GOF

Creazionali: Abstract Factory e Singleton

Risolvono problematiche inerenti l'istanziazione degli oggetti:

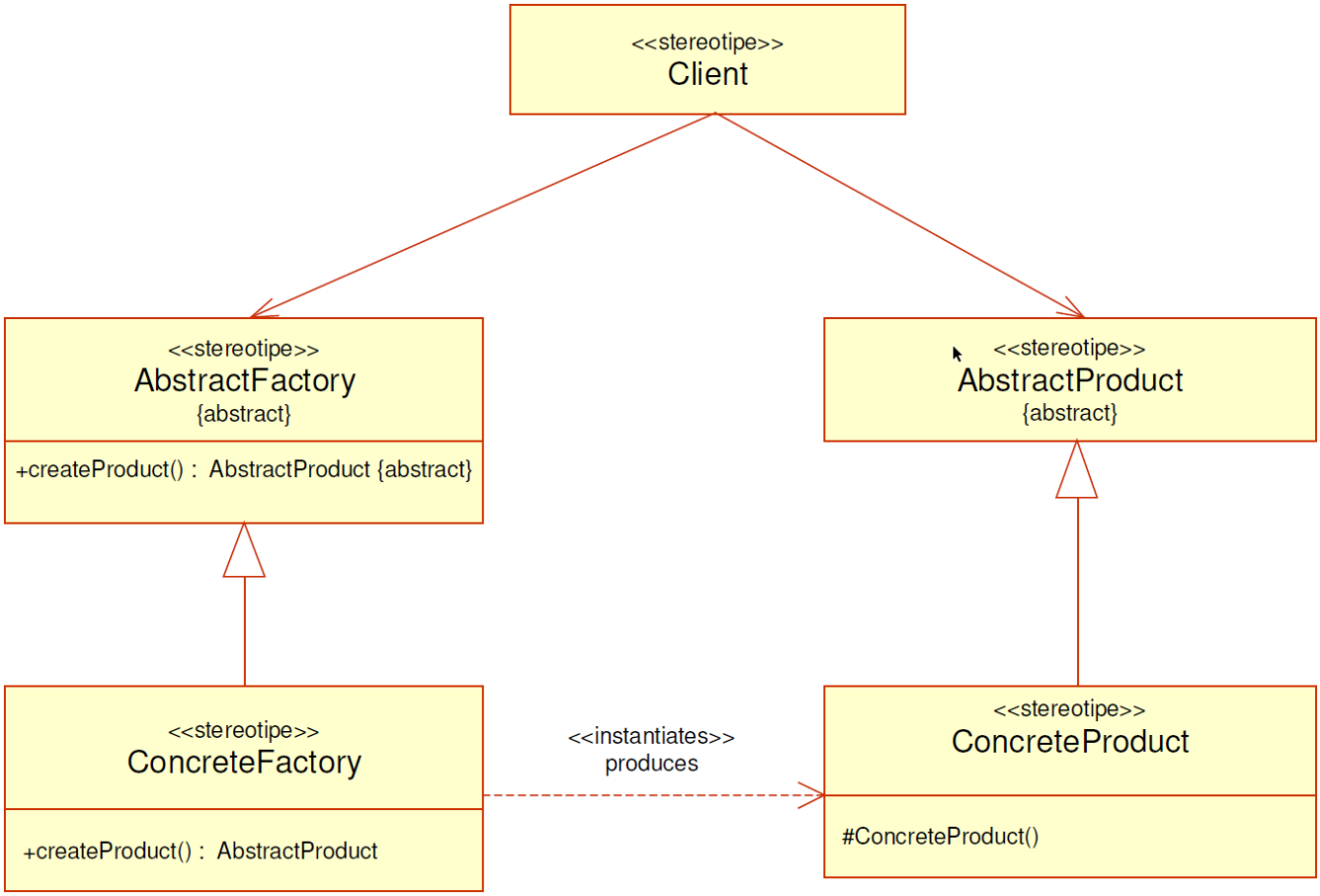
Abstract Factory

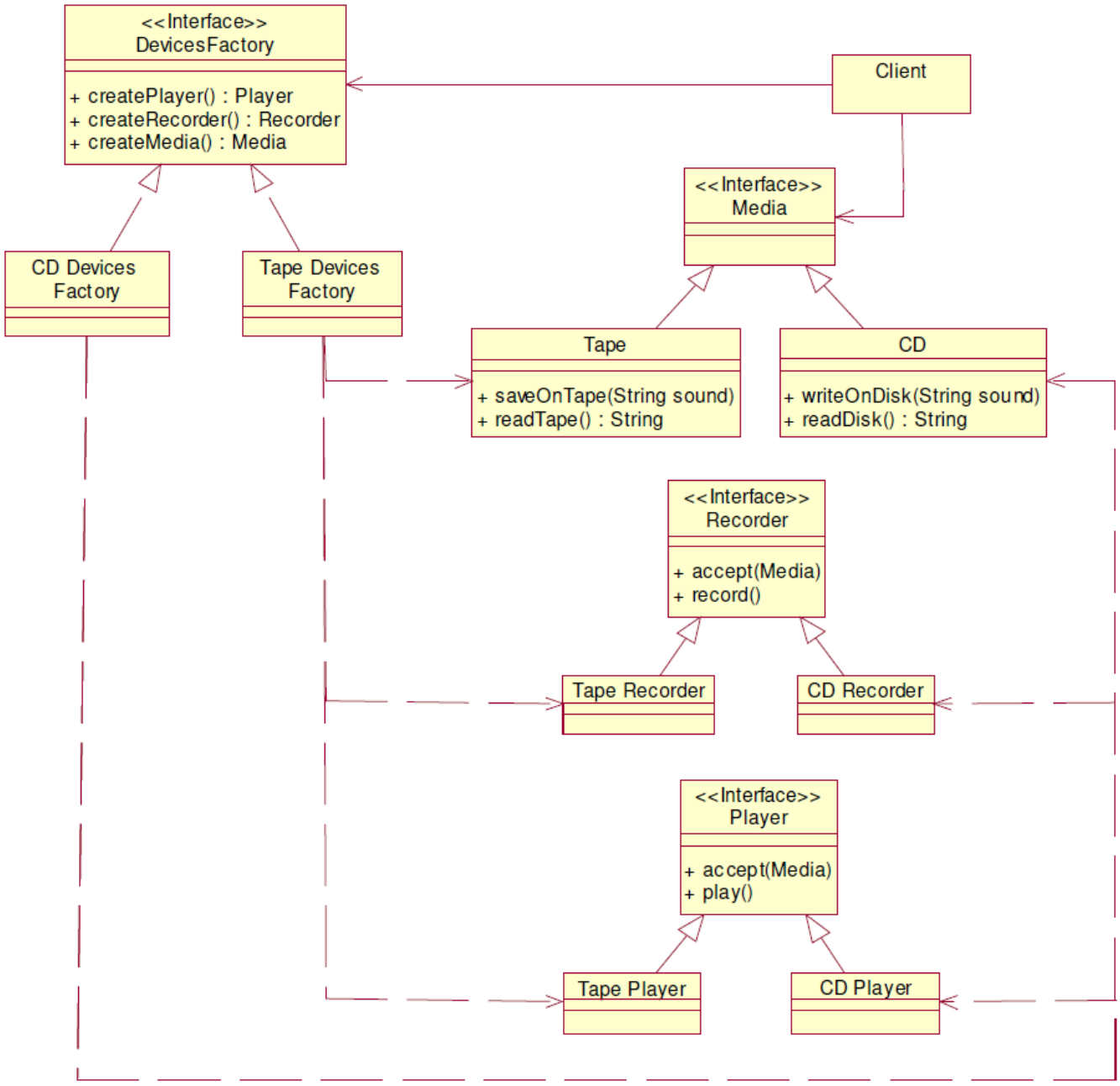
Nome: Abstract Factory

Problema: Come creare famiglie di classi correlate che implementano un'interfaccia comune?

Soluzione: Definire un'interfaccia factory (la factory astratta). Definire una classe factory concreta per ciascuna famiglia di elementi da creare. Opzionalmente, definire una vera classe astratta che implementa l'interfaccia factory e fornisce servizi comuni alle factory concrete che la estendono.

- Presenta un'interfaccia per la creazione di famiglie di prodotti, in modo tale che il cliente che li utilizza non abbia conoscenza delle loro concrete classi. Questo consente:
- di assicurarsi che il cliente crei soltanto prodotti vincolati fra di loro l'utilizzo di diverse famiglie di prodotti da parte dello stesso cliente
- Una variante comune di Abstract Factory consiste nel creare una classe astratta factory a cui si accede utilizzando il pattern Singleton
- è usata nelle librerie Java per la creazione di famiglie di elementi GUI per diversi sistemi operativi e sottosistemi GUI





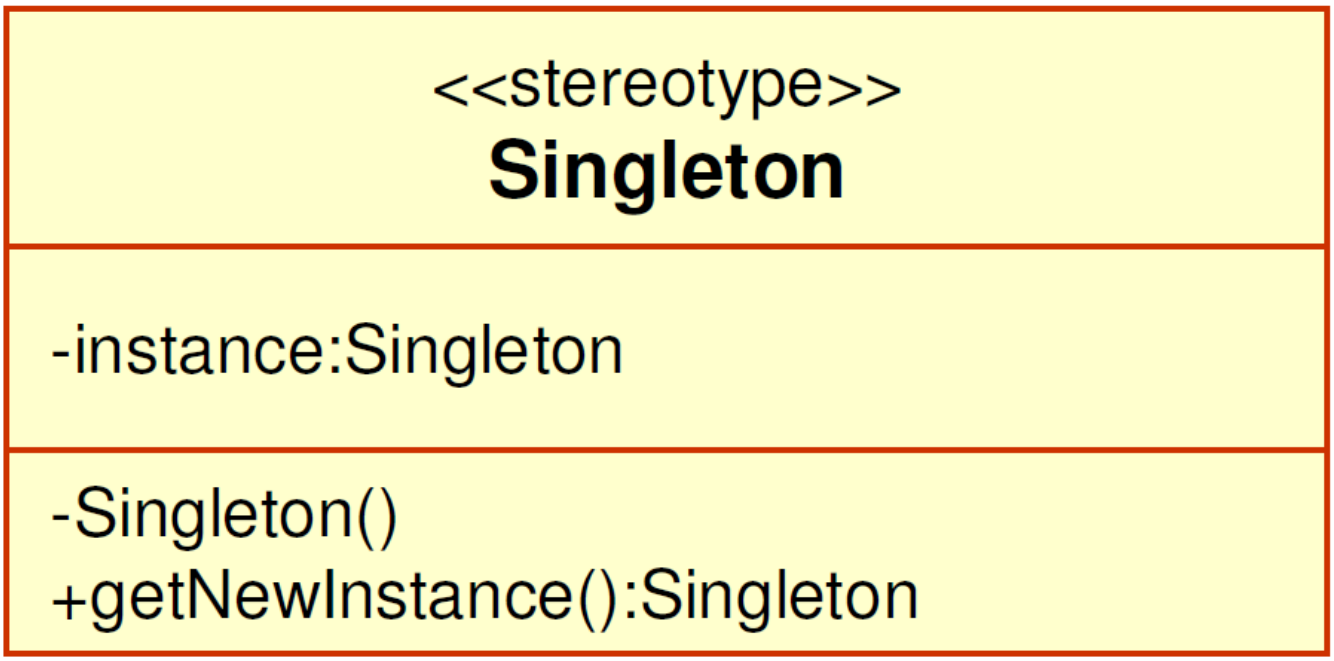
Singleton

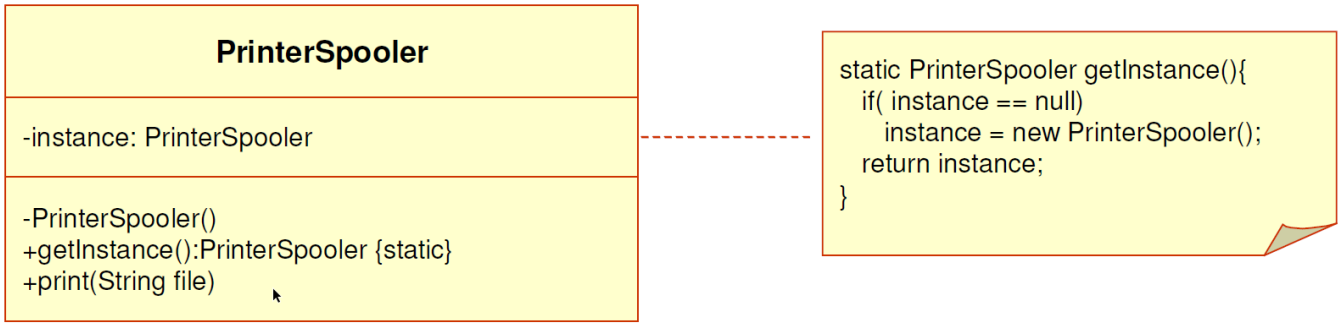
Nome: Singleton

Problema: è consentita (o richiesta) esattamente una sola istanza di una classe, ovvero un "singleton". Gli altri oggetti hanno bisogno di un punto di accesso globale e singolo a questo oggetto.

Soluzione: Difinisci un metodo statico (di classe) della classe che restituisce l'oggetto singleton.

- Il "Singleton" pattern definisce una classe della quale è possibile la istanziazione di un unico oggetto, tramite l'invocazione a un metodo della classe, incaricato della produzione degli oggetti
- Le diverse richieste di istanziazione comportano la restituzione di un riferimento allo stesso oggetto
- In UML un singleton viene illustrato con un "1" nella sezione del nome, in alto a destra





Strutturali: Adapter, Composite e Decorator

Risolvono problematiche inerenti la struttura delle classi e degli oggetti:

Adapter

Nome: Adapter

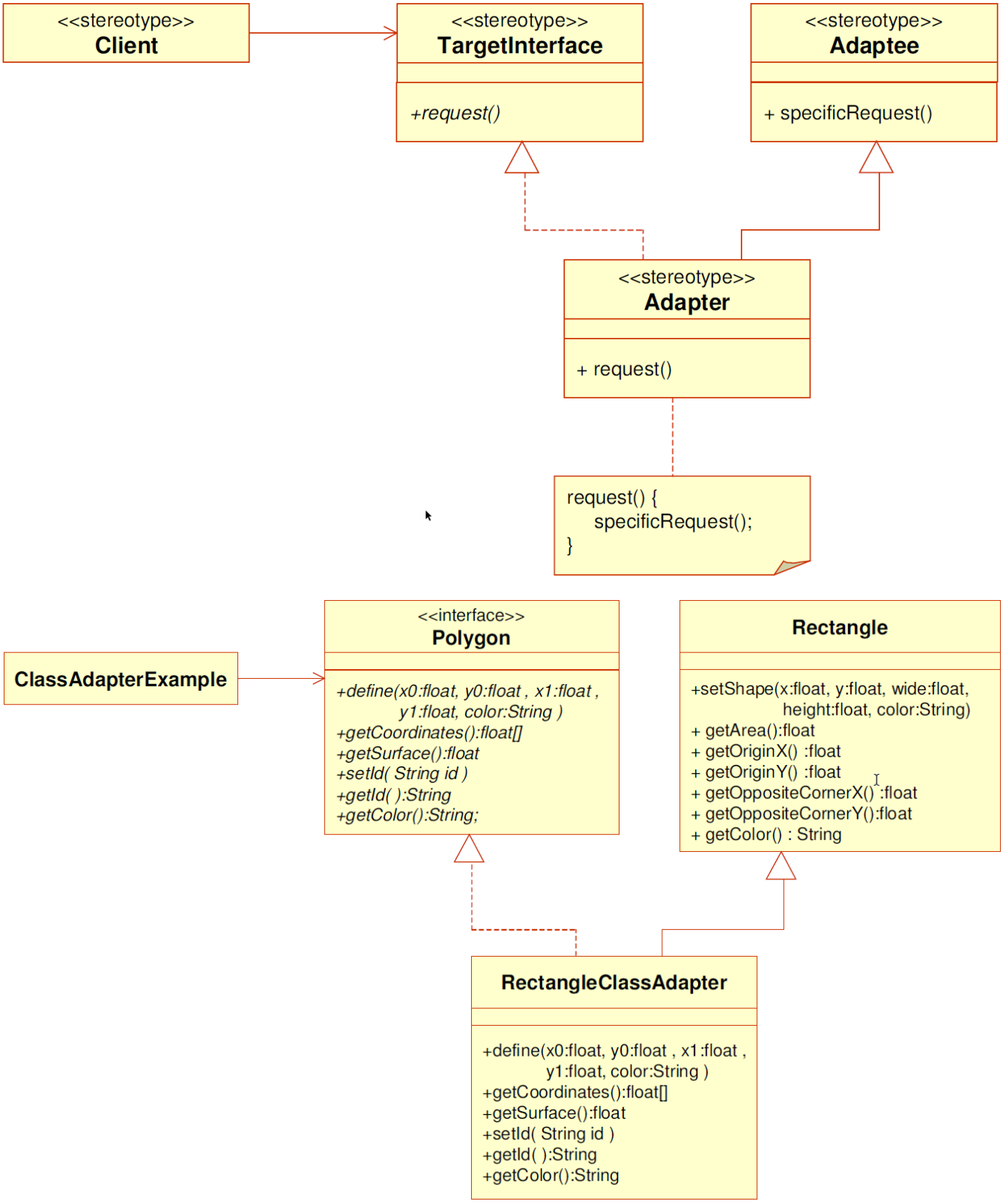
Problema: Come gestire interfacce incompatibili, o fornire un'interfaccia stabile a comportamenti simili ma con interfacce diverse?

Soluzione: Converti l'interfaccia originale di un componente in un'altra interfaccia, attraverso un oggetto adattatore intermedio.

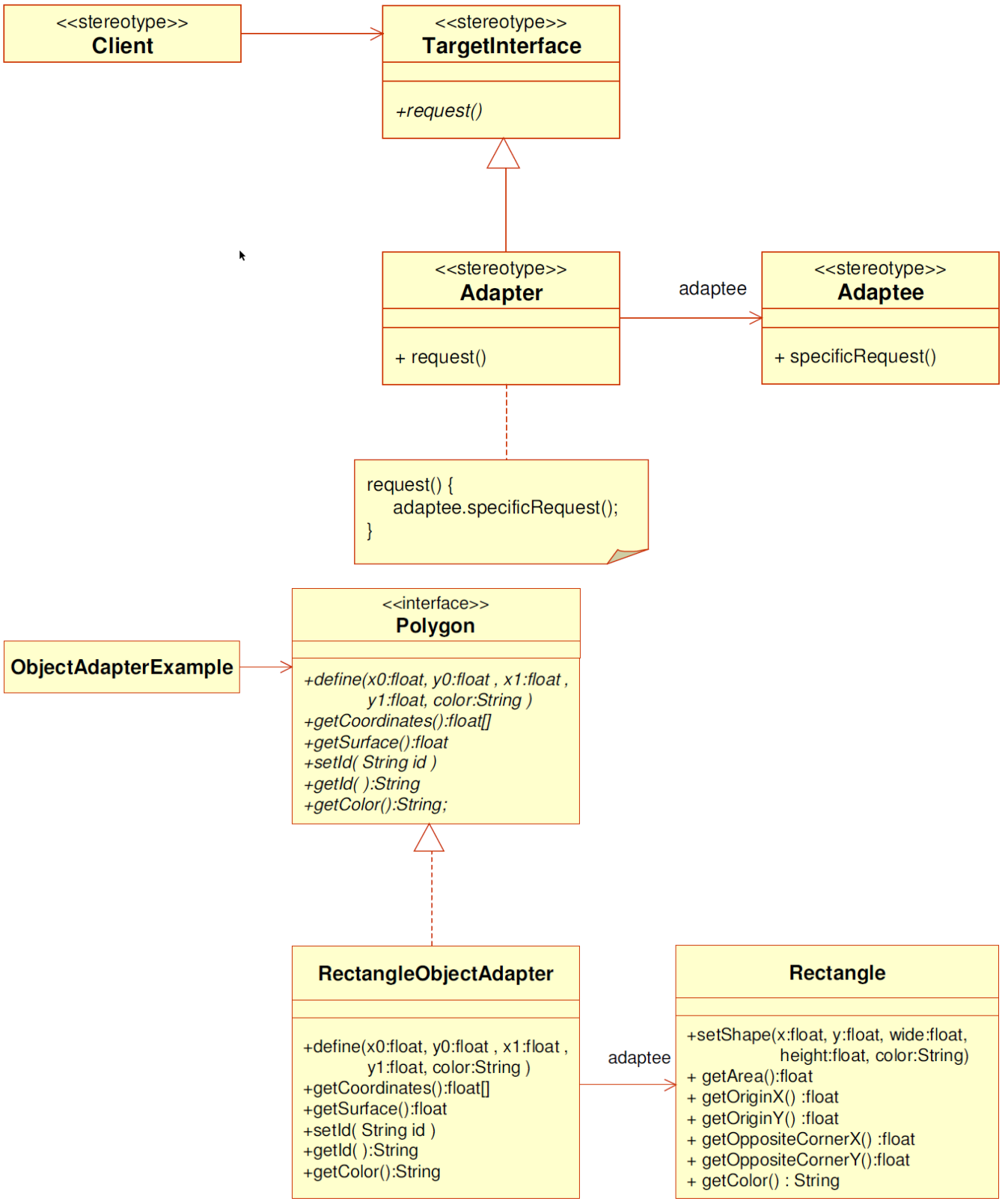
- Si consideri una coppia di oggetti software in una relazione client-server. Si parla di interfacce incompatibili quando l'oggetto server offre servizi di interesse per l'oggetto client ma l'oggetto client vuole fruire di questi servizi in una modalità diversa da quella prevista dall'oggetto server (interfacce imcompatibili)
- Ci sono più oggetti server che offrono servizi simili; questi oggetti hanno interfacce simili ma diverse tra loro. Un oggetto client vuole fruire dei servizi offerti da uno tra questi oggetti server (componenti simili con interfacce diverse)
- In generale, un adattatore riceve richieste dai suoi client, per esempio da un oggetto dello strato di dominio, nel formato client dell'adattatore
- L'adattatore poi adatta, trasforma, una richiesta ricevuta in una richiesta nel formato del server

- L'adattatore invia la richiesta al server
- Se il server fornisce una risposta, lo fa nel formato del server
- L'adattatore adatta, trasforma, la risposta ricevuta dal server in una risposta nel formato del client e poi la restituisce al suo client

Classe



Object



Composite

Nome: Composite

Problema: Come trattare un gruppo o una struttura composta di oggetti (polimorficamente) dello stesso tipo nello stesso modo di un oggetto non composto (atomico)?

Soluzione: Definisci le classi per gli oggetti composti e atomici in modo che implementino la stessa interfaccia

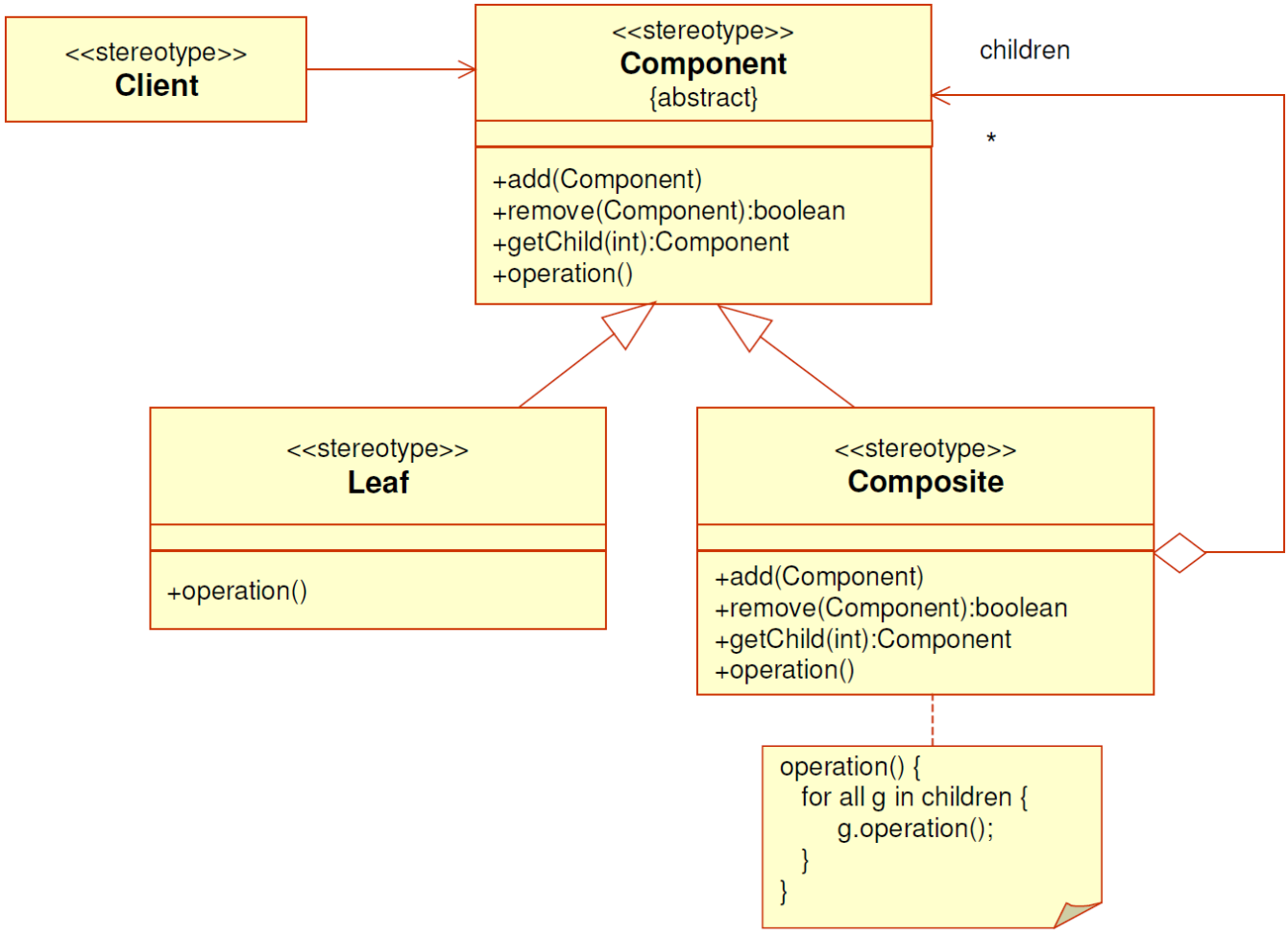
Consente la costruzione di gerarchie di oggetti composti. Gli oggetti composti possono essere formati da oggetti singoli, oppure da altri oggetti composti. Questo pattern è utile nei casi in cui si vuole:

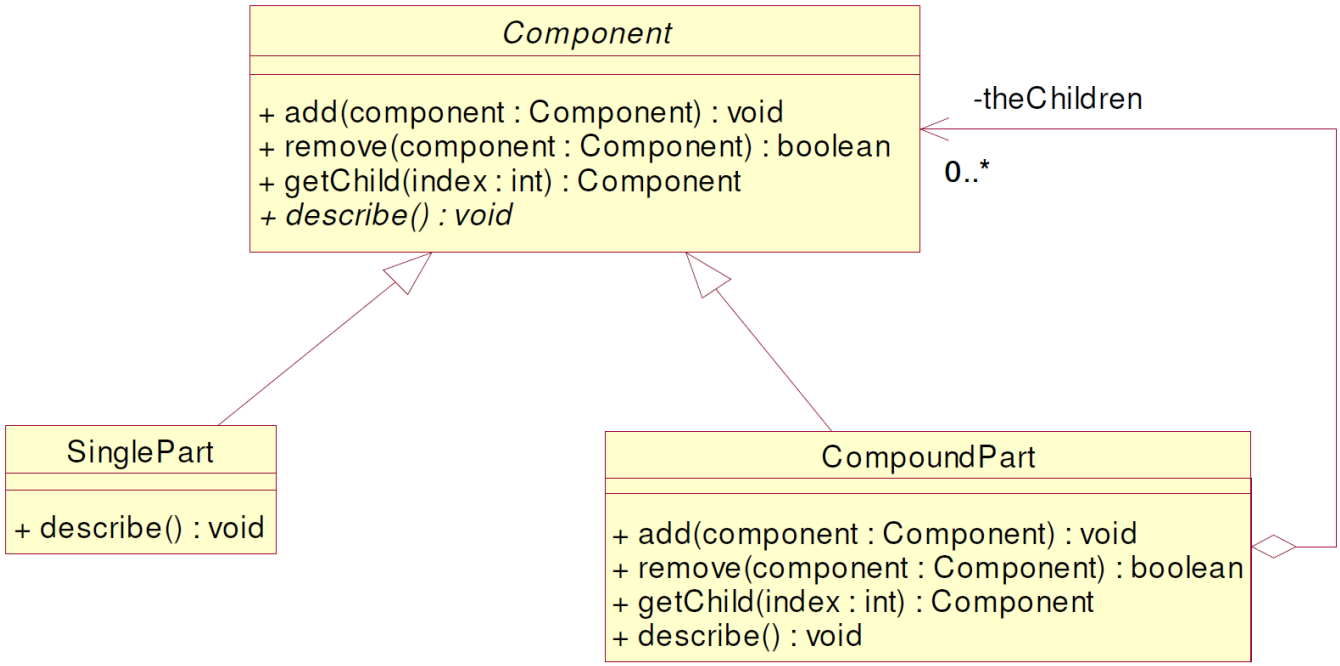
- Rappresentare gerarchie di oggetti tutto-parte
- Essere in grado di ignorare le differenze tra oggetti singoli e oggetti composti
- è nota anche come struttura ad albero, composizione ricorsiva, struttura induttiva: foglie e nodi hanno la stessa funzionalità
- Implementa la stessa interfaccia per tutti gli elementi contenuti

Il pattern definisce la classe astratta componente che deve essere estesa in due sottoclassi:

- Una che rappresenta i singoli componenti (foglia)
- L'altra che rappresenta i componenti composti e che si implementa come contenitore di componenti

Nota: i componenti composti possono immagazzinare sia componenti singoli sia altri contenitori.





> **In breve**
>
> Il pattern composite permette di costruire strutture ricorsive (ad esempio un albero di elementi) in modo che ad un cliente (una classe che usa la struttura) l'intera struttura sia vista come una singola entità. Quindi l'interfaccia alle entità atomiche (foglie) è esattamente la stessa dell'interfaccia delle entità composte. In essenza tutti gli elementi della struttura hanno la stessa interfaccia senza considerare se sono composti o atomici.

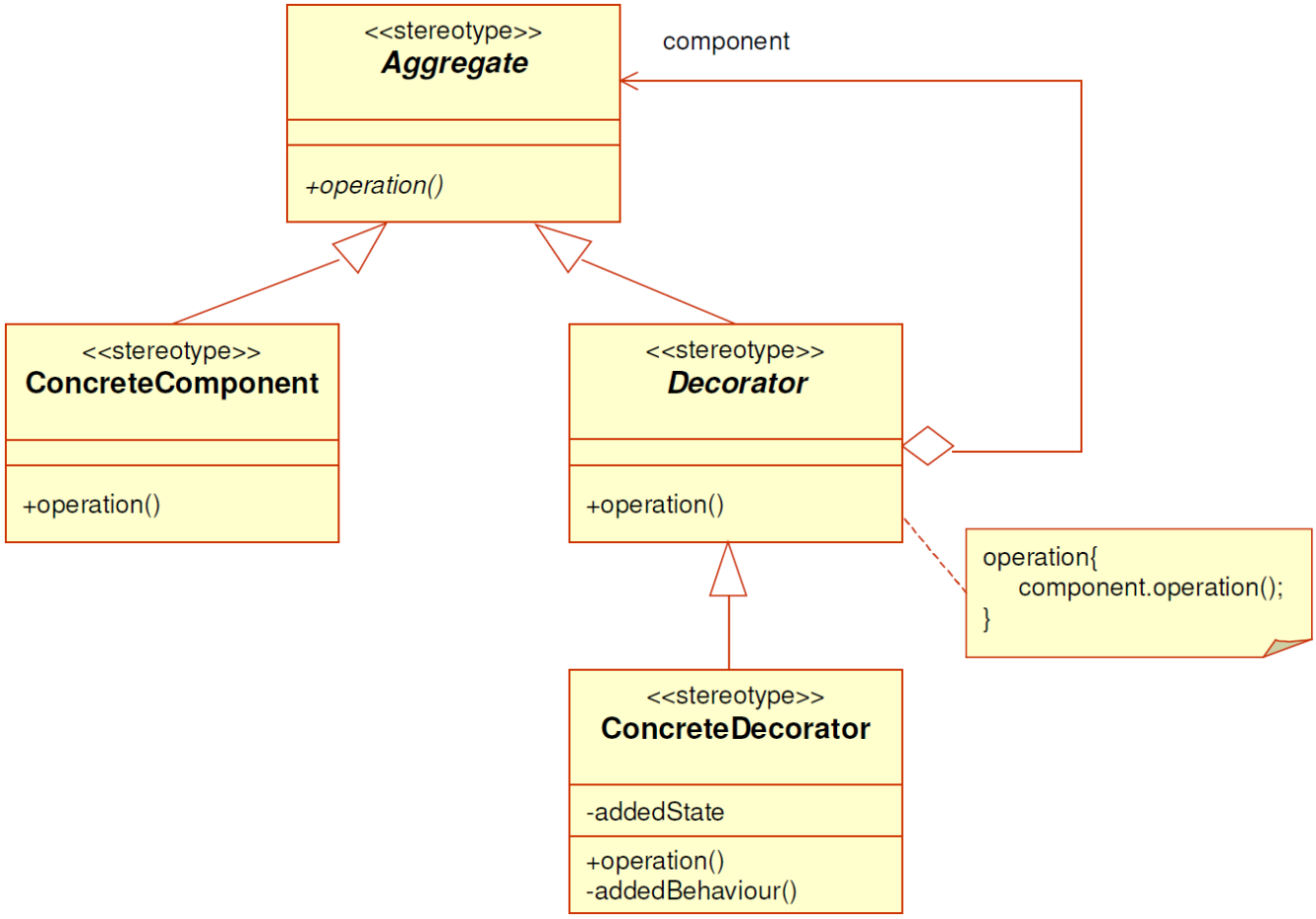
Decorator

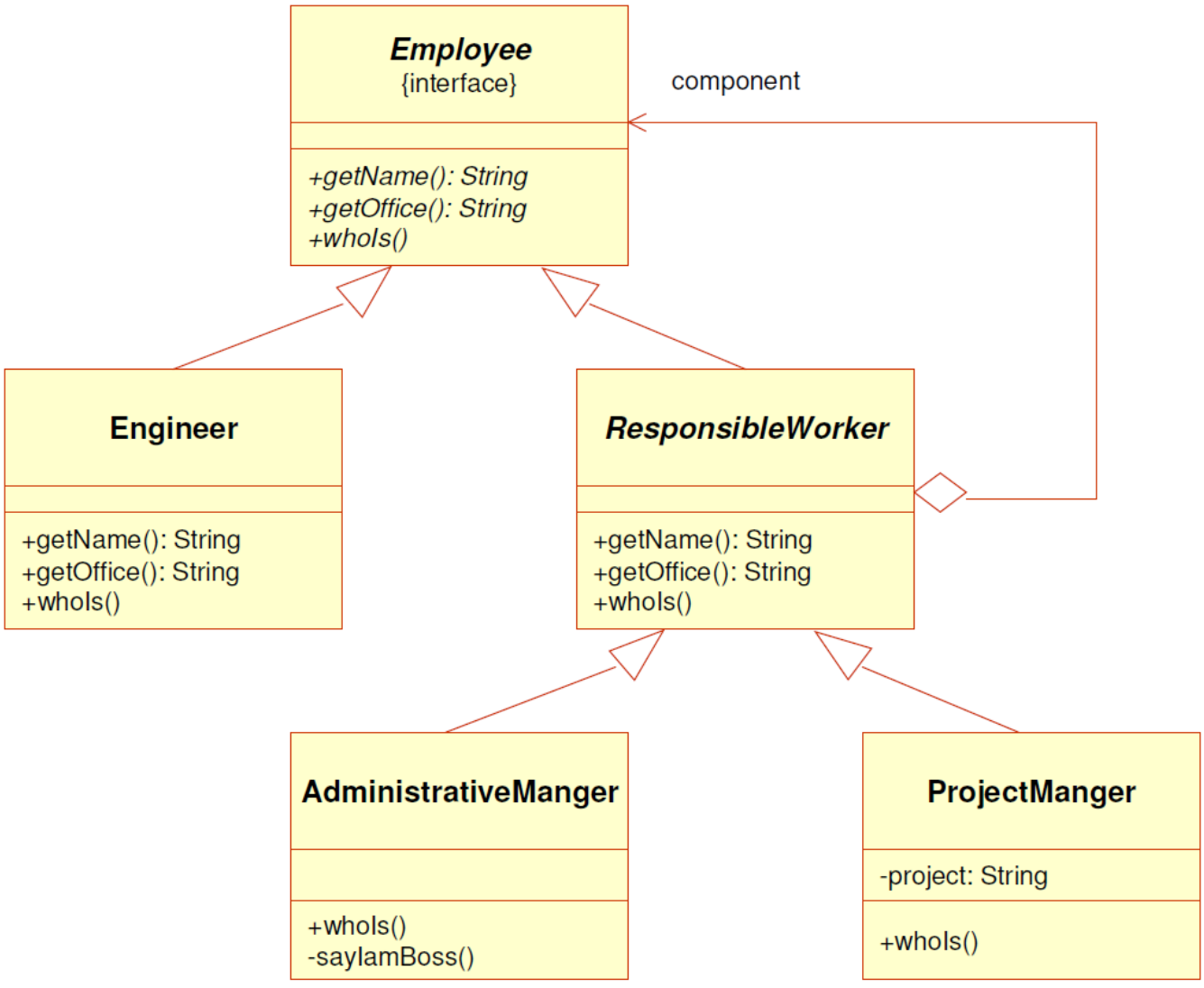
Nome: Decorator (wrapper)

Problema: Come permettere di assegnare una o più responsabilità addizionali ad un oggetto in maniera dinamica ed evitare il problema della relazione statica? Come provvedere una alternativa più flessibile al meccanismo di sottoclasse ed evitare il problema di avere una gerarchia di classi complessa?

Soluzione: Inglobare l'oggetto all'interno di un altro che aggiunge le nuove funzionalità

- Permette di aggiungere responsabilità ad oggetti individualmente, dinamicamente e in modo trasparente, ossia senza impatti sugli altri oggetti
- Le responsabilità possono essere ritirate
- Permette di evitare l'esplosione delle sottoclassi per supportare un ampio numero di estensioni e combinazioni di esse oppure quando le definizioni sono nascoste e non disponibili alle sottoclassi


<<stereotype>>
Aggregate
+operation()
component
<<stereotype>>
ConcreteComponent
+operation()
<<stereotype>>
Decorator
+operation()
operation{
component.operation();
}
<<stereotype>>
ConcreteDecorator
-addedState
+operation()
-addedBehaviour()


Employee
{interface}
+getName(): String
+getOffice(): String
+whoIs()
component
Engineer
+getName(): String
+getOffice(): String
+whoIs()
ResponsibleWorker
+getName(): String
+getOffice(): String
+whoIs()
AdministrativeManger
+whoIs()
-saylamBoss()
ProjectManger
-project: String
+whoIs()

> **In breve**
>
> Il pattern decorator permette ad una entità di contenere completamente un'altra entità così che l'utilizzo del decoratore sia identico all'entità contenuta. Questo consente al decoratore di modificare il comportamento e/o il contenuto di tutto ciò che sta incapsulato senza cambiare l'aspetto esteriore dell'entità. Ad esempio, è possibile utilizzare un decoratore per aggiungere l'attività di logging dell'elemento contenuto senza cambiare il conportamento di questo.

> **Composite vs Decorator**
>
> - Il patterm composite: fornisce un'interfaccia comune a elementi atomici (foglie) e composti
> - Il pattern decorator: fornisce caratteristiche addizionali ad elementi atomici (foglie), mantenendo un'interfaccia comune

Comportamentali: Observer, State, Strategy e Visitor

Forniscono soluzione alle più comuni tipologie di interazione tra gli oggetti:

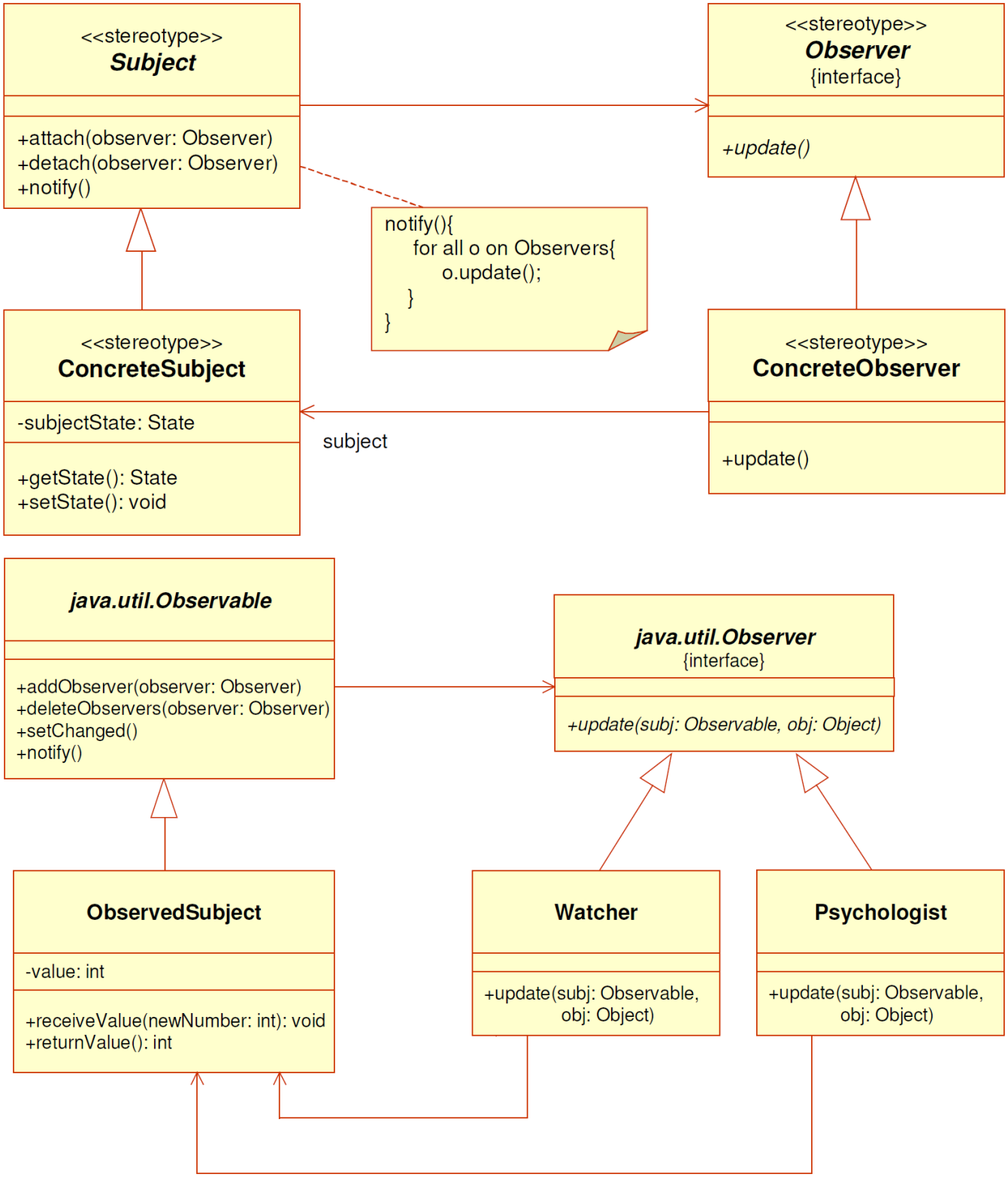
Observer

Nome: Observer

Problema: Diversi tipi di oggetti subscriber (abbonato) sono interessati ai cambiamenti di stato o agli eventi di un oggetto publisher (editore). Ciascun subscriber vuole reagire in un suo modo proprio quando il publisher genera un evento. Inoltre, il publisher vuole mantenere un accoppiamento basso verso i subscriber. Che cosa fare?

Soluzione: Definisci un'interfaccia subscriber o listener (ascoltatore). Gli oggetti subscriber implementano questa interfaccia. Il publisher registra dinamicamente i subscriber che sono interessati ai suoi eventi, e li avvisa quando si verifica un evento.

- Definisce una dipendenza tra oggetti di tipo uno-a-molti: quando lo stato di un oggetto cambia, tale evento viene notificato a tutti gli oggetti dipendenti, essi vengono automaticamente aggiornati
- L'oggetto che notifica il cambiamento di stato non fa alcuna assunzione sulla natura degli oggetti notificati: le due tipologie di oggetti sono disacoppiati
- Il numero degli oggetti affetti dal cambiamento di stato di un oggetto non è noto a priori
- Fornisce un modo per accoppiare in maniera debole degli oggetti che devono comunicare (eventi). I publisher conoscono i subscriber solo attraverso un'interfaccia, e i subscriber possono registrarsi (o cancellare la propria registrazione) dinamicamente con il publisher



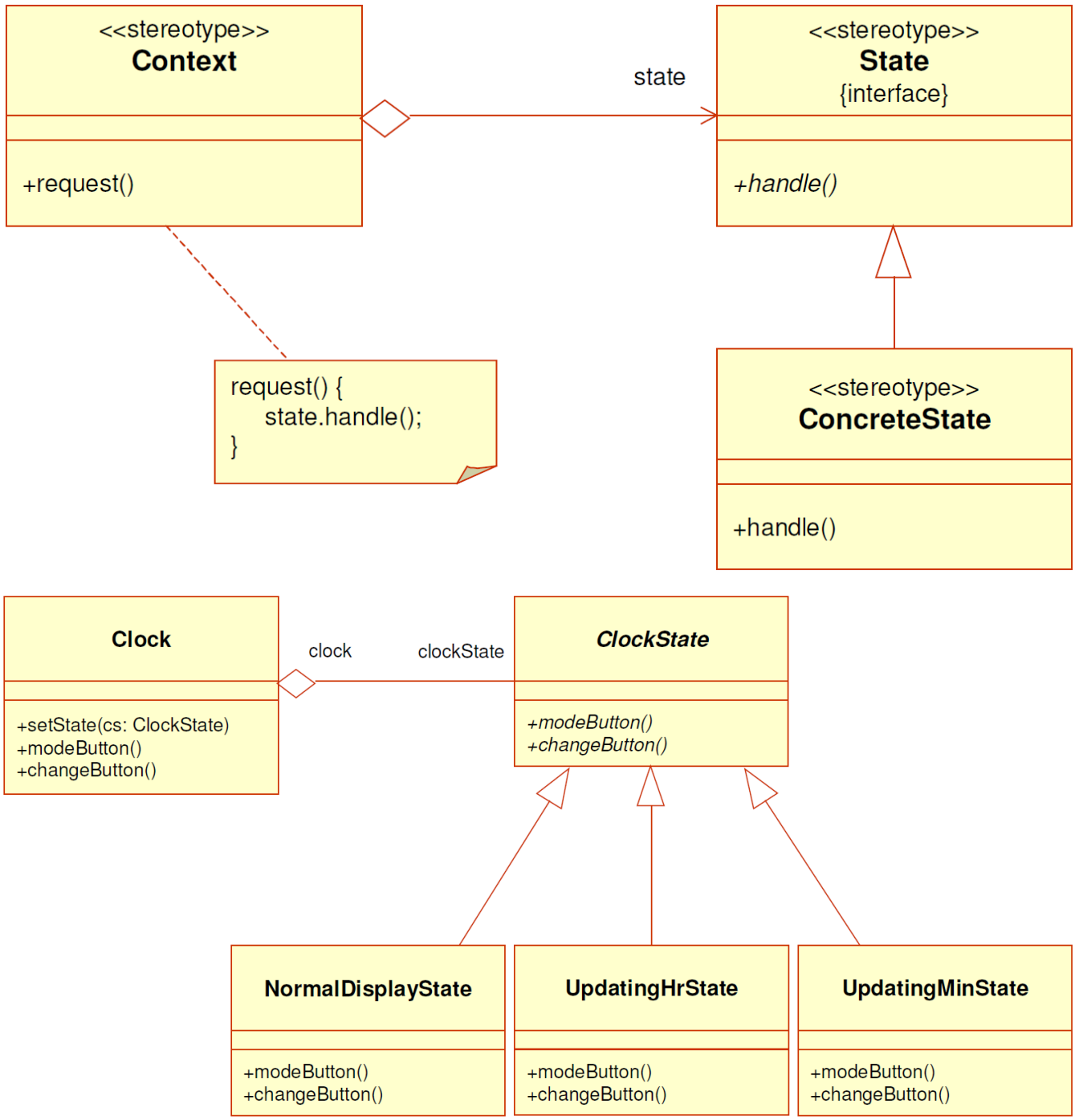
State

Nome: State

Problema: Il comportamento di un oggetto dipende da suo stato e i suoi metodi contengono logica condizionale per casi che riflette le azioni condizionali che dipendono dallo stato. C'è un'alternativa alla logica condizionale?

Soluzione: Crea delle classi stato per ciascuno stato, che implementano un'interfaccia comune. Delega le operazioni che dipendono dallo stato dall'oggetto contesto all'oggetto stato corrente corrispondente. Assicura che l'oggetto contesto referenzi sempre un oggetto stato che riflette il suo stato corrente.

- Permette ad un oggetto di modificare il suo comportamento quando cambia il suo stato interno
- Può sembrare che l'oggetto modifichi la sua classe



Strategy

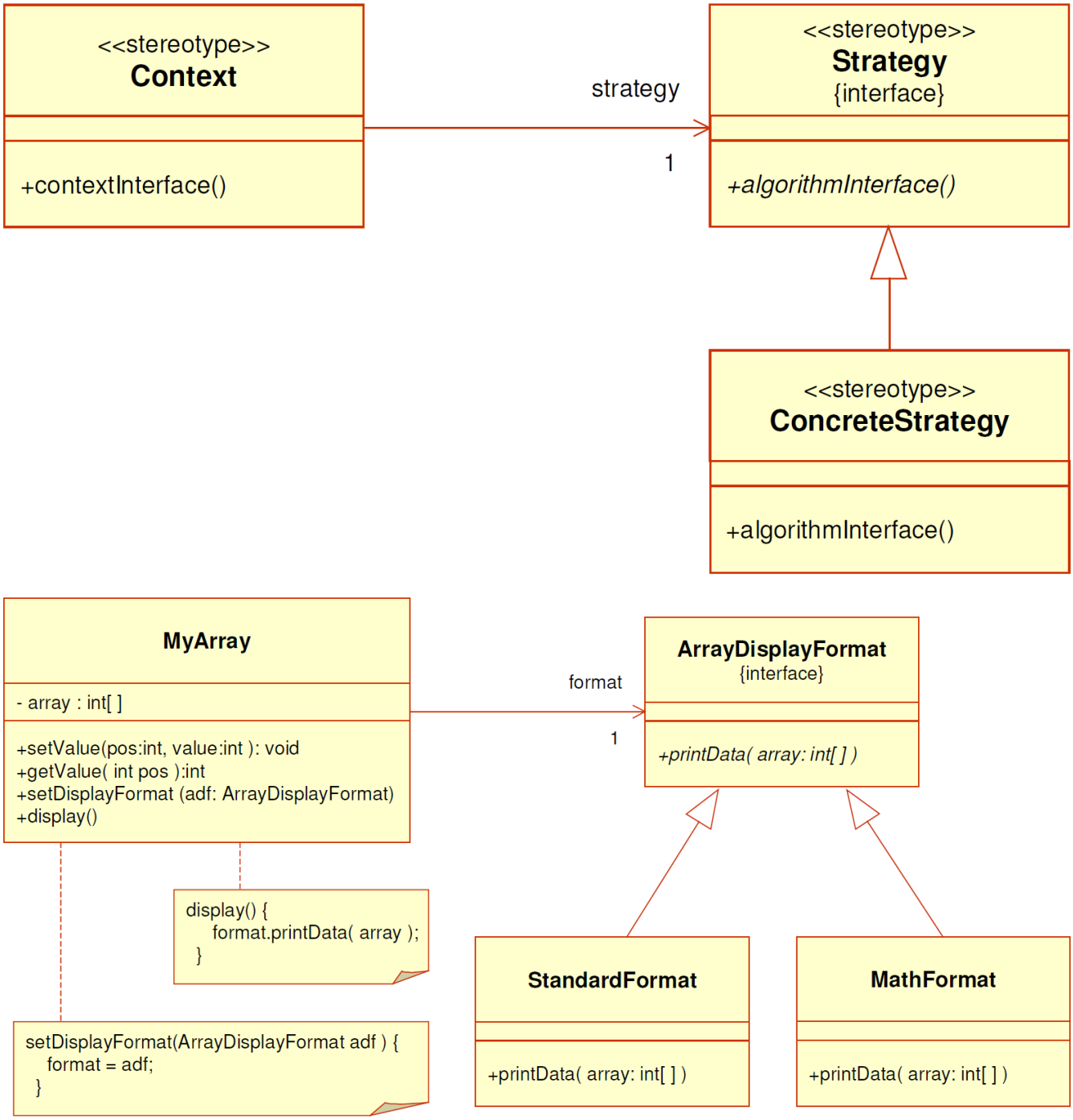
Nome: Strategy (policy)

Problema: Come progettare per gestire un insieme di algoritmi o politiche variabili ma correlati? Come progettare per consentire di modificare questi algoritmi o politiche?

Soluzione: Definisci ciascun algoritmo/politica/strategia in una classe separata, con un'interfaccia comune.

- L'oggetto contesto è l'oggetto a cui va applicato l'algoritmo
- L'oggetto contesto è associato a un oggetto strategia, che è l'oggetto che implementa un algoritmo
- Consente la definizione di una famiglia di algoritmi, incapsula ognuno e li rende intercambiabili tra di loro

- Permette di modificare gli algoritmi in modo indipendente dai clienti che fanno uso di essi
- Disaccoppia gli algoritmi dai clienti che vogliono usarli dinamicamente
- Permette che un oggetto client possa usare indifferentemente uno o l'altro algoritmo
- è utile dove è necessario modificare il comportamento a runtime di una classe
- Usa la composizione invece dell'ereditarietà: i comportamenti di una classe non dovrebbero essere ereditati ma piuttosto incapsulati usando la dichiarazione di interfaccia



Il pattern State si occupa di che cosa (stato o tipo) un oggetto è (al suo interno) e incapsula un comportamento dipendente dallo stato. Fare cose diverse in base allo stato, lasciando il chiamante sollevato dall'onere di soddisfare ogni stato possibile.

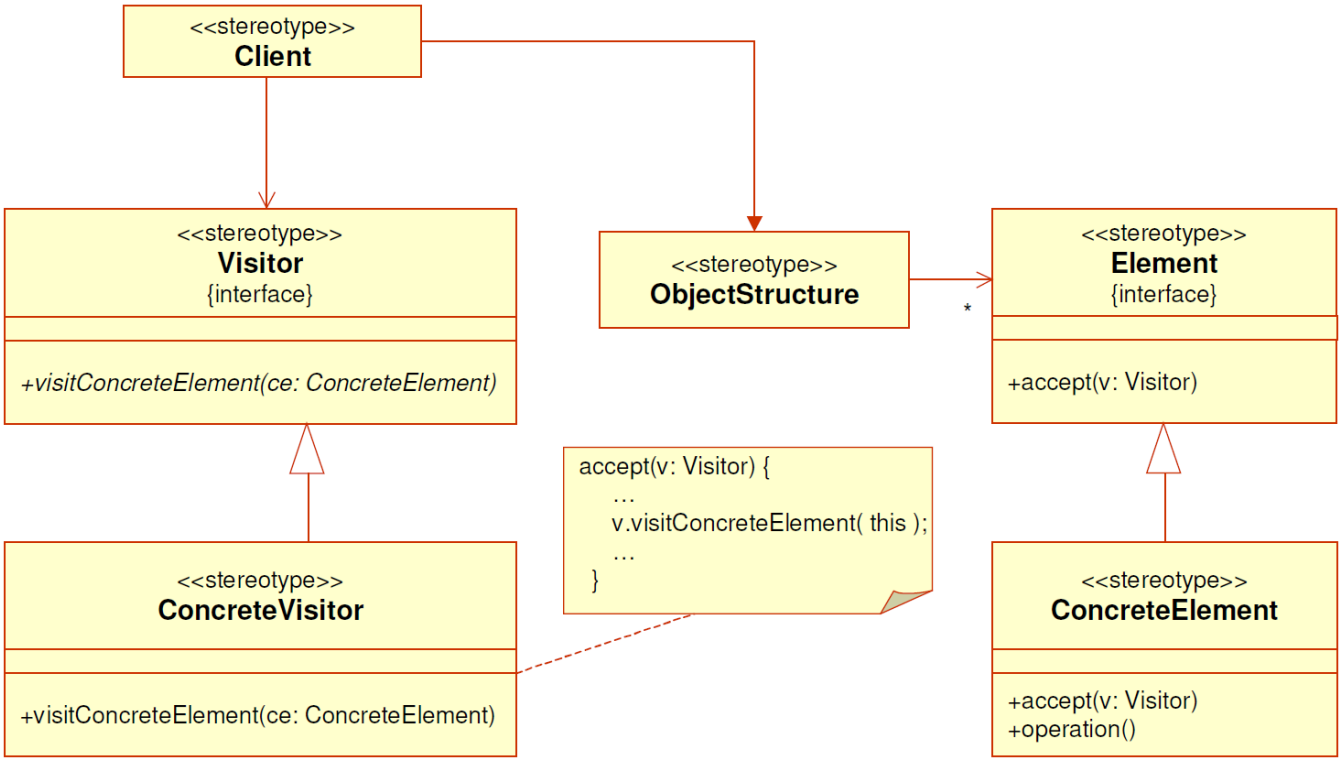
Visitor

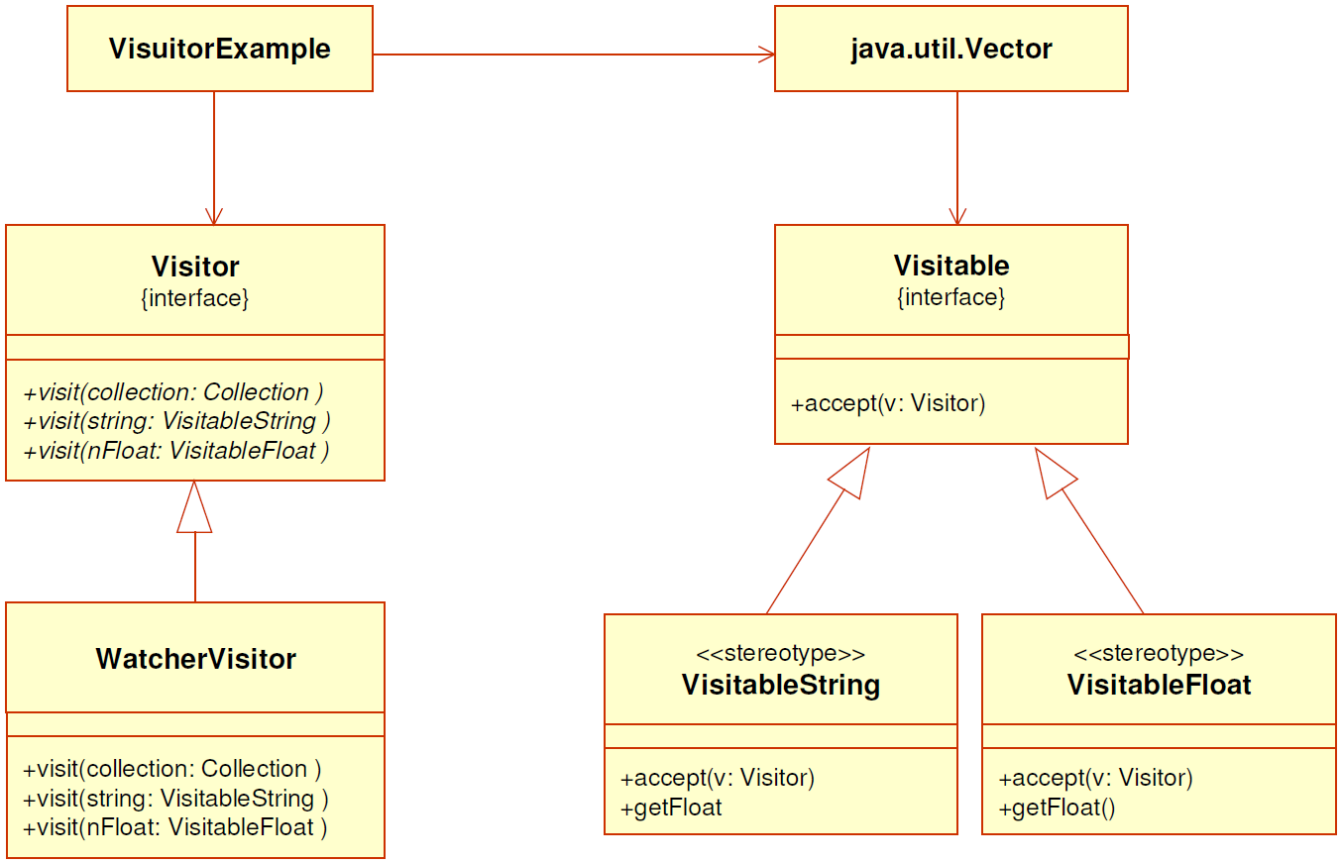
Nome: Visitor

Problema: Come separare l'operazione applicata su un contenitore complesso dalla struttura dati cui è applicata? Come poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa? Come attraversare il contenitore complesso i cui elementi sono eterogenei applicando azioni dipendenti dal tipo degli elementi?

Soluzione: Creare un oggetto (ConcreteVisitor) che è in grado di percorrere la collezione, e di applicare un metodo proprio su ogni oggetto (Element) visitato nella collezione (avendo un riferimento a questi ultimi come parametro). Ogni oggetto della collezione aderisce ad un'interfaccia (Visitable) che consente al ConcreteVisitor di essere accettato da parte di ogni Element. Il Visitor analizza il tipo di oggetto ricevuto, fa l'invocazione alla particolare operazione che deve eseguire.

- Flessibilità delle operazioni
- Organizzazione logica
- Visita di vari tipi di classe
- Mantenimento di uno stato aggiornabile ad ogni visita
- Le diverse modalità di visita della struttura possono essere definite come sottoclassi del Visitor.




VisuitorExample
java.util.Vector
Visitor
{interface}
+visit(collection: Collection)
+visit(string: VisitableString)
+visit(nFloat: VisitableFloat)
Visitable
{interface}
+accept(v: Visitor)
WatcherVisitor
+visit(collection: Collection)
+visit(string: VisitableString)
+visit(nFloat: VisitableFloat)
<<stereotype>>
VisitableString
+accept(v: Visitor)
+getFloat
<<stereotype>>
VisitableFloat
+accept(v: Visitor)
+getFloat()