

# Comandi base per interagire con la riga di comando di Windows

Il sistema operativo DOS non ha interfaccia grafica a finestre. Per navigare tra le cartelle, o per compiere un qualsiasi altro tipo di operazione sui file, bisogna digitare un comando. Segue una lista di comandi basilari che dovrebbero permettere al lettore di affrontare lo studio del testo in questione senza problemi.

Ogni comando descritto deve essere seguito dalla pressione del tasto “invio” (“enter”) sulla tastiera per avere effetto.

Comando	Spiegazione
<code>cd nomeCartella</code>	Spostamento in una cartella contenuta nella cartella di lavoro.
<code>cd ..</code>	Spostamento nella cartella che contiene la cartella di lavoro.
<code>dir</code>	Elenca il contenuto della cartella di lavoro con allineamento verticale
<code>dir /w</code>	Elenca il contenuto della cartella di lavoro con allineamento orizzontale.

<code>dir /p</code>	Elenca il contenuto della cartella di lavoro con un allineamento verticale. Se i file da elencare eccedono la disponibilità visiva della finestra, vengono visualizzati inizialmente solo i file che rientrano nella finestra stessa. Alla pressione di un qualsiasi ulteriore tasto, il sistema visualizzerà la schermata successiva.
<code>Control c</code>	(pressione del tasto “ctrl” contemporaneamente al tasto “c”) Interrompe il processo in corso (utile nel caso di processo con ciclo di vita infinito).

---

## **Preparazione dell'ambiente operativo su sistemi operativi Microsoft Windows: installazione del Java Development Kit**

- 1.** Scaricare il JDK Standard Edition da <http://java.sun.com>, possibilmente la versione più recente. Il nome del file varia da un rilascio all'altro, ma non di molto. Quello della versione 1.6.0 beta è "jdk-6-beta-windows-i586.exe", ed il file è ampio circa 51 Mb.
- 2.** Scaricare anche la documentazione (si tratta di un file .zip). Anche qui il nome del file varia da un rilascio all'altro; per la versione 1.6.0 beta è "jdk-6-beta-doc.zip", ed il file è ampio circa 50 Mb. Una volta ottenuto il file di installazione, eseguirlo e rispondere sempre positivamente a tutte le domande poste (accetti la licenza? Deve installare qui il JDK?...)
- 3.** Impostare la variabile d'ambiente PATH, facendola puntare alla cartella "bin" del JDK:
  - a) per i sistemi Windows 2000/XP/NT, eseguire i seguenti passi: Tasto destro su "Risorse del Computer", fare clic su "Proprietà". Selezionare il tab "Avanzate" e cliccare su "Variabili d'ambiente" (su Windows NT selezionare solo il tab "Variabili d'ambiente"). Tra le

“Variabili di sistema” (o se preferite tra le “Variabili utente”), selezionare la variabile PATH e fare clic su “Modifica”. Spostarsi nella casella “Valore variabile” e portarsi con il cursore alla fine della riga. Se non c’è già, aggiungere un “;”. Infine aggiungere il percorso alla cartella “bin” del JDK, che dovrebbe essere simile a :

C:\programmi\java\jdk1.6.0\bin

Fare clic su “OK” e l’installazione è terminata.

- b) Per i sistemi Windows 95/98, eseguire i seguenti passi: Fare clic su “Start” e poi su “Esegui”. Inserire il comando “sysedit” e poi premere Invio. Selezionare la finestra “Autoexec.bat”. Cercare l’istruzione “PATH = ...” (se non c’è, aggiungerla) e portarsi con il cursore alla fine dell’istruzione. Se non c’è già, aggiungere un “;”. Aggiungere il percorso alla cartella bin del JDK, che dovrebbe essere simile a:

C:\programmi\java\jdk1.6.0\bin

Fare clic su “OK” e riavviare la macchina: l’installazione è terminata.

- c) Per i sistemi Windows ME, eseguire i seguenti passi:  
Dal menu Start, scegliere “Programmi”, “Accessori”, “Strumenti di sistema” e “Informazioni di sistema”. Scegliere il menu “Strumenti” dalla finestra che viene presentata e fare clic su “Utilità di configurazione del sistema”. Fare clic sul tab “Ambiente”, selezionare PATH e premere “Modifica”. Portarsi con il cursore alla fine della riga. Se non c’è già, aggiungere un “;”. Aggiungere il percorso alla cartella “bin” del JDK, che dovrebbe essere simile a :

C:\programmi\java\jdk1.6.0\bin

Fare clic su “OK” e riavviare la macchina: l’installazione è terminata.

# Documentazione di EJE (Everyone's Java Editor)

## C.1 Requisiti di sistema

Per poter essere eseguito, EJE ha bisogno di alcuni requisiti di sistema. Per quanto riguarda l'hardware, la scelta minima deve essere la seguente:

Memoria RAM, minimo 16 Mb (raccomandati 32 Mb)

Spazio su disco fisso: 542 kb circa

Per quanto riguarda il software necessario ad eseguire EJE:

Piattaforma Java: Java Platform Standard Edition, v1.4.x (j2sdk1.4.x) o superiore

Sistema operativo: Microsoft Windows 9x/NT/ME/2000/XP (TM), Fedora Core 5, 4, 3, 2 & Red Hat Linux 9 (TM), Sun Solaris 2.8 (TM). Non testato su altri sistemi operativi...

## C.2 Installazione ed esecuzione di EJE

EJE è un programma multiplatforma, ma richiede due differenti script per essere eseguito sui sistemi operativi Windows o Linux. Sono quindi state create due distribuzioni, ma queste si differenziano solo per gli script di lancio...

### **C.2.1 Per utenti Windows (Windows 9x/NT/ME/2000/XP)**

Una volta scaricato sul vostro computer il file `eje.zip`:

Scompattare tramite un utility di zip come WinRar o WinZip il file `eje.zip`

Lanciare il file `eje.bat` (`eje_win9x.bat` per windows 95/98) con un doppio clic.

### **C.2.2 Per utenti di sistemi operativi Unix-like (Linux, Solaris...)**

Una volta scaricato sul computer il file `eje.tar.gz` (o `eje.tar.bz2`):

Dezippare tramite `gzip` (o `bzip2`) il file `eje.tar.gz` (o `eje.tar.bz2`). Verrà creato il file `eje.tar`.

Scompattare il file `eje.tar`. Verrà creata la directory `EJE`.

Cambiare i permessi del file `eje.sh` con il comando `chmod`. Da riga di comando digitare:

```
chmod a+x eje.sh
```

Lanciare il file `eje.sh` nella directory `EJE`.

### **C.2.3 Per utenti che hanno problemi con questi script (Windows 9x/NT/ME/2000/XP & Linux, Solaris)**

Da riga di comando (prompt DOS o shell Unix) digitare il seguente comando:

```
java -classpath . com.cdsc.eje.gui.EJE
```

**Se il sistema operativo non riconosce il comando, allora non avete installato il Java Development Kit (1.4 o superiore), oppure non avete settato la variabile d'ambiente `PATH` alla directory `bin` del JDK (vedi “installation notes” del JDK).**

## **C.3 Manuale d'uso**

`EJE` è un semplice e leggero editor per programmare in Java. Esistono tanti altri strumenti per scrivere codice Java, ma spesso si tratta di pesanti IDE che hanno bisogno a loro volta di un periodo di apprendimento piuttosto lungo per essere sfruttati con profitto. Inoltre tali strumenti hanno bisogno di ampie risorse di sistema, che potrebbero essere assenti o non necessarie per lo scopo dello sviluppatore. `EJE` si propone come editor Java non per sostituire gli strumenti di cui sopra, bensì

per scrivere codice in maniera veloce e personalizzata. E' stato creato pensando appositamente a chi si avvicina al linguaggio, contemporaneamente alla realizzazione di questo testo.

Le principali caratteristiche di EJE sono le seguenti:

- 1.** Possibilità di compilare ed eseguire file (anche con argomenti) direttamente dall'editor (anche appartenenti a package).
- 2.** Supporto di file multipli tramite il tasto TAB.
- 3.** Colorazione delle parole significative per la sintassi Java.
- 4.** Veloce esplorazione del filesystem tramite un'alberazione delle directory, e possibilità di organizzare ad albero cartelle di lavoro.
- 5.** Navigabilità completa di tutte le funzionalità tramite tastiera.
- 6.** Possibilità di annullare e riconfermare l'ultima azione per un numero infinito di volte.
- 7.** Utilità di ricerca e sostituzione di espressioni nel testo.
- 8.** Inserimenti dinamici di template di codice (è anche possibile selezionare testo per circondarlo con template di codice) e di attributi incapsulati (proprietà JavaBean).
- 9.** Personalizzazione dello stile di visualizzazione.
- 10.** Possibilità di commentare testo selezionato.
- 11.** Possibilità di impostare messaggi da visualizzare dopo uno specificato periodo di tempo.
- 12.** Popup di introspezione di classi automatico, per visualizzare i membri da utilizzare dopo aver definito un oggetto
- 13.** Possibilità di aprire la documentazione della libreria standard del JDK in un browser Java.
- 14.** Possibilità di generare documentazione automaticamente dei propri sorgenti mediante l'utilità javadoc.
- 15.** Indentazione automatica del codice in stile C o Java.
- 16.** Navigazione veloce tra file aperti.

**17.** E' possibile impostare molte opzioni: tipo, stile e dimensione del font, abilitazione e disabilitazione del popup di introspezione, stile di indentazione, compilazione in base alla versione di Java di destinazione, abilitazione/disabilitazione asserzioni, lingua, stile del look and feel

**18.** E' possibile stampare i file sorgente.

**19.** Supporto di Java versione 6!

L'interfaccia che EJE mette a disposizione dello sviluppatore è molto semplice ed intuitiva. La figura 1 mostra EJE in azione (EJE si mostrerà nella versione inglese se lanciato su un sistema operativo in lingua non italiana).



**Figura C.1 - "Eje in azione".**













Il pannello 1 mostra l'alberatura delle cartelle disponibili nel file system. Il contenuto delle cartelle non è visibile a meno che non si tratti di file sorgenti Java. E' possibile aprire nel pannello tali file facendo clic su essi.















Il pannello 2 mostrerà il contenuto dei file aperti.











Il pannello 3 invece mostrerà i messaggi relativi ai processi mandati in esecuzione dall'utente, come la compilazione e l'esecuzione dei file Java.















## C.4 Tabella descrittiva dei principali comandi di EJE:

Comando	Icona	Dove si trova	Scorciatoia	Sinossi
Nuovo		Menu File, Barra degli strumenti	CTRL-N	Crea un nuovo file
Apri		Menu File, Barra degli strumenti	CTRL-O	Apri un file presente nel filesystem
File recenti...		Menu File		Permette di aprire un file aperto recentemente
Salva		Menu File, Barra degli strumenti, Popup su area testo	CTRL-S	Salva il file corrente
Salva Tutto		Menu File, Barra degli strumenti	CTRL-SHIFT-S	Salva tutti i file aperti
Salva con nome		Menu File		Salva un file con un nome diverso dall'originale
Stampa...		Menu File, Barra degli strumenti	CTRL-P	Stampa il file corrente
Options		Menu File	F12	Apri la finestra delle opzioni
Chiudi File		Menu File, Popup su area testo	CTRL-SHIFT-Q	Chiude il file corrente
Esci		Menu File	CTRL-Q	Termina EJE
Annulla		Menu Modifica, Barra degli strumenti	CTRL-Z	Annulla l'ultima azione
Ripeti		Menu Modifica, Barra degli strumenti	CTRL-Y	Ripeti ultima azione
Taglia		Menu Modifica, Barra degli strumenti, Popup su area testo	CTRL-X	Sposta selezione negli appunti

Copia		Menu Modifica, Barra degli strumenti, Popup su area testo	CTRL-C	Copia selezione negli appunti
Incolla		Menu Modifica, Barra degli strumenti, Popup su area testo	CTRL-V	Incolla appunti
Cancella		Menu Modifica, Popup su area testo		Cancella selezione
Seleziona Tutto		Menu Modifica, Popup su area testo	CTRL-A	Seleziona tutto il testo
Rendi Maiuscolo		Menu Modifica, Popup su area testo	CTRL-U	Rende maiuscolo testo selezione
Rendi Minuscolo		Menu Modifica, Popup su area testo	CTRL-L	Rende minuscolo testo selezione
Trova		Menu Cerca, Barra degli strumenti	CTRL-F	Cerca espressioni nel testo
Trova Successivo		Menu Cerca	F3	Cerca la successiva espressione nel testo
Sostituisci		Menu Cerca	CTRL-H	Cerca e sostituisce espressioni nel testo
Vai alla riga		Menu Cerca	CTRL-G	Sposta il cursore alla riga
Template di classe		Menu Inserisci	CTRL-0	Inserisce (o circonda il testo selezionato con) un template di classe
Metodo main		Menu Inserisci	CTRL-1	Inserisce (o circonda il testo selezionato con) un template di metodo main
If		Menu Inserisci	CTRL-2	Inserisce (o circonda il testo selezionato con) un template di costruito if
Switch		Menu Inserisci	CTRL-3	Inserisce (o circonda il testo selezionato con) un template di costruito switch

For		Menu Inserisci	CTRL-4	Inserisce (o circonda il testo selezionato con) un template di costruito for
While		Menu Inserisci	CTRL-5	Inserisce (o circonda il testo selezionato con) un template di costruito while
Do While		Menu Inserisci	CTRL-6	Inserisce (o circonda il testo selezionato con) un template di costruito do-while
Try/catch		Menu Inserisci	CTRL-7	Inserisce (o circonda il testo selezionato con) un template di blocco try/catch
Commenta selezione		Menu Inserisci	CTRL-8	Inserisce (o circonda il testo selezionato con) un template commento
Proprietà_JavaBean		Menu Inserisci	CTRL-9	Apri wizard per creare proprietà_JavaBean
Barra degli strumenti		Menu Visualizza		Nasconde/visualizza barra degli strumenti
Barra di stato		Menu Visualizza		Nasconde/visualizza barra di stato
Scegli cartella di lavoro		Menu Strumenti		Permette di scegliere una cartella di lavoro che verrà aperta nell'albero del pannello 1
Prossimo file		Menu Visualizza, Barra degli strumenti	F5	Seleziona il prossimo file
File precedente Barra degli strumenti		Menu Visualizza,	F4	Seleziona il file precedente
Compila		Menu Sviluppo Barra degli strumenti, Popup su area testo	CTRL-F9	Compila file corrente

Compila progetto		Menu Sviluppo, Barra degli strumenti	SHIFT-F7	Compila tutti i file aperti
Esegui		Menu Sviluppo, Barra degli strumenti, Popup su area testo	F9	Esegue file corrente
Esegui con argomenti		Menu Sviluppo	SHIFT-F9	Esegue file corrente sfruttando gli argomenti specificati
Interrompi processo		Menu Sviluppo, Barra degli strumenti		Interrompe processo corrente
Sveglia		Menu Strumenti		Permette di impostare un timeout per mostrare un messaggio
Monitor Risorse		Menu Strumenti		Mostra la memoria allocata e usata da EJE
Genera Documentazione		Menu Strumenti		Genera documentazione javadoc del file corrente
Formatta il codice		Menu Strumenti, Barra degli strumenti, Popup su area testo	CTRL-SHIFT-F	Formatta il codice
Commenta Selezione		Menu Strumenti	CTRL-SHIFT-C	Commenta il testo selezionato
Guida all'utilizzo		Menu Aiuto	F1	Mostra questo manuale utente
Documentazione Java		Menu Aiuto	F2	Mostra la documentazione della libreria standard Java
Informazioni		Menu Aiuto su EJE	CTRL-F1	Visualizza informazioni su EJE

# Model View Controller Pattern (MVC)

## D.1 Introduzione

Il pattern in questione è molto famoso ma è spesso utilizzato con superficialità dagli sviluppatori. Ciò è probabilmente dovuto alla sua complessità, dal momento che stiamo parlando di una vera e propria “composizione di pattern”. Venne introdotto nel mondo del software per la costruzione di interfacce grafiche con Smalltalk-80, ma oggi deve gran parte della sua fama a Java. L'MVC è stato infatti utilizzato per la struttura di alcuni componenti Swing e, soprattutto, è coerente con l'architettura Java 2 Enterprise Edition (J2EE).

Questo documento è liberamente ispirato, per quanto riguarda la sua struttura, alla descrizione fornita proprio dal catalogo dei pattern J2EE (Java Blueprints).

## D.2 Contesto

L'applicazione deve fornire una interfaccia grafica (GUI) costituita da più schermate, che mostrano vari dati all'utente. Inoltre le informazioni che devono essere visualizzate devono essere sempre quelle aggiornate.

**Questo punto non è solitamente valido per architetture come J2EE, dove le GUI non sono costituite da applicazioni, ma da pagine HTML. Con l'avvento della tecnologia Ajax però, le cose stanno cambiando...**

## D.3 Problema

L'applicazione deve avere una natura modulare e basata sulle responsabilità, al fine di ottenere una vera e propria applicazione component-based. Questo è conveniente per poter più facilmente gestire la manutenzione dell'applicazione. Per esempio, ai nostri giorni, con la massiccia diffusione delle applicazioni enterprise, non è possibile prevedere al momento dello sviluppo in che modo e con quale tecnologia gli utenti interagiranno con il sistema (WML?, XML?, Wi-Fi?, HTML?). Appare quindi chiaro il bisogno di un'architettura che permetta la separazione netta tra i componenti software che gestiscono il modo di presentare i dati e i componenti che gestiscono i dati stessi.

## D.4 Forze

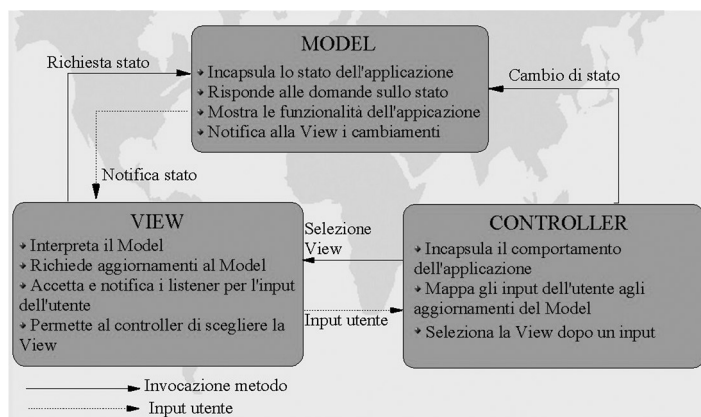
- ❑ E' possibile accedere alla gestione dei dati con diverse tipologie di GUI (magari sviluppate con tecnologie diverse)
- ❑ I dati dell'applicazione possono essere aggiornati tramite diverse interazioni da parte dei client (messaggi SOAP, richieste HTTP...)
- ❑ Il supporto di varie GUI ed interazioni non influisce sulle funzionalità di base dell'applicazione

## Soluzione e struttura

L'applicazione deve separare i componenti software che implementano il modello delle funzionalità di business dai componenti che implementano la logica di presentazione e di controllo che utilizzano tali funzionalità. Vengono quindi definite tre tipologie di componenti che soddisfano tali requisiti:

- ❑ il Model, che implementa le funzionalità di business
- ❑ la View, che implementa la logica di presentazione
- ❑ il Controller, che implementa la logica di controllo

La seguente figura D.1 rappresenta un diagramma di interazione, che evidenzia le responsabilità dei tre componenti.



**Figura D.1 - “MVC: diagramma di interazione”.**

## D.6 Partecipanti e responsabilità

### MODEL:

Analizzando la figura D.1, si evince che il core dell'applicazione viene implementato dal Model, il quale incapsulando lo stato dell'applicazione definisce i dati e le operazioni che possono essere eseguite su questi. Quindi definisce le regole di business per l'interazione con i dati, esponendo alla View ed al Controller rispettivamente le funzionalità per l'accesso e l'aggiornamento. Per lo sviluppo del Model quindi è vivamente consigliato utilizzare le tipiche tecniche di progettazione object-oriented, al fine di ottenere un componente software che astragga al meglio concetti importati dal mondo reale. Il Model può inoltre avere la responsabilità di notificare ai componenti della View eventuali aggiornamenti verificatisi in seguito a richieste del Controller, al fine di permettere alle View di presentare agli occhi degli utenti dati sempre aggiornati.

### VIEW:

La logica di presentazione dei dati viene gestita solo e solamente dalla View. Ciò implica che questa deve fondamentalmente gestire la costruzione dell'interfaccia grafica (GUI), che rappresenta il mezzo mediante il quale gli utenti interagiranno con il sistema. Ogni GUI può essere costituita da schermate diverse che presentano più modi di interagire con i dati dell'applicazione. Per far sì che i dati presentati siano sempre aggiornati è possibile adottare due strategie, note come “push model”

e “pull model”. Il push model adotta il pattern Observer, registrando le View come osservatori del Model. Le View possono quindi richiedere gli aggiornamenti al Model in tempo reale grazie alla notifica di quest’ultimo. Benché questa rappresenti la strategia ideale, non è sempre applicabile. Per esempio nell’architettura J2EE, se le View che vengono implementate con JSP restituiscono GUI costituite solo da contenuti statici (HTML) e quindi non in grado di eseguire operazioni sul Model. In tali casi è possibile utilizzare il “pull Model”, dove la View richiede gli aggiornamenti quando “lo ritiene opportuno”. Inoltre la View delega al Controller l’esecuzione dei processi richiesti dall’utente dopo averne catturato gli input e la scelta delle eventuali schermate da presentare.

### **CONTROLLER:**

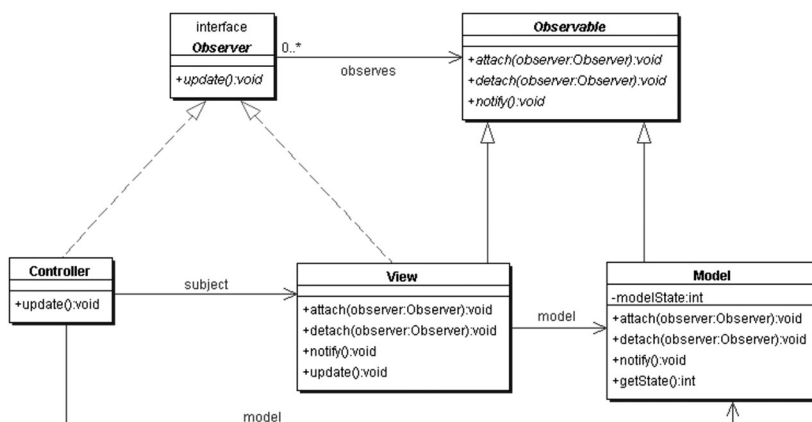
Questo componente ha la responsabilità di trasformare le interazioni dell’utente della View in azioni eseguite dal Model. Ma il Controller non rappresenta un semplice “ponte” tra View e Model. Realizzando la mappatura tra input dell’utente e processi eseguiti dal Model e selezionando la schermate della View richieste, il Controller implementa la logica di controllo dell’applicazione.

## **D.7 Collaborazioni**

In figura D.2 viene evidenziata, mediante un diagramma delle classi, la vera natura del pattern MVC. In pratica, View e Model sono relazionati tramite il pattern Observer, dove la View “osserva” il Model. Il pattern Observer caratterizza anche il legame tra View e Controller, ma in questo caso è la View ad essere “osservata” dal Controller. Ciò supporta la registrazione dinamica al runtime dei componenti. Inoltre il pattern Strategy potrebbe semplificare la possibilità di cambiare la mappatura tra gli algoritmi che regolano i processi del Controller e le interazioni dell’utente con la View.

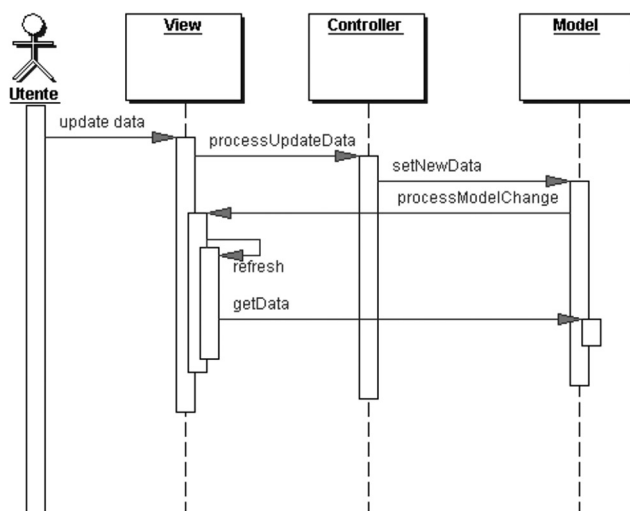
**Che si tratti di un pattern che ne contiene altri, risulta abbastanza evidente. Ma nella sua natura originaria l’MVC comprendeva anche l’implementazione di altri pattern, come il Factory Method per specificare la classe Controller di default per una View, il Composite per costruire View ed il Decorator per aggiungere altre proprietà (per esempio lo scrolling).**





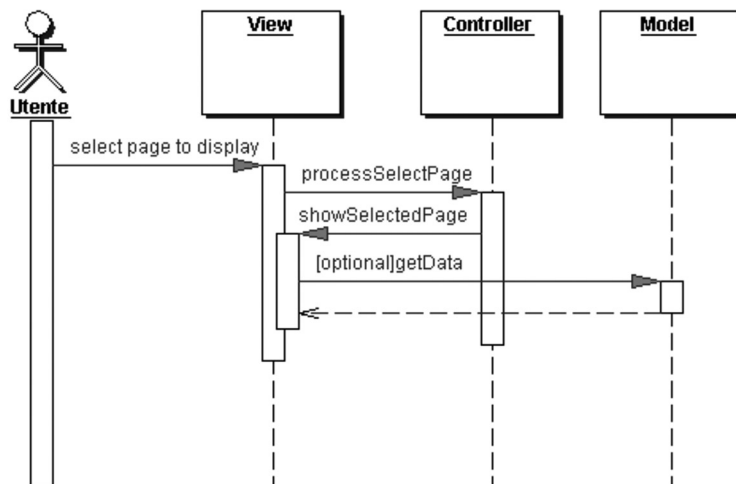
**Figura D.2 - “MVC: diagramma delle classi”.**

Evidenziamo ora solo due dei possibili scenari che potrebbero presentarsi utilizzando un'applicazione MVC-based: la richiesta dell'utente di aggiornare dati e la richiesta dell'utente di selezionare una schermata, rispettivamente illustrate tramite diagrammi di sequenze nelle figure D.3 e D.4.



**Figura D.3 - “Scenario aggiornamento dati”.**

Dalla figura D.3 evidenziamo il complesso scambio di messaggi tra i tre partecipanti al pattern il quale, benché possa sembrare inutile a prima vista, garantisce pagine sempre aggiornate in tempo reale all'utente.

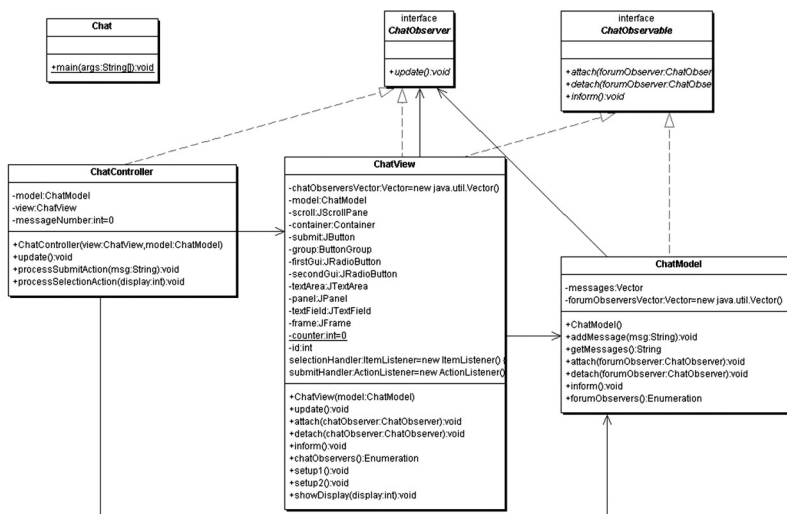


**Figura D.4 - “Scenario selezione schermata”.**

Nella figura D.4 vengono enfatizzati i ruoli di View e Controller. La View, infatti non decide quale sarà la schermata richiesta, delegando ciò alla decisione del Controller (che grazie al pattern Strategy può anche essere anche cambiato dinamicamente al runtime); piuttosto si occupa della costruzione e della presentazione all’utente della schermata stessa.

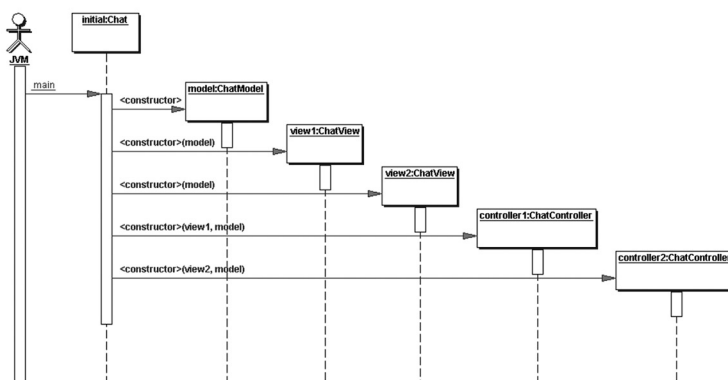
## D.8 Codice d’esempio

Nell’esempio presentato di seguito viene implementata una piccola applicazione che simula una chat ma funziona solo in locale, allo scopo di rappresentare un esempio semplice di utilizzo del pattern MVC. E’ un esempio didattico e quindi si raccomanda al lettore di non perdersi nei dettagli inutili, ma si consiglia piuttosto, di concentrarsi sulle caratteristiche del pattern. Nella figura D.5 è rappresentato il diagramma delle classi della chat dal punto di vista dell’implementazione.



**Figura D.5 - “Diagramma delle classi dal punto di vista dell’implementazione”.**

La classe `Chat` definisce il metodo `main()` che semplicemente realizza lo startup dell’applicazione, così come descritto dal diagramma di sequenza in figura D.6.



**Figura D.6 - “Startup dell’applicazione”.**

La classe `ChatView` definisce due metodi, `setup1()` e `setup2()`, per realizzare due schermate (molto simili in realtà, ma è solo un esempio). Essa definisce anche due classi interne anonime che fungono da listener. I listener, una volta notificati, delegano all’oggetto (o agli oggetti) `ChatController` (registratisi in fase di startup

come observer della View) tramite cicli di notifica come il seguente:

```
public void itemStateChanged(ItemEvent e) {
    ChatController con;
    for (int i = 0; i < chatObserversVector.size(); i++) {
        con = (ChatController)chatObserversVector.get(i);
        con.processSelectionAction(e.getStateChange());
    }
}
```

Di seguito è riportata l'intera classe ChatController:

```
public class ChatController implements ChatObserver {
    private ChatModel model;
    private ChatView view;
    private int messageNumber = 0;

    public ChatController(ChatView view, ChatModel model) {
        this.view = view;
        this.model = model;
        view.attach(this);
    }

    public void update() {
        messageNumber++;
        System.out.println(messageNumber);
    }

    public void processSubmitAction(String msg) {
        model.addMessage(msg);
    }

    public void processSelectionAction(int display) {
        view.showDisplay(display);
    }
}
```

Si può notare come in questo esempio il metodo `update()` si limiti a incrementare un contatore di messaggi, mentre la vera logica di controllo è implementata mediante i metodi (che potrebbero essere strategy, ma in questo caso abbiamo semplificato) `processSubmitAction()` e `processSelectionAction()`.

Per avere un quadro più completo, riportiamo la classe ChatModel per intero (per scaricare l'intero codice in file zip basta collegarsi a

[http://www.claudiodesio.com/download/mvc\\_code.zip](http://www.claudiodesio.com/download/mvc_code.zip)):

```
import java.util.*;

public class ChatModel implements ChatObservable {
    private Vector messages;
    private Vector forumObserversVector = new Vector();

    public ChatModel() {
        messages = new Vector();
    }

    public void addMessage(String msg) {
        messages.add(msg);
        inform();
    }

    public String getMessages() {
        int length = messages.size();
        String allMessages = "";
        for (int i = 0; i < length; ++i) {
            allMessages += (String)messages.elementAt(i)+"\n";
        }
        return allMessages;
    }

    public void attach(ChatObserver forumObserver){
        forumObserversVector.addElement(forumObserver);
    }

    public void detach(ChatObserver forumObserver){
        forumObserversVector.removeElement(forumObserver);
    }

    public void inform(){
        java.util.Enumeration enumeration = forumObservers();
        while (enumeration.hasMoreElements()) {
            ((ChatObserver)enumeration.nextElement()).update();
        }
    }

    public Enumeration forumObservers(){
        return ((java.util.Vector) forumObserversVector.clone()).elements();
    }
}
```

## D.9 Conseguenze

- ❑ Riutilizzo dei componenti del Model; la separazione tra Model e View permette a diverse GUI di utilizzare lo stesso Model. Conseguentemente, i componenti del Model sono più semplici da implementare, testare e mantenere, giacché tutti gli accessi passano tramite questi componenti.
- ❑ Supporto più semplice per nuovi tipi di client; infatti basterà creare View e Controller per interfacciarsi ad un Model già implementato.
- ❑ Complessità notevole della progettazione; questo pattern introduce molte classi extra ed interazioni tutt'altro che scontate.

## D.10 Conclusioni

Il pattern MVC introduce una notevole complessità all'applicazione, ed è sicuramente non di facile comprensione. Tuttavia il suo utilizzo si rende praticamente necessario in tantissime applicazioni moderne, dove le GUI sono insostituibili. Non conviene avventurarsi alla scoperta di nuove interazioni tra logica di business e di presentazione, se una soluzione certa esiste già.

# Introduzione all'HTML

HTML è l'acronimo per HyperText Markup Language, ovvero linguaggio di marcatura per ipertesti. Un ipertesto è una tipologia di documento che include, oltre che a semplice testo, collegamenti ad altre pagine.

Non si tratta di un vero linguaggio di programmazione, ma solo di un linguaggio di formattazione di pagine. Non esistono per esempio strutture di controllo come if o for. Le istruzioni dell'HTML si dicono tag. I tag, in italiano "etichette", sono semplici istruzioni con la seguente sintassi:

```
<NOME_TAG [LISTA DI ATTRIBUTI]>
```

Dove ogni attributo, opzionale, ha una sintassi del tipo:

```
CHIAVE=VALORE
```

Il valore di un attributo potrebbe e dovrebbe essere compreso tra due apicetti o due virgolette. Ciò è però necessario se e solo se il valore è costituito da più parole separate. Inoltre, quasi Tutti i tag HTML, salvo rare eccezioni, vanno chiusi, con una istruzione del tipo:

```
</NOME_TAG>
```

Ad esempio, il tag `<HTML>` va chiuso con `</HTML>`; il tag `<applet code='StringApplet' width='100' height='100'>` va chiuso con `</applet>`.

Per scrivere una semplice pagina HTML non occorre altro che un semplice editor di testo come il blocco note. Un browser come Internet Explorer può poi farci visualizzare la pagina formattata.

Se salviamo il seguente file di testo con suffisso “htm” o “html” contenente i seguenti tag:

```
<HTML>
  <HEAD>
    <TITLE>Prima pagina HTML</TITLE>
  </HEAD>
  <BODY>
    <CENTER>
      Hello HTML
    </CENTER>
  </BODY>
</HTML>
```

possiamo poi visualizzare il risultato aprendo il nostro file con un browser come Internet Explorer.

L'HTML non è case-sensitive.

Non resta altro che procurarci un manuale HTML (da Internet se ne possono scaricare centinaia), o semplicemente imparare da pagine già fatte andando a spiare il codice che è sempre disponibile. Esistono tantissimi tag da scoprire...



# Introduzione allo Unified Modeling Language

## F.1 Che cos'è UML (What)?

Oggigiorno si sente parlare molto spesso di UML, ma non tutte le persone che parlano di UML sanno di che cosa realmente si tratti. Qualcuno pensa che sia un linguaggio di programmazione e quest'equivoco è dovuto alla parola “language”. Qualcun altro pensa si tratti di una metodologia object-oriented e questo è probabilmente dovuto a cattiva interpretazione di letture non molto approfondite. Infatti si sente spesso parlare di UML congiuntamente a varie metodologie. Per definire quindi correttamente che cos'è UML, è preferibile prima definire per grandi linee che cos'è una metodologia.

Una **metodologia** object-oriented, nella sua definizione più generale, potrebbe intendersi come una coppia costituita da un processo e da un linguaggio di modellazione.

**Quella riportata, per precisione, è la definizione di metodo. Una metodologia è tecnicamente definita come “la scienza che studia i metodi”. Spesso però questi termini sono considerati sinonimi.**

A sua volta un **processo** potrebbe essere definito come la serie di indicazioni riguardanti i passi da intraprendere per portare a termine con successo un progetto.

Un **linguaggio di modellazione** è invece lo strumento che le metodologie utilizzano per descrivere (possibilmente in maniera grafica) tutte le caratteristiche statiche e dinamiche di un progetto.

In realtà UML non è altro che un linguaggio di modellazione. Esso è costituito per linee generali da una serie di diagrammi grafici i cui elementi sono semplici linee, triangoli, rettangoli, omini stilizzati e così via. Questi diagrammi hanno il compito di descrivere in modo chiaro tutto ciò che durante un progetto potrebbe risultare difficile o troppo lungo con documentazione testuale.

## F.2 Quando e dove è nato (When & Where)

A partire dai primi anni Ottanta la scena informatica mondiale iniziò ad essere invasa dai linguaggi orientati ad oggetti quali SmallTalk e soprattutto C++. Questo perché col crescere della complessità dei software, e della relativa filosofia di progettazione, la programmazione strutturata mostrò i suoi limiti e si rivelò insufficiente per soddisfare le sempre maggiori pretese tecnologiche. Ecco allora l'affermarsi della nuova mentalità ad oggetti e la nascita di nuove teorie, che si ponevano come scopo finale il fornire tecniche più o meno innovative per realizzare software, ovviamente sfruttando i paradigmi degli oggetti. Nacquero una dopo l'altra le metodologie object-oriented, in gran quantità e tutte più o meno valide. Inizialmente esse erano strettamente legate ad un ben determinato linguaggio di programmazione, ma la mentalità cambiò abbastanza presto. A partire dal 1988 iniziarono ad essere pubblicati i primi libri sull'analisi e la progettazione orientata agli oggetti. Nel '93 si era arrivati ad un punto in cui c'era gran confusione: analisti e progettisti esperti come James Rumbaugh, Jim Odell, Peter Coad, Ivar Jacobson, Grady Booch ed altri proponevano tutti una propria metodologia ed ognuno di loro aveva una propria schiera di entusiasti seguaci. Quasi tutti gli autori più importanti erano americani ed inizialmente le idee che non provenivano dal Nuovo continente erano accolte con sufficienza, e qualche volta addirittura derise. In particolare Jacobson, prima di rivoluzionare il mondo dell'ingegneria del software con il concetto di "Use Case", fu denigrato da qualche autore americano, che definì "infantile" il suo modo di utilizzare omini stilizzati come fondamentali elementi dei suoi diagrammi. Probabilmente fu solo un tentativo di sbarazzarsi di un antagonista scomodo, o semplicemente le critiche provennero da fonti non certo lungimiranti...

Questi episodi danno però l'idea del clima di competizione in quel periodo (si parla di "guerra delle metodologie"). UML ha visto ufficialmente la luce a partire dal 1997... i tempi dovevano ancora maturare...

## F.3 Perché è nato UML(Why)

Il problema fondamentale era che diverse metodologie proponevano non solo diversi pro-

cessi, il che può essere valutato positivamente, ma anche diverse notazioni. Era chiaro allora a tutti che non doveva e poteva esistere uno standard tra le metodologie. Infatti i vari processi esistenti erano proprietari di caratteristiche particolarmente adatte a risolvere alcune particolari problematiche. In pratica, quando si inizia un progetto, è giusto avere la possibilità di scegliere tra diversi stratagemmi risolutivi (processi). Il fatto che ogni processo sia strettamente legato ad un determinato linguaggio di modellazione ovviamente non rappresenta altro che un intralcio per i vari membri di un team di sviluppo. L'esigenza di un linguaggio standard per le metodologie era avvertita da tanti, ma nessuno degli autori aveva intenzione di fare il primo passo.

## F.4 Chi ha fatto nascere UML (Who)

Ci pensò allora la Rational Software Corporation (<http://www.rational.com>), che annoverava tra i suoi esperti Grady Booch, autore di una metodologia molto famosa all'epoca, nota come "Booch 93". Nel '94 infatti, James Rumbaugh, creatore della Object Modelling Technique (OMT), probabilmente la più utilizzata tra le metodologie orientate agli oggetti, si unisce a Booch alla Rational. Nell'ottobre del '95 fu rilasciata la versione 0.8 del cosiddetto "Unified Method". Ecco che allora lo svedese Ivar Jacobson nel giro di pochi giorni si unisce a Booch e Rumbaugh, iniziando una collaborazione che poi è divenuta storica. L'ultimo arrivato portava in eredità, oltre che il fondamentale concetto di "Use Case", la famosa metodologia "Object Oriented Software Engineering" (OOSE) conosciuta anche come "Objectory", che in realtà era il nome della società di Jacobson, oramai inglobata da Rational. Ecco allora che i "tres amigos" (così vennero soprannominati) iniziarono a lavorare per realizzare lo "Unified Software Development Process" ("USDP") e soprattutto al progetto UML. L'Object Management Group (OMG, <http://www.omg.org>), un consorzio senza scopi di lucro che si occupa delle standardizzazioni, le manutenzioni e le creazioni di specifiche che possano risultare utili al mondo dell'information technology, nello stesso periodo inoltra a tutti i più importanti autori una "richiesta di proposta" ("Request For Proposal", RFP) di un linguaggio di modellazione standard. Ecco che allora Rational, insieme con altri grossi partner quali IBM e Microsoft, propose la versione 1.0 di UML nell'ottobre del 1997. OMG rispose istituendo una "Revision Task Force" (RTF) capitanata attualmente da Cris Kobryn per apportare modifiche migliorative ad UML.

La versione attuale di UML è la 2.0, ma c'è molta confusione tra gli utenti di UML. Infatti, la difficoltà di "interpretare le specifiche", i punti di vista differenti da parte degli autori più importanti e l'ancoraggio di alcuni autori alle prime versioni del lin-

guaggio (tres amigos in testa) rendono purtroppo il quadro poco chiaro. Effettivamente OMG ha come scopo finale far diventare UML uno standard ISO e per questo le specifiche sono destinate, più che agli utenti di UML, ai creatori dei tool di sviluppo di UML. Ecco perché le specifiche hanno la forma del “famigerato metamodello UML”. Ovvero, UML viene descritto tramite se stesso. In particolare, il metamodello UML è diviso in quattro sezioni e, giusto per avere un’idea, viene definito un linguaggio (l’“Object Constraint Language - OCL”) solo allo scopo di definire impeccabilmente la sintassi. Tutto questo nella apprezzabilissima ipotesi futura di creare applicazioni solo trascinando elementi UML uno sull’altro tramite tool come Together (<http://www.borland.com>)... pratica che per chi scrive attualmente costituisce dal 50% al 70% del ciclo di sviluppo del software. Intanto bisogna orientarsi tra tante informazioni diverse...

# UML Syntax Reference

## Unified Modeling Language Syntax Reference

<b>Use Case Diagram</b>	Actor	Use Case
	Relationship Link	System Boundary
	Inclusion	Extension
	Generalization	Actor Generalization

Diagramma dei casi d'uso: rappresentano le interazioni tra il sistema e gli utenti del sistema stesso.

<b>Class Diagram</b>	Class/Object	Association/link	Navigability
	Attribute	Aggregation	Multiplicity
	Operation	Composition	Qualified Association
	Member Properties	Extension	Association Class
	Abstract Class/Interface	Implementation	Roles Names

Diagramma delle classi: descrive le classi che compongono il sistema e le relazioni statiche esistenti tra esse.

N.B.: Quando un diagramma mostra gli oggetti del sistema, spesso ci si riferisce ad esso come diagramma degli oggetti (Object Diagram).

<b>Component Diagram</b>	Component	Dependency
--------------------------	-----------	------------

Diagramma dei componenti: descrive i componenti software e le loro dipendenze.

<b>Deployment Diagram</b>	Node	Link
---------------------------	------	------

Diagramma di installazione (di dispiegamento, schieramento): mostra il sistema fisico.

<b>Interaction Diagrams:</b>	Actor	Message
<b>Sequence &amp; Collaboration</b>	Object	Asynchronous Message
	Creation	Life Line
	Destruction	Activity Line

	State	Transition
	Diagrammi di interazione: mostrano come gruppi di oggetti collaborano in un determinato lasso temporale.	
	Diagramma di sequenza: esalta la sequenza dei messaggi.	
	Diagramma di collaborazione: esalta la struttura architetturale degli oggetti.	
<b>State Transition Diagram</b>	Start	Action
	End	History
	Diagramma degli stati (di stato): descrive il comportamento di un oggetto mostrando gli stati e gli eventi che causano i cambiamenti di stato (transizioni).	
<b>Activity Diagram</b>	Activity	Flow
	Branch/Merge	Fork/Join
	Swimlane	Other elements
	Diagramma delle attività: descrive i processi del sistema tramite sequenze di attività sia condizionali sia parallele.	
<b>General Purpose Elements &amp; Extension Mechanism</b>	Package	Iteration mark
	Stereotype	Condition
	Constraint	Tagged Value
	Elementi generici e meccanismi d'estensione: elementi UML utilizzabili nella maggior parte dei diagrammi	

# Use Case Diagram Syntax Reference

## Actor



Attore: ruolo interpretato dall'utente nei confronti del sistema.  
N.B.: un utente potrebbe non essere una persona fisica.

## Use Case



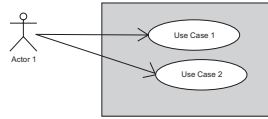
Caso d'uso: insieme di scenari legati da un obiettivo comune per l'utente. Uno scenario è una sequenza di passi che descrivono l'interazione tra l'utenza ed il sistema.  
NB: è possibile descrivere scenari mediante diagrammi dinamici.

## Relationship link



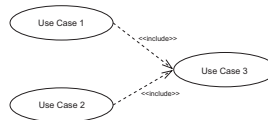
Associazione (o relazione): relazione che associa logicamente un attore ad un caso d'uso.

## System Boundary



Sistema (o delimitatore del sistema): delimitatore del dominio del sistema.

## Inclusion



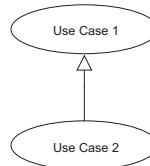
Inclusione: relazione logica tra casi d'uso, che estrae un comportamento comune a più casi d'uso.

## Extension



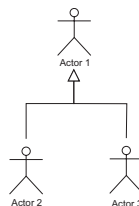
Estensione: relazione logica che lega casi d'uso, che hanno lo stesso obiettivo semantico. Il caso d'uso specializzato raggiunge lo scopo aggiungendo determinati punti d'estensione, che sono esplicitati nel caso d'uso base. Punto d'estensione: descrive un comportamento di un caso d'uso specializzato, non utilizzato dal caso d'uso base.

## Generalization



Generalizzazione: relazione logica che lega casi d'uso, che hanno lo stesso obiettivo semantico. Il caso d'uso specializzato, raggiunge lo scopo aggiungendo nuovi comportamenti non utilizzati dal caso d'uso base, ma senza formalismi sintattici.

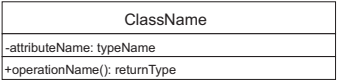
## Actor generalization



Generalizzazione tra attori: relazione logica che lega attori. Un attore che specializza un attore base, può relazionarsi a qualsiasi caso d'uso relazionato al caso d'uso base. Inoltre può relazionarsi anche ad altri casi d'uso, non relazionati con il caso d'uso base.

# Class Diagram Syntax Reference

## Class & Object



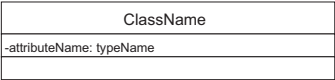
Class



Object

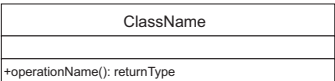
Classe: astrazione per un gruppo di oggetti che condividono stesse caratteristiche e funzionalità.  
Oggetto: creazione fisica (o istanza) di una classe.

## Attribute



Attributo (o variabile d'istanza o variabile membro o caratteristica di una classe): caratteristica di una classe/oggetto.

## Operation



Operazione (o metodo o funzione membro): funzionalità di una classe/oggetto.

## Member Properties

+ memberName “public member”

# memberName “protected member”

– memberName “private member”

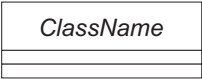
memberName o \$memberName “static member”

operation o operation {abstract} “abstract operation”

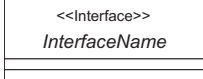
/attributeName “derived attribute”

Public, protected, private: modificatori di visibilità.  
Membro statico (o membro della classe): membro della classe.  
Operazione astratta: “signature” del metodo (metodo senza implementazione).  
Attributo derivato: attributo ricavato da altri.

## Abstract Class & Interface



Abstract Class

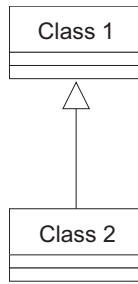


Interface

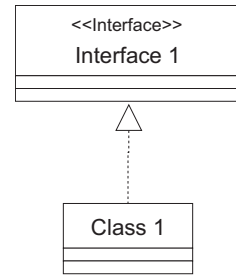
Classe astratta: classe che non si può istanziare e che può dichiarare metodi astratti.  
Interfaccia: struttura dati non istanziabile che può dichiarare solo metodi astratti (e costanti statiche pubbliche).



## Extension & Implementation



Extension

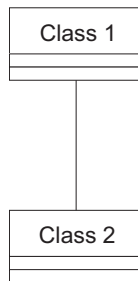


Implementation

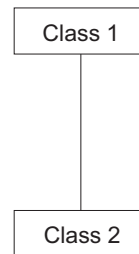
Estensione: relazione di ereditarietà tra classi.

Implementazione: estensione ai fini dell'implementazione dei metodi astratti di un'interfaccia.

## Association



Association



Link

Associazione: relazione tra classi dove una utilizza i servizi (le operazioni) dell'altra.

Link: relazione tra associazione tra oggetti.

## Aggregation



Aggregazione: associazione caratterizzata dal contenimento.

## Composition



Composizione: aggregazione con cicli di vita coincidenti.

## Navigability



Navigabilità: direzione di un'associazione.

**Multiplicity**



Molteplicità: corrispondenza tra la cardinalità del numero di oggetti delle classi coinvolte nell’associazione.

---

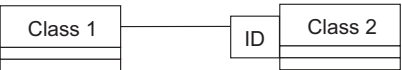
**Role Names**



Ruoli: descrizione del comportamento di una classe relativamente ad un’associazione.

---

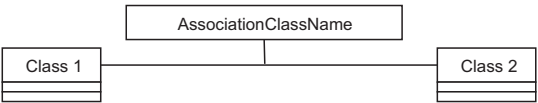
**Qualified Association**



Associazione qualificata: associazione nella quale un oggetto di una classe è identificato dalla classe associata mediante una “chiave primaria”.

---

**Association Class**



Class d’associazione: codifica in classe di un’associazione.

---

# Component & Deployment Diagram Syntax Reference

---

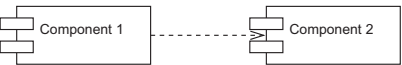
**Component**



Componente : modulo software eseguibile, dotato di identità e con un’interfaccia ben specificata, di cui di solito è possibile uno sviluppo indipendente.

---

**Dependency**



Dipendenza: relazione tra due elementi di modellazione, nella quale un cambiamento sull’elemento indipendente avrà impatto sull’elemento dipendente.

---

**Link**



Associazione (o relazione): relazione logica nella quale un partecipante (componente o nodo o classe) utilizza i servizi dell’altro partecipante.

---

Node



Nodo : rappresentazione di una piattaforma hardware.

Interaction Diagram Syntax Reference

Actor



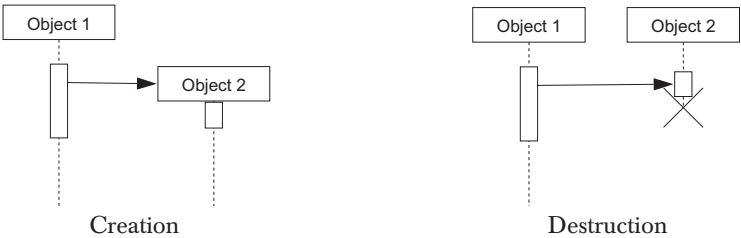
Attore: ruolo interpretato dall'utente nei confronti del sistema.  
N.B.: un utente potrebbe non essere una persona fisica.

Object



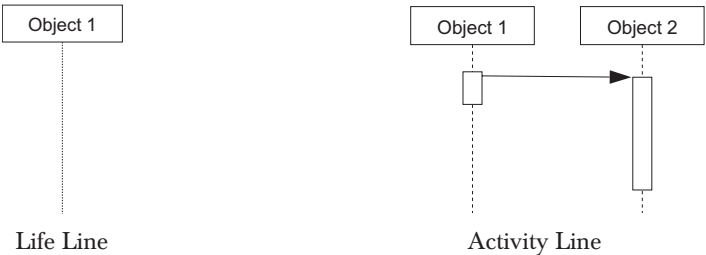
Oggetto: creazione fisica (o istanza) di una classe

Creation & Destruction



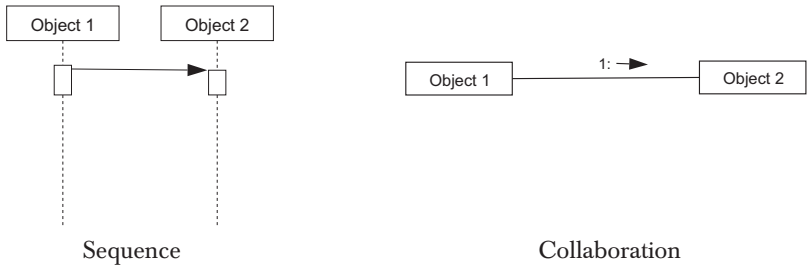
Creazione: punto d'inizio del ciclo di vita di un oggetto (può coincidere con una chiamata al costruttore dell'oggetto creato).  
Distruzione: punto terminale del ciclo di vita di un oggetto (può coincidere con una chiamata al distruttore dell'oggetto creato).

Life Line & Activity Line



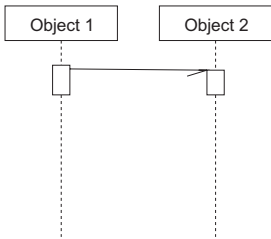
Linea di vita: linea di vita di un oggetto.  
Linea di attività: linea di attività di un oggetto (rappresenta il blocco d'esecuzione di un metodo).

**Message**



Messaggio: messaggio di collaborazione tra oggetti (può coincidere con una chiamata al costruttore dell'oggetto di destinazione).

**Asynchronous Message**



Messaggio asincrono: messaggio che può essere eseguito in maniera asincrona.

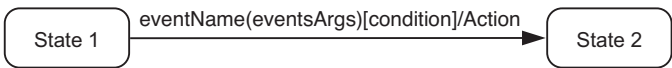
# State Transition Diagram Syntax Reference

**State**



Stato: stato di un oggetto. Può essere caratterizzato da azioni (transizioni interne).

**Transition**



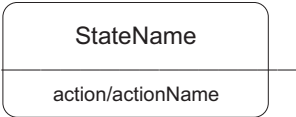
Transizione: attività che termina portando un oggetto in uno stato.

**Start & End**



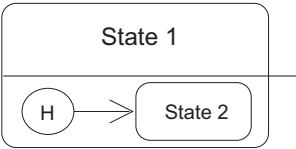
Stato iniziale: punto iniziale di uno state transition diagram. Stato finale: punto terminale di uno state transition diagram.

Action



Azione: attività che caratterizza uno stato.

History



Stato con memoria: stato capace di ripristinare situazioni precedentemente attraversate.

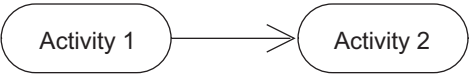
Activity Diagram Syntax Reference

Activity



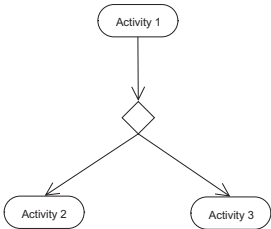
Attività: processo assolto dal sistema.

Flow

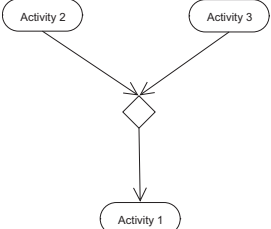


Flusso (di attività): insieme di scenari legati da un obiettivo comune per l'utente.

Branch & Merge



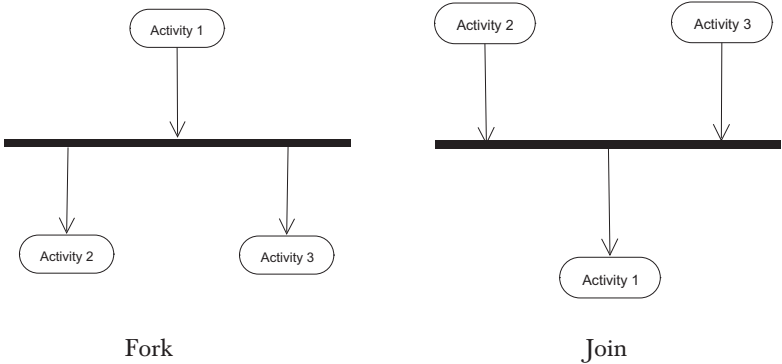
Branch



Merge

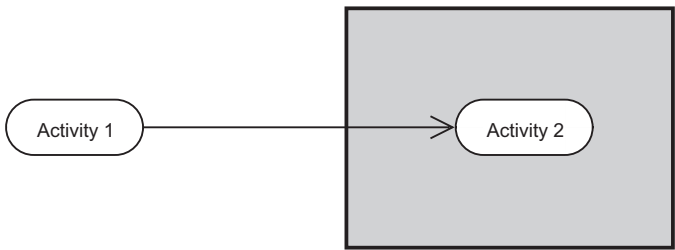
Ramificazione: rappresenta una scelta condizionale.  
Unione: rappresenta un punto di terminazione di blocchi condizionali.

**Fork & Join**



Biforcazione: rappresenta un punto di partenza per attività concorrenti.  
Riunione: rappresenta un punto di terminazione per attività concorrenti .

**SwimLane**



Corsia: area di competenza di attività.

**Start & End**



Stato iniziale: punto iniziale di uno state transition diagram.  
Stato finale: punto terminale di uno state transition diagram.

**Object & State**



Oggetto: creazione fisica (o istanza) di una classe.  
Stato: stato di un oggetto. Può essere caratterizzato da azioni (transizioni interne).

# General Purpose Elements Syntax Reference

**Package & Stereotype**



<<Stereotype>>

Package

Stereotype

Package: notazione che permette di raggruppare elementi UML.  
Stereotipo: meccanismo di estensione che permette di stereotipare costrutti non standard in UML.

**Constraint &  
Tagged value**

{constraint}

{key = value}

Constraint

Tagged Value

Vincolo: meccanismo di estensione che permette di esprimere vincoli.  
Valore etichettato: vincolo di proprietà.

**Iteration Mark &  
Condition**

\*

[condition]

Iteration Mark

Condition

Marcatore di iterazione: rappresenta un iterazione.  
Condition: rappresenta una condizione.

# Introduzione ai Design Pattern

L'applicazione dell'Object Orientation ai processi di sviluppo software complessi, comporta non poche difficoltà. Uno dei momenti maggiormente critici nel ciclo di sviluppo è quello in cui si passa dalla fase di analisi a quella di progettazione. In tali situazioni bisogna compiere scelte di design particolarmente delicate, dal momento che potrebbero pregiudicare il funzionamento e la data di rilascio del software. In tale contesto (ma non solo) si inserisce il concetto di design pattern, importato nell'ingegneria del software direttamente dall'architettura. Infatti la prima definizione di pattern fu data da Cristopher Alexander, un importante architetto austriaco (insegnante all'università di Berkeley - California) che iniziò a formalizzare tale concetto sin dagli anni '60. Nel suo libro "Pattern Language: Towns, Buildings, Construction" (Oxford University Press, 1977) Alexander definisce un pattern come una soluzione architeturale che può risolvere problemi in contesti eterogenei.

La formalizzazione del concetto di Design Pattern (pattern di progettazione) è ampiamente attribuita alla cosiddetta Gang of Four (brevemente GOF). La gang dei quattro è costituita da Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides che, nonostante provenissero da tre continenti diversi, in quattro anni di confronti catalogarono la prima serie di 23 pattern che costituisce il nucleo fondamentale della tecnica. Nel 1994 vede la luce il libro considerato la guida di riferimento per la comunità dei pattern: "Design Pattern: elements of reusable object oriented software" (Addison-Wesley). Altri autori in seguito hanno pubblicato testi che estendono il numero dei pattern noti...



## H.1 Definizione di Design Pattern

I Design Pattern rappresentano soluzioni di progettazione generiche applicabili a problemi ricorrenti all'interno di contesti eterogenei. Consapevoli che l'asserzione precedente potrebbe non risultare chiara al lettore che non ha familiarità con determinate situazioni, cercheremo di descrivere il concetto presentando, oltre che la teoria, anche alcuni esempi di pattern. In questo modo si potranno meglio apprezzare sia i concetti sia l'applicabilità.

L'idea di base, però, è piuttosto semplice. E' risaputo che la bontà della progettazione è direttamente proporzionale all'esperienza del progettista. Un progettista esperto risolve i problemi che si presentano utilizzando soluzioni che in passato hanno già dato buoni risultati. La GOF non ha fatto altro che confrontare la (ampia) esperienza dei suoi membri nel trovare soluzioni progettuali, scoprendo così intersezioni abbastanza evidenti. Siccome queste intersezioni sono anche soluzioni che risolvono frequentemente problemi, in contesti eterogenei, possono essere dichiarate e formalizzate come Design Pattern. In questo modo si mettono a disposizione soluzioni a problemi comuni, anche ai progettisti che non hanno una esperienza ampia come quella della GOF. Quindi stiamo parlando di una vera e propria rivoluzione!

## H.2 GOF Book: formalizzazione e classificazione

La GOF ha catalogato i pattern utilizzando un formalismo ben preciso. Ogni pattern, infatti, viene presentato tramite il nome, il problema a cui può essere applicato, la soluzione (non in un caso particolare) e le conseguenze. In particolare, ogni pattern viene descritto tramite l'elenco degli elementi a loro parere maggiormente caratterizzanti:

1. Nome e classificazione: importante per il vocabolario utilizzato nel progetto
2. Scopo: breve descrizione di cosa fa il pattern e suo fondamento logico
3. Nomi alternativi (se ve ne sono): molti pattern sono conosciuti con più nomi
4. Motivazione: descrizione di uno scenario che illustra un problema di progettazione e la soluzione offerta
5. Applicabilità: quando può essere applicato il pattern

6. **Struttura (o modello):** rappresentazione grafica delle classi del pattern mediante una notazione (in questa sede si utilizzerà UML, ma sul libro la cui nascita è antecedente alla nascita di UML vengono utilizzati OMT di Rumbaugh e i diagrammi di interazione di Booch)
7. **Partecipanti:** le classi/oggetti con le proprie responsabilità
8. **Collaborazioni:** come collaborano i partecipanti per poter assumersi le responsabilità
9. **Conseguenze:** pro e contro dell'applicazione del pattern
10. **Implementazione:** come si può implementare il pattern
11. **Codice d'esempio:** frammenti di codice che aiutano nella comprensione del pattern (in questa sede si utilizzerà Java, ma sul libro la cui nascita è antecedente alla nascita di Java, vengono utilizzati C++ e SmallTalk)
12. **Pattern correlati:** relazioni con altri pattern
13. **Utilizzi noti:** esempi di utilizzo reale del pattern in sistemi esistenti

I 23 pattern presentati nel libro della GOF sono classificati in tre categorie principali, basandosi sullo scopo del pattern:

- ☐ **Pattern creazionali:** propongono soluzioni per creare oggetti
- ☐ **Pattern strutturali:** propongono soluzioni per la composizione strutturale di classi e oggetti
- ☐ **Pattern comportamentali:** propongono soluzioni per gestire il modo in cui vengono suddivise le responsabilità delle classi e degli oggetti

Inoltre viene anche fatta una distinzione sulla base del raggio d'azione del pattern:

- ☐ **Class pattern:** offrono soluzioni tramite classi e sottoclassi in relazioni statiche tra loro
- ☐ **Object Pattern:** offrono soluzioni dinamiche basate sugli oggetti

Possiamo riassumere i 23 pattern del libro della GOF tramite questa tabella:

Scopo		Creazionale	Strutturale	Comportamentale
<b>Raggio d'azione</b>	<b>Class</b>	Factory Method	Adapter (class)	Interpreter Template Method
	<b>Object</b>	Abstract Factory	Adapter (object)	Chain of responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	Strategy

# Compilazione con Java 5

Supponiamo di avere a disposizione un Java Development Kit 1.5 o superiore. Esistono due possibilità:

1. compilare un file che fa uso delle nuove feature di Tiger (per esempio il nuovo ciclo `foreach`)
2. compilare un file che non fa uso delle nuove feature di Tiger (per esempio utilizza una `Collection` senza sfruttare la parametrizzazione tramite generics)

Nel primo caso e la compilazione rimane sempre la stessa:

```
javac [opzioni] NomeFile.java
```

e non dovrebbe sorgere nessun problema.

Nel secondo caso, in cui ci troveremmo per esempio se volessimo compilare file scritte prima dell'avvento di Java 5, potremmo imbatterci in qualche warning. Addirittura potremmo anche andare incontro a qualche errore, per esempio utilizzando la parola `enum` come identificatore. Infatti, `enum` è ora una nuova parola chiave del linguaggio. Per esempio, non è raro trovare codice antecedente a Java 5 che dichiara i reference di tipo `Enumeration` proprio con l'identificatore `enum`.

Nel secondo caso, quindi, è opportuno specificare al momento della compilazione l'opzione:

```
-source 1.4
```

per la verità già incontrata nel Modulo 10 relativamente all'uso delle asserzioni. In questo modo avvertiremo il compilatore di compilare il file senza tener conto delle nuove caratteristiche di Java 5.

Il punto di forza di Java 5 è proprio il compilatore. Infatti, il bytecode prodotto sarà eseguibile anche da una Java Virtual Machine di versione precedente alla 5. Questo significa che, una volta compilato un file con Java 5, questo è eseguibile anche su JVM versione 1.4 o 1.3. Quindi le novità dal linguaggio sono gestite accuratamente dal compilatore. Per ottenere questo risultato è necessario specificare il flag “-source 1.4” (o “1.3” etc...), e l’opzione “-target 1.4” (o “1.3” etc...). Infatti, l’opzione “-source” dichiara che il codice contenuto del file (o dei file) da compilare contiene una sintassi valida per la versione 1.4. Mentre l’opzione “-target” dichiara la versione di Java Virtual Machine su cui sarà possibile eseguire il bytecode compilato. Se non vengono specificate opzioni, il valore di default sia per “-source” che per “-target” sarà “1.5”.

**Quanto detto sopra può essere anche specificato con EJE. Basta lanciarlo con un JDK 1.5, o lanciarlo con un JDK 1.4.x, e specificare un JDK 1.5 dal menu File-Opzioni (la scorciatoia è F12). Dal tab “Java” potete, oltre che abilitare o meno le asserzioni, anche specificare la versione di Java da utilizzare (che di default dovrebbe rimanere 1.4, anche se avete lanciato EJE con Java 5). Questa operazione equivale a impostare il flag “-source” da riga di comando. Il combobox “target” permetterà di specificare il valore dell’opzione “-target”.**