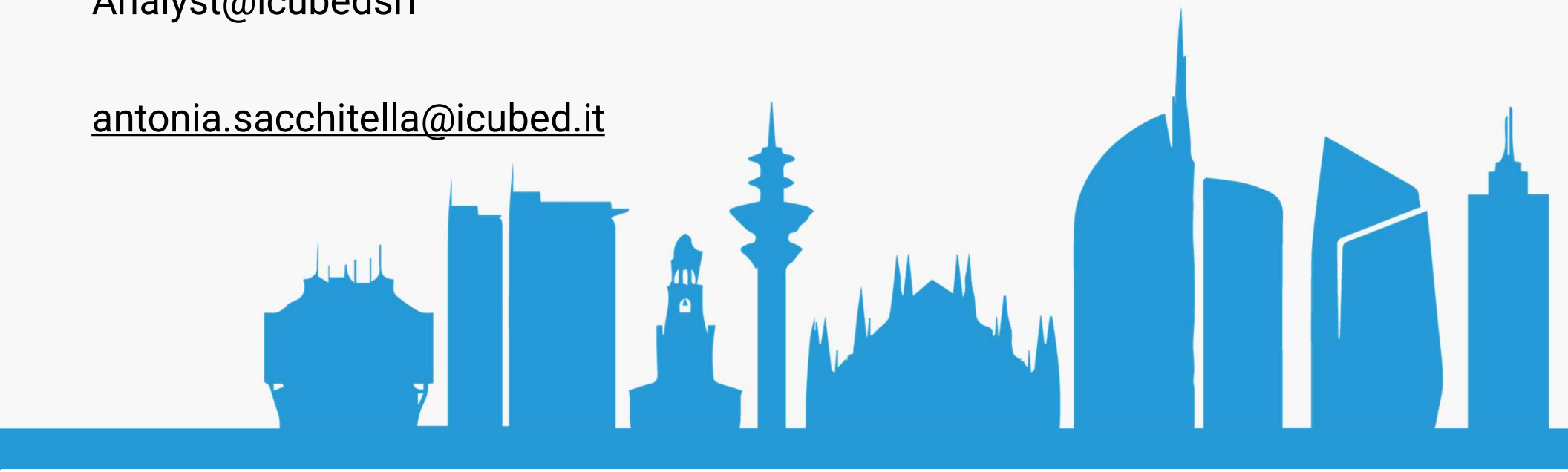
Week 2 - OOP

Antonia Sacchitella
Analyst@icubedsrl

antonia.sacchitella@icubed.it



Classi

Una classe è come un costruttore di oggetti o un "blueprint" per la creazione di oggetti.

```
public class MyClass {
    //...
}
```

Una classe può contenere ed eventualmente esporre una sua interfaccia:

- Dati (**campi** e **proprietà**)
- Funzioni (**metodi**)
- Eventi

Classi, campi e proprietà

Campo

```
public class MyClass
{
    public string Name;
}

MyClass c = new MyClass();
c.Name = "C#";
Console.WriteLine(c.Name);
```

Classi e proprietà

Proprietà

```
public class MyClass
{
    public string Name { get; set; }
}

MyClass c = new MyClass();
c.Name = "C#";
Console.WriteLine(c.Name);
```

Classi e proprietà

Proprietà 'condensata'

```
public class MyClass
{
    public string Name { get; set; }
}

MyClass c = new MyClass();
c.Name = "C#";
Console.WriteLine(c.Name);
```

Proprietà tradizionale

```
public class MyClass
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

MyClass c = new MyClass();
c.Name = "C#";
```

Classi e proprietà

Proprietà in sola Lettura

```
public class MyClass
{
    public string Name { get; }
}

MyClass c = new MyClass();
c.Name = "C#"; // KO
Console.WriteLine(c.Name); // OK
```

Proprietà in sola Scrittura

```
public class MyClass
{
    public string Name { set; }
}

MyClass c = new MyClass();
c.Name = "C#"; // OK
Console.WriteLine(c.Name); // KO
```

Classi e proprietà

Proprietà calcolata

```
public class MyClass
{
    public string FullName {
            get { return $"{FirstName} {LastName}"; }
    }
}

MyClass c = new MyClass();
c.FullName = "C#"; // KO
Console.WriteLine(c.FullName); // OK
```

La classe Object

Tutto in .NET deriva dalla **classe Object**

- Se non specifichiamo una classe da cui ereditare, il compilatore assume automaticamente che stiamo ereditando da Object

System.Object

- Tutto ciò che deriva da Object **ne eredita anche i metodi**
- Questi metodi sono disponibili **per tutte le classi** che definiamo

La classe Object

- **ToString:** converte l'oggetto in una stringa
- **GetHashCode:** ottiene il codice hash dell'oggetto
- **Equals:** permette di effettuare la comparazione tra oggetti
- **Finalize:** chiamato in fase di cancellazione da parte del garbage collector
- **GetType:** ottiene il tipo dell'oggetto
- **MemberwiseClone:** effettua la copia dell'oggetto e ritorna una reference alla copia

Ereditarietà

- Si applica quando tra due classi esiste una relazione “è un tipo di“. Esempio: **Customer** è un tipo di **Person**.
- Consente di specializzare e/o estendere una classe.
- Si chiama *ereditarietà* perché la classe che deriva (**classe derivata**) può usare tutti i membri della classe ereditata (**classe base** – keyword **base**) come se fossero propri, ad eccezione di quelli dichiarati privati.

```
public class Person {
    protected string name;
}

public class Customer : Person {

    public void ChangeName(string newName) {
        base.name = newName;
    }
}
```

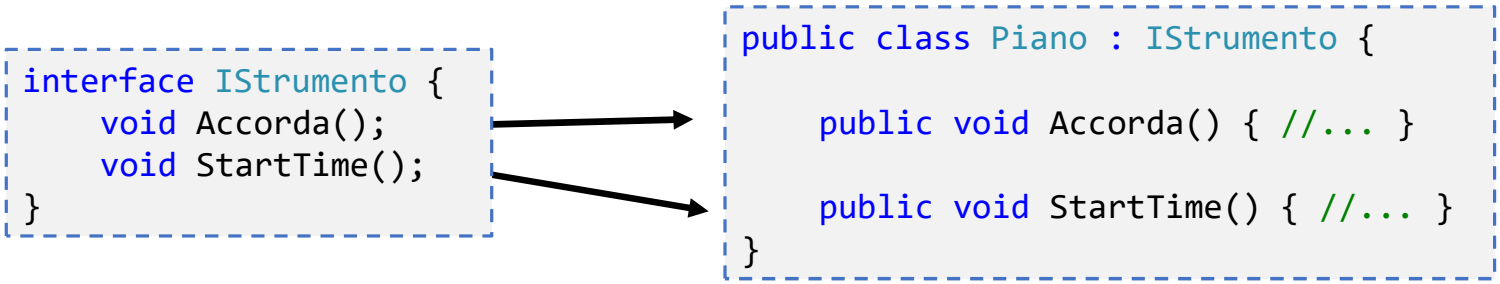
Polimorfismo

- Il *polimorfismo* è la possibilità di trattare un'istanza di un tipo come se fosse un'istanza di un altro tipo.
- Il polimorifismo è subordinato all'esistenza di una relazione di derivazione tra i due tipi.
- Affinchè un metodo possa essere polimorfico, deve essere marcato come **virtual** o **abstract**.


```
public class Strumento
{
    public virtual void Accorda() { }
}

public class Violino : Strumento
{
    public override void Accorda()
    {
        base.Accorda();
    }
}

public class Orchestra
{
    public Strumento violino, chitarra, pianoforte;

    public Orchestra()
    {
        violino = new Violino();
        violino.Accorda();
    }
}
```

Interfacce

- Un’interfaccia definisce un contratto che la classe che la implementa deve rispettare
- Un’interfaccia è priva di qualsiasi implementazione e di modificatore di accessibilità (**public**, **private**, ecc.)
- Una classe può implementare più interfacce contemporaneamente.



```
interface IStrumento {
    void Accorda();
    void StartTime();
}
```

```
public class Piano : IStrumento {

    public void Accorda() { //... }

    public void StartTime() { //... }
}
```

Demo

Design Patterns

- **Un design pattern descrive una soluzione generale a un problema di progettazione ricorrente.**

- Ogni pattern
- Possiede un nome
- Descrive quando e come può essere applicato
- Identifica le classi e le istanze partecipanti e la distribuzione delle responsabilità tra esse

Design Patterns

- **Un design pattern descrive una soluzione generale a un problema di progettazione ricorrente.**

- La principale raccolta di pattern è il volume
  - "Design Patterns: Elements of Reusable Object-Oriented Software"
- del 1994, scritto da un gruppo di 4 sviluppatori noti col nome collettivo di "Gang of Four" (GoF).

Design Patterns

- La "Gang of Four", ha identificato e descritto 23 design pattern classificati in tre gruppi principali:
  - Creazionali (Creational)
    - **Factory**
    - Abstract Factory
    - Builder
    - Prototype
    - Singleton

Design Patterns

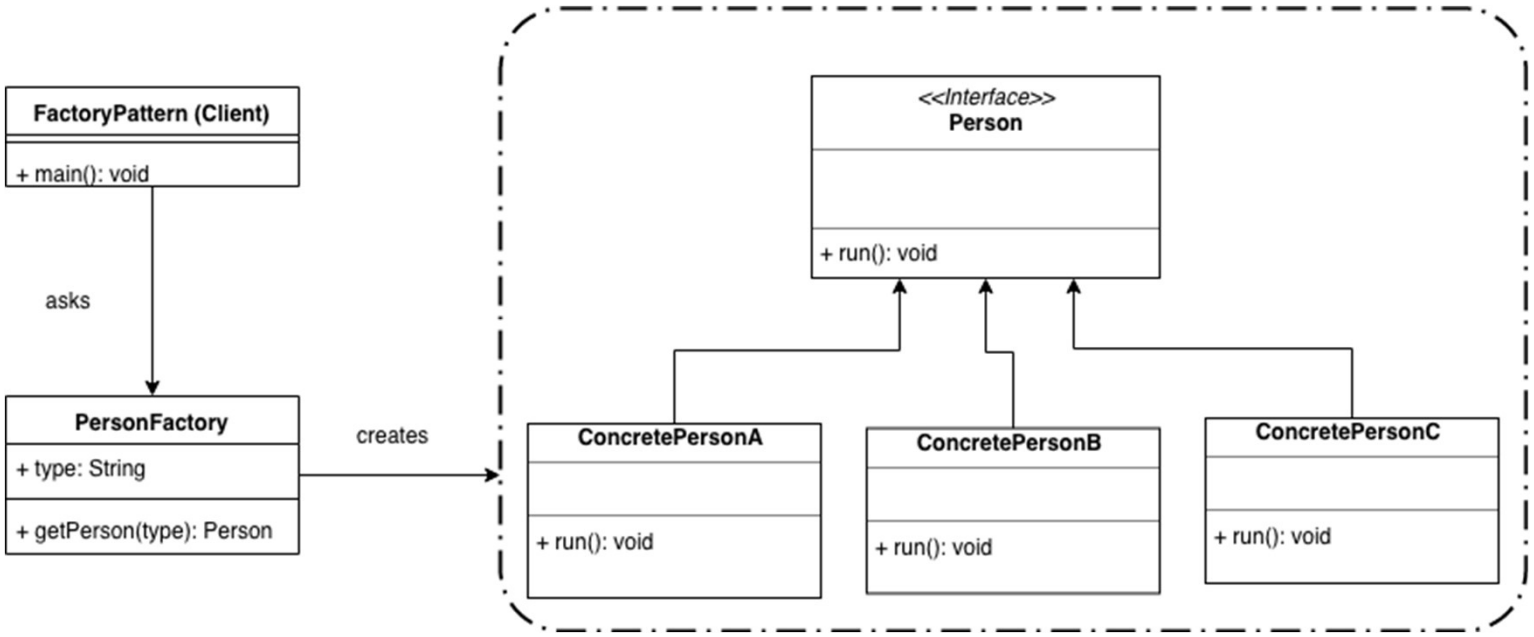
- La "Gang of Four", ha identificato e descritto 23 design pattern classificati in tre gruppi principali:
  - Strutturali (Structural)
    - Adapter
    - Bridge
    - Composite
    - **Decorator**
    - Façade
    - Flyweight
    - Proxy

Design Patterns

- La "Gang of Four", ha identificato e descritto 23 design pattern classificati in tre gruppi principali:
  - Comportamentali (Behavioral)
    - **Chain of responsibility**
    - Command
    - Interpreter
    - Iterator
    - Mediator
    - Memento
    - Observer
    - State
    - Strategy
    - Template
    - Visitor

Design Patterns - Factory

- Tipo: Creazionale

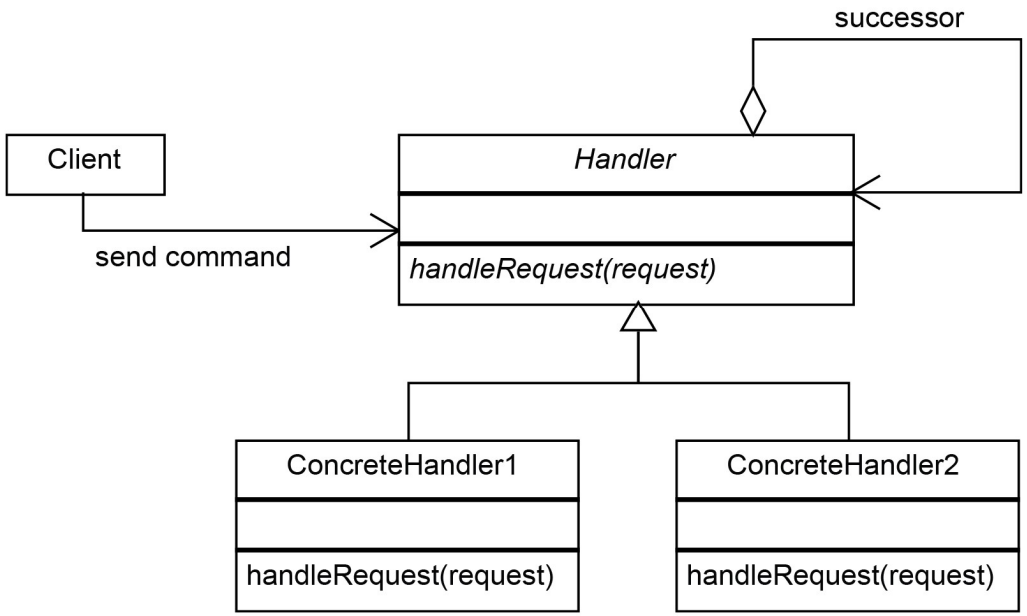


Demo

Design Pattern – Factory

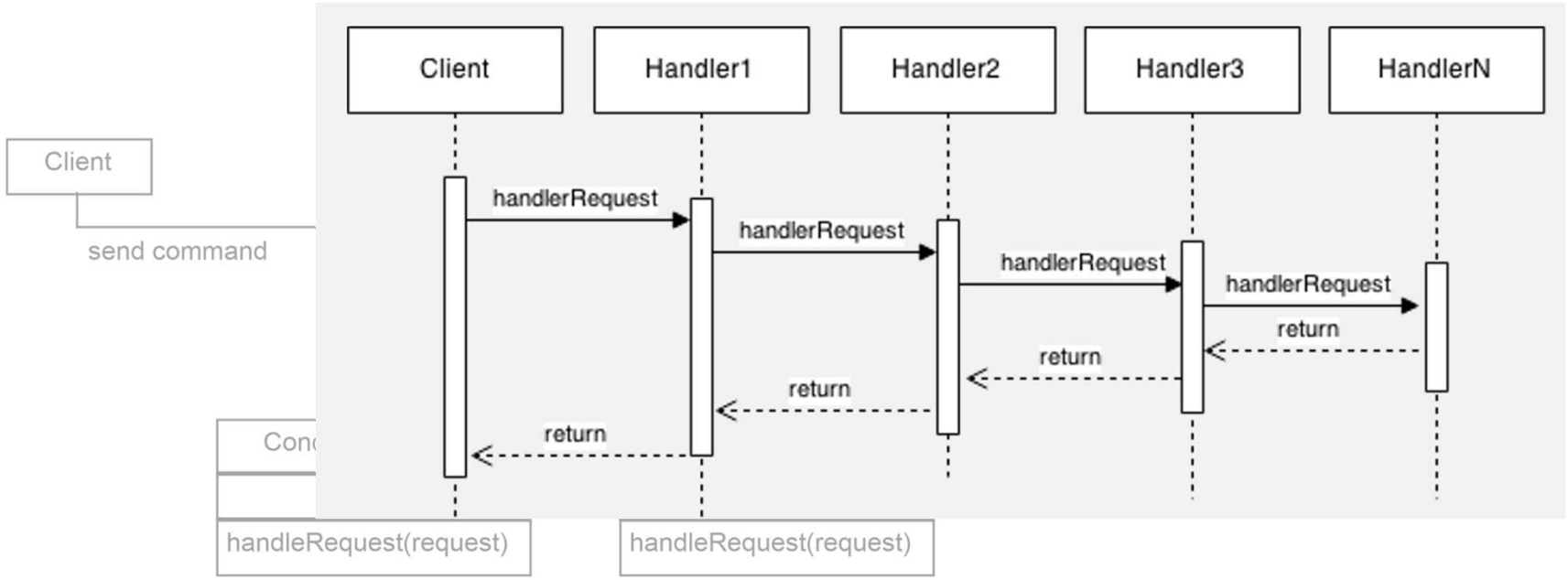
Design Patterns - Chain of Responsibility

- Tipo: Comportamentale



Design Patterns - Chain of Responsibility

- Tipo: Comportamentale

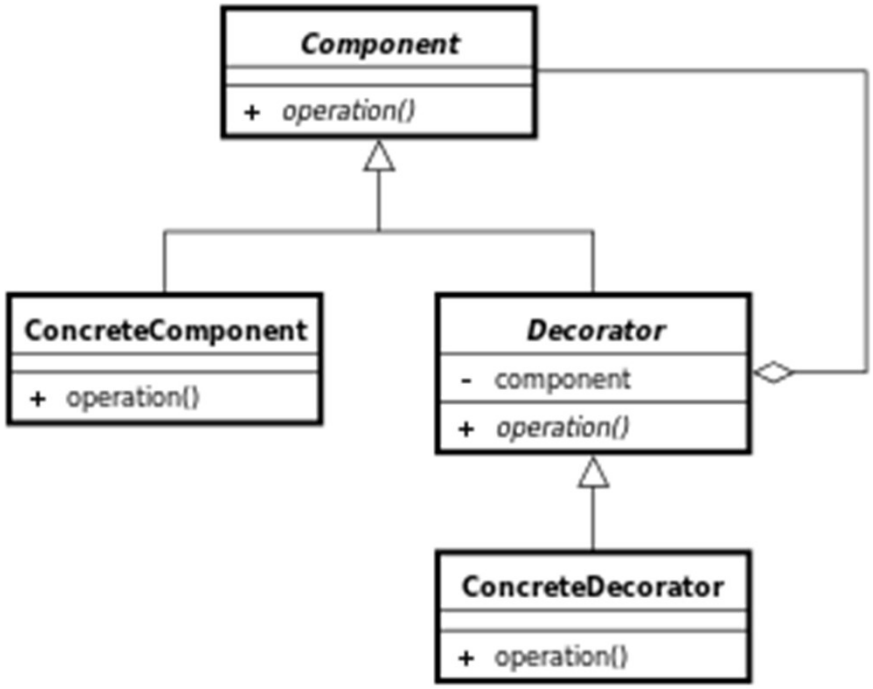


Demo

Design Pattern – Chain of Responsibility

Design Patterns - Decorator

- Tipo: Strutturale



Demo

Design Pattern – Decorator

Esercitazione 1

Si vuole implementare una soluzione software che consenta di gestire diverse tipologie aziende sulla base del numero di dipendenti che dichiara di possedere.

In particolare in base al numero di dipendenti dichiarati si dovrà distinguere tra:

- **piccola azienda** fino a 20 dipendenti,
- **media** fino a 100 dipendenti,
- **grande** fino a 500 dipendenti,
- **multinazionale** da 500 in su.
- Una volta creata l'azienda il programma dovrà consentire l'inserimento di una lista di dipendenti Ciascun dipendente sarà caratterizzato da:
  - *Nome* (string)
  - *Cognome* (string)
  - *Data di nascita* (DateTime)
  - *Data di assunzione* (DateTime)
  - *Tasso di produttività* (Int)
  - *Tasso di assenza* (Int)

Esercitazione 1

- Per ogni dipendente si dovrà implementare un modulo in grado di attribuire premi agli impiegati sulla base di opportune opzioni di selezione.

Ciascuna opzione è mutualmente esclusiva (non tutte assieme). Sia possibile individuare le seguenti possibilità:

- Opzione **PRODUTTIVITA'** - Impiegati con età < Y e produttività > W%;
- Opzione **PRESENZA** - Impiegati con età < Y e tasso di assenza annuo < Z%;
- Opzione **ANZIANITA' DI SERVIZIO** - Impiegati con un'anzianità di servizio > 43 anni
- Opzione **BENESSERE COLLETTIVO** - Impiegati con una produttività >= W;

- Siano Y, W, Z parametri di input

Esercitazione 1

- Implementare poi la possibilità per un Dipendente di avere uno o più dei seguenti benefit assegnati (tra parentesi le proprietà extra da assegnare):
  - **Posto Auto** (*Codice Posto Auto* (string))
  - **Assicurazione Sanitaria** (*Codice Cliente* (string))
  - **Ticket restaurant** (*Nr Tessera* (string), *Nr Ticket Mensili* (int))
  - **Auto Aziendale** (*Nr di Targa* (string), *Modello* (string))

Utilizzare opportunamente i design pattern visti a lezione