Programmazione a Oggetti
Modulo B

Lezione 19

Dott. Alessandro Roncato

09/04/2013

Riassunto

- Product Trader => Plug-in
- Service Manager

Oggi

- Type Object
- Manager
- Indirection
- Adapter

Type Object

Un altro problema ricorrente è quello della gestione di “sottotipi” illimitati.

- Specie di animali
- Tipi di Prodotti
- Libri/Istanze di libri
- …

Type Object

Altro problema frequente è quello della gestione di categorie e sottocategorie in una gerarchia illimitata.

- Negozio: Elettrodomestici/Telfoni/ Cellulari/Smartphone/Android/ Schermo 3,2/
- Annunci: Veicoli/Auto/Berlina
-

Type Object

Anche se la P.O dovrebbe suggerire di adottare il polimorfismo, dato che il numero di classi è illimitato non possiamo (anche pensando di usare Product Trader) implementarle come sottoclassi.

Type Object

Esempio: un'applicazione per i negozi di animali non potrà implementare tutte le specie di animali possibili con una sottoclasse di animale. Altrimenti se capita una nuova specie di animale non prevista nell'applicazione, bisognerebbe scrivere il codice della stessa.

(serve veramente polimorfismo?)

Polimorfismo

Non è possibile applicare il polimorfismo perché richiederebbe di implementare nuove classi e ricompilare l'applicazione (o compilare solo le nuove classi in un file jar) per ogni specie di animale da aggiungere.

In ogni caso il polimorfismo serve solo se il comportamento di alcuni metodi cambia in base al tipo.

Polimorfismo

In alcuni casi, anche se è possibile applicare il polimorfismo se il comportamento delle sottoclassi non cambia, può essere conveniente applicare Type Object.

Type Object

Esempio

Equivalente con Polimorfismo

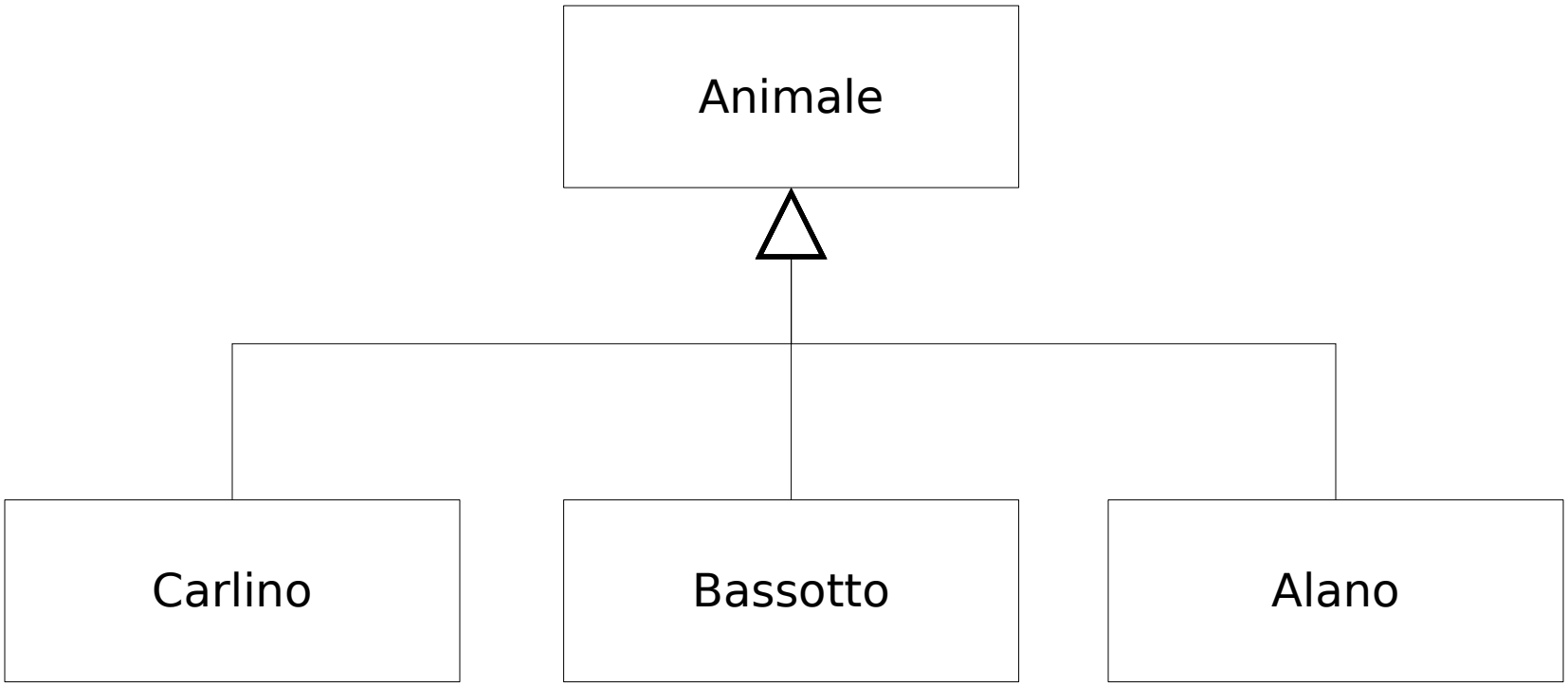
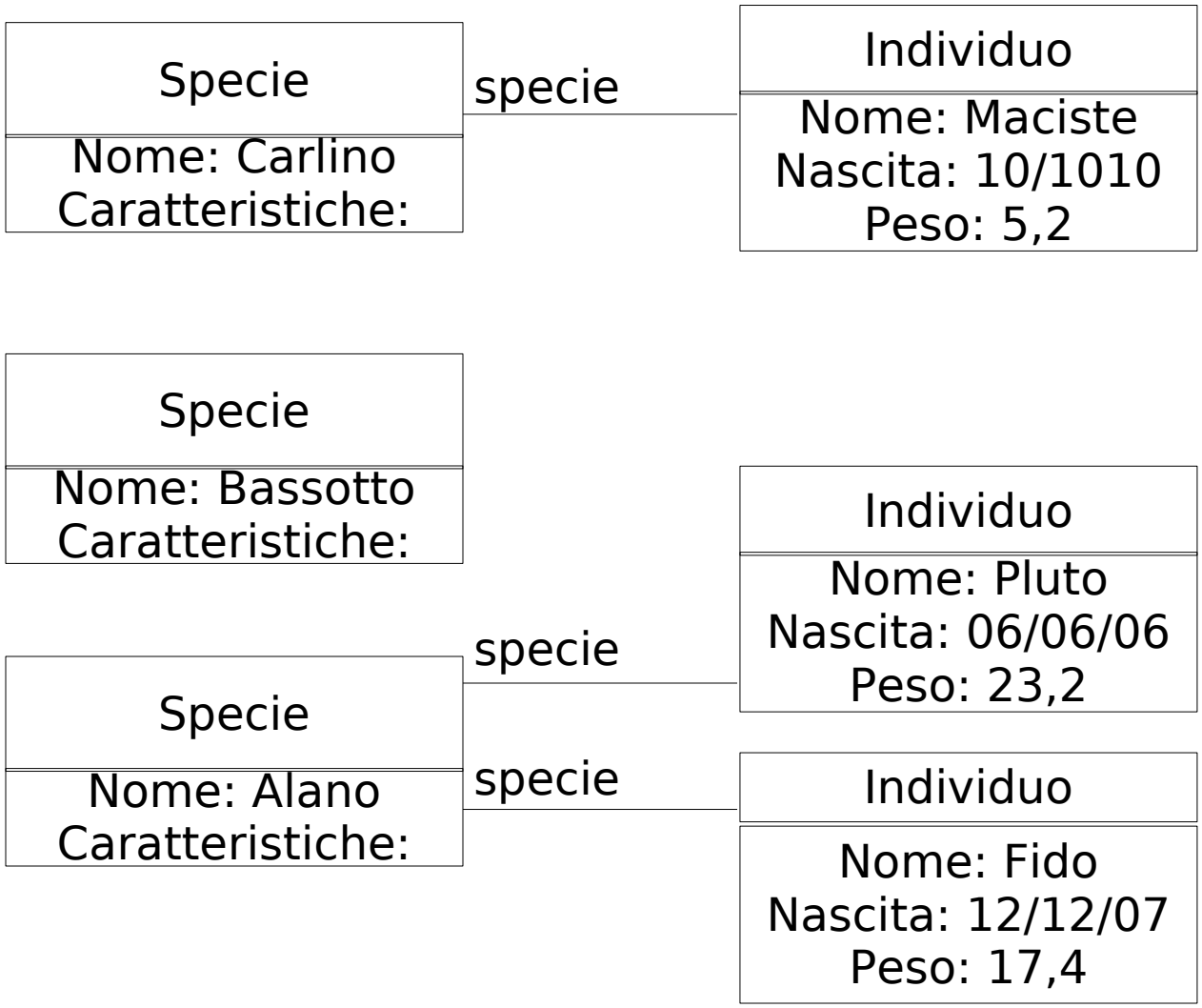


Diagramma oggetti



Polimorfismo=>Type Object

- Ogni istanza di Animale diventa una istanza di Individuo (Object)
- Ogni sottoclasse di Animale diventa una istanza di Specie (Type):

  3 sottoclassi Carlino, Bassotto e Alano sono diventate le 3 istanze di Specie.

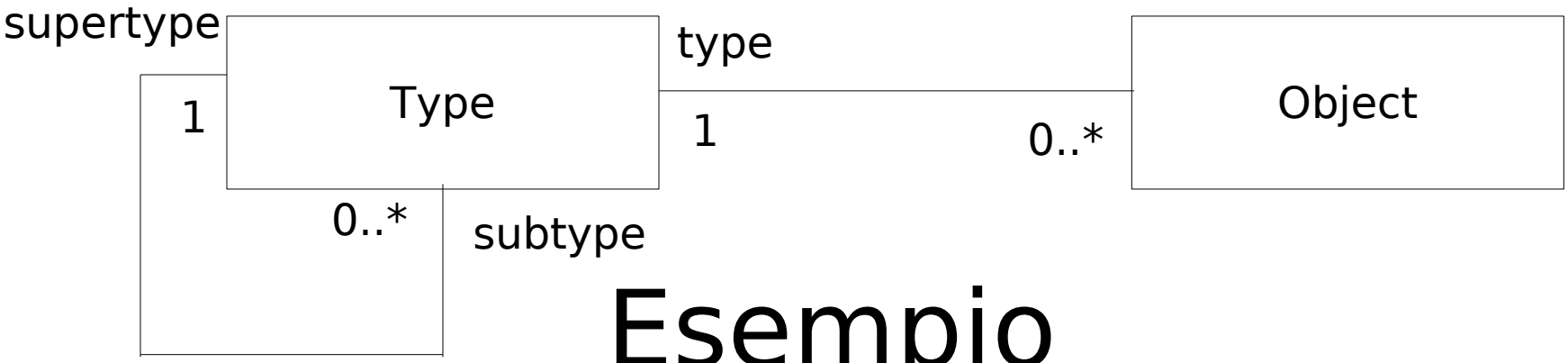
Specie

```
public class Specie { // Type
   String nome;
   String caratteristiche;
   …
   Set<Individuo> individui;//specie-1
   public Set<Individuo> getIndividui(){
     return individui;
   }
}
```

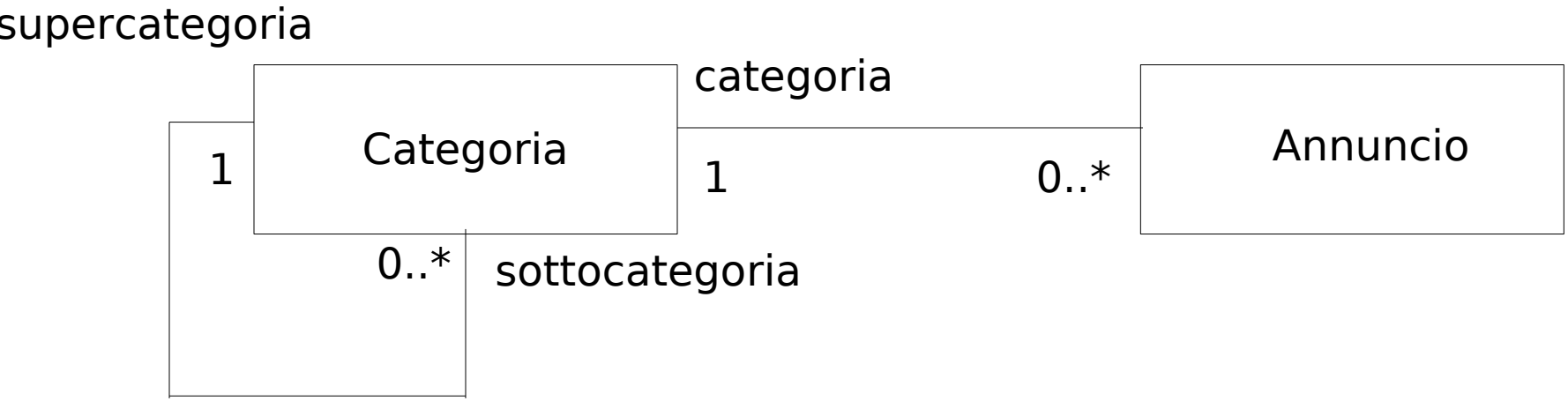
Individuo

```
public class Individuo { // Object
   Specie specie; //type!!!
   String nome;
   Date nascita;
   double peso;
   ...
   public String getCaratteristiche(){
      //delega;
      return specie.getCaratteristiche();
    }
...
```

Ricorsivo



Esempio



Equivalente Polimorfismo

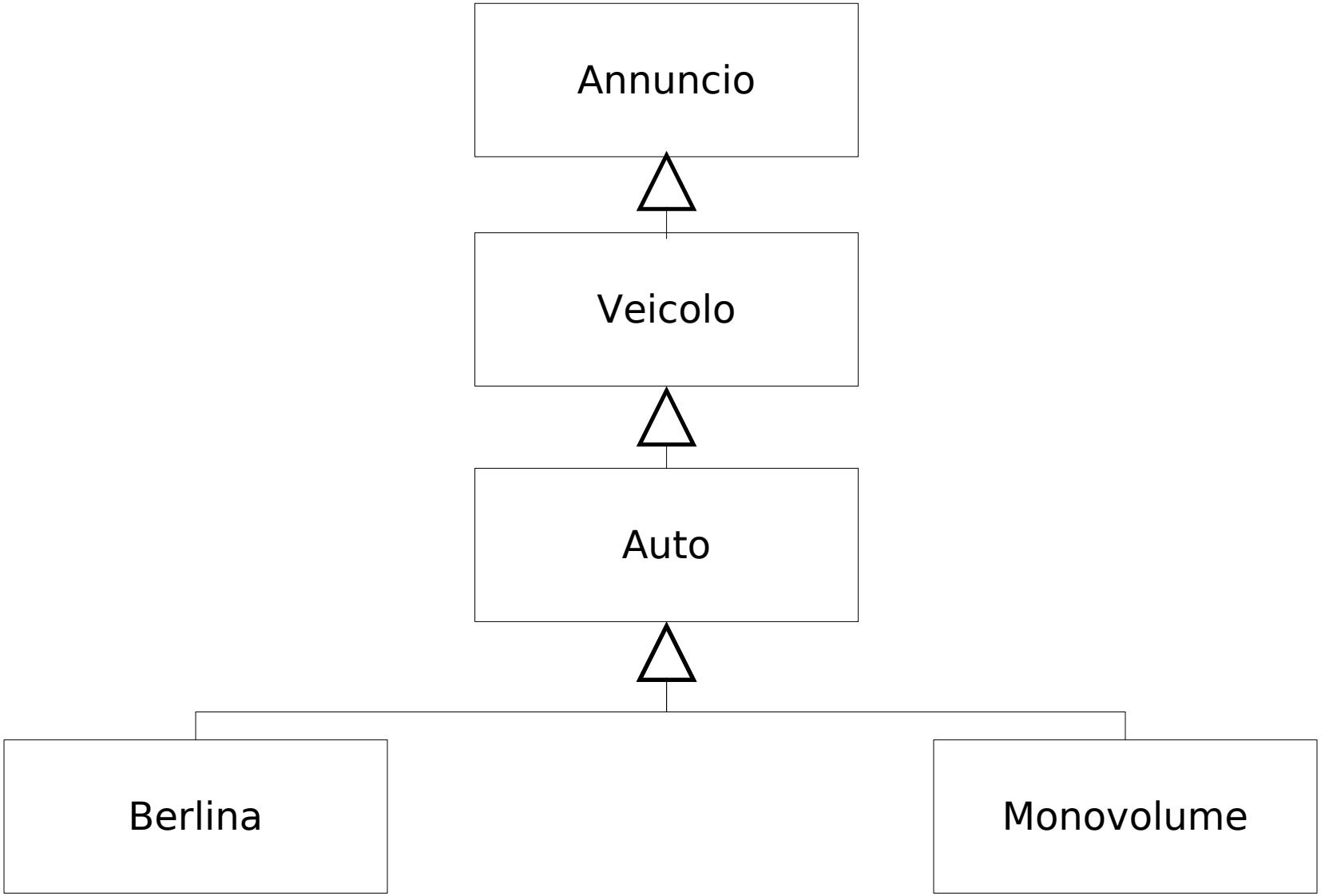
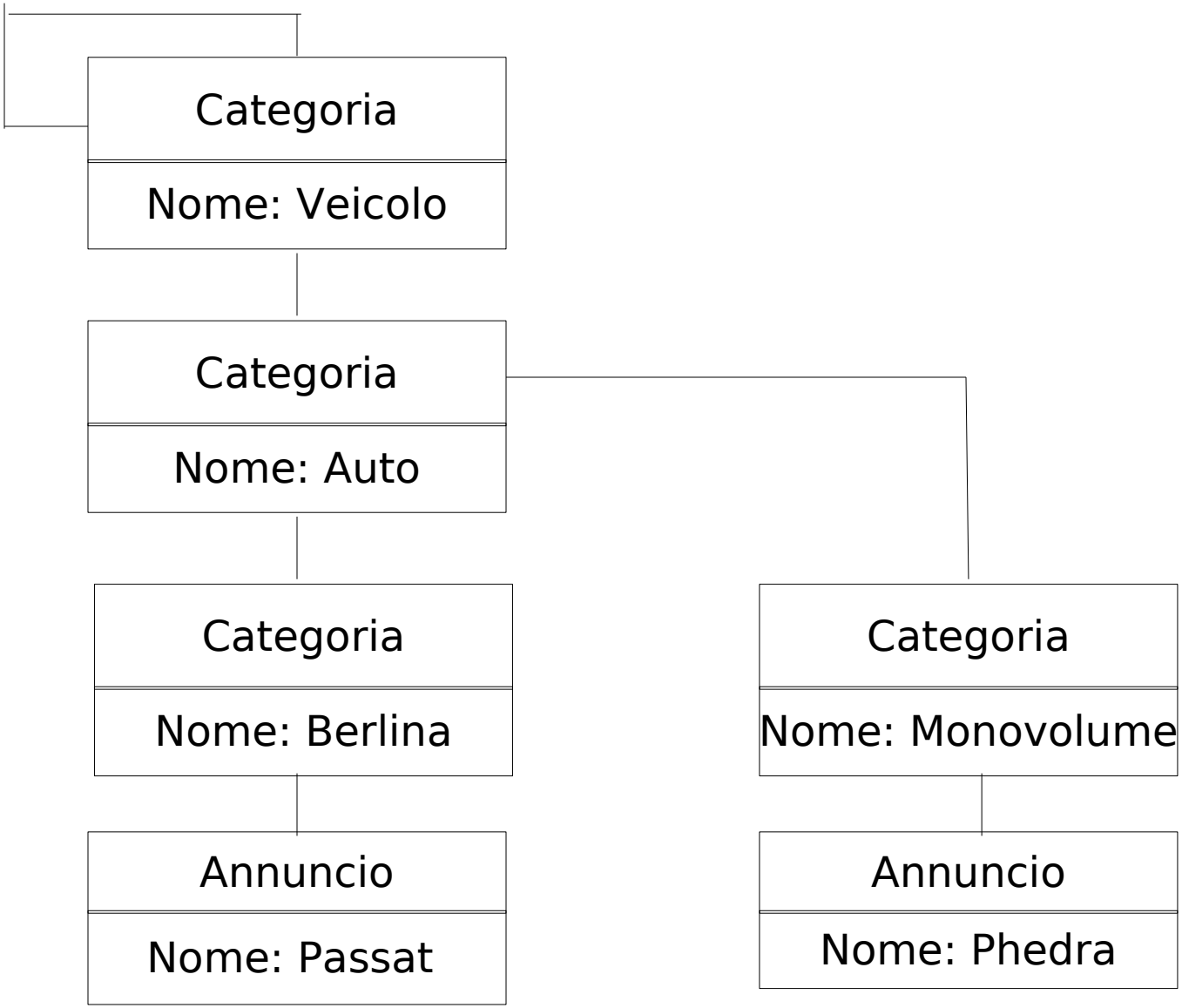


Diagramma oggetti



Polimorfismo=>Type Object

- Le relazioni di estensioni tra classi diventano relazioni di associazione tra istanze di Categoria (Type)

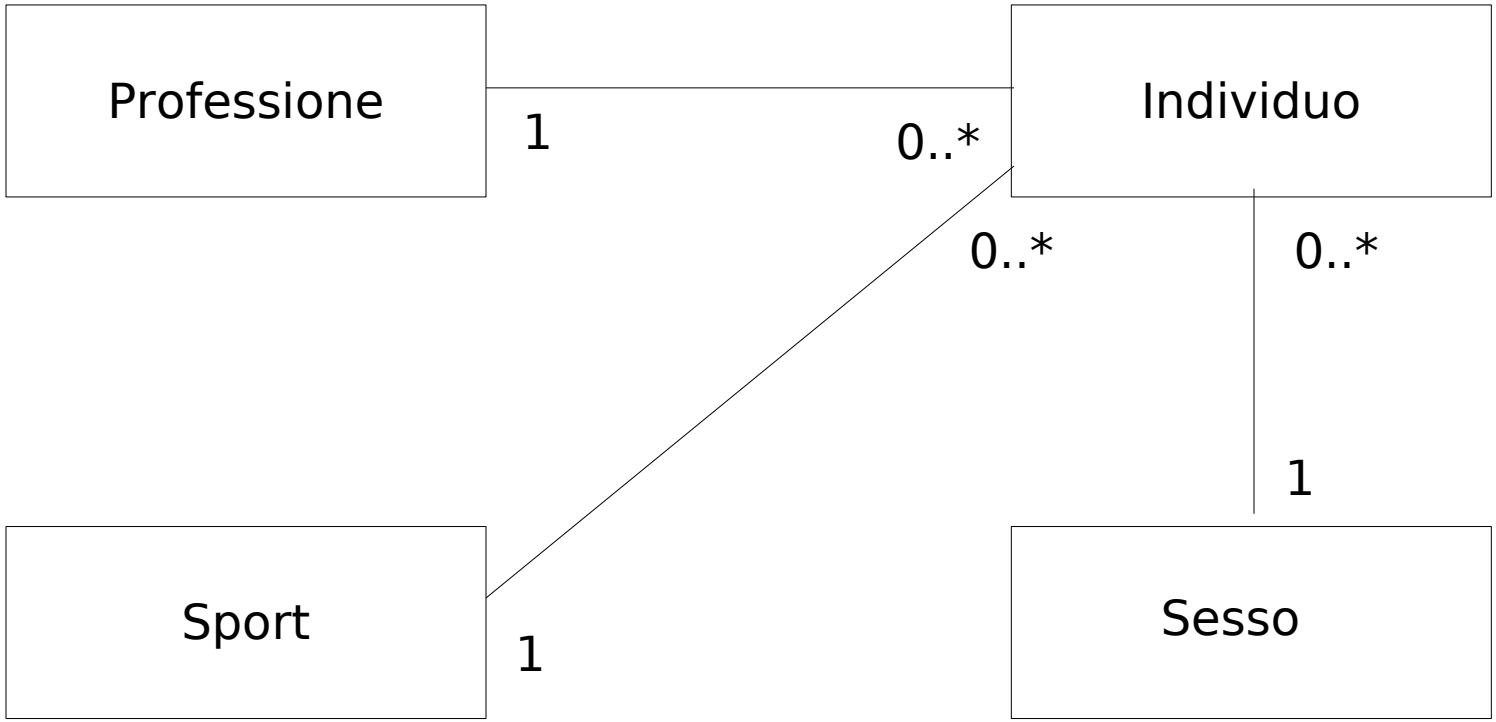
Categoria

```
public class Categoria { // Type
   String nome;
   Categoria supercategoria;
   Set<Categoria> sottocategorie;
   …
   Set<Annuncio> annunci;//categoria
   public Set<Annuncio> getAnnunci(){
     return annunci;
   }
}
```

Annuncio

```
public class Annuncio { // Object
   Categoria categoria; //type!!!
   String nome;
   ...
   public String getNameCategoria(){
     return categoria.getName();
    }
...
```

Esempio Multiplo



Equivalente Polimorfismo

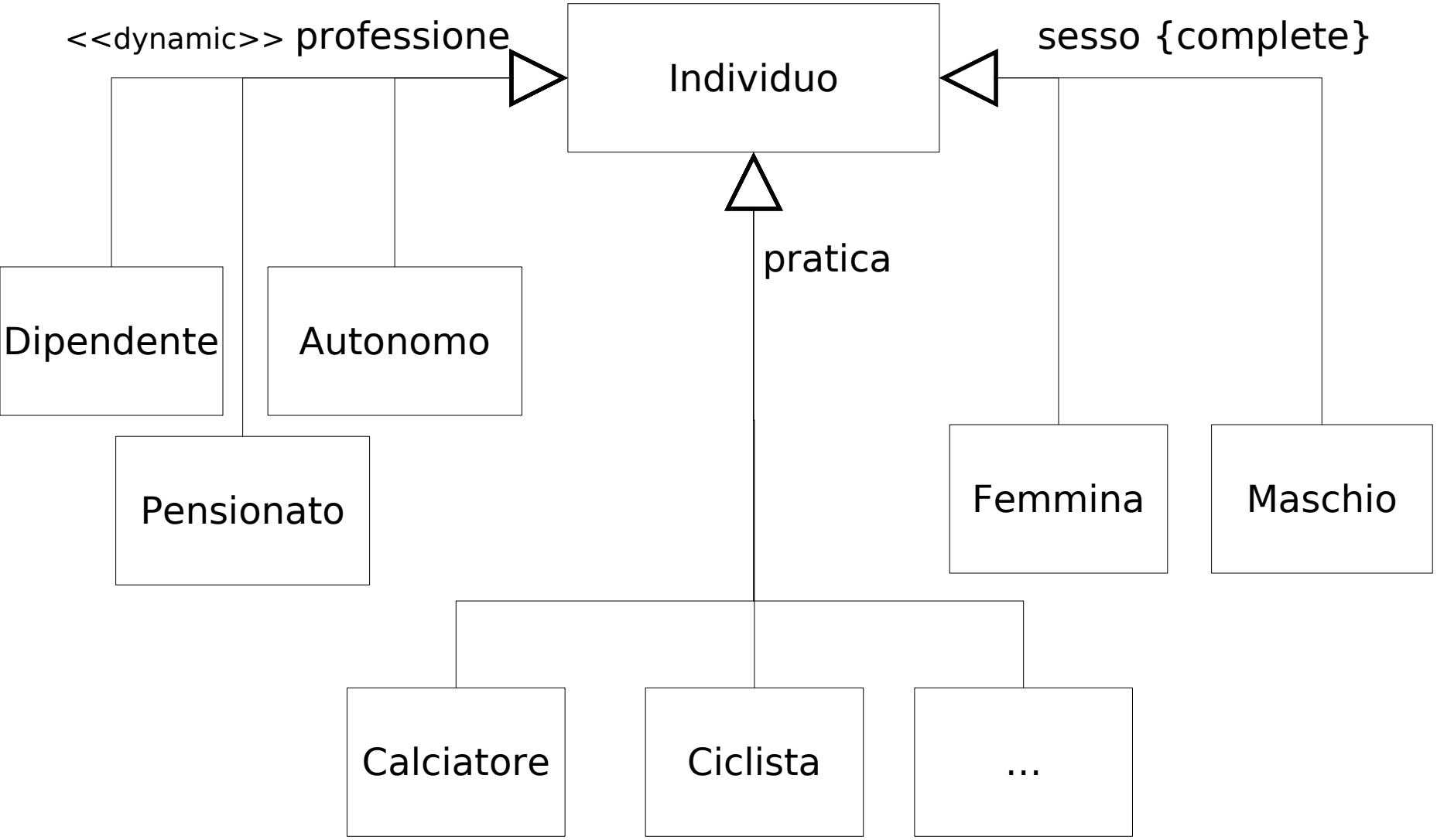
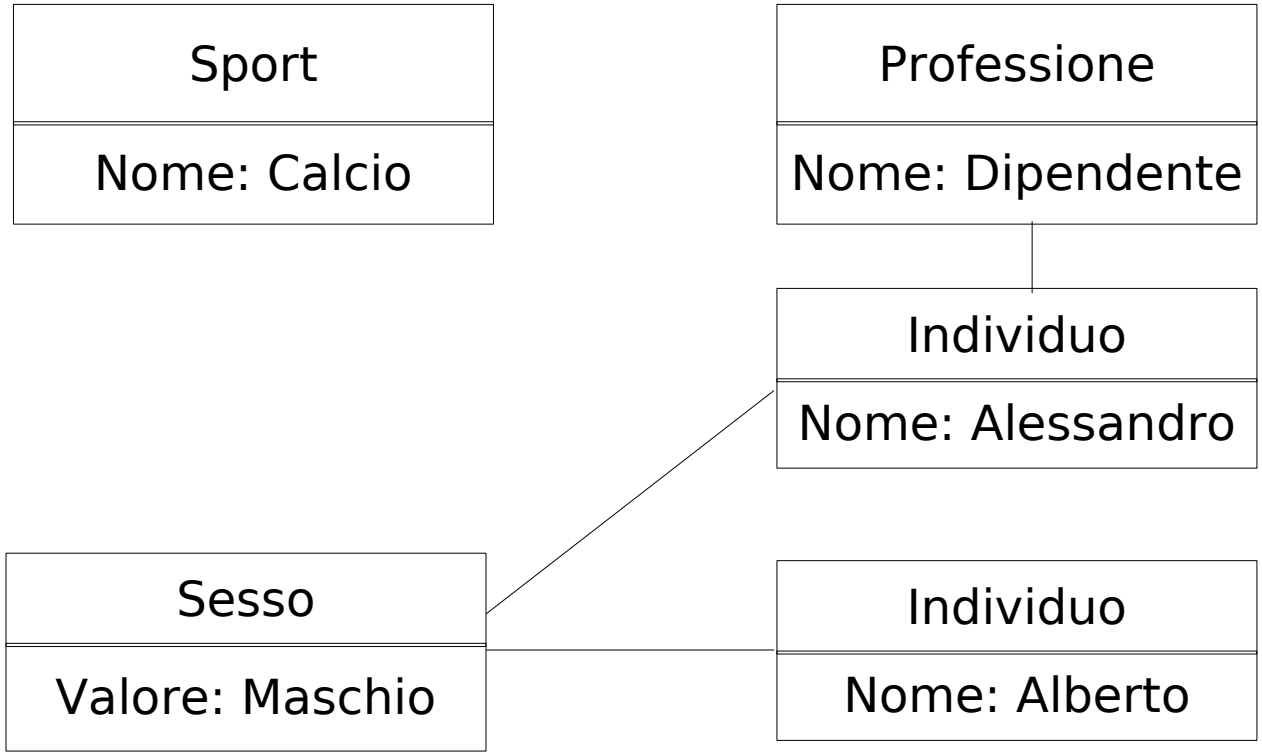


Diagramma oggetti



Polimorfismo=>Type Object

- Nella classificazione multipla ogni “generalization set” diventa una nuova classe Type e gli Object possono avere (o devono avere nel caso {complete}) associazioni con una istanza di questa classe Type.
- In Java per implementare la classificazione multipla con il Polimorfismo bisogna implementare un numero elevato (2x3x3) di classi e N oggetti rispetto a Type/Object (1+1+1)+1 e 2+3+3 istanze di Type e N oggetti.

Type

```
public class Sesso { // Type
   Valore nome;
   Set<Individuo> individui;}
public class Professione {//Type
   String nome;
   Set<Individuo> individui;}
Public class Sport {//Type
   String nome;
   Set<Individuo> individui;}
```

Object

```
public class Individuo { // Object
   Sesso sesso; //type
   Professione professione; //type
   Sport sport; //type
   ...
 }
```

Quando Type Object?

- Le istanze della classe devono essere raggruppate per tipo/categoria
- La classe richiede un grande numero di sottoclassi e/o il cui numero è sconosciuto
- Le istanze possono cambiare tipo dopo essere state create (tipo dinamico)
- Le istanze stesse possono essere raggruppate ricorsivamente (categorie e sottocategori ricorsive)

Type Object PRO

+Creazione Runtime Type

+Riduce il numero di sottoclassi

+ Nasconde la Separazione istanze dal tipo:

Object implementa l'interfaccia di Type che implementa tramite delega;

+ Cambiamento dinamico del Tipo

+ Estensione indipendente di Type e Obejct

+ Classificazione multipla

Type Object Contro

- Progettazione + complessa
- Implementazione + complessa
- Ogni oggetto deve mantenere il riferimento al proprio tipo
- Non risolve il problema di Oggetti con comportamento diverso: l'implementazione dei metodi della classe Type è unica. (eventualmente programmazione guidata dai dati, es. parser)

Esempio

Gestione impiegati: funzioni/mansioni/ruoli non noti a priori, cambiano nel tempo e possono essere + di 1.

- Nuove cariche
- Promozione
- Incarichi multipli

Mansione

```
public class Mansione { // Type
   String nome;
   double stipendio;
   Set<Benefit> benefits;
   Set<Impiegato> impiegati;
   …
   public Set<Impiegato> getImpiegati(){
     return impiegati;
   }
}
```

Impiegato ...

```
public class Impiegato { // Object
    Mansione mansione; //type!!!
    String nome;
    String cognome;
    ...
    public double getStipendio(){
        //delega;
        return mansione.getStipendio();
    }
...
```

.. Impiegato

```
public class Impiegato { // Object
    ...
    public Impiegato(Mansione m, … ){
      mansione = m;
      …
    }

    public void promuovi(Mansione m) {
       mansione=m;
  }
```

Dirigente

```
public class Dirigente extends Impiegato
{ // Object
   Mansione mansione2

   public double getStipendio(){
      //delega;
      return mansione.getStipendio()
          + mansione2.getStipendio();
   }
```

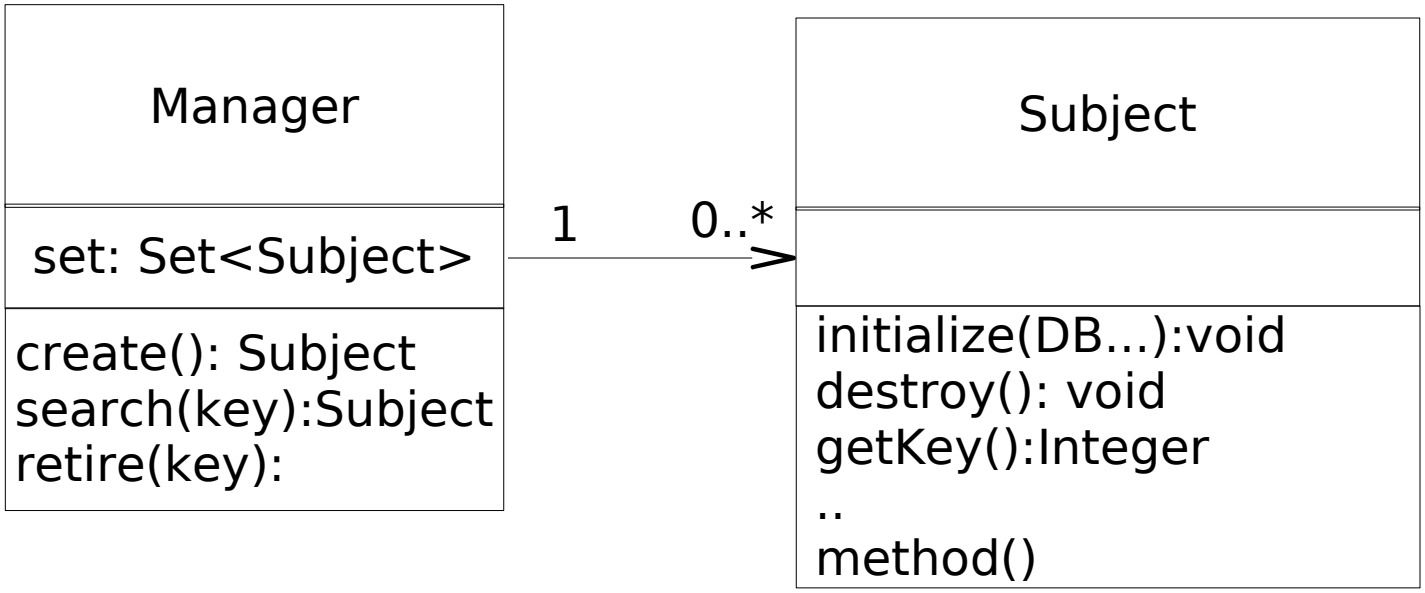
Manager

- Caso particola di Pure Fabrication
- + funzioni rispetto a Factory
- Intero ciclo di vita degli oggetti: creazione, distruzione, ricerca, persistenza ...

Quando Manager

- Tutti gli oggetti accessibili insieme
- Modifiche all'implementazione degli oggetti non devono riflettersi nella gestione del ciclo di vita
- Molte classi diverse hanno la stessa gestione del ciclo di vita (riuso manager)

Manager


Manager
set: Set<Subject>
create(): Subject
search(key):Subject
retire(key):
1
0..*
Subject
initialize(DB...):void
destroy(): void
getKey():Integer
..
method()

Manager PRO

+Indipendenza degli oggetti dalla gestione del ciclo di vita (es. possibile cambiare la gestione della persistenza senza agire sugli oggetti stessi)

+Riuso del codice del Manager con + oggetti

+ Insieme di oggetti visibili come unicum

+ Subclassing del Manager

Manager Contro

- La divisione tra Manager e Subject può rompere l'incapsulamento
- Difficile decidere come dividere il codice tra Manager e Subject.
- Se l'istanza del Manager gestisce una gerarchia di Subject (ovvero istanze di sottoclassi diverse di Subject) allora ha bisogno di altre informazioni per istanziare gli oggetti (es. Reflection)

Manager...

```
public class ContoManager {//static o
  Map<Integer,Conto> conti = … ;
  public Conto search(int key){
    Conto c = conti.get(key);
    if (c==null){
      c = new Conto();
      c.initialize(db);
      conti.add(c);}
    return c;
  }
```

...Manager

```
public Conto create(int key){
    Conto c = conti.get(key);
    if (c!=null)
        throw new Duplicate...
    c = new Conto();
    conti.add(c);
    return c;}
  public void retire(Conto c) {
   conti.remove(c);
   c.destroy();
 }
```

Manager

- Con una implementazione diversa che tiene conto dei riferimenti multipli può gestire anche gli accessi concorrenti
- Da ogni altro oggetto posso recuperare un riferimento a un oggetto gestito dal manager:

```
Manager.search(key)
Manager.getInstance().search(key)
```

Indirection

- Dove assegnare una responsabilità per evitare l'accoppiamento diretto tra due (o più) oggetti?
- Assegna la responsabilità a un oggetto intermediario.
- Casi particolari sono Adapter, Facade e Observer
- Molti Indirection sono Pure Fabrication
- Sostenere Low Coupling/Protect Var.

Indirection

- Dove assegnare una responsabilità per evitare l'accoppiamento diretto tra due (o più) oggetti?
- Assegna la responsabilità a un oggetto intermediario.

Adapter

- Dove gestire interfacce incompatibili o fornire un'interfaccia stabile a componenti simili con interfacce diverse?
- Converti l'interfaccia di un componente in un'altra interfaccia attraverso un oggetto adattatore intermedio

Esempio Interfaccie incompatibili

Una libreria usata da un'applicazione non deve essere sostituita da una nuova libreria che però ha una interfaccia diversa.

Ovviamente si vuole modificare meno codice possibile

Esempio cont.

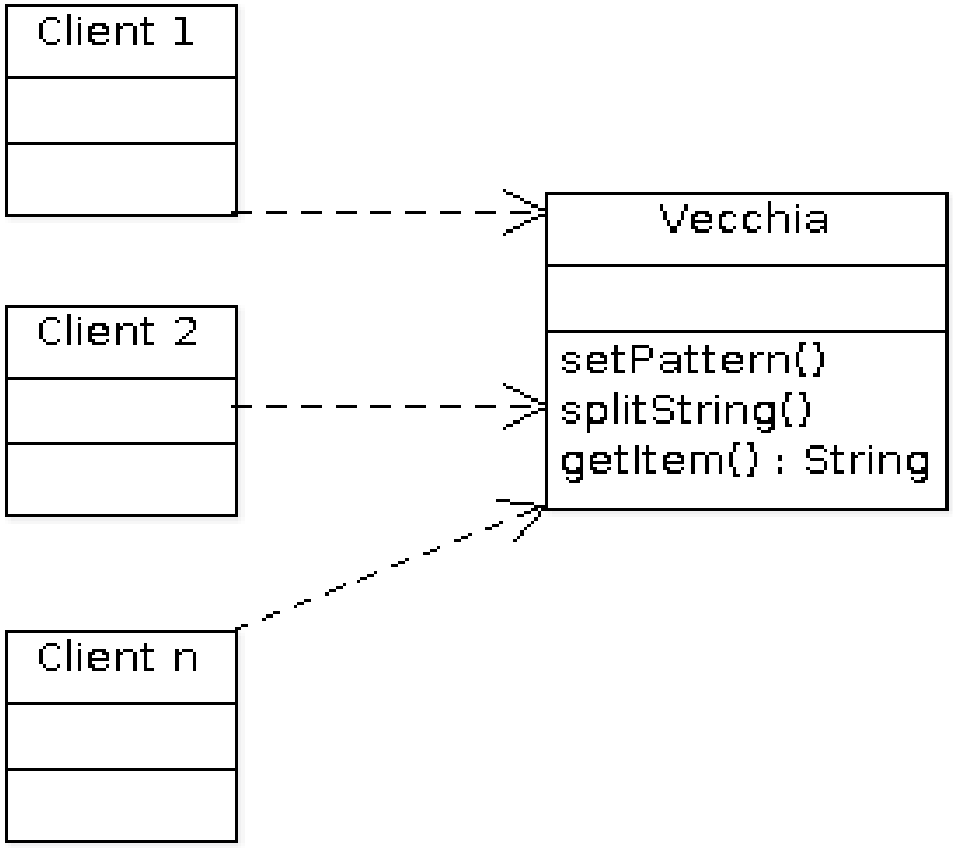
Interfaccia vecchio oggetto

- splitString(s);
- setPattern(p);// se non impostato si usa ','
- getItem(i): String //se non viene chiamato
  prima splitString o i non esiste ritorna null

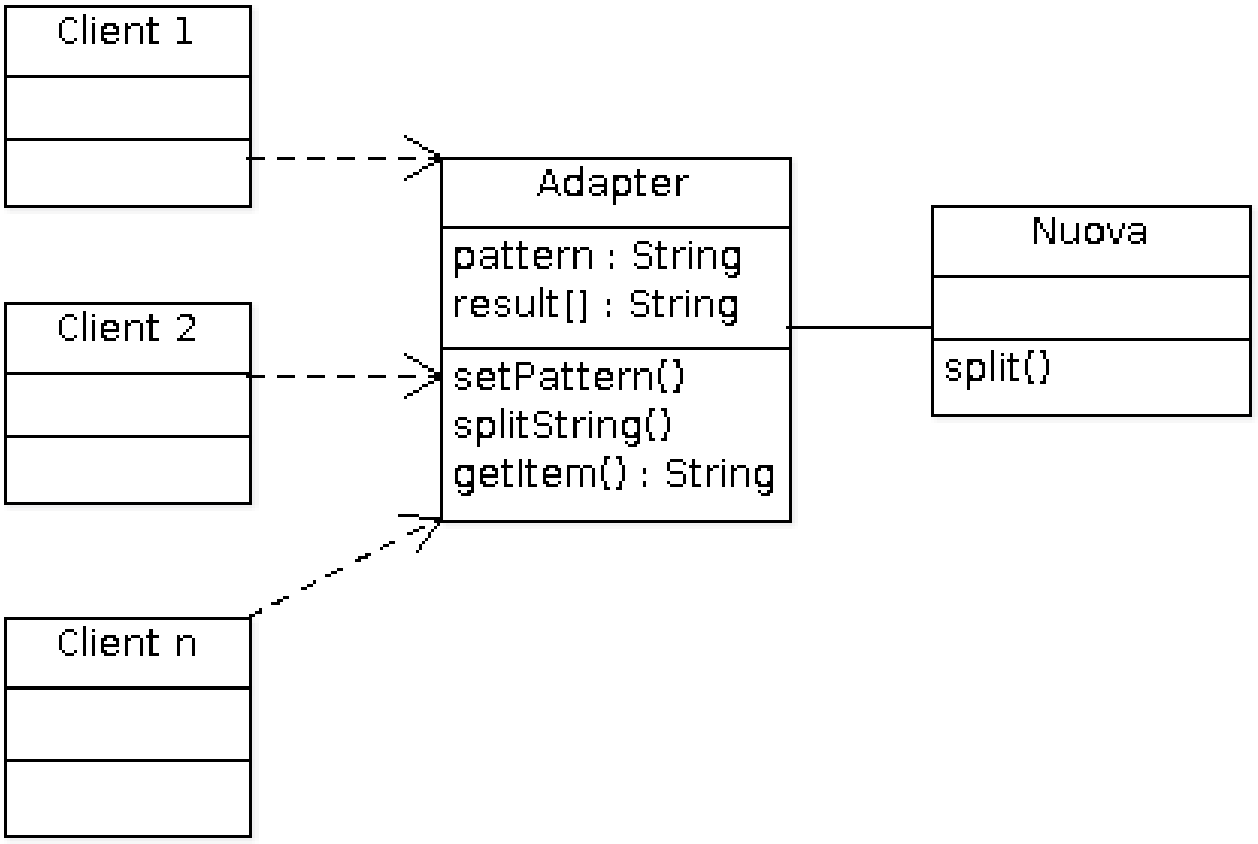
Interfaccia nuovo oggetto

- split(s,p): String[]

Prima di Adapter



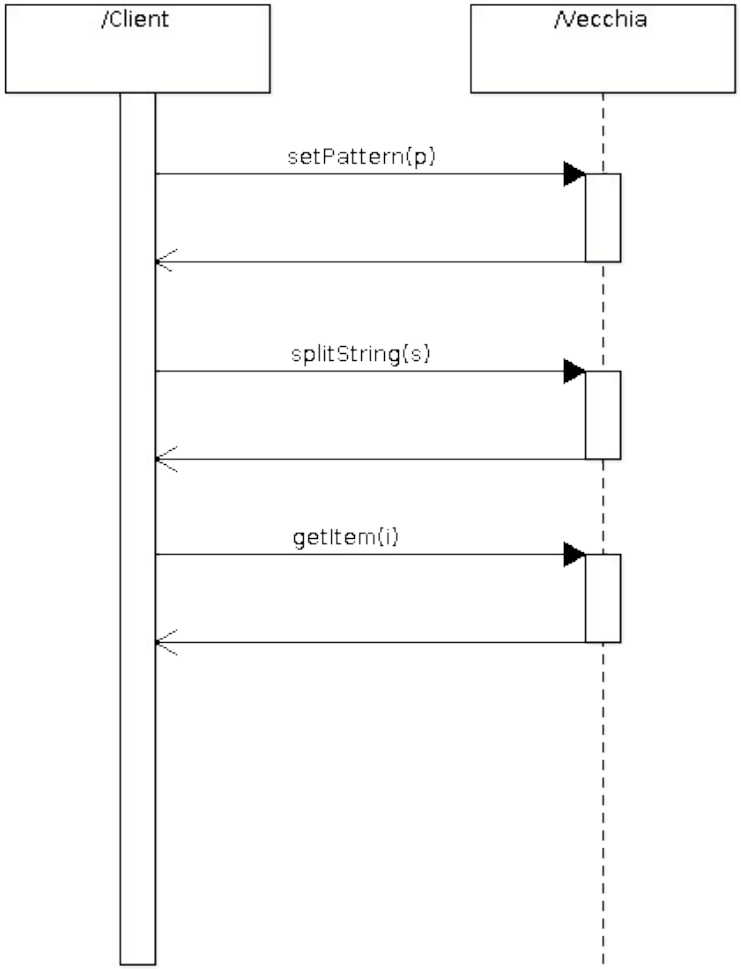
Con Adapter


Client 1
Client 2
Client n
Adapter
pattern : String
result[] : String
setPattern()
splitString()
getItem() : String
Nuova
split()

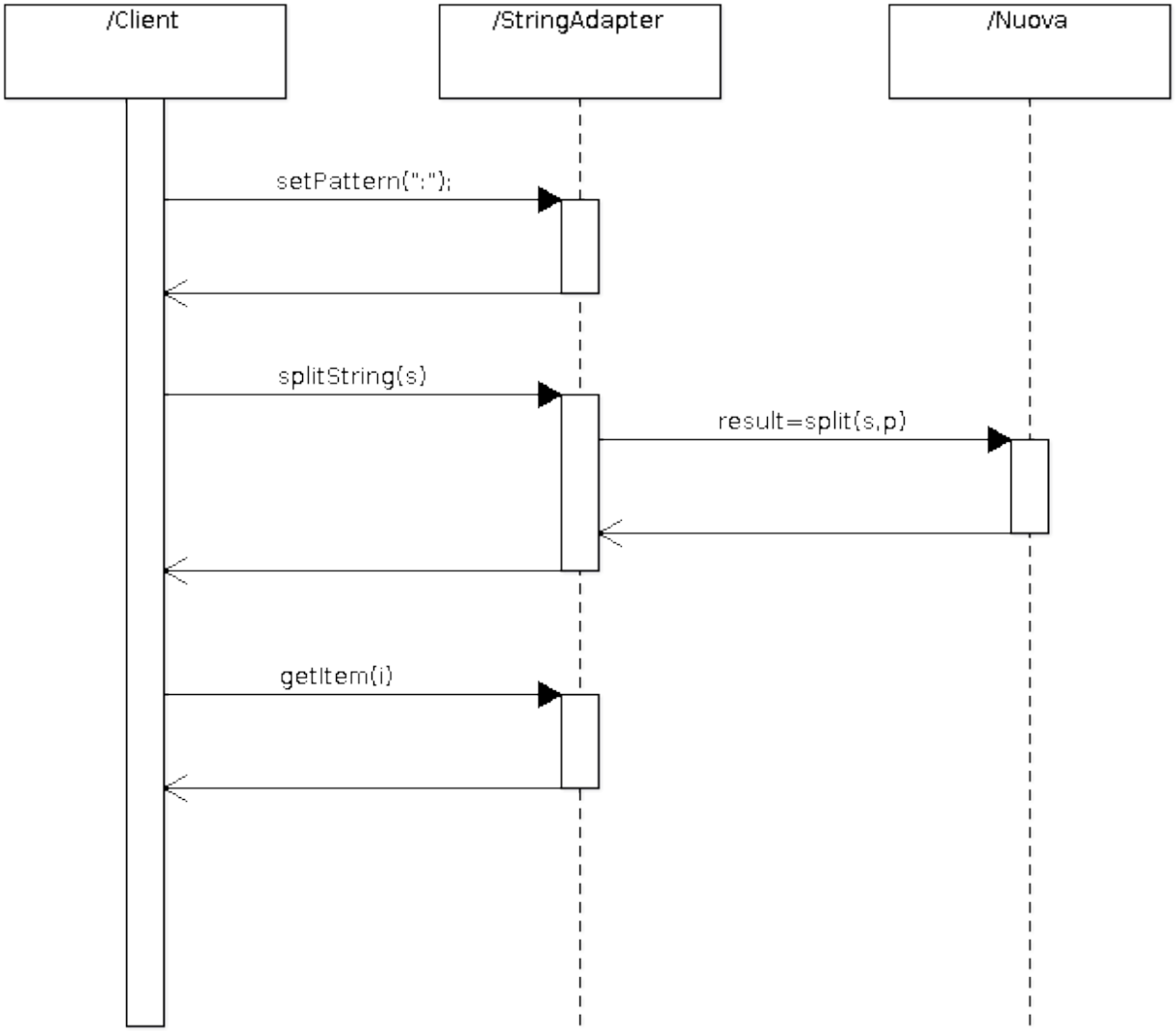
Adapter

```
public class StringAdapter{
  String p=”,”;
  String[] result=null;
  Nuova nuovo = …;
  public void setPattern(String p){
    this.p=p;}
  public void splitString(String s){
    result = nuovo.split(s,p);}
  public String getItem(i) {
    if (result!=null && i<result.lenght)
      return result[i];
    else return null;}
}
```

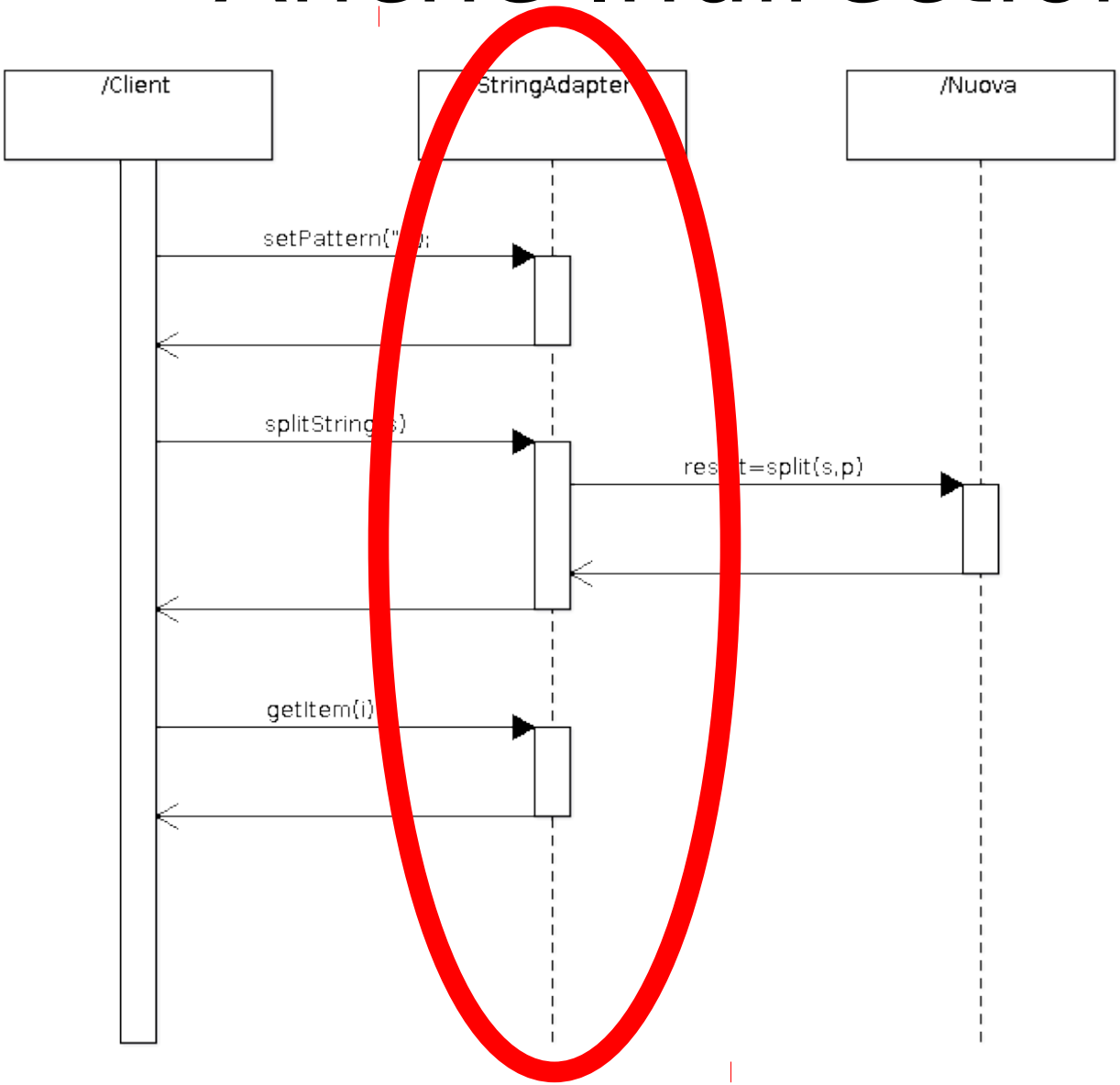
Prima Adapter



Con Adapter


/Client
/StringAdapter
/Nuova
setPattern(":");
splitString(s)
result=split(s,p)
getItem(i)

Anche Indirection



Domande