L3 Info – INF5A1
Jour 9
le 8 novembre 2021

Design Patterns
suite

Sommaire

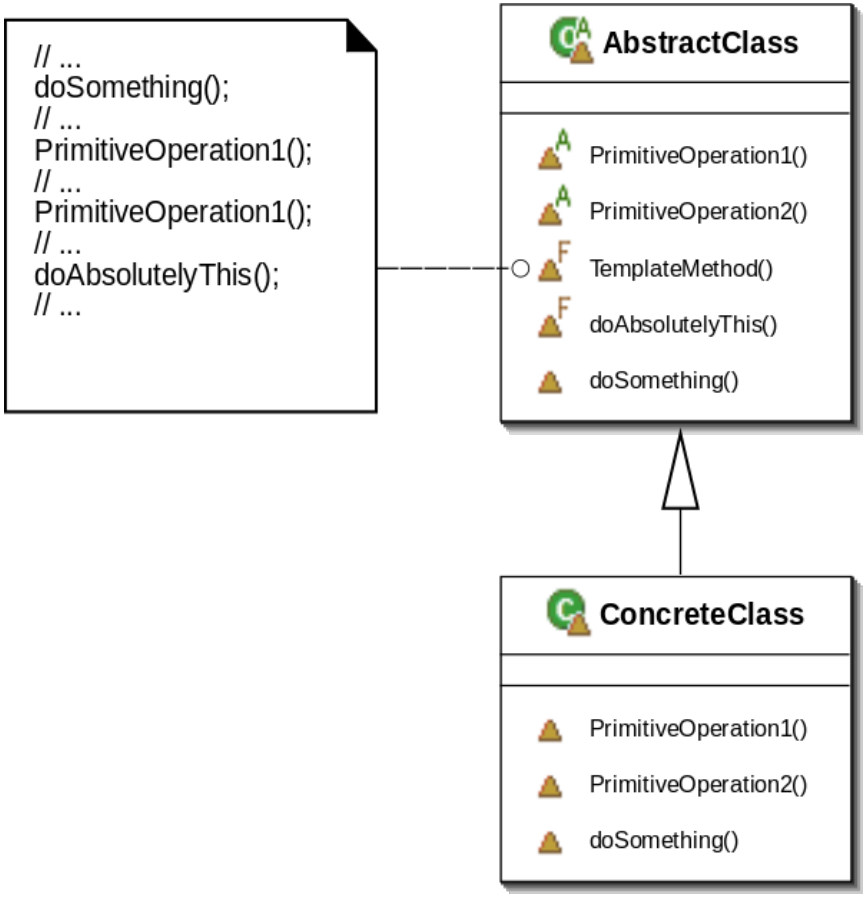
- Patterns :
  - Template method
  - Le pattern Composite
  - Le pattern Chain of Responsibility
- Principe des Packages

Pattern Template method

- Un pattern concernant les classes abstraites que vous utilisez peut-être sans le savoir !
- Idée : dans une classe abstraite, une méthode non abstraite utilise des méthodes abstraites
- Problème auquel on répond : proposer un algorithme générique pouvant avoir différentes implémentations (sans dupliquer le patron général de l'algorithme)
- Solution : le patron général est une méthode non abstraite, s'appuyant sur des méthodes qui seront implémentées au niveau des sous-classes

Template method : diagramme

crédit : wikipedia



Abstraction : classes versus instances

- L'une des difficultés de ce pattern pour les novices est qu'il nécessite de bien comprendre l'abstraction :
  - comment une méthode non abstraite peut-elle utiliser une méthode abstraite, donc non encore implémentée dans la classe ?
- L'abstraction concerne les classes, pas les instances. Une instance provient toujours d'une classe concrète, donc toutes ses méthodes sont implémentées.
- En d'autres termes, dans un objet, aucune méthode n'est jamais abstraite, même si par polymorphisme on manipule cet objet via un super-type abstrait (interface ou classe abstraite). Du code peut donc faire appel à des méthodes abstraites, c'est d'ailleurs ce que l'on fait en permanence avec les interfaces...

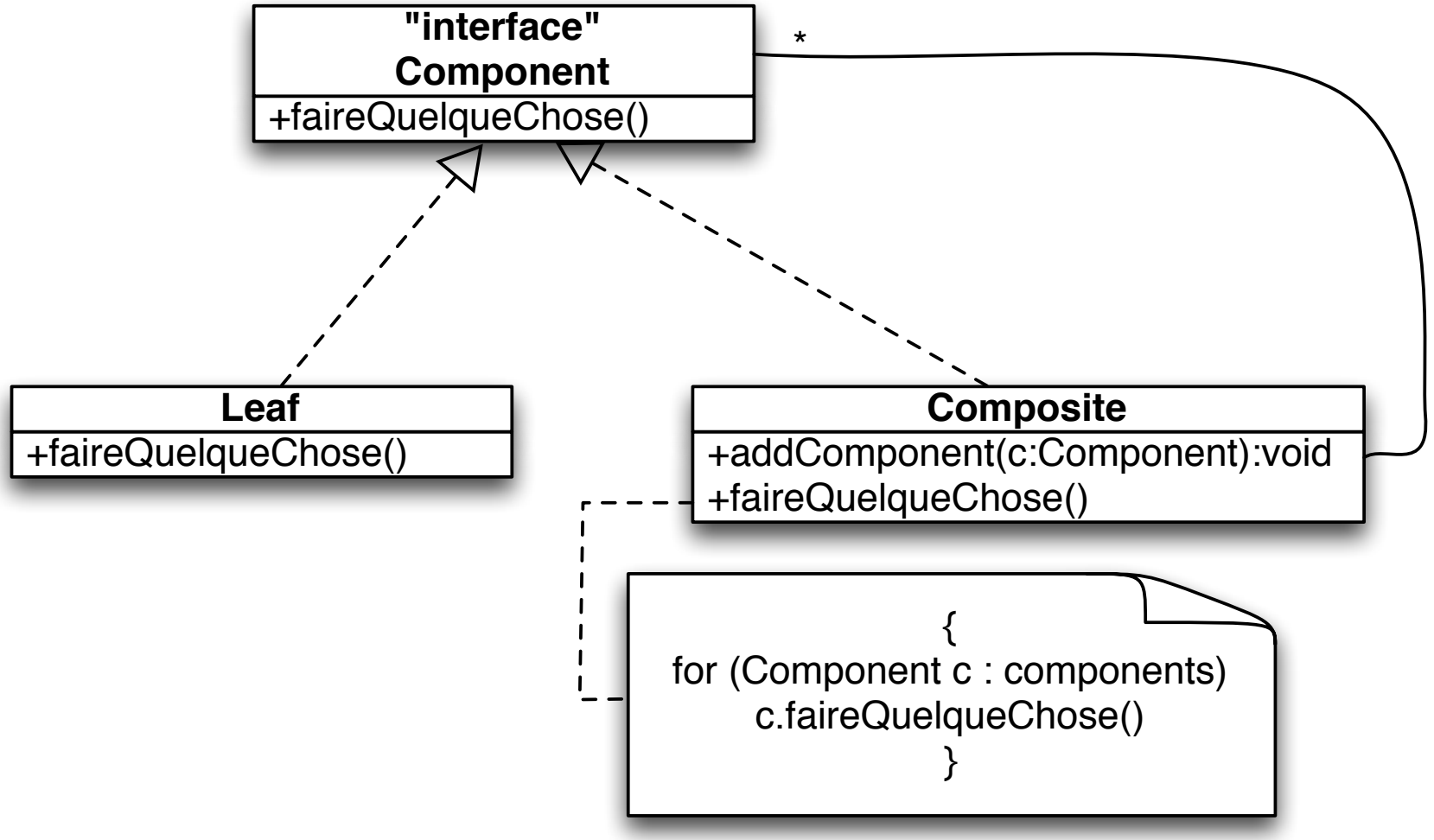
Pattern Composite

- Permettre le traitement uniforme d'objets et de collections d'objets
- Exemple : on veut pouvoir appliquer une translation à une forme, mais aussi à un ensemble de formes, de la même façon
- Il faut donc pouvoir considérer une partie ou un tout (i.e. un ensemble de parties) comme des entités d'un même type

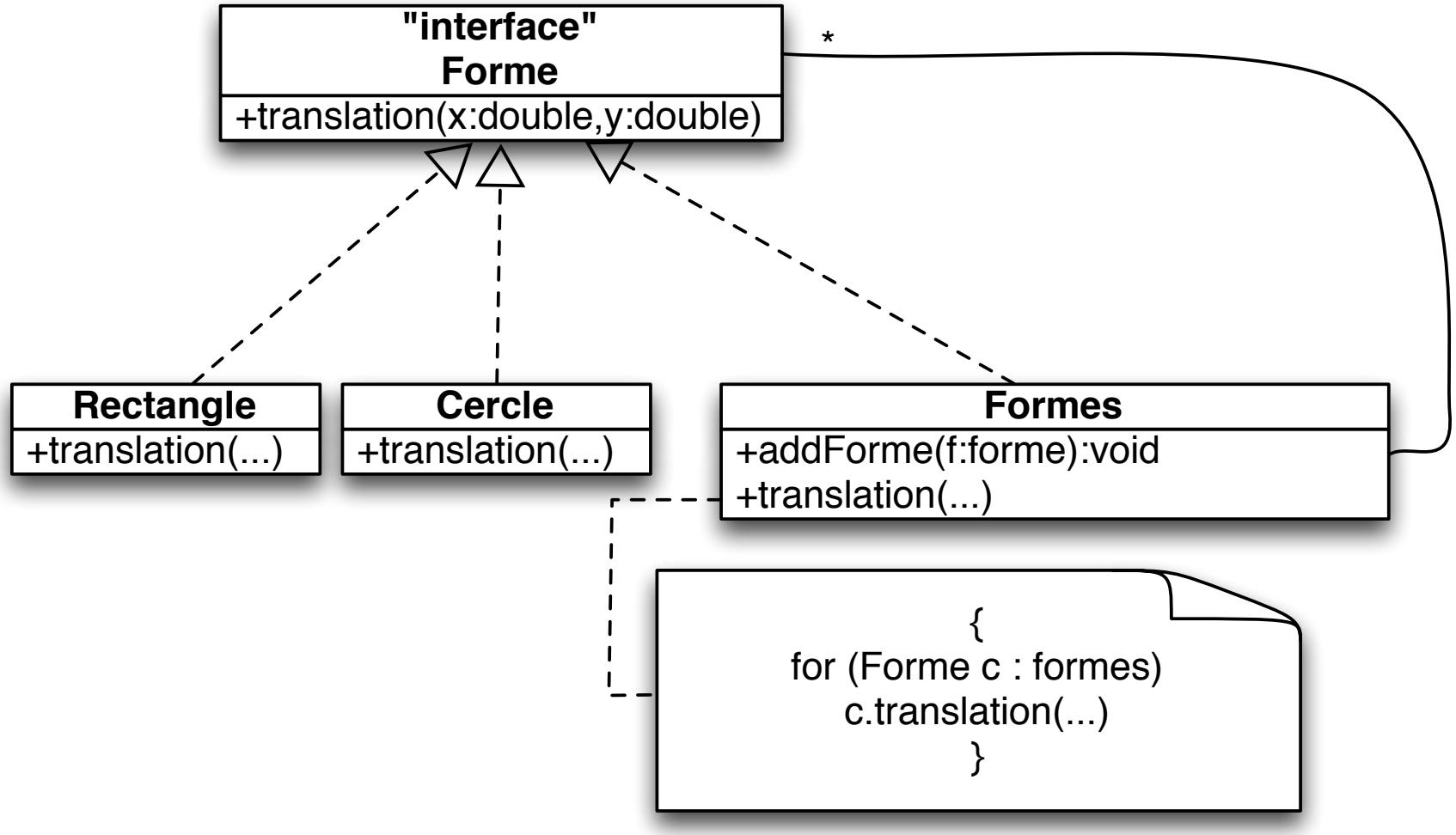
Composite (suite)

- Solution :
  - Proposer un super type qui soit commun aux parties et aux touts. On le nomme Component
  - Faire hériter des entités simples (Leaf) et des entités composées (Composite) de ce super type
  - Créer une association afin qu'un Composite puisse contenir des Components.
  - On obtient ainsi une structure hiérarchique sous forme d'arbre, composée classiquement de nœuds et de feuilles, mais lesquels partagent le fait d'être des Components, et donc savoir faire les mêmes choses.

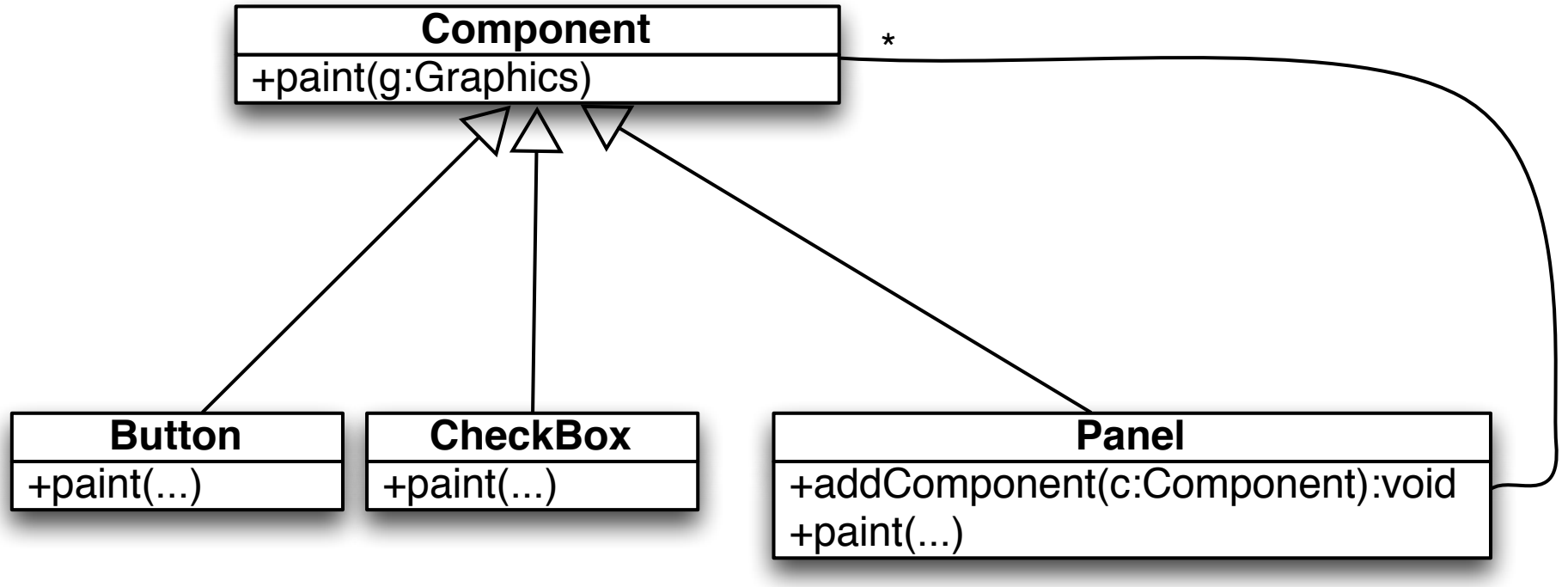
Composite : diagramme de classes



Composite exemple 1



Composite : exemple 2



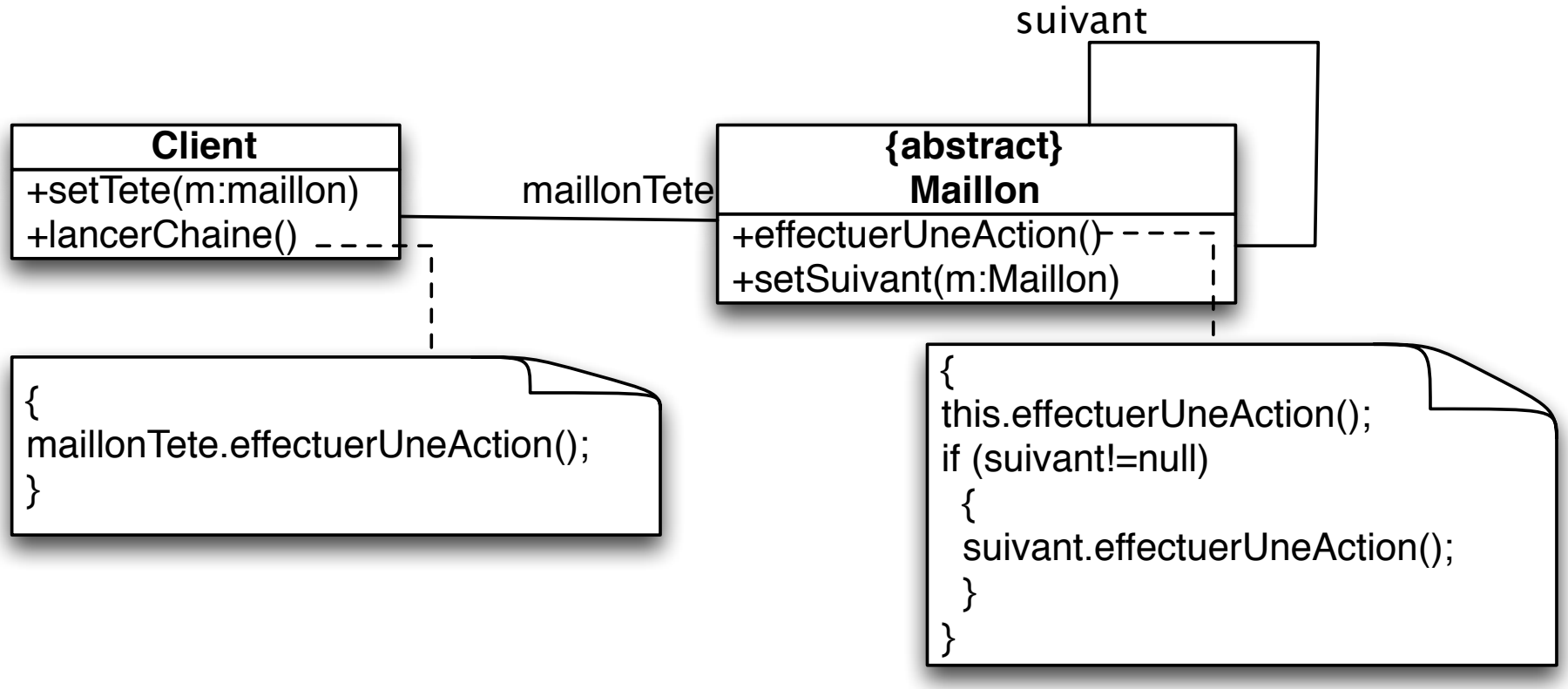
Pattern Chain of Responsibility

- Permettre à différents composants indépendants de participer au traitement d'une requête
- Les composants sont des maillons qui se suivent au sein d'une chaine
- Chaque maillon tente de répondre à la requête, puis, éventuellement, passe cette dernière au maillon suivant, et ainsi de suite

Chain of Responsibility (2)

- On évite ainsi le couplage entre les différents maillons : les classes correspondantes ne sont pas liées
- On peut créer de nouveaux maillons en créant de nouvelles classes
- On peut enfin, dynamiquement (à l'exécution), créer différentes chaînes par création et assemblage de maillons

Chain of Responsibility : UML



Chain Of Responsibility : variantes

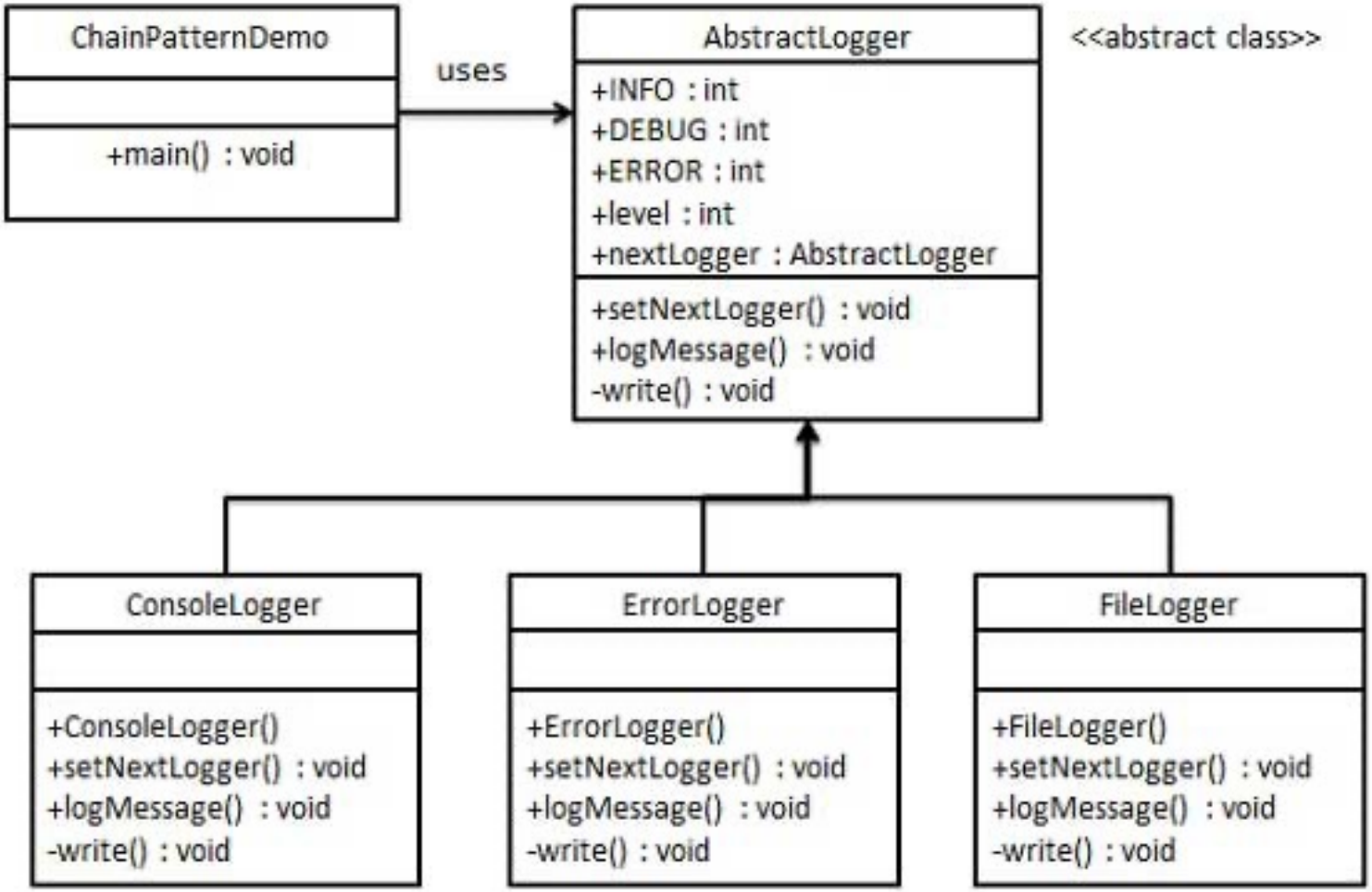
- Selon le type de chaîne que l'on veut créer :
  - soit tous les maillons sont successivement appelés, et chacun apporte sa contribution
  - Soit chaque maillon essaye de répondre à la requête, et passe cette dernière au maillon suivant s'il ne sait pas répondre

Exemple 1 : vérifier des droits

- Un utilisateur a le droit de réserver un court de tennis si son abonnement est à jour et si au moins un terrain est libre sur le créneau demandé
- Deux maillons sont créés, un pour chacune de ces exigences
- La chaîne est créée pour aller jusqu'à son terme : tous les maillons doivent répondre oui, y compris le dernier. Le droit est avéré si un maillon répond vrai et s'il n'a pas de successeur (ce qui signifie que l'on est en bout de chaîne).
- De façon économique, on peut pré-implémenter le code de vérification en chaîne dans la classe abstraite sous forme de « Template method » s'appuyant sur une méthode abstraite de vérification locale (définie dans chaque classe d'implémentation de maillon)

Exemple 2 : un logger

(tutorialspoint.com)



Révisions du L2 : les packages

- Organiser les classes dans des paquets comme on organise ses fichiers dans des dossiers
- Plusieurs avantages :
  - S'y retrouver facilement : les classes sont regroupées de façon logique
  - Pouvoir importer/exporter facilement des ensembles de classes cohérents
  - S'affranchir des difficultés posées par des classes ayant le même nom lorsqu'on utilise des classes externes

Principes d'organisation des packages

- On essaie de regrouper des ensembles de types (classes, interfaces) formant un tout
- Essayer de limiter au maximum les dépendances entre packages
- Règle vitale : interdire l'interdépendance
  - Si le paquet A dépend du paquet B
  - Et que le paquet B dépend du paquet A
  - Alors A et B sont interdépendants, il s'agit d'un découpage articifiel, et ces paquets ne pourront jamais être utilisés l'un sans l'autre : autant les regrouper

Correspondance packages & fichiers

- Les paquets sont organisés de façon arborescente, un paquet pouvant contenir d'autres paquets
- Le principe est donc le même que celui des fichiers/dossiers
- Et ce parallélisme est mis en œuvre réellement, car à tout paquet correspond un dossier dans le code source et dans le code compilé

Exemple

- Je souhaite créer un package fr.unicaen.l3
- Dans un endroit de mon arborescence de fichiers, je crée par exemple un répertoire src/ puis je crée l'arborescence correspondant aux paquets :
  - fr/
  - fr/unicaen/
  - fr/unicaen/l3

Convention de nommage

- Pour éviter la collision entre noms de classe, l'idée est que chaque développeur ait ses propres chemins de packages
- Pour cela, on utilise un nom de domaine inversé, par exemple fr.unicaen.l3.numeroEtudiant.nomDuPremierPaquet

Le vrai nom des classes…

- Le vrai nom d'une classe (ou d'une interface) est en réalité le chemin complet menant jusqu'à elle.
- Exemple : java.util.ArrayList
- Le nom ArrayList est le nom court, une abréviation permettant de ne pas encombrer le code, mais pour le compilateur, le nom pris en compte est java.util.ArrayList

Code source et code compilé

- Ce qui est vrai pour le code source l'est aussi pour le code compilé :
  - L'arborescence de packages (et donc de répertoires) définie dans le code source est reproduite à l'identique dans le code compilé

Les importations

- Les importations n'importent rien du tout
- Il s'agit seulement de dire au compilateur de quoi on parle vraiment lorsque l'on utilise un nom court.
- Exemple :
  - J'écris import java.util.ArrayList
  - Ensuite, dans tout le fichier, je peux écrire ArrayList au lieu de java.util.ArrayList

Importation groupées

- Comme pour les fichiers, on peut utiliser le signe * pour signifier tous les types situés dans un package
- Par exemple import java.util.* importe java.util.ArrayList, java.util.Collection, etc.
- Attention : les importations ne sont pas récursives :
  - import java.*; ne dispense pas d'écrire :
  - import java.util.*;

Faut-il éviter d'utiliser * dans les import ?

- Aucune surchage mémoire :
  - import n'importe rien, mais dit seulement où aller chercher les noms complets
  - C'est seulement lorsqu'un type est utilisé dans le code qu'il est chargé en mémoire
  - Aucune contre-indication de ce côté là
- Mais possibilité de collision :
  - java.util.Date
  - Java.sql.Date

Pallier les collisions ?

- Soit en les évitant :

  - Ne pas mettre import java.util.* et import java.sql.* mais faire les imports individuels

  - Mais pas toujours possible si on a besoin d'utiliser les deux types dans le même fichier

- Soit, tout simplement, en utilisant le nom complet des classes dans le code :
  - java.util.Date date=new java.util.Date();
  - java.sql.Date dateSql=new java.sql.Date();

Remarque

- Pour bien comprendre le rôle des imports, pensez qu'il est essentiellement syntaxique : on peut supprimer les imports de n'importe quelle classe en mettant le nom complet des classes dans le code. Ils ne sont donc jamais indispensables, ils évitent juste des écritures plus lourdes.

Utilisation concrète en Java

- Chaque classe doit définir dans sa première ligne le package auquel elle appartient :
  - package fr.unicaen.l3.monPaquet
- Comme déjà vu, le fichier correspondant doit être placé dans une arborescence finissant par /fr/unicaen/l3/monPaquet
- Dans les lignes suivantes, on fait tous les imports souhaités
- Puis on rédige le code de la classe

Lancer le main() d'une classe située dans un package

- Attention, il faut se placer dans le répertoire à partir duquel on voit la racine de l'arborescence de packages, dans le répertoire où la compilation a été faite
- Dans notre exemple, ce répertoire contient donc un répertoire fr qui contient un répertoire unicaen, etc.
- Puis on lance avec :
  - java fr.unicaen.l3.monPaquet.MainClass