

Ingegneria del Software A

Ing. F. Bergenti

E-mail `Bergenti@CE.UniPR.IT`

Tel. `0521 90 5708`

Web `http://www.ce.unipr.it`

Alcuni Riferimenti Bibliografici

- E. Damiani, M. Madravio. *UML Pratico*. Pearson Education Italia, 2003.
- B. Eckel. *Thinking in Java*. Disponibile dal sito <http://www.mindview.net>
- B. Eckel. *Thinking in Patterns*. Disponibile dal sito <http://www.mindview.net>
- E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Pearson Education Italia, 2002.
- C. Ghezzi et al. *Ingegneria del Software*. Mondadori Informatica, 1996.
- P. Kruchten. *Rational Unified Process*. Pearson Education Italia, 2000.

Cos'è l'Ingegneria del Software?

- L'**ingegneria del software** tratta della realizzazione di sistemi software di dimensioni e complessità tali da richiedere uno o più team di persone.
- La nascita e lo sviluppo sono una conseguenza diretta dell'aumento di complessità del software.



Alcune Possibili Definizioni

- L'ingegneria del software è l'approccio **sistematico** allo sviluppo, all'operatività, alla manutenzione ed al ritiro del software.
- L'ingegneria del software è la disciplina tecnologica e manageriale che riguarda la **produzione sistematica** e la manutenzione dei prodotti software che vengono sviluppati e modificati entro tempi e costi preventivati.
- L'ingegneria del software è un corpus di teorie, metodi e strumenti, sia di tipo tecnologico che organizzativo, che consentono di **produrre** applicazioni con le desiderate caratteristiche di **qualità**.

Software e Ingegneria Classica

- Il software è un **prodotto** dell'ingegno e non di un processo industriale.
- L'ingegneria classica (civile, meccanica) progetta il prodotto e (spesso) il processo industriale.
- L'ingegneria del software progetta solo il prodotto e (spesso) non utilizza un processo industriale formalizzato.

Cenni Storici (1/3)



- L'ingegneria del software nasce con la conferenza NATO del 1968.
- Nuovi punti di vista
 - software **crisis**,
 - software **reuse**,
 - software **engineering**.

Cenni Storici (2/3)



- Anni '90, sviluppo delle **tecnologie orientate agli oggetti**
 - programmazione orientata agli oggetti,
 - UML (Unified Modeling Language),
 - Design pattern.

Cenni Storici (3/3)



- Da metà anni '90, nuove prospettive
 - Web e e-commerce,
 - open source.
- Java
 - linguaggio orientato agli oggetti,
 - multi piattaforma,
 - Web oriented,
 - ben accettato dalla comunità dell'open source.

Java

- Piattaforma sviluppata da Sun Microsystems
 - linguaggio orientato agli oggetti che migliora il C++,
 - strumenti sia a **compile-time** che a **run-time**.
- Nasce dall'esigenza di svincolare il software dall'hardware e dal sistema operativo
 - sistemi Web-oriented (inizialmente lato client, oggi principalmente lato server),
 - sistemi embedded,
 - recentemente, sistemi mobili (telefoni cellulari, PDA).

Java Runtime Environment (JRE)

- Java Virtual Machine (**JVM**)
 - macchina astratta realizzata su una macchina fisica,
 - come le macchine fisiche, ha un insieme di istruzioni e manipola la memoria e le periferiche,
 - esegue un codice (**bytecode**) indipendente dalla macchina fisica.
- Java class library
 - classi e oggetti disponibili a corredo della JVM,
 - offre soluzioni ottimizzate a molti problemi applicativi.

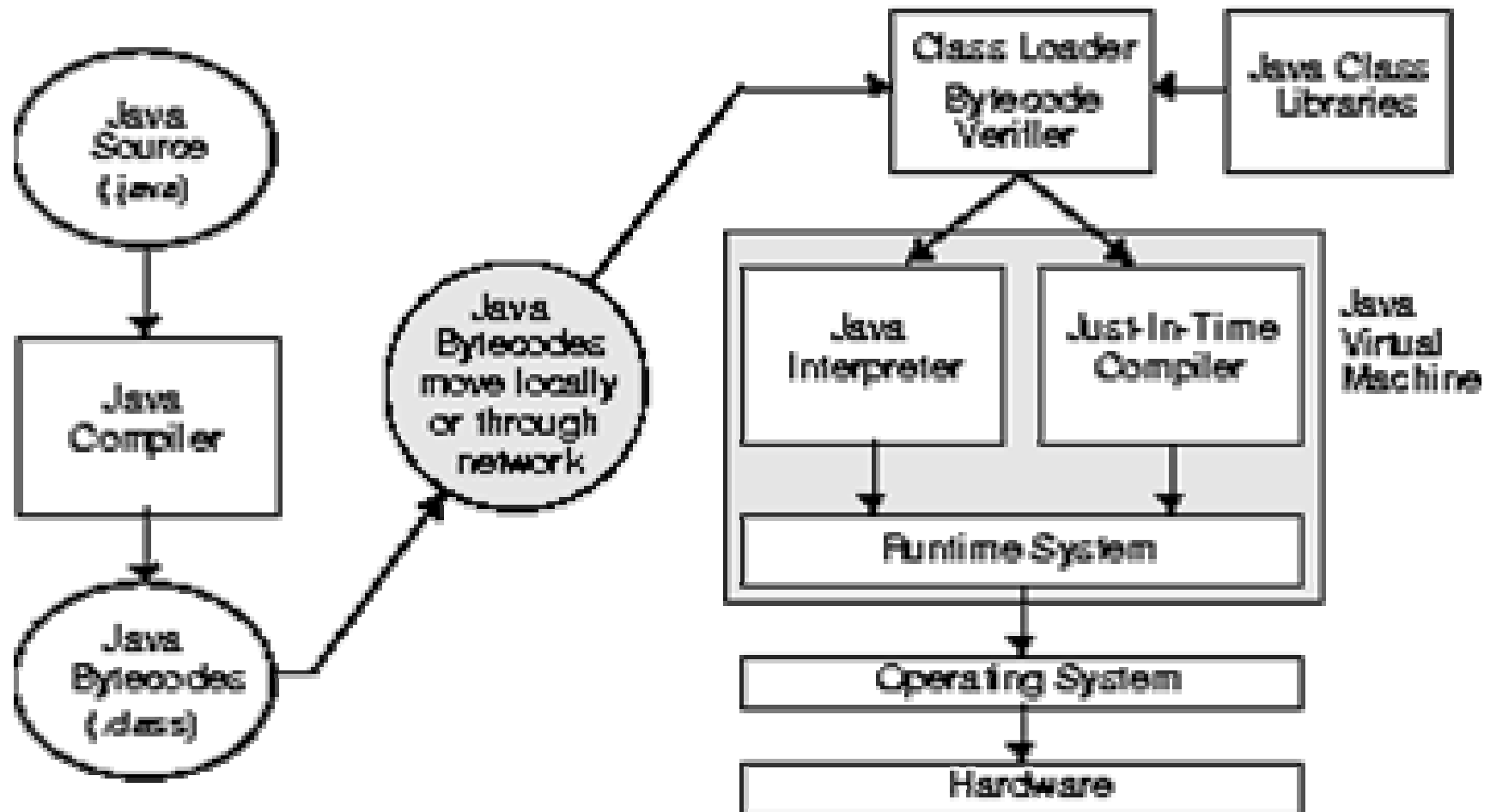
Java Development Toolkit (JDK)

- JRE corredato dagli strumenti per lo sviluppatore
 - compilatore (**javac**),
 - documentazione delle API della Java Library,
 - esempi,
 - ...
- Disponibile dal sito `http://java.sun.com`

Ciclo di Vita di un'Applicazione Java

Compile-time Environment

Runtime Environment
(Java Platform)



Perché Conviene Utilizzare Java?

- **Vantaggi**
 - ottimo linguaggio di programmazione orientato agli oggetti che migliora il C++,
 - class library molto potente,
 - multi-piattaforma,
 - nessun costo dell'ambiente di sviluppo.
- **Svantaggi**
 - lentezza dovuta all'interpretazione del bytecode.

Compilatore Just-in-Time (JIT)

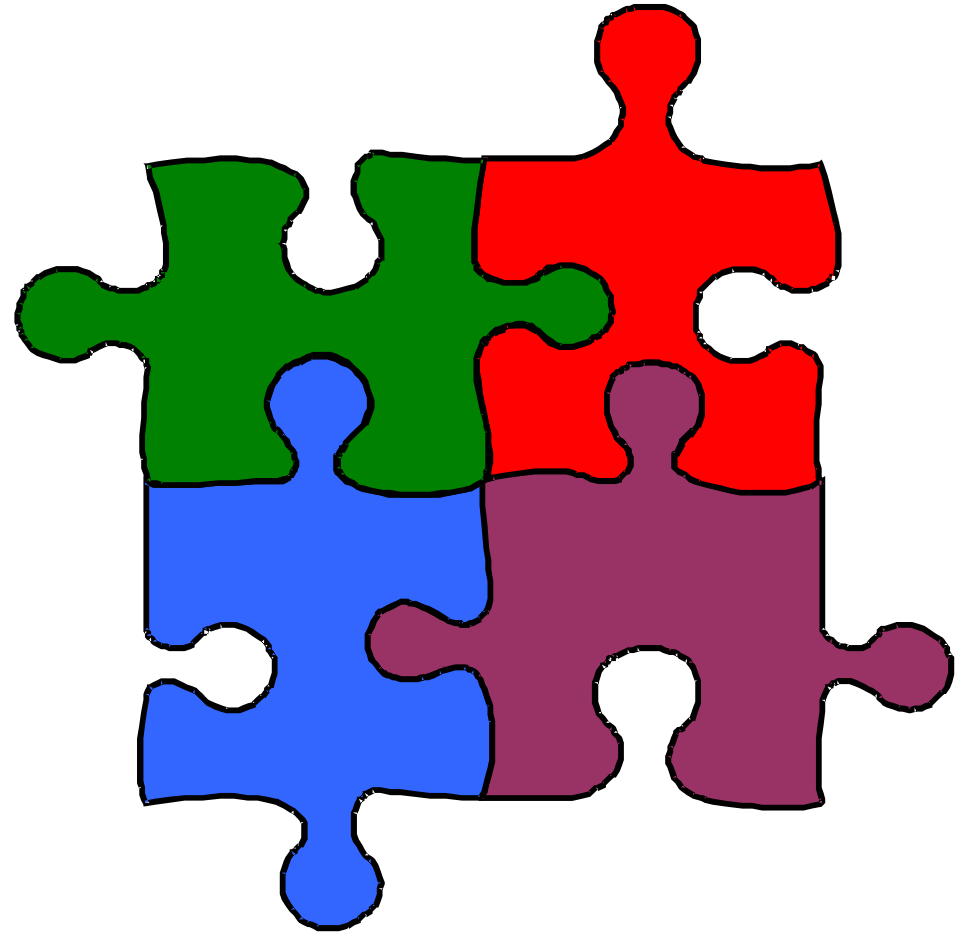
- Il compilatore JIT viene introdotto nella JVM per ovviare alla necessità di interpretare il bytecode.
- Genera codice eseguibile dalla macchina fisica ogni volta che nuovo bytecode viene caricato.
- Il codice generato dal compilatore JIT è più lento di un codice generato da un linguaggio come il C++, ma ha prestazioni ragionevoli.

Microsoft .NET

- Estende in vario modo le funzionalità offerte da Java.
- Utilizza un codice multi piattaforma chiamato Intermediate Language (IL)
 - multi-linguaggio,
 - IL può essere generato da vari linguaggi (VisualBasic.NET, Eiffel, ...),
 - è possibile integrare componenti scritti in linguaggi differenti,
 - migliori prestazioni rispetto a Java perché IL è progettato ed ottimizzato per la compilazione JIT.

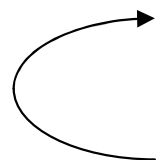
Programmazione Orientata agli Oggetti (in Java)

- Cambia il paradigma della programmazione procedurale.
- Spinta dallo sviluppo delle interfacce grafiche.
- Cerca di massimizzare il riuso di codice
 - minore time to market,
 - minori costi (personale).

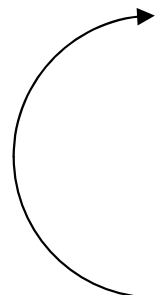


Nuovo Approccio

- Programmazione procedurale

- 
- analisi del problema per trovare un algoritmo risolutore,
 - scomposizione dell'algoritmo risolutore in un insieme di problemi più piccoli.

- Programmazione orientata agli oggetti

- 
- analisi del problema e descrizione dei concetti (oggetti) che ne fanno parte,
 - scomposizione del problema sui singoli oggetti e ricerca di un algoritmo risolutore,
 - scomposizione dell'algoritmo risolutore in un insieme di problemi più piccoli.

Oggetti

- Sono astrazioni che descrivono
 - oggetti fisici,
 - concetti astratti, che fanno parte del problema (o della soluzione).
- Caratterizzati da
 - uno **stato**,
 - un **insieme di servizi** che offrono agli altri oggetti.
- Esempio, un telefono cellulare
 - stato: carica della batteria, potenza del segnale, ...
 - servizi: chiama un numero, rispondi alla chiamata, ...

Classi di Oggetti (1/2)

- Un oggetto può essere descritto tramite le caratteristiche della **classe** (insieme, famiglia) di oggetti di cui fa parte
 - tutti i telefoni cellulari
 - hanno un'indicazione del livello della batteria e del segnale ricevuto,
 - consentono di rispondere e di chiamare.
- Spesso si dice **istanza** della classe X anziché dire oggetto di classe X.

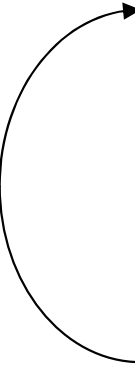
Classi di Oggetti (2/2)

- Un oggetto può appartenere a più classi
 - i telefoni cellulari sono anche oggetti fisici, e tutti gli oggetti fisici hanno un peso ed una posizione nello spazio.
- Tra oggetti è possibile stabilire delle **relazioni**
 - tutte le batterie hanno un'indicazione del livello di carica,
 - tutti i telefoni cellulari contengono una batteria.
- Tra classi è possibile stabilire delle relazioni
 - la classe dei telefoni cellulari è contenuta (sottoinsieme) nella classe degli oggetti fisici.

Sistema ad Oggetti

- Sistema procedurale
 - insieme di procedure che si chiamano l'una con l'altra.
- Sistema ad oggetti
 - insieme di oggetti che richiedono l'esecuzione di servizi ad altri oggetti del sistema.
- La complessità diminuisce tanto più
 - lo stato dei singoli oggetti è semplice,
 - l'insieme dei servizi offerti dai singoli oggetti è specifico (ma generale).

Sistema ad Oggetti in Java

- Nei linguaggi **class based** (come Java e C++)
 - gli oggetti vengono descritti solo tramite le loro classi,
 - gli oggetti vengono creati partendo da una classe,
 - un oggetto ha una sola classe origine.
 - Per realizzare un sistema ad oggetti in Java
 - descrizione delle classi
 - stato, mediante un insieme di **attributi**,
 - servizi, mediante un insieme di **metodi**,
 - relazioni con altre classi, mediante un insieme di attributi e metodi.
 - creazione degli oggetti del sistema e invio di messaggi.
- 

Classi in Java

```
public class CellularTelephone {  
    public void answer() {  
        ...  
    }  
  
    public void call(String number) {  
        ...  
    }  
  
    private int field = 0;  
    private int battery = 0;  
}
```

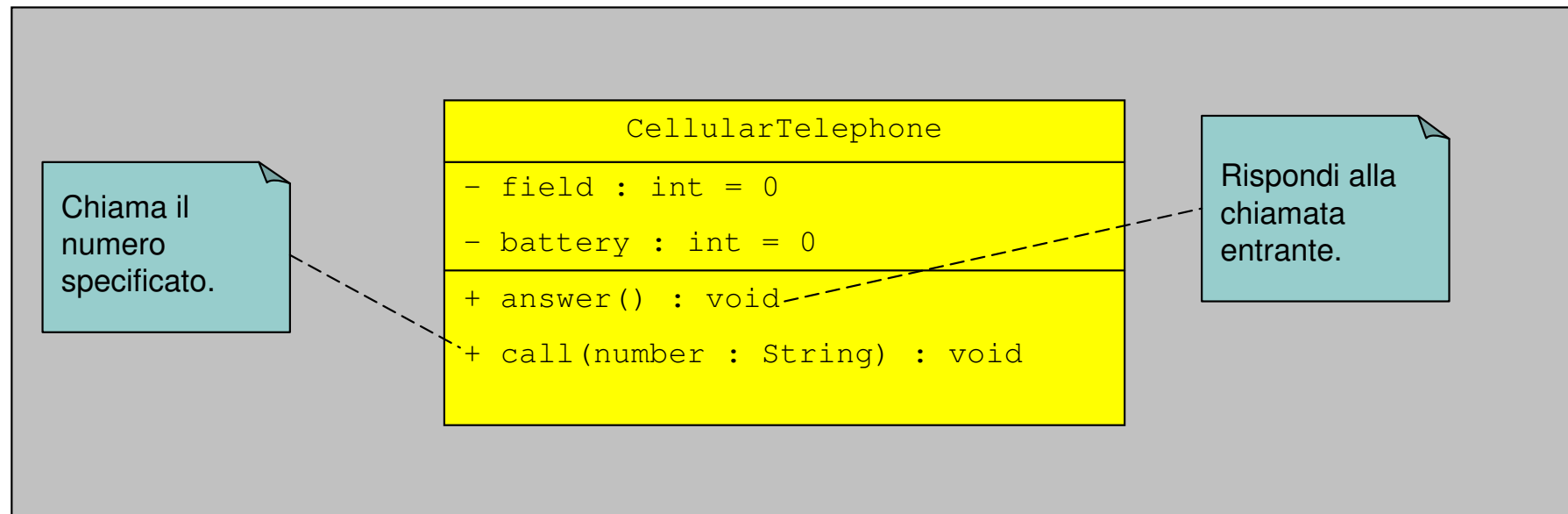
metodi che descrivono
i servizi

attributi che descrivono
lo stato

1/CellularTelephone.java

Classi con UML

Un diagramma che contiene classi e relazioni tra classi si chiama **class diagram**.



Esempio

Area dei Rettangoli

- Realizzare un sistema che calcola l'area e il perimetro di tre rettangoli.
- Primo passo è descrivere gli oggetti che fanno parte del problema
 - classe dei rettangoli.
- Secondo passo è creare gli oggetti ed inviare i messaggi necessari.

Classe dei Rettangoli

```
public class Rectangle {  
    public int area() {  
        return width*height;  
    }  
  
    public int perimeter() {  
        return 2*width + 2*height;  
    }  
  
    private int width = 0;  
    private int height = 0;  
}
```

Rectangle
- width : int = 0
- height : int = 0
+ area() : int
+ perimeter() : int

2/Rectangle.java

Compilazione di un Sorgente Java

- Tutti i sorgenti sono messi nella stessa directory ed hanno estensione '`.java`'.
- Se un file si chiama `X.java`, allora contiene una classe pubblica che si chiama `X`.
- Per compilare, con una shell DOS posizionata nella directory dei sorgenti
`javac <nome_del_file_compreso_.java>`
- Partendo da `X.java`, viene generato il bytecode `X.class`.

Generazione della Documentazione

- Per generare la documentazione partendo dai commenti nel codice, con una shell DOS posizionata nella directory dei sorgenti

```
javadoc -d docs <nome_del_file>
```
- I commenti contengono HTML e pseudo-tag tipo: `@param` e `@return`.

Creazione ed Utilizzo degli Oggetti

- Gli oggetti vengono creati partendo dalla loro classe di appartenenza
 - creazione di un oggetto rettangolo
`Rectangle obj = new Rectangle();`
 - invio di un messaggio all'oggetto `obj`
`obj.area();`
- La variabile `obj` è detta **object reference** ed è una sorta di puntatore all'oggetto rettangolo.

Inizializzazione degli Oggetti

- Problema: l'oggetto creato con `new` ha larghezza e altezza nulle.
- Prima soluzione (non soddisfacente)
 - introdurre i metodi

```
public int setWidth(int w) { width = w; }  
public int setHeight(int h) { height = h; }
```
 - far seguire a `new` l'inizializzazione dell'oggetto

```
Rectangle r = new Rectangle();  
r.setWidth(3);  
r.setHeight(4);
```

Metodi Getter e Setter (1/2)

- Spesso si introducono dei metodi che consentono di manipolare direttamente lo stato di un oggetto

```
public void setWidth(int w)    { width = w;    }  
public int  getSize()         { return size; }
```

- I metodi getter e setter dovrebbero essere minimizzati
 - solo l'oggetto è responsabile di come il suo stato viene memorizzato,
 - tutti i servizi utili che si basano sullo stato dell'oggetto dovrebbero essere offerti dall'oggetto stesso.
- I metodi **getter** e **setter** sono utili se non si limitano a leggere o scrivere variabili.

Metodi Getter e Setter (2/2)

- La soluzione dei setter ha vari problemi
 - i metodi setter sono stati aggiunti solo allo scopo di inizializzare l'oggetto,
 - il programmatore potrebbe dimenticarsi di invocarli dopo la costruzione dell'oggetto.
- È necessario che il linguaggio consenta di tenere unite la **costruzione** e l'**inizializzazione** degli oggetti.

Costruttori (1/2)

- Principio **instantiation is initialization**
La creazione di un oggetto coincide con la sua inizializzazione ad uno stato iniziale.
- Per soddisfare questo principio è necessario prevedere che venga chiamata una procedura d'inizializzazione, il **costruttore**, al momento della creazione.

```
public Rectangle(int w, int h) {  
    width = w; height = h;  
}
```

Costruttori (2/2)

- L'istruzione:
`new Rectangle (4, 5)`
ha l'effetto di
 - creare l'oggetto,
 - chiamare il costruttore dell'oggetto.
- Una classe può dichiarare più costruttori differenziati in base al numero e al tipo dei parametri.

Classe dei Rettangoli

```
public class Rectangle {  
    public int area() {  
        return width*height;  
    }  
  
    public int perimeter() {  
        return 2*width + 2*height;  
    }  
  
    public Rectangle(int w, int h) {  
        width = w; height = h;  
    }  
  
    private int width = 0;  
    private int height = 0;  
}
```

Rectangle
- width : int = 0
- height : int = 0
+ area() : int
+ perimeter() : int
+ Rectangle(w : int, h : int)

3/Rectangle.java

Metodo `Main` (1/2)

- Per avviare un sistema è necessario fornire alla JVM una procedura da eseguire.

- Questa procedura (metodo) si chiama:

```
public static void main(String[] args)
```

e può essere definita in ogni classe.

- Di solito si definisce una classe con il solo scopo di contenere il metodo `main`.

Metodo Main (2/2)

```
public class Main {  
    public static void main(String[] args) {  
        // Creazione dei tre rettangoli.  
        Rectangle r1 = new Rectangle(5, 7);  
        Rectangle r2 = new Rectangle(4, 8);  
        Rectangle r3 = new Rectangle(10, 10);  
  
        // Calcolo delle tre aree.  
        int area1 = r1.area();  
        int area2 = r2.area();  
        int area3 = r3.area();  
  
        // Stampa delle tre aree.  
        System.out.println("Area del rettangolo 1: " + area1);  
        System.out.println("Area del rettangolo 2: " + area2);  
        System.out.println("Area del rettangolo 3: " + area3);  
    }  
}
```

3/Main.java

Esecuzione

- Per eseguire un'applicazione Java, con una shell DOS posizionata nella directory contenenti tutti i bytecode, lanciare

```
java <nome_della_classe_contente_un_main>
```

- Il comando `java X <argomenti>` ha l'effetto di
 - eseguire il metodo `main` della classe `X`
 - passare come parametri al `main` le parole della stringa `<argomenti>` spezzate utilizzando gli spazi come separatori.

Distruzione degli Oggetti

- La creazione di un oggetto richiede memoria per memorizzare lo stato dell'oggetto.
- Quando un oggetto non serve più è necessario liberare questa memoria.
- La JVM offre un **garbage collector** per gestire in modo automatico la restituzione della memoria.
- Quando un oggetto non serve più, il garbage collector lo distrugge.

Garbage Collector

- Vantaggi
 - non è possibile dimenticare di liberare la memoria,
 - non è possibile liberare della memoria che dovrà essere utilizzata in seguito.
- Svantaggi
 - il garbage collector decide autonomamente quando liberare la memoria,
 - liberare e compattare la memoria richiede del calcolo.

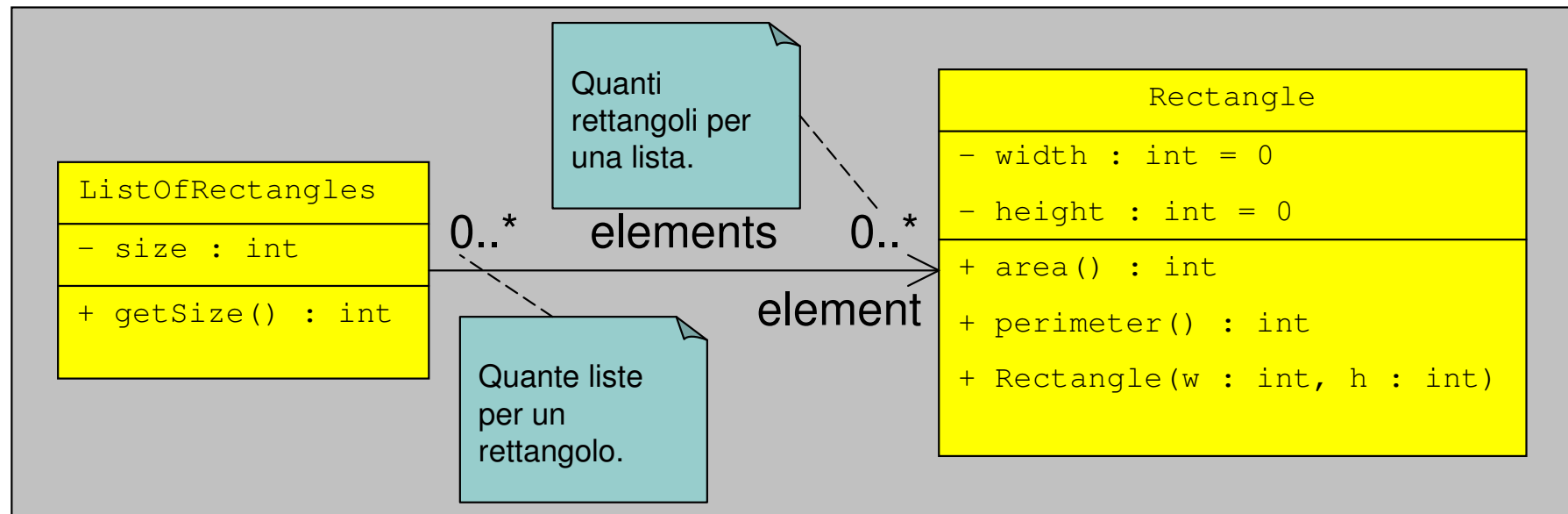
Esempio

Lista di Rettangoli

- Realizzare un programma che stampi l'area di 10 rettangoli contenuti in una lista.
- Classi necessarie
 - classe dei rettangoli,
 - classe delle liste di rettangoli.

Associazioni (1/4)

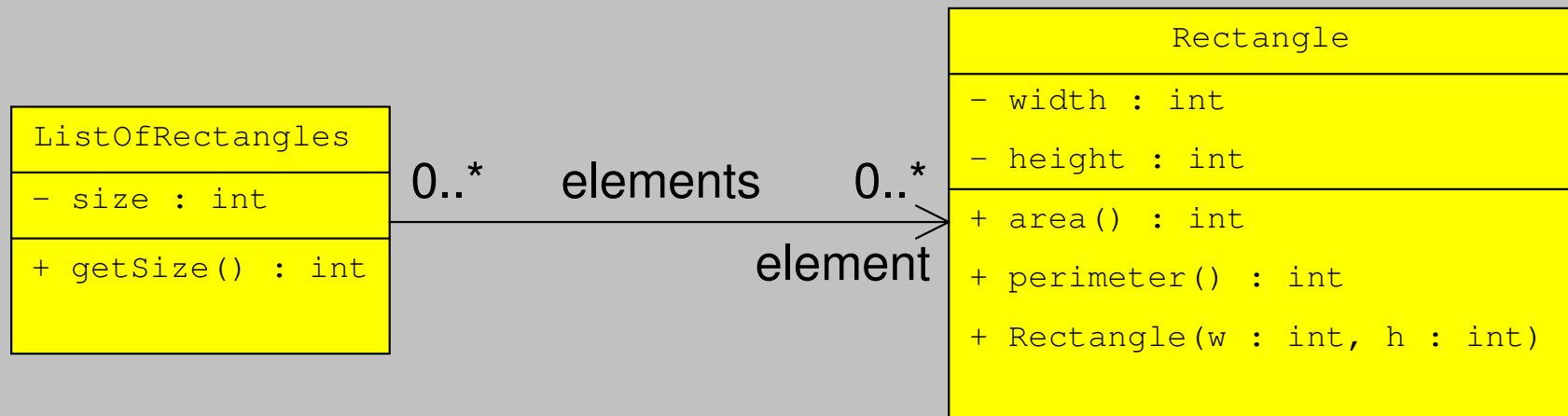
- C'è una relazione tra un oggetto lista di rettangoli e un insieme di rettangoli.
- Una relazione tra oggetti genera un'associazione tra le classi.



Associazioni (2/4)

- Un'associazione ha un nome che la descrive.
- Ognuno dei lati
 - ha un nome,
 - ha una cardinalità,
 - può o no essere navigabile.
- Un'associazione equivale a codice Java.

Associazioni (3/4)



```
public class ListOfRectangles {
    public void      addElement(Rectangle r) { elements[size++] = r; }
    public Rectangle getElement(int i)      { return elements[i]; }

    private Rectangle[] elements;
}
```

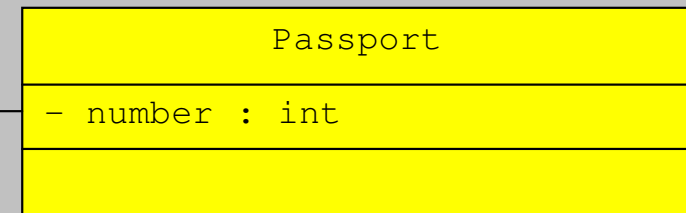
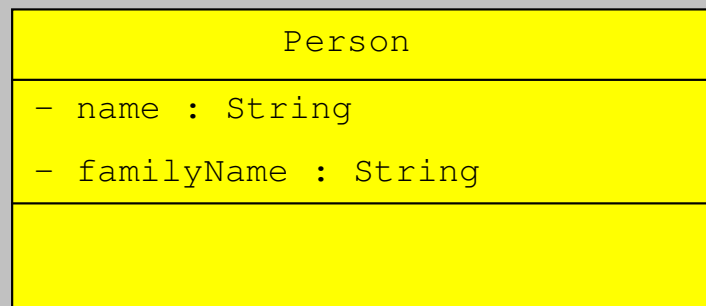
4/ListOfRectangles.java

Associazioni (4/4)

- È importante notare che
 - solo i lati navigabili implementano l'associazione,
 - è necessario controllare il vincolo sulla cardinalità nei metodi `addElement()` e `removeElement()`,
 - anziché un array è possibile implementare l'associazione con un **oggetto aggregato** (esempio `java.util.ArrayList` o `java.util.TreeSet`).

Associazione 1 a 1

- Un'associazione particolarmente importante è l'associazione 1 a 1.



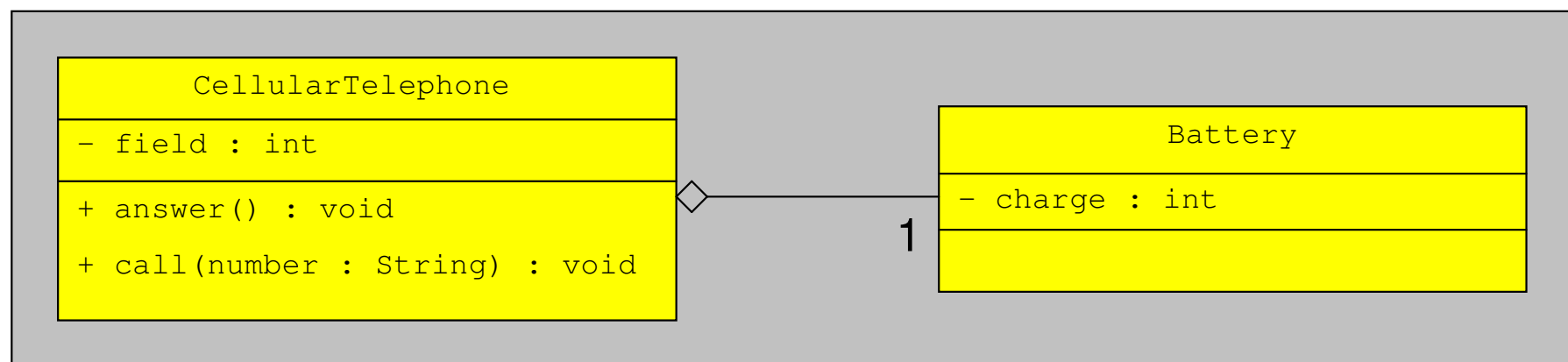
```
public class Person {
    public Passport getPassport()          { return passport; }
    public void      setPassport(Passport p) { passport = p;    }

    private Passport passport = null;
}
```

5/Person.java

Contenimento

- Se un'associazione esprime un contenimento fisico, o una relazione **part-of** o una relazione **has-a**, allora si parla di **relazione di contenimento**.
- In Java non c'è nessuna vera differenza tra il contenimento ed un più generica associazione.



Associazioni e Attributi

- Attributi e associazioni generano codice
 - le associazioni vengono usate per collegare tra loro le classi del problema (o della soluzione),
 - negli altri casi
 - tipi primitivi, come `int` o `float`,
 - classi della Java library, come `String` o `Object`,
 - classi di library che non si vogliono includere esplicitamente nel problema (o nella soluzione).si utilizzano gli attributi.

Esempio

Ordinamento dei Rettangoli

- Realizzare un programma che ordini una lista di 10 rettangoli in base alla loro area.
- Classi necessarie
 - classe dei rettangoli,
 - classe delle liste di rettangoli.
- L'ordinamento verrà effettuato richiedendo il servizio 'ordinati' ad un oggetto di classe lista di rettangoli.

Implementazione del Servizio di Ordinamento

```
public class ListOfRectangles {
    public void sort() {
        for(int i = 0; i < size; i++)
            for(int j = i + 1; j < size; j++) {
                Rectangle left  = elements[i];
                Rectangle right = elements[j];

                if(left.area() > right.area()) swap(i, j);
            }
    }

    private void swap(int i, int j) {
        Rectangle t = elements[i];

        elements[i] = elements[j]; elements[j] = t;
    }
}
```

6/ListOfRectangles.java

Metodi Privati

- Il metodo `swap()` è privato
 - non è un servizio che l'oggetto offre ad altri oggetti,
 - può essere chiamato solo all'interno dell'oggetto stesso.
- I metodi privati sono assimilabili a procedure o funzioni nell'approccio procedurale.

Classi o Metodi?

- Esistono vari algoritmi di ordinamento applicabili ad una lista, il metodo `sort()` utilizza il bubble sort.
- Problema: abbiamo legato l'algoritmo alla struttura dati.
- Quando si vuole implementare un algoritmo che può essere **sostituito**, conviene utilizzare una classe anzichè un metodo.

Implementazione del Servizio di Ordinamento

```
public class BubbleSorter {
    public void sort(ListOfRectangles list) {
        for(int i = 0; i < list.getSize(); i++)
            for(int j = i + 1; j < list.getSize(); j++) {
                Rectangle left  = list.getElement(i);
                Rectangle right = list.getElement(j);

                if(left.area() > right.area()) swap(list, i, j);
            }
    }

    private void swap(ListOfRectangles list, int i, int j) {
        Rectangle t = list.getElement(i);

        list.setElement(i, list.getElement(j));
        list.setElement(j, t);
    }
}
```

7/BubbleSorter.java

Accesso allo Stato (1/2)

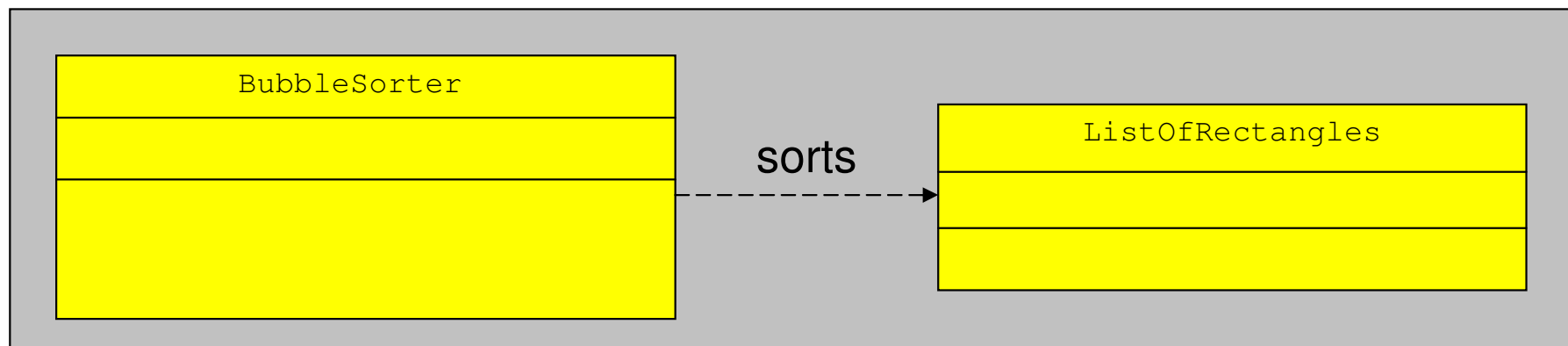
- Per implementare il `BubbleSorter` dobbiamo dare accesso agli elementi della lista
 - metodi: `setElement()`, `getElement()`.
- È importante che l'accesso agli elementi interni non sveli come la lista memorizza i propri elementi (array, lista concatenata, ...)
 - possibilità di cambiare la memorizzazione interna,
 - aumenta la riusabilità perchè `BubbleSorter` è applicabile indipendentemente da come la lista è memorizzata.

Accesso allo Stato (2/2)

- Principio di **information hiding**
Bisogna nascondere tutti i dettagli implementativi di un oggetto.
- Consente di rendere
 - insensibili gli utilizzatori rispetto ad eventuali cambiamenti dell'implementazione,
 - una classe pubblicabile senza mostrare dettagli implementativi.

Dipendenza (1/2)

- Il `BubbleSorter` prevede che
 - esista nel sistema una classe `ListOfRectangles`,
 - la classe `ListOfRectangles` abbia i metodi `getElement()`, `setElement()` e `getSize()`.
- `BubbleSorter` **dipende** da `ListOfRectangles`.



Dipendenza (2/2)

- Una classe dipende da servizi di un'altra classe, ma non ha nessuna struttura interna che dipende da quest'ultima
 - la forma più comune di dipendenza si ha quando un metodo di una classe come tipo di un argomento un'altra classe con cui non è associata.
- Dipendenze e associazioni limitano la riusabilità perchè **accoppiano** le classi.

Modelli Dinamici con UML

- UML mette a disposizione due tipi di diagrammi (equivalenti), detti **interaction diagram**, per descrivere in modo grafico le interazioni tra gli oggetti
 - **sequence diagram**, descrivono una sequenza di interazione tra più oggetti,
 - **collaboration diagram**, descrivono una sequenza di interazione tra più oggetti mettendone in evidenza le relazioni.

Esempio

Parte Centrale del Bubble Sort

- In Java

```
Rectangle left  = list.getElement(i);  
Rectangle right = list.getElement(j);
```

```
if(left.area() > right.area()) swap(i, j);
```

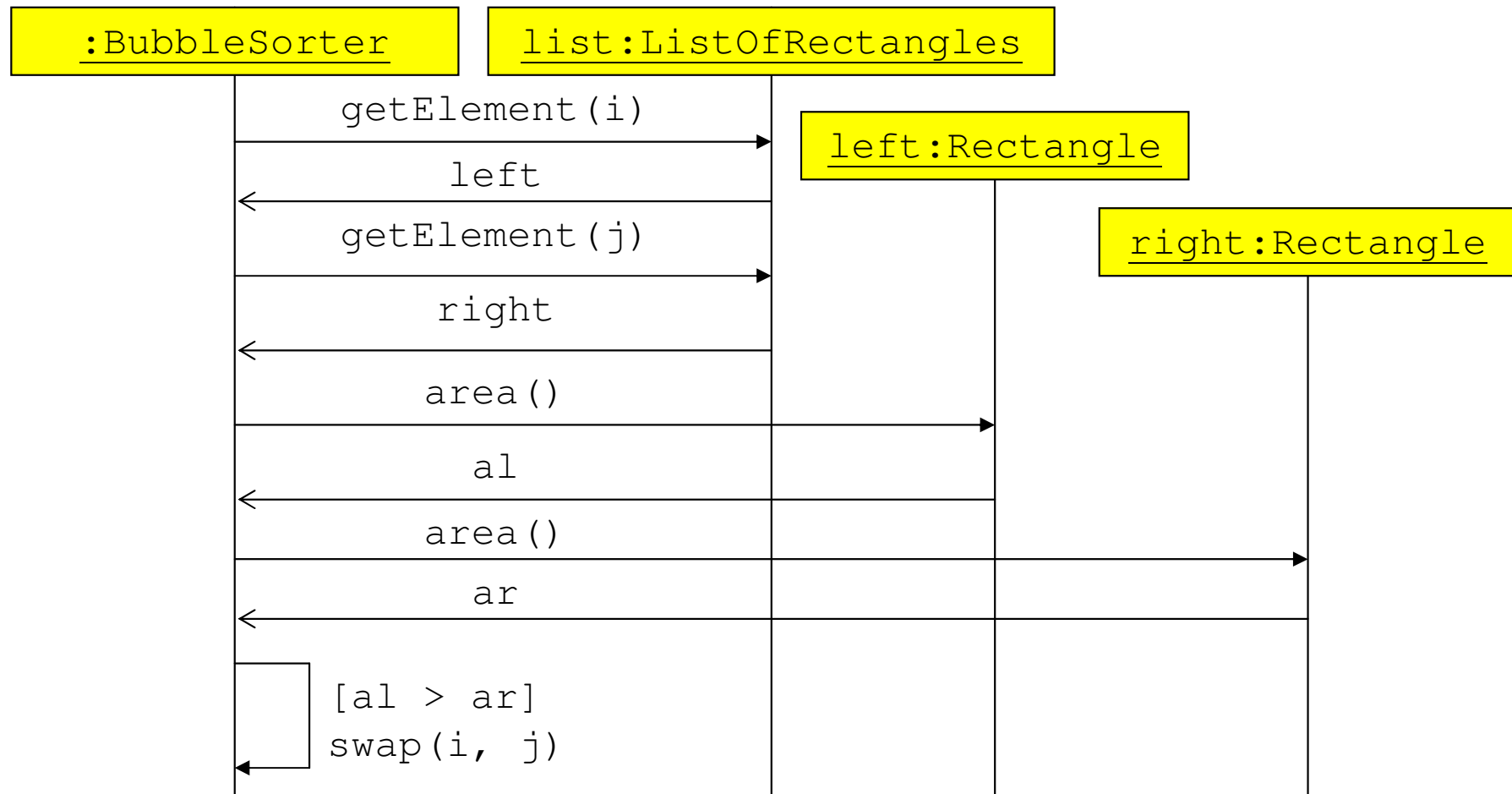
- Con uno pseudo-codice

```
Rectangle left  = i-esimo elemento di list;  
Rectangle right = j-esimo elemento di list;
```

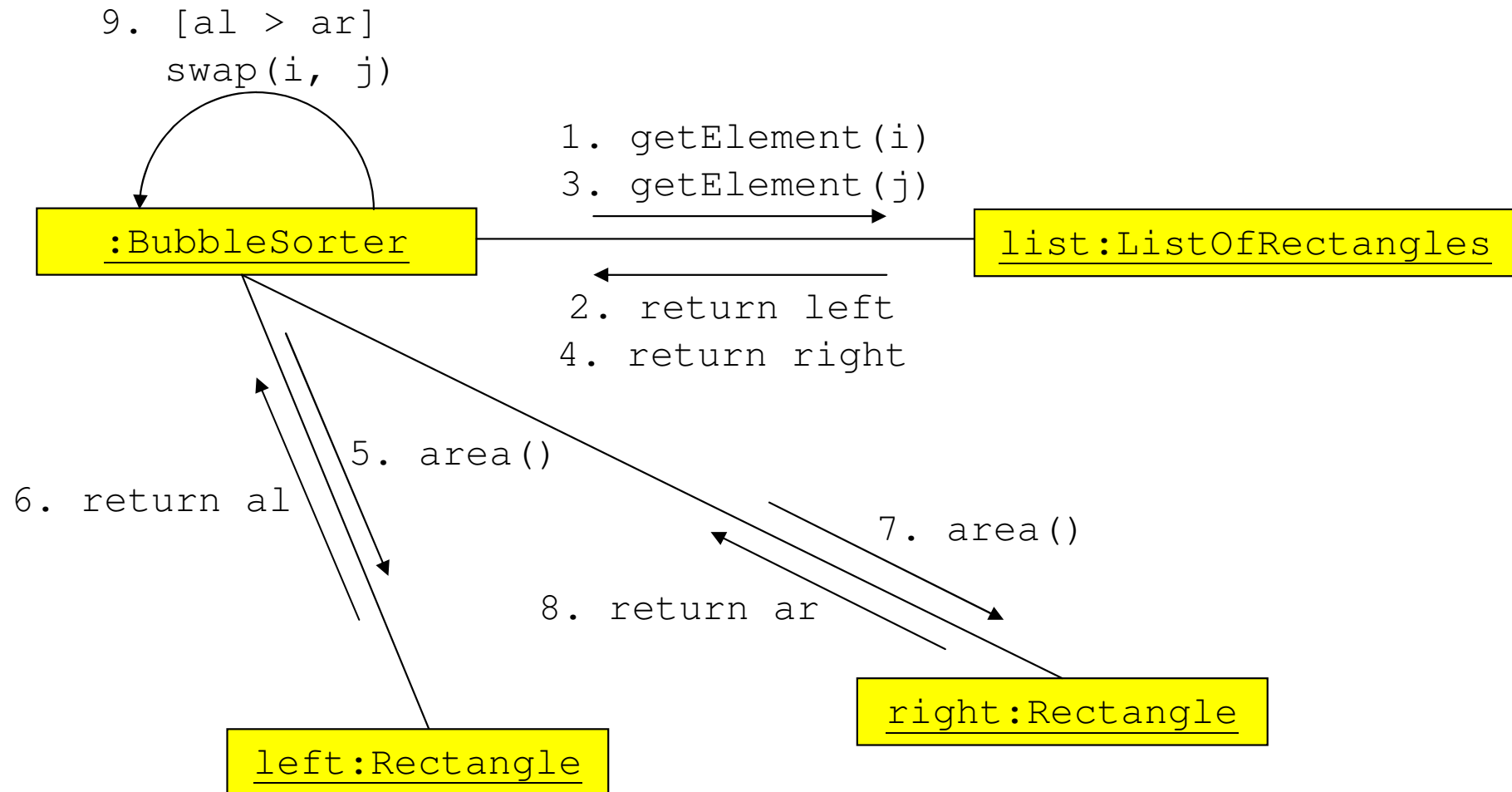
```
int al = area di left;  
int ar = area di right;
```

```
if (al > ar) swap(i, j);
```

Sequence Diagram



Collaboration Diagram



Interaction Diagram

- **Vantaggio**
 - rappresentazione grafica utile per comprendere meglio interazioni complesse.
- **Svantaggi**
 - non sufficientemente espressivi,
 - potenzialmente ambigui,
 - più ridondanti di uno pseudo-codice.
- **Utili**
 - in fase di analisi e di progettazione ad alto livello,
 - come documentazione.

Ereditarietà (1/3)

- Tra le classi è possibile definire una relazione di sotto ~~classe~~ (sotto insieme)
 - classi `Animale`, `Felino`, `Gatto`,
 - `Gatto` sotto-classe (classe **derivata**) di `Felino`,
`Gatto` sotto-classe (classe derivata) di `Animale`.
 - `Animale` classe **base** di `Felino` e di `Gatto`.
- Consente di comporre un insieme di classi per creare una nuova classe.

Ereditarietà (2/3)

- La sotto-classe
 - eredita tutte le caratteristiche della classe base,
 - non può accedere alle caratteristiche `private` della classe base,
 - può dichiarare nuove caratteristiche che non sono visibili dalle classi base.
- La classe base
 - può definire delle caratteristiche `protected` a cui solo lei e le sotto-classi possono accedere.

Ereditarietà (3/3)

- Principio di **sostituibilità**

*L'ereditarietà implementa la relazione **is-a**: è sempre possibile usare un oggetto di una sotto-classe al posto di un oggetto di una classe base.*

```
list.addElement(new Rectangle(5, 7));  
...  
list.addElement(new Square(6));  
list.addElement(new Square(200));  
  
BubbleSorter sorter = new BubbleSorter();  
sorter.sort(list);
```

8/Main.java

Clausola `protected`

- Consente di dichiarare metodi e attributi accessibili solo dalla classe e dalle derivate
 - non sono visibili all'esterno dell'oggetto, come se fossero privati,
 - consentono di rendere visibili alcune parti di una classe alle sue sotto-classi,
- Da utilizzare con cautela. Solo per metodi
 - che non conviene rendere pubblici perchè non offrono servizi significativi, come `swap()` in `BubbleSorter`,
 - potenzialmente utili a chi estende la classe.

Ereditarietà in Java

- Una classe derivata viene costruita partendo dalla classe base

```
public class Square extends Rectangle {  
    ...  
    caratteristiche aggiuntive  
    ...  
}
```

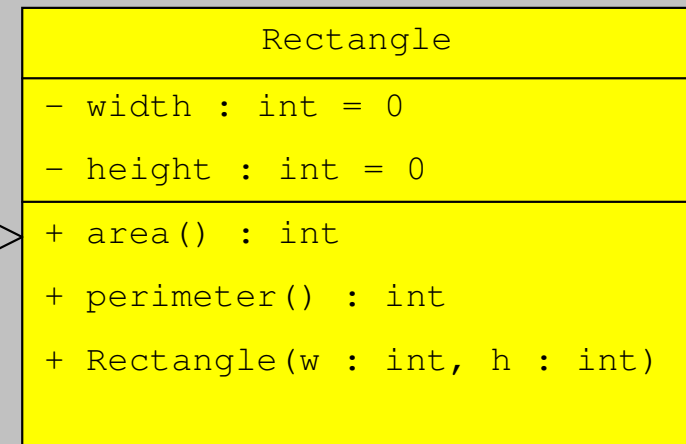
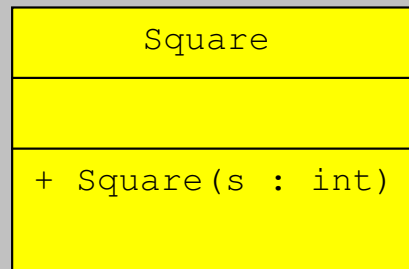
- L'ereditarietà tra le classi Java è **singola**: solo una classe base per ogni classe derivata.
- Tutte le classi sono derivate (implicitamente) da `Object`.

```
Object obj = new Rectangle(4, 3);  
obj.equals(new Rectangle(5, 5));
```

Ereditarietà e Costruzione (1/3)

- I costruttori non vengono ereditati
 - non contengono il codice di costruzione degli attributi aggiunti nella classe derivata.
- È possibile accedere ai costruttori della classe base mediante `super`.
- `super` deve essere chiamato all'inizio del nuovo costruttore
 - prima è necessario costruire completamente le caratteristiche della classe base per poi passare a quelle della classe derivata.

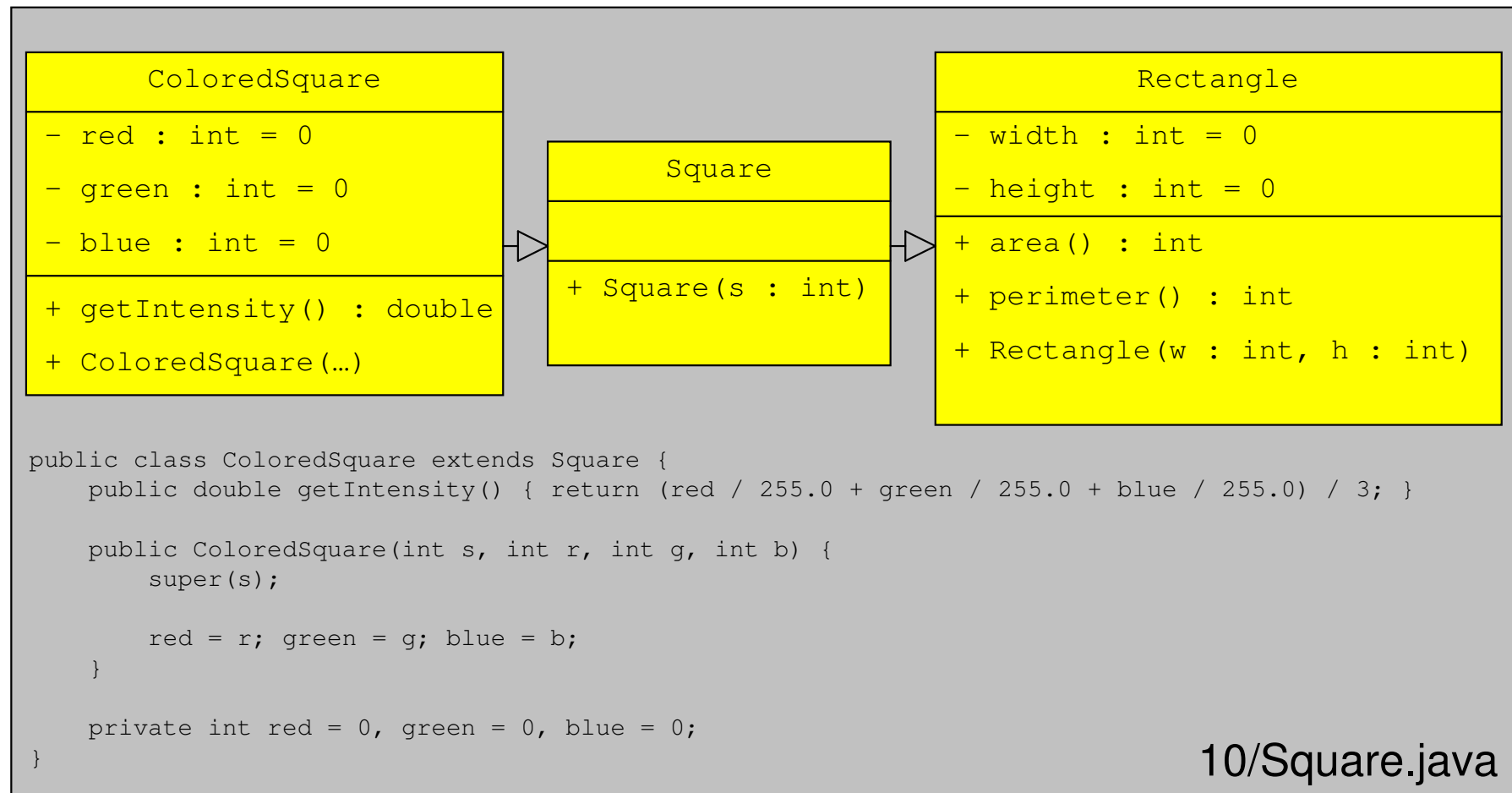
Ereditarietà e Costruzione (2/3)



```
public class Square extends Rectangle {  
    public Square(int s) {  
        super(s, s);  
  
        ...costruzione degli attributi aggiunti...  
    }  
}
```

9/Square.java

Ereditarietà e Costruzione (3/3)



Polimorfismo

- Un metodo della classe base può essere ridefinito nelle classi derivate (metodo **polimorfo**).

```
public class Square extends Rectangle {  
    public int area() { return getWidth()*getWidth(); }  
    public int perimeter() { return 4*getWidth(); }  
}
```

```
Rectangle r1 = new Rectangle(3, 4);  
Rectangle r2 = new Square(6);
```

```
r1.area(); // invoca il metodo di Rectangle  
r2.area(); // invoca il metodo di Square
```

9/Square.java

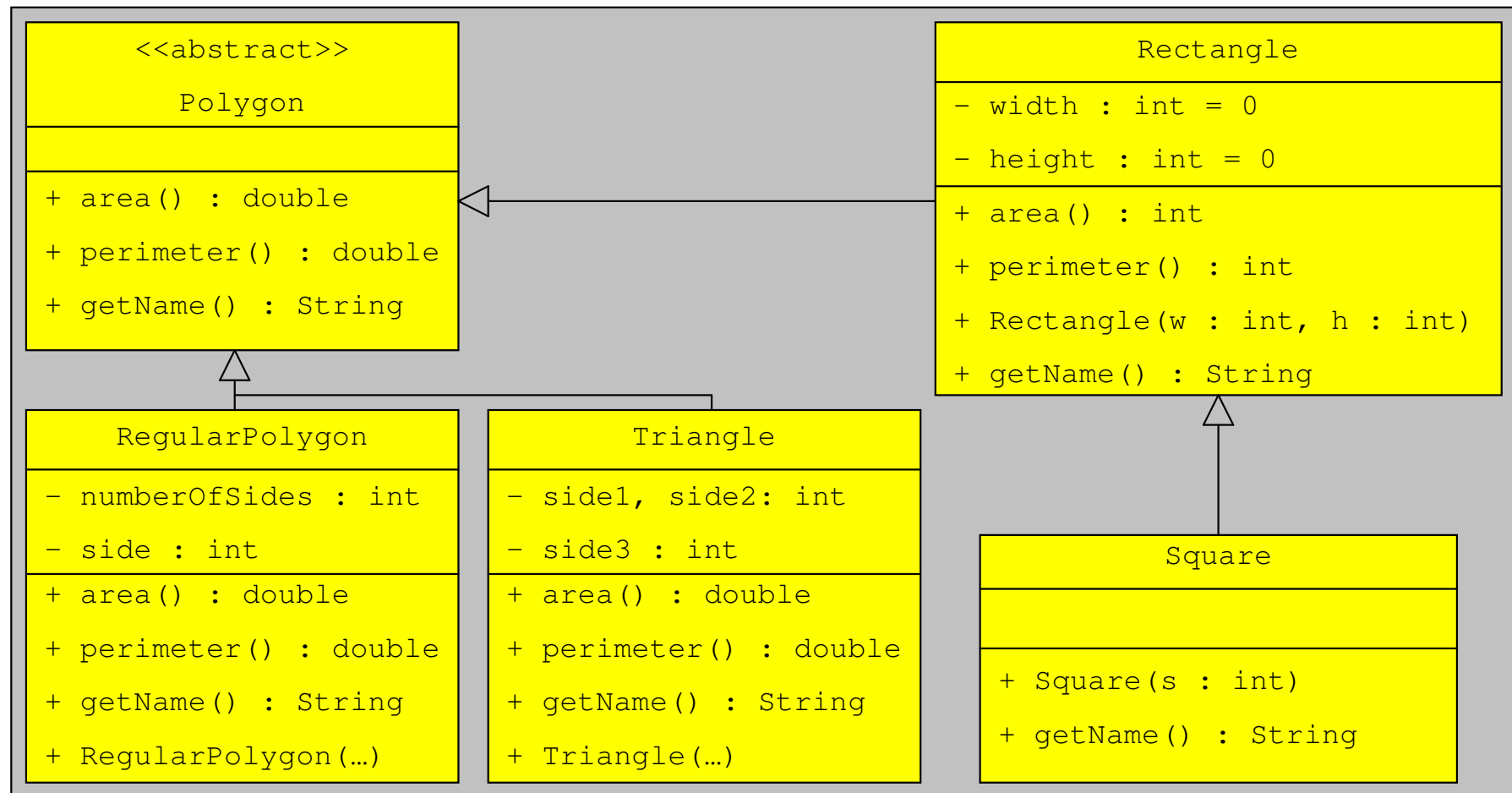
Reference `this`, `super` e `null`

- `null`, punta ad un oggetto nullo
 - una reference punta ad un oggetto valido o a `null`.
- `this`, punta all'oggetto corrente
 - nella classe `Rectangle`, è possibile chiamare `this.area()` o `this.perimeter()`.
- `super`, punta all'oggetto corrente e consente di accedere alle caratteristiche della classe base
 - nella classe `Square` è possibile utilizzare il metodo `area()` di `Rectangle` mediante `super.area()`.

Classi Astratte (1/3)

- Se una classe contiene metodi che possono essere implementati solo dalle sue sotto-classi viene detta **astratta**
 - informazioni non sufficienti nella classe base,
 - meccanismi specifici per implementare un metodo,
 - ...
- Una classe astratta non può essere istanziata.

Classi Astratte (2/3)



Classi Astratte (3/3)

```
public abstract class Polygon {
    public abstract double area();
    public abstract double perimeter();

    public abstract String getName();
}

public class Triangle extends Polygon {
    public double area() {
        double s = perimeter()/2;
        double a2 = s*(s-side1)*(s-side2)*(s-side3);

        return Math.sqrt(a2);
    }

    public double perimeter() { return side1 + side2 + side3; }

    public String getName() { return "Triangle"; }

    private int side1 = 0, side2 = 0, side3 = 0;
}
```

11/Polygon.java, Triangle.java

Ereditarietà e Miglioramento della Riutilizzabilità

- Strutture dati ed algoritmi possono essere implementati in funzione della classe `Polygon` anziché di `Rectangle`
 - `ListOfPolygons`,
 - `BubbleSorter` sfrutta il fatto che tutti i poligoni hanno un `area`.
- Esprimere strutture dati ed algoritmi in funzione della classe più alta nella gerarchia che offra le caratteristiche richieste consente di massimizzare il riuso.

Generalizzazione delle Strutture Dati

```
public class ListOfPolygons {  
    public void addElement(Polygon r)          { elements[size++] = r; }  
    public void setElement(int i, Polygon r)   { elements[i] = r;          }  
    public Polygon getElement(int i)           { return elements[i];    }  
  
    public int getSize() { return size; }  
  
    public ListOfPolygons() { elements = new Polygon[100]; }  
  
    private int size = 0;  
  
    private Polygon[] elements;  
}
```

11/ListOfPolygons.java

Generalizzazione dell'Algoritmo di Ordinamento

```
public class BubbleSorter {  
    public void sort(ListOfPolygons list) {  
        for(int i = 0; i < list.getSize(); i++)  
            for(int j = i + 1; j < list.getSize(); j++) {  
                Polygon left  = list.getElement(i);  
                Polygon right = list.getElement(j);  
  
                if(left.area() > right.area()) swap(list, i, j);  
            }  
    }  
  
    private void swap(ListOfPolygons list, int i, int j) {  
        Polygon t = list.getElement(i);  
  
        list.setElement(i, list.getElement(j));  
        list.setElement(j, t);  
    }  
}
```

11/BubbleSorter.java

Utilizzo delle Classi Generalizzate

```
public class Main {
    public static void main(String[] args) {
        // Creazione della lista dei poligoni.
        ListOfPolygons list = new ListOfPolygons();

        // Creazione dei poligoni e aggiunta alla lista.
        list.addElement(new Rectangle(5, 7));
        list.addElement(new Square(6));
        list.addElement(new Triangle(3, 4, 5));
        list.addElement(new RegularPolygon(5, 7));

        // Creazione di un oggetto ordinatore
        BubbleSorter sorter = new BubbleSorter();

        // Ordinamento della lista.
        sorter.sort(list);

        // Stampa delle aree degli elementi della lista.
        Printer printer = new Printer();
        printer.print(list);
    }
}
```

11/Main.java

Ereditarietà Multipla

- Un quadrato è contemporaneamente
 - un rettangolo,
 - un poligono regolare.
- La classe `Square` dovrebbe estendere sia `RegularPolygon` che `Rectangle`
 - quale stato usare?
 - quale metodi chiamare?
- Java non permette l'ereditarietà multipla tra le classi.

Interfacce

- L'interfaccia di un oggetto è l'insieme delle **signature** dei metodi pubblici della sua classe
 - modo che ha per interagire con il mondo,
 - servizi che offre agli altri oggetti.
- I corpi dei metodi, cioè come i servizi vengono **implementati**, non sono parte dell'interfaccia
 - l'interfaccia indica cosa un oggetto sa fare e non come lo fa.
- L'interfaccia di un rettangolo è
 - `double area()`,
 - `double perimeter()`.
- In Java, le interfacce vengono implementate da classi.

Interface in Java (1/3)

```
public interface Polygon {
    public double area();
    public double perimeter();
    public String getName();
}

public interface RegularPolygon extends Polygon {}

public class ConcreteRegularPolygon implements RegularPolygon {
    public double area()      { ... }
    public double perimeter() { return numberOfSides*side; }
    public String getName()   { return "Regular polygon"; }

    public ConcreteRegularPolygon(int n, int s) { ... }

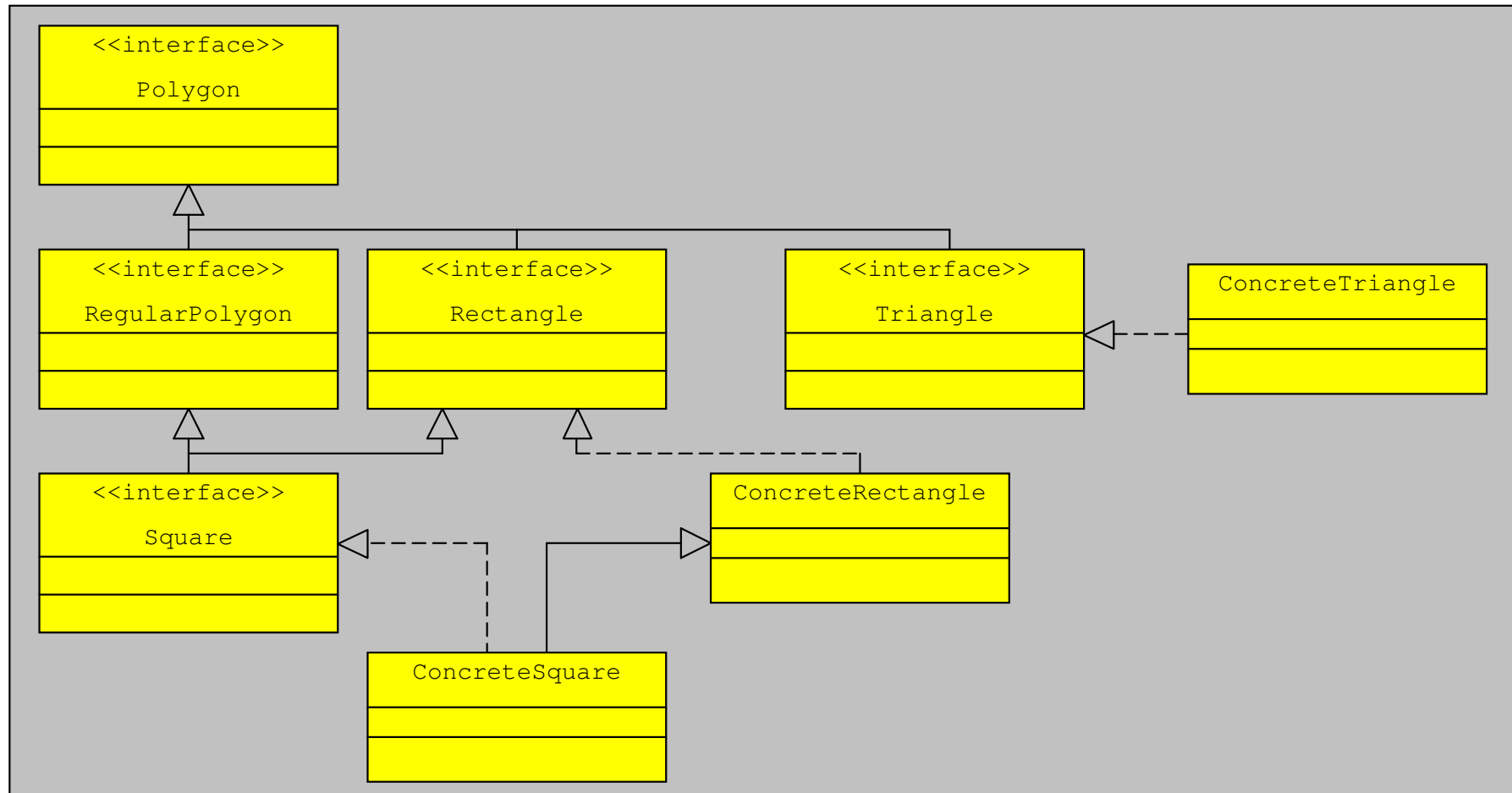
    private int numberOfSides = 0;
    private int side = 0;
}
```

12/Triangle.java,
RegularPolygon.java,
ConcreteRegularPolygon.java

Interfacce in Java (2/3)

- Java permette l'ereditarietà multipla tra le interfacce
 - è la classe implementazione che sceglie come memorizzare lo stato ed implementare i metodi.
- Usando le interfacce
 - migliore pulizia del modello ed aderenza alla realtà modellizzata,
 - possibilità di migliorare la riusabilità.
- Introdurre un'interfaccia per ogni classe non è l'approccio migliore.

Interface in Java (3/3)



A Cosa Serve l'Ereditarietà?

- Due utilizzi principali
 - modellizzare il problema (o la soluzione), molto importante nella fase di analisi,
 - massimizzare il riuso, molto importante nella fase di progettazione.
- I due utilizzi sono legati perchè la prima bozza di un progetto è il modello che analizza il **dominio del problema** (o della soluzione).

Due Modi per Massimizzare il Riuso

- Aggiungendo nuovi servizi a classi già esistenti
 - si riutilizza il codice della classe esistente,
 - si possono sfruttare anche parti `protected` della classe base.
- Costruendo sistemi che vengono specializzati mediante oggetti che implementano ben determinate interfacce
 - si riutilizza un intero sistema, andando a modificarne il comportamento,
 - oggi, questo meccanismo di riuso è il più apprezzato.

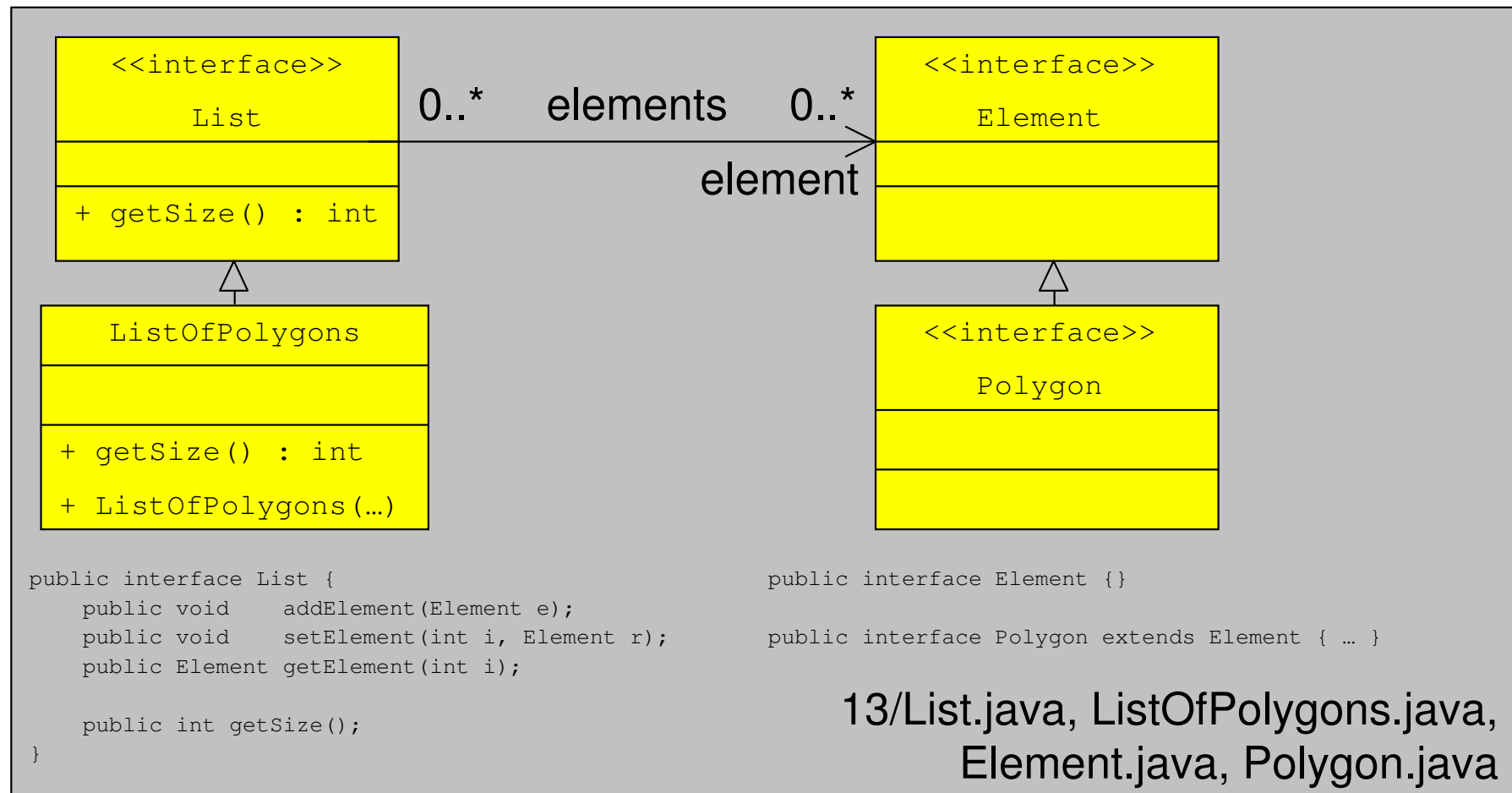
Generalizzazione del Servizio di Ordinamento (1/2)

- Il servizio attuale ordina solo
 - poligoni,
 - in base all'area.
- Per massimizzarne la riusabilità bisogna agire su entrambe queste limitazioni
 - definendo una gerarchia di classi con il solo scopo di rendere l'algoritmo personalizzabile.
- Il riuso non viene dalla possibilità di riutilizzare parti dell'algoritmo di ordinamento, ma dall'aver reso l'algoritmo personalizzabile.

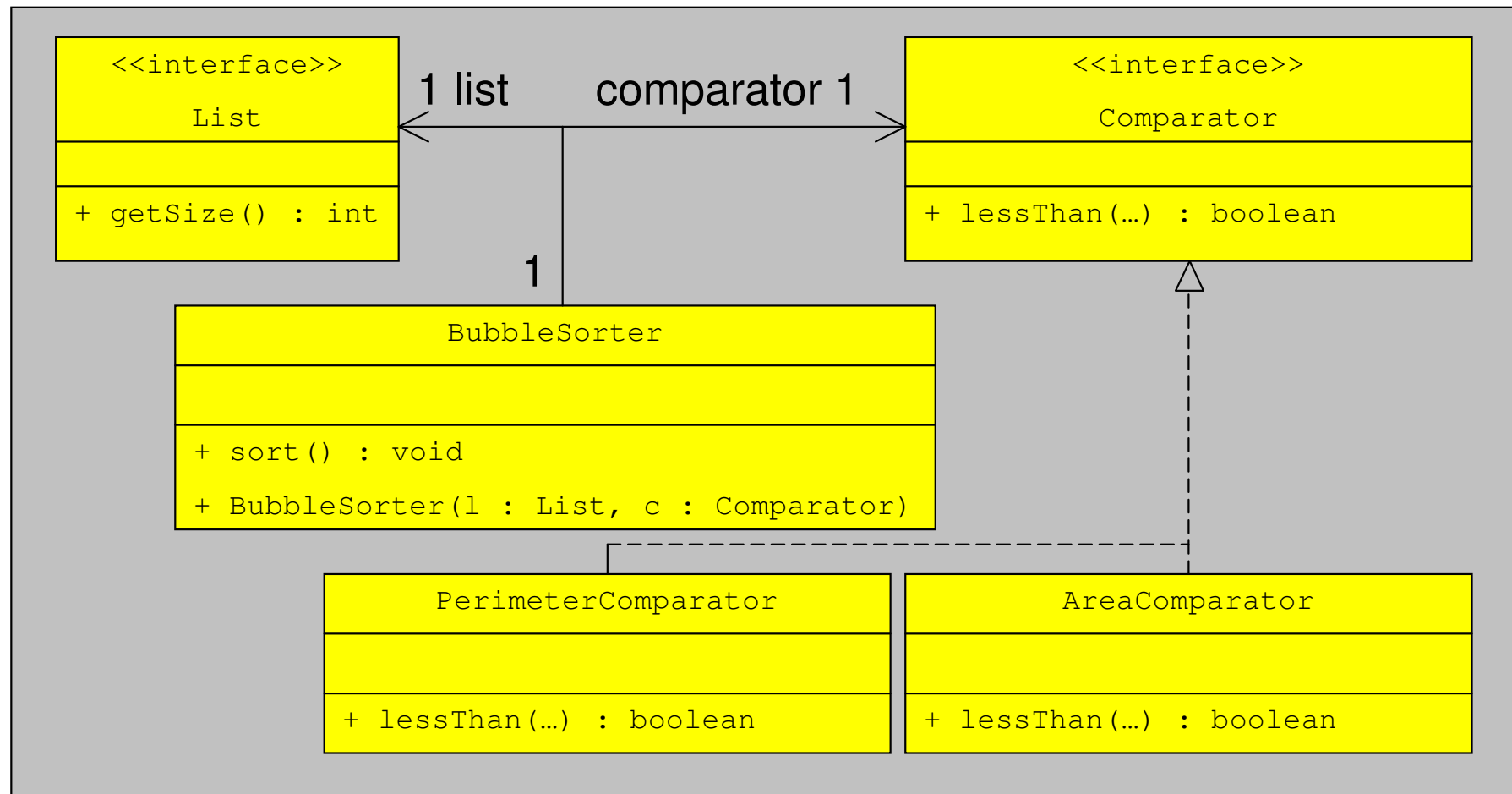
Generalizzazione del Servizio di Ordinamento (2/2)

- Introduciamo le classi
 - `List`, interfaccia che modella un tipo dato aggregato in cui è possibile accedere agli elementi mediante un indice,
 - `Element`, interfaccia che implementano gli elementi di una lista,
 - `Comparator`, interfaccia di oggetti capaci di confrontare due oggetti in base a qualche caratteristica.

Una Lista Astratta



Un BubbleSorter Personalizzabile (1/2)



Un BubbleSorter Personalizzabile (2/2)

```
public class BubbleSorter {
    public void sort() {
        for(int i = 0; i < list.getSize(); i++)
            for(int j = i + 1; j < list.getSize(); j++) {
                Element left = list.getElement(i);
                Element right = list.getElement(j);

                if(!comparator.lessThan(left, right)) swap(i, j);
            }
    }

    public BubbleSorter(List l, Comparator c) {
        list = l;
        comparator = c;
    }

    private void swap(int i, int j) {
        Element t = list.getElement(i);

        list.setElement(i, list.getElement(j));
        list.setElement(j, t);
    }

    private Comparator comparator = null;
    private List list = null;
}
```

13/BubbleSorter.java

Up-cast (1/2)

- L'interfaccia `Comparator` è espressa in modo generale.
- Per implementarla è necessario convertire gli `Element` da confrontare in `Polygon` per poter calcolare area e perimetro.
- Java permette di assegnare un oggetto appartenente ad una classe base ad una reference ad un oggetto di una classe derivata mediante un **up-cast**

```
ClasseBase      b = new ClasseDerivata();  
ClasseDerivata d = (ClasseDerivata)b;
```

- La classe origine di `b` deve essere `ClasseDerivata` o una sua sotto-classe, altrimenti la JVM genera un'eccezione a run-time.

Up-cast (2/2)

```
public interface Comparator {
    public boolean lessThan(Element e1, Element e2);
}

public class AreaComparator implements Comparator {
    public boolean lessThan(Element e1, Element e2) {
        Polygon p1 = (Polygon)e1;
        Polygon p2 = (Polygon)e2;

        return p1.area() < p2.area();
    }
}

public class PerimeterComparator implements Comparator {
    public boolean lessThan(Element e1, Element e2) {
        Polygon p1 = (Polygon)e1;
        Polygon p2 = (Polygon)e2;

        return p1.perimeter() < p2.perimeter();
    }
}
```

13/Comparator.java,
AreaComparator.java,
PerimeterComparator.java

Struttura di un Progetto (1/2)

- Quando la complessità del progetto aumenta
 - aumenta il numero di file (sorgenti e bytecode),
 - conviene strutturare il progetto in moduli isolati e redistribuibili, detti **package**.
- Un package è un insieme di file (sorgente o bytecode) che sono logicamente visti come un unico gruppo
 - è possibile strutturare i package come alberi,
 - un sotto package isola un gruppo di funzionalità del suo package padre.
- I file di un package
 - devono essere tutti nella stessa directory, che si deve chiamare con il nome (completo) del package,
 - indicano esplicitamente di che package fanno parte.

Struttura di un Progetto (2/2)

- Solitamente la struttura delle directory di un progetto in Java è

<code>src/</code>	i sorgenti,
<code>classes/</code>	i bytecode,
<code>images/</code>	eventuali immagini,
<code>lib/</code>	eventuali library esterne che si intende utilizzare.

- All'interno della directory `src` si mette una directory per ogni package che si intende utilizzare

<code>src/editor</code>	i file iniziano con package editor;
<code>src/editor/model</code>	i file iniziano con package editor.model;
<code>src/editor/view</code>	i file iniziano con package editor.view;

- La directory `classes` rispecchia la struttura di `src`.

Library

- Una **library** è un insieme di package che implementano funzionalità comuni
 - tutte le classi con funzionalità matematiche avanzate,
 - tutte le classi necessarie a gestire l'input/output,
 - tutte le classi dell'interfaccia grafica.
- Solitamente le library sono distribuite come file di tipo **Java Archive (JAR)**
 - è uno ZIP del contenuto della directory `classes/` del progetto della library,
 - possono essere gestiti come file ZIP o mediante il comando `jar`.

Compilazione di un Progetto

- Per compilare un progetto è necessario includere
 - tutte le directory dei sorgenti,
 - tutte le library esterne.
- La linea di comando diventa

```
java -d classes -classpath  
classes;lib\<library1>.jar;lib\<library1>.jar;...  
src/<package1>/*.java src/<package2>/*.java ...
```

Esecuzione del Programma

- Per eseguire il programma è necessario includere
 - tutte le directory contenenti i bytecode,
 - tutti i file JAR delle library utilizzate,
 - tutte le directory di eventuali risorse.
- Questa lista è nota come **classpath**.
- La linea di comando diventa

```
java -classpath  
classes;images;lib\<library1>.jar;lib\<library2>.jar;...  
<package>.Main
```

Interfacce Grafiche in Java

- La programmazione orientata agli oggetti si è sviluppata a fronte del successo delle **Graphic User Interface (GUI)**.
- Capire il funzionamento delle GUI
 - consente di esplorare meglio il modello ad oggetti,
 - è importante perchè oggi tutte le applicazioni hanno una GUI.

Progettazione di una GUI

- Progettare una GUI è un'attività molto complessa.
- Il progettista deve
 - conoscere la tipologia degli utenti e i loro bisogni,
 - prevenire gli errori degli utenti, quando possibile,
 - snellire il più possibile l'accesso ai dati ed ai comandi.
- Una GUI deve essere
 - auto consistente, cioè avere un modo uniforme per presentare i dati e per accettare i comandi,
 - tenere presente le convenzioni del sistema in cui l'applicazione verrà eseguita.

AWT and Swing

- L'idea iniziale di Java era quella di fornire una library che consentisse di realizzare GUI indipendentemente dalla piattaforma in cui l'applicazione sarebbe stata eseguita
 - **Abstract Windowing Toolkit (AWT)**, non sufficientemente potente per realizzare GUI complesse.
- Java 2 (JDK 1.2) introduce nelle sue specifiche **Swing**
 - nuova library completamente riprogettata che si appoggia ad AWT solo per i servizi di base.
- Attualmente, Java contiene sia AWT che Swing e il progettista può scegliere quale utilizzare.

GUI in Java

- Una GUI è composta da almeno tre tipi di oggetti
 - **componenti**, come bottoni o menù, che vengono visualizzati,
 - **eventi**, che reificano le azioni dell'utente,
 - **listener**, che rispondono agli eventi sui componenti.
- Un contenitore (**container**) è uno speciale tipo di componente che racchiude ed organizza altri componenti
 - una finestra, un pannello, una combo-box.

Model/View/Controller (1/2)

- Il modo migliore per progettare una GUI è detto **Model/View/Controller (MVC)**.
- La parte dell'applicazione dedicata alla GUI viene spezzata in tre categorie di classi
 - classi model: implementano il modello di quello che si vuole rappresentare, senza dire nulla su come verrà rappresentato,
 - classi view: utilizzano le classi model per dare una veste grafica, una vista, al modello,
 - classi controller: descrivono come il modello cambia in reazione agli eventi che l'utente genera sulla GUI. Ad ogni cambiamento significativo del modello, anche la vista viene informata.

Model/View/Controller (2/2)

- Esempio, nel caso di un editor di poligoni
 - il modello è una lista di poligoni che l'utente ha introdotto nel suo disegno,
 - la vista è come questi poligoni vengono disegnati
 - con quali colori, in quale ordine.
 - il controller è responsabile di
 - modificare il modello cambiando la posizione di un poligono quando l'utente trascina il mouse,
 - informare la vista che qualcosa sta cambiando.

Finestre con Swing (1/2)

- Una finestra è realizzata creando un oggetto di classe `JFrame`.
- Per disegnare all'interno della finestra è possibile creare un pannello disegnabile.
- In Swing, i pannelli sono oggetti di classe `JPanel`
 - è possibile disegnare al loro interno,
 - sono contenitori di altri componenti.

Finestre con Swing (2/2)

- Un `JFrame` consiste di quattro piani. Normalmente si lavora con il piano detto **content pane**.
- Per aggiungere un `JPanel` al `JFrame`.

```
JFrame f = new JFrame("Title");  
JPanel p = new JPanel();  
Container contentPane = f.getContentPane();  
contentPane.add(p);
```

Disegnare in un JPanel (1/2)

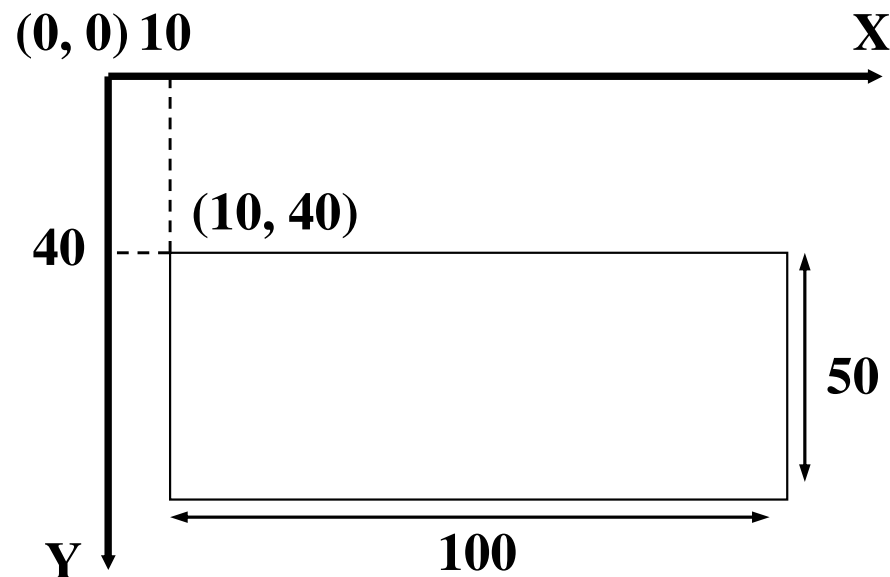
- JPanel è una sotto-classe di JComponent.
- JComponent contiene un metodo `paintComponent(Graphics)`, che viene invocato dalla JVM tutte le volte che si presenta la necessità di (ri-)disegnare un componente.
- Le sotto-classi di JComponent devono re-implementare questo metodo per fornire un algoritmo di disegno del componente.
- Per disegnare nel JPanel costruiamone una sotto-classe e re-implementiamo opportunamente `paintComponent(Graphics)`.

Disegnare in un JPanel (1/2)

- `paintComponent(Graphics)` riceve un oggetto `Graphics` che utilizza per disegnare.
- `Graphics` offre tutti i metodi necessari per disegnare e per gestire i colori ed i font di caratteri.

```
public class EditorView extends JPanel {  
    public void paintComponent(Graphics g) {  
        // disegna lo sfondo  
        super.paintComponent(g);  
  
        ...utilizza il modello per disegnare sul JPanel  
        sfruttando g..  
    }  
}
```


Sistema di Coordinate di Graphics



- Ogni **pixel** (**picture element**) all'interno del `JPanel` è identificato da due numeri interi.
- Graphics utilizza il sistema di coordinate detto **raster**

```
g.drawRect (10, 40,  
            100, 50);
```

Eventi

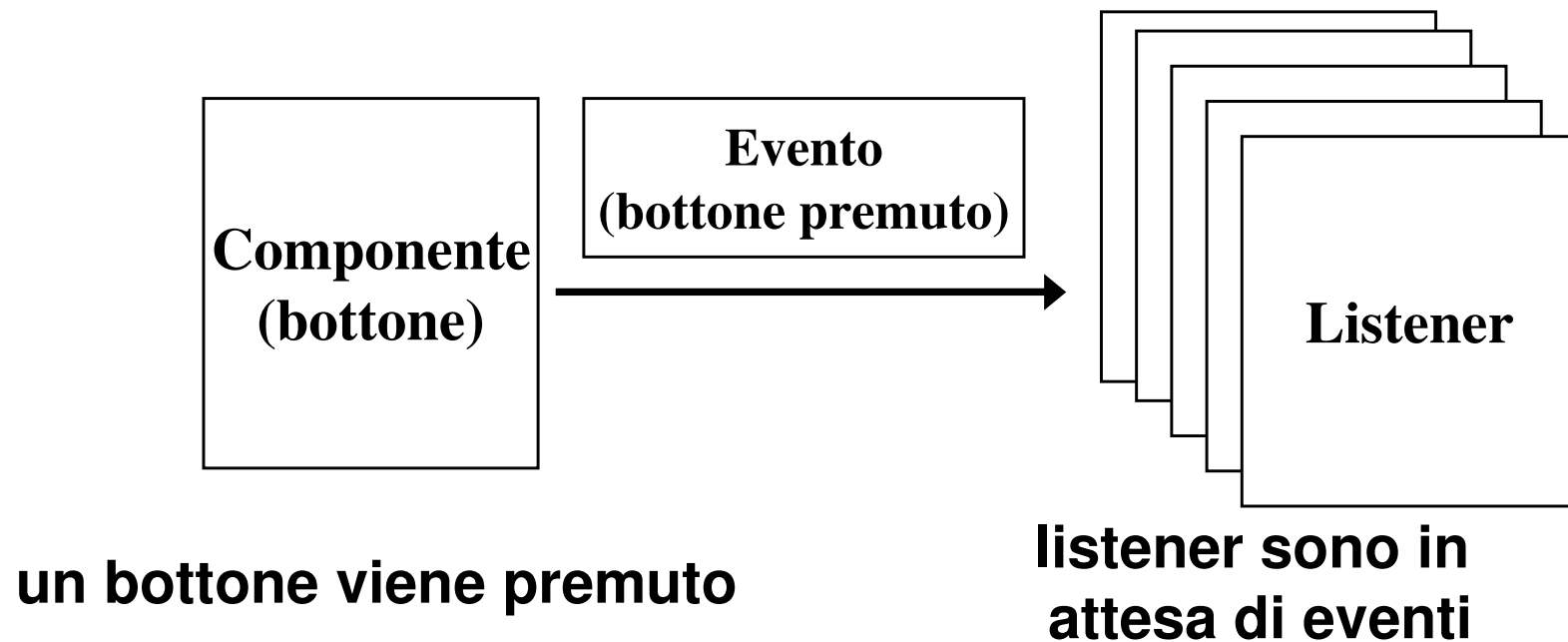
- Un evento è un oggetto che rappresenta un'attività dell'utente sulla GUI
 - il mouse è stato mosso,
 - Il bottone del mouse è stato premuto,
 - è stata scelta una voce di menù.
- Gli eventi vengono generati (**fire**) da un componente della GUI.
- La Java library contiene varie classi che rappresentano i più comuni tipi di eventi.

Eventi e Listener (1/2)

- Un listener è un oggetto che aspetta che un componente generi un particolare tipo di evento.
- La Java library mette a disposizione una serie di interfacce corrispondenti a listener per i più comuni tipi di eventi.
- Per reagire ad un evento si implementa l'interfaccia listener apposita e la si registra sul componente che potrebbe generare l'evento.

Eventi e Listener (2/2)

- Quando un evento accade, il componente genera un oggetto che viene passato a tutti i listener interessati a questa categoria di eventi.



Le Interfacce Listener

- È possibile creare un oggetto listener implementando un'interfaccia listener
 - l'interfaccia `MouseListener` è quella da utilizzare per creare listener collegati alle pressioni del bottone del mouse.
- Gli eventi di `MouseListener` sono
 - `mouse pressed`, il bottone del mouse è stato premuto,
 - `mouse released`, il bottone del mouse è stato rilasciato,
 - `mouse clicked`, il bottone del mouse è stato prima premuto e poi rilasciato senza muovere il mouse,
 - `mouse entered`, il puntatore del mouse è entrato nel componente che ha attivato il listener,
 - `mouse exited`, il puntatore del mouse è uscito dal componente che ha attivato il listener.

Gerarchia di Contenimento (1/2)

- Una GUI viene organizzata mediante componenti contenitori e componenti contenuti
 - i contenitori consentono di organizzare i loro contenuti mediante degli oggetti **layout manager**,
 - i contenuti offrono funzionalità all'utente.
- L'aspetto della GUI è determinato da
 - la gerarchia di contenimento,
 - i layout manager associati ad ogni contenitore,
 - il tipo dei singoli componenti e le loro proprietà.

Gerarchia di Contenimento (2/2)

The entire interface is governed by a border layout with a panel in each area.

North: two labels

East: two labels, a slider, and a combo box with vertical spacing in a box layout

West: four smaller panels in a vertical box layout. One panel for each label / text field combination and another for the gender radio buttons

Center: several check boxes, a label, and a text field

South: two buttons preceded by horizontal glue in a box layout

Survey

Demographic Survey

Please complete this form. Press the Submit button when complete.

First Name:

Last Name:

Middle Initial:

Gender

☐ Male

☐ Female

Hobbies

☐ Reading

☐ Hiking

☐ Woodworking

☐ Origami

☐ Classic Sports

☐ Xtreme Sports

Other:

Age:

Salary Range:

Clear Submit

Layout Manager (1/2)

- Un layout manager è un oggetto che determina il modo in cui i componenti sono disposti all'interno di un contenitore.
- La Java library mette a disposizione vari layout manager
 - in `java.awt`: flow layout, border layout, card layout, grid layout, gridbag layout,
 - in `javax.swing`: box layout, overlay layout.

Layout Manager (2/2)

- Ogni contenitore ha un layout manager di default e un nuovo layout manager può essere impostato mediante `setLayout (LayoutManager)`.
- Ogni layout manager ha le sue regole per disporre i componenti all'interno del contenitore
 - alcuni utilizzano le dimensioni preferite dei componenti, altri utilizzano le dimensioni massime o minime.
 - il layout manager di un contenitore dispone gli oggetti contenuti tutte le volte che un componente è aggiunto al contenitore o che ne cambiano le dimensioni.

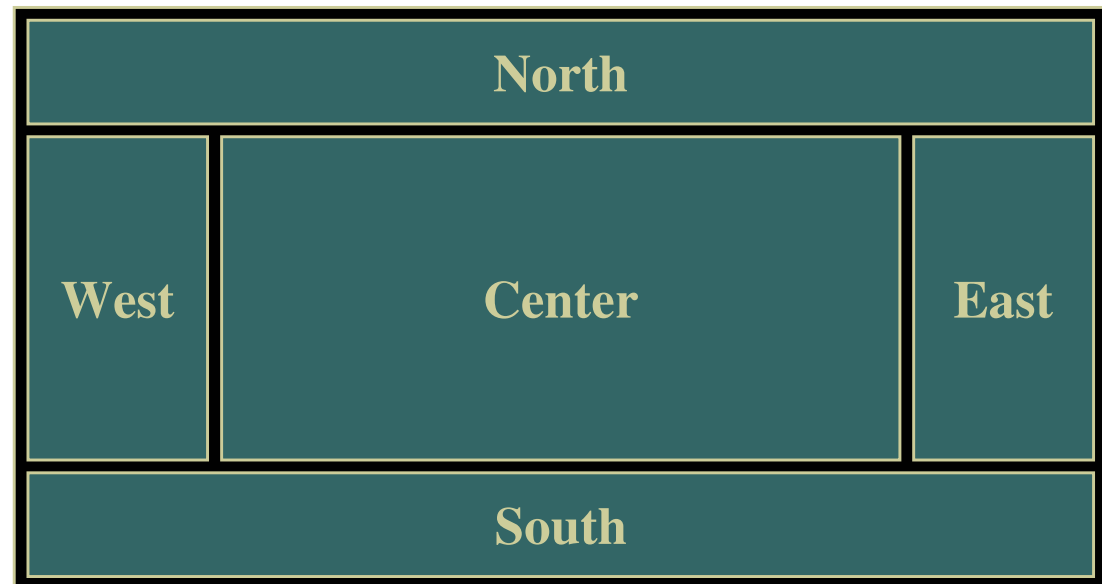
Flow Layout

- Il flow layout mette il maggior numero di componenti su una riga e poi si sposta a quella successiva
 - i componenti sono disposti nello stesso ordine con cui sono aggiunti,
 - il default è che i componenti sono centrati sulle righe.



Border Layout

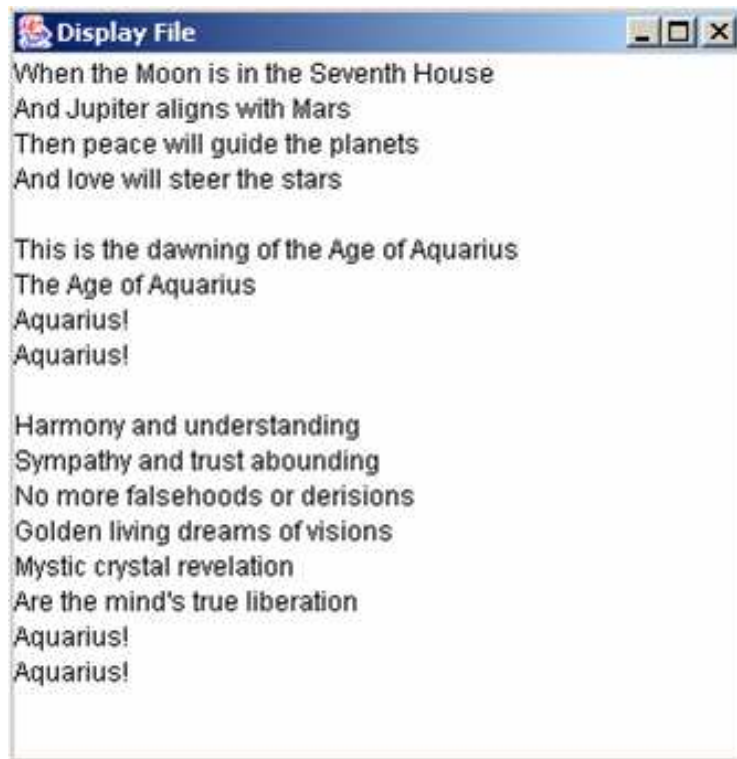
- Il border layout definisce cinque aree in cui un componente può essere aggiunto.
 - L'area di centro si allarga al massimo in modo da riempire tutto lo spazio non utilizzato.



Action Listener

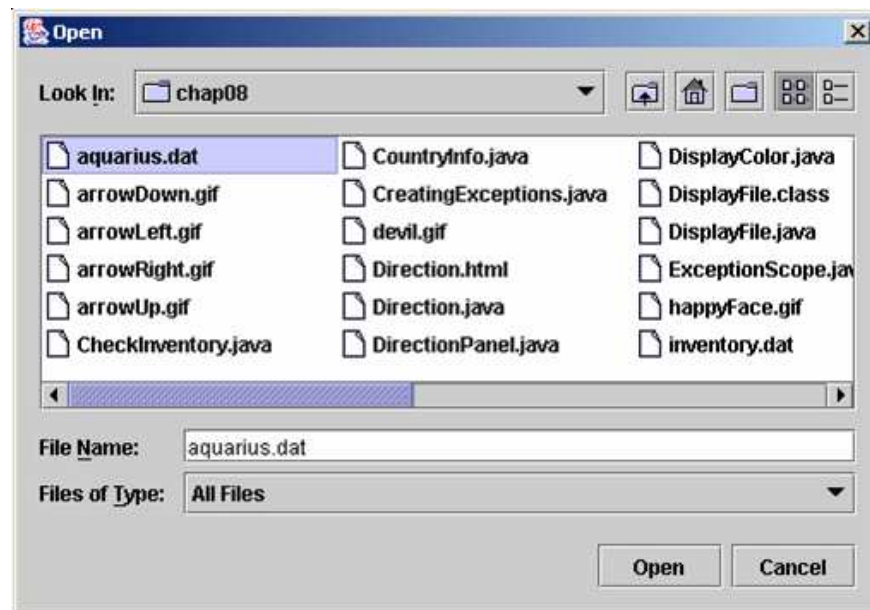
- L'interfaccia `ActionListener` è utilizzata per implementare listener di eventi di molti componenti
 - un bottone è stato premuto,
 - una voce di menù è stata selezionata,
 - un bottone toggle ha cambiato stato.
- Quando un evento di questo tipo accade, a tutti i listener viene invocato il metodo `actionPerformed(ActionEvent)`.

Text Field e Text Area



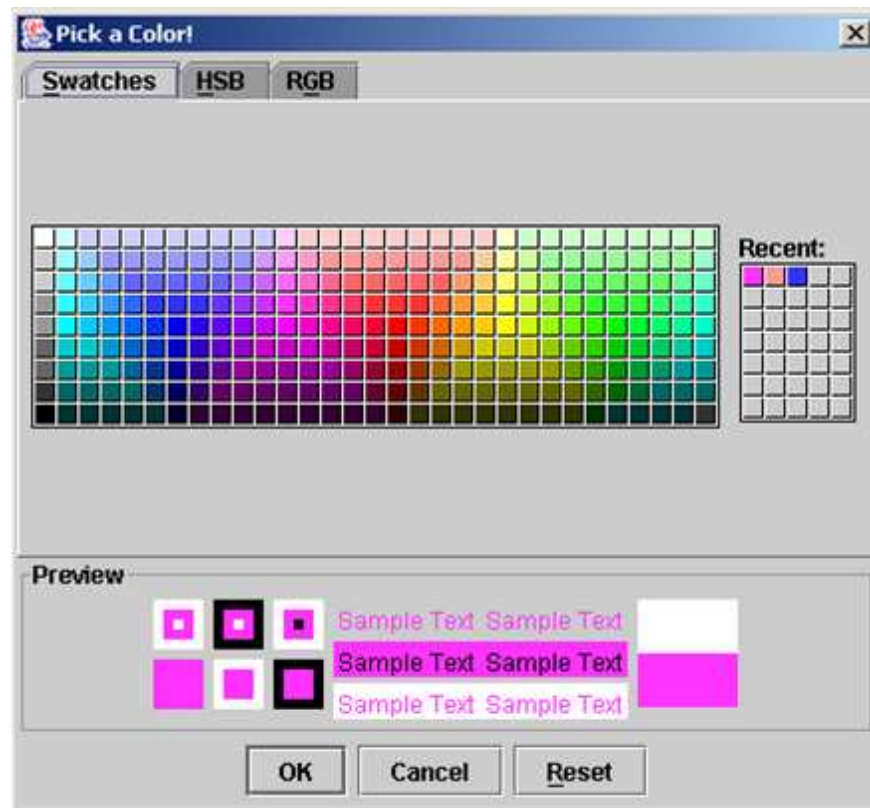
- Un `JTextField` è un editor di testo su una singola linea.
- Una `JTextArea` è un editor di testo multi-linea.
- Possono essere personalizzati per supportare funzionalità complesse.

File Chooser



- Swing mette a disposizione componenti complessi per offrire funzionalità comuni.
- `JFileChooser` consente di far scegliere un file all'utente.

Color Chooser



- JOptionPane consente di porre domande all'utente.



- JColorChooser consente di far scegliere un colore all'utente.

Gestione degli Errori (1/2)

- Un'**eccezione** è un evento che accade a run-time e forza l'uscita del processo dal suo flusso nominale.
- Le eccezioni sono generalmente causate da errori
 - divisione per zero,
 - tentativo di indirizzare un array al di fuori della sua dimensione massima.

Gestione degli Errori (2/2)

- Gestione degli errori in C
 - non è parte del linguaggio e viene effettuata in modo specifico per ogni situazione,
 - tipicamente una funzione ritorna un valore speciale al chiamante per indicare che è accaduto qualche errore,
 - si ipotizza che il chiamante controlli il valore di ritorno e in caso di errore agisca di conseguenza,
 - non c'è nessun meccanismo che forza questo controllo,
 - codice applicativo e codice di gestione dell'errore sono mescolati.

Gestione degli Errori in Java (1/3)

- Quando accade un errore in un metodo Java, questo crea un **oggetto eccezione** e lo consegna alla JVM.
- L'oggetto eccezione contiene le informazioni riguardo all'errore che si è verificato.
- La JVM cerca un **handler** in grado di gestire l'errore prendendo in consegna l'oggetto eccezione,
- Quando la JVM trova un handler adatto, lo esegue passandogli l'oggetto eccezione come parametro.

Gestione degli Errori in Java (2/3)

- La JVM ripercorre all'indietro la sequenza delle chiamate finchè trova un metodo che contiene un handler appropriato.
- Si dice che l'handler individuato **cattura** l'eccezione **lanciata** a causa dell'errore.
- Se la JVM non trova nessun handler, il processo termina e viene visualizzata la sequenza delle chiamate che hanno portato all'errore.

Gestione degli Errori in Java (3/3)

- Il codice di gestione dell'errore è separato dal codice nominale del programma
 - il codice nominale è più leggibile.
- Gli errori vengono propagati automaticamente finchè qualcuno è in grado di gestirli.
- Gli errori sono reificati in oggetti
 - mantengono uno stato che li descrive,
 - possono essere strutturati utilizzando l'ereditarietà.

Blocco try/catch

- Java richiede che ogni metodo catturi un'eccezione mediante un **catch**, oppure che indichi quali eccezioni non gestisce.
- Un blocco catch cattura tutti gli oggetti eccezione del tipo del suo parametro

```
try {  
    ...codice che può generare un'eccezione...  
} catch(Class1 id1) {  
    ...handler delle eccezioni di tipo Class1...  
} catch (Class2 id2) {  
    ...handler delle eccezioni di tipo Class2...  
}
```

Generazione delle Eccezioni (1/2)

- Se un metodo vuole lanciare un'eccezione, crea un oggetto (che implementa `Throwable`) e lo passa alla JVM

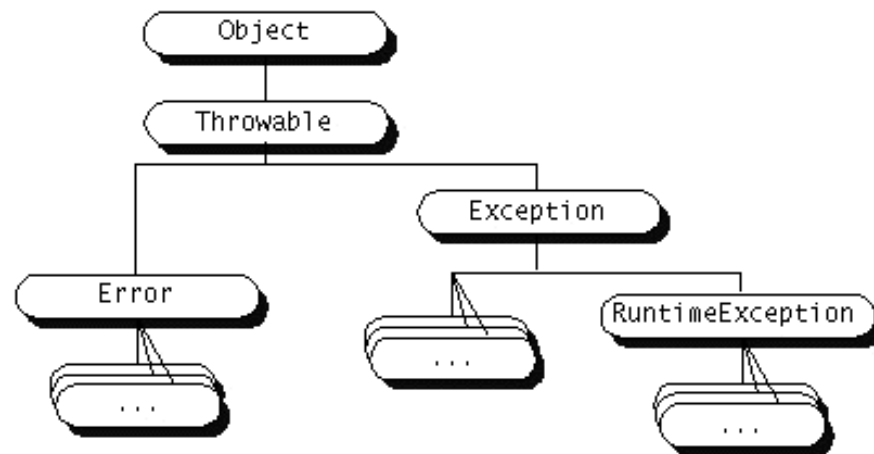
```
if (file.length() == 0)
    throw new FileEmptyException(file);
```

- L'oggetto viene assegnato al parametro del primo blocco `catch` compatibile trovato nella sequenza delle chiamate.

Generazione delle Eccezioni (2/2)

```
public class EmptyStackException extends Exception {  
    public MyException(String msg){ super(msg); }  
}  
  
public class Stack {  
    public Object pop() throws EmptyStackException {  
        if(v.size() == 0)  
            throw new EmptyStackException();  
  
        Object obj = v.get(0);  
        v.remove(0);  
  
        return obj;  
    }  
}
```

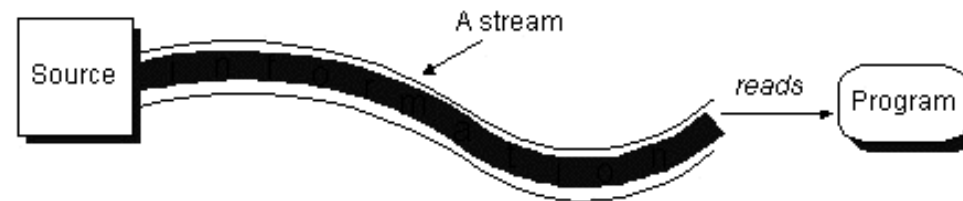
Gerarchia delle Eccezioni



- Quando si verifica un malfunzionamento nella JVM, viene lanciato un `Error`.
- Le applicazioni lanciano eccezioni che estendono `Exception`.
- Una `RuntimeException` è un'eccezione che non è necessario catturare o dichiarare nell'intestazione del metodo
 - `NullPointerException`,
 - `DivisionByZero`.

Input/Output in Java

- La Java library mette a disposizione, nel package `java.io`, delle classi per gestire le operazioni di input/output.
- Gli **input stream** sono utilizzati per leggere dati.

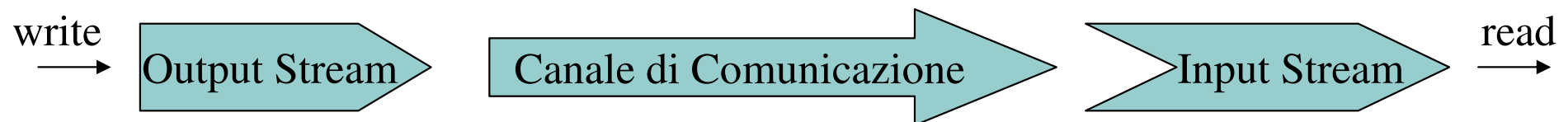


- Gli **output stream** sono utilizzati per scrivere dati.



Input e Output Stream (1/3)

- Uno stream può essere visto come un lato di un canale di comunicazione mono-direzionale
 - collega un output stream ad un corrispondente input stream.
- Tutto quello che viene scritto nell'output stream, viene letto dall'input stream.



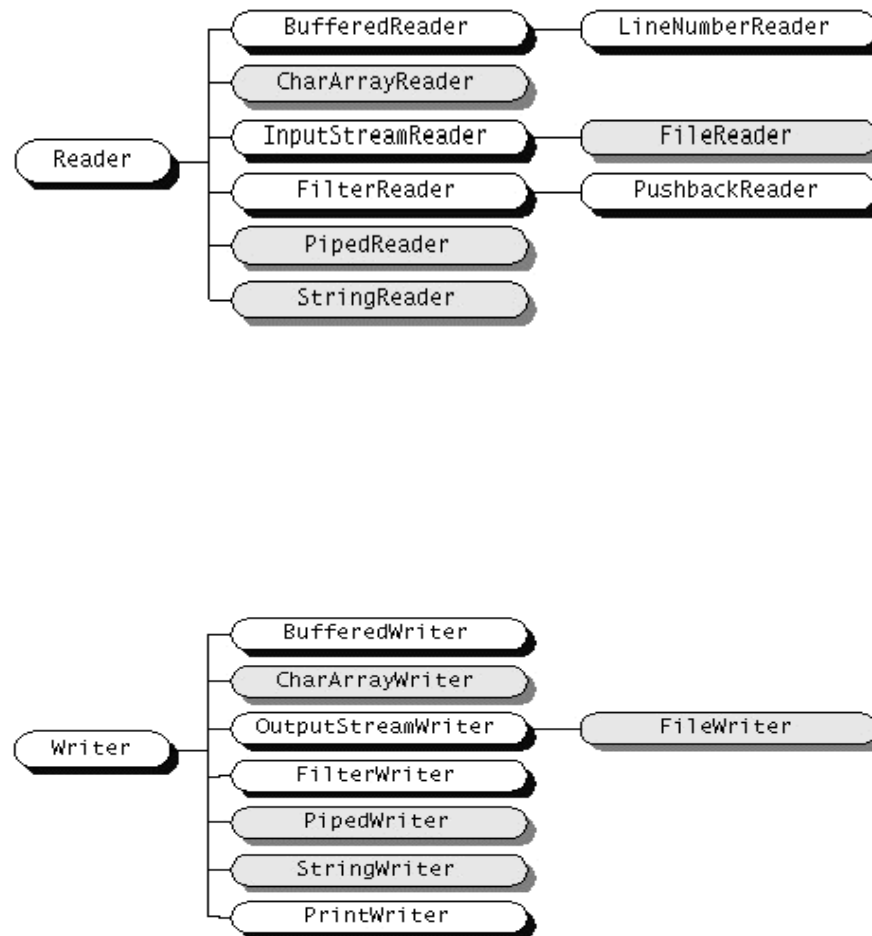
Input e Output Stream (2/3)

- I canali di comunicazione più comuni sono
 - un collegamento via rete,
 - un buffer in memoria,
 - un file,
 - tastiera e video di un terminale.
- Gli stream offrono un'interfaccia uniforme per le operazioni di input/output
 - non importa il tipo o l'origine dei dati, gli algoritmi che manipolano i dati rimangono (circa) gli stessi.

Input e Output Stream (3/3)

- Gli stream sono FIFO (First-In-First-Out).
- Gli stream sono bloccanti
 - le operazioni di scrittura e lettura bloccano il (thread del) processo in attesa che l'operazione sia completata.
- Il package `java.io` contiene una collezione di classi stream
 - divise in due gerarchie sulla base del tipo di dato (byte o caratteri) che viene manipolato.

Character Stream



- Estendono le classi Reader e Writer.
- Manipolano stream di caratteri Unicode a 16 bit.
- Forniscono le conversioni necessarie per manipolare i file indipendentemente dalla piattaforma
 - conversione degli `'\n'`,
 - gestione del set di caratteri.

Byte Stream



- Estendono le classi `InputStream` e `OutputStream`.
- Manipolano stream di byte non codificati.

Character e Byte Stream (1/3)

- Le classi `Reader` and `InputStream` definiscono metodi simili

- `Reader` gestisce i caratteri

```
int read()
```

```
int read(char cbuf[])
```

```
int read(char cbuf[], int offset, int length)
```

- `InputStream` gestisce i singoli byte

```
int read()
```

```
int read(byte cbuf[])
```

```
int read(byte cbuf[], int offset, int length)
```

Character e Byte Stream (2/3)

- Le classi `Writer` and `OutputStream` definiscono metodi simili

- `Writer` gestisce i caratteri

```
void write(char c)
```

```
void write(char cbuf[])
```

```
void write(char cbuf[], int offset, int length)
```

- `OutputStream` gestisce i singoli byte

```
void write(int b)
```

```
void write(byte cbuf[])
```

```
void write(byte cbuf[], int offset, int length)
```


Character e Byte Stream (3/3)

```
public class Main {
    public static void main(String[] args) {
        if(args.length != 2) {
            System.err.println("Usage: java Main <source file> <dest file>");
            System.exit(-1);
        }

        File inputFile = new File(args[0]);
        File outputFile = new File(args[1]);

        try {
            FileReader in = new FileReader(inputFile);
            FileWriter out = new FileWriter(outputFile);

            int c;

            while((c = in.read()) != -1) out.write(c);

            in.close(); out.close();
        } catch(FileNotFoundException fnfe) {
            System.err.println("File not found: " + args[0]);
        } catch(IOException e) {
            System.err.println("Unable to write file: " + args[1]);
        }
    }
}
```

15/Main.java

Collection Framework (1/2)

- Un **framework** è un insieme di **classi** che si appoggiano a delle **interfacce** che l'applicazione implementa per personalizzarne il comportamento (**algoritmi**)
 - il `BubbleSorter` è un (piccolo) framework che sfrutta i servizi del `Comparator` e della `List`,
 - `Swing` è un framework.
- Non è l'applicazione che chiama il framework, è il framework che chiama l'applicazione.

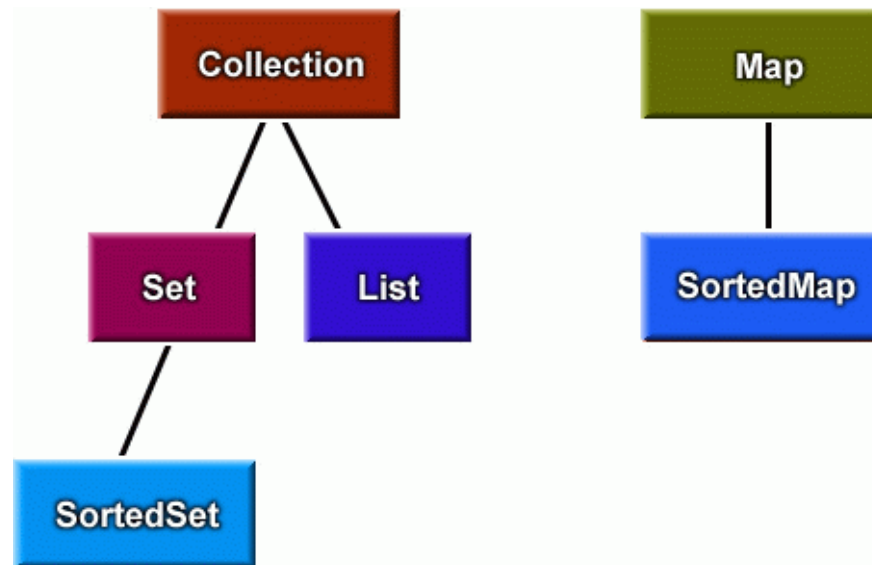
Collection Framework (2/4)

- La Java library comprende il **collection framework**, modulo di gestione di tipi dato **aggregati**
 - liste,
 - insiemi,
 - tabelle associative.
- Il collection framework di Java è costituito da
 - interfacce, che rappresentano i vari tipi dato aggregati,
 - classi (concrete), che implementano le interfacce,
 - algoritmi, di uso generale e definiti sui vari tipi dato.
- Contenuto nel package: `java.util`.

Collezioni

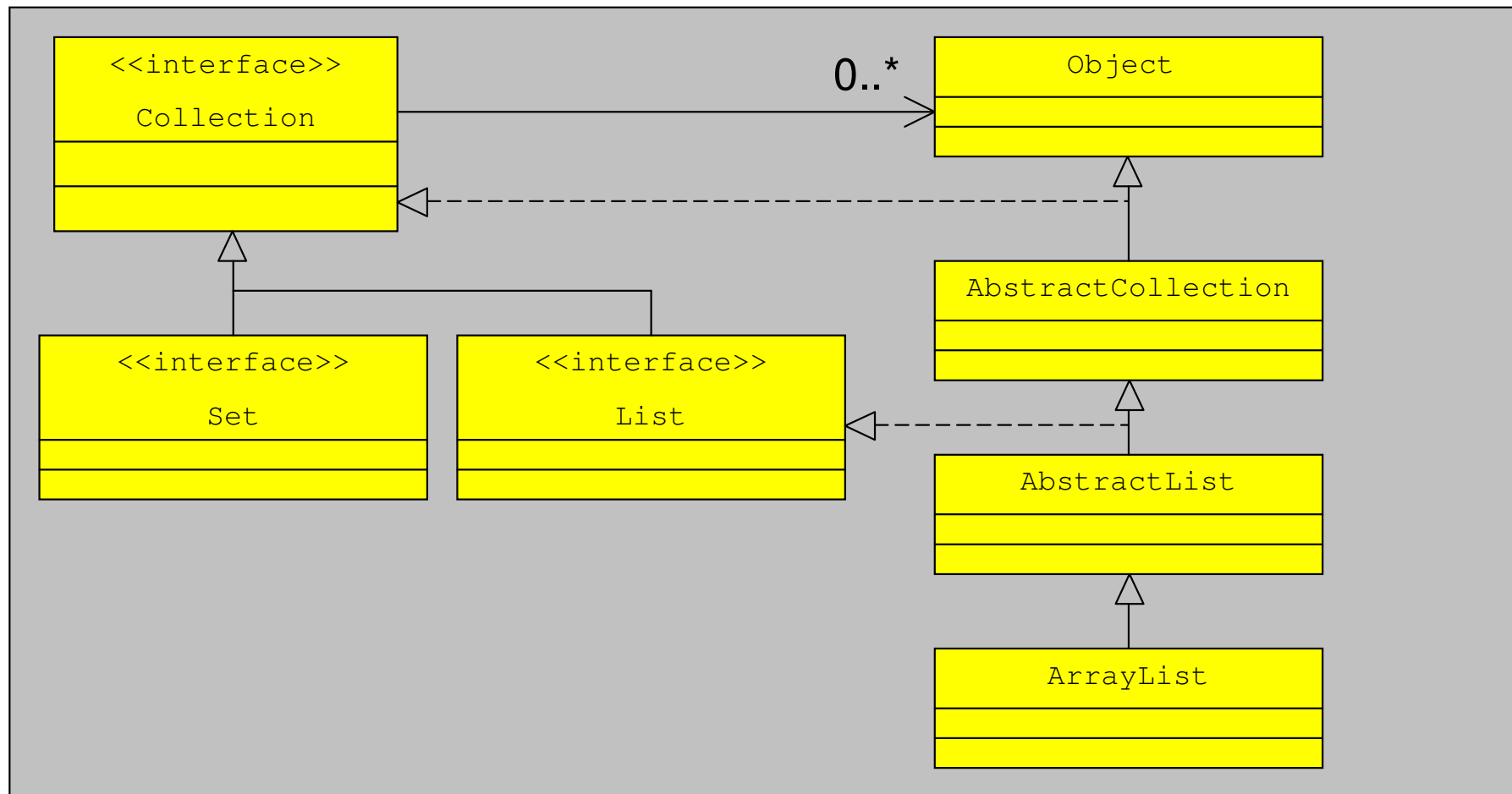
- Una **collezione** (**container**) è un oggetto che raggruppa un insieme di altri oggetti.
- Le collezioni sono usate per memorizzare e manipolare gruppi di dati.
- Le collezioni rappresentano tipicamente oggetti che sono raggruppati per contenimento
 - un elenco telefonico,
 - le parole in un vocabolario,
 - una lista di studenti.

Collection (List, Set) e Map



- `Collection`, interfaccia implementata dai dati aggregati lineari.
- `Map`, interfaccia per i dati aggregati associativi.
- `Object`, utilizzata per massimizzare la generalità dei dati contenuti nelle `Collection` e nelle `Map`.

Interfacce e Classi Concrete



L'Interfaccia `Set`

- Un `Set` è una collezione non posizionale che non può contenere dati duplicati.
- `Set` **estende** `Collection` e non aggiunge altri metodi.
- Se `s1` ed `s2` sono entrambi `Set`
 - `s1.containsAll(s2)`, vero se `s2` è sotto insieme di `s1`,
 - `s1.addAll(s2)`, trasforma `s1` nell'unione di `s1` ed `s2`.
 - `s1.retainAll(s2)`, trasforma `s1` nell'intersezione di `s1` ed `s2`.

L'Interfaccia `List`

- Una `List` è un aggregato posizionale (**sequenza**).
- Una `List` può contenere elementi duplicati.
- In aggiunta ai metodi di `Collection`,
 - **accesso posizionale**, `get(int)`, `set(int, Object)`, `add(int, Object)`, `remove(int, Object)`, `addAll(int, Object)`,
 - **ricerca**, `indexOf(Object)`, `lastIndexOf(Object)`.

Il Metodo `equals` (1/2)

- Insiemi e liste richiedono di poter verificare se due oggetti contenuti sono uguali.
- `Object` mette a disposizione un metodo che tutto il collection framework utilizza per verificare se due oggetti sono uguali

```
boolean equals(Object o)
```

II Metodo equals (2/2)

```
public class Person {  
    public boolean equals(Object o) {  
        if(!(o instanceof Person)) return false;  
  
        Person p = (Person)o;  
  
        return name.equals(p.name) && familyName.equals(p.familyName);  
    }  
  
    private String name = null;  
    private String familyName = null;  
}
```

16/Person.java

L'Interfaccia Map (1/2)

- I tipi dato associativi sono aggregati indirizzabili per contenuto anzichè per posizione
 - dizionario, permette di accedere ad un contenuto mediante una parola,
 - pagine bianche, permettono di accedere ad un numero di telefono mediante un nome.
- Da utilizzare con cautela: non possono sostituire le relazioni tra gli oggetti.

L'Intefaccia Map (2/2)

```
import java.util.Map;
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        Person person = new Person("Paolo", "Rossi");
        Passport passport = new Passport(12345);

        person.setPassport(passport);
        passport.setPerson(person);

        Map passports = new HashMap();
        passports.put(person, passport);

        Passport p = (Passport)passports.get(person);

        if(p == null)
            System.out.println("...");
        else
            System.out.println("..." + p.getNumber());
    }
}
```

16/Main.java

II Metodo `hashCode` (1/2)

- Per implementare una `Map` in modo efficiente è possibile utilizzare un **codice hash** (numero intero) associato ad ogni oggetto contenuto
 - se due oggetti sono uguali (secondo `equals`) allora hanno lo stesso codice hash,
 - se due oggetti non sono uguali (secondo `equals`), non è necessario (ma è preferibile) che abbiano lo stesso codice hash.
- Una tabella hash sfrutta i codici hash degli oggetti contenuti per velocizzare l'indirizzamento
 - il codice hash viene utilizzato come indirizzo virtuale e non univoco del contenuto.

II Metodo hashCode (2/2)

```
public class Person {  
    public boolean equals(Object o) {  
        if(!(o instanceof Person)) return false;  
  
        Person p = (Person)o;  
  
        return name.equals(p.name) && familyName.equals(p.familyName);  
    }  
  
    public int hashCode() { return (name + familyName).hashCode(); }  
  
    private String name = null;  
    private String familyName = null;  
}
```

16/Person.java

Implementazioni

- Il collection framework mette a disposizione più implementazioni per ogni interfaccia.
- Conviene sempre riferirsi alle classi concrete solo al momento della creazione dell'oggetto,

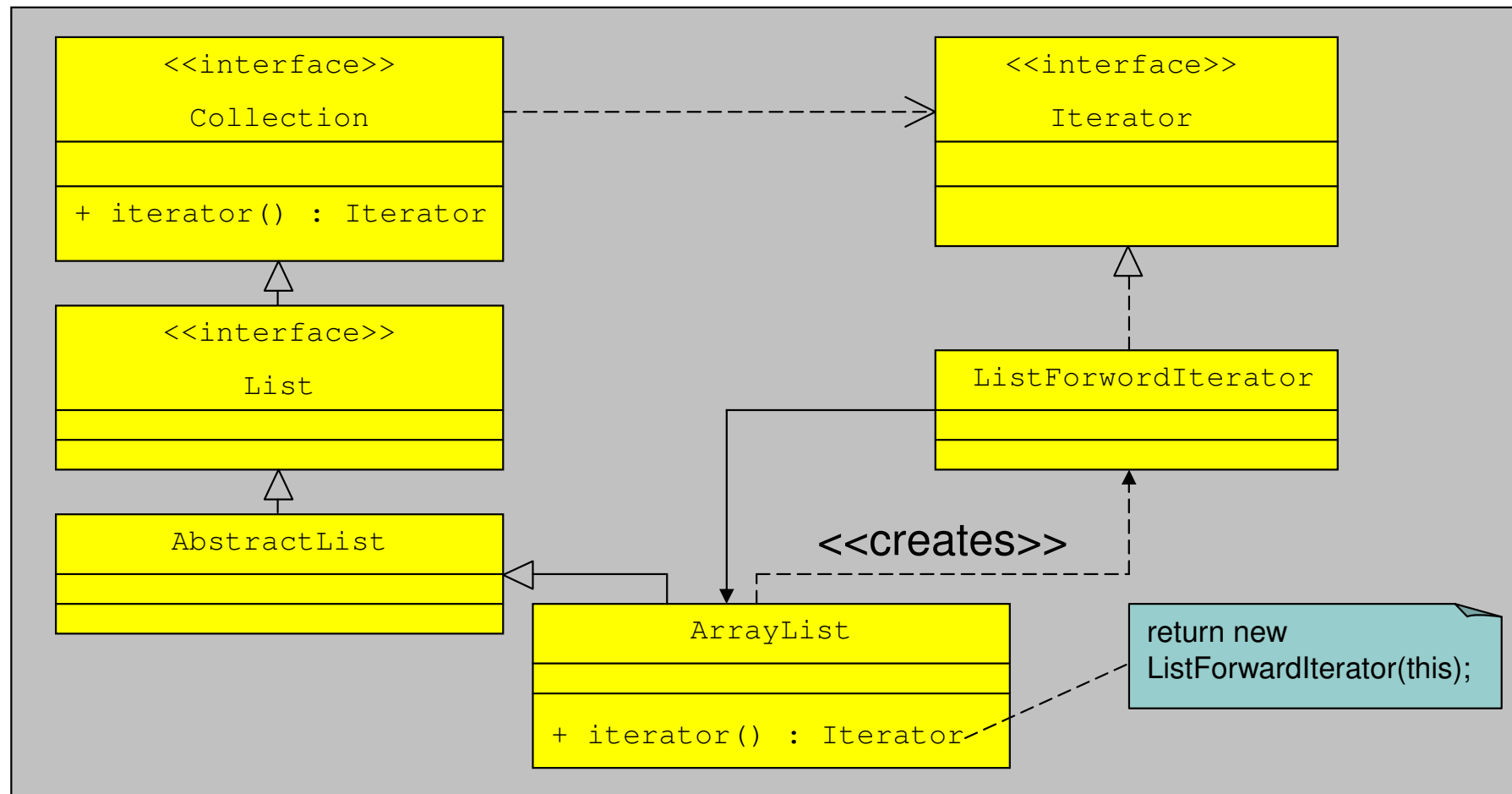
```
List list = new ArrayList()
```

- Le implementazioni disponibili sono
 - Map, HashMap e TreeMap,
 - List, ArrayList e LinkedList,
 - Set, TreeSet e HashSet.

Iteratori (1/3)

- Tutto il collection framework utilizza un unico modo per accedere agli singoli elementi dei tipi dato aggregati.
- **Iteratore**, oggetto che permette di accedere agli elementi di un tipo dato aggregato secondo un certo ordine
 - vengono creati dal tipo dato aggregato,
 - la politica di attraversamento viene cambiata cambiando la classe dell'iteratore.

Iteratori (2/3)



Iteratori (3/3)

```
public class ArraySet implements Set, Cloneable {
    ...
    private class ArraySetIterator implements Iterator {
        public Object next() {
            if(current == collection.size) throw new NoSuchElementException();
            return collection.buffer[current++];
        }

        public boolean hasNext() {
            return current < collection.size;
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }

        private ArraySetIterator(ArraySet arraySet) {
            collection = arraySet;
        }

        private int current = 0;

        private ArraySet collection = null;
    }
}
```

17/ArraySet.java

Oggetti Valore (1/2)

- L'operatore di assegnazione copia le reference tra gli oggetti (**shallow copy**)

`A obj2 = obj1;`

`obj2` e `obj1` puntano allo stesso oggetto.

- Per consentire che un oggetto venga copiato (**deep copy**), la convenzione è quella di implementare `Cloneable`, che offre il metodo pubblico `clone()`

`A obj2 = (A) obj1.clone();`

`obj2` e `obj1` sono due copie dello stesso oggetto.

Oggetti Valore (2/2)

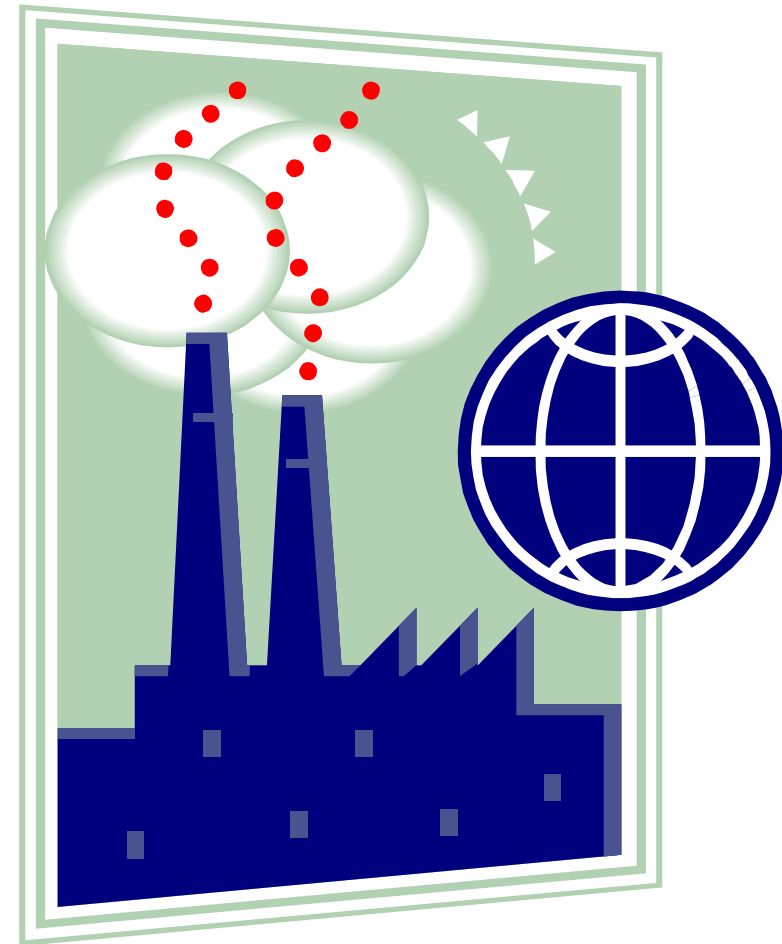
- Gli oggetti per cui sono implementare `clone()` e `equals(Object)` vengono detti **oggetti valore** perché si comportano come i valori primitivi
- Spesso implementano anche `hashCode()`.

```
public class Person {  
    public Object clone() {  
        return new Person(name, familyName);  
    }  
  
    private String name = null;  
    private String familyName = null;  
}
```

16/Person.java

La Produzione del Software

- Il software è **ubiquo**
 - tutti i sistemi sono controllati da software.
- L'ingegneria del software ha a che fare con
 - teorie,
 - metodi,
 - strumenti.per lo sviluppo
industriale di software.



Cos'è il Software?

- In una visione semplicistica
 - programmi eseguibili,
 - documentazione prodotta in fase di sviluppo,
 - documentazione operativa per gli utenti.
- Un **prodotto** software può essere sviluppato
 - per un particolare **committente** (prodotto **custom**), seguendo le specifiche date dal committente stesso,
 - per un particolare mercato (prodotto **general-purpose**), seguendo delle specifiche (dettate dal marketing) che cercano di massimizzare le vendite.

Cos'è l'Ingegneria del Software?

- Non esiste una definizione accettata
 - disciplina ingegneristica che riguarda tutti gli aspetti della produzione del software.
- L'ingegneria del software adotta un approccio **organizzato** e **sistematico** alla produzione di software mediante l'uso di strumenti e tecniche che tengano conto
 - dei vincoli specifici del problema,
 - delle risorse disponibili.

Ingegneria del Software ed Informatica

- L'informatica è una disciplina scientifica è tratta dei fondamenti e delle teorie di base
 - *Computer science is no more about computers than astronomy is about telescopes, E. Dijkstra.*
- L'ingegneria del software ha come obiettivo principale quello di essere d'aiuto alla produzione di **software utile** in modo **efficace**.
- Oggi, le teorie dell'informatica sono ancora insufficienti per supportare completamente l'ingegneria del software.

Ingegneria del Software ed Ingegneria dei Sistemi Informatici

- L'ingegneria dei sistemi informatici considera lo sviluppo di un sistema includendo
 - l'hardware,
 - le persone,
 - il software.
- L'ingegneria del software è parte dell'ingegneria dei sistemi informatici.

Processo di Sviluppo

- Insieme di attività il cui scopo è lo sviluppo (**iniziale** o **evolutivo**) di un sistema software.
- Attività presenti in tutti i **modelli di processo** di sviluppo
 - **specifica**, comprensione di cosa il sistema deve fare e di tutti i vincoli allo sviluppo, alla manutenzione ed alla evoluzione,
 - **sviluppo**, produzione effettiva del sistema,
 - **validazione**, controllo che il sistema sia esattamente quello che il committente ha richiesto,
 - **manutenzione** ed **evoluzione**, modifica del sistema in risposta a cambiamenti delle necessità.

Metodologia di Produzione (1/2)

- Rappresentazione semplificata ed astratta di un processo di sviluppo.
- Si basa su un modello del processo di sviluppo.
- Esistono vari modelli del processo di sviluppo, i più semplici sono
 - cascata,
 - evolutivo,
 - a prototipi.

Metodologia di Produzione (2/2)

- Una metodologia include
 - **artefatti** (modelli), che è necessario produrre seguendo un ben preciso flusso,
 - flusso di produzione degli artefatti, che ne mette in luce le dipendenze,
 - notazioni, che è possibile utilizzare per produrre gli artefatti,
 - regole, a cui è necessario sottostare nella produzione degli artefatti,
 - suggerimenti e aiuti, che guidano l'ingegnere del software.

CASE tool

- I **Computer-Aided Software Engineering (CASE)** tool sono sistemi software che offrono supporto automatico alle attività all'interno di una metodologia.
- Possono essere usati,
 - per supportare la metodologia in tutte le sue parti,
 - solo per avere una notazione ed un repository comune.

Qualità del Software (1/5)

- Le qualità su cui si basa la valutazione di un sistema possono essere
 - **interne**, riguardano le caratteristiche legate allo sviluppo del software e non sono visibili agli utenti,
 - **esterne**, riguardano le funzionalità fornite dal sistema e sono visibili agli utenti.
- Le categorie sono legate, non è possibile ottenere qualità esterne se il sistema non gode di qualità interne.

Qualità del Software (2/5)

- È anche possibile valutare un sistema secondo qualità
 - **relative al prodotto**, riguardano le caratteristiche stesse del sistema e sono sempre valutabili,
 - **relative al processo**, riguardano i metodi utilizzati durante lo sviluppo del software.
- Le categorie sono legate, non è possibile ottenere qualità del prodotto se il sistema non gode di qualità del processo
 - *product quality is process quality.*

Qualità del Software (3/5)

- **Correttezza**, un sistema è corretto se rispetta le specifiche.
- **Affidabilità** (**dependability**), un sistema è affidabile se l'utente può dipendere da esso.
- **Robustezza**, un sistema è robusto se si comporta in modo ragionevole anche in circostanze non previste dalle specifiche.
- **Efficienza**, un sistema è efficiente se usa bene le risorse di calcolo.
- **Facilità d'uso**, un sistema è facile da usare se l'interfaccia che presenta all'utente gli permette di esprimersi in modo naturale.

Qualità del Software (4/5)

- La valutazione
 - della correttezza e dell'affidabilità è basata sulle specifiche,
 - della robustezza riguarda tutti i casi non trattati dalle specifiche.
- La valutazione
 - della correttezza e della facilità d'uso è affidata ai committenti,
 - della robustezza e dell'efficienza, agli sviluppatori.

Qualità del Software (5/5)

- **Verificabilità**, un sistema è verificabile se le sue caratteristiche sono verificabili.
- **Riusabilità**, un sistema è riusabile se può essere usato, in tutto o in parte, per costruire nuovi sistemi.
- **Portabilità**, un software è portabile se può funzionare su più piattaforme hardware/software.
- **Facilità di manutenzione**, un sistema è facile da mantenere se
 - è strutturato in modo tale da facilitare la ricerca degli errori,
 - la sua struttura permette di aggiungere nuove funzionalità al sistema,
 - la sua struttura permette di adattarlo ai cambiamenti del dominio applicativo.
- **Interoperabilità** si riferisce all'abilità di un sistema di cooperare con altri sistemi, anche di altri produttori.

Qualità del Processo di Produzione

- **Produttività**, misura l'efficienza del processo nei termini della velocità di consegna del sistema,
- **Tempestività**, misura la capacità del processo di valutare e rispettare i tempi di consegna del sistema,
- **Trasparenza**, un processo di produzione è trasparente se permette di capire il suo stato e di controllarne i passi.
- **Agilità**, misura la capacità del processo di consentire la produzione in tempi ridotti (ridotto **time-to-market**).

Costi del Software (1/3)

- I costi del software sono spesso predominanti nei costi di un sistema
 - il costo del software in un PC spesso è maggiore di quello dell'intero hardware.
- Il costo del software è principalmente nel **mantenimento** piuttosto che nello sviluppo.
- Uno degli obiettivi principali dell'ingegneria del software è ottenere un sviluppo **cost-effective** prevenendo per quanto possibile i costi di manutenzione.

Costo del Software (2/3)

- Costo delle risorse per lo sviluppo (**costi diretti**)
 - costo del personale sviluppatore, normalmente predominante sugli altri,
 - costo del personale di supporto,
 - costo delle risorse di sviluppo,
 - materiali di consumo,
 - altri costi generali della struttura.

Costo del Software (3/3)

- Altri costi (**costi indiretti**)
 - capacità, motivazione e coordinamento del personale,
 - complessità del sistema da realizzare,
 - stabilità dei requisiti,
 - caratteristiche dell'ambiente di sviluppo.
- Costo di manutenzione ed evoluzione.

Manutenzione ed Evoluzione (1/3)

- Il software deve evolvere perchè
 - non sono stati colti correttamente i requisiti,
 - i requisiti non sono inizialmente noti,
 - cambiano le condizioni operative,
 - ...
- L'evoluzione è ineliminabile per gran parte delle tipologie di sistemi, anche se i requisiti iniziali sono corretti e completi.

Manutenzione ed Evoluzione (2/3)

- Manutenzione è il processo di modifica di un sistema dopo il suo **rilascio** al fine di
 - eliminare anomalie (**manutenzione correttiva**),
 - migliorare le prestazioni o altri attributi di qualità (**manutenzione perfettiva**),
 - adattare a mutamenti dell'ambiente (**manutenzione adattativa**).
- Costi della manutenzione:
 - spesso maggiori del 50% del costo totale del software,
 - 75% (fonte Hewlett-Packard),
 - 70% (fonte Dipartimento della Difesa degli Stati Uniti).
 - suddivisi in:
 - Manutenzione correttiva: 20%,
 - Manutenzione perfettiva: 60%,
 - Manutenzione adattativa: 20%.

Manutenzione ed Evoluzione (3/3)

- Esperienza di Hewlett-Packard
 - la maggior parte degli errori potrebbe essere scoperta mediante tecniche sistematiche di revisione di progetto,
 - i moduli con maggior complessità del flusso di controllo hanno maggiore probabilità di contenere errori.
- Alcuni dati
 - su 10 difetti scoperti durante il test, 1 si propaga nella manutenzione,
 - eliminare i difetti costa, in tempo, 4 10 volte per sistemi grossi e maturi rispetto a piccoli e ancora in sviluppo,
 - il costo di rimozione degli errori aumenta con il ritardo rispetto al quale gli errori sono introdotti.

Problemi Principali dell'Ingegneria del Software Oggi

- Gestire sistemi vecchi ma non sostituibili (**legacy system**)
 - necessità di manutenzione adattativa,
 - problemi dovuti alla tecnologia obsoleta.
- Gestire l'eterogeneità dei sistemi
 - i sistemi sono sempre distribuiti e sono composti da una grande varietà di sistemi hardware e software.
- Riduzione dei tempi di consegna
 - c'è una sempre maggiore pressione per ottenere tempi di consegna brevi.

Responsabilità Etiche e Professionali

- All'ingegnere del software sono affidate responsabilità che vanno al di là delle semplici questioni tecniche/tecnologiche.
- L'ingegnere del software deve comportarsi in modo **professionale**
 - onesto ed eticamente responsabile.
- Mantenere un comportamento etico è più che attenersi ai dettami di legge.

Principali Responsabilità Professionali (1/2)

- Riservatezza
 - l'ingegnere del software deve rispettare la riservatezza, indipendentemente da formali accordi (**Non-Disclosure Agreement, NDA**).
- Competenza
 - l'ingegnere del software non deve falsare il proprio livello di competenza,
 - non deve accettare compiti al di fuori delle proprie competenze.

Principali Responsabilità Professionali (2/2)

- Diritto di proprietà intellettuale
 - l'ingegnere deve conoscere le leggi riguardanti la proprietà intellettuale quali quelle sui brevetti e sulle invenzioni,
 - deve garantire a sé e agli altri il rispetto della proprietà intellettuale.
- Uso anomalo delle competenze tecniche
 - l'ingegnere del software non deve sfruttare le proprie competenze tecniche per usi non istituzionali.

Codice Etico ACM/IEEE

- Association for Computer Machinery (**ACM**) ed Institute of Electrical and Electronics Engineers (**IEEE**) anno emanato un codice di etica professionale.
- Questo codice contiene otto principi riguardo
 - il comportamento,
 - Il processo decisionaledi ingegneri del software di tutti i livelli, da professionisti affermati agli studenti.

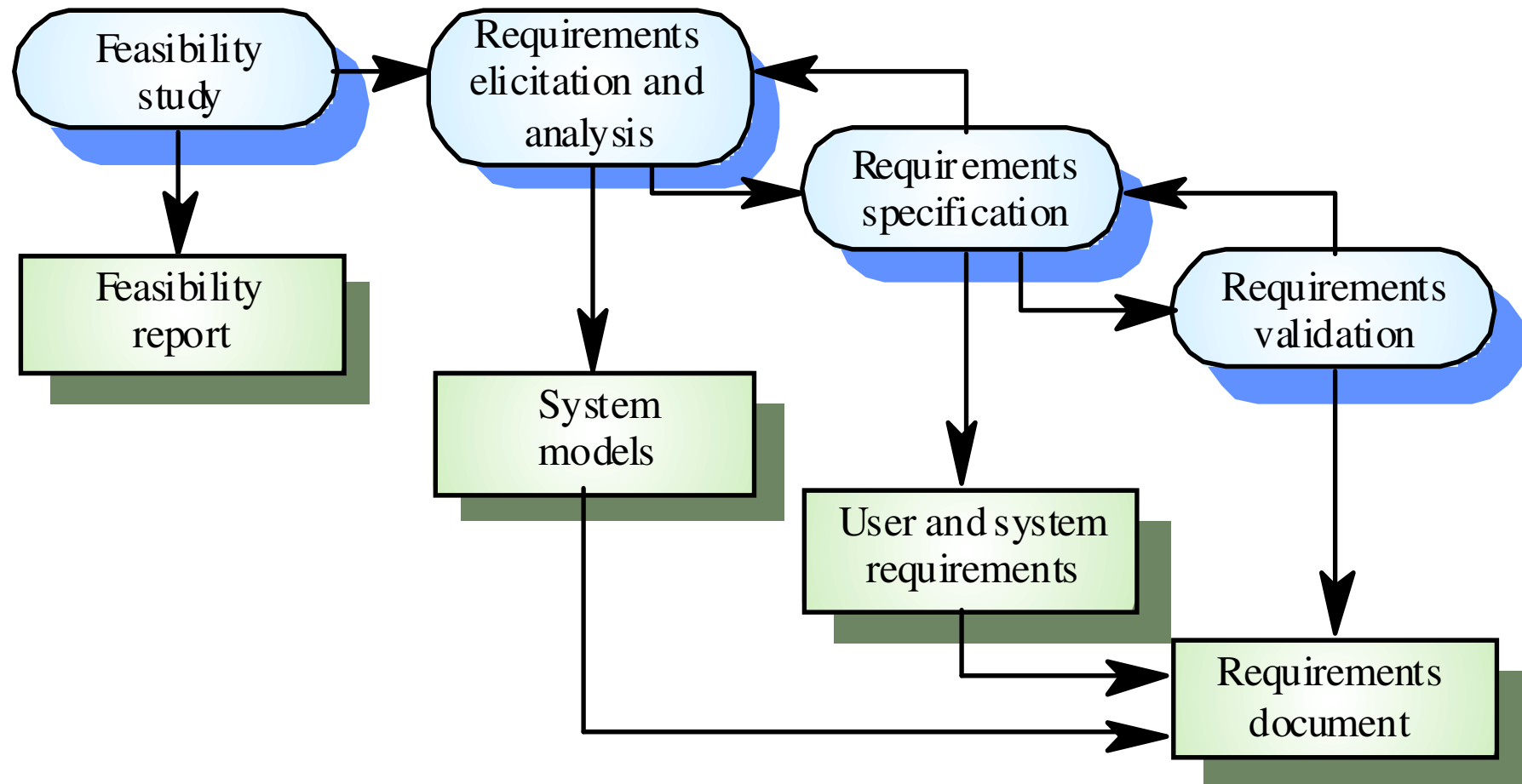
Modello del Processo di Sviluppo

- Un **processo di sviluppo** è insieme strutturato di attività richieste per sviluppare un sistema software
 - specifica,
 - progettazione ed implementazione,
 - verifica e validazione,
 - manutenzione ed evoluzione.
- Un modello di un processo di sviluppo è una rappresentazione astratta del processo.

Specifica

- Attività che consente di stabilire
 - quali sono i **requisiti** (espliciti o impliciti) dei committenti,
 - quali sono i **vincoli** (espliciti o impliciti) sul sistema e sul suo sviluppo.
- Processo di ingegneria dei requisiti
 - studio di fattibilità,
 - estrazione (**elicitation**) dei requisiti ed analisi dei requisiti,
 - specifica dei requisiti,
 - validazione dei requisiti.

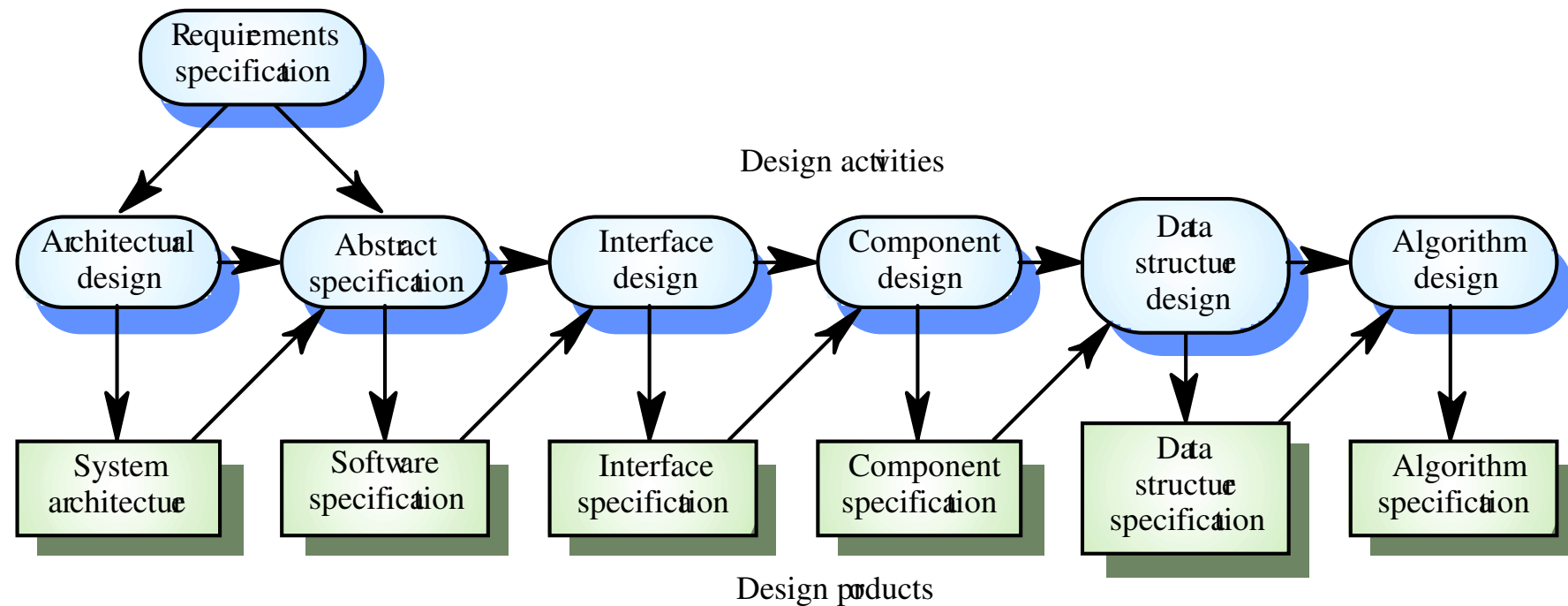
II Processo di Ingegneria dei Requisiti



Progettazione ed Implementazione

- Processo di conversione della specifica di un sistema in un sistema funzionante.
- Progettazione
 - progettazione della struttura (**architettura**) che a vari livelli di dettaglio realizza la specifica.
- Implementazione
 - converte le architetture individuate nella progettazione in codice eseguibile.
- Progettazione ed implementazione sono fortemente correlate ed (alle volte) sono interlacciate.

Un Processo di Progettazione

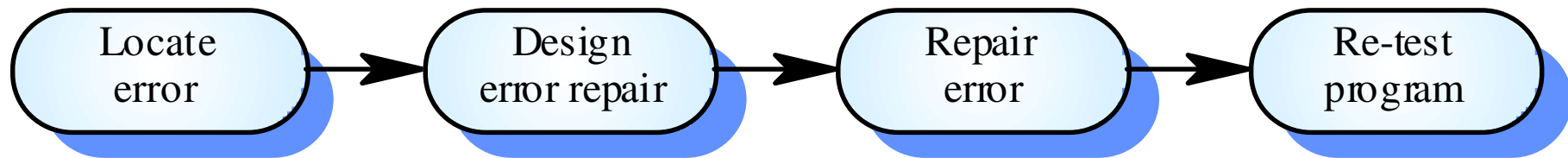


Metodologie di Progettazione

- Approccio sistematico allo sviluppo di un progetto software.
- Il progetto è tipicamente documentato mediante un insieme di modelli grafici
 - data-flow diagram,
 - entity-relation diagram,
 - tutti i modelli UML
 - class diagram,
 - interaction diagram.

Programmazione e Debugging

- Trasformazione di un progetto in un programma e rimozione degli errori di codifica.
- Ogni programmatore esegue il testing delle unità che sta sviluppando per scoprire errori di codifica.



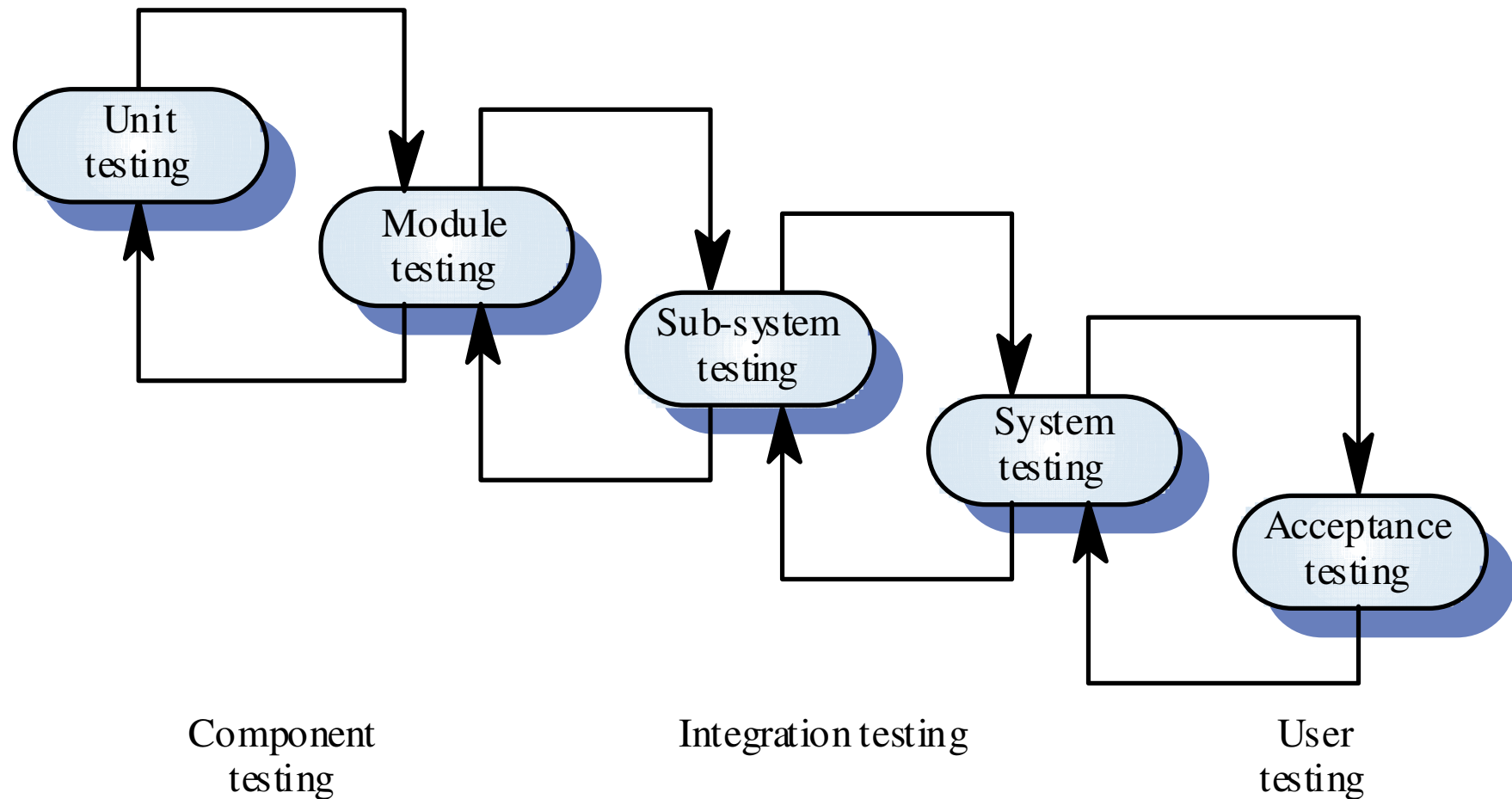
Verifica e Validazione

- La verifica e la validazione servono per mostrare che il sistema
 - è **conforme** alle specifiche,
 - incontra i requisiti dell'utente.
- Comprende revisione e testing del sistema.
- Il testing di un sistema richiede di eseguire il sistema su casi di test (**test case**) derivati dalle specifiche.

Testing

- A cosa serve il testing
 - *Le operazioni di testing possono individuare la presenza di errori nel software ma non ne possono dimostrare la correttezza, E. Dijkstra.*
 - Verificare il comportamento del sistema in un insieme di casi sufficientemente ampio da rendere plausibile che il suo comportamento sia analogo anche nelle restanti situazioni.
- Le operazioni di testing si suddividono in:
 - Testing in the small, riguardano moduli singoli,
 - Testing in the large, riguardano il sistema nella sua globalità.

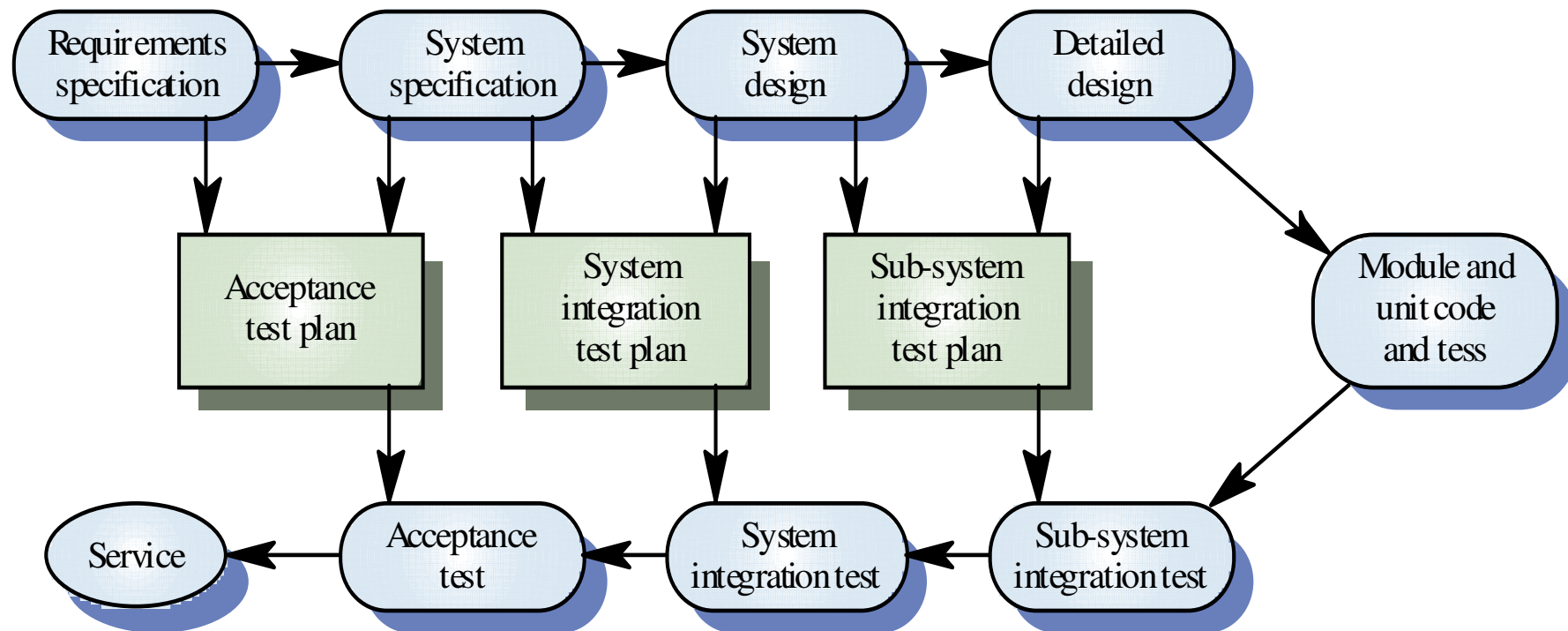
Processo di Testing



Fasi di Testing (1/2)

- Unit testing
 - testing di ogni singolo componente.
- Module testing
 - testing di ogni modulo, insieme di componenti inter-dipendenti.
- Sub system testing
 - testing dell'integrazione tra moduli in sotto-sistemi. L'obiettivo è individuare problemi nelle interfacce tra i moduli.
- System testing
 - testing del sistema nel suo complesso. Testing di eventuali proprietà emergenti del sistema.
- Acceptance testing
 - testing con i committenti per verificare l'accettabilità del sistema.

Fasi di Testing (2/2)



Testing in the Small (1/3)

- Valuta il corretto funzionamento di una porzione del codice
 - analizzando il suo output in relazione ad input significativi.
- Verifica di copertura dei programmi (**statement test**)
 - un errore non può essere scoperto se la parte di codice che lo contiene non viene eseguita almeno una volta,
 - criterio: selezionare un insieme di test T tali che, a seguito dell'esecuzione del programma P su tutti i casi di T, ogni istruzione elementare di P venga eseguita almeno una volta,
 - può essere eseguito solo conoscendo la struttura interna della porzione di codice (**white box testing**).

Testing in the Small (2/3)

- Verifica di copertura delle decisioni (**branch test**)
 - per ogni condizione presente nel codice è utilizzato un test che produca risultato vero e falso,
 - criterio: selezionare un insieme di test T tali che, a seguito dell'esecuzione del programma P su tutti i casi di T, ogni arco del grafo di controllo di P sia attraversato almeno una volta,
 - può essere eseguito solo conoscendo la struttura interna della porzione di codice.

Testing in the Small (3/3)

- Verifica di copertura delle decisioni e delle condizioni (**branch and condition test**)
 - per ogni porzione di condizione composta presente nel codice, sia utilizzato un test che produca il risultato vero e falso,
 - criterio: selezionare un insieme di test T tali che, a seguito dell'esecuzione del programma P su tutti i casi di T, ogni arco del grafo di controllo di P sia attraversato e tutti i possibili valori delle condizioni composte siano valutati almeno una volta,
 - produce un'analisi più approfondita rispetto al criterio di copertura delle decisioni,
 - può essere eseguito solo conoscendo la struttura interna della porzione di codice.

Testing in the Large

- Il white-box testing è impossibile per sistemi di grandi dimensioni.
- Il sistema è visto come una scatola nera (**black-box testing**) e si vanno a verificare le corrispondenze di input e output.
- L'insieme di test da utilizzare viene selezionato sulla base delle specifiche che
 - Problema: il sistema riceve come input una fattura di cui è nota la struttura dettagliata. La fattura deve essere inserita in un archivio ordinato per data. Se esistono altre fatture con la stessa data fa fede l'ordine di arrivo. È inoltre necessario verificare che: 1) il cliente sia già stato inserito in archivio, vi sia corrispondenza tra la data di inserimento del cliente e quella della fattura, ...
Test set:
 - 1) Fattura con data odierna,
 - 2) Fattura con data passata e per la quale esistono altre fatture,
 - 3) Fattura con data passata e per la quale non esistono altre fatture,
 - 4) Fattura il cui cliente non è stato inserito....

Ispezione del Software

- Analisi del codice per capirne le caratteristiche e le funzionalità
 - può essere effettuata sul codice oppure sullo pseudocodice,
 - permette la verifica unitaria di un insieme di condizioni,
 - è soggetta agli errori di colui che la effettua,
 - si basa su un modello della realtà e non su dati reali.
- I due principali approcci
 - code walk through;
 - code inspection.

Code Walk-Through

- Analisi informale eseguita da un team di persone
 - dopo aver selezionato opportune porzioni del codice e opportuni valori di input simulano su carta il comportamento del sistema,
 - il numero di persone coinvolte deve essere ridotto,
 - il progettista deve fornire in anticipo la documentazione scritta relativa al codice,
 - l'analisi non deve durare più di alcune ore,
 - l'analisi deve essere indirizzata solamente alla ricerca dei problemi e non alla loro soluzione,
 - al fine di aumentare il clima di cooperazione all'analisi non devono partecipare i manager.

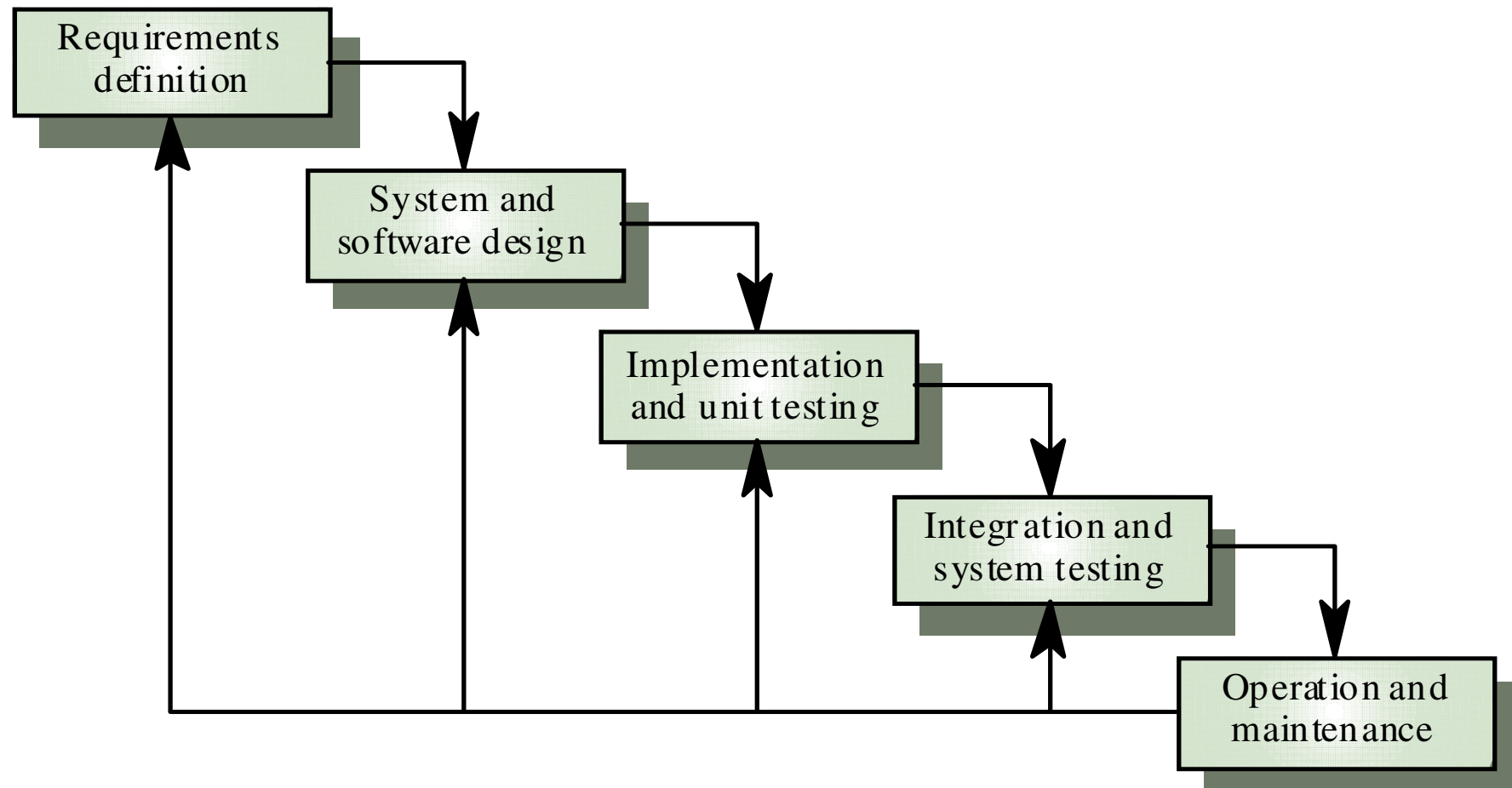
Code Inspection

- Analisi eseguita da un team di persone e organizzata come nel caso del code walk-through
 - mira a ricercare classi specifiche di errori,
 - il codice viene esaminato controllando soltanto la presenza di una particolare categoria di errore, piuttosto che simulando una generica esecuzione.
- Le classi di errori che vengono solitamente ricercate con questa tecnica sono
 - uso di variabili non inizializzate,
 - loop infiniti,
 - letture di porzioni di memoria non allocata,
 - rilascio improprio della memoria.

Processi di Sviluppo Generici

- Modello a cascata (**waterfall**)
 - separa in modo netto le fasi di specifica e di sviluppo.
- Modello evolutivo
 - specifica e sviluppo sono interlacciate.
- Modello a componenti
 - il sistema è sviluppato assemblando un insieme di componenti disponibili.

Modello a Cascata



Fasi del Modello a Cascata

- Analisi e specifica dei requisiti
 - capire cosa i committenti vogliono e formalizzare il più possibile questi requisiti.
- Progettazione del sistema
 - costruzione di un modello del sistema a vari livelli di dettaglio.
- Implementazione e unit testing
 - realizzazione delle singole unità di progetto e testing delle unità.
- Integrazione
 - tra le unità del progetto,
 - con eventuali altri sistemi già presenti.
- Operatività e manutenzione
 - supporto all'operatività,
 - manutenzione correttiva.

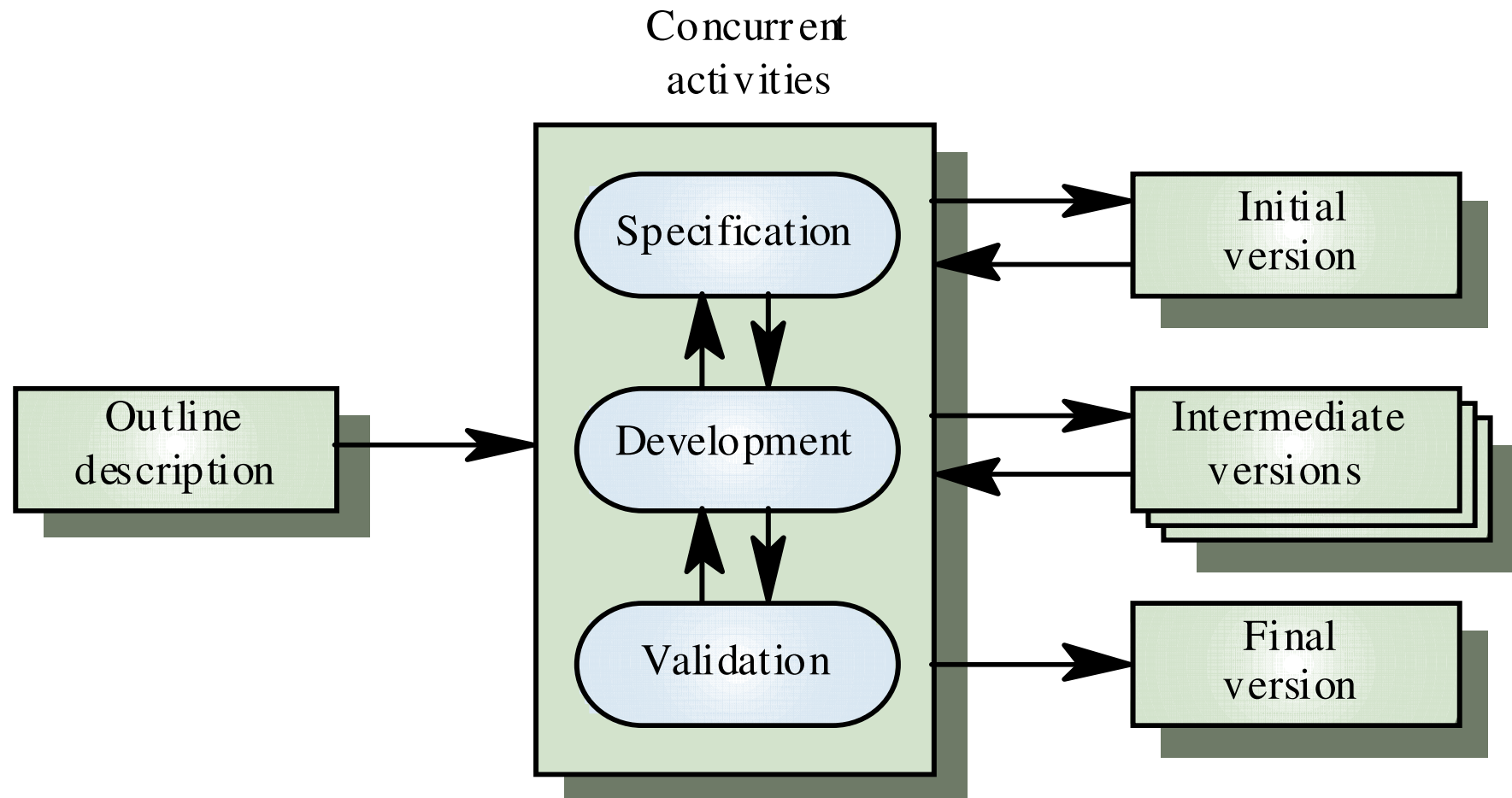
Problemi del Modello a Cascata

- Difficoltà di gestire cambiamenti quando il processo è finito.
- Non c'è flessibilità nella partizionare a livelli
 - difficile gestire i cambiamenti dei requisiti dei committenti.
- Il modello a cascata è appropriato solo per tipologie di sistema già note.

Sviluppo Evolutivo (1/3)

- Sviluppo esplorativo
 - l'obiettivo è lavorare con i committenti per evolvere il sistema partendo da una versione iniziale della specifica,
 - deve iniziare da requisiti ragionevoli.
- Prototipazione **throw-away**
 - l'obiettivo è catturare i requisiti del sistema,
 - i requisiti che si adottano per realizzare il prototipo spesso non vengono mantenuti.

Sviluppo Evolutivo (2/3)



Sviluppo Evolutivo (3/3)

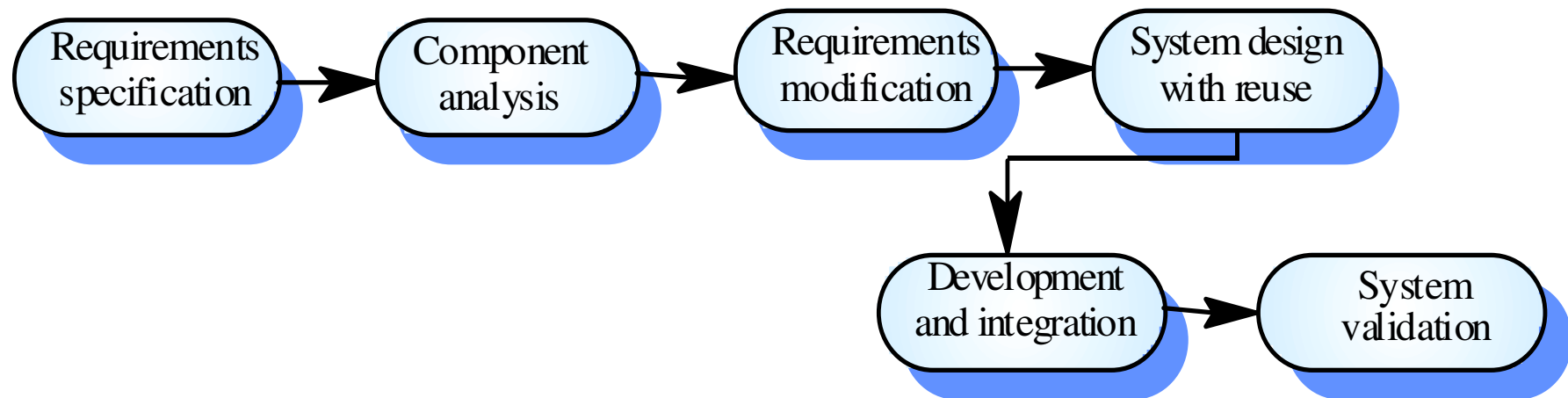
- Problemi
 - il processo è poco ispezionabile e prevedibile,
 - i sistemi risultanti spesso sono poco strutturati,
 - conviene sfruttare tecniche e linguaggi per una prototipazione veloce.
- Applicabilità
 - per sistemi medio piccoli fortemente interattivi, sfruttando dei **Rapid Application Development (RAD)** tool,
 - per parti specifiche di sistemi grandi,
 - per sistemi a breve vita.

Sviluppo a Componenti (1/2)

- Basato sul riutilizzo sistematico di componenti
 - sviluppati **in-house**,
 - **Commercial-Off-The-Shelf (COTS)**.
- Il sistema è realizzato assemblando i componenti disponibili.
- Questo approccio è immaturo, ma è molto promettente.

Sviluppo a Componenti (2/2)

- Stadi del processo
 - specifica dei requisiti,
 - analisi dei componenti disponibili,
 - modifica dei requisiti,
 - progettazione basata sull'integrazione dei componenti scelti,
 - sviluppo ed integrazione,
 - validazione.



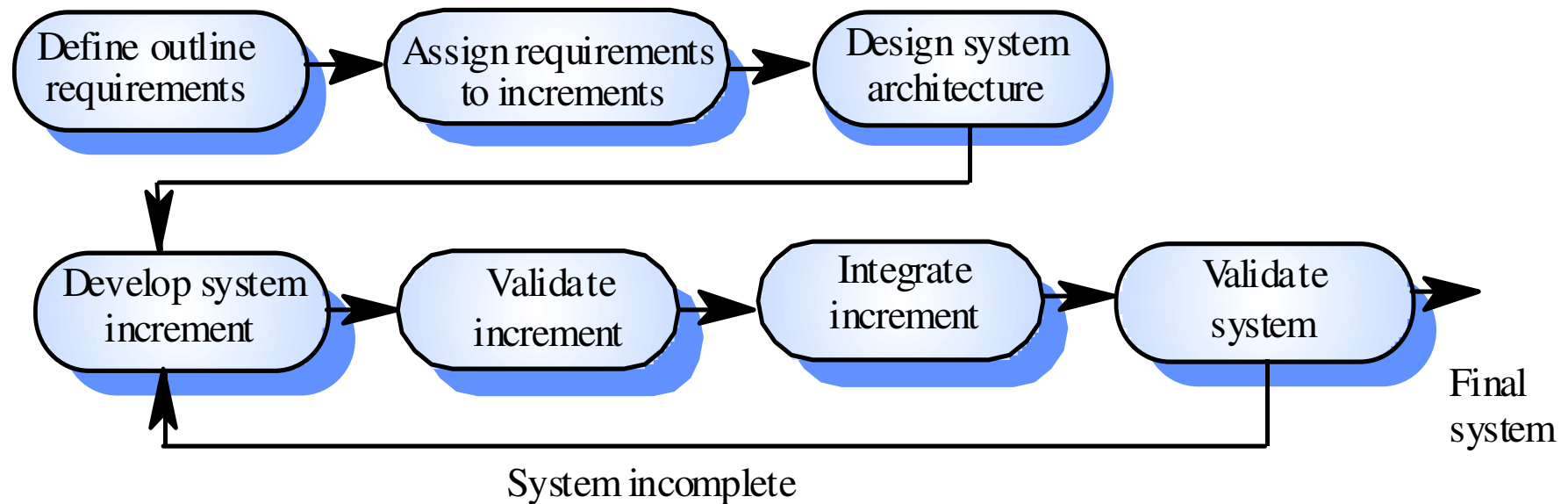
Iterazione dei Processi

- I requisiti di un sistema evolvono sempre durante lo sviluppo di un progetto
 - necessità di iterare le fasi di un processo per rivedere quello che è già stato fatto a fronte dei nuovi requisiti.
- L'iterazione può essere applicata a qualsiasi modello di processo.
- Due approcci allo sviluppo iterativo
 - sviluppo incrementale,
 - sviluppo a spirale.

Sviluppo Incrementale (1/2)

- Anziché rilasciare il sistema tutto insieme, lo sviluppo è spezzato in revisioni incrementali.
- I requisiti del committente sono prioritizzati e sviluppati in ordine di priorità.
- All'inizio dello sviluppo di una nuova revisione, i requisiti sono bloccati fino alla fine della stessa.

Sviluppo Incrementale (2/2)



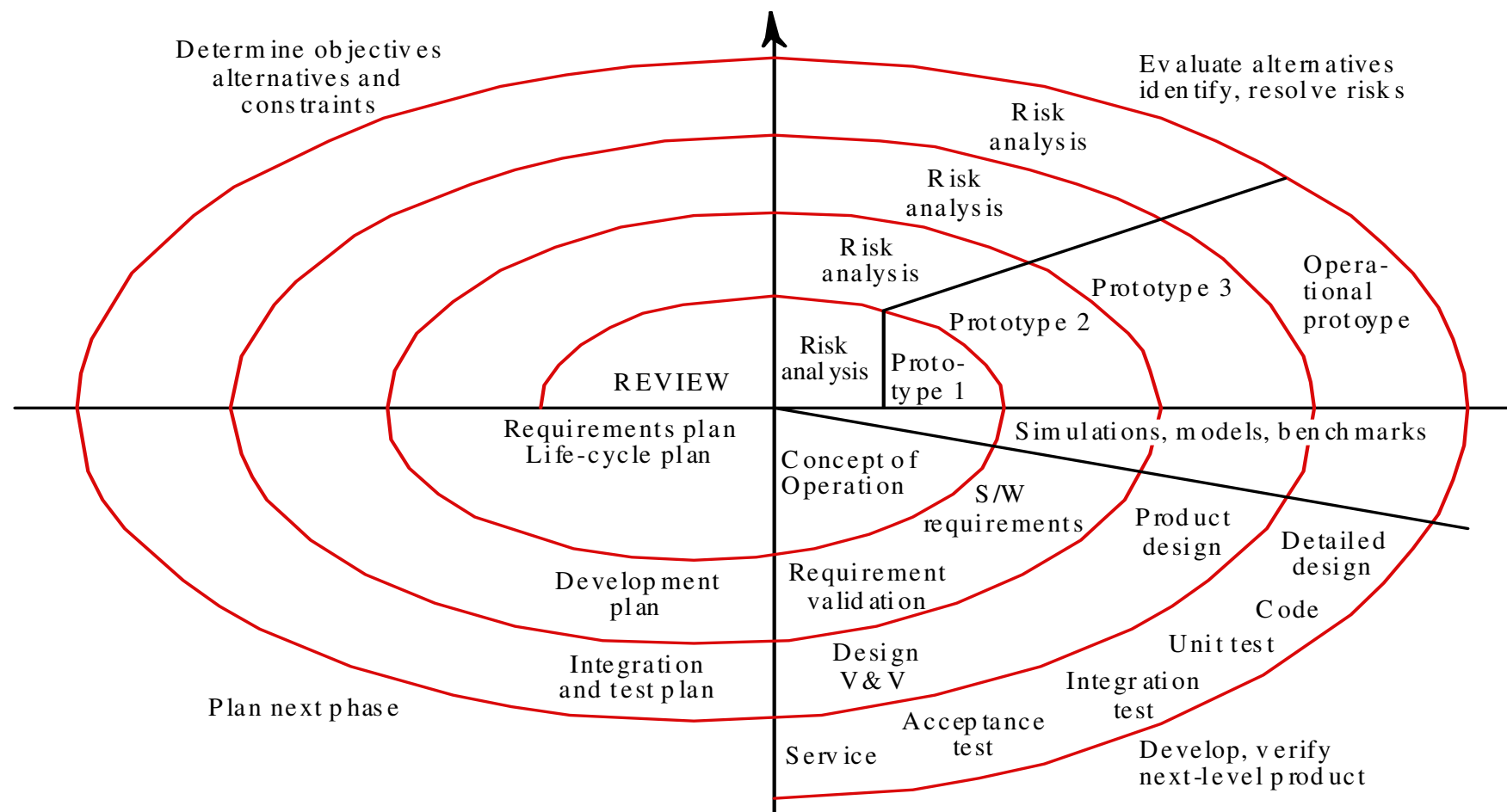
Vantaggi dello Sviluppo Incrementale

- Il committente può disporre di alcune funzionalità in tempi brevi.
- Le revisioni iniziali possono avere lo stessa funzione di prototipi in grado di mettere in evidenza ulteriori requisiti.
- Basso rischio di un completo fallimento del progetto.
- Le funzionalità più importanti tendono ad essere testate maggiormente.

Sviluppo a Spirale (1/2)

- Il processo è una spirale anziché una sequenza con periodici ritorni.
- Ogni ciclo della spirale rappresenta una fase del processo.
- Non ci sono fasi fisse, le attività dei vari cicli sono scelte in dipendenza delle necessità.
- L'analisi dei rischi è parte esplicita del processo e viene effettuata periodicamente.

Sviluppo a Spirale (2/2)



Settori del Modello a Spirale

- Individuazione degli obiettivi della fase.
- Analisi e riduzione dei rischi
 - i rischi sono analizzati e le attività organizzate in modo da ridurre i rischi specifici della fase.
- Sviluppo e validazione
 - viene applicato un modello di processo per la specifica fase. In linea di principio, potrebbe essere un modello diverso per ogni fase.
- Pianificazione
 - il progetto viene rivisto e la fase successiva viene pianificata.

Gestione (**Management**) dei Progetti Software

- Attività coinvolte nel garantire che un sistema venga consegnato
 - rispettando le scadenze,
 - nei costi preventivati.
- Chi sviluppa un sistema ha sempre dei vincoli
 - di tempo,
 - di costo (**budget**).
- Spesso i progetti vengono gestiti con un approccio **one shot**.

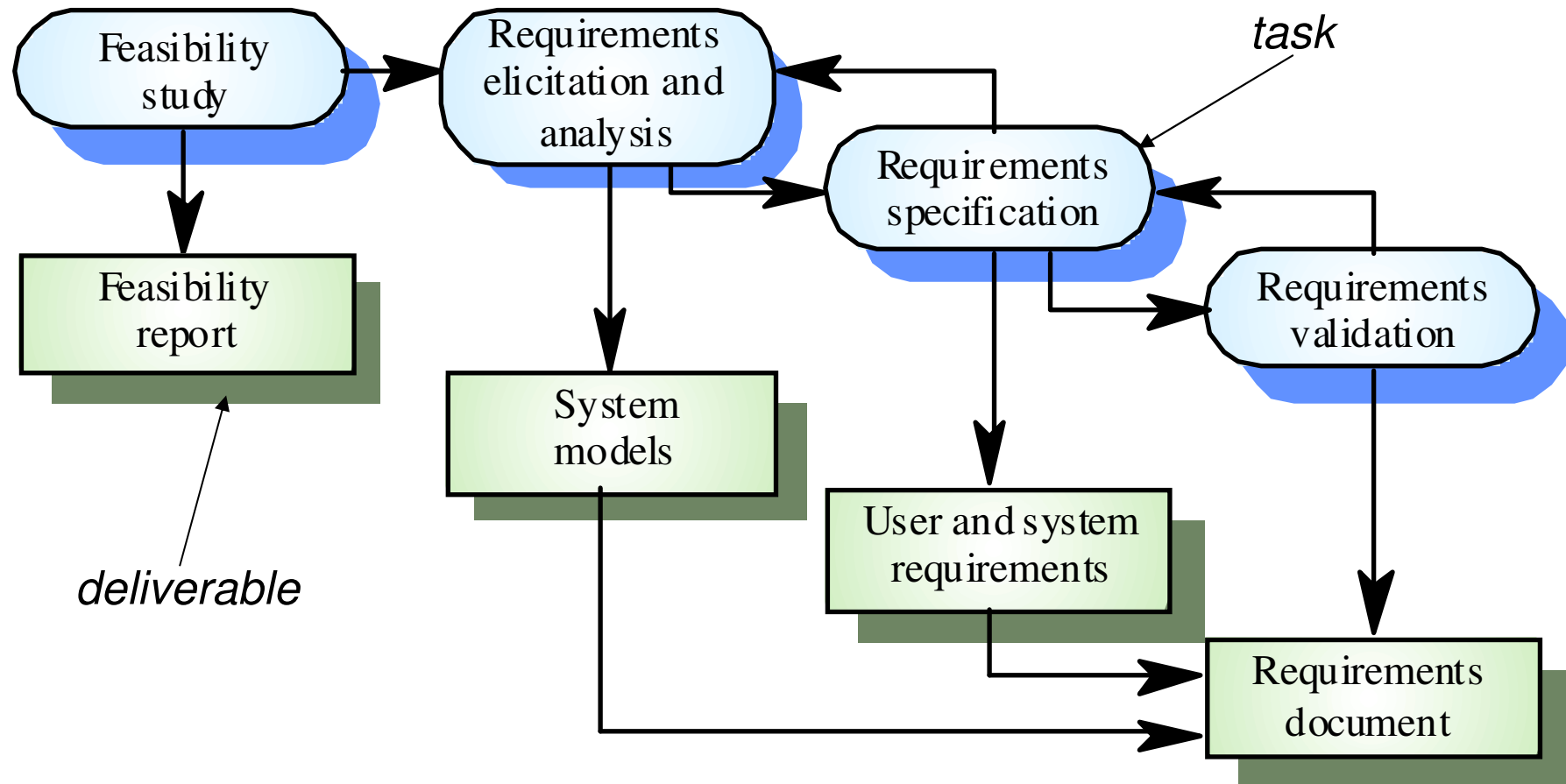
Attività di Gestione

- Scrittura di una proposta di progetto da sottomettere al committente
 - pianificazione delle attività,
 - pianificazione dei costi.
- Selezione e valutazione del personale.
- Controllo dello stato di avanzamento e revisione
 - delle attività,
 - dei costi.
- Produzione di **report** periodici.

Organizzazione delle Attività

- Le attività in un progetto dovrebbero essere organizzate per
 - produrre risultato tangibile e misurabile,
 - consentire di giudicare lo stato d'avanzamento.
- Le **milestone** sono le date di terminazione delle singole attività.
- I **deliverable** sono i risultati del progetto rilasciati al committente
 - ogni azione visibile al committente (**task**) deve produrre un deliverable.

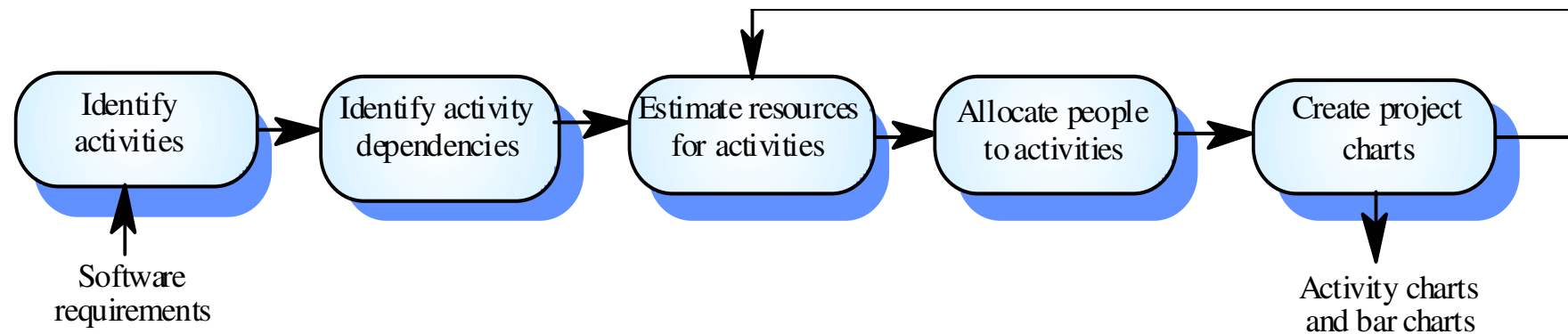
Task e Deliverable



Attività di **Scheduling** di un Progetto

- Organizzazione del progetto in attività e stima per ogni attività
 - delle risorse necessarie,
 - del tempo necessario.
- Organizzazione delle attività (in modo concorrente) per utilizzare in modo ottimo le risorse
 - minimizzare le dipendenze tra le attività,
 - minimizzare i **percorsi critici**, percorsi in cui un ritardo di un'attività non può essere ammortizzato.

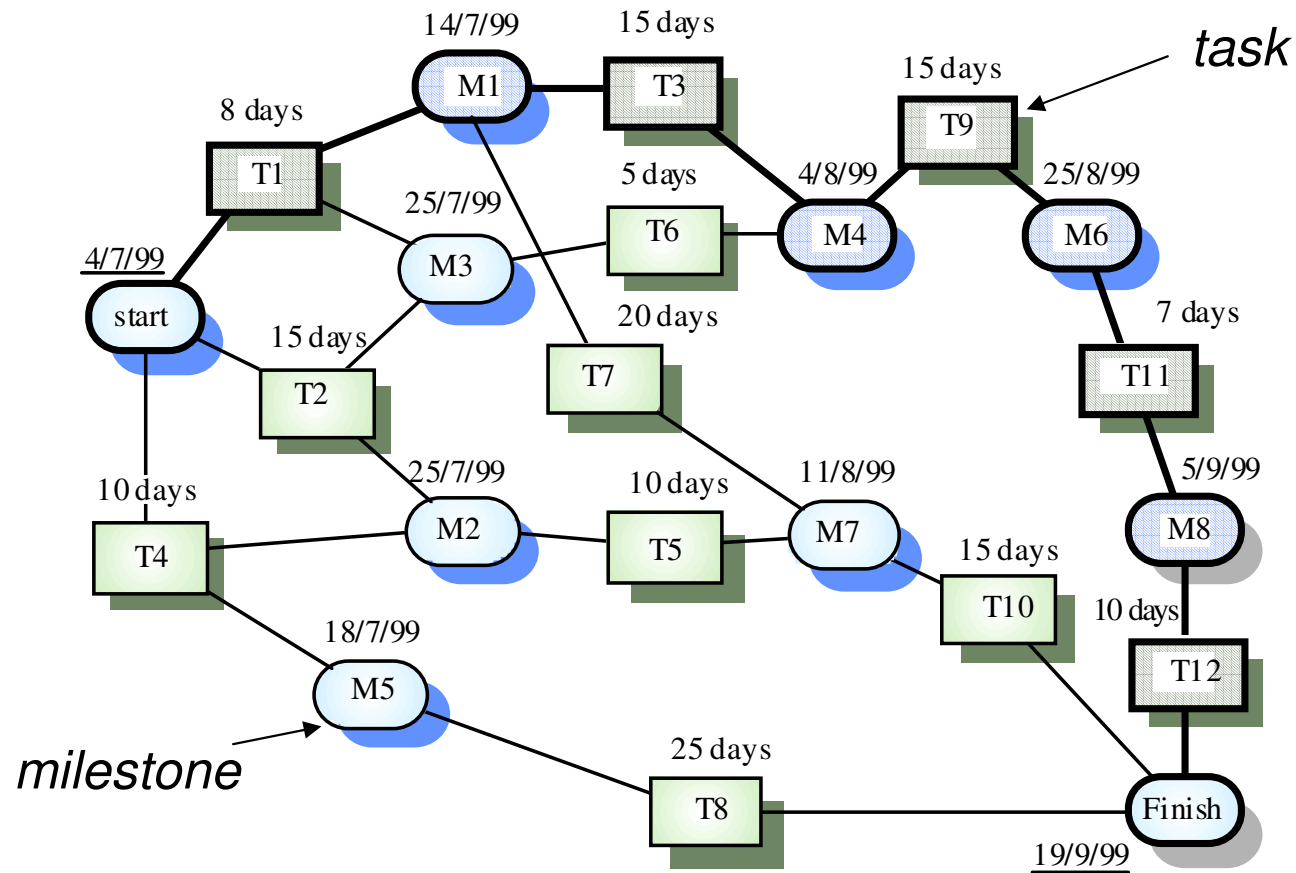
Il Processo di Scheduling di un Progetto



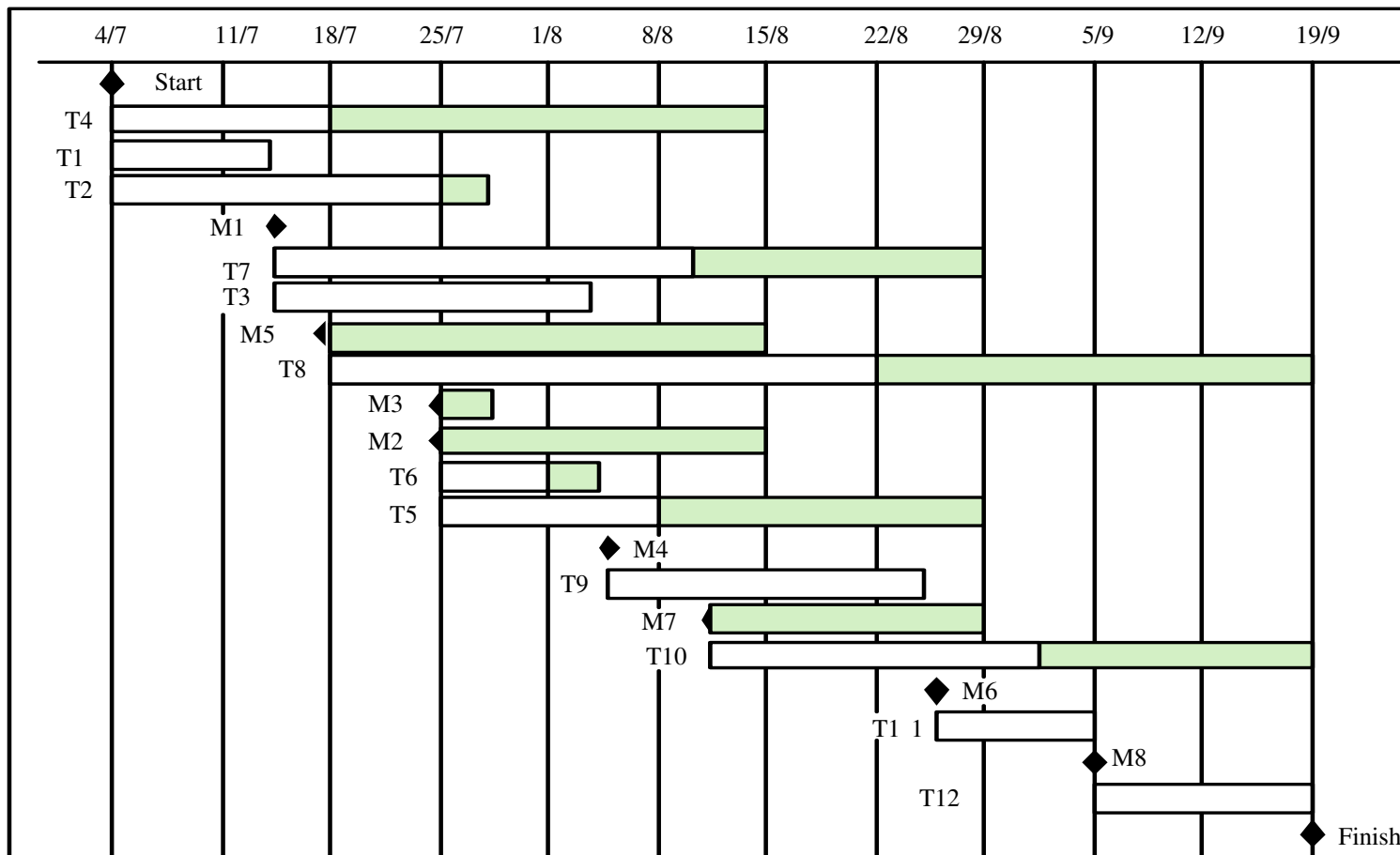
Diagrammi di Scheduling

- Notazioni grafiche utili per descrivere lo scheduling di un progetto
 - PERT (Program Evaluation Review Technique) chart, mostra le attività e le loro dipendenze,
 - Gantt chart, mostra lo schedule in funzione del tempo,
 - Man-month (MM) chart, mostra come viene impiegato il personale nel tempo.

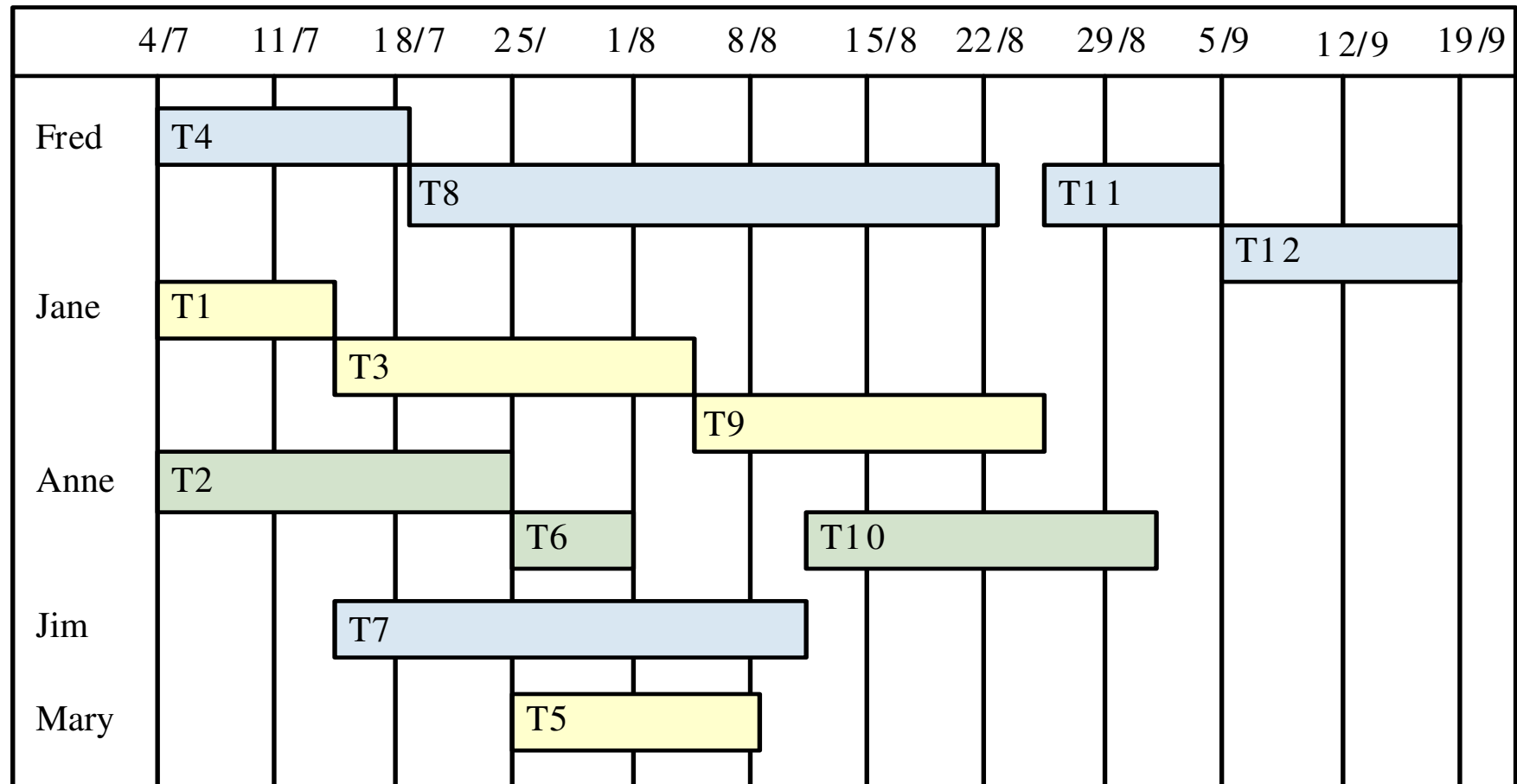
PERT Chart



Gantt Chart



Man-Month Chart



Approccio Orientato agli Oggetti (1/2)

- Nell'approccio orientato agli oggetti
 - modello di sviluppo non viene alterato,
 - cambia l'approccio adottato nei vari passi.
- Analisi
 - si basa sulle astrazioni di **attore** e **ruolo**,
 - assegna **responsabilità** ai ruoli,
 - modelli
 - statico del problema, **modello del dominio**,
 - dinamico del comportamento atteso dal sistema.

Approccio Orientato agli Oggetti (2/2)

- Progettazione
 - descrive il sistema con modelli statici e dinamici implementabili,
 - modelli
 - sono legati a scelte tecnologiche,
 - estendono i modelli di analisi per definire la parte del sistema non visibile agli utilizzatori,
 - statici, definiscono l'**architettura** del sistema,
 - dinamici, definiscono il comportamento del sistema e gli algoritmi implementati.
- Realizzazione
 - sfrutta le caratteristiche dei linguaggi orientati agli oggetti per implementare il progetto,
 - implementa classi ed interazioni tra oggetti.

Analisi e Specifica dei Requisiti (1/2)

- La **specifica** è un accordo tra il produttore di un servizio e il suo consumatore.
- Nelle diverse fasi dello sviluppo, le specifiche devono essere fornite ad un diverso livello di dettaglio
 - **requirement specification**, con cui il committente e lo sviluppatore si accordano sulle funzionalità del software,
 - **design specification**, con cui il progettista e i programmatori si accordano sulle caratteristiche strutturali dei moduli da implementare e sui servizi da mettere a disposizione,
 - **module specification**, con cui il progettista e i programmatori si accordano sulle interfacce dei vari moduli.

Analisi e Specifica dei Requisiti (2/2)

- Attributi di qualità dell'analisi e specifica dei requisiti sono
 - chiarezza,
 - non ambiguità
 - consistenza.
- Un **requisito**
 - è una descrizione delle caratteristiche richieste al sistema,
 - riguarda le interfacce che vengono esposte all'utilizzatore,
 - deve essere essenziale, per individuare le caratteristiche minime richieste al sistema.
- Gli artefatti di questa fase comprendono
 - modello del dominio,
 - casi d'uso.

Tipi di Requisiti

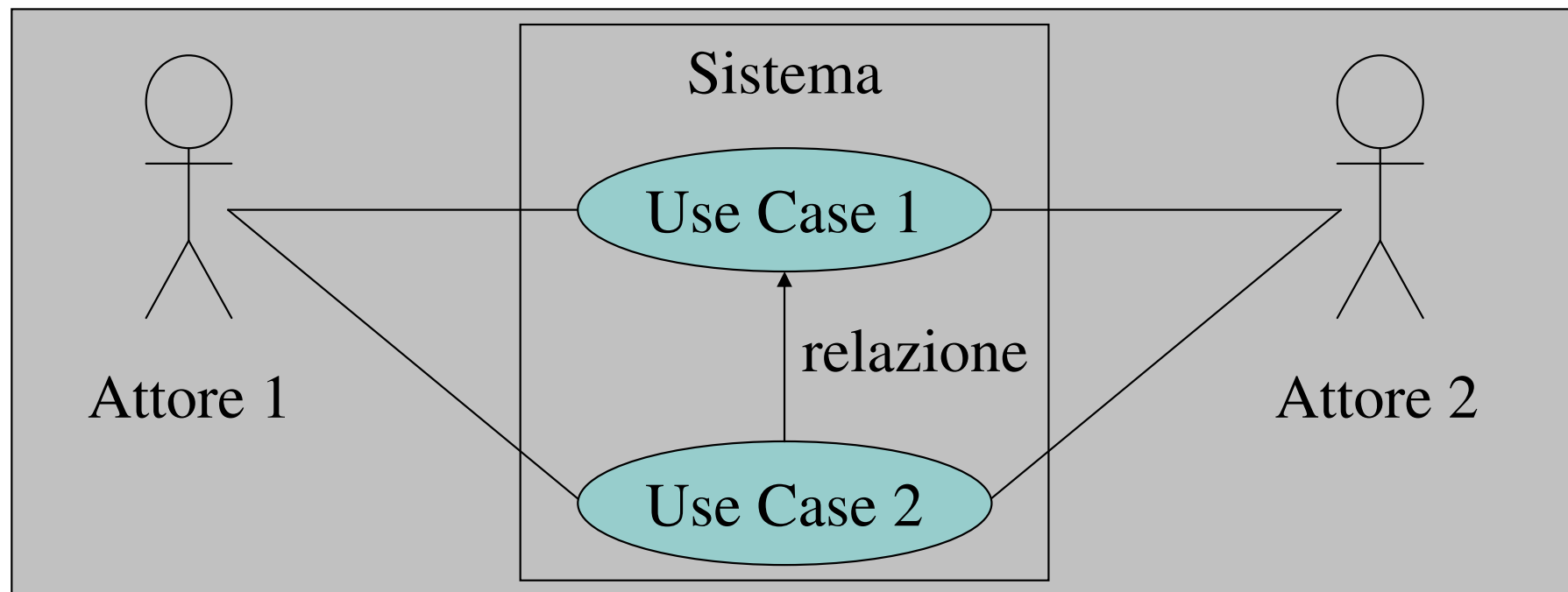
- **Requisiti funzionali**
 - come il sistema deve reagire agli input,
 - come il sistema deve comportarsi nelle varie situazioni.
- **Requisiti non-funzionali**
 - caratteristiche dei servizi offerti dal sistema
 - reattività,
 - affidabilità,
 - tolleranza ai guasti,
 - ...
- **Requisiti di dominio**
 - derivano dal particolare dominio applicativo.

Casi d'Uso

- Strumenti per analizzare e specificare il comportamento atteso dal sistema.
- Descrivono l'interazione di un attore con il sistema e sono composti da
 - use case diagram,
 - descrizione tabellare.
- Non descrivono i requisiti del sistema
 - illustrano il comportamento atteso dal sistema.
- Usati anche per la verifica funzionale.

Use Case Diagram

- Elenca i casi d'uso e le loro relazioni.
- Specifica quali attori sono coinvolti in quali casi d'uso.



Descrizione Tabellare

Use case: Use Case 1

Attori: Attore 1 (iniziatore), Attore 2.

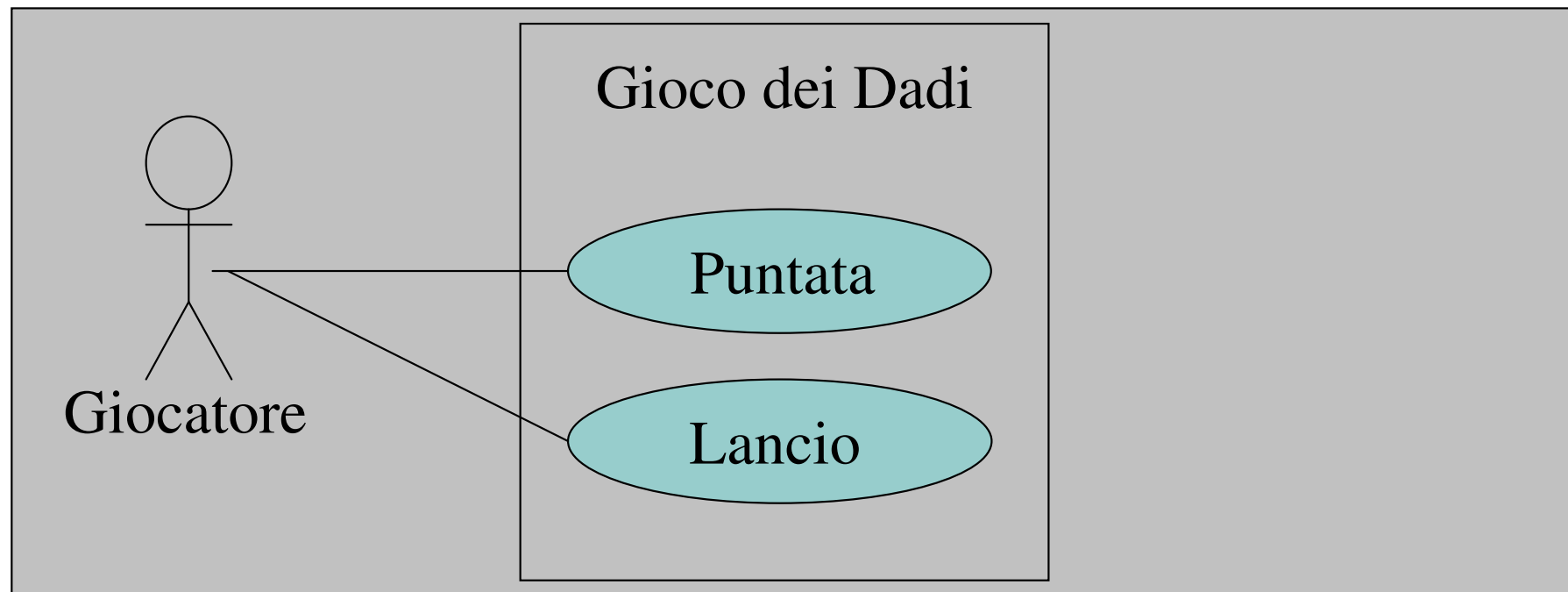
Tipo: Primario, secondario, essenziale.

Descrizione: Descrizione informale dello scenario.

Azione Attore	Risposta del Sistema
1. Attore1. Azione 1	
2. Attore 2. Azione 2	
	3. Risposta all'azione 2.
4. Attore 1. Azione 3.	
	5. Risposta all'azione 3.
...	...

Gioco dei Dadi (1/2)

- Due casi d'uso, Puntata e Lancio
 - nessuna relazione tra i casi d'uso.
 - l'unico attore è coinvolto in entrambi i casi d'uso.



Gioco dei Dadi (2/2)

Use case: Puntata.
Attori: Giocatore (iniziatore).
Tipo: Essenziale
Descrizione: Il giocatore punta una somma sul numero che uscirà nel prossimo lancio, indicando il numero atteso.

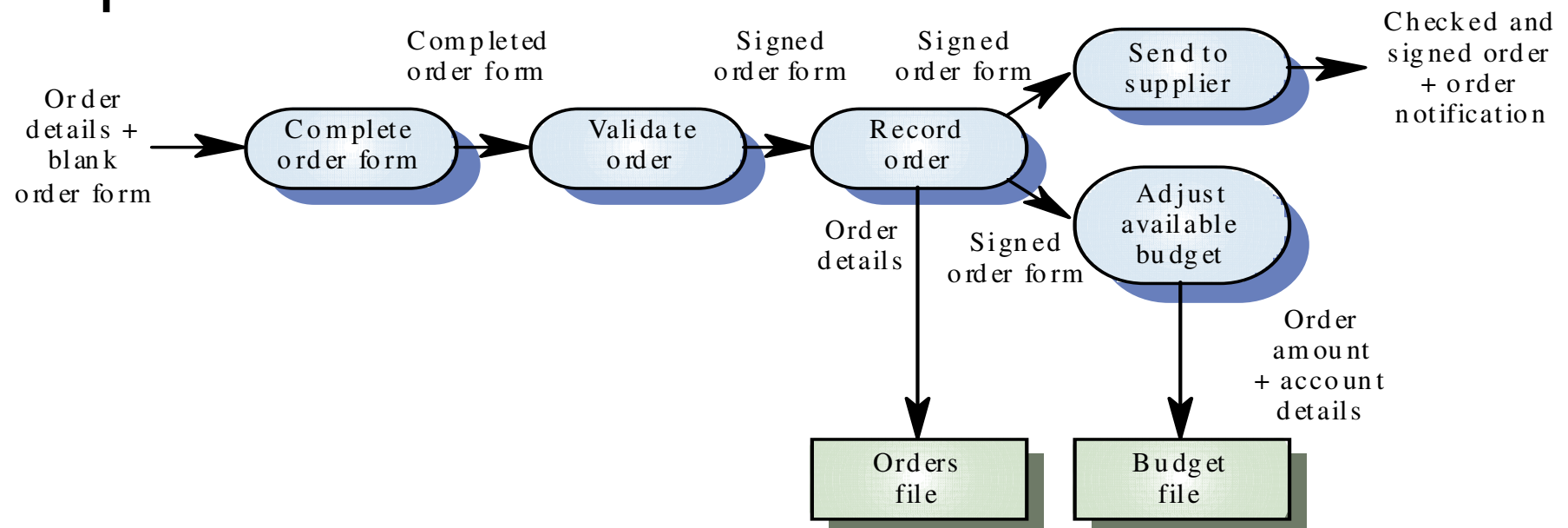
Azione Attore	Risposta Sistema
1. Il giocatore indica la somma ed il numero su cui punta.	
	2. Il sistema accetta la puntata, indicando numero e somma ricevuta.
Eccezioni	
1. Il numero giocato non è compreso tra 1 e 6.	
2. La somma giocata è maggiore della disponibilità del giocatore.	
...	

Identificazione dei Casi d'Uso

- Identificazione **actor-based**
 - attori nel dominio del problema,
 - per ogni attore, processi a cui partecipa.
- Identificazione **process-based**
 - processi a cui il sistema partecipa,
 - relazioni tra attori e processi.
- Gli interaction diagram di UML consentono di descrivere i casi d'uso.

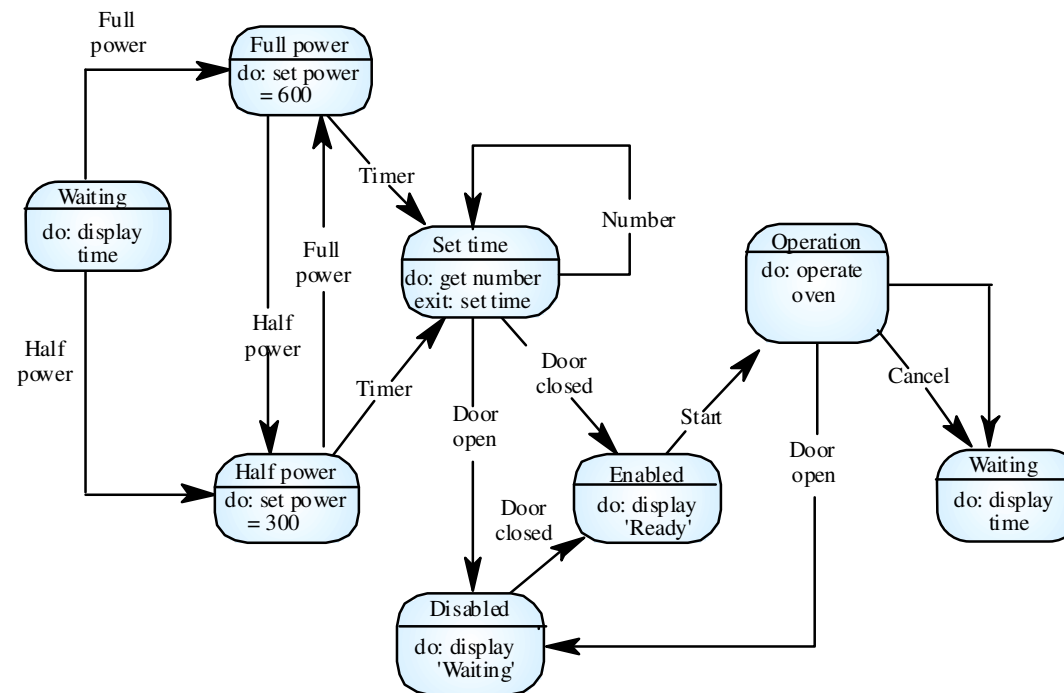
Diagramma di Flusso dei Dati

- I **data-flow diagram** sono usati per modellizzare come i dati vengono processati nel sistema.



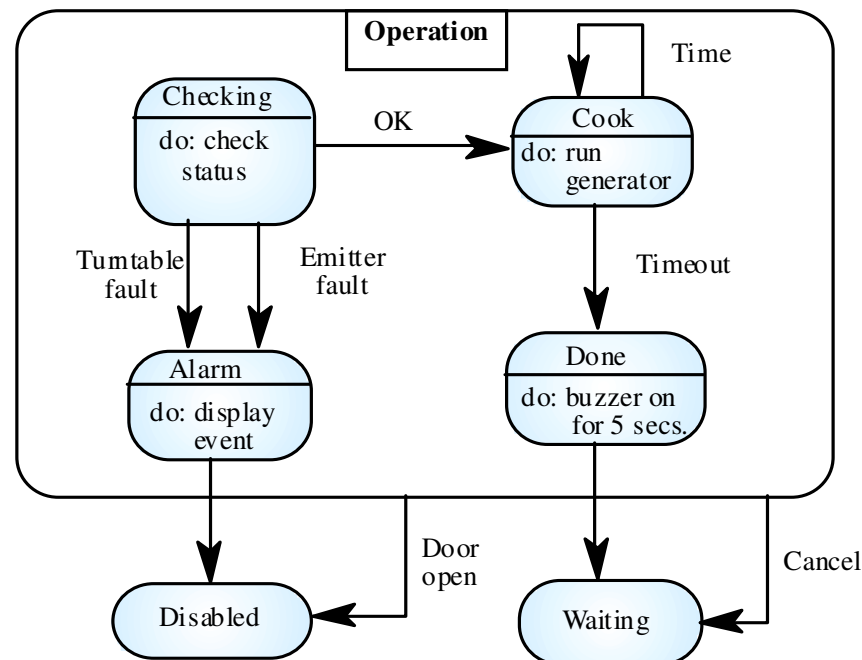
Macchine a Stati

- Modello a stati del comportamento del sistema
 - **stati** in cui si può trovare il sistema,
 - **eventi** che causano le transizioni di stato.



Statechart UML

- Macchine a stati decomponibili in modo iterativo.
- Devono essere completati da tabelle che descrivano gli stati e gli eventi.



Dizionario dei Dati

- Lista di tutti i termini (con relativo significato) utilizzato nei modelli.
- Scopo
 - dare un significato comune a tutti di termini anche comuni,
 - evitare la replicazione dei termini,
 - offrire una base per l'individuazione di astrazioni utili durante la fase di progetto.
- Può essere costruito facilmente partendo dalle descrizioni tabellari degli use case.

Modello del Dominio

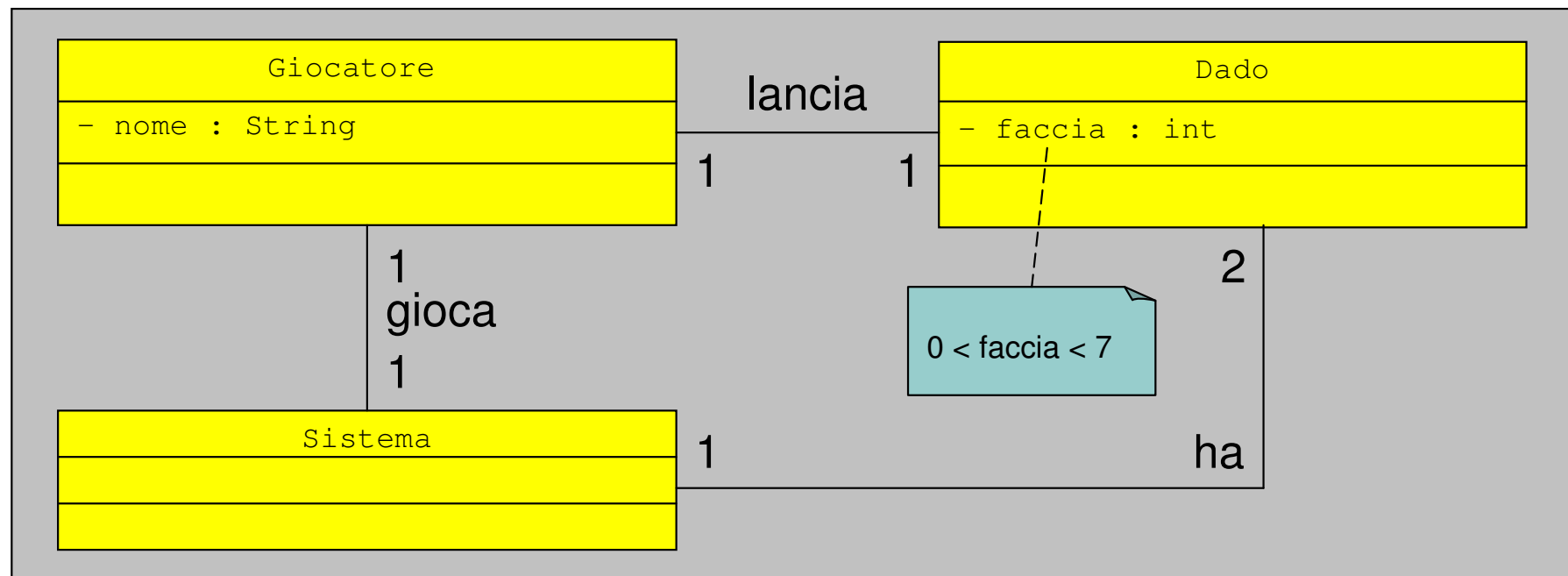
- Descrive il dominio del problema
 - **classi di entità** nel dominio,
 - **classi di attori**, classi di entità che interagiscono con il sistema,
 - **relazioni** entità/entità, entità/attori.
- Ogni entità è caratterizzata da una classe di appartenenza definita da
 - un nome,
 - un insieme di attributi.
- Ogni attributo è definito da un insieme di valori possibili.

Identificazione del Modello del Dominio

- Partendo dalla descrizione tabellare dei casi d'uso, si utilizzano le regole
 - un sostantivo indica un attore o un'entità,
 - un verbo indica una relazione o un'azione compiuta da un attore,
 - gli attributi sono relazioni che interessano tipi dato primitivi.
- I class diagram di UML consentono di descrivere i modelli del dominio.
- I class diagram ricordano i modelli entità/relazioni, ma
 - nei modelli entità/relazioni, le relazioni vengono definite tra le singole entità,
 - nei modelli ad oggetti, le relazioni vengono definite tra le classi di entità.

Gioco dei Dati

- Dai casi d'uso
 - due sostantivi: giocatore, dado;
 - un verbo: lancia.



Relazioni Comuni

- Alcune relazioni sono molto comuni
 - A è fisicamente/logicamente parte di B,
 - A è fisicamente/logicamente contenuto in B,
 - A descrive B,
 - A possiede B,
 - A usa/gestisce B,
 - A comunica con B,
 - A è posseduto da B.
- Le relazioni comuni consentono di distinguere facilmente tra relazioni ed attributi.

Modello Comportamentale

- Il modello del dominio descrive staticamente il sistema.
- Il modello comportamentale
 - descrive il comportamento dinamico del sistema in reazione ad alcuni stimoli o eventi,
 - viene formalizzato con un insieme di interaction diagram che contengono
 - gli attori,
 - il sistema,
 - gli eventi ed i messaggi generati o ricevuti dal sistema e dagli attori.

Messaggi e Contratti

- Ogni messaggio viene descritto mediante un contratto

Nome:	Nome del messaggio
Responsabilità:	Effetti associati al messaggio.
Output:	Output del messaggio.
Pre-condizioni:	Condizioni da verificarsi.
Post-condizioni:	Condizioni garantite.
Eccezioni:	Condizioni di non nominali.

- Esempio

Nome:	<code>punta(numero, somma)</code>
Responsabilità:	Puntare una <code>somma</code> su un <code>numero</code> .
Output:	<code>nessuno</code> .
Pre-condizioni:	<code>somma</code> minore della disponibilità. <code>somma</code> minore del massimo.
Post-condizioni:	<code>nessuna</code> .
Eccezioni:	<code>somma</code> maggiore della disponibilità. <code>somma</code> minore del massimo.

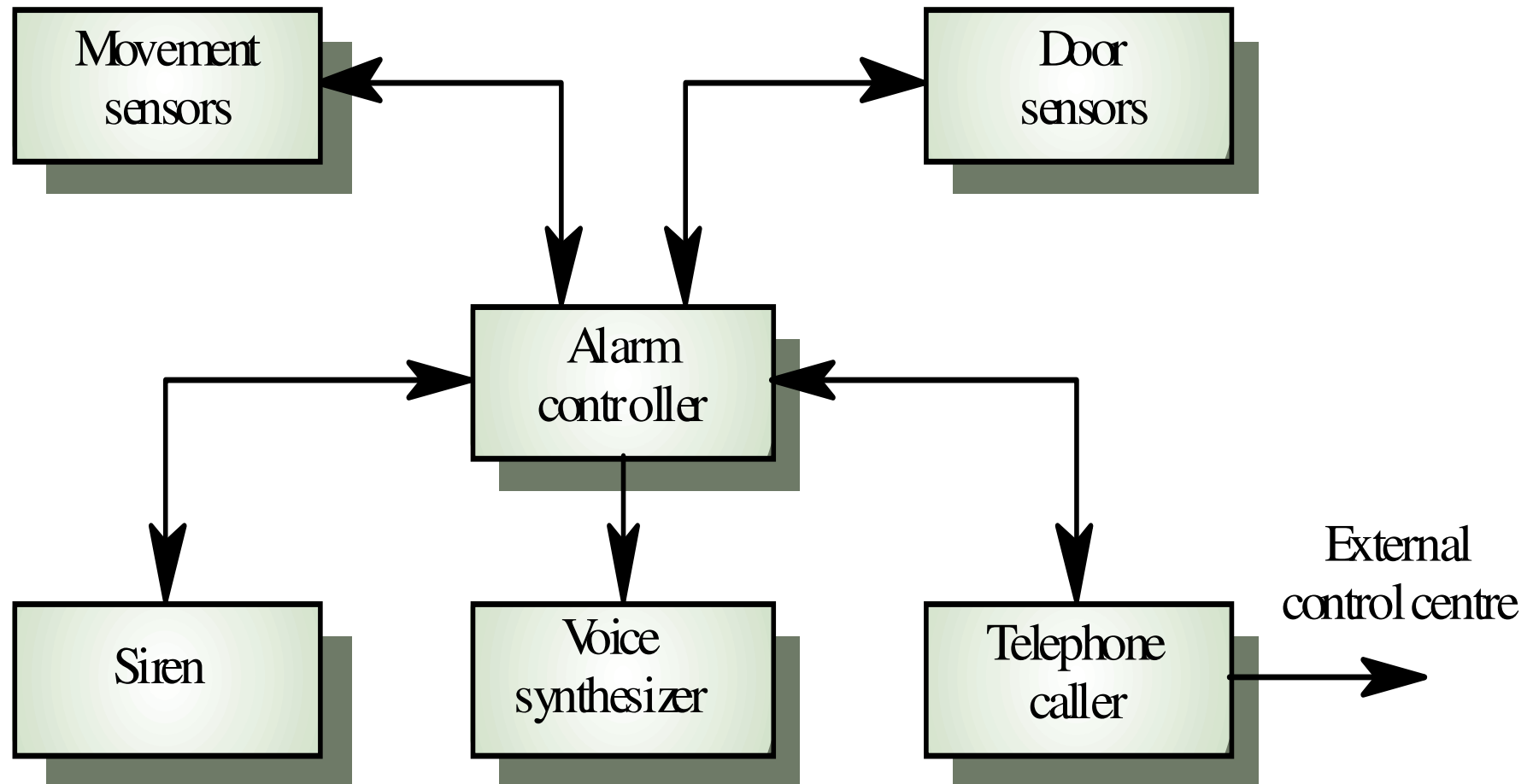
Progettazione (1/2)

- Processo che trasforma le specifiche del committente in un insieme di specifiche direttamente utilizzabili dai programmatori.
- Il risultato è l'**architettura** del sistema
 - insieme dei moduli che compongono il sistema,
 - descrizione delle loro funzioni,
 - descrizione delle loro relazioni.

Progettazione (2/2)

- Modularizzazione
 - il sistema è decomposto in **sotto-sistemi** e **moduli**.
- Modello del controllo
 - un modello di come il controllo viene distribuito tra la parti del sistema.
- Decomposizione dei moduli
 - i singoli moduli sono decomposti in **componenti**.
- Diversi livelli di dettaglio
 - un sotto-sistema è un sistema le cui operazioni non dipendono dai servizi di altri sotto-sistemi,
 - un modulo è una parte di un (sotto-)sistema che fornisce servizi ad altri moduli,
 - un componente è una parte di un modulo che può essere implementato direttamente
 - library di classi e oggetti in Java.

Sistema d'Allarme



Modularizzazione (1/3)

- Un modulo
 - raccoglie un insieme di funzionalità tra loro strettamente legate,
 - deve essere definito in due fasi
 - design **architetturale**, specifica il comportamento di un modulo in relazione all'interazione con altri moduli,
 - design **in dettaglio**, definisce le funzionalità di ogni singolo modulo, indicando come le funzionalità debbano essere realizzate.
- Un modulo è completamente specificato quando il livello di dettaglio delle specifiche ha un'interpretazione non ambigua e completa da parte dei programmatori.

Modularizzazione (2/3)

- Suddividere un sistema in moduli necessita di tracciare le interazioni tra i moduli.
- Le relazioni più comuni sono
 - utilizzo (**uses**), indica quali moduli vengono utilizzati per completare i servizi forniti da un particolare modulo,
 - composizione (**part-of**), descrive la struttura del sistema a diversi livelli di dettaglio,
 - dipendenza (**depends-on**), descrive la sequenza con cui possono essere realizzati i diversi moduli.

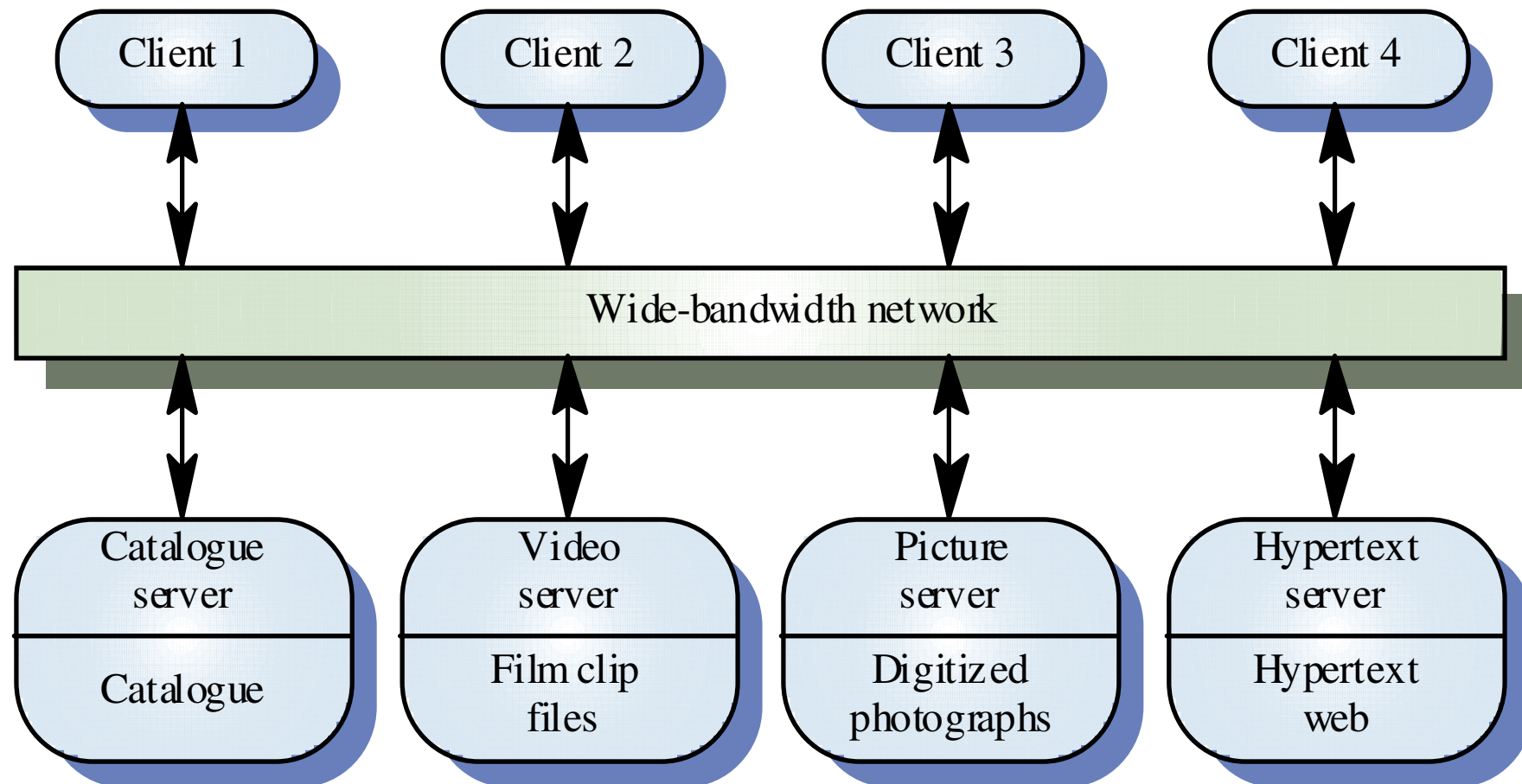
Modularizzazione (3/3)

- La strategia da utilizzare nella definizione dei moduli può essere
 - **top down**, decompone progressivamente i moduli in unità più piccole a partire da una visione globale ed astratta
 - garantisce che la progettazione sia fatta partendo da una visione globale,
 - permette di realizzare una documentazione di più facile comprensione.
 - **bottom up**: aggrega i componenti di base in moduli che descrivono il sistema a complessità crescente
 - permette di analizzare in dettaglio le strutture dati utilizzate dalle procedure,
 - non comporta un prematuro irrigidimento della struttura del sistema.
- I progettisti non operano mai seguendo una sola strategia.

Architettura Client-Server

- Tipica dei sistemi distribuiti
 - descrive come l'elaborazione e i dati sono gestiti da un insieme di componenti.
- È formata da
 - un insieme di **server** che offrono servizi di uso generale,
 - un insieme di **client** di sfruttano i servizi messi a disposizione dai server.
 - una rete, con una certa topologia, che garantisce la connettività tra client e server.

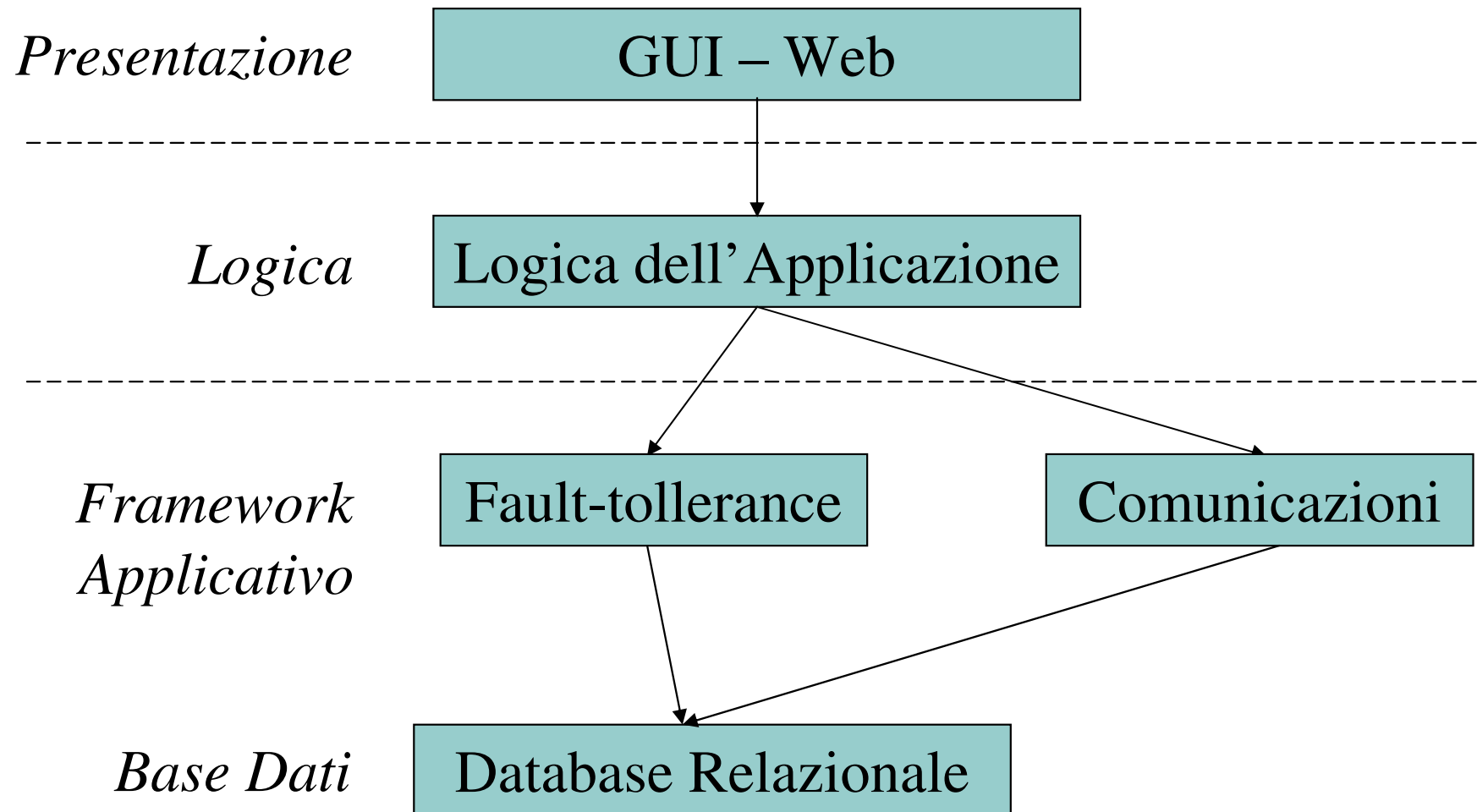
Servizi Multimediali



Caratteristiche dell'Architettura Client-Server

- Vantaggi
 - la distribuzione dei dati e del controllo è semplice,
 - è semplice aggiungere nuovi client,
 - è semplice aggiungere e/o sostituire i server.
- Svantaggi
 - il solo scambio dei messaggi può impegnare molte risorse,
 - gestione spesso ridondante dei singoli server.

Architettura a Tre Livelli (1/2)



Architettura a Tre Livelli (2/2)

- Tipica di un sistema informativo
 - modulare,
 - generale ed adattabile a molte applicazioni,
 - isola la logica dell'applicazione dai componenti riutilizzabili,
 - consente di distribuire i livelli o i componenti tra diversi processi.
- Alla base della **Java 2 Enterprise Edition (J2EE)**.

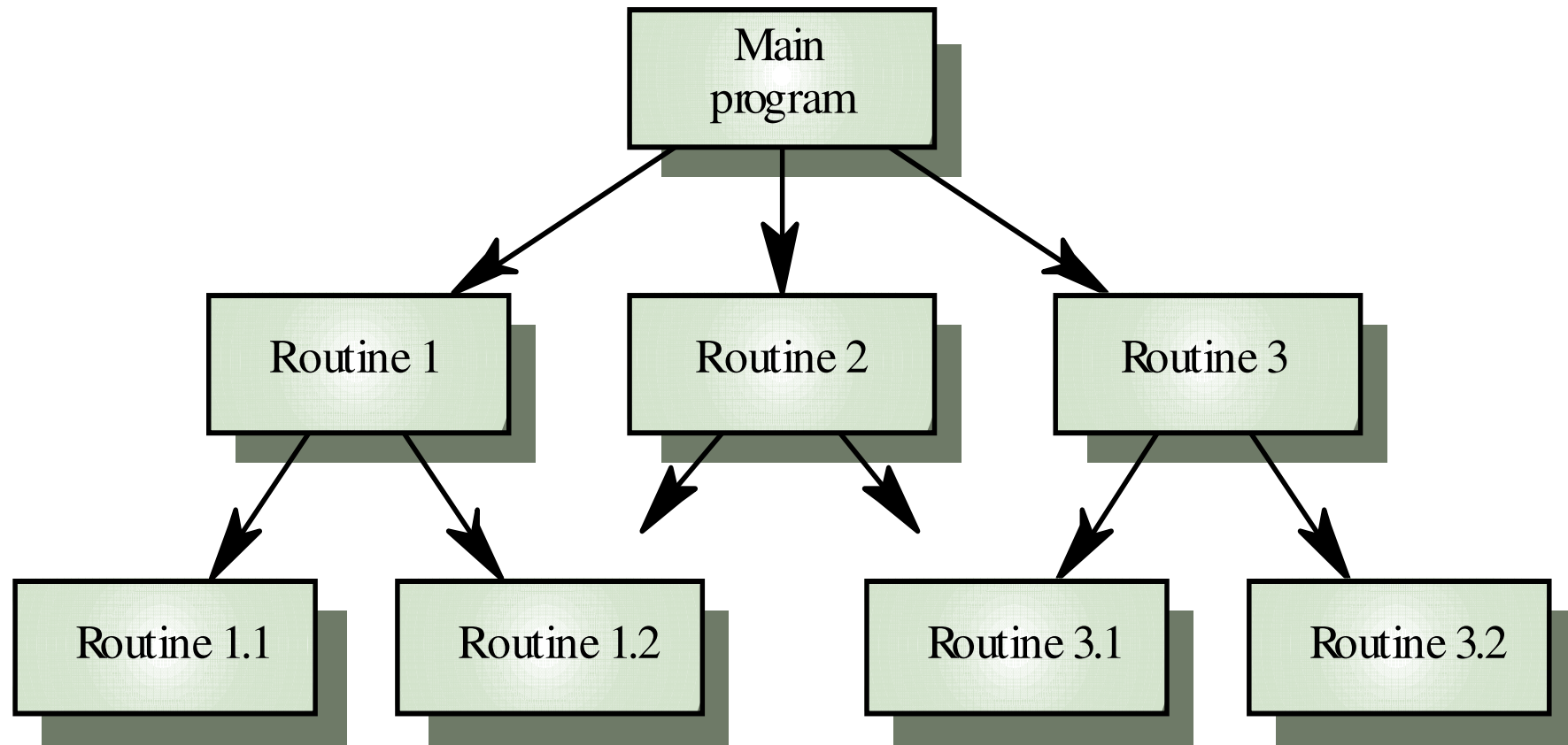
Progettazione del Controllo

- Consiste nel distribuire il controllo tra i sotto-sistemi.
- Controllo centralizzato
 - un sotto-sistema ha la responsabilità completa del controllo.
- Controllo ad eventi
 - ogni sotto-sistema può rispondere autonomamente agli eventi provenienti
 - da altri sotto-sistemi,
 - dall'ambiente.

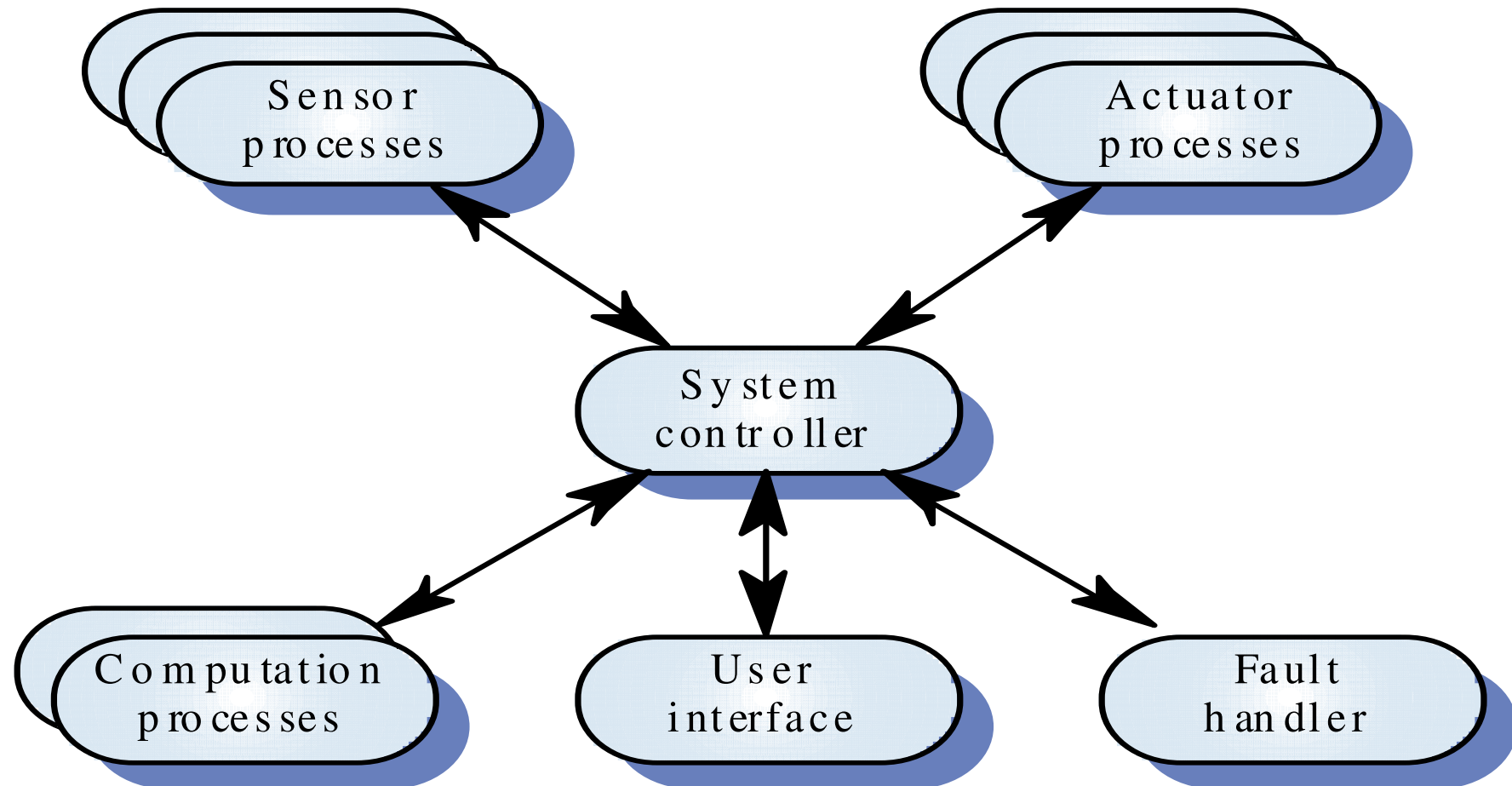
Controllo Centralizzato

- Nell'architettura è presente un sotto-sistema con lo scopo di garantire il controllo degli altri sotto-sistemi.
- Modello **request/response**
 - modello a sub-routine in cui il controllo parte dall'alto e si dirama nella gerarchia di sub-routine,
 - applicabile praticamente solo a sistemi sequenziali.
- Modello a **master/slave**
 - un sotto-sistema controlla
 - l'attivazione,
 - lo spegnimento,
 - la coordinazione delle attività degli altri sotto-sistemi,
 - applicabile a sistemi concorrenti.

Modello Request/Response



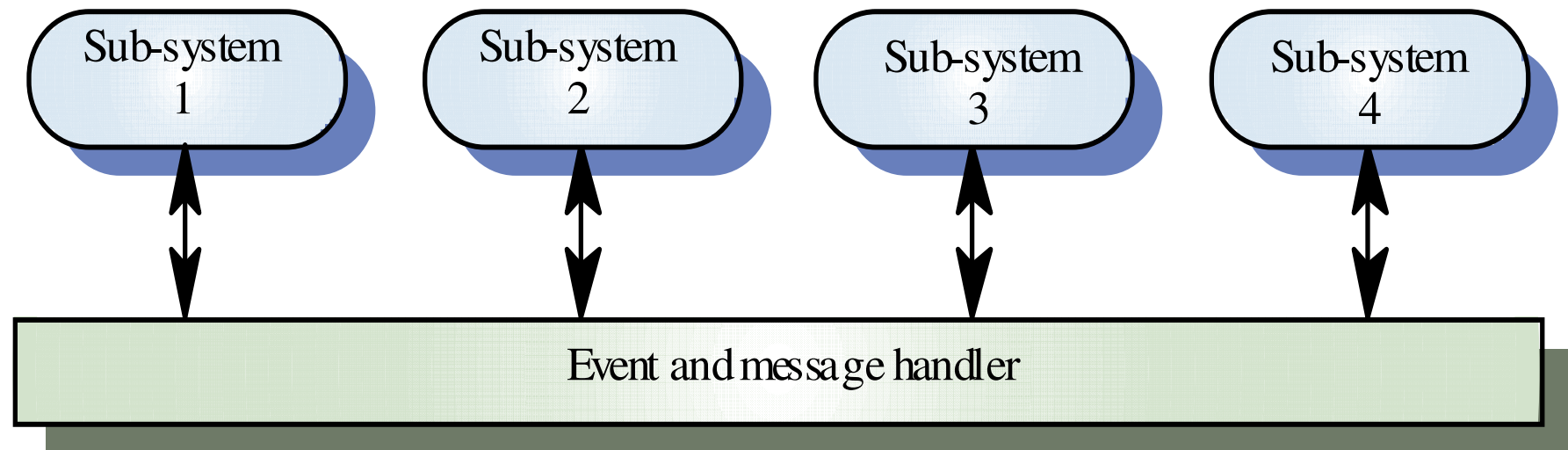
Modello Master/Slave



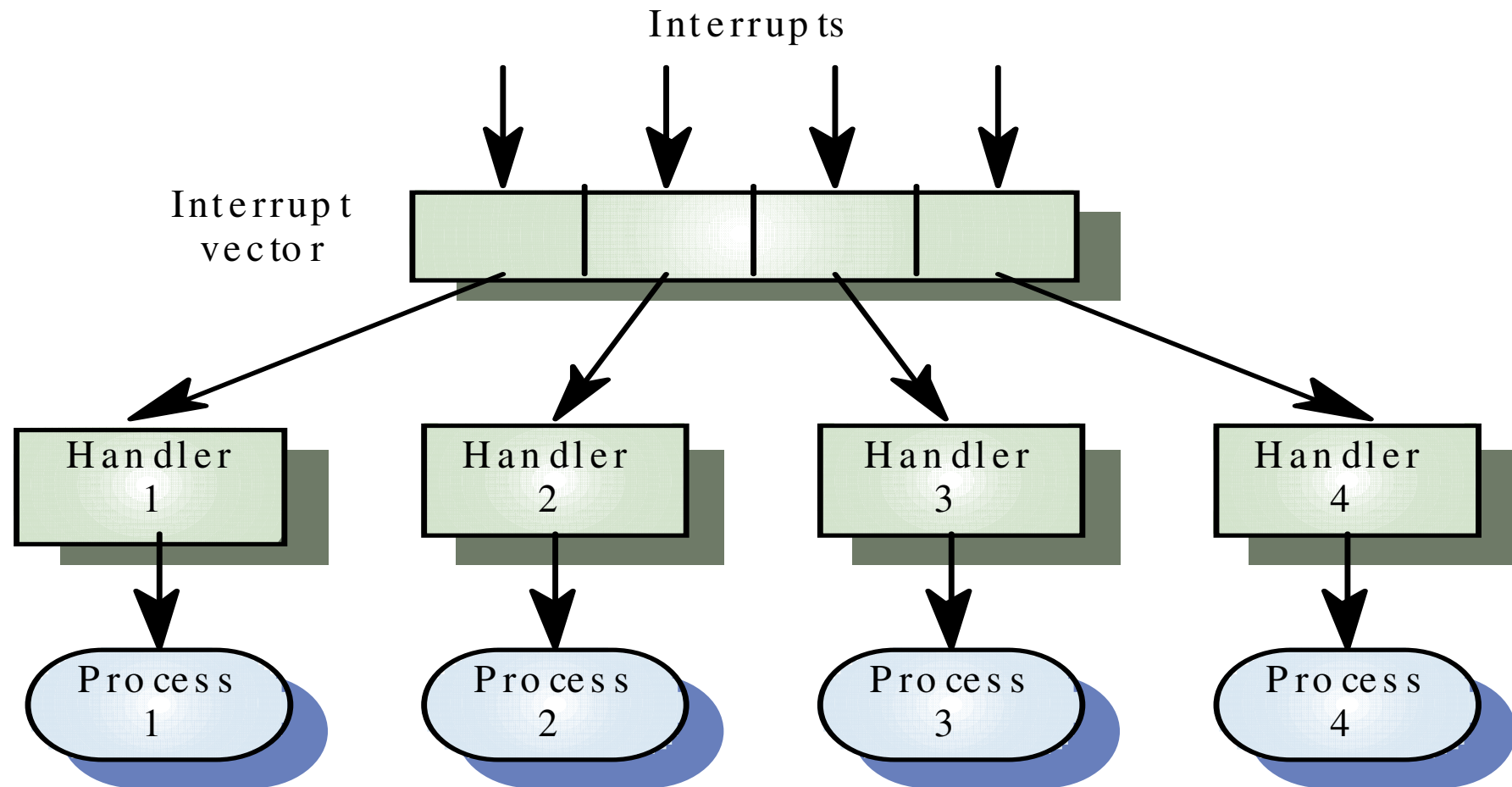
Controllo ad Eventi

- Ogni sotto-sistema risponde autonomamente agli eventi esterni
 - modello **broadcasting**, in cui ogni evento è inviato a tutti i sotto-sistemi,
 - modello ad **interrupt**, usato nei controlli tempo-critici dove c'è un **interrupt handler**
 - riceve gli interrupt,
 - opera un **dispatch** ai sotto-sistemi interessati.

Modello Broadcasting



Modello ad Interrupt



Progettazione in Dettaglio

- Specifica le caratteristiche proprie di un modulo ed indica al programmatore quali servizi esso debba fornire.
- Nella suddivisione delle funzionalità del sistema in moduli il progettista deve considerare che
 - ogni modulo viene realizzato in modo il più possibile indipendente da ogni altro,
 - i programmatori devono essere in grado di operare su un modulo avendo una conoscenza minima del contenuto degli altri,
 - tutti i servizi strettamente connessi devono appartenere allo stesso modulo.
- Particolarmente importante è la definizione dell'**interfaccia** del modulo, ossia la descrizione dei servizi fruibili da parte degli utilizzatori.

Interfaccia di un Modulo

- L'interfaccia di un modulo contiene tutte le informazioni necessarie all'utilizzo del modulo
 - funzionalità a disposizione, deve essere ben chiaro quali servizi sono realizzati dal modulo,
 - modalità di fruizione di un servizio, per ogni servizio è necessario indicare la sequenza di funzioni da invocare,
 - definizione dei parametri di input, il tipo e il numero dei parametri di input devono essere specificati in modo chiaro,
 - descrizione dell'output, il tipo dei valori restituiti dalle funzioni deve essere completamente specificato, comprese le condizioni di eccezione e di errore.

Progettazione Orientata agli Oggetti

- Scomposizione orientata agli oggetti
 - ogni modulo viene strutturato in un insieme di oggetti
 - tra loro il più possibile disaccoppiati,
 - con interfacce significative e ben definite,
 - vengono identificate le classi, i metodi e gli attributi,
 - il livello di dettaglio è ancora sufficientemente alto.
- Due tipologie di modelli
 - modello statici,
 - class diagram,
 - modelli dinamici
 - interaction diagram.

Relazione con i Modelli di Analisi

- I modelli di progetto derivano dai modelli di analisi.
- UML prevede notazioni simili sia per l'analisi che per il progetto
 - in fase di analisi, le notazioni descrivono gli attori e le relazioni nel dominio del problema;
 - in fase di progetto, le notazioni descrivono gli oggetti e le cooperazioni tra gli oggetti.

Progettare per il Riuso

- Un sistema di classi è ben progettato se
 - le classi esprimono concetti ben individuabili,
 - gli stati descritti delle classi sono semplici,
 - i metodi delle classi sono pochi e il loro scopo è ben preciso (ma generale),
 - sono minimizzate associazioni e dipendenze,
 - le strutture dati sono tenute separate dagli algoritmi,
 - gli algoritmi non dipendono da come le strutture dati sono memorizzate,
 - ...
- Per una migliore progettazione conviene sfruttare dei **design pattern**.

Design Pattern (1/2)

- Nella progettazione si incontrano problemi ricorrenti.
- È utile trovare soluzioni **riusabili** per trattare problemi ricorrenti
 - **design pattern.**



Design Pattern (2/2)

- Sono soluzioni utilizzate in progetti reali per risolvere problemi comuni.
- Sono indipendenti dal linguaggio di programmazione.
- Vengono raggruppati in **pattern language**
 - raccolte divise secondo categorie di problemi affrontati.
- Variano da semplici soluzioni programmatiche a complesse architetture di sistema.
- Se utilizzati durante la fase di sviluppo, vengono detti **idiomi programmatici**.

Storia dei Design Pattern

- Lavori dell'architetto Christopher Alexander
 - introduce il termine “pattern” nel 1977.
- Erich Gamma, tesi di Ph.D., 1991.
- James Coplien, Advanced C++ Idioms, 1989.
- Gamma, Helm, Johnson, Vlissides (**Gang of Four - GoF**), Design Patterns: Elements of Reusable Object-Oriented Software, 1991.
- Buschmann, Meunier, Rohnert, Sommerland, Stal, Pattern-Oriented Software Architecture: A System of Patterns, 1996.
- Conferenze PLoP, iniziate nel 1994.

Definizioni

- *...describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice, C. Alexander.*
- *...the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts, D. Riehle.*
- *...both a thing and the instructions for making the thing, J. Coplien.*

Caratteristiche

- I design pattern consentono di
 - definire un vocabolario comune,
 - comunicare concetti complessi in modo semplice,
 - documentare i progetti,
 - catturare parti essenziali di un progetto in una forma compatta,
 - descrivere astrazioni.
- I design pattern non offrono
 - soluzioni esatte e complete,
 - soluzioni a tutti i problemi di progettazione.

Formato GoF di un Design Pattern (1/2)

- Nome e tipo
- Intento
 - cosa fa il pattern e quando la soluzione è applicabile e può portare benefici.
- AKA (Also Known As)
 - altri nomi in uso per il pattern.
- Motivazione
 - problemi che sono stati risolti mediante l'uso del pattern
- Applicabilità
 - situazioni dove il pattern può essere applicato e può portare beneficio

Formato GoF di un Design Pattern (2/2)

- **Struttura**
 - descrizione della soluzione,
 - indica i partecipanti e le collaborazioni.
- **Conseguenze**
 - trade off e questioni che nascono dall'impiego del pattern.
- **Implementazione**
 - suggerimenti e tecniche utili per implementare il pattern.
- **Codice d'esempio.**
- **Usi noti**
 - sistemi reali in cui il pattern è stato utilizzato con successo.
- **Pattern correlati.**

Pattern Language

- Definizioni di Coplien
 - *...a structured collection of patterns that build on each other to transform needs and constraints into an architecture.*
 - *...defines collection of patterns and rules to combine them into an architectural style...describe software frameworks or families of related systems.*

Pattern GoF

- I pattern più noti sono i GoF
- I pattern GoF sono classificati in
 - **creazionali**, gestiscono la creazione dinamica degli oggetti all'interno di un sistema;
 - **strutturali**, micro-architetture statiche di utilizzo generale;
 - **comportamentali**, descrivono il comportamento di un'architettura di oggetti.

Pattern Creazionali

- **Abstract factory**, utilizzata per creare oggetti senza conoscerne l'implementazione concreta.
- **Builder**, usato per creare oggetti complessi indipendentemente dalla loro rappresentazione interna.
- **Prototype**, usato per creare oggetti partendo da un'istanza prototipo.
- **Singleton**, usato per garantire che una sola istanza di un oggetto venga creata nel sistema.

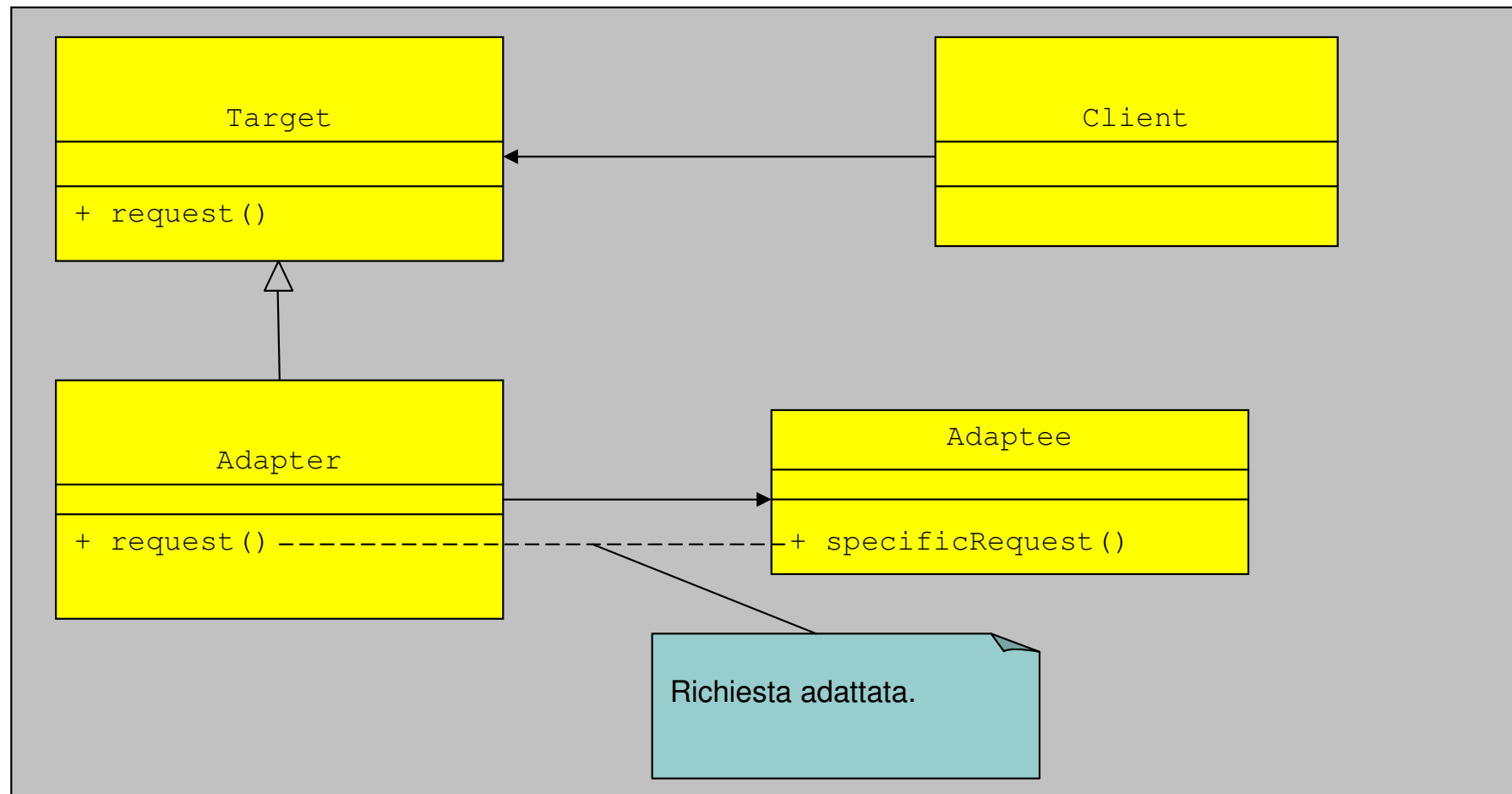
Pattern Strutturali

- Gestiscono il modo in cui classi e oggetti vengono composti per formare architetture complesse.
- **Adapter**, usato per risolvere problemi di incompatibilità tra interfacce.
- **Bridge**, usato per fornire più implementazioni ad un'interfaccia.
- **Composite**, usato per trattare in modo uniforme oggetti singoli e gruppi di oggetti.
- **Façade**, usato per fornire un'interfaccia unificata ad un gruppo di oggetti.
- **Proxy**, usato per fornire funzionalità aggiuntive ad un oggetto.

Adapter (1/2)

- Consente ad un oggetto di essere utilizzato mediante un'interfaccia che non implementa.
- L'adapter è uno dei pattern più usati
 - consente a oggetti diversi di essere utilizzati insieme anche se le rispettive interfacce non sono compatibili,
 - aumenta il riuso delle classi perché consente di utilizzarle in situazioni non previste inizialmente senza doverle modificare.
- Un adapter viene usato
 - quando si vuole utilizzare una classe che non offre un'interfaccia corretta, senza modificarla,
 - per creare una classe che utilizza i servizi di un'altra classe di cui non è ancora nota l'interfaccia.

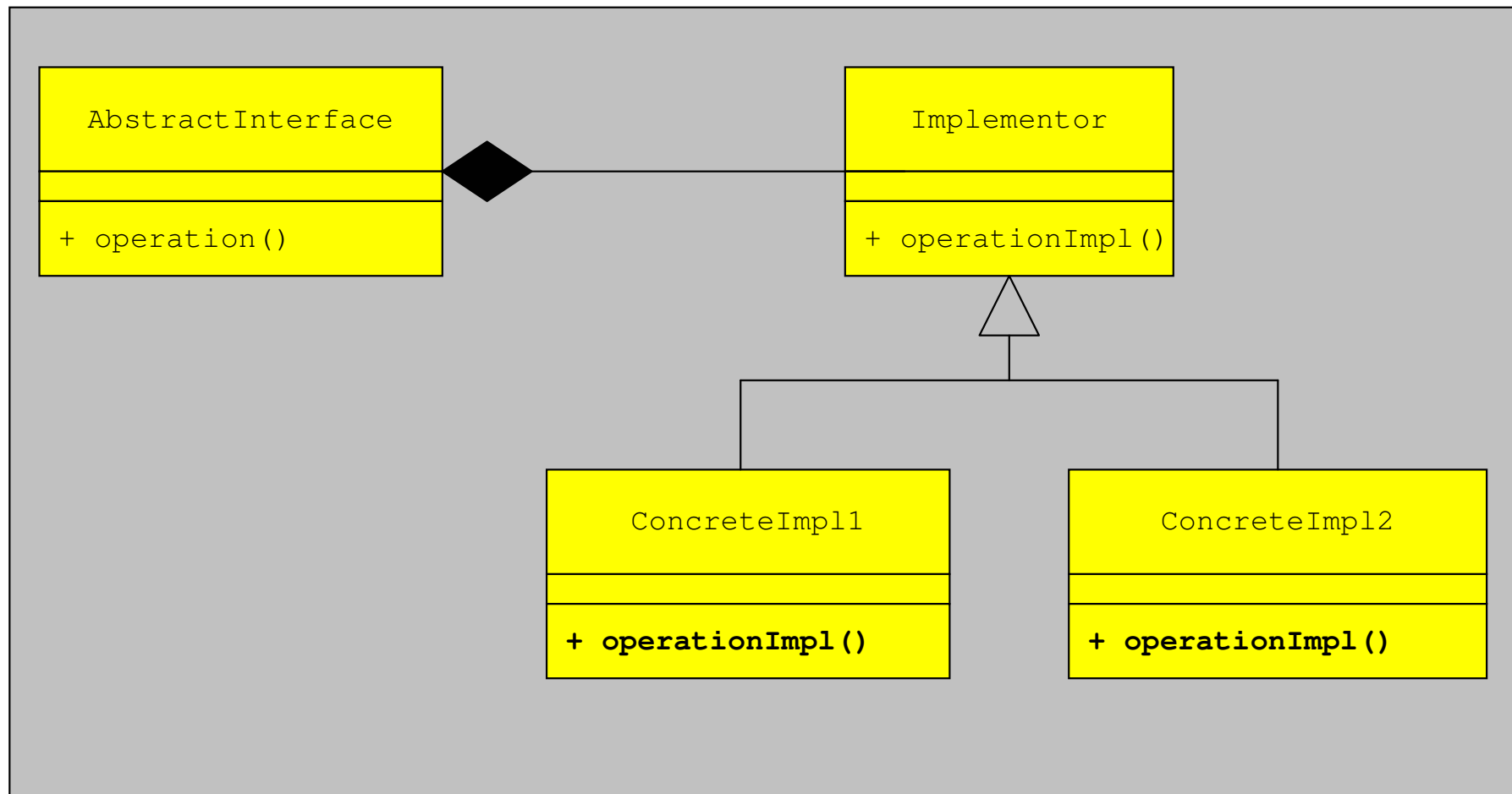
Adapter (2/2)



Bridge (1/2)

- Disaccoppia un'astrazione dalla sua implementazione
 - la classe di implementazione e la classe degli utilizzatori possono variare indipendentemente.
- Un bridge viene usato quando
 - si vuole evitare di legare un'astrazione ad una sola implementazione,
 - ogni cambiamento di un'implementazione non è influente sugli utilizzatori,
 - per nascondere dettagli implementativi.

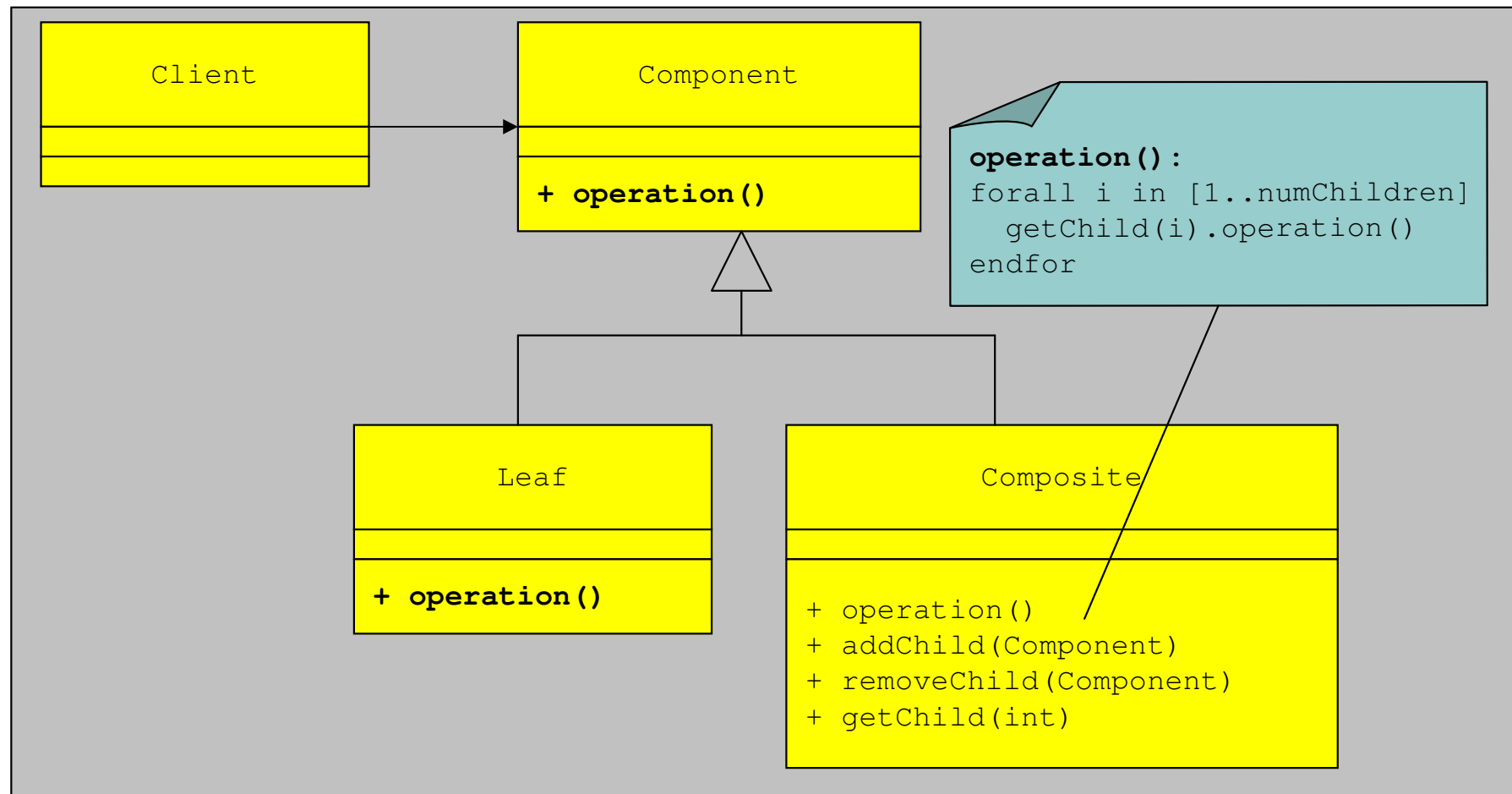
Bridge (2/2)



Composite (1/2)

- Consente ad oggetti utilizzatori di creare strutture ad albero in modo che sia possibile trattare in modo uniforme
 - oggetti **foglia**,
 - oggetti **contenitore**.
- Un composite viene usato quando
 - si vuole rappresentare gerarchie di oggetti,
 - si vuole consentire di ignorare le differenze tra oggetti foglia e oggetti contenitore.

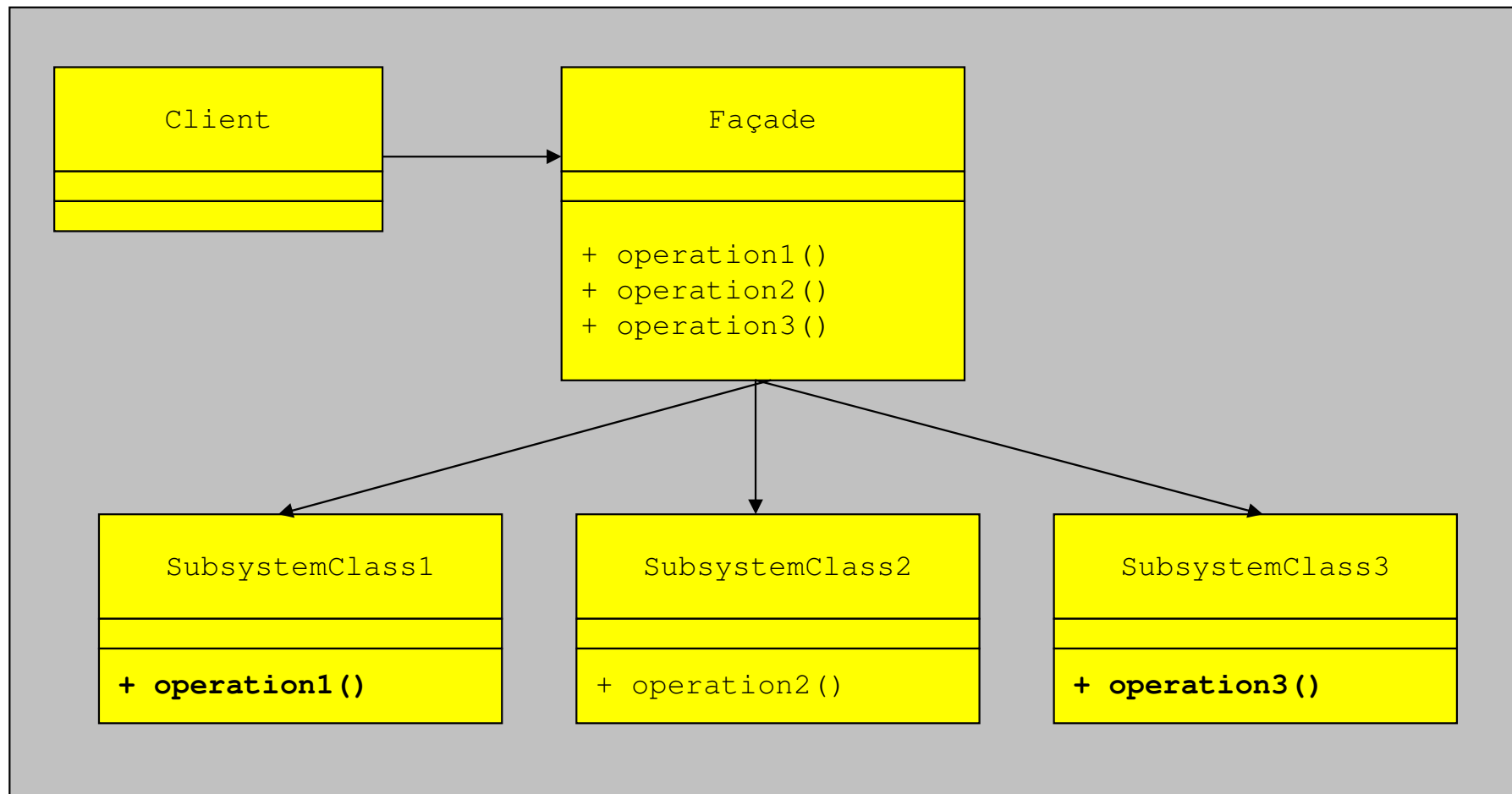
Composite (2/2)



Façade (1/2)

- Offre un'interfaccia uniforme ad un insieme di interfacce correlate.
- Una Façade viene utilizzata quando
 - si vuole offrire un'interfaccia uniforme ad un sotto-sistema complesso,
 - si vuole disaccoppiare un sotto-sistema dai suoi utilizzatori, riducendo le inter-dipendenze.

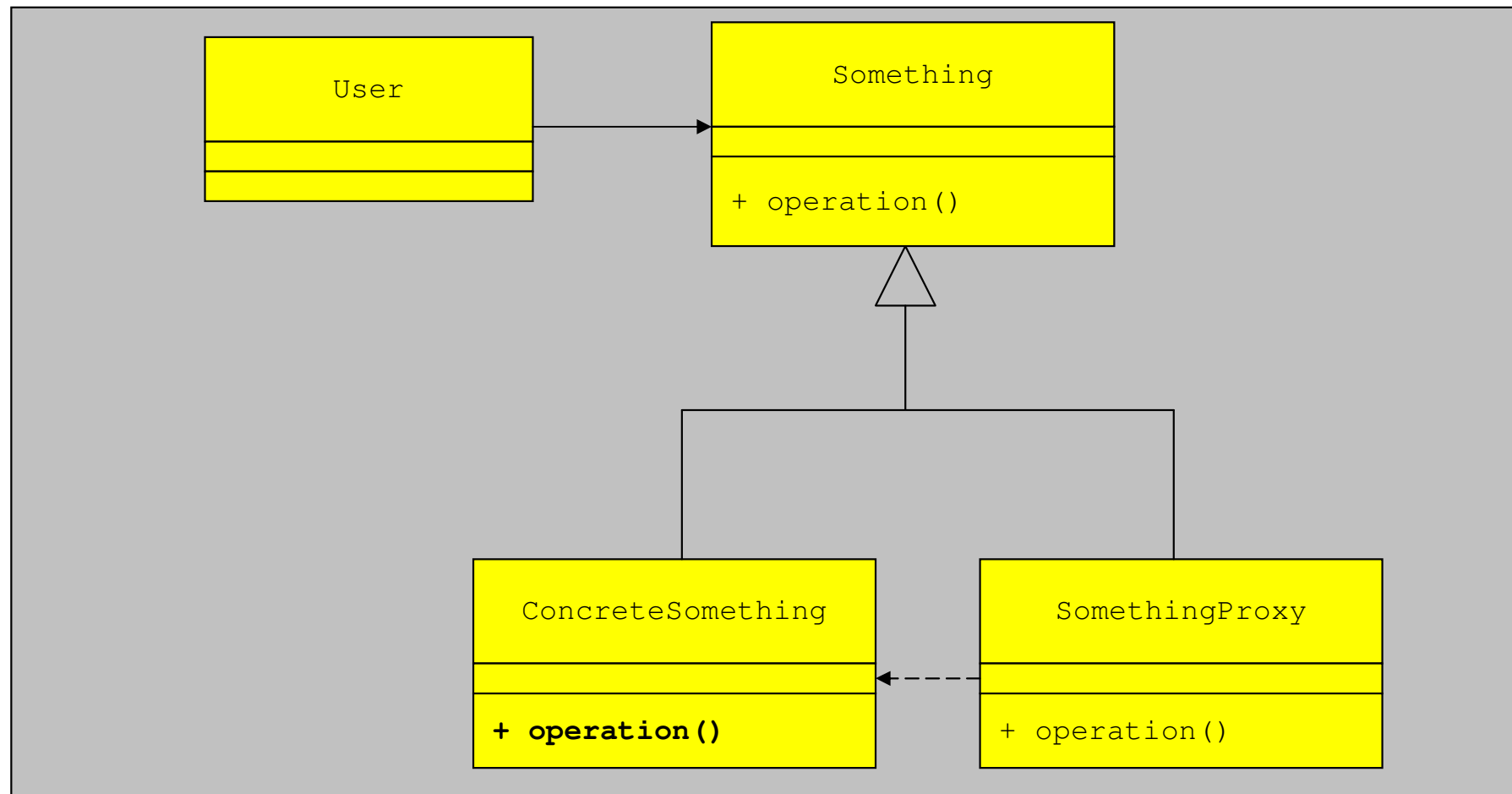
Façade (2/2)



Proxy (1/2)

- Un proxy è un delegato di un altro oggetto.
- I proxy hanno diversi usi
 - per controllare l'accesso alle risorse
 - per implementare politiche di sicurezza,
 - per implementare una gestione sofisticata di un oggetto
 - accesso sincronizzato,
 - persistenza,
 - per implementare oggetti remoti
 - il proxy e l'oggetto reale sono su due host diversi.

Proxy (2/2)



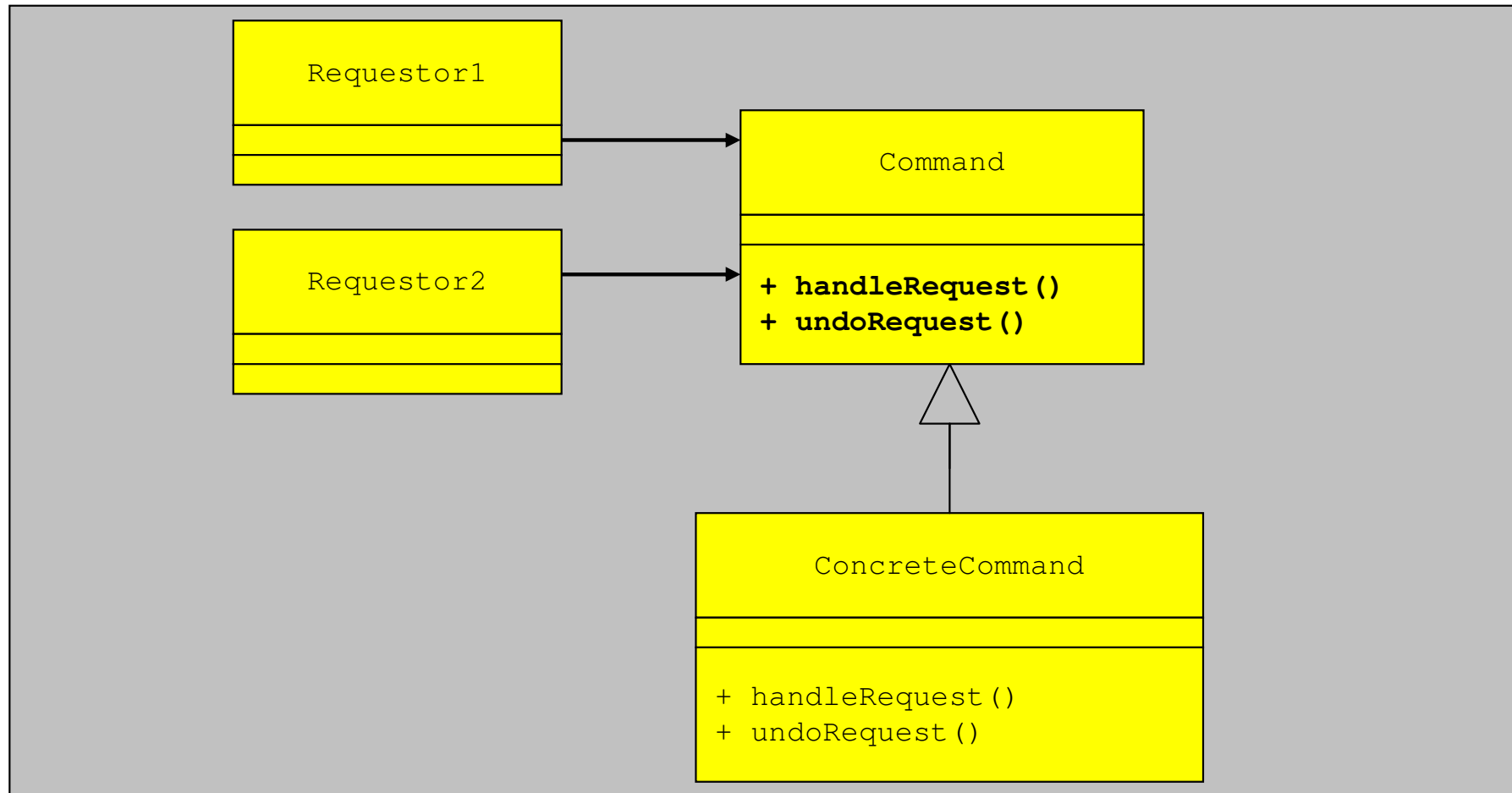
Pattern Comportamentali

- I pattern comportamentali definiscono collaborazioni tra oggetti
 - essenzialmente definiscono comunicazioni tra oggetti.
- **Command**, usato per incapsulare l'astrazione di richiesta.
- **Iterator**, usato per incapsulare la navigazione di un oggetto composto.

Command (1/2)

- Incapsula una richiesta in un oggetto
 - consente a molte sorgenti di generare richieste e le gestisce secondo una politica,
 - consente ad un oggetto di modificare il proprio comportamento in base alle richieste.
- Un command viene usato quando:
 - si vuole consentire di creare una richiesta in molti modi diversi,
 - si vuole consentire di fare **undo** delle richieste.

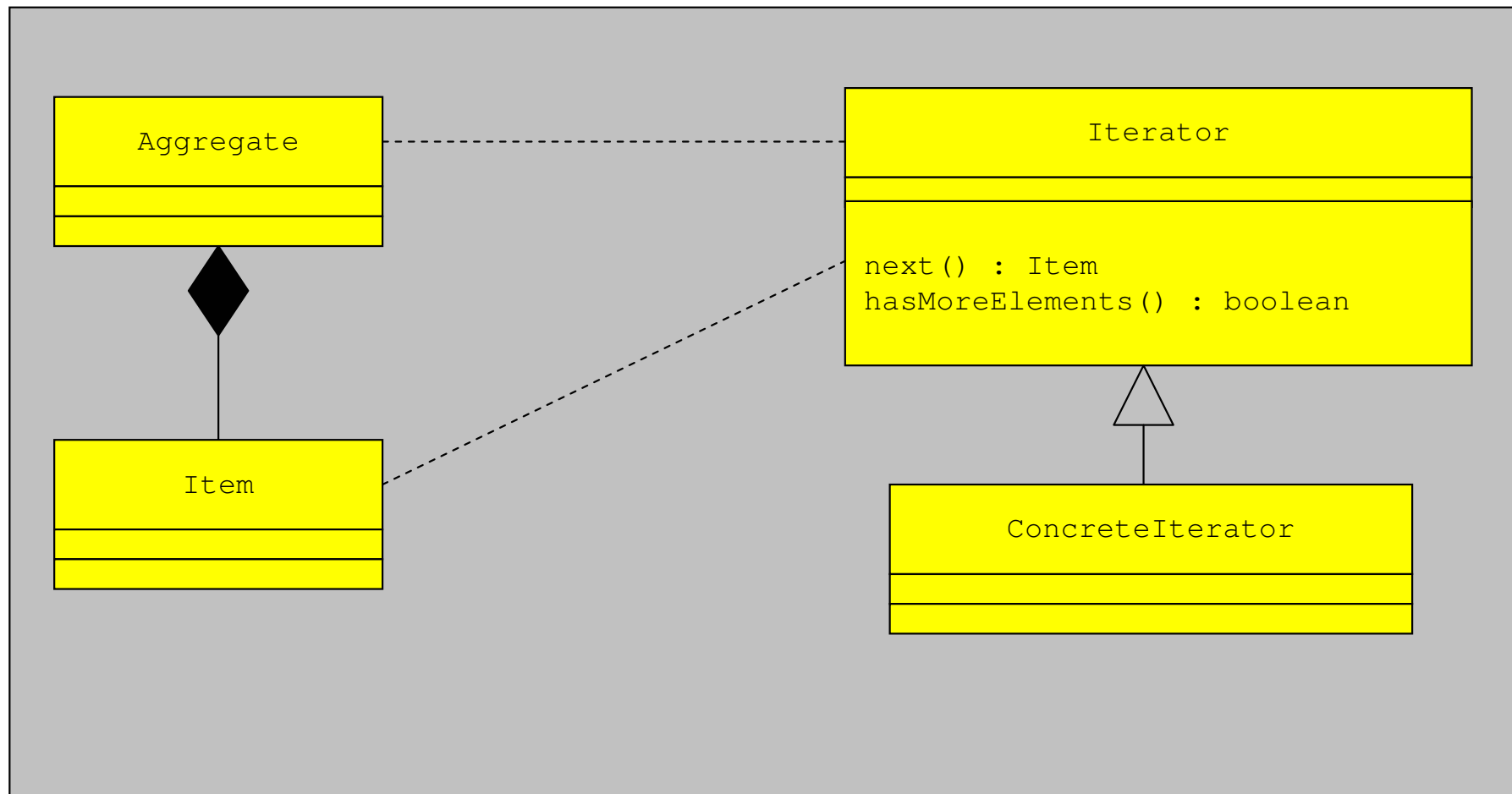
Command (2/2)



Iterator (1/2)

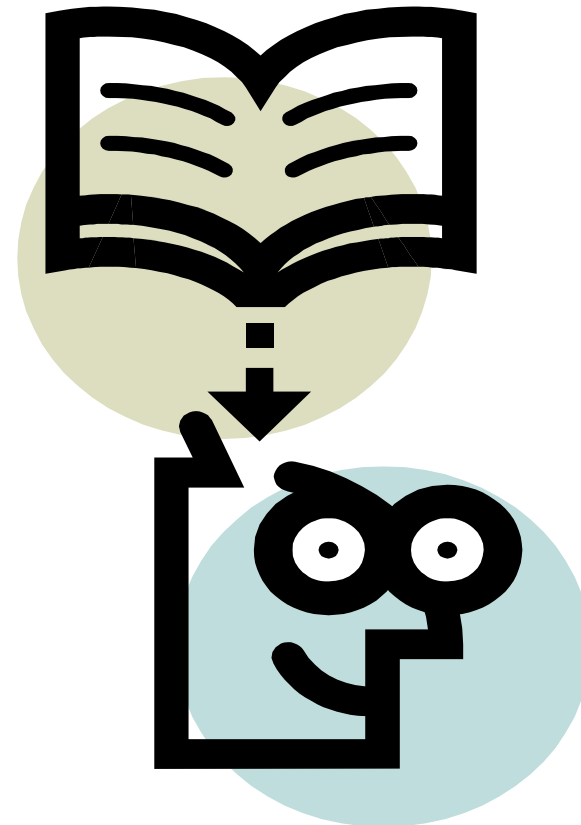
- Implementa un meccanismo di accesso ad oggetti aggregati in modo sequenziale
 - un iteratore naviga sequenzialmente un oggetto aggregato.
- Un iteratore viene utilizzato
 - per accedere al contenuto di un oggetto aggregato in modo indipendente da come i componenti sono memorizzati nell'oggetto aggregato,
 - quando si vuole offrire diverse politiche di attraversamento di un oggetto aggregato.

Iterator (2/2)



Come Scrivere un Programma in Java

- È possibile utilizzare un editor di testo e il compilatore in linea di comando.
- Ambienti integrati
 - TextPad
<http://www.textpad.com>
 - JCreator
<http://www.jcreator.com>
 - Netbeans
<http://www.netbeans.org>



Regole Generali

- Un programma deve essere **comprensibile** anche da chi non lo ha scritto.
- Un programma deve
 - essere **indentato** orizzontalmente e verticalmente,
 - contenere commenti significativi,
 - seguire le convenzioni sui nomi.

Indentazione Orizzontale (1/3)

- Esempio di blocco

```
if (a > b) {  
    . . . System.out.println("A" + a);  
    . . . System.out.println("B" + b);  
}
```

- Esempio di dichiarazione di variabile

```
NomeDellaClasse nomeDellOggetto;  
double . . . . . v1 = 0, v2 = 0,  
    . . . . . var1 = 0;
```

Indentazione Orizzontale (2/3)

- Esempio di scelta innestata

```
if (a > b) {  
    . . . if (a > b) . System.out.println("A") ;  
    . . . else . . . . . System.out.println("B") ;  
} else {  
    . . . System.out.println("C") ;  
    . . . System.out.println("D") ;  
}
```

Indentazione Orizzontale (3/3)

- Esempio di dichiarazione di classe e di metodo

```
public · class · NomeClasse · { ¶  
    · · · · int · metodo (int · a, · char · b) · { ¶  
        · · · · · System.out.println ("A" · + · a) ; ¶  
        · · · · · System.out.println ("B" · + · b) ; ¶  
        · · · · } ¶  
    } ¶
```

Indentazione Verticale

- Utilizzare una linea bianca per separare parti di codice logicamente separate

```
import java.net.*;
```

```
public class NomeClasse {
```

```
    int metodo(int a, char b) {
```

```
        int n;
```

```
        while (n < 10) {
```

```
            System.out.println("N" + n); n++;
```

```
        }
```

```
        System.out.println("Ciclo terminato");
```

```
    }
```

```
}
```

Convenzione sui Nomi

- I nomi devono essere significativi e il più possibile brevi.
- In generale vengono utilizzate le seguenti convenzioni
 - nomi di variabili e metodi iniziano con la minuscola,
 - nomi di classi iniziano con la maiuscola,
 - nei nomi composti, ogni parola inizia con la maiuscola,
 - le costanti sono tutte in maiuscolo e le parole si compongono con '_'.

Convenzione sui Commenti

- I commenti precedono il codice a cui si riferiscono.
- Commenti lunghi vanno spezzati su più righe.
- Per commentare un metodo lo si precede con una descrizione
 - di ciò che fa,
 - del significato dei parametri,
 - ...
- Non si utilizzano commenti lunghi nel corpo dei metodi.

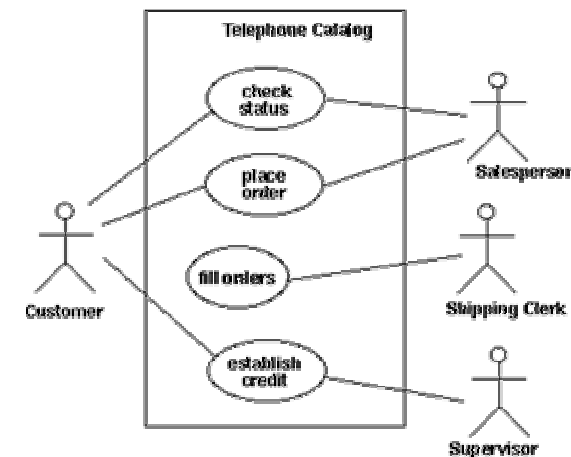
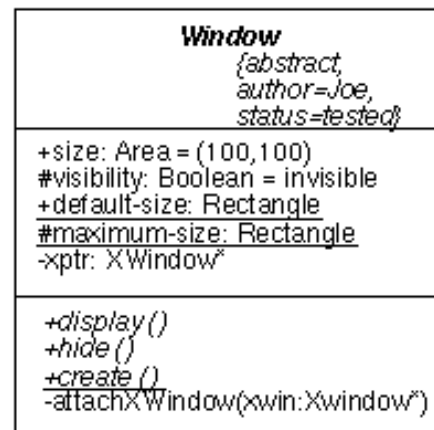
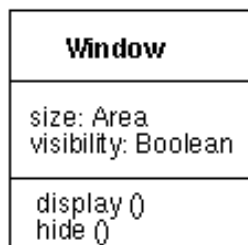
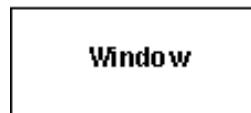
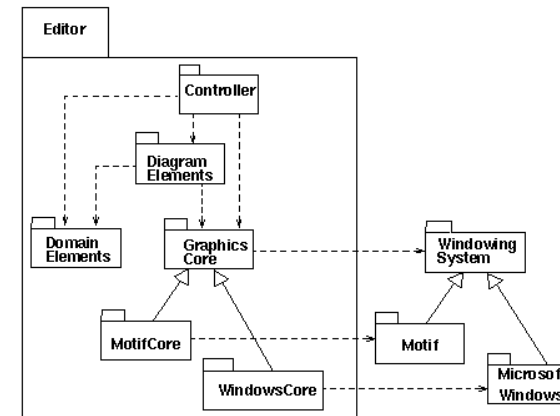
Compendio di UML (1/4)

- Compendio della notazione UML
 - use case diagram
 - class diagram
 - sequence diagram
 - collaboration diagram
 - state charts

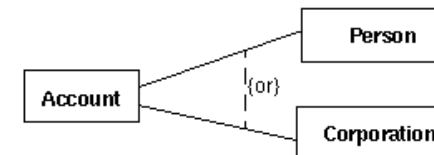
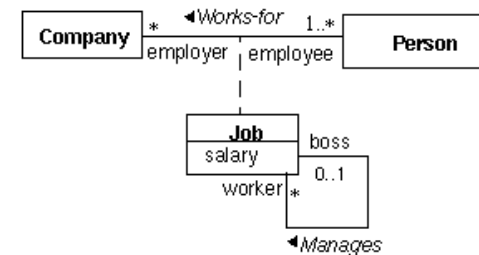
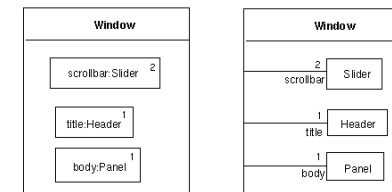
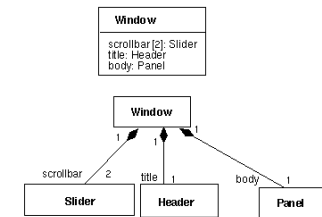
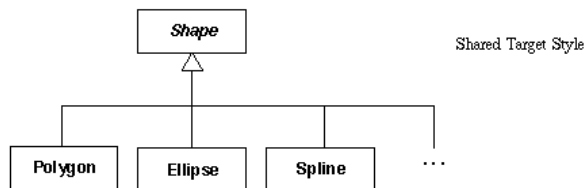
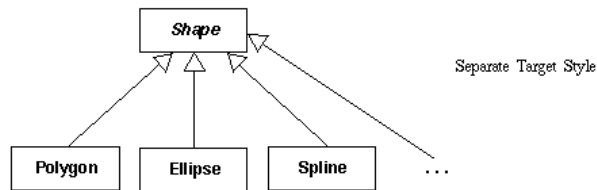
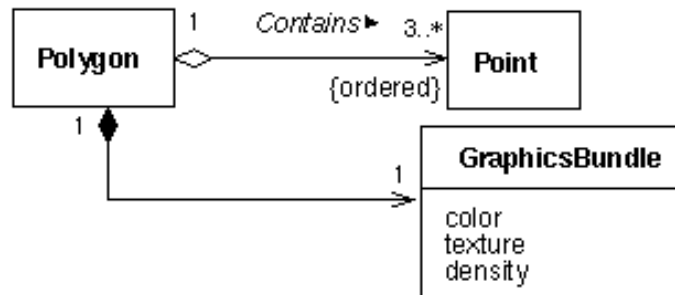


Compendio di UML (2/4)

This model was built by Alan Wright after meeting with the mission planning team.



Compendio di UML (3/4)



Compendio di UML (4/4)

