Design pattern

Design pattern

- Un design pattern est Schéma formant une solution reconnue comme fiable et robuste à un problème connu ou récurrent.
- Un design pattern est une capitalisation sur l'ensemble du travail déjà réaliser pour :
- Évite de réinventer la roue !
- Gain de temps
- Permet de s'assurer que l'on applique une solution fiable, robuste et éprouvée.

Design pattern – Principe du SOLID

- **SRP : Single Responsability Principle**
  - Une classe doit avoir une et une seule raison de changer. Si une classe a plus d'une responsabilité, alors ces responsabilités deviennent couplées.
- **OCP : Open/Close Principle**
  - Les entités doivent être extensible mais non modifiable.
- **LSP : Liskov Substitution Principle**
  - Tout type de base doit pouvoir être remplacé par l'un de ses sous-types.
- **ISP : Interface Segregation Principle**
  - Découper au mieux nos interfaces par besoin
- **DIP : Dependency Inversion Principle**
  - Il ne faut pas dépendre (directement) des implémentation bas niveau

Design pattern

- Les designs pattern sont découpés en 3 types:
  - Les structural patterns
  - Les creational patterns
  - Les behavioral patterns

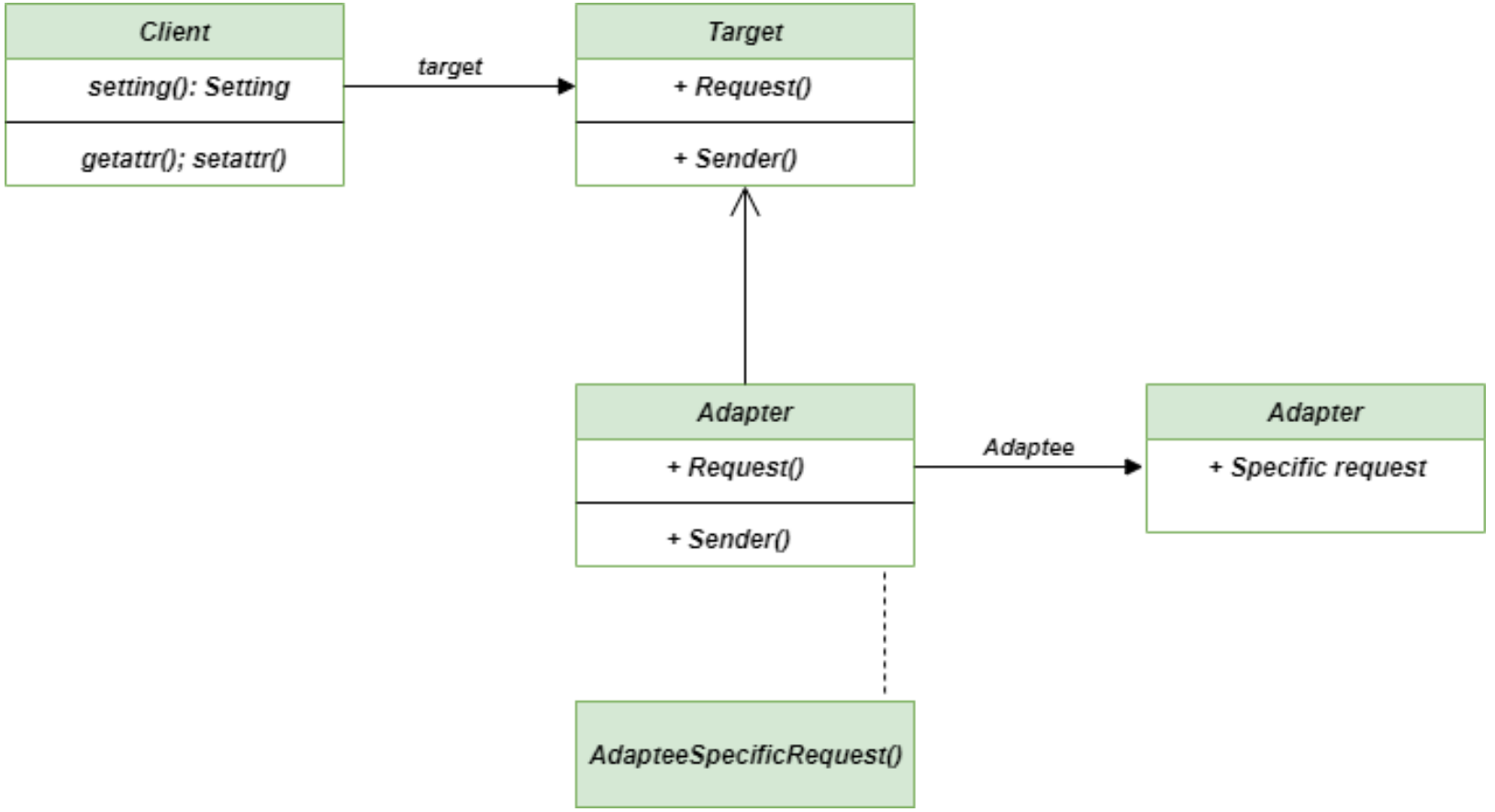
Design pattern – pattern de structure

- Description de la manière dont un objet ou un ensemble d'objets peuvent être créés, initialisés, et configurés
- Isolation du code relatif à la création, à l'initialisation afin de rendre l'application indépendante de ces aspects
  - Adapter Pattern
  - Bridge Pattern
  - Decorator Pattern
  - Composite Pattern

Design pattern – pattern de structure – Adapter pattern

- Problème
  - Utilisation d'une classe existante dont l'interface ne nous convient pas (convertir l'interface d'une classe en une autre)
  - Utilisation de plusieurs sous-classes dont l'adaptation des interfaces est impossible par dérivation (Object Adapter)
- Conséquences
  - Adapter de classe
    - il n'introduit qu'une nouvelle classe, une indirection vers la classe adaptée n'est pas nécessaire
    - MAIS il ne fonctionnera pas dans le cas où la classe adaptée est racine d'une dérivation
  - Adapter d'objet
    - il peut fonctionner avec plusieurs classes adaptées
    - MAIS il peut difficilement redéfinir des comportements de la classe adaptée

Design pattern – pattern de structure – Adapter pattern



Exercice

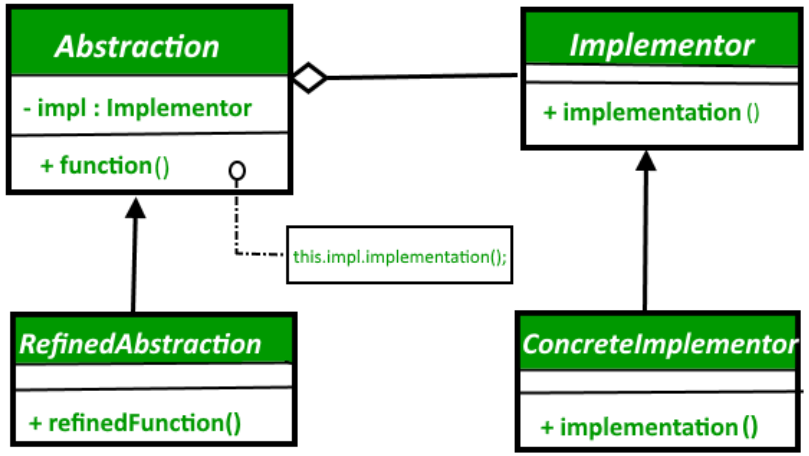
EXERCICE

- Nous souhaitons créer une application qui permet de récupérer des données concernant plusieurs types de véhicules (moto, camion, voiture,…).
- Notre application récupèrera dans un premier temps les données en format xml.
- Par la suite nous souhaitons récupérer les données en json pour les mettre à disposition d'une autre application par exemple.
- Implémentez les différentes classes nécessaires (moto, camion, voiture,…)
- Créez un adapter qui permettra de résoudre la problématique du passage du xml vers json

Design pattern – pattern de structure – Bridge pattern

- Problème
  - ce motif est à utiliser lorsque l'on veut découpler l'implémentation de l'abstraction de telle sorte que les deux puissent varier indépendamment
- Conséquences
  - interfaces et implémentations peuvent être couplées/découplées lors de l'exécution

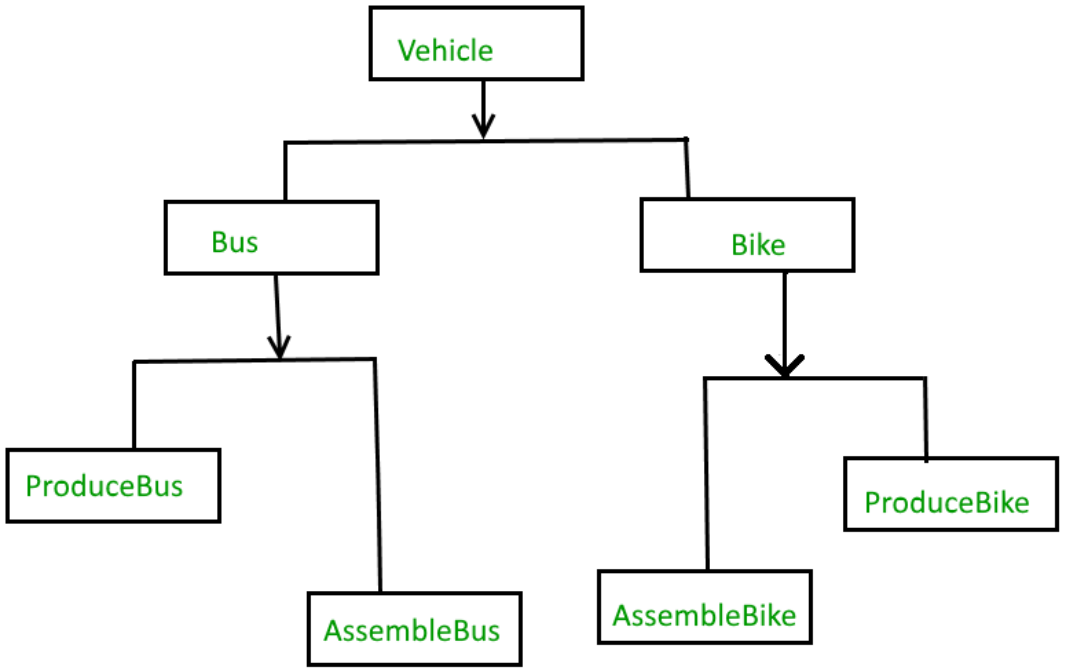
Design pattern – pattern de structure – Bridge pattern



Exercice

EXERCICE

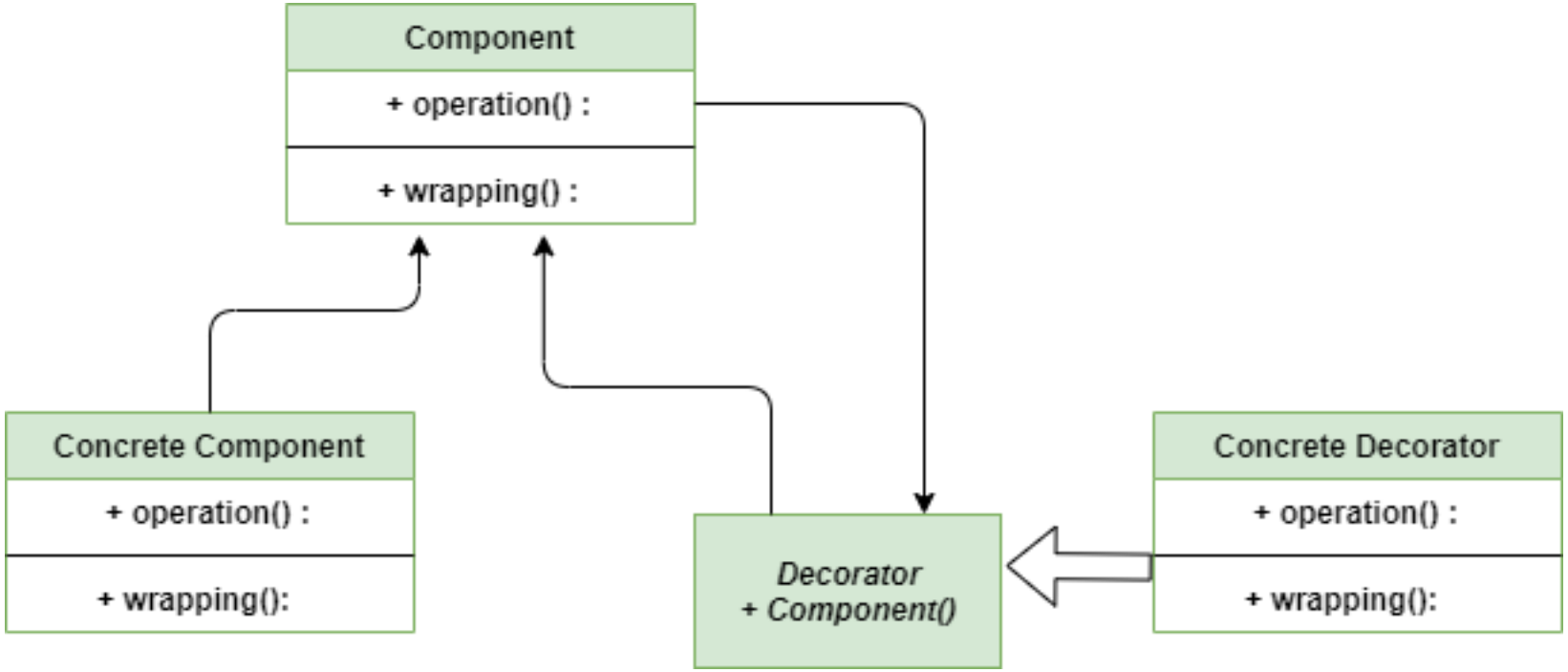
- Convertir la modélisation suivante en une modélisation en Bridge



Design pattern – pattern de structure – Decorator pattern

- Problème
  - on veut ajouter/supprimer des responsabilités aux objets en cours de fonctionnement
  - l'héritage est impossible à cause du nombre de combinaisons, ou à cause de droits d'accès
- Conséquences
  - plus de flexibilité que l'héritage
  - réduction de la taille des classes du haut de la hiérarchie
  - MAIS beaucoup de petits objets

Design pattern – pattern de structure – Decorator pattern



Exercice

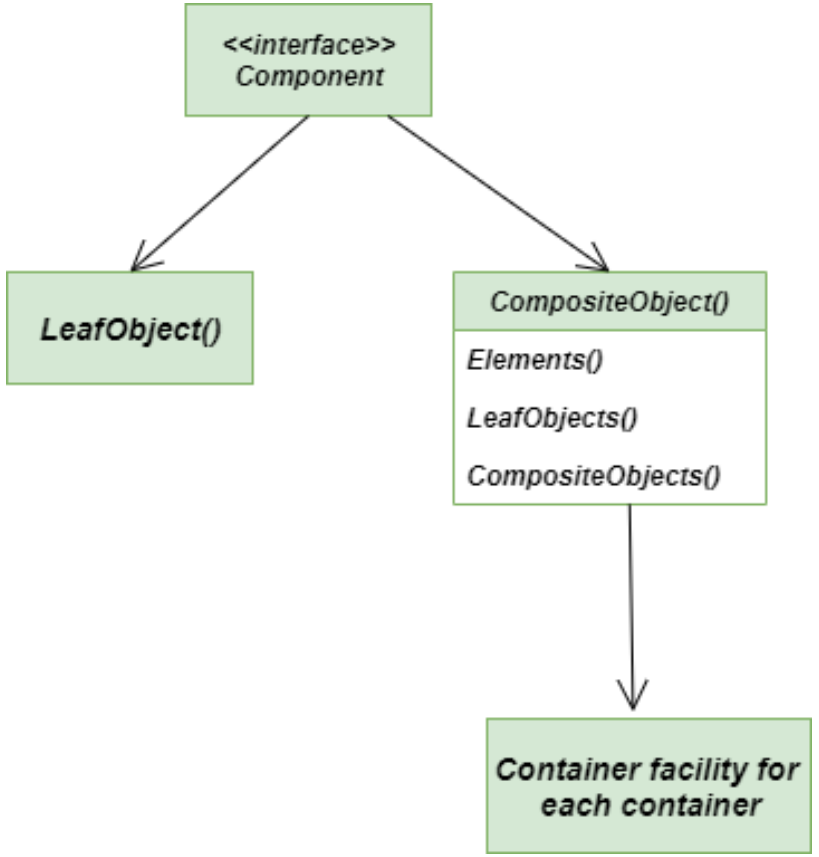
EXERCICE

- Nous souhaitons créer une application qui permet fabriquer différents types de cafés, chaque café à un cout et un nom:
  - Colombia, Expresso, Deca,…
- L'application permet d'ajouter différents ingrédients à votre café, chaque ingrédient a un cout également
  - Sirop de vanille, chocolat, lait, noisette,…
- Modélisez et implémentez cette application en utilisant le pattern décorateur.

Design pattern – pattern de structure – Composite pattern

- Problème
  - établir des structures arborescentes entre des objets et les traiter uniformément
- Conséquences
  - hiérarchies de classes dans lesquelles l'ajout de nouveaux composants est simple
  - simplification du client qui n'a pas à se préoccuper de l'objet accédé
  - MAIS il est difficile de restreindre et de vérifier le type des composants

Design pattern – pattern de structure – Composite pattern



Exercice

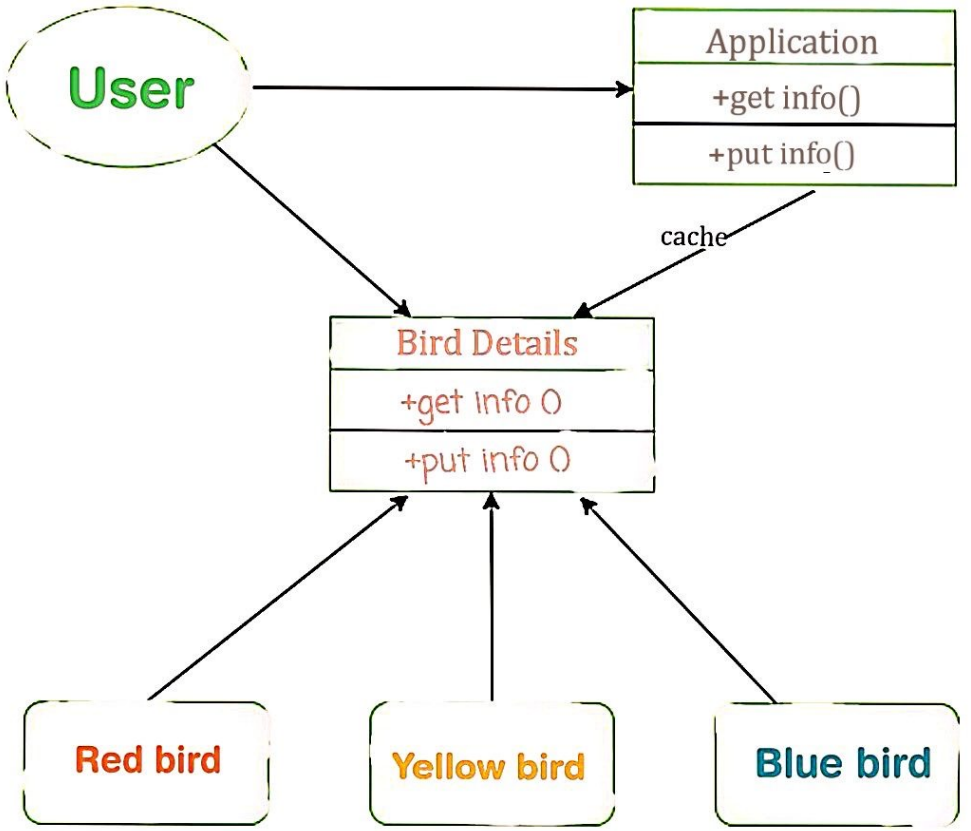
EXERCICE

- En utilisant le pattern composite, réalisez une application qui permet de visualiser les fichiers / sous-dossiers d'un dossier donné.
- L'application doit permettre d'exécuter un nombre d'opération quelque soit l'élément du dossier soit fichier ou sous-dossier.
- Les opérations sont : ajout, déplacement, suppression

Design pattern – pattern de structure – Flyweight pattern

- Problème
  - grand nombre d'objet
  - le coût du stockage est élevé
  - l'application ne dépend pas de l'identité des objets
- Conséquences
  - réduction du nombre d'instances
  - coût d'exécution élevé
  - plus d'états par objet

Design pattern – pattern de structure – Flyweight pattern



Exercice

EXERCICE

- Soit une application qui permet d'effectuer un inventaire pour les ventes de voitures.
- Chaque voiture individuelle a un numéro de série spécifique et une couleur spécifique.
- La plupart des autres détails concernant cette voiture sont les mêmes pour toutes les voitures d'un modèle particulier, par exemple :
  - Le modèle Honda Fit DX est une voiture simple avec peu de fonctionnalités.
  - Le modèle LX est équipé de la climatisation, de l'inclinaison, du régulateur de vitesse, des vitres et des serrures électriques.
  - Le modèle Sport a des roues fantaisie, un chargeur USB et un spoiler.
- En utilisant le pattern flyweight, donnez une modélisation de cette application.

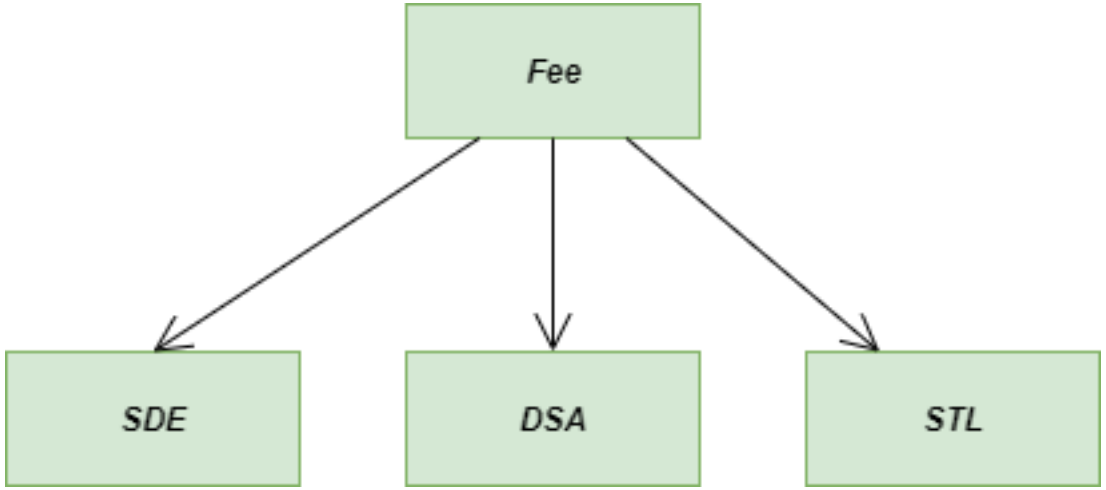
Design pattern – pattern de création

- Rendre le système indépendant de la manière dont les objets sont créés, composés et représentés
  - Encapsulation de la connaissance des classes concrètes à utiliser
  - Cacher la manière dont les instances sont créées et combinées
- Permettre dynamiquement ou statiquement de préciser QUOI (l'objet), QUI (l'acteur), COMMENT (la manière) et QUAND (le moment) de la création
- Exemple :
  - Abstract Factory Pattern
  - Builder Pattern
  - Factory Method Pattern
  - Prototype Pattern
  - The Singleton Pattern

Design pattern – pattern de structure – Abstract Factory pattern

- Problème
  - ce motif est à utiliser dans les situations où existe le besoin de travailler avec des familles de produits tout en étant indépendant du type de ces produits
  - doit être configuré par une ou plusieurs familles de produits
- Conséquences
  - Séparation des classes concrètes, des classes clients
    - les noms des classes produits n'apparaissent pas dans le code client
    - Facilite l'échange de familles de produits
    - Favorise la cohérence entre les produits
  - Le processus de création est clairement isolé dans une classe
  - la mise en place de nouveaux produits dans l'AbstractFactory n'est pas aisée

Design pattern – pattern de structure – Abstract Factory pattern



Exercice

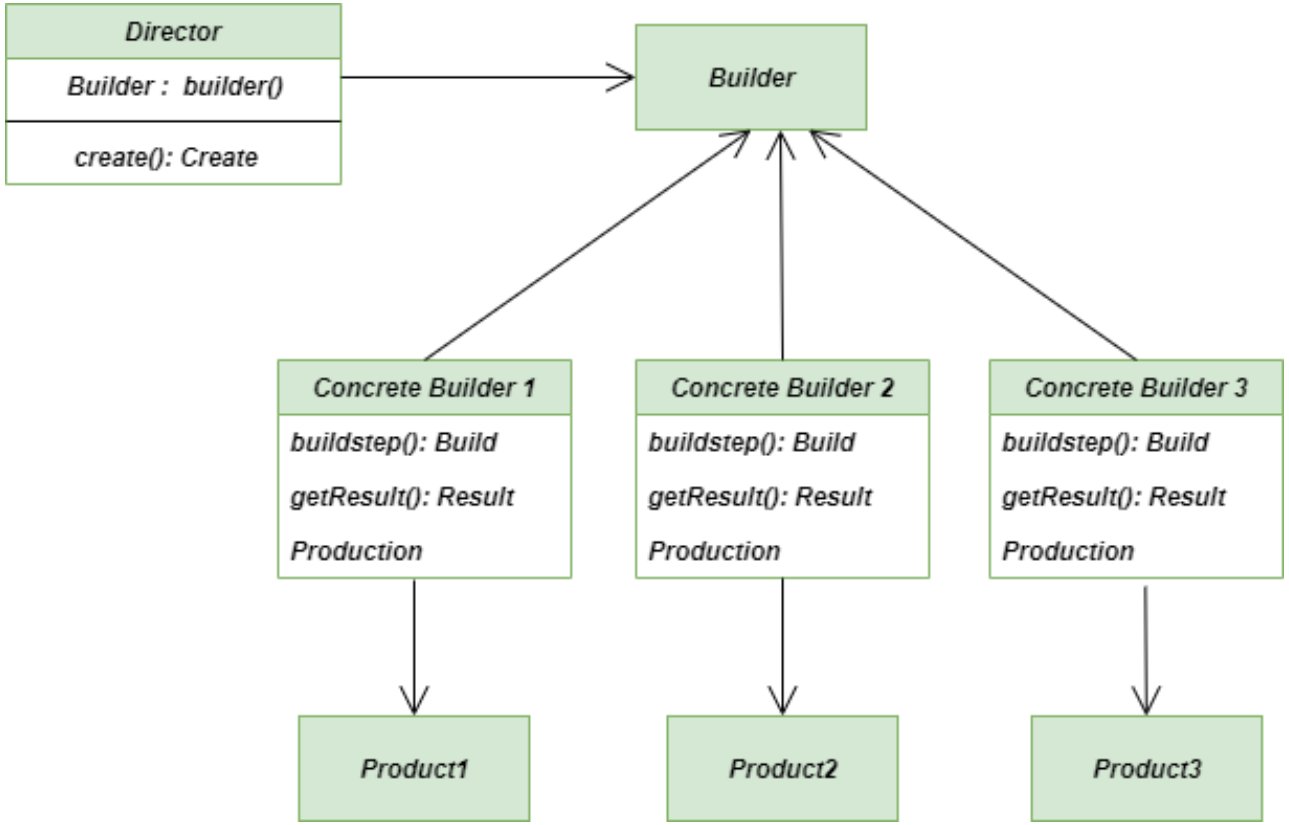
EXERCICE

- Nous souhaitons mettre en place une application qui permet de commander plusieurs type pizza en fonction d'un menu.
- Chaque pizza passe par les étapes suivantes : préparer, cuire, couper et emballer.
- L'application doit nous permettre d'ajouter facilement d'autre types de pizza dans le menu.
- En utilisant l'abstract factory, implémentez cette application.

Design pattern – pattern de structure – builder pattern

- Problème
  - ce motif est intéressant à utiliser lorsque l'algorithme de création d'un objet complexe doit être indépendant des constituants de l'objet et de leurs relations, ou lorsque différentes représentations de l'objet construit doivent être possibles
- Conséquences
  - Variation possible de la représentation interne d'un produit
    - l'implémentation des produits et de leurs composants est cachée au Director
    - Ainsi la construction d'un autre objet revient à définir un nouveau Builder
  - Isolation du code de construction et du code de représentation du reste de l'application
  - Meilleur contrôle du processus de construction

Design pattern – pattern de structure – Builder pattern



Exercice

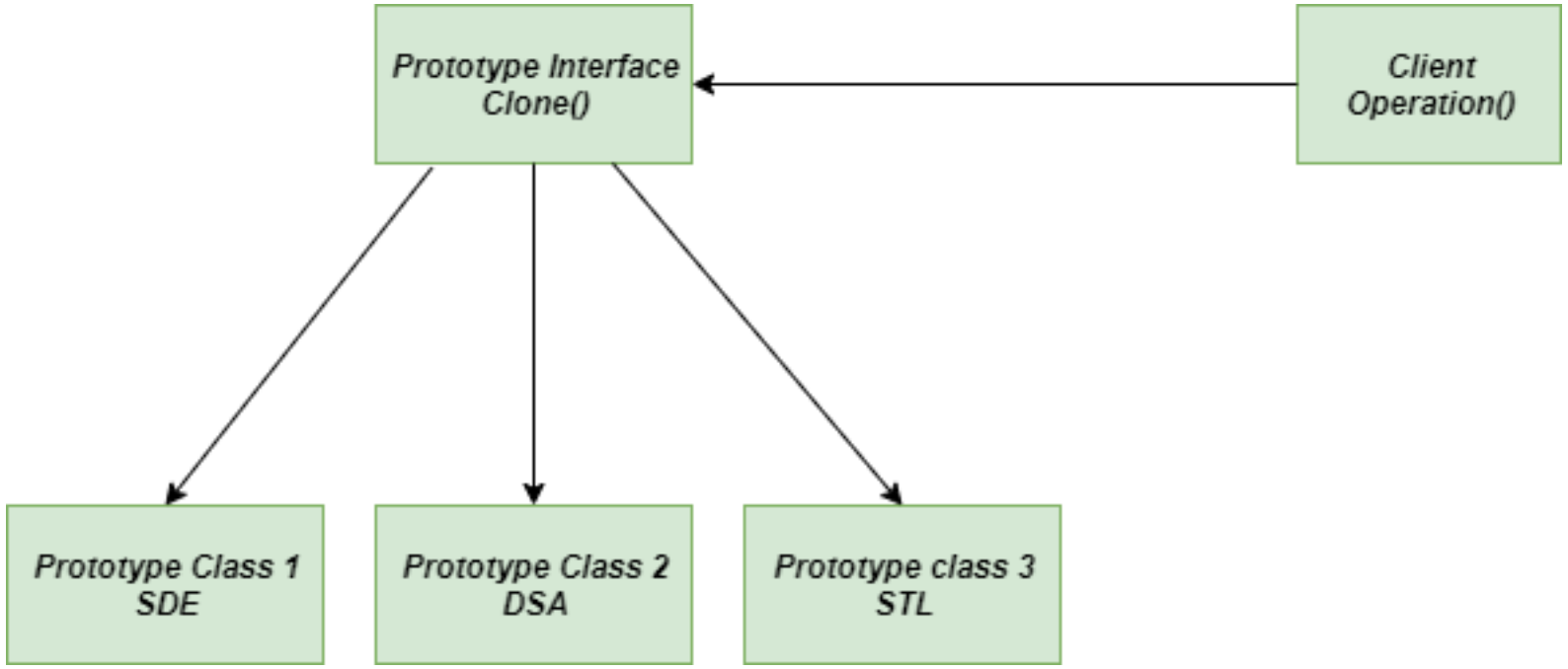
EXERCICE

- Nous souhaitons avoir une application qui permet de construire des véhicules (Voiture, Camion), chaque véhicule est composé de plusieurs autres objets (moteur, roue) en fonction du type du véhicule.
- En utilisant le pattern builder implémentez cette application.

Design pattern – pattern de structure – prototype pattern

- Problème
  - Le système doit être indépendant de la manière dont ses produits sont créés, composés et représentés : les classes à instancier sont spécifiées au moment de l'exécution
  - La présence de hiérarchies de Factory similaires aux hiérarchies de produits doivent être évitées. Les combinaisons d'instances sont en nombre limité
- Conséquences
  - mêmes conséquences que Factory et Builder

Design pattern – pattern de structure – prototype pattern



Exercice

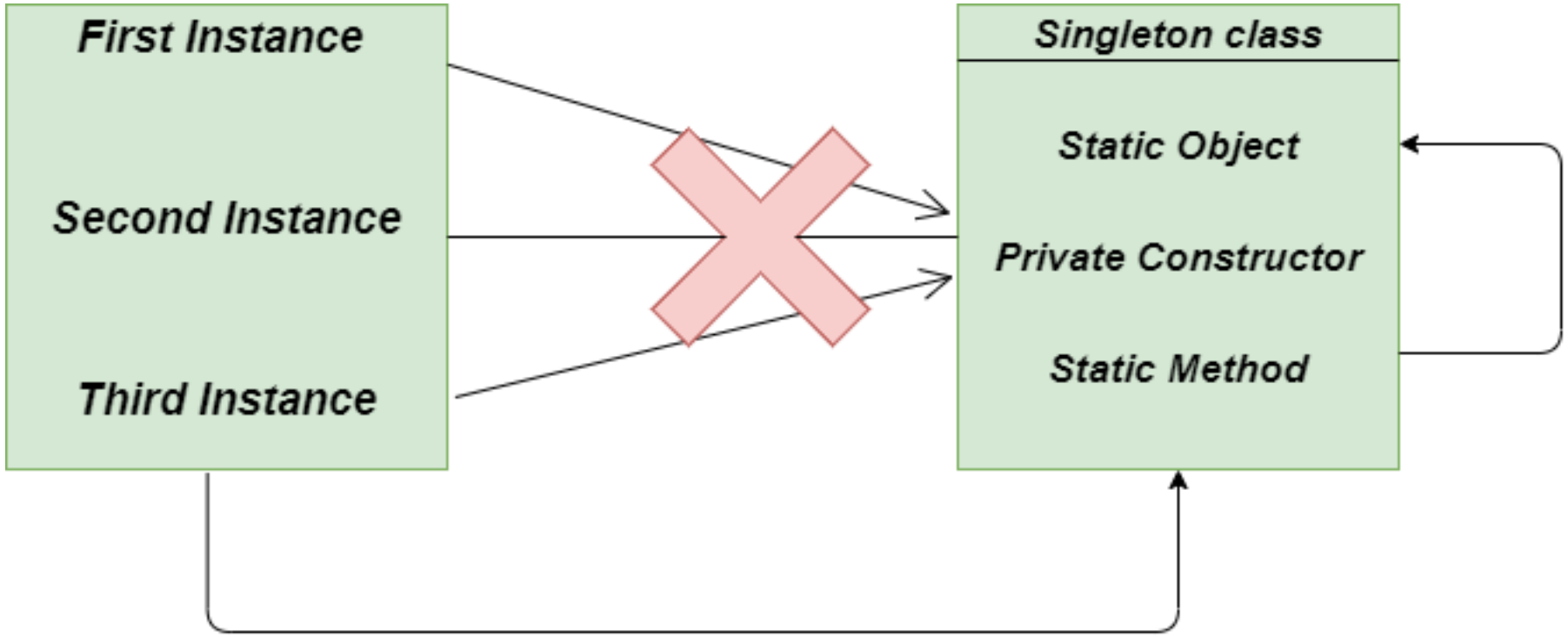
EXERCICE

- Nous souhaitons créer des personnages pour un jeu vidéo.
- Commerçant, un guerrier, et un mage.
- Les personnages possèdent des caractéristiques communes et des spécificités. Par exemple commerçant a du charisme, guerrier une force et le mage un pouvoir.
- En utilisant le pattern Prototype Implémentez les différentes classes.

Design pattern – pattern de structure – singleton pattern

- Problème
  - avoir une seule instance d'une classe et pouvoir l'accéder et la manipuler facilement
- Solution
  - une seule classe est nécessaire pour écrire ce motif
- Conséquences
  - l'unicité de l'instance est complètement contrôlée par la classe elle même. Ce motif peut facilement être étendu pour permettre la création d'un nombre donné d'instances

Design pattern – pattern de structure – Singleton pattern



Exercice

EXERCICE

- Créer un singleton thread safe pour extraire des informations de configuration à partir d'un fichier.

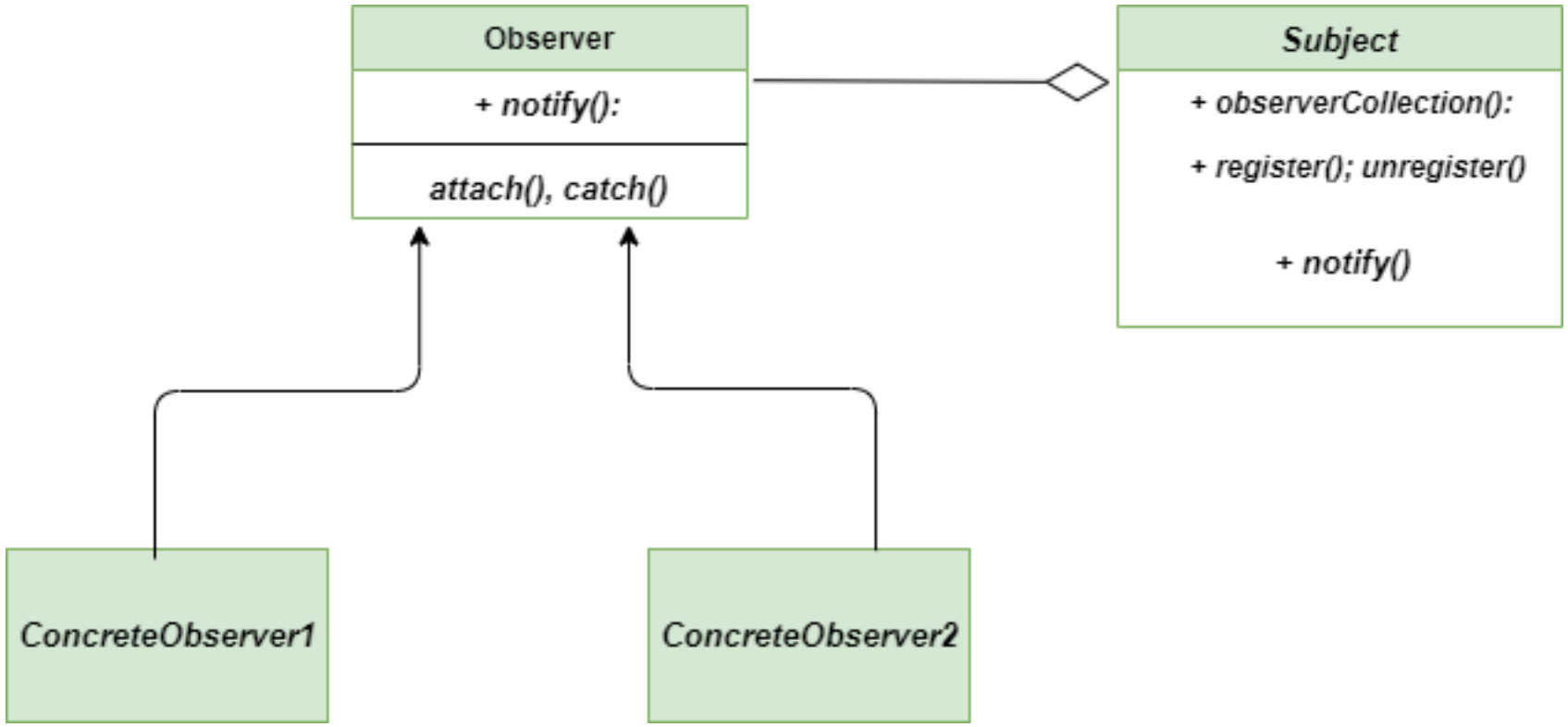
Design pattern – pattern de comportement

- Description de structures d'objets ou de classes avec leurs interactions
- Deux types de motifs
  - Motifs de comportement de classes :
    - utilisation de l'héritage pour répartir les comportements entre des classes (ex : Interpreter)
  - Motifs de comportement d'objets avec l'utilisation de l'association entre objets :
    - pour décrire comment des groupes d'objets coopèrent (ex : Mediator)
    - pour définir et maintenir des dépendances entre objets (ex : Observer)
    - pour encapsuler un comportement dans un objet et déléguer les requêtes à d'autres objets (ex : Strategy, State, Command)
    - pour parcourir des structures en appliquant des comportements (ex :Visitor, Iterator)

Design pattern – pattern de structure – observer pattern

- Problème
  - on veut assurer la cohérence entre des classes coopérant entre elles tout en maintenant leur indépendance
- Conséquences
  - couplage abstrait entre un sujet et un observeur, support pour la communication par diffusion,
  - MAIS des mises à jour inattendues peuvent survenir, avec des coûts importants

Design pattern – pattern de structure – observer pattern



Exercice

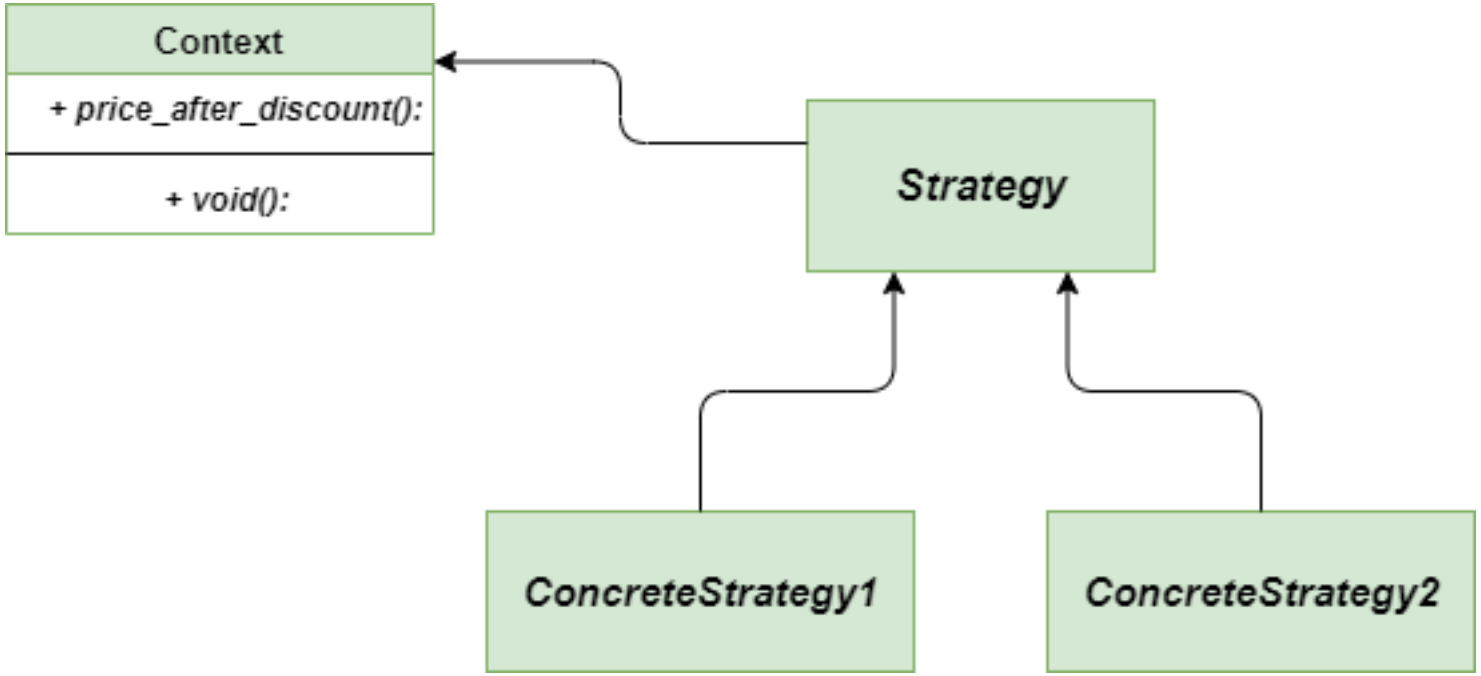
EXERCICE

- Nous souhaitons réaliser une classe inventaire qui contient un set de produits et qui permet d'ajouter ou supprimer des produits du set.
- Cette classe doit permettre à une classe extérieure d'être informer des changements de produits et quel produit est concerné par le changement.
- Implémentez la classe inventaire et une exemple d'une classe extérieure (observer).
- La classe extérieure affichera juste un message dans la console.

Design pattern – pattern de structure – strategy pattern

- Problème
  - on veut
    - définir une famille d'algorithmes
    - encapsuler chacun et les rendre interchangeables tout en assurant que chaque algorithme peut évoluer indépendamment des clients qui l'utilisent
- Conséquences
- Expression hiérarchique de familles d'algorithmes, élimination de tests pour sélectionner le bon algorithme, laisse un choix d'implémentation et une sélection dynamique de l'algorithme
- Les clients doivent faire attention à la stratégie, surcoût lié à la communication entre Strategy et Context, augmentation du nombre d'objets

Design pattern – pattern de structure – strategy pattern



Exercice

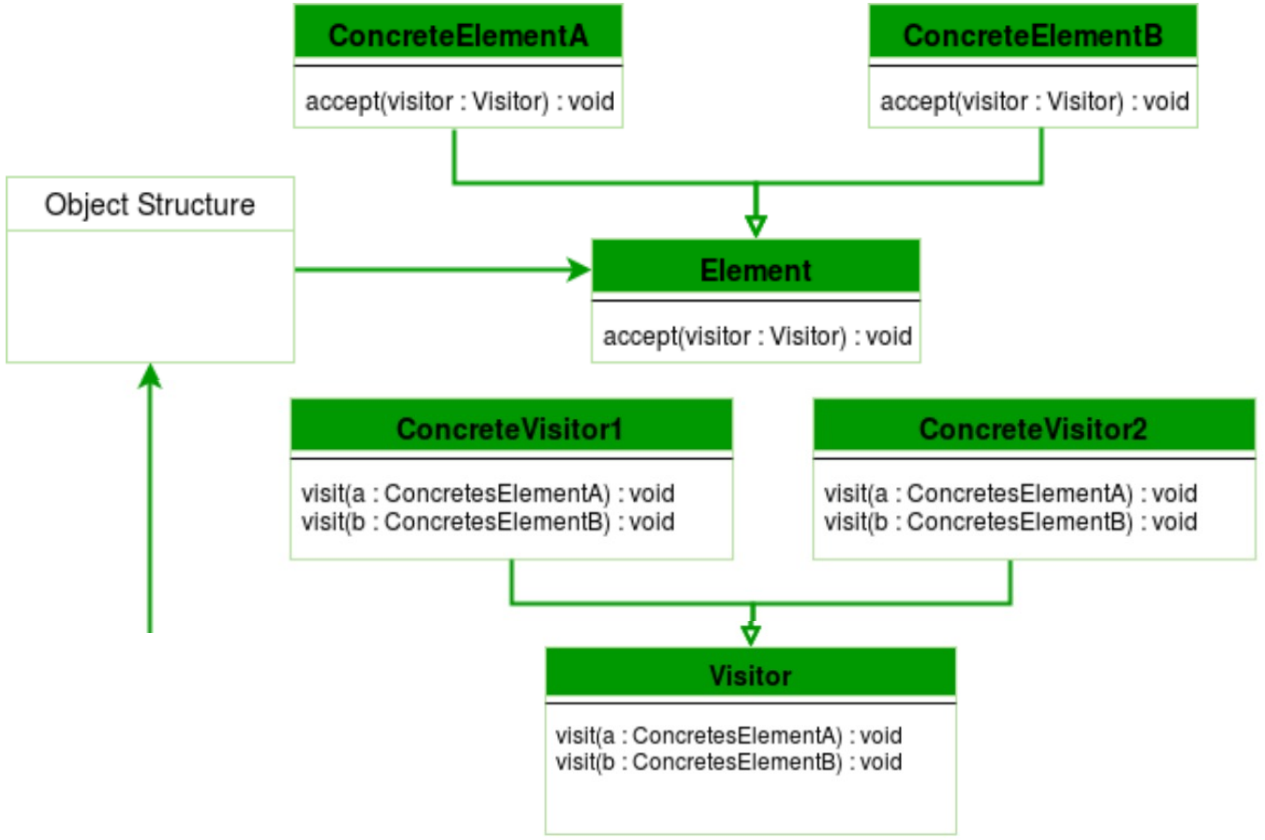
EXERCICE

- Nous souhaitons créer un jeu de papier pierre ciseaux en utilisant un modèle de stratégie.
- Vous pouvez sélectionner n'importe quelle stratégie parmi pierre, papier, ciseaux et une stratégie au hasard pour l’ordinateur.
- Implémentez le jeu.

Design pattern – pattern de structure – visitor pattern

- Problème
  - Des opérations doivent être réalisées dans une structure d'objets comportant des objets avec des interfaces différentes
  - Plusieurs opérations distinctes doivent être réalisées sur des objets d'une structure
  - La classe définissant la structure change rarement mais de nouvelles opérations doivent pouvoir être définies souvent sur cette structure
- Conséquences
  - l'ajout de nouvelles opérations est aisé
  - union de différentes opérations et séparations d'autres
  - MAIS l'ajout de nouvelles classes concrètes est freinée

Design pattern – pattern de structure – visitor pattern



Exercice

EXERCICE

- Nous souhaitons réaliser une application de commerce électronique, où nous pouvons ajouter différents types d'articles dans un panier, nous voulons également avoir une fonctionnalité pour calculer le montant total.
- Étant donné que chaque élément sera un objet distinct. Nous pouvons utiliser le pattern visitor pour obtenir le montant total