Design Patterns structurels

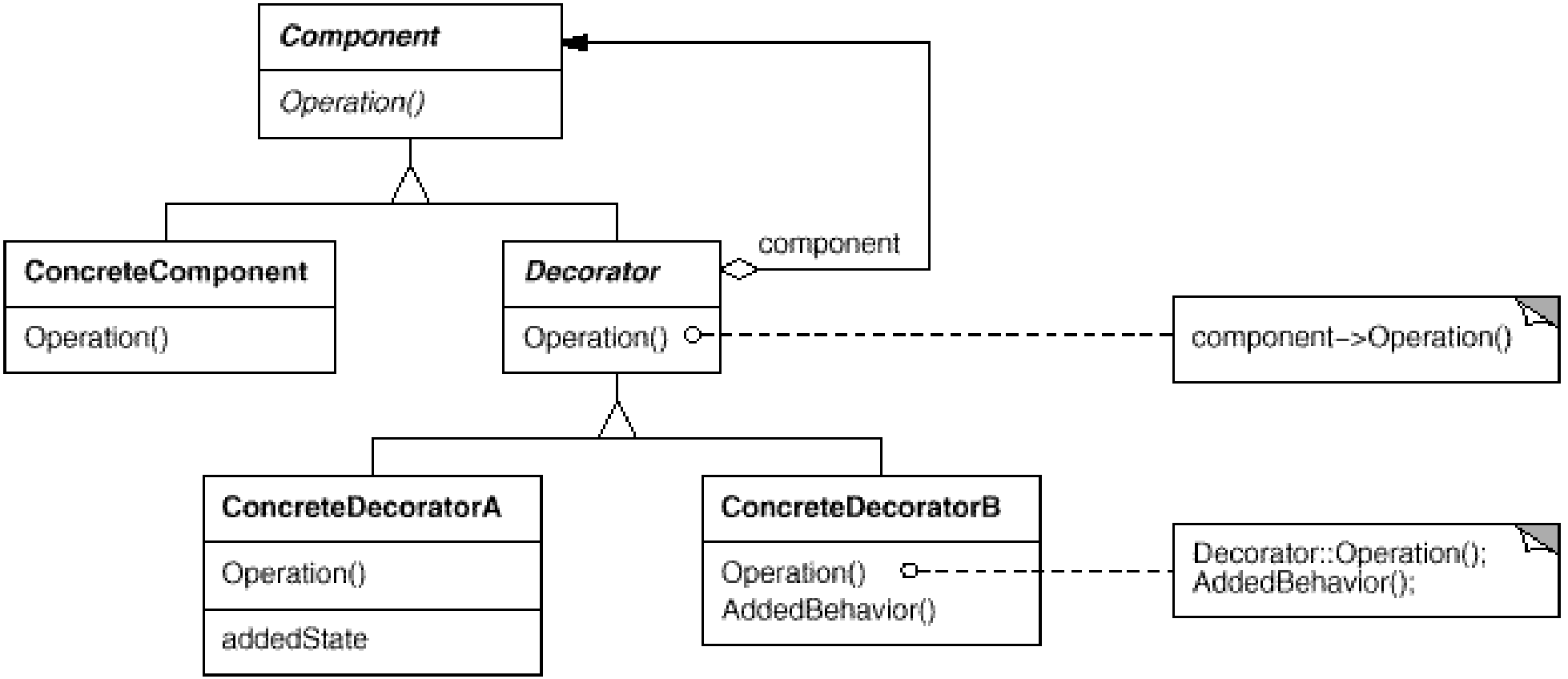
- Design Patterns liés au problème d'organisation des objets dans un logiciel
  - Composition des classes et des objets
  - Permet de concevoir des structures plus larges, et extensibles. Un peu comme des préfabriqués avec des interfaces normalisées
  - Renforce aussi la localité (donc le S de SOLID !).
    - Ex sécurisation et proxy

- Quelques exemples :
  - Decorator interne et externe : ajouter une caractéristique à un objet
  - Proxy : l'objet est loin, le calcul est lourd, l'accès est réservé
  - Composite : traiter de manière identique un groupe et un élément
  - Flyweight : (vous l'apprendrez par vos lectures)

Decorator 's

- Etendre les fonctionalités d'une classe sans la modifier,
  - Sans la compliquer
  - Sans lui ajouter une deuxième responsabilité !!
- Etendre avec souplesse → pas toutes les mêmes instances de la même manière
- Rester tranparent pour le client !!
- Peut être récursif
  - Donc aide à garder des décorateurs avec notre 'S' !

Decorator externe (GoF)



Decorator interne

- Quand la décoration peut être “prévue”
- On délègue la fonctionnalité
  - En ne connaissant que l'interface
  - Exemple: bordure

- Différences avec decorator externe
  - Pour garder les this
    - Ça veut dire quoi ?
  - Pour garder le type des objets
    - Ça veut dire quoi ?

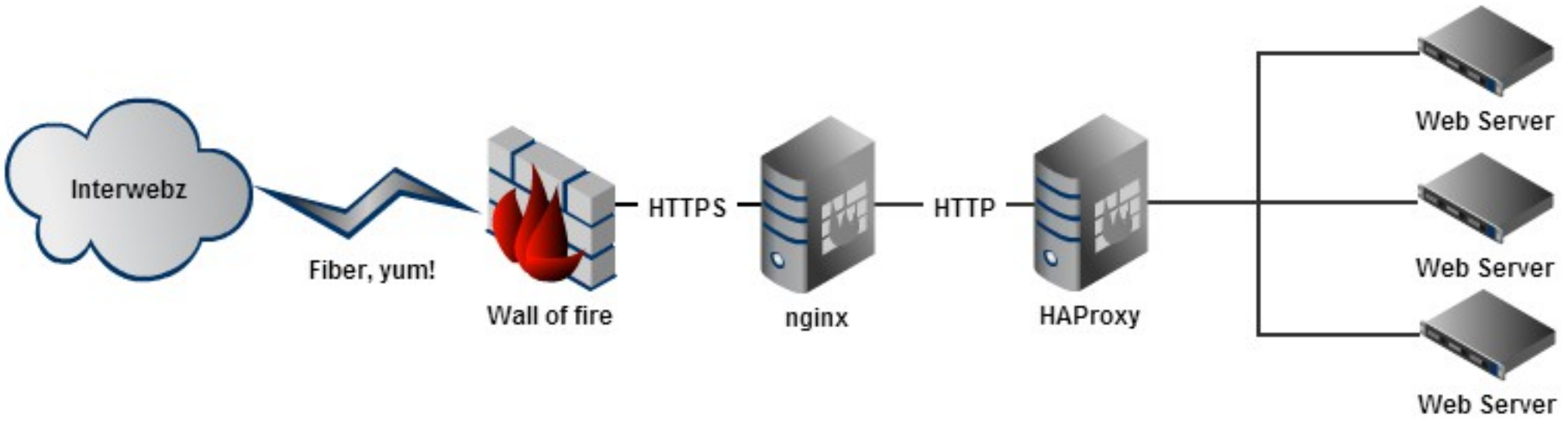
Proxy

- Un intermédiaire
- Pour ne pas compliquer un objet
- Rester tranparent pour le client !!

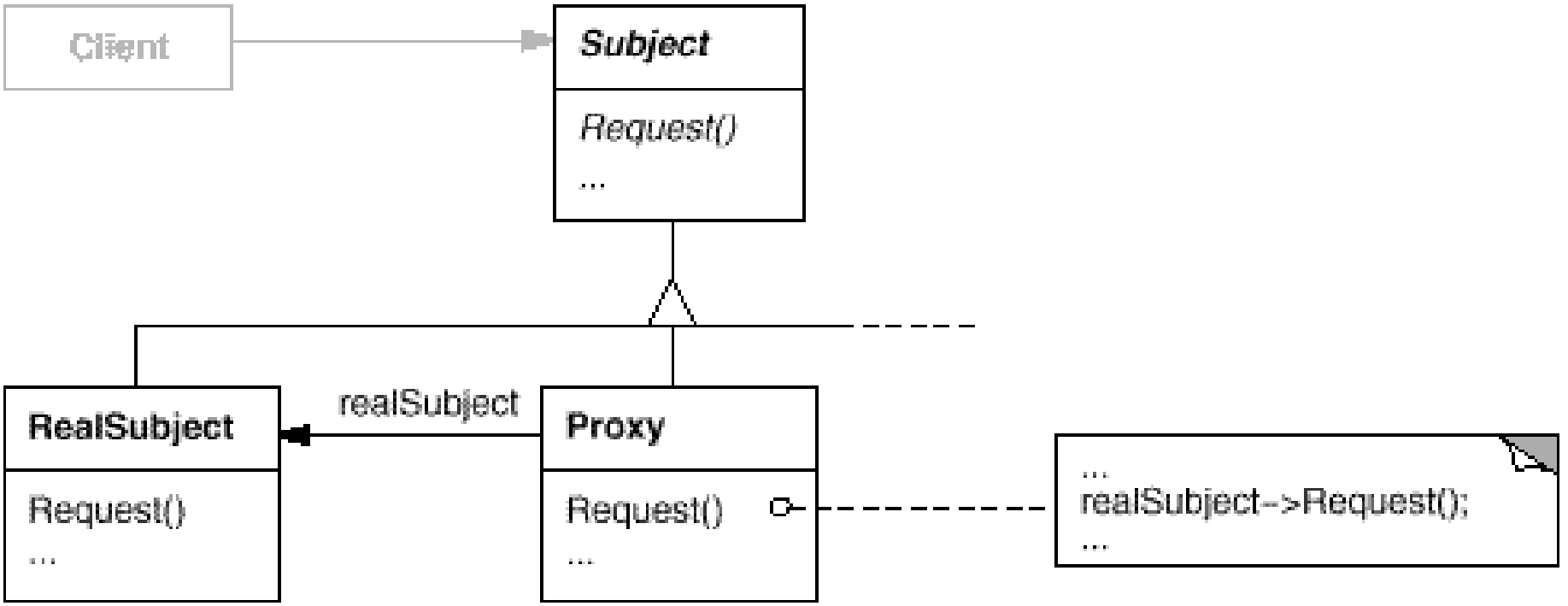
- Cas d'utilisation :
  - Vérifications autorisations,
  - gestion cache,
  - Remote,
  - ...

Proxy : du classique !

- Vous avez codé un serveur http
  - Il faut ajouter https
  - Un load balancer
  - Une détection d'intrusion

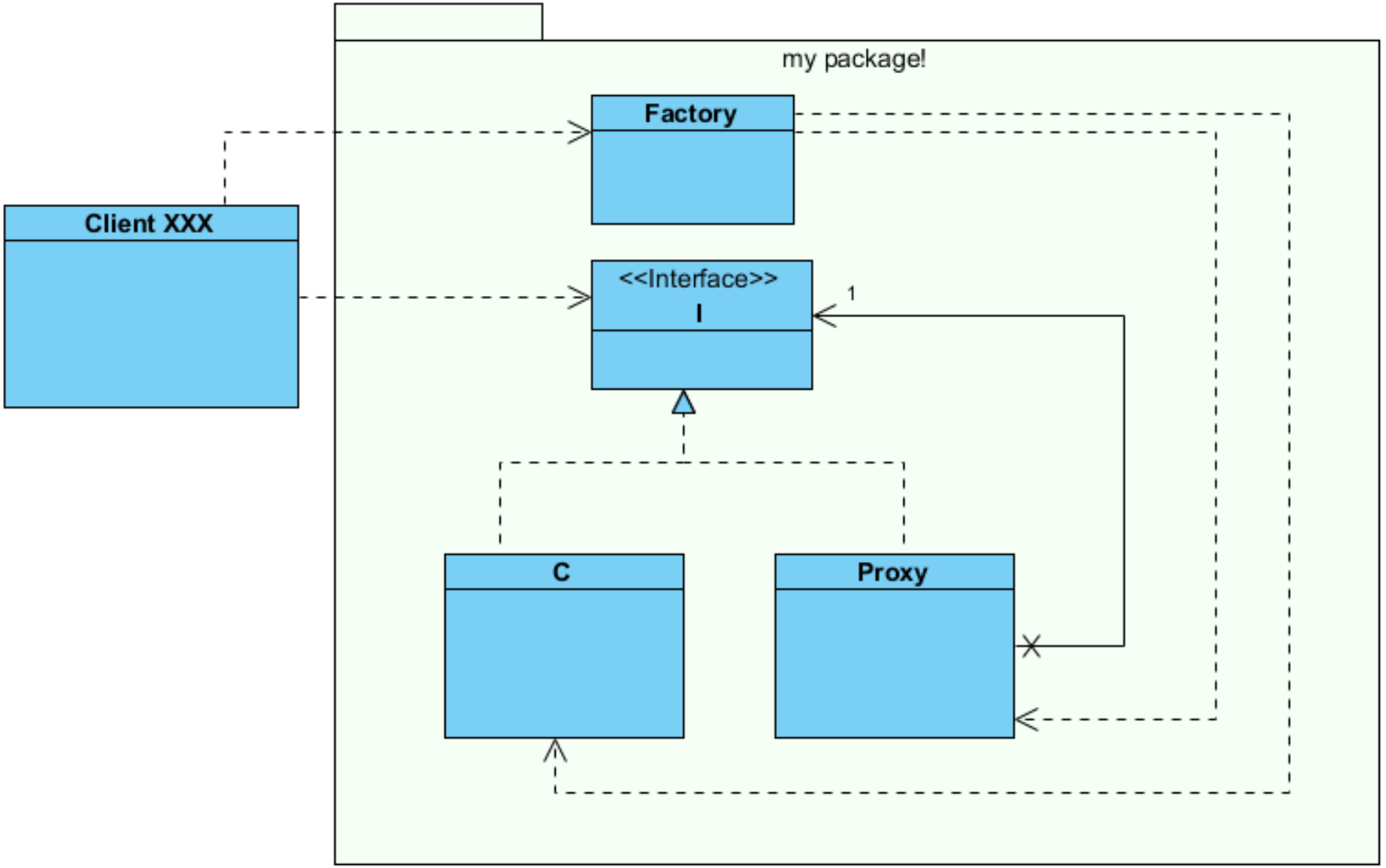


Proxy (GoF)


Client
Subject
Request()
...
RealSubject
Request()
...
realSubject
Proxy
Request()
...
...
realSubject->Request();
...

Proxy (réel !)

- Ca vous rappelle quelque chose !?

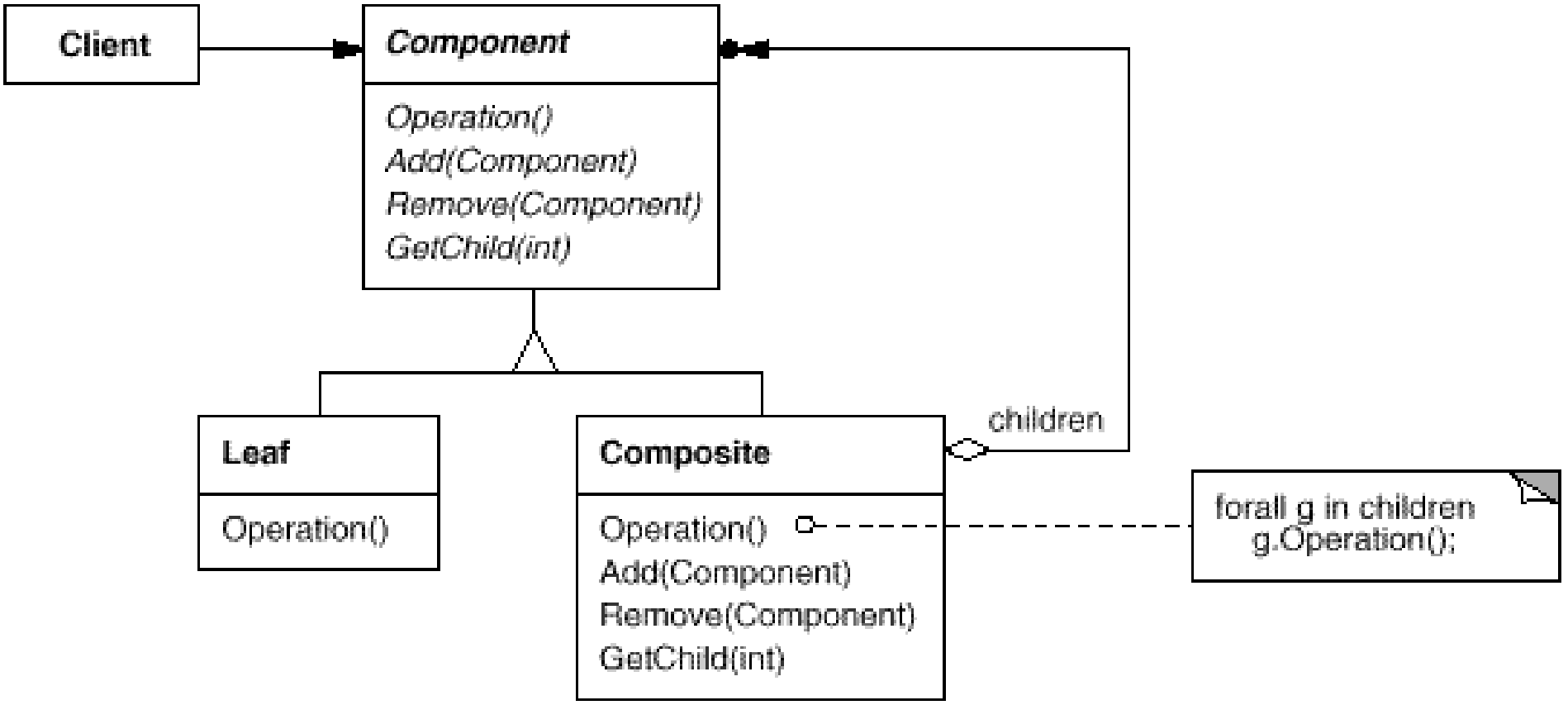


Composite

- On veut pouvoir traiter de manière indifférencier un ensemble et un élément
  - Exemple : un dessin qui contient des formes géométriques que l'on peut déplacer
  - Exemples : les composants graphiques du jdk
    - Les menus auxquels on peut ajouter des entrées ou des sous-menus
    - ...

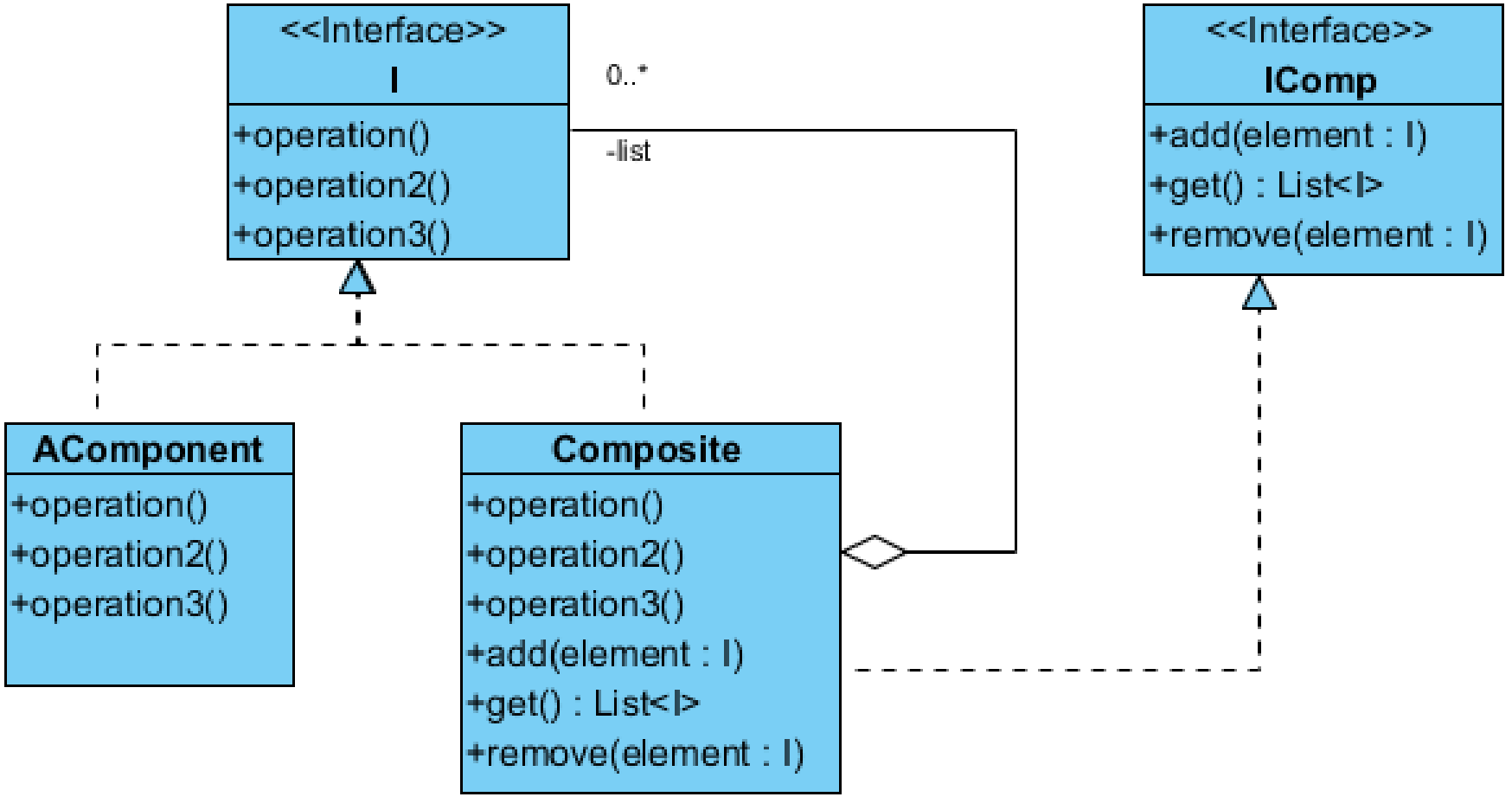
Composite transparent

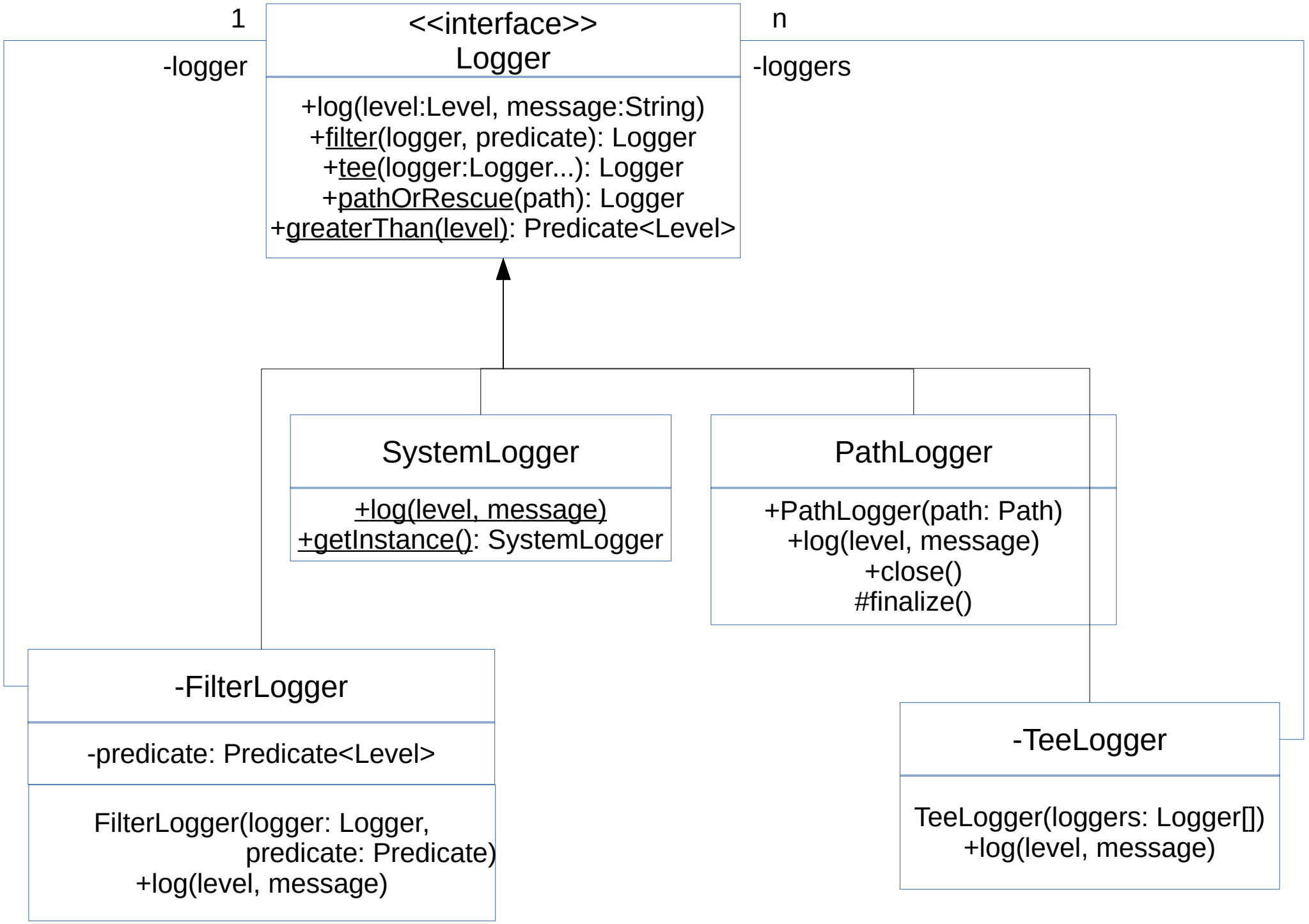
- Donc un composant et un composite (le contenant) doivent avoir la même interface



Composite sûr

- L'interface commune est vraiment la partie commune !




1
-logger
<<interface>>
Logger
+log(level:Level, message:String)
+filter(logger, predicate): Logger
+tee(logger:Logger...): Logger
+pathOrRescue(path): Logger
+greaterThan(level): Predicate<Level>
n
-loggers
SystemLogger
+log(level, message)
+getInstance(): SystemLogger
PathLogger
+PathLogger(path: Path)
+log(level, message)
+close()
#finalize()
-FilterLogger
-predicate: Predicate<Level>
FilterLogger(logger: Logger,
predicate: Predicate)
+log(level, message)
-TeeLogger
TeeLogger(loggers: Logger[])
+log(level, message)

TD !

Synthèse

Decorator interne / externe

- Attention aux question de terminologie
  - Decorator GoF = Decorator externe
  - Decorator interne = même intention mais implémentation par simple strategy
- Un schéma UML ne suffit pas

Proxy / Decorator externe

- 2 patterns du GoF
- Quasi même schéma UML
- Intention différente ?
  - Proxy : intermédiaire technique. En général transparent, caché par une factory
  - Decorator externe : apport fonctionnel. Souvent créé explicitement

- Frontière floue donc confusion régulière !!

Hiérarchie de classes ?

- Ne pas faire systématiquement de hiérarchie
- En particulier, ne pas faire une classe dérivée juste pour positionner une propriété différente
  - Une méthod factory suffit
  - Ex: newDestroyer dans la classe abstraite Carrier