Anno Accademico
2006/2007

Linguaggi e compilatori

Progetto di un linguaggio per la definizione di digrammi UML

Luca Cividini 51360
Fabio Marini 52169
Antonio Riva 52171

Indice generale

1 Introduzione

Il progetto che abbiamo sviluppato si pone come obiettivo la definizione di un linguaggio per la descrizione di diagrammi UML.
UML è un linguaggio di modellazione semi-formale; il diagramma che genera è costituito da elementi grafici, elementi testuali formali, ed elementi di testo libero. Ha una semantica molto precisa e un grande potere descrittivo.
Nella definizione della grammatica si è prestata attenzione al controllo di dichiarazione/uso/non re-dichiarazione delle classi e a controlli di non re-dichiarazione di package.
Il linguaggio definito si ispira molto a Java e alla sua sintassi per la dichiarazione di package, classi e metodi.

2 Capacità di rappresentazione del nostro linguaggio

2.1 Gestione di più diagrammi

Il linguaggio definito è predisposto alla gestione di diversi tipi di diagrammi attraverso la prima definizione:

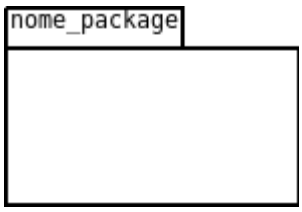
```
diagram uml "nome_diagramma"{ ... }
```

UML è solo una delle possibilità però è l'unica implementata.

2.2 Rappresentazione di un package

Con lo stesso sistema di annidamento tra parentesi graffe è possibile definire una lista di package:

```
package nome_package{ ... }
```



La rappresentazione grafica corrispondente è quella in figura.

2.3 Rappresentazione di una classe/interfaccia

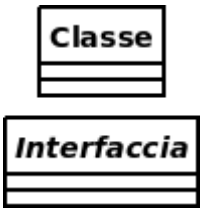
La definizione di classe avviene con il seguente codice:

```
class nome_classe{ ... }
```

Le interfacce invece sono rappresentate così:

```
interface nome_intefaccia{ ... }
```

Come al solito in figura la rappresentazione grafica.



Per la definizione di una classe è necessario rappresentare tre cose:

- relazioni
- attributi
- metodi

Questa operazione viene effettuata all'interno delle parentesi graffe.

2.4 Relazioni

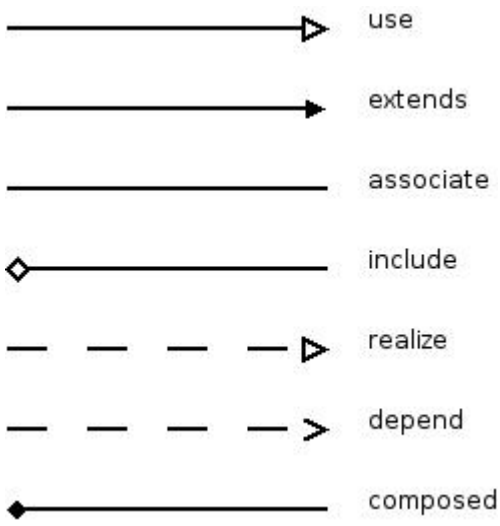
La rappresentazione delle relazioni è effettuata attraverso un codice di questo tipo:

```
relations{ ... }
```

Al suo interno vi saranno tante definizioni di questo tipo:

```
RELATION_TYPE nome_classe,nome_classe(1,label,1)
```

Dove viene indicato il tipo di relazione, i nomi delle classi che sono coinvolte nella relazione e se necessario delle informazioni descrittive della relazione in cui dono indicate le cardinalità o il ruolo ai lati e l'etichetta.



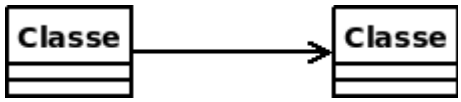
I tipi di relazioni contemplate sono:

- uso
- ereditarietà (estensione)
- associazione
- inclusione
- realizzazione
- dipendenza
- composizione

Ognuna avrà rappresentazione grafiche differenti; verranno ora descritte una per una

2.4.1 Uso

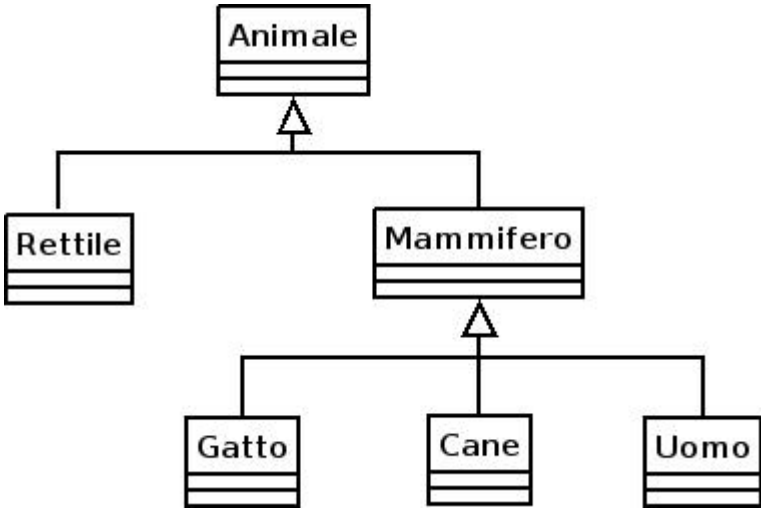
La relazione d'uso o navigabilità fa si che da una classe può essere raggiunta un'altra classe: viene rappresentata attraverso una freccia semplice.



2.4.2 Ereditarietà (estensione)

Una classe figlia (o sottoclasse) può ereditare gli attributi e le operazioni da un'altra classe (che viene definita classe padre o super classe) che sarà sempre più generica della classe figlia.

Nella generalizzazione, una classe figlia può rappresentare un valido sostituto della classe padre.

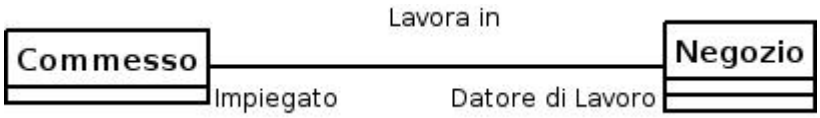


In UML l'ereditarietà viene rappresentata con una linea che connette la classe padre alla classe discendente e dalla parte della classe padre si inserisce un triangolo (una freccia).

2.4.3 Associazione

Quando più classi sono connesse l'una con l'altra da un punto di vista concettuale, tale connessione viene denominata associazione.

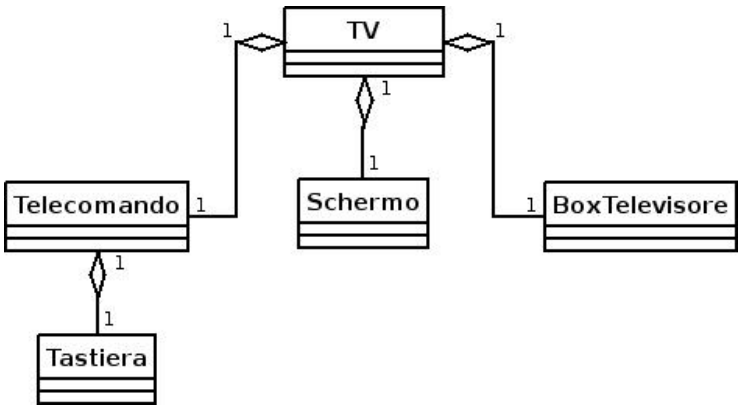
Graficamente, un'associazione viene rappresentata con una linea che connette due classi, con il nome dell'associazione appena sopra la linea stessa.



Quando una classe si associa con un'altra, ognuna di esse gioca un ruolo all'interno dell'associazione. E' possibile mostrare questi ruoli sul diagramma, scrivendoli vicino la linea orizzontale dalla parte della classe che svolge un determinato ruolo, come si vede nella figura precedente.

2.4.4 Inclusione (aggregazione)

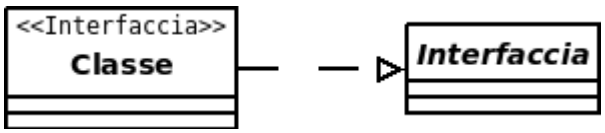
Una classe può rappresentare il risultato di un insieme di altre classi che la compongono. Le classi che costituiscono i componenti e la classe finale sono in una relazione particolare del tipo: parte – intero. Un'aggregazione è rappresentata come una gerarchia in cui “l'intero” si trova in cima e i componenti (“parte”) al di sotto. Una linea unisce “l'intero” ad un componente con un rombo raffigurato sulla linea stessa vicino “all'intero”.



Graficamente è rappresentata come le frecce in figura.

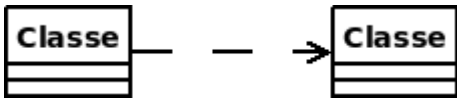
2.4.5 Realizzazione (classe/interfaccia)

Un'interfaccia è un insieme di operazioni che specifica alcuni aspetti del comportamento di una classe; una interfaccia rappresenta un insieme di operazioni che una classe offre ad altre classi. Per modellare un'interfaccia si utilizza lo stesso modo utilizzato per modellare una classe, con un rettangolo. La differenza consiste nel fatto che un'interfaccia non ha attributi ma soltanto operazioni (metodi). Nella rappresentazione grafica oltre la solita freccia viene anche esplicitato all'interno della classe il nome dell'interfaccia implementata.



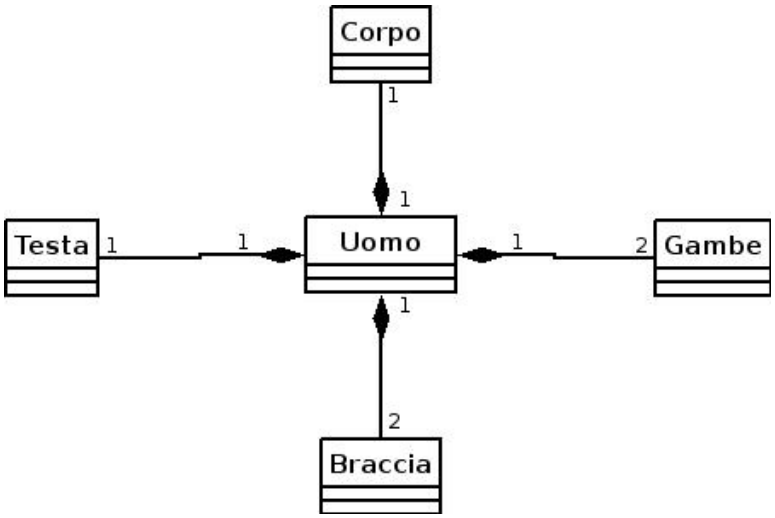
2.4.6 Dipendenza

La relazione di dipendenza viene definita con la semplice freccia ma tratteggiata.



2.4.7 Composizione

É un tipo di più forte di aggregazione. Ogni componente in una composizione può appartenere soltanto ad un “intero”. Es: Una persona ha una testa, un corpo, due braccia e due gambe.
Graficamente è rappresentata come le frecce in figura.

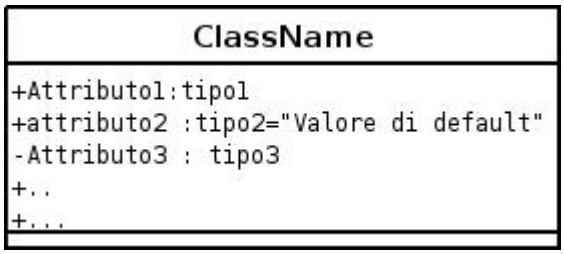


2.5 Attributi

Un attributo rappresenta una proprietà di una classe. Esso descrive un insieme di valori che la proprietà può avere quando vengono istanziati oggetti di quella determinata classe. Una classe può avere zero o più attributi.
La lista degli attributi di una classe viene separata graficamente dal nome della classe a cui appartiene tramite una linea orizzontale.
La convenzione UML definisce che gli attributi siano preceduti da:

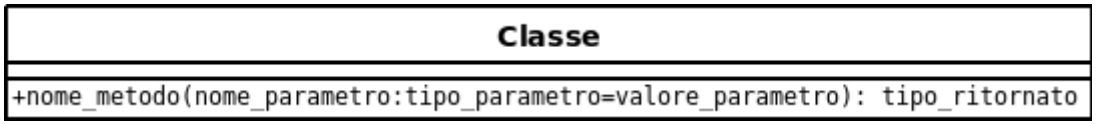
- + nel caso di attributo pubblico
- # nel caso di attributo protetto
- - nel caso di attributo privato

Oltre al nome può essere definito il tipo e il valore dell'attributo come rappresentato in figura.
Il codice per rappresentare un metodo attributo completo è questo:
VISIBILITY TYPE nome_attributo=value



2.6 I metodi

I metodi sono le operazioni che la classe può svolgere. La sintassi è più complessa perché devono essere definiti anche i parametri dell'operazione. Graficamente questi parametri vengono disegnati sotto gli attributi in un rettangolo separato e la successione dei vari elementi è rappresentata come in figura; vari parametri del metodo sono rappresentati separati da virgole.



UML permette anche di definire la direzione del metodo, per semplicità non è stata implementata.

Come per gli attributi vale la solita regole per la visibilità

- + nel caso di attributo pubblico
- # nel caso di attributo protetto
- - nel caso di attributo privato

Il codice per rappresentare un metodo è questo

```
VISIBILITY TYPE nome_metodo{
    TYPE nome_arg1=value,
    TYPE nome_arg1
}
```

2.7 Commenti

Un po' dappertutto è possibile inserire commenti. Un commento è identificato da una qualsiasi linea preceduta da cancelletto.

3 Vista d'insieme del linguaggio

```
diagram uml "nome_diagramma"{
     package nome_package{
                #
          CLASS_TYPE nome_classe{
                relations{
                     #
                     RELATION_TYPE nome_classe,nome_classe(1,label,1)
                }
                #
                VISIBILITY TYPE nome_attributo=value
                #
                VISIBILITY TYPE nome_metodo{
                     #
                     TYPE nome_arg1=value,

                     #
                     TYPE nome_arg1
                }
          }

          #
          CLASS_TYPE nome_classe{
                #
                VISIBILITY TYPE  nome_attributo=value
                #
                VISIBILITY TYPE nome_metodo{
                     #
                     TYPE nome_arg1=value,
                     #
                     TYPE nome_arg1
                }
          }
     }
}
```

4 Grammatica e regole semantiche

4.1 I campi

- **clist:** lista ereditata delle classi definite precedentemente
- **rclist:** lista sintetizzata delle classi definite precedentemente
- **plist:** lista ereditata dei package definiti precedentemente
- **rplist:** lista sintetizzata dei package definiti precedentemente
- **ulist:** lista ereditata delle classi usate all'interno delle relazioni
- **rulist:** lista sintetizzata delle classi usate all'interno delle relazioni
- **error:** attributo booleano contenete lo stato dell'errore 0: nessun errore 1: errore
- **diff:** attrubuto booleano contenete il risultato tra il confronto della lista delle classi usate con la lista delle classi dichiarate

4.2 Le funzioni semantiche

- **or():** corrisponde all'OR logico, usato per effettuare un unione del flag di errore
- **exist():** controlla che una nome di package/classe non sia presente nella lista di package/classi già dichiarate al momento della definizione
- **in_array():** controlla che il contenuto del primo array sia un sottoinsieme del secondo array, restituisce 0 se il primo è sottoinsieme del secondo altrimenti 1; questo valore di ritorno di per se contrario alla normale convenzione è necessario per poter poi essere confrontato con il flag di errore!
- **push():** inserisce il valore passato come primo parametro nell'array passato come secondo parametro

4.3 Le produzioni

S → diagram DIAGRAM_TYPE diagram_name { PACKAGES_DEFINITIONS }

```
//gestione dichiarazione classi
S[0].clist := null
PACKAGES_DEFINITIONS[5].clist := S[0].clist
S[0].rclist := PACKAGES_DEFINITIONS[5].rclist

//gestione uso classi
S[0].ulist := null
PACKAGES_DEFINITIONS[5].ulist := S[0].ulist
S[0].rulist := PACKAGES_DEFINITIONS[5].rulist

//gestione dichiarazione package
S[0].plist := null
PACKAGES_DEFINITIONS[5].plist := S[0].plist
S[0].rplist := PACKAGES_DEFINITIONS[5].rplist
```

```
    //confronto finale delle liste d'uso con quelle di dichiarazione
    S[0].diff := in_array(S[0].rulist,S[0].rclist)
    S[0].error := or(S[0].diff,PACKAGES_DEFINITIONS[5].error)
```

DIAGRAM_TYPE → uml

PACKAGES_DEFINITIONS → PACKAGE_DEFINITION PACKAGES_DEFINITIONS

```
    //gestione dichiarazione classi
    PACKAGE_DEFINITION[1].clist := PACKAGES_DEFINITIONS[0].clist
    PACKAGES_DEFINITIONS[2].clist:= PACKAGE_DEFINITION[1].rclist
    PACKAGES_DEFINITIONS[0].rclist:=PACKAGES_DEFINITIONS[2].rclist
    PACKAGES_DEFINITIONS[0].error:=
    or(PACKAGE_DEFINITION[1].error,PACKAGES_DEFINITIONS[2].error)

    //gestione uso classi
    PACKAGE_DEFINITION[1].ulist := PACKAGES_DEFINITIONS[0].ulist
    PACKAGES_DEFINITIONS[2].ulist:= PACKAGE_DEFINITION[1].rulist
    PACKAGES_DEFINITIONS[0].rulist:=PACKAGES_DEFINITIONS[2].rulist

    //gestione dichiarazione package
    PACKAGE_DEFINITION[1].plist := PACKAGES_DEFINITIONS[0].plist
    PACKAGES_DEFINITIONS[2].plist:= PACKAGE_DEFINITION[1].rplist
    PACKAGES_DEFINITIONS[0].rplist:= PACKAGES_DEFINITIONS[2].rplist
    PACKAGES_DEFINITIONS[0].error:=
    or(PACKAGE_DEFINITION[1].error,PACKAGES_DEFINITIONS[2].error)
```

PACKAGE_DEFINITION → package package_name { CLASSES_DEFINITIONS }

```
    //gestione dichiarazione classi
    CLASSES_DEFINITIONS[4].clist:=PACKAGE_DEFINITION[0].clist
    PACKAGE_DEFINITION[0].rclist:=CLASSES_DEFINITIONS[4].rclist
    PACKAGE_DEFINITION[0].error:=CLASSES_DEFINITIONS[4].error

    //gestione uso classi
    CLASSES_DEFINITIONS[4].ulist:=PACKAGE_DEFINITION[0].ulist
    PACKAGE_DEFINITION[0].rulist:=CLASSES_DEFINITIONS[4].rulist

    //gestione dichiarazione package
    PACKAGE_DEFINITION[0].errore:=
    exist(package_name[2].value,PACKAGE_DEFINITION[0].plist)
    PACKAGE_DEFINITION[0].rplist:=
    push(package_name[2].value,PACKAGE_DEFINITION[0].plist)
```

PACKAGES_DEFINITIONS → ϵ

```
    //gestione dichiarazione classi
    PACKAGES_DEFINITIONS[0].rclist :=  PACKAGES_DEFINITIONS[0].clist
    PACKAGES_DEFINITIONS[0].error:=false

    //gestione uso classi
    PACKAGES_DEFINITIONS[0].rulist :=  PACKAGES_DEFINITIONS[0].ulist
```

```
    //gestione dichiarazione package
    PACKAGES_DEFINITIONS[0].rplist :=  PACKAGES_DEFINITIONS[0].plist
    PACKAGES_DEFINITIONS[2].error :=  false

OPTIONAL_COMMENT → ϵ
OPTIONAL_COMMENT → COMMENT
COMMENT → comment

CLASSES_DEFINITIONS → CLASS_DEFINITION CLASSES_DEFINITIONS
    //gestione dichiarazione classi
    CLASS_DEFINITION[1].clist:=CLASSES_DEFINITIONS[0].clist
    CLASSES_DEFINITIONS[2].clist:=CLASS_DEFINITION[1].rclist
    CLASSES_DEFINITIONS[0].rclist:=CLASSES_DEFINITIONS[2].rclist
    CLASSES_DEFINITIONS[0].error:=
    or(CLASS_DEFINITION[1].error,CLASSES_DEFINITIONS[2].error)

    //gestione uso classi
    CLASS_DEFINITION[1].ulist:=CLASSES_DEFINITIONS[0].ulist
    CLASSES_DEFINITIONS[2].ulist:=CLASS_DEFINITION[1].rulist
    CLASSES_DEFINITIONS[0].rulist:=CLASSES_DEFINITIONS[2].rulist

CLASSES_DEFINITIONS → ϵ
    //gestione dichiarazione classi
    CLASSES_DEFINITIONS[0].rclist:=CLASSES_DEFINITIONS[0].clist
    CLASSES_DEFINITIONS[0].error:=false

    //gestione uso classi
    CLASSES_DEFINITIONS[0].rulist:=CLASSES_DEFINITIONS[0].ulist

CLASS_DEFINITION → OPTIONAL_COMMENT VISIBILITY CLASS_TYPE class_name
{ RELATIONS ATTRIBUTES METHODS }
    //gestione dichiarazione classi
    CLASS_DEFINITION[0].error:=
    exist(class_name[4].value,CLASS_DEFINITION[0].clist)
    CLASS_DEFINITION[0].rclist:=
    push(class_name[4].value,CLASS_DEFINITION[0].clist)

    //gestione uso classi
    RELATIONS[6].ulist := CLASS_DEFINITION[0].ulist
    CLASS_DEFINITION[0].rulist := RELATIONS[6].rulist

RELATIONS → relations { RELATIONS_DEFINITIONS }
    //gestione uso classi
    RELATIONS_DEFINITIONS[3].ulist := RELATIONS[0].ulist
    RELATIONS[0].rulist := RELATIONS_DEFINITIONS[3].rulist


RELATIONS_DEFINITIONS → RELATION_DEFINITION RELATIONS_DEFINITIONS
    //gestione uso classi
```

```
    RELATION_DEFINITION[1].ulist := RELATIONS_DEFINITIONS[0].ulist
    RELATIONS_DEFINITIONS[2].ulist := RELATION_DEFINITION[1].rulist
    RELATIONS_DEFINITIONS[0].rulist :=
    RELATIONS_DEFINITIONS[2].rulist

RELATIONS_DEFINITIONS → ϵ
    //gestione uso classi
    RELATIONS_DEFINITIONS[0].rulist :=
    RELATIONS_DEFINITIONS[0].ulist

RELATION_DEFINITION → OPTIONAL_COMMENT RELATION_TYPE class_name
RELATION_CARDINALITY CLASSES_LIST_WITH_CARDINALITY
    //gestione uso classi
    CLASSES_LIST_WITH_CARDINALITY[5].ulist :=
    push(class_name[4].value,RELATION_DEFINITION[0].ulist)
    RELATION_DEFINITION[0].rulist :=
    CLASSES_LIST_WITH_CARDINALITY[5].rulist

RELATION_TYPE → use
RELATION_TYPE → extend
RELATION_TYPE → associate
RELATION_TYPE → include
RELATION_TYPE → composed
RELATION_TYPE → realize
RELATION_TYPE → depend

CLASSES_LIST_WITH_CARDINALITY → , class_name RELATION_CARDINALITY
CLASSES_LIST_WITH_CARDINALITY
    //gestione uso classi
    CLASSES_LIST_WITH_CARDINALITY[4].ulist :=
    push(class_name[2].value,CLASSES_LIST_WITH_CARDINALITY[0].ulist)
    CLASSES_LIST_WITH_CARDINALITY[0].rulist :=
    CLASSES_LIST_WITH_CARDINALITY[4].rulist

CLASSES_LIST_WITH_CARDINALITY → ϵ
    //gestione uso classi
    CLASSES_LIST_WITH_CARDINALITY[0].rulist :=
    CLASSES_LIST_WITH_CARDINALITY[0].ulist

RELATION_CARDINALITY → ϵ
RELATION_CARDINALITY → (CARDINALITY, CARDINALITY, CARDINALITY)

CARDINALITY → string
CARDINALITY → ϵ

ATTRIBUTES → ATTRIBUTE ATTRIBUTES
ATTRIBUTES → ϵ
ATTRIBUTE → COMMENT VISIBILITY ATTRIBUTE_TYPE attribute_name
DEFAULT_VALUE
```

DEFAULT_VALUE → ∈
DEFAULT_VALUE → **= value**

METHODS → **METHOD METHODS**
METHODS → ∈
METHOD → **COMMENT VISIBILITY METHOD_TYPE method_name { METHOD_ARGS }**
METHOD_ARGS → **METHOD_ARG METHOD_ARGS_ITERATION**
METHOD_ARGS_ITERATION → **, METHOD_ARG METHOD_ARGS_ITERATION**
METHOD_ARGS_ITERATION → ∈
METHOD_ARG → **COMMENT ARG_TYPE arg_name DEFAULT_VALUE**

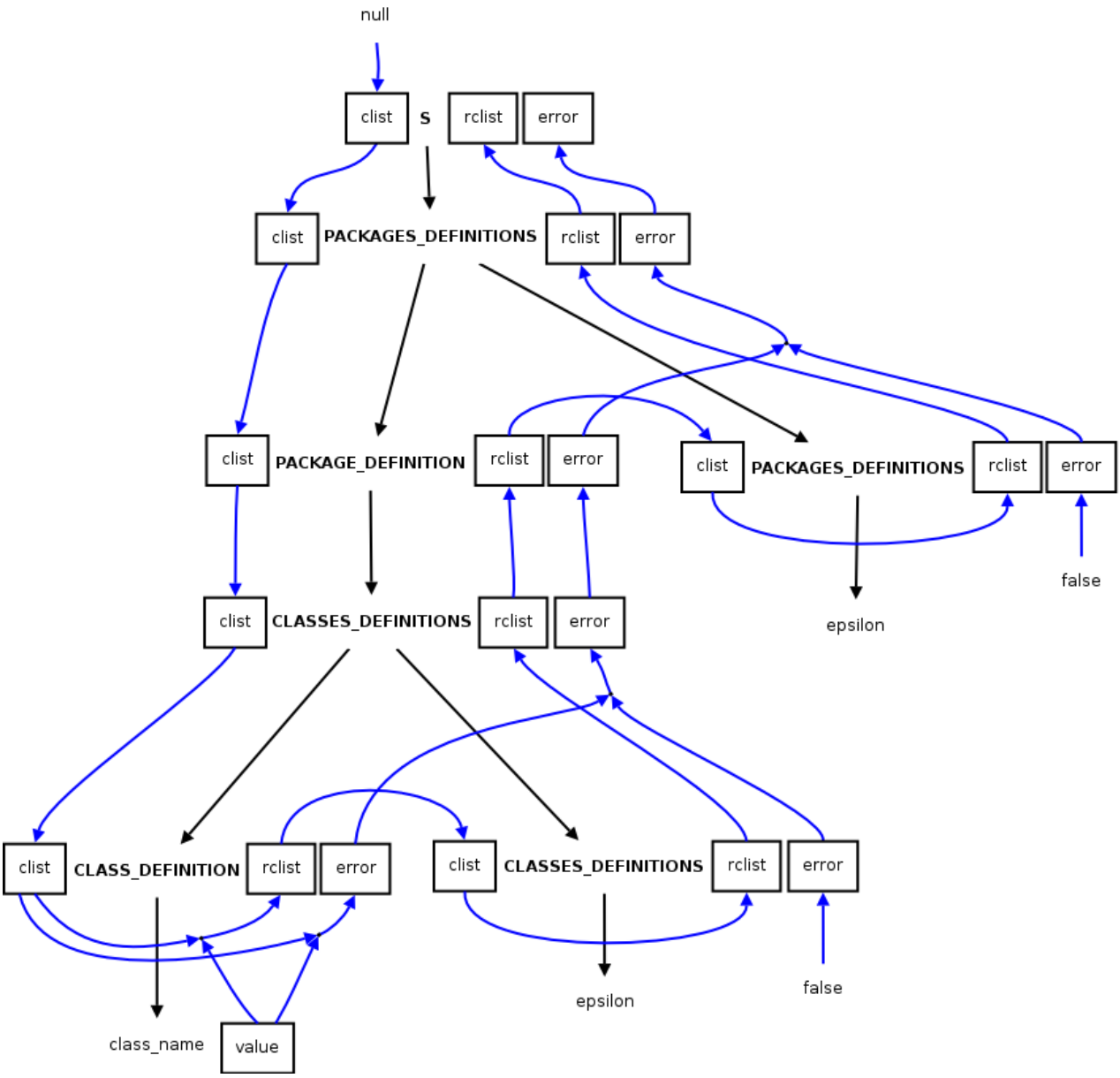
CLASS_TYPE → **interface**
CLASS_TYPE → **class**

METHOD_TYPE → **TYPE**
ARG_TYPE → **TYPE**
ATTRIBUTE_TYPE → **TYPE**
TYPE → **string**
TYPE → ∈

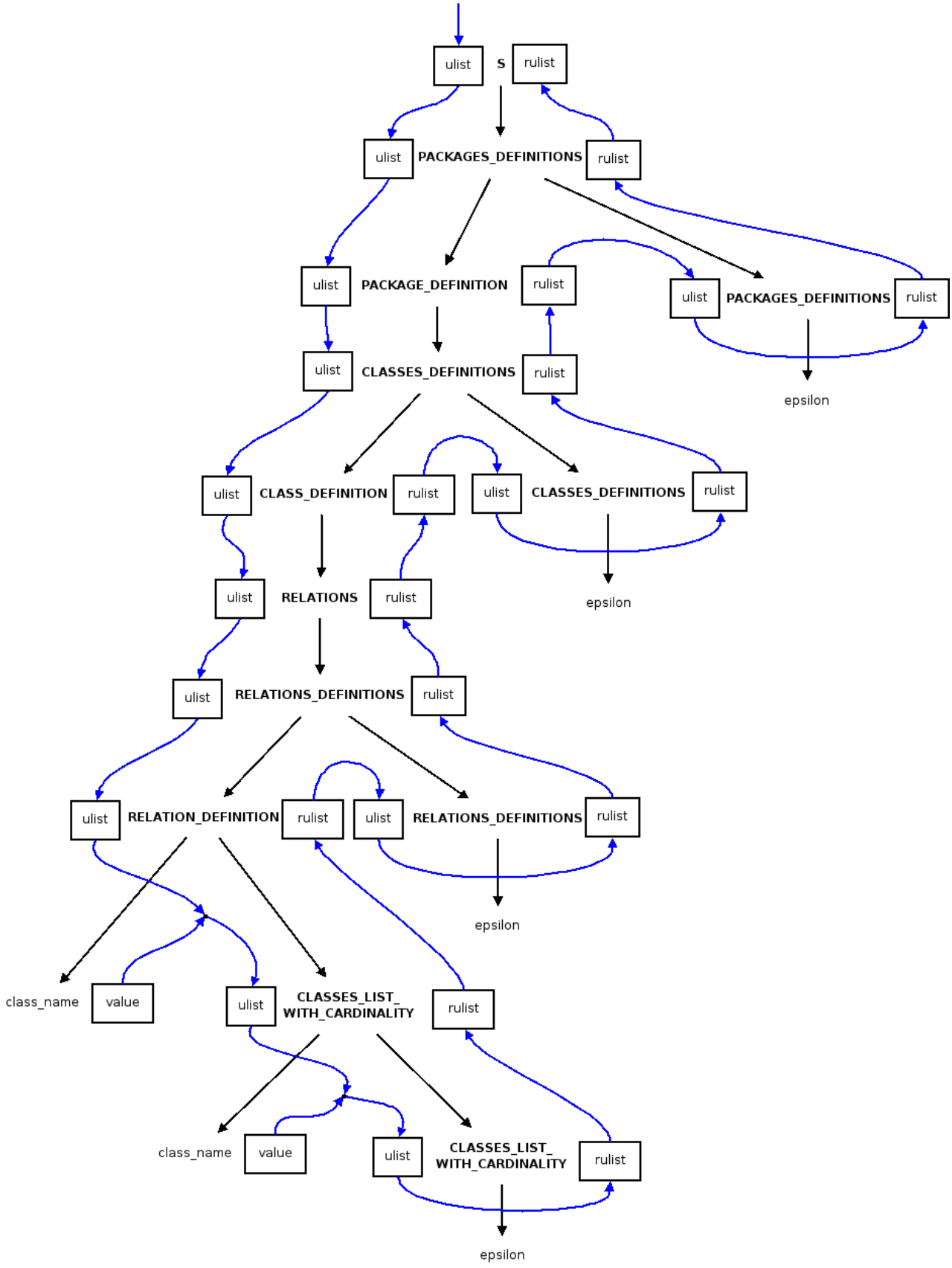
VISIBILITY → **private**
VISIBILITY → **public**
VISIBILITY → **string**
VISIBILITY → ∈

5 Diagrammi

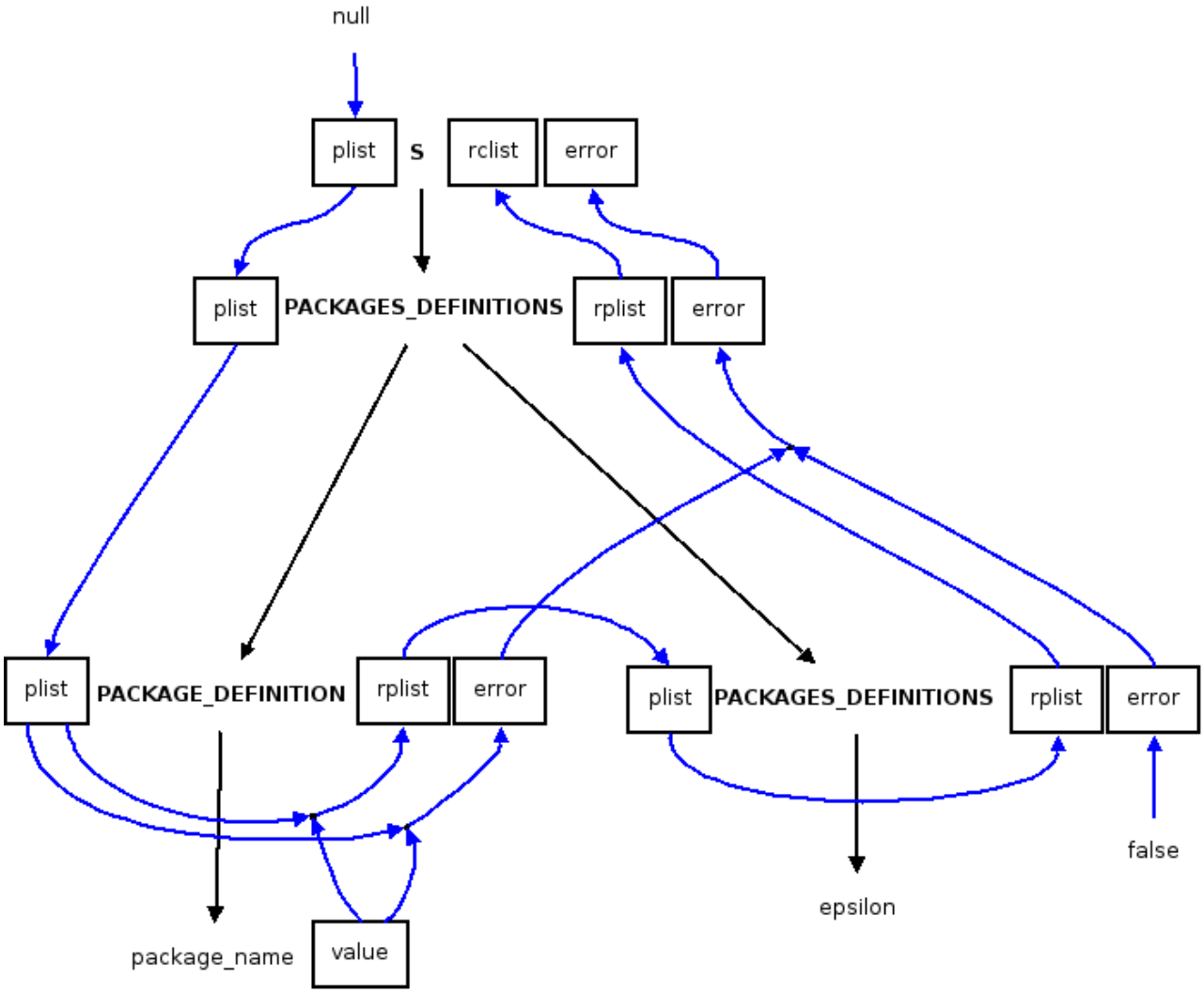
5.1 Creazione della lista di classi dichiarate e controllo di non re-dichiarazione di una classe



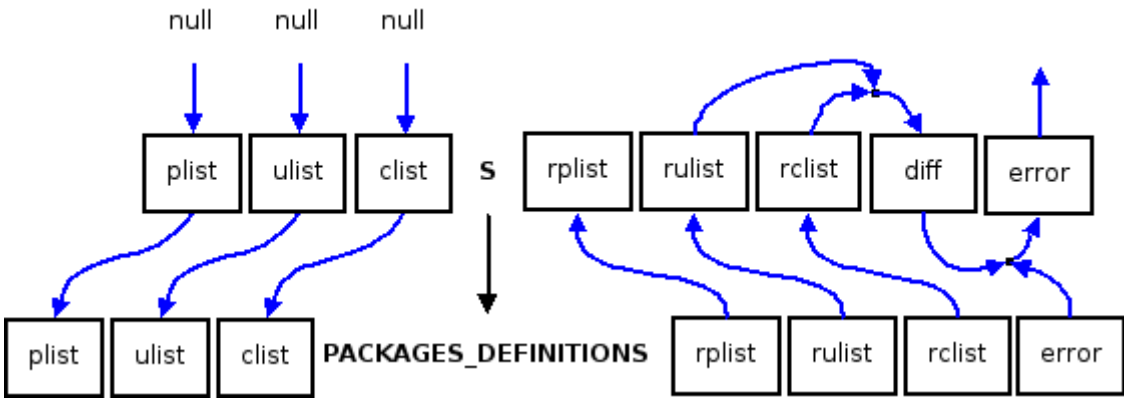
5.2 Creazione della lista di classi usate



5.3 Controllo della non re-dichiarazione di un package

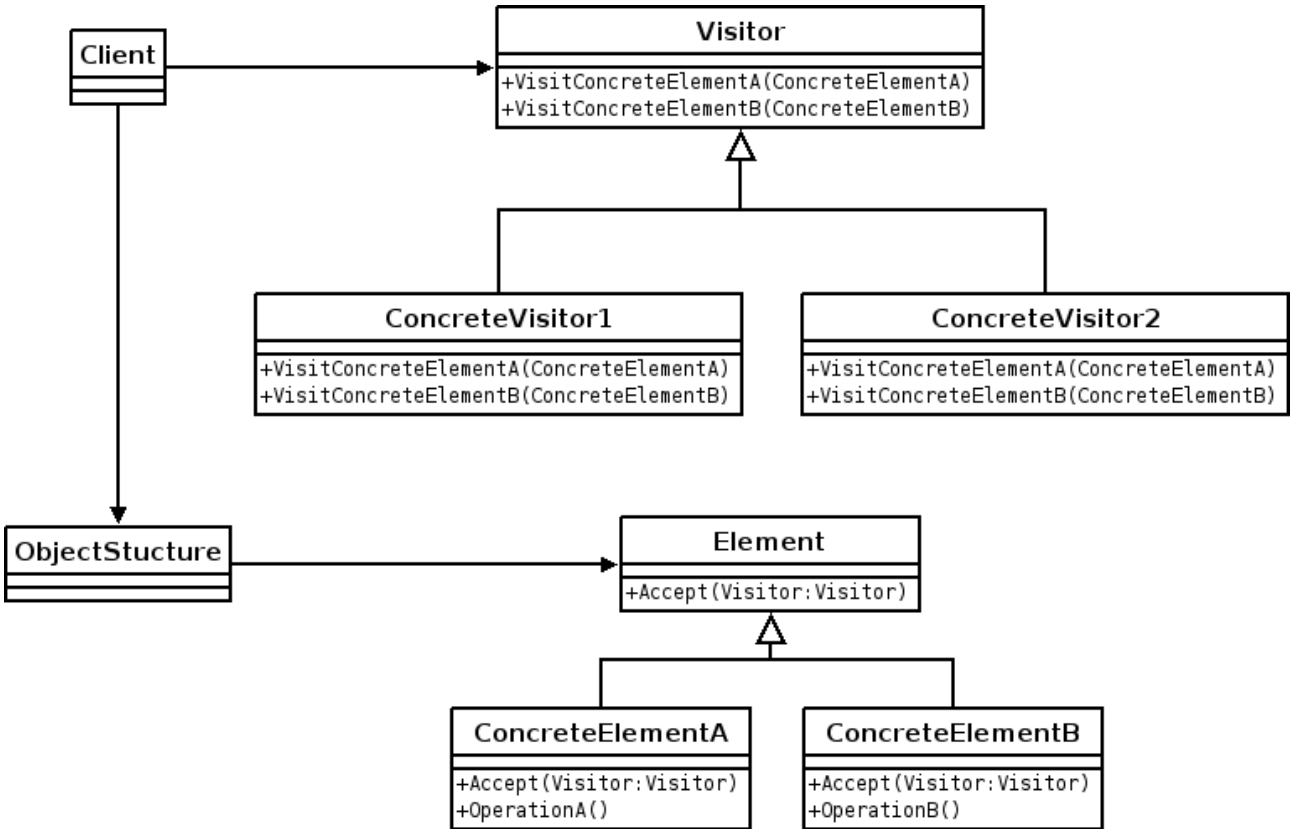


5.4 Confronto finale delle liste d'uso con quelle di dichiarazione



6 Esempi

6.1 Visitor Pattern

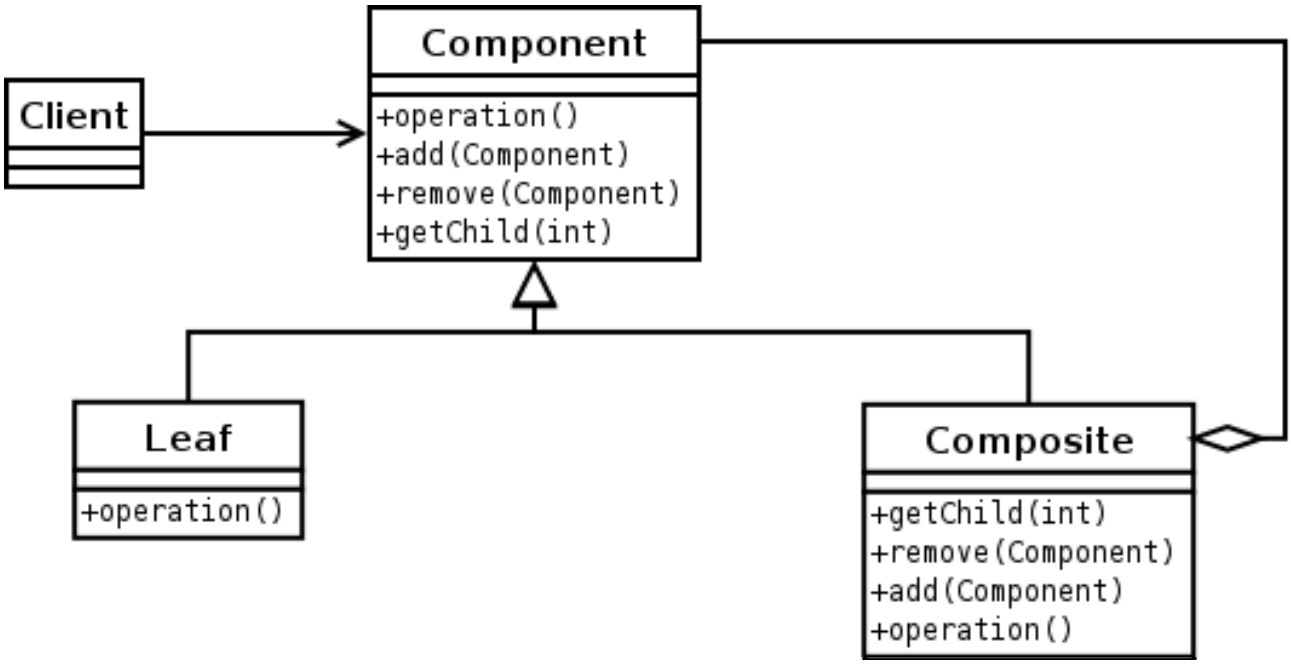



```
diagram uml "visitor pattern"{
    class Client{
        relations{
            use Visitor,ObjectStructure
        }
    }
    class Visitor{
        public VisitConcreteElementA{
            ConcreteElementA
        }
        public VisitConcreteElementB{
            ConcreteElementB
        }
    }
    class ConcreteVisitor1{
```

```
        relations{
            extend Visitor
        }
        public VisitConcreteElementA{
            ConcreteElementA
        }
        public VisitConcreteElementB{
            ConcreteElementB
        }
    }
    class ConcreteVisitor2{
        relations{
            extend Visitor
        }
        public VisitConcreteElementA{
            ConcreteElementA
        }
        public VisitConcreteElementB{
            ConcreteElementB
        }
    }
    class ObjectStructure{
        relations{
            use Element
        }
    }
    class Element{
        public Accept{
            Visitor Visitor
        }
    }
    class ConcreteElementA{
        relations{
            extend Element
        }
        public Accept{
```

```
                Visitor Visitor
            }
            public OperationA{
            }
        }
        class ConcreteElementB{
            relations{
                extend Element
            }
            public Accept{
                Visitor Visitor
            }
            public OperationB{
            }
        }
}
```

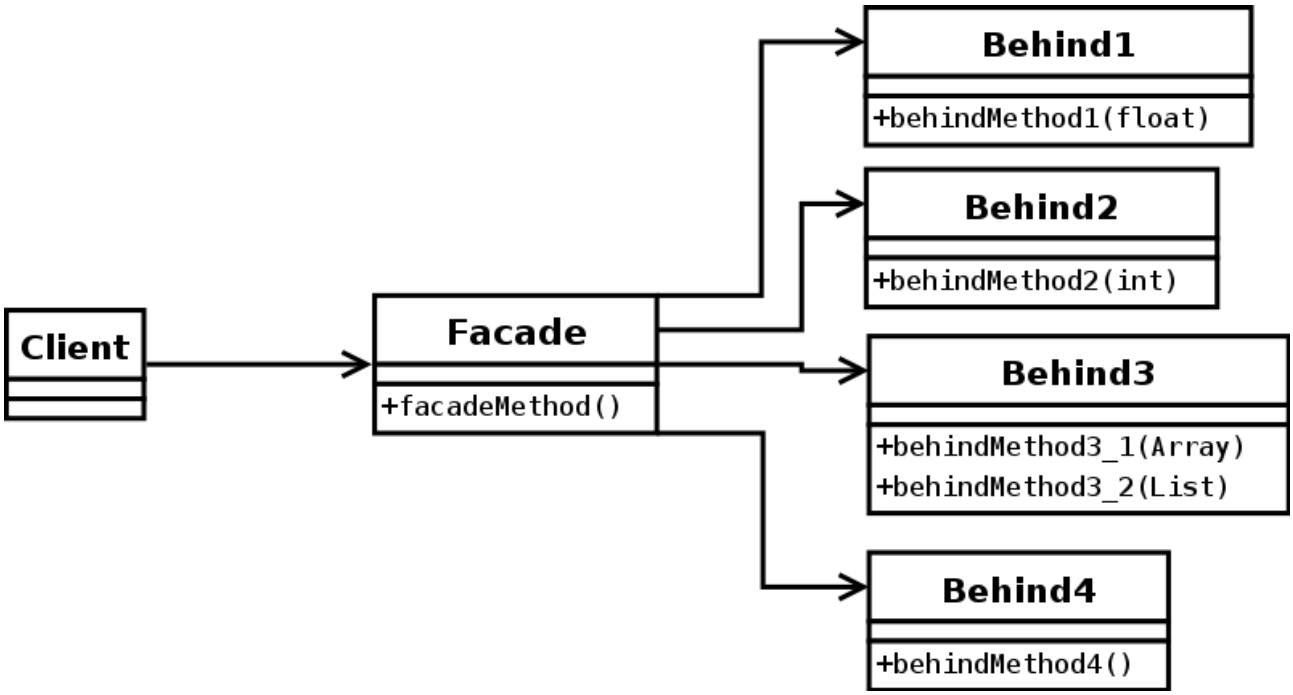
6.2 Composite Pattern




```
diagram uml “composite pattern” {
    class Client{
        relations{
            use Component
        }
    }
    interface Component{
        public operation{
        }
        public add{
            Component
        }
        public remove{
            Component
        }
        public getChild{
            int
        }
    }
```

```
    class Leaf{
          relations{
               implement Component
          }
          public operation{
          }
    }
    class Composite{
          relations{
               implement Component
          }
          public operation{
          }
          public add{
               Component
          }
          public remove{
               Component
          }
          public getChild{
               int
          }
    }
}
```

6.3 Façade Pattern




```
diagram uml “facade pattern”{
    class Client{
        relations{
            use Facade
        }
    }
    class Facade{
        relations{
            use Behind1, Behind2, Behind3, Behind4
        }
        public facadeMethod{
        }
    }
    class Behind1{
        public behindMethod1{
            float
        }
    }
```

```
    class Behind2{
          public behindMethod2{
                int
          }
    }
    class Behind3{
          public behindMethod3_1{
                Array
          }
          public behindMethod3_2{
                List
          }
    }
    class Behind4{
          public behindMethod1{
          }
    }
}
```