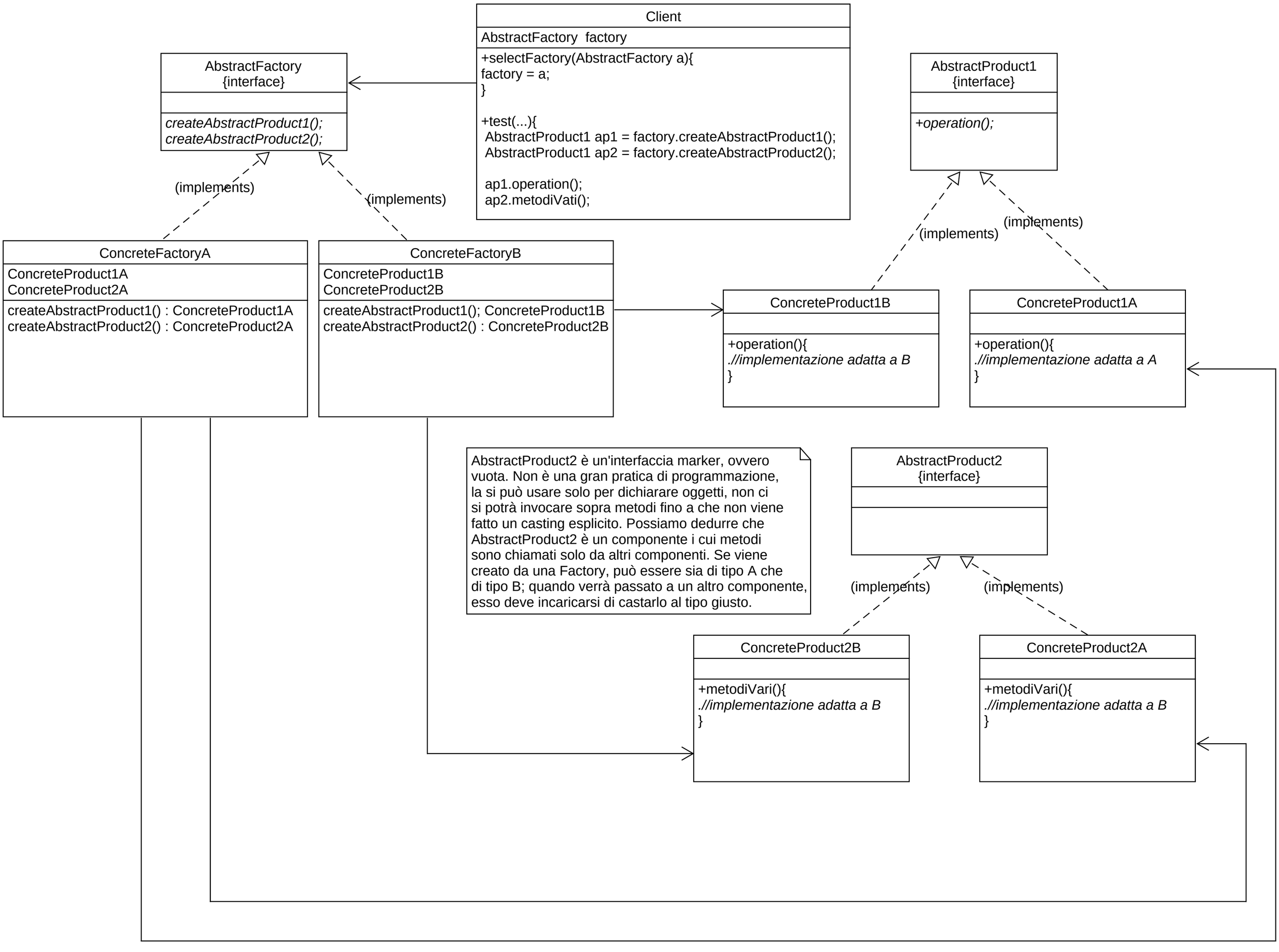
AbstractFactory



Singleton versione statica

Singleton (static class)
.//costruttore privato per evitare *.//che vengano istanziati oggetti* *.//metodi necessari al client*

difetti:
1)Devo conoscere a tempo di compilazi
tutte le informazioni necessarie a creare
Singleton
2)Non posso implementare interfacce

Singleton creato da metodo statico

Singleton
private static Singleton instance

.//costruttore privato per evitare
.//che vengano istanziati oggetti
 	
public static Singleton getInstance(){
  if(instance == null){
  instance = new Singleton();
  }
 return instance;
}

Problema: non supporta il multithreading. Esempio:
-t1 chiama per la prima volta getInstnace()
-subito dopo l'istruzione "
che ci farà oltre al ramo if, t1 viene sospeso.
-parte t2, che invoca getInstance
-Singleton non è ancora stato creato, allora anche per t2 "if(instance == null)", sarà true e quindi creerà un nuovo singleton
-t2 viene sospeso
-t1 riprende e crea anche lui un nuovo singleton

Singleton multithread

Singleton
private static Singleton instance

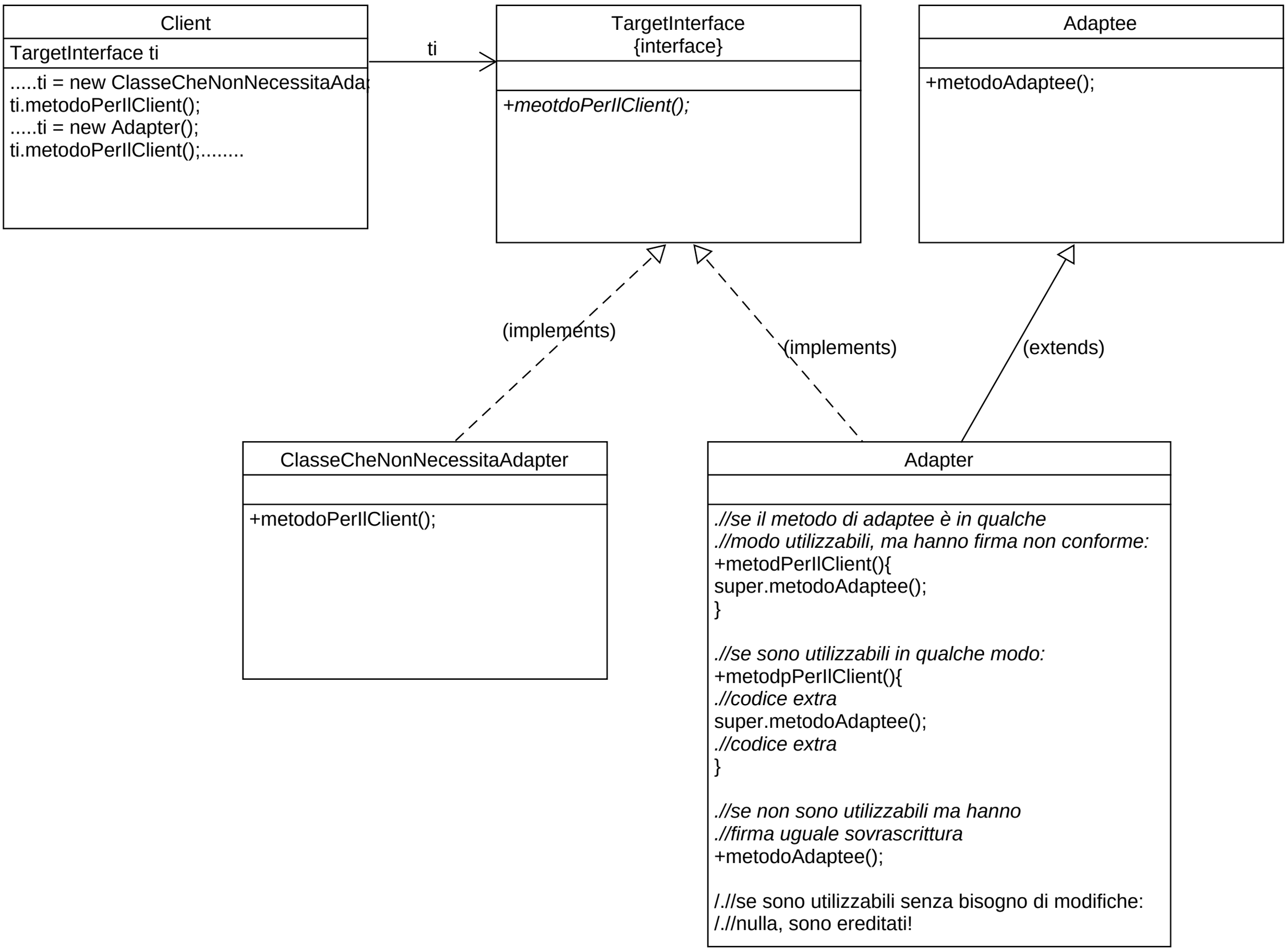
.//costruttore privato per evitare
.//che vengano istanziati oggetti
 	
public static synchronized Singleton getInstance(){
  if(instance == null){
  instance = new Singleton();
  }
 return instance;
}

Soluzione inefficiente poiché va acquisito
il lock a ogni invocazione di getInstance();
La soluzione "double-checked lcoking" aggira
il problema modificando il metodo statico in
modo che si sincronizzi sull'oggetto Class:

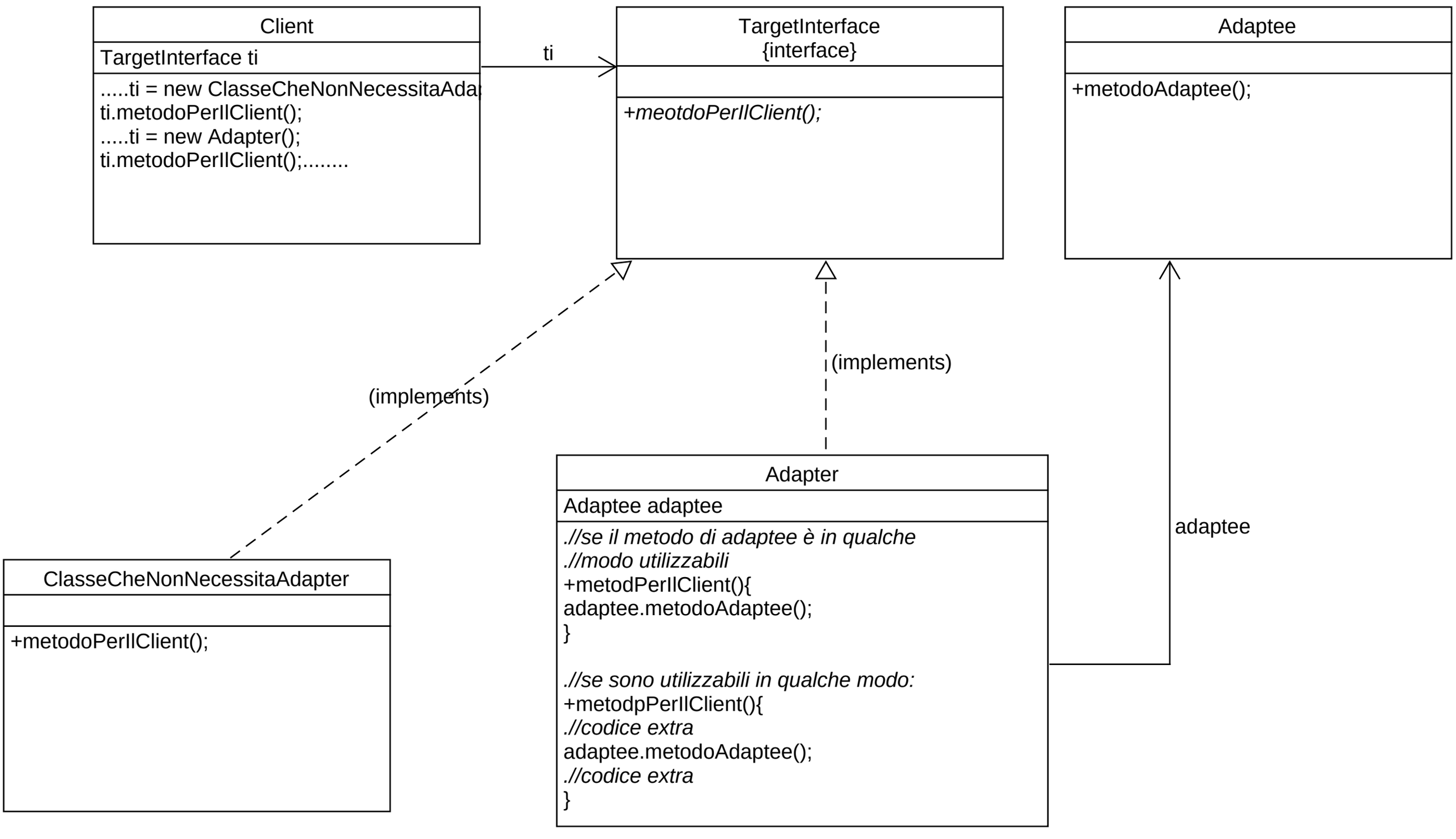
```
public static Singleton getInstance(){
   if(instance == null){
      synchronized(Singleton.class){
         if(instance == null){
            instance = new Singleton();
         }
      }
   }
  return instance;
}
```

Ma essa è scorretta perché non tiene conto dei criteri di ottimizzazione che possono far generare più di una istanza.

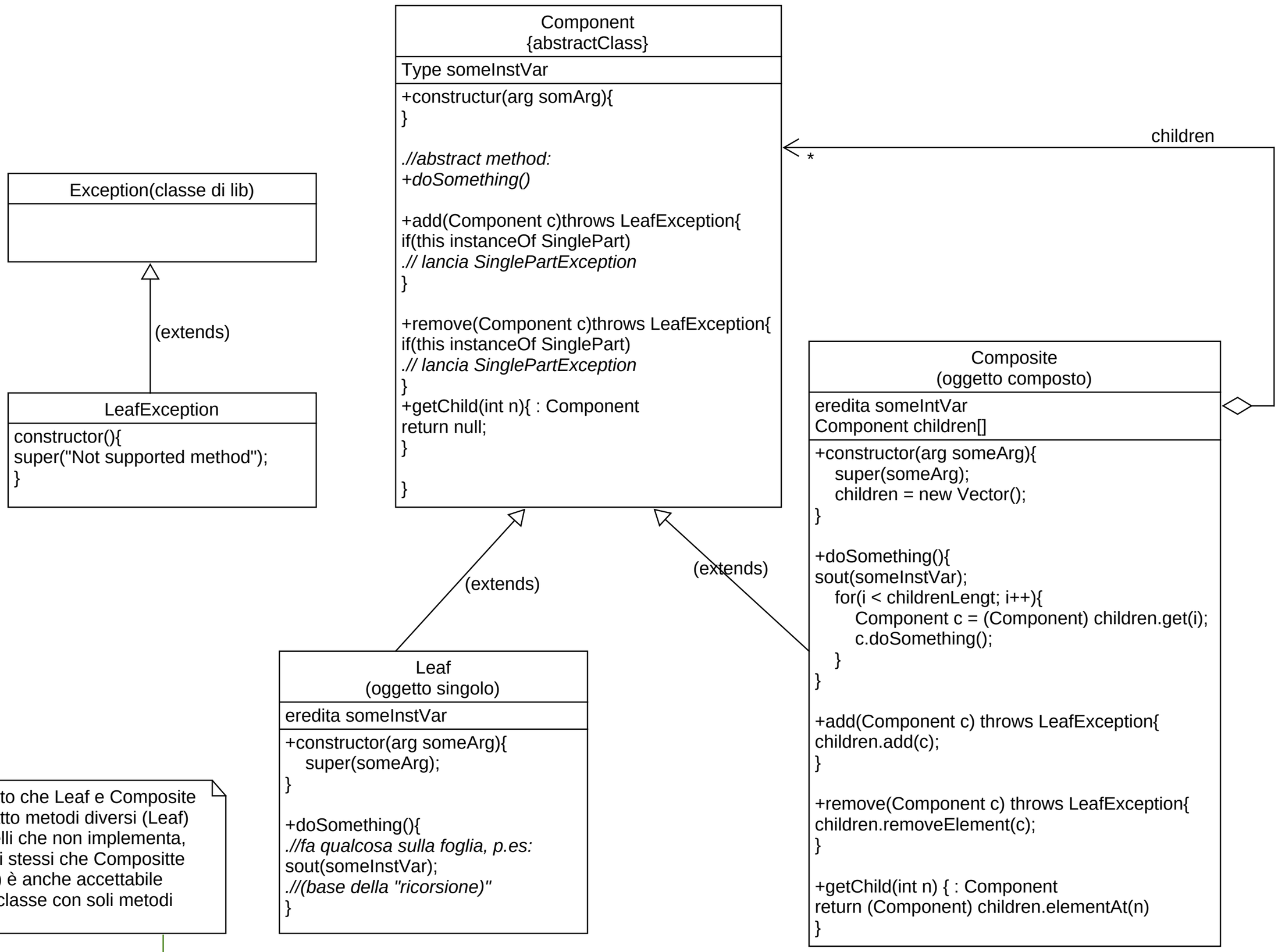
Class Adapter



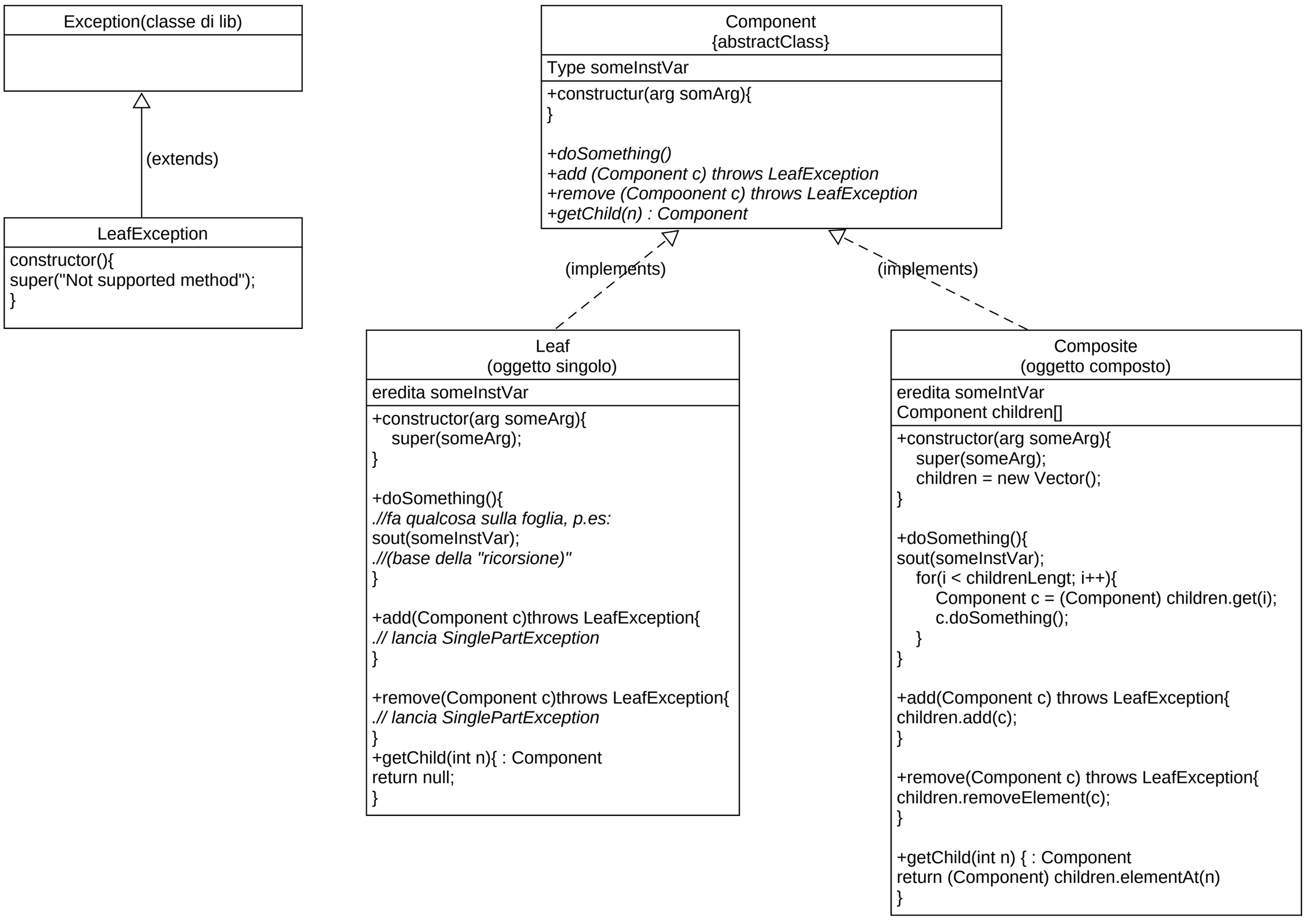
Object Adapter



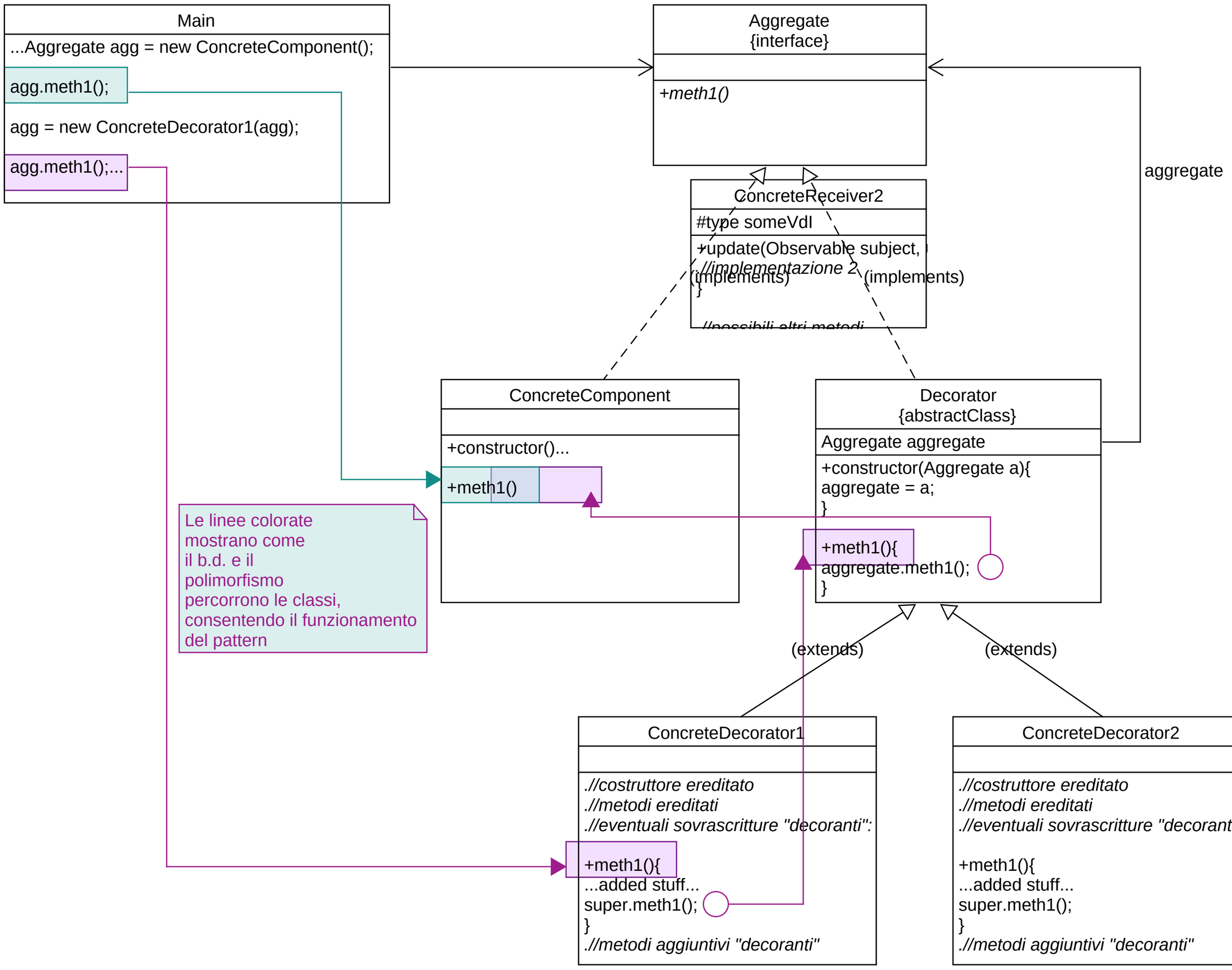
Composite - metodi ereditati



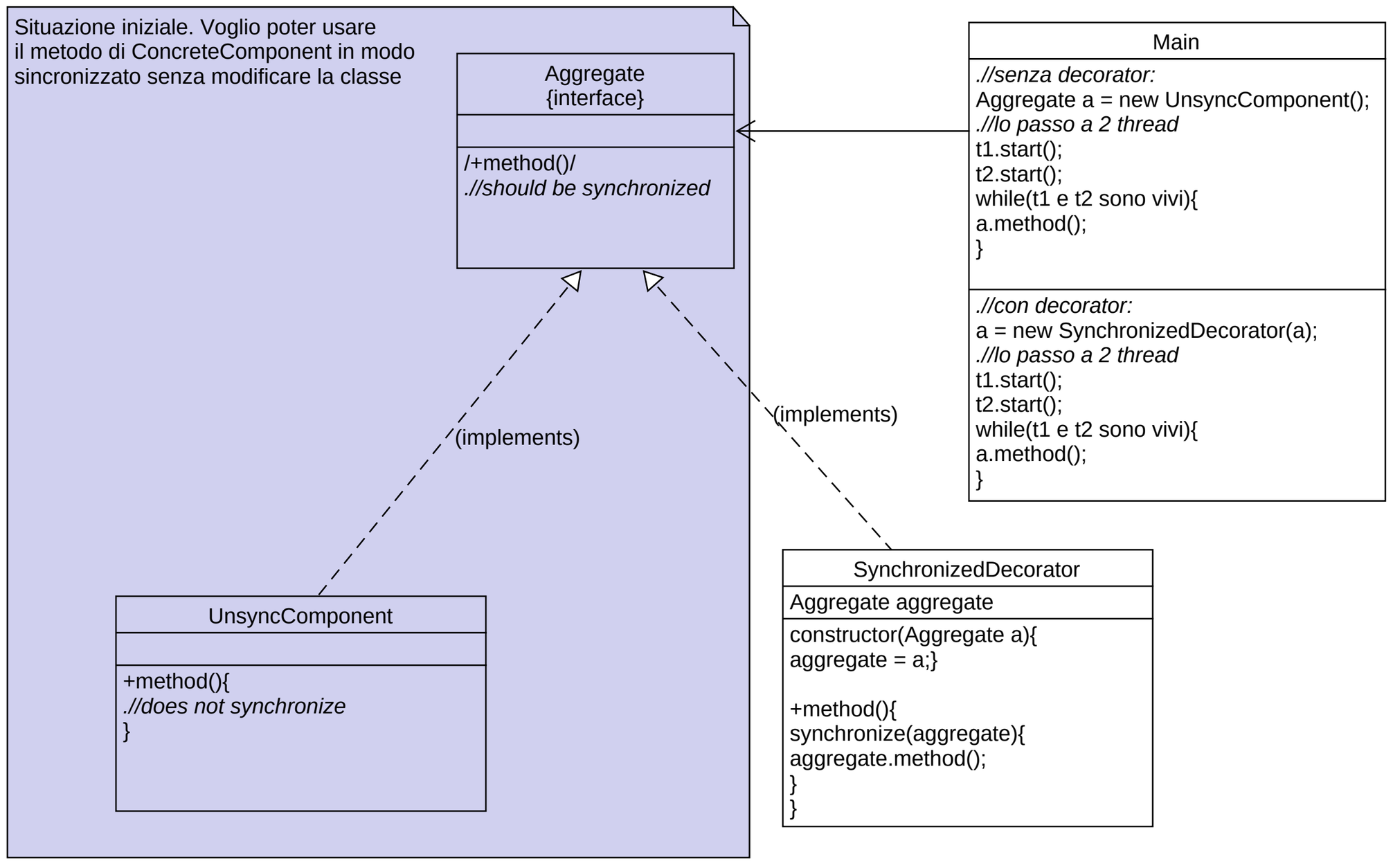
Composite - metodi implementati



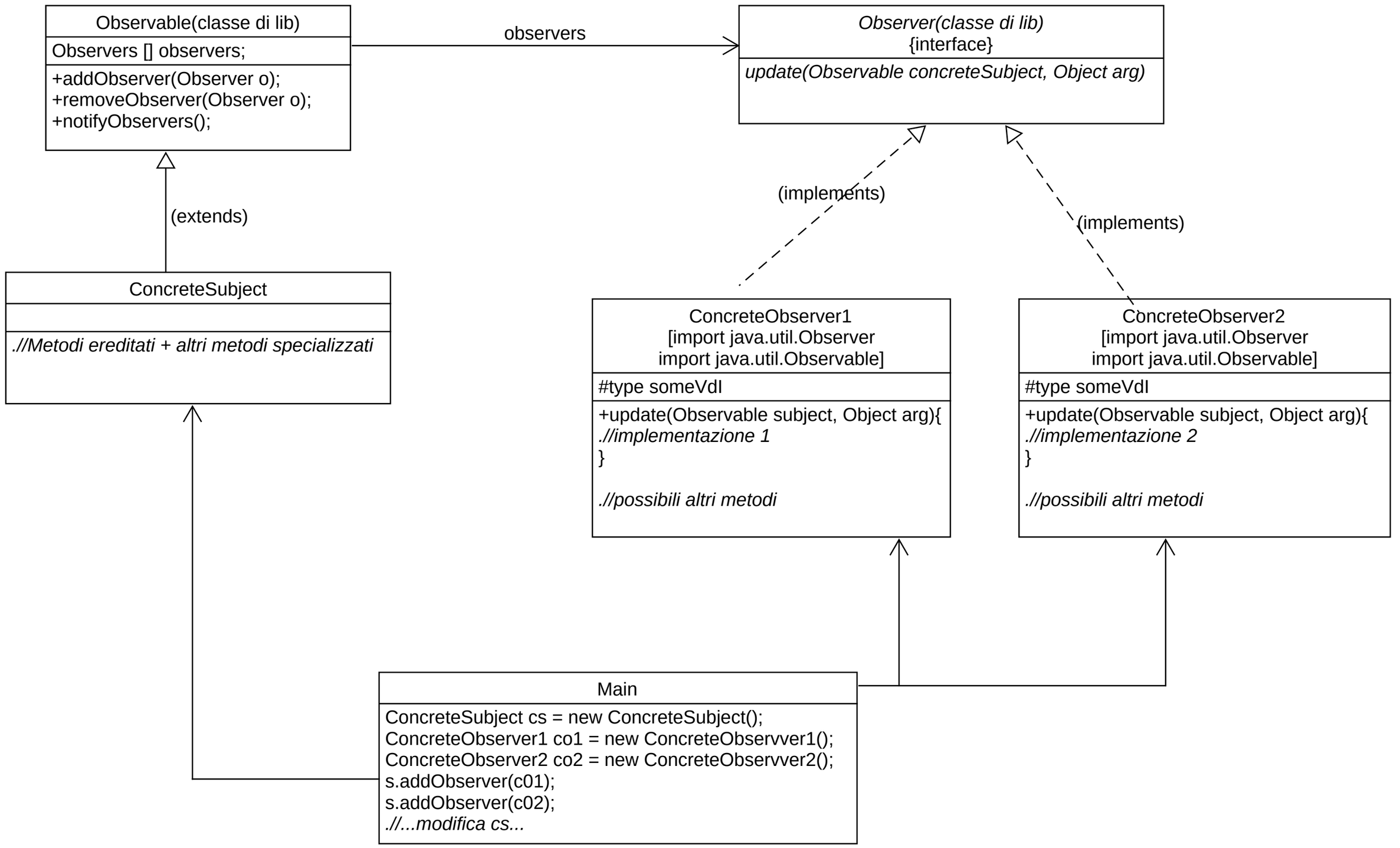
Decorator



Synchronized Decorator



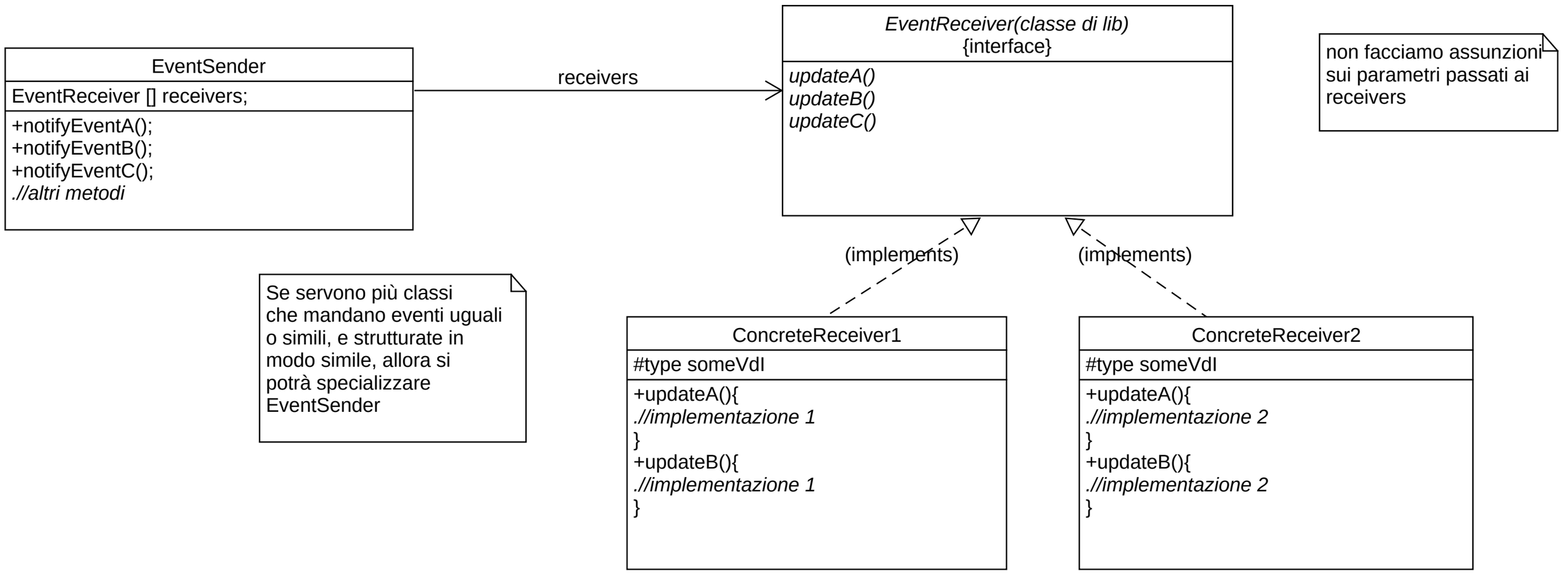
Observer - Observable



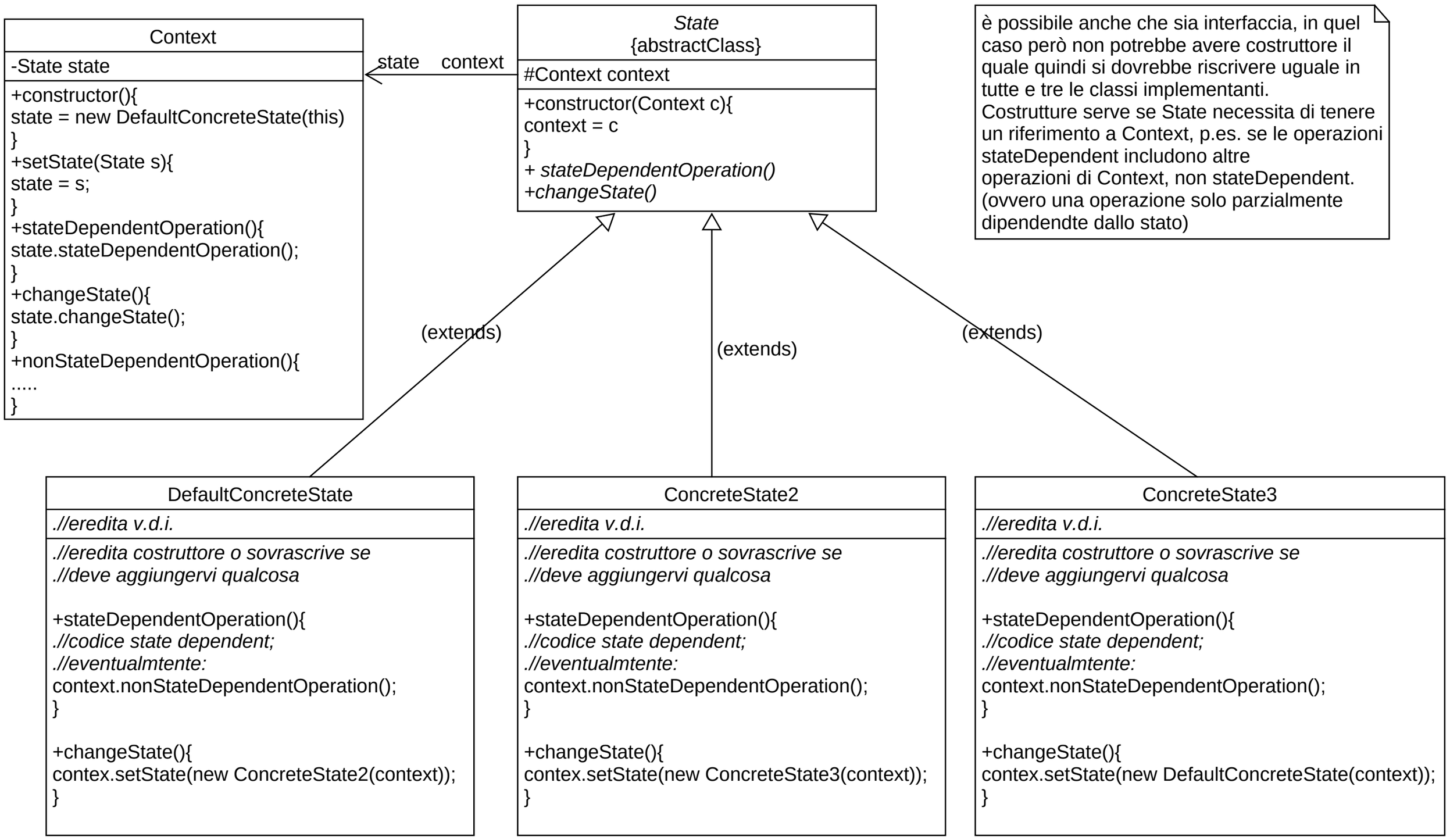
Altro vantaggio: se ho solo un tipo di EventSender, mi risparmio una estensione

Observer - Observable
Variante Event Receiver

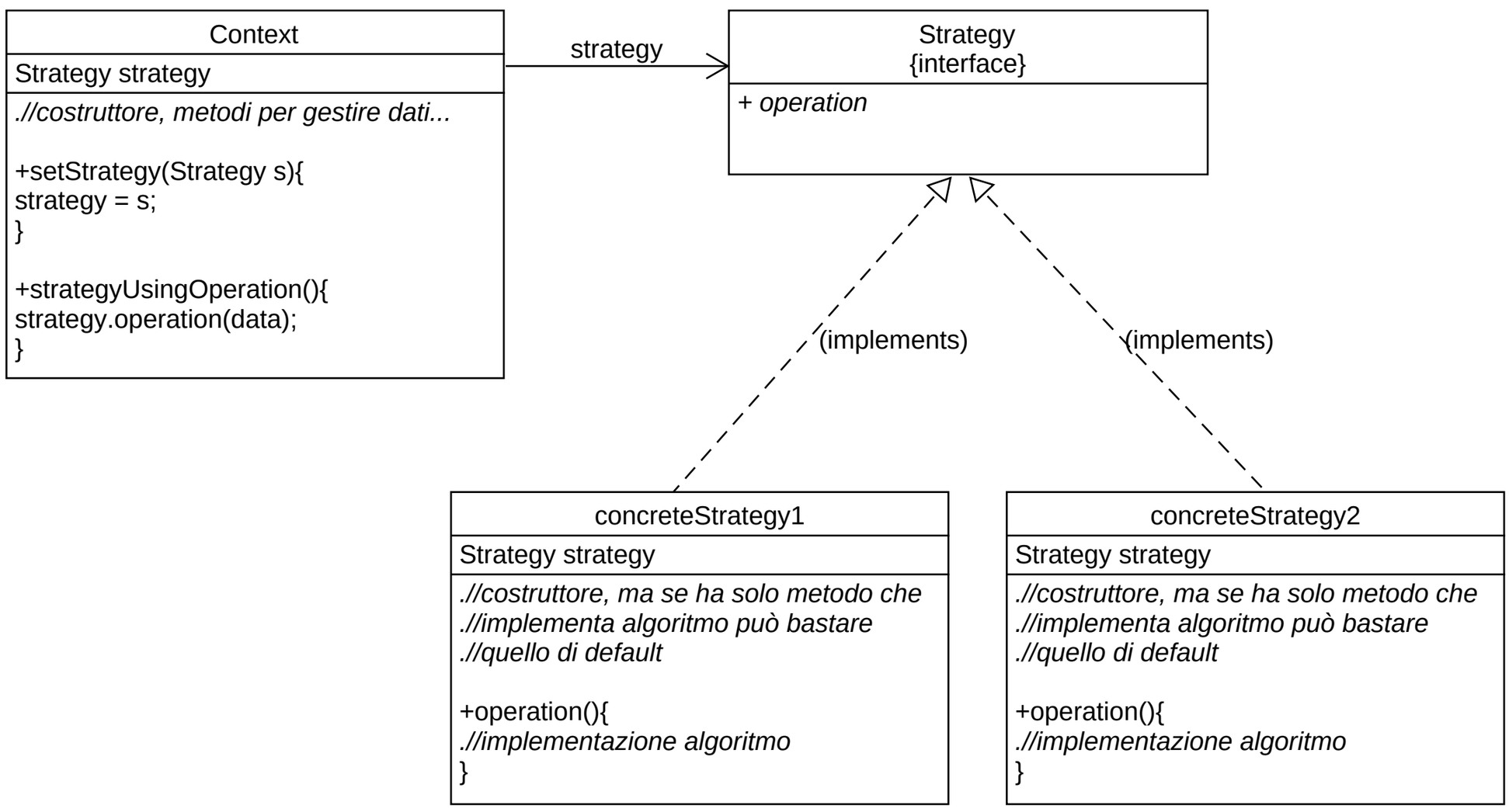
EventSender è un Subject più complesso che ha cambiamenti più specifici e variegati.
EventReceiver è una interfaccia che raccoglie tutte le possibili update



State



Strategy



Visitor versione base

posso aggiungere metodi extra all'implementazione dell'interfaccia, ma non posso chiamarli da fuori senza prima fare downcast

Main

typeDataStructure
(supporta tipi eterogenei)

Visitor
{interface}

+visit(typeA a)
+visit(typeB b)
+visit(typeDataStructure ds)

(implements)

ConcreteVisitor

```
+visit(typeDataStructure ds){
   Iterator iterator = ds.iterator();
   while(iterator.hasNext()){
      Object o = iterator.next();
      if(o instanceof Visitable){
        ((Visitable)o).accept(this);
        } else if (o instanceof typeDataStructure){
        visit((typeDataStructure) o);
      }
   }
}

+visit(typeA a){
.//codice adeguato al tipo a...
}
+visit(typeB b){
.//codice adeguato al tipo b...
}
```

le classi che implementano visitable sono uguali, quindi Visitable potrebbe anche essere classe astratta con metodo accept già implementato e poi ereditato dalle sottoclassi, ma questo farebbe "sprecare" l'estensione

Visitable ("Element")
{interface}

+accept(Visitor v)

(implements)

(implements)

VisitableTypeA
("ConcreteElementA")

```
.//costruttore
+accept(Visitor v){
v.visit(this);
}
```

VisitableTypeB
("ConcreteElementB")

```
.//costruttore
+accept(Visitor v){
v.visit(this);
}
```

difetti di questa versione di visitor:
1)Se voglio aggiungere un nuovo ConcreteElement, devo modificare e ricompilare l'interfaccia Visitor
2)I ConcreteElement devono implementare Visitable
Soluzione: versione con Reflection

Visitor Reflection

ReflectiveVisitor
{interface}

+visit(Object o)

(implements)

ConcreteReflectiveVisitor

```
import java.lang.reflect.Method;
import java.lang.reflect.invocationTargetException;

+visito(Collection collection){
    Iterator iterator = collection.iterator();
    while(iterator.hasNext()){
        Object o = iterator.next();
        visit(o);
     }
}

+visit(typeA a){
.//codice adeguato al tipo a...
}
+visit(typeB b){
.//codice adeguato al tipo b...
}

defaultVisit(Object o){
    if(o instanceof Collection){
        visit((Collection) o);
    } else {
            //comportamento di default
            //per tipo non previsto
    }
}

+visit(Object o){
try {
      Method m = getClass().getMethod("visit", new Class[] {o.getClass});
      m.invoke(this, new Object[] {o});
      }
catch(NoSuchMethodException e){
    defaultVisit(o);
}
catch(IllegalAccessException e){
    defaultVisit(o);
}
catch(InvocationTargetException e){
    defaultVisit(o);
```

se viene invocato visit() su un tipo non previsto, scatta la NoSuchMethodException e viene usata la visita di default.
Attenzione, Collection è una interfaccia quindi non arriverà mai un oggetto con tipo reale Collection. Ma se ne arriva una sua implementazione, per esempio Vector, il ConcreteRelativeVisitor non potrà usare direttamente il metodo visit(Collection c) perché sarebbe downcast.
Per questo motivo userebbe il metodo visit(Object), il quale lancerebbe una NoSuchMethodException per poi chiamare defaultVisit. A questo punto con l'opportuno upcast a Collection potrà essere usato visit(Collection c).