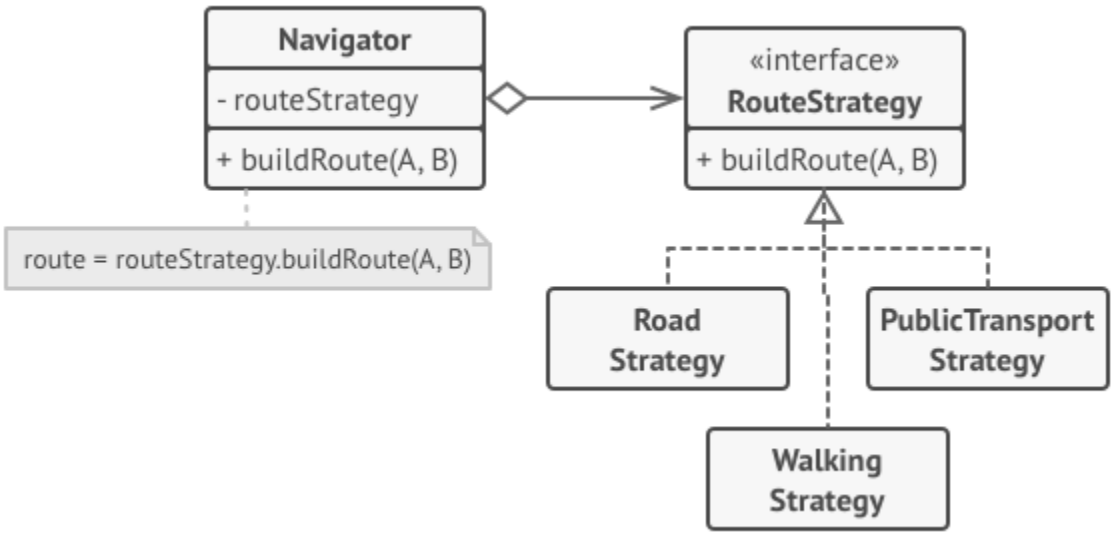
Solution
Le patron de conception stratégie vous propose de prendre une classe dotée d'un comportement spécifique mais qui l'exécute de différentes façons, et de décomposer ses algorithmes en classes séparées appelées stratégies.

La classe originale (le contexte) doit avoir un attribut qui garde une référence vers une des stratégies. Plutôt que de s'occuper de la tâche, le contexte la délègue à l'objet stratégie associé.

Le contexte n'a pas la responsabilité de la sélection de l'algorithme adapté, c'est le client qui lui envoie la stratégie. En fait, le contexte n'y connait pas grand-chose en stratégies, c'est l'interface générique qui lui permet de les utiliser. Elle n'expose qu'une seule méthode pour déclencher l'algorithme encapsulé à l'intérieur de la stratégie sélectionnée.

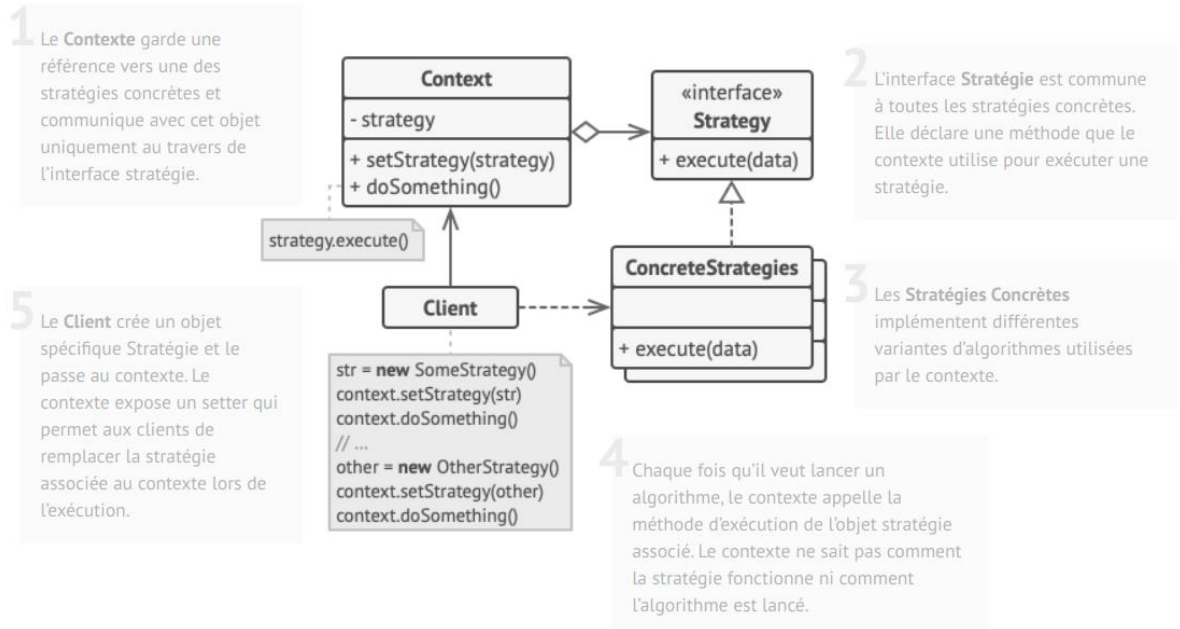
Le contexte devient indépendant des stratégies concrètes. Vous pouvez ainsi modifier des algorithmes ou en ajouter de nouveaux sans toucher au code du contexte ou aux autres stratégies.

Dans notre application de navigation, chaque algorithme d'itinéraire peut être extrait de sa propre classe avec une seule méthode tracer Itinéraire. La méthode accepte une origine et une destination, puis retourne une liste de points de passage.



Quand bien même les différentes classes itinéraires ne donneraient pas un résultat identique avec les mêmes paramètres, la classe navigateur principale ne se préoccupe pas de l'algorithme sélectionné, car sa fonction première est d'afficher les points de passage sur la carte. La classe navigateur possède une méthode pour changer la stratégie d'itinéraire active afin que ses clients (les boutons de l'interface utilisateur par exemple) puissent remplacer le comportement sélectionné par un autre.

Structure



Quand utiliser le pattern stratégie ?

- Utilisez le patron de conception stratégie si vous voulez avoir différentes variantes d'un algorithme à l'intérieur d'un objet à disposition, et pouvoir passer d'un algorithme à l'autre lors de l'exécution ;
- Utilisez la stratégie si vous avez beaucoup de classes dont la seule différence est leur façon d'exécuter un comportement ;
- Utilisez la stratégie pour isoler la logique métier d'une classe, de l'implémentation des algorithmes dont les détails ne sont pas forcément importants pour le contexte ;
- Utilisez ce patron si votre classe possède un gros bloc conditionnel qui choisit entre différentes variantes du même algorithme.

4-3-3- Pattern État

Intention

Le pattern état est un patron de conception comportemental qui permet de modifier le comportement d'un objet lorsque son état interne change. L'objet donne l'impression qu'il change de classe.

Problème

Le principe repose sur le fait qu'un programme possède un nombre fini d'états. Le programme se comporte différemment selon son état et peut en changer instantanément. En revanche, selon l'état dans lequel il se trouve, certains états ne lui sont pas accessibles. Ces règles de changement d'état sont appelées transitions. Elles sont également finies et prédéterminées.

Vous pouvez appliquer cette approche aux objets. Imaginons une classe Document. Un document peut être dans l'un des trois états suivants : brouillon (draft), modération et publié. La méthode publier du document fonctionne un peu différemment en fonction de son état :

- Dans brouillon, elle passe le document en modération ;
- Dans modération, elle rend le document public si l'utilisateur actuel est un administrateur ;
- Dans publié, elle ne fait rien du tout.

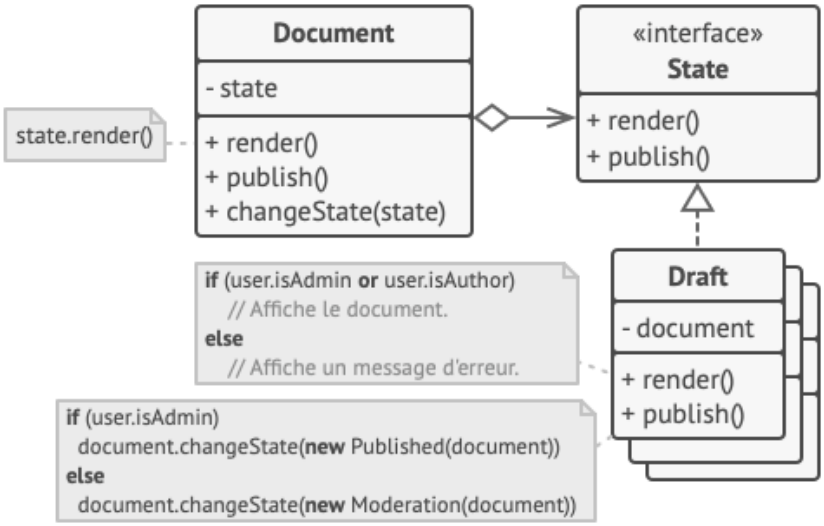
Lorsque l'on commence à ajouter de plus en plus d'états et de comportements à la classe Document, le code devient difficile à maintenir, car tout changement dans la

logique de transition demande de modifier les états conditionnels dans chaque méthode.
Plus le projet évolue et plus Il est difficile de prédire tous les états et transitions possibles lors de la phase de conception.

Solution

Le patron de conception état propose de créer de nouvelles classes pour tous les états possibles d'un objet et d'extraire les comportements liés aux états dans ces classes.

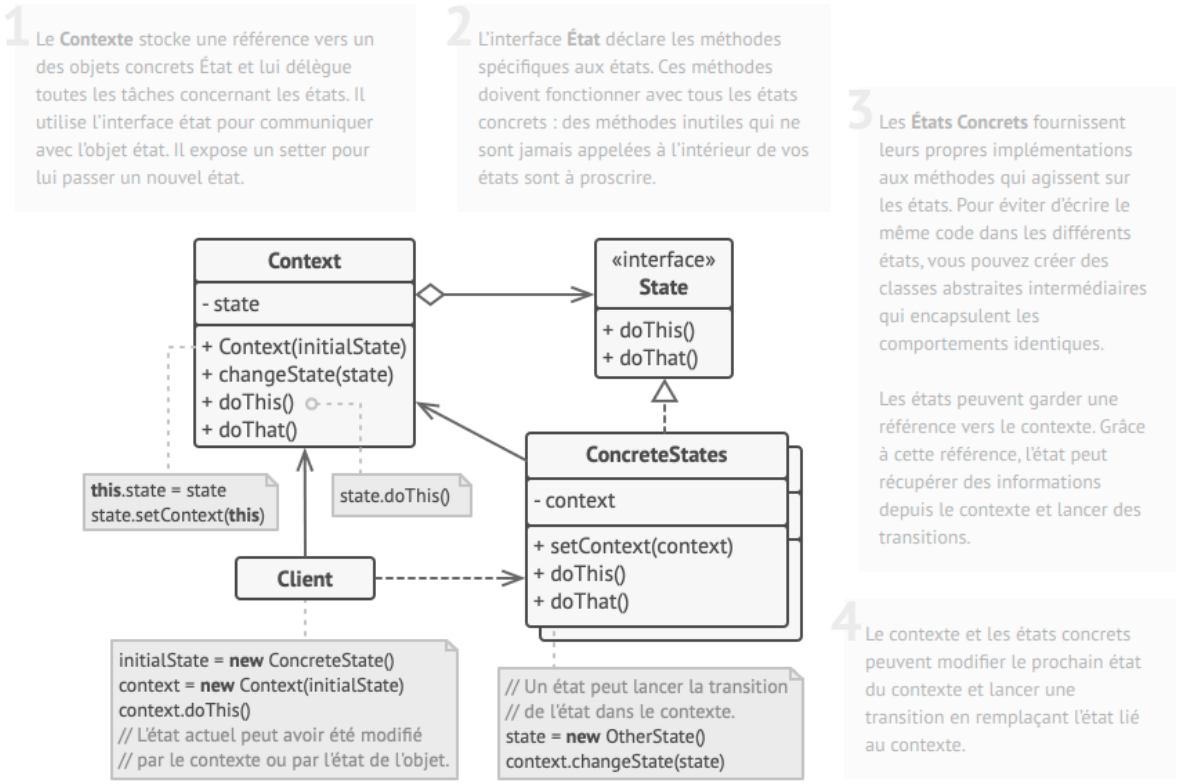
Plutôt que d'implémenter tous les comportements de lui-même, l'objet original que l'on nomme contexte, stocke une référence vers un des objets état qui représente son état actuel. Il délègue tout ce qui concerne la manipulation des états à cet objet.



Pour faire passer le contexte dans un autre état, remplacez l'objet état par un autre qui représente son nouvel état. Vous ne pourrez le faire que si toutes les classes suivent la même interface et si le contexte utilise cette dernière pour manipuler ces objets.

Cette structure ressemble de près au patron de conception Stratégie, mais il y a une différence majeure. Dans le patron de conception état, les états ont de la visibilité entre eux et peuvent lancer les transitions d'un état à l'autre, alors que les stratégies ne peuvent pas se voir.

Structure



Quand utiliser le pattern état ?

- Utilisez le patron de conception état lorsque le comportement de l'un de vos objets varie en fonction de son état, qu'il y a beaucoup d'états différents et que ce code change souvent ;
- Utilisez ce patron si l'une de vos classes est polluée par d'énormes blocs conditionnels qui modifient le comportement de la classe en fonction de la valeur de ses attributs ;
- Utilisez ce patron de conception si vous avez trop de code dupliqué dans des états et transitions similaires de votre automate.

5- Conclusion

Une bonne partie des patrons de conception utilise la notion d'interface et de classes abstraites pour leurs différentes implémentations. Maîtriser ces concepts clés de la programmation orientée objet est un bon moyen pour comprendre les design patterns. Les design patterns demeurent à bien des égards la meilleure manière de résoudre certains problèmes de la conception logicielle. Cependant quelques

critiques leurs sont adressées, critiques que nous n'évoquerons volontairement pas ici. Nous dirons juste que s'il faut utiliser les design patterns, il faut être convaincu que ce soit une utilisation optimale car il arrive quelques fois qu'un code simple résolve le problème.

Références bibliographiques

[1] Alexander Shvets. (2021). *Plongez au cœur des patrons de conception.*

[2] Mohamed Youssfi. (2014). *Design patterns première partie*. Université Hassan II, Casablanca, Maroc.

[3] Steven John Metsker, William C. Wake. (2006). *Les Design Patterns en Java, Les 23 modèles de conception fondamentaux.* Paris, France : Addison-Wesley.