RIVISTA DI

INFORMATICA

AICA
Associazione Italiana per
l'Informatica ed il Calcolo Automatico

1/96

Volume XXVI - N. 1
gennaio - aprile 1996
Spedizione in abbonamento
postale /50% - Milano
Prezzo a copia L. 30.750

MASSON
MASSON S.p.A.
DIV. PERIODICI TECNICI

ESTENSIONI AL PARADIGMA MODEL-VIEW-CONTROLLER PER IL PROGETTO DI APPLICAZIONI GRAFICHE INTERATTIVE

GIANLUCA CHIOZZI[1]

IBM Semea
Cir.ne Idroscalo
20090 Segrate, Milano, ITALIA

The Model-View-Controller (MVC) paradigm provides a proper design architecture, based over object oriented principles, for applications where human-computer interaction and graphical components play an important role. This paper analyses the MVC paradigm to put in evidence characteristics and limitations. Then we introduce model extensions in order to get over these limitations.

Il paradigma Model-View-Controller (MVC) mette a disposizione una architettura progettuale, basata su principi object oriented, per applicazioni in cui giocano un ruolo primario l'interazione uomo-computer e le componenti grafiche. Questo lavoro analizza il paradigma MVC per metterne in evidenza caratteristiche e limiti. Vengono quindi introdotte estensioni al modello per il loro superamento.

1. INTRODUZIONE

L'utilizzo di tecnologie di sviluppo software Object-Oriented [**boo 94**] [**cox 86**] [**mey 88**] risponde alle esigenze delle applicazioni grafiche a forte componente interattiva, per le quali l'unico metodo affidabile di progettazione è quello di realizzare dei prototipi e di farli evolvere verso l'applicazione finale sulla base delle osservazioni espresse dagli utilizzatori, alternando fasi di analisi, sviluppo e test [**mey 92**] [**mey 92b**].

Tali metodologie permettono infatti di suddividere l'applicazione in elementi indipendenti che possono evolvere senza interferire fra di loro.

Il paradigma Model-View-Controller (MVC), introdotto per la prima volta in Smalltalk-80[**kra 88**], si innesta sui principi base delle metodologie ad oggetti e mette a disposizione uno schema architetturale per le applicazioni interattive. Vengono infatti identificati con precisione i ruoli e le responsabilità delle diverse classi di oggetti che

[1]Indirizzo attuale: European Southern Observatory, Karl-Schwarschildstrasse, 2
Garching bei München D-85748 Germany
Email: gchiozzi@eso.org

costituiscono l'applicazione e le regole di colloquio e interazione fra questi[**cox 86**].

Il presente lavoro analizza il paradigma MVC evidenziandone caratteristiche e limiti. Vengono quindi presentate estensioni al modello che ne consentono il superamento.

Le idee presentate sono implementate in un toolkit C++ utilizzato in IBM Semea per lo sviluppo di sistemi di grafica interattiva 2D.

In particolare questa metodologia è stata utilizzata in sistemi complessi quali le stazioni operatore per un sistema di gestione guasti su rete di distribuzione elettrica, un prototipo per un sistema di gestione del traffico ferroviario e un sistema per il monitoraggio e il controllo del traffico autostradale.

Nel corso dell'articolo gli esempi faranno riferimento al classico e ben noto caso di un editor grafico vettoriale [**fol 90**] [**shn 92**].

2. IL PARADIGMA MVC

Il paradigma Model-View-Controller è pensato per fornire una guida al progetto delle applicazioni interattive.

L'interfaccia grafica di Smalltalk-80 è basata interamente sul modello MVC [**ada 88**] [**kra 88**]. Gli stessi concetti vengono introdotti da Cox [**cox 86**] e sono utilizzati in numerosi ambienti di sviluppo Object-Oriented. In particolare la forma qui descritta per il paradigma MVC corrisponde a quella utilizzata da Knolle [**kno 89**] [**kno 90**].

Il principio fondamentale alla base di questo modello è la netta separazione fra le strutture dati dell'applicazione, la loro rappresentazione e l'interazione con l'utilizzatore.

Vengono individuate tre componenti, che si scambiano messaggi secondo le relazioni descritte in Fig. 1:

Models: che descrivono le strutture dati dell'applicazione.
Views: che sono le loro rappresentazioni, utilizzate per mostrarle agli utilizzatori.
Controllers: che gestiscono l'interazione con gli utenti.

Questa suddivisione nasce da una serie di considerazioni:

- Modello dei dati (Model) e sue rappresentazioni (Views) vengono modificati in maniera indipendente durante lo sviluppo di una applicazione. Mentre le strutture dati sono sostanzialmente stabili, l'interfaccia utente è soggetta a sostanziali modifiche.
- La stessa struttura dati ha spesso più rappresentazioni (quali una rappresentazione grafica e un pannello alfanumerico).
- I modelli per i dati e le viste possono essere riutilizzati in applicazioni diverse.

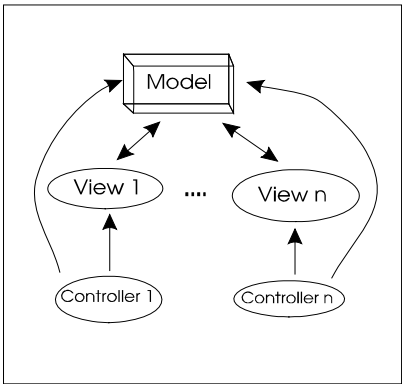

Fig. 1: Relazioni fra le componenti MVC

- L'interazione con una vista ha modalità diverse a seconda della situazione (lo stesso **elemento grafico** può esser spostato o ruotato); pertanto la gestione dell'interazione deve essere affidata ad una entità indipendente (il Controller) e intercambiabile.

3. CLASSI MODEL

Le classi **Model** forniscono i servizi per accedere ai propri attributi; nel caso, ad esempio, della **linea**, occorreranno dei servizi per modificarne posizione e colore.

La richiesta di modificare un attributo del Model viene effettuata invocando il relativo servizio da parte di qualunque oggetto dell'applicazione, e in particolare da una View relativa al Model stesso o da un suo Controller.

Come conseguenza, tutte le View che dipendono dal Model ricevono automaticamente un messaggio di notifica.

Nell'esempio di Fig. 2, il Model viene modificato nel parametro *colore*, e quindi manda a tutte le view un messaggio per comunicare la variazione.

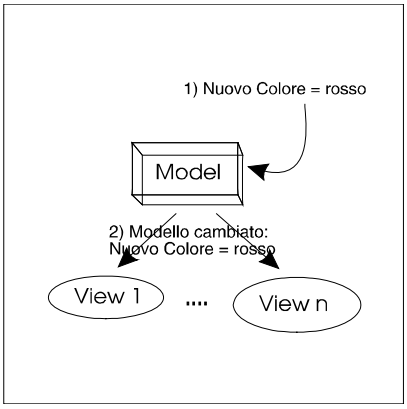

Fig. 2: Tutte le View vengono avvisate delle modifiche apportate al Model

4. CLASSI VIEW

La classe di base **View** fornisce i servizi necessari all'implementazione del concetto di "rappresentazione *inerte* di un modello", cioè priva di meccanismi di interazione.

In particolare ogni View deve poter essere associata ad un Model e deve potervi essere installato un Controller per la gestione delle procedure di interazione. A tale scopo, sono stati introdotti i servizi

- SetModel(Model)
- SetController(Controller)

Il modello al quale è associata una vista deve essere unico, ma deve poter essere sostituito dinamicamente, in modo da utilizzare la stessa View, in tempi successivi, per visualizzare diversi Model.

Analogamente vi potrà essere al più un Controller installato per ciascuna View.

5. CLASSI CONTROLLER

I Controller implementano il comportamento delle View, e pertanto ricevono e gestiscono gli eventi ad esse relativi.

Il loro compito è quello di filtrare gli input degli utenti e modificare di conseguenza la propria View e il proprio Model.

Nel caso, ad esempio, del pannello di dialogo per l'editing di una stringa, il relativo Controller dovrà occuparsi di gestire l'editing vero e proprio e la pressione del tasto OK, in risposta al quale deve invocare il servizio per modificare la stringa del Model corrispondente.

6. RELAZIONI FRA COMPONENTI

La classe Model è il fulcro della comunicazione fra le tre diverse tipologie di componenti (Fig. 1). Il Model infatti riceve messaggi e richieste di servizi dalle View, dai relativi Controller e da ogni altro oggetto dell'applicazione.

Se queste richieste comportano modifiche ai parametri del modello, tutte le View che vi

fanno riferimento ricevono automaticamente un messaggio di notifica.

I Controller invece ricevono gli eventi generati dagli utenti e a loro volta inviano messaggi e richieste alla propria View e al relativo Model.

Il paradigma quindi prevede una comunicazione bidirezionale fra il Model ed una View. Il Controller invece può solamente comunicare in modo unidirezionale verso View e Model. È importante osservare come non sia prevista comunicazione fra diverse viste relative anche allo stesso modello.

Se nell'esempio di Fig. 2, la richiesta di cambiare colore al modello fosse stata inviata dal Controller della View 1 si sarebbe verificato il flusso di messaggi in Fig. 3.

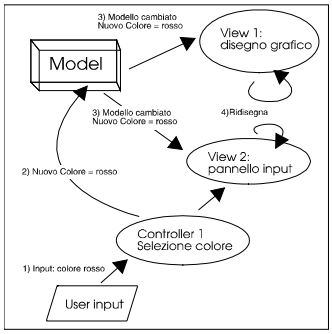

Fig. 3 - Flusso dei messaggi in risposta ad un evento

Il paradigma MVC, per quanto molto elegante e semplice, si rivela tuttavia insufficiente per applicazioni complesse; occorre quindi introdurre alcune estensioni.

7. STRUTTURE GERARCHICHE

In una applicazione reale, le strutture dati sono generalmente organizzate in modo complesso, spesso gerarchicamente [**fol 90**] [**shn 92**].

Nel semplice esempio dell'**editor grafico**, è ragionevole immaginare l'oggetto **disegno** come un insieme di **elementi grafici**. A sua volta un elemento grafico complesso può essere costituito da elementi più semplici.

I Model per gli oggetti di complessità maggiore dovranno quindi gestire fra i propri attributi i Model per i loro elementi costituenti; ad esempio il modello per il **disegno** conterrà fra i propri attributi una lista di modelli per **elementi grafici**.

Una modifica ad un componente è allo stesso tempo una modifica per il Model che lo contiene, e questo deve riceverne notifica.

Nello stesso tempo però, per garantire la modularità e la riusabilità delle classi costruite, i Model non devono avere alcuna pre-conoscenza degli oggetti che li useranno come componenti, così come non devono averla delle View.

Simmetricamente una View relativa a un Model strutturato, dovrà a sua volta poter gestire le View corrispondenti ai singoli componenti.

Per questo motivo abbiamo introdotto nelle classi Model e View la gestione di relazioni *Owner-Member*. In maniera esattamente analoga a quanto avviene per la relazione *Model-View*, ogni oggetto può essere posto in relazione con un Owner attraverso il servizio

- SetOwner(Owner)

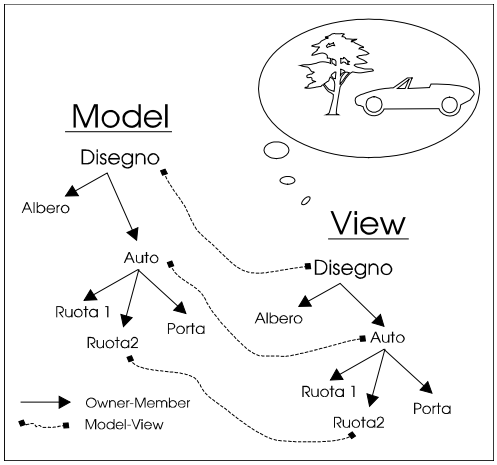

Fig. 4 - Relazione *Owner-Member*

equivalente a SetModel(). I messaggi vengono quindi inviati in modo automatico verso Owner e Members.

L'architettura ad oggetti dell'applicazione risulta quindi definita mediante una struttura a piramidi gerarchiche comunicanti: l'elemento cardine è la piramide dei Model, legati dalle rispettive relazioni *Owner-Membership*, alla quale si affiancano le corrispondenti piramidi per le View.

Consideriamo, ad esempio, il flusso di messaggi fra le componenti dell'**editor grafico** per l'azione di "spostamento" di un oggetto. Per introdurre il vincolo che gli oggetti possano posizionarsi solamente sulle maglie di una griglia, è sufficiente che il Model per il **disegno**, quando riceve il messaggio di *move* da un **elemento grafico** del quale egli è *Owner*, verifichi la correttezza della posizione ed eventualmente gli richieda di spostarsi in una nuova posizione ammessa. La presenza e la gestione della griglia è quindi di esclusiva responsabilità del **disegno** e non influisce in alcun modo con i componenti di livello inferiore.

Utilizzando questi meccanismi si realizzano architetture applicative modulari, nelle quali è possibile sostituire elementi o introdurne dei nuovi facilmente, a patto di rispettare il vincolo che nessun oggetto richieda la conoscenza di caratteristiche del proprio Owner o delle proprie View.

8. LEGAMI DIRETTI FRA OGGETTI

I legami gerarchici *Model-View* e *Owner-Member* consentono di realizzare quelle architetture applicative in cui l'interazione con l'utente o altri eventi producano modifiche ai Model e queste debbano essere propagate alle loro rappresentazioni.

Esistono tuttavia delle situazioni nelle quali è necessario collegare fra di loro due oggetti, altrimenti indipendenti, affinché possano coordinare la propria attività.

Un tipico caso è quello in cui si vogliano garantire caratteristiche di omogeneità fra diverse View dello stesso Model, ad esempio qualora si voglia garantire che utenti diversi di una applicazione distribuita vedano tutti sul proprio schermo una scena dallo stesso punto di vista e alla stessa scala.

La violazione del vincolo che non vi siano comunicazioni trasversali fra le viste è giustificata in quanto riguarda caratteristiche, come l'area visualizzata o il fattore di scala, che sono di esclusiva pertinenza delle View e non coinvolgono in alcun modo il Model cui appartengono.

Abbiamo quindi deciso di rendere disponibile per le classi Model, View e Controller il servizio

- RegisterOnChange(object)

grazie al quale un altro oggetto può dichiarare di essere interessato a ricevere notifica in caso di modifiche.

9. EREDITARIETÀ DEI CONTROLLER

La corrispondenza biunivoca

View <=> Controller

implica l'installazione di un controllore per ciascuna vista.

Tuttavia una applicazione grafica interattiva spesso presenta all'utente un numero molto elevato di View per oggetti appartenenti a poche classi; ad esempio nel caso dell'**editor grafico** il **disegno** può essere costituito da centinaia di **elementi grafici**.

L'installazione di un nuovo Controller risulta quindi onerosa sia in termini di performance che di memoria, poiché implica l'installazione per ciascuna View di uno specifico Controller.

Abbiamo pertanto introdotto il concetto di Class-Controller: è cioè possibile installare un controllore non solo a livello di singolo oggetto, ma anche a livello di classe, comune a tutte le

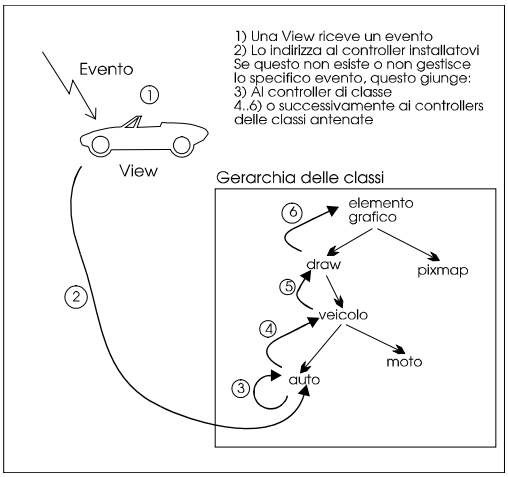

Fig. 5 - Gestione degli eventi con i Class-Controller

istanze.

Quando si verifica un evento all'interno di una View, il sistema lo invia all'eventuale Controller. Se questo non gestisce l'evento, viene passato al controllore della classe, a quello della classe padre e così via, fino a quando l'evento non viene consumato o si giunge alla base della gerarchia.

Gli eventi vengono pertanto sempre gestiti dal Controller più specializzato fra quelli installati sulla gerarchia di classi della View.

Ad esempio, per gestire l'*editing* di un **elemento grafico** occorreranno dei Controllers diversi per ciascuna sottoclasse, ma non per ogni singolo oggetto. D'altro canto tutti gli **elementi grafici** sono dotati dell'attributo posizione e quindi l'operazione di *move* può essere gestita da un unico Controller installato sulla classe base.

10. DISPATCHING E "GRAB" DEGLI EVENTI

Durante le fasi interattive, gli eventi generati dagli utenti dell'applicazione vengono indirizzati dal sistema alla View corrispondente. Questa a sua volta li redirige al Controller per la gestione.

Nel caso di View gerarchiche, legate fra di loro utilizzando relazioni *Owner-Member*, l'evento raggiunge prima la cima della piramide delle istanze di oggetti e, se non viene consumato dal Controller corrispondente, si propaga ai Member.

Questo meccanismo è stato scelto (preferendolo a quello in cui si procede dalle foglie della gerarchia delle istanze verso la radice), poiché è sembrato più adatto a gestire entità grafiche strutturate [**shn 92**].

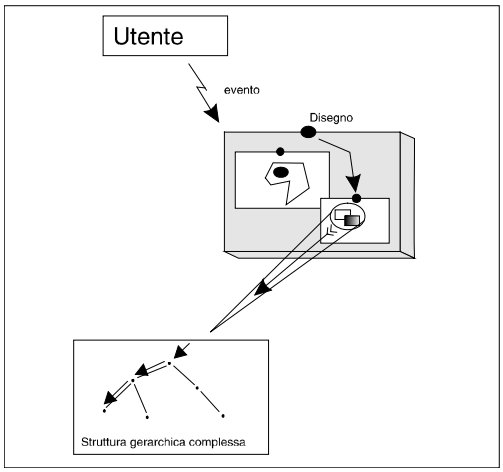

Fig. 6 - Propagazione degli eventi per View gerarchiche

Ad esempio, nel caso dell'editor grafico, selezionando un punto corrispondente alla ruota di una **automobile**, l'evento raggiunge prima i Controller installati sull'oggetto **disegno**, poi quelli relativi all'istanza di **automobile**, infine quelli per la **ruota** specificamente selezionata.

In molti casi inoltre, una interazione programma-utente risulta da un insieme di eventi, la cui successione rispetta regole specifiche [**fol 90**] [**shn 92**]. È opportuno quindi che questi eventi siano gestiti da un unico Controller.

Consideriamo ad esempio il tipico caso dello spostamento di un oggetto grafico in una nuova posizione. Un possibile schema di interazione è costituito da due passi [**fol 90**]:

1. Selezione con il mouse dell'oggetto
2. Selezione della nuova posizione

La prima interazione viene ovviamente indirizzata all'oggetto selezionato, ma la seconda corrisponde a puntare il cursore in una posizione del disegno lontana dall'oggetto, e quindi in condizioni normali questo non riceverebbe alcun messaggio.

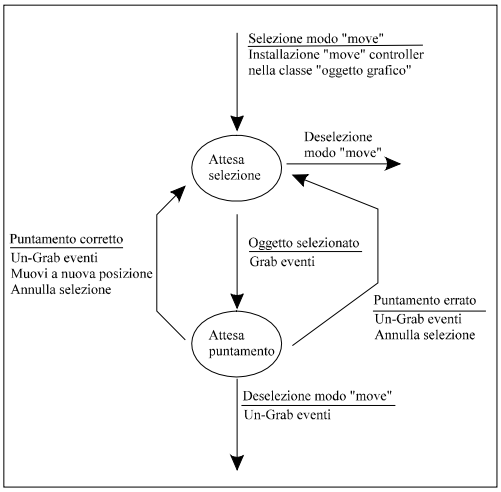

Fig. 7 - Automa a stati per il "Move Controller"

La soluzione a questo problema consiste nella definizione di un protocollo attraverso il quale un Controller può catturare (eseguire un *grab*) gli eventi generati dall'interazione con l'utente e decidere di gestirli, anche se non gli competerebbero.

In questo modo un piccolo automa a stati all'interno del Controller può gestire completamente una fase interattiva a più passi.

Analizziamo in dettaglio come viene gestito tale tipo di interazione (fig. 7):

- L'utente seleziona la funzione "move". Come conseguenza viene installato sugli **elementi grafici** il "Move Controller". L'automa viene posto nello stato di *attesa selezione*.

- L'utente seleziona un **elemento grafico**. L'evento viene passato al Controller, che richiede un *grab* degli eventi successivi e si pone nello stato *attesa puntamento*.
- L'utente seleziona una posizione. L'evento viene rediretto al Controller, che acquisisce le coordinate, libera il *grab*, notifica la nuova posizione al Model e torna nello stato iniziale di *attesa selezione*.
- La modifica del Model scatena i messaggi di notifica a *Views, Owner e Members*.

11. L'IMPLEMENTAZIONE DEL PARADIGMA MVC

Il modello concettuale MVC qui descritto è stato implementato in C++ all'interno di un ambiente per lo sviluppo di applicazioni grafiche interattive 2D [**chi 93**].

È stata realizzata la classe di base **MVC_object**, che implementa i servizi di base per la gestione dei messaggi e delle relazioni *Owner-Member*. Da questa derivano le classi **Model**, **View** e **Controller.**

Il principale problema implementativo è stata la scelta del meccanismo di gestione dei messaggi. Il C++ infatti non dispone di un meccanismo di invocazione di messaggi generici.

Questo implica da parte di chi manda un messaggio ad un oggetto una conoscenza a priori dei servizi messi a disposizione, conoscenza che è contraria ai principi che abbiamo messo alla base del modello MVC.

Abbiamo pertanto deciso di realizzare un metodo generico per la gestione dei messaggi:

- Message(messageID, messageData)

in cui il cui primo parametro sia un identificatore per il messaggio e il secondo parametro sia una struttura contenente i dati che caratterizzano il messaggio stesso.

Il riconoscimento del messaggio e l'invocazione del metodo opportuno, se esiste, sono lasciati come compito a questo singolo metodo centralizzato.

Nella nostra realizzazione vi è un *database di messaggi centralizzato*; le singole classi vi registrano i messaggi che vogliono gestire e i metodi da invocare per la loro gestione.

Il gestore dei messaggi verifica a run-time se per la classe cui appartiene l'oggetto esiste una registrazione per uno specifico messaggio e invoca il relativo metodo.

Sono stati anche introdotti strumenti per il logging e l'analisi del flusso dei messaggi a run-time, indispensabili in fase di debugging.

È anche importante sottolineare come nella implementazione attuale i messaggi siano **sincroni**: il metodo Message() è bloccante e ritorna solo quando il messaggio è stato gestito.

12. CONCLUSIONI

Il paradigma MVC così esteso, si è nella pratica dimostrato uno strumento prezioso per lo sviluppo di applicazioni grafiche interattive anche molto complesse.

In particolare si sono riscontrati tangibili vantaggi per quanto riguarda:

- Capacità di far evolvere con continuità il codice dallo stadio di prototipo a quello di applicazione finale.
- Riutilizzabilità del codice.
- Facilità di intervento per operazioni di manutenzione e modifica.

D'altro canto si è evidenziato come sia importante mantenere sotto controllo il flusso dei messaggi, per non penalizzare le prestazioni in modo significativo. Per questo motivo sono stati introdotti specifici strumenti.

Gli utilizzatori hanno anche evidenziato la necessità di introdurre meccanismi di sincronizzazione più evoluti e la possibilità di gestire messaggi asincroni.

13. BIBLIOGRAFIA

Ada 88 Adams Sam S. - "Meta Methods - The MVC Paradigm", HOOPLA!, pp.5-21, July 1988

Boo 94 Booch G. - *Object Oriented Analysis and Design with Applications*, Benjamin Cummings Pub., 1994

Chi 93 Chiozzi G., Ghezzi G., Pantarotto S., Tucci M. - "A C++ Application Framework to integrate X-Windows and graPHIGS in a coherent object oriented environment", proceedings VIII Convegno ICO GRAPHICS, Milan, Italy, March 1993

Cox 86 Cox Brad J. - *Object Oriented Programming - An Evolutionary Approach*, Addison-Wesley, Reading, MA, 1986

Fol 90 Foley J., Van Dam A., Feiner S., Hughes J. - *Computer Graphics, principles and practice*, Addison-Wesley, Reading, MA, 1990

Kno 89 Knolle N. - "Variations of Model-View-Controller", *Journal of Object Oriented Programming*, Sept/Oct 1989

Kno 90 Knolle N., Fong M., Lang R. - "SITMAP: A Case Study in Object-Oriented Design and Object-Oriented Programming", *Applications of Object-Oriented Programming*, Pinson L.J., Wiener R.S., editors, Addison-Wesley, Reading, MA 1990

Kra 88 Krasner G.E., Pope S.T. - "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80", *Journal of Object Oriented Programming*, pp.26-49, Aug/Sept 1988

Mey 88 Meyer B. - *Object-Oriented Software Construction*, Prentice Hall, New Jersey, 1988

Mey 92 Meyer B., Rosson M.B. - "Survey on User Interface Programming", proceedings SIGCHI'92, Monterrey, CA, May 3-7, 1992

Mey 92b Meyer B. - "State of the Art in User Interface Tools", *Advances in Human-Computer Interaction, Vol. 4*, Hartson H.R. and Hix D. editors, Ablex Pub., NJ, 1992

Shn 92 Shneiderman B. - *Designing the User Interface*, Addison-Wesley, New York, 1992