

# Génie logiciel

Yannick Loiseau  
yannick.loiseau@uca.fr

Université Clermont Auvergne

Licence informatique 3<sup>e</sup> année

# Introduction

# Génie logiciel ?

*Software Engineering*

artisanat  $\Rightarrow$  industriel

- ▶ méthodes
- ▶ processus
- ▶ pratiques
- ▶ techniques
- ▶ outils

$\rightarrow$  rigueur efficacité  $\Rightarrow$  logiciels de qualité

# Qualité ?

**bien** réaliser le logiciel

- ▶ stable
- ▶ robuste
- ▶ performant
- ▶ facile à utiliser
- ▶ sans bug
- ▶ maintenable
- ▶ ...

⇒ architecture : conception

⇒ code : réalisation

# Qualité ?

réaliser le bon logiciel

- répond aux besoins
- besoins changeants
- adapté au contexte

⇒ analyse fonctionnelle & contextuelle

# Champs d'application



# Aspect organisationnels

- ▶ délais
- ▶ coûts
- ▶ planification
- ▶ affectation des tâches
- ▶ gestion humaine
- ▶ communication



# Aspect techniques

- ▶ formalisation
- ▶ conception
- ▶ spécification
- ▶ réalisation
- ▶ outils
- ▶ qualité

# Contenu

# Cours

Architecture, conception, modélisation

- ▶ UML
- ▶ Principes de conceptions
- ▶ Design Patterns

10h30

# TD

- ▶ exercices sur le cours
- ▶ sujets  $\rightarrow$  ENT  $\Rightarrow$  préparation

10h30, 2 groupes

# TP

techniques et outils

- ▶ gestion de version
- ▶ automatisation
- ▶ tests unitaires
- ▶ ...

sujets → ENT

12h, 3 groupes

## Cycle de vie du logiciel

## Définition (Cycle de vie)

- ▶ tâches
- ▶ organisation

## Tâches

éditeur  $\neq$  *ad-hoc*

- analyse
- décomposition
- conception
- réalisation
- intégration
- tests
- déploiement
- maintenance



# Analyse

- ▶ faisabilité
- ▶ besoins
- ▶ contraintes
- ▶ problèmes
- ▶ existant
- ▶ analyse fonctionnelle

# Décomposition

- ▶ sous problèmes
- ▶ composants indépendants
- ▶ architecture

# Conception

- ▶ détails des solutions
- ▶ modélisation
- ▶ choix technologiques

# Réalisation

- ▶ production de code
- ▶ assurance qualité
- ▶ vérification

# Intégration

- communication des composants
- fonctionnement global

# Tests

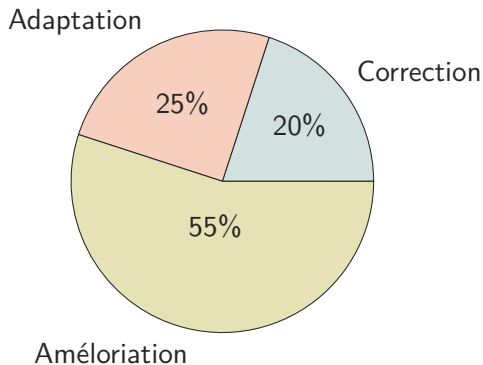
- ▶ fonctionnalités
- ▶ performances
- ▶ robustesse
- ▶ respect des besoins (acceptation)

# Déploiement

- ▶ mise en place
- ▶ exploitation / production
- ▶ migration

# Maintenance

+90% du coût





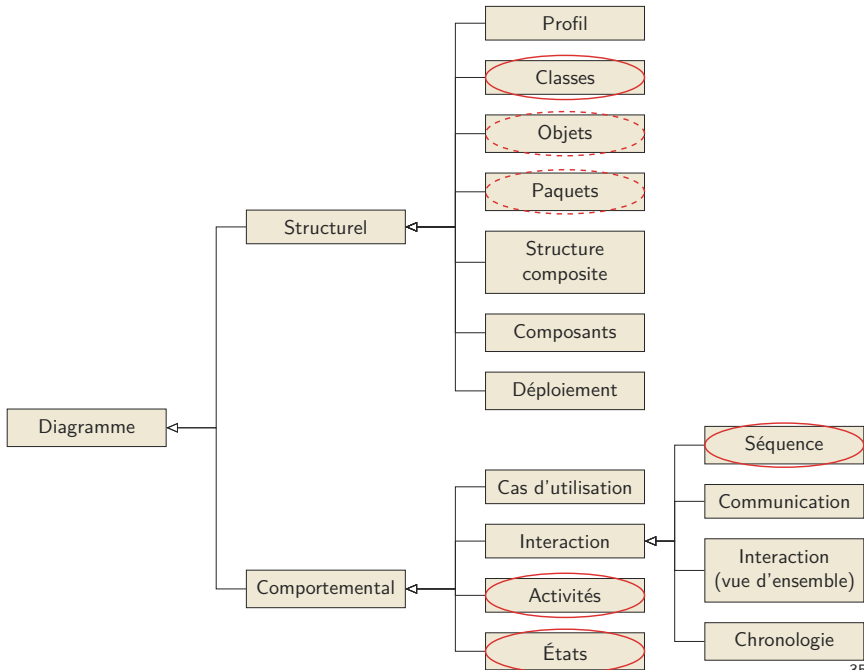
# Modélisation

# UML

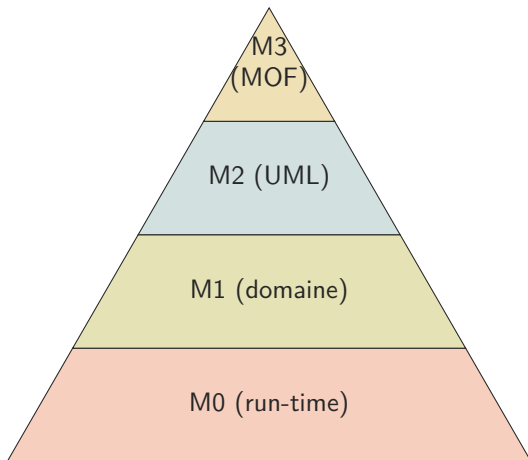
Unified Modeling Language

<http://www.uml.org/>

- ▶ langage modélisation
- ▶ graphique
- ▶ objet
- ▶ indépendant d'un langage
- ▶ conceptuel → implémentation
- ▶ standard : OMG <http://www.omg.org/spec/UML/>
  - ▶ 1.4.2 (ISO/IEC 19501) : 2001
  - ▶ 2.4.0 (ISO/IEC 19505-1 19505-2) : 2011
  - ▶ 2.5 : 2015



Modèle, méta-modèle, méta-métamodèle, . . .

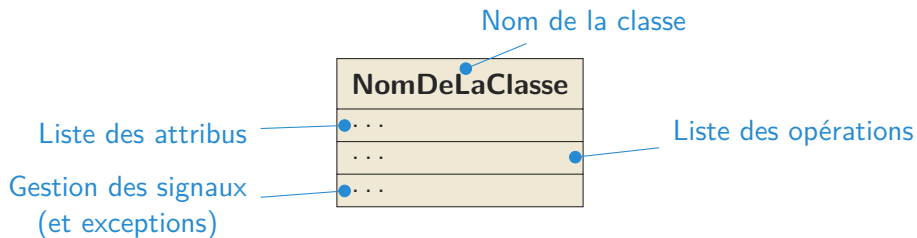


# Modélisation structurelle

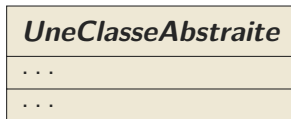
## Diagramme de classes



# Représentation

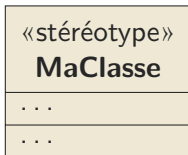


# Abstraite

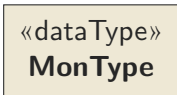
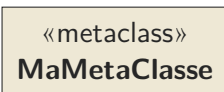


- non instanciable

# Stéréotypes



- ▶ metaclass
- ▶ exception
- ▶ dataType
- ▶ signal
- ▶ control
- ▶ entity
- ▶ boundary
- ▶ ...



MonControl

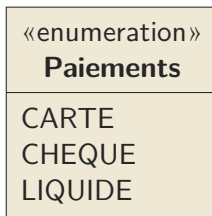


MonEntite



MonInterface

# Énumération



# Attributs

Personne
+ nom : String = "Toto" - dateNaissance : Date {readOnly} # /age : Integer {age > 0} ~ amis : Personne [0..*] {ordered,nonunique}

*visibilité nomAttribut : type [multiplicité] = défaut {propriétés}*

# Multiplicité

## Définition (Multiplicité)

Cardinalité de l'élément  $\Rightarrow$  collection d'instances

- ▶ *min..max*
- ▶ min : 0, 1, ... (default 1)
- ▶ max : 1, \*, ... (default 1)

## Exemple

- ▶ 1..1 : un et un seul élément (default)
- ▶ 3 : au moins 1 et au plus 3 éléments
- ▶ 0..1 : élément optionnel
- ▶ 5..7 : entre 5 et 7 éléments
- ▶ 1..\* : au moins un élément

# Propriétés et contraintes

Sur les collections

- ▶ ordered / unordered
- ▶ unique / nonunique

	<i>ordered</i>	<i>unique</i>
Bag	✗	✗
Sequence	✓	✗
Set	✗	✓
Ordered Set	✓	✓

# Propriétés et contraintes

Sur tout attribut

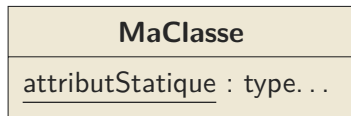
- ▶ readOnly
- ▶ union, subsets *prop*
- ▶ redefines *prop*
- ▶ id
- ▶ attribut dérivé : /age
- ▶ expression : {age > 0}



# Visibilité

- ▶ publique : +
- ▶ paquet : ~
- ▶ protégée : #
- ▶ privée : -

# Attribut statique



- ▶ attribut de la classe
- ▶ commun à toutes les instances

# Opérations

Forme
<ul style="list-style-type: none"><li># dessiner()</li><li>– déplacer(...) : void</li><li>+ getSurface() : Integer {query}</li></ul>

*visibilité nomOpération(paramètres) : type retour [mult.] {prop.}*

# Propriétés d'opération

## Propriétés :

- ▶ redefines *op*
- ▶ query
- ▶ ordered, unique, ...

## Contraintes :

- ▶ pré-conditions
- ▶ post-conditions
- ▶ *body conditions*
- ▶ exceptions

# Paramètres d'opération

## MaClasse

op(a : Integer[0..\*], b : String) : void

op(a : Integer[0..\*] {unique}, b : String = "") : void

op(in a : Integer[0..\*] {unique}, out b : String = "") : void

*direction nom : type [multiplicité] = default {propriétés}*

# Direction ?

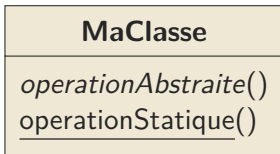
- ▶ **in** (valeur)
- ▶ out (référence)
- ▶ inout
- ▶ return : monOperation(return : String)  $\Leftrightarrow$  monOperation() : String

## Principe : CQS (*Command Query Separation*)

une opération est :

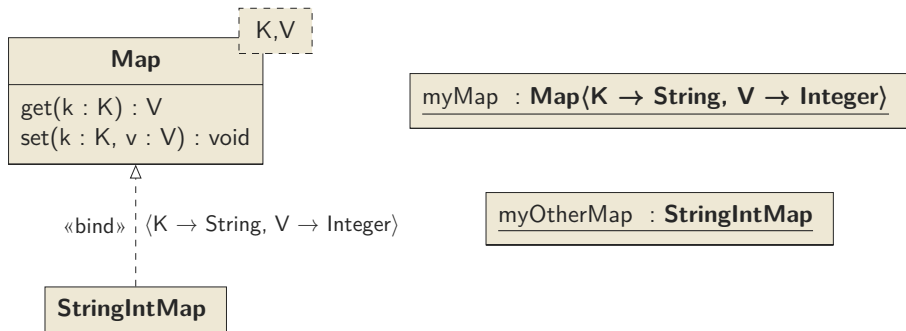
- ▶ une requête (query) sans effet de bord, retourne une valeur
- ▶ une commande (command) avec effet de bord, retourne *void*
- ▶ procédure vs. fonction
- ▶ return vs. output
- ▶ passage par valeur vs. passage par référence
- ▶ contrainte query : transparence référentielle

# Opération abstraite ou statique





# Template



- ▶ Classe avec types génériques  $\Rightarrow$  polymorphisme générique
- ▶ Création de type en fixant les types génériques (*binding*)
- ▶ Association des type à l'instanciation (création de classe anonyme)

```
class StringIntMap extends Map<String,Integer>{}  
StringIntMap myOtherMap = new StringIntMap()  
Map<String,Integer> myMap = new Map<String,Integer>()
```

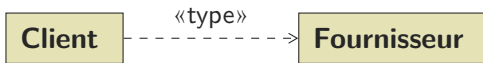
# Relations

# Relations

## Définition

Élément UML liant 2 ou plus éléments

# Dépendance



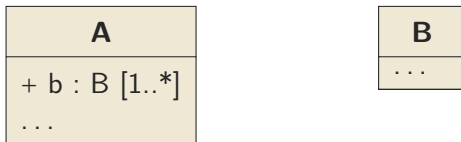
- ▶ Relation orientée
- ▶ *nécessite* ou *utilise*
- ▶ Dépendance *sémantique* ou *structurelle*
- ▶ *couplage* faible (temporaire)
- ▶ types : «create», «call», «instanciate», «send», «use», ...

# Association

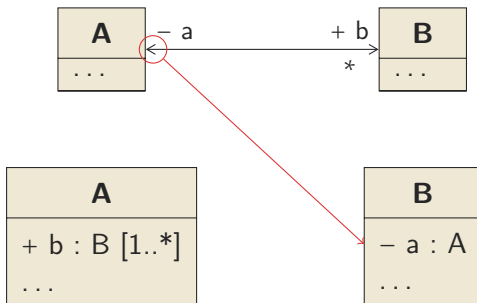


- ▶ relation sémantique
- ▶ multiplicité (propriétés)
- ▶ nom
- ▶ rôles (visibilité)
- ▶ navigabilité

# Navigabilité



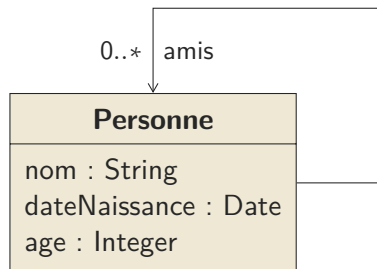
# Navigabilité



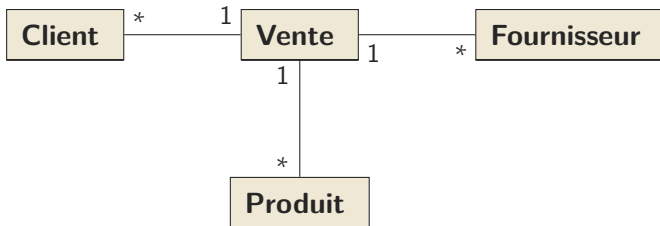
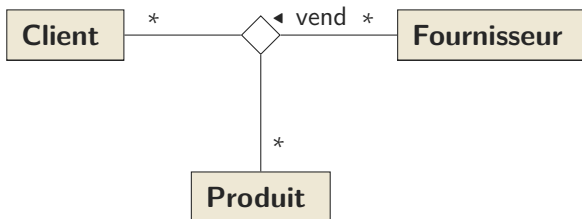
Modifier  $A.b \Rightarrow$  modifier  $B.a$



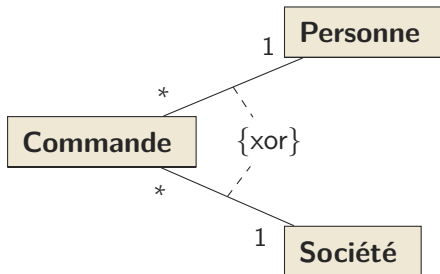
# Association réflexive



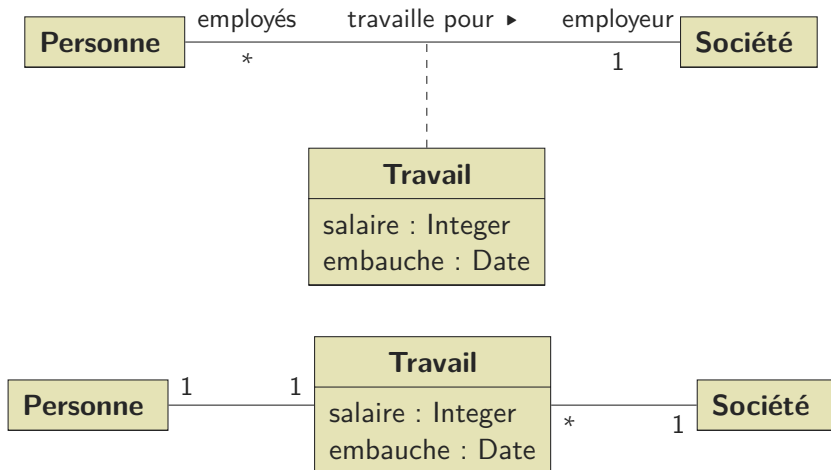
# Association n-aire



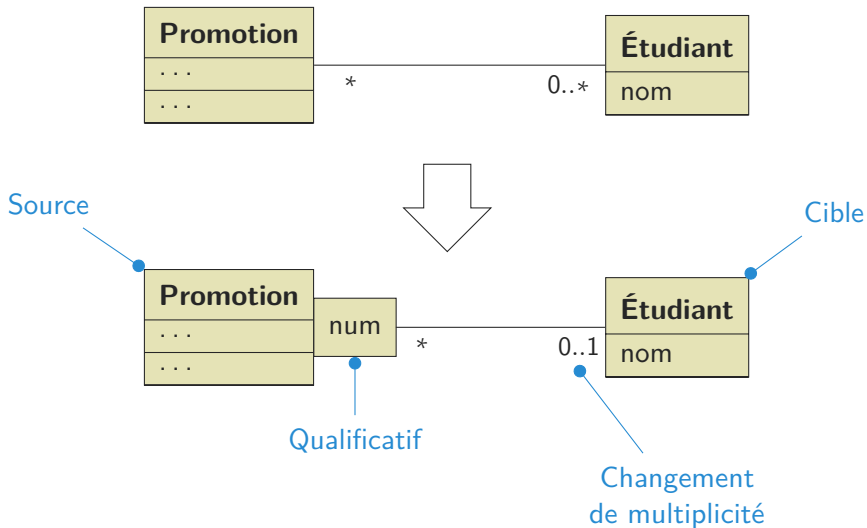
# Contraintes



# Classe d'association



# Association qualifiée

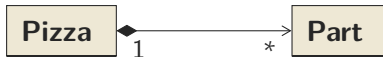


# Agrégation



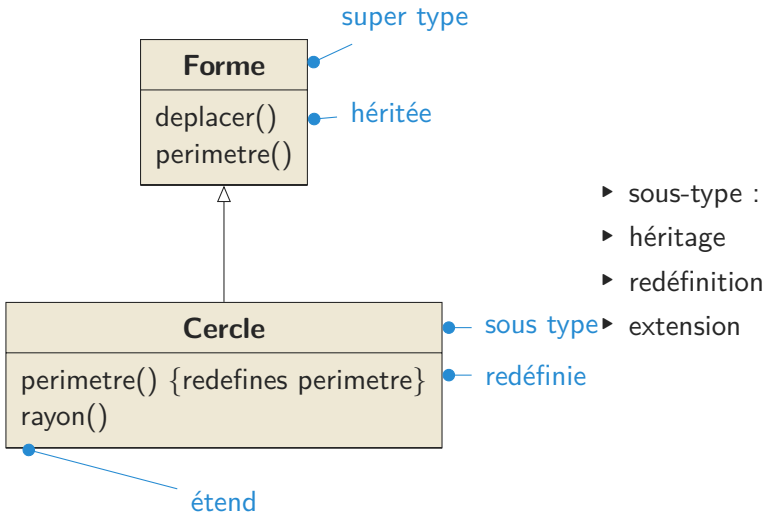
- ▶ collection d'instance
- ▶ interface de type collection
- ▶ indépendance des éléments
- ▶ partage possible

# Composition



- ▶ élément composite
- ▶ gestion de création par composé
- ▶ dépendance des composants
- ▶ pas de partage des composants
- ▶ pas de références extérieures

# Généralisation – spécialisation





# Sous-type

- ▶ remplaçabilité
  - ▶ encapsulation
  - ▶ polymorphisme par sous-typage
- ▶ nominal ou structurel
- ▶ covariance des propriétés

## Principe : Substituabilité Liskov (87)

### Sous-type comportemental

- ▶ *covariance* des sorties (retours, exceptions)
- ▶ *contravariance* des entrées
- ▶ pas de renforcement des *préconditions*
- ▶ pas de relachement des *postconditions*
- ▶ conservation des *invariants*
- ▶ conservation de la *mutabilité* et des *états*

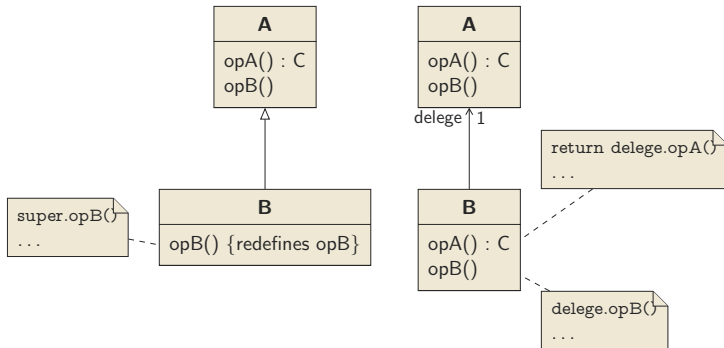
SOLID

## Principe : OCP : open-closed principle (88,96)

- ouverte par extension (héritage, redéfinition)
- fermée à la modification (encapsulation)

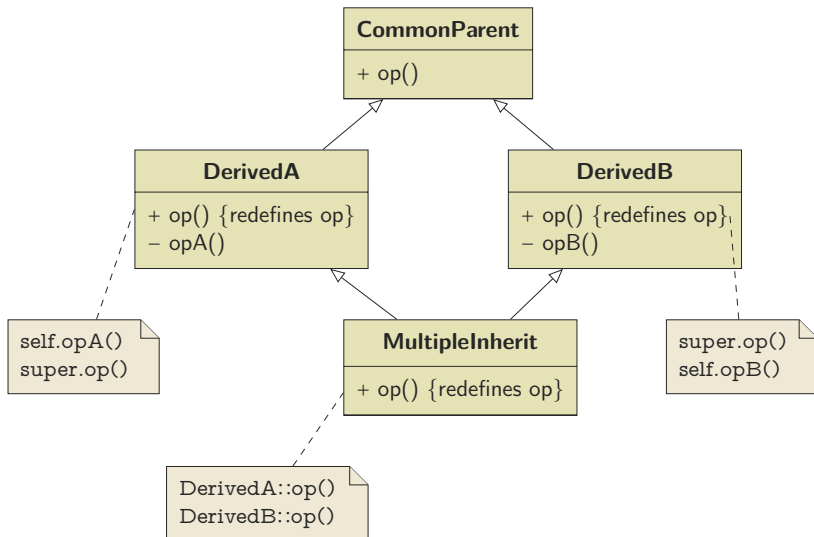
SOLID

# Délégation



- ▶ plus flexible : couplage plus faible
- ▶ plus sélectif
- ▶ gestion « manuelle »

# Héritage multiple

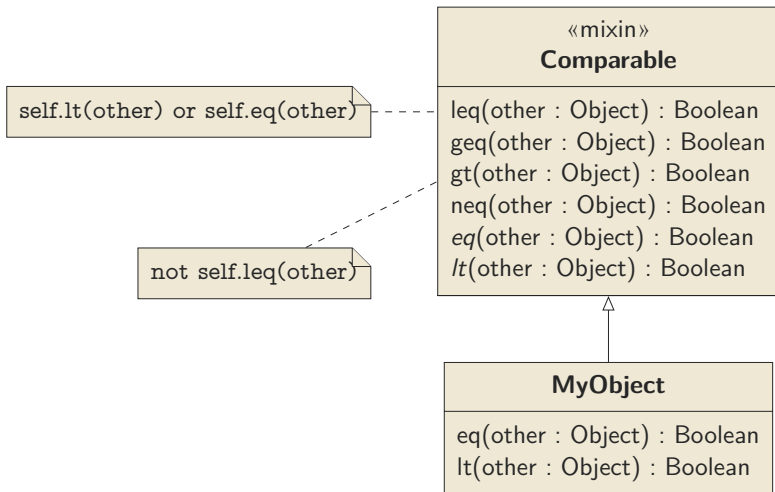


# Mixins

- ▶ classe concrète
- ▶ implémentation partielle
- ▶ non instanciable
- ▶ composable (sans conflit)
- ▶ traits, rôles, ...

# Mixins

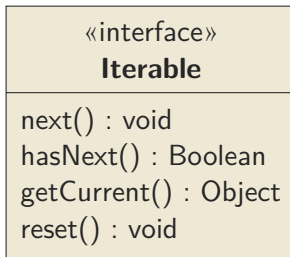
## exemple



# Interfaces

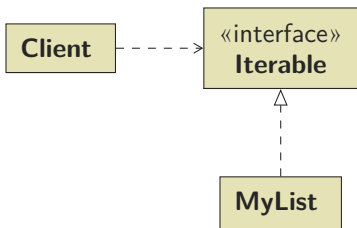


# Interface



- ▶ classe purement abstraite
- ▶ pas d'implémentation
- ▶ spécifie un contract

# Implémentation



- ▶ Une classe respecte la spécification d'une interface
- ▶ Implémente toutes les méthodes
- ▶ Respecte les contraintes
- ▶ Une classe peut implémenter plusieurs interfaces

# Design

## Principe : Ségrégation des interfaces

- interfaces minimales
- dépendance des interfaces
- $\Rightarrow$  découplage

SOLID

# Design

- ▶ interface de rôle minimale → découplage \*able
- ▶ interface complète → tests I\*

# Paquets

# Paquet

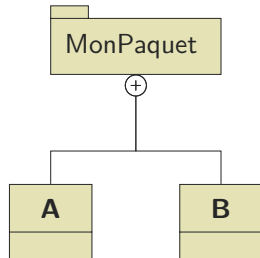
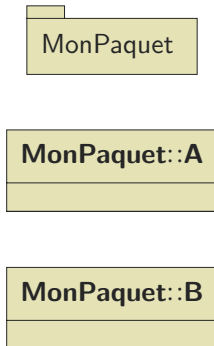
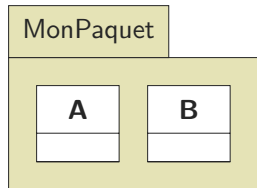
Package

- ▶ groupe
- ▶ espace de nom

# Design

- ▶ minimiser le couplage
- ▶ maximiser la cohésion

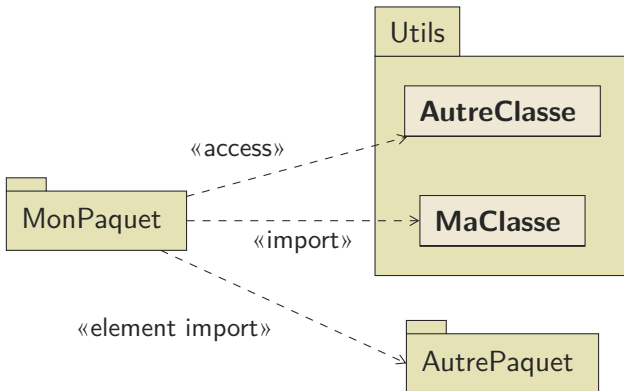
⇒ peu de dépendances entre paquets





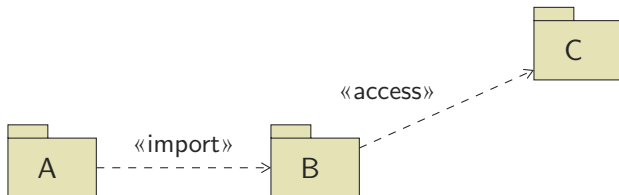
# Import d'éléments

- ▶ Utilisation d'éléments d'un autre paquet
- ▶ Sans qualificatif
- ▶ Possibilité d'alias
- ▶ Visibilité pour la cascade :
  - ▶ Publique : «**import**»
  - ▶ Privée : «**access**»



# Import de paquets

⇒ Import de tous les éléments publics



# Modélisation comportementale

## Diagramme d'état

- ▶ états (objet, composant, système)
- ▶ événement
- ▶ changement d'état
- ▶  $\Rightarrow$  automate

- ▶ automate de comportement
- ▶ automate de protocole



- ▶ événements → état courant
- ▶ état → comportement
- ▶ signaux (asynchrones)

- ▶ protocole : contrat d'utilisation
- ▶ → cycle de vie, ordre d'invocation, . . .

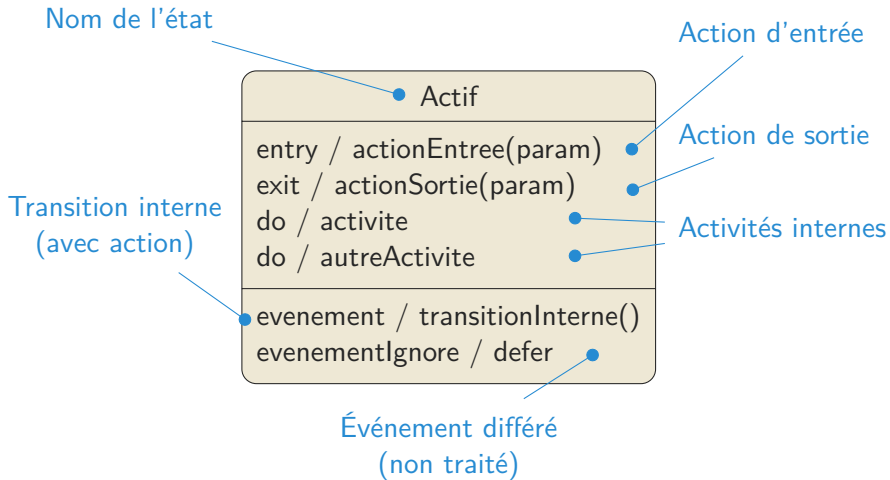
État

## Définition (État)

partie de la vie de l'objet

- ▶ conditions
- ▶ activités
- ▶ événements

événements/actions passés  $\Rightarrow$  état courant

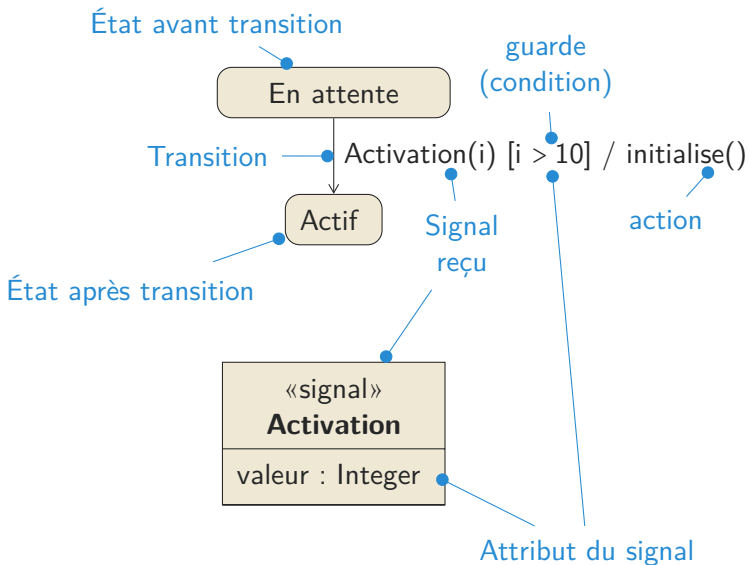


# Événements

- ▶  $\Rightarrow$  changement d'état : transition
- ▶ synchrone ou asynchrone
- ▶  $\rightarrow$  IPC / threads

- ▶ événement **externe** action utilisateur, signal système
- ▶ événement **interne** appel de méthode, exception





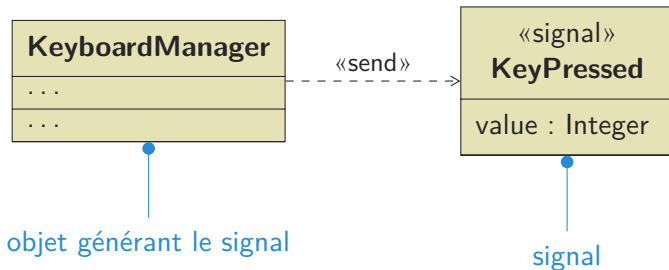
- ▶ signaux : `SignalEvent`
- ▶ invocations : `CallEvent`
- ▶ délais : `TimeEvent` *at*, *after*
- ▶ changements d'état : `ChangeEvent` *when*
- ▶ générique : `AnyReceiveEvent` *all*

# Signaux

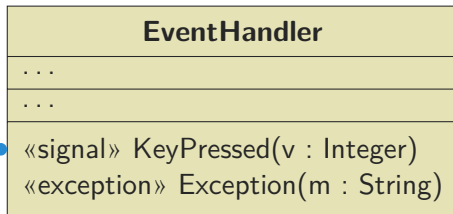
## objet `SignalEvent`

- ▶ envoyé asynchrone
- ▶ reçu
- ▶  $\Rightarrow$  programmation événementielle GUI
- ▶ cas particulier : exception

- ▶ objet → attributs / méthodes
- ▶ stéréotype «[signal](#)»
- ▶ spécialisation → hiérarchie
- ▶ ⇒ polymorphisme



signaux gérés



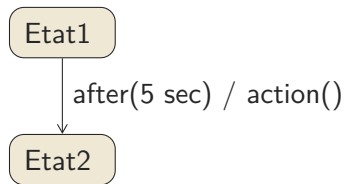
# Appels

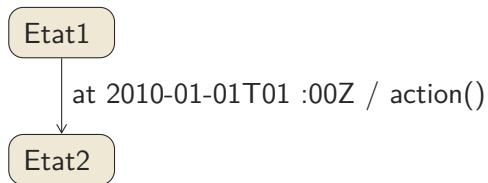


- ▶ invocation d'opération
- ▶ synchrone
- ▶ changement d'état

## Délais et événements temporels

- ▶ moment temporel
- ▶ absolu / relatif
- ▶ **at** / **after**





## Changements d'état

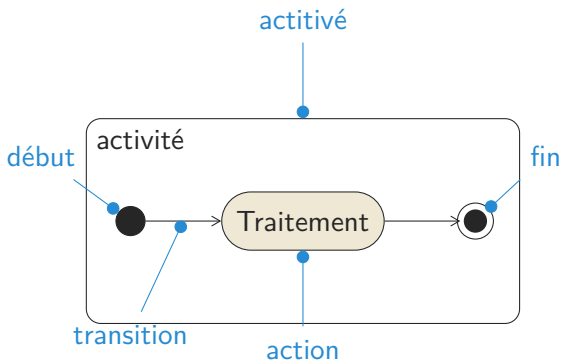
- ▶ modification d'état
- ▶ → contrainte  $\Rightarrow$  transition conditionnelle implicite
- ▶ évaluation continue  $\Rightarrow$  setter, notification (DP observateur)
- ▶ [when](#)

## Diagramme d'activité



- ▶ processus métiers
- ▶ traitements
- ▶ algorithmes (méthodes)

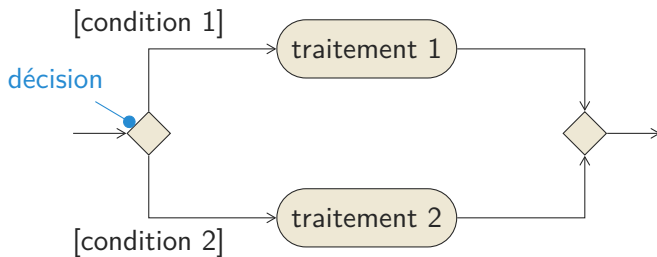
- ▶ ensemble d'activités
- ▶ enchainement
- ▶ conditions



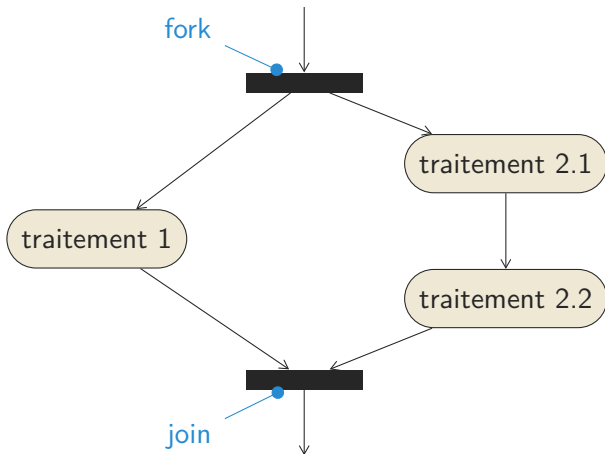
# Sous-activités



# Conditions

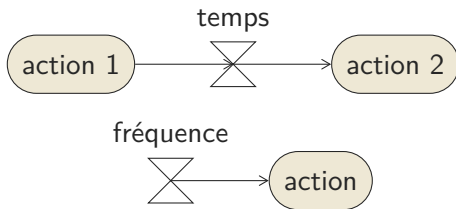


# Parallélisme



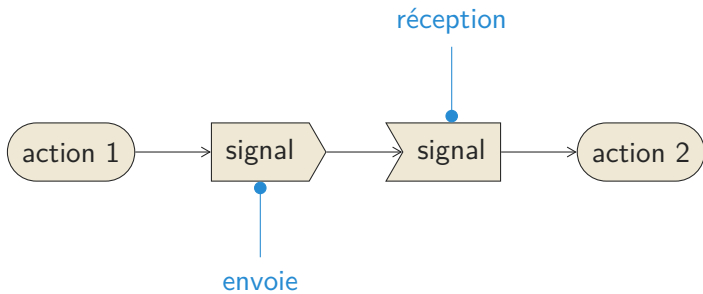
# Événements

temporel



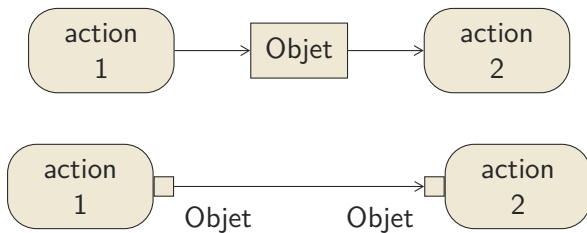
# Événements

signaux



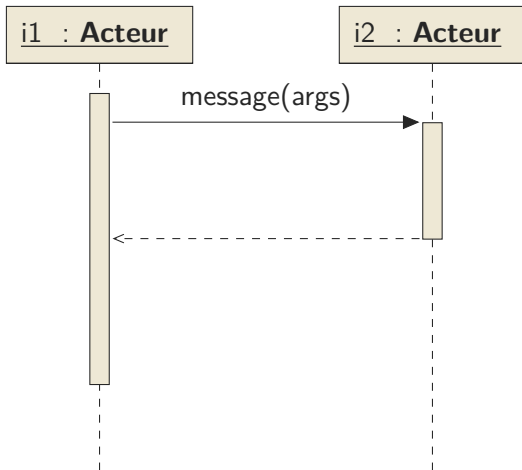


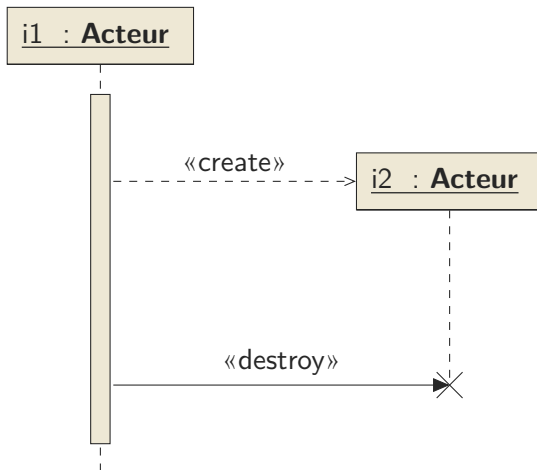
# Flux de données



## Diagramme de séquence

- ▶ interaction  $\Rightarrow$  messages
- ▶ instance
- ▶ objets, composants, ...
- ▶ dimension temporelle





# Patrons de conception

# Quoi ?

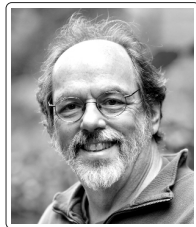
## Définition (Patrons de conception)

- ▶ modèles de conception / architecture
- ▶ solutions éprouvées
- ▶ réutilisable
- ▶ problèmes récurrents
- ▶ modélisation, architecture
- ▶ mise en œuvre

Kent Beck



Ward Cunningham



OOPSLA (Object-Oriented Programming, Systems, Languages & Applications)  
1987



## « Gang of Four »



*Design Patterns : Elements of  
Reusable Object-Oriented Software*  
(1994)

Erich Gamma



Richard Helm



Ralph Johnson

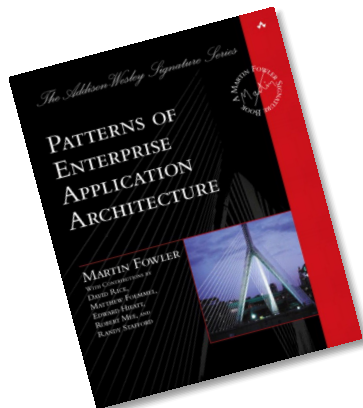


John Vlissides

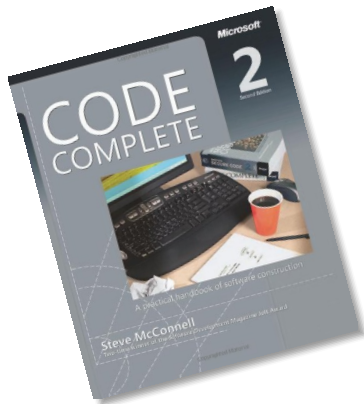


## *Patterns of Enterprise Application Architecture (2002)*

Martin Fowler



Steve McConnell

*Code Complete (2004)*

## ✓ avantages

- ▶ formalisation de bonnes pratiques
- ▶ éviter les erreurs
- ▶  $\Rightarrow$  meilleure efficacité / qualité
- ▶ vocabulaire commun
- ▶  $\Rightarrow$  meilleur lisibilité
- ▶ ↗ flexibilité

## X inconvénients

- ▶ plus complexe (indirection, . . .)
- ▶ performance
- ▶ choix
- ▶ fonctionnalités du langage
- ▶  $\Rightarrow$  intégration

# Classification

- ▶ *creational*
- ▶ *structural*
- ▶ *behavioural*
- ▶ *architectural*
- ▶ *concurrency*
- ▶ ...

# Creational

- ▶ séparation : instanciation vs. utilisation
- ▶ encapsulation de la classe concrète
- ▶ encapsulation de la création / initialisation
- ▶ inversion de dépendance
- ▶  $\Rightarrow$  classe de premier ordre

# Structural

- ▶ simplification des relations
- ▶  $\Rightarrow$  classes et fonctions de premier ordre
- ▶  $\Rightarrow$  modules



# Behavioural

- ▶ communication
- ▶ interaction
- ▶  $\Rightarrow$  fonctions de premier ordre
- ▶  $\Rightarrow$  fonctions d'ordre supérieur
- ▶  $\Rightarrow$  typage dynamique

# Concurrency

- ▶ système concurents
- ▶ threads

# Architectural

- ▶ plus larges
- ▶ architecture, système global
- ▶ performances
- ▶ échelle

# Création

# Singleton

creational

- ▶ existence d'une seule instance
- ▶ point d'accès global

## Singleton

– instance : Singleton = null

...

– «create» Singleton()

+ getInstance() : Singleton

...

# Avantages

- ▶ contrôle d'accès
- ▶ espace de noms clair
- ▶ extensible
- ▶ nombre variable d'instances (*multiton*)

# Variable globale ?

- ▶ pas d'unicité (ou classe anonyme)
- ▶ pollution de l'espace de noms
- ▶ initialisation tôt



# Classe statique

- ▶ problème si instance nécessaire (paramètre)
- ▶ pas extension d'une autre classe, interface
- ▶ pas extensible
- ▶ pas d'initialisation tardive
- ▶ pas d'instances multiples

# Mise en œuvre

Java

```
public class Singleton {  
    private static Singleton instance = null;  
  
    private Singleton() { }  
  
    public static Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

# Mise en œuvre

## Problèmes

- ▶ multi-thread mutex : (synchronized)
- ▶ clone (CloneNotSupportedException)
- ▶ persistance (serialize)

# Extensions

- ▶ « multiton »
- ▶ registre
- ▶ memoization, caching, répartition

# Critiques

état global

- ▶ difficile à tester (couplage fort)
- ▶ masque les dépendances

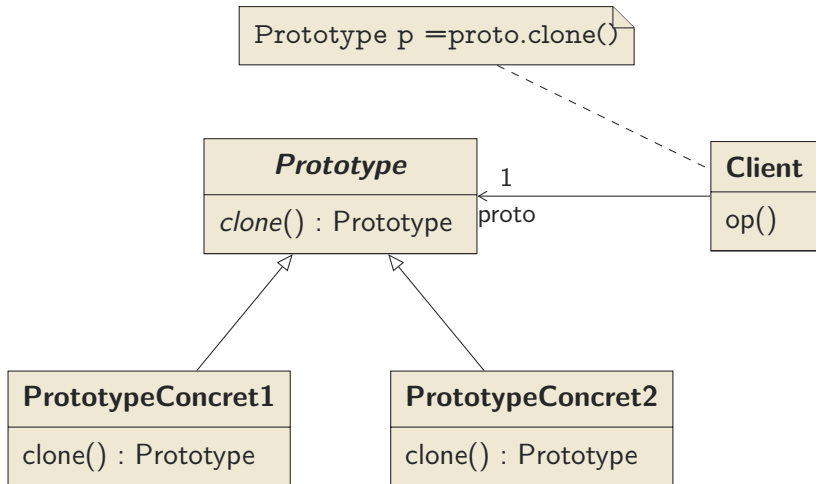
# Prototype

creational

- ▶ instance prototype
- ▶ création par copy (clone)
- ▶ p. ex. Javascript

# Utilisation

- ▶ indépendance création / composition / utilisation
- ▶ type des dépendances spécifié à l'exécution
- ▶ structure de type complexe
- ▶ hiérarchie de types complexe (pas de d'instanciation)
- ▶ peu de différences entre instance





# Avantages

- ▶ ↘ complexité des hiérarchies
- ▶ ↘ coût d'instanciation
- ▶ ↘ couplage (injection de dépendance)
- ▶ simplification d'objets complexes comme façade

# Mise en œuvre

- ▶ *shallow* vs. *deep*
- ▶ aliasing (attribut partagé)
- ▶ références circulaires

# Extensions

- méthode `init`
- registre de prototypes

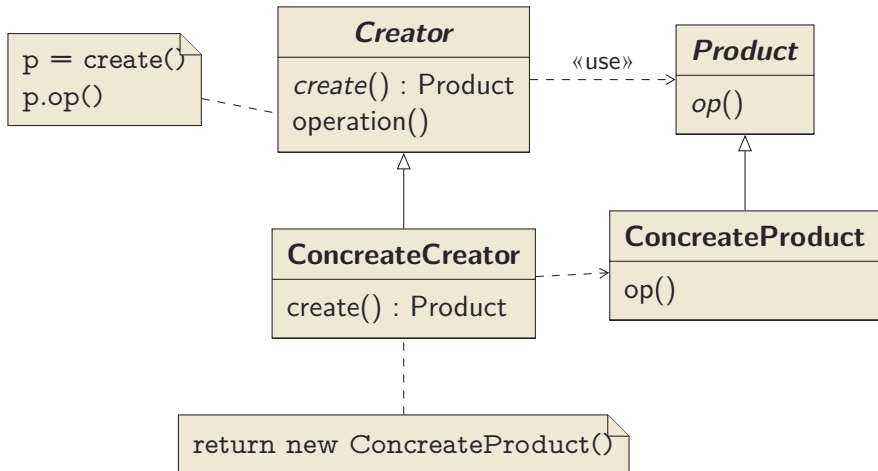
# Alternatives

- ▶ factory (abstract)
- ▶ builder
- ▶ classes de premier ordre

# Factory Method

creational

- ▶ différer l'instanciation
- ▶ type concret inconnu
- ▶ « constructeur virtuel »
- ▶ *framework*



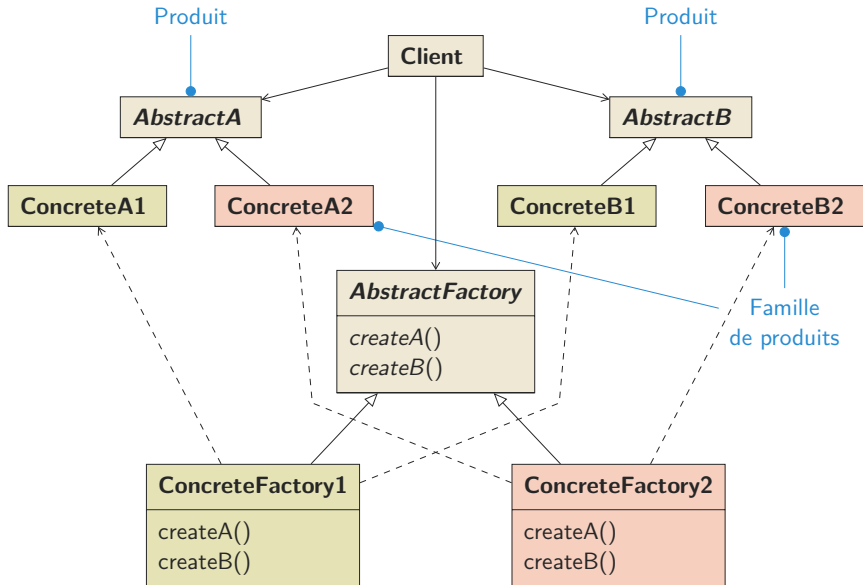
# Mise en œuvre


- ▶ méthode paramétrée
- ▶ classes génériques
- ▶ objet classe

# Abstract Factory

- ▶ création d'une famille d'instance
- ▶ classe concrète inconnue
- ▶ extension de *factory method*





- ▶ classes concrètes isolées
- ▶  $\Rightarrow$  homogénéité des produits
- ▶ facilite le changement de familles d'objets
- ▶  nouveaux produits

- ▶ singleton

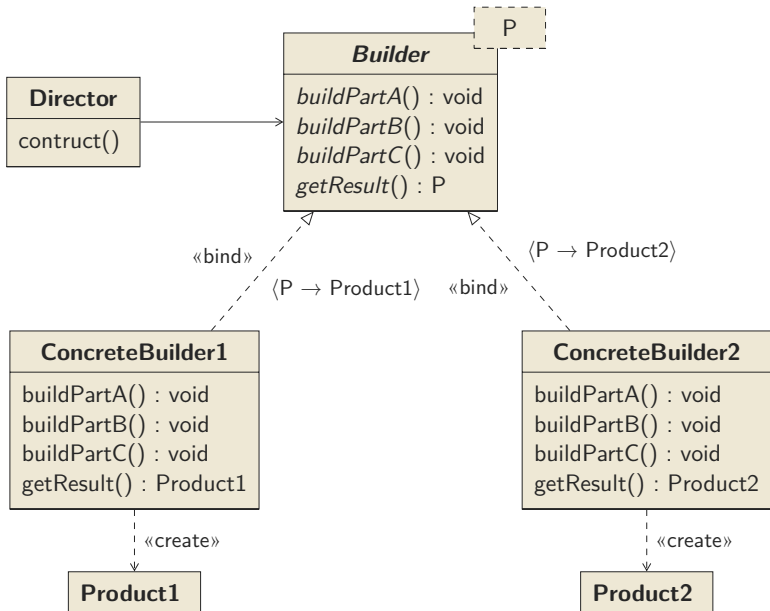
- ▶ classes de premier ordre + dictionnaire

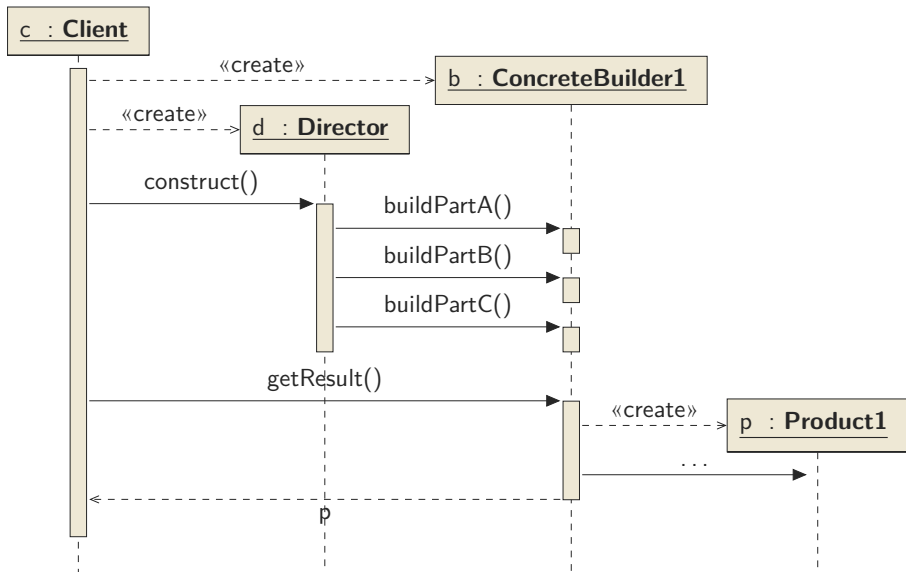
# Builder

- ▶ création similaire d'objets différents
- ▶ instanciation d'objets similaires
- ▶ objet/structure complexe
  - interface de création simple
- ▶ construction incrémentale
  - garantir l'état final
  - objet non modifiable

# Builder

- ▶ processus de création → Builder
- ▶ client indépendant des produits concrets
- ▶  $\neq$  builders  $\Rightarrow$   $\neq$  produits







- ▶ méthodes par défaut
- ▶ `builderFrom(product)` → prototype
- ▶ classe interne au produit
- ▶ fluent interface

## Définition (Fluent Interface)

- ▶ noms des méthodes **non** représentatifs
- ▶ retourne **this**
- ▶ enchaînement

# Builder

## Fluent Interface

```
FlightBuilder parisToSidney = FlightBuilder.defineFlight()  
    .of(myCompany)  
    .leavingFrom("Paris").the(2010,10,10).at(16, 0)  
    .to("Sidney").for(18, HOURS)  
    .passingBy("Singapore").for(2, HOURS).after(8, HOURS);
```

```
Flight f1 = parisToSidney.create();  
Flight f2 = parisToSidney.leavingAt(22, 0).create();
```

- ▶ *abstract factory*
- ▶ *structure composite*

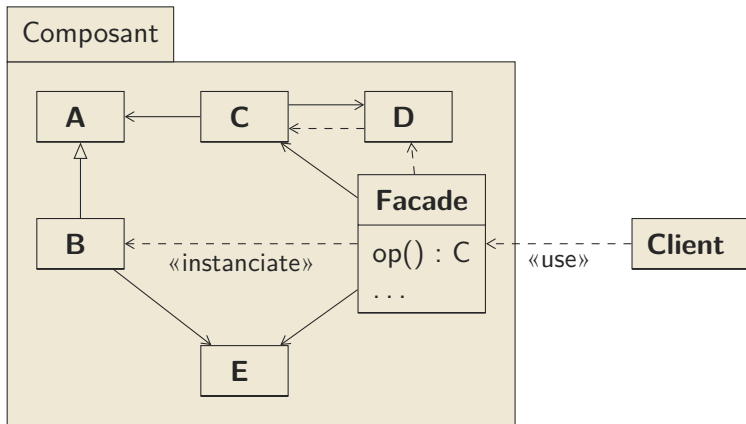
# Structure

# Façade

- ▶ composant
- ▶ plus haut niveau d'abstraction (métier)

interface :

- ▶ simplifiée
- ▶ unifiée
- ▶ unique



# Avantages

- ▶ facilité d'utilisation
- ▶ compréhensible
- ▶ test
- ▶ encapsulation du sous-système
- ▶ ↘ dépendances  $\Rightarrow$  ↘ couplage



# Extensions

- ▶ façade abstraite (interface)
- ▶ singleton

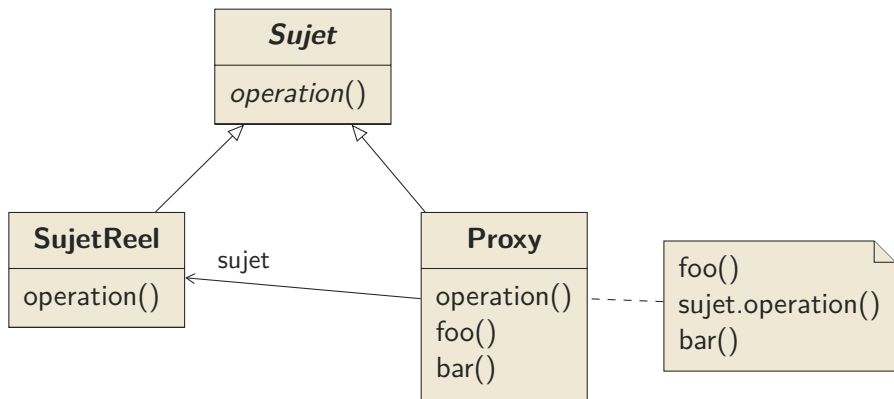
# Alternatives

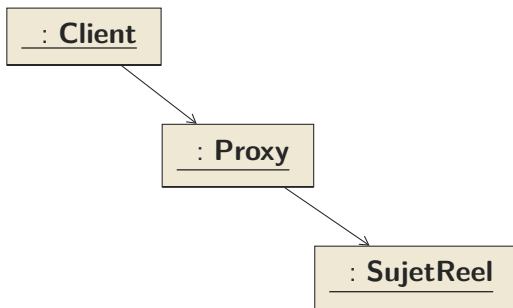
modules. . .

# Intermédiaire

## Proxy

- ▶ objet de substitution
- ▶ référence « intelligente »
- ▶ ajoute une indirection
- ▶ *proxy virtuel* : gestion paresseuse (création à la demande, cache)
- ▶ *proxy distant* : accès à service distant
- ▶ *proxy de protection* : contrôle d'accès, verrous





# Implémentation

- ▶ surcharge de l'accès aux attributs  
(`->`, `__getitem__`, `doesNotUnderstand`)
- ▶ `SujetReel` concret ou interface ?
- ▶ instantiation par le proxy

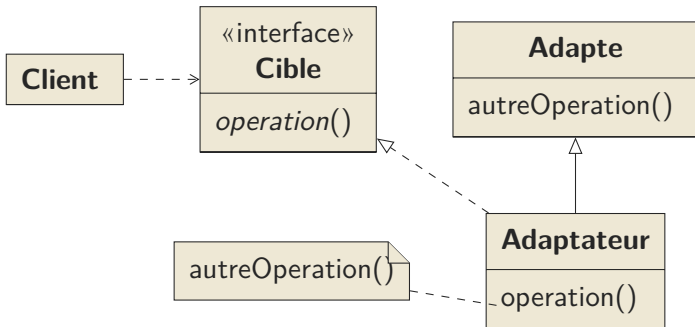
# Adaptateur

- ▶ conversion d'interface
- ▶ réutiliser les fonctionnalités
- ▶ interfaces non compatibles

## 2 versions

- ▶ adaptateur de classe : héritage multiple
- ▶ adaptateur d'objet : délégation

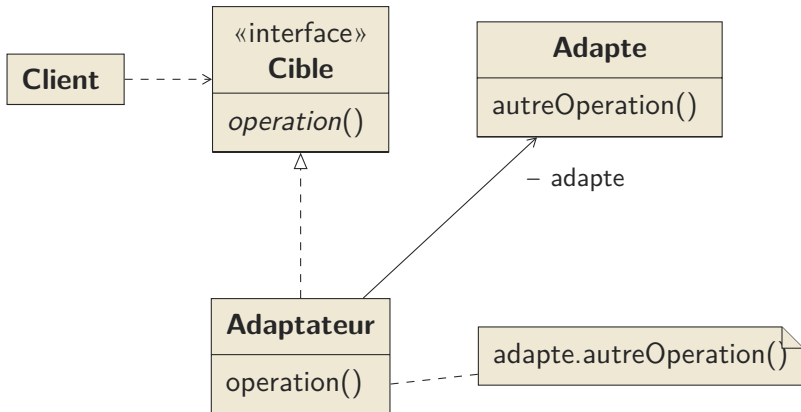




# Adaptateur

de classe

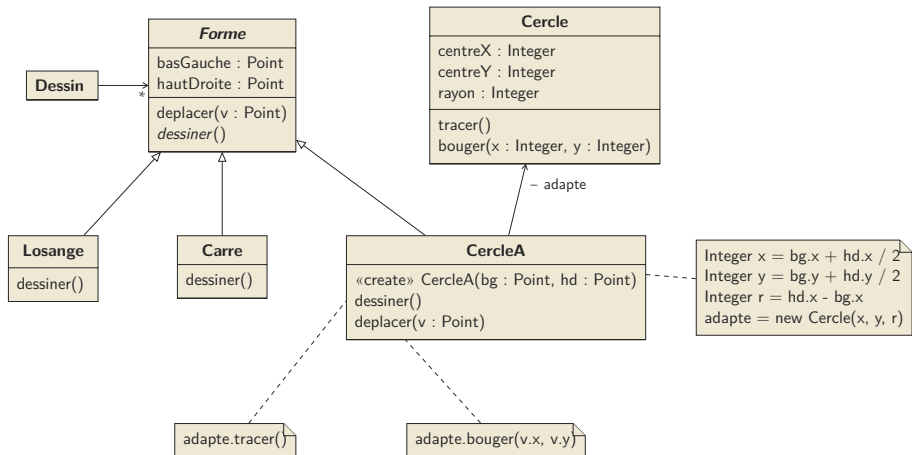
- ▶ uniquement une classe, pas ses sous-classes
- ▶ redéfinition de l'adapté
- ▶ exploite l'héritage



# Adaptateur

d'objet

- ▶ classe adaptée dynamique
- ▶ délégation manuelle
- ▶ redéfinition difficile : pas de polymorphisme



- ▶ renommage → opérations différentes
- ▶ adaptateur à double sens

# Implémentation

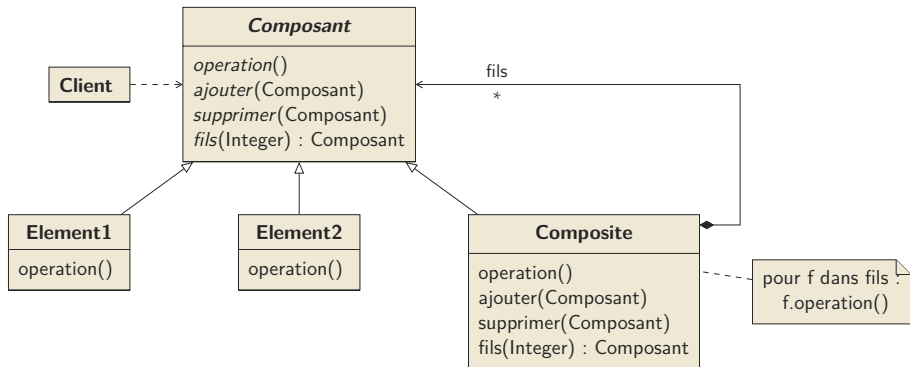
- ▶ adaptateur de classe : héritage privé de l'adapté
- ▶ interface restreinte

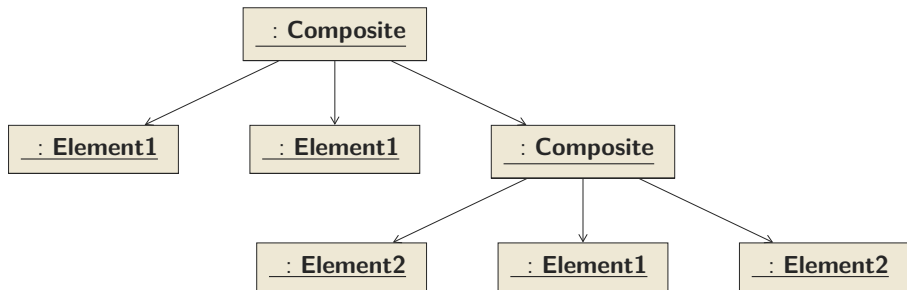
# Composite

- ▶ structures hiérarchiques
- ▶ composition indéfinie
- ▶ traitement uniforme objet et composition



- ▶ éléments primitifs
- ▶ conteneurs
- ▶  $\Rightarrow$  composition récursive





- ▶ composition récursive
- ▶ client : traitement uniforme
- ▶ ajout de types de composants
- ▶ pas de contraintes sur le type de contenu

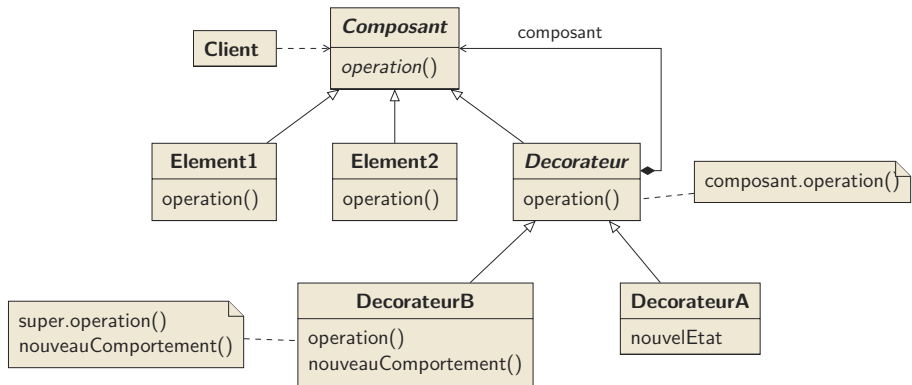
# Implémentation

- ▶ double ref. composant  $\leftrightarrow$  composite
- ▶ partage des fils : cf. Flyweight
- ▶ interface du composant maximale (impl. par défaut)
- ▶ gestion de la hiérarchie : composant ou composite ?
- ▶ transparence vs. sécurité
- ▶ ordre des fils : cf. Iterator

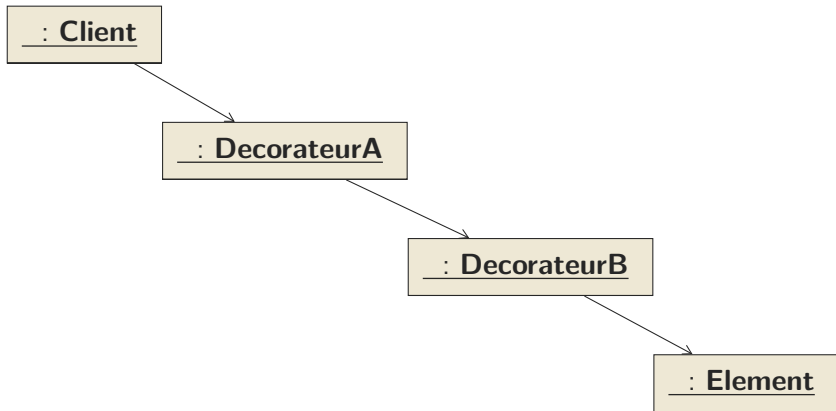
# Décorateur

- ▶ ajout **dynamique** de responsabilités
- ▶ sur une **instance**
- ▶ ex : ajout de défilement (IHM)

- ▶ héritage pas flexible et traite la classe
- ▶ encapsulation et délégation
- ▶ transparent pour le client
- ▶ composition récursive







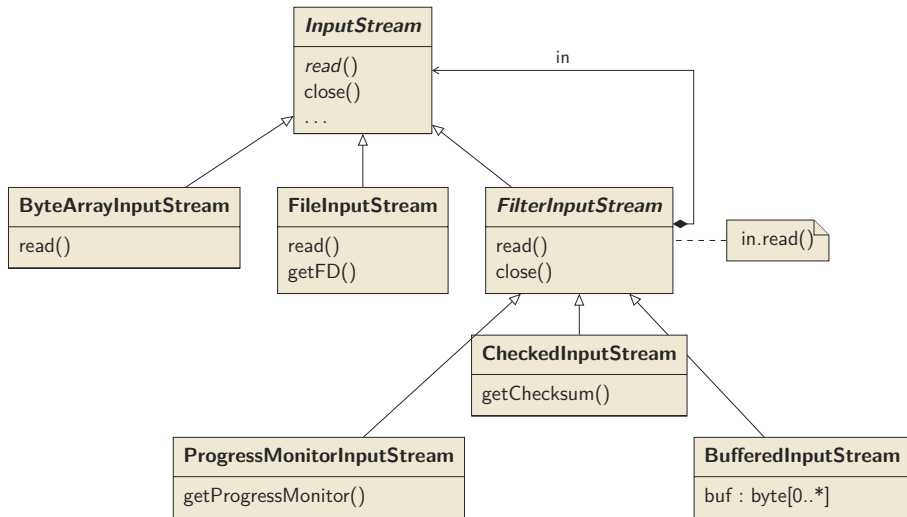
- ▶ décorateur substituable
- ▶ transparence totale (client et élément)
- ▶ dépend de l'ordre de décoration

- ▶ plus flexible que héritage
- ▶ niveau instance
- ▶ dynamique
- ▶ séparation des responsabilités
- ▶ composition libre
- ▶ décoration multiple avec le même décorateur

- ▶ identités différentes
- ▶ types homogènes
- ▶ difficile à comprendre et corriger

# Exemple

Java : [InputStream](#)



```
InputStream in = new BufferedInputStream(  
    new ProgressMonitorInputStream(parentComponent, "Reading",  
        new FileInputStream(fileName)));
```

# Alternatives

- ▶ stratégie : interface trop lourde
- ▶ orienté aspect
- ▶ composition de fonction
- ▶ fonctions/classes d'ordre supérieur

# Flyweight

poid-mouche

partage d'instances

- ▶ grand nombre d'objets
- ▶ fine granularité
- ▶ efficacité mémoire



- ▶ instances partagées
- ▶ contextes multiples
- ▶ semblent des objets indépendants

# États des instances

séparé en 2 sous états

- ▶ *intrinsèque*
- ▶ *extrinsèque*

## Définition (État intrinsèque)

- ▶ indépendant du contexte
- ▶  $\Rightarrow$  partageable
- ▶  $\rightarrow$  stocké dans le *flyweight*

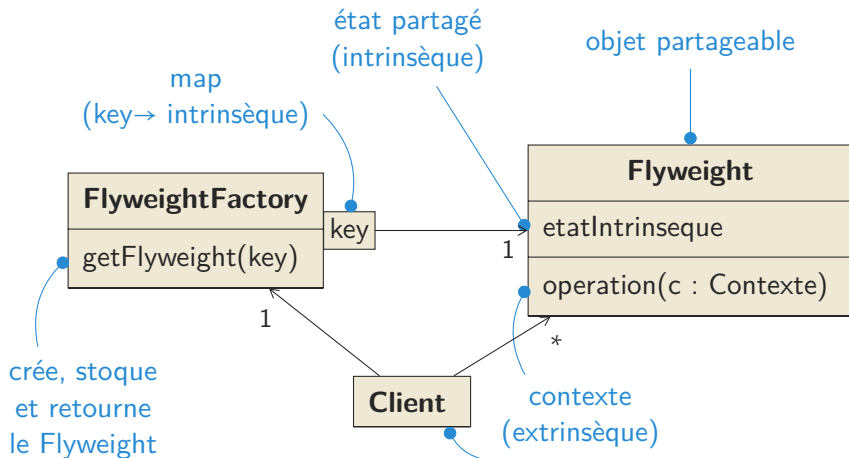
## Définition (État extrinsèque)

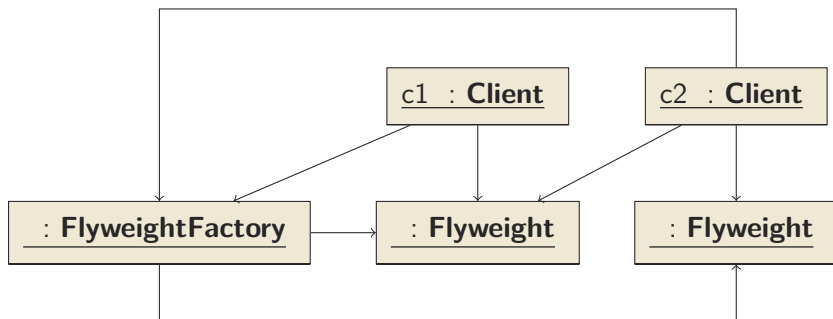
- ▶ dépendant du contexte
- ▶  $\Rightarrow$  non partageable
- ▶  $\rightarrow$  passé au *flyweight* par le client

pas d'instanciation directe

# Utilisation

- ▶ grand nombre d'objets
- ▶ grand coût mémoire
- ▶ état surtout extrinsèque
- ▶ groupe remplacé par 1 instance
- ▶ indépendant de l'identité







# Implémentation

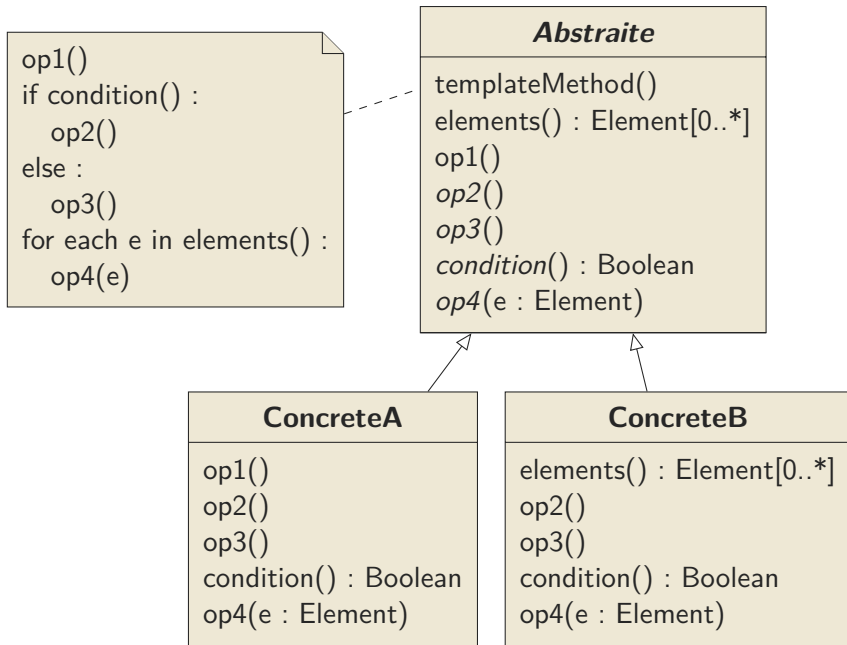
- ▶ état extrinsèque : calculé ou peu de valeurs
- ▶ G.C.
- ▶ spécialisation

# Comportement

# Template method

- définition générique d'un algo
- détails des étapes différés

- ▶ factorise les invariants d'un algo
- ▶ évite la duplication de code
- ▶ facilite la réutilisation
- ▶ contrôle l'extension (*hooks*)
- ▶ inversion de contrôle



# Alternatives

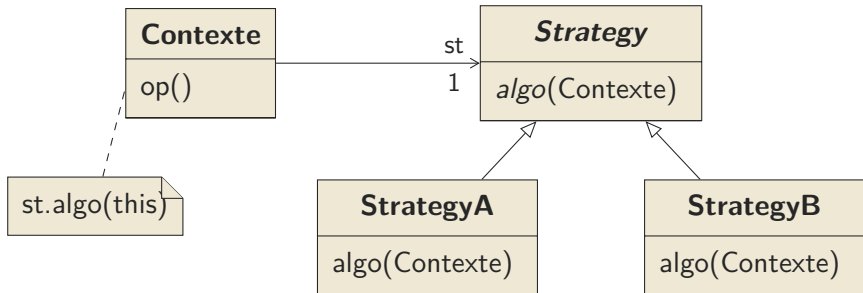
- stratégie
- fonction ordre supérieur + composition

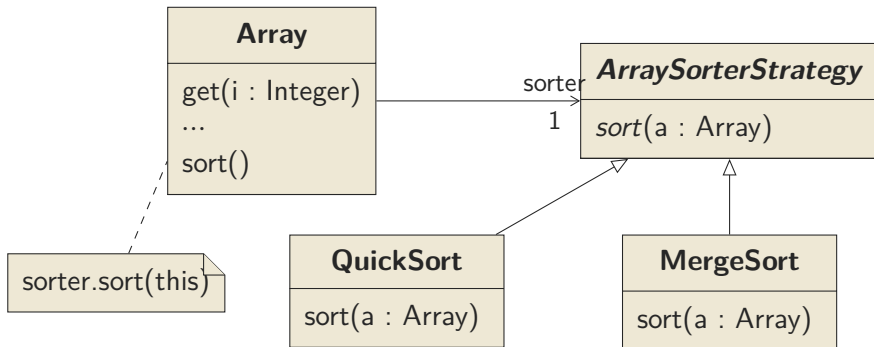
# Stratégie

- ▶ famille d'algo
- ▶ interchangeable
- ▶ indépendant du client

- ▶ classes similaires :  $\neq$  implémentations
- ▶ variantes d'algo (compromis taille – temps)
- ▶ encapsulation de données propres à l'algo
- ▶ comportement alternatifs  $\Rightarrow$  polymorphisme > tests







- ▶ définie une famille d'algo réutilisables
- ▶ alternative à l'héritage
- ▶  $\Rightarrow$  changement dynamique
- ▶ polymorphisme plutôt que tests
- ▶ choix d'implémentation

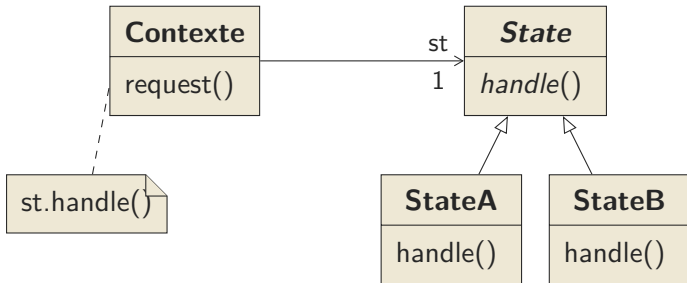
- ▶ connaissance des implémentations
- ▶ communication
- ▶ nombre d'objets

- ▶ template method
- ▶ method object (*callable*)
- ▶ singleton ou flyweight
- ▶ fonction de premier ordre

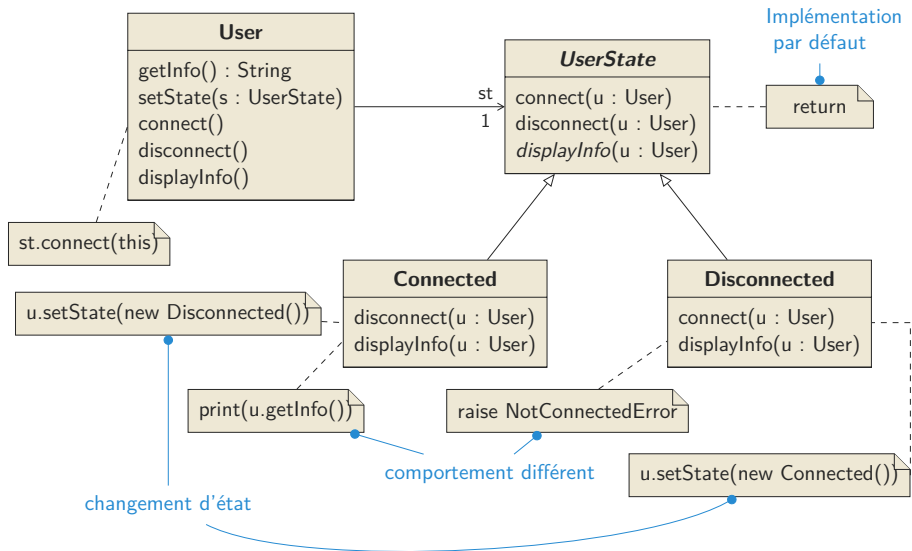
# État

- ▶ objet  $\rightarrow$  états
- ▶ état  $\Rightarrow \neq$  comportement

- ▶ automate (parser, ...)
- ▶ protocole (connexion, ordre d'appel, ...)



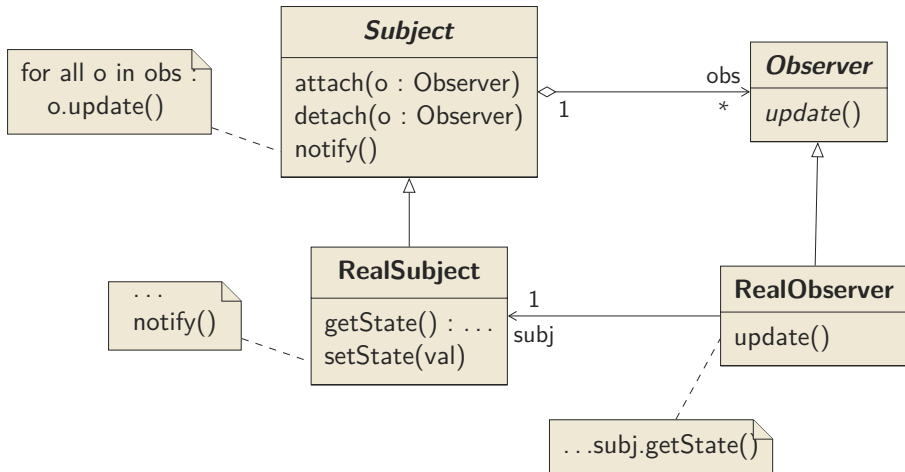




# Observateur

- ▶ notification
- ▶ changement d'état
- ▶ événements

- ▶ synchronisation
- ▶ couplage faible
- ▶ p. ex. : IHM (Vue ↔ Modèle)



- ▶ couplage faible et abstrait
- ▶  $\Rightarrow$  couches différentes
- ▶ diffusion de messages
- ▶ ajout / suppression dynamique

# Variantes

- ▶ sujet en paramètre du `update`
- ▶  $\Rightarrow$  observation de différents sujets
- ▶ le client appelle `notify()`
- ▶ infos dans `update` : *push* vs. *pull*
- ▶ type de notification  $\leftrightarrow$  *handler*
- ▶  $\Rightarrow$  événementiel
- ▶ *manager* intermédiaire pour sujet  $\leftrightarrow$  observateur

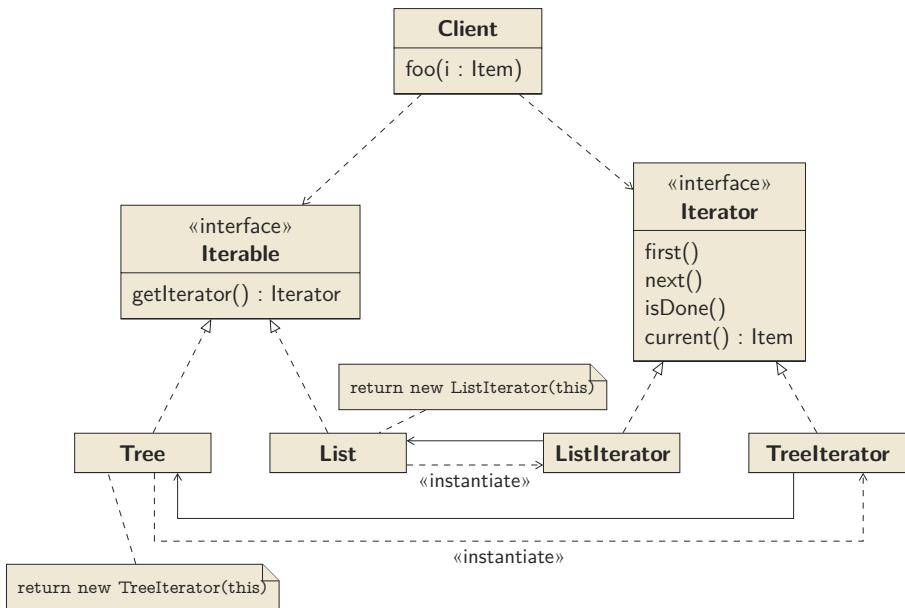
# Itérateur

- ▶ abstraction des collections
- ▶ (ou structure quelconque)
- ▶ parcours séquentiel générique : *itération polymorphe*
- ▶ parcours alternatif facile

- ▶ extraction du parcours
- ▶ interface prédéfinie
- ▶ maintenance état courant



- ▶ itérateur par structure (interface d'accès)
- ▶ structure → itérateur (factory)



```
Iterator it = myStructure.getIterator();
```

```
it.first();
```

```
while (!it.isDone()) {
```

```
    foo(it.current());
```

```
    it.next();
```

```
}
```

```
// ou
```

```
for (it.first(); !it.isDone(); it.next()) {
```

```
    foo(it.current());
```

```
}
```

- ▶  $\neq$  itérateurs  $\Rightarrow \neq$  parcours
- ▶ (profondeur, largeur, ...)
- ▶ interface + simple de la structure

- ▶ type générique (template)
- ▶ robuste pour la modif.

# Variantes et extensions

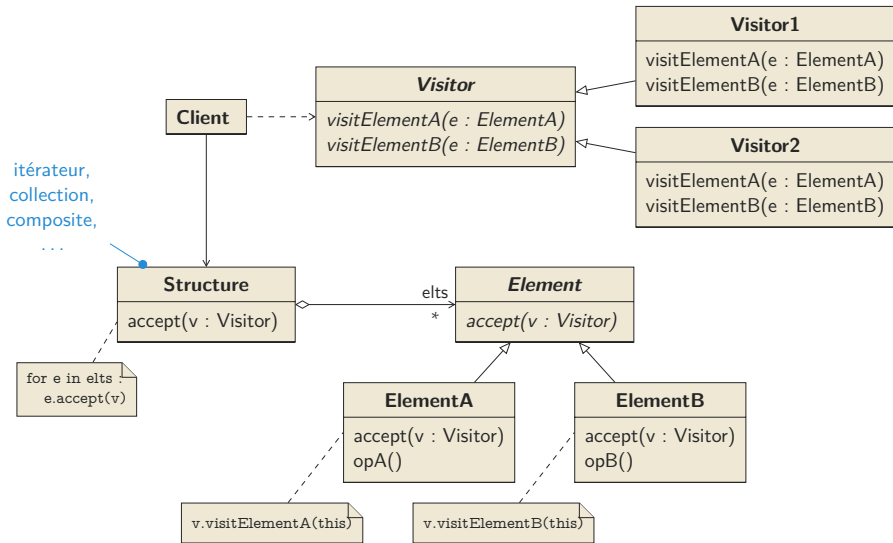
- ▶ intégration dans le langage (for each)
- ▶ `next` avec effet de bord
- ▶ accès privilégié
- ▶ externe vs. interne (map/visiteur, lambda)
- ▶ générateurs
- ▶ ajout de *décorateur*  $\Rightarrow$  chaînage d'itérateur (*template meth.* ou *stratégie*)

# Visiteur

- ▶ extension de l'itérateur interne
- ▶ types hétérogènes
- ▶ structure complexe
- ▶ nombreux traitements
- ▶ *double dispatch*

- ▶ traitement extérieur
  - ▶ séparation des responsabilités
  - ▶ ajout facile de traitements
  - ▶ maintenance facile (shotgun surgery)
- ▶ indépendant de la structure
  - ▶ évolution facile
  - ▶ semi-structuré / dynamique
- ▶ agrégateur
- ▶ ajout d'éléments difficile





# Variantes


- ▶ parcours :
  - ▶ **structure**
  - ▶ visiteur (template meth.)
    - parcours dépendant du traitement
  - ▶ itérateur (pas de double dispatch)

# Alternatives

- ▶ dispatch multiple (résolution tardive sur les paramètres)
- ▶ fonction d'ordre supérieur (map, reduce)

Autres

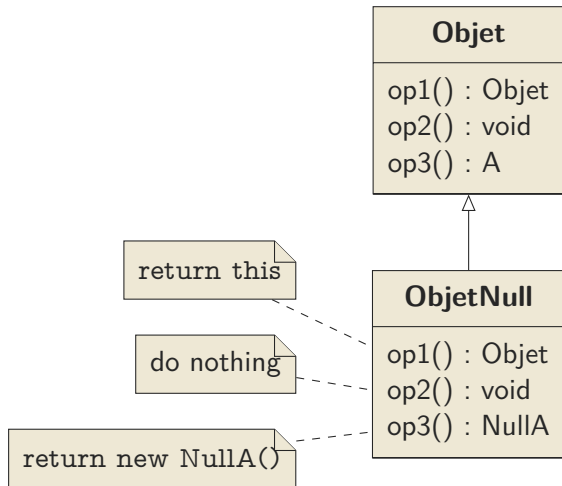
# Null Object

- ▶ référence nulle
- ▶ vérification (`NullPointerException`)
- ▶  polymorphisme
- ▶  $\Rightarrow$  duplication de code

## Objet nul spécifique

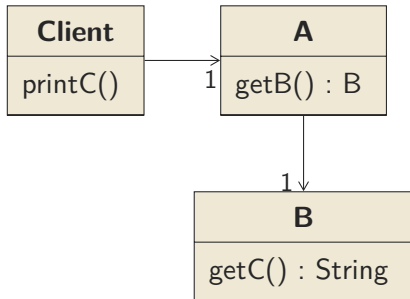
- ▶ même interface
- ▶ méthodes vides ou retourne un autre objet nul

- ▶ pas d'effet de bord
- ▶ prévisible
- ▶ chaînage





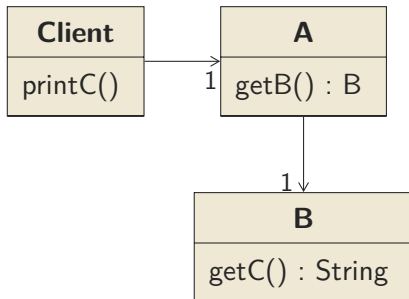
# Mise en œuvre



```

class Client {
    void printC() {
        String r = "?";
        if (a != null) {
            b = a.getB();
            if (b != null) {
                r = b.getB();
            }
        }
        System.out.println(r);
    }
}
    
```

# Mise en œuvre



```

class NullA extends A {
    B getB() { return new NullB(); }
}

class NullB extends B {
    String getC() { return "?"; }
}

class Client {
    void printC() {
        System.out.println(
            a.getB().getC());
    }
}
    
```

# Variantes et extensions

- ▶ singleton, flyweight, inner class
- ▶ test de nullité :
  - ▶ méthode `isNull()` : `Boolean`
  - ▶ transtypage booléen ou opérateur de test
  - ▶ interface `Null` et test `isinstance Null`
- ▶ `Null` générique (*duck typing*, Objective-C)
- ▶ *special case* (`NaN`, `Infinity`, valeur inconnue)

# Variantes et extensions

```
class MyObject {  
    private String name;  
  
    String name() { return name;}  
    void name(String n) { name = n;}  
  
    void foo() {  
        System.out.println("Foo");  
    }  
}
```

# Variantes et extensions

```
class MyObject {  
    private String name;  
  
    String name() { return name;}  
    void name(String n) { name = n;}  
  
    void foo() {  
        System.out.println("Foo");  
    }  
  
    static final MyObject NULL = new MyObject() {  
        @Override  
        String name() { return "Unknown";}  
  
        @Override  
        void name(String n) {}  
  
        @Override  
        void foo() {}  
    };  
}
```

# Variantes et extensions

- ▶ singleton, flyweight, inner class
- ▶ test de nullité :
  - ▶ méthode `isNull()` : `Boolean`
  - ▶ transtypage booléen ou opérateur de test
  - ▶ interface `Null` et test `isinstance Null`
- ▶ `Null` générique (*duck typing*, Objective-C)
- ▶ *special case* (`NaN`, `Infinity`, valeur inconnue)

# Type Option

- ▶ langages fonctionnels
  - ▶ OCaml : `type 'a option = None | Some of 'a`
  - ▶ Haskell : `data Maybe a = Just a | Nothing`
  - ▶ Scala, Rust, ...
- ▶ théorie des monades
- ▶ *pattern matching*