IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Patron de conception

A. Beugnard & J.-C. Royer
DLR – MAPD – C5
2023

Objectifs de ce cours

1. Des éléments de conception
2. La notion de patron (de conception)
3. Un brin de philosophie et d'histoire
4. Un bout de catalogue de patrons

Contribution aux compétences

1. Conception
2. Développement

Je vous invite à prendre des notes...

1 Des éléments de conception

2 Patron

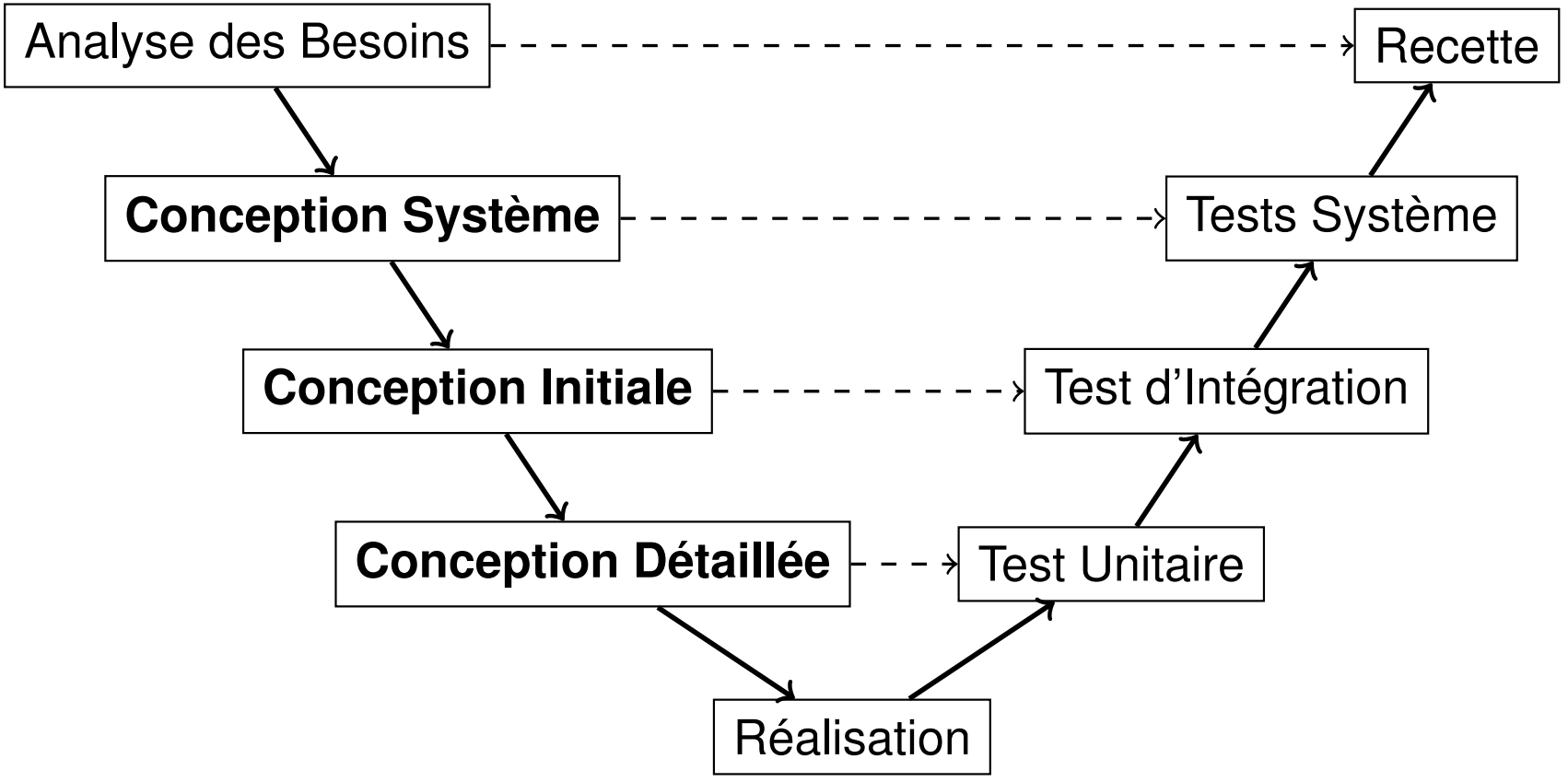
3 Un brin de philosophie et d'histoire

4 Un bout de catalogue de patrons de conception

5 Un exemple

6 Conclusion

Pour les grandes phases du développement



Après avoir compris le problème (identifié les exigences, élaboré un cahier des charges), on s'attaque aux solutions.

Il y a rarement une unique solution à un problème.

On va étudier plusieurs solutions (selon la taille du problème) :

- D'abord sans détails : l'architecture (conception système)
  - Choix d'une architecture
- Puis avec un peu de détails (conception initiale)
  - Choix des composants
- Puis avec un beaucoup de détails (conception détaillée)
  - Choix du contenu des composants

Trouver des arguments (justifications) pour choisir.

Ce qui fait un bon ingénieur, ce n'est pas de trouver une solution, c'est de *justifier* la solution proposée.

On s'appuie sur :

- Les exigences fonctionnelles, non fonctionnelles
- Les contraintes de coût, de délai, légales, etc.
- Son expertise, son expérience

Ce qui fait un bon ingénieur, ce n'est pas de trouver une solution, c'est de *justifier* la solution proposée.

On s'appuie sur :

- Les exigences fonctionnelles, non fonctionnelles
- Les contraintes de coût, de délai, légales, etc.
- Son expertise, son expérience

Il est donc fondamental de capitaliser son expérience.

Ce qui fait un bon ingénieur, ce n'est pas de trouver une solution, c'est de *justifier* la solution proposée.

On s'appuie sur :

- Les exigences fonctionnelles, non fonctionnelles
- Les contraintes de coût, de délai, légales, etc.
- Son expertise, son expérience

Comment ?

- Dans sa tête (se partage peu)
- Dans des livres (individuels ou collectifs)
- Dans des standards (consensus métier)
- Dans des catalogues de bonnes pratiques (consensus)
- ...

Il faut :

- Reconnaître une bonne pratique
- La décrire, la mettre en forme
- La partager

Ce n'est pas facile...

1 Des éléments de conception

2 Patron

3 Un brin de philosophie et d'histoire

4 Un bout de catalogue de patrons de conception

5 Un exemple

6 Conclusion

Que faut-il décrire ?

- le problème
- la solution connue proposée

Que faut-il décrire ?

- le problème
- la solution connue proposée
- le contexte d'application

Que faut-il décrire ?

- le problème
- la solution connue proposée
- le contexte d'application
- les avantages et inconvénients connus

Que faut-il décrire ?

- le problème
- la solution connue proposée
- le contexte d'application
- les avantages et inconvénients connus
- des exemples d'utilisation

Que faut-il décrire ?

- le problème
- la solution connue proposée
- le contexte d'application
- les avantages et inconvénients connus
- des exemples d'utilisation
- un nom

Le nom permet d'enrichir son vocabulaire. On peut dire : « j'utilise un <nom> » ou « je reconnais un <nom>. »

Un nom, un problème, une solution, un contexte, des propriétés, des exemples.

Solution particulière mais éprouvée à un problème récurrent

Permet de justifier des choix de conception

Les patrons ne résolvent pas tout

Problème 1 + patron => Problème 2

Deux façons d'utiliser un patron :

1. Pour décrire une solution (descriptif, passif)
2. Pour élaborer une solution (prescriptif, actif)

Une brique Un patron est dépendant d’un contexte, d’un environnement

Une règle Ne s’applique pas mécaniquement (sans réfléchir, sans adaptation)

Une méthode Ne dit pas quelle question se poser ni dans quelle ordre pour prendre une décision. Un patron est le résultat d’une décision

Nouveau On en utilise en architecture depuis longtemps... pas toujours consciemment

- Encapsule des connaissance, normalise un vocabulaire
- Aide à trouver une solution
- Abstrait, dans le cas des patrons de conception (objet) ; préconise des usages des classes, de l'héritage et de la liaison dynamique efficaces (même sans le comprendre)

- Des efforts pour les apprendre, les connaître
- Difficiles à identifier (prise de recul) – nécessite de l'expérience et de l'expertise
- Il en existe beaucoup, beaucoup (par problème, par domaine, etc.)

1 Des éléments de conception

2 Patron

3 Un brin de philosophie et d'histoire

4 Un bout de catalogue de patrons de conception

5 Un exemple

6 Conclusion

Patrons... en peinture

Patrons... en peinture

The purpose is to figure out the universal patterns that lies hidden underneath the apparent chaos on the surface

. . . if you see them, you win !

If you can only read the surface, you can not build anything.

If you want to build something strong, use the patterns.

"Architectures can't be made, but only generated, indirectly, by the ordinary actions of the people, just as a flower cannot be made but only generated from the seed."

Christopher Alexander

There is something fundamentally wrong with 20th century architectural design methods and practices. Contemporary methods fail :

- To generate products that satisfy the true requirement placed upon them
- To meet the real demands of the real users
- To "improve the human condition"

*“By following the **way**, one may pass through the **gate** to reach the **quality**”*
Christopher Alexander

- The quality. . . is the purpose, the requirements
- The gate. . . is the tool
- The way. . . is acting

Nom	du patron. Enrichit le vocabulaire
Exemple	un usage réel
Contexte	une situation qui fait apparaître le problème
Problème	le problème dans ce contexte
Forces	des contraintes
Solution	une résolution éprouvée du problème

Exemple

Contexte	Il faut attendre
Problème	faire passer le temps (activement ou passivement)
Forces	quelle durée d'attente ? faut-il occuper l'esprit ?
Solution	pièce/zone selon le contexte, relaxante et non confinée
Nom	Salle d'attente
Exemple	Chez le dentiste, à l'aéroport...

Contexte	passer d'une pièce à une autre
Problème	ne pas être gêné dans le déplacement
Forces	séparation des pièces ? visuelle ? sonore ?
Solution	un trou dans le mur du sol de hauteur d'homme (au moins) fermé par un dispositif qui dépend du contexte

Contexte	passer d'une pièce à une autre
Problème	ne pas être gêné dans le déplacement
Forces	séparation des pièces ? visuelle ? sonore ?
Solution	un trou dans le mur du sol de hauteur d'homme (au moins) fermé par un dispositif qui dépend du contexte
Nom	Porte
Exemple	porte blindée, porte vitrée, porte insonorisée

1 Des éléments de conception

2 Patron

3 Un brin de philosophie et d'histoire

4 Un bout de catalogue de patrons de conception

5 Un exemple

6 Conclusion

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, 1995, Addison Wesley, ISBN : 0- 201-63361-2

Dit le *Gang of Four* [GoF].

Creational	Singleton, Prototype, Factory method, *Abstract Factory*[1], Builder
Structural	Composite, *Proxy*, Facade, Adaptor,. . .
Behavioural	Observer, *State*, Strategy, Mediator, *Visitor*, . . .

1. En annexe.

Motivation

- Assurer qu'une classe a une unique instance et fournit un seul point d'accès

Applicabilité/contraintes

- Une seule instance nécessaire

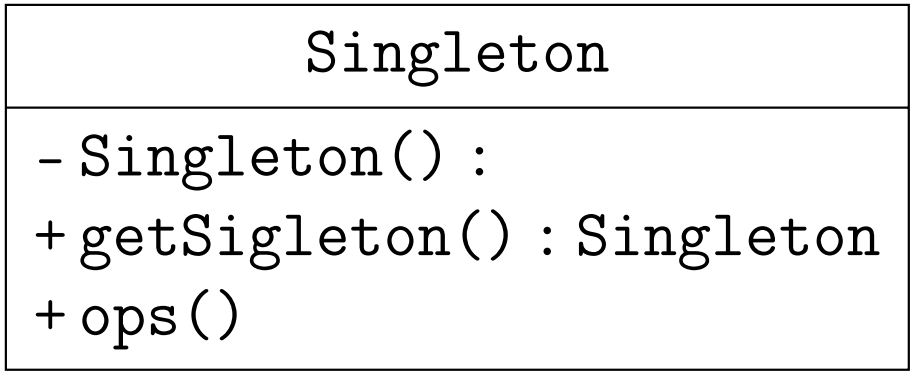
Participants/rôles

- Singleton : l'instance unique

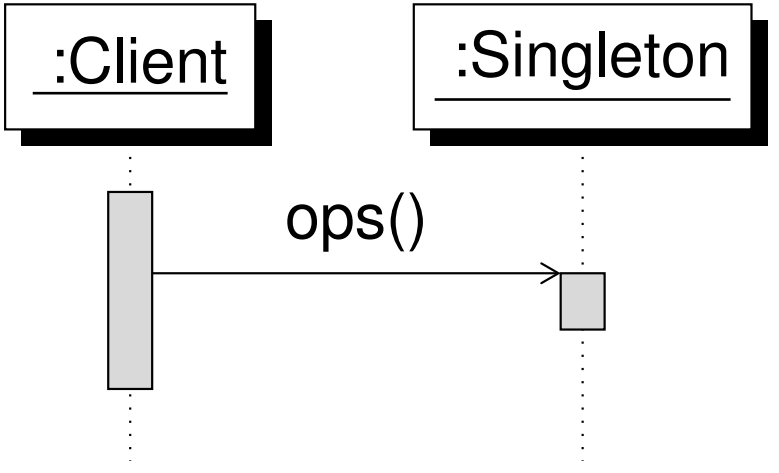
Conséquences

- création/suppression efficace
- point d'accès unique

Structure



Collaboration



- Une constante dans une interface pourrait remplir ce rôle (depuis Java 8)
- Une solution un peu rigide est de recourir au qualificatif `static` qui remplit en partie ce rôle mais est considérée comme une mauvaise pratique
- Toutefois la création d'une instance est plus flexible et plus évolutive grâce à l'héritage
- Mettre le constructeur en privé évite des créations intempestives...
- Ici cas simple : héritage et concurrence peuvent notablement compliquer les choses... `https://www.oracle.com/technetwork/articles/java/singleton-1577166.html`, attention très avancé !

Prototype (1/2)

Motivation

- Créer un objet à partir une instance modèle

Applicabilité/contraintes

- Quand le type est décidé dynamiquement

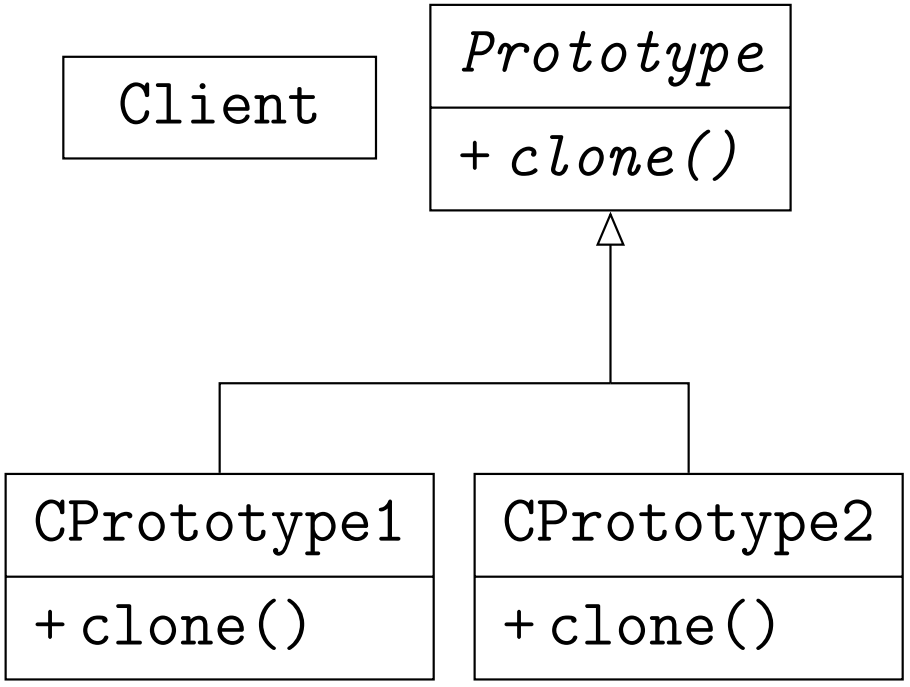
Participants/rôles

- Client : celui qui veut créer un objet
- Prototype : interface du service de clonage
- ConcretePrototype : l'implantation du service de clonage

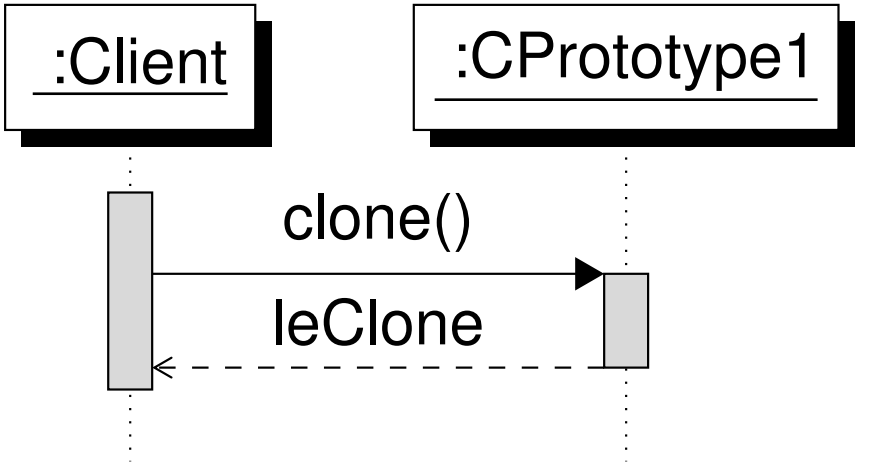
Conséquences

- type du clone déterminé à l'exécution

Structure



Collaboration



Il y a une hiérarchie des créations selon la forme du « plan » de construction.

1. Singleton : pas de plan
2. Clone : le plan est une instance (le clone)
3. Factory method : le plan est dans une méthode
4. Abstract Factory : le plan est lié à une classe
5. Builder : le plan est réifié en un objet qui doit être interprété pour construire un objet

Composite (1/3)

Motivation

- Manipuler des objets simples et composés de la même façon

Applicabilité/contraintes

- quand on ne veut pas faire de différence entre un objet isolé et des objets assemblés

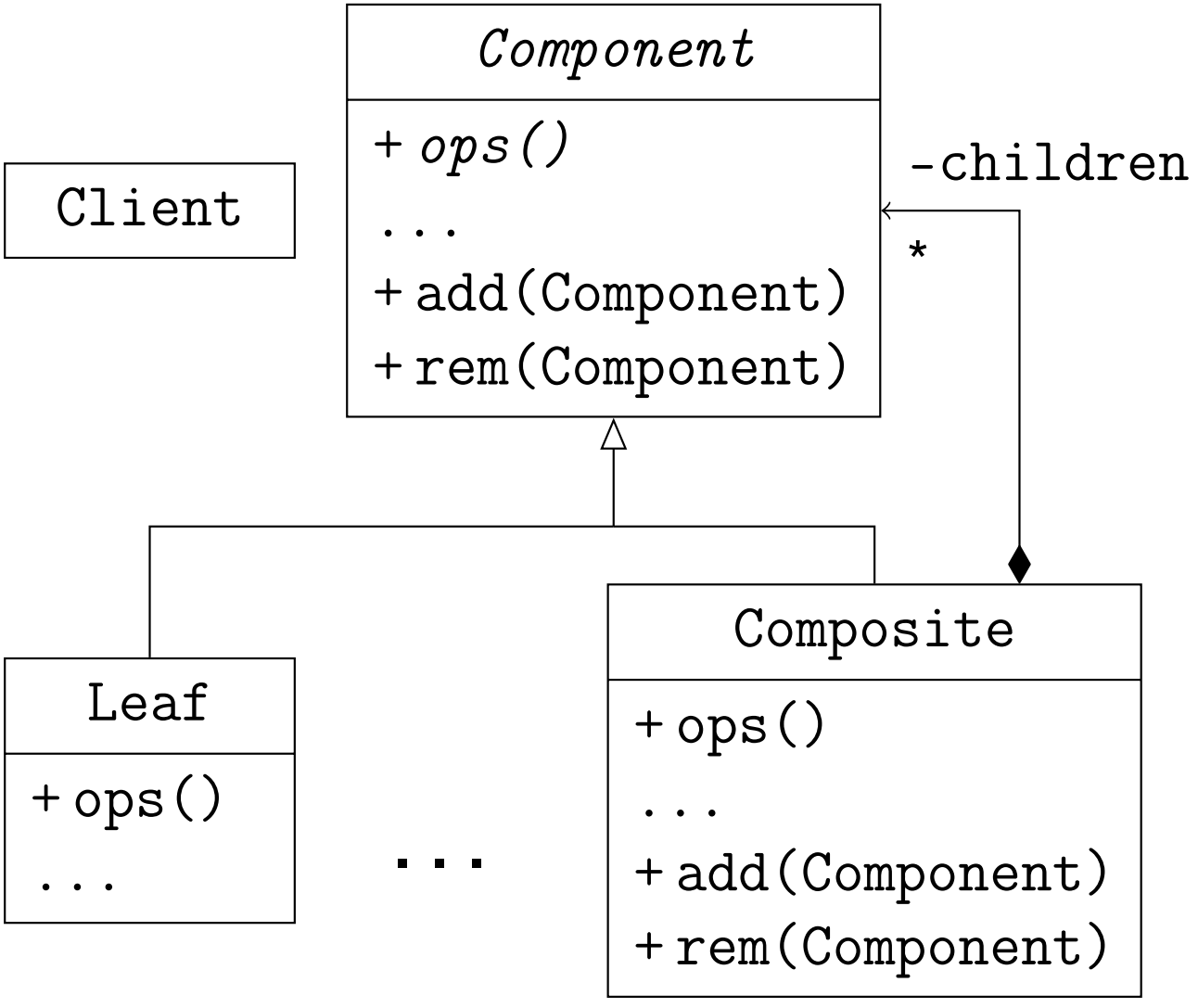
Participants/rôles

- Client : celui qui veut manipuler les objets
- Component : déclare l'interface des comportements communs
- Leaf : implémente les comportements élémentaires
- Composite : stocke les composants et implemente les opérations de gestion des composants

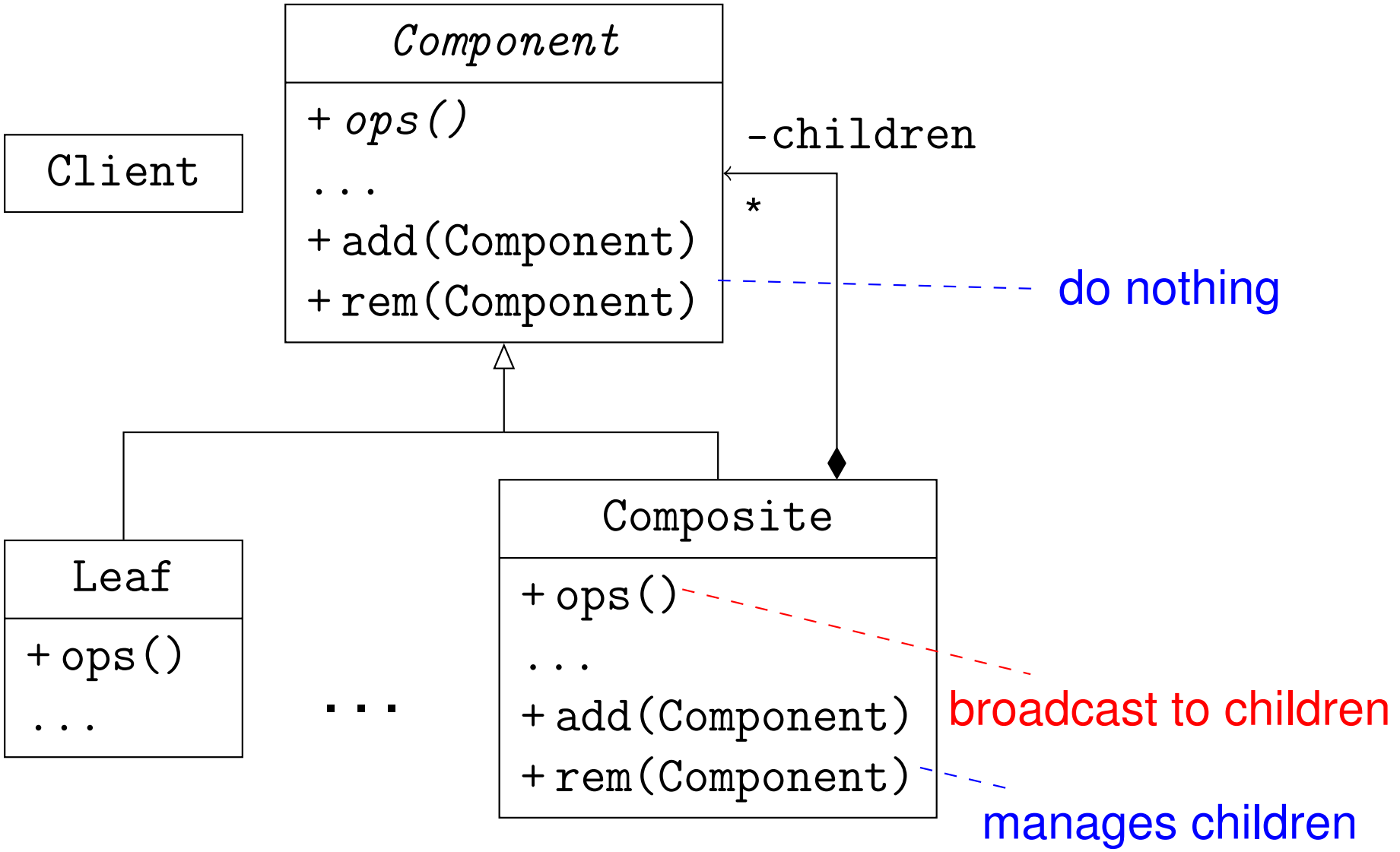
Conséquences

- simplifie la gestion
- extensible latéralement

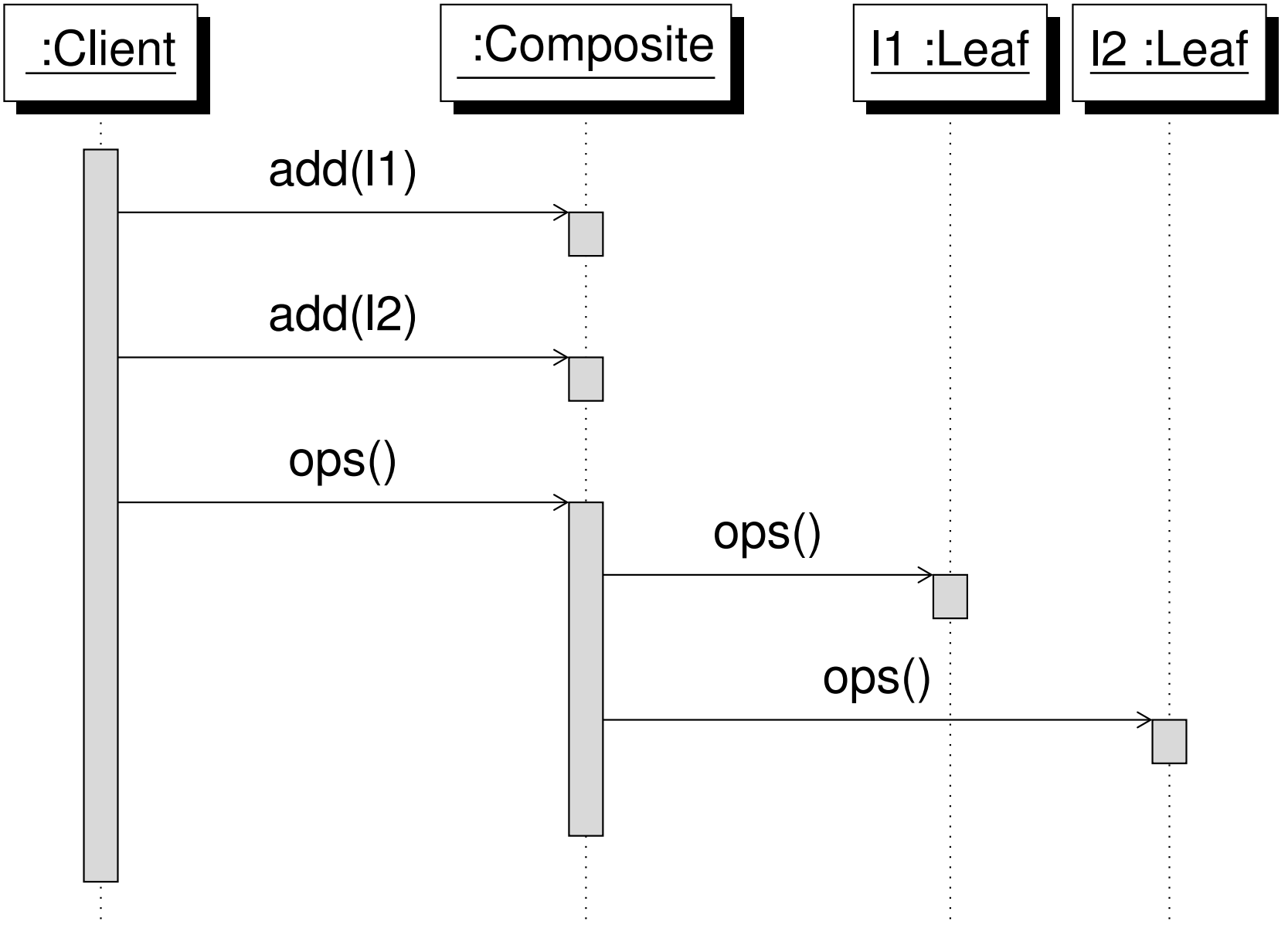
Structure


Component
+ ops()
...
+ add(Component)
+ rem(Component)
Client
-children
*
Leaf
+ ops()
...
...
Composite
+ ops()
...
+ add(Component)
+ rem(Component)

Structure


Component
+ ops()
...
+ add(Component)
+ rem(Component)
-children
*
Client
do nothing
Leaf
+ ops()
...
...
Composite
+ ops()
...
+ add(Component)
+ rem(Component)
broadcast to children
manages children

Collaboration



Façade (1/2)

Motivation

- regrouper des services dispersés dans plusieurs classes

Applicabilité/contraintes

- Pour créer une interface centralisée (voire unique en couplant avec Singleton)

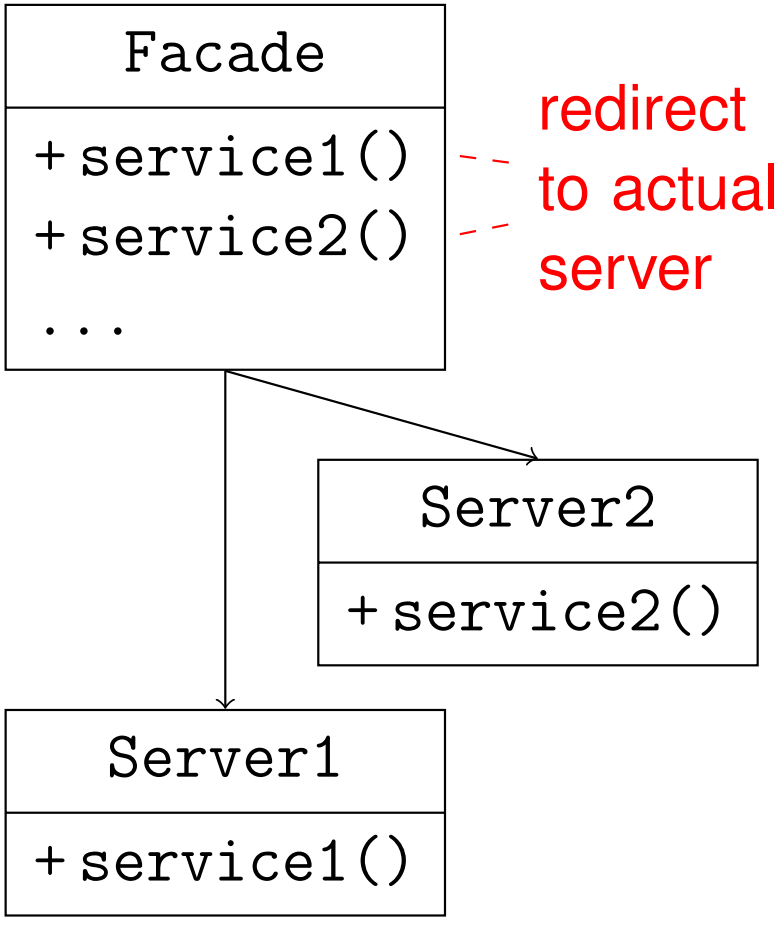
Participants/rôles

- Façade : l'interface
- Server : l'implantation du service

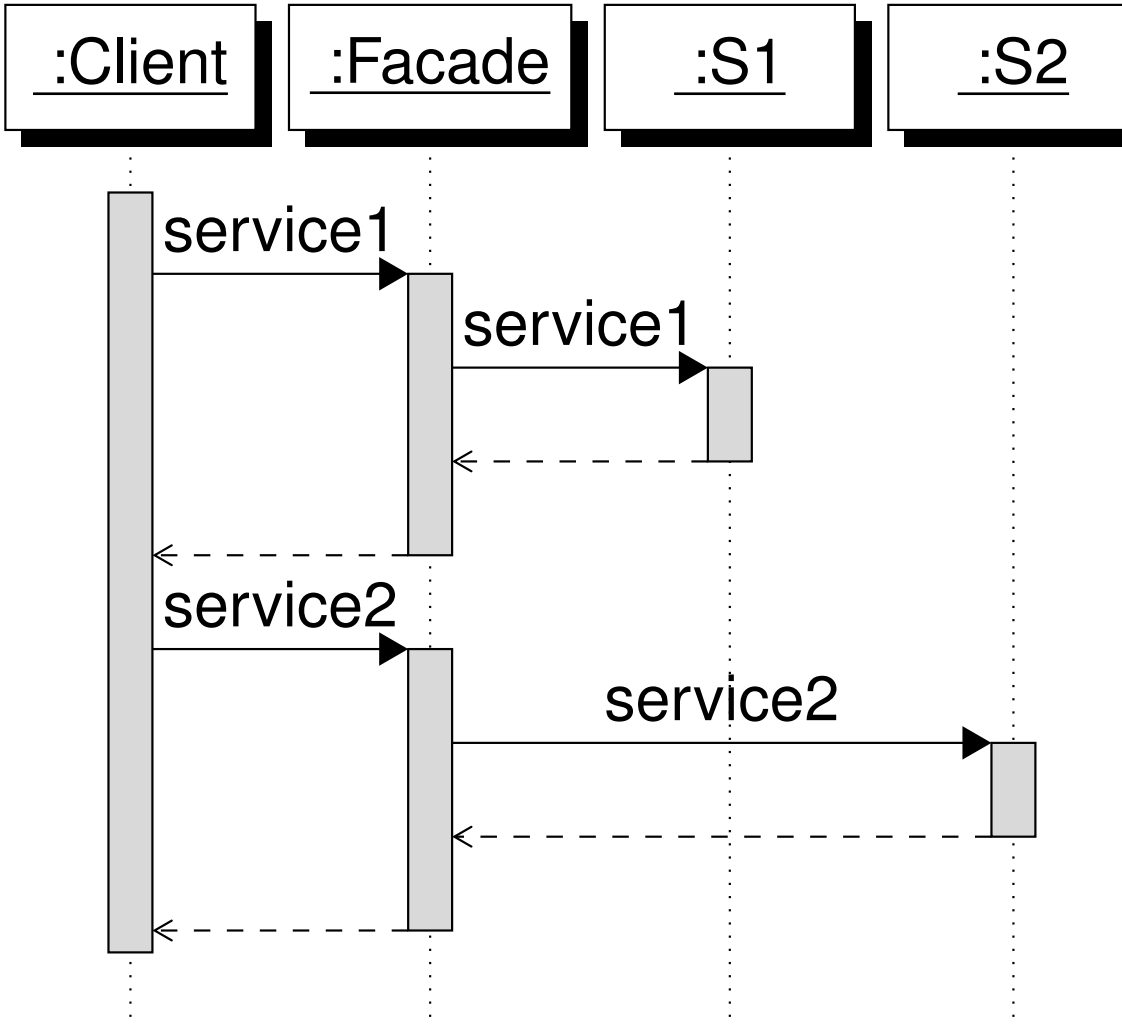
Conséquences

- point d'accès unique
- simplification de la présentation aux objets utilisateurs

Structure



Collaboration



Motivation

- Mécanisme de notification entre un Subject et plusieurs Obervers inconnus du Subject

Applicabilité/contraintes

- le Subject est indépendant des Observers

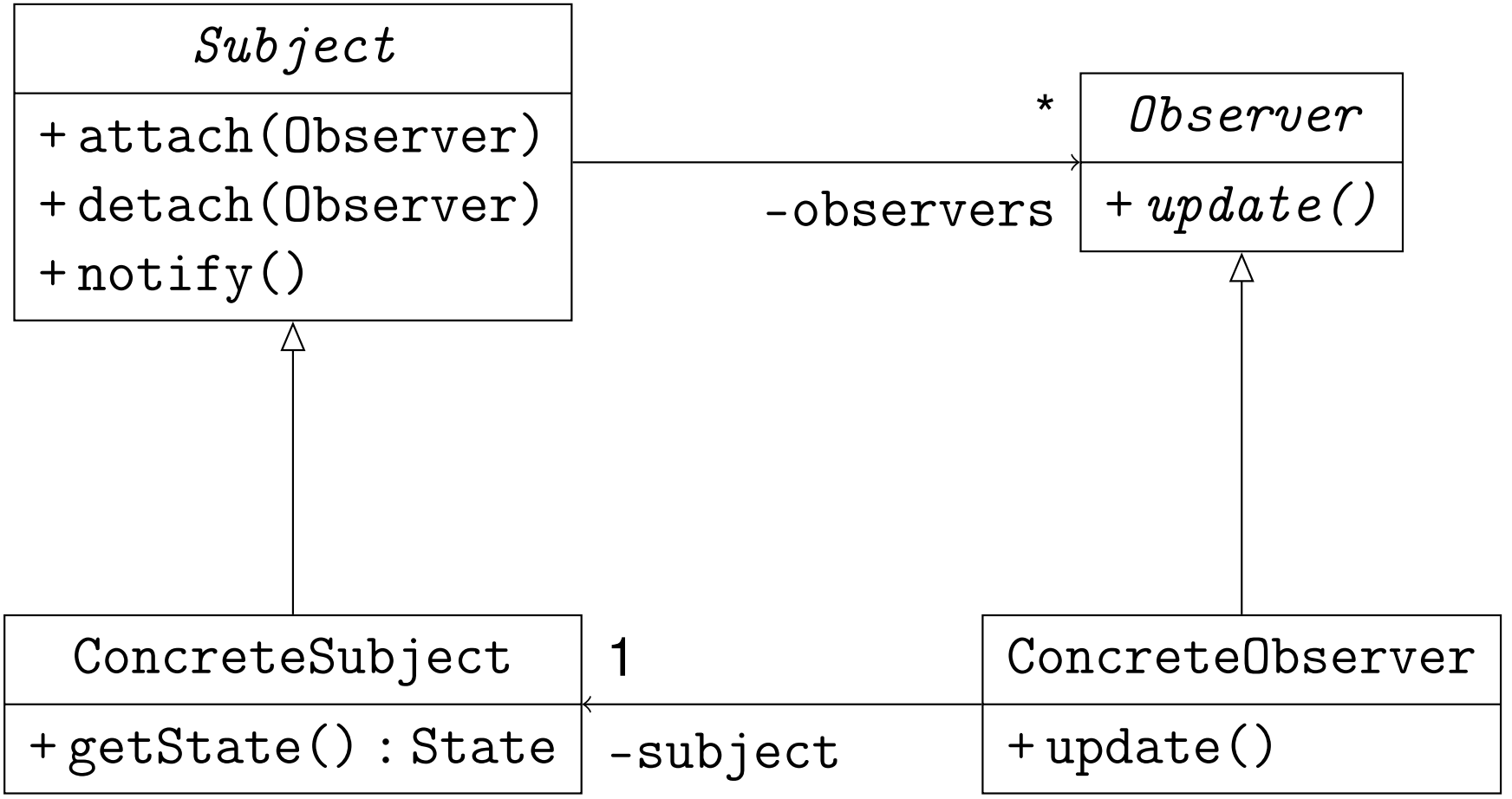
Participants/rôles

- Subject : gère une liste d'Observers et transmet les notifications
- ConcreteSubject : génère les notifications
- Observer : définit l'interface
- ConcreteObserver : s'enregistre auprès du Subject et réalise les mises à jour

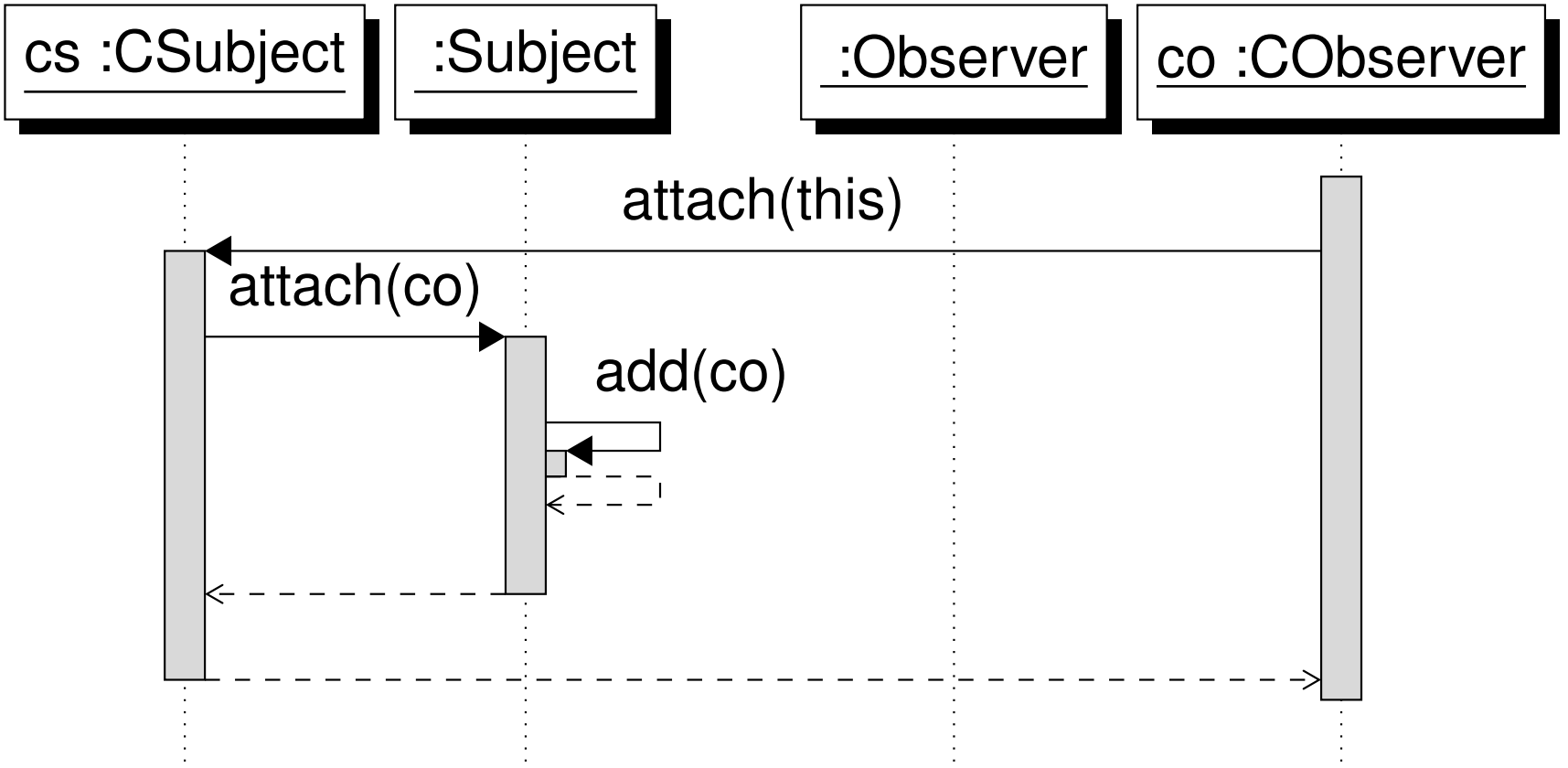
Conséquences

- belle séparation des responsabilités
- un ConcreteObserver ne peut observer qu'un Subject

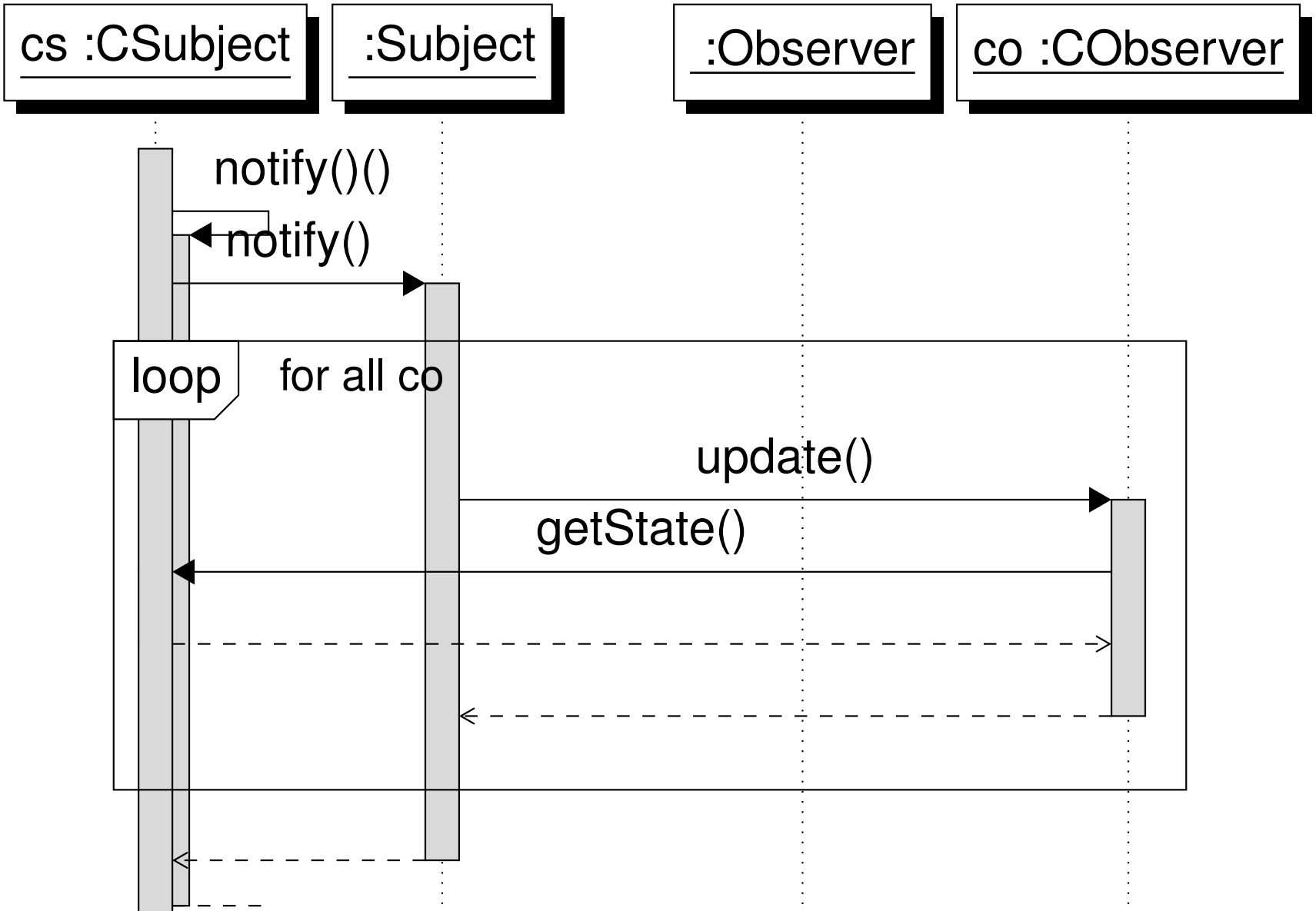
Structure


Subject
+attach(Observer)
+detach(Observer)
+notify()
*
Observer
+update()
-observers
ConcreteSubject
+getState() : State
1
-subject
ConcreteObserver
+update()

Collaboration (register)



Collaboration (update)



Motivation

- Définir une famille de comportements interchangeables. Le comportement change en fonction d'un contexte.

Applicabilité/contraintes

- quand on a besoin de manières différentes de faire des calculs.

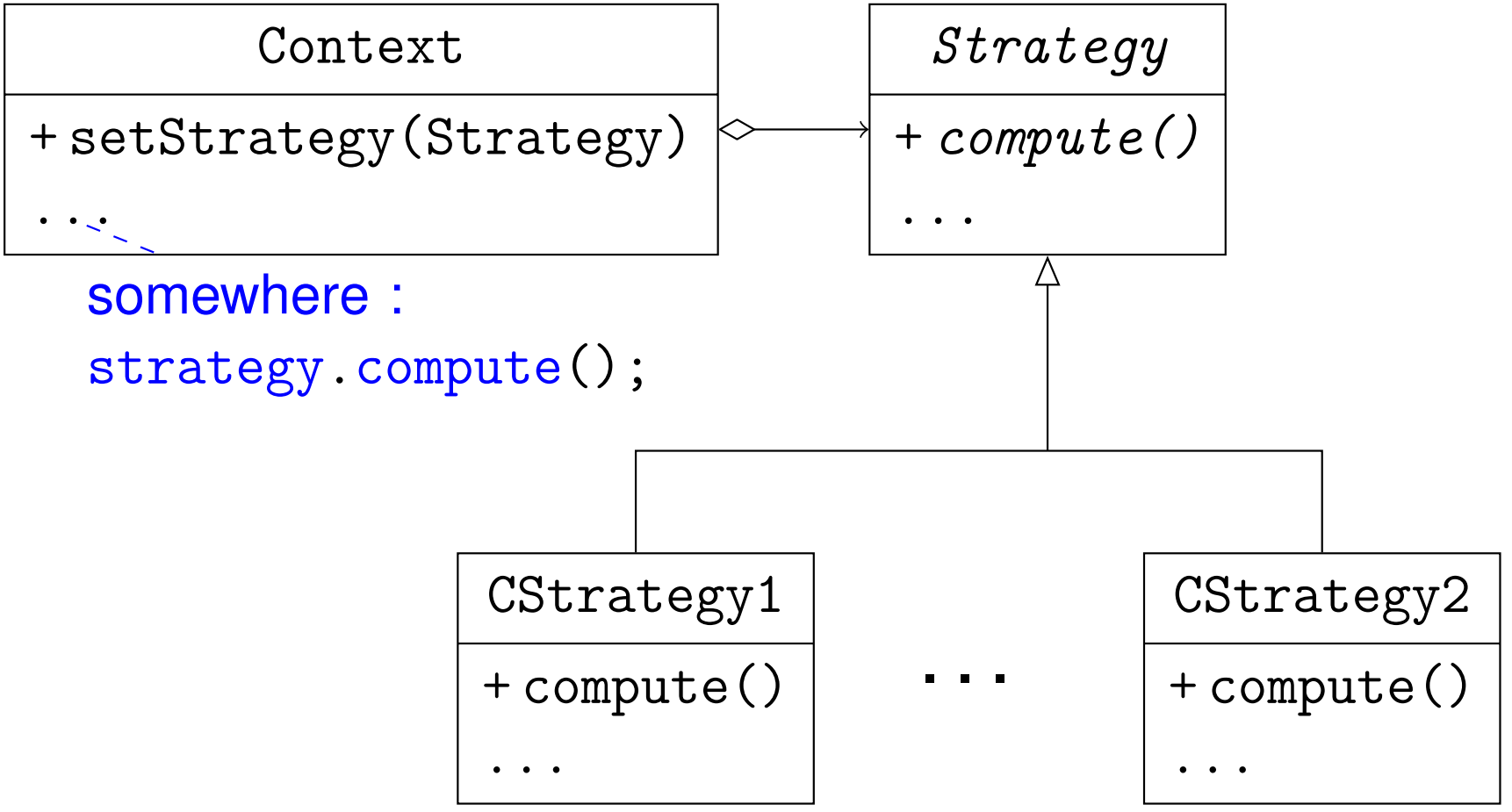
Participants/rôles

- Context : définit le contexte et utilise un calcul/comportement variable
- Strategy : l'abstraction qui fournit l'interface
- ConcreteStrategy : implémente le calcul/comportement

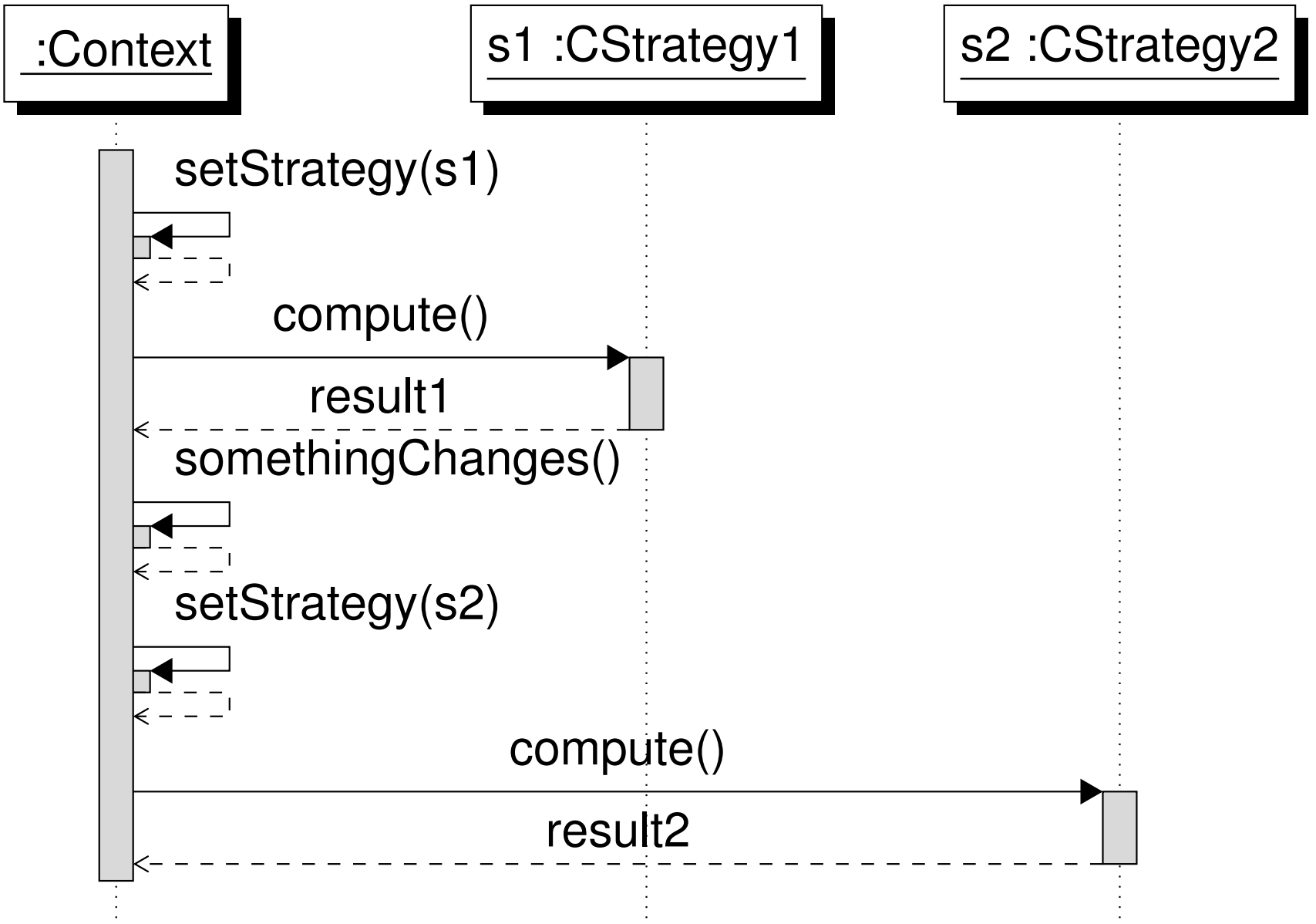
Conséquences

- extension horizontale
- algorithmes réutilisables

Structure


Context
+ setStrategy(Strategy)
...
Strategy
+ compute()
...
somewhere :
strategy.compute();
CStrategy1
+ compute()
...
...
CStrategy2
+ compute()
...

Collaboration



1 Des éléments de conception

2 Patron

3 Un brin de philosophie et d'histoire

4 Un bout de catalogue de patrons de conception

5 Un exemple

6 Conclusion

Une « spécification » :

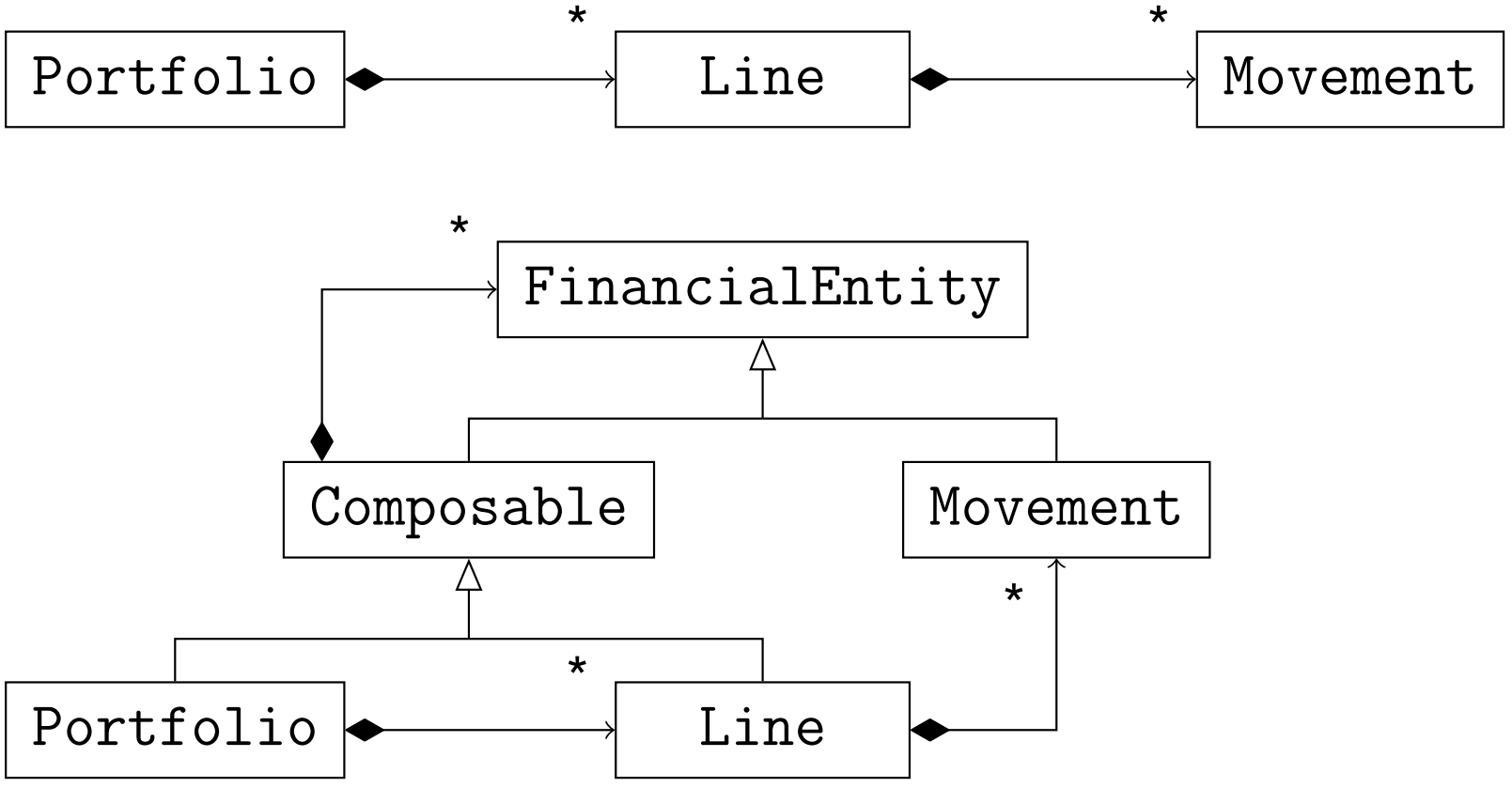
Un portefeuille contient des mouvements groupés par lignes.

Une « spécification » :

Un portefeuille contient des mouvements groupés par lignes.

Une « spécification » :

Un portefeuille contient des mouvements groupés par lignes.



Modèle	3 classes	5 classes
Avantages	Simple Objet réel direct	Abstraction Flexible Maintenabilité Code plus réparti (petit)
Inconvénients	Évolution Évolutivité Code plus concentré (gros)	Complexité apparente

1 Des éléments de conception

2 Patron

3 Un brin de philosophie et d'histoire

4 Un bout de catalogue de patrons de conception

5 Un exemple

6 Conclusion

Il y a de nombreux autres patrons intéressants :

- Factory Method, Builder,
- Adapter, Bridge, Decorator, Flyweight
- Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Template Method

Il y a des patrons plus spécialisés : pour la concurrence, pour la distribution, pour les télécoms, etc.

Il y a aussi des anti-patrons. De fausses bonnes idées. Voir : `http://wiki.c2.com/?AntiPatternsCatalog`. Exemple God class

- Les patrons offrent des solutions éprouvées et aident à décrire ou construire un système.
- Ils permettent de capitaliser des bonnes pratiques
- Ils permettent d'enrichir le vocabulaire
- D'un point de vue conception objet, ils permettent :
  - de faire du code plus maintenable, plus extensible
  - de bien séparer les responsabilités ; ils aident à choisir ce qui peut être réifié.
  - d'utiliser les redéfinitions et la liaison dynamique efficacement, même sans comprendre...
  - de montrer que la réification est puissante (voir State, Strategy, Command, Builder, etc.)
  - que l'héritage doit être utilisé avec parcimonie ; les relations sont plus flexibles et évolutives

- Réifier et réutiliser
- Séparer les responsabilités
- Hériter avec parcimonie

Motivation

- Créer *des* objets sans spécifier de classes concrètes

Applicabilité/contraintes

- séparer le service de création des classes concrètes

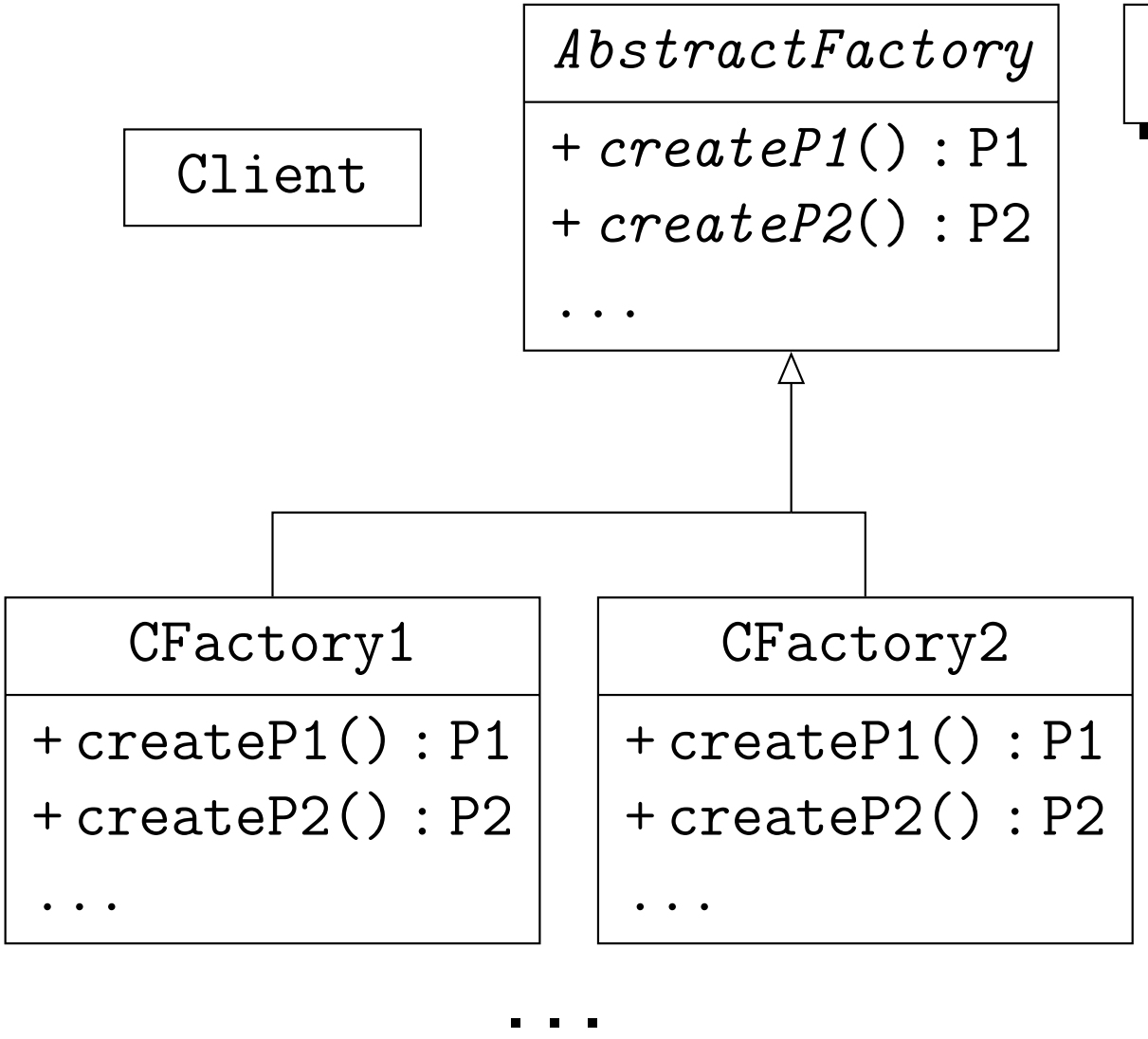
Participants/rôles

- Client : celui qui veut créer un objet
- AbstractFactory : interface du service de création
- ConcreteFactory : l'implantation du service de création

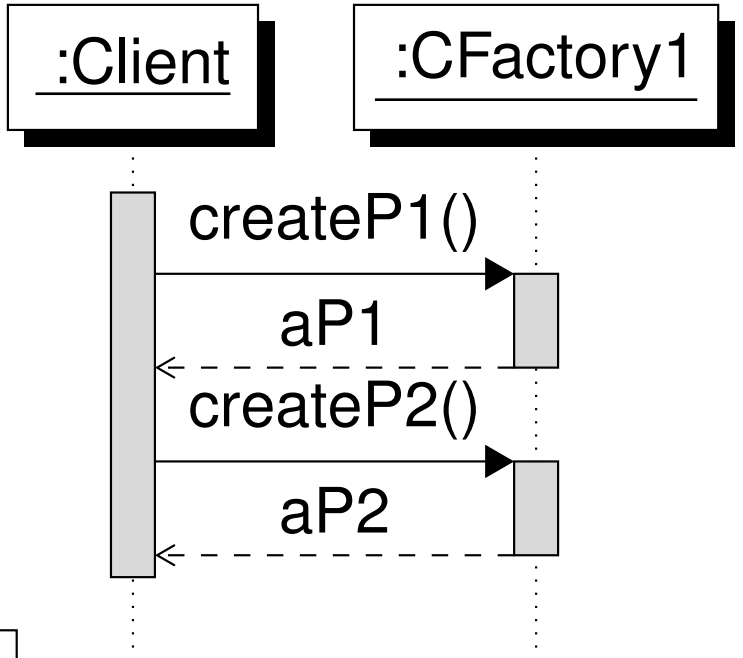
Conséquences

- sépare la fabrique du produit.
- l'interface de la fabrique abstraite définit les produits disponibles
- extensible

Structure



Collaboration



- En fait la création est déportée dans une autre classe et repose sur une autre création d'instances
- La création peut être un processus plus complexe qu'une simple instanciation
- La solution de la fabrique peut être généralisée à plusieurs hiérarchies

Motivation

- Fournir un substitut à un objet

Applicabilité/contraintes

- pour gérer une référence complexe (pour un objet distant, pour un objet chiffré, protégé, stocké...

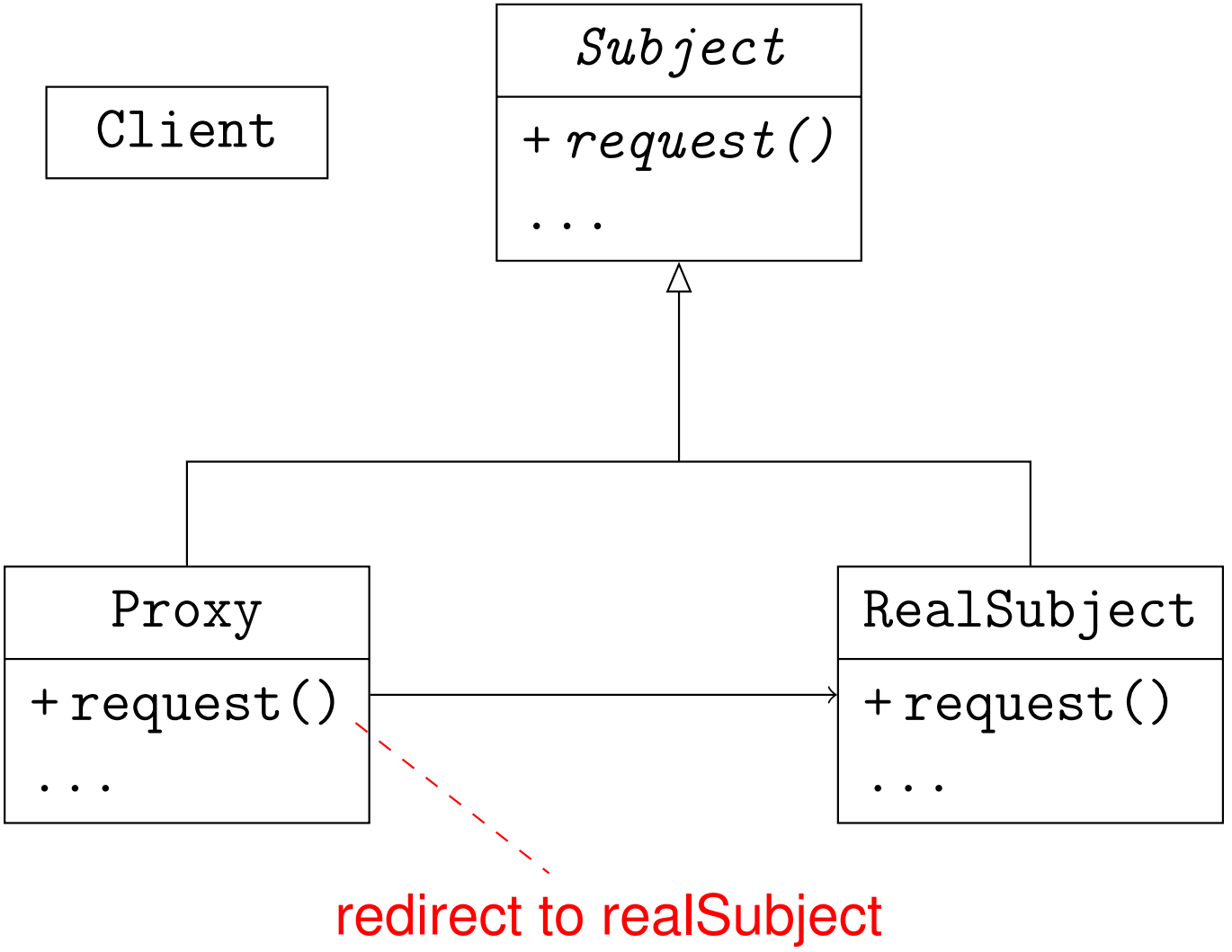
Participants/rôles

- Client : celui qui veut accéder à un object
- Proxy : l'abstraction qui fournit l'interface
- Subject : le substitut, qui calcule l'accès à l'objet réel
- RealSubject : l'objet cible

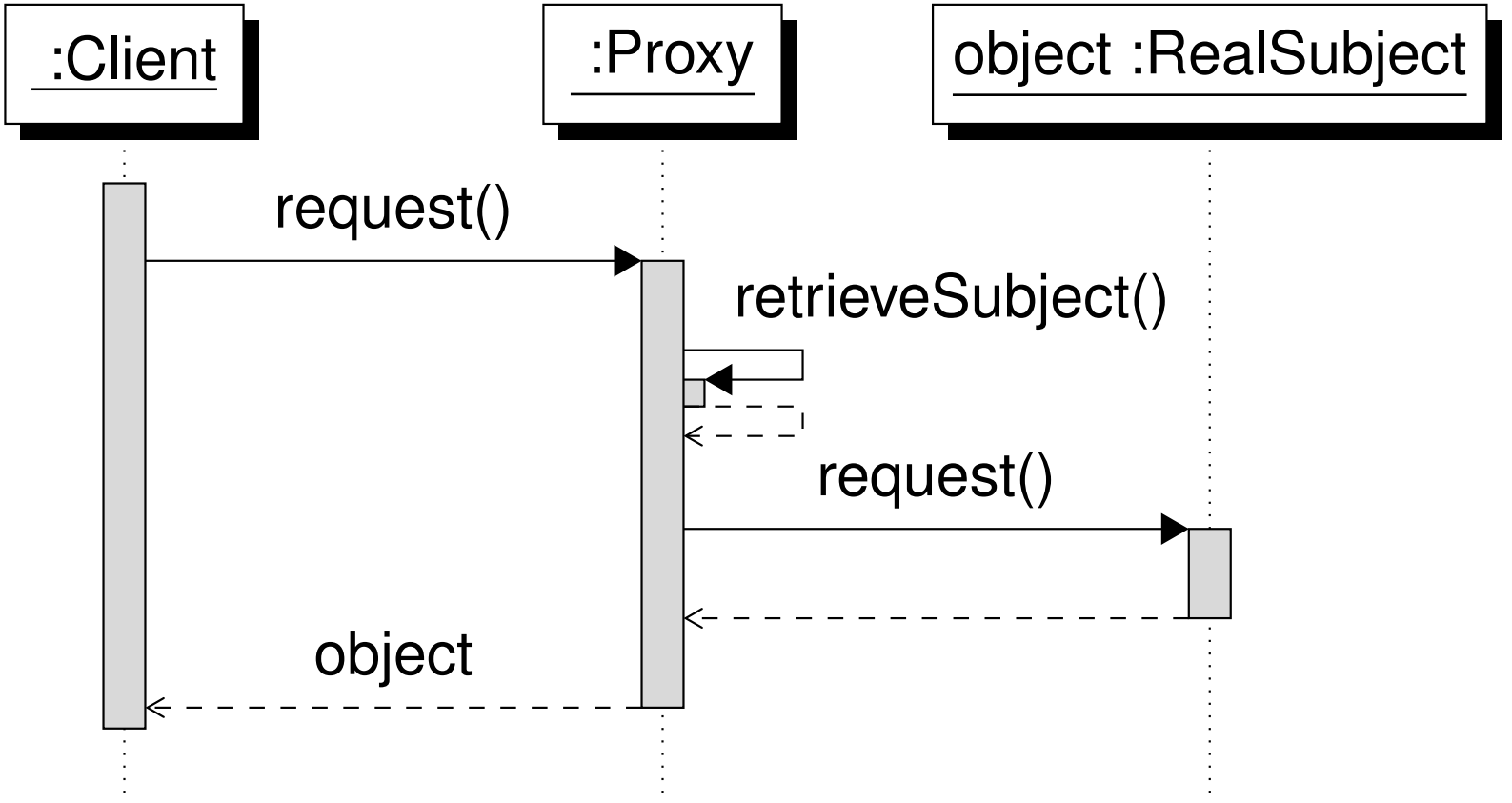
Conséquences

- introduit une indirection vers un objet

Structure


Client
Subject
+ request()
...
Proxy
+ request()
...
RealSubject
+ request()
...
redirect to realSubject

Collaboration



Des variantes : le proxy est conservé... ou, la référence au proxy est remplacée par l'objet retrouvé.

State (1/3)

Motivation

- Définir une famille de comportements interchangeables. Le comportement change en fonction d'un état d'un objet.

Applicabilité/contraintes

- quand on a besoin de différentes manières de faire des calculs selon l'état d'un objet...

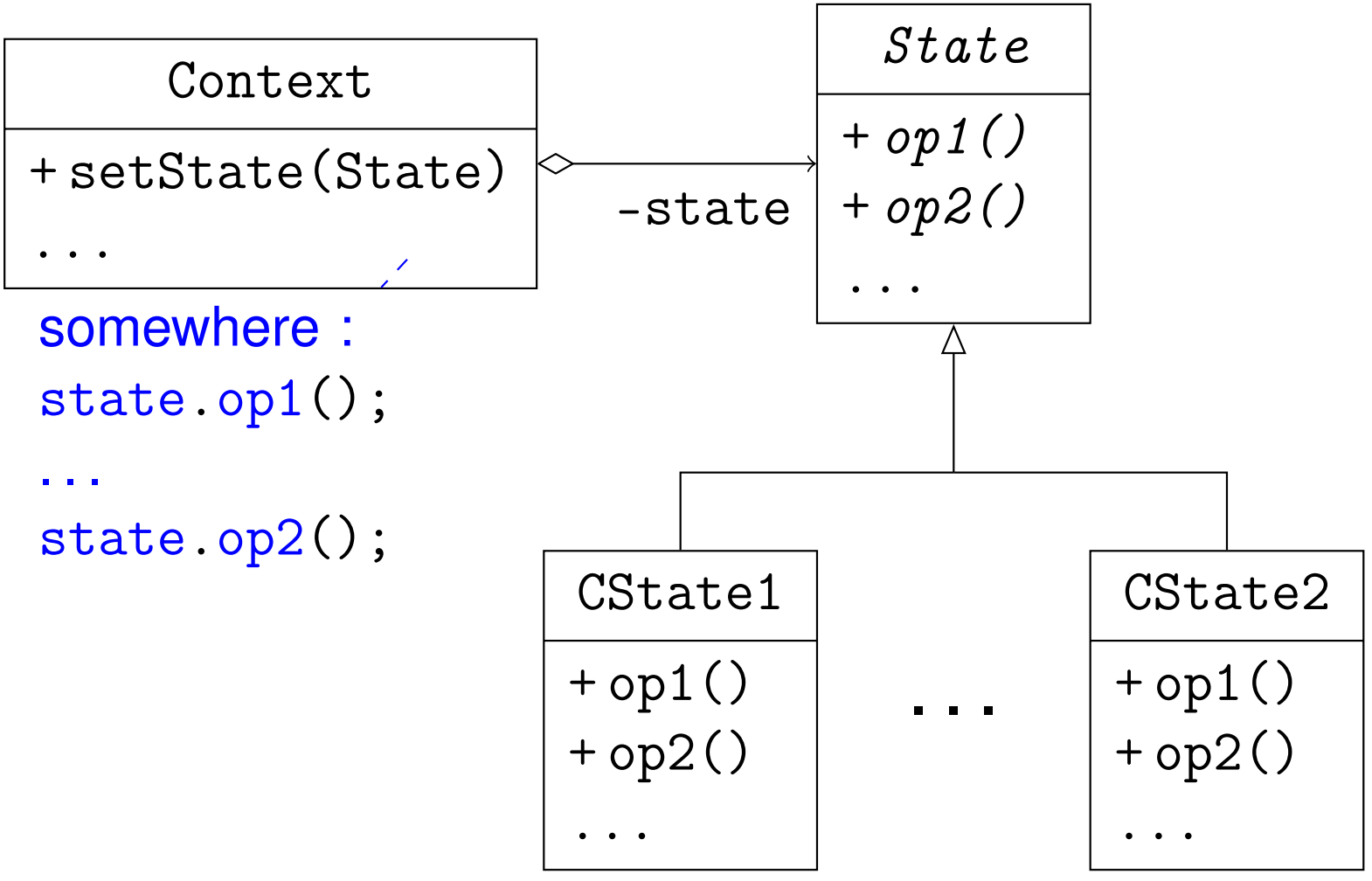
Participants/rôles

- Context : définit le contexte et utilise un calcul/comportement variable
- State : l'abstraction qui fournit l'interface
- ConcreteState : implémente les calculs/comportements

Conséquences

- extension horizontale

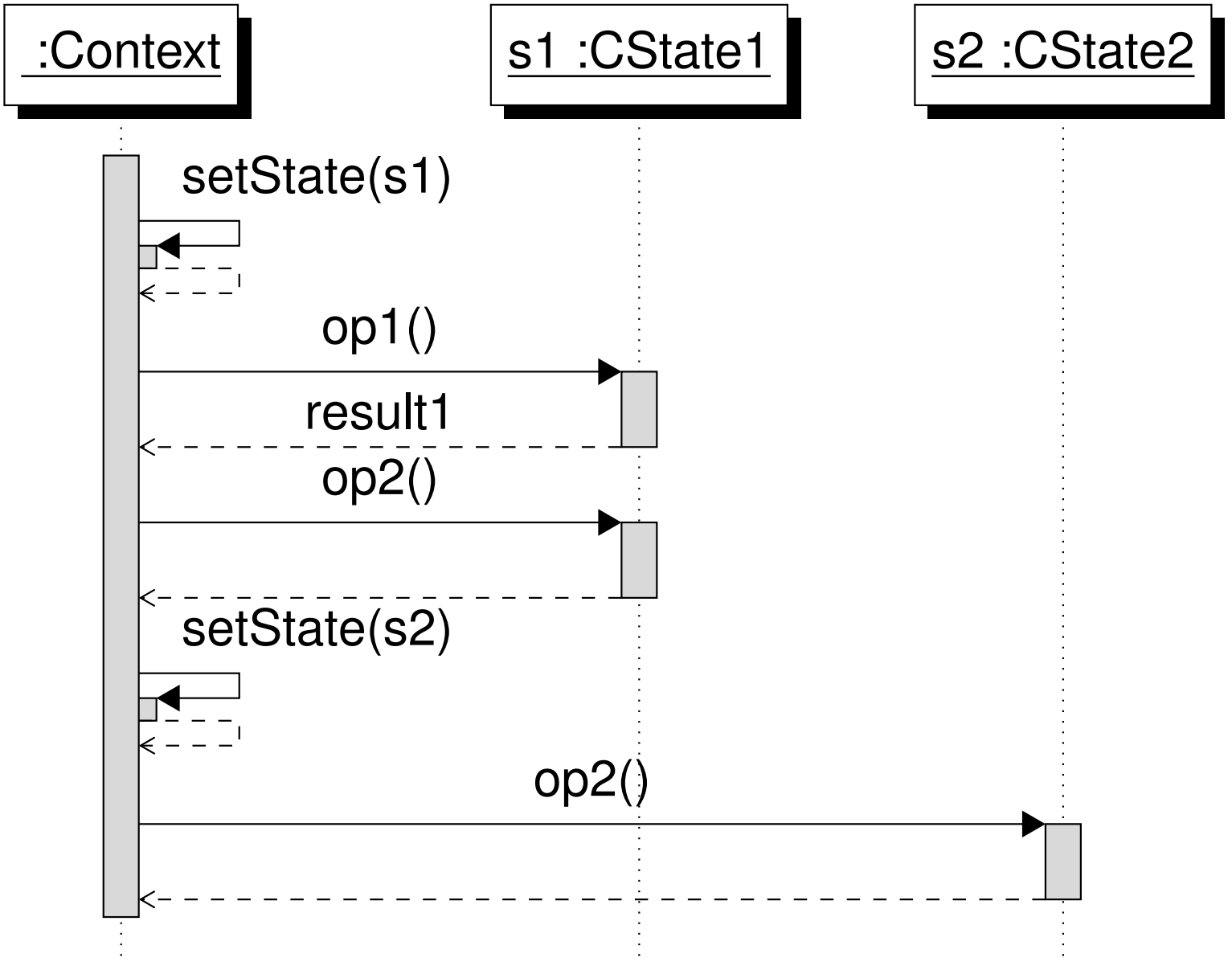
Structure



somewhere :

```
state.op1();
...
state.op2();
```

Collaboration



Motivation

- Ajouter des comportements à un objet sans le modifier

Applicabilité/contraintes

- s'applique à une structure d'objets
- la structure d'objet est (plutôt) stable

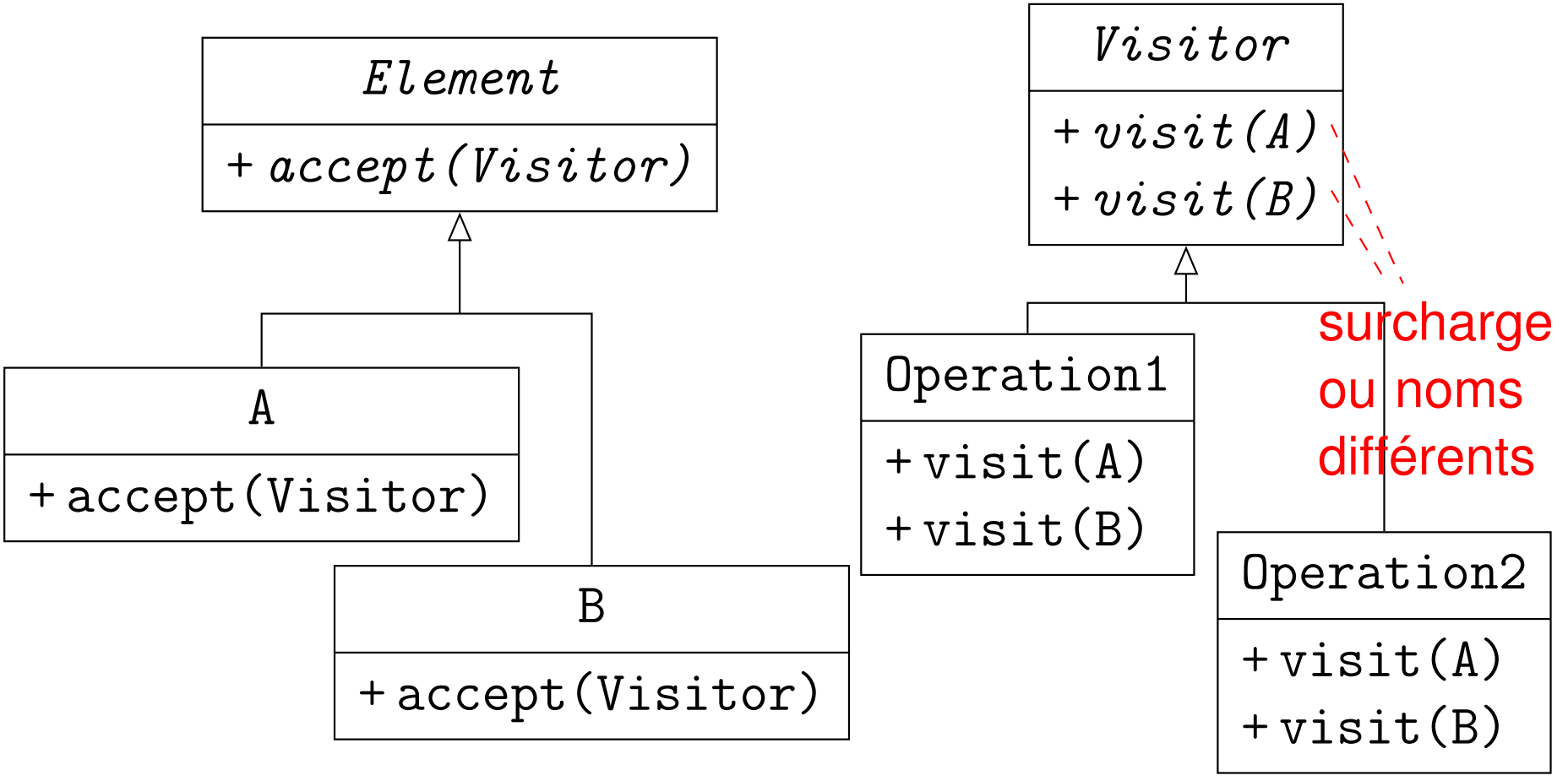
Participants/rôles

- Visitor : déclare les opérations de visite
- Operation : implémente les opérations de visite
- Element : d'une structure, déclare une opération d'acception de visite
- ConcreteElement : implante l'opération de visite

Conséquences

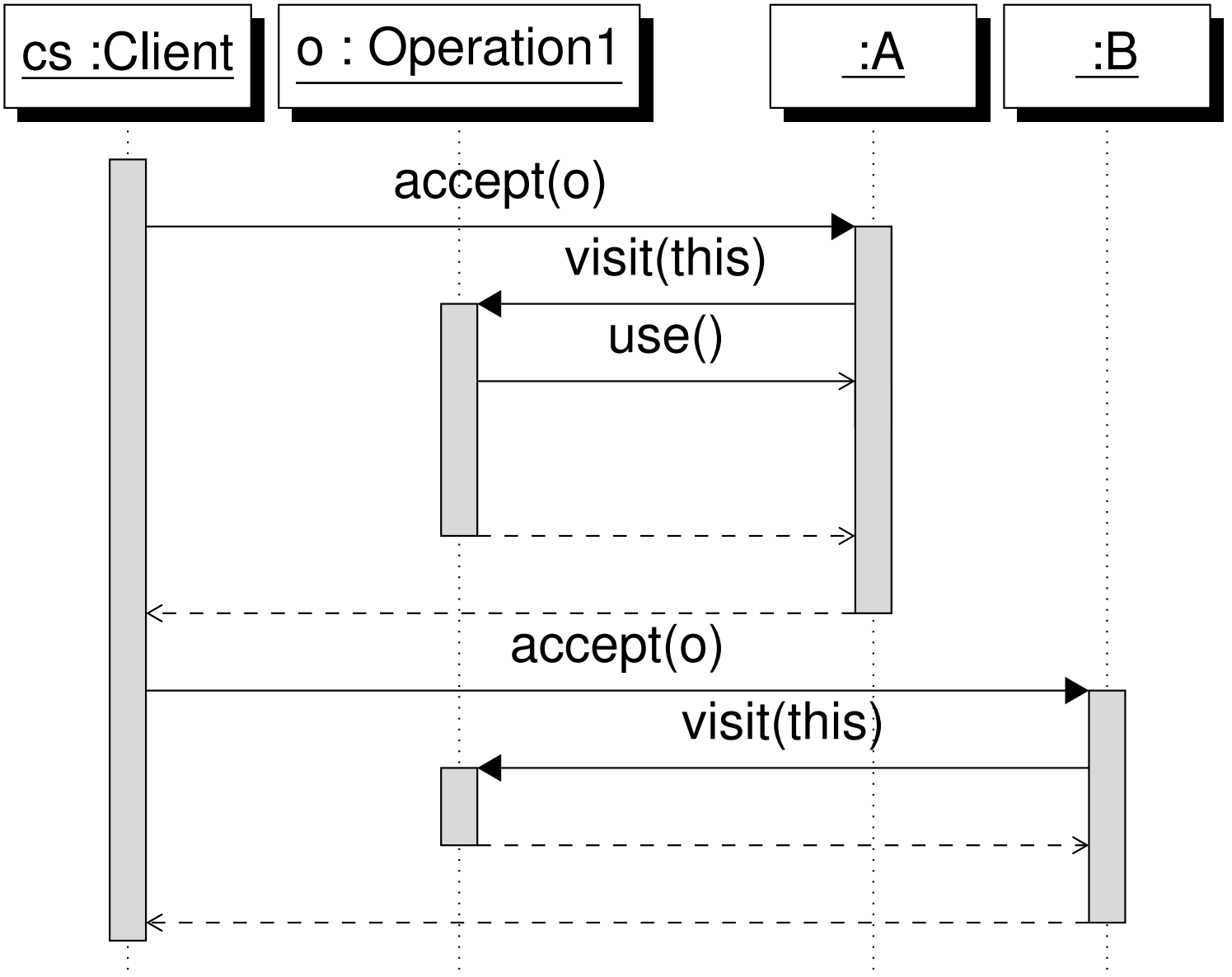
- rompt l'encapsulation ; un visiteur peut vouloir accéder à une partie privée des éléments
- ajout d'opération simple, mais coût élevé d'ajout d'élément.

Structure

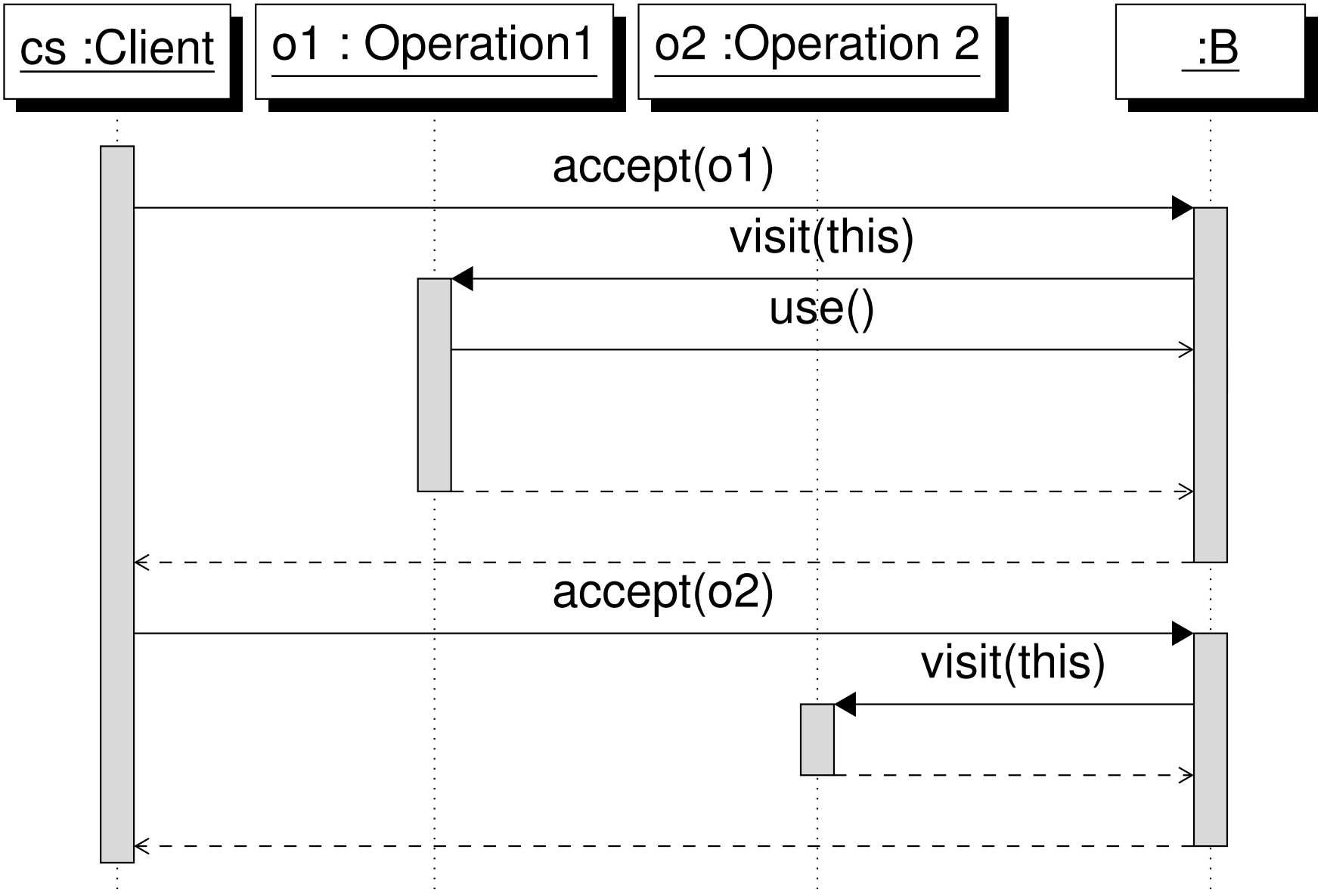


Ajouter un élément `C` implique d'ajouter un `visit(C)` dans toute la hiérarchie des visiteurs.

Collaboration (extend with Operation1, use on 2 objects)



Collaboration (2 different Operation on the same object)



- S'appuie sur le Composite qui sert de source d'éléments.
- On peut avoir un accept par défaut qui ne fait rien pour chacun des éléments.
- On peut utiliser ce patron récursivement. Un `visit` peut appeler une autre opération avec `accept`.
- La pratique qui, si A appelle B puis B appelle A sachant B, s'appelle le *double dispatching* ; elle permet une liaison dynamique selon 2 types (B puis A/B). Utile pour des opérations binaires (avec 2 objets). Le visiteur l'applique systématiquement.