Classes et Patterns

DUT M3105 : Conception et programmation objet avancées

Serge Rosmorduc
serge.rosmorduc@lecnam.net
Conservatoire National des Arts et Métiers

2019–2020

Principes d'architecture orientée objet

Principes d'architecture OO

- Le but recherché ;
- Comment bien concevoir ses classes et les combiner ;
- Comment définir une classe ;
- Les outils techniques ;

Bibliographie

- Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, « Design Patterns : Elements of Reusable Object-Oriented Software », Addison-Wesley, 1994
- Bertrand Meyer, « Object Oriented Software Construction », Prentice Hall, 1997
- Eric Evans, « Domain Driven Design », Addison-Wesley, 2003
- Emmanuel Puybaret, « Les Cahiers du Programmeur Swing », Eyrolles, 2006 : présente le développement d'une application conséquente en utilisant les méthodes agiles et les patterns.
- `https://martinfowler.com/` : blog très complet, par l'un des « papes » de la conception orientée objet ;
- `http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod` ; pages d'Uncle Bob, alias Robert C. Martin, une autre personnalité du domaine ;

Buts de l'architecture

Concevoir des logiciels qui :

- répondent au cahier des charges !
- soient fiables ;
- soient faciles à développer : éléments (classes) de petite taille, testables individuellement ;
- soient faciles à faire évoluer : une modification des spécifications doit dans l'idéal concerner peu de classes ;
- soient éventuellement réutilisables dans d'autre contextes.

Qu'est-ce qu'une classe bien définie

- la classe représente **un** concept et le représente bien ;
- donc : on préfère avoir beaucoup de petites classes simples plutôt que peu de très grosses classes ;
- proche de la notion de 3ième forme normale en BD ;
- moins une classe dépend des autres, plus elle est facile à comprendre et à tester.

Quelques outils et principes

Type Abstrait de Données (*Abstract Data Types*)

Définition

Un type de donnée abstrait est défini par

- une liste d'opérations ;
- des propriétés formelles sur ces opérations

C'est la spécification d'une classe sans faire référence à son implémentation.

Bibliographie

- Bertrand Meyer (1988,) *Conception et programmation orientées objet*
- Barbara Liskov & Stephen Zilles (1974) *Programming with abstract data types*

Exemple : définition abstraite d'un tableau de double

Opérations

- creer(taille : entier) : tableau de double
- taille() : entier
- get(i : entier) : double
- set(i : entier, v : double)

Spécifications

- `t= creer(a)` a comme précondition $a \geq 0$ et comme postconditions : $t.taille() = a \wedge \forall i, (0 \leq i < a) \Rightarrow t.get(i) = 0$
- v= t.get(i) a comme préconditions $0 \leq i < t.taille()$
- t.set(i,v) a comme précondition $0 \leq i < t.taille()$ et comme postcondition $t.get(i) = v$

SOLID

- **S**ingle Responsability Principle ;
- **O**pen/Closed Principle ;
- **L**iskov Substitution Principle ;
- **I**nterface Segregation Principle ;
- **D**ependency Inversion Principle ;

Webographie

http ://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Single Responsability Principle

Définition

A class should have one, and only one, reason to change (Uncle Bob)

- Une classe doit avoir une seule responsabilité.
- proche de la notion de forme normale en BD

Par ailleurs, un autre principe, légèrement différent, est énoncé par Meyer (OOSC) : Single Choice.

Principe Ouvert/Fermé

Définition

- une classe doit être ouvert à l'extension : on doit pouvoir lui rajouter des fonctionnalités (par héritage, composition, modification du code...)
- une classe doit être fermé à la modification de son *contrat* : le code extérieur qui utilise la classe doit *continuer à fonctionner sans modifications* même si on modifie la classe.

Héritage et Principe de Substitution de Liskov

En java, dire que la classe B hérite de la classe A implique que tout B **est_un** A.

- si b est de classe B, on peut écrire `A a= b;`
- tout ce qu'on peut faire sur un objet de classe A, on peut le faire sur B ;
- une méthode de B qui étend une méthode de A :
  - ne peut normalement pas imposer de restrictions supplémentaires sur ses arguments ;
  - peut éventuellement s'imposer des limitations sur le type des données retournées (covariance).
- donc, si la classe Oiseau a une méthode voler(), on a un problème pour faire descendre Autruche de Oiseaux !
- en java, une méthode d'une sous classe peut déclarer comme résultat une sous classe du résultat de la méthode qu'elle redéfinit ; c'est la **covariance** ;
- l'inverse, pour les *arguments,* appelé **contravariance,** ,n'est pas géré par java (en scala, si).

Interface Segregation Principle

Définition

Créer des interfaces à grain fin qui sont spécifiques aux clients

- À la limite, quand une classe A utilise un objet à travers une interface B, B ne devrait pas montrer à de méthodes dont celui-ci n'a pas besoin.
- par exemple, si je dois simplement consulter un objet, mais pas le modifier, je n'ai pas besoin de voir ses setters.
- le monde informatique est plein d'exemples où ces considérations sont négligées. . .

Dependency Inversion Principle

Définition

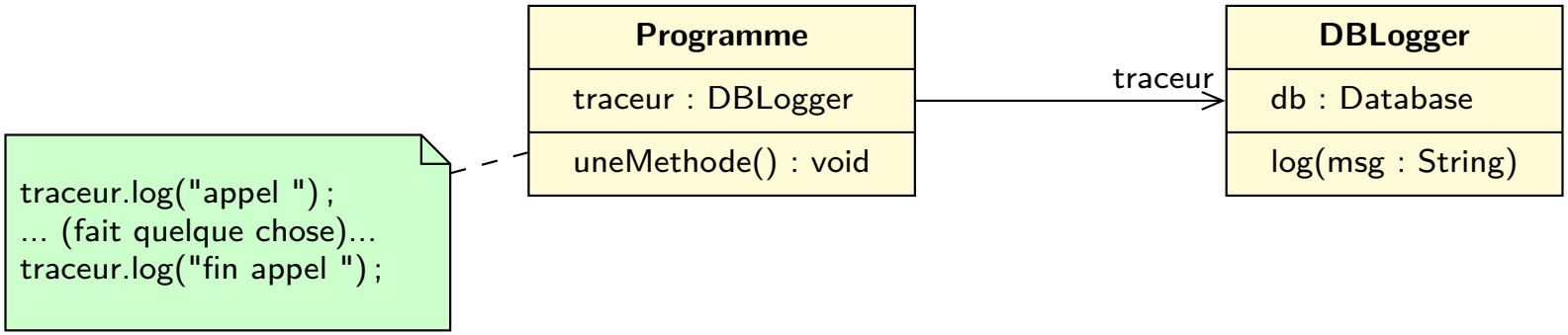
Une classe doit faire référence à des abstractions, non à leur implémentation.

- Quand une classe A dit qu'elle a besoin d'une collection pour travailler, la façon dont la collection est implémentée ne nous intéresse pas ;
- ce qui nous intéresse, c'est la collection des méthodes disponibles, pas la manière dont celles-ci sont écrites ;
- boîtes noires ; interfaces plutôt que classes.
- En suivant ce principe, une classe java ne devrait pas dépendre d'autres classes java, mais seulement d'**interfaces**.

- En pratique : cette règle s'applique surtout entre différentes couches d'un logiciel ;
- Si la classe `Facture` a besoin d'une `Adresse`, on ne va probablement pas créer une interface pour `Adresse`

Coder pour des interfaces, non pour des classes

Variante : *dépendre des abstractions, non des implémentations*
(en conception objet, ils aiment bien les slogans).
Un exemple : dans la classe `Programme`, on souhaite enregistrer (logger) les actions réalisées par l'utilisateur d'un programme dans une base de données.
On a donc deux classes :

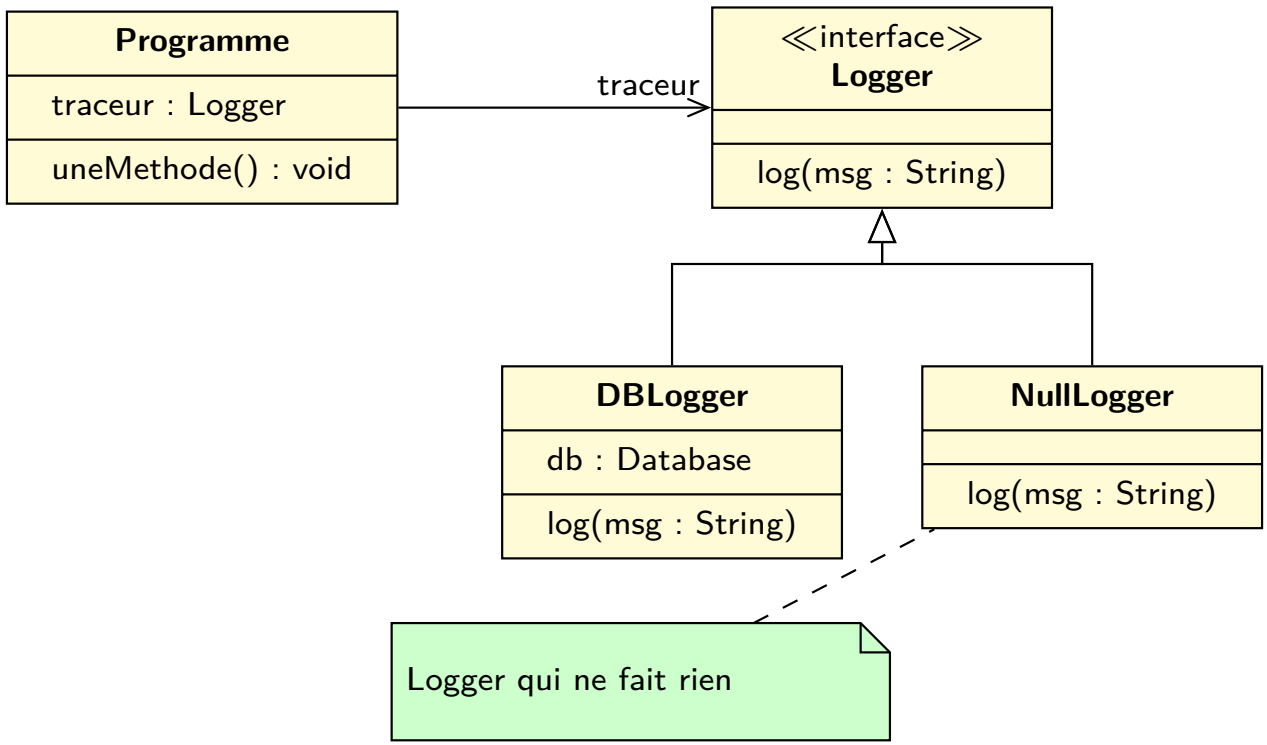


Problèmes :

- pas pratique pour les tests, il faut une `Database` ;
- et si on veut envoyer les logs ailleurs ?

Coder pour des interfaces, non pour des classes

Solution : Programme ne connaît que l'interface de `DBLogger`...



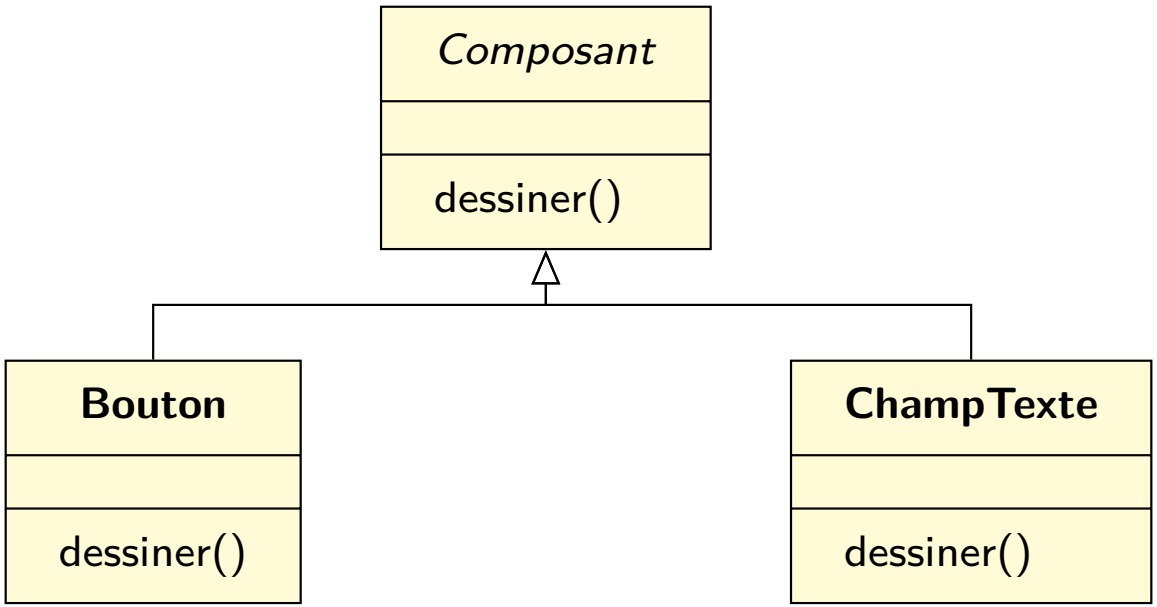
- avantage : beaucoup plus souple et facile à tester ;
- la classe Programme *ne dépend plus de la classe DBLogger* ! ! !
- inconvénient : complexifie le code.

Délégation

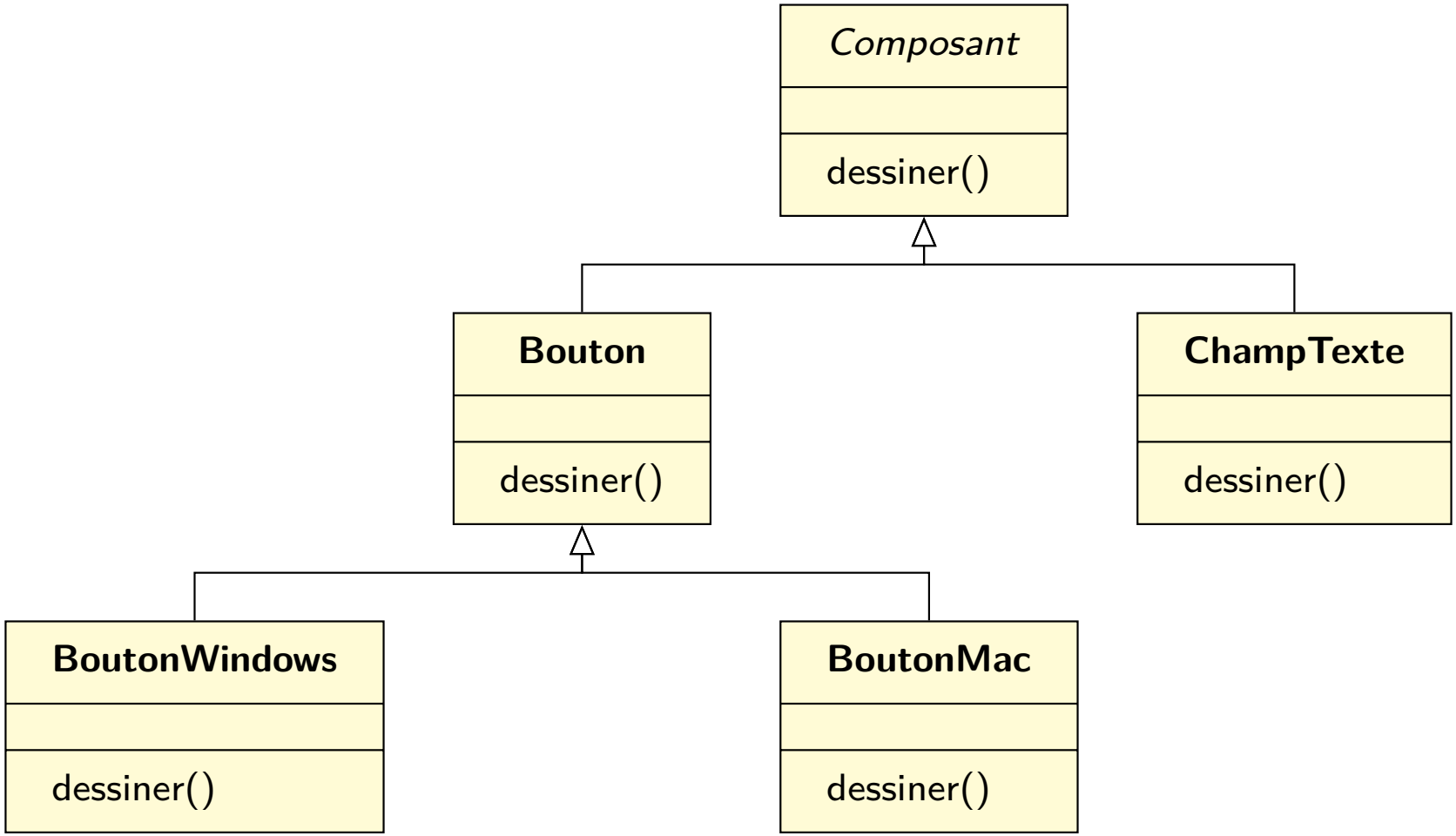
Définition

La délégation, c'est de faire faire le travail par quelqu'un d'autre.
Pour un objet, c'est faire réaliser une opération par un de ses composants.

Exemple : soit une bibliothèque d'interface graphique. On veut pouvoir avoir un aspect différent des objets sur Mac et sur PC



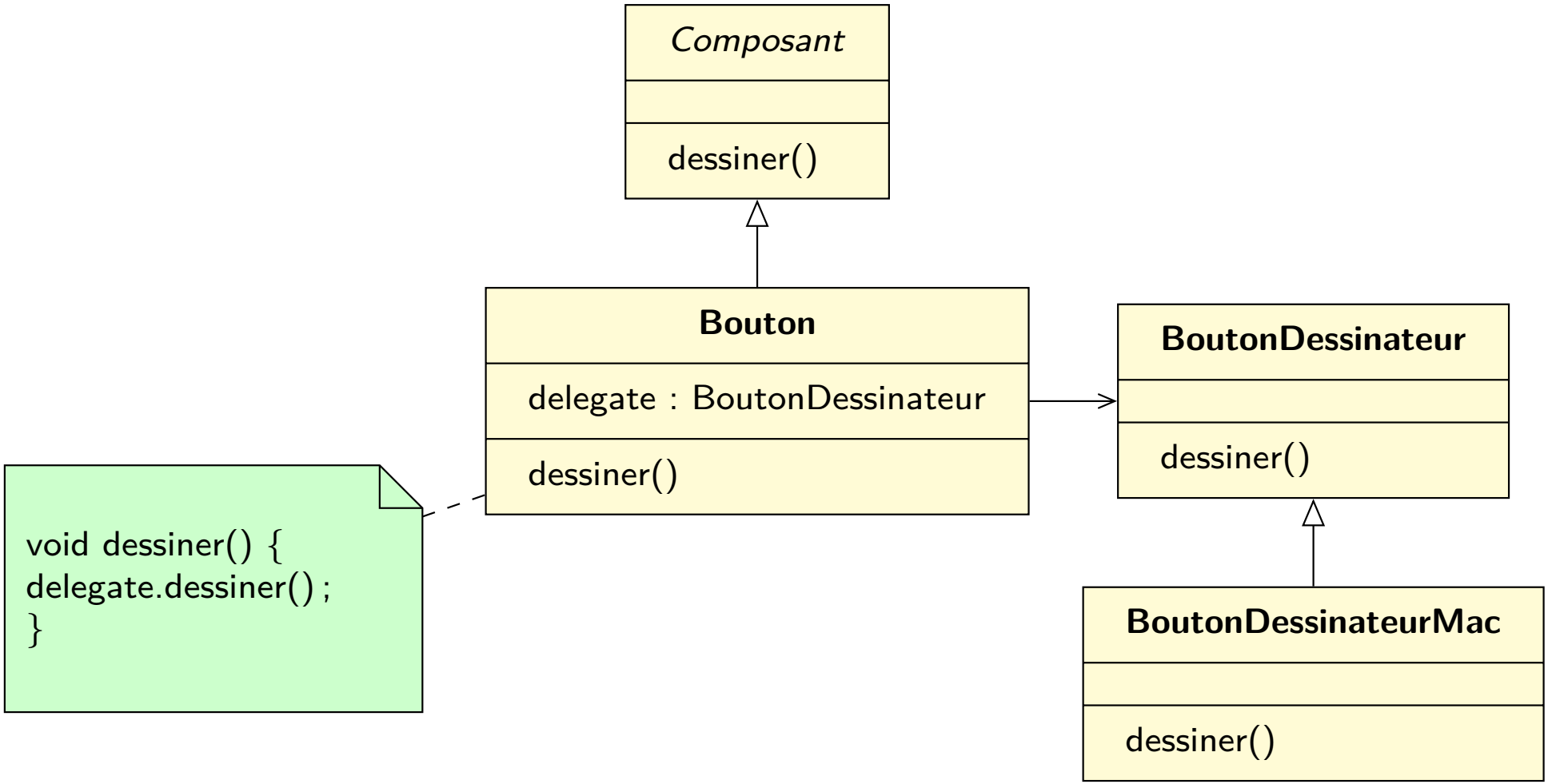
Solution 1 : avec l'héritage.



Solution 1 : héritage

- assez simple à comprendre et à configurer.
- pas souple si on veut pouvoir changer d'aspect « à la volée » !
- ici on a deux axes (type de composant et skin) ; avec cinq composants et deux skins, ça nous donnerait 10 classes ; si on ajoutait un nouveau type de variations avec 9 possibilités : $90 = 5 \times 2 \times 9$ classes, etc...

Solution 2 : avec la délégation



Solution 2 : délégation

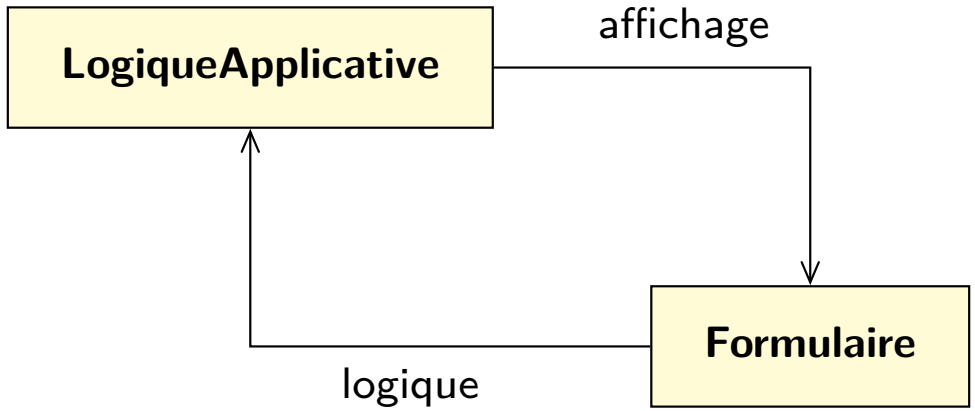
- Plus souple : on peut changer de solution de dessin à la volée si on veut ;
- pas d'explosition combinatoire : on a à peu près le même ordre de grandeur avec deux types de variations (les composants/les skins), mais si on ajoute un nouveau type de variation avec 9 possibilités, on aura $5 \times 2 + 9 = 19$ classes au lieu de 90 !
- plus complexe à comprendre : si on délègue tout, il devient difficile d'aborder le système (exemple : Swing).

Notion de couplage

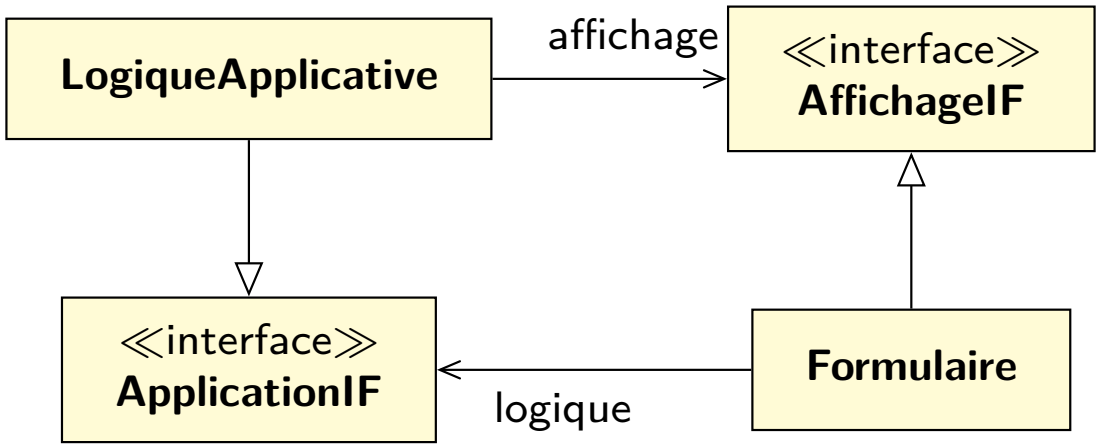
- *limiter* les cycles dans le schéma UML ;
- pourquoi : parce qu'ils empêchent de travailler séparément sur les classes ;
- proscrire absolument : les cycles entre packages ;
- outil pour supprimer un cycle :
  - abstraire les éléments par des interfaces ;
  - ou redécouper l'application pour briser la symétrie.
- il n'est pas interdit d'avoir des cycles, mais ça lie très fortement les classes en question.

Suppression du couplage

Avant :



Après :



Les Design Patterns

Définition

Un design pattern est un modèle d'organisation de classes pour accomplir un certain type de travail.

Il ne s'agit donc *a priori* pas de classes toutes faites, mais de guides architecturaux pour combiner les classes

Éléments d'un pattern

Nom : vocabulaire commun pour les programmeurs

Motivation décrit le problème que le pattern résoud ;

Description forme générale du pattern ; variantes et implémentations.

Conséquences avantages et les inconvénients de ce pattern.

Un exemple simple : le Singleton

Motivation

Certains objets doivent exister en un seul exemplaire : par exemple un Spooler d'imprimante ; l'accès à une BD dans certaines applications.

- On veut garantir l'existence d'une instance unique de cet objet ;
- On veut aussi pouvoir *récupérer* facilement l'objet en question (par exemple sans devoir le transporter d'une classe à l'autre).

Description

Voici une solution en java :

```
public class PrinterSpooler {
  private static PrinterSpooler instance=
     new PrinterSpooler();
  public static PrinterSpooler getInstance() {
     return instance;
  }
  private PrinterSpooler() {....}
     ...
  }
  public void printDoc(String text) {....}
}
```

Utilisation :

```
PrinterSpooler s= PrinterSpooler.getInstance();
s.printDoc("salut!");
```

Remarque et exemples

- le nom de la méthode `getInstance()` fait partie du pattern ;
- usage très fréquents dans les bibliothèques java
  - sans toujours suivre les conventions de nommages ;
  - avec parfois des « faux » singletons (`getInstance()` retournant une valeur « par défaut »)
  - exemples :

```
1 Console c= System.console();
2 Calendar rightNow = Calendar.getInstance();
```

- une autre solution pour créer un Singleton en java, c'est d'utiliser un `enum`.

Singleton : conséquences

- en fait : une variable globale un peu plus sophistiquée que la moyenne ;
- facile à réaliser ;
- **pas adapté du tout** au test (on ne peut pas le remplacer facilement par un « mock » ;
- meilleures solutions : entrepôts (**JNDI**), ou **Injection de dépendances** (Spring, Guice)
- à la limite de *l'anti-pattern*.

Un exemple plus complexe : Le Composite

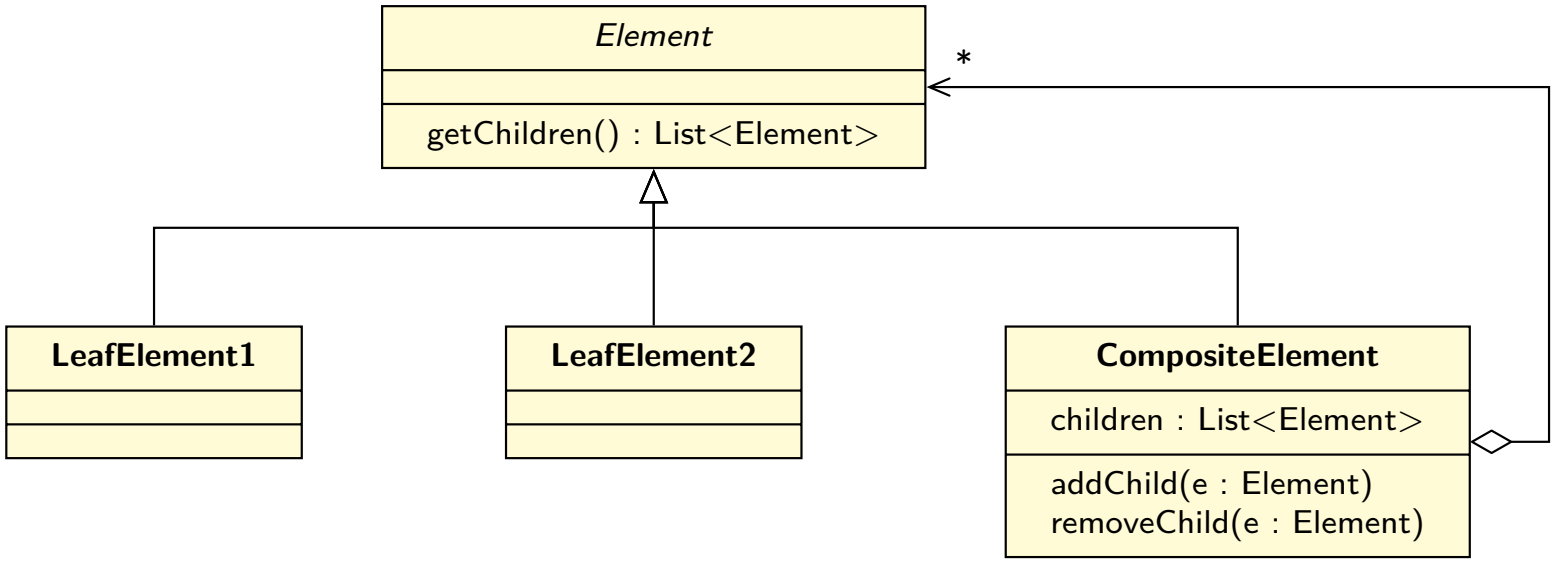
Motivation

On manipule un objet complexe, composé d'éléments ; certains de ces éléments sont eux-même composés d'éléments.

Exemples très nombreux :

- fenêtre d'interface graphique ;
- dessin vectoriel (formes et groupes de formes) ;
- textes formatés ;
- documents XML ;

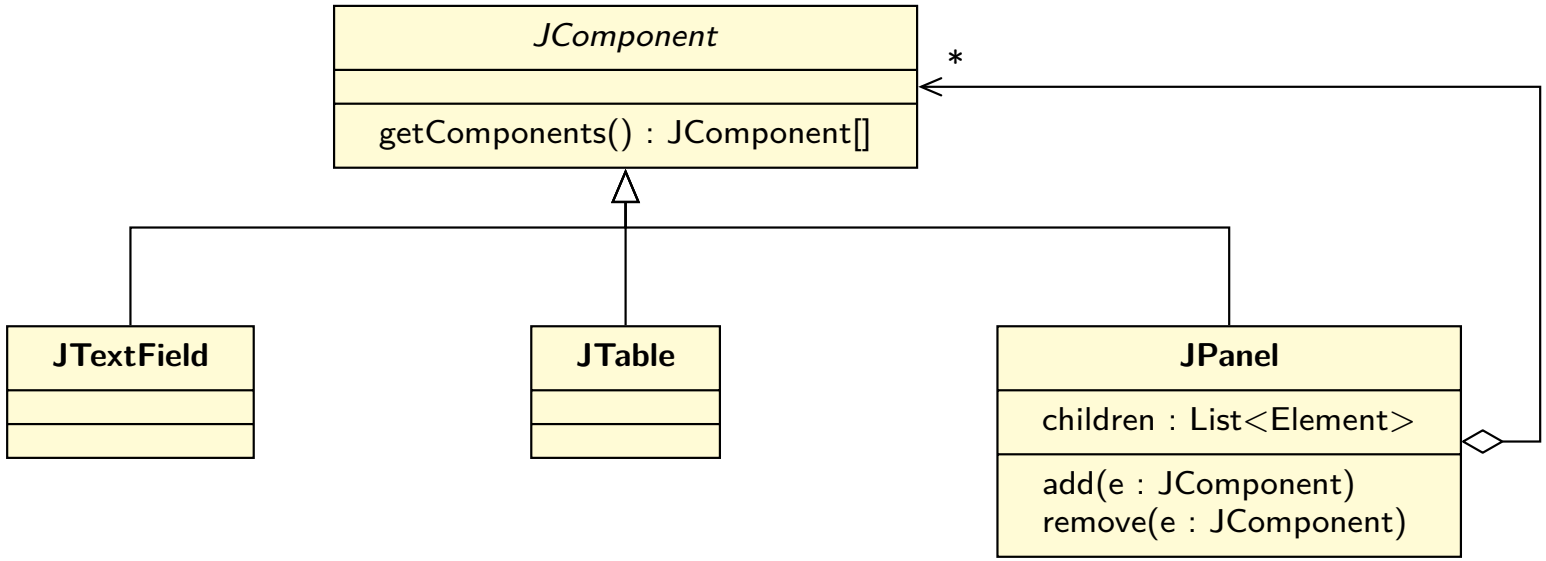
Composite : description



- beaucoup de variantes possibles ;
- `getChildren()` de Element renvoie une liste vide ;
- dans le pattern d'origine, `addChild` est placé sur `Element` (mais ça n'est pas logique) ;

Composite en Swing

(vue très simplifiée)



Composite, suite...

Le placement de la méthode `getChildren()` à la racine permet un parcours récursif facile :

```
public static void parcourirEtFaire(Element e) {
  ... traiter e ...
  for (Element child: e.getChildren()) {
    parcourirEtFaire(child);
  }
}
```

Exemples

- org.w3c.dom.Node and Element
- bibliothèque `JDOM` ;
- composants AWT/Swing : `Container` ;

Utilisation des patterns

- recettes de cuisine ;
- langage commun entre développeurs ;
- élément de compréhension d'une bibliothèque ;
- élément structurant dans l'architecture d'un programme : *refactoring to patterns*

Tous les patterns du GoF

Classification des patterns

- Le livre du GoF classe les patterns en trois catégories ;
- elles sont étendues par d'autres ouvrages (par exemple : architectures concurrentes) ;
- Ces catégories sont :

  création : des patterns qui gèrent la création de nouveaux objets ;

  structure : des patterns qui décrivent des manières de construire des objets complexes ;

  comportement : des patterns qui se focalisent sur la réalisation d'un comportement souhaité.

Les patterns de création

Gèrent la création d'objet. En gros, qui appelle `new` ?

- permettent d'appliquer la règle *coder pour des interfaces, pas pour des implémentations* ;
- délèguent la création d'objets à des spécialistes.

Abstract Factory fabrique pour une hiérarchie d'objets ;

Builder constructeur pour un objet composite ;

Factory Method délégation de la construction à une méthode ;

Prototype création par copie d'un objet existant ;

Singleton création d'une classe avec instance unique.

Les patterns structuraux

Décrivent la manière de composer des classes en des structures plus importantes

Adapter : adapte une classe existante à une nouvelle interface (lien entre deux bibliothèques par exemple) ;

Bridge : permet de faire évoluer séparément une interface et son implémentation ;

Composite représente des hiérarchies de composants avec une notion de groupes ;

Decorator attache dynamiquement des responsabilités supplémentaires à un objet ;

Facade : simplifie l'accès à une couche logicielle en en cachant la complexité ;

Flyweight : réutilise des objets le plus possible pour réduire l'occupation mémoire ;

Proxy : permet de différer la création effective d'un objet sans compliquer outre mesure le reste du programme.

Les patterns comportementaux (Behavioural patterns)

S'intéressent à la communication et à la collaboration d'objets pour accomplir une tâche.

Chain of Responsibility : permet de laisser se propager une requête, en donnant l'opportunité aux membres d'une structure de la traiter ou non ;

Command : réifie une commande pour pouvoir la stocker, la rejouer ou l'annuler ;

Interpreter : représente un programme pour pouvoir l'exécuter ;

Iterator : découple le parcours d'un ensemble d'objet de son implémentation ;

Mediator : permet de faire communiquer un ensemble d'objets tout en réduisant leur couplage ;

Memento : permet de stocker un état d'un objet et de le restaurer sans rompre l'encapsulation ;

Les patterns comportementaux (Behavioural patterns)

Observer : permet à un objet de prévenir quand il est modifié ;

State : permet de gérer les cas où le comportement d'un objet est sensiblement modifié selon son état ;

Strategy : permet de choisir entre plusieurs algorithmes pour un traitement ;

Template Method : permet de réaliser une séquence d'opérations, dont certaines seront définies dans des sous classes ;

Visitor : permet de parcourir une hiérarchie d'objets sans briser l'encapsulation.