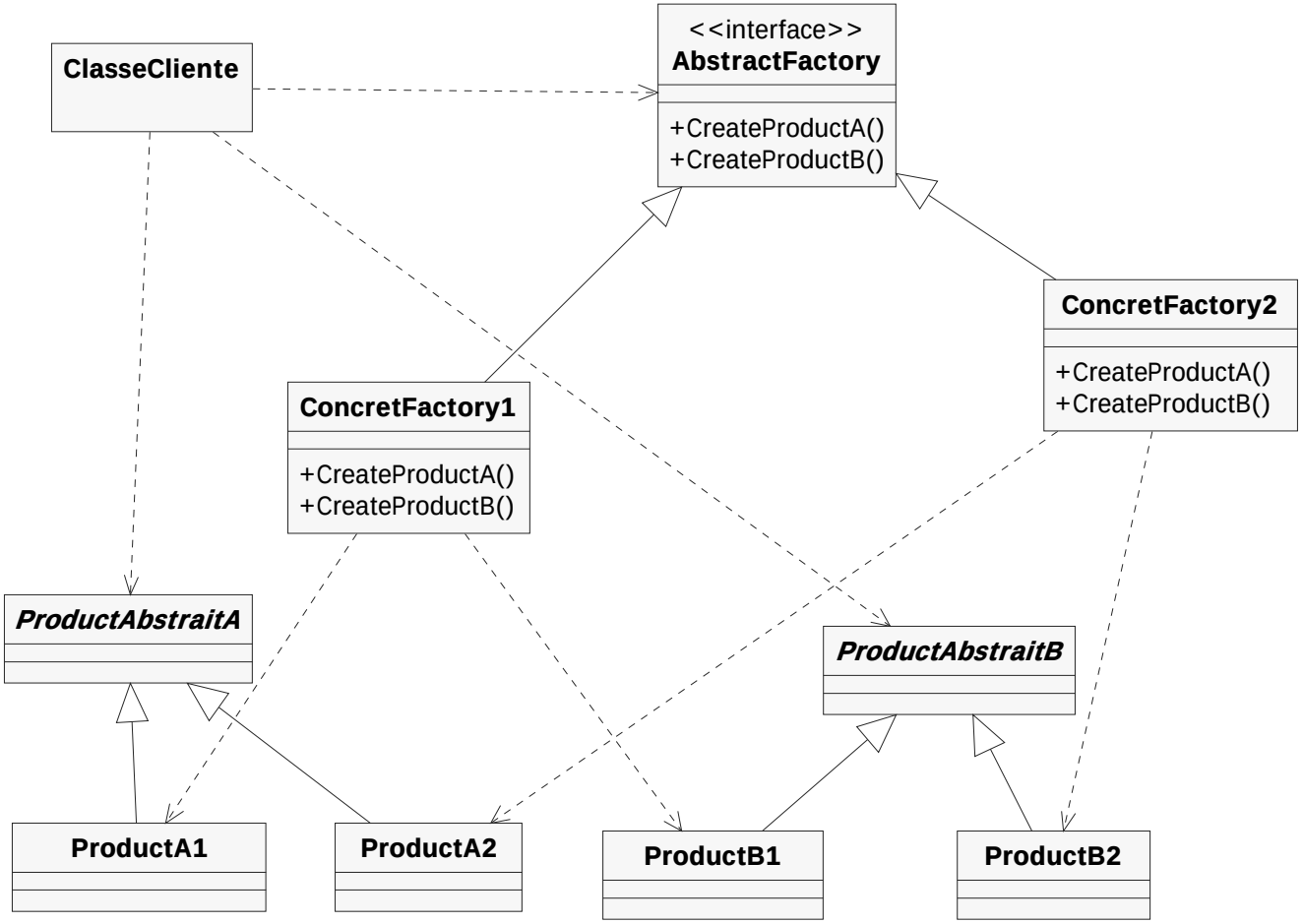
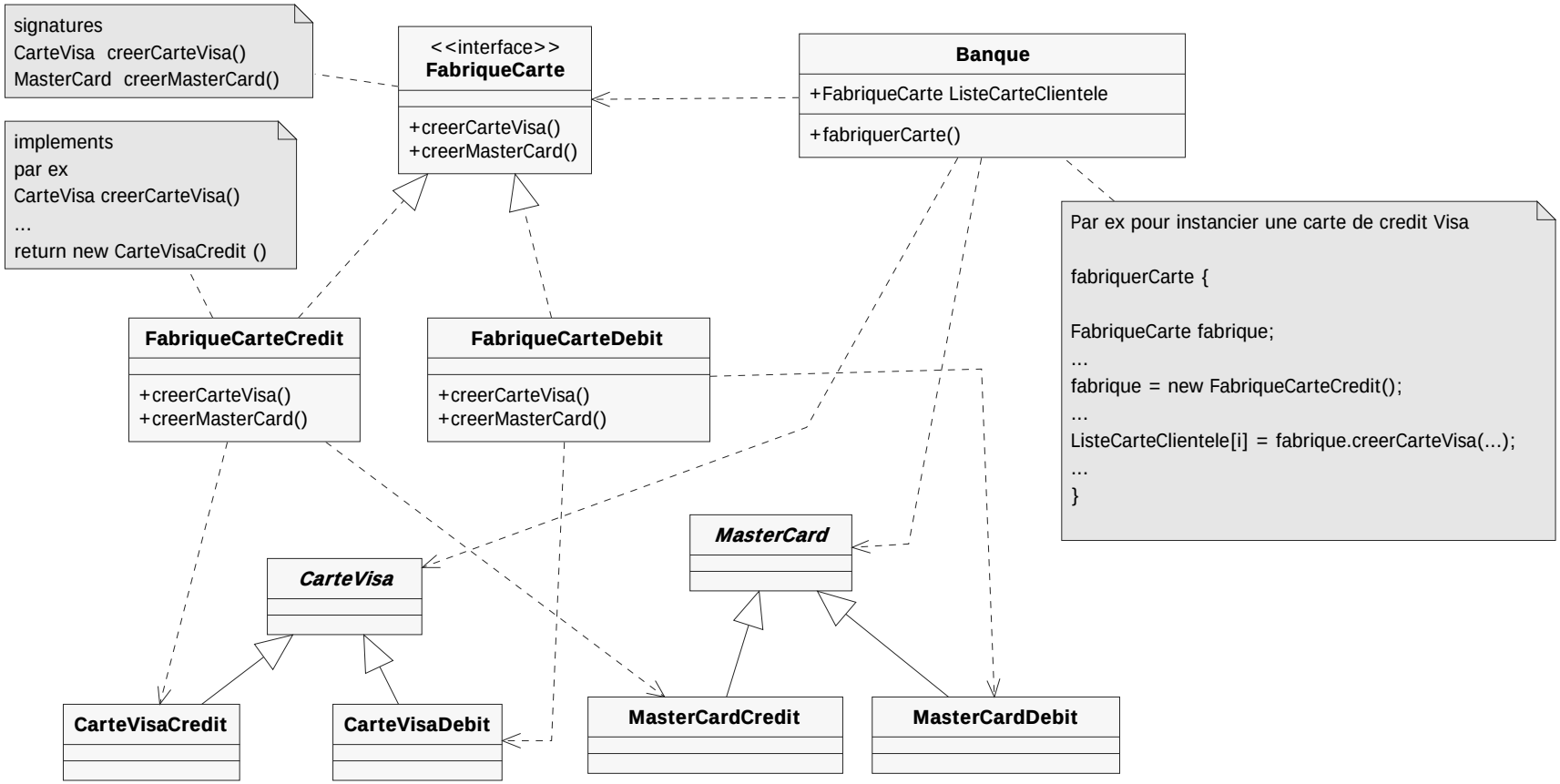
Les DP sont classés selon le GOF en 3 catégories :

1) Patterns de construction
2) Patterns de structuration
3) Patterns decomportement

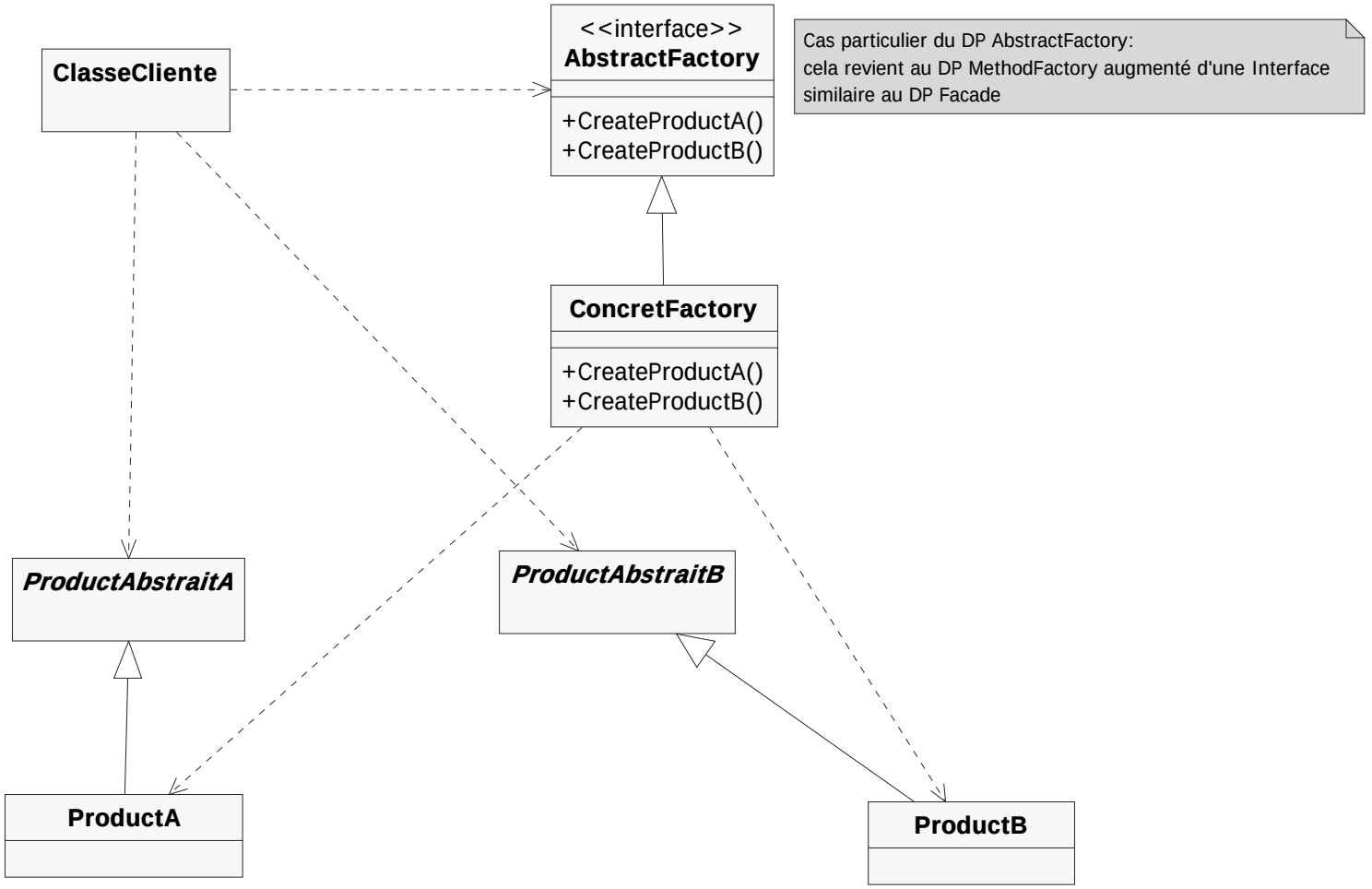
Untitled ::ModelesDP::DP_Construction::AbstractFactoryDP::AbstractFactoryDP::AbstractFactoryCD

But : créer des objets de 2 hiérarchies sans connaitre l'implémentation des classes



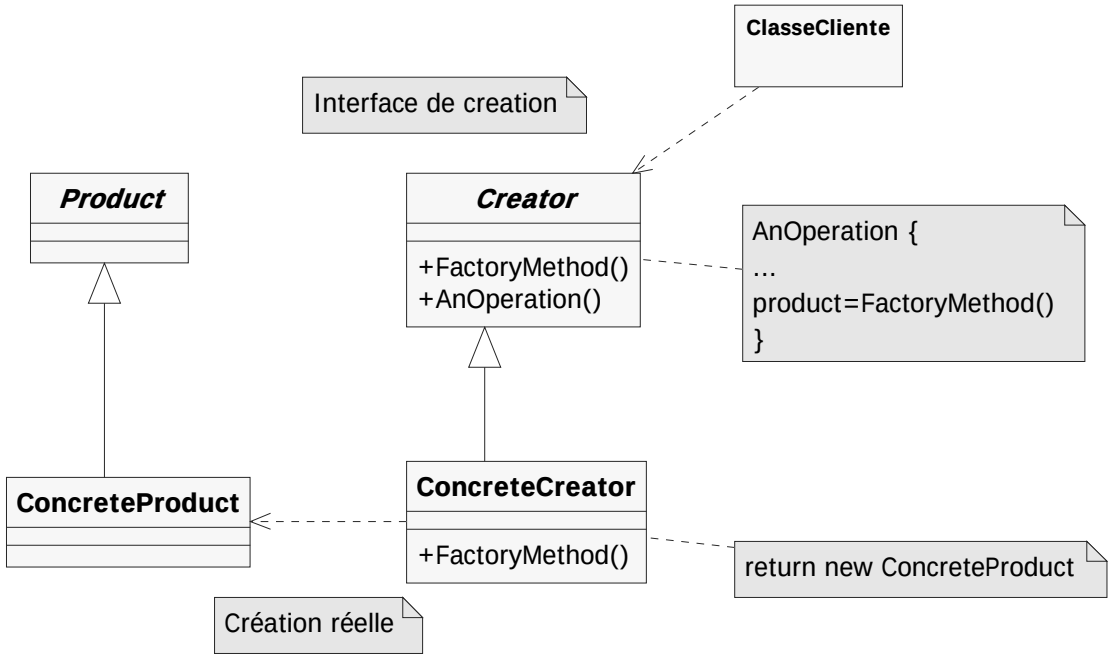

signatures
CarteVisa creerCarteVisa()
MasterCard creerMasterCard()
<<interface>>
FabriqueCarte
+creerCarteVisa()
+creerMasterCard()
Banque
+FabriqueCarte ListeCarteClientele
+fabriquerCarte()
implements
par ex
CarteVisa creerCarteVisa()
...
return new CarteVisaCredit ()
Par ex pour instancier une carte de credit Visa
fabriquerCarte {
FabriqueCarte fabrique;
...
fabrique = new FabriqueCarteCredit();
...
ListeCarteClientele[i] = fabrique.creerCarteVisa(...);
...
}
FabriqueCarteCredit
+creerCarteVisa()
+creerMasterCard()
FabriqueCarteDebit
+creerCarteVisa()
+creerMasterCard()
MasterCard
CarteVisa
CarteVisaCredit
CarteVisaDebit
MasterCardCredit
MasterCardDebit

Untitled ::ModelesDP::DP_Construction::AbstractFactoryVarianteDP::AbstractFactoryVarianteCD

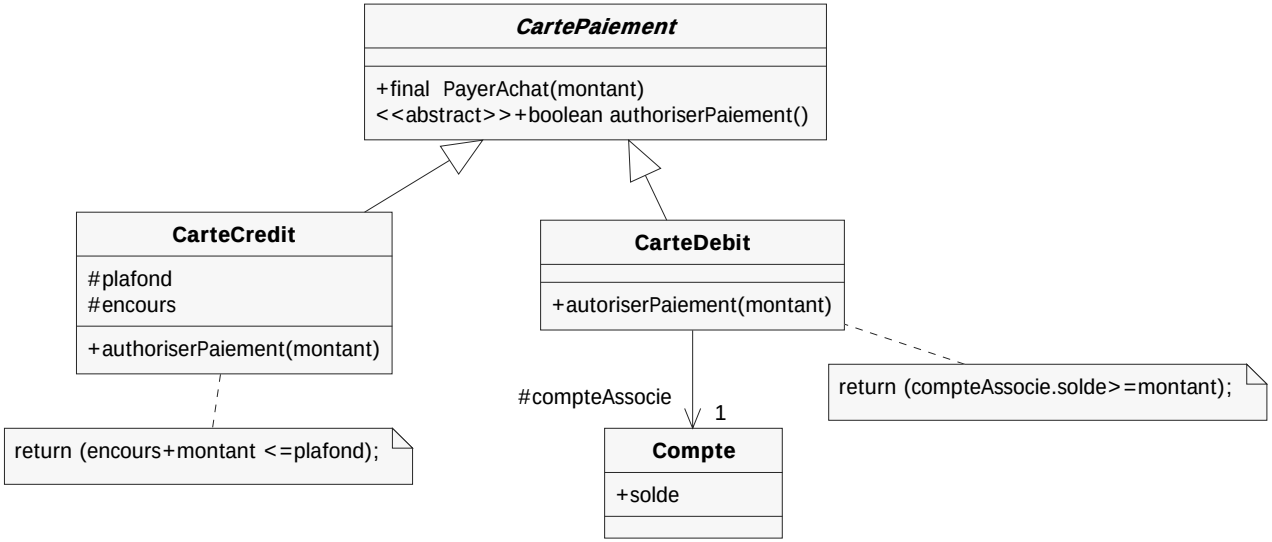


Untitled ::ModelesDP::DP_Construction::FactoryMethodDP::FactoryMethodDP::MethodFactoryCD

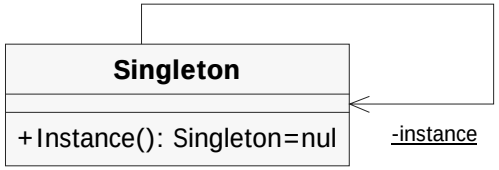
But : introduire une méthode abstraite de création d'un objet en reportant aux sous-classes concrètes la création effective

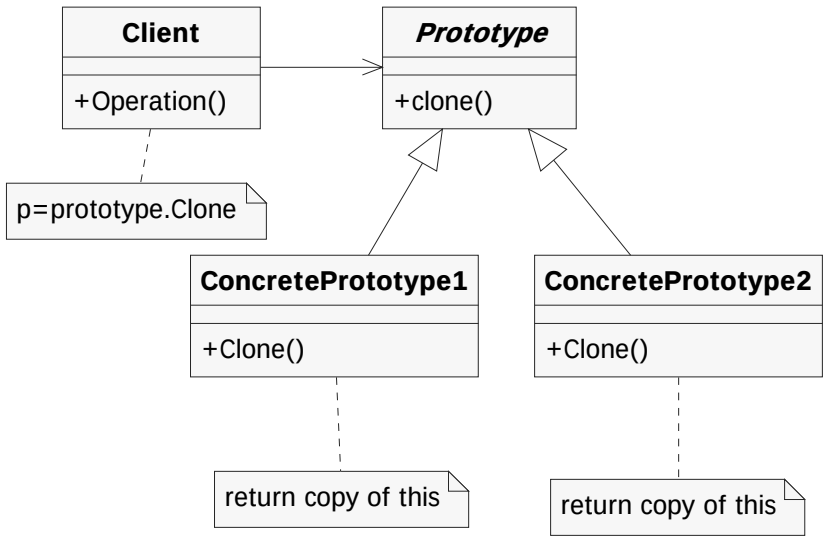


Untitled ::ModelesDP::DP_Construction::FactoryMethodDP::TemplateMethodEx::TM_cartesBancairesCD



Untitled ::ModelesDP::DP_Construction::Singleton::SingletonCD



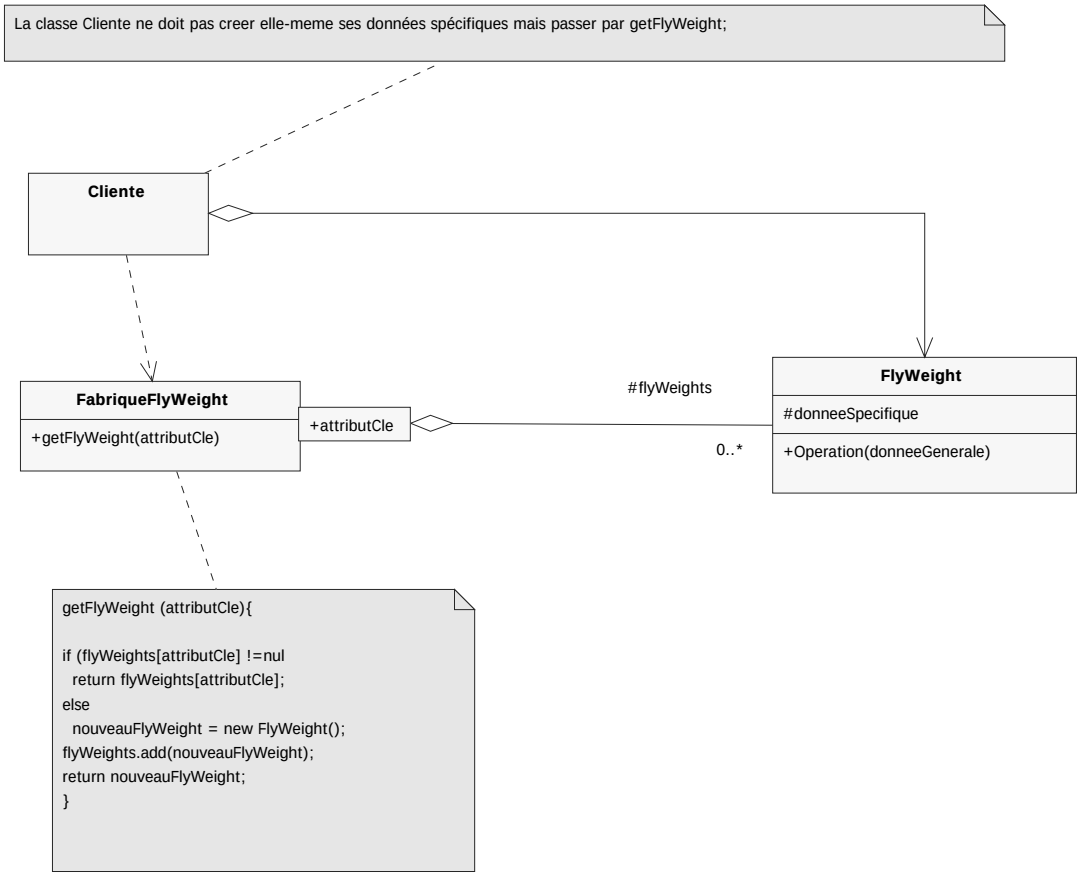

Client
+Operation()
Prototype
+clone()
p=prototype.Clone
ConcretePrototype1
+Clone()
ConcretePrototype2
+Clone()
return copy of this
return copy of this

Untitled ::ModelesDP::DP_Construction::FlyWeight::FlyWeihtCD

But: partager de façon efficace de nombreux objets de petite taille

Ce DP propose de partager les données communes entre objets et donne un mécanisme pour créer les données spécifiques

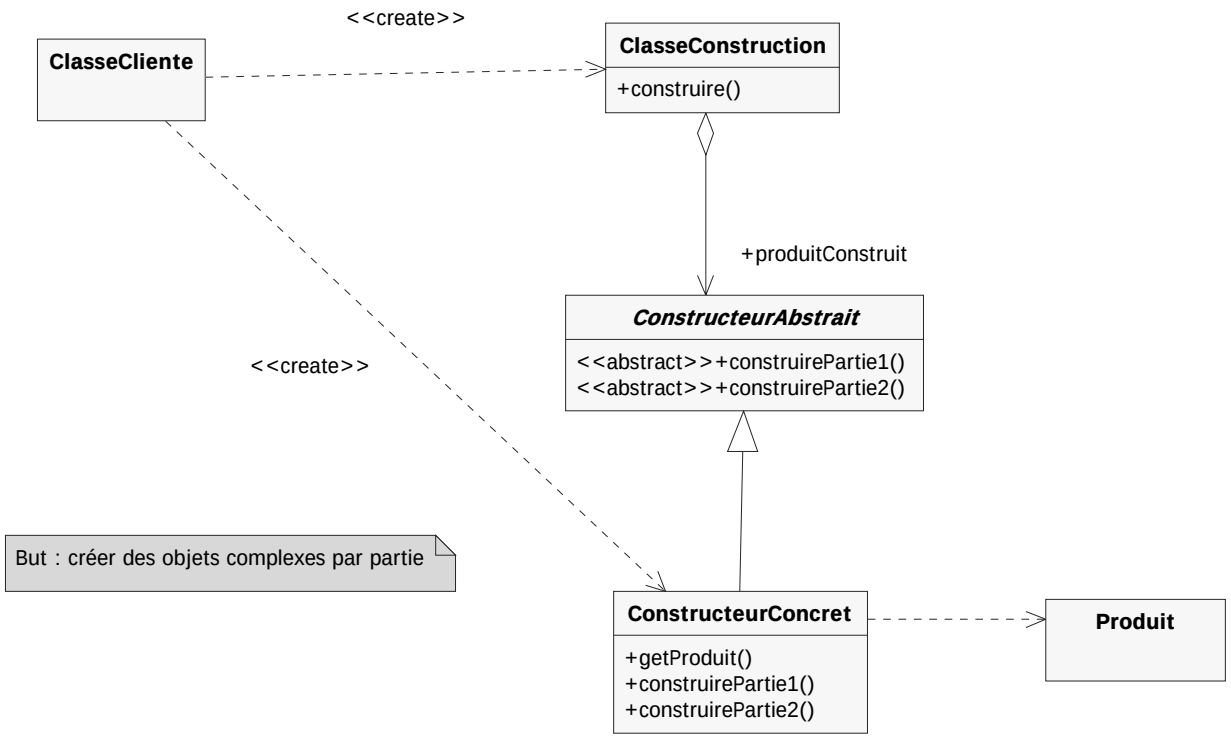
La classe Cliente ne doit pas creer elle-meme ses données spécifiques mais passer par getFlyWeight;

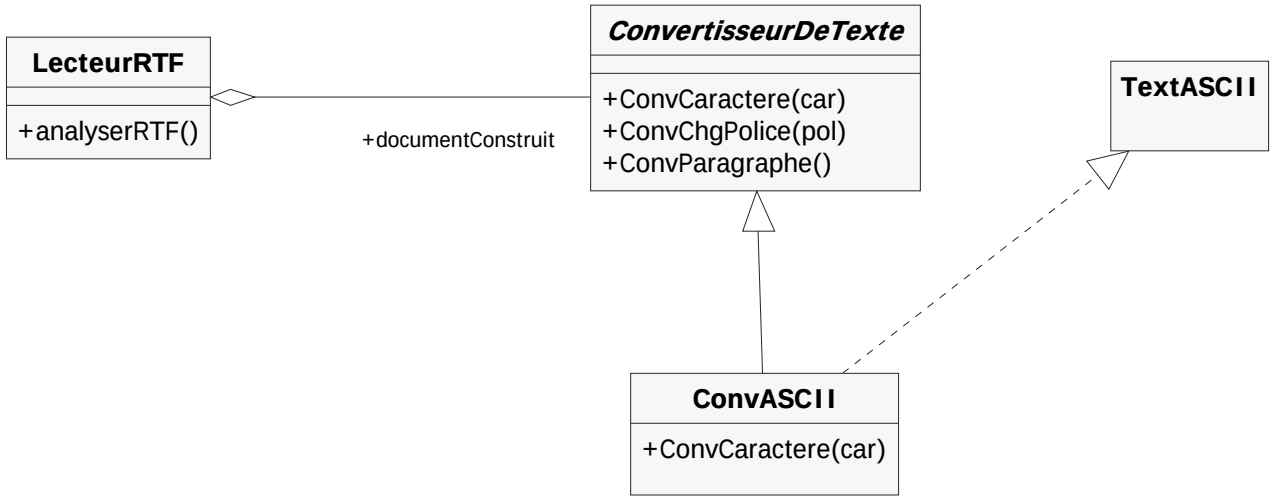


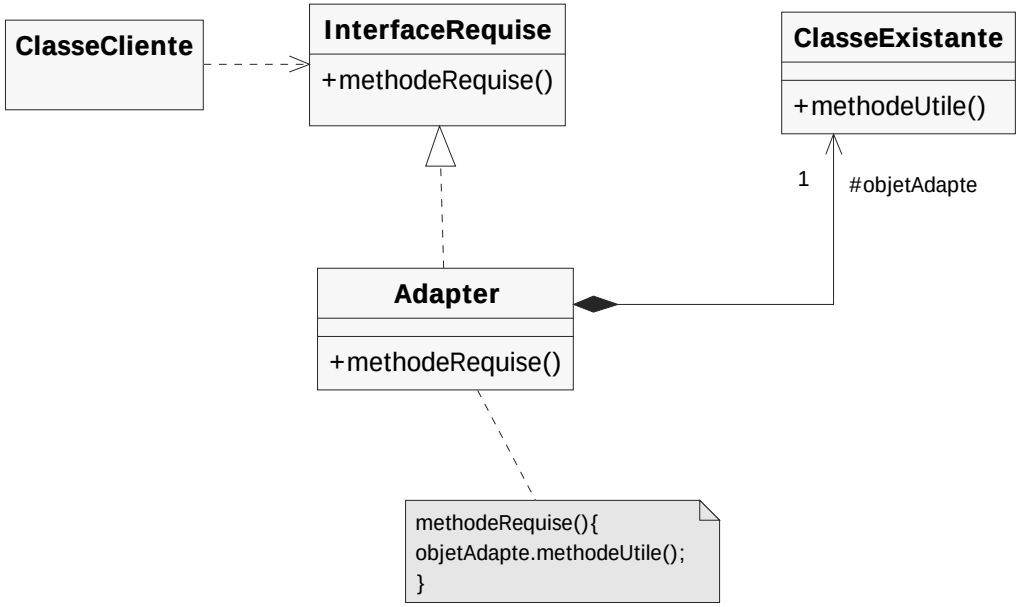

```
getFlyWeight (attributCle){

if (flyWeights[attributCle] !=nul
  return flyWeights[attributCle];
else
  nouveauFlyWeight = new FlyWeight();
flyWeights.add(nouveauFlyWeight);
return nouveauFlyWeight;
}
```

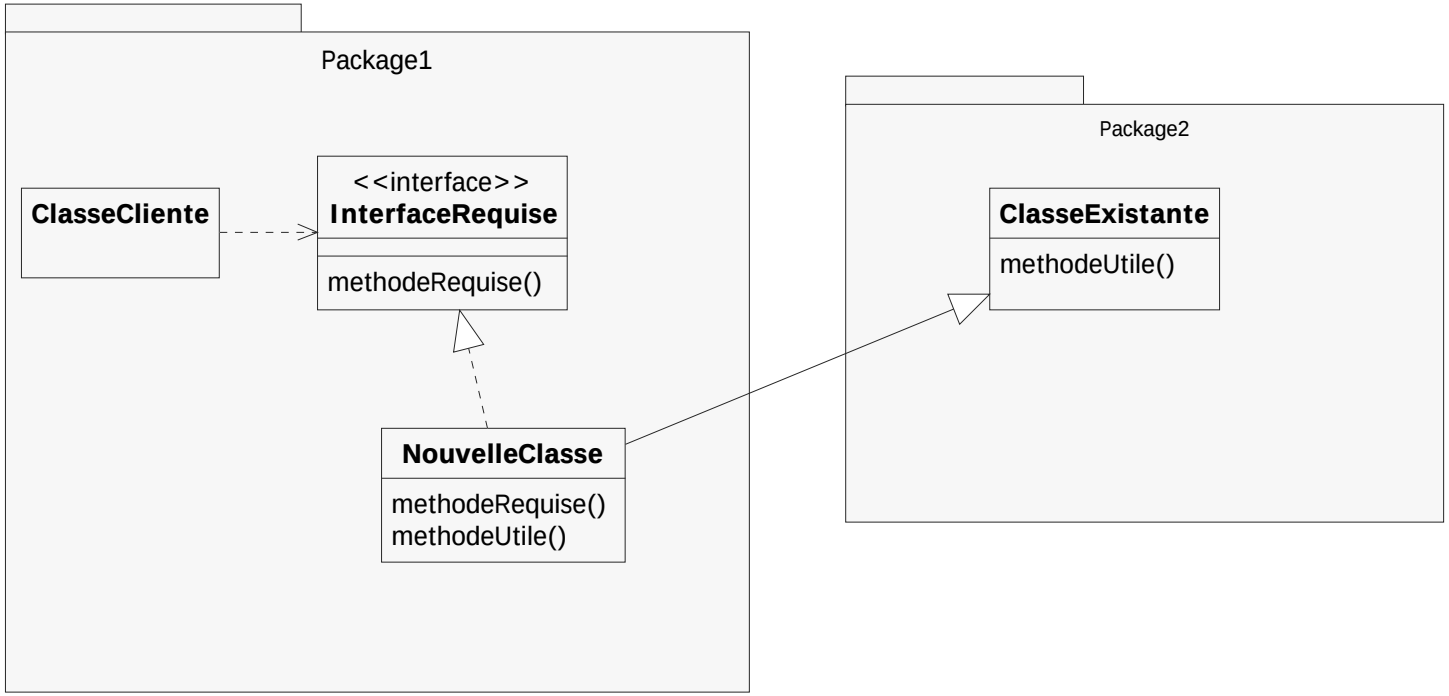
Untitled ::ModelesDP::DP_Construction::Builder::BuilderDP::BuilderCD

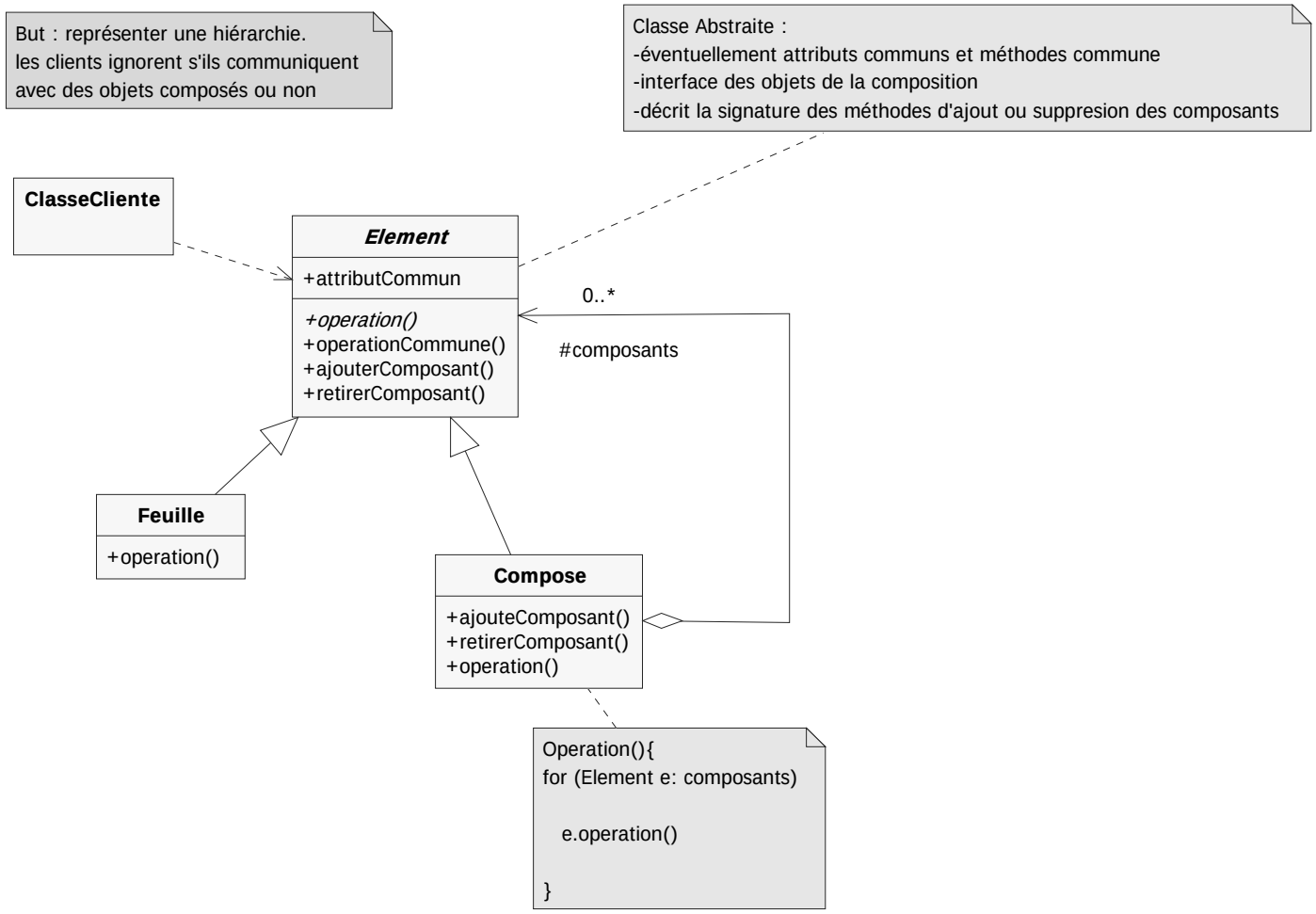



LecteurRTF
+analyserRTF()
+documentConstruit
ConvertisseurDeTexte
+ConvCaractere(car)
+ConvChgPolice(pol)
+ConvParagraphe()
TextASCII
ConvASCII
+ConvCaractere(car)

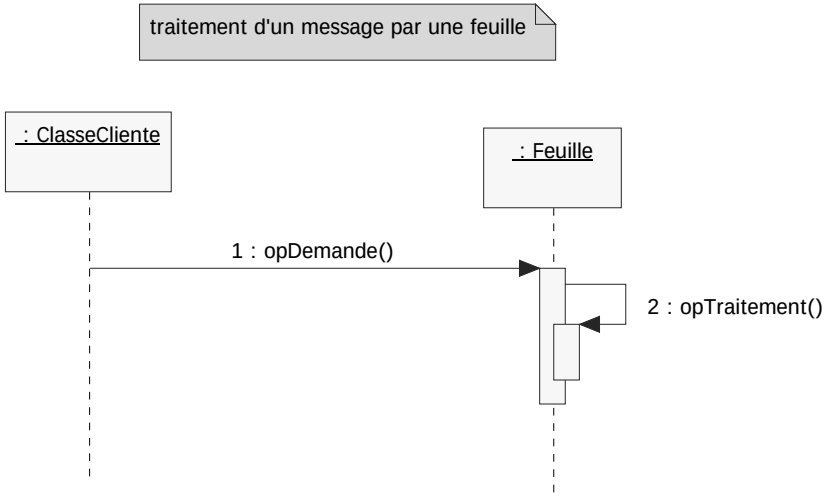

ClasseCliente
InterfaceRequise
+methodeRequise()
ClasseExistante
+methodeUtile()
1
#objetAdapte
Adapter
+methodeRequise()
methodeRequise(){
objetAdapte.methodeUtile();
}

Untitled ::ModelesDP::DP_Structuration::AdapterClasse::CD_ADAPTER_Classe

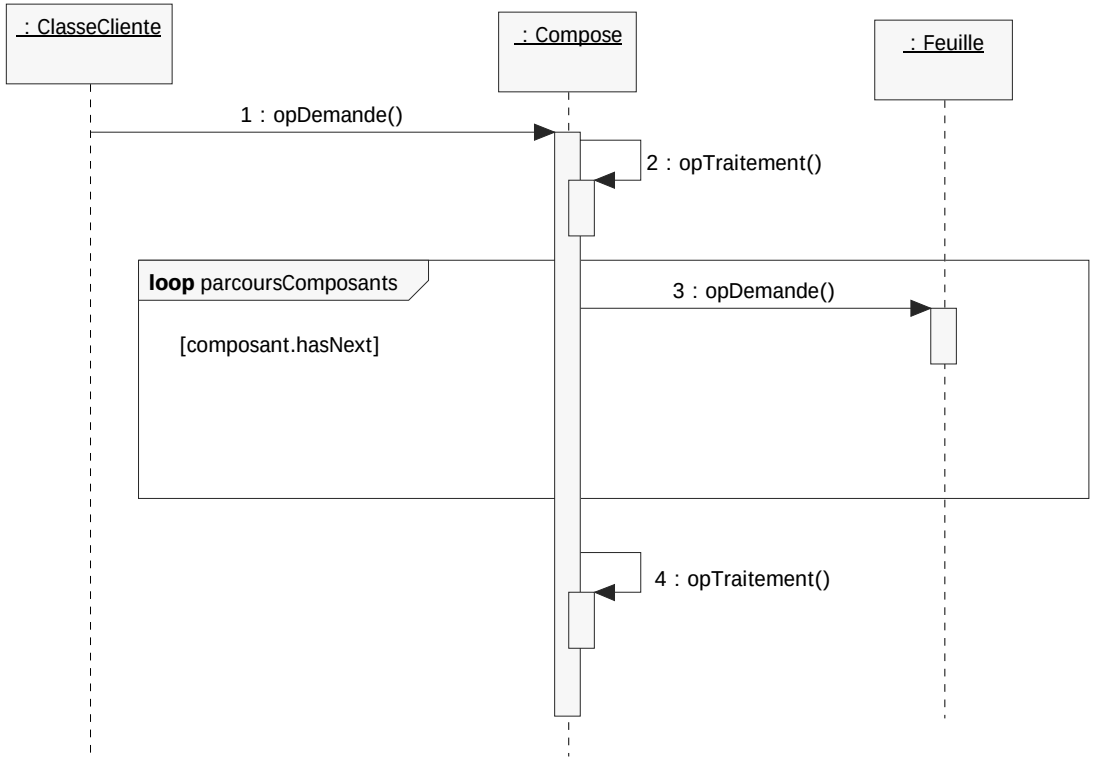



But : représenter une hiérarchie.
les clients ignorent s'ils communiquent
avec des objets composés ou non
Classe Abstraite :
-éventuellement attributs communs et méthodes commune
-interface des objets de la composition
-décrit la signature des méthodes d'ajout ou suppresion des composants
ClasseCliente
Element
+attributCommun
+operation()
+operationCommune()
+ajouterComposant()
+retirerComposant()
0..*
#composants
Feuille
+operation()
Compose
+ajouteComposant()
+retirerComposant()
+operation()
Operation(){
for (Element e: composants)
e.operation()
}

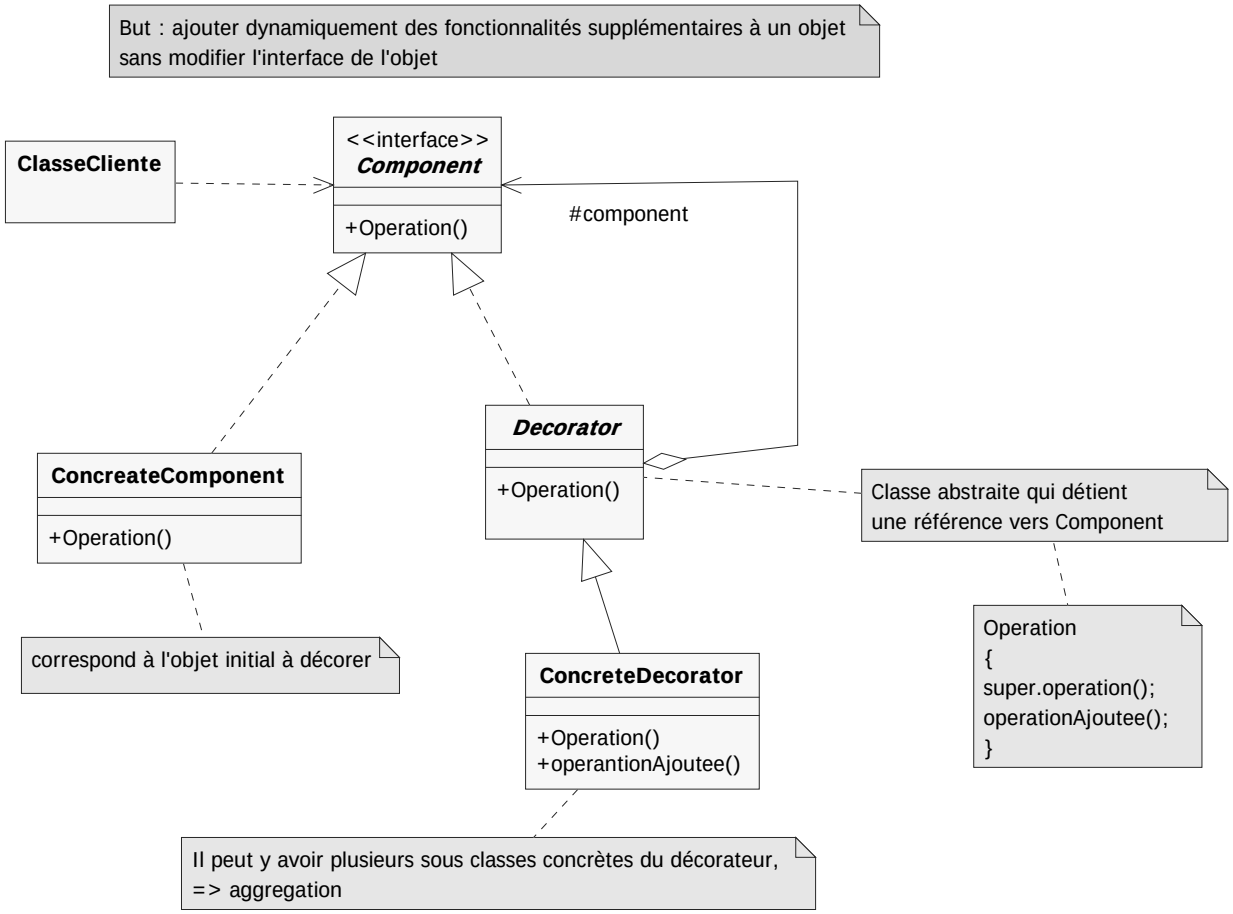
Untitled ::ModelesDP::DP_Structuration::CompositeDP::CollaborationInstanceSet1::InteractionInstanceSet1::SDfeuille



Untitled ::ModelesDP::DP_Structuration::CompositeDP::CollaborationInstanceSet2::InteractionInstanceSet1::SDCompose

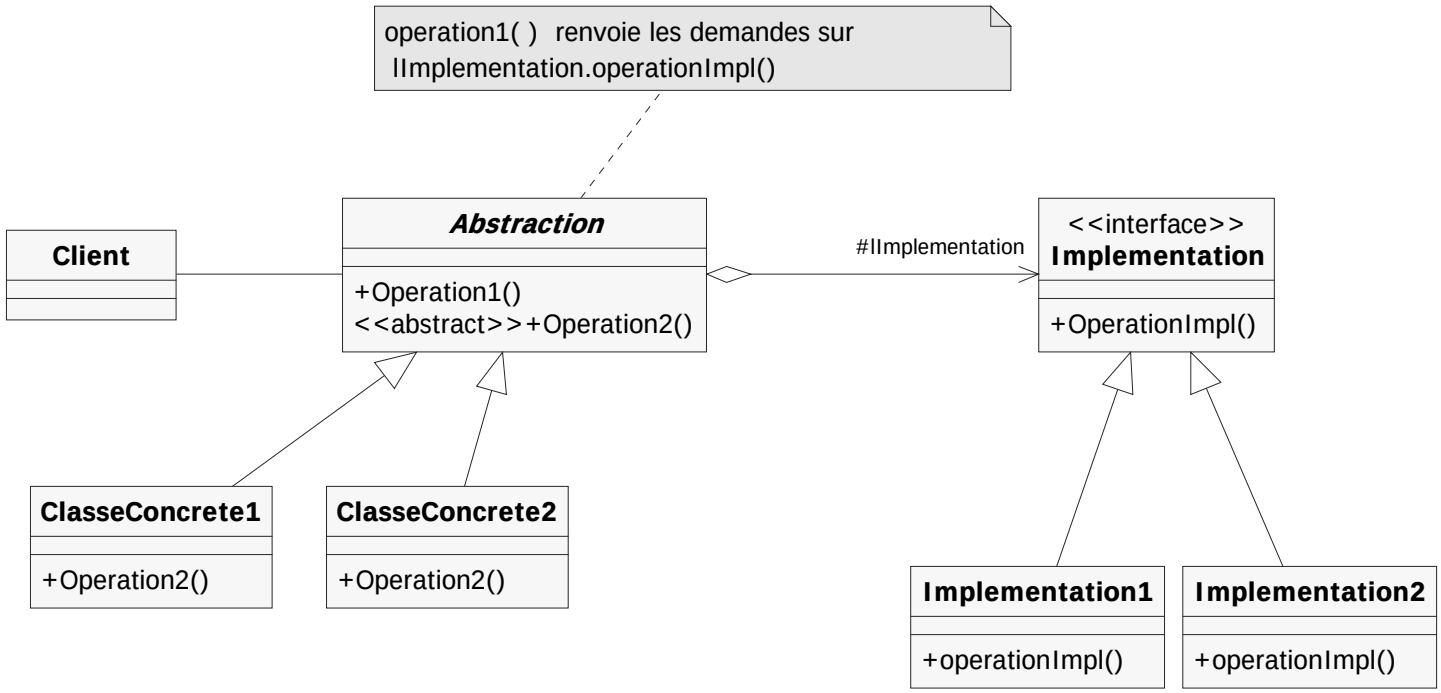


Untitled ::ModelesDP::DP_Structuration::DecoratorDP::DecoratorCD

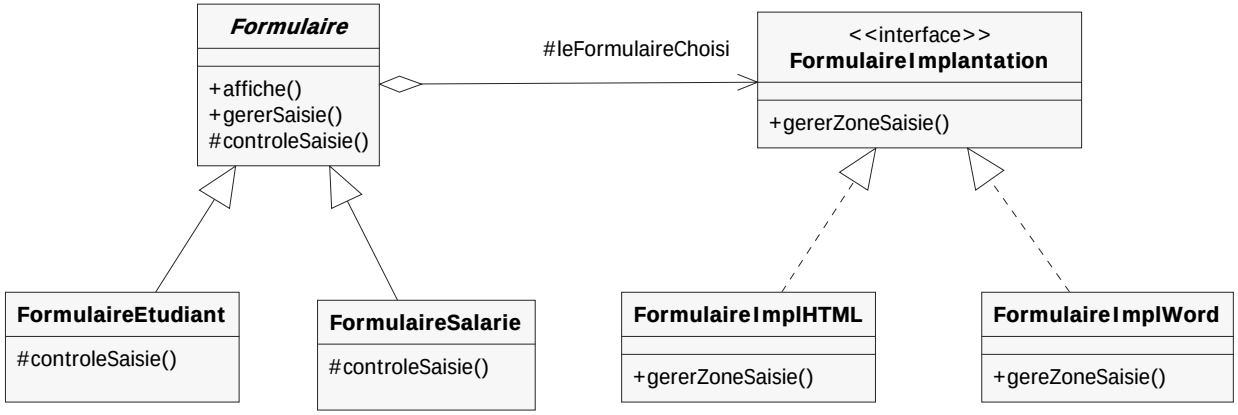


Untitled ::ModelesDP::DP_Structuration::BridgeDP::BridgeDP::BridgeCD

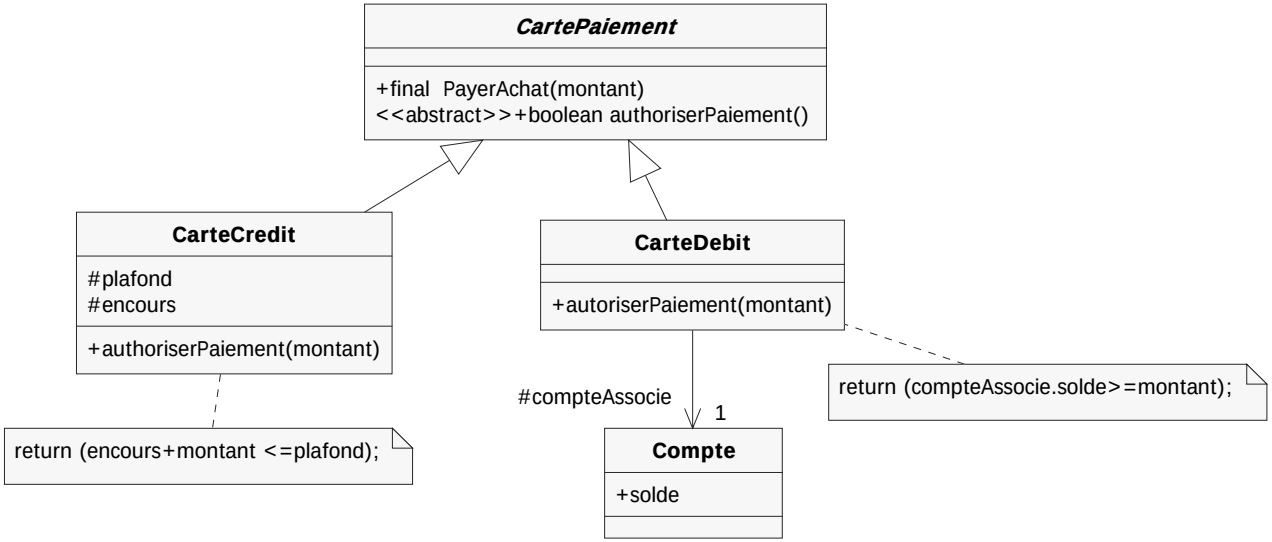
But : séparer l'aspect implémentation de sa modélisation (methodes d'interface)
ainsi l'implémentation et sa représentation par l'interface peuvent évoluer de façon indépendante



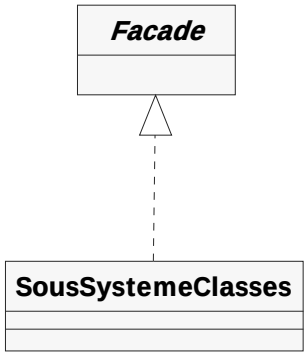
Untitled ::ModelesDP::DP_Structuration::BridgeDP::BridgeEx::BridgeFormulaireCD



Untitled ::ModelesDP::DP_Structuration::BridgeDP::BridgeEx::TemplateMethodEx::ClassDiagram1



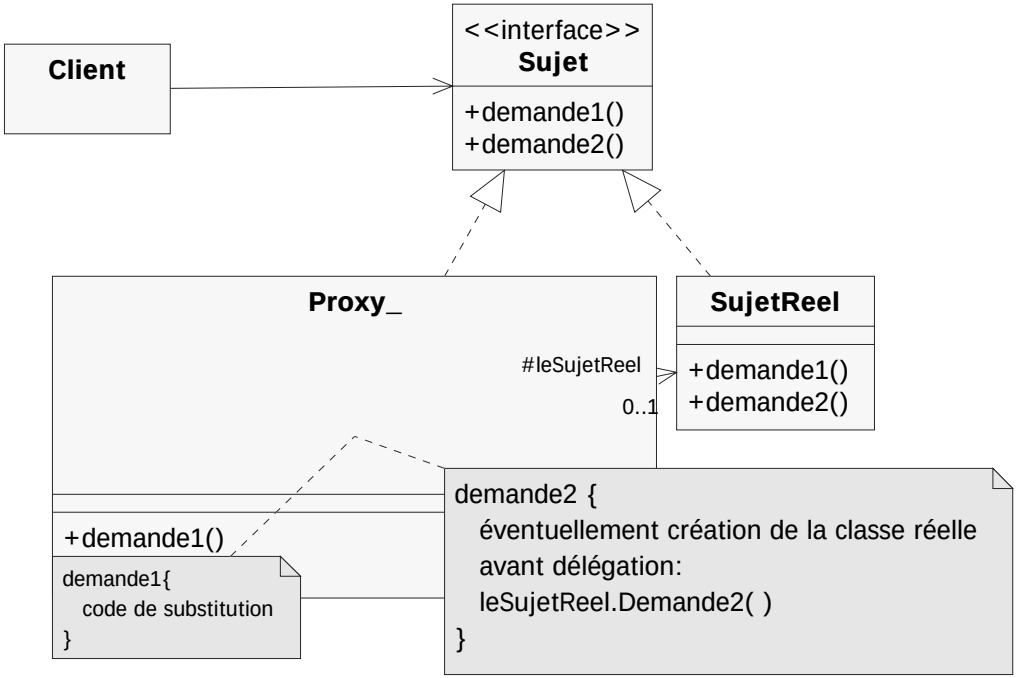
But: regouper dans une interface commune les interfaces de classes
pour rendre plus facile l'utilisation à une classe cliente.
Ce DP contribue à la modélisation de composants logiciels.



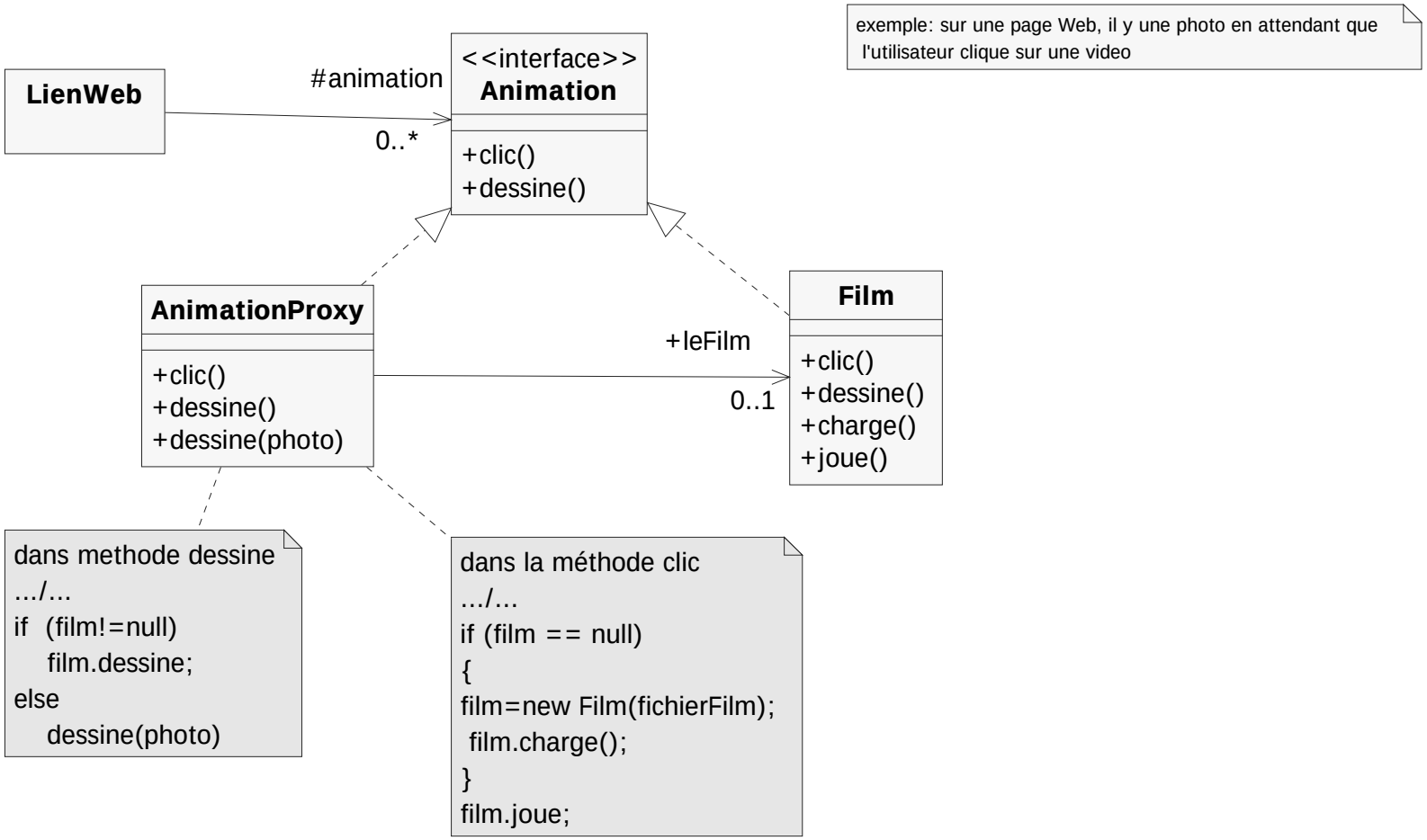
Chaque sous sytème est une hiérarchie de classes
qui implémente partiellement les signatures de la façade

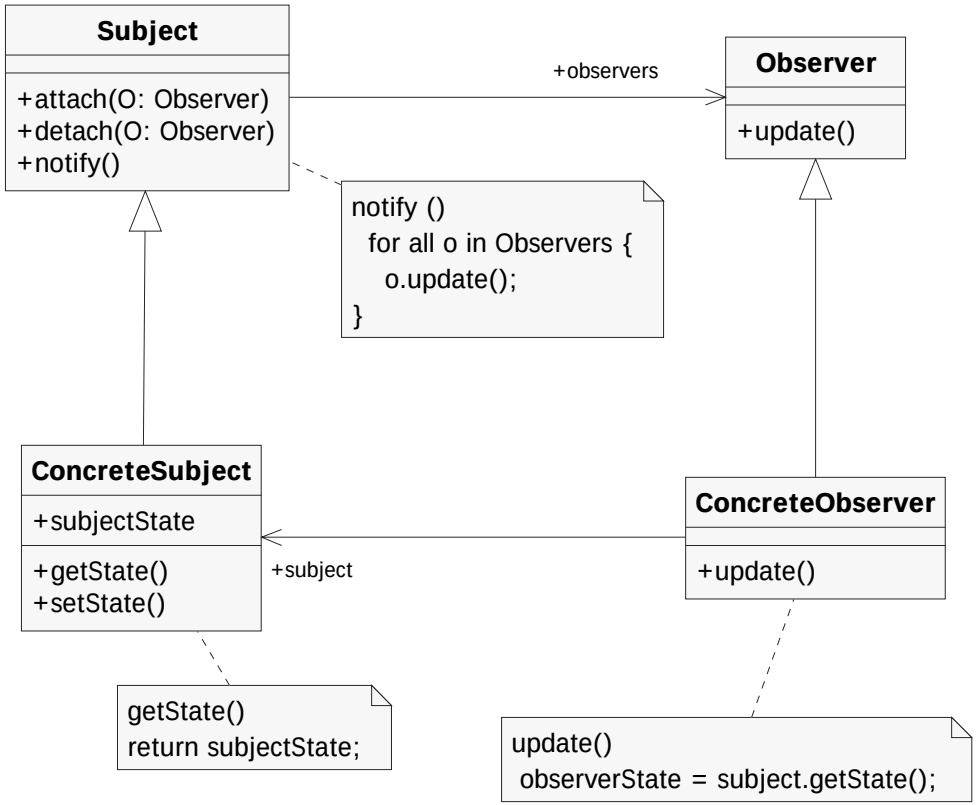
Untitled ::ModelesDP::DP_Structuration::ProxyDP::ProxyDP::ProxyCD

But : Le Proxy se substitue à la classe réelle de façon transparente
pour la classe cliente .
Ce mécanisme permet de retarder la création de la classe réelle quand on en a réellement besoin.

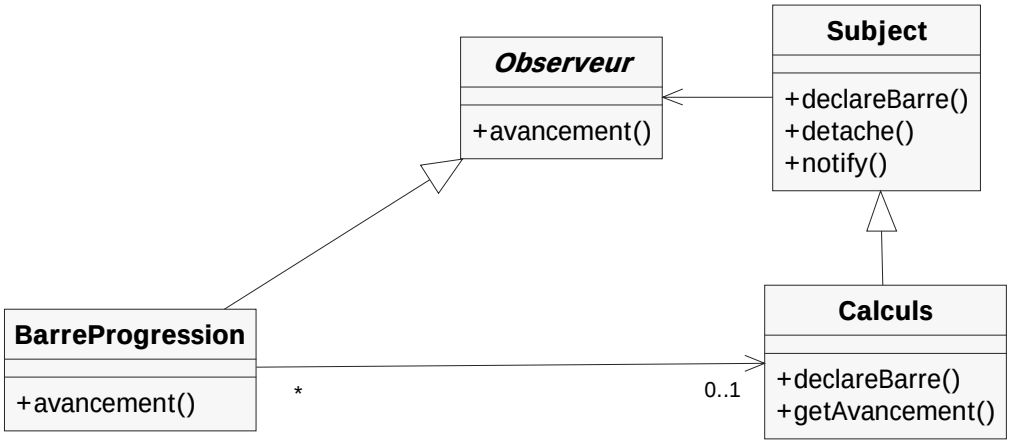


En fait 2 comportements sont possibles:
soit le Proxy effectue lui-même la demande par un code de substitution,
soit il la délègue à la classe réelle.

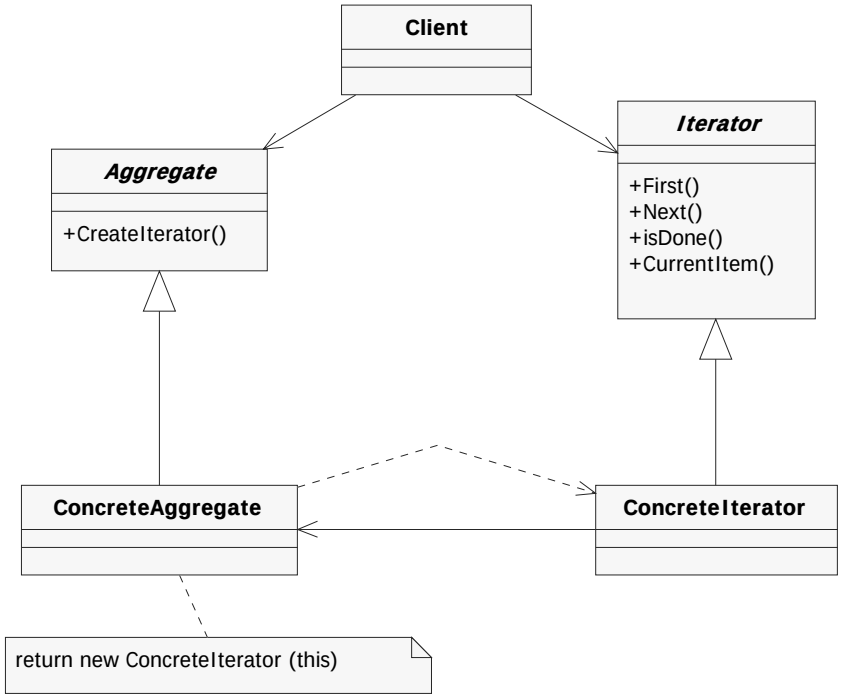

exemple: sur une page Web, il y une photo en attendant que l'utilisateur clique sur une video
LienWeb
#animation
0..*
<<interface>>
Animation
+clic()
+dessine()
AnimationProxy
+clic()
+dessine()
+dessine(photo)
+leFilm
0..1
Film
+clic()
+dessine()
+charge()
+joue()
dans methode dessine
.../...
if (film!=null)
film.dessine;
else
dessine(photo)
dans la méthode clic
.../...
if (film == null)
{
film=new Film(fichierFilm);
film.charge();
}
film.joue;


Subject
+attach(O: Observer)
+detach(O: Observer)
+notify()
+observers
Observer
+update()
notify ()
for all o in Observers {
o.update();
}
ConcreteSubject
+subjectState
+getState()
+setState()
+subject
ConcreteObserver
+update()
getState()
return subjectState;
update()
observerState = subject.getState();

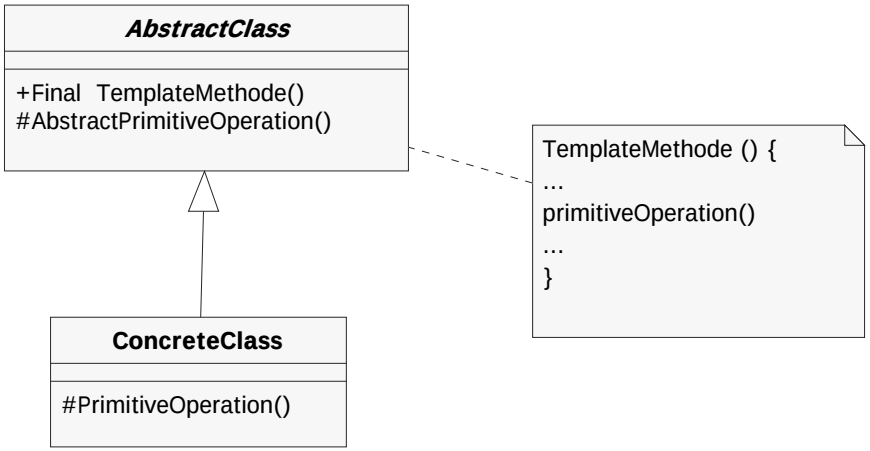
Untitled ::ModelesDP::DP_Comportement::Observer::ObserverExBarreProgression::ObserverExCD



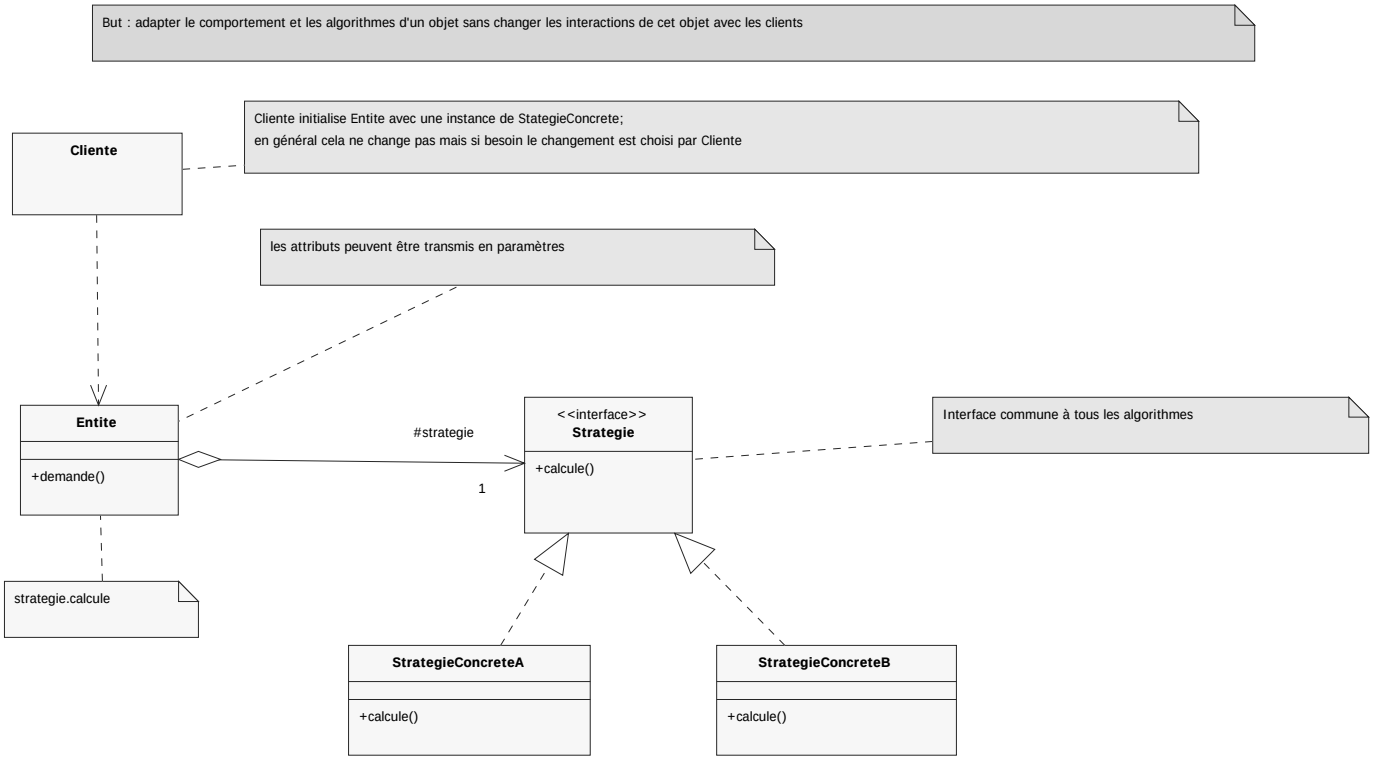
Untitled ::ModelesDP::DP_Comportement::Iterator::IteratorCD



Untitled ::ModelesDP::DP_Comportement::TemplateMethod::TemplateMethodCD



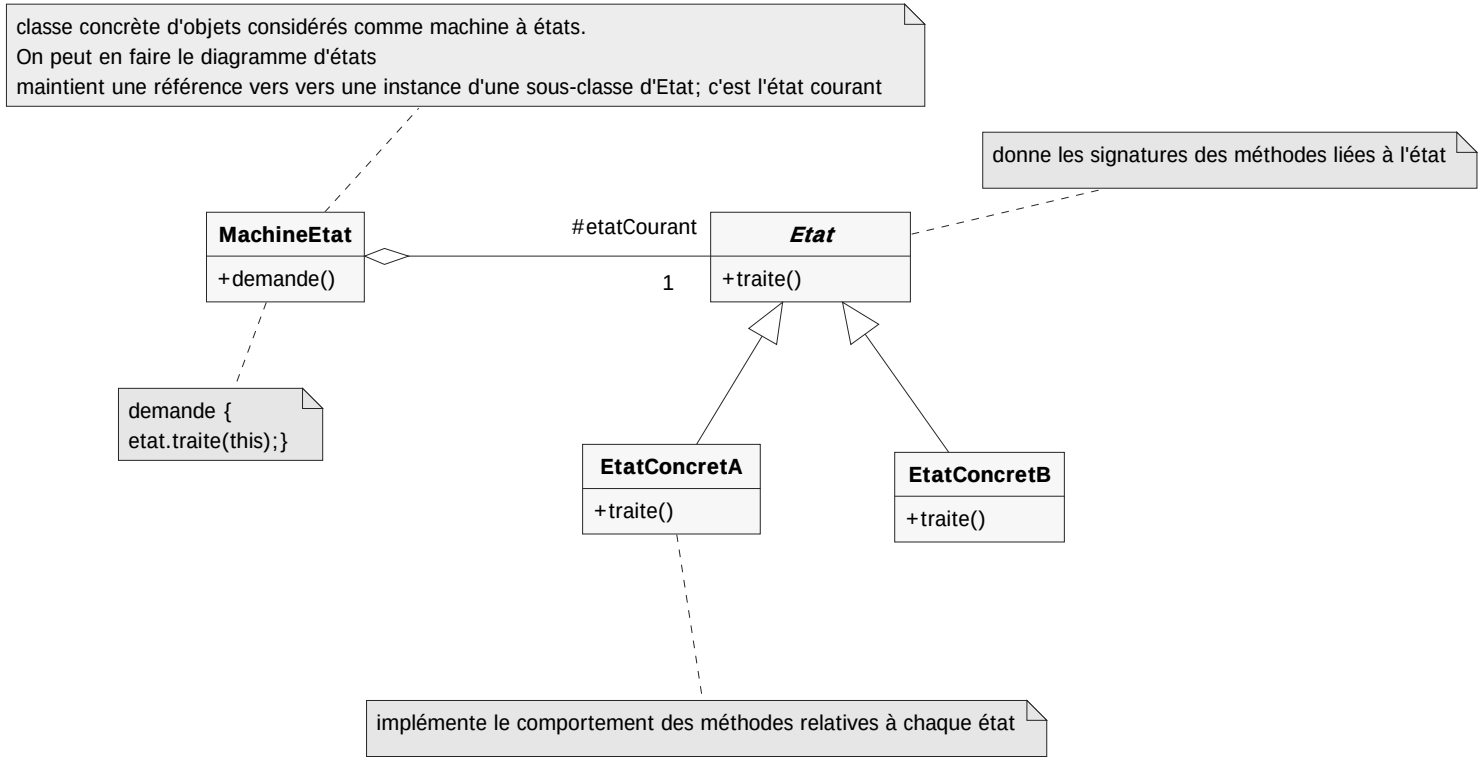
Untitled ::ModelesDP::DP_Comportement::StrategyPattern::StrategyCD



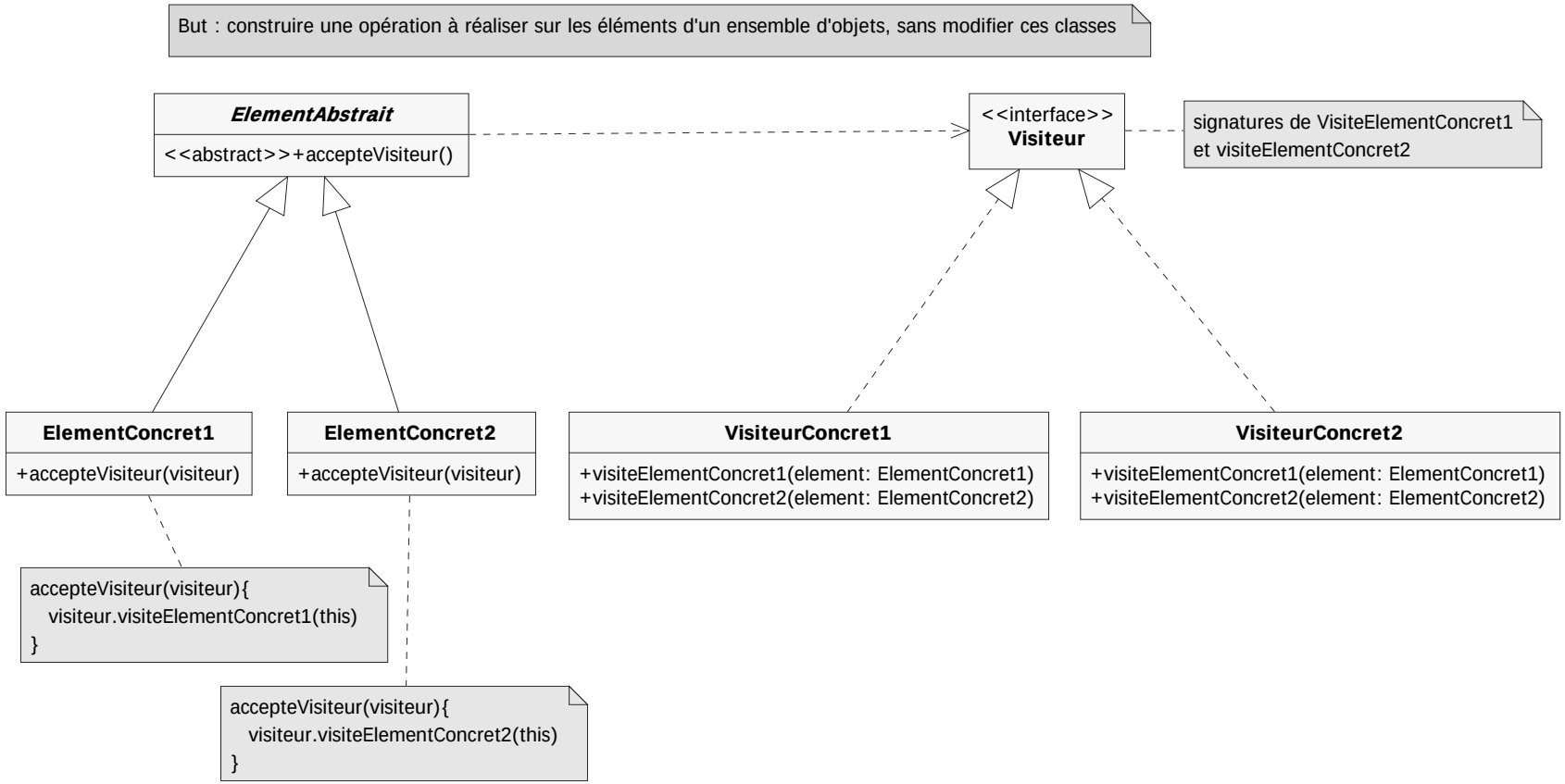
Untitled ::ModelesDP::DP_Comportement::State::StateCD

But : permettre à un objet d'adapter son comportement en fonction de son état interne

en évitant l'implantation de cett dépendance par des



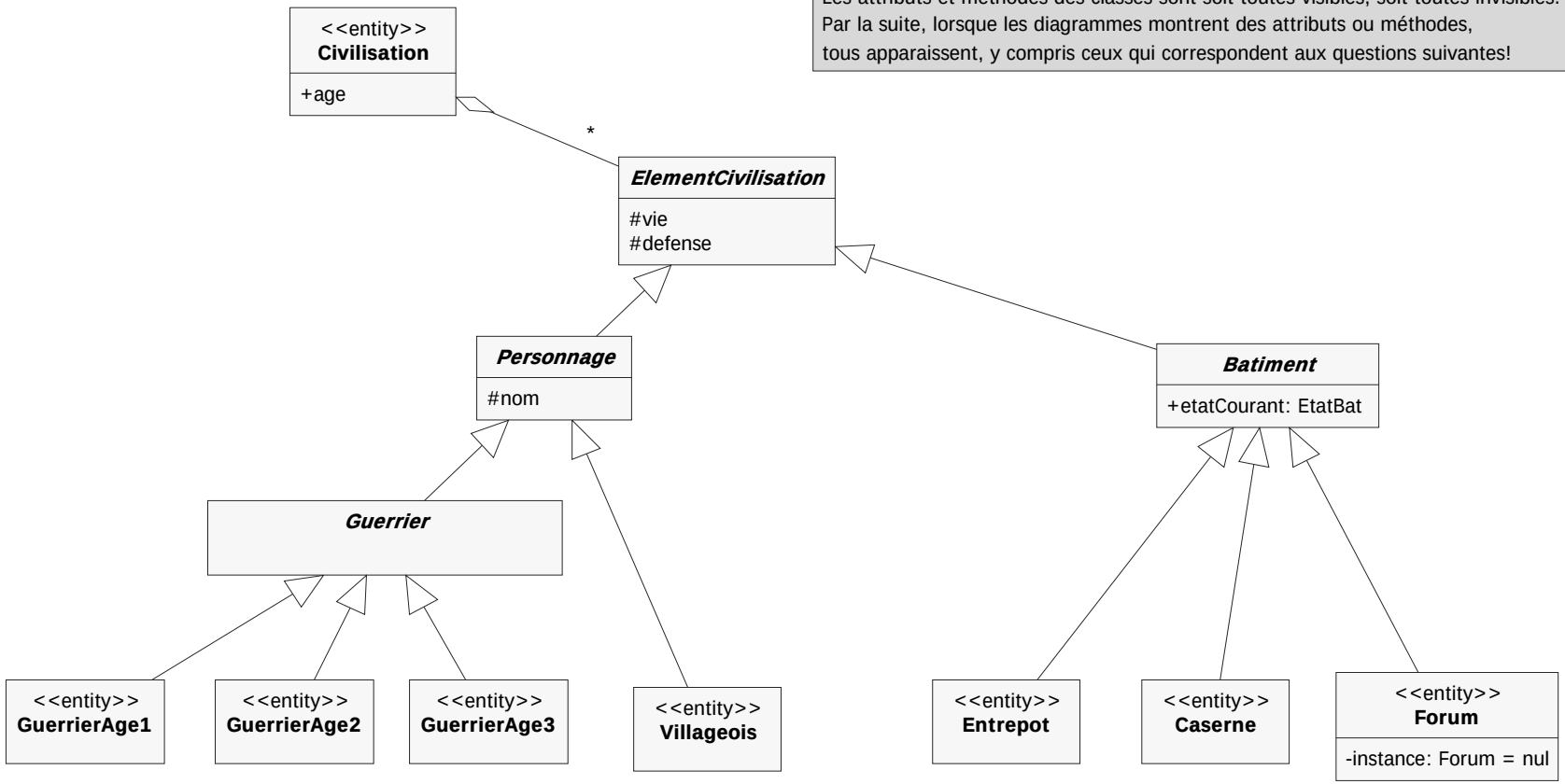
Untitled ::ModelesDP::DP_Comportement::Visitor::ClassDiagram1



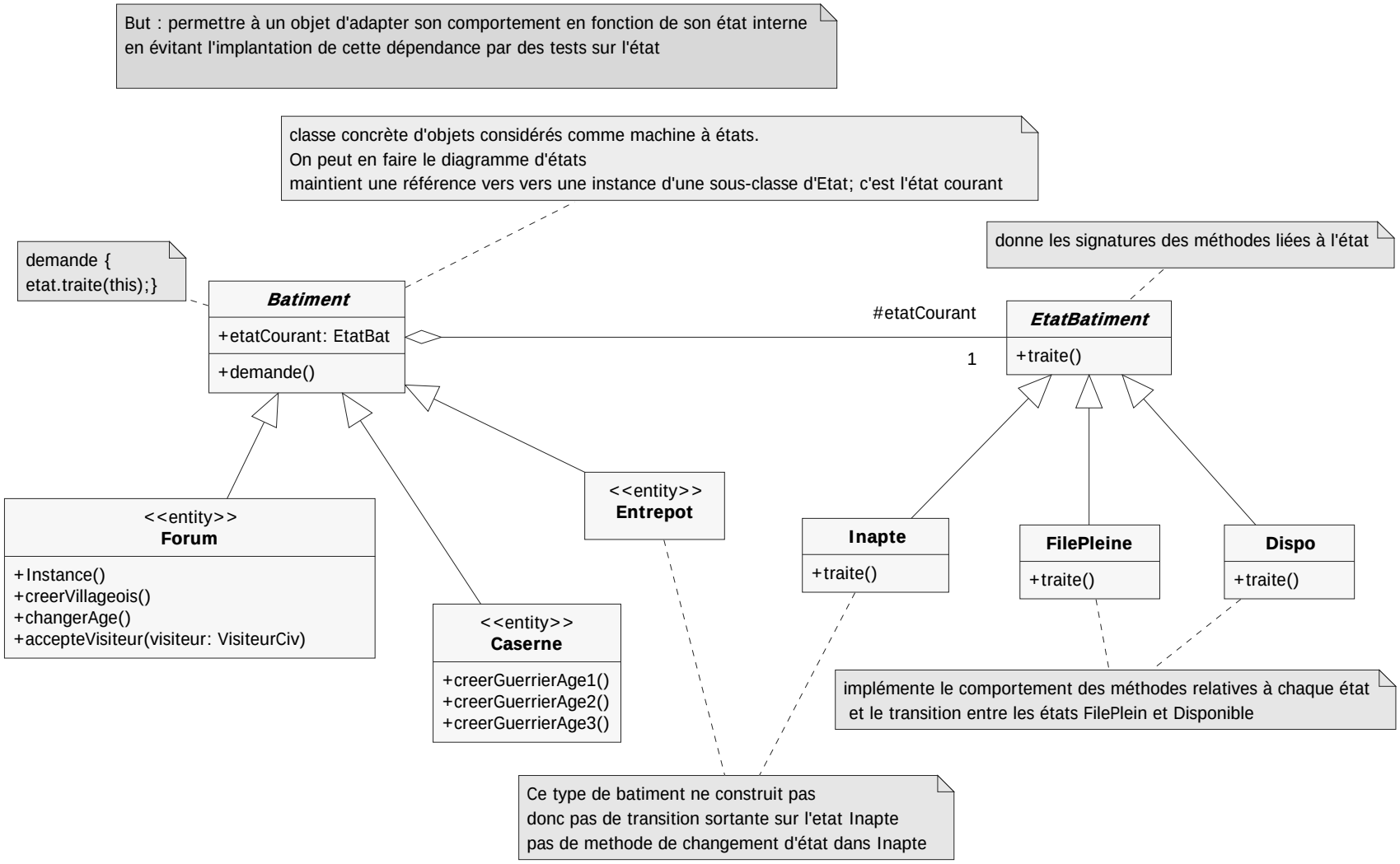
Untitled ::Jeu::ClassesCandidatesEntity::ClassesEntityJeuCD

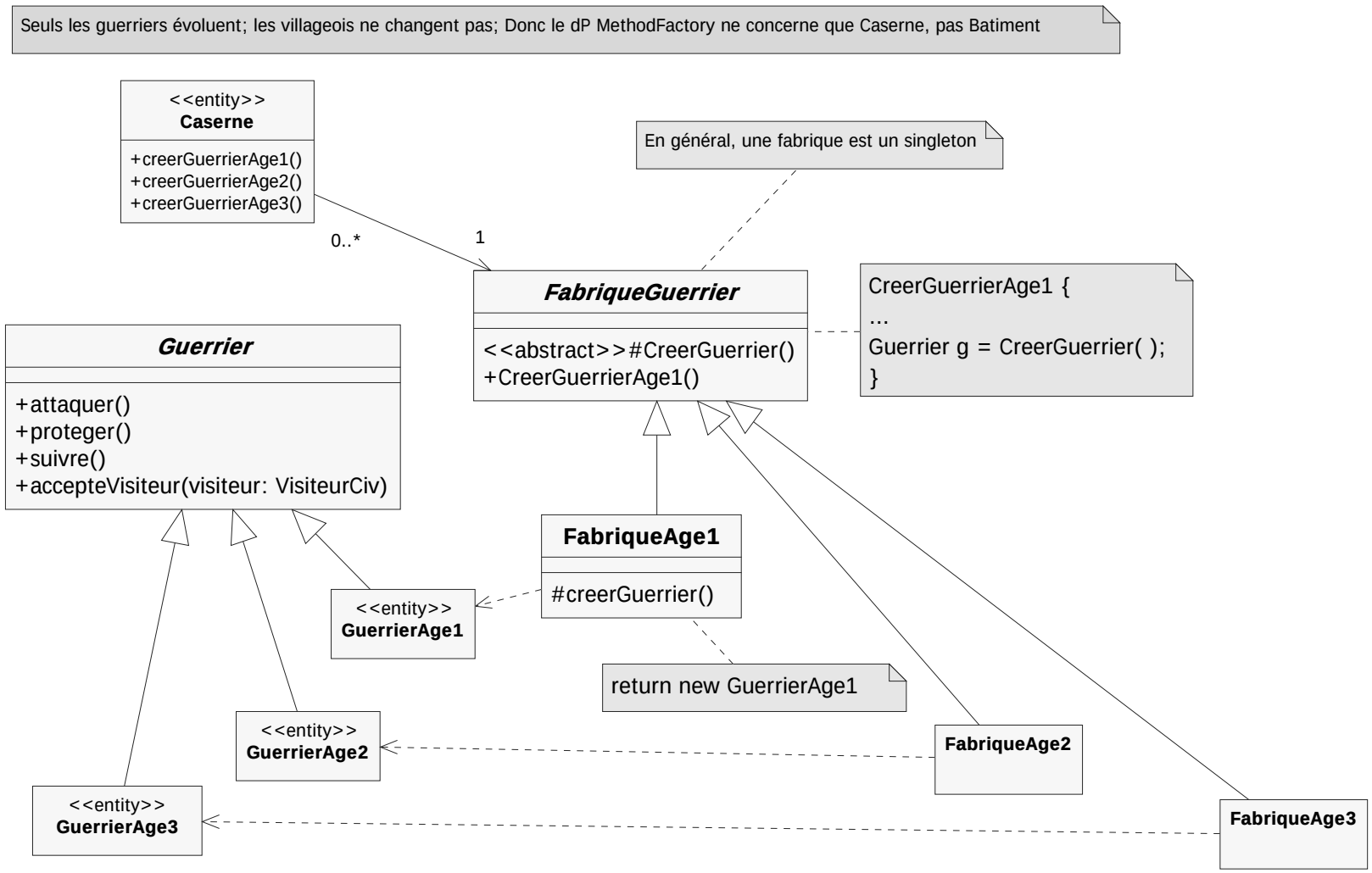
Ce diagramme de classes du "domaine" représente la civilisation d'un joueur
La attributs sont significatifs pour la structure d'une civilisation

Remarque sur l'utilisation de StarUML:
Les attributs et méthodes des classes sont soit toutes visibles, soit toutes invisibles.
Par la suite, lorsque les diagrammes montrent des attributs ou méthodes,
tous apparaissent, y compris ceux qui correspondent aux questions suivantes!

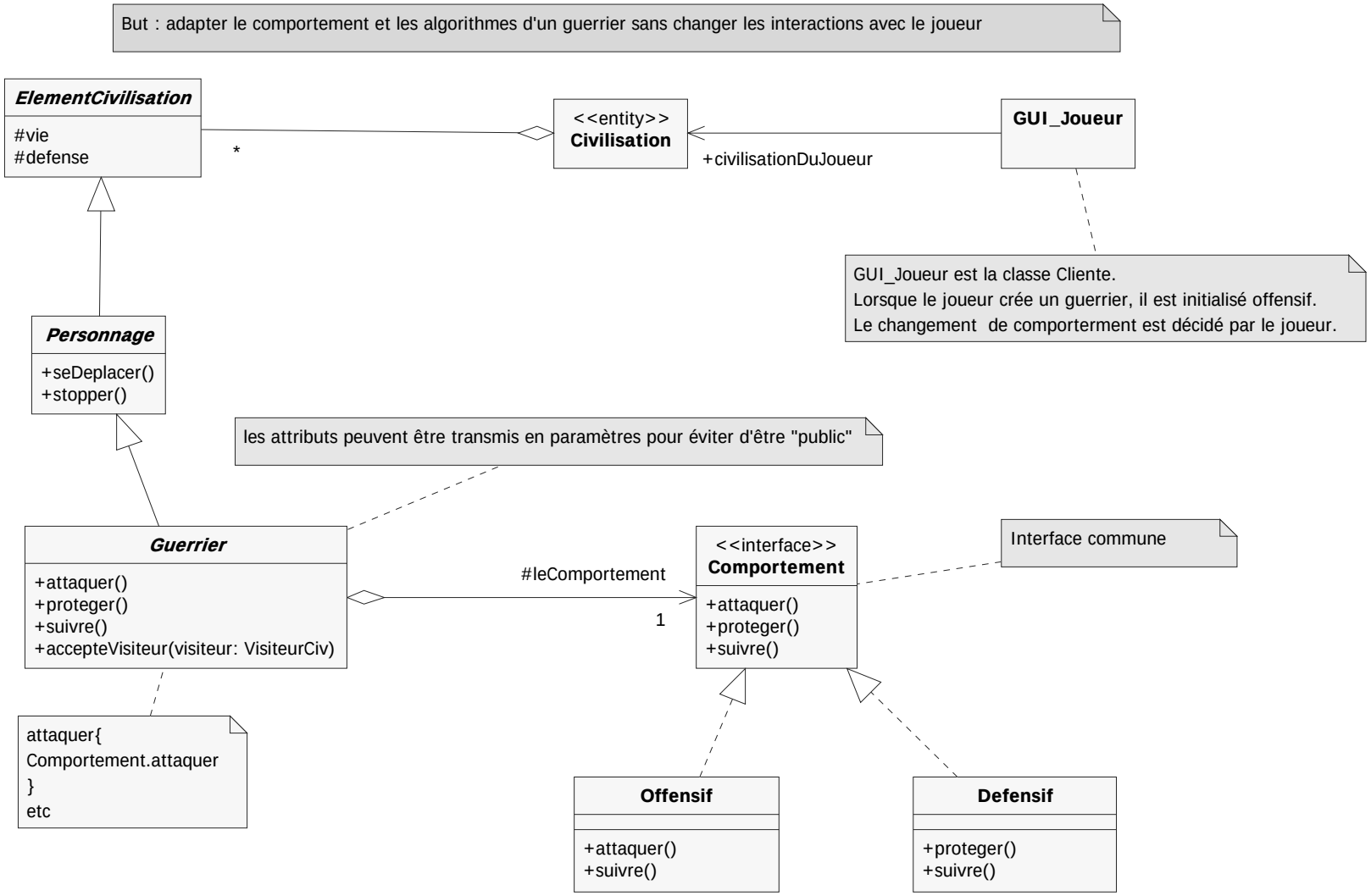


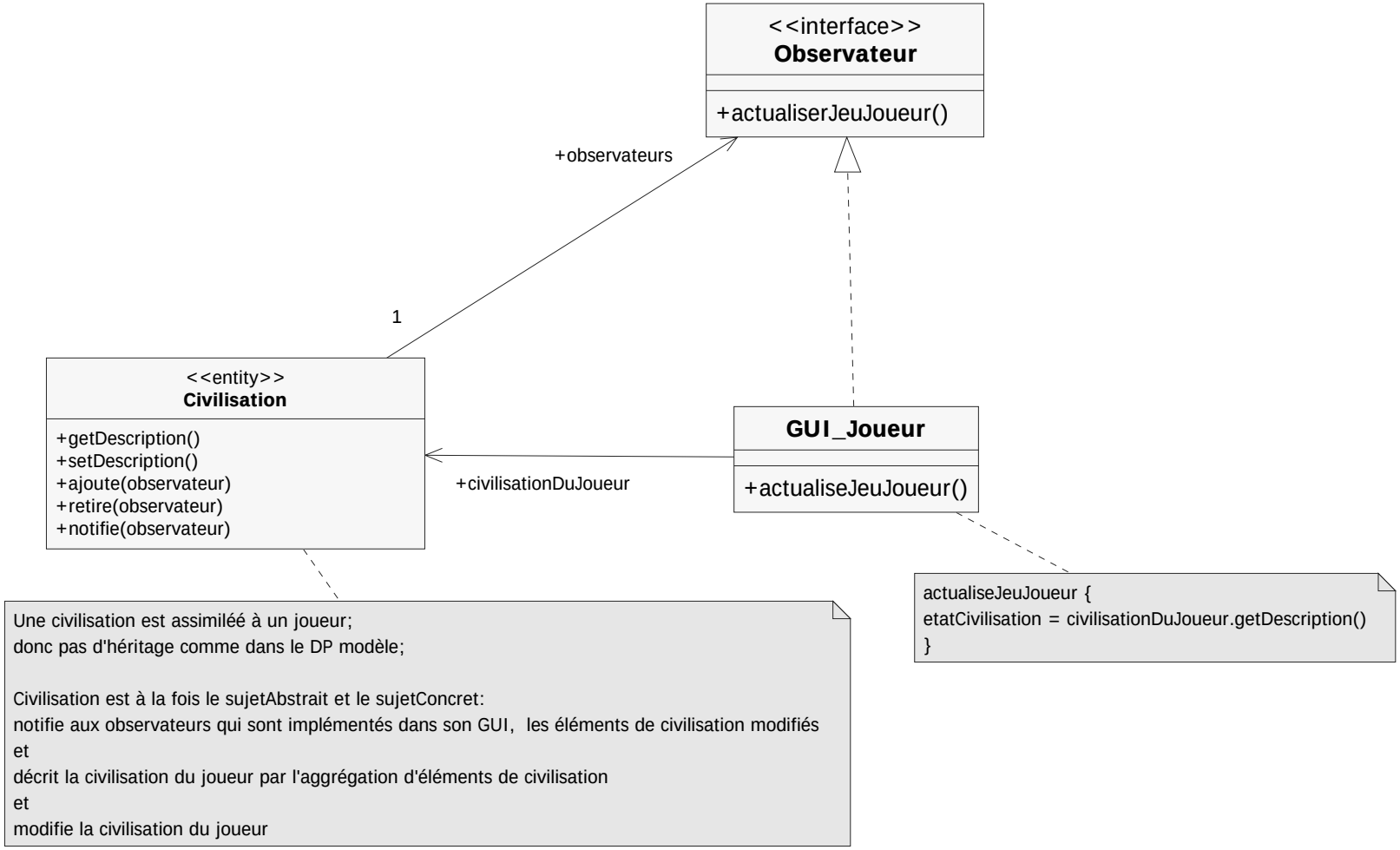
Untitled ::Jeu::State::StateCD



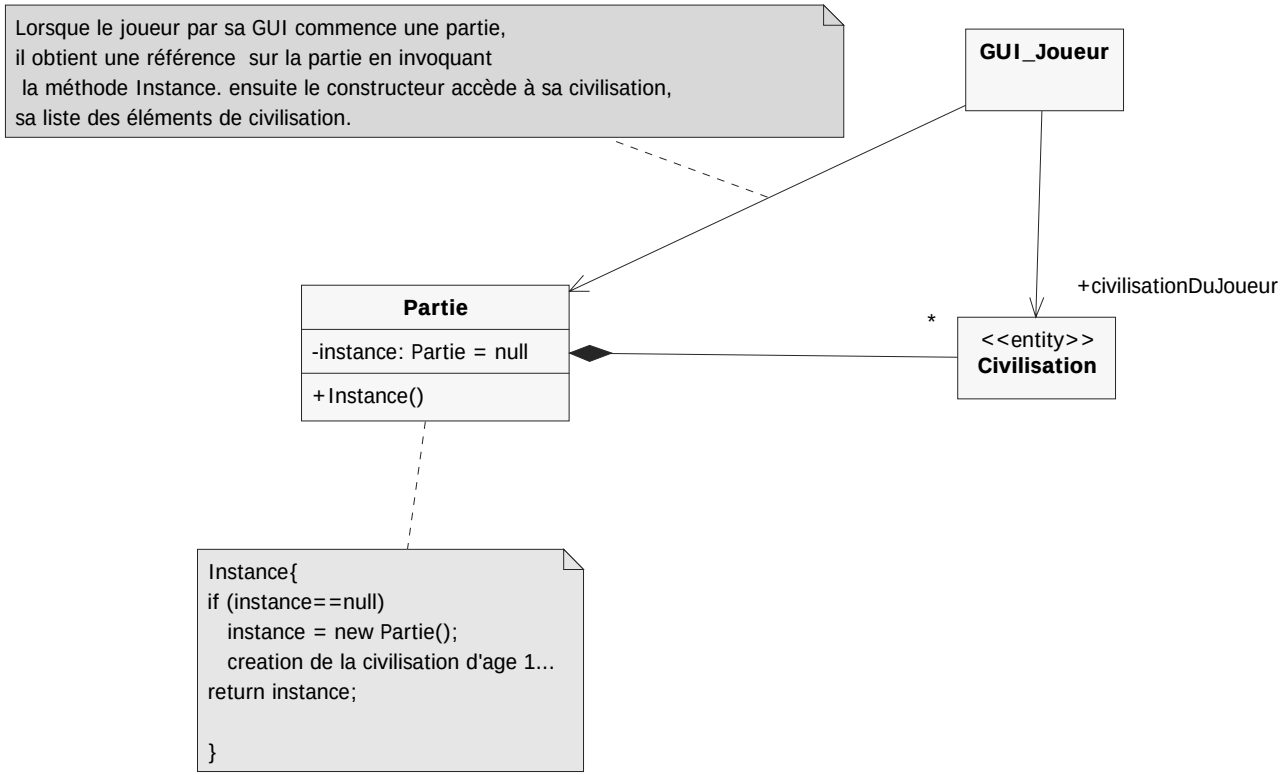

Seuls les guerriers évoluent; les villageois ne changent pas; Donc le dP MethodFactory ne concerne que Caserne, pas Batiment
<<entity>>
Caserne
+creerGuerrierAge1()
+creerGuerrierAge2()
+creerGuerrierAge3()
0..*
1
En général, une fabrique est un singleton
FabriqueGuerrier
<<abstract>>#CreerGuerrier()
+CreerGuerrierAge1()
CreerGuerrierAge1 {
...
Guerrier g = CreerGuerrier( );
}
Guerrier
+attaquer()
+proteger()
+suivre()
+accepteVisiteur(visiteur: VisiteurCiv)
FabriqueAge1
#creerGuerrier()
<<entity>>
GuerrierAge1
return new GuerrierAge1
<<entity>>
GuerrierAge2
FabriqueAge2
<<entity>>
GuerrierAge3
FabriqueAge3

Untitled ::Jeu::StrategyDP::StrategyCD

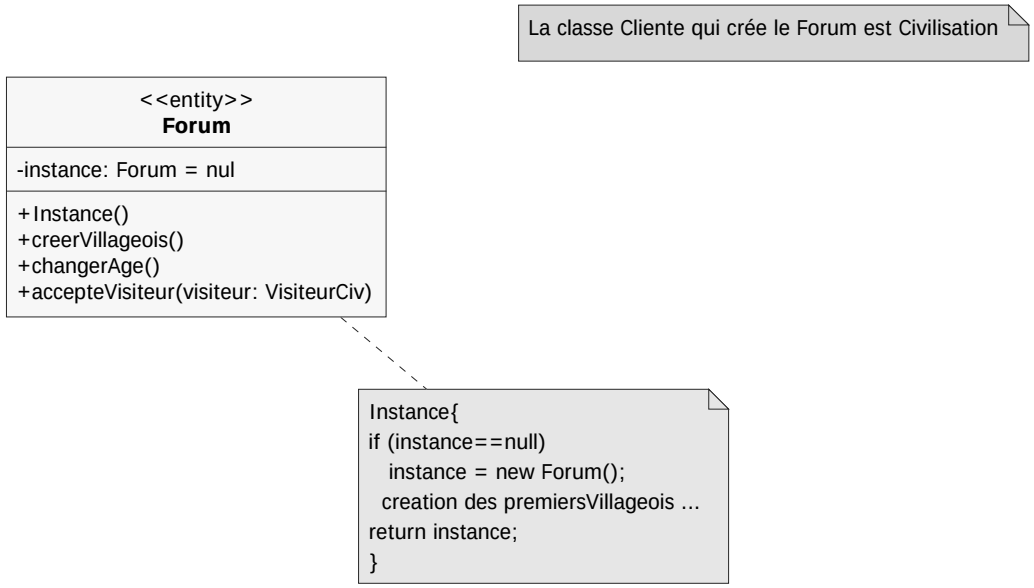



<<interface>>
Observateur
+actualiserJeuJoueur()
+observateurs
1
<<entity>>
Civilisation
+getDescription()
+setDescription()
+ajoute(observateur)
+retire(observateur)
+notifie(observateur)
+civilisationDuJoueur
GUI_Joueur
+actualiseJeuJoueur()
actualiseJeuJoueur {
etatCivilisation = civilisationDuJoueur.getDescription()
}
Une civilisation est assimiléé à un joueur;
donc pas d'héritage comme dans le DP modèle;
Civilisation est à la fois le sujetAbstrait et le sujetConcret:
notifie aux observateurs qui sont implémentés dans son GUI, les éléments de civilisation modifiés
et
décrit la civilisation du joueur par l'aggrégation d'éléments de civilisation
et
modifie la civilisation du joueur

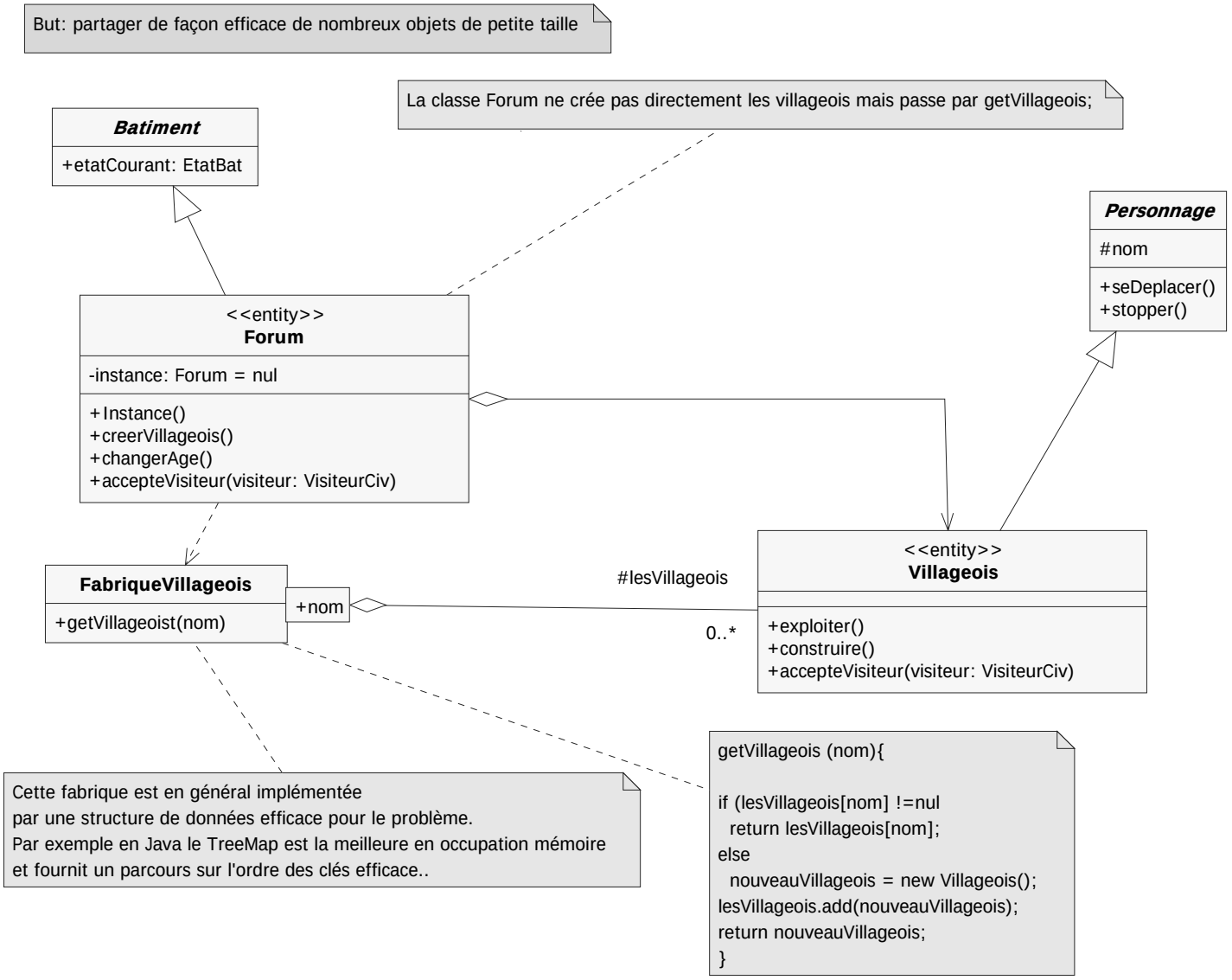
Untitled ::Jeu::SingletonDP::SingletonPartie::SingletonPartieCD

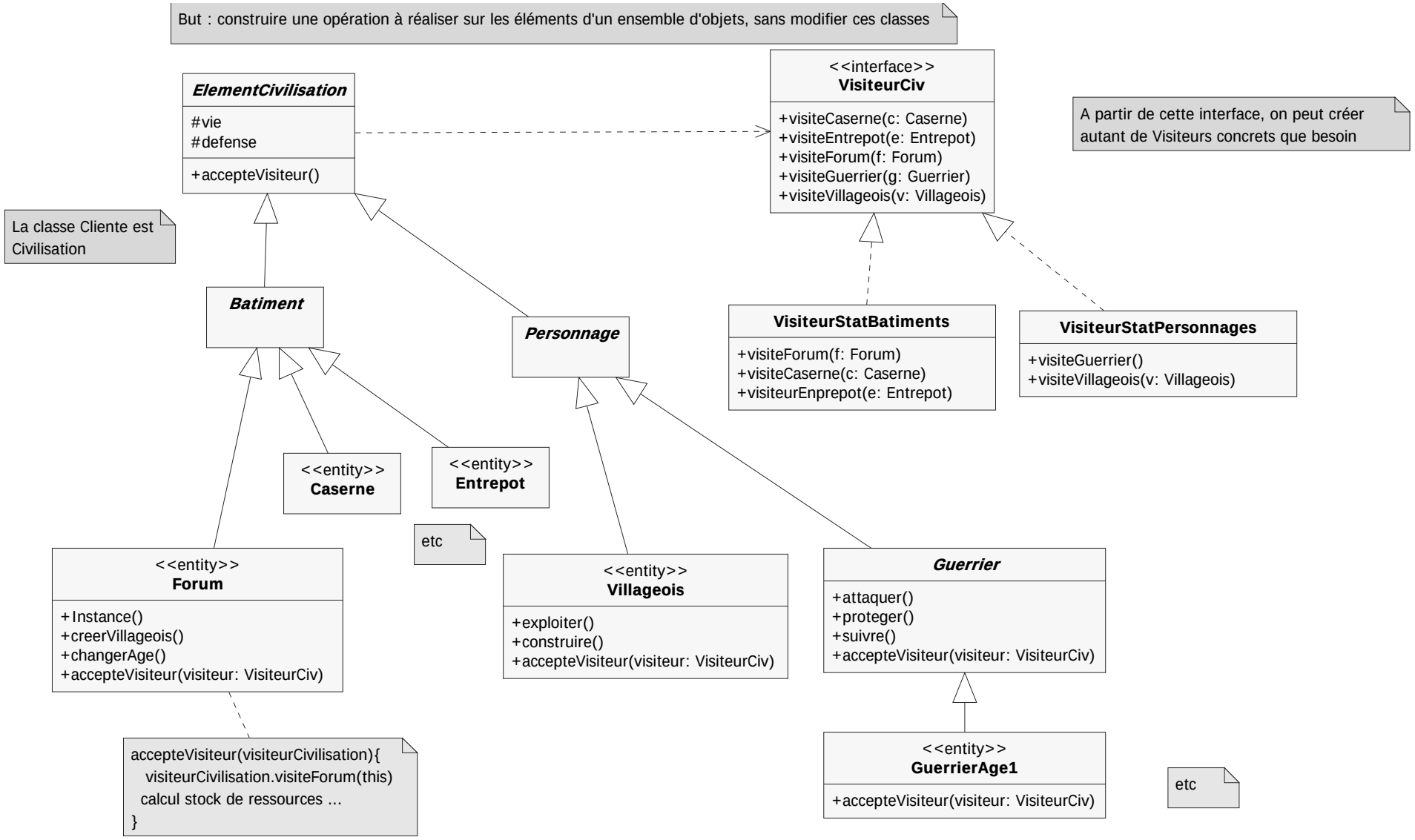


Untitled ::Jeu::SingletonDP::SingletonForum::SingletonForumCD



Untitled ::Jeu::FlyWeightDP::FlyWeihtCD




But : construire une opération à réaliser sur les éléments d'un ensemble d'objets, sans modifier ces classes
ElementCivilisation
#vie
#defense
+accepteVisiteur()
<<interface>>
VisiteurCiv
+visiteCaserne(c: Caserne)
+visiteEntrepot(e: Entrepot)
+visiteForum(f: Forum)
+visiteGuerrier(g: Guerrier)
+visiteVillageois(v: Villageois)
A partir de cette interface, on peut créer autant de Visiteurs concrets que besoin
La classe Cliente est Civilisation
Batiment
Personnage
VisiteurStatBatiments
+visiteForum(f: Forum)
+visiteCaserne(c: Caserne)
+visiteurEnprepot(e: Entrepot)
VisiteurStatPersonnages
+visiteGuerrier()
+visiteVillageois(v: Villageois)
<<entity>>
Caserne
<<entity>>
Entrepot
etc
<<entity>>
Forum
+Instance()
+creerVillageois()
+changerAge()
+accepteVisiteur(visiteur: VisiteurCiv)
<<entity>>
Villageois
+exploiter()
+construire()
+accepteVisiteur(visiteur: VisiteurCiv)
Guerrier
+attaquer()
+proteger()
+suivre()
+accepteVisiteur(visiteur: VisiteurCiv)
accepteVisiteur(visiteurCivilisation){
visiteurCivilisation.visiteForum(this)
calcul stock de ressources ...
}
<<entity>>
GuerrierAge1
+accepteVisiteur(visiteur: VisiteurCiv)
etc