INGEGNERIA DEL SOFTWARE

Prof. Federico Bergenti

Indice dei Contenuti

1. Introduzione

def. Per **ingegneria del software** si intende quella disciplina che si occupa dei processi produttivi e delle metodologie di sviluppo finalizzate alla realizzazione di sistemi software. Nasce in seguito alla necessità di realizzare sistemi software di *dimensioni* e *complessità* tali da richiedere uno o più team di persone.

L'ingegneria del software è l'approccio sistematico allo sviluppo, all'operatività, alla manutenzione ed al ritiro del software.

Approccio sistematico:
- Sviluppo
- Operatività
- Manutenzione
- Ritiro

L'ingegneria del software nasce con la conferenza NATO del 1968.

Conferenza NATO 1968 →
- Software Crisis
- Software Reuse
- SW Engineering

→ **Anni '90** →
- OO Programming
- UML
- Design Pattern

2. Java e Programmazione Object Oriented

2.1 Introduzione

Java è una piattaforma sviluppata da Sun Microsystem, cercando di creare un linguaggio completamente orientato agli oggetti che migliorasse il C++.

Nasce dall'esigenza di svincolare il Software dall'Hardware. All'inizio questo ha avuto un costo prestazionale notevole, ora si sta assottigliando. Il pacchetto ci mette a disposizione:

- JRE Java Runtime Enviroment
  - Java Virtual Machine JVM, macchina astratta che esegue il codice bytecode
  - Java Class Library, classi e oggetti disponibili a corredo della JVM
- JDK Java Development Toolkit
  - JRE corredato dagli strumenti per lo sviluppatore
  - compilatore (javac)
  - documentazione delle API della Java Library
  - esempi

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. La programmazione orientata agli oggetti è nata, spinta dallo sviluppo di interfacce grafiche e cambia il paradigma della programmazione procedurale.

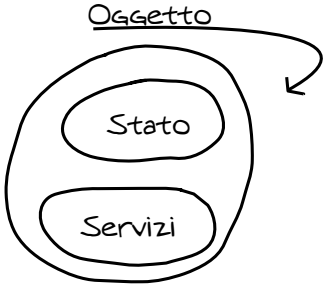
2.2 Nuovo Approccio

La programmazione OO propone un nuovo approccio:

• **Programmazione procedurale**	• **Programmazione orientata agli oggetti**
◦ analisi del problema per trovare un algoritmo risolutore. ◦ scomposizione dell'algoritmo risolutore in un insieme di algoritmi risolutori più piccoli.	◦ 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 algoritmi risolutori più piccoli.

2.3 Oggetti

- Sono astrazioni che descrivono:
  - oggetti *fisici*.
  - concetti *astratti*, che fanno parte del problema (o della soluzione).

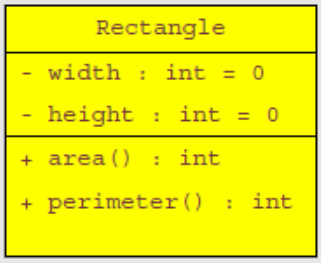


- Caratterizzati da
  - uno stato
  - un insieme di servizi che offrono agli altri oggetti.
- Esempio, un telefono cellulare
  - stato: carica della batteria, potenza del segnale, …
  - servizi: chiama un numero, rispondi alla chiamata, …

La complessità **diminuisce** tanto più lo stato dei singoli oggetti è **semplice**, l'insieme dei servizi offerti dai singoli oggetti è **specifico** (ma generale).

Esempio definizione di una classe con relativo **Class Diagram**:

```
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;
}
```



Una volta definito lo *stampo* possiamo creare un oggetto:

```
Rectangle obj = new Rectangle();
```

La variabile *obj* è detta object reference ed è una sorta di puntatore all'oggetto rettangolo.

2.4 Getter e Setter

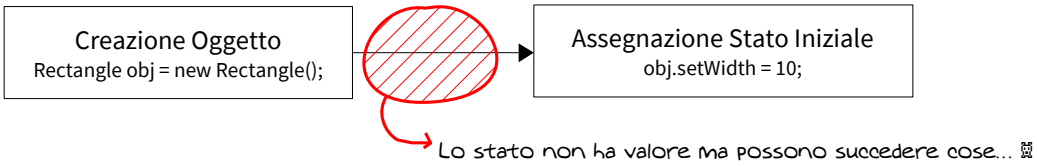
Metodi che consentono di manipolare direttamente lo stato di un oggetto. Sono metodi che vanno minimizzati, per quanto possibile.

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

Regola: I metodi getter e setter sono utili se non si limitano a leggere o scrivere variabili.

2.5 Inizializzazione di un Oggetto: Costruttori

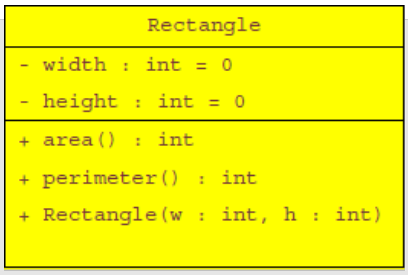
Se usassimo i metodi setter per inizializzare un oggetto, potremmo avere il seguente problema:



Instantiation is initialization: *La creazione di un oggetto coincide con la sua inizializzazione a uno stato iniziale.*

Per soddisfare questo principio è necessario prevedere che venga chiamata una procedura d'inizializzazione, il *costruttore*, al momento della creazione.

```
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;
}
```



L'istruzione:

```
new Rectangle(4, 5);
```

Ha l'effetto di:

- *creare* l'oggetto
- *chiamare il costruttore* dell'oggetto.

2.6 Metodo Main

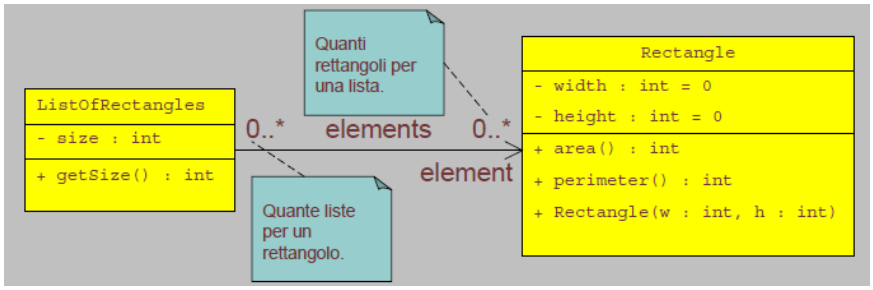
Per avviare un sistema è necessario fornire alla JVM una procedura da eseguire. Questa procedura (metodo) si chiama:

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

Regola: Di solito si definisce una classe con il solo scopo di contenere il metodo main.

2.7 Associazioni

Una relazione tra oggetti genera un'*associazione tra le classi*. 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:

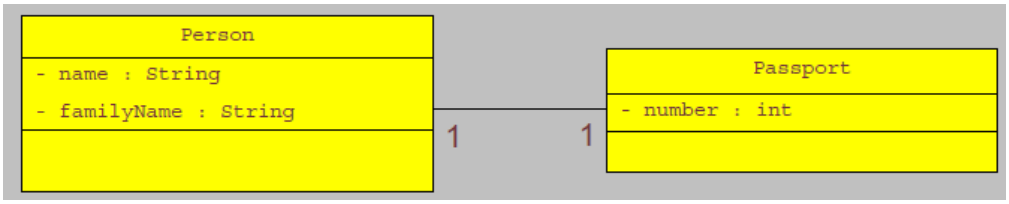
```
public class ListOfRectangles {
  public void addElement(Rectangle r) { elements[size++] = r; }
  public Rectangle getElement(int i) { return elements[i]; }
  private Rectangle[] elements;
}
```

E' necessario controllare il vincolo sulla cardinalità nei metodi addElement() e removeElement()

Nota: Solo i lati navigabili implementano l'associazione.

Associazione 1 a 1

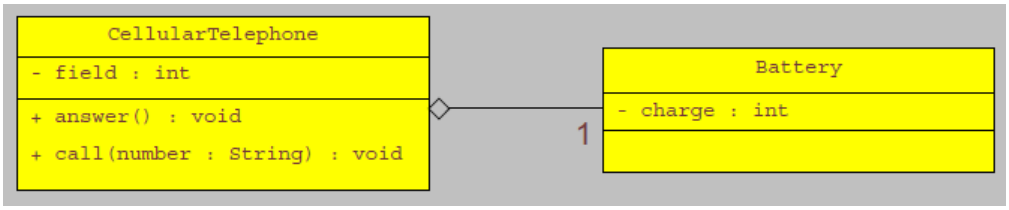
Un'associazione particolarmente importante è l'**associazione 1 a 1**.




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

Relazione di contenimento

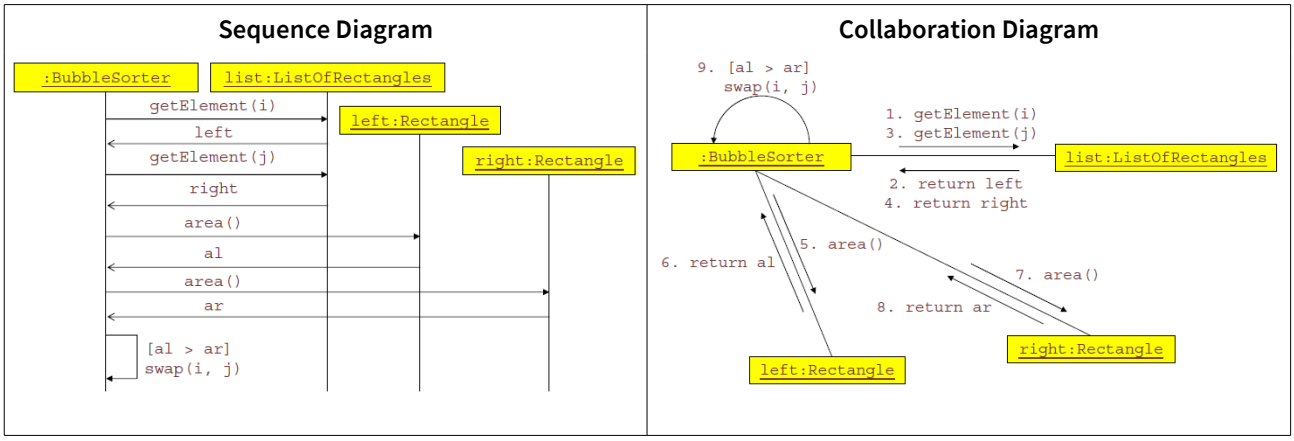
Se un'associazione esprime un *contenimento fisico*, o una *relazione part-of* o una *relazione has-a*, allora si parla di relazione di contenimento.



Nota: In Java **non** c'è nessuna vera differenza tra il contenimento ed un più generica associazione.

2.8 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. Sono utili in fase di *analisi* e di *progettazione ad alto livello*, come *documentazione*.



+ Vantaggio

- rappresentazione grafica utile per comprendere meglio interazioni complesse.

- Svantaggi

- non sufficientemente espressivi,
- potenzialmente ambigui,
- più ridondanti di uno pseudo-codice.

2.9 Ereditarietà

Tra le classi è possibile definire una relazione di sotto-classe (sotto-insieme), consente di comporre un insieme di classi per creare una nuova classe.

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
  - da utilizzare con **cautela**, solo per metodi utili a chi estende ma che non offrono servizi

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

Una classe derivata viene costruita partendo dalla classe base. L'ereditarità tra le classi Java è singola: solo una classe base per ogni classe derivata, così si evitano **DDD** (*Deadly Diamond of Death*).

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

Essendo l'eredità singola, tutte le classi sono derivate (implicitamente) da un ancestore comune: **`Object`**.

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

Eredità e costruttori

I costruttori non vengono ereditati, è possibile tuttavia accedere ai costruttori della classe base mediante `super`.

Nota: `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.

Polimorfismo

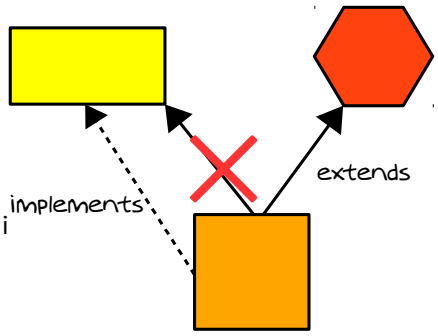
Un metodo della classe base può essere ridefinito nelle classi derivate (metodo **polimorfo**).

Classi Astratte

Se una classe contiene metodi che possono essere implementati solo dalle sue sotto-classi viene detta **astratta**.

Una classe astratta **non può essere istanziata**.

Interfacce



Un quadrato è contemporaneamente

- un rettangolo
- un poligono regolare

ma Java *non permette l'ereditarietà multipla* tra le classi.

def. 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

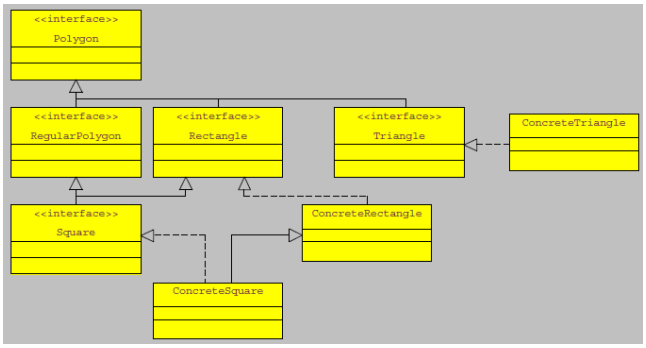
I corpi dei metodi, cioè come i servizi vengono implementati, **non** sono parte dell'interfaccia:

- l'interfaccia indica cosa un oggetto sa fare e non come lo fa

Nota: In Java, le interfacce vengono implementate da classi.

```
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;
}
```

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.

A Cosa Serve l'Ereditarità?

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** perché la prima bozza di un progetto è il modello che *analizza il dominio* del problema (o della soluzione).

Come Massimizzare il Riuso: Due modi

- **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, il **secondo** meccanismo di riuso è il più apprezzato.

2.10 Applicazione: Generalizzazione del Servizio di Ordinamento

Vogliamo creare un servizio di ordinamento generico. In una prima esercitazione abbiamo creato un servizio che ordinasse solo *poligoni*, in base *all'area*.

Obiettivo: Per massimizzarne la riusabilità bisogna agire su entrambe queste limitazioni definendo una *gerarchia di classi* con il solo *scopo* di **rendere l'algoritmo personalizzabile**.

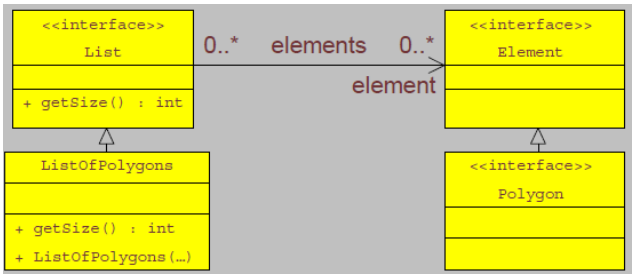
Il riuso non viene dalla possibilità di riutilizzare parti dell'algoritmo di ordinamento, ma dall'aver reso l'algoritmo **personalizzabile**.

Introduciamo le classi:

- **List:** interfaccia che modellizza 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.

Lista Astratta

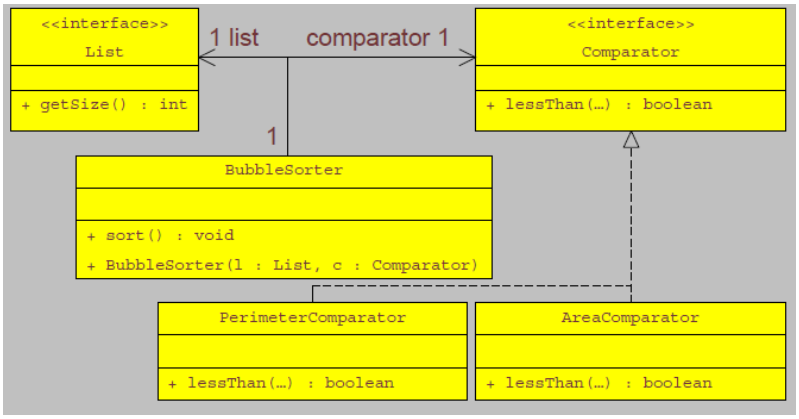
```
public interface List {
  public void     addElement(Element e);
  public void     setElement(int i, Element r);
  public Element  getElement(int i);

  public int getSize();
}
public interface Element {}
public interface Polygon extends Element { … }
```



Bubble Sorter Personalizzabile

```
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;
  }
```



Down-cast

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 a una classe base a un oggetto di una classe derivata median - te un **down-cast.**

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

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

```
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();
  }
}
```

3. Struttura di un progetto Java

Quando la complessità del progetto aumenta (aumenta il numero di file sorgenti e bytecode) conviene strutturare il progetto il moduli isolati e redistribuibili, detti *package*.

3.1 Package

Un **package** è un insieme di file (sorgente o bytecode) che sono logicamente visti come un unico *gruppo:*

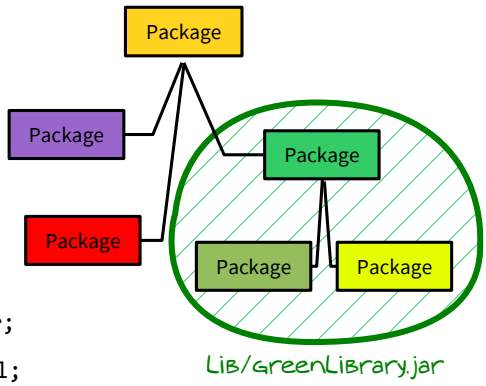
- è possibile strutturare i package come *alberi*
- un sotto-package isola un gruppo di *funzionalità* del suo package padre

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

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

- `src/` i sorgenti
- `classes/` i bytecode
- `images/` eventuali immagini
- `lib/` eventuali library esterne che si intende utilizzare



All'interno della directory `src` si mette una directory *per ogni package*:

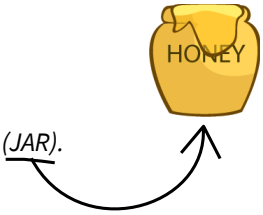
- `src/` editor i file iniziano con `package editor;`
- `src/editor/model` i file iniziano con `package editor.model;`
- `src/editor/view` i file iniziano con `package editor.view;`

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

3.2 Library

Una library è un insieme di package che implementano funzionalità comuni, come ad esempio:

- 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).*

Il file `.jar` è uno `ZIP` del contenuto della directory `classes/`.

Per **compilare** un progetto è necessario includere tutte le *directory dei sorgenti* e 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 …
```

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 e** linea di comando diventa:

```
java –classpath
classes;images;lib\<library1>.jar;lib\<library2>.jar;…
```

```
<package>.Main
```

4. 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 perché oggi tutte le applicazioni hanno una GUI

4.1 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

4.2 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).

4.3 Model/View/Controller

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

Esempio, nel caso di un **editor di poligoni:**

- il **model** è una lista di poligoni che l'utente ha introdotto nel suo disegno
- la **view** 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

4.4 Finestre con Swing

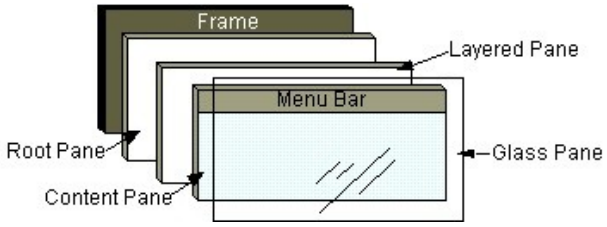
Una finestra è realizza creando un oggetto di classe `JFrame`. Per disegnare all'interno della finestra è possibile creare un pannello disegnabile. In Swing, i pannelli sono oggetti di classe `JPanel`:

- è possibile disegnare al loro interno
- sono contenitori di altri componenti

Ogni **top level container** (e `JInternalFrame`) di Swing ha quello che è chiamato `JRootPane`. Questo è responsabile di gestire effettivamente il layout globale della finestra.

Un JFrame consiste di **quattro piani**, normalmente si lavora con il piano detto *content pane*.

Quando aggiungi qualcosa a un `JFrame`, questa è automaticamente aggiunta al `ContentPane`, nelle vecchie versioni di Java era obbligatorio chiamare `getContentPane().add(...)`.



Per aggiungere un `JPanel` al `JFrame`:

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

4.5 Disegnare in un `JPanel`

`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)`:

```
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 Raster: `Graphics` utilizza un sistema dove ogni pixel (picture element) all'interno del JPanel è identificato da due numeri interi, questo sistema è detto *raster*.

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

4.6 Eventi

Nel caso delle interfacce grafiche, 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

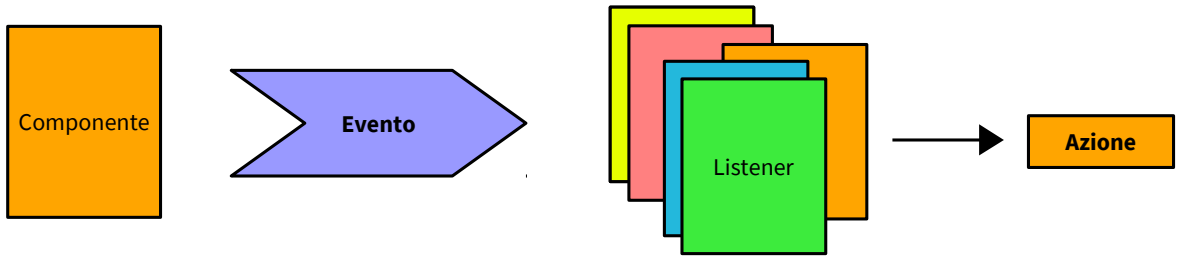
Gli eventi vengono generati (*fire*) da un componente della GUI. La Java library contiene varie classi che rappresentano i più comuni tipi di eventi.

Listener

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.



4.7 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).

4.8 Layout Manager

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

Ogni contenitore ha un layout manager di default e un nuovo layout manager può essere impostato mediante `setLayout(LayoutManager)`.

5. Gestione degli errori

5.1 Eccezioni

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, come ad esempio:

- divisione per zero
- tentativo di indirizzare un array al di fuori della sua dimensione massima

Gestione degli errori in C

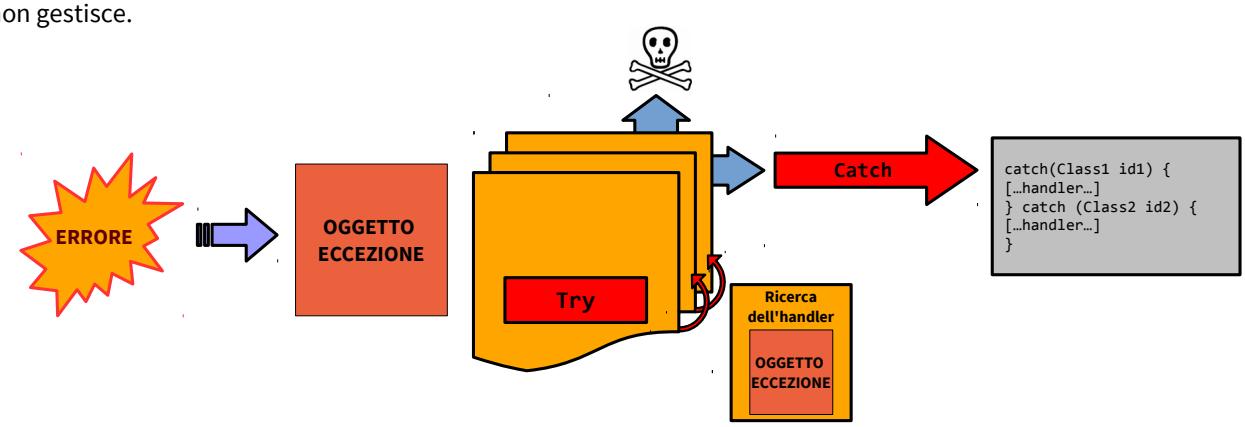
- non è parte del linguaggio e viene effettuata in **modo specifico** per ogni situazione
- tipicamente una funzione **ritorna un valore speciale** al chiamante per indicare che è accaduto qualche errore
- si ipotizza che il chiamante **controlli il valore di ritorno** e in caso di errore agisca di conseguenza

Problema! non c'è nessun meccanismo che forzi questo controllo, codice applicativo e codice di gestione dell'errore sono mescolati

Gestione degli Errori in Java

- 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.
- La JVM ripercorre all'indietro la sequenza delle chiamate finché 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.

Nota: Java richiede che **ogni metodo catturi** un'eccezione mediante un **catch**, oppure che indichi quali eccezioni non gestisce.



5.2 Generazione delle Eccezioni

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

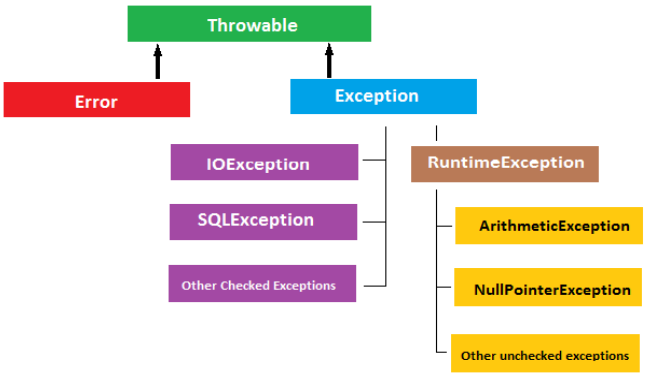
```
if(file.length() == 0)
  throw new FileEmptyException(file);
```

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

```
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;
  }
}
```

5.3 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.



5.4 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.**

Stream come lato di un canale di comunicazione

Uno stream può essere visto come un **lato di un canale di comunicazione mono-direzionale**, collega un output stream ad un corrispondente input stream.

Tastiera (cin) **>>** **Variabile**

Tutto quello che viene scritto nell'output stream, viene letto dall'input stream.

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.

Caratteristiche importanti degli stream:

- 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.
    - E' **fondamentale chiudere lo stream** `stream.close();` tipicamente nella `finally{}` del try-catch nel quale viene inserita l'operazione di lettura/scrittura dello stream.

5.5 Stream nella JCL (Java Class Library)

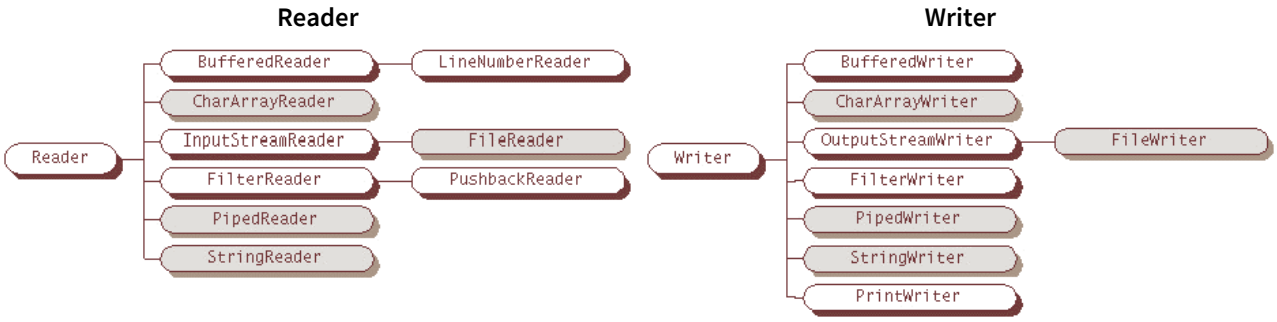
Il package `java.io` contiene una collezione di classi stream **divise in due gerarchie** sulla base del tipo di dato (byte o caratteri) che viene manipolato.

Character Stream

Estendono le classi `Reader` e `Writer`. Manipolano stream di caratteri Unicode a 16 bit.

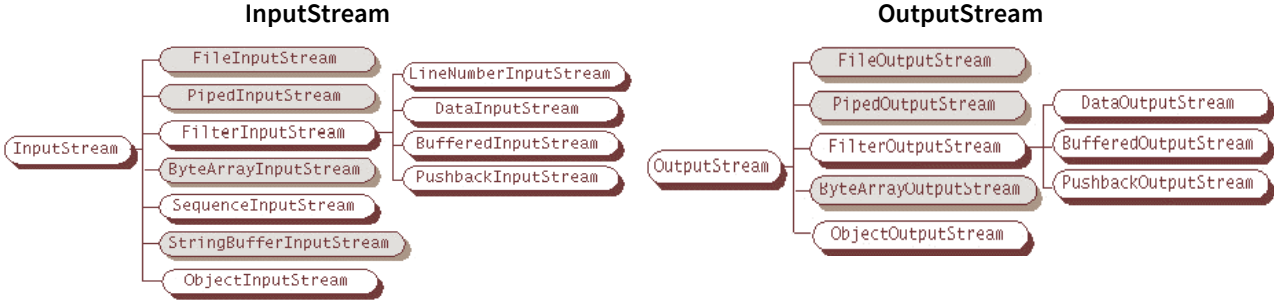
Forniscono le conversioni necessarie per manipolare i file **indipendentemente dalla piattaforma**:

- 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

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)`

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)`

Esempio di Codice

```
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]);
     }
  }
}
```

6. Collection Framework

6.1 Framework

def. Un *framework* è un insieme di **classi** che si appoggiano a delle **interfacce** che l'applicazione implementa per personalizzarne il comportamento (**algoritmi**).

Es.

- il **`BubbleSorter`** è un (piccolo) framework che sfrutta i servizi del `Comparator` e della `List`
- **Swing** è un framework

Nota: Non è l'applicazione che chiama il framework, è il framework che chiama l'applicazione.

6.2 Collection Framework

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

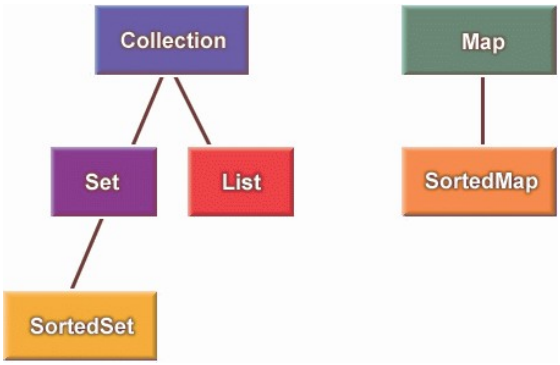
Contenuto nel package: `java.util`.

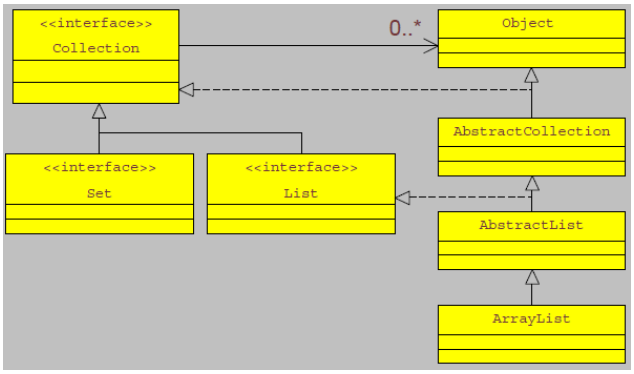
Collezione: Una *collezione* (*container*) è un oggetto che raggruppa un insieme di altri oggetti. Le collezioni sono usate per memorizzare e manipolare **gruppi di dati**. Le collezioni rappresentano tipicamente oggetti che sono raggruppati per contenimento:

- un elenco telefonico
- le parole in un vocabolario
- una lista di studenti

6.3 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`.





L'Interfaccia Set (Insieme Matematico)

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), 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

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)`

```
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;
}
```

L'interfaccia Map

I tipi dato associativi sono aggregati indirizzabili per **contenuto** anziché 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.

Nota: Da utilizzare con cautela: non possono sostituire le relazioni tra gli oggetti.

```
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());
  }
}
```

Il Metodo hashCode

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.

Nota: 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.

```
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;
}
```

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`

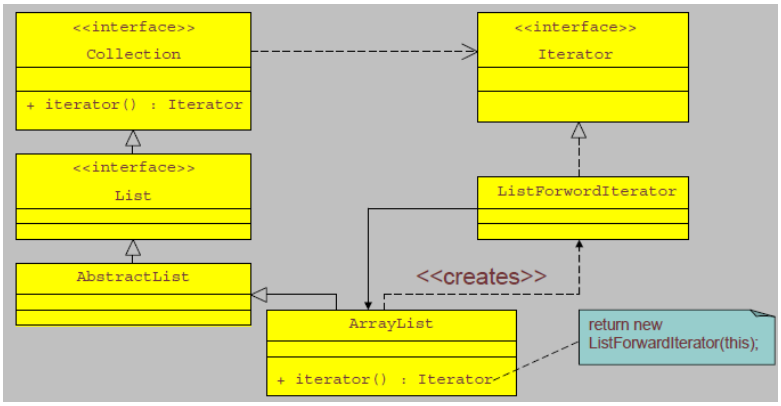
- `Set`, `TreeSet` e `HashSet`

6.4 Iteratori

Tutto il collection framework utilizza un unico modo per **accedere** agli singoli **elementi** dei tipi dato aggregati.

def. **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




```
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;
  }
}
```

6.5 Oggetti Valore

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()`

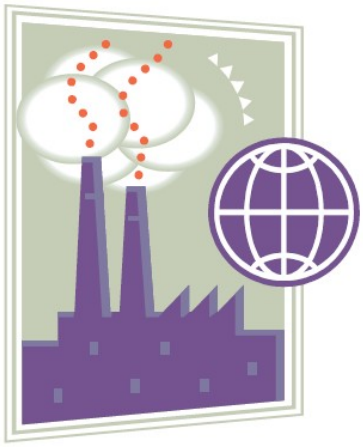
```
A obj2 = (A)obj1.clone();
```

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

Gli oggetti per cui sono implementate `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;
}
```

7. La Produzione di 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

L'ingegneria del software ha a che fare con:

- teorie
- metodi
- strumenti

per lo sviluppo **industriale** di software.

7.1 Ingegneria del Software e 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.

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**.

7.2 Processo di Sviluppo

def. Il **processo di sviluppo** è l'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
- **realizzazione** (o **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

7.3 Metodologia di Produzione

def. Con **Metodologia di Produzione** si intende la 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

Una metodologia di produzione **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

def. 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

7.4 Qualità del software

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.

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.
    - *Product quality is process quality*

Parametri di qualità esterni

- **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.

Parametri di qualità interni

- **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 manutenere 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 co-operare con altri sistemi, anche di altri produttori

7.5 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*)

7.6 Costo del Software

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.

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 (costi indiretti)

- altri costi generali della struttura
- capacità, motivazione e coordinamento del personale
- complessità del sistema da realizzare
- stabilità dei requisiti
- caratteristiche dell'ambiente di sviluppo

Mantenimento

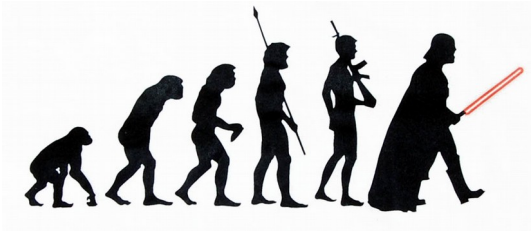
Il costo del software è principalmente nel **mantenimento** piuttosto che nello sviluppo.

7.7 Mantenimento del Software

Uno degli **obiettivi** principali dell'ingegneria del software è ottenere un **sviluppo cost-effective** prevenendo per quanto possibile i costi di manutenzione.

Il software **deve evolvere** perché:

- non sono stati colti correttamente i requisiti
- i requisiti non sono inizialmente noti
- cambiano le condizioni operative
- …



Importante! L'evoluzione è **ineliminabile** per gran parte delle tipologie di sistemi, anche se i requisiti iniziali sono corretti e completi.

def. **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%.

Esperienza sul Campo

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

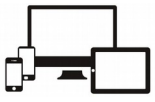
Alcuni dati:

- 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

7.8 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.

8. Responsabilità Etiche e Professionali

Etica

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

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 della proprie competenze

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**

8.1 Codice Etico ACM/IEEE

Association for Computer Machinery (**ACM**) ed **Institute of Electrical and Electronics Engineers** (**IEEE**) hanno emanato un **codice di etica professionale**.

Questo codice contiene otto principi riguardo:

- il comportamento
- Il processo decisionale

di **ingegneri** del software di **tutti i livelli**, da professionisti affermati agli studenti.

9. Modello del Processo di Sviluppo

def. 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

def. Un **modello di un processo** di sviluppo è una rappresentazione astratta del processo.

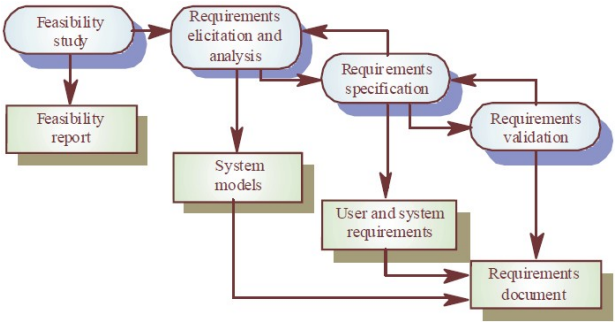
9.1 Specifica

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



9.2 Progettazione ed Implementazione

Processo di **conversione** della specifica di un sistema in un sistema funzionante.

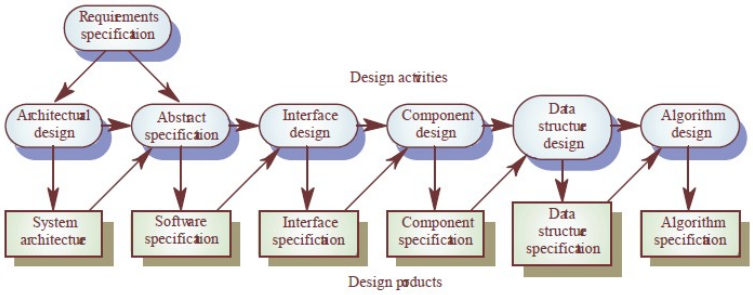
- Progettazione
  - progettazione della struttura (architettura) che a vari livelli di dettaglio realizza la specifica
- Implementazione
  - converte le architetture individuate nella progettazione in codice eseguibile

Progettazione ed implementazione sono fortemente correlate ed (alle volte) sono interlacciate.

Processo di Progettazione

Approccio sistematico allo sviluppo di un progetto software. Il progetto è tipicamente documentato mediante un insieme di modelli grafici:

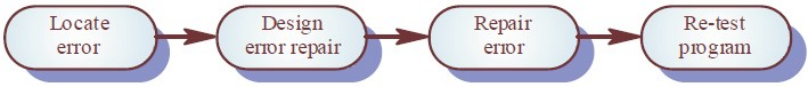
- **data-flow** diagram
- **entity-relation** diagram
- tutti i **modelli UML**
  - **class** diagram
  - **interaction** diagram



Implementazione

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.



9.3 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

Le operazioni di testing possono individuare la presenza di errori nel software ma non ne possono dimostrare la correttezza.
E. Dijkstra

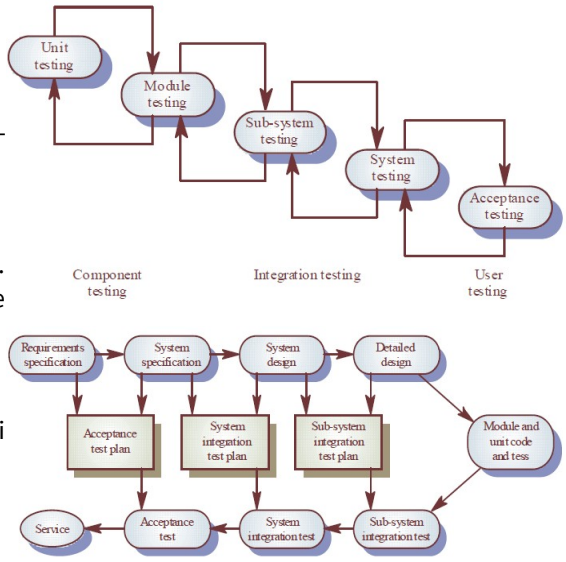
def. Verificare il comportamento del sistema in un insieme di casi sufficientemente ampio da rendere plausibile che il suo comportamento sia analogo anche nelle restanti situazioni.

Le operazioni di testing si suddividono in:

- Testing **in the small**, riguardano moduli singoli
- Testing **in the large**, riguardano il sistema nella sua globalità

Fasi di Testing

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



Testing in the small

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**)
  - **Problema:** 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**
    - ***Nota:*** può essere eseguito solo conoscendo la struttura interna della porzione di codice (**white-box testing**)
- 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
    - ***Nota:*** può essere eseguito solo conoscendo la struttura interna della porzione di codice (**white-box testing**)
- 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
    - ***Nota:*** produce un'analisi più approfondita rispetto al criterio di copertura delle decisioni, può essere eseguito solo conoscendo la struttura interna della porzione di codice (**white-box testing**)

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, come ad esempio:

- **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:
        - il cliente sia già stato inserito in archivio, vi sia corrispondenza tra la data di inserimento del cliente e quella della fattura
        - […]
    - **Test set:**
        - fattura con data odierna
        - fattura con data passata e per la quale esistono altre fatture
        - fattura con data passata e per la quale non esistono altre fatture
        - fattura il cui cliente non è stato inserito
        - […]

9.4 Ispezione del Software

def. Analisi del codice per capirne le caratteristiche e le funzionalità

- può essere effettuata sul codice oppure sullo pseudo-codice
- 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

def. Analisi informale eseguita da un team di persone che, 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

def. Analisi eseguita da un team di persone e organizzata, come nel caso del code walk-through, che mira però 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:

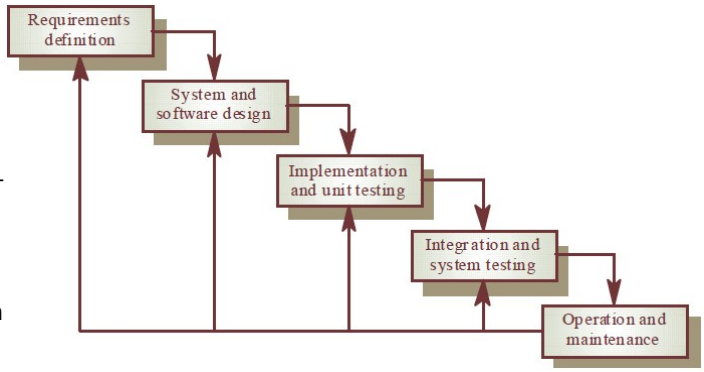
- uso di variabili non inizializzate
- loop infiniti
- letture di porzioni di memoria non allocata
- rilascio improprio della memoria

10. 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

10.1 Modello a Cascata

Fasi del Modello a Cascata



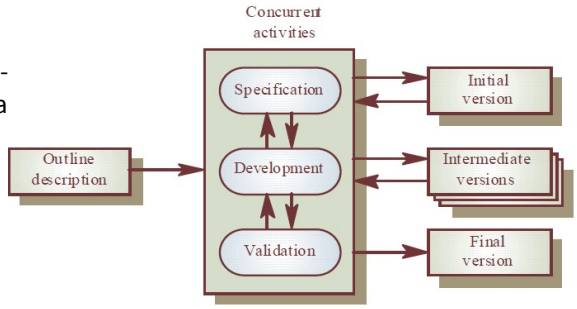
- **Analisi e specifica dei requisiti**
  - capire cosa i committenti vogliono
  - formalizzare il più possibile questi requisiti
- **Progettazione del sistema**
  - costruzione di un modello del sistema a vari livelli di dettaglio
- **Implementazione e unit testing**
  - realizzazione delle singole 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

10.2 Sviluppo Evolutivo

- **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

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**

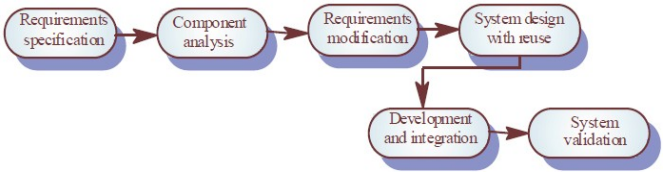
10.3 Sviluppo a Componenti

È basato sul **riuso sistematico** di componenti. Questi possono essere:

- sviluppati in-house,
- Commercial-Off-The-Shelf (COTS).

Il sistema è realizzato assemblando i **componenti disponibili**. Questo approccio è **immaturo**, ma è molto **promettente**.

Stadi del processo



- specifica dei requisiti
- analisi dei componenti disponibili
- modifica dei requisiti
- progettazione basata sull'integrazione dei componenti scelti
- sviluppo ed integrazione
- validazione

10.4 Iterazione dei Processi

I requisiti di un sistema evolvono sempre durante lo sviluppo di un progetto, c'è la necessità di **iterare** le fasi di un processo per rivedere quello che è già stato fatto a fronte dei nuovi requisiti.

Nota: L'iterazione può essere applicata a **qualsiasi modello** di processo.

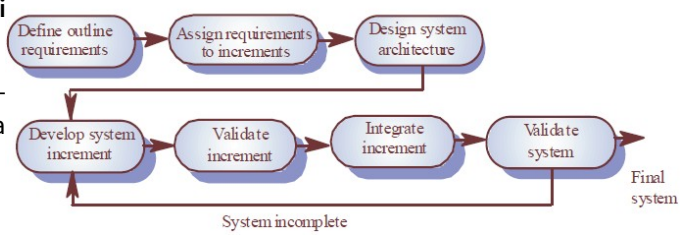
Due approcci allo sviluppo iterativo sono:

- **sviluppo incrementale**
- **sviluppo a spirale**

10.5 Sviluppo Incrementale

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



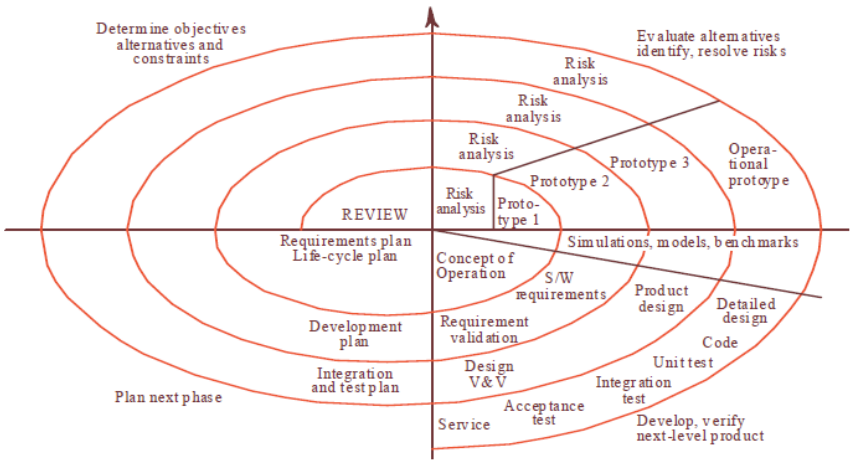
stessa

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**

10.6 Sviluppo a Spirale

- 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



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

11. Gestione (Management) dei Progetti Software

def. La gestione dei progetti software (Software Project Management), è l'insieme delle 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**.

11.1 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

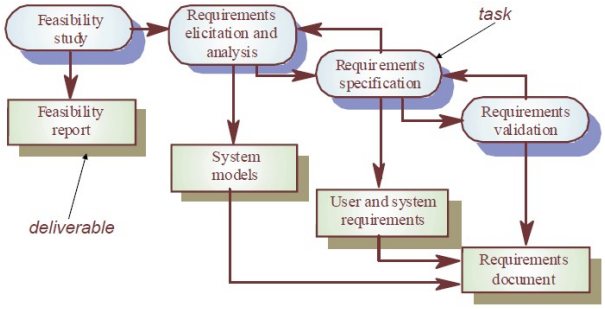
11.2 Organizzazione delle Attività

Le attività in un progetto dovrebbero essere organizzate per:

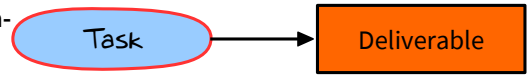
- produrre risultato **tangibile** e **misurabile**
- consentire di giudicare lo **stato d'avanzamento**

def. Le **milestone** sono le date di terminazione delle singole attività.

def. I **deliverable** sono i risultati del progetto rilasciati al committente.



Ogni azione visibile al committente (**task**) deve produrre un **deliverable**.



11.3 Attività di Scheduling di un Progetto

Organizzazione del progetto in attività e stima per ogni attività:

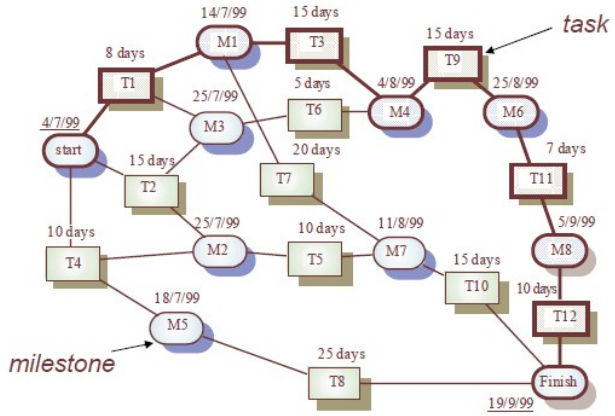
- delle **risorse** necessarie
- del **tempo** necessario

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

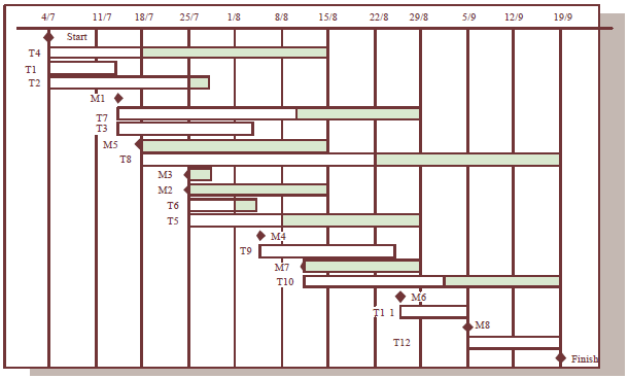
11.4 Diagrammi di Scheduling

Notazioni grafiche utili per descrivere lo scheduling di un progetto



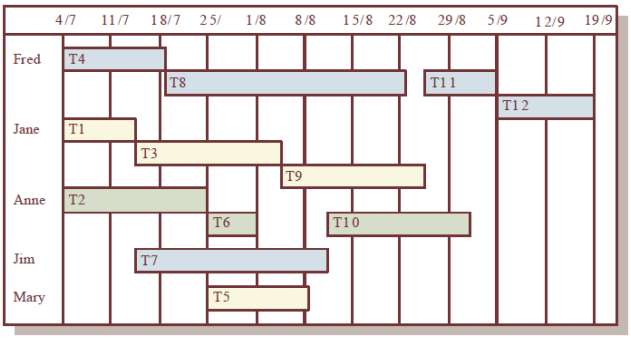
PERT (Program Evaluation Review Technique) chart

mostra le attività e le loro dipendenze



Gantt chart

mostra lo schedule in funzione del tempo



Person-month (PM) chart

mostra come viene impiegato il personale nel tempo

11.5 Approccio Orientato agli Oggetti

Nell'approccio orientato agli oggetti il 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

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

11.6 Analisi e Specifica dei Requisiti

def. 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

Attributi di qualità dell'analisi e specifica dei requisiti sono

- chiarezza
- non ambiguità
- consistenza

def. 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**

11.7 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)

11.8 Casi d'Uso

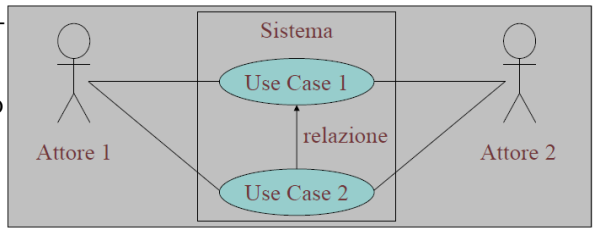
Strumenti per analizzare e specificare il comportamento atteso dal sistema.

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.

Sono usati anche per la verifica funzionale.

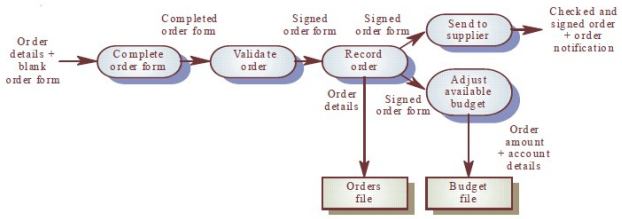


Use case: Use Case 1
Attori: Attore 1 (iniziatore), Attore 2.
Tipo: Primario, secondario, essenziale.
Descrizione: Descrizione informale dello scenario.

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.
…	…

11.9 Diagramma di Flusso dei Dati

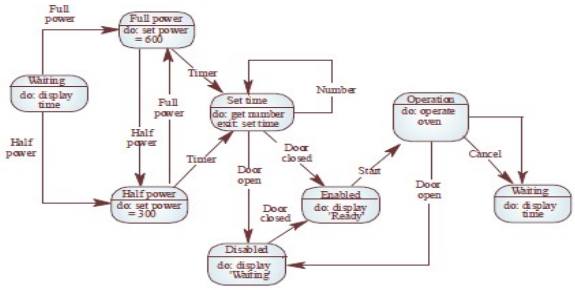
I **data-flow diagram** sono usati per modellizzare come i dati vengono processati nel sistema.



11.10 Macchine a Stati

Modello a stati del comportamento del sistema:

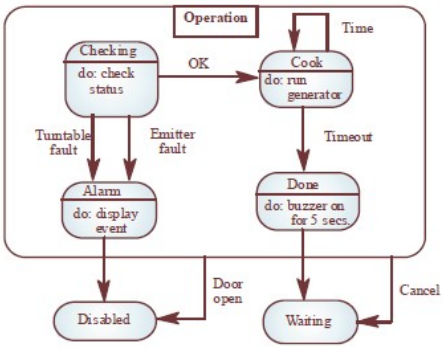
- stati in cui si può trovare il sistema
- eventi che causano le transizioni di stato



11.11 Statechart UML

Macchine a stati decomponibili in modo iterativo.

Devono essere completati da tabelle che descrivano gli stati e gli eventi.



11.12 Dizionario dei Dati

def. Il **Dizionario dei Dati** è la lista di tutti i termini (con relativo significato) utilizzato nei modelli. Lo scopo del dizionario dei dati è:

- 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.

11.13 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.

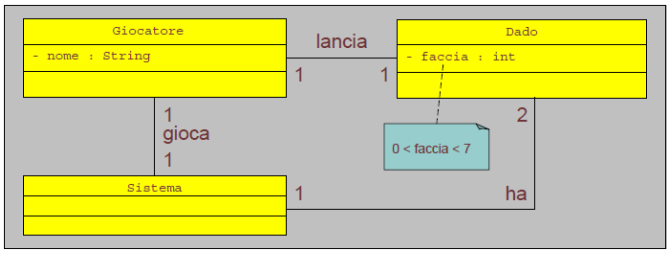
I class diagram ricordano i modelli **entità/relazioni, ma**:

- 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**

11.14 Esempio Gioco dei Dadi

Dai casi d'uso:

- due sostantivi: giocatore, dado;
- un verbo: lancia.



11.15 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.

11.16 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

11.17 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:             punta(numero, somma)
Responsabilità:   Puntare una somma su un numero.
Output:           nessuno.
Pre-condizioni:   somma minore della disponibilità. somma minore del massimo.
Post-condizioni:  nessuna.
Eccezioni:        somma maggiore della disponibilità. somma minore del massimo.
```

12. Progettazione

def. La p**rogettazione** è il processo che trasforma le specifiche del committente in un insieme di specifiche direttamente utilizzabili dai programmatori.

Il risultato è l'**architettura** del sistema, ovvero:

- **insieme dei moduli** che compongono il sistema,
- descrizione delle loro **funzioni**
- descrizione delle loro **relazioni**

12.1 Fasi della Progettazione

- **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

12.2 Modularizzazione

def. 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.

Relazioni tra i Moduli

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

Strategia di Approccio alla Modularizzazione

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

Nota: I progettisti non operano **mai** seguendo una sola strategia.

12.3 Architettura Client-Server

def. L'**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.

+ Vantaggi

- la **distribuzione** dei dati e del controllo è semplice
- è semplice **aggiungere** nuovi client
- è semplice **aggiungere** e/o sostituire i server

- Svantaggi

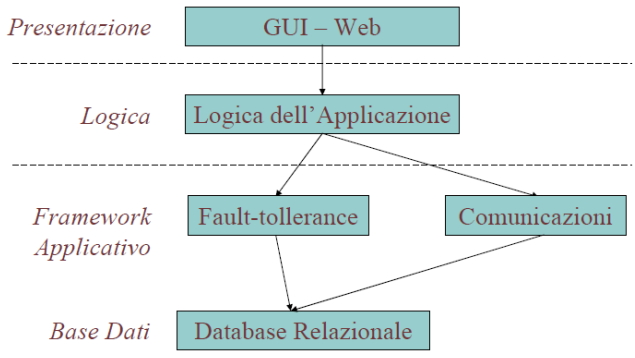
- il solo scambio dei messaggi può impegnare **molte risorse**
- gestione spesso **ridondante** dei singoli server.

12.4 Architettura a Tre Livelli

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).



12.5 Progettazione del Controllo

Consiste nel distribuire il controllo tra i sottosistemi.

- **Controllo centralizzato**
  - un sotto-sistema ha la responsabilità completa del controllo.
- **Controllo ad eventi**

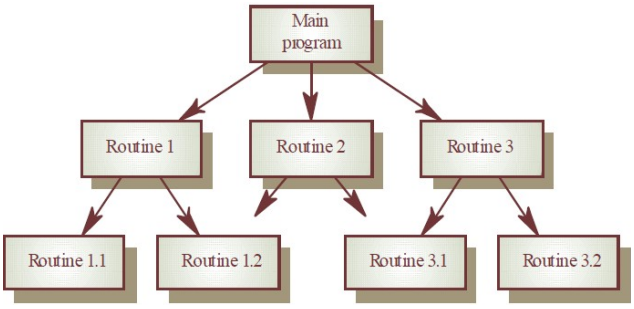
- ogni sotto-sistema può rispondere autonomamente agli eventi provenienti da altri sotto-sistemi dall'ambiente.

Controllo Centralizzato

Nell'architettura è presente un **sotto-sistema** con lo scopo di garantire il **controllo** degli altri sotto-sistemi.

Modello request/response

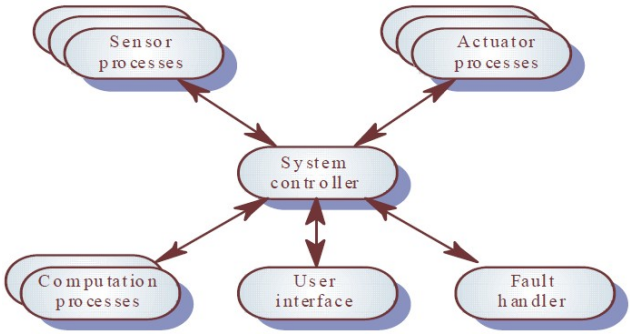
- modello a **sub-routine** in cui il controllo parte dall'alto e si dirama nella gerarchia di sub-routine
- applicabile praticamente solo a **sistemi sequenziali**



Modello a master/slave

un **sotto-sistema** controlla:

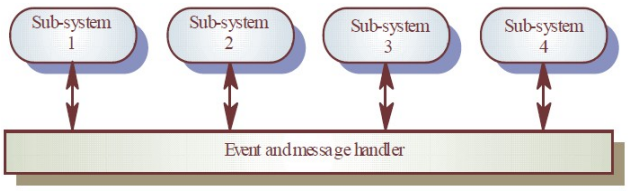
- l'attivazione
- lo spegnimento
- la coordinazione delle attività degli altri sotto-sistemi
- applicabile a **sistemi concorrenti**



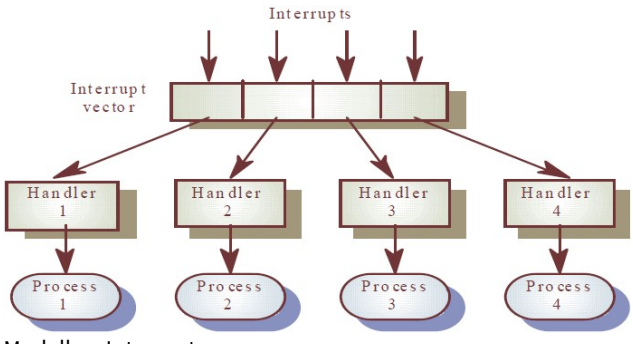
Controllo ad Eventi

Nel 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 a Interrupt

12.6 Progettazione in Dettaglio

La **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 inpu**t, 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

12.7 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

12.8 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

12.9 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**.

13. Design Pattern

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

I design pattern sono soluzioni utilizzate in progetti reali per risolvere **problemi comuni, s**ono **indipendenti dal linguaggio di programmazione.**

Vengono raggruppati in **pattern language,** raccolte divise secondo categorie di problemi affrontati.

- Variano da semplici soluzioni programmative a complesse architetture di sistema.

Se utilizzati durante la fase di sviluppo, vengono detti **idiomi programmativi**.

13.1 Storia

Il termine fu inizialmente introdotto in architettura in un celebre saggio di Christopher Alexander del 1977:

> *...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.*

In seguito, proprio l'opera di Alexander ispirò la nascita di un settore dell'**ingegneria del software** dedicato all'applicazione del concetto di design pattern alle architetture software, soprattutto object-oriented.

La nascita del "*movimento*" dei pattern in informatica si deve al celebre libro **Design Patterns: Elementi per il riuso di software ad oggetti** di Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (Gang of Four - GoF) (**1995**).

13.2 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

13.3 Formato GoF di un Design Pattern

- **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
- **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**

13.4 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.*
> *Jim Coplien*

13.5 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

13.6 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.

13.7 Pattern Strutturali

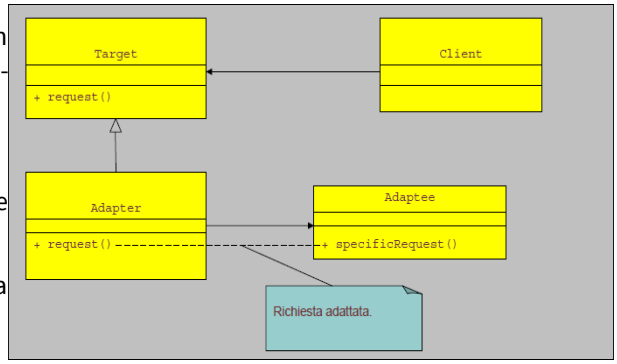
Gestiscono il modo in cui classi e oggetti vengono **composti** per formare architetture complesse.

- **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

def. L'**Adapter** 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.

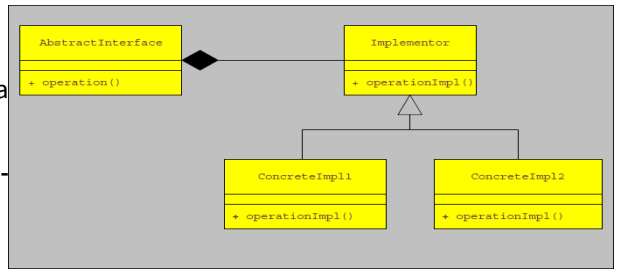


Un adapter **viene usato**:

- 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**

Bridge

def. Il **Bridge** disaccoppia un'astrazione dalla sua implementazione. La classe di implementazione e la classe degli utilizzatori possono variare indipendentemente.

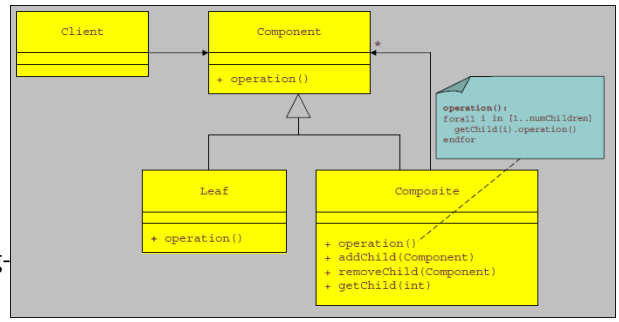


Un bridge **viene usato quando:**

- 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

Composite

def. Il **Composite** 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:

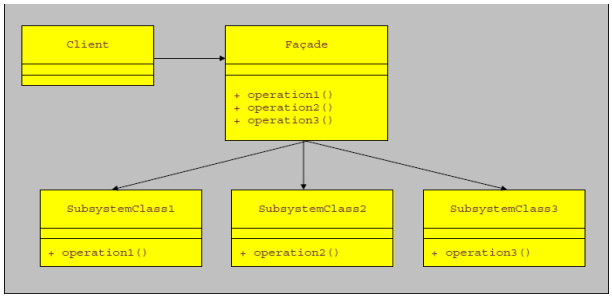
- si vuole rappresentare **gerarchie di oggetti**
- si vuole consentire di **ignorare le differenze** tra oggetti foglia e oggetti contenitore

Façade

def. La Façade 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 interdipendenze

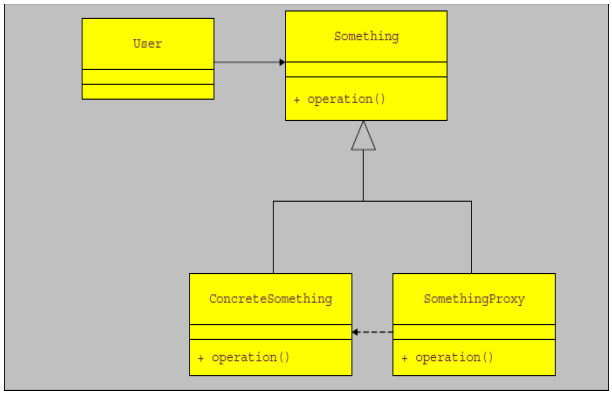


13.8 Proxy

def. 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



13.9 Pattern Comportamentali

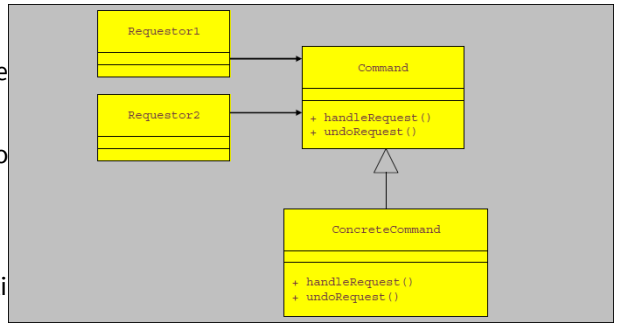
I pattern comportamentali definiscono collaborazioni tra oggetti, essenzialmente definiscono comunicazioni tra oggetti.

- **Command**, usato per incapsulare l'astrazione di richiesta.
- **Iterator**, usato per incapsulare la navigazione di un oggetto composto.

Command

def. Un **command** 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

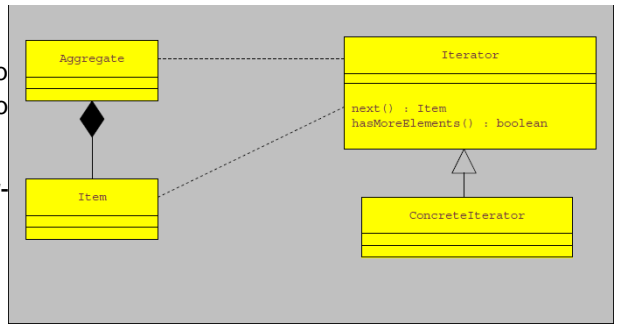
Iterator

def. Un Iteratore implementa un meccanismo di accesso ad oggetti aggregati in modo sequenziale. Un iteratore **na-**

viga 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



14. Appendice

14.1 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 '_'

14.2 Convenzione sui Commenti

In generale vengono utilizzate le seguenti **convenzioni**:

- 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.